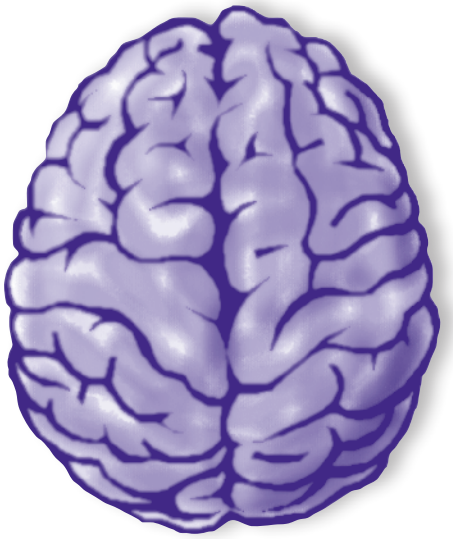
Un guide d'apprentissage

Couvre Java 5.0

Java

Tête la première



Kathy Sierra
& Bert Bates
Traduction de Marie-Cécile Baland

DIGIT BOOKS

Propriété de André MAFFEI <andre.maffei@yahoo.fr>

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Java

Tête la première

2e édition

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Java™
Tête la première

2e édition

Kathy Sierra
Bert Bates

Traduction de
Marie-Cécile Baland

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

À nos cerveaux, toujours à notre disposition.

Enfin plus ou moins ...

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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

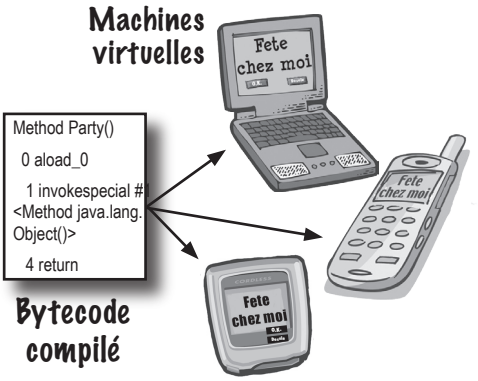
Votre cerveau et Java. Voilà que *vous* essayez d'*apprendre* quelque chose tandis que votre *cerveau* fait tout ce qu'il peut pour vous empêcher de *mémoriser*. Votre cerveau pense « Mieux vaut laisser de la place pour les choses vraiment importantes, comme savoir quels sont les animaux sauvages qu'il 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

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

1 Jetons-nous à l'eau

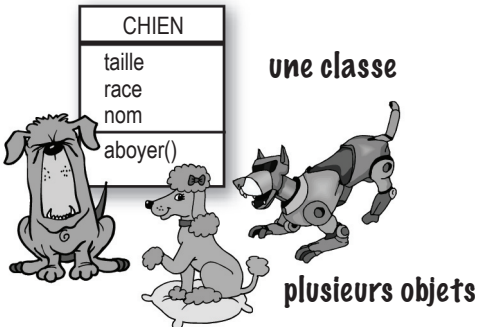
Java vous emmène ailleurs. Depuis l'humble révélation au public de la version 1.02, Java a séduit les programmeurs par sa syntaxe conviviale, ses caractéristiques orientées objet, sa gestion de la mémoire, et, surtout, sa promesse de portabilité. Nous allons piquer une tête, écrire du code et le compiler.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

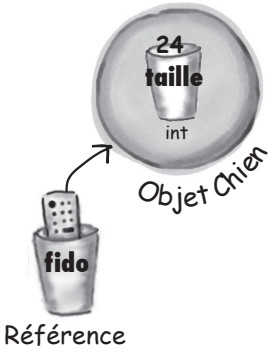


La guerre des chaises (Kevin (OO) et Bernard (procédural))	28
Héritage (introduction)	31
Redéfinir les méthodes (introduction)	32
Qu'y a-t-il dans une classe ? (méthodes, variables d'instance)	34
Créer votre premier objet	36
Utiliser main()	38
Codons un jeu	39
Exercices et jeux	42

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

3 Maîtrisez vos variables

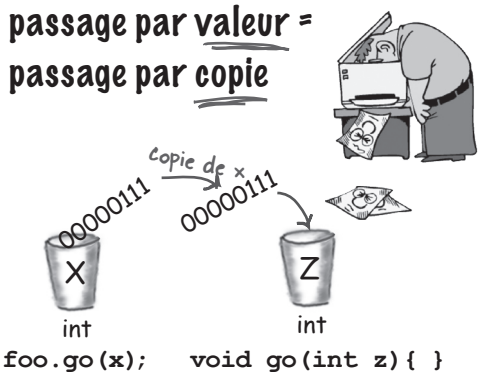
Il existe deux sortes de variables : les types primitifs et les références. Il y a autre chose dans la vie que des entiers, par exemple des chaînes de caractères et des tableaux. Que se passe-t-il si vous avez un objet Maître avec une variable d'instance Chien ? Ou une Voiture avec un Moteur ? Dans ce 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.***



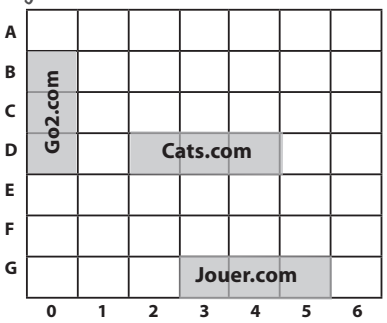
Les méthodes utilisent l'état des objets (aboyez autrement)	73
Arguments et types de retour	74
Passage par valeur (la variable est *toujours* copiée)	77
Méthodes get et set	78
Encapsulation (faites-le ou vous risquez d'être humilié)	80
Utiliser des références dans un tableau	83
Exercices et jeux	88

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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).

Nous allons construire un jeu : Couler un Point Com.



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

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

7 Mieux vivre à Objectville

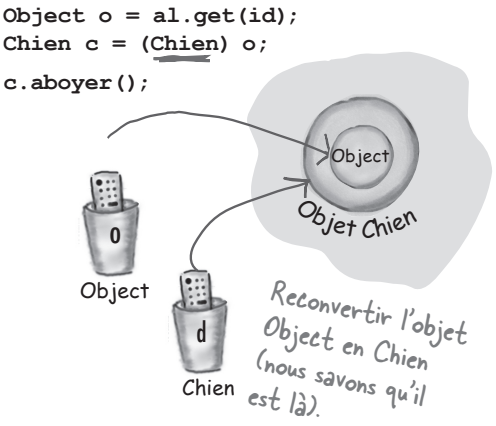
Planifiez vos programmes en pensant à l'avenir. Et si vous pouviez écrire du code que quelqu'un *d'autre* puisse étendre **facilement** ? Et si ce code était suffisamment souple pour autoriser ces satanés changements de spécifications de dernière minute ? Quand vous aurez adhéré au Plan Polymorphisme, 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…

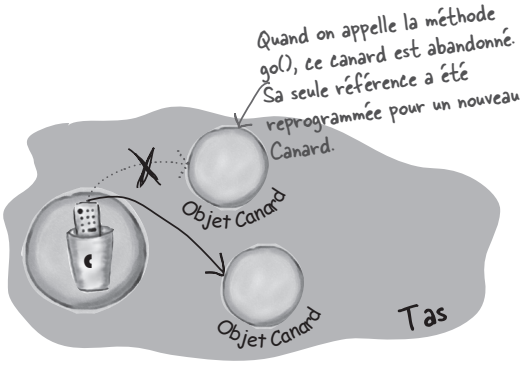
Certaines classes ne doivent pas être instanciées	200
Classes abstraites (*non instanciables*)	201
Méthodes abstraites (doivent être implémentées)	203
Le polymorphisme en action	206
La classe Object (la superclasse ultime)	208
Extraire des objets d'une ArrayList (ils sont de type Object)	211
Le compilateur 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



Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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

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*.

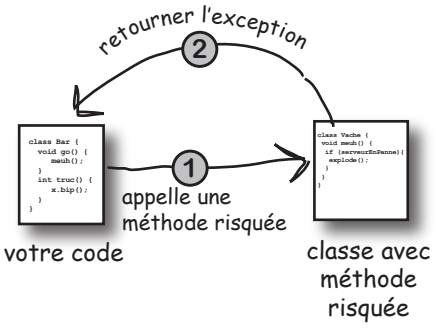
Les variables statiques sont partagées par toutes les instances d'une classe.

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

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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.



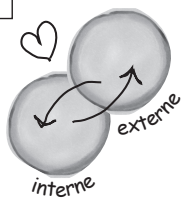
Créer une boîte à rythmes (la BeatBox)	316
Et si vous devez appelrr un code risqué ?	319
Les exceptions signalent la survenue d'une erreur	320
Le compilateur vérifie que vous êtes conscient des risques	321
Intercepter les exceptions avec un bloc *try/catch*	322
Contrôle de flots dans les blocs *try/catch*	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

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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

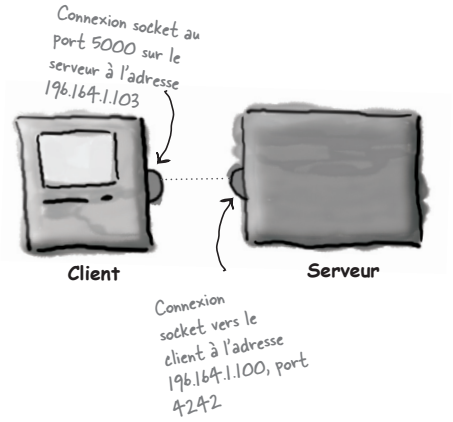
On peut compresser et décompresser les objets. Les objets ont un état et un comportement. Le *comportement* se trouve dans la *classe*, mais l'*état* réside dans chaque *objet* individuel. Si vous devez sauvegarder un état *vous pouvez emprunter la voie difficile,* interroger chaque objet et écrire laborieusement 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

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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* ?

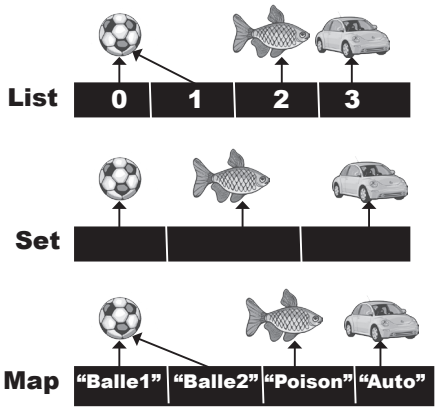


Aperçu du programme de discussion	473
Se connecter, émettre et recevoir	474
Sockets réseau	475
Ports TCP	476
Lire des données dans une socket (avec BufferedReader)	478
Écrire des données dans une socket (avec PrintWriter)	479
Conseil du jour : le code du client	480
Écrire un serveur simple	483
Conseil 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

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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 ?

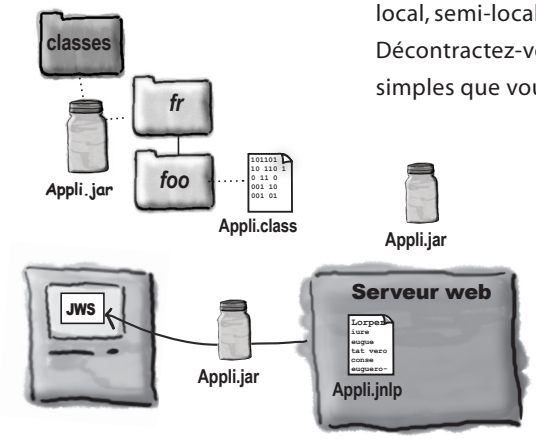


Collections	533
Trier une ArrayList avec Collections.sort()	534
Génériques et fiabilité du type	540
Généricité et typage	540
Trier des objets qui implémentent l'interface Comparable	547
Trier des objets avec un Comparateur personnalisé	552
L'API des collections : List, Set et Map	557
Éviter les doublons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

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

17 Déployez votre code

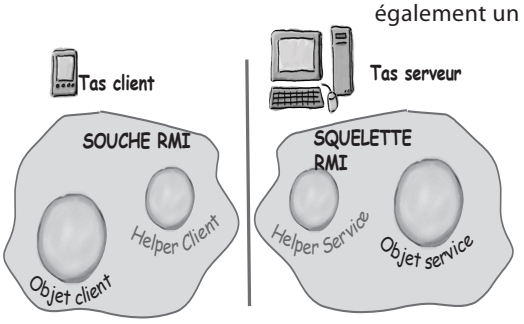
Il est temps de lâcher prise. Vous avez écrit votre code. Vous l'avez testé. Vous l'avez peaufiné. Vous avez dit à tout le monde que vous adoreriez ne plus jamais voir une ligne de code. Mais vous avez fini par créer un chef d'œuvre. Et en plus il tourne ! Et maintenant ? Dans les deux derniers chapitres, 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

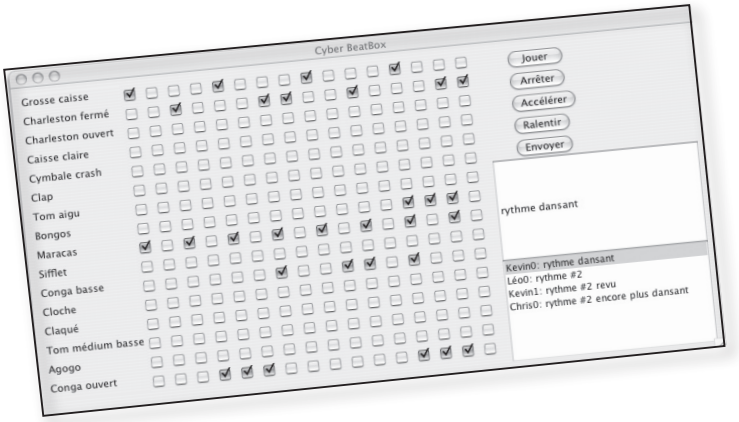
Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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

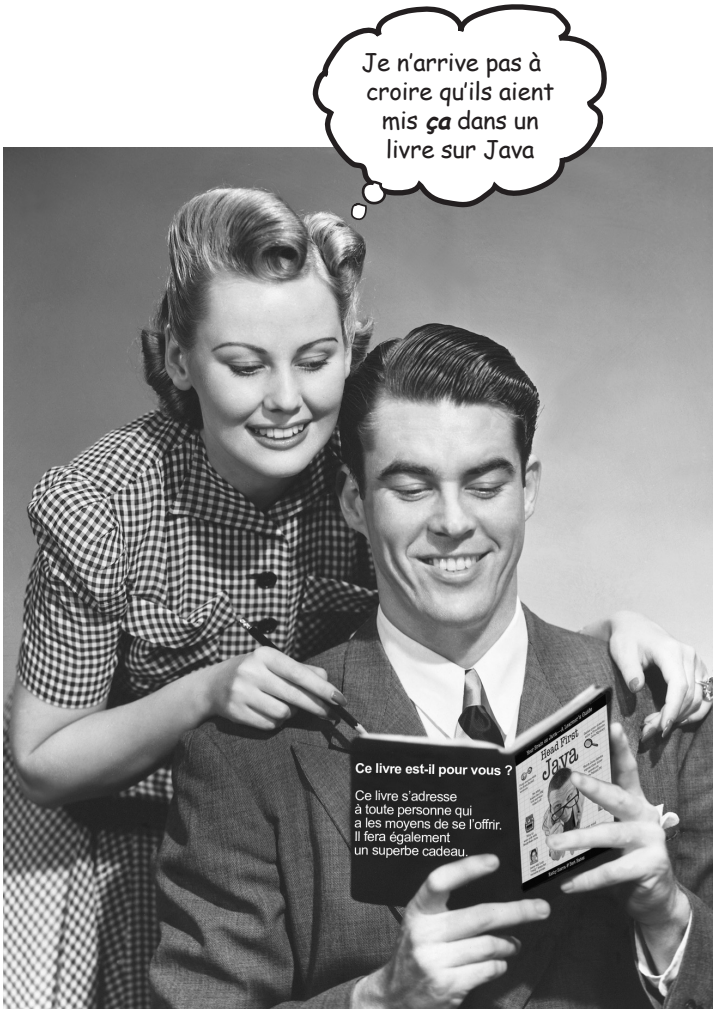
Énumérations 671

i Index

677

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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 ? »

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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]

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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 ?»

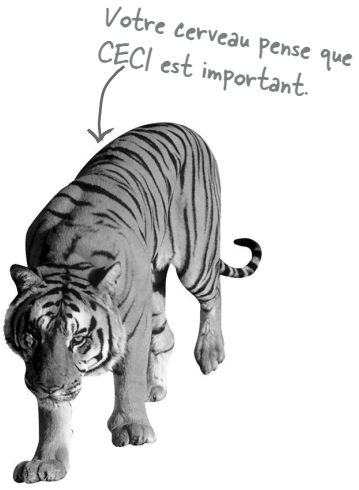
«Je mangerais bien une pizza.»

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*.

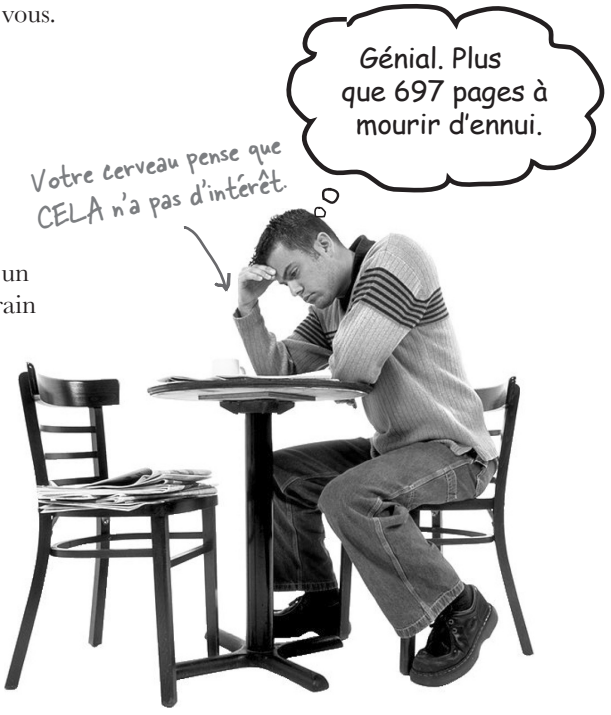
Et votre cerveau ne sait rien d'autre…

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.

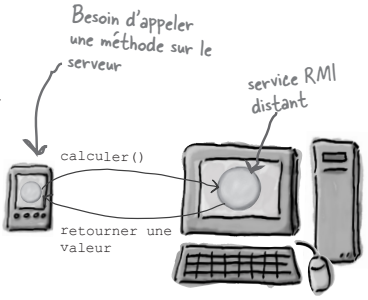
Et il n'y a a aucun moyen simple de lui dire : «Hé, cerveau, merci bien, mais même si ce livre est ennuyeux comme la pluie, et même si j'ai le moral à zéro, je *veux* que tu fasses attention.»



Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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 ?

C'est terrible d'être une méthode abstraite. On n'a pas de **corps**.

```
abstract void roam();
```

Pas de corps de méthode! Terminer par un point-virgule.

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.

Est-ce qu'une Salle de Bains EST-UNE Baignoire ? Ou bien est-ce une relation A-UN ?

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.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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. »

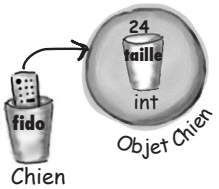
La voie rapide consiste à faire *tout ce qui non seulement augmente mais aussi diversifie l'activité cérébrale*. Les éléments de la page précédente constituent une grande partie de la solution, et ils se sont tous révélés capables d'aider votre cerveau à travailler en votre faveur. Certaines études démontrent 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.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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.

POINTS D'IMPACT

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.

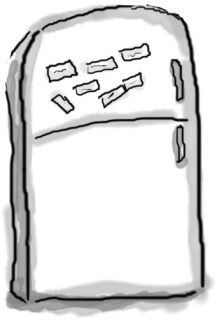
Nous avons également incorporé des ***histoires*** et des ***exercices*** qui présentent ***plusieurs points*** de vue, parce que votre cerveau est ainsi fait que vous apprenez plus en profondeur s'il est forcé de faire des évaluations et de porter des jugements.

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.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>



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*

Propriété de Andre MAFFEI <andre.maffei@yahoo.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.

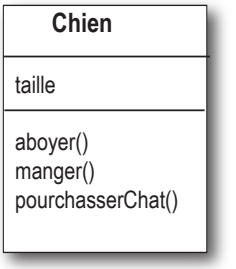
Note 2 : Pratiquement tout le code de ce livre est disponible sur tetelapremiere.fr

Propriété de Andre MAFFEI <andre.maffei@yahoo.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.)

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.

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

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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*.)

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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 :

Corey McGlone

Jef Cumps

Johannes de Jong

Jason Menard

Thomas Paul

Marilym de Queiroz

Rodney J. Woodruff

James Cubeta Terri Cubeta

Ira Becker

John Nyquist

Chris Jones

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

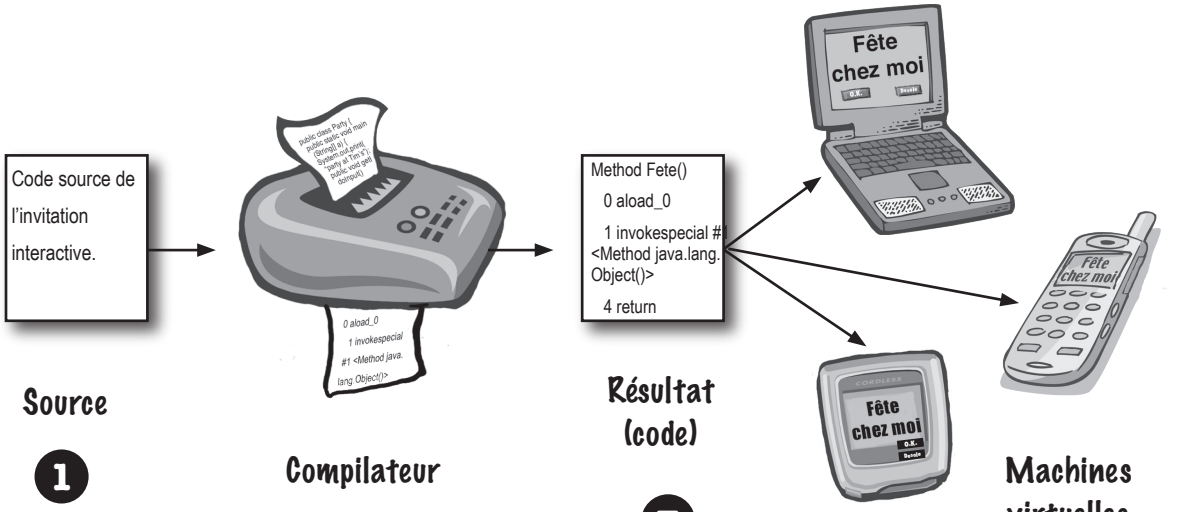
Jetons-nous à l'eau

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.**

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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.



Source

1

Créez un document source. Utilisez un protocole établi (en l'occurence le langage Java).

Compilateur

2

Passez votre document dans un compilateur de code source. Le compilateur recherche les erreurs et refuse de compiler tant qu'il n'est pas sûr que tout fonctionnera correctement.

Résultat (code)

3

Le compilateur crée un nouveau document, codé en ***bytecode*** Java. Tout équipement capable d'exécuter Java saura interpréter ce fichier et le traduire en quelque chose qu'il pourra exécuter. Le bytecode compilé est indépendant de la plate-forme.

Machines virtuelles

4

Vos amis n'ont pas de machine Java physique, mais ils ont tous une machine ***virtuelle*** Java (implémentée au niveau logiciel) qui fonctionne sur leurs gadgets électroniques. La machine virtuelle lit et *exécute* le bytecode.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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

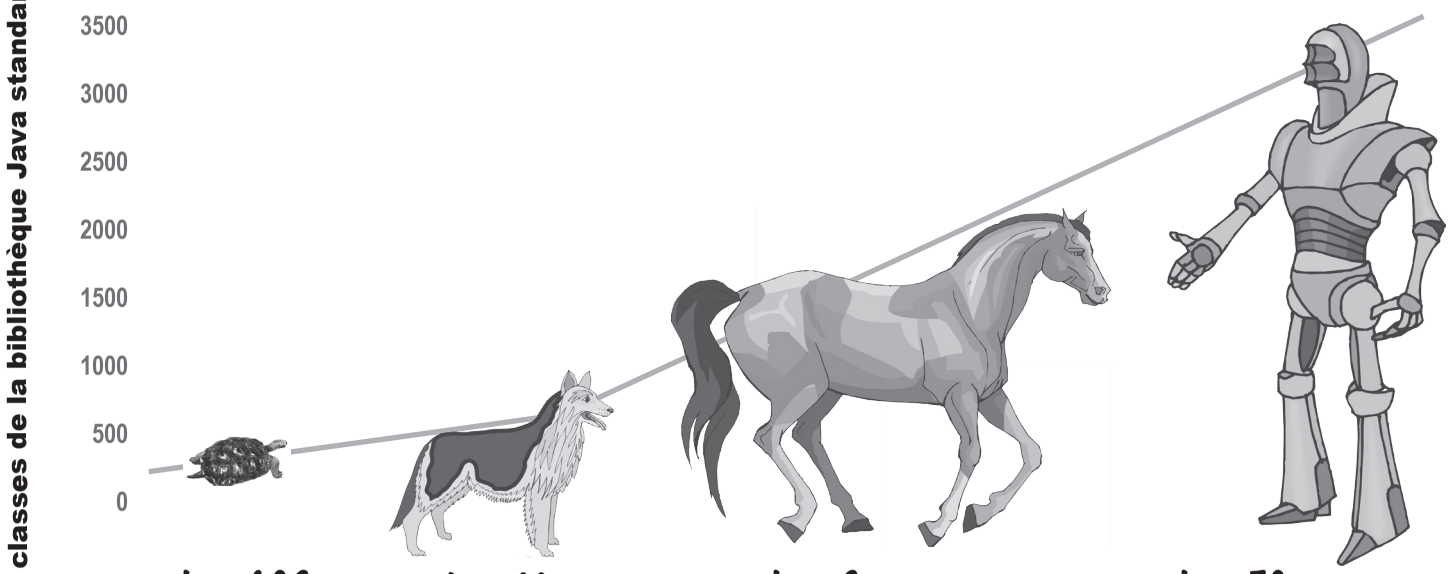
4

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.)

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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...

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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...)

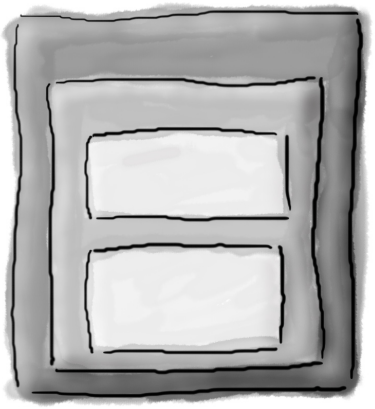
ici, vous saurez que votre essai n'a pas fonctionné...

dans ce cas, afficher "Fichier non trouvé" sur la ligne de commandes

entre les { }, ce qu'il faut faire en cas de problème...

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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

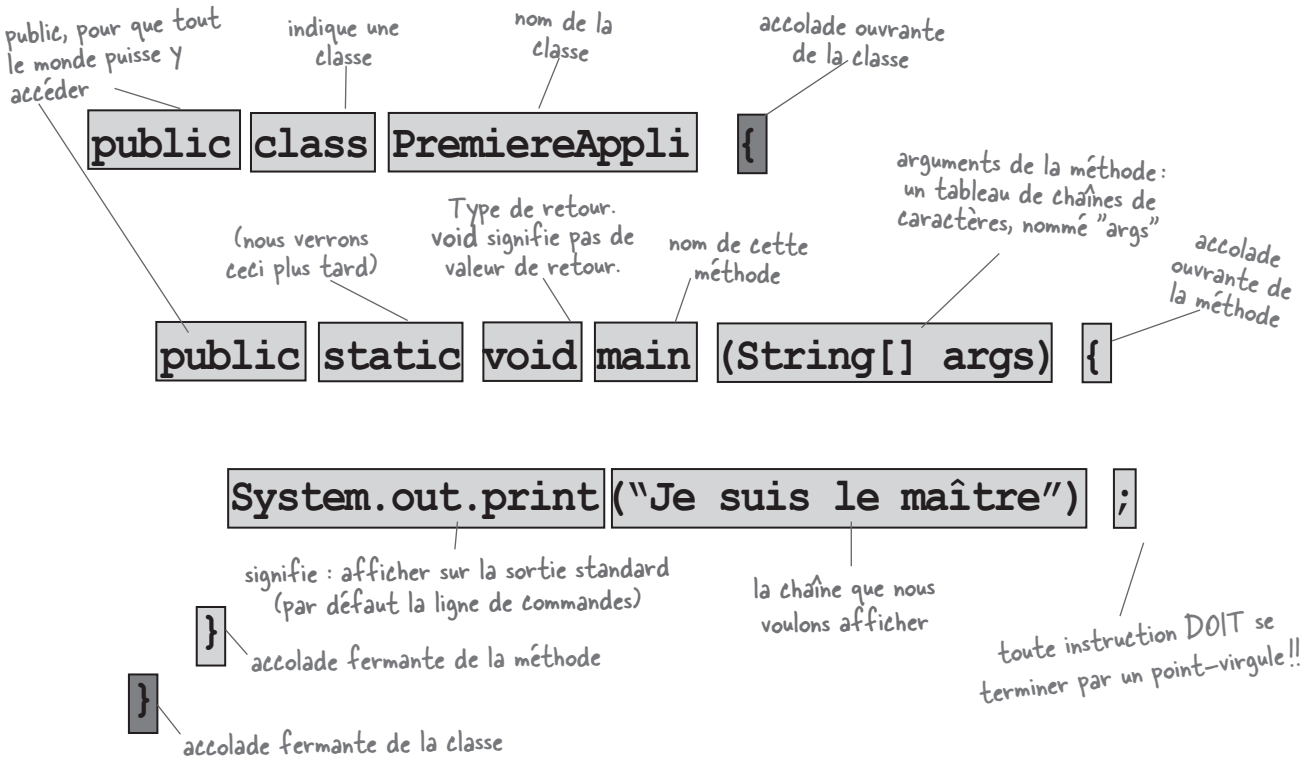
Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Écrire une classe avec une méthode main()

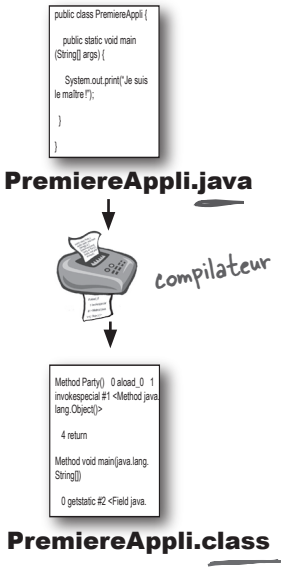
En Java, tout se passe dans une classe. Vous tapez le fichier qui contient votre code source (avec une extension *.java*), puis vous le compilez pour obtenir un nouveau fichier (avec une extension *.class*). Quand vous exécutez votre programme, c'est en réalité une classe que vous exécutez.

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!
```


Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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
}
```


Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

```
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
}
```


Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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) { }`

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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");
      }
   }
}
```


Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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
```

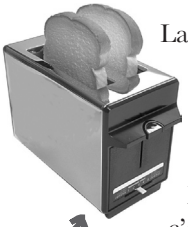
Il y a encore un petit défaut dans notre code. Il se compile et s'exécute, mais le résultat n'est pas parfait à 100 %. Voyez si vous pouvez repérer le problème et le corriger.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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...

VU À LA TV

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.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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);
   }
}
```


Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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

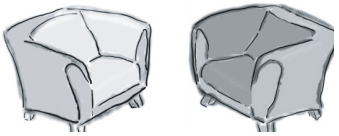
composante paradigmatique incrémentale

stratégie communicante pleinement fonctionnelle

technologie sémantique de quatrième génération

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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

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...

Je n'apprécie pas ce ton.

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 ?

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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...

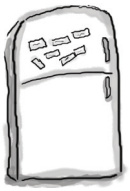
Excusez-moi, mais je n'avais pas fini. Oui, *il y a* des exceptions liées au typage qui peuvent surgir à l'exécution, mais certaines d'entre elles sont permises, car elles sont nécessaires à une autre fonctionnalité importante de Java : la liaison dynamique. Lorsqu'il s'exécute, un programme Java peut 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.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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
```


Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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

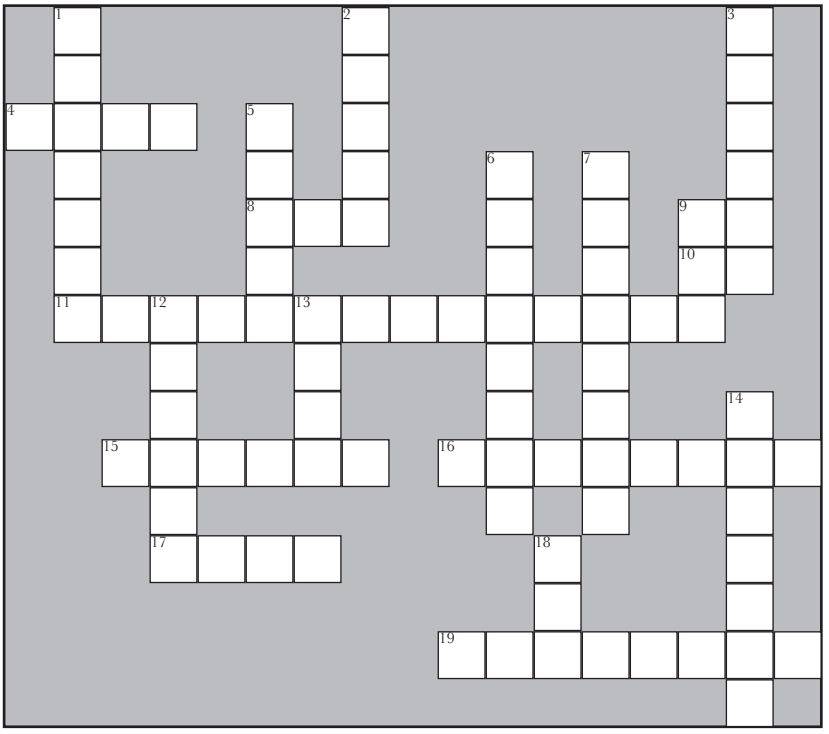
```
class Exercice1b {
    int x = 5;
    while ( x > 1 ) {
      x = x - 1;
      if ( x < 3) {
        System.out.println("petit x");
      }
    }
}
```


Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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
```


Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

La piscine

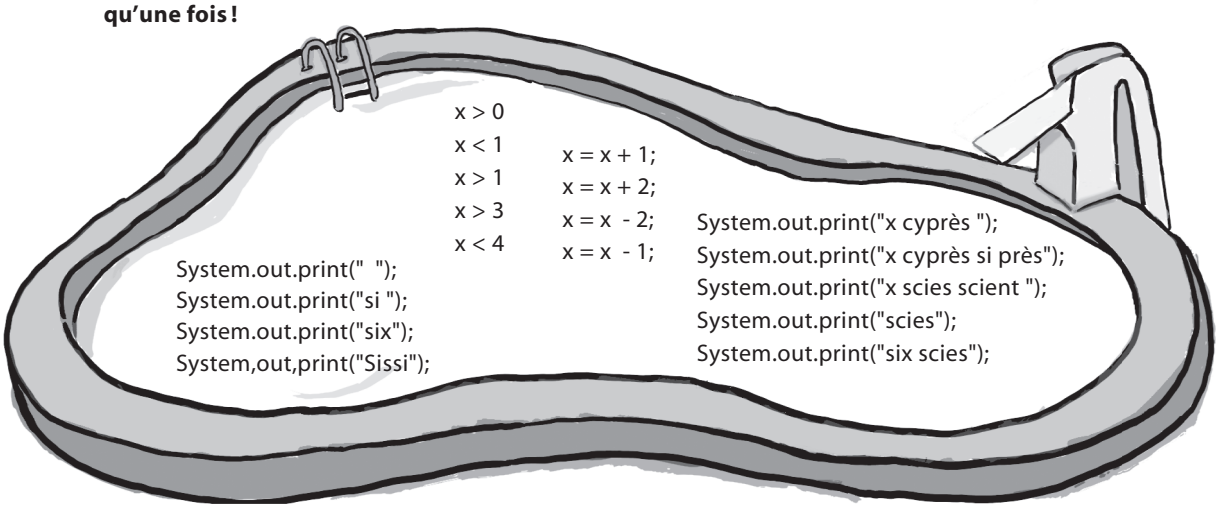
Votre *tâche* consiste à prendre des fragments de code dans la piscine et à les remettre dans le programme à la place des blancs. Vous ne pouvez *pas* utiliser le même fragment deux fois, et vous n'avez pas besoin de les utiliser tous. Votre *but* est de créer une classe qui se compile, s'exécute et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



Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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");
      }
    }
  }
}
```

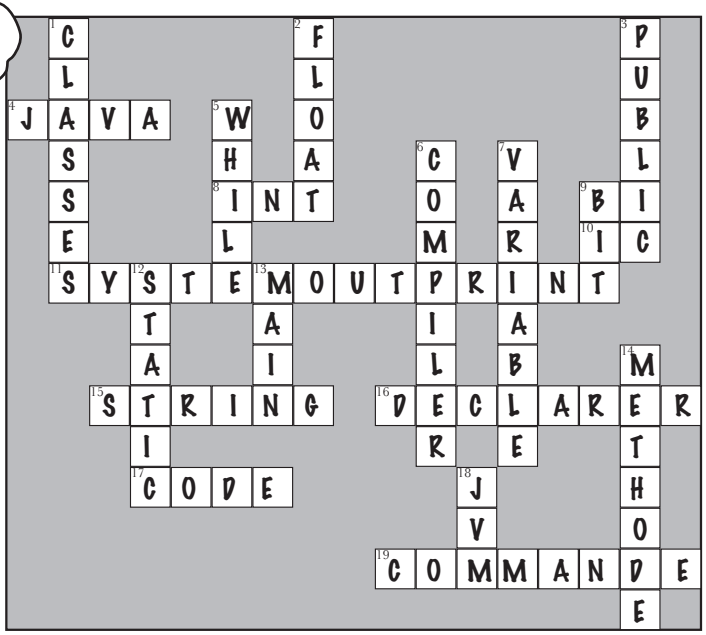
Ce fichier ne se compilera pas sans une **déclaration de classe, et n'oubliez pas** l'accolade correspondante!

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.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>




```
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
```


Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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 ? »***

Les spécifications

Trois formes s'afficheront sur une interface graphique : un cercle, un carré et un triangle. Lorsque l'utilisateur cliquera sur un bouton, la forme pivotera de 360° dans le sens des aiguilles d'une montre (autrement dit, elle fera un tour complet sur elle-même), et le système jouera un fichier son AIF spécifique à cette forme particulière.

La chaise

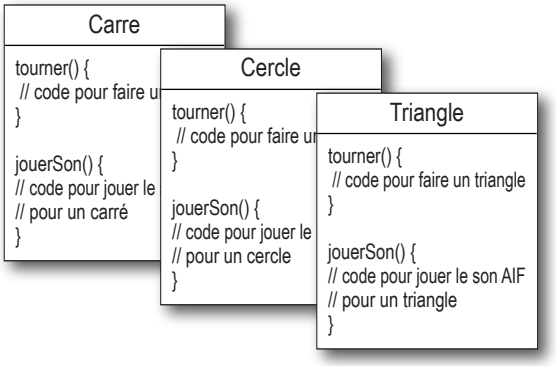
Dans le box de Bernard

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.



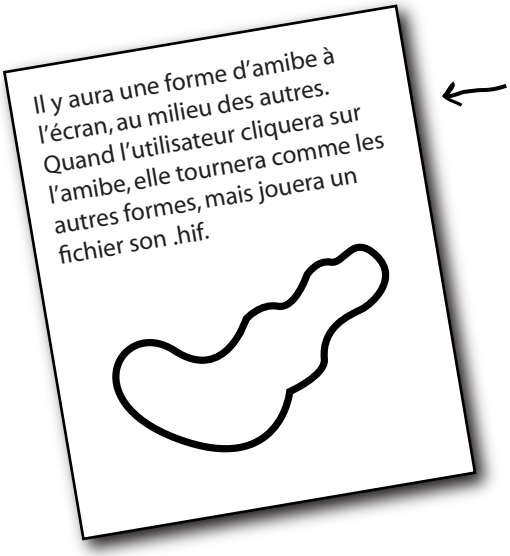
Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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

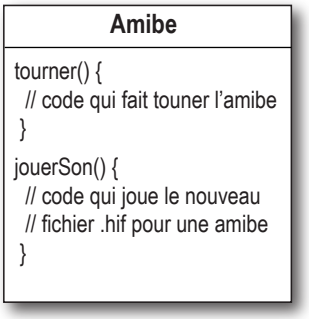
La procédure tourner fonctionnerait toujours, puisque le code utilisait une table pour apparier TypeForme à une forme réelle. Mais ***il fallait modifier jouerSon.*** Et qu'était-ce donc qu'un fichier .hif ?

```
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.

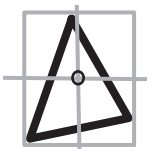


Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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 :

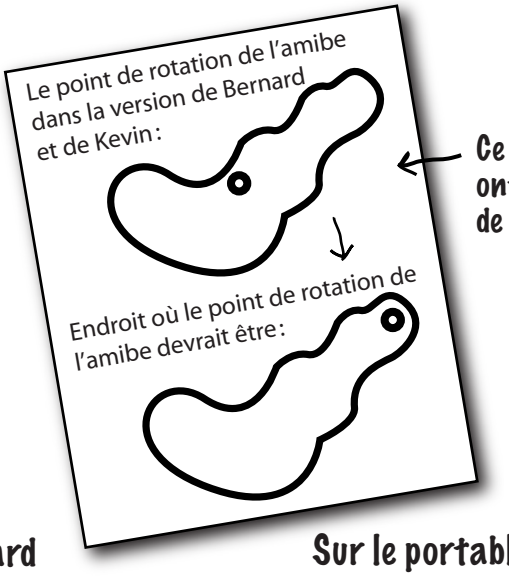


1) déterminer le rectangle dans lequel la forme est inscrite

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
 }
```


Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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.

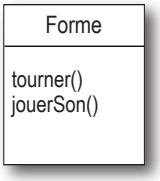
BERNARD : Qu'importe ! C'est stupide comme conception. Ça t'oblige à maintenir quatre « méthodes » tourner différentes. Comment peux-tu justifier ça ?

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

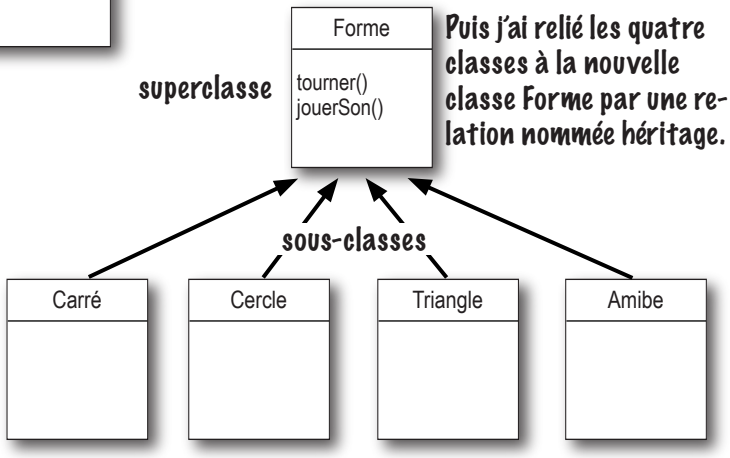
Ce que voulait Bernard
(il se figurait que la chaise l'impressionnerait)

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.*

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

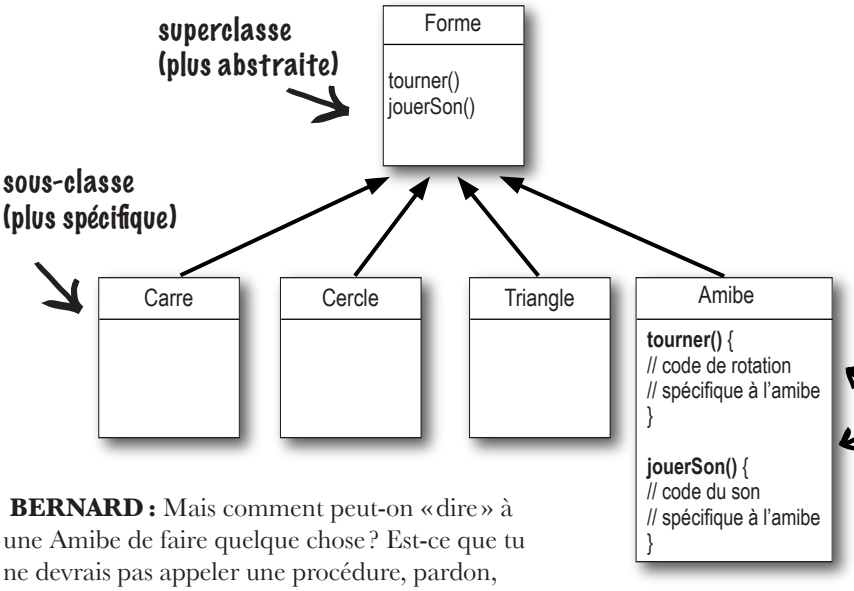
Et la méthode tourner() de l'amibe ?

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.

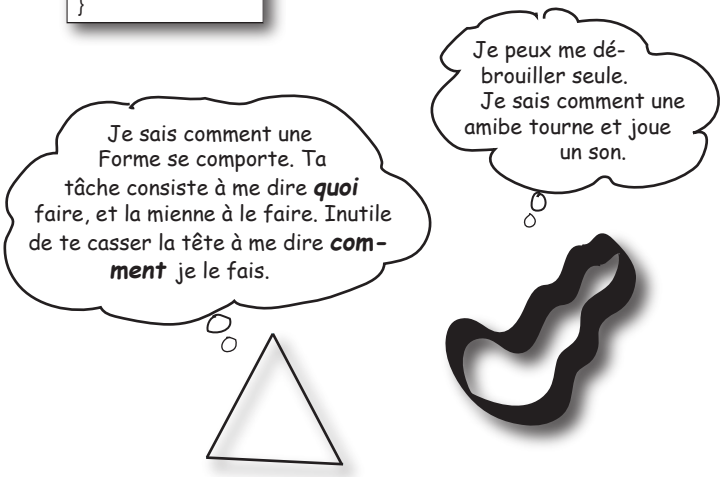


4

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.**



Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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**

Panier	savoir
contenu	
ajouter() supprimer() régler()	faire

Bouton	savoir
libellé couleur	
coloriiser() appuyer() relâcher()	faire

Reveil	savoir
heureAlarme modeAlarme	
setHeureAlarme() setAlarme() alarmeProgramme() pause()	faire

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

- variable d'instance

Ce qu'un objet peut *faire* est une

- méthode

Variables d'instance (état)

Méthodes (comportement)

Chanson	savoir
titre artiste	
setTitre() setArtiste() jouer()	faire

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.

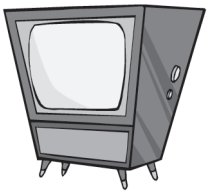
Les **méthodes** correspondent au comportement de l'objet, à ce qu'il peut faire. Quand vous écrivez une classe, vous réfléchissez à ce que l'objet doit savoir, puis aux méthodes qui vont opérer sur ces données. Il est courant pour un objet d'avoir des méthodes qui lisent ou modifient les valeurs des 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.

À vos crayons

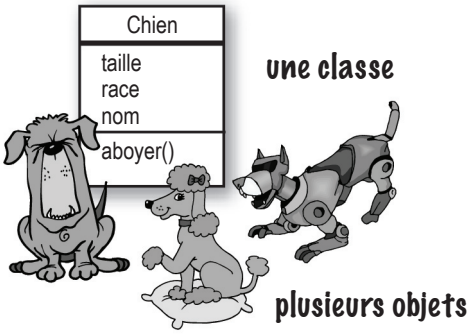
Écrivez ce qu'un objet Television peut avoir besoin de savoir et de faire

Television
variables d'instance
méthodes



Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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

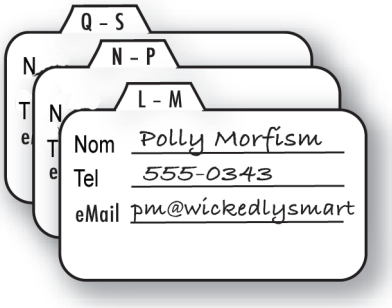


Les classes ne sont pas des objets.

(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.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Vite! Sortons de la méthode main()!

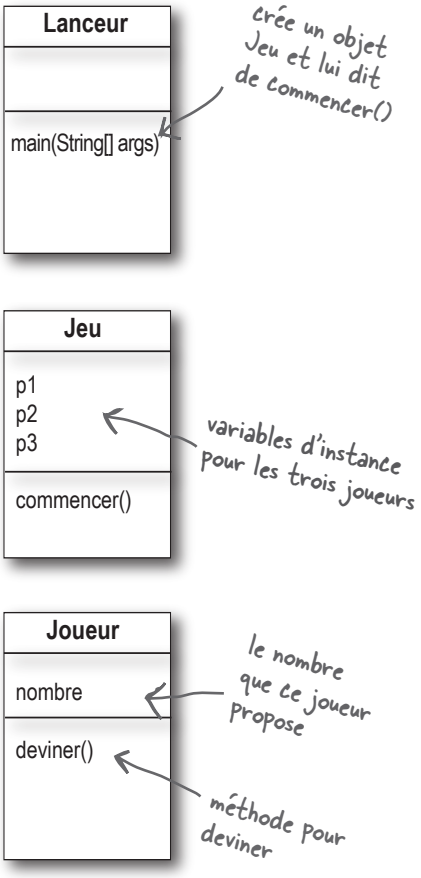
Tant que vous êtes dans la méthode main(), vous n'êtes pas vraiment à Objectville. Un programme de test peut s'exécuter sans problème dans la méthode main(), mais une vraie application OO nécessite des objets qui communiquent, ce qui n'est pas le cas dans une méthode main() statique qui crée et teste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

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

```
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.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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>.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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.

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.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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();
    }
  }
}
```


Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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
```


Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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);
  }
}
```

Résultat

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

```
class ____________ {
  int _________ = 0;
  void ___________ {
    System.out.println("helloooo... ");
  }
}
```

Question bonus !

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

Note : Vous pouvez utiliser chaque fragment plusieurs fois !

x
y
e2
count

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

Echo
Tester
echo( )
count( )
hello( )

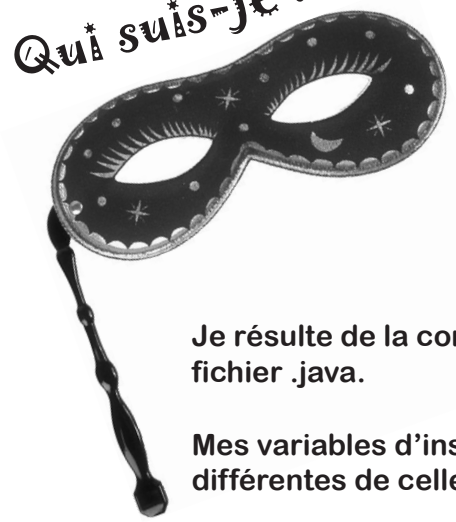
e2 = e1;
Echo e2;
Echo e2 = e1;
Echo e2 = new Echo( );

x = = 3
x = = 4

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

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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.	

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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
```

Vous êtes le compilateur :

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) {

    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!

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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 ?

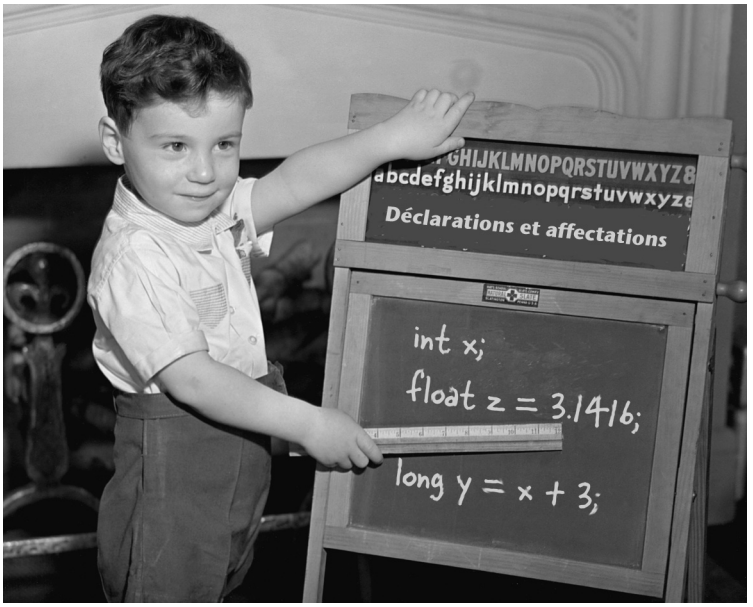
Je résulte de la compilation d'un fichier .java	classe
Mes variables d'instance peuvent être différentes de celles de mes copains.	objet
Je me comporte comme un patron.	classe
J'aime faire des choses.	objet, méthode
Je peux avoir plusieurs méthodes.	classe, objet
Je représente un « état ».	variable d'instance
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.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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*.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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.)

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

« 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 :

petite moyenne grande géante

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.

long int short byte

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 :

byte 8 | short 16 | int 32 | long 64

float 32 | double 64

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.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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;
```


Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

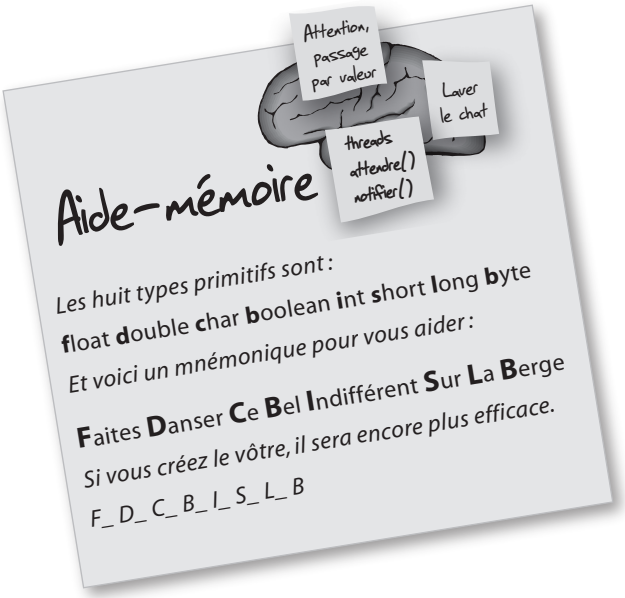
Touche pas à mes mots-clés!

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.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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.

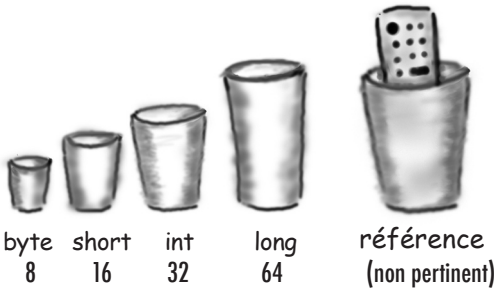
On utilise l'opérateur point (.) sur une variable référence pour dire « utilise ce qui précède le point pour aller me chercher ce qu'il y a après ». Par exemple :

```
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).

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>



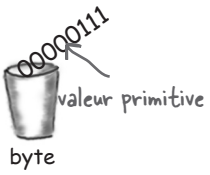
Une référence est simplement une autre valeur de variable.

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).

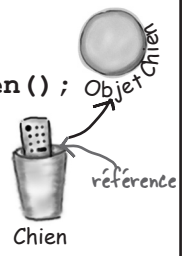


Variable référence

```
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

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

1 Déclarer une variable référence

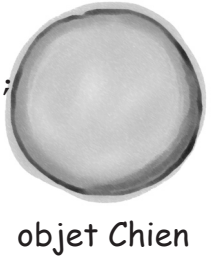
```
Chien monChien = new Chien();
```

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();
```

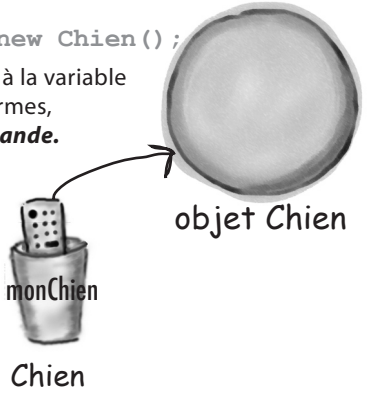
Dit à la JVM d'allouer de l'espace sur le tas pour un nouvel objet Chien (nous étudierons ce processus en détail, notamment au chapitre 9).



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.***



Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

il n'y a pas de
Questions stupides

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

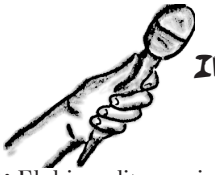
R : On ne sait pas. À part les intimes des développeurs de la JVM, personne ne sait comment une référence est représentée. Il y a des pointeurs quelque part, mais on ne peut pas y accéder. Ce n'est pas nécessaire. (Bon, si vous insistez, vous pouvez imaginer que c'est une valeur sur 64 bits.) Mais quand 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…

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Vie sur le tas et ramasse-miettes

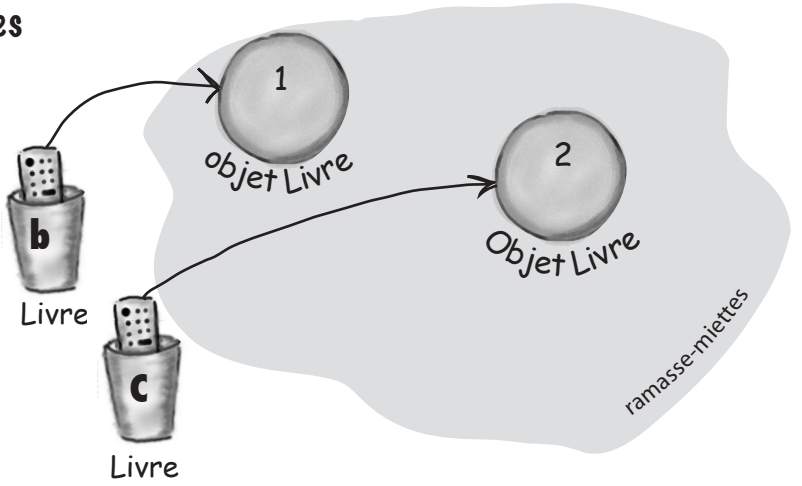
```
Livre b = new Livre();
Livre c = new Livre();
```

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;
```

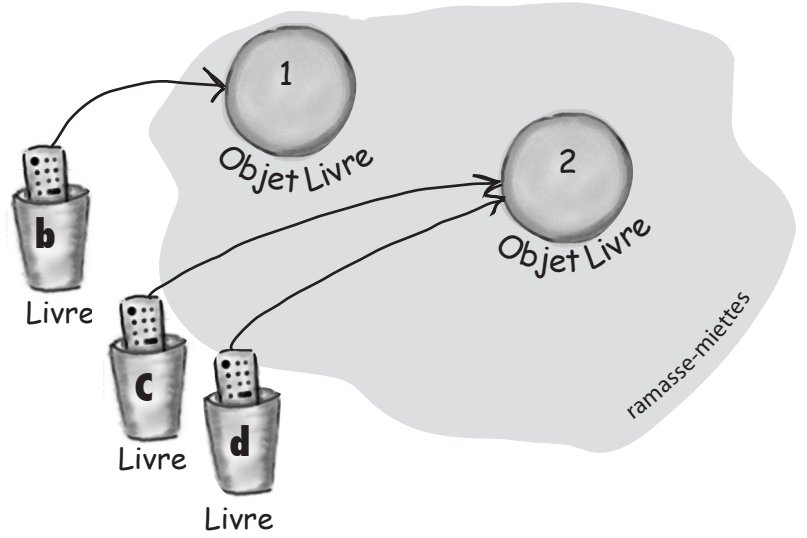
Déclare une nouvelle variable référence Livre. Au lieu de créer un troisième objet Livre, affecte la valeur de la variable ***c*** à la variable ***d***. C'est comme si on disait « Prends les bits qui sont dans ***c***, fais-en une copie et colle-la dans ***d***."

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



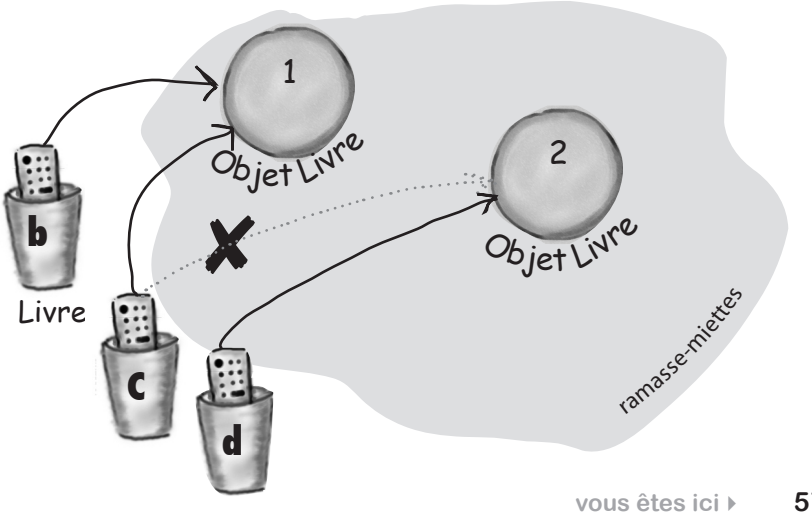
```
c = b;
```

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



Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Vie et mort sur le tas

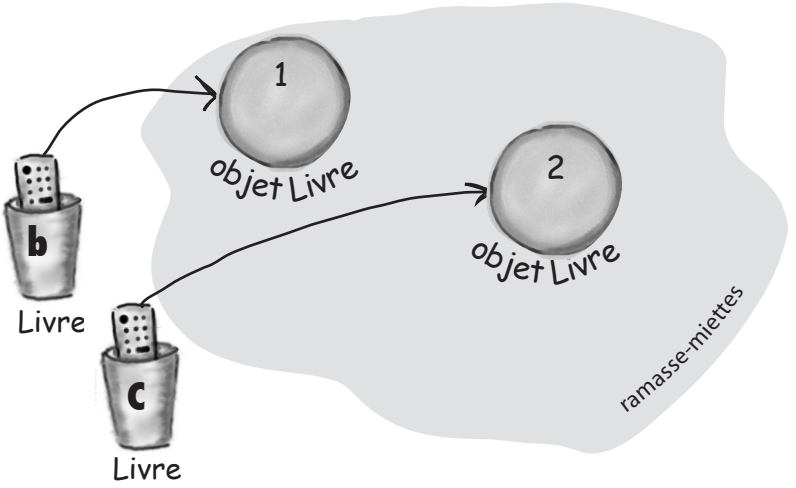
```
Livre b = new Livre();
Livre c = new Livre();
```

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.

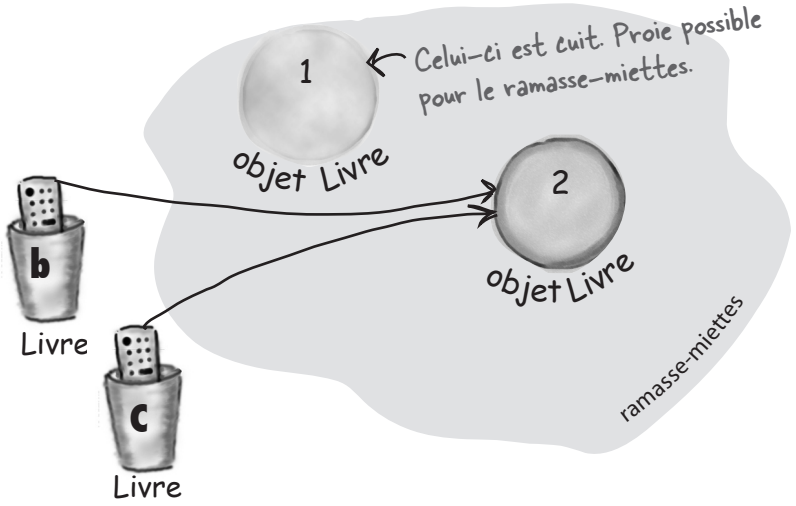
b ET c référencent le même objet. L'objet 1 est abandonné et devient candidat au ramasse-miettes (RM).

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.

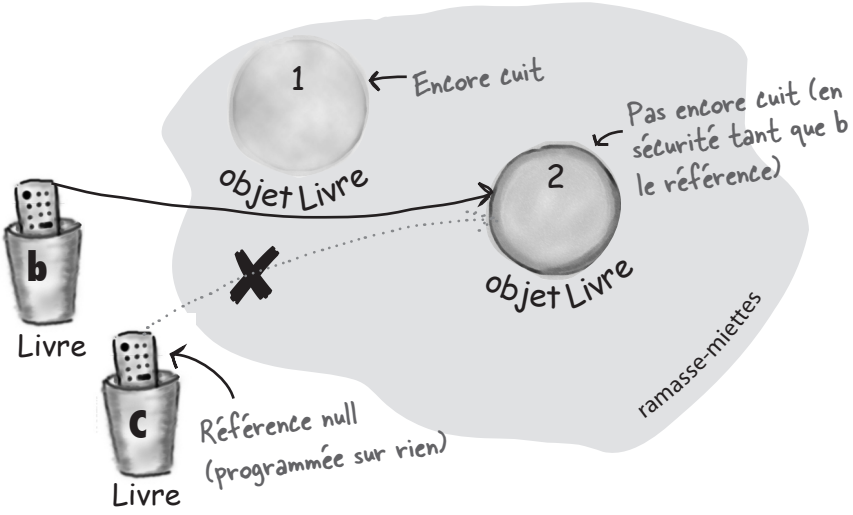
Objet 2 a toujours une référence active (b). Tant que c'est le cas, l'objet ne craint pas le ramasse-miettes.

Références actives : 1

Références *null* : 1

Objets accessibles : 1

Objets abandonnés : 1



Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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;
```

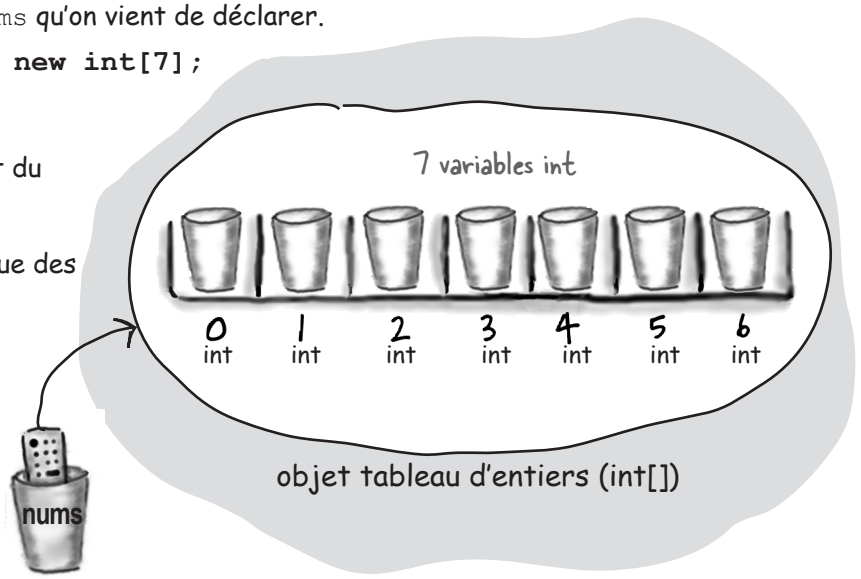
2 On crée un nouveau tableau d'entiers de 7 éléments, et on l'affecte à la variable `int[]` nommée `nums` qu'on vient de déclarer.

```
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 !

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

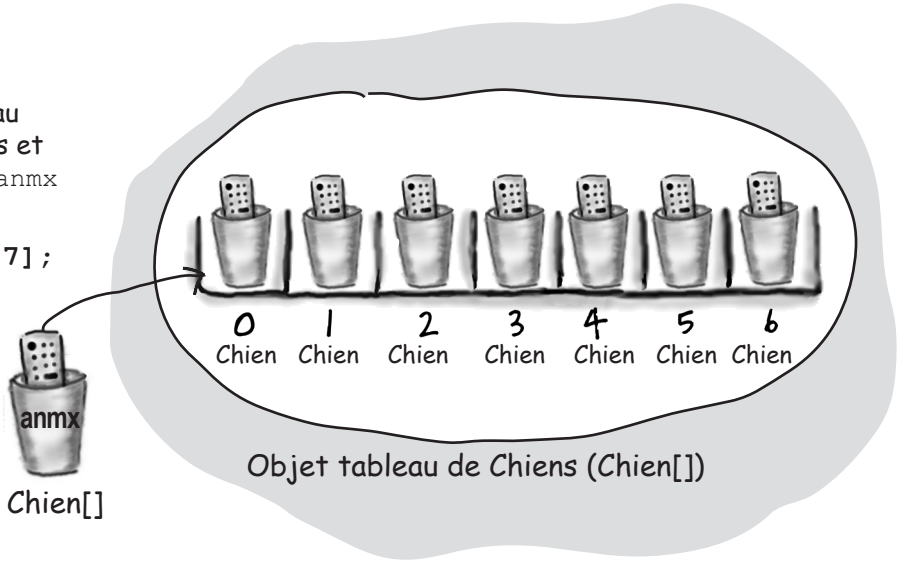
Créons un tableau de Chiens

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];
```



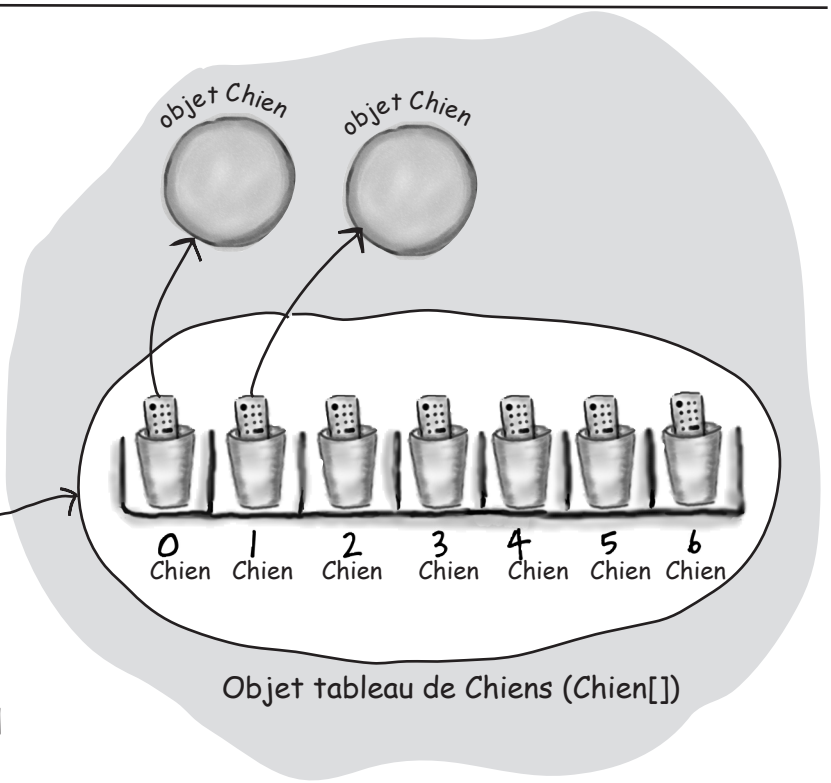
Qu'est-ce qui manque ?

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 ?

_____

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Chien
nom
aboyer() manger() poursuivreChat()

Contrôlez votre Chien

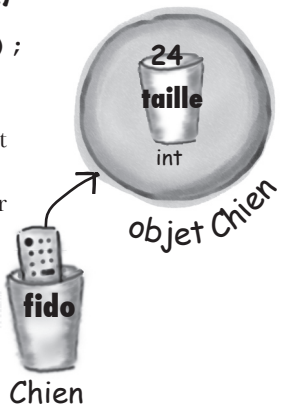
(avec une variable référence)

```
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.

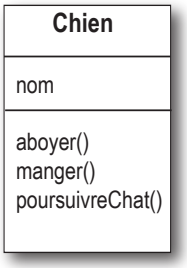
Par exemple, vous ne pouvez pas placer un Chat dans un tableau de Chiens (ce serait affreux si quelqu'un pensait que le tableau ne contient que des Chiens, demandait à chacun d'eux d'aboyer et découvrait avec horreur qu'il y a un Chat qui traîne par là). Vous ne pouvez pas non plus placer un `double` 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.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

```
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.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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");
    }
  }
}
```


Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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
```


Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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 !

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

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

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

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

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

28.0
30.0

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

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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).

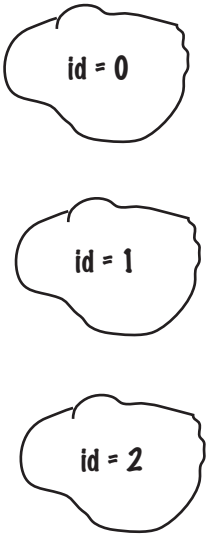
```
class CasseTete {
  int id = 0;
  public static void main(String [] args) {
    int x = 0;
    CasseTete [ ] ct = new CasseTete[5];
    while ( x < 3 ) {
      ct[x] = new CasseTete();
      ct[x].id = x;
      x = x + 1;
    }
    ct[3] = ct[1];
    ct[4] = ct[1];
    ct[3] = null;
    ct[4] = ct[0];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

Reliez les variables aux objets correspondants.

Vous n'aurez peut-être pas besoin de toutes les références.

Variables référence :

Objets CasseTete :



Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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?

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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 !

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

```
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;
  }
}
```

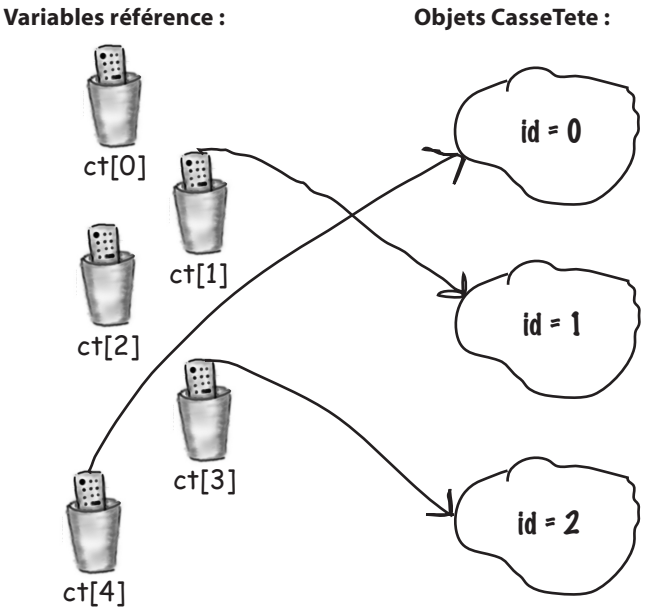
Fichier Edition Fenêtre Aide Bermuda

```
%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.)



Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

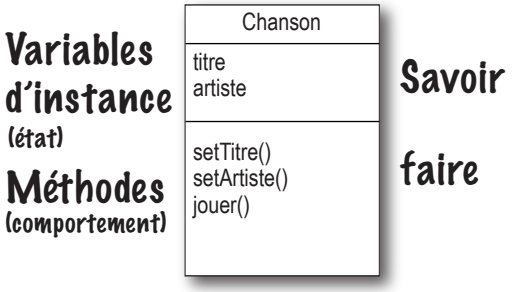
Comportement des objets

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.***

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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 ?

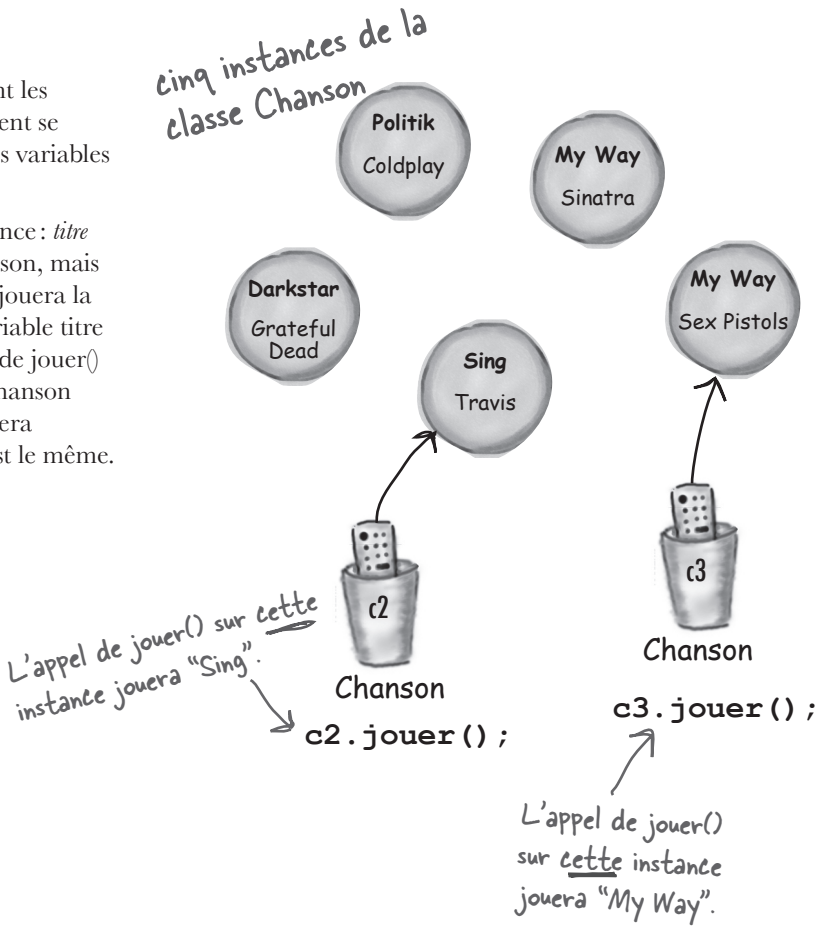
Eh bien… ***pour ainsi dire.****

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");
```



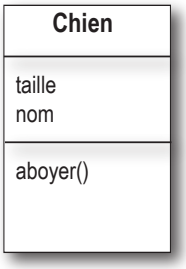
*Oui, encore une réponse d'une clarté éblouissante !

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Tous les chiens n'aboient pas pareil

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

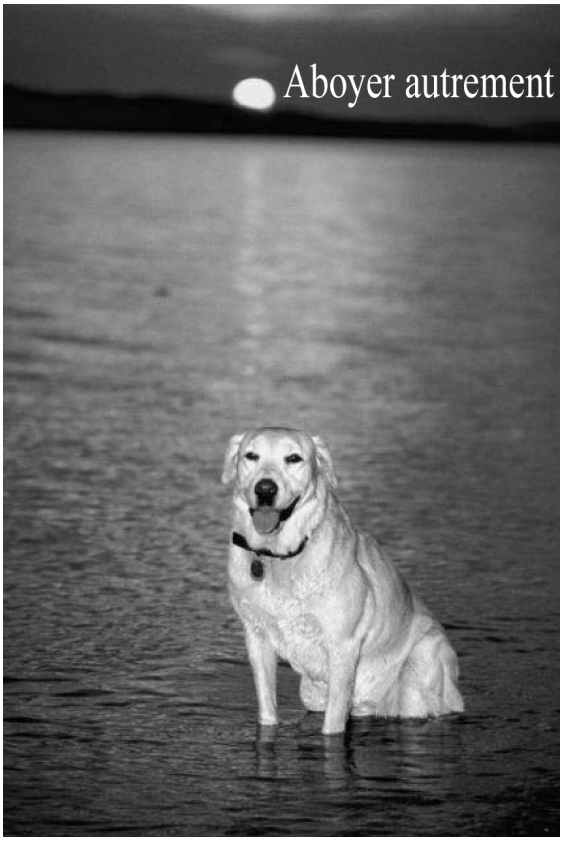
La classe Chien a une variable d'instance nommée taille, que la méthode *aboyer()* utilise pour décider quelle sorte de son produire.




```
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!
```



Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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.

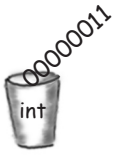
```
Chien c = new Chien();
c.aboyer(3);
```

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.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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 :

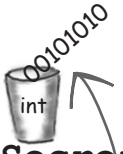
```
int direSecret() {
   return 42;
}
```

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.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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().

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

En Java on transmet les arguments par valeur

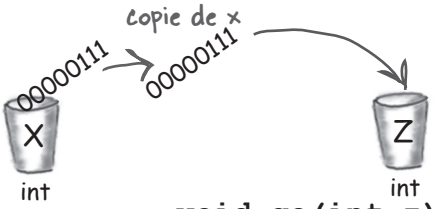
Autrement dit par copie.

```
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){ }
```

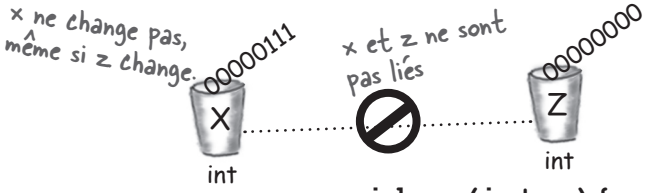
2 On déclare une méthode de type int. Elle a un paramètre nommé z.




```
foo.go(x);
```

```
void go(int 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.




```
void go(int z){
    z = 0;
}
```

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.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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

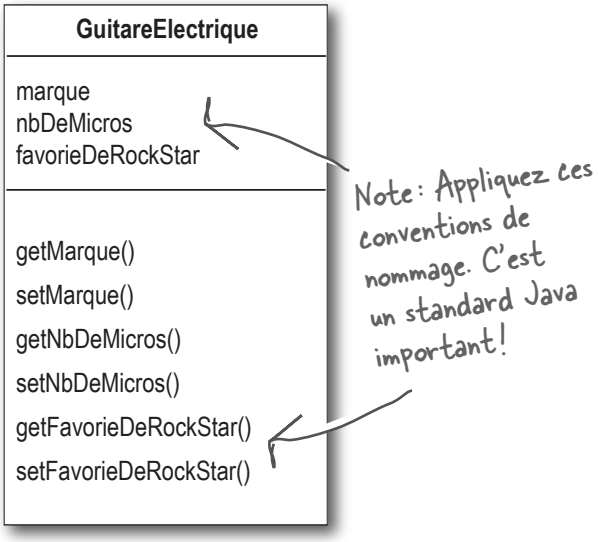
- Les classes définissent ce qu'un objet sait et ce qu'un objet fait.
- On nomme ce qu'un objet sait **variables d'instance** (son état).
- Et les **méthodes** correspondent à ce que l'objet fait (son comportement).
- Les méthodes utilisent les variables d'instance pour que des objets du même type puissent 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.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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;
   }
}
```

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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.

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();

```
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();
  }
}
```

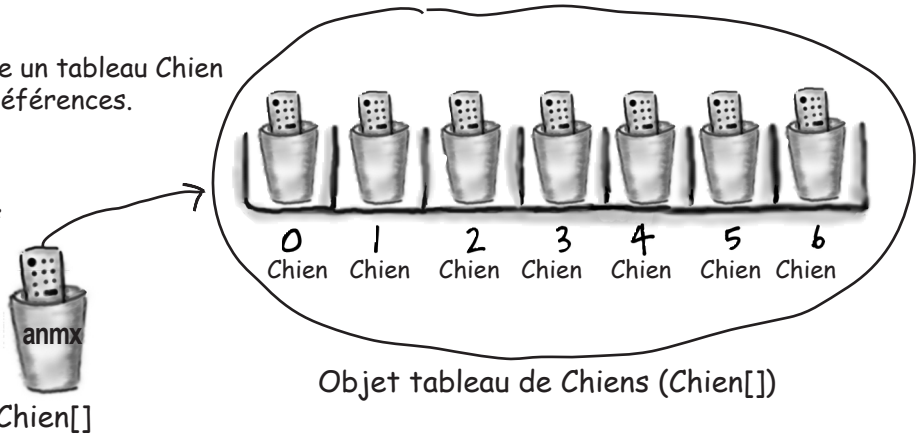

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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];
```



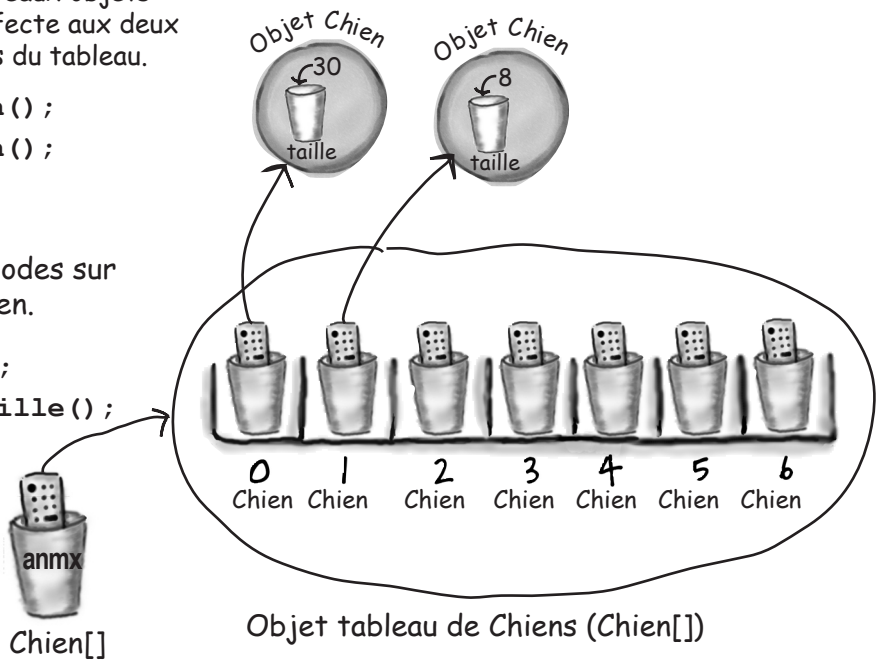
---

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);
```



Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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 ?

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

```
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.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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.

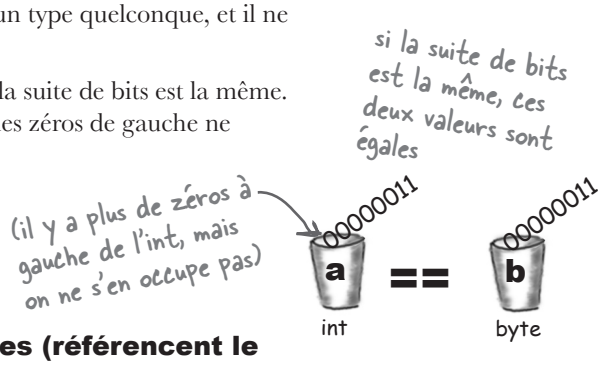
(Comme deux objets String distincts qui représentent tous deux les caractères de « Fred ».)

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.

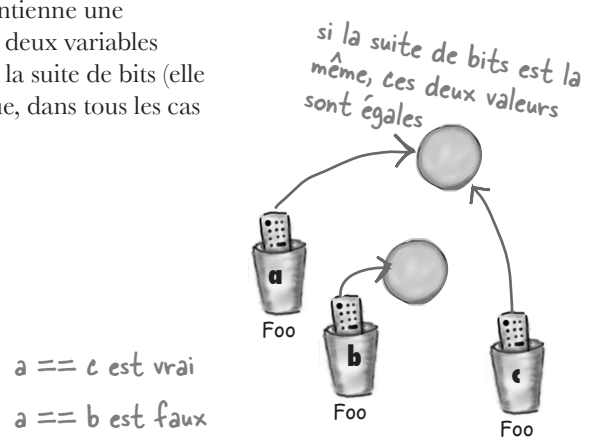
```
int a = 3;
byte b = 3;
if (a == b) { // vrai }
```



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 }
```



Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Aide-mémoire

Attention, passage par valeur

Laver le chat

threads attendre() notifier()

Une rose est une rose,
ce poème est en prose,
transmettre une valeur
c'est transmettre une copie.

Ah, vous pouvez faire mieux ? Essayez. Remplacez la deuxième ligne par un vers de votre cru. Mieux encore : remplacez le tout par vos propres mots et vous n'oublierez *jamais*.

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);
```


Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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);
  }
}
```


Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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.

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. _____

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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
```


Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

La piscine

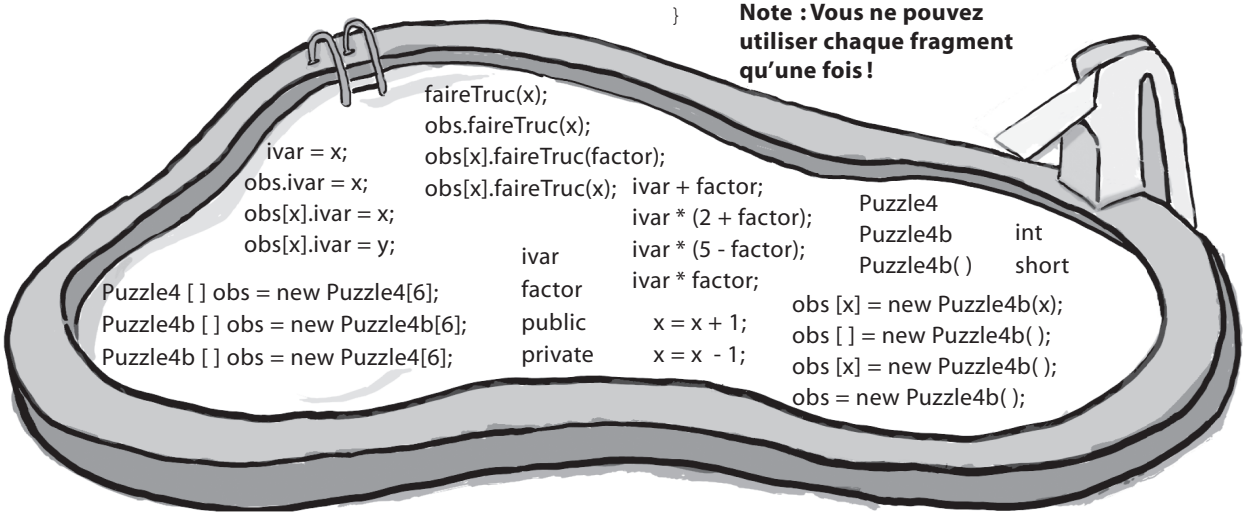
Votre ***tâch***e consiste à prendre des fragments de code dans la piscine et à les remettre dans le programme à la place des blancs. Vous ne pouvez **pas** utiliser le même fragment deux fois, et vous n'avez pas besoin de les utiliser tous. Votre ***but*** est de créer une classe qui se compile, s'exécute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



Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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 ?

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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( ).

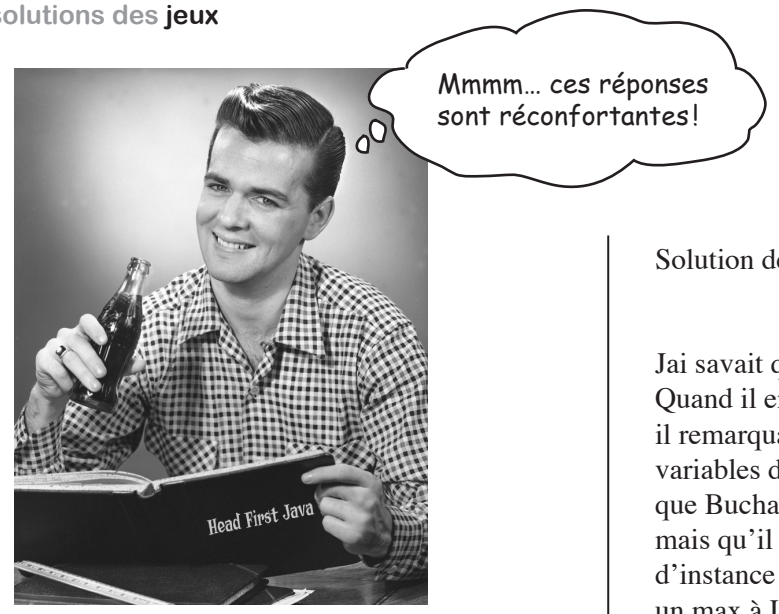
B les méthodes utilisent les variables d'instance

```
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**

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>




```
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.

Candidats :	**Résultat possible :**
`x < 9` `indice < 5`	`14 7`
	`9 5`
`x < 20` `indice < 5`	`19 1`
	`14 1`
	`25 1`
`x < 7` `indice < 7`	`7 7`
	`20 1`
`x < 19` `indice < 1`	`20 5`

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Jouons à la bataille navale : « Coulez un Point Com »

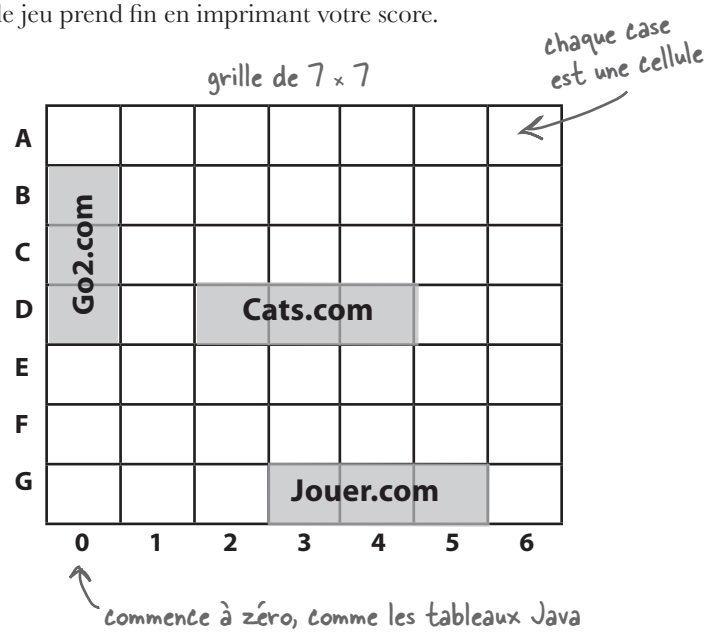
Vous jouez contre l'ordinateur, avec des règles légèrement différentes de celles de la vraie bataille navale. Vous ne placez vous-même aucun bateau, mais vous devez couler les siens en jouant le moins de coups possible.

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  : (
```


Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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

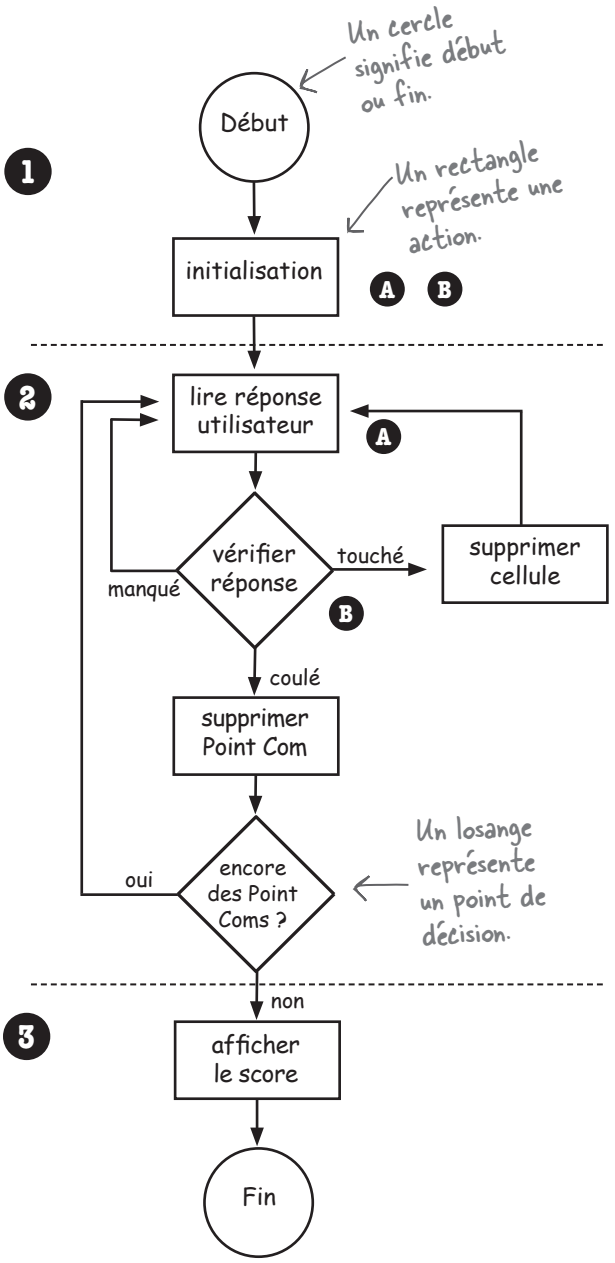
Répétez les étapes suivantes tant qu'il reste des Point Coms :

- **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 !

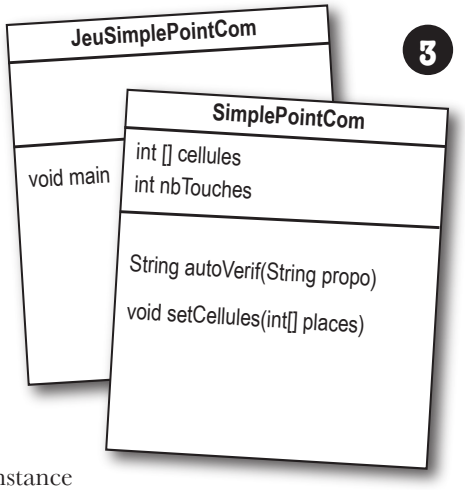
Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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
```


Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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 SimplePointCom.

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 JeuSimplePointCom

- ❑ écrire pré-code
- écrire tests [non]
- ❑ écrire code Java final

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

SimplePointCom
int [] cellules int nbTouches
String autoVerif(String propoUtil) void setCellules(int[ ] placeCellules)

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

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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 »).

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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
```

convertir la chaîne en entier

créer une variable pour contenir le résultat retourné. lui affecter "manqué" comme valeur par défaut

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!

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

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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 !

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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
```


Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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
```


Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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).

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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.

DECLARER une variable pour contenir le nombre de coups, l'initialiser à 0

CREER un objet SimplePointCom

CALCULER un nombre aléatoire compris entre 0 et 5

CREER un tableau d'int avec les 3 cellules

APPELER setCellules sur l'objet point com

DECLARER un booléen estVivant

TANT QUE le point com est toujours vivant

LIRE la saisie

// la VERIFIER

APPELER autoVerif() sur le point com

INCREMENTER nbPropo

SI resultat est "coulé"

POSITIONNER estVivant à false

AFFICHER le nombre de coups

```
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()
```

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

créer l'objet point com

générer un nombre aléatoire pour la première cellule, et l'utiliser comme point de départ du tableau

affecter le point com à ses emplacements (le tableau)

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.

lire la saisie de l'utilisateur

demander au point com de vérifier ; mettre le résultat retourné dans un String

incrémenter le nombre de coups

Est-il "coulé" ? Si oui, positionner estVivant à false (ce qui permet de sortir de la boucle) et afficher le nombre de coups.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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 Code prêt à l'emploi 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.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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)

ici le code de l'action à répéter (le corps)

opérateur d'incrémentation

```
for(int i = 0; i < 100; i++){ }
```

initialisation — test booléen — expression d'itération

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*

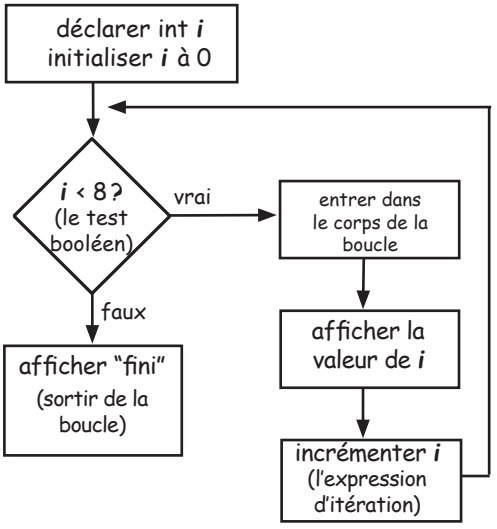
C'est là que vous spécifiez une ou plusieurs actions qui doivent avoir lieu à chaque passage dans la boucle. N'oubliez pas qu'elles se produisent à la *fin* de chaque itération.

répétez 100 fois :

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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

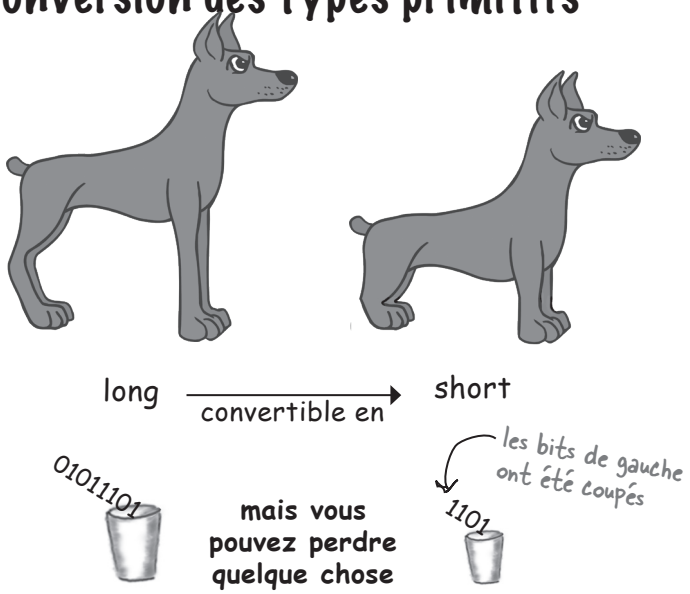
Cette partie déclare et initialise une variable qui sera utilisée dans le corps de la boucle. À chaque itération, celle-ci contiendra un autre élément de la collection. Le type de cette variable doit être compatible avec les éléments du tableau ! Par exemple, vous ne pouvez pas déclarer une variable 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.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Conversion des types primitifs



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";

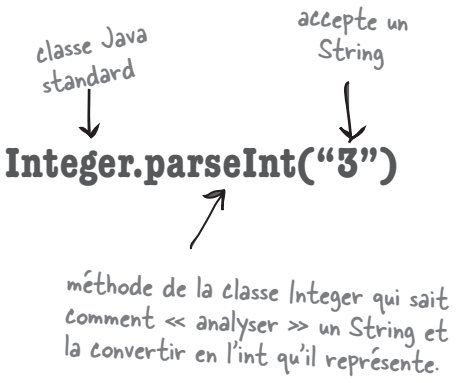
int x = 2;

if (x == num) // horrible explosion !

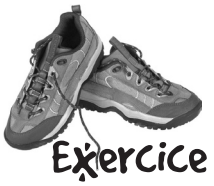
Si vous essayez de compiler, le compilateur se moque de vous :

```
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.



Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>



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
```


Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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--) {
```

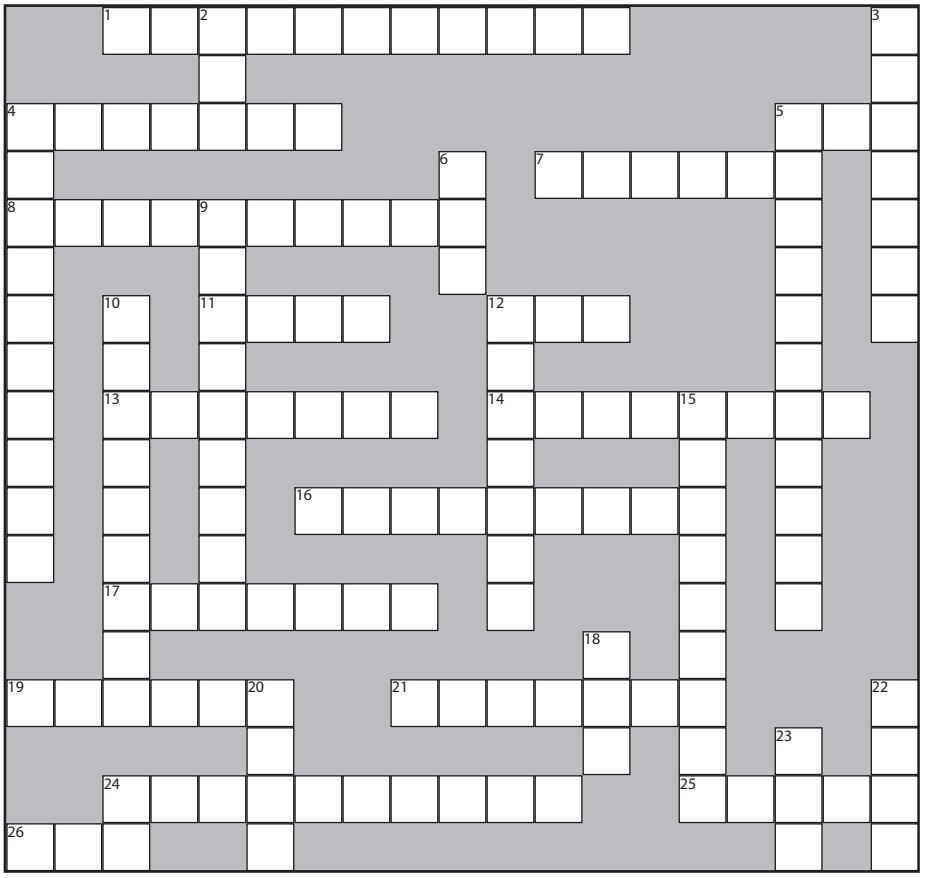
```
for(int x = 0; x < 4; x++) {
```

```
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
```


Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>



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

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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--) {

        [                    ]   ← emplacement du code candidat

        y = y - 2;
        if (x == 6) {
          break;
        }
        x = x + 3;
      }
      y = y - 2;
    }
    System.out.println(x + " " + y);
  }
}
```

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

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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 :

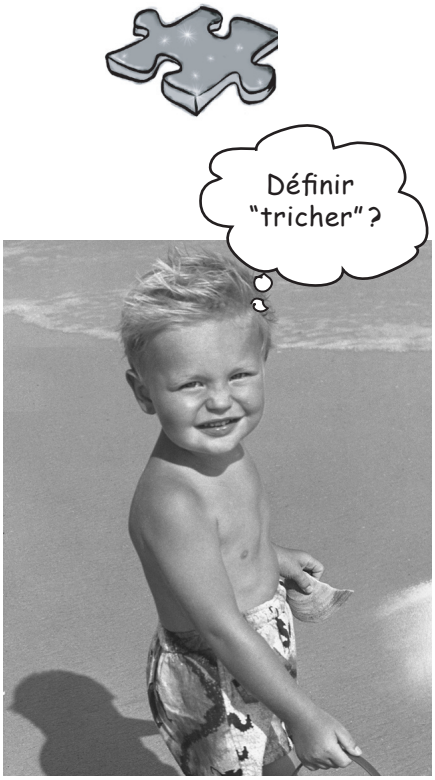
```
class MultiFor {

  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
```


Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>



			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`

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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!

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Comment allons-nous le corriger ?

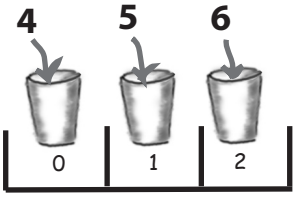
Il nous faut un moyen de savoir si une cellule a déjà été touchée. Nous allons explorer quelques possibilités, mais commençons par examiner ce que nous savons déjà…

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)



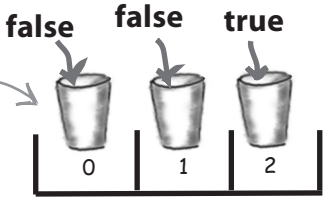
la variable d'instance (tableau) qui contient les cellules occupées par le PointCom. Ici, il contient les 3 valeurs 4, 5 et 6. Ce sont les nombres que l'utilisateur doit deviner.

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.

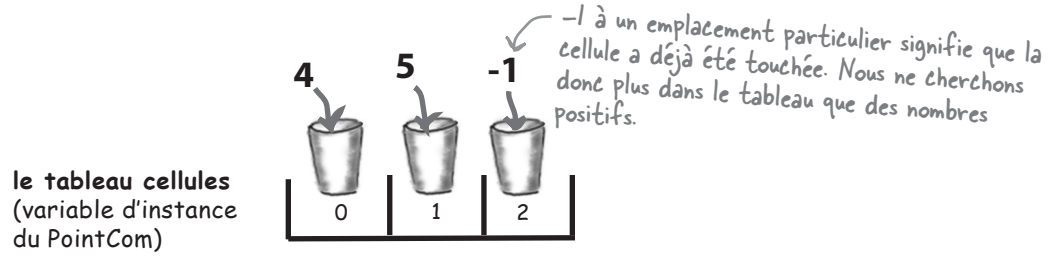
Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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

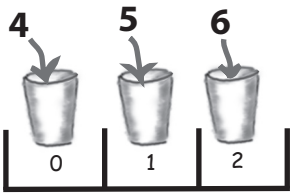
L'option deux est un peu plus élégante que la première, mais elle n'est pas très efficace. Il faut toujours parcourir les trois emplacements (positions d'indice), même si un ou plusieurs d'entre eux sont déjà invalides parce qu'ils ont été touchés (et ont la valeur -1). On doit pouvoir trouver encore mieux…

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

③ Option trois

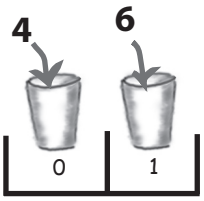
Nous détruisons les cellules à mesure qu'elles sont touchées, puis nous réduisons la taille du tableau. Sauf qu'un tableau ne peut pas changer de taille, ce qui nous oblige à créer un **nouveau** tableau et à y copier les cellules qui restent dans l'ancien.

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

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Ce que vous pouvez faire avec une ArrayList

Avez-vous vu la nouvelle syntaxe de <Oeuf> ? Cela signifie «faire une liste des objets Oeufs».

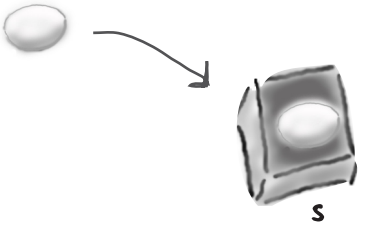
Un nouvel objet ArrayList est créé sur le tas. Il est petit parce qu'il est vide.

1. En créer une

```
ArrayList<Oeuf> maListe = new ArrayList();
```

2. Y placer quelque chose

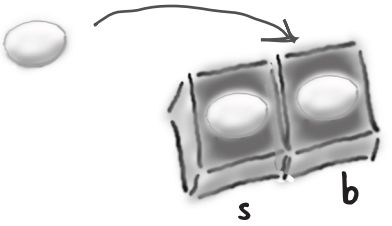
```
Oeuf s = new Oeuf();

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

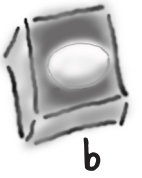
7. Savoir si elle est vide

```
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 !

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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);`	

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

il n'y a pas de
Questions stupides

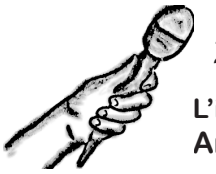
Q : L'ArrayList est super, mais comment aurais-je pu savoir qu'elle existait ?

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.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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*.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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!

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

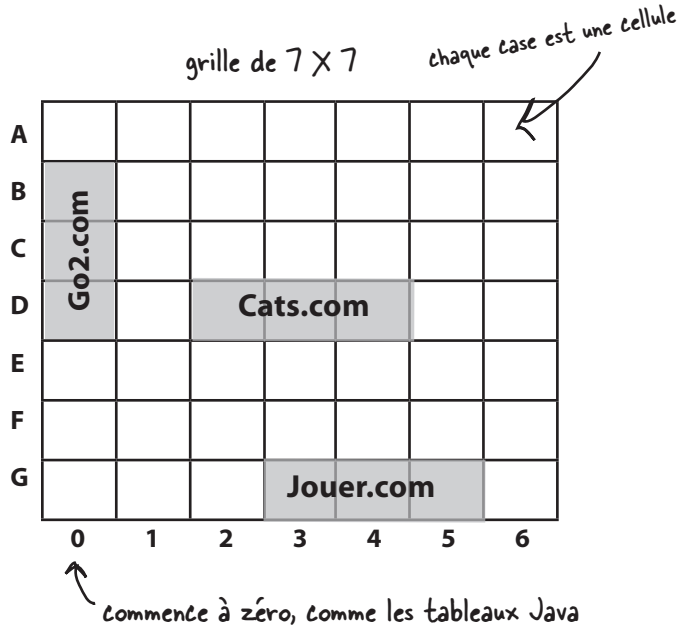
Construisons le VRAI jeu : « Coulez un Point Com »

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  : (
```


Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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

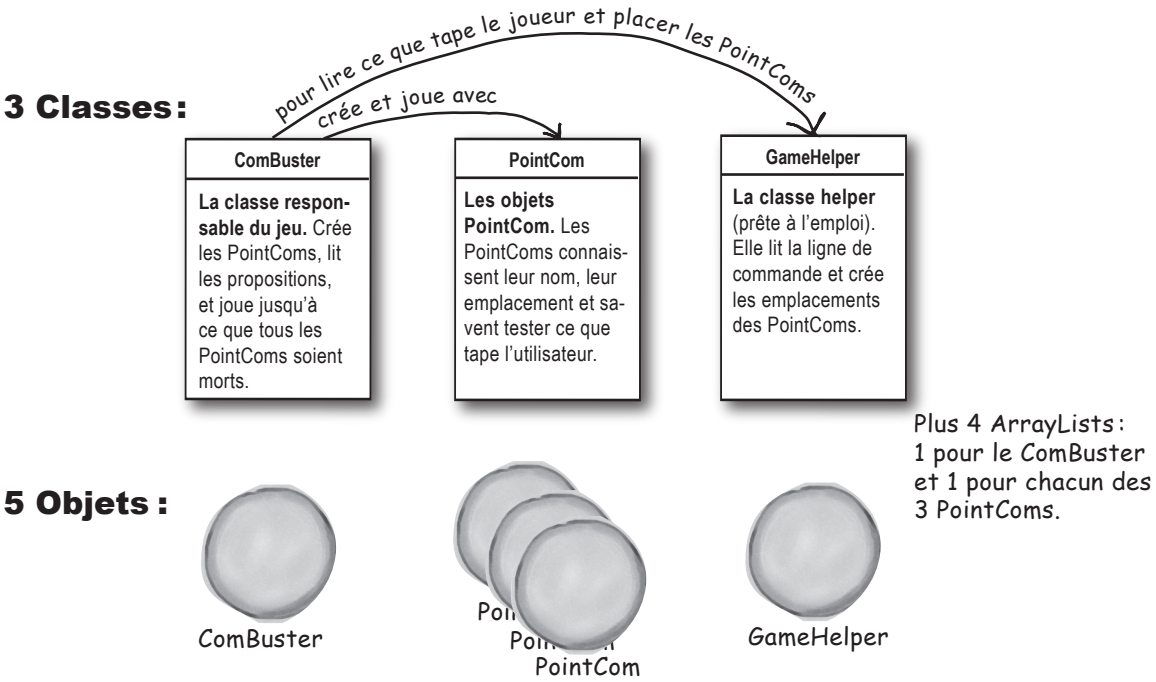
- **Ajouter un nom de variable** qui contiendra le nom du PointCom («Jouer.com», «Go2.com», etc.) pour que chaque PointCom puisse afficher son nom quand il est coulé (voir l'écran de la page de gauche).

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...

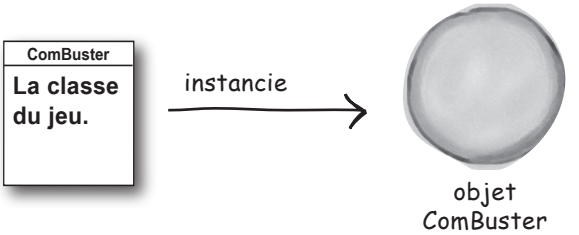
- **Placer les trois PointComs dans une grille et non dans une ligne.**
  Cette étape est beaucoup plus complexe qu'auparavant car nous devons placer les PointComs au hasard. Comme nous ne sommes pas là pour nous occuper des fonctions mathématiques, nous plaçons l'algorithme qui place les PointComs dans 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.



Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

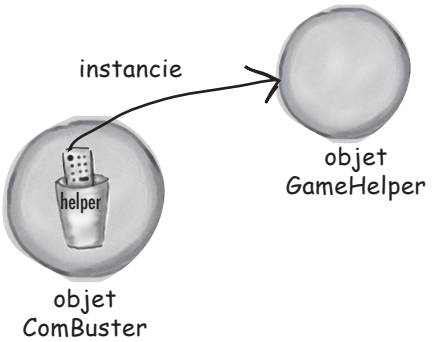
Qui fait quoi dans le jeu ComBuster
(et quand)

1



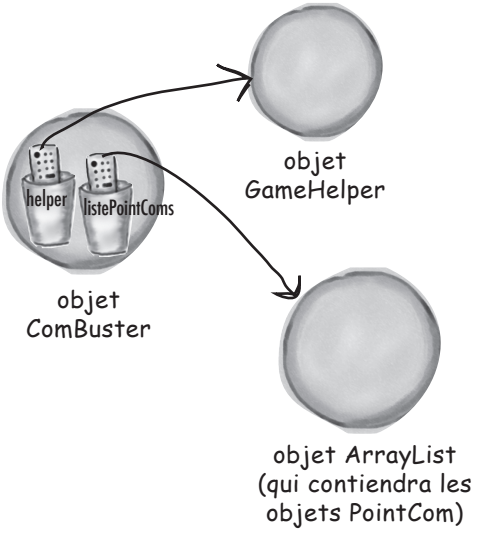
La méthode main() de la classe ComBuster instancie l'objet ComBuster qui fait fonctionner tout le jeu.

2



L'objet ComBuster (le jeu) instancie un objet GameHelper, l'objet qui aide le jeu à faire son travail.

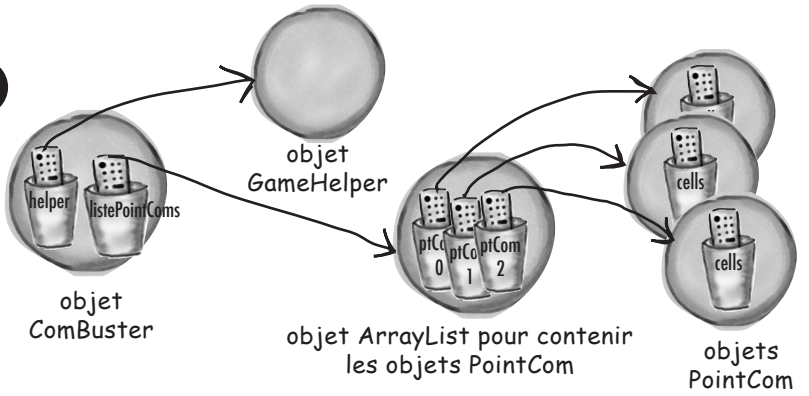
3



L'objet ComBuster instancie une ArrayList qui contiendra les 3 objets PointCom.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

4

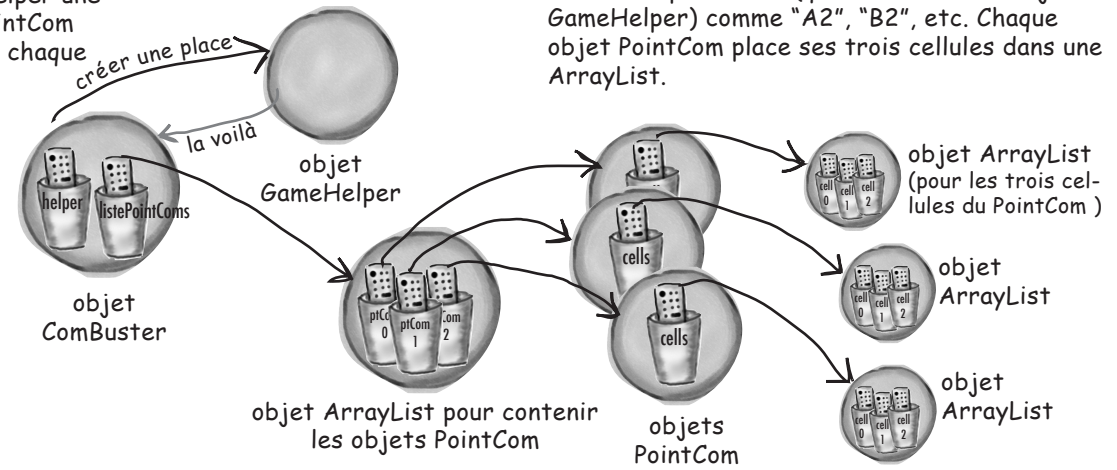


L'objet ComBuster crée trois objets PointCom (et les place dans l'ArrayList).

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.

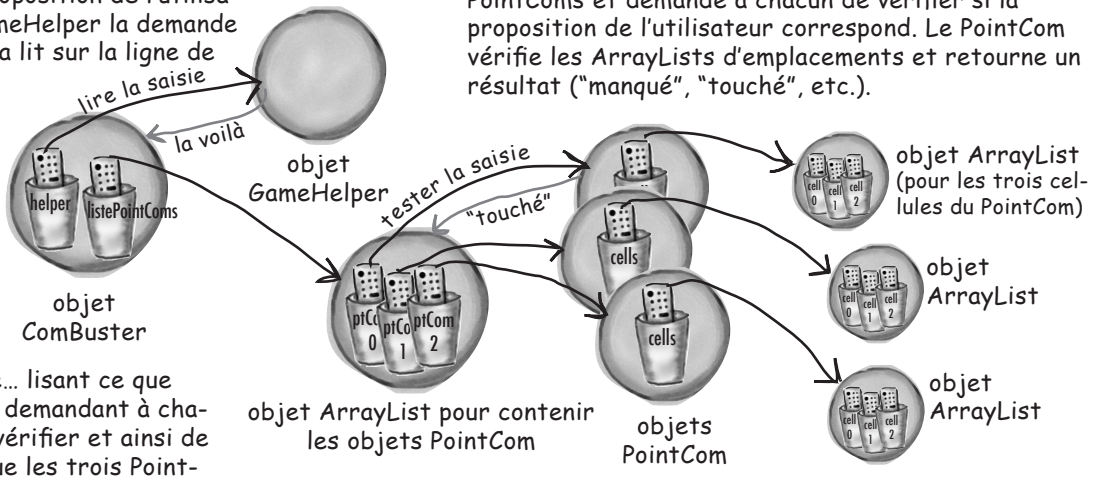
5



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.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

pré-code | code test | **vrai code**

Quoiqu'il arrive, ne tournez PAS la page !

Pas avant d'avoir fini l'exercice.

Notre version se trouve page suivante.

```
  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)

— 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

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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 ()

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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
}
```


Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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.

Retourne 'manqué' ou 'touché' ou 'coulé'.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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.

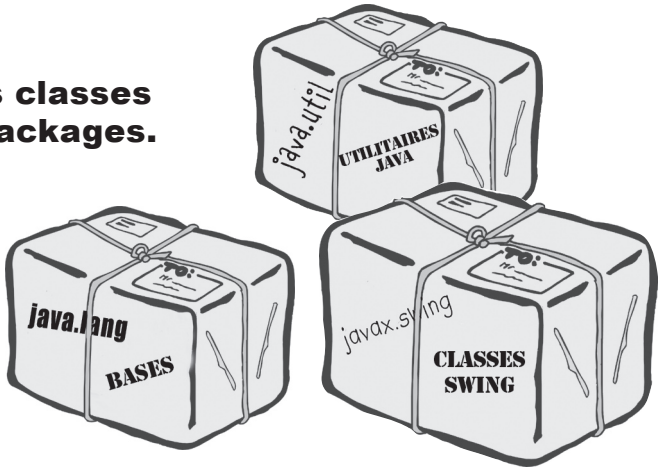
```
    }

    // System.out.println("\n");

    return cellAlpha;
  }
}
```


Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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**.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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 ?

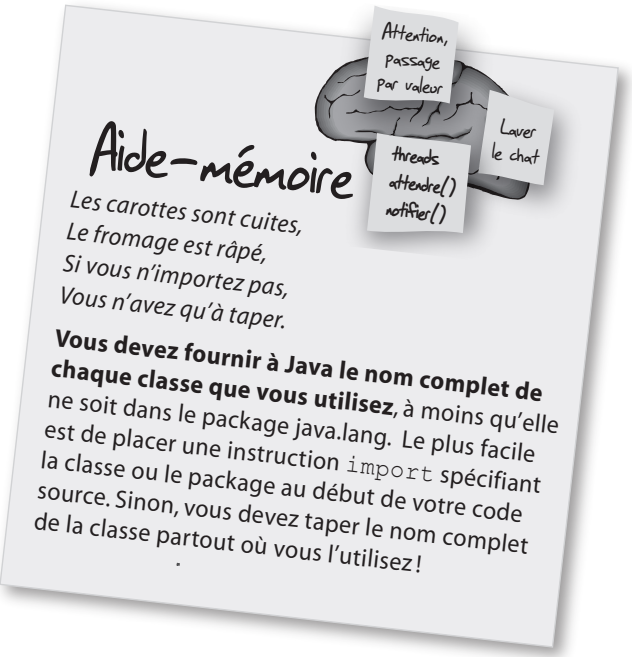
R : Peut-être êtes-vous un programmeur C ? Un `import` n'est pas un `include`. La réponse est donc non et non. Répétez après moi : « Une instruction `import` vous évite de taper ». C'est tout. N'ayez pas peur que votre code soit encombré ou ralenti par de trop nombreux imports. Un `import` n'est que 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.



Aide-mémoire

Les carottes sont cuites,
Le fromage est râpé,
Si vous n'importez pas,
Vous n'avez qu'à taper.

Vous devez fournir à Java le nom complet de chaque classe que vous utilisez, à moins qu'elle ne soit dans le package java.lang. Le plus facile est de placer une instruction `import` spécifiant la classe ou le package au début de votre code source. Sinon, vous devez taper le nom complet de la classe partout où vous l'utilisez !

Encore une fois, dans le cas très improbable où ce ne serait pas encore tout à fait clair :

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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

- Julie. 31 ans, manucure

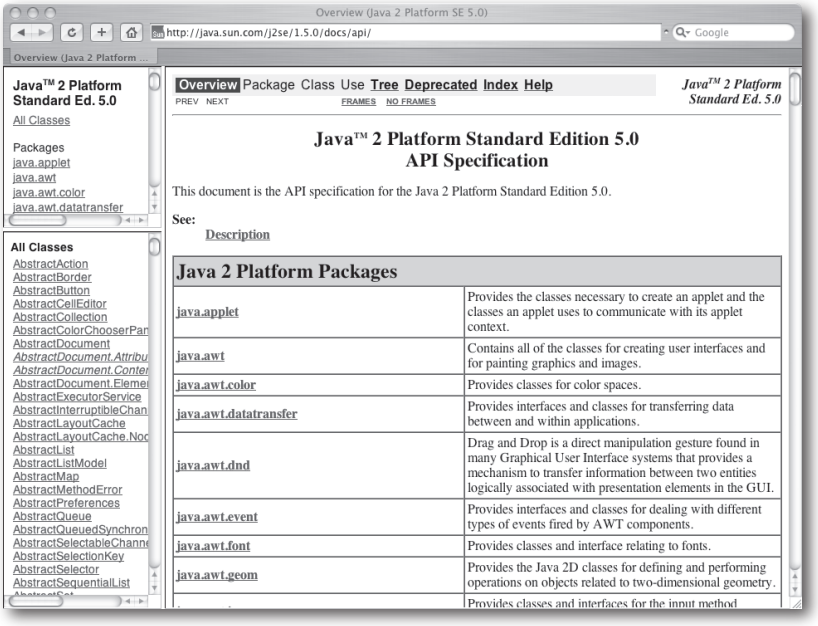
Comment découvrir l'API

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



Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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.

Object — Date; 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);
```

méthodes (et autres éléments dont nous parlerons plus loin)

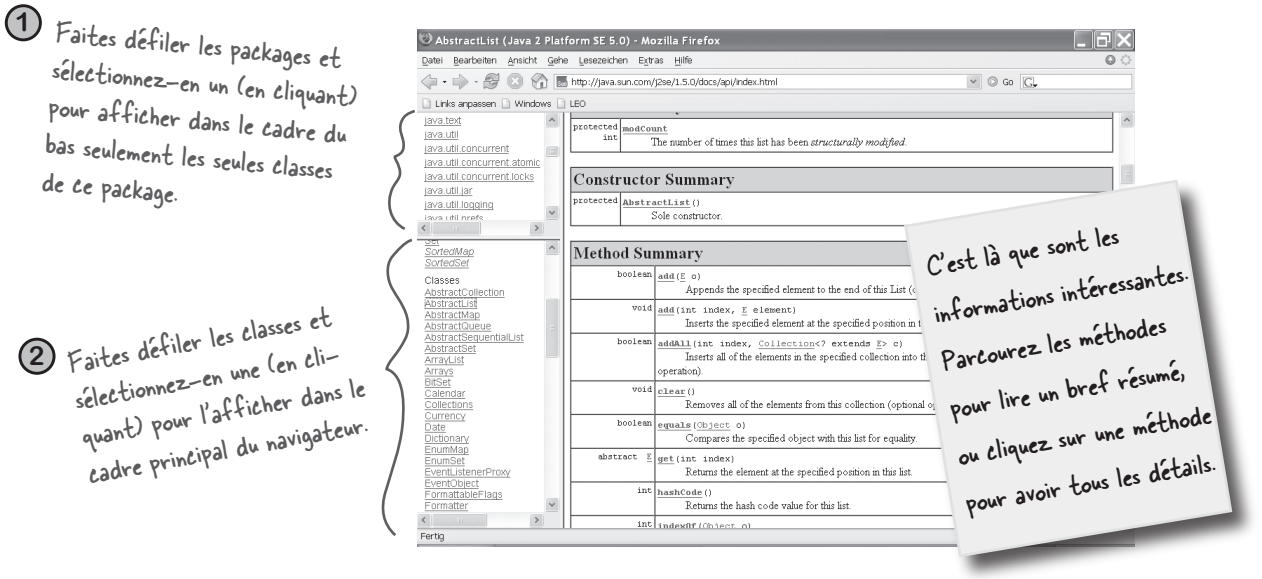
Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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.



Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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

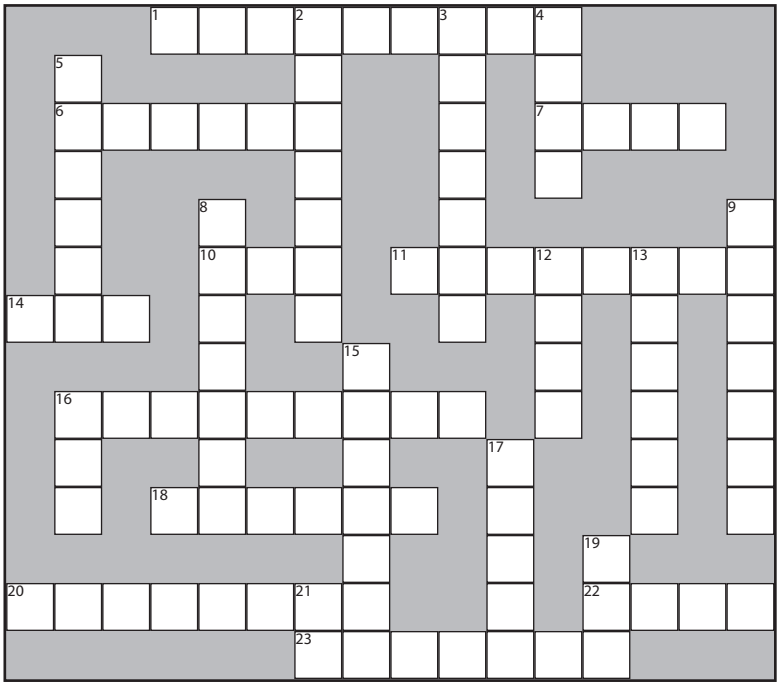
```
% java ArrayListMagnet
zéro  un  deux  trois
zéro  un  trois  quatre
zéro  un  trois  quatre  4.2
zéro  un  trois  quatre  4.2
```


Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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 :

Verticalement
1. 8 variétés
7. Le____ et l'infini
11. Pensez à ArrayList
23. Un tableau en a au moins 2

Horizontalement
2. Peut être redéfinie
12. Une variable a un nom et un ____
19. Accesseur

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

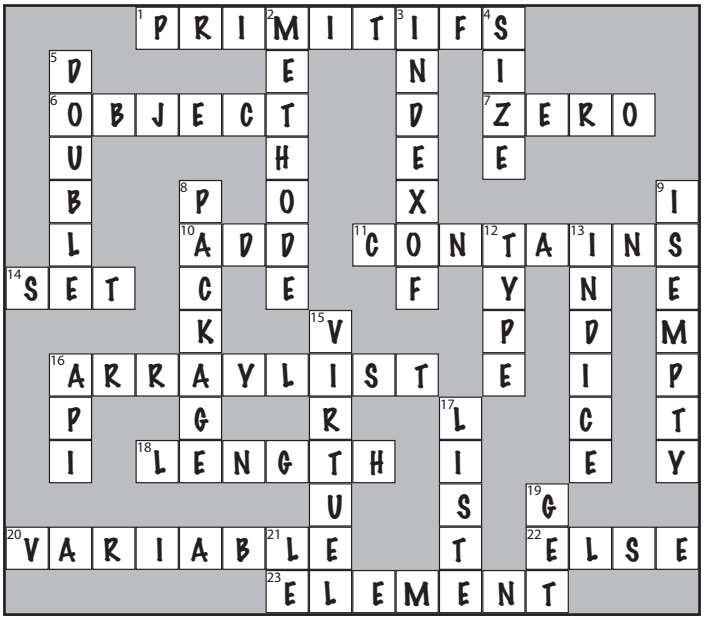
Solutions des exercices

```
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
```


Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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. _____

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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 !

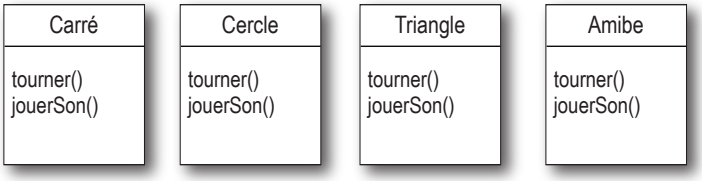
Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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.

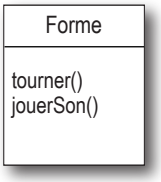
BERNARD : Tu as du code dupliqué ! Il y a une procédure tourner dans tes quatre trucs. C'est stupide comme conception. Ça t'oblige à maintenir quatre « méthodes » tourner différentes. Comment peux-tu justifier ça ?

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

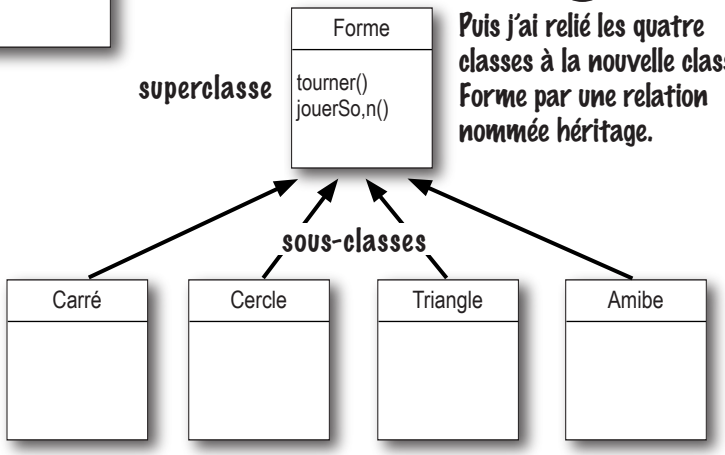


1 J'ai regardé ce que les quatre classes avaient en commun.

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.*

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Et la méthode tourner() de l'amibe ?

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

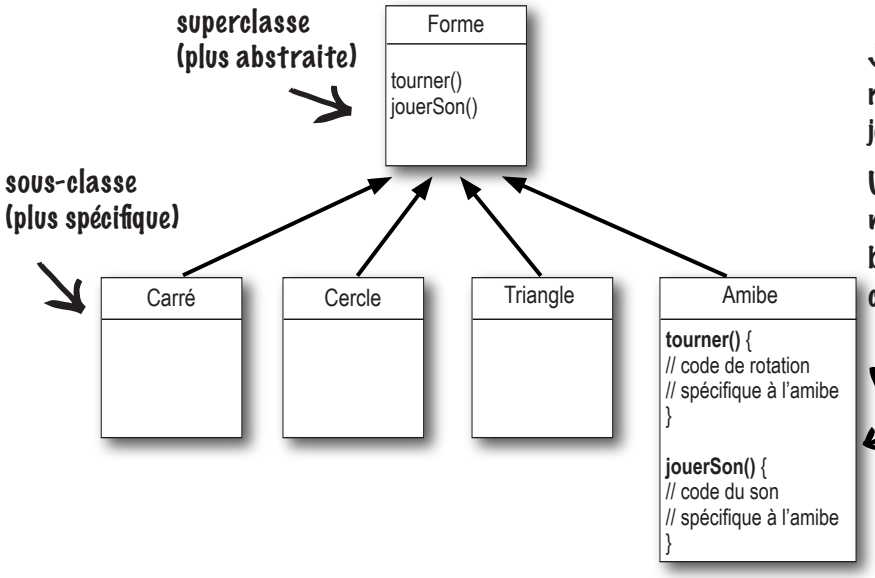
KEVIN : Des **méthodes.**

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.

Écraser maintenant

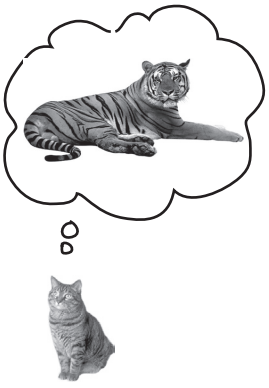
Me demander



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 ?

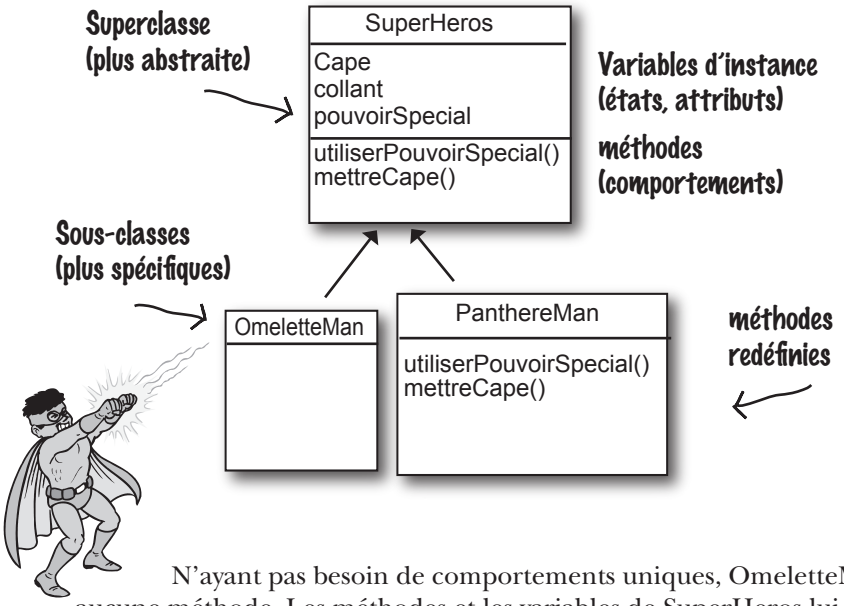
Réfléchissez-y. *Avant* de tourner la page.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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.

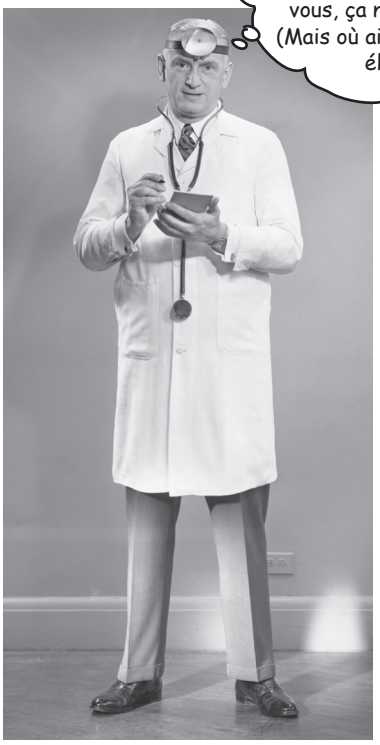
Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Un exemple d'héritage :

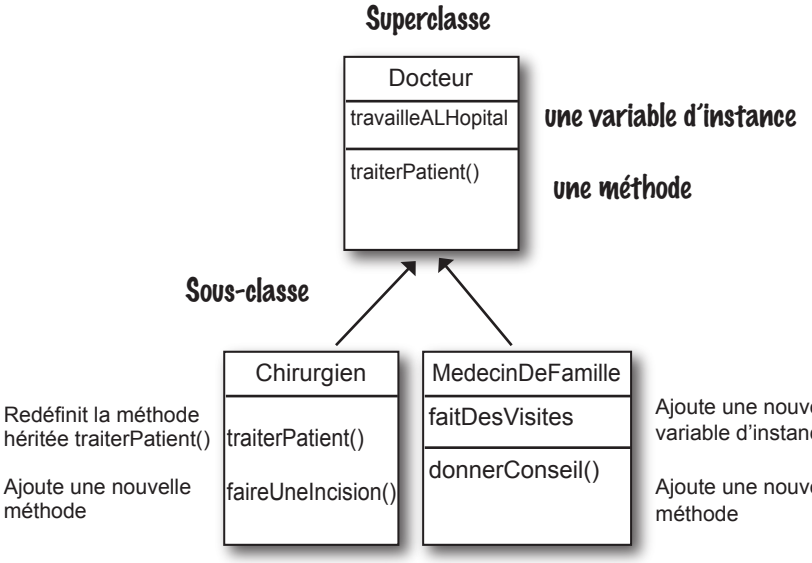
```
public class Docteur {

    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() ? _____

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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.

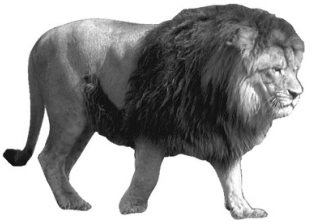
Et nous voulons que d'autres programmeurs puissent ajouter de nouvelle sortes d'animaux au programme à n'importe quel moment.

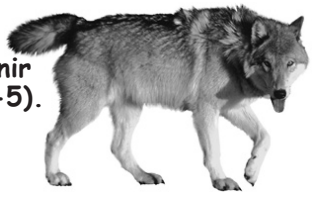
Nous devons d'abord déterminer les caractéristiques abstraites que tous les animaux partagent, puis intégrer ces caractéristiques dans une classe dont tous ces animaux puissent dériver.

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).





Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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.

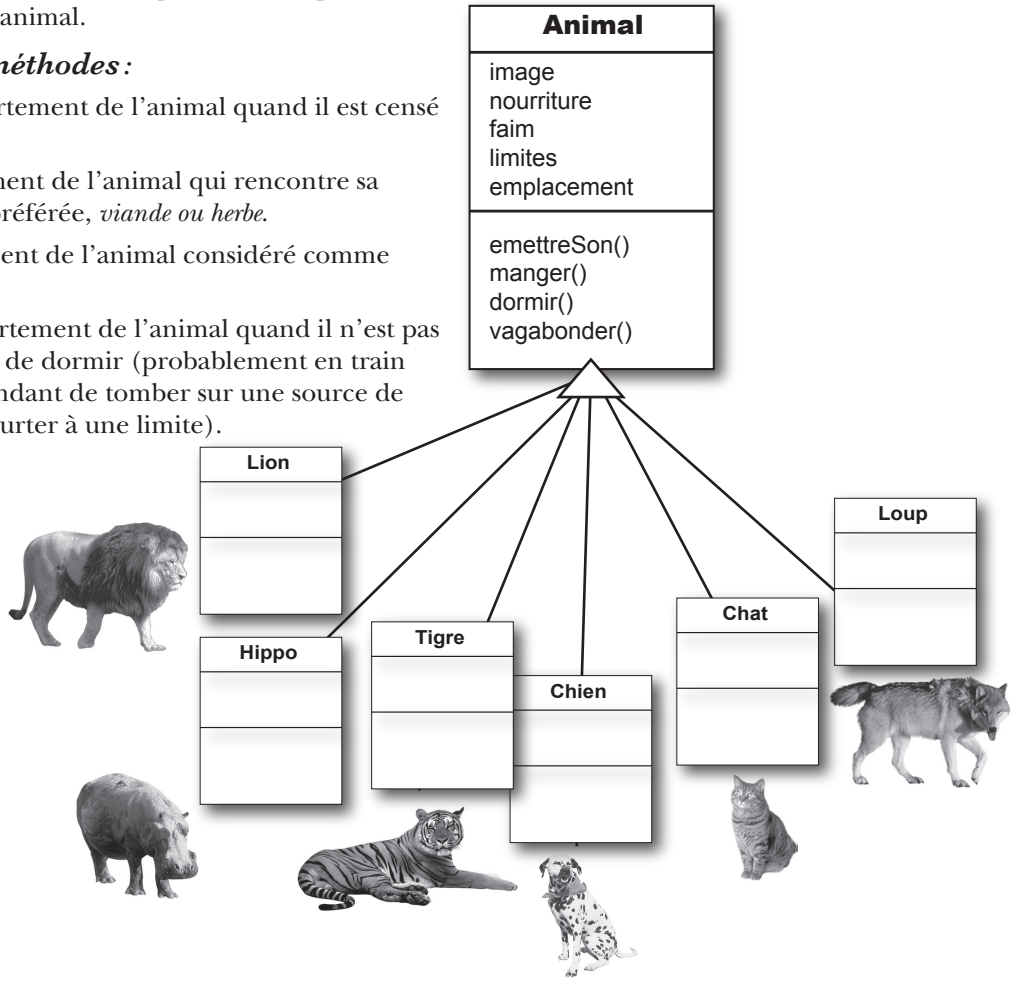
vagabonder() – comportement de l'animal quand il n'est pas en train de manger ou de dormir (probablement en train d'errer ça et là en attendant de tomber sur une source de nourriture ou de se heurter à une limite).

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.



Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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 ?

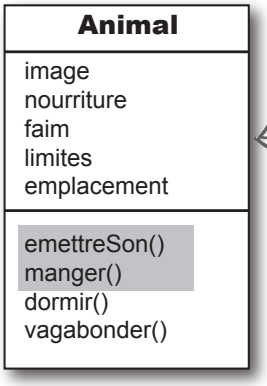
3 Déterminer si une sous-classe nécessite des comportements (implémentations de méthodes) spécifiques à ce type particulier.

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.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Chercher d'autres occasions d'héritage

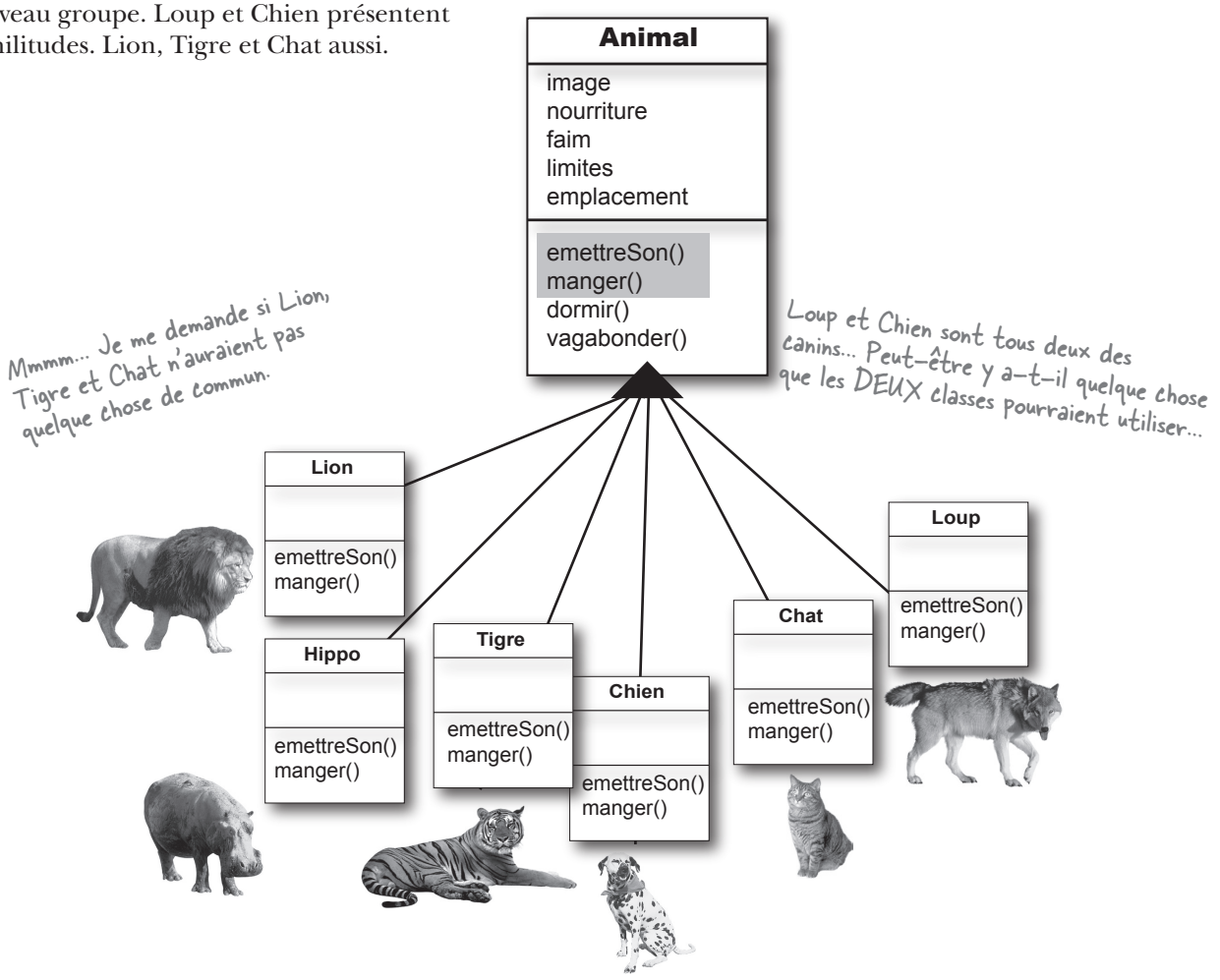
L'arborescence des classes commence à prendre forme. Chaque sous-classe redéfinit les méthodes *emettreSon()* et manger*()*, de sorte qu'il est impossible de confondre le ouah d'un Chien avec le miaou d'un Chat (nos excuses aux deux parties) et qu'un Hippo ne mangera pas comme un Lion.

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.



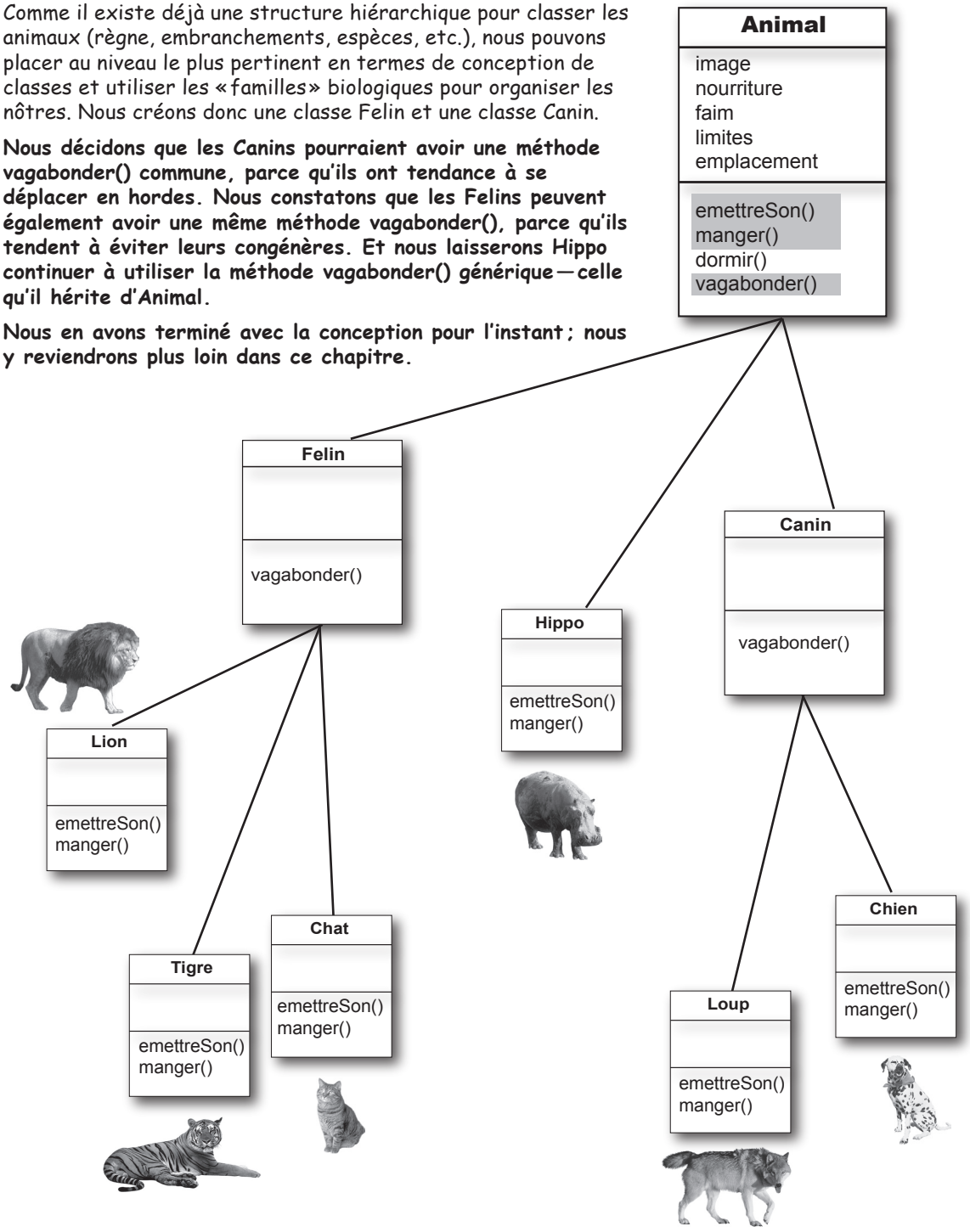
Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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.



Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Concevoir une hiérarchie d'héritage

Classe	Superclasse	Sous-classes
Vêtements	—	Short, Chemise
Short	Vêtements	
Chemise	Vêtements	

Table d'héritage

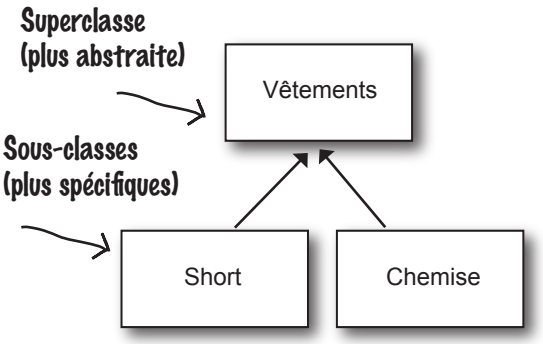


Diagramme de classes

À vos crayons

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.

Tracez votre diagramme ici.

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***.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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.

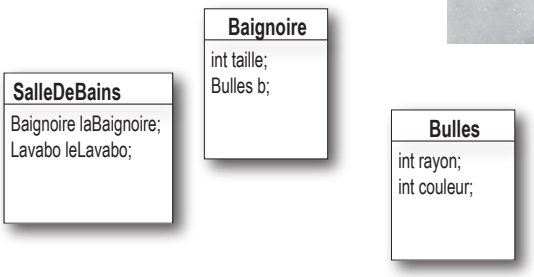
Chirurgien EST-UN Docteur. Tout va bien.

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.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Comment savoir si la hiérarchie d'héritage est correcte ?

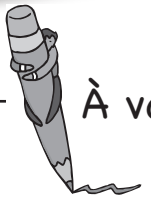
Nous n'avons de toute évidence pas épuisé le sujet, mais nous reviendons sur d'autres questions OO au chapitre suivant (dans lequel nous approfondirons et améliorerons le travail de conception effectué dans *ce* chapitre).

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.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

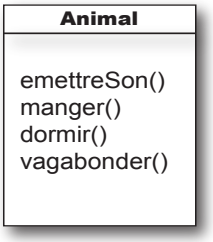
L'héritage garantit que les classes regroupées sous un certain supertype possèdent toutes les méthodes de ce supertype.*

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.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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

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

1 Déclarer une variable référence

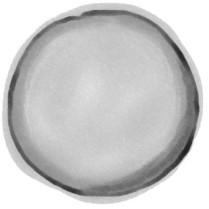
```
Chien monChien = new Chien();
```

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();
```

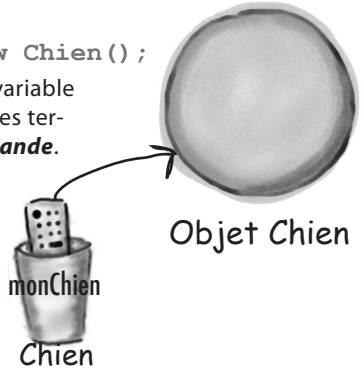
Dit à la JVM d'allouer de l'espace à un nouvel objet Chien sur le tas.



3 Lier l'objet et la référence

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

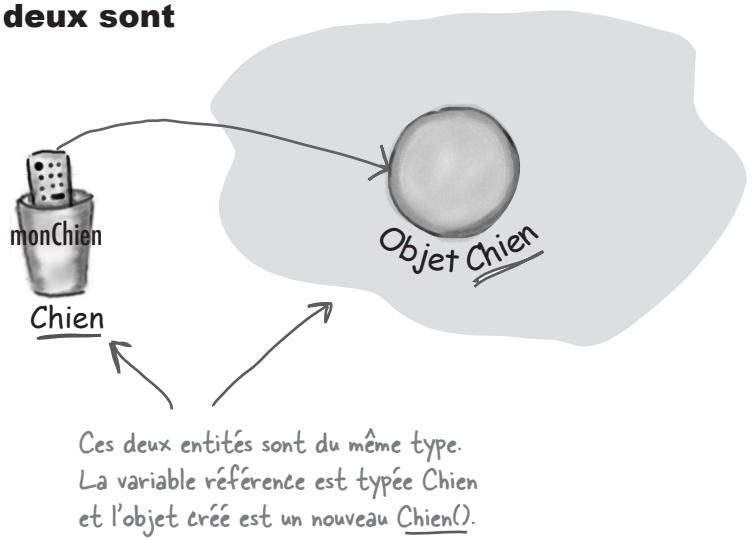
Affecte le nouveau Chien à la variable référence monChien. En d'autres termes, ***programme la télécommande***.



Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

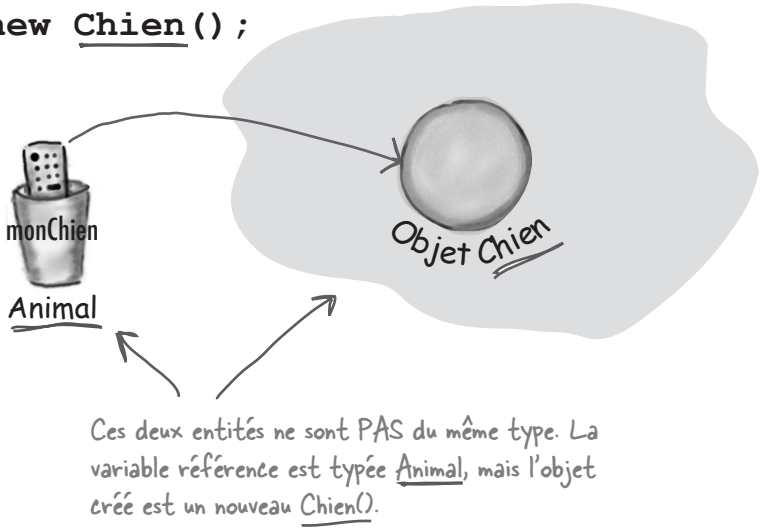
Le point important est que la référence ET l'objet sont du même type.

Dans cet exemple, les deux sont du type Chien.



Mais le polymorphisme autorise la référence et l'objet à être de types *différents*.

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



Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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.

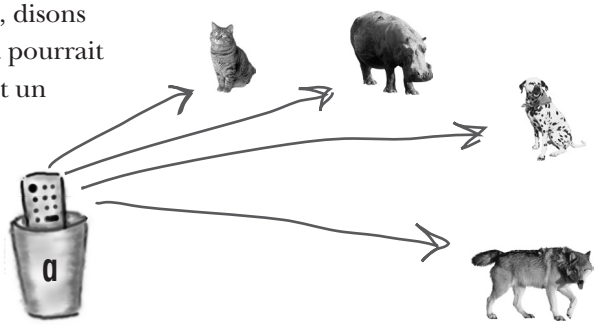
Pareil pour vagabonder().

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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.

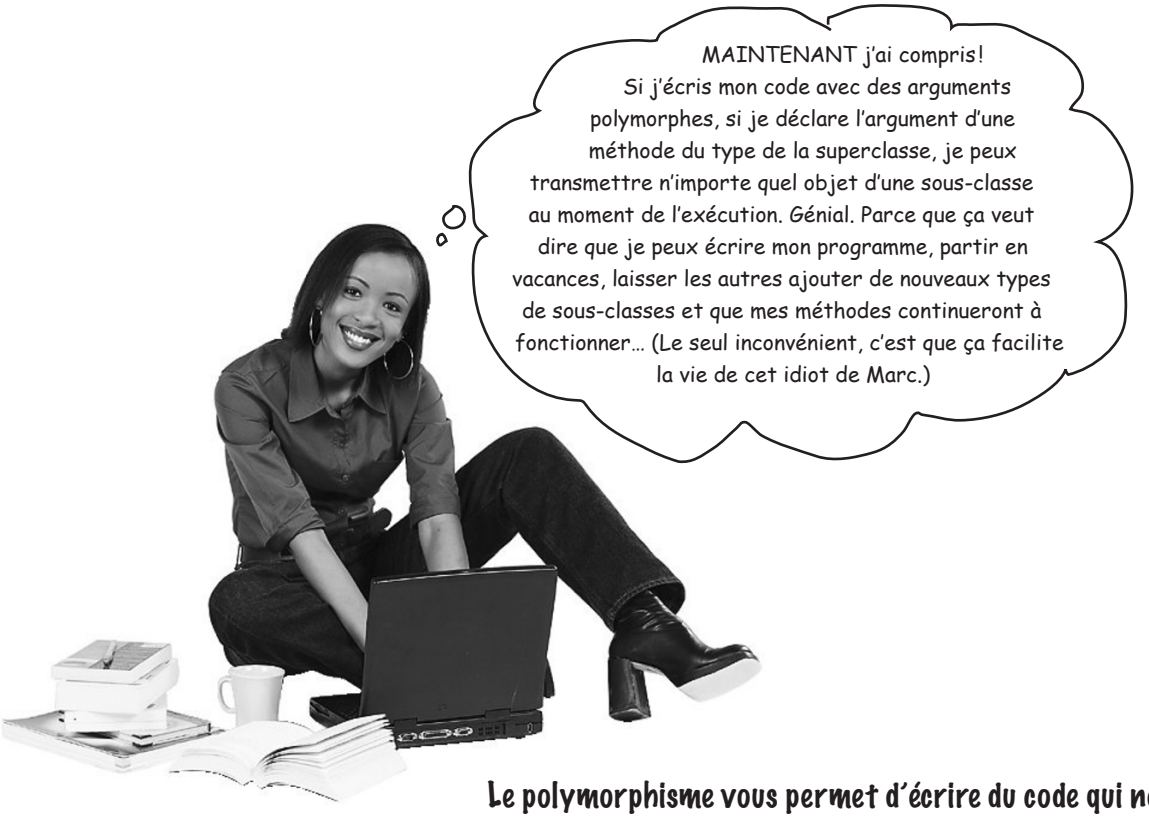
```
class ProprietaireDAnimaux {
   public void start() {
      Veto v = new Veto();
      Chien c = new Chien();
      Hippo h = new Hippo();
      v.soigner(c);
      v.soigner(h);
   }
}
```

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.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>



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) ?

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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.

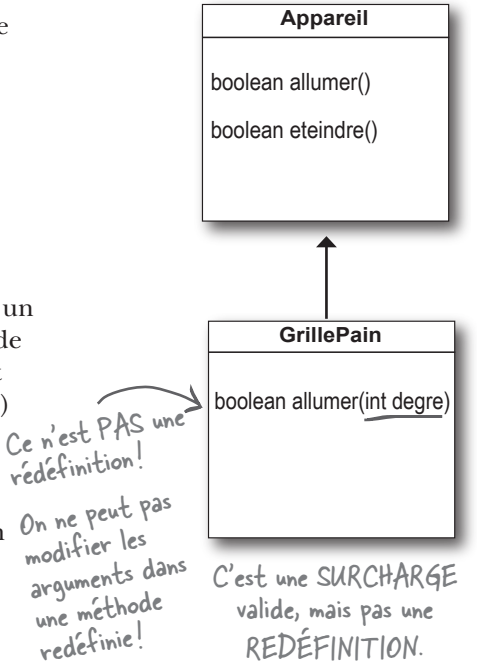
Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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!***



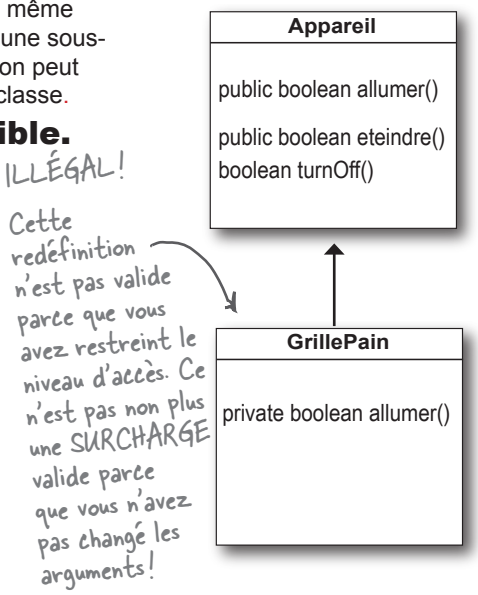
① Arguments et types de retour doivent être identiques.

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.



Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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.

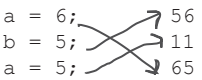
Vous avez le droit de surcharger une méthode en la rendant plus restrictive. Cela n'a pas d'importance, puisque la nouvelle méthode n'est pas obligée de remplir le contrat de la première.

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);
   }
}
```


Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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
```


Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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;
}
```


Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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.

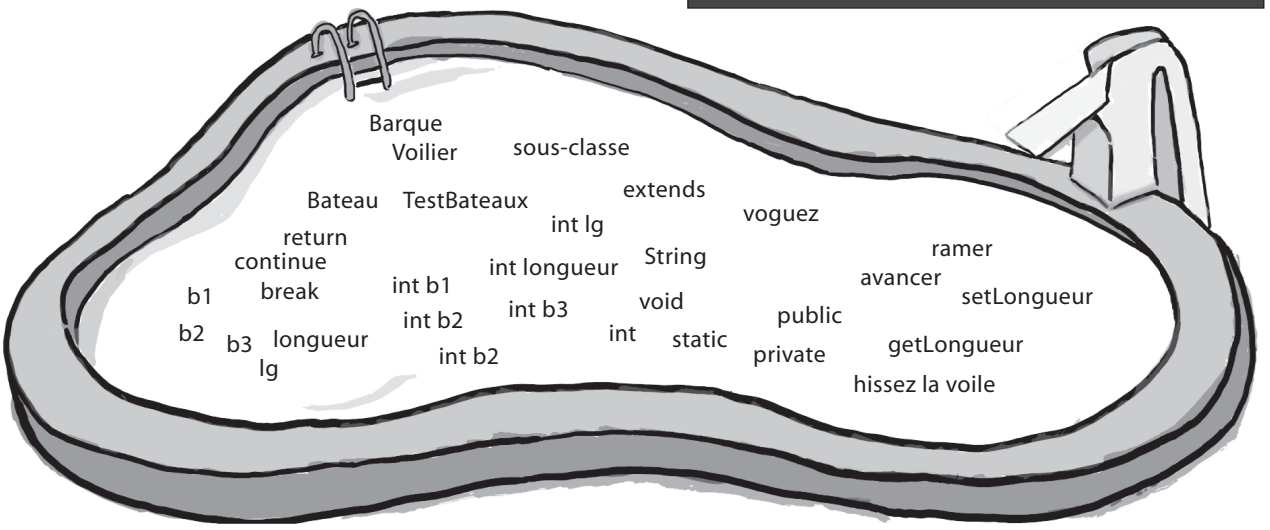
```
public class Barque ________ ________ {
   public ___________ ramer() {
      System.out.print("la mer est belle");
   }
}
```

```
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`



Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Vous ÊTES le compilateur

Solutions des exercices

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
```


Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

```
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`

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

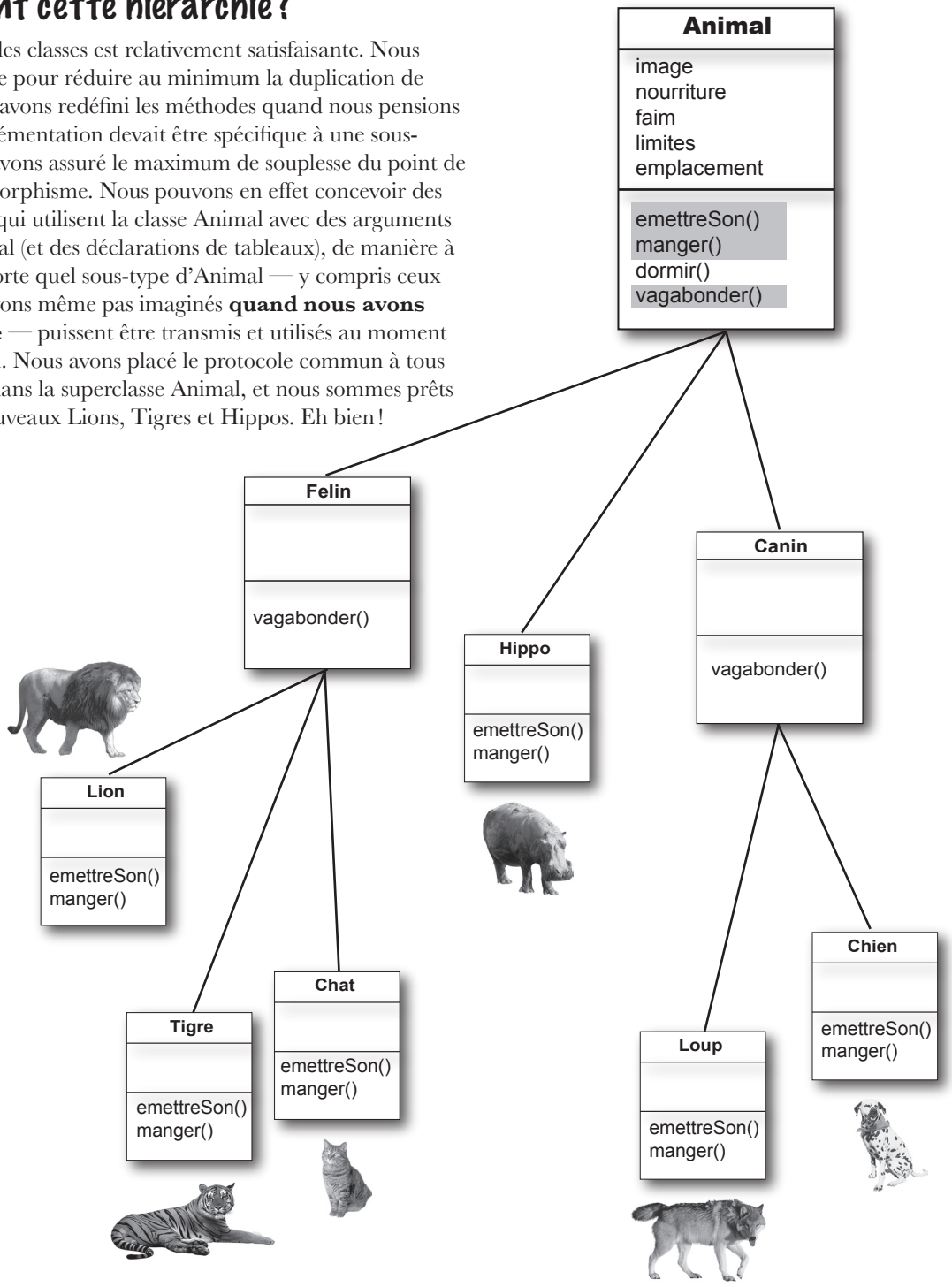
Le polymorphisme dans tous ses états

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.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Avons-nous oublié quelque chose en concevant cette hiérarchie ?

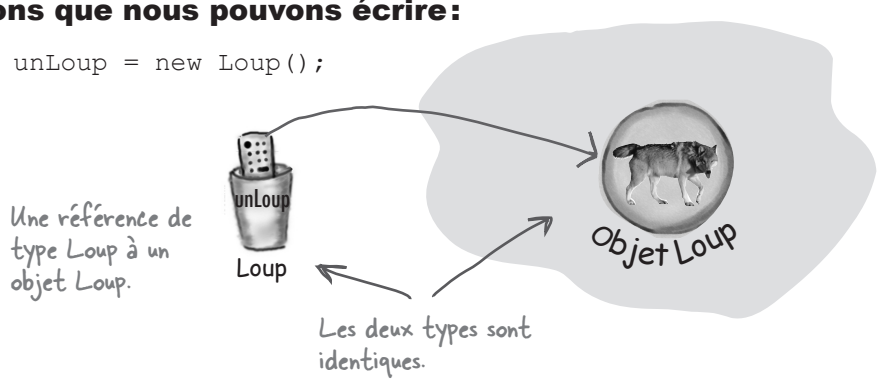
La structure des classes est relativement satisfaisante. Nous l'avons conçue pour réduire au minimum la duplication de code, et nous avons redéfini les méthodes quand nous pensions que leur implémentation devait être spécifique à une sous-classe. Nous avons assuré le maximum de souplesse du point de 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 !



Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

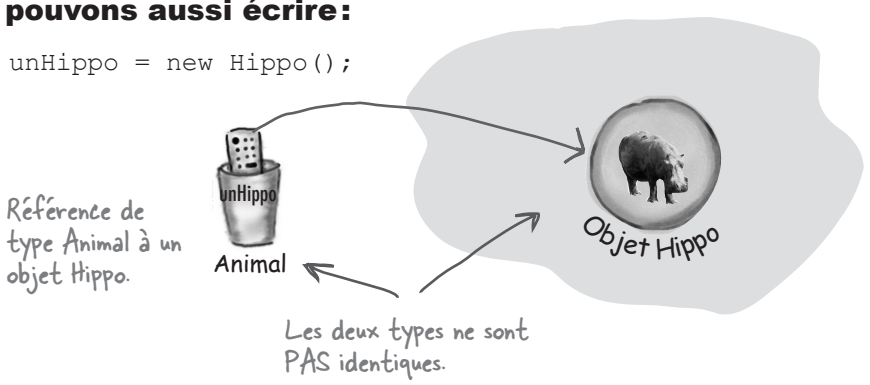
Nous savons que nous pouvons écrire :

```
Loup unLoup = new Loup();
```



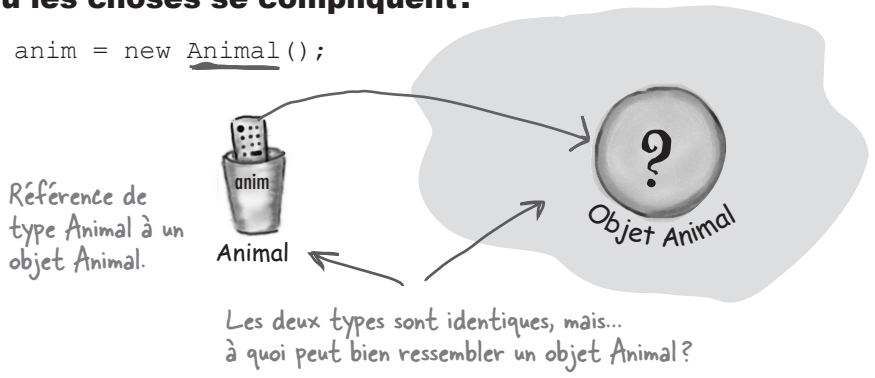
Et que nous pouvons aussi écrire :

```
Animal unHippo = new Hippo();
```



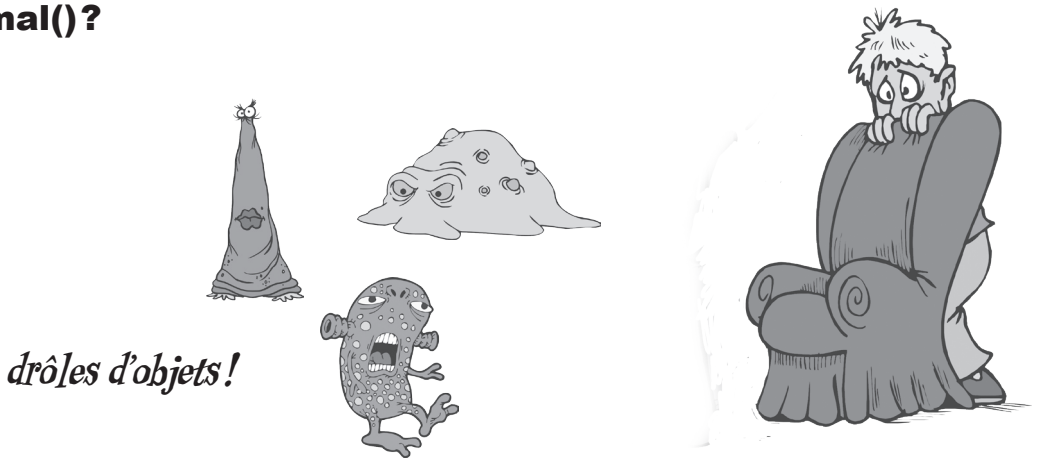
Mais voilà où les choses se compliquent :

```
Animal anim = new Animal();
```



Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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() { }
}
```


Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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.

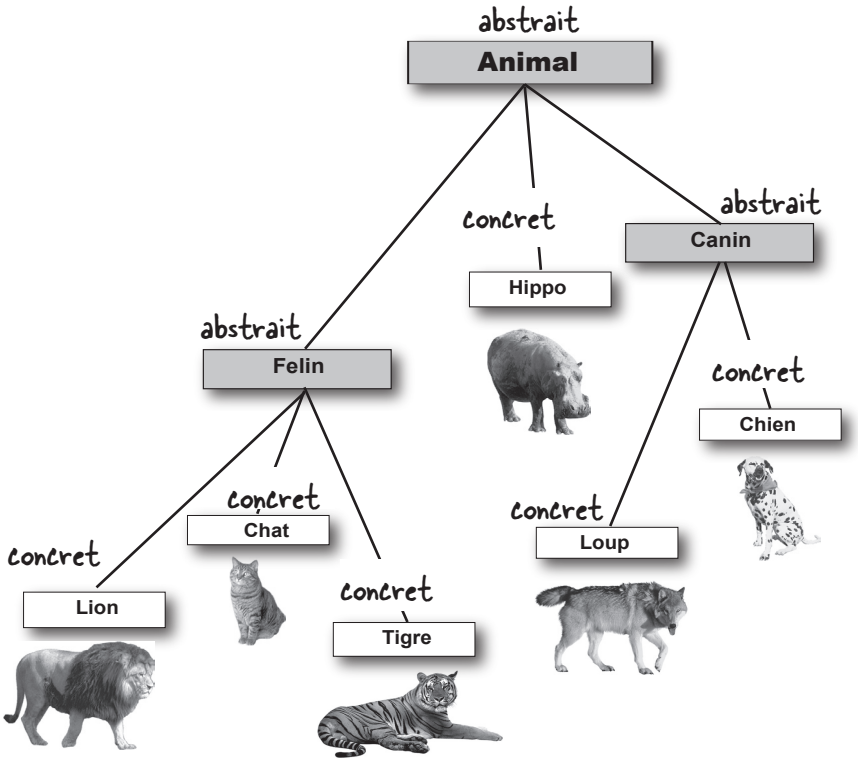
* Il y a toutefois une exception : une classe abstraite peut avoir un membre statique (voir le chapitre 10).

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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 ?

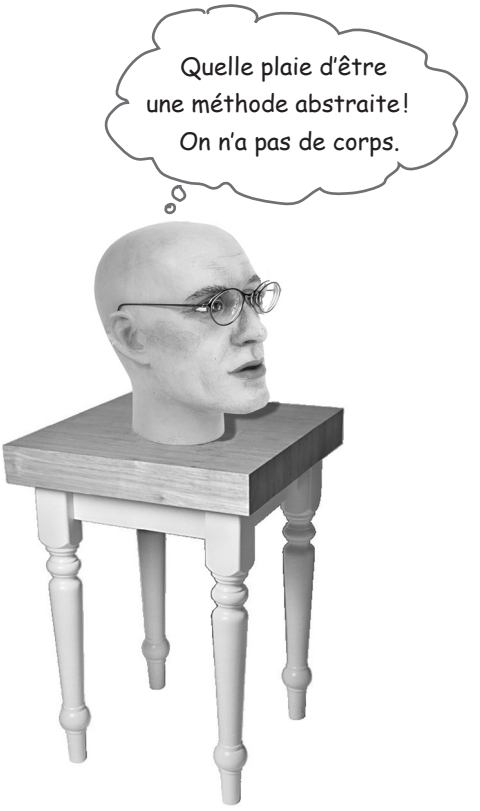
La classe ***PremieresCotesDeBlaye*** est-elle abstraite ou concrète ? Le Premières côtes 90 de la Croix-Simon est sans doute concret, quoi qu'il arrive. Mais comment en être sûr ?

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) ?

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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	_____

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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
```


Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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

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.

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 !

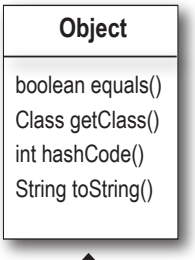
(Note : à partir de Java 5.0, les méthodes get() et add() sont un peu différentes de celles que nous montrons ici, mais c'est pour l'instant une façon de vous les représenter. Nous entrerons dans les détails un peu plus loin.)

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

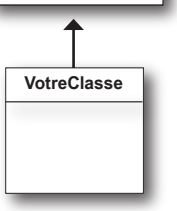
Qu'y a-t-il donc dans cette ultra-super-mégaclasse Object ?

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.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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

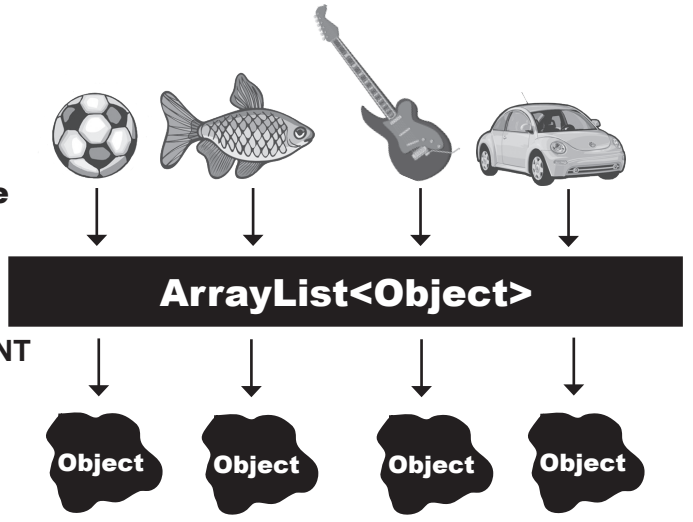
Ajouter le Chien à la liste

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 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.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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

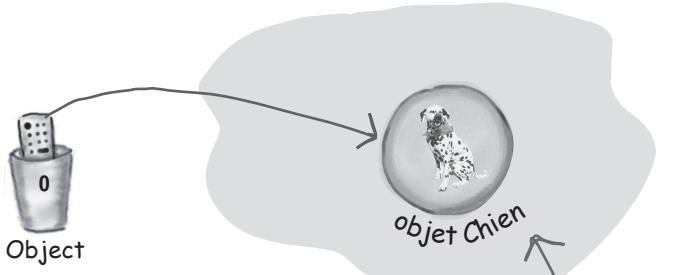
```
public void go() {
    Chien unChien = new Chien();
    Object memeChien = getObject(unChien);
}

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.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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();
```

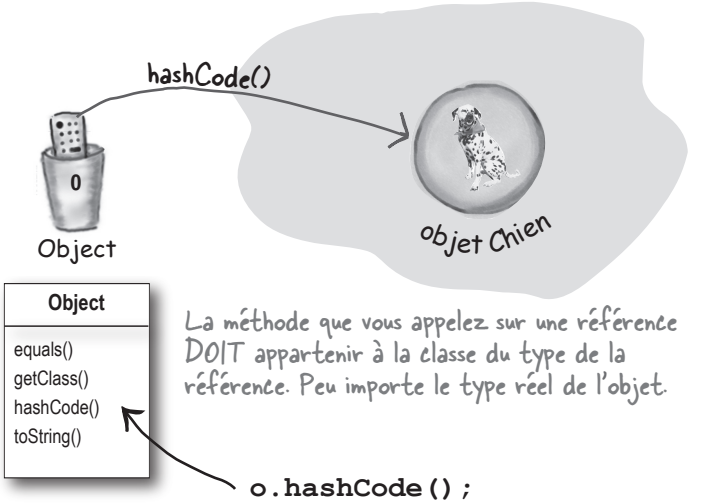
Légal. La classe Object a une méthode hashCode() : vous pouvez appeler cette méthode sur TOUT objet.

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.

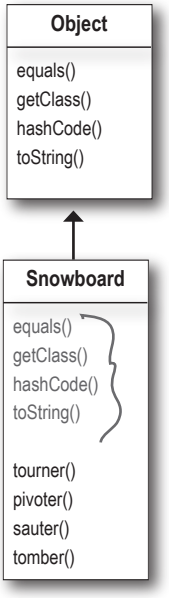
```
o.hashCode();
```

La référence "o" a été déclarée de type Object : vous ne pouvez appeler que des méthodes de la classe Object.

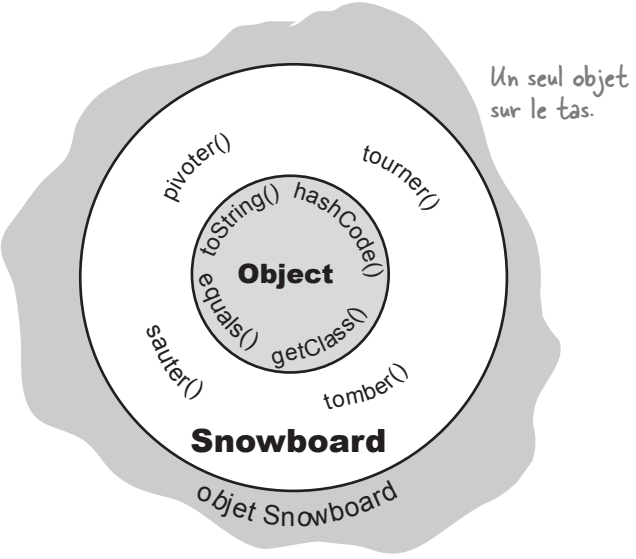
Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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).



Snowboard hérite des méthodes de la superclasse Object et en ajoute quatre autres.



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.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

« 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.

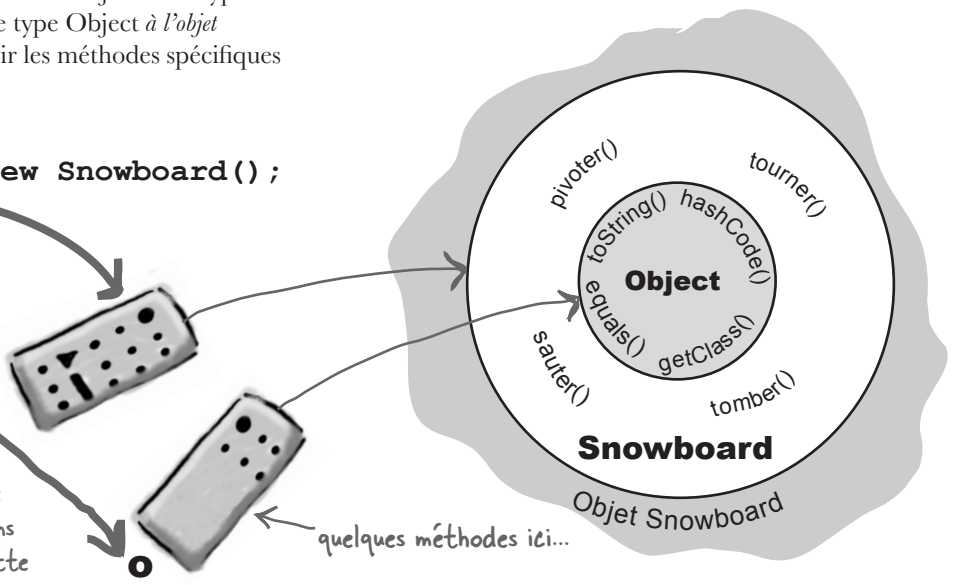
Naturellement, ce n'est pas toujours vrai ; une sous-classe peut ne pas ajouter de nouvelles méthodes, mais se contenter de redéfinir celles de sa superclasse. Mais l'essentiel est que même si l'objet est de type Snowboard, une *référence* de type Object *à l'objet* Snowboard ne peut pas voir les méthodes 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;
```

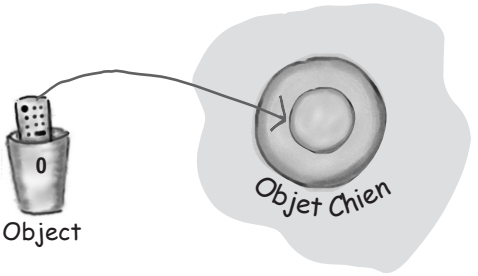


La télécommande (référence) de type Snowboard a plus de boutons qu'une télécommande Object. Cette télécommande peut voir toute la Snowboarditude de l'objet Snowboard. Elle peut accéder à toutes ses méthodes, qu'elles soient héritées de la classe Object ou spécifiques à la classe Snowboard.

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.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

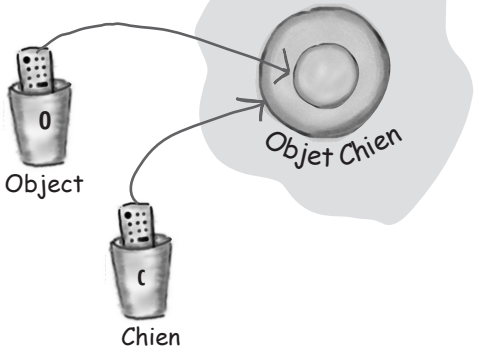
Reconvertir une référence à un objet dans son type *réel*.



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;
}
```


Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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.)

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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.

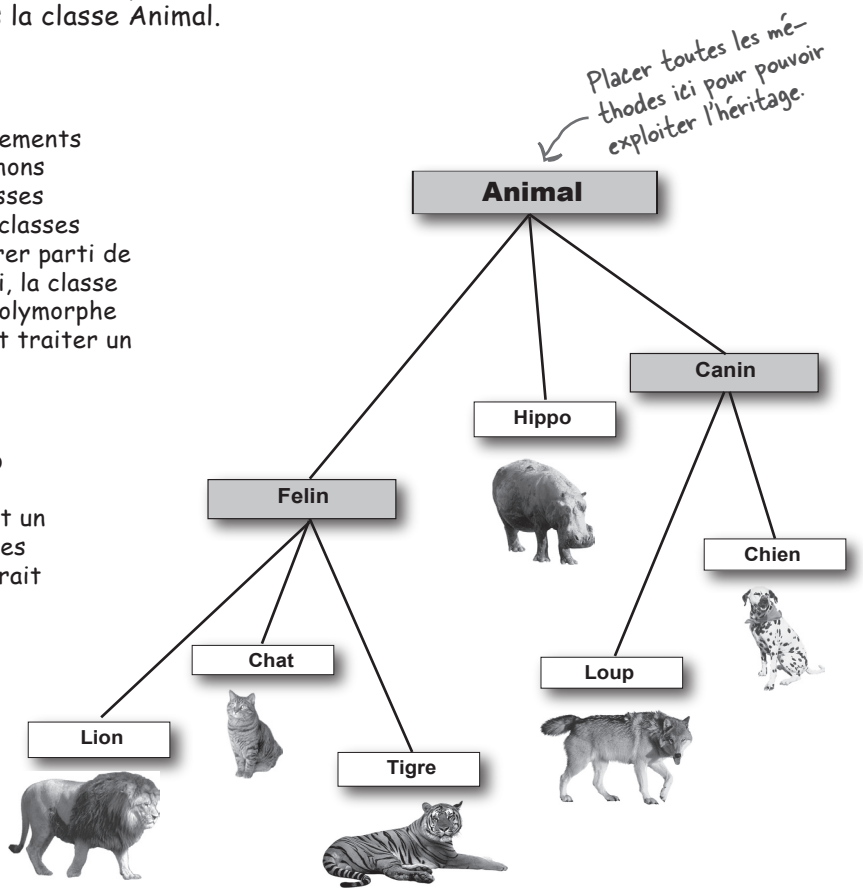
Pour :

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.



Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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.

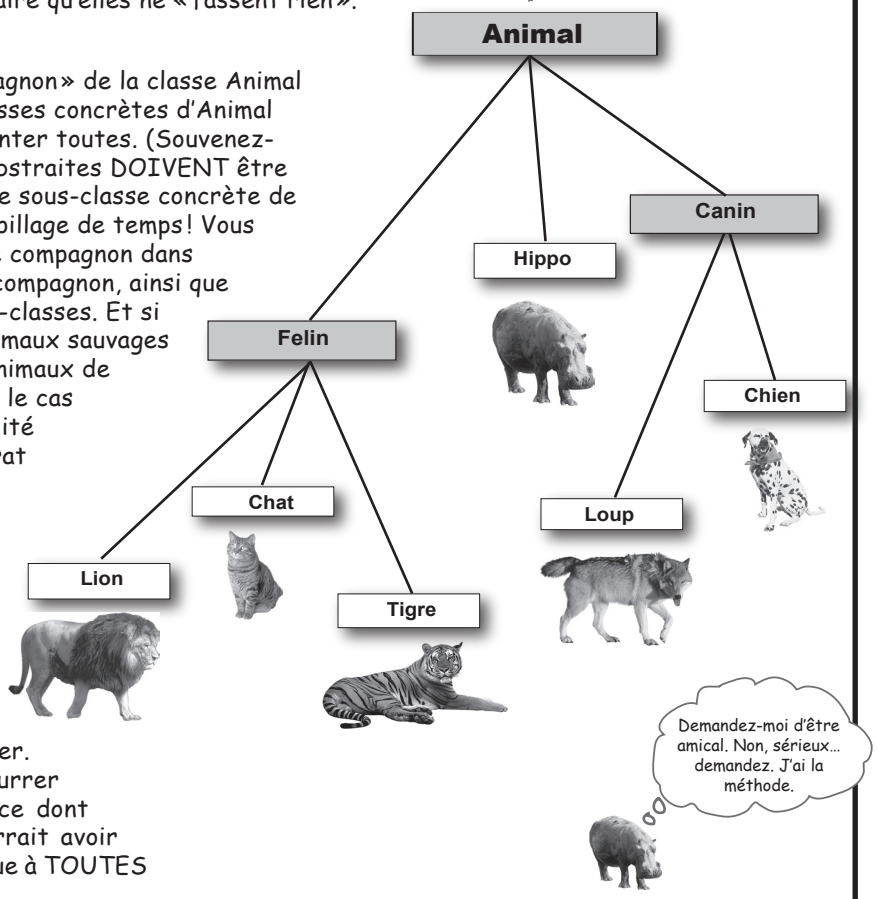
Pour :

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.

Placer toutes les méthodes de compagnon ici, mais sans implémentations. Les rendre toutes abstraites.



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 .

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

③ Option trois

Ne placer les méthodes de compagnon QUE dans les classes auxquelles elles appartiennent.

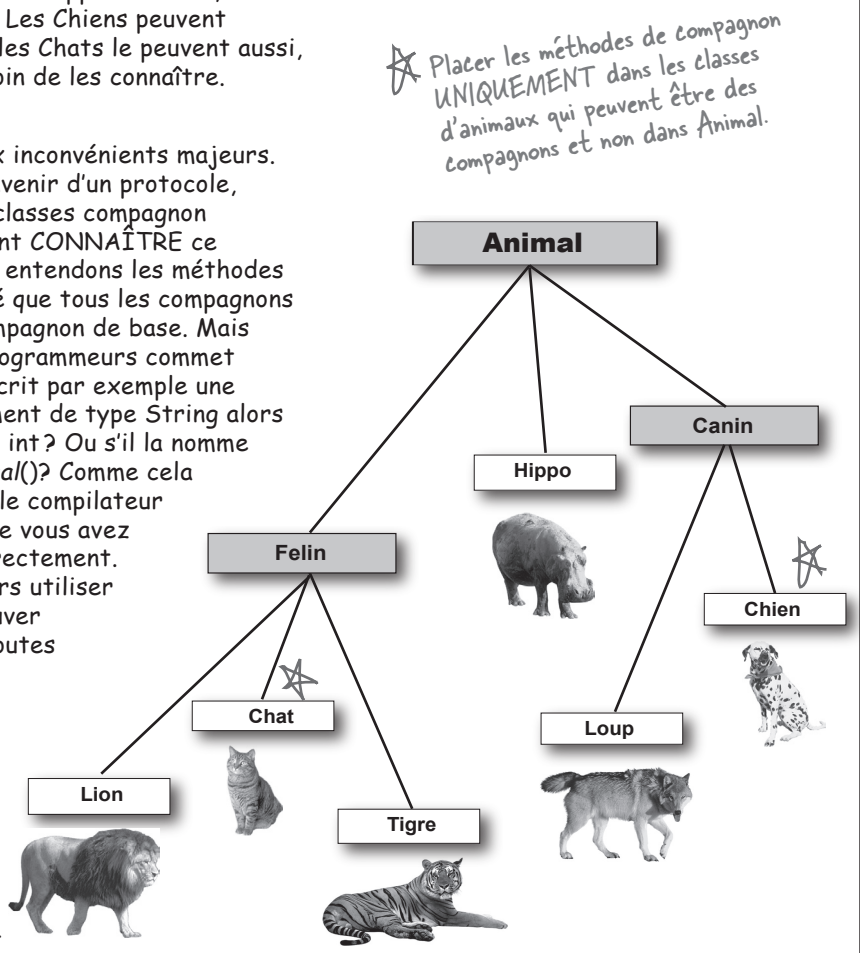
Pour:

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.

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.

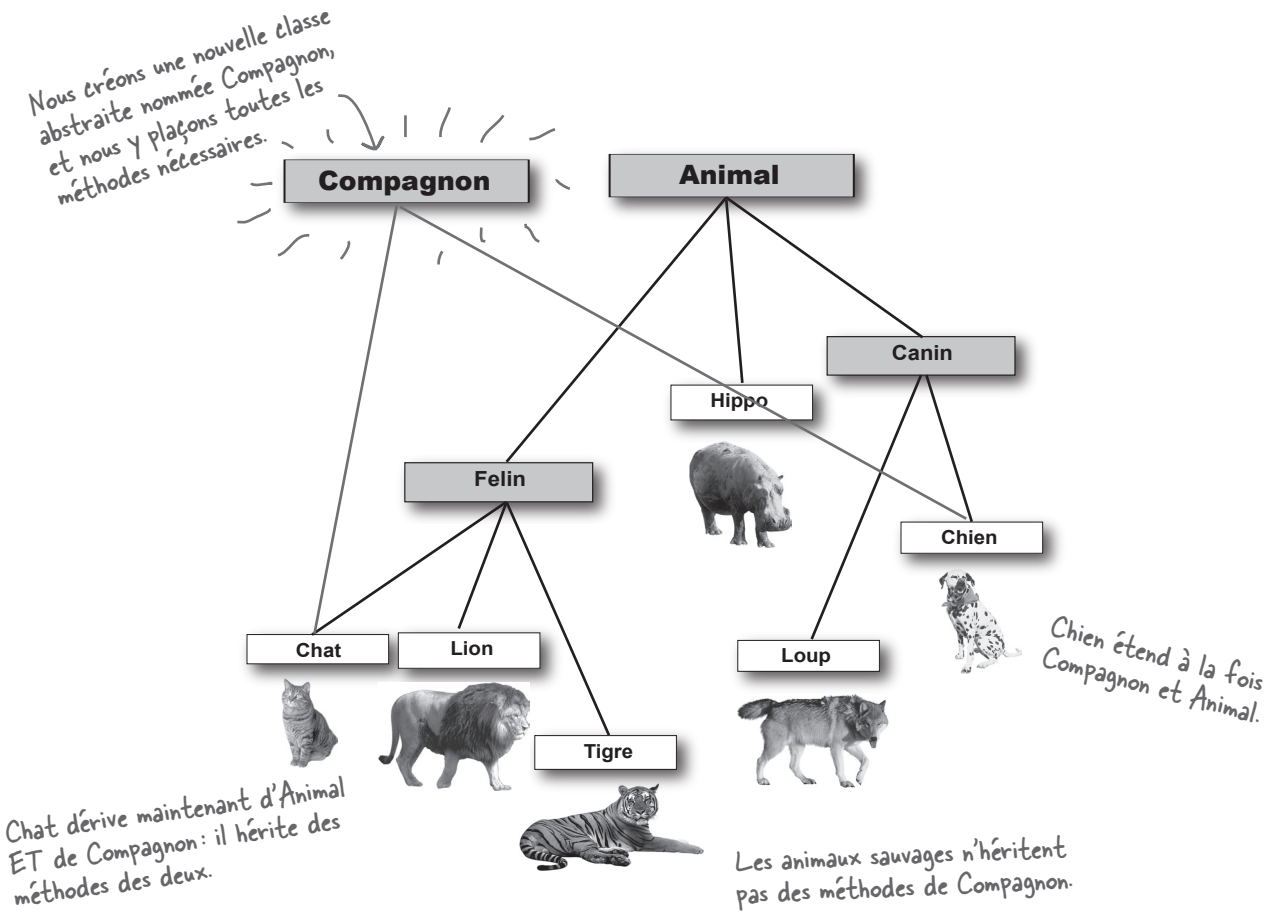


Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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



Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

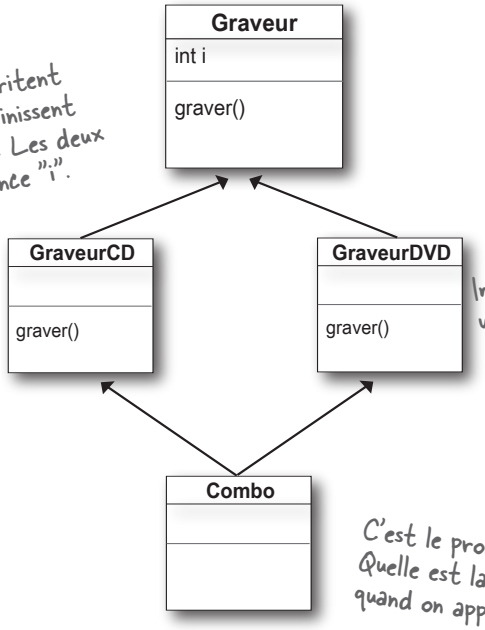
Mais l'approche à « deux superclasses » pose un petit problème…

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



GraveurCD et GraveurDVD héritent tous deux de Graveur et redéfinissent tous deux la méthode graver(). Les deux héritent de la variable d'instance "i".

Imaginez que GraveurCD et GraveurDVD utilisent tous deux la variable d'instance "i" avec des valeurs différentes. Que se passe-t-il si Combo a besoin des deux valeurs de "i" ?

C'est le problème de l'héritage multiple. Quelle est la méthode graver() qui s'exécute quand on appelle graver() sur le Combo ?

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 ?*

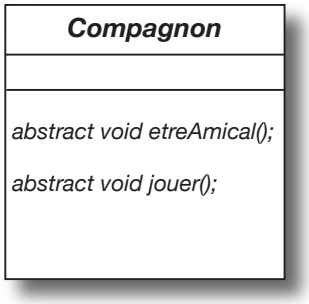
Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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.

il n'y a pas de

Questions stupides

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 ?

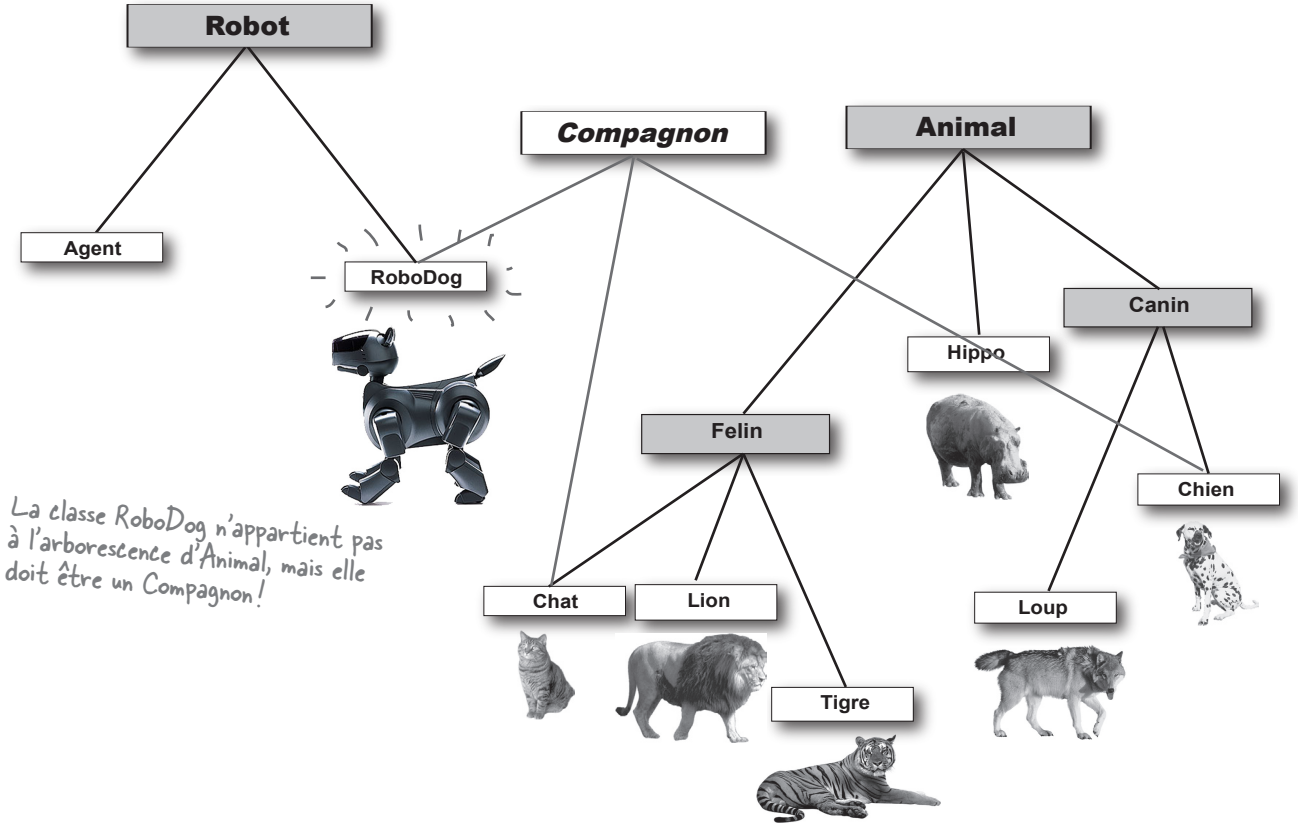
R : Polymorphisme, polymorphisme et polymorphisme. Les interfaces sont le *nec plus ultra* de la souplesse. Si vous substituez des interfaces aux classes concrètes (ou même aux superclasses abstraites) pour les arguments et les types de retour, vous pouvez transmettre n'importe quoi qui implémente cette 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.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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.

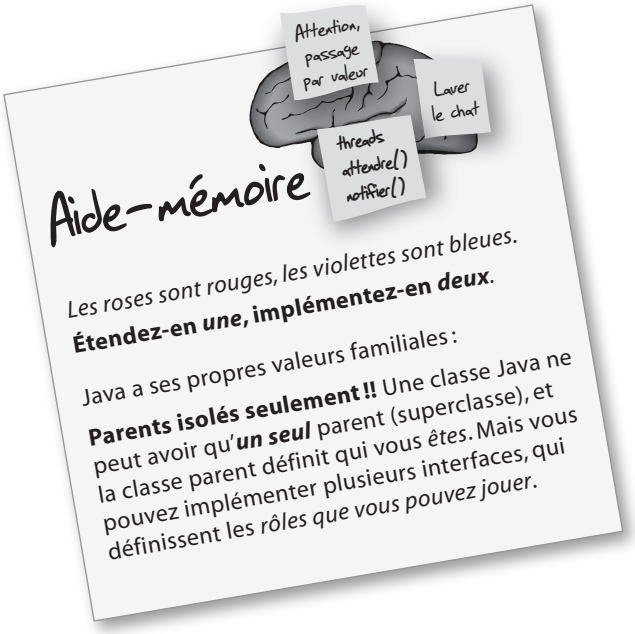
Mais quand vous utilisez une interface comme type polymorphe (comme pour un tableau de Compagnons), les objets peuvent venir de n'importe quel point de la hiérarchie d'héritage. La seule exigence est qu'ils appartiennent à une classe qui implémente l'interface. Autoriser des classes provenant de structures 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 { ... }
```


Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>



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.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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() {...}
}
```

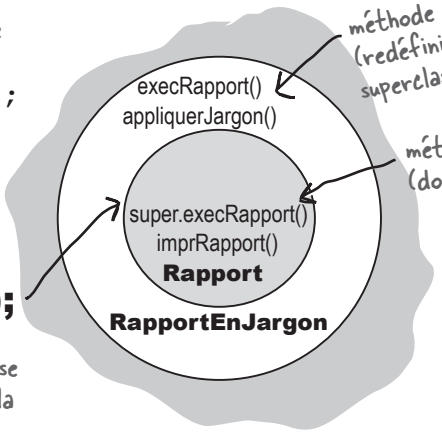
la version de la superclasse fait un travail important qui peut être utile aux sous-classes.

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.



méthode de la sous-classe (redéfinit la version de la superclasse).

méthodes de la superclasse (dont execRapport()).

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.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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 { }
```

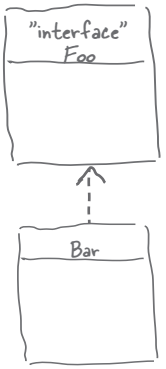
3)
```
public abstract class Zig implements Puce { }
public class Alfred extends Zig { }
public interface Puce { }
```

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)

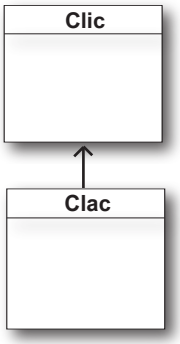
4)

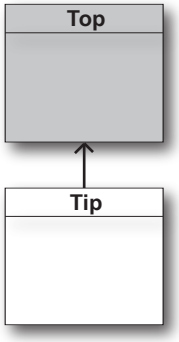
5)

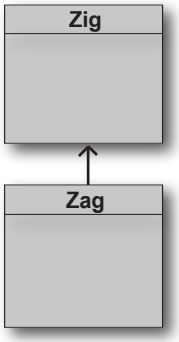
Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

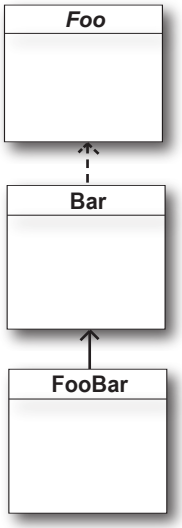
Vous trouverez à gauche des diagrammes de classes. Votre tâche consiste à les traduire en déclarations Java valides. Nous avons fait le numéro 1 pour vous (et c'était difficile).

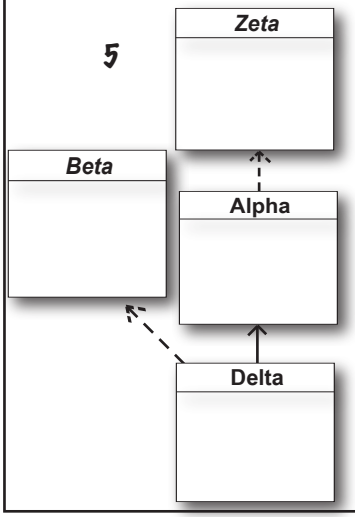
Diagrammes

1


2


3


4


5


Quelle est la déclaration ?

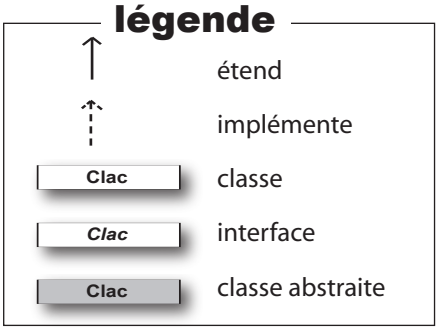
1) `public class Clic { }`
`public class Clac extends Clic { }`

2)

3)

4)

5)



Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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 ( ) { }

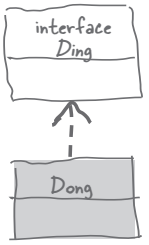
i.jouer(x)
i(x).jouer[ ]
i[x].jouer( )
i[x].jouer[ ]

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

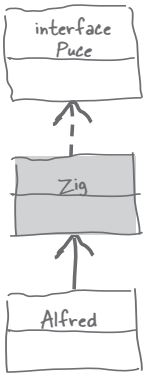
Solutions des exercices

Quel est le diagramme ?

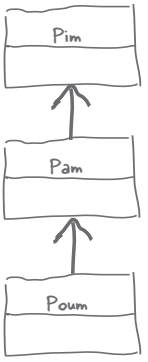
2)



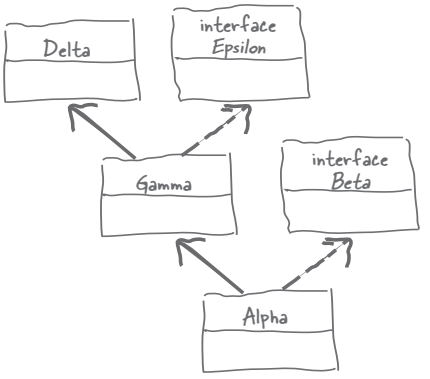
3)



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 { }
```


Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

```
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
```


Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

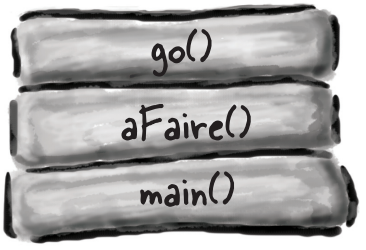
Deux lieux de résidence: la pile et le tas

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.

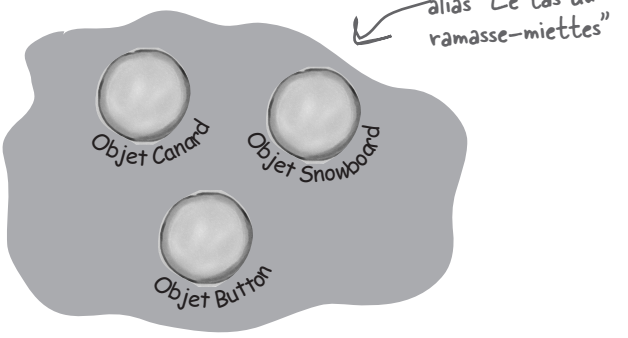
La Pile

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.

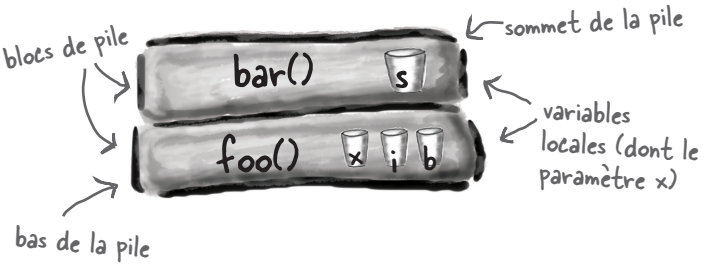
Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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.

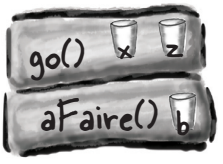
```
public void aFaire() {
   boolean b = true;
   go(4);
}

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).

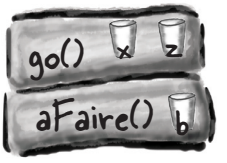
1. Le code d'une autre classe appelle **aFaire()**, et **aFaire()** va dans un bloc au sommet de la pile. La variable booléenne nommée '**b**' va dans le bloc de pile de **aFaire()**.

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()**.



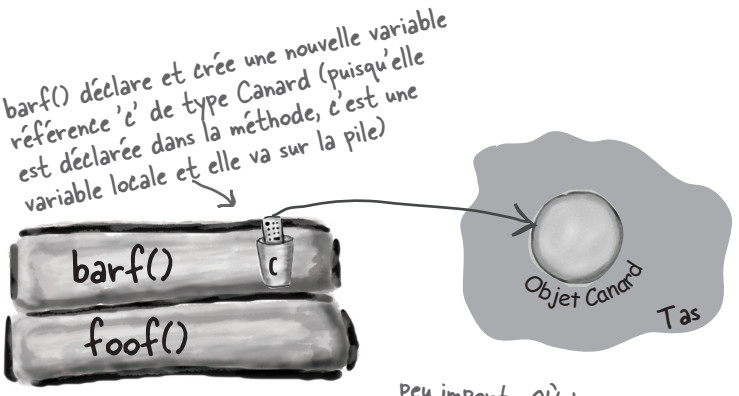
Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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);
   }
}
```



Peu importe OÙ la variable référence est déclarée (dans une méthode ou comme une variable d'instance d'une classe), l'objet va toujours toujours toujours sur le tas

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.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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.

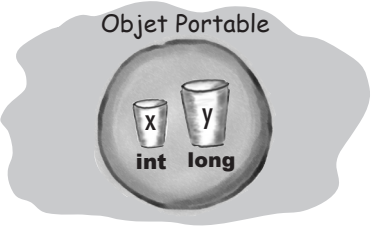
Quand le nouvel objet a des variables d'instance qui sont des références à des objets et non des types primitifs, la vraie question qui se pose est celle de savoir si cet objet a besoin d'espace pour tous les objets pour lesquels il contient des références. La réponse est : *pas exactement.* Quoi qu'il 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.

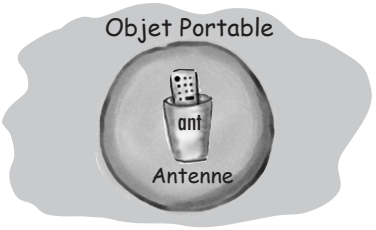
```
private Antenne ant;
```

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();
```

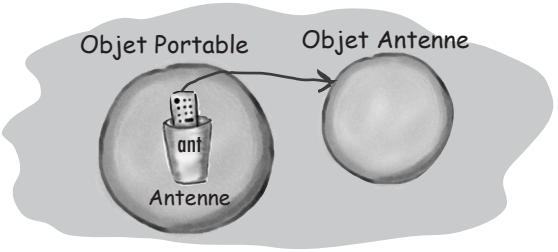


Objet avec deux variables d'instance de type primitif. L'espace pour les variables se trouve dans l'objet.



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();
}
```


Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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.*

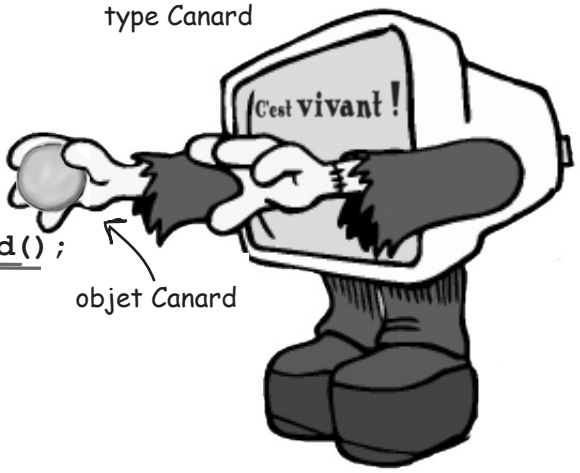
Revoyons les 3 étapes de la déclaration, de la création et de l'affectation d'un objet :

1 Déclarer une variable référence

créer une nouvelle variable référence du type d'une classe ou d'une interface

```
Canard monCanard = new Canard();
```

référence de type Canard



2 Créer un objet

Miracle

```
Canard monCanard = new Canard();
```

objet Canard

3 Lier l'objet et la référence

affecter le nouvel objet à la référence

```
Canard monCanard = new Canard();
```

objet Canard

référence de type Canard

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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.

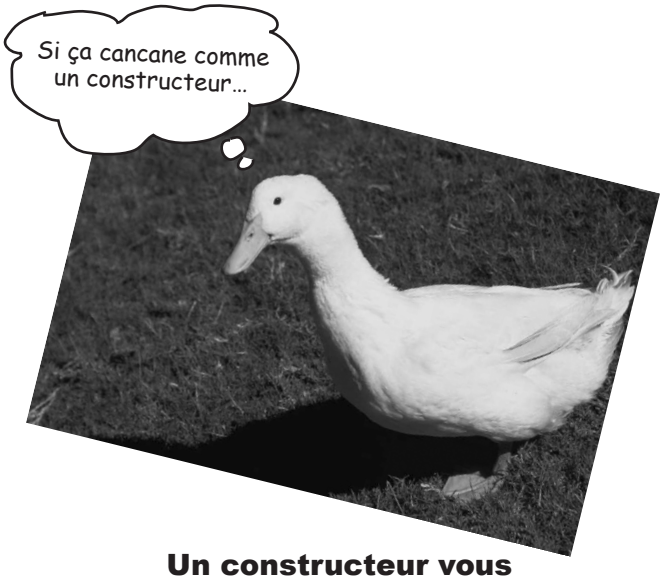
```
public  Canard() {
   // ici, le code du constructeur
}
```

Où est le type de retour ? Si c'était une méthode, il y aurait un type de retour entre "public" et "Canard()".

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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.




```
public class Canard {

   public Canard() {
      System.out.println("Coin");
   }
}
```

code du constructeur

Un constructeur vous permet d'intervenir au milieu de `new`.

```
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).

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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);
   }
}
```

Transmettre une valeur au constructeur.

Cette fois, il suffit d'une seule instruction pour créer le nouveau Canard et définir sa taille.

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…

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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.***

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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
```


Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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… »

MUSCLEZ VOS NEURONES

Qu'en est-il des superclasses ?

Quand on crée un Chien, le constructeur de Canin doit-il s'exécuter aussi ?

Si la superclasse est abstraite, doit-elle même *avoir* un constructeur ?

Nous allons voir ceci dans les pages qui suivent. Arrêtez-vous de lire et réfléchissez aux implications des constructeurs et des superclasses.

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 !

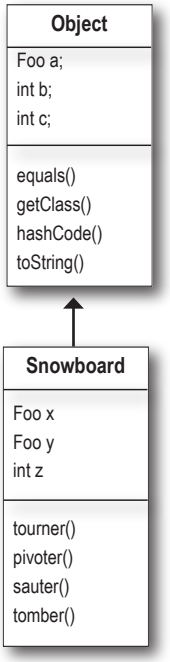
R : Mais ce n'est pas tout à fait vrai. private ne veut pas dire que *rien* ne peut y accéder, mais que *rien en dehors de la classe* ne peut y accéder. Situation inextricable, pensez-vous ! Seul le code de la *même* classe, celle qui contient le constructeur privé, peut créer un objet à partir de cette 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.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Attendez une minute... nous n'avons JAMAIS parlé des rapports entre les superclasses, l'héritage et les constructeurs.

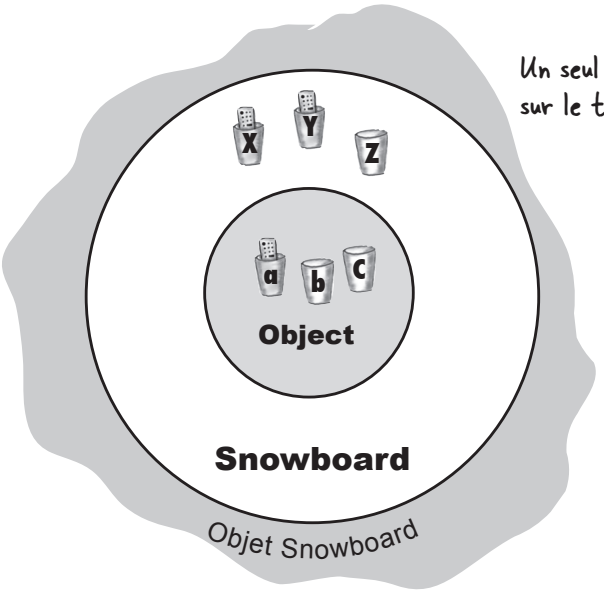
C'est là que cela devient amusant. Souvenez-vous : nous avons vu au chapitre précédent que l'objet Snowboard enveloppait un noyau intérieur représentant la portion Object de la classe Snowboard. Le point essentiel ici, c'est que tout objet ne contient pas seulement ses propres variables d'instance, mais 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.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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.

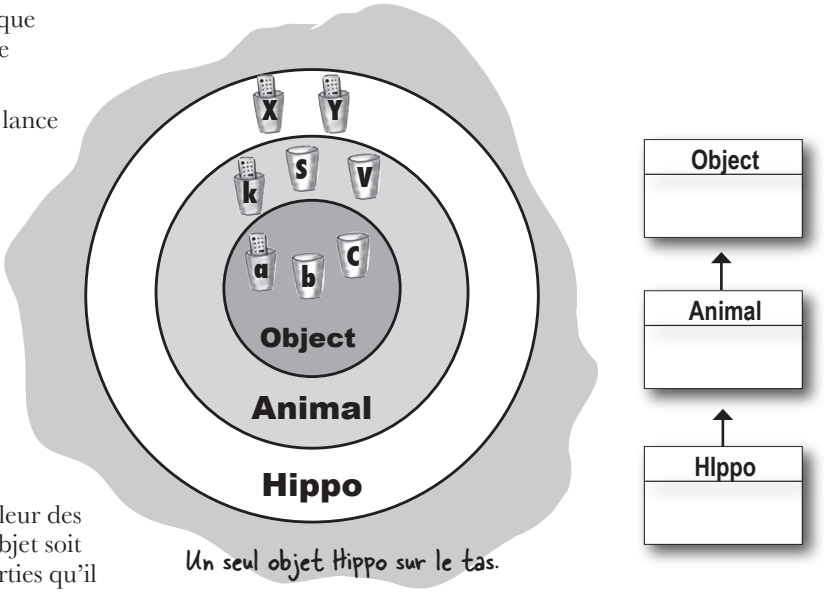
Cela signifie que chaque superclasse a un constructeur (parce que toute classe a un constructeur), et que chaque constructeur de la hiérarchie s'exécute lorsqu'un nouvel objet est créé.

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.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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).

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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;
  }

⊘ public Bip(int i) {
      taille = i;
      super();
  }
```

Légal, puisque le compilateur insérera l'appel de super() en première instruction.

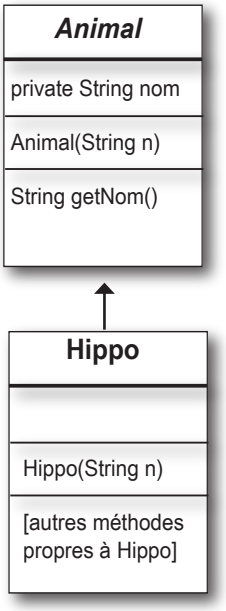
ILLEGAL !! Ne compilera pas ! Vous ne pouvez pas placer explicitement l'appel de super() après quoi que ce soit...

*Cette règle connaît une exception ; nous l'aborderons page 252.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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
```


Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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();
            ^
```

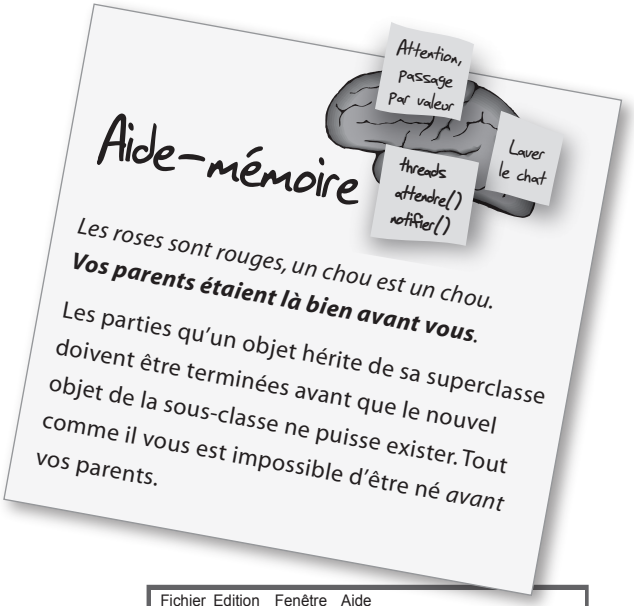

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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()
```


Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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.

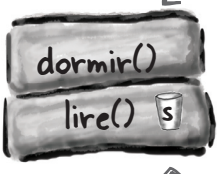
Mais si la vie d'un objet dépend de celle de sa référence, combien de temps celle-ci vit-elle ?

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*.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Différence entre **durée de vie** et **portée** des variables locales :

Durée de vie

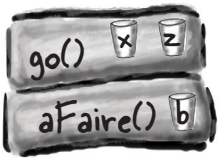
Une variable locale est en vie tant que son bloc est sur la pile. Autrement dit, *tant que la méthode n'est pas terminée*.

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***.

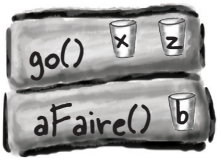
```
public void aFaire() {
   boolean b = true;
   go(4);
}
public void go(int x) {
   int z = x + 24;
   couac();
   // le reste du code
}
public void couac() {
   char c = 'a';
}
```

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).

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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".

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

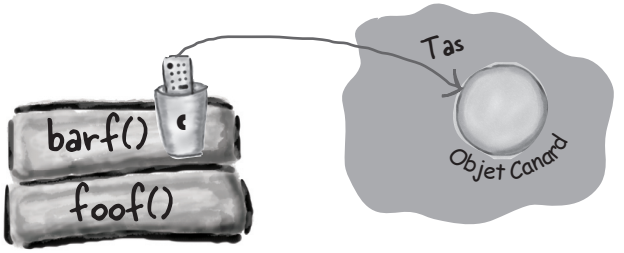
Tueur d'objets n°1

La référence sort définitivement de la portée.

```
public class RefPile  {
   public void foof() {
      barf();
   }

   public void barf() {
      Canard c = new Canard();
   }
}
```

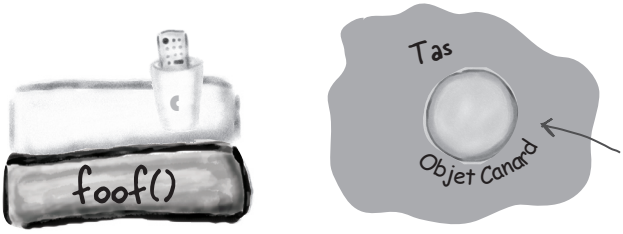
1. *foof()* est poussée sur la pile. Aucune variable n'est déclarée.

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".



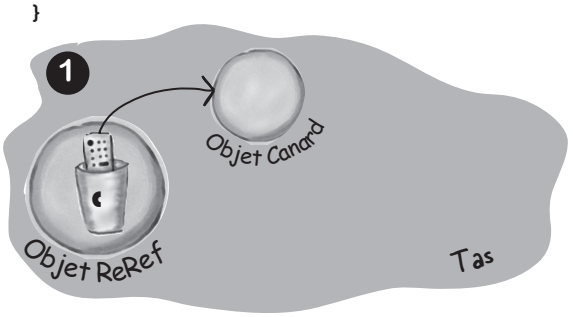
Oh-oh. La variable "c" a disparu quand le bloc de barf() a été dépilé. Le Canard est abandonné. Une proie rêvée pour le ramasse-miettes.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Tueur d'objets n°2

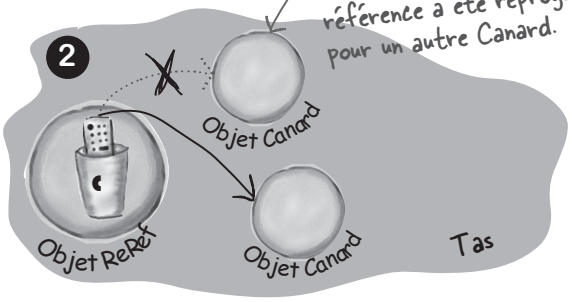
Affecter la référence à un autre objet

```
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...

Quand on appelle la méthode go(), ce Canard est abandonné. Sa seule référence a été reprogrammée pour un autre Canard.



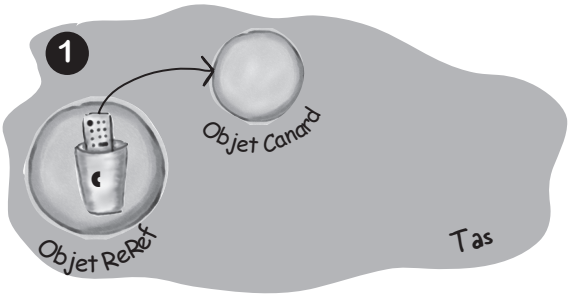
"c" se voit affecter un nouvel objet Canard, et le Canard précédent est abandonné. C'est comme si ce premier Canard était mort.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Tueur d'objets n°3

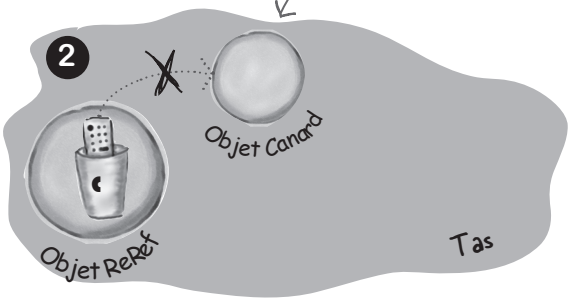
Positionner explicitement la référence à null

```
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`

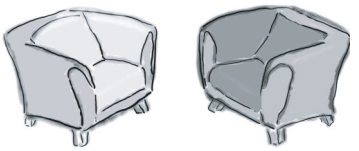
Quand vous positionnez une référence à null, vous déprogrammez la télécommande. Autrement dit, vous avez bien une télécommande, mais pas de télé à l'autre bout. Cette référence a des bits qui représentent « null » (*nous* ne savons pas ce que sont ces bits, et nous ne nous en soucions pas, du moment 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.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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

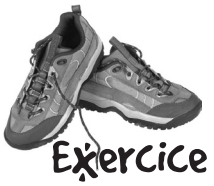
D'un côté, nous sommes de nouveau actives. De l'autre, les heures nous sont comptées. Notre vie est si courte. Plus le temps passe, et plus nous nous rapprochons de la fin de la méthode. Et nous savons *tous* ce qui arrive alors.

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.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>



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;`

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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"

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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"));
    }
  }
}
```


Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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. »

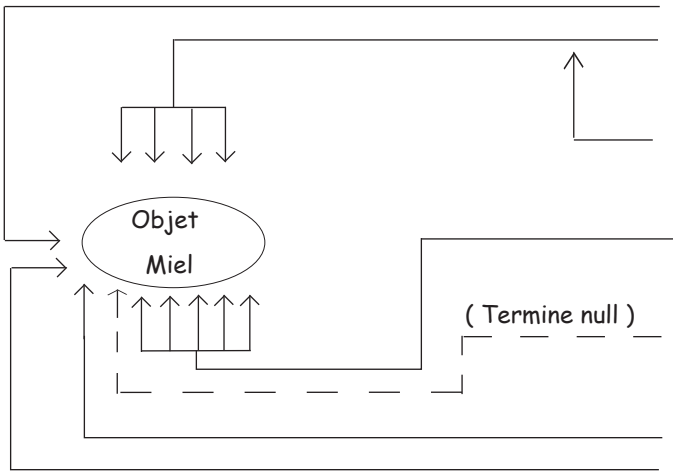
Que suspectait Léo ? Comment a-t-il compris l'erreur et quelles lignes de code pourriez-vous ajouter à ce programme pour pouvoir le déboguer plus facilement ?

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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()
}
```


Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Solution du jeu

La clé du mystère

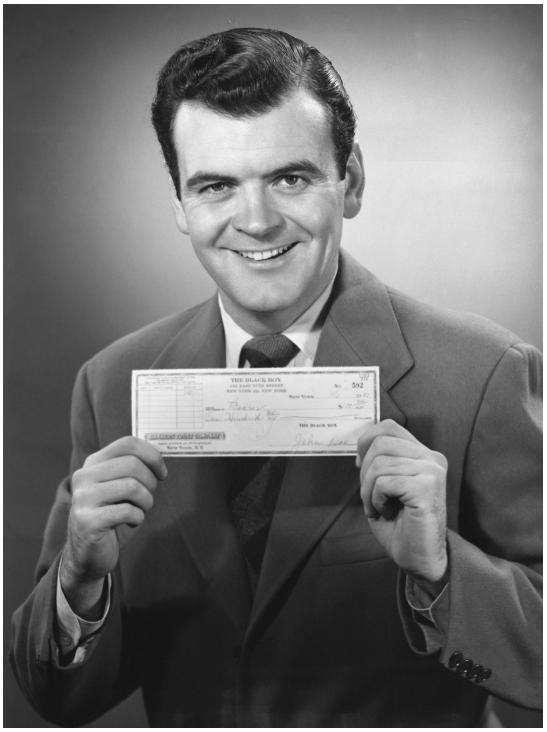
Léo avait remarqué que le constructeur de la classe RadiateurV2 acceptait une ArrayList. Chaque fois que le constructeur de RadiateurV3 était appelé, il transmettait une ArrayList dans son appel de super() au constructeur de RadiateurV2. Cela signifiait que cinq SimUnits RadiateurV2 supplémentaires étaient 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 !

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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*.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Méthodes ordinaires (non statiques) vs. méthodes statiques

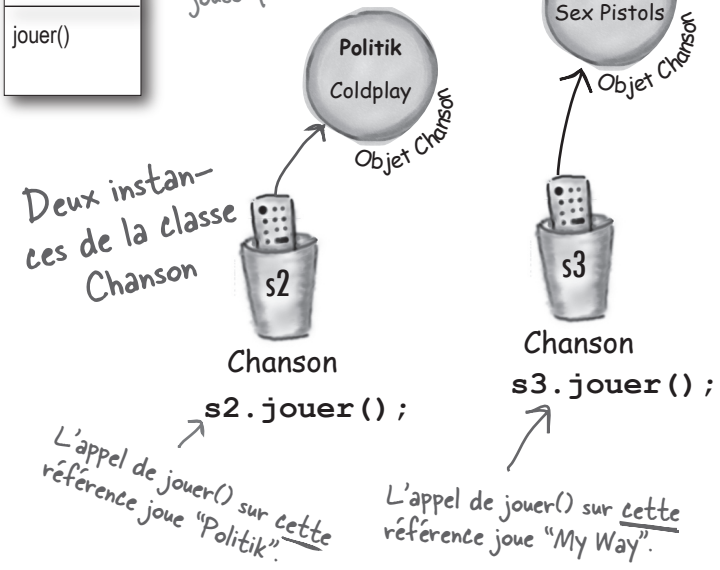
Java est orienté objet, mais on rencontre parfois un cas spécial, généralement une méthode utilitaire (comme les méthodes de la classe Math), où une instance de la classe est inutile. Le mot-clé **`static`** permet à une méthode de s'exécuter en l'absence de toute ***instance de la classe***. Une méthode 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);
   }
```

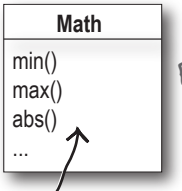
La valeur de la variable d'instance affecte le comportement de la méthode jouer().

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
}
```



Pas de variables d'instance. Le comportement de la méthode ne change pas en fonction de l'état d'une variable.

```
Math.min(42,36);
```

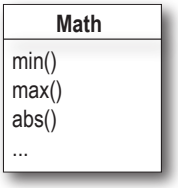
On utilise le nom de la classe, non celui d'une variable référence.

PAS D'OBJETS!!

Absolument AUCUN OBJET nulle part!

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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 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);
                                  ^
```

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.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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 !

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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*.

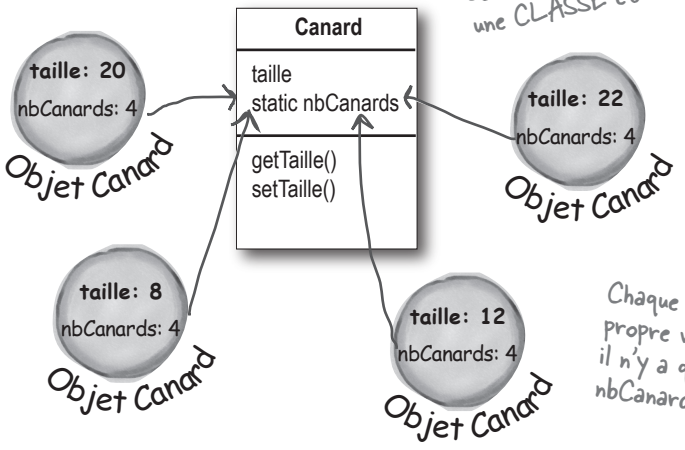
La variable statique nbCanards n'est initialisée QUE quand la classe est chargée, et NON chaque fois qu'une nouvelle instance est créée.

```
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.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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**

MUSCLEZ VOS NEURONES

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 ?

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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.

Fichier Edition Fenêtre Aide Combien?

```
% java TestJoueur
0
1
```

0 ← avant toute création d'instance

1 ← après création d'un objet

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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
```


Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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;
    final int whuffie;

    Foof() {
       whuffie = 42;
    }

    void faireQuelqueChose(final int x) {
       // vous ne pouvez pas modifier x
    }

    void faireAutreChose() {
       final int z = 7;
       // vous ne pouvez pas modifier z
    }
}
```

vous ne pouvez plus modifier taille

vous ne pouvez plus modifier whuffie

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).

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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);
   }
}
```


Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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
```

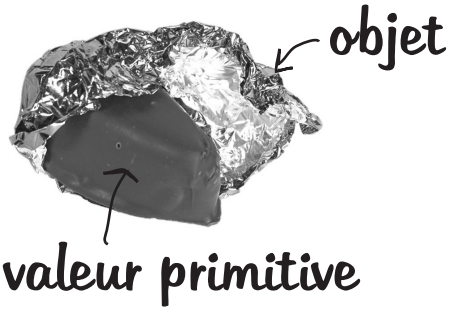

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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.)



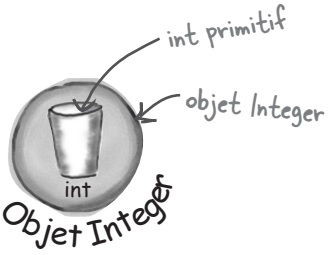
Quand vous devez traiter une valeur primitive comme un objet, enveloppez-la. Si vous utilisez une version antérieure à Java 5.0 vous le ferez, par exemple, quand vous voudrez stocker une telle valeur dans une collection – une ArrayList ou une HashMap.

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();
```


Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

C'est stupide. Vous voulez dire que je ne peux tout simplement pas créer une ArrayList d'entiers??? Que je dois emballer chacun d'eux dans un nouvel objet Integer puis le déballer quand j'essaie d'accéder à cette valeur dans l'ArrayList? C'est une perte de temps et une erreur qui n'attend que l'occasion de se produire

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.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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 !

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>.)

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.

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>.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

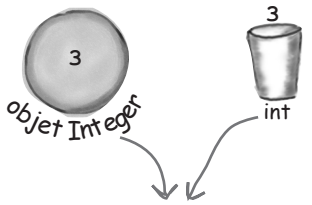
L'autoboxing fonctionne presque partout

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.

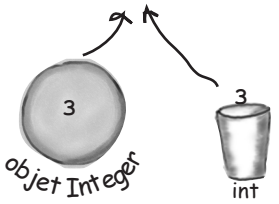



```
void prendreNombre(Integer i) { }
```

Valeurs de retour

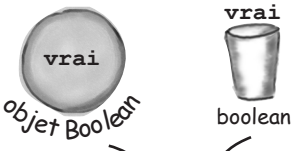
Si une méthode déclare un type de retour primitif, vous pouvez retourner soit un primitif compatible soit une référence à l'enveloppe de ce type primitif. Et si le type de retour est une enveloppe, vous pouvez retourner soit une référence au type enveloppe soit un primitif du type correspondant.

```
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");
}
```


Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Opérations sur les nombres

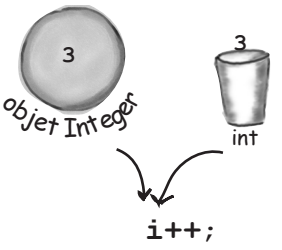
C'est probablement la possibilité la plus étrange de toutes : oui, vous pouvez maintenant utiliser un type enveloppe comme opérande dans les opérations où le type primitif est attendu. Vous pouvez par exemple appliquer l'opérateur d'incrémentation à une référence à un objet Integer!

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++;
```

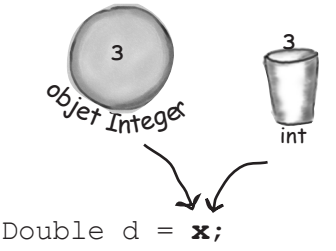
Et cela signifie que vous pouvez aussi écrire des choses du genre :

```
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);
   }
}
```


Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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.)

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Et maintenant l'inverse... transformer un nombre en chaîne de caractères

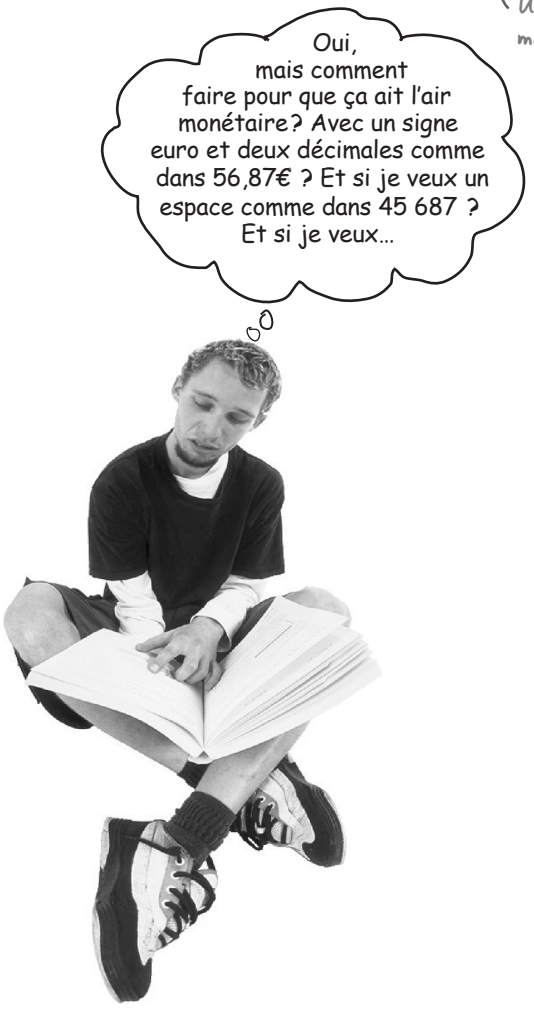
Il y a plusieurs façons de transformer un nombre en chaîne de caractères. La plus simple consiste à le concaténer à une chaîne existante.

```
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.



Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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... (1) à cela (2)

```
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.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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 chaîne 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.

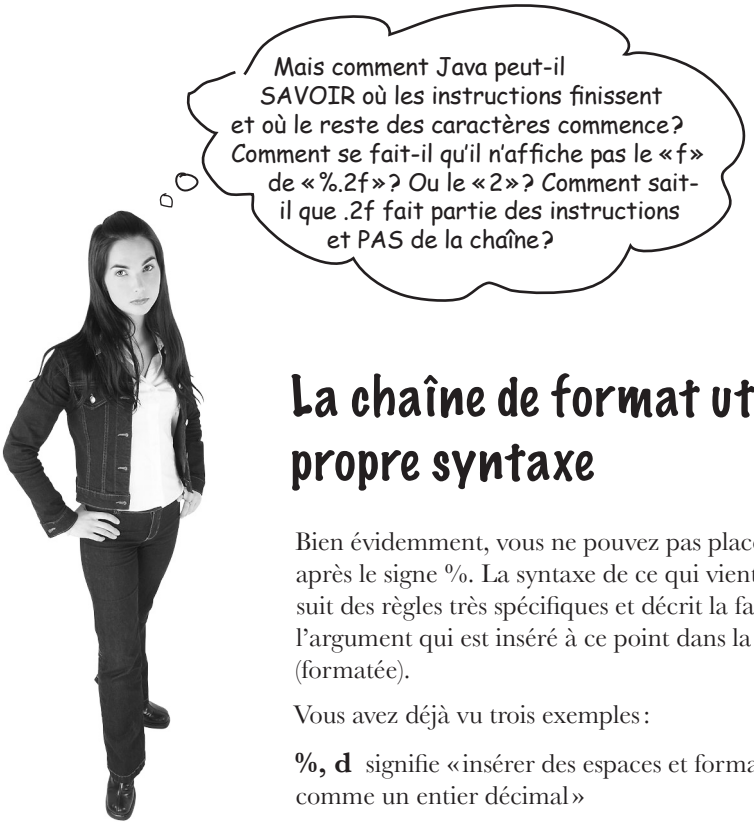
Ajouter un espace

```
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.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>



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.)

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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
```

Quand vous avez plusieurs arguments, ils sont insérés dans l'ordre dans lequel vous les passez à la méthode format().

Nous avons ajouté des espaces aux deux variables et limité le nombre en virgule flottante (la seconde variable) à deux décimales.

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.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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.

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.

```
Sunday, November 28
```

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.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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*...

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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 :

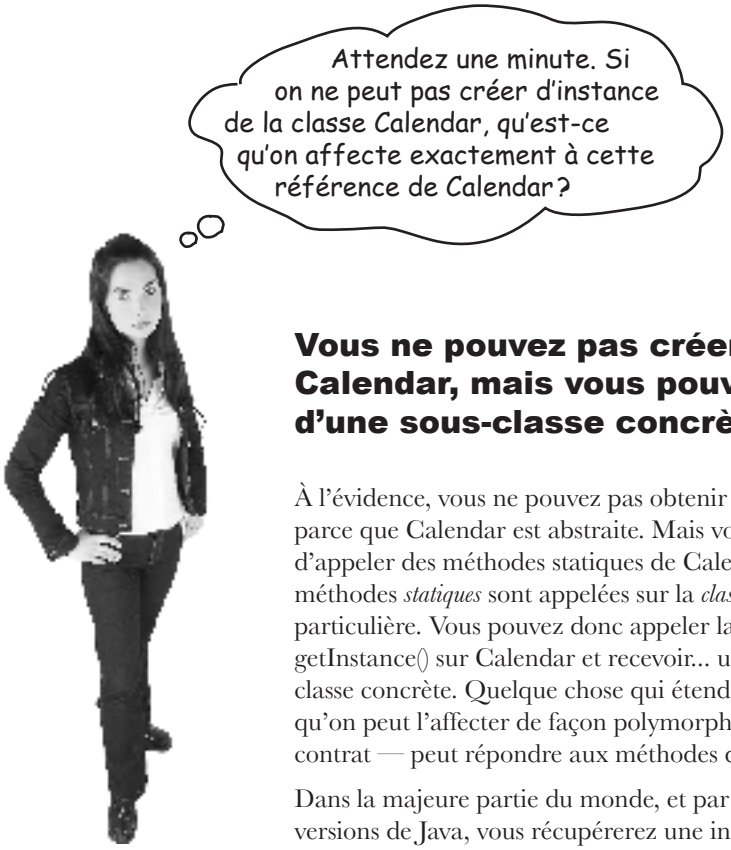
```
Calendar cal = new Calendar();
```

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**.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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...

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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.

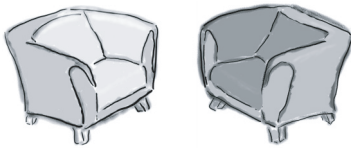
Imports statiques en action.

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 ?

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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.

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.

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.

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.

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.

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.

Mais c'est un autre cas à part. Et personne ne l'utilise de toute façon, sauf pour le débogage.

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 !

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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
```


Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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
```


Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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**

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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
```


Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Comportements 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. Quelque chose peut mal tourner. *Très* mal. Quand vous écrivez une méthode risquée, il vous faut du code capable de gérer les éventuelles retombées. Mais comment *sait-on* qu'une méthode est risquée ? Et où placer le code qui *gérera* une situation **exceptionnelle** ? Jusqu'ici, nous n'avons pas *vraiment* pris de risques. Nous avons eu quelques problèmes d'exécution, mais ils provenaient surtout d'erreurs de codage. De bogues. Et ceux-là, nous les devons les corriger au moment du développement. Non, le code de gestion des erreurs dont nous parlons ici concerne le code dont vous ne *pouvez* pas garantir qu'il s'exécutera correctement. Le code qui escompte que le fichier sera dans le bon répertoire, que le serveur fonctionnera et que le Thread restera inactif. Et nous en avons besoin *maintenant*. Parce que dans *ce* chapitre, nous allons utiliser une API à risques : JavaSound. Nous allons construire un lecteur MIDI.

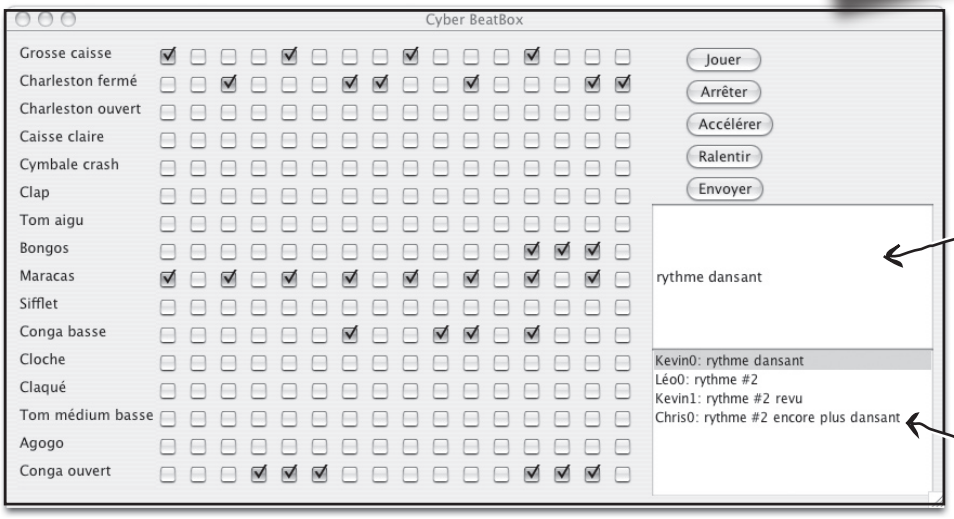
Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Construisons une boîte à rythmes

Dans les trois chapitres suivants, nous allons construire différentes applications de son, notamment une boîte à rythmes : la Cyber BeatBox. En fait, avant la fin de ce livre, nous aurons une version multi-utilisateurs, qui vous permettra d'envoyer vos oeuvres à d'autres joueurs, un peu comme dans une « chat room ». Vous allez l'écrire de bout en bout, à moins que vous ne choisissiez d'utiliser le code prêt à l'emploi pour la partie interface graphique. O.K., tous les services informatique n'ont pas l'usage d'un serveur de rythmes. Mais c'est une bonne façon d'apprendre *Java*, et construire une BeatBox est plus amusant que de développer une gestion de stocks.

La BeatBox terminée ressemble à ceci :

Vous construisez une boucle (un motif de batterie à 16 temps) en cochant les cases.



Le message que vous envoyez à vos amis avec la boucle en cliquant sur Envoyer.

Ici, les messages entrants. Cliquez sur l'un d'eux pour charger la boucle qui l'accompagne, puis sur «Jouer» pour la jouer.

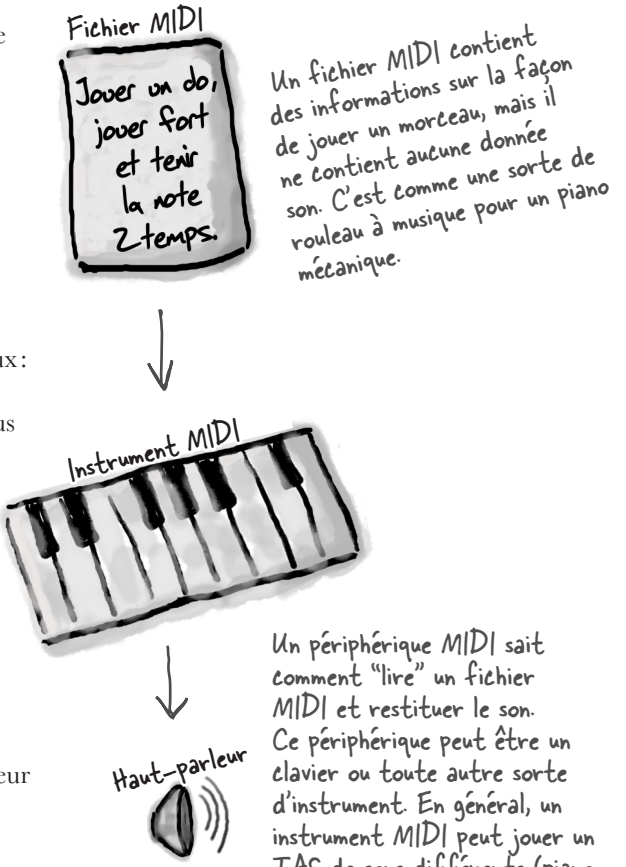
Cochez les cases pour chacun des 16 « temps ». Par exemple, la grosse caisse et les maracas jouent sur le temps 1 (sur 16), rien sur le temps 2, et sur le temps 3 les maracas et le charleston fermé... Vous voyez l'idée. Cliquer sur « Jouer » joue votre motif en boucle jusqu'à ce que vous appuyiez sur « Arrêter ». Vous pouvez « capturer » à tout moment l'un de vos propres motifs en l'envoyant au serveur BeatBox (ce qui signifie que ceux qui sont sur le réseau peuvent l'écouter). Vous pouvez également charger l'un des motifs entrants en cliquant sur le message qui l'accompagne.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Commençons par les bases

Nous avons de toute évidence un certain nombre de choses à apprendre avant de pouvoir terminer le programme, notamment comment construire une IHM avec Swing, comment se connecter à une autre machine *via* le réseau et suffisamment d'E/S pour pouvoir envoyer un message à un autre ordinateur.

Oh, et puis l'API JavaSound. C'est par là que nous allons commencer dans ce chapitre. Oubliez pour l'instant l'interface graphique, le réseau et les E/S, et concentrez-vous sur la façon d'arriver à faire sortir un son MIDI de votre ordinateur. Et ne vous inquiétez pas si vous ne connaissez rien à MIDI, si vous ne jouez d'aucun instrument et si vous ne savez pas lire une partition. Tout ce que vous avez besoin d'apprendre est abordé ici. Vous allez bientôt signer le contrat du siècle



L'API JavaSound

JavaSound est une collection de classes et d'interfaces ajoutées à Java à partir de la version 1.3. Ce ne sont pas des additifs spéciaux : elles font partie de la bibliothèque de classes standard de J2SE. JavaSound est composé de deux parties : MIDI et Sampled. Nous n'utiliserons que MIDI dans ce livre. MIDI (*Musical Instrument Digital Interface*) est un protocole standard qui permet à différentes sortes d'équipements son électroniques de communiquer. Mais pour notre application BeatBox, vous pouvez vous représenter un fichier MIDI comme une sorte de rouleau à musique que vous insérez dans un périphérique que vous pouvez voir comme *un «piano mécanique»* high-tech. Autrement dit, les données MIDI ne sont pas des sons, mais des *instructions* qu'un instrument MIDI peut restituer. Ou, pour prendre une autre analogie, vous pouvez voir un fichier MIDI comme un document HTML et l'instrument comme un navigateur Web.

Les données MIDI indiquent *quoi* faire (jouer un do, comment le frapper et combien de temps tenir la note, etc.) mais sont totalement muettes sur le *son* réel que vous entendez. MIDI ne sait pas comment créer le son d'une flûte, d'un piano ou de la guitare de Jimmy Hendrix. Pour obtenir le *son* réel, il nous faut un instrument (un périphérique MIDI) qui sache lire et exécuter un fichier MIDI. Ce périphérique ressemble généralement plus à un groupe ou à un orchestre. Il peut s'agir d'un périphérique physique, comme les claviers électroniques utilisés par les groupes de rock, ou même d'un instrument entièrement logiciel, qui réside dans votre ordinateur.

Pour notre BeatBox, nous n'utiliserons que l'instrument logiciel intégré fourni avec Java. On le nomme *synthétiseur* parce qu'il fabrique des sons. Des sons que vous pouvez *entendre*.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Il nous faut d'abord un séquenceur

Avant de pouvoir jouer un son quelconque, il nous faut un objet Sequencer. Le séquenceur est l'objet qui reçoit toutes les données MIDI et les envoie aux bons instruments. C'est lui qui joue la musique. Un séquenceur peut faire beaucoup de choses différentes, mais, dans ce livre, nous l'utilisons strictement comme un périphérique de restitution. Comme un lecteur de CD sur votre chaîne stéréo, mais avec quelques fonctionnalités supplémentaires. La classe Sequencer se trouve dans le package javax.sound.midi (qui fait partie de la bibliothèque standard Java à partir de la version 1.3). Commençons donc par nous assurer que nous pouvons créer (ou obtenir) un objet Sequencer.

```
import javax.sound.midi.*;

public class MusicTest1 {

    public void jouer() {
        Sequencer sequenceur = MidiSystem.getSequencer();
        System.out.println("Nous avons un séquenceur");
    } // fin de la méthode jouer()

    public static void main(String[] args) {
        MusicTest1 mt = new MusicTest1();
        mt.jouer();
    } // fin de la méthode main()
} // fin de la classe
```

importation du package javax.sound.midi.

Nous avons besoin d'un objet Sequencer. C'est le composant central du périphérique/instrument MIDI que nous utilisons, celui qui... séquence toutes les informations MIDI pour les transformer en « morceau.» Mais nous n'allons pas en créer un de toutes pièces – nous allons demander à MidiSystem de nous en donner un.

Quelque chose ne va pas!

Ce code ne se compilera pas! Le compilateur dit qu'il y a une "exception non gérée" qui doit être interceptée ou déclarée.

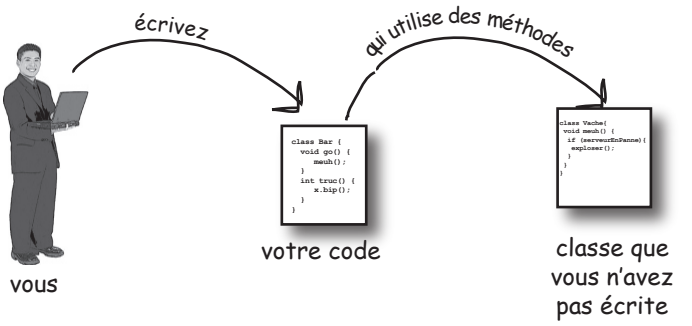
Fichier Edition Fenêtre Aide Plaît-il?

```
% javac MusicTest1.java
MusicTest1.java:13: unreported exception javax.sound.midi.
MidiUnavailableException; must be caught or declared to be
thrown
    Sequencer sequenceur = MidiSystem.getSequencer();
                                      ^
1 errors
```

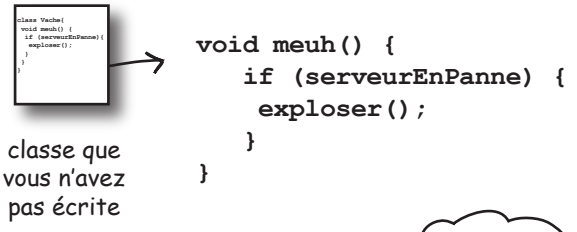

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Que se passe-t-il quand une méthode que vous voulez appeler (probablement dans une classe que vous n'avez pas écrite) est risquée ?

1. **Supposons que vous voulez appeler une méthode d'une classe que vous n'avez pas écrite.**

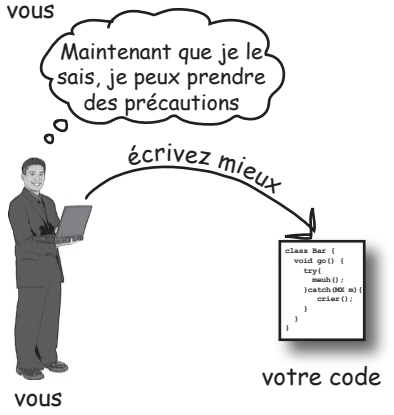


2. **Cette méthode fait quelque chose de risqué, qui peut ne pas fonctionner au moment de l'exécution**



3. **Vous devez *savoir* que la méthode que vous appelez est risquée.**

4. **Vous écrivez alors du code qui gère l'erreur *si elle se produit*. Vous devez être préparé, au cas où.**



Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

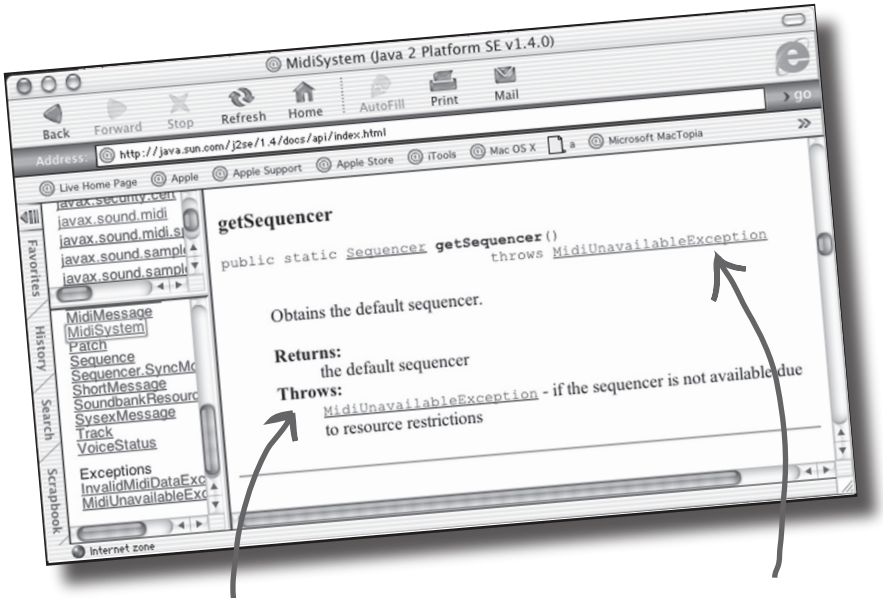
Les méthodes Java utilisent des *exceptions* pour dire au code appelant :

« Il y a un problème. J'ai échoué. »

En Java, la gestion des exceptions est un mécanisme bien pensé qui permet de traiter toutes les « situations exceptionnelles » qui peuvent surgir au moment de l'exécution, et de placer tout le code de gestion des erreurs de façon très lisible à un seul endroit. Elle s'appuie sur le fait que vous savez que la méthode que vous appelez est risquée (autrement dit qu'elle *peut* générer une exception), pour que vous puissiez écrire une portion de code qui prenne en compte cette possibilité. Si vous savez qu'une exception peut se produire quand vous appelez une méthode donnée, vous êtes *préparé* au problème qui a provoqué l'exception, et peut-être même en mesure de le résoudre.

Mais comment savez-vous si une méthode lance (déclenche) une exception ? Vous trouvez une clause **`throws`** dans sa déclaration.

La méthode `getSequencer()` prend un risque. Elle peut échouer lors de l'exécution. Elle doit donc « déclarer » le risque que *vous* prenez quand vous l'appelez.



Cette partie vous dit QUAND vous pouvez avoir cette exception – dans ce cas à cause d'une restriction des ressources (ce qui peut signifier que le séquenceur est déjà utilisé).

La documentation de l'API vous indique que getSequencer() peut lancer une exception : MidiUnavailableException. Une méthode doit déclarer les exceptions qu'elle peut lancer.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Le compilateur doit savoir que VOUS savez que vous appelez une méthode risquée.

Si vous enveloppez le code risqué dans un bloc **try/catch**, le compilateur se décontracte.

Un bloc try/catch indique au compilateur que vous *savez* que quelque chose d'exceptionnel peut arriver et que vous êtes préparé à le gérer. Il ne se soucie pas de la *façon* dont vous le gérez, mais il veut simplement savoir que vous vous en occupez.

Cher compilateur,
Je sais que je prends un risque, mais ne pensez-vous pas que ça en vaut la peine? Que dois-je faire?

Cher lecteur,
La vie est courte (surtout sur le tas). Prenez le risque. Essayez. Mais au cas où les choses tourneraient mal, faites bien attention à intercepter tous les problèmes avant que l'enfer ne se déchaîne.

```
import javax.sound.midi.*;

public class MusicTest1 {
    public void jouer() {

        try {
            Sequencer sequenceur = MidiSystem.getSequencer();
            System.out.println("Nous avons un séquenceur");
        } catch(MidiUnavailableException ex) {
            System.out.println("La poisse");
        }
    } // fin de la méthode jouer()

    public static void main(String[] args) {
        MusicTest1 mt = new MusicTest1();
        mt.play();
    } // fin de la méthode main()
} // fin de la classe
```

On place l'élément risqué dans un bloc "try".

Et on écrit un bloc "catch" pour spécifier quoi faire si la situation exceptionnelle se présente – ici une MidiUnavailableException est lancée par l'appel de getSequencer().

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

N'essayez pas ceci chez vous.

Une exception est un objet... de type Exception.

C'est une chance, parce que ce serait beaucoup plus difficile à mémoriser si les exceptions étaient de type Brocoli.

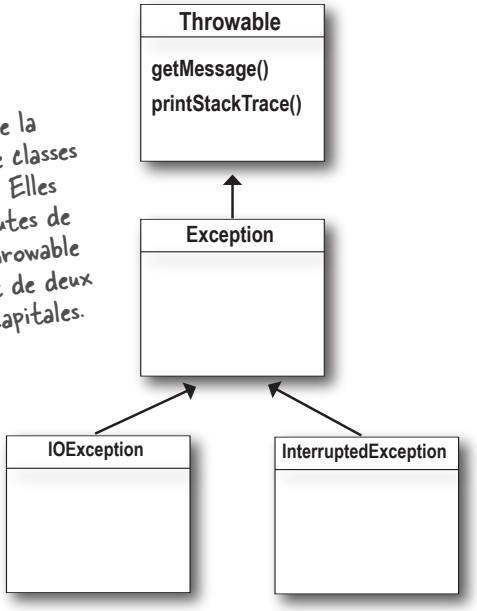
Souvenez-vous des chapitres sur le polymorphisme : un objet de type Exception *peut être une* instance de n'importe quelle sous-classe d'Exception.

Puisqu'une *Exception* est un objet, ce que vous interceptez avec **`catch`** *est un objet. Dans le* code suivant, l'argument de **`catch`** est déclaré de type Exception, et la variable référence passée en paramètre est *ex*.

```
try {
    // méthode risquée

} catch(Exception ex) {
    // essai de récupération
}
```

C'est comme si on déclarait un argument de méthode.

Ce code ne s'exécute que si une exception est lancée.



Une partie de la hiérarchie de classes d'Exception. Elles dérivent toutes de la classe Throwable et héritent de deux méthodes capitales.

Ce que vous écrivez dans un bloc catch dépend de l'exception lancée. Si par exemple le serveur est en panne, vous pouvez utiliser le bloc catch pour essayer un autre serveur. Si un fichier est absent, vous pouvez demander à l'utilisateur de le chercher.

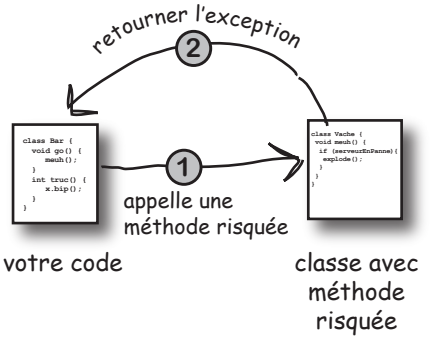
Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Si c'est *votre* code qui intercepte l'exception, quel est celui qui la lance ?

Lorsque vous programmerez en Java, vous passerez beaucoup plus de temps à gérer des exceptions qu'à les créer et les lancer vous-même. Pour l'instant, contentez-vous de savoir que quand votre code appelle une méthode risquée — une méthode qui déclare une exception — c'est cette méthode qui renvoie l'exception, à vous, l'appelant.

En réalité, vous avez peut-être écrit les deux classes. Mais peu importe qui a écrit le code... ce qui compte, c'est de savoir quelle est la méthode qui lance l'exception et quelle est celle qui l'intercepte.

Quand quelqu'un écrit du code qui pourrait lancer une exception, il doit déclarer l'exception.



① Code lançant l'exception :

Cette méthode DOIT dire au monde (déclarer) qu'elle lance une exception MauvaiseException.

```
public void prendreRisque() throws MauvaiseException {
   if (abandonnerToutEspoir) {
      throw new MauvaiseException();
   }
}
```

créer une nouvelle Exception (un nouvel objet) et la lancer.

Une méthode intercepte ce qu'une autre méthode lance. Une exception est toujours relancée à l'appelant.

La méthode qui lance doit déclarer qu'elle peut lancer l'exception.

② Votre code qui *appelle* la méthode risquée :

```
public void croiserLesDoigts() {
   try {
      unObjet.prendreRisque();
   } catch (MauvaiseException ex) {
      System.out.println("Aaargh!");
      ex.printStackTrace();
}
```

Si l'exception n'est pas récupérable, affichez AU MOINS l'état de la pile d'appels avec la méthode printStackTrace() dont toutes les exceptions héritent.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

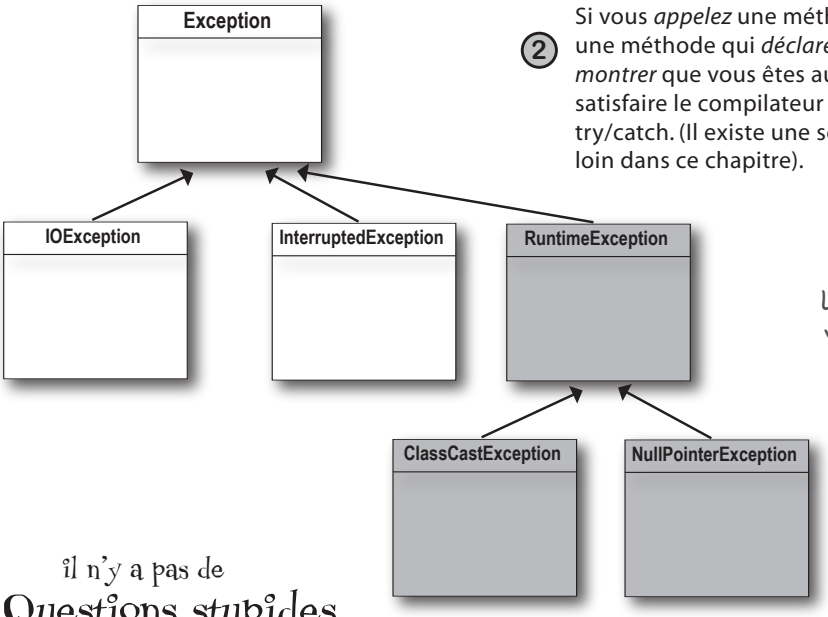
Le compilateur vérifie tout sauf les RuntimeExceptions.

Le compilateur exige :

1. Si vous *lancez* une exception dans votre code, vous *devez* la déclarer au moyen du mot-clé *throws* dans votre déclaration de méthode.

2. Si vous *appelez* une méthode qui lance une exception (autrement dit une méthode qui *déclare* qu'elle lance une exception), vous devez *montrer* que vous êtes au courant de cette possibilité. Une façon de satisfaire le compilateur consiste à encapsuler l'appel dans un bloc try/catch. (Il existe une seconde façon que nous verrons un peu plus loin dans ce chapitre).

Les exceptions qui ne sont PAS des sous-classes de RuntimeException sont vérifiées par le compilateur. On parle à leur propos d' « exceptions vérifiées.»



Les RuntimeExceptions ne sont PAS vérifiées par le compilateur. On les appelle (grosse surprise) « exceptions non vérifiées ». Vous pouvez lancer, intercepter et déclarer des RuntimeExceptions, mais vous n'y êtes pas obligé et le compilateur ne s'en occupera pas.

il n'y a pas de
Questions stupides

Q : Attendez une minute ! Pourquoi est-ce la PREMIÈRE fois que nous devons gérer une exception avec try/catch ? Et les exceptions que j'ai déjà rencontrées comme NullPointerException et DivideByZeroException ? J'ai même eu une NumberFormatException avec la méthode Integer.parseInt(). Comment se fait-il que nous n'ayons pas eu à les intercepter ?

R : Le compilateur s'occupe de toutes les sous-classes d'Exception, *sauf* si elles sont d'un type spécial, RuntimeException. Toute classe d'exception qui dérive de RuntimeException a un laisser-passer. Les RuntimeExceptions peuvent être lancées n'importe où, avec ou sans clause throws, avec ou sans bloc try/catch. Le compilateur ne se donne pas la peine de vérifier si une méthode déclare qu'elle lance une RuntimeException, ni si l'appelant reconnaît que cette exception pourrait se produire lors de l'exécution.

Q : Mais POURQUOI le compilateur ne s'occupe-t-il pas des exceptions qui se produisent à l'exécution ? Est-ce qu'elles ne sont pas tout aussi gênantes ?

R : La plupart des RuntimeExceptions viennent d'un problème dans la logique de votre code, non d'une condition qui échoue au moment de l'exécution et que vous pourriez prédire ou empêcher. Vous *ne pouvez pas* garantir la présence d'un fichier. Vous *ne pouvez pas* garantir que le serveur fonctionnera. Mais vous *pouvez* veiller à ce qu'un indice ne dépasse pas les limites d'un tableau (c'est la raison d'être de l'attribut .length).

Vous VOULEZ que les RuntimeExceptions se produisent lors du développement et des tests. Vous ne voulez pas coder un try/catch, avec toute la surcharge que cela implique, pour intercepter quelque chose qui ne devrait pas se produire.

Un try/catch sert à gérer les situations exceptionnelles, pas les bogues. Utilisez les blocs catch pour récupérer une situation que vous ne pouvez pas garantir. Ou, au strict minimum, affichez un message et l'état de la pile pour que tout le monde puisse comprendre ce qui s'est passé.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

POINTS D'IMPACT

- Une méthode peut lancer une exception quand quelque chose échoue au moment de l'exécution.
- Une exception est toujours un objet de type Exception. (Ce qui signifie que l'objet, en vertu du polymorphisme, appartient à une classe située quelque part en-dessous d'Exception dans la hiérarchie d'héritage.)
- Le compilateur ne prête PAS attention aux exceptions du type **RuntimeException**. Une RuntimeException n'a pas besoin d'être déclarée ni encapsulée dans un bloc try/catch (même si vous avez le droit de faire l'un ou l'autre, ou les deux).
- Toutes les exceptions dont le compilateur s'occupe sont appelées « exceptions vérifiées », ce qui signifie en réalité exceptions vérifiées *par le compilateur*. Seules les RuntimeExceptions sont exclues de cette vérification. Toutes les autres doivent être traitées dans votre code.
- Une méthode lance une exception avec le mot-clé **`throw`**, suivi d'un nouvel objet de type Exception :

```
throw new PasDeCafeException();
```

- Les méthodes qui *pourraient* lancer une exception vérifiée ***doivent*** l'annoncer avec une déclaration **`throws Exception`**.
- Si vous appelez une méthode qui appelle une exception vérifiée, vous devez dire au compilateur que vous avez pris vos précautions.
- Si vous êtes prêt à gérer l'exception, encapsulez l'appel dans un bloc try/catch, et placez le traitement de l'exception dans le bloc catch.
- Si vous n'êtes pas prêt à gérer l'exception, vous pouvez satisfaire le compilateur en « l'esquivant » officiellement. Nous en parlerons un peu plus loin dans ce chapitre.

Astuce métacognitive

Si vous essayez d'apprendre quelque chose de nouveau, que ce soit la *dernière* chose avant d'aller vous coucher. Dès que vous aurez posé ce livre (en supposant que vous puissiez vous y arracher) ne lisez rien de plus difficile que le dos d'une boîte de conserve. Votre cerveau a besoin de temps pour traiter ce que vous avez lu et appris. Cela peut prendre quelques *heures*. Si vous essayez d'empiler quelque chose de nouveau sur Java, Java pourrait bien disparaître.

Bien sûr, cela ne concerne pas les apprentissages moteurs. Travailler votre droite au punching-ball ne devrait pas affecter votre apprentissage de Java.

Pour obtenir les meilleurs résultats, lisez ce livre (ou regardez au moins les images) juste avant d'aller dormir.

À vos crayons

Selon vous, lesquelles de ces actions pourraient lancer une exception dont le compilateur se soucierait ? Nous ne nous intéressons qu'à ce que vous ne pouvez pas contrôler dans votre code. Nous avons fait la première.

(Parce que c'était la plus facile.)

Ce que vous voulez faire	Problème possible
✓ vous connecter à un serveur distant	le serveur est en panne
__ dépasser les bornes d'un tableau	
__ afficher une fenêtre à l'écran	
__ interroger une base de données	
__ voir si un fichier est là où vous *pensez*	
__ créer un nouveau fichier	
__ lire un caractère sur la ligne de commande	

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Contrôle de flot dans les blocs try/catch

Quand vous appelez une méthode risquée, deux choses peuvent se produire. Soit la méthode réussit et le bloc try se termine, soit la méthode retourne une exception à la méthode appelante.

Si le bloc try réussit

(prendreRisque() ne lance *pas* d'exception)

```
try {
  1 Foo f = x.prendreRisque();
    int b = f.getNum();

} catch (Exception ex) {
    System.out.println("Raté !");
}
  2 System.out.println("Réussi !");
```

Le bloc try s'exécute d'abord, puis le code qui suit le catch.

Le code du bloc catch ne s'exécute jamais.

```
Fichier Edition Fenêtre Aide RisqueTout
%java Tester
Réussi !
```

Si le bloc try échoue

(prendreRisque() *lance* une exception)

```
try {
  1 Foo f = x.prendreRisque();
    int b = f.getNum();

} catch (Exception ex) {
  2   System.out.println("Raté!");
}
  3 System.out.println("Réussi!");
```

Le bloc try s'exécute, mais l'appel de prendreRisque() lance une exception, et le reste du bloc try ne s'exécute pas.

Le bloc catch s'exécute, puis la méthode continue.

Le reste du bloc try ne s'exécute jamais, ce qui est une bonne chose parce que le reste du bloc dépend du succès de l'appel de prendreRisque().

```
Fichier Edition Fenêtre Aide RisqueTout
%java Tester
Raté !
Réussi !
```


Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

finally : pour ce que vous voulez exécuter *dans tous les cas*

Si vous voulez cuire un gâteau, vous commencez par allumer le four.

Si votre essai est un **échec** complet, ***vous devez éteindre le four.***

Si votre essai est **une réussite**, ***vous devez éteindre le four.***

Vous devez éteindre le four dans tous les cas !

Un bloc finally est l'endroit où placer le code qui s'exécute *indépendamment* de l'exception.

```
try {
  allumerLeFour();
  x.cuire();
} catch (CuissonException ex) {
     ex.printStackTrace();
} finally {
     eteindreLeFour();
}
```

Sans finally, vous devez placer la méthode eteindreLeFour() à la *fois* dans le bloc try et dans le bloc catch, parce que ***vous devez éteindre le four quoi qu'il arrive***. Un bloc finally vous permet de placer tout le code de « nettoyage » à *un seul* endroit, au lieu de le dupliquer comme ceci :

```
try {
  allumerLeFour();
  x.cuire();
  eteindreLeFour();
} catch (CuissonException ex) {
     ex.printStackTrace();
     eteindreLeFour();
}
```

Si le bloc try échoue (exception), le contrôle de flot passe immédiatement au bloc catch. Quand le bloc catch se termine, le bloc finally s'exécute. Quand le bloc finally se termine, le reste de la méthode continue.

Si le bloc try réussit (*pas* d'exception), le contrôle de flot saute le bloc catch et passe au bloc finally. Quand le bloc finally se termine, le reste de la méthode continue.

Si le bloc try ou le bloc catch a une instruction return, finally s'exécute quand même ! Le flot saute à finally, puis revient au return.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

À vos crayons

Contrôle de flot

Regardez le code à gauche. Selon vous, quel sera le résultat de ce programme ? Et quel serait-il si on transformait la troisième ligne en :
`String test = "oui";` **?**

Il est entendu que HorribleExeption étend Exeption.

```
public class TestExceptions {

  public static void main(String [] args) {

    String test = "non";
    try {
       System.out.println("début de try");
       prendreRisque(test);
       System.out.println("fin de try");
    } catch (HorribleException he) {
        System.out.println("horrible exception");
    } finally {
        System.out.println("finally");
    }
    System.out.println("fin de main");
  }

  static void prendreRisque(String test) throws HorribleException {
    System.out.println("début de risque");
    if ("oui".equals(test)) {

      throw new HorribleException();
    }
    System.out.println("fin de risque");
    return;
  }
}
```

Résultat quand `test = "non"`

Résultat quand `test = "oui"`

Quand test = "non" : début de try - début de risque - fin de risque - fin de try - finally - fin de main
Quand test = "oui" : début de try - début de risque - horrible exception - finally - fin de main

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Avons-nous dit qu'une méthode peut lancer plusieurs exceptions ?

Une méthode peut lancer plusieurs exceptions si elle en a besoin. Mais elle doit déclarer *toutes* les exceptions vérifiées qu'elle peut lancer (mais si deux exceptions ou plus ont une superclasse commune, la méthode peut se contenter de déclarer la superclasse).

Intercepter plusieurs exceptions

Le compilateur vérifiera que vous avez géré toutes les exceptions vérifiées lancées par la méthode que vous appelez. Empilez les blocs *catch* en-dessous du *try*, l'un après l'autre. Il arrive que l'ordre dans lequel vous empilez les blocs catch importe, mais nous verrons cela un peu plus tard.

```
public class Lessive {
    public void faireLaLessive() throws PantalonException, LingerieException {
       // code pouvant lancer l'une ou l'autre exception
    }
}
```

Cette méthode déclare DEUX exceptions.

```
public class Foo {
    public void go() {
       Lessive lessive = new Lessive();
       try {
            lessive.faireLaLessive();
        } catch(PantalonException pex) {
            // code de récupération

        } catch(LingerieException lex) {
            // code de récupération
        }
    }
}
```

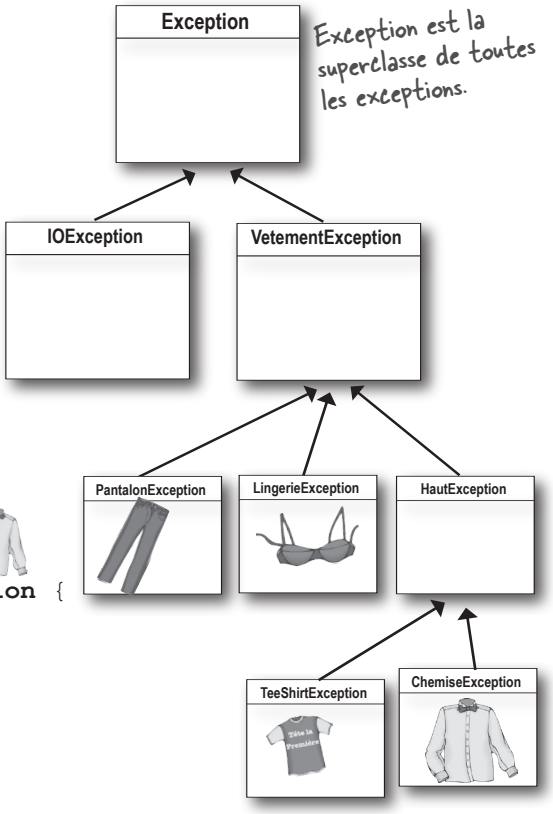
Si faireLaLessive() lance une PantalonException, elle atterrit dans le bloc catch de PantalonException.

Si faireLaLessive() lance une LingerieException, elle atterrit dans le bloc catch de LingerieException.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Les exceptions sont polymorphes

Souvenez-vous que les exceptions sont des objets. Ces objets n'ont rien de spécial, excepté qu'on peut les ***lancer***. Donc, comme tout bon objet, une Exception peut être référencée de façon polymorphe. Un *objet* LingerieException, par exemple, peut être affecté à une *référence* VetementException. Un objet PantalonException peut être affecté à une référence Exception. Vous voyez l'idée. L'avantage des exceptions est qu'une méthode n'a pas besoin de déclarer explicitement toutes les exceptions possibles qu'elle pourrait lancer : elle peut déclarer une superclasse des exceptions. Pareil pour les blocs catch — vous n'avez pas besoin d'écrire un catch pour chaque exception possible tant que le ou les blocs catch existants peuvent gérer chaque exception lancée.



① Vous pouvez DÉCLARER des exceptions en utilisant un supertype des exceptions que vous lancez.

```
public void faireLaLessive() throws VetementException {
```

Déclarer une VetementException vous permet de lancer n'importe quelle sous-classe de VetementException. La méthode faireLaLessive() peut alors lancer une PantalonException, une LingerieException, une TeeShirtException et une ChemiseException sans les déclarer explicitement.

② Vous pouvez INTERCEPTER des exceptions en utilisant un supertype de l'exception lancée.

```
try {
    lessive.faireLaLessive();
} catch(VetementException vex) {
    // code de récupération
}
```

Peut intercepter une sous-classe de VetementException quelconque.

```
try {
    lessive.faireLaLessive();
} catch(HautException hex) {
    // code de récupération
}
```

Ne peut intercepter que TeeShirtException et ChemiseException

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Ce n'est pas parce que vous POUVEZ tout intercepter avec un seul super bloc catch polymorphe que vous DEVEZ toujours le faire.

Vous pourriez écrire le code qui gère les exceptions en ne spécifiant qu'un unique bloc catch, et en n'utilisant que le supertype Exception dans la clause catch, pour pouvoir intercepter toute exception susceptible d'être lancée.

```
try {
    lessive.faireLaLessive();
} catch(Exception ex) {
      // code de récupération...
}
```

Récupération de QUOI ? Ce bloc catch interceptera ABSOLUMENT TOUTES les exceptions, vous empêchant automatiquement de savoir ce qui s'est passé.

Écrivez un bloc catch pour chaque exception que vous devez gérer de façon spécifique.

Si par exemple votre code traite différemment (ou récupère différemment) une TeeShirtException et une LingerieException, écrivez deux blocs catch distincts. Mais si vous traitez tous les autres types de VetementException de la même manière, ajoutez une clause catch avec une VetementException pour gérer le reste.

```
try {
    lessive.faireLaLessive();
} catch(TeeShirtException tex) {
    // récupération de TeeShirtException
} catch(LingerieException lex) {
    // récupération de LingerieException
} catch(VetementException vex) {
    // récupération de toutes les autres exceptions
}
```

TeeShirtException et LingerieException devant être récupérées différemment, vous avez besoin de deux blocs catch.

Toutes les autres VetementExceptions sont interceptées ici.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Plusieurs blocs catch doivent être ordonnés, du plus petit au plus grand

Les TeeShirtExceptions sont interceptées ici, mais aucune autre exception ne rentre.

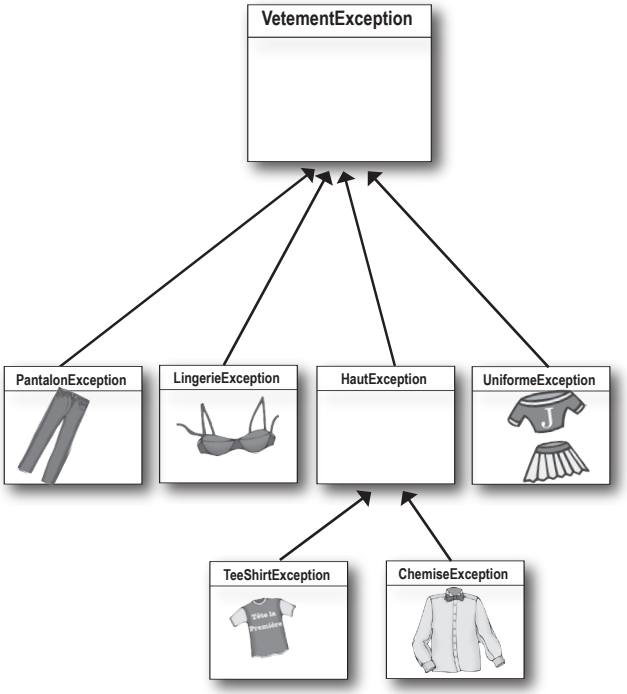
```
catch(TeeShirtException tex)
```

Les TeeShirtExceptions n'arriveront jamais jusqu'ici, mais toutes les autres sous-classes de HautException sont interceptées.

```
catch(HautException hex)
```

Toutes les VetementExceptions sont interceptables ici, même si TeeShirtException et HautException n'iront jamais aussi loin.

```
catch(VetementException vex)
```



Plus on est proche du sommet de la hiérarchie d'héritage, plus le « panier » de catch est grand. Plus on descend, plus les sous-classes d'Exception sont spécialisées et plus le « panier » est petit. Ce n'est que du bon vieux polymorphisme.

Un catch utilisant HautException est suffisamment grand pour accepter une TeeShirtException ou une ChemiseException (et toute autre future sous-classe de quoi que ce soit qui dérive de HautException). Une VetementException est encore plus grande (autrement dit, on peut référencer plus de choses en utilisant un type VetementException). Elle peut accepter une exception de type VetementException (évidemment), et de tous ses sous-types : PantalonException, UniformeException, LingerieException et HautException. La mère de tous les arguments de catch est le type **Exception** ; elle interceptera *toutes* les exceptions, y compris celles qui ne sont pas vérifiées par le compilateur et se produisent lors de l'exécution, et vous ne l'utiliserez probablement pas en dehors des phases de tests.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Vous ne pouvez pas mettre les grands paniers dans les petits.

Vous pouvez le faire, mais cela ne compilera pas. Les blocs catch ne sont pas comme les méthodes surchargées, où la meilleure correspondance est choisie. La JVM commence simplement par le premier bloc catch, et continue jusqu'à ce qu'elle en trouve un suffisamment grand (suffisamment élevé dans la hiérarchie d'héritage) pour gérer l'exception. Si votre premier bloc est **`catch(Exception ex)`**, le compilateur sait qu'il ne sert à rien de s'occuper des autres — ils ne seront jamais atteints.

La taille compte quand vous avez plusieurs blocs catch. Celui qui a le plus grand panier doit être tout en bas. Sinon, ceux qui en ont de plus petits ne servent à rien.

Ne faites pas ça!

```
try {
  lessive.faireLaLessive();

 }  catch(VetementException vex) {
    // récupération de VetementException

 }  catch(LingerieException lex) {
   // récupération de LingerieException

 }  catch(HautException hex) {
    // récupération de HautException
 }
```

Les «frères et soeurs» peuvent être dans n'importe quel ordre, parce que les uns ne peuvent pas intercepter les exceptions des autres.

Vous pouvez placer HautException au-dessus de LingerieException, et personne n'y verra d'objection. Même si HautException est d'un type plus grand et peut intercepter d'autres types (ses propres sous-types) elle ne peut pas intercepter de LingerieException, donc il n'y a pas de problème.

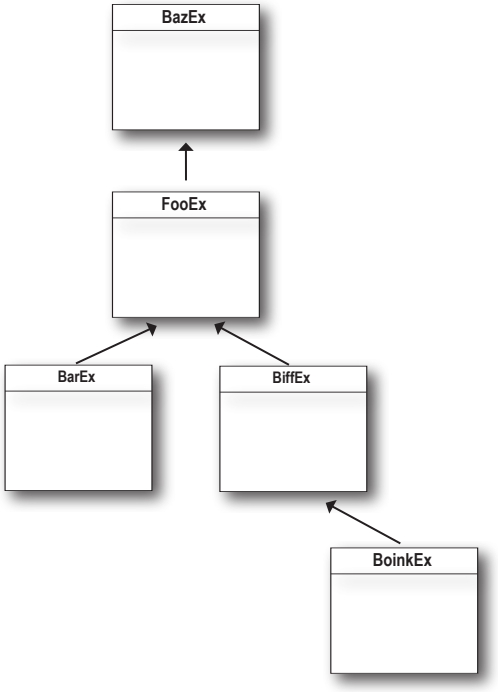
Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

À vos crayons

Supposons que le code de ce bloc try/catch soit valide. Votre tâche consiste à tracer deux diagrammes de classes différents qui représentent fidèlement les sous-classes d'Exception. Autrement dit, quelles seraient les structures d'héritage qui rendraient légal les clauses try et catch de cet exemple de code ?

```
try {
  x.prendreRisque();
}  catch(AlphaEx a) {
    // récupération de AlphaEx
}  catch(BetaEx b) {
   // récupération de BetaEx
}  catch(GammaEx c) {
    // récupération de GammaEx
}  catch(DeltaEx d) {
    // récupération de DeltaEx
}
```

Votre tâche consiste à créer deux structures try / catch *légales* différentes (similaires au code ci-dessus), pour traduire le diagramme de classe figurant à gauche. On supposera que TOUTES ces exceptions peuvent être lancées par la méthode du bloc try.



Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Quand vous ne voulez pas gérer une exception...

esquivez-la

Si vous ne voulez pas gérer une exception, vous pouvez l'**esquiver** en la **déclarant**.

Quand vous appelez une méthode risquée, le compilateur doit savoir que vous êtes au courant. La plupart du temps, cela signifie « encapsuler » l'appel dans un bloc try/catch. Mais vous disposez d'une autre solution : l'esquiver et laisser la méthode appelée intercepter l'exception.

C'est facile — il suffit de *déclarer* que *vous* lancez les exceptions. Même si, techniquement, ce n'est pas *vous* qui les lancez. Peu importe. Vous êtes celui par qui l'exception arrive.

Mais si vous esquivez l'exception, vous n'avez pas de bloc try/catch. Que se passe-t-il donc quand la méthode (faireLaLessive()) *lance* l'exception ?

Quand une méthode lance une exception, elle est dépilée immédiatement, et l'exception est lancée à la méthode suivante dans la pile — la méthode *appelante*. Mais si la méthode appelante esquive l'exception, elle n'a pas de clause catch pour l'intercepter. Elle est donc dépilée immédiatement, l'exception est lancée à la méthode suivante et ainsi de suite... Où cela se termine-t-il ? Vous allez le savoir bientôt.

```
public void foo() throws HorribleException {
    // appel d'une méthode risquée sans try/catch
    lessive.faireLaLessive();
}
```

Vous ne la lancez pas VRAIMENT, mais puisque vous n'avez pas de bloc try/catch, VOUS êtes maintenant la « méthode risquée ». Parce que maintenant, ce qui VOUS appelle doit gérer l'exception.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Esquiver (en déclarant) ne fait que retarder l'inévitable

Tôt ou tard, *quelqu'un* doit gérer l'exception. Mais si *main()* l'esquive également ?

```
public class MachineALaver {
    Lessive lessive = new Lessive();

    public void foo() throws VetementException {
        lessive.faireLaLessive();
    }

   public static void main (String[] args) throws VetementException {
       MachineALaver a = new MachineALaver();
       a.foo();
   }
}
```

Les deux méthodes esquivant l'exception (en la déclarant), il n'y a plus personne pour la gérer ! Pour le compilateur, tout va bien.

1 faireLaLessive() lance une VetementException

main() appelle foo()

foo() appelle faireLaLessive()

faireLaLessive() s'exécute et lance une VetementException

2 foo() esquive l'exception

faireLaLessive() est immédiatement dépilée et l'exception passe à foo().

Mais foo() n'a pas de try/catch, donc…

3 main() esquive l'exception

foo() est dépilée immédiatement et l'exception passe à… Qui ? Quoi ? Il ne reste plus personne que la JVM, et elle pense « Vous ne pensez quand-même pas que c'est MOI qui vais vous sortir de là ? ».

4 la JVM refuse de continuer

Nous utilisons le tee-shirt pour représenter une VetementException. On sait, on sait... Vous auriez préféré les blue-jeans.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Gérer ou déclarer. Telle est la loi.

Nous connaissons maintenant les deux façons de satisfaire le compilateur quand nous appelons une méthode risquée (qui lance une exception).

1 GÉRER

Envelopper l'appel risqué dans un bloc try/catch

```
try {
    lessive.faireLaLessive();
} catch(VetementException vex) {
      // code de récupération
}
```

Mieux vaudrait que ce catch soit assez "grand" pour gérer toutes les exceptions que faireLaLessive() pourrait lancer. Sinon le compilateur se plaindra que vous n'interceptez pas toutes les exceptions.

2 DÉCLARER (esquiver)

Déclarer que VOTRE méthode lance les mêmes exceptions que la méthode risquée que vous appelez.

```
void foo() throws VetementException {
    lessive.faireLaLessive();
}
```

La méthode faireLaLessive() lance une VetementException, mais la méthode foo() l'esquive en la déclarant. Pas de try/catch.

Mais cela signifie maintenant que tout code qui appelle la méthode foo() doit respecter la loi «gérer ou déclarer». Si foo() esquive l'exception (en la déclarant) et que main() appelle foo(), alors main() doit gérer l'exception.

```
public class MachineALaver {
    Lessive lessive = new Lessive();

    public void foo() throws VetementException {
        lessive.faireLaLessive();
    }

   public static void main (String[] args) {
        MachineALaver a = new MachineALaver();
        a.foo();
   }
}
```

PROBLÈME!!

Maintenant main() ne compile plus, et nous avons un message signalant une exception non gérée. Pour le compilateur, la méthode foo() lance une exception.

Comme la méthode foo() esquive la VetementException lancée par faireLaLessive(), main() doit envelopper a.foo() dans un bloc try/catch, ou bien déclarer qu'elle aussi lance une VetementException!

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Revenons à notre application musicale...

Vous avez peut-être totalement oublié, mais nous avons commencé ce chapitre en jetant un coup d'œil sur du code JavaSound. Nous avons créé un objet Sequencer, mais il ne compilait pas parce que la méthode Midi.getSequencer() déclarait une exception vérifiée par le compilateur (MidiUnavailableException). Mais nous pouvons maintenant le corriger en enveloppant l'appel dans un bloc try/catch.

```
public void jouer() {
        try {

            Sequencer sequenceur = MidiSystem.getSequencer();
            System.out.println("Nous avons un séquenceur");

        } catch(MidiUnavailableException ex) {
            System.out.println("La poisse");
        }
    } // fin de la méthode jouer()
```

L'appel de getSequencer() ne pose plus de problème, maintenant que nous l'avons enveloppé dans un bloc try/catch.

Le paramètre de catch doit être la « bonne » exception. Si nous avions écrit " catch(FileNotFoundException f) ", le code ne compilerait pas. En raison du polymorphisme, MidiUnavailableException ne « rentre » pas dans une FileNotFoundException.

N'oubliez pas qu'il ne suffit pas d'avoir un bloc catch ... Vous devez intercepter l'exception lancée!

Règles des exceptions

1 Pas de catch ni de finally sans try

```
void go() {
   Foo f = new Foo();
   f.foof();
   catch(FooException ex) { }
 }
```

ILLÉGAL! Où est le try?

2 Aucun code entre le try et le catch

```
try {
   x.faireQqch();
}
int y = 43;
} catch(Exception ex) { }
```

ILLÉGAL ! Vous ne pouvez pas placer de code entre le try et le catch.

3 try DOIT être suivi de catch ou de finally

```
try {
   x.faireQqch();
} finally {
    // nettoyage
}
```

LÉGAL parce qu'il y a un finally, même s'il n'y a pas de catch. Mais vous ne pouvez pas avoir de try tout seul.

4 try suivi de finally (sans catch) doit toujours déclarer l'exception.

```
void go() throws FooException {
    try {
       x.faireQqch();
    } finally { }
}
```

Un try sans catch ne respecte pas la loi « gérer ou déclarer ».

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Recettes de code

Vous n'êtes pas obligé de tout faire vous-même, mais c'est beaucoup plus amusant si vous le faites.

Le reste de ce chapitre est optionnel : vous pouvez utiliser notre Code prêt à l'emploi pour toutes les applis musicales.

Mais si vous voulez en savoir plus sur JavaSound, tournez la page.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Créer de vrais sons

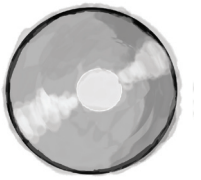
Souvenez-vous du début du chapitre : nous avons vu que les données MIDI contenaient les instructions sur ce qu'il fallait jouer (et *comment* le jouer), mais qu'elles *ne créaient pas vraiment les sons que vous entendez*. Pour qu'un son sorte des haut-parleurs, il faut envoyer les données MIDI à un équipement MIDI, qui lit les instructions et les transforme en son, en déclenchant un instrument réel ou un instrument « virtuel » (un programme synthétiseur). Nous n'utilisons dans ce livre que des périphériques logiciels. Voici donc comment JavaSound fonctionne :

Il vous faut QUATRE composants :

1. **Celui qui joue la musique, le séquenceur.**
2. **La musique à jouer..., une séquence, un morceau.**
3. **La partie de la séquence qui contient les vraies informations, une piste.**
4. **Les vraies informations : quelles notes jouer, combien de temps, etc.**

Séquenceur —joue une→ **Séquence** —a une→ **Piste** —contient→ événement Midi, événement Midi, événement Midi, événement Midi

C'est le séquenceur (un objet Sequencer) qui restitue réellement le son. Vous pouvez le voir comme un **lecteur de CD.**



La séquence (un objet Sequence) est le morceau de musique que le séquenceur va jouer. Dans ce livre, imaginez que c'est un CD, mais que **tout le CD ne contient qu'un seul morceau.**

Dans ce livre, nous n'avons besoin que d'une seule piste (un objet Track). Vous pouvez donc vous représentez un CD « un titre », avec une seule piste. C'est sur cette piste que se trouvent toutes les informations sur le morceau (les données MIDI).

Un événement MIDI est un message compréhensible par le séquenceur. Il pourrait dire (s'il parlait) : « Maintenant joue un do, joue-le à cette vitesse et tiens la note pendant tant de temps ».

Un événement MIDI pourrait également dire par exemple : « Change l'instrument courant en flûte ».

Pour ce livre, représentez-vous la séquence comme un CD ne contenant qu'un seul morceau (une seule piste). Les informations sur la façon de jouer le morceau se trouvent sur la piste, et la piste fait partie de la séquence.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Et le processus comprend CINQ étapes:

1. Obtenir un séquenceur, un objet **Sequencer**, et l'ouvrir

```
Sequencer lecteur = MidiSystem.getSequencer();
lecteur.open();
```

2. Créer un nouvel objet **Sequence**

```
Sequence seq = new Sequence(timing,4);
```

3. Demander à la Sequence de créer une piste piste de type **Track**

```
Track piste = seq.createTrack();
```

4. Remplir la piste d'événements MIDI — **MidiEvents** — et transmettre la séquence au séquenceur

```
piste.add(monMidiEvent1);
lecteur.setSequence(seq);
```

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Votre toute première application son

Tapez-la et exécutez-la. Vous entendrez le son de quelqu'un qui joue une seule note sur un piano ! (O.K., peut-être pas quelqu'un mais quelque chose.)

```
import javax.sound.midi.*;

public class MiniMiniMusicApp {

     public static void main(String[] args) {
        MiniMiniMusicApp mini = new MiniMiniMusicApp();
        mini.jouer();
     } // fin de la méthode main()

    public void jouer() {

      try {

1       Sequencer lecteur = MidiSystem.getSequencer();
        lecteur.open();

2       Sequence seq = new Sequence(Sequence.PPQ, 4);

3       Track piste = seq.createTrack();

4       ShortMessage a = new ShortMessage();
        a.setMessage(144, 1, 44, 100);
        MidiEvent noteOn = new MidiEvent(a, 1);
        piste.add(noteOn);

        ShortMessage b = new ShortMessage();
        b.setMessage(128, 1, 44, 100);
        MidiEvent noteOff = new MidiEvent(b, 16);
        piste.add(noteOff);

        lecteur.setSequence(seq);

        lecteur.start();
        Thread.sleep (60*20); // Demande de dormir pendant 20 secondes
                              // (le Thread MIDI a alors une chance de jouer)
        lecteur.close(); // Ferme le séquenceur
      } catch (Exception ex) {
          ex.printStackTrace();
      }
  } // fin de la méthode jouer()
} // fin de la classe
```

N'oubliez pas d'importer le package midi.

Obtenir un séquenceur. Et l'ouvrir pour pouvoir l'utiliser (un séquenceur n'est pas ouvert par défaut).

Ne vous inquiétez pas des arguments du constructeur. Contentez-vous de les copier (imaginez que ce sont des arguments "tout prêts").

Demander une piste à la séquence. Souvenez-vous que la piste est dans la séquence, et les données MIDI dans la piste.

Remplir la piste de quelques événements MIDI. Cette partie est presque entièrement constituée de code prêt à l'emploi. La seule difficulté réside dans les arguments de la méthode setMessage(), et dans ceux du constructeur de MidiEvent. Nous étudierons ces arguments page suivante.

Transmettre la séquence au séquenceur (comme si on insérait le CD dans le lecteur).

Démarrer le séquenceur (comme si on appuyait sur un bouton).

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Créer un MidiEvent (les données du morceau)

Un MidiEvent est une instruction correspondant à une partie d'un morceau. Une suite de MidiEvents est comparable à une partition, ou encore à un rouleau de piano mécanique. La plupart des MidiEvents qui nous concernent décrivent ***quoi faire*** et ***quand le faire***. La partie «quand» est importante, puisque le tempo est crucial en musique. Cette note suit cette note et ainsi de suite. Et comme les MidiEvents sont détaillés, vous devez indiquer à quel moment commencer à jouer les notes (un événement NOTE ON) et à quel moment *arrêter* (événement NOTE OFF). Vous imaginez donc bien qu'émettre le message «arrêter de jouer un sol» (NOTE OFF) *avant le message* «commencer à jouer un sol» (NOTE ON) ne fonctionnera pas.

En réalité, l'instruction MIDI va dans un objet Message. Le MidiEvent est une combinaison du Message et du moment auquel ce message doit se «déclencher». Autrement dit, le Message pourrait dire «Commence à jouer un do» tandis que le MidiEvent dirait «Déclenche ce message sur le quatrième temps».

Nous avons donc toujours besoin d'un Message et d'un MidiEvent. Le Message indique ce qu'il faut faire et le MidiEvent précise *quand* le faire.

Un MidiEvent dit quoi faire et quand le faire.

Chaque instruction doit comporter une indication de tempo.

Autrement dit, sur quel temps l'événement doit se produire.

1. Créer un **Message**

```
ShortMessage a = new ShortMessage();
```

2. Placer l'**Instruction** dans le Message

```
a.setMessage(144, 1, 44, 100);
```

Ce message dit « Commence à jouer la note 44 ». (Nous verrons les nombres page suivante.)

3. Créer un **MidiEvent** en utilisant le Message

```
MidiEvent noteOn = new MidiEvent(a, 1);
```

Les instructions sont dans le message, mais le MidiEvent ajoute le moment où l'instruction doit être exécutée. Ce MidiEvent dit de déclencher le message "a" sur le premier temps (1).

4. Ajouter le MidiEvent à l'objet **Track**

```
piste.add(noteOn);
```

La piste, l'objet Track, contient tous les objets MidiEvent. La séquence les organise dans l'ordre dans lequel ils sont censés se produire, puis le séquenceur les restitue dans cet ordre. Des quantités d'événements peuvent se produire au même moment, par exemple deux notes jouées simultanément, ou différents instruments jouant différents sons en même temps.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Message MIDI : le cœur d'un MidiEvent

Un message MIDI contient la partie de l'événement qui indique quoi faire. L'instruction réelle que vous voulez que le séquenceur exécute. Le premier argument d'une instruction est toujours le type du message. Les valeurs des trois autres arguments dépendent du type de message. Par exemple, un message de type 144 signifie «NOTE ON». Mais pour pouvoir exécuter NOTE ON, le séquenceur doit disposer d'un certain nombre d'informations. Imaginez le séquenceur disant «O.K., mais sur quel canal? Voulez-vous que je joue une note de Batterie ou une note de Piano? Un do? Un mi bémol? Et pendant que nous y sommes, combien de temps dois-je la tenir?».

Pour obtenir un message MIDI, créez une instance de ShortMessage et appelez la méthode setMessage() en lui transmettant les quatre arguments nécessaires. Mais n'oubliez pas que le message n'indique que ce qu'il faut faire, et que vous devez toujours le placer dans un événement qui indiquera quand ce message doit se «déclencher».

Anatomie d'un message

Le *premier* argument de setMessage() représente toujours le 'type' du message, tandis que les trois *autres* représentent différentes choses en fonction du type de message.

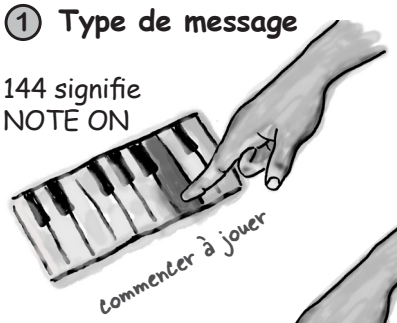
```
                  type de message   canal   note   vélocité
a.setMessage(144,                   1,      44,    100);
```

Les 3 arguments varient selon le type de message. Celui-ci est un message NOTE ON, et les autres arguments indiquent ce que le séquenceur doit savoir pour jouer une note.

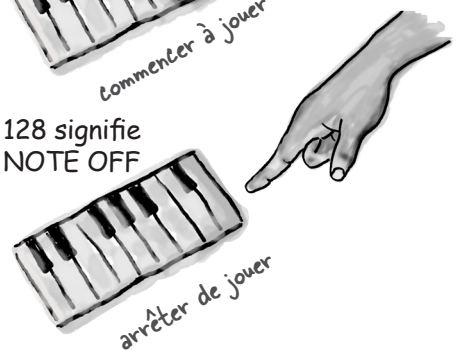
Le *Message* dit quoi faire et le *MidiEvent* dit quand.

① Type de message

144 signifie NOTE ON



128 signifie NOTE OFF



② Canal

Représentez-vous un canal comme un musicien dans un groupe. Le canal 1 est le musicien 1 (le pianiste), le canal 9 le batteur, etc.

③ Note

Un nombre compris entre 0 et 127, de la note la plus basse à la plus haute.

④ Vélocité

À quelle rapidité et avec quelle force avez-vous appuyé sur le touche? 0 est si doux que vous n'entendrez probablement rien, mais 100 est une bonne valeur par défaut.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Modifier un message

Maintenant que vous savez ce que contient un message Midi, Vous pouvez faire vos propres expériences. Modifiez la note jouée, sa tenue, ajoutez des notes ou changez d'instrument.

1 Modifier la note

Essayez un nombre entre 0 et 127 dans les messages note on et note off.

```
a.setMessage(144, 1, 20, 100);
```

2 Modifier la durée de la note

Modifiez l'événement note off (pas le *message*) pour qu'il se produise plutôt ou plus tard.

```
b.setMessage(128, 1, 44, 100);
MidiEvent noteOff = new MidiEvent(b, 3);
```

3 Changez d'instrument

Ajoutez un nouveau message, AVANT celui qui joue la note, pour que l'instrument du canal 1 soit autre chose que le piano par défaut. L'argument qui indique de changer d'instrument est "192", et le troisième argument représente l'instrument (essayez un nombre entre 0 et 127)

```
first.setMessage(192, 1, 102, 0);
```

transformer l'instrument du canal 1 (musicien 1) en instrument 102

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Version 2: utiliser la ligne de commande pour essayer des sons

Cette version ne joue toujours qu'une seule note, mais vous allez utiliser les arguments de la ligne de commande pour changer d'instrument et de note. Testez en transmettant deux valeurs de type int entre 0 et 127. Le premier int définit l'instrument, le second int la note.

```
import javax.sound.midi.*;

public class MiniMusicCmdLine {   // c'est la première

   public static void main(String[] args) {
        MiniMiniMusicCmdLine mini = new MiniMusicCmdLine();
        if (args.length < 2) {
            System.out.println("Vous avez oublié les arguments");
        } else {
            int instrument = Integer.parseInt(args[0]);
            int note = Integer.parseInt(args[1]);
            mini.play(instrument, note);
        }
   } // fin de la méthode main()
   public void play(int instrument, int note) {

      try {

         Sequencer player = MidiSystem.getSequencer();
         player.open();
         Sequence seq = new Sequence(Sequence.PPQ, 4);
         Track track = seq.createTrack();

         MidiEvent event = null;

         ShortMessage first = new ShortMessage();
         first.setMessage(192, 1, instrument, 0);
         MidiEvent changeInstrument = new MidiEvent(first, 1);
         track.add(changeInstrument);

         ShortMessage a = new ShortMessage();
         a.setMessage(144, 1, note, 100);
         MidiEvent noteOn = new MidiEvent(a, 1);
         track.add(noteOn);

         ShortMessage b = new ShortMessage();
         b.setMessage(128, 1, note, 100);
         MidiEvent noteOff = new MidiEvent(b, 16);
         track.add(noteOff);
         player.setSequence(seq);
         player.start();

      } catch (Exception ex) {ex.printStackTrace();}
   } // fin de la méthode play()
} // fin de la classe
```

Exécutez le programme avec deux arguments de 0 à 127. Essayez ceux-ci pour commencer:

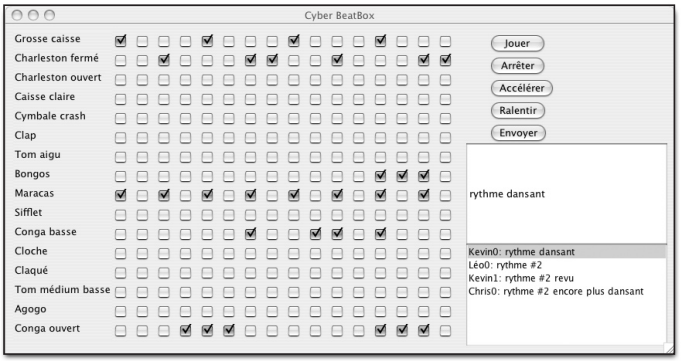
```
Fichier Edition Fenêtre Aide Jouer
%java MiniMusicCmdLine 102 30
%java MiniMusicCmdLine 80 20
%java MiniMusicCmdLine 40 70
```


Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Suite du programme des Recettes de code

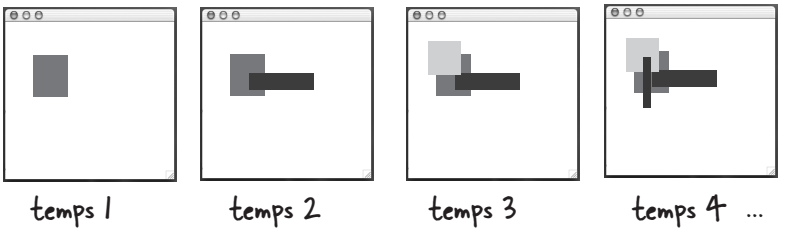
Chapitre 15: le but final

Quand nous aurons terminé, nous aurons une boîte à rythmes pleinement fonctionnelle en réseau. Nous devrons étudier les IHM (notamment la gestion des événements), les E/S, les connexions réseau et les threads. C'est ce que nous ferons dans les trois prochains chapitres (12, 13 et 14).



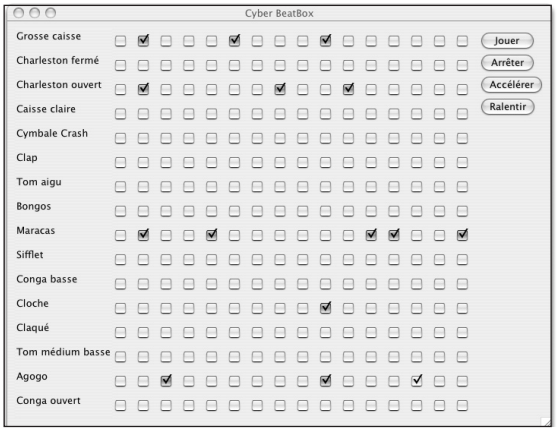
Chapitre 12: événements MIDI

Cette recette de code va nous permettre de construire un petit "clip" qui trace des rectangles aléatoires au rythme de la musique MIDI. Nous apprendrons à construire et restituer de nombreux événements MIDI (et pas seulement deux comme dans le présent chapitre).



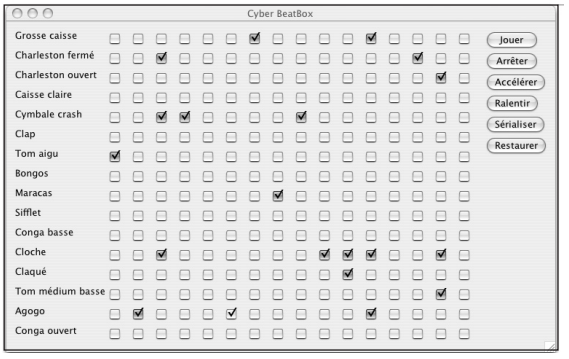
Chapitre 13: BeatBox autonome

Maintenant nous construisons la vraie BeatBox, interface graphique et tout. Mais elle est limitée: dès que vous modifiez un motif, le précédent est perdu. Il n'y a aucune fonction de sauvegarde et de restauration, et elle ne communique pas avec le réseau. (Mais vous pouvez toujours l'utiliser pour vous entraîner.)



Chapitre 14: sauvegarde et restauration

Vous avez créé le modèle parfait. Vous pouvez maintenant l'enregistrer dans un fichier et le recharger quand vous voulez le rejouer. Nous sommes prêts pour la version finale (chapitre 15), dans laquelle nous n'écrivons pas modèle dans un fichier, mais où nous l'envoyons au serveur *via* le réseau.



Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Dans ce chapitre, nous avons exploré le merveilleux monde des exceptions. Vous devez maintenant déterminer si les affirmations suivantes sont vraies ou fausses.

Vrai ou Faux

1. Un bloc try doit être suivi d'un bloc catch *et d'un bloc* finally.

2. Si on écrit une méthode susceptible de déclencher une exception vérifiée par le compilateur, on doit envelopper le code dans un bloc try / catch.

3. Les blocs catch peuvent être polymorphes.

4. Seules les exceptions « vérifiées par le compilateur » peuvent être interceptées.

5. Si on définit un bloc try / catch, le bloc finally correspondant est optionnel.

6. Si on définit un bloc try, on peut l'apparier avec un bloc catch ou un bloc finally, ou les deux.

7. Si on écrit une méthode qui déclare qu'elle peut lancer une exception vérifiée par le compilateur, on doit envelopper le code qui lance l'exception dans un bloc try / catch.

8. La méthode main( ) doit gérer toutes les exceptions non gérées qui lui parviennent.

9. Un seul bloc try peut avoir plusieurs blocs catch différents.

10. Une méthode ne peut lancer qu'un seul type d'exception.

11. Un bloc finally peut s'exécuter sans qu'une exception soit lancée.

12. Un bloc finally peut exister sans bloc try.

13. Un bloc try peut exister seul, sans catch ni finally.

14. Lorsqu'on gère une exception, on dit parfois qu'on « l'esquive ».

15. L'ordre des blocs catch n'a jamais d'importance.

16. Une méthode ayant un bloc try et un bloc finally peut éventuellement déclarer l'exception.

17. Les exceptions susceptibles de survenir à l'exécution doivent être gérées ou déclarées.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Le frigo

Un programme Java a été découpé en fragments ; ceux-ci ont été collés 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 !

```
System.out.print("n");
```

```
try {
```

```
System.out.print("l");
```

```
prendreRisque(test);
```

```
System.out.println("r");
```

```
} finally {
```

```
System.out.print("c");
```

```
class MyEx extends Exception { }

public class ExTestDrive {
```

```
System.out.print("e");
```

```
if ("non".equals(t)) {
```

```
System.out.print("v");
```

```
throw new MyEx();
```

```
} catch (MyEx e) {
```

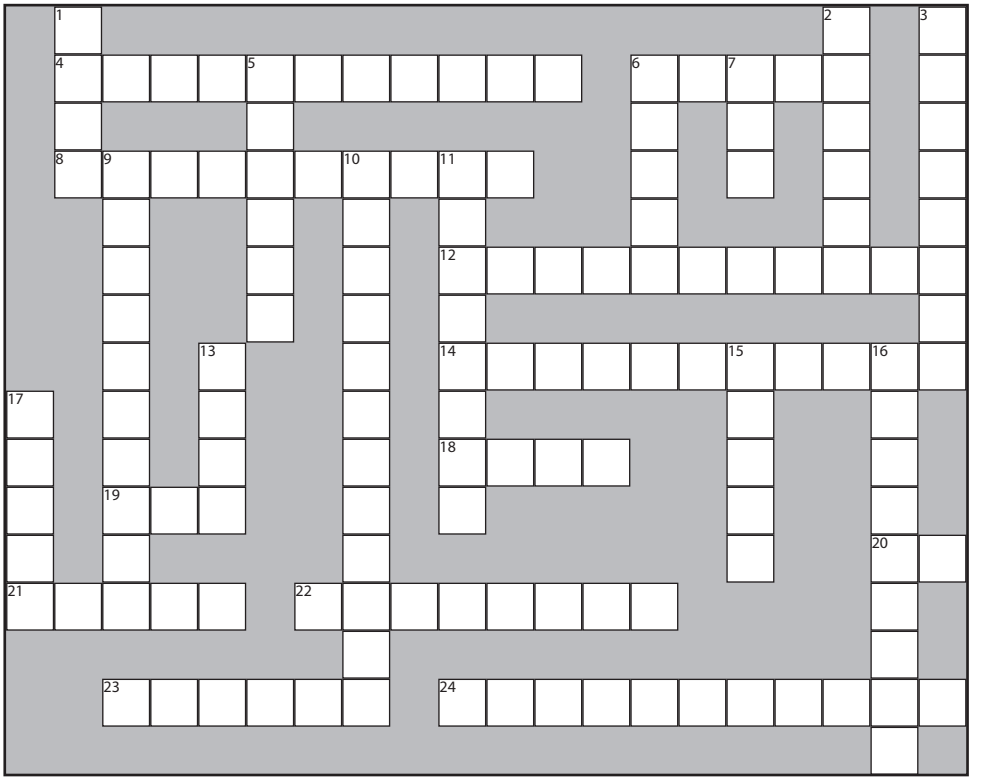
```
static void prendreRisque(String t) throws MyEx {
  System.out.print("a");
```

```
public static void main(String [] args) {
  String test = args[0];
```

```
Fichier Edition Fenêtre Aide Lancer
% java ExTestDrive non
laver

% java ExTestDrive oui
lancer
```


Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Mots-Croisés 7.0



Vous savez quoi faire !

Horizontalement

4. Donne une valeur
6. Intercepte
8. Arbre généalogique
12. On en hérite
14. Recette de code
18. Programme
19. Le plus petit primitif
20. Constante
21. Lance
22. Non abstraite
23. Instances de classes
24. Gestion ou… telle est la loi

Verticalement

1. A beaucoup de méthodes statiques
2. Déclare une exception
3. Ôtées de la pile
5. Patron d'objet
6. Programmer
7. Avant le catch
9. Faire du nouveau
10. Transformerions en bytecode
11. Objet
13. Pas un comportement
15. Posent des conditions
16. Erreur ou problème
17. À des décimales

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Solutions des exercices

Vrai ou Faux

1. Faux. L'un ou l'autre, ou les deux.

2. Faux. Vous pouvez déclarer l'exception.

3. Vrai.

4. Faux. Les exceptions se produisant à l'exécution peuvent être interceptées.

5. Vrai.

6. Vrai, les deux sont acceptables.

7. Faux, la déclaration suffit.

8. Faux. Mais si elle ne le fait pas, la JVM peut s'arrêter.

9. Vrai.

10. Faux.

11. Vrai. C'est ce qu'on fait souvent pour « faire le ménage » après une tâche partiellement terminée.

12. Faux.

13. Faux.

14. Faux, esquiver est synonyme de déclarer.

15. Faux. Les exceptions les plus générales doivent être interceptées par les derniers blocs catch.

16. Faux. En l'absence de bloc catch, vous *devez* déclarer l'exception.

17. Faux.

Le frigo :

```
class MyEx extends Exception { }

public class ExTestDrive {

  public static void main(String [] args) {
    String test = args[0];
    try {

       System.out.print("l");

       prendreRisque(test);

       System.out.print("c");

    } catch ( MyEx e) {

        System.out.print("v");

    } finally {

        System.out.print("e");
    }
    System.out.println("r");
  }

  static void prendreRisque(String t) throws MyEx {
    System.out.print("a");

    if ("non".equals(t)) {

      throw new MyEx();
    }

    System.out.print("n");

  }
}
```

Fichier Edition Fenêtre Aide Laver

```
% java ExTestDrive non
laver

% java ExTestDrive oui
lancer
```


Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Solution des mots-croisés

	1 M																		2 T				3 D	
	4 A	F	F	E	5 C	T	A	T	I	O	N		6 C	A	7 T	C	H		E					
	T				L								O		R		R		P					
	8 H	9 I	E	R	A	R	10 C	H	11 I	E			D		Y		O		I					
		N			S		O		N				E				W		L					
		S			S		M		12 S	U	P	E	R	C	L	A	S	S	E					
		T			E		P		T										E					
		A		13 E			I		14 A	L	G	O	R	I	15 T	H	M	16 E	S					
17 F		N		T			L		N						E			X						
L		C		A			E		18 C	O	D	E			S			C						
O		19 I	N	T			R		E						T			E						
A		E					I								S			20 P	I					
21 T	H	R	O	W		22 C	O	N	C	R	E	T	E					T						
							N											I						
		23 O	B	J	E	T	S		24 D	E	C	L	A	R	A	T	I	O	N					
																		N						

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Une histoire très graphique

Soyons honnêtes, vous allez devoir créer des interfaces graphiques (GUI). On parle aussi d'interfaces homme machine (IHM). Si vous construisez des applications destinées à des utilisateurs, vous en aurez *besoin*. Si vous programmez pour vous-même, vous en aurez *envie*. Même si vous pensez que vous allez passer votre vie à écrire du code côté serveur, pour lequel l'interface des clients est une page web, vous devrez tôt ou tard écrire des outils, et il vous faudra aussi une interface graphique. Oui, les applications en mode ligne de commande sont rétro, mais elles n'en ont pas le charme. Elles manquent de souplesse et de convivialité. Nous allons consacrer deux chapitres aux interfaces graphiques et découvrir des fonctionnalités capitales de Java, notamment la **gestion des événements** et les **classes internes**. Dans ce chapitre, nous allons afficher un bouton à l'écran et le rendre « cliquable ». Nous dessinerons à l'écran, nous afficherons une image JPEG et nous aborderons l'animation.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Tout commence par une fenêtre

Un JFrame est l'objet qui représente une fenêtre à l'écran. C'est dans cette fenêtre que vous placez tous les composants de l'interface : boutons, cases à cocher, champs de texte, etc. Elle peut posséder une honnête barre de menus avec les éléments habituels. Et elle dispose de toutes les icônes (les widgets) correspondant à votre plate-forme pour agrandir, réduire et fermer la fenêtre.

L'aspect d'un JFrame varie en fonction de votre plate-forme. Voici un JFrame sous Windows :

Si je vois encore une application en mode ligne de commande, vous êtes viré.

Un JFrame avec une barre de menus et deux « widgets » (un bouton et un bouton radio).

Placer des widgets dans la fenêtre

Une fois que vous avez un JFrame, vous pouvez y placer des widgets. Il existe une tonne de composants Swing que vous pouvez ajouter : vous les trouverez dans le package javax.swing. Les plus courants sont JButton, JRadioButton, JCheckBox, JLabel, JList, JScrollPane, JSlider, JTextArea, JTextField et JTable. La plupart sont très simples à utiliser, mais certains (comme JTable) peuvent être un peu plus complexes.

Une interface graphique est facile à construire :

1. Créer un cadre (un JFrame)
```
JFrame cadre = new JFrame();
```

2. Créer un widget (bouton, champ de texte, etc.)
```
JButton bouton = new JButton("cliquez-moi");
```

3. Ajouter le widget au cadre
```
cadre.getContentPane().add(bouton);
```

On n'ajoute rien au cadre directement. Imaginez que le cadre est une bordure qui entoure la fenêtre et que vous ajoutez les widgets au panneau de cette fenêtre.

4. L'afficher (fixer sa taille et le rendre visible)
```
cadre.setSize(300,300);
cadre.setVisible(true);
```


Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Votre première interface graphique: un bouton dans un cadre

```
import javax.swing.*;

public class SimpleIHM1 {
    public static void main (String[] args) {

        JFrame cadre = new JFrame();
        JButton bouton = new JButton("cliquez-moi");

        cadre.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        cadre.getContentPane().add(bouton);

        cadre.setSize(300,300);

        cadre.setVisible(true);
    }
}
```

Ne pas oublier d'importer le package swing.

Créer un cadre et un bouton.

(Vous pouvez passer au constructeur le texte que vous voulez voir sur le bouton.)

Cette ligne termine le programme dès que vous fermez la fenêtre. Si vous l'oubliez, elle restera indéfiniment affichée à l'écran.)

Ajouter le bouton au panneau.

Fixer la taille du cadre en pixels.

Enfin, le rendre visible!! (Si vous oubliez cette étape, vous ne verrez rien quand vous exécuterez ce code.)

Exécutons le programme et voyons ce qui se passe:

```
%java SimpleIHM1
```

Ouah! Ce bouton est gigantesque.

Le bouton remplit tout l'espace disponible dans le cadre. Plus tard, nous apprendrons à contrôler son emplacement (et sa taille).

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

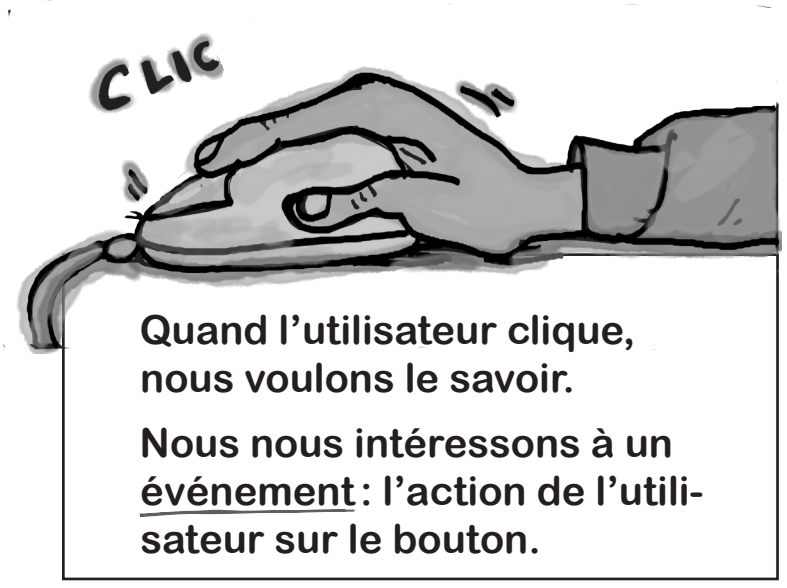
Mais rien ne se passe quand je clique...

Ce n'est pas la stricte vérité. Quand vous cliquez sur le bouton, son aspect change (il varie en fonction du « look and feel » de votre plate-forme, mais vous verrez toujours s'il a été cliqué).

La vraie question est : « Comment amener le bouton à faire quelque chose de précis quand l'utilisateur clique dessus ? ».

Il nous faut deux éléments :

1. Une ***méthode*** à appeler quand l'utilisateur clique sur le bouton (le résultat que vous voulez que le clic génère).
2. Un moyen de ***savoir*** quand déclencher cette méthode. Autrement dit de savoir quand l'utilisateur clique sur le bouton !



il n'y a pas de Questions stupides

Q : Est-ce qu'un bouton ressemble à un bouton Windows si je suis sous Windows ?

R : Si vous le voulez. Vous avez le choix entre plusieurs types de « look and feel » — les classes qui contrôlent l'aspect de l'interface. Dans la plupart des cas, vous avez au moins deux options : l'aspect Java standard, alias ***Metal***, et l'aspect natif de votre plate-forme. Les captures d'écran de ce livre utilisent soit le look and feel ***Aqua*** d'OS X, soit le look ***Metal***.

Q : Est-ce que je peux donner le look Aqua à une interface en permanence ? Même si le programme s'exécute sous Windows ?

R : Non. Tous les choix ne sont pas disponibles sur toutes les plates-formes. Pour plus de sécurité, vous pouvez définir explicitement le look Metal, pour que l'interface ait toujours le même aspect indépendamment de la machine, ou ne rien spécifier et accepter la valeur par défaut.

Q : J'ai entendu dire que Swing était lente comme une limace et que personne ne l'utilisait.

R : C'était vrai dans le passé, mais ça ne l'est plus. Vous risquez de souffrir un peu si vous manquez de ressources. Mais, sur les machines récentes et à partir de Java 1.3, vous ne remarquerez même pas la différence entre une interface Swing et une interface native. De nos jours, Swing est employée intensivement dans toutes sortes d'applications.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Gérer un événement utilisateur

Imaginez que le texte du bouton soit «cliquez moi» et que vous vouliez le transformer en «*j'ai été cliqué!*» quand l'utilisateur appuie. Nous pouvons commencer par écrire une méthode qui modifie ce texte, (un bref coup d'œil sur l'API vous expliquera comment):

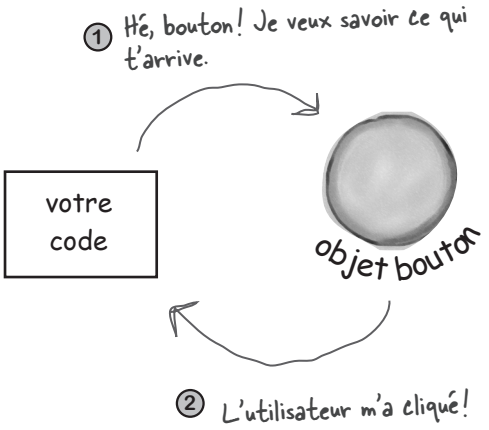
```
public void modifier() {
     bouton.setText("J'ai été cliqué!");
}
```

Et maintenant? Comment saurons-nous quand cette méthode doit s'exécuter? ***Et comment saurons-nous que le bouton a été cliqué?***

En Java, le processus qui consiste à lire un événement utilisateur et à le traiter se nomme *gestion des événements*. Il existe de nombreux types d'événements différents, bien que la plupart soient associés à une action de l'utilisateur de l'IHM. Si celui-ci clique sur un bouton, c'est un événement. Et cet événement dit «L'utilisateur veut que l'action associée à ce bouton se produise». Si c'est un bouton «Tempo lent», l'utilisateur veut que le tempo ralentisse. Si c'est un bouton Envoi sur un client de «chat», il veut que l'envoi de son message soit déclenché. L'événement le plus simple est donc le clic de l'utilisateur, qui indique que celui-ci veut qu'une action ait lieu.

Avec les boutons, vous ne vous souciez généralement pas des événements intermédiaires (on appuie sur le bouton, on le relâche, etc). Ce que vous voulez dire au bouton, c'est: «Peu m'importe ce que l'utilisateur fabrique avec le bouton, pendant combien de temps il le survole avec la souris, combien de fois il va changer d'avis avant de se décider, etc. ***Dis-moi juste quand il veut vraiment quelque chose!*** Autrement dit, ne m'appelle pas tant que l'utilisateur n'a pas cliqué d'une façon qui indique qu'il veut que le fichu bouton fasse ce qu'il dit qu'il va faire!».

D'abord, le bouton doit savoir que nous voulons savoir.



Ensuite, le bouton doit avoir un moyen de nous rappeler quand l'événement se produit.

MUSCLEZ VOS NEURONES

1) Comment pouvez-vous dire à un objet bouton que ses événements vous intéressent? Que vous êtes un auditeur concerné?

2) Comment le bouton va-t-il vous rappeler? Supposez que vous n'ayez aucun moyen de lui communiquer le nom de votre unique méthode (modifier()). Que pouvez-vous utiliser d'autre pour l'assurer que vous avez une méthode spécifique qu'il peut appeler quand l'événement survient?

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Si les événements des boutons vous intéressent, **implémentez une interface** qui dise : ***j'écoute*** vos événements.

Une **interface auditeur** est le pont entre l'**auditeur** (vous) et la **source de l'événement** (le bouton).

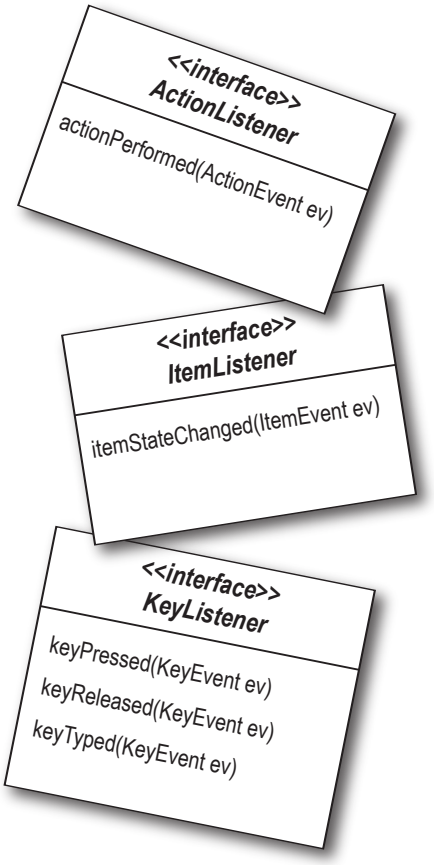
Les composants Swing sont des sources d'événements. Dans la terminologie Java, une source d'événement est un objet qui transforme les actions de l'utilisateur (cliquer un bouton, appuyer sur une touche, fermer une fenêtre) en événements. Et, comme virtuellement tout en Java, un événement est représenté par un objet. Un objet d'une classe correspondant à l'événement. Si vous parcourez la documentation du package java.awt.event, vous verrez un paquet de classes d'événements (faciles à repérer, leurs noms finissent tous par ***Event***). Vous y trouverez MouseEvent, KeyEvent, WindowEvent, ActionEvent et plusieurs autres.

Une ***source d'événement*** (comme un bouton) crée un ***objet événement*** quand l'utilisateur fait quelque chose de significatif (*cliquer* dessus par exemple). La majeure partie du code que vous écrirez (et tout le code de ce livre) recevra des événements au lieu d'en *créer*. Autrement dit, vous serez la plupart du temps un *auditeur* d'événements et non une *source*.

Chaque type d'événement a une interface auditeur qui lui correspond. Si vous voulez un MouseEvent, implémentez l'interface MouseListener. Un WindowEvent ? Implémentez WindowListener. Vous voyez l'idée. Et rappelez-vous les règles des interfaces — pour implémenter une interface, vous *déclarez* que vous l'implémentez (class Chien implements Compagnon), ce qui signifie que vous devez *écrire l'implémentation* de chaque méthode de l'interface.

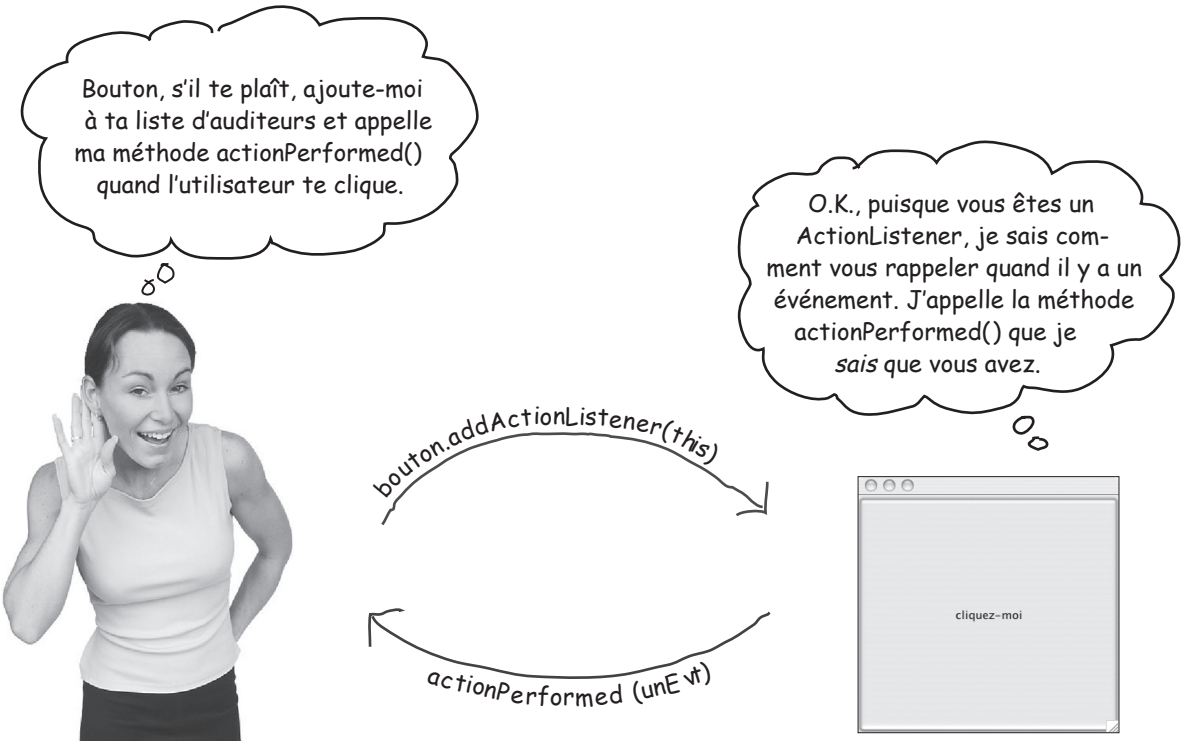
Certaines interfaces ont plusieurs méthodes parce que les événements eux-mêmes ont différentes variantes. Si vous implémentez MouseListener, vous pouvez avoir des événements pour mousePressed, mouseReleased, mouseMoved, etc. Chacun de ces événements de la souris a une méthode séparée dans l'interface, même s'il s'agit toujours d'un MouseEvent. La méthode mousePressed() est appelée quand l'utilisateur (vous l'avez deviné) appuie sur le bouton de la souris. Et quand l'utilisateur le lâche, c'est la méthode mouseReleased() qui est appelée. Il n'y a donc qu'un seul *objet*, MouseEvent, pour les différents événements de la souris, mais il y a plusieurs méthodes qui représentent les différents *types* d'événements.

Quand vous implémentez une interface auditeur, vous fournissez au bouton le moyen de vous rappeler. C'est dans l'interface que la méthode de rappel est déclarée.



Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Comment l'utilisateur et la source communiquent :



L'auditeur

Si votre classe doit être informée des ActionEvents d'un bouton, vous implémentez l'interface ActionListener. Comme le bouton doit savoir que vous êtes intéressé, vous vous enregistrez auprès du bouton en appelant sa méthode addActionListener() et en lui transmettant une référence à l'ActionListener (dans ce cas, *vous* êtes l'ActionListener et vous passez donc *this*). Et comme le bouton doit avoir un moyen de vous rappeler quand l'événement se produit, il appelle la méthode de l'interface auditeur. En tant qu'ActionListener, vous *devez* implémenter la seule méthode de l'interface, actionPerformed(). Le compilateur le garantit.

La source d'événement

Un bouton étant une source d'ActionEvents, il doit savoir quels objets sont des auditeurs intéressés. Le bouton a une méthode addActionListener() qui fournit aux auditeurs un moyen de *dire* au bouton qu'ils sont intéressés.

Quand la méthode addActionListener() du bouton s'exécute (parce qu'un auditeur potentiel l'a appelée), le bouton lit le paramètre (une référence à l'objet auditeur) et le mémorise dans une liste. Quand l'utilisateur clique, le bouton déclenche l'événement en appelant actionPerformed() sur chaque auditeur de la liste.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Lire l'ActionEvent d'un bouton

1. Implémenter l'interface ActionListener.
2. S'enregistrer auprès du bouton (lui dire qu'on veut écouter ses événements).
3. Définir la méthode de gestion de l'événement (implémenter la méthode actionPerformed() de l'interface ActionListener).

```
import javax.swing.*;
import java.awt.event.*;

                                        1
public class SimpleIhm1B implements ActionListener {
    JButton bouton;

    public static void main (String[] args) {
       SimpleIhm1B ihm = new SimpleIhm1B();
       ihm.go();
     }

    public void go() {
       JFrame cadre = new JFrame();
       bouton = new JButton("cliquez moi");

2      bouton.addActionListener(this);

       cadre.getContentPane().add(bouton);
       cadre.SetDefaultCloseOperation (JFrame.EXIT_ON_CLOSE);
       cadre.setSize(300,300);
       cadre.setVisible(true);
    }

3   public void actionPerformed(ActionEvent event) {
       bouton.setText("J'ai été cliqué!");
    }
}
```

Importer le package qui contient ActionListener et ActionEvent.

Implémenter l'interface. Cette instruction dit "Une instance de SimpleIhm1B EST-UN ActionListener".

(Le bouton enverra des événements uniquement aux implémenteurs de l'ActionListener).

Informer le bouton de votre intérêt. Cette instruction lui dit "Ajoute-moi à ta liste d'auditeurs". L'argument transmis DOIT être un objet d'une classe qui implémente ActionListener!!

Implémenter la méthode actionPerformed() de l'interface ActionListener. C'est cette méthode qui gère réellement l'événement!

Le bouton appelle cette méthode pour vous informer qu'un événement s'est produit. Il vous envoie un objet ActionEvent comme argument, mais nous n'en avons pas besoin. Il nous suffit de savoir que l'événement est arrivé.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Auditeurs, sources et événements

Durant la majeure partie de votre éblouissante carrière en Java, vous ne serez pas la *source* des événements.

(Peu importe que vous fantasmiez en croyant être le centre de votre univers social.)

Vous devez vous y faire. ***Votre tâche consiste à bien écouter.***

(Ce qui, si vous le faites sincèrement, peut améliorer vos rapports sociaux.)

En tant qu'auditeur, mon rôle consiste à **implémenter** l'interface, **m'enregistrer** auprès du bouton et **fournir** la gestion de l'événement.

L'auditeur REÇOIT l'événement

En tant que source, mon rôle consiste à **accepter** les enregistrements (des auditeurs), **recevoir** les événements de l'utilisateur et **appeler** la méthode de gestion de l'événement de l'auditeur (quand l'utilisateur me clique).

La source ENVOIE l'événement

Hé! Et moi? J'ai un rôle aussi, vous savez! En tant qu'objet Event, je suis l'**argument** de la méthode de rappel de l'événement (de l'interface) et ma tâche consiste à **transporter les données** concernant l'événement à l'auditeur.

L'objet Event CONTIENT LES DONNÉES sur l'événement

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

il n'y a pas de Questions stupides

Q : Pourquoi est-ce que je ne peux pas être une source d'événements?

R : Vous POUVEZ l'être. Nous avons simplement dit que, *la plupart* du temps, vous seriez le récepteur et non l'origine de l'événement (au moins au *début* de votre brillante carrière de programmeur Java). Or la plupart des événements qui vous intéressent sont « déclenchés » par les classes de l'API Java, et votre rôle se borne à les écouter. Mais vous pourriez écrire un programme qui nécessite un événement personnalisé, par exemple un événement BourseEvent déclenché quand votre application boursière détecte quelque chose qu'elle juge important. Dans ce cas, vous auriez un objet VeilleBoursiere qui serait une source d'événement, et vous feriez tout ce que fait un bouton (ou toute autre source) — créer une interface auditeur, fournir une méthode d'enregistrement (addBourseListener()), qui, quand on l'appellerait, ajouterait l'appelant à la liste des auditeurs. Puis, quand un événement boursier se produirait, vous instancieriez un objet BourseEvent (une autre classe que vous écririez) et l'enverriez aux auditeurs de la liste en appelant leur méthode coursChange(BourseEvent ev). Et n'oubliez pas que tout *type d'événement* doit avoir une *interface auditeur correspondante* (et que vous devez créer une interface BourseListener avec une méthode coursChange()).

Q : Je ne vois pas l'importance de l'objet event qui est transmis aux méthodes de rappel. Si quelqu'un appelle ma méthode mousePressed, de quelle autre info pourrais-je avoir besoin ?

R : Dans la plupart des cas, vous n'avez pas besoin de l'objet event. Ce n'est rien de plus qu'un petit « transporteur de données » qui transmet des informations supplémentaires sur l'événement. Mais vous devez parfois interroger l'événement pour obtenir des détails spécifiques. Si par exemple votre méthode mousePressed() est appelée, vous savez que le bouton de la souris a été cliqué. Mais si vous voulez savoir sur quoi pointait la souris lors du clic ? Autrement dit si vous voulez connaître les coordonnées X et Y de l'écran à cet instant ?

Ou bien vous voulez enregistrer le *même* auditeur auprès de *plusieurs* objets. Une calculette, par exemple, a 10 touches numériques. Comme elles font toutes la même chose, vous ne voulez pas créer un auditeur différent pour chaque touche. Vous pouvez enregistrer un seul auditeur auprès des 10 touches. Quand vous recevez un événement (parce que la méthode de rappel a été appelée) vous pouvez appeler une méthode sur l'objet event pour connaître la source réelle, autrement dit *quelle touche a envoyé cet événement.*

À vos crayons

Chacun de ces widgets (objets d'interface utilisateur) est la source d'un ou plusieurs événements. Reliez les widgets aux événements qu'ils peuvent générer. Certains widgets peuvent être source de plusieurs événements, et certains événements peuvent être générés par plusieurs widgets

Widgets	Méthodes
case à cocher	windowClosing()
champ de texte	actionPerformed()
liste déroulante	itemStateChanged()
bouton	mousePressed()
boîte de dialogue	keyTyped()
bouton radio	mouseExited()
élément de menu	focusGained()

Comment SAVOIR si un objet est une source d'événement?

On cherche dans l'API.

O.K. Et on cherche quoi?

Une méthode qui commence par « add », se termine par « Listener » et prend pour argument une interface auditeur. Si vous voyez :

addKeyListener(KeyListener k)

vous savez que la classe qui contient cette méthode est une source de KeyEvents. C'est une convention de nommage.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Revenons aux objets graphiques...

Maintenant que nous savons mieux comment les événements fonctionnent (nous en reparlerons plus tard), revenons à l'affichage d'objets à l'écran. Nous allons consacrer quelques minutes à jouer avec le graphisme avant de retourner à la gestion des événements.

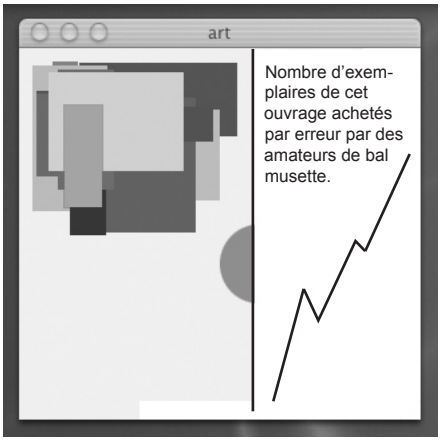
Trois façons d'ajouter des objets à votre IHM :

① **Placer des widgets dans un cadre**

Ajouter des boutons, des menus, des boutons radio, etc.

```
cadre.getContentPane().add(monBouton);
```

Le package javax.swing contient une bonne douzaine de types de widgets.



② **Dessiner un graphisme 2D sur un widget**

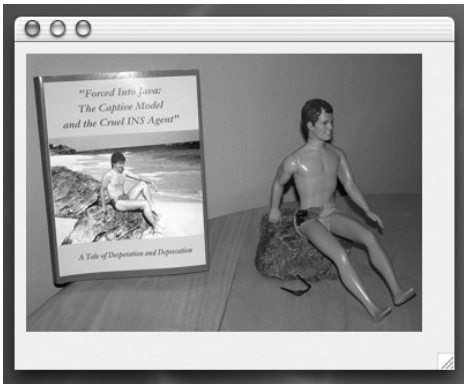
Utiliser un objet graphics pour «peindre» des formes.

```
graphics.fillOval(70,70,100,100);
```

Vous n'êtes pas limité aux carrés et aux cercles. L'API Java2D est bourrée de méthodes graphiques élaborées et amusantes.

art, jeux, simulations, etc.

courbes, histogrammes, camemberts, etc.



③ **Placer un JPEG sur un widget**

Vous pouvez placer vos propres images sur un widget.

```
graphics.drawImage(monImage,10,10,this);
```

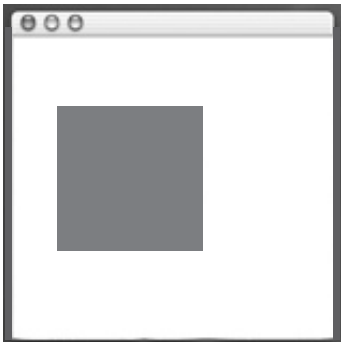

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Créez votre propre widget de dessin

Si vous voulez afficher vos propres graphismes à l'écran, la meilleure solution consiste à créer votre propre widget de dessin. Vous le posez sur le cadre, exactement comme un bouton ou comme n'importe quel widget, mais il servira de support à vos images quand vous l'afficherez. Vous pourrez même animer ces images, ou faire changer les couleurs chaque fois que vous cliquerez sur un bouton.

C'est vraiment du gâteau.

Créez une sous-classe de JPanel et redéfinissez une seule méthode, paintComponent().



Tout votre code graphique sera placé dans la méthode paintComponent(). Pensez que cette méthode paintComponent() est la méthode appelée par le système pour dire « Hé, widget ! C'est le moment de te peindre toi-même.» Si vous voulez tracer un cercle, la méthode paintComponent() aura du code pour tracer un cercle. Quand le cadre contenant votre widget s'affiche, paintComponent() est appelée et votre cercle apparaît. Si l'utilisateur réduit la fenêtre en icône, la JVM sait que le cadre doit être « réparé » quand il est agrandi de nouveau, et elle rappelle paintComponent(). Chaque fois que la JVM pensera que l'écran a besoin d'être rafraîchi, votre méthode paintComponent() sera appelée.

Autre chose. ***Vous n'appelez jamais cette méthode vous-même !*** Son argument (un objet Graphics) est le canevas que vous avez appliqué sur l'affichage. Vous ne pouvez pas l'obtenir vous-même, il doit vous être fourni par le système. Mais nous verrons plus tard que vous pouvez *demander* au système de rafraîchir l'affichage (repaint()), ce qui provoque en fin de compte l'appel de paintComponent().

```
import java.awt.*;
import javax.swing.*;

class MonPanneau extends JPanel {

      public void paintComponent(Graphics g) {

         g.setColor(Color.orange);

         g.fillRect(20,50,100,100);
      }
}
```

Vous avez besoin de ces deux packages.

Créez une sous-classe de JPanel, un widget que vous pouvez ajouter à un cadre comme n'importe quoi d'autre. Sauf que maintenant c'est votre propre widget personnalisé.

Voilà LA Méthode Importante. Vous ne l'appellerez JAMAIS vous-même. Le système l'appelle et dit « Voici une belle surface toute neuve, de type Graphics, et maintenant tu peux peindre dessus ».

Imaginez que "g" est une machine à peindre. Vous lui indiquez la couleur à prendre et la forme à peindre (avec des coordonnées pour son emplacement et sa taille).

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Jouer avec paintComponent()

Vous pouvez faire bien d'autres choses avec paintComponent(). Mais le plus amusant commence quand vous faites vos propres expériences. Essayez de jouer avec les nombres. Regardez la documentation de la classe Graphics (nous verrons plus tard que vous disposez *d'autres* ressources que la classe Graphics).

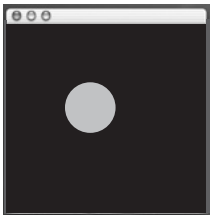
Afficher une image JPEG

```
public void paintComponent(Graphics g) {

    Image image = new ImageIcon("catzilla.jpg").getImage();

    g.drawImage(image,3,4,this);

}
```

Ici, le nom de votre fichier.

Les coordonnées (x,y) du point où doit se trouver le coin supérieur gauche de l'image. Autrement dit, à 3 pixels du bord gauche et à 4 du haut du panneau. Ces nombres sont toujours relatifs au widget (dans ce cas votre sous-classe de JPanel), non au cadre lui-même.

Peindre un cercle coloré de façon aléatoire sur fond noir



```
public void paintComponent(Graphics g) {

   g.fillRect(0,0,this.getWidth(), this.getHeight());



   int rouge = (int) (Math.random() * 255);
   int vert = (int) (Math.random() * 255);
   int bleu = (int) (Math.random() * 255);

   Color randomColor = new Color(rouge, vert, bleu);
   g.setColor(randomColor);
   g.fillOval(70,70,100,100);
}
```

Remplir tout le panneau de noir (la couleur par défaut).

Les deux premiers arguments définissent les coordonnées (x,y) du coin supérieur gauche par rapport au panneau, déterminant le point où le dessin commence. 0, 0 signifie donc « commencer à 0 pixel du bord gauche et à 0 pixel du haut ». Les deux autres arguments signifient « donner à ce rectangle la largeur du panneau (this.getWidth()), et la hauteur du panneau (this.getHeight) ».

Vous pouvez créer une couleur en passant 3 arguments qui représentent des valeurs RGB.

Commencer à 70 pixels de la gauche, 70 du haut, et tracer une figure de 100 pixels de large et 100 pixels de haut.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Derrière toute bonne référence à Graphics il y a un objet Graphics2D.

L'argument de paintComponent() est déclaré de type Graphics (java.awt.Graphics).

```
public void paintComponent(Graphics g) { }
```

Donc, le paramètre "g" EST-UN objet Graphics. Ce qui signifie qu'il *pourrait* être une *sous-classe* de Graphics (à cause du polymorphisme). Et de fait, il l'est.

L'objet référencé par le paramètre "g" est en réalité une instance de la classe Graphics2D.

Pourquoi est-ce important ? Parce qu'il y a des choses que vous pouvez faire avec une référence Graphics2D qui sont impossibles avec une référence Graphics. Un objet Graphics2D a plus de possibilités qu'un objet Graphics, et c'est en fait un objet Graphics2D qui se cache derrière une référence Graphics.

Pensez au polymorphisme. Le compilateur décide quelles méthodes vous pouvez appeler en fonction du type de la référence, non du type de l'objet. Si vous avez un objet Chien référencé par une variable référence de type Animal :

```
Animal a = new Chien();
```

Vous ne pouvez PAS dire :

```
a.aboyer();
```

Même si vous savez que c'est un chien qui est derrière. Le compilateur regarde "a", voit qu'il est de type Animal et qu'il n'y a pas de bouton de télécommande nommé aboyer() dans la classe Animal. Mais vous pouvez toujours faire en sorte que l'objet redevienne le chien qu'il est réellement en écrivant :

```
Chien c = (Chien) a;
c.aboyer();
```

L'intérêt de l'objet Graphics est donc le suivant :

Si vous avez besoin d'une méthode de la classe Graphics2D, vous ne pouvez pas *utiliser* le paramètre de paintComponent ("g") directement. Mais vous pouvez le *convertir* avec une nouvelle variable de type Graphics2D.

```
Graphics2D g2d = (Graphics2D) g;
```

Méthodes que vous pouvez appeler sur une référence Graphics :

- drawImage()
- drawLine()
- drawPolygon
- drawRect()
- drawOval()
- fillRect()
- fillRoundRect()
- setColor()

Pour convertir l'*objet* Graphics2D en *référence* Graphics2D :

```
Graphics2D g2d = (Graphics2D) g;
```

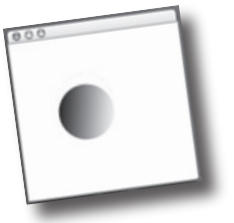
Méthodes que vous pouvez appeler sur une référence Graphics2D :

- fill3DRect()
- draw3DRect()
- rotate()
- scale()
- shear()
- transform()
- setRenderingHints()

(Cette liste n'est pas complète. Consultez la documentation.)

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Parce que la vie est trop courte pour peindre un cercle d'une seule couleur quand un mélange de nuances vous attend.



Objet Graphics2D déguisé en simple objet Graphics.

```
public void paintComponent(Graphics g) {

    Graphics2D g2d = (Graphics2D) g;

    GradientPaint gradient = new GradientPaint(70,70,Color.blue, 150,150, Color.orange);

    g2d.setPaint(gradient);
    g2d.fillOval(70,70,100,100);
}
```

Convertissez-le pour pouvoir appeler une méthode de Graphics2D que Graphics ne possède pas.

coordonnées de départ — couleur de départ — coordonnées d'arrivée — couleur d'arrivée

Indique au pinceau de peindre en dégradé (gradient).

La méthode fillOval() signifie « remplir l'ovale de la "couleur" chargée dans le pinceau (le dégradé) ».

```
public void paintComponent(Graphics g) {
    Graphics2D g2d = (Graphics2D) g;

    int red = (int) (Math.random() * 255);
    int green = (int) (Math.random() * 255);
    int blue = (int) (Math.random() * 255);
    Color startColor = new Color(red, green, blue);

    red = (int) (Math.random() * 255);
    green = (int) (Math.random() * 255);
    blue = (int) (Math.random() * 255);
    Color endColor = new Color(red, green, blue);

    GradientPaint gradient = new GradientPaint(70,70,startColor, 150,150, endColor);
    g2d.setPaint(gradient);
    g2d.fillOval(70,70,100,100);
}
```

Le code précédent modifié pour définir des couleurs de départ et d'arrivée aléatoires. Essayez-le !

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

POINTS D'IMPACT

ÉVÉNEMENTS

- Pour créer une IHM, commencez par une fenêtre, généralement un cadre de type JFrame :
```
JFrame cadre = new JFrame();
```
- Vous pouvez ajouter des widgets (boutons, champs de texte, etc.) au JFrame avec :
```
frame.getContentPane().add(bouton);
```
- Contrairement à beaucoup d'autres composants, on ne peut pas ajouter d'éléments directement à un JFrame. On les insère dans son panneau de contenu.
- Pour afficher le JFrame, vous devez définir sa taille et le rendre visible :
```
cadre.setSize(300,300);
cadre.setVisible(true);
```
- Pour savoir quand l'utilisateur clique sur un bouton (ou effectue tout autre action dans l'interface graphique), vous devez écouter un événement.
- Pour pouvoir écouter un événement, vous devez faire part de votre intérêt à sa source. Une source d'événements est l'élément (bouton, case à cocher, etc.) qui déclenche l'événement.
- L'interface auditeur donne à la source d'événements un moyen de vous rappeler, parce qu'elle définit les méthodes que la source appelle quand un événement se produit.
- Pour s'enregistrer auprès d'une source, appelez sa méthode d'enregistrement. Celle-ci prend toujours la forme :
  add<Type d'événement>Listener. Par exemple, pour les ActionEvents, d'un bouton, appelez :
```
bouton.addActionListener(this);
```
- Implémentez l'interface auditeur en écrivant toutes ses méthodes de gestion des événements. Placez le code dans la méthode de rappel de l'auditeur. Pour les ActionEvents, la méthode est :
```
public void actionPerformed(ActionEvent
                            event) {
    bouton.setText("c'est cliqué!");
}
```
- L'objet event transmis à la méthode de gestion de l'événement contient des informations sur celui-ci, notamment sa source.

GRAPHISMES

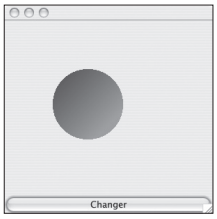
- Vous pouvez dessiner des graphismes en 2D directement sur un widget.
- Vous pouvez directement placer un fichier .gif ou .jpeg sur un widget.
- Pour créer vos propres graphismes (y compris les .gif et les .jpeg), créez une sous-classe de JPanel et redéfinissez la méthode paintComponent().
- La méthode paintComponent() est appelée par le système. VOUS NE L'APPELEZ JAMAIS VOUS-MÊME. L'argument de paintComponent() est un objet Graphics qui vous fournit la surface sur laquelle dessiner. Vous ne pouvez pas construire cet objet vous-même.
- Les méthodes usuelles à appeler sur un objet Graphics (le paramètre de paintComponent ()) sont :
```
g.setColor(Color.blue);
g.fillRect(20,50,100,120);
```
- Pour un .jpg, construisez une Image comme suit :
```
Image image = new ImageIcon("catzilla.
jpg").getImage();
```
  et affichez l'image avec :
```
g.drawImage(image,3,4,this);
```
- L'objet référencé par le paramètre de paintComponent() est en réalité une instance de la classe Graphics2D. Cette classe possède de nombreuses méthodes, notamment :
  fill3DRect(), draw3DRect(), rotate(), scale(), shear(), transform()
- Pour pouvoir invoquer les méthodes de la classe Graphics2D, vous devez convertir le paramètre et transformer l'objet Graphics en objet Graphics2D :
```
Graphics2D g2d = (Graphics2D) g;
```


Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

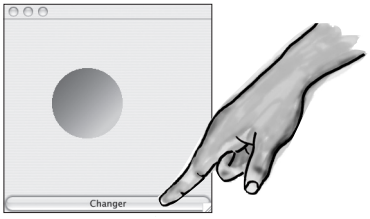
Nous savons gérer un événement.
Nous savons dessiner.
Mais pouvons-nous dessiner *quand* nous recevons un événement?

Lions un événement à un changement dans notre panneau. Nous allons faire en sorte que le cercle change de couleur à chaque clic sur un bouton. Voici comment le programme se déroule :

Lancer l'application

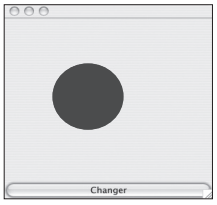


1. Le cadre est construit avec deux widgets (votre panneau et un bouton). Un auditeur est créé et enregistré auprès du bouton. Puis le cadre s'affiche et attend que l'utilisateur clique.



2. L'utilisateur clique sur le bouton. Le bouton crée un objet event et appelle le gestionnaire d'événements de l'auditeur.

3. Le gestionnaire d'événements appelle repaint() sur le cadre. Le système appelle paintComponent() sur le panneau.



4. Voila! Une nouvelle couleur est affichée, parce que paintComponent() s'exécute de nouveau et remplit le cercle avec une couleur aléatoire.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Agencement d'une interface graphique : placer plusieurs widgets dans un cadre

Nous abordons les questions d'agencement au chapitre suivant, mais nous allons en voir les rudiments ici. Par défaut, un cadre possède cinq régions, et on ne peut ajouter qu'un seul élément à une région. Mais pas de panique ! Ce seul élément peut être un panneau qui contient trois autres éléments, dont un panneau qui contient deux éléments de plus et… vous voyez l'idée. En fait, nous avons « triché » quand nous avons ajouté un bouton au cadre comme ceci :

```
cadre.getContentPane().add(bouton);
```

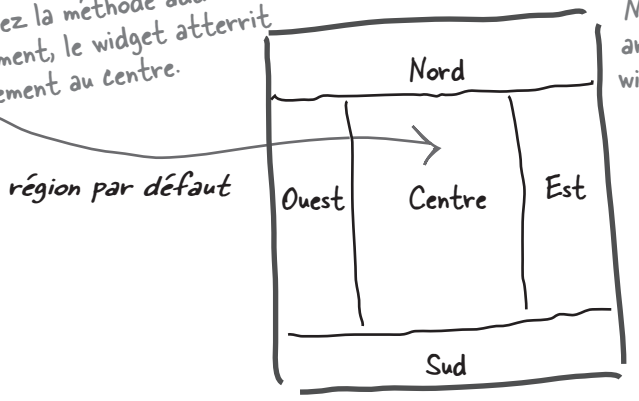
Ce n'est vas vraiment ainsi que vous êtes censé procéder (une méthode à un seul argument).

Méthode préférable (et généralement obligatoire) pour ajouter un panneau au cadre. Toujours spécifier OÙ (dans quelle région) vous voulez placer le widget.

```
cadre.getContentPane().add(BorderLayout.CENTER, bouton);
```

Nous appelons la méthode add() avec deux arguments : la région (une constante) et le widget à ajouter à cette région.

Si vous appelez la méthode add() avec un seul argument, le widget atterrit automatiquement au centre.



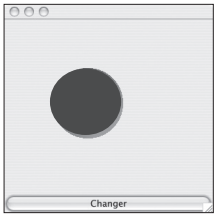
À vos crayons

Étant donné les images de la page 351, écrivez le code qui ajoute le bouton et le panneau au cadre.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Le cercle change de couleur chaque fois que vous cliquez sur le bouton.

Le panneau personnalisé (instance de MonPanneau) est dans la région "CENTER" du cadre.



Le bouton est dans la région "SOUTH" du cadre.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;


public class SimpleIhm3C implements ActionListener {

    JFrame cadre;

    public static void main (String[] args) {
       SimpleIhm3C ihm = new SimpleIhm3C();
       ihm.go();
    }

    public void go() {
       cadre = new JFrame();
       cadre.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

       JButton bouton = new JButton("Changer");
       bouton.addActionListener(this);

       MonPanneau panneau = new MonPanneau();

       cadre.getContentPane().add(BorderLayout.SOUTH, bouton);
       cadre.getContentPane().add(BorderLayout.CENTER, panneau);
       cadre.setSize(300,300);
       cadre.setVisible(true);
    }

    public void actionPerformed(ActionEvent event) {
        cadre.repaint();
    }
}
```

Ajouter l'auditeur (this) au bouton.

Ajouter les deux widgets (bouton et panneau) aux deux régions du cadre.

Quand l'utilisateur clique, dire au cadre de se repeindre lui-même. Cela signifie que la méthode paintComponent() est appelée sur chaque widget du cadre!

```
class MonPanneau extends JPanel {

   public void paintComponent(Graphics g) {
     // Code pour remplir l'ovale avec une couleur aléatoire
     // Voir page 347 pour le code
   }

}
```

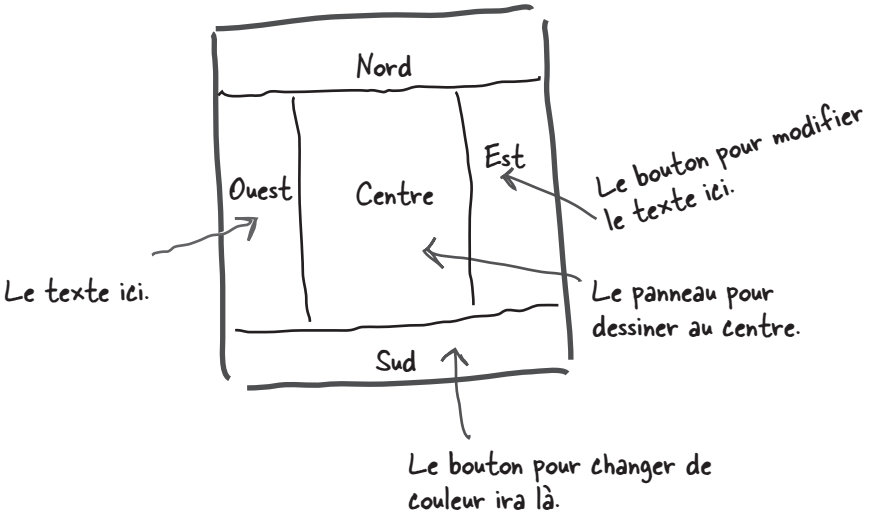
La méthode paintComponent() du panneau est appelée chaque fois que l'utilisateur clique.

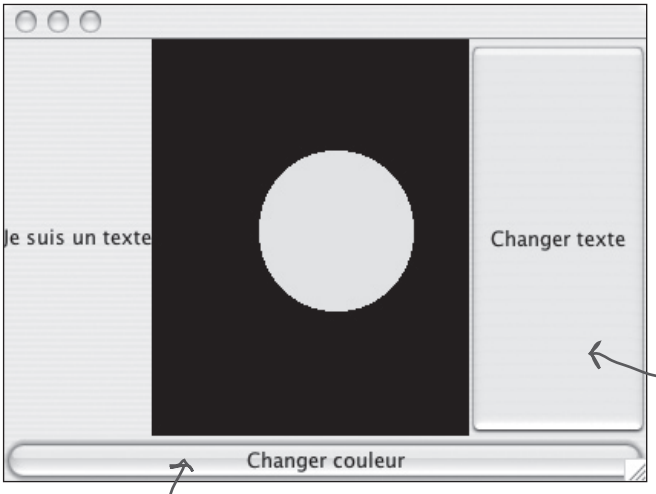
Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Essayons avec DEUX boutons

Le bouton sud se comportera comme maintenant, se bornant à appeler la méthode repaint() sur le cadre. Le second bouton (que nous plaçons à l'est) modifiera le texte d'une étiquette. (Une étiquette n'est rien d'autre qu'un texte affiché à l'écran.)

Il nous faut donc QUATRE widgets





... et DEUX événements

Mmmm…

Est-ce qu'au moins c'est possible ? Comment peut-on avoir deux événements quand on n'a qu'une seule méthode actionPerformed() ?

Ce bouton modifie le texte de l'autre côté de l'écran.

Ce bouton modifie la couleur du cercle.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Comment obtenir des événements pour deux boutons, si chaque bouton doit faire quelque chose de différent?

1 *option un*
Implémenter deux méthodes actionPerformed()

```
class MonIhm implements ActionListener {
    // un tas de code ici, puis:

    public void actionPerformed(ActionEvent event) {
        cadre.repaint();
    }

    public void actionPerformed(ActionEvent event) {
        etiquette.setText("Ouille!");
    }
}
```

Mais ceci est impossible!

Problème. Vous ne pouvez pas! On ne peut pas implémenter la même méthode deux fois dans une classe Java. Le compilateur refuse. Et même si vous le *pouviez*, comment la source d'événements saurait-elle *laquelle* des deux méthodes appeler?

2 *option deux*
Enregistrer le même auditeur auprès des deux boutons

```
class MonIhm implements ActionListener {
    // déclarer un paquet de variables d'instance ici

    public void go() {
       // construction de l'IHM
       boutonCouleur = new JButton();
       boutonTexte = new JButton();
       boutonCouleur.addActionListener(this);
       boutonTexte.addActionListener(this);
       // suite du code ...
    }

   public void actionPerformed(ActionEvent event) {
        if (event.getSource() == boutonCouleur) {
            cadre.repaint();
        } else {
           etiquette.setText("Ouille!");
        }
    }
}
```

Enregistrer le même auditeur auprès des deux boutons.

Tester l'objet event pour déterminer quel bouton l'a déclenché, puis décider quoi faire en conséquence.

Problème. Cela marche, mais ce n'est pas très OO. Un gestionnaire d'événements qui fait plusieurs choses différentes signifie que vous avez une seule méthode qui fait plusieurs choses différentes. Si vous devez modifier la façon dont *une* source est gérée, vous devrez intervenir dans *tous* les gestionnaires d'événements. C'est parfois une bonne solution, mais qui va à l'encontre de la maintenabilité et de l'extensibilité.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Comment obtenir des événements pour deux boutons, si chaque bouton doit faire quelque chose de différent?

3 *option trois*
Créer deux classes ActionListener distinctes

```
class MonIhm  {
   JFrame cadre;
   JLabel etiquette;
   void go() {
      // code pour instancier les deux auditeurs et en enregistrer
      // un auprès de boutonCouleur et l'autre auprès de boutonTexte
   }
} // fin de la classe
```

```
class boutonCouleurListener implements ActionListener {
    public void actionPerformed(ActionEvent event) {
        cadre.repaint();
    }
}
```

Ne fonctionne pas! Cette classe n'a pas de référence à la variable "cadre" de MonIHM.

```
class LabelButtonListener implements ActionListener {
    public void actionPerformed(ActionEvent event) {
        etiquette.setText("Ouille!");
    }
}
```

Problème! Cette classe n'a pas de référence à la variable "etiquette".

Problème. Ces classes n'auront pas accès aux variables sur lesquelles elles doivent agir, « cadre » et « etiquette ». Vous pourriez corriger le problème, mais il faudrait donner à chacune des classes auditeurs une référence à la classe principale de l'IHM, pour que, à l'intérieur des méthodes actionPerformed(), l'auditeur puisse utiliser cette référence pour accéder aux variables de la classe. Mais cela viole le principe d'encapsulation, et nous devrions probablement créer des méthodes get pour les widgets (getCadre(), getEtiquette(), etc.). Et il faudrait probablement ajouter un constructeur à la classe auditeur pour pouvoir lui transmettre la référence à l'IHM au moment où l'auditeur est instancié. Et le code va devenir de plus en plus complexe et se transformer en véritable gâchis.

Il doit y avoir un meilleur moyen !

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Les classes internes à la rescousse!

Vous pouvez imbriquer une classe dans une autre classe. C'est facile. Vérifiez simplement que la définition de la classe interne est bien à *l'intérieur* des accolades de la classe externe.

Classe interne simple :

```
class MaClasseExterne  {

   class MaClasseInterne {
      void go() {
       }
    }

}
```

La classe interne est complètement encapsulée dans la classe externe.

Une classe interne a accès à toutes les méthodes et les variables de la classe externe, même celles qui sont privées.

Elle utilise ces méthodes et ces variables exactement comme si elles étaient déclarées dans la classe interne.

Une classe interne a un privilège spécial pour utiliser les méthodes et les variables de la classe externe. *Même celles qui sont privées.* Et elles sont utilisées comme si elles étaient définies dans la classe interne. C'est cela qui est si pratique dans les classes internes : elles présentent tous les avantages d'une classe normale, mais avec des droits d'accès spéciaux.

Classe interne utilisant une variable d'une classe externe

```
class MaClasseExterne  {

    private int x;

   class MaClasseInterne {
      void go() {
         x = 42;
       }
    } // fin de la classe interne

} // fin de la classe externe
```

"x" est utilisé comme si c'était une variable de la classe interne !

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

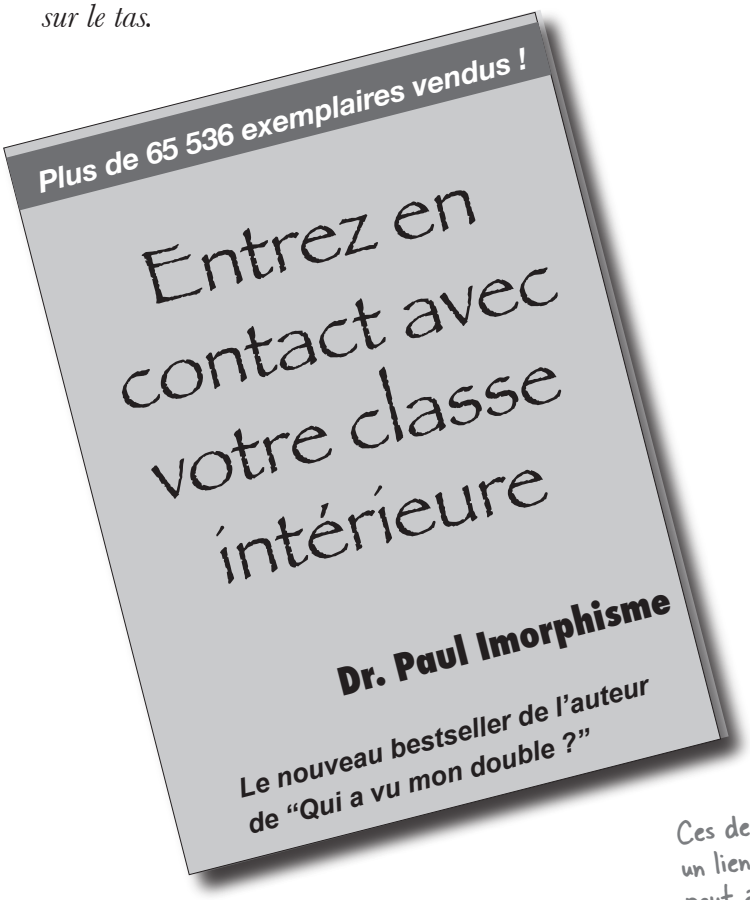
Une instance de classe interne doit être liée à une instance de classe externe*.

Quand nous parlons de classe interne accédant aux variables et aux méthodes de la classe externe, n'oubliez pas qu'il s'agit en réalité d'une instance de la classe interne qui accède aux variables et aux méthodes d'une instance de la classe externe. Mais *quelle* instance ?

Une instance arbitraire quelconque de la classe interne peut-elle accéder aux variables et aux méthodes de n'importe quelle instance de la classe externe? **Non !**

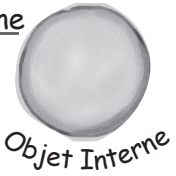
Un objet ***interne*** *doit être lié à un objet spécifique* ***externe*** *sur le tas.*

Un objet interne partage un lien particulier avec un objet externe.

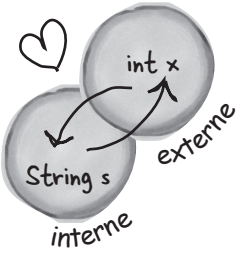


1. Créer une instance de la classe externe.

2. Créer une instance de la classe interne en utilisant l'instance de la classe externe



3. Ces deux objets sont maintenant intimement liés.



Ces deux objets sur le tas ont un lien spécial. L'objet interne peut accéder aux variables de l'externe (et inversement).

*Il y a une exception à cette règle, dans un cas très particulier — une classe interne définie dans une méthode statique. Mais nous n'irons pas jusque là, et vous pourrez probablement vivre toute votre vie de programmeur Java sans jamais en rencontrer une.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Comment créer une instance de classe interne

Si vous instanciez une classe interne *à partir* du code d'une classe externe, l'instance de la classe externe est celle à laquelle l'objet interne sera «lié». Par exemple, si le code d'une méthode instancie la classe interne, l'objet interne sera lié à l'instance dont la méthode s'exécute.

Le code d'une classe externe peut instancier l'une de ses propres classes internes, exactement de la même manière qu'il instancie toute autre classe… **`new MonInterne()`**.

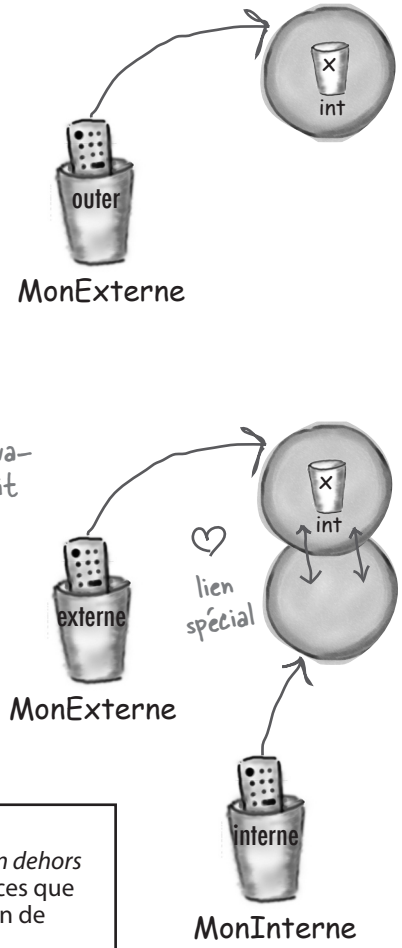
```
class MyExterne  {

   private int x;

   Mon Interne interne = new MonInterne();

   public void faireQqch() {
      interne.go();
   }

   class MonInterne {
      void go() {
         x = 42;
      }
   } // fin de la classe interne

} // fin de la classe externe
```

La classe externe a une variable d'instance "x" privée.

Créer une instance de la classe interne.

Appeler une méthode de la classe interne.

La méthode de la classe interne accède à la variable d'instance "x", comme si "x" appartenait à la classe interne.



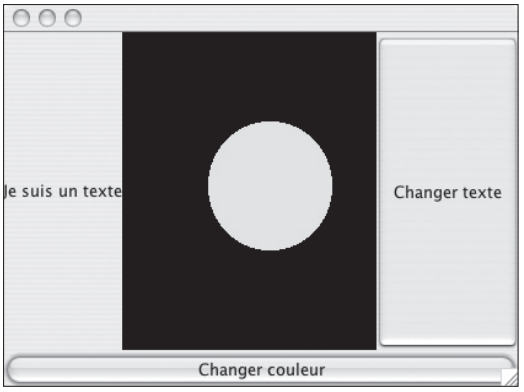
Aparté

Vous *pouvez* instancier une classe interne dans une portion de code qui s'exécute *en dehors* de la classe externe, mais vous devez employer une syntaxe spéciale. Il y a des chances que toute votre carrière de programmeur Java se passe sans que vous ayez jamais besoin de créer un objet interne de l'extérieur, mais au cas où vous seriez intéressé…

```
class Foo {
   public static void main (String[] args) {
      MonExterne objetExt = new MonExterne();
      MonExterne.MonInterne ObjetInt = objetExt.new MonInterne();
   }
}
```


Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

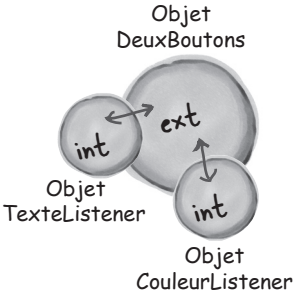
Maintenant, nous pouvons créer deux boutons




```
public class DeuxBoutons {

    JFrame cadre;
    JLabel etiquette;

    public static void main (String[] args) {
       DeuxBoutons ihm = new DeuxBoutons ();
       ihm.go();
    }

    public void go() {
       cadre = new JFrame();
       cadre.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

       JButton boutonTexte = new JButton("Changer texte");
       boutonTexte.addActionListener(new TexteListener());

       JButton boutonCouleur = new JButton("Changer couleur");
       boutonCouleur.addActionListener(new CouleurListener());

       etiquette = new JLabel("Je suis un texte");
       MonPanneau panneau = new MonPanneau();

       cadre.getContentPane().add(BorderLayout.SOUTH, boutonCouleur);
       cadre.getContentPane().add(BorderLayout.CENTER, panneau);
       cadre.getContentPane().add(BorderLayout.EAST, boutonTexte);
       cadre.getContentPane().add(BorderLayout.WEST, etiquette);

       cadre.setSize(300,300);
       cadre.setVisible(true);
    }

    class TexteListener implements ActionListener {
        public void actionPerformed(ActionEvent event) {
            etiquette.setText("Aïe!");
        }
    } // fin de classe interne

    class CouleurListener implements ActionListener {
        public void actionPerformed(ActionEvent event) {
            cadre.repaint();
        }
    } // fin de classe interne

}
```

La classe principale n'implémente plus ActionListener.

Au lieu de passer (this) à la méthode d'enregistrement de l'auditeur du bouton, passer une nouvelle instance de la classe appropriée.

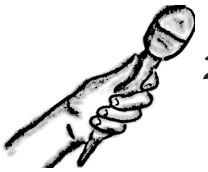


Maintenant, nous avons DEUX ActionListeners dans une seule classe.

La classe interne connaît "etiquette".

La classe interne utilise la variable d'instance "cadre" sans avoir de référence explicite à l'objet de la classe externe.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>



INTERVIEW

L'interview de cette semaine : une instance de classe interne.

JTLP : Quel est l'intérêt des classes internes ?

Objet interne : Par où commencer ? Nous vous donnons une chance d'implémenter la même interface plusieurs fois dans une classe. Vous vous souvenez qu'une classe Java normale ne permet pas d'implémenter une méthode plus d'une fois. Mais comme chaque classe *interne* peut implémenter la *même* interface, vous pouvez avoir plein d'implémentations *différentes* des mêmes méthodes d'une interface.

JTLP : Et pourquoi voudrait-on implémenter la même méthode deux fois ?

Objet interne : Revoyons les gestionnaires d'événements des IHM. Réfléchissez… Si vous voulez que *trois* boutons aient chacun un comportement différent, vous utiliserez *trois* classes internes qui implémenteront toutes trois ActionListener — ce qui signifie que chaque classe implémentera sa propre méthode actionPerformed().

JTLP : Les gestionnaires d'événements sont donc la seule raison d'utiliser des classes internes ?

Objet interne : Bien sûr que non. Les gestionnaires d'événements ne sont qu'un exemple évident. Chaque fois que vous avez besoin d'une classe séparée, mais que vous voulez quand même que cette classe se *comporte* comme si elle faisait partie d'une autre *classe*, une classe interne est la meilleure solution — et parfois la seule.

JTLP : Il y a encore quelque chose que je ne comprends pas. Si vous voulez que la classe interne se *comporte* comme si elle appartenait à la classe externe, pourquoi avoir une classe séparée ? Pourquoi le code de la classe interne ne serait-il pas dans la classe externe ?

Objet interne : Je viens de vous citer un scénario dans lequel vous avez besoin de plusieurs implémentations d'une même interface. Mais même si vous n'utilisez pas d'interfaces, vous pouvez avoir besoin de deux classes différentes parce que ces deux classes représentent deux *entités* différentes. C'est de la bonne approche objet.

JTLP : Stop. Je croyais qu'une bonne partie de la conception OO concernait la réutilisation et la maintenance. Vous savez, l'idée selon laquelle si on a deux classes séparées, on peut les modifier et les utiliser indépendamment, au lieu de tout fourrer dans une seule classe, etc. Mais avec une classe *interne*, il n'y a jamais qu'une seule classe à la fin, non ? La classe externe est la seule qui soit réutilisable et séparée de tout le reste. Les classes internes ne sont pas précisément réutilisables. En fait, j'ai entendu dire qu'on les qualifiait de « réinutilisables ».

Objet interne : Oui, il est vrai qu'une classe interne n'est pas aussi réutilisable, quelquefois même pas du tout, parce qu'elle est intimement liée aux variables d'instance et aux méthodes de la classe interne. Mais elle…

JTLP : Ce qui ne fait qu'apporter de l'eau à mon moulin ! Si elle n'est pas réutilisable, pourquoi se soucier de créer une classe séparée. Je veux dire en dehors du problème des interfaces, qui m'a plutôt l'air d'un expédient.

Objet interne : Comme je vous l'ai dit, vous devez penser au polymorphisme.

JTLP : O.K. Et pourquoi penserais-je au polymorphisme ?

Objet interne : Parce que les classes externes et internes peuvent devoir être de types différents ! Commençons par l'exemple de l'auditeur polymorphe d'une IHM. De quel type est l'argument de la méthode qui enregistre l'auditeur du bouton ? Autrement dit, si vous vérifiez dans la documentation, de quel type (classe ou interface) est l'argument que vous passez à la méthode addActionListener() ?

JTLP : Vous devez passer un auditeur. Quelque chose qui implémente une interface auditeur particulière, en l'occurrence ActionListener. Tout le monde sait ça. Où voulez-vous en venir ?

Objet interne : Je veux en venir au fait que, du point de vue du polymorphisme, vous avez une méthode qui n'accepte qu'un *type* particulier. Quelque chose qui EST-UN ActionListener. Mais — et c'est là le point important — que se passe-t-il si votre classe doit avoir une relation EST-UN avec un type de *classe* et non d'interface ?

JTLP : Vous ne pouvez pas juste faire dériver votre classe de celle dont elle doit être une partie ? Est-ce que ce n'est pas justement tout l'intérêt du sous-classement ? Si B est une sous-classe de A, on peut utiliser un B partout où on attend un A ? Toute cette histoire de passer un Chien là où un Animal est le type déclaré ?

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Objet interne : Bingo ! Et si vous devez réussir le test EST-UN pour deux classes différentes ? Des classes qui ne sont pas dans la même hiérarchie d'héritage ?

JTLP : Oh, eh bien, on peut juste… hmmm. Je crois que j'ai compris. On peut toujours *implémenter* plusieurs interfaces, mais on ne peut étendre qu'une *seule* classe. Vous ne pouvez réussir qu'un seul test EST-UN quand il s'agit de types de *classes*.

Objet interne : Exactement ! Vous ne pouvez pas être à la fois un Chien et un Bouton. Mais si vous êtes un Chien qui a parfois besoin d'être un Bouton (pour pouvoir vous transmettre vous-même à des méthodes qui acceptent un Bouton), la classe Chien (qui étend Animal et ne peut donc étendre Bouton) peut avoir une classe *interne* qui agit en tant que Bouton de la part du Chien en étendant la classe Bouton. Et chaque fois qu'un Bouton est requis, le Chien peut transmettre son Bouton interne au lieu de se transmettre lui-même. Autrement dit, au lieu de dire x.takeButton(this), l'objet Chien appelle x.takeButton(new monBoutonInterne()).

JTLP : Est-ce que je peux avoir un exemple clair ?

Objet interne : Vous souvenez-vous du panneau que nous avons utilisé quand nous avons créé notre propre sous-classe de JPanel ? Cette classe est une classe séparée, non une classe interne. Et c'est bien, parce qu'elle n'a pas besoin d'un accès spécial aux variables d'instance de l'IHM principale. Mais si elle en avait besoin ? Si nous créions une animation, et qu'elle obtenait ses coordonnées de l'application principale (par exemple en fonction de quelque chose que l'utilisateur fait quelque part ailleurs dans l'IHM). Dans ce cas, si nous faisons du panneau une classe interne, il devient une sous-classe de JPanel, tandis que la classe externe est libre d'être une sous-classe d'autre chose.

JTLP : Je vois ! Et de toute façon le panneau n'est pas suffisamment réutilisable pour être une classe séparée, puisque ce que ce que nous dessinons est spécifique à cette seule application.

Objet interne : Voilà ! Vous y êtes !

JTLP : Bien. Nous pouvons donc passer à la nature de la relation que vous entretenez avec l'instance externe.

Objet interne : Mais qu'est-ce que vous avez tous ? Il n'y a donc pas assez de ragots sordides sur le polymorphisme ?

JTLP : Vous n'imaginez pas ce que le public est prêt à payer pour un bon scandale. Donc quelqu'un vous crée et vous vous trouvez instantanément lié à l'objet externe ? Non ?

Objet interne : Oui. Certains parlent à ce propos de mariage de raison. Nous n'avons pas notre mot à dire sur l'objet auquel nous sommes liés.

JTLP : Pour reprendre l'analogie, pouvez-vous divorcer et vous remarier avec quelqu'un d'autre ?

Objet interne : Non, c'est pour la vie entière.

JTLP : La vie de qui ? La vôtre ? Celle de l'objet externe ? Les deux ?

Objet interne : La mienne. Je ne peux être lié à aucun autre objet externe. Ma seule issue est le ramasse-miettes.

JTLP : Et l'objet externe ? Peut-il être associé à d'autres objets internes ?

Objet interne : Nous y sommes. C'est ça que vous vouliez ? Oui, mon prétendu « partenaire » peut en avoir autant qu'il en a envie.

JTLP : Est-ce de la… monogamie en série ? Ou est-ce qu'il peut les avoir tous en même temps ?

Objet interne : Tous en même temps. Vous êtes content ?

JTLP : Logique. Mais n'oubliez pas que c'est vous qui chantiez les louanges de « l'implémentation multiple d'une même interface ». Et si la classe externe a trois boutons, il est normal qu'elle ait trois classes internes (et donc trois objets) pour gérer les événements. Merci pour tout. Tenez, voilà un mouchoir.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Utiliser une classe interne pour l'animation

Nous avons vu pourquoi les classes internes étaient pratiques pour les auditeurs d'événements : elles permettent d'implémenter la même méthode de gestion des événements plusieurs fois. Nous allons maintenant voir l'utilité d'une classe interne utilisée comme sous-classe d'une superclasse dont la classe externe ne dérive pas. Autrement dit quand la classe externe et la classe interne appartiennent à deux hiérarchies d'héritage différentes !

Notre but est de créer une animation simple, dans laquelle notre cercle va traverser l'écran, de bas en haut et de gauche à droite.

Fonctionnement d'une animation simple

1. Dessiner un objet à une position (x,y) donnée

```
g.fillOval(20,50,100,100);
```

20 pixels de la gauche,
50 pixels du haut.

2. Redessiner l'objet à une position (x,y) différente

```
g.fillOval(25,55,100,100);
```

25 pixels de la gauche,
55 pixels du haut

(L'objet s'est un peu déplacé vers le bas et la droite

3. Répéter l'étape précédente en modifiant les valeurs de x et y aussi longtemps que l'animation est censée continuer.

il n'y a pas de
Questions stupides

Q : Pourquoi apprendre l'animation ? je ne pense pas que j'écrirai jamais des jeux.

R : Vous n'écrirez peut-être jamais de jeux, mais vous créerez peut-être des simulations, dans lesquelles les représentations évoluent dans le temps pour montrer le résultat d'un processus donné. Ou bien un outil de visualisation qui par exemple actualise un graphique pour montrer combien un programme utilise de mémoire, ou le volume de trafic qui transite par un serveur. Les animations sont utiles dans tous les programmes qui traduisent des nombres qui changent constamment en données visuelles plus parlantes. Convaincant, n'est-ce pas ? Bien sûr, ce n'est que la « justification officielle ». La vraie raison d'aborder ce sujet ici, c'est que c'est un exemple simple d'utilisation des classes internes. (Et aussi parce que nous *aimons* l'animation, que notre prochain ouvrage est sur J2EE, et que nous *savons* qu'il n'y aura pas d'animation dans celui-là.)

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Ce qu'il nous faudrait vraiment...

```
class MonPanneau extends JPanel {
      public void paintComponent(Graphics g) {
         g.setColor(Color.orange);
         g.fillOval(x,y,100,100);
      }
}
```

Chaque fois que paintComponent() est appelée, l'ovale est redessiné à un emplacement différent.

À vos crayons

Mais où prenons-nous les nouvelles coordonnées x et y ?

Et qui appelle repaint() ?

Voyez si vous pouvez **imaginer une solution simple** pour animer la balle et lui faire traverser le panneau de haut en bas et de gauche à droite. Notre réponse se trouve page suivante : ne tournez pas celle-ci avant d'avoir terminé !

Super indice : faites du panneau une classe interne.

Autre indice : ne placez aucune espèce de boucle dans la méthode paintComponent().

Notez vos idées (ou votre code) ici :

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Le code complet de l'animation

```
import javax.swing.*;
import java.awt.*;


public class SimpleAnimation {

    int x = 70;
    int y = 70;

    public static void main (String[] args) {
       SimpleAnimation ihm = new SimpleAnimation ();
       ihm.go();
   }

   public void go() {
       JFrame cadre = new JFrame();
       cadre.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

       MonPanneau panneau = new MonPanneau();

       cadre.getContentPane().add(panneau);
       cadre.setSize(300,300);
       cadre.setVisible(true);

       for (int i = 0; i < 130; i++) {

          x++;
          y++;

          panneau.repaint();

          try {
            Thread.sleep(50);
          } catch(Exception ex) { }
       }

   }// fin de la méthode go()

   class MonPanneau extends JPanel {

      public void paintComponent(Graphics g) {
         g.setColor(Color.green);
         g.fillOval(x,y,40,40);

      }
   } // fin de la classe interne
} // fin de la classe externe
```

Créer deux variables d'instance dans la classe principale de l'IHM pour représenter les coordonnées x et y du cercle.

Rien de nouveau ici. Créer les widgets et les placer dans le cadre.

L'action est ici!

Répéter 130 fois.

Incrémenter les coordonnées x et y.

Dire au panneau de se repeindre lui-même (pour afficher le cercle au nouvel emplacement).

Ralentir un peu, sinon il bougera si vite que nous ne le VERRONS pas se déplacer. Ne vous inquiétez pas, vous n'êtes pas censé connaître déjà les threads. Nous en parlerons au chapitre 15.

Maintenant, c'est une classe interne.

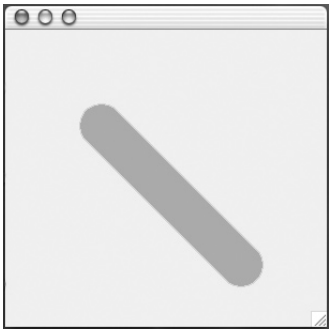
Utiliser les coordonnées x et y continuellement mises à jour de la classe externe.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Hum. Il n'a pas bougé... il a *bavé*.

Où est l'erreur ?

Il y a juste un petit défaut dans la méthode paintComponent().



Ce n'est pas vraiment le résultat recherché !

Nous avons oublié d'effacer au fur et à mesure, si bien que nous avons une trace !

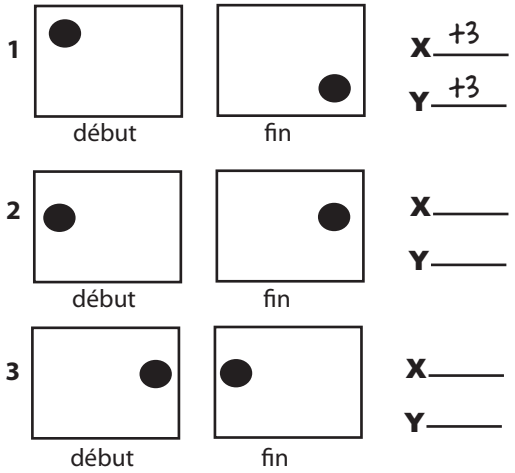
Pour corriger ce bogue, il suffit de remplir tout le panneau avec la couleur de fond avant de repeindre le cercle à chaque fois. Le code ci-dessous ajoute deux lignes au début de la méthode : l'une pour initialiser la couleur à blanc (pour le fond du panneau) et l'autre pour remplir tout le rectangle qui forme le panneau avec cette couleur. En bon français, le code ci-dessous signifie « Remplis le rectangle commençant aux coordonnées 0,0 (0 pixels de la gauche et 0 pixels du haut) et donne lui la taille actuelle du panneau.

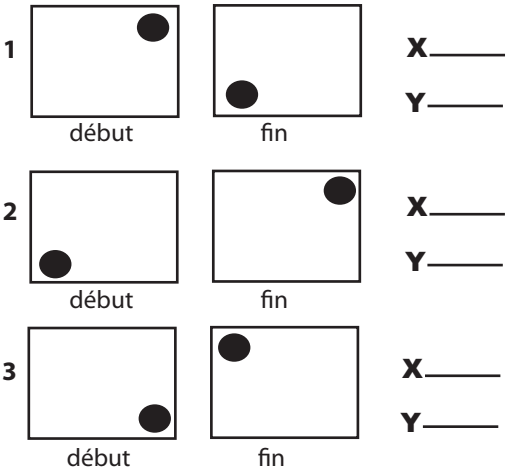
```
public void paintComponent(Graphics g) {
   g.setColor(Color.white);
   g.fillRect(0,0,this.getWidth(), this.getHeight());

   g.setColor(Color.green);
   g.fillOval(x,y,40,40);

}
```

getWidth() et getHeight() sont des méthodes héritées de JPanel.

À vos crayons (optionnel, juste pour le plaisir)

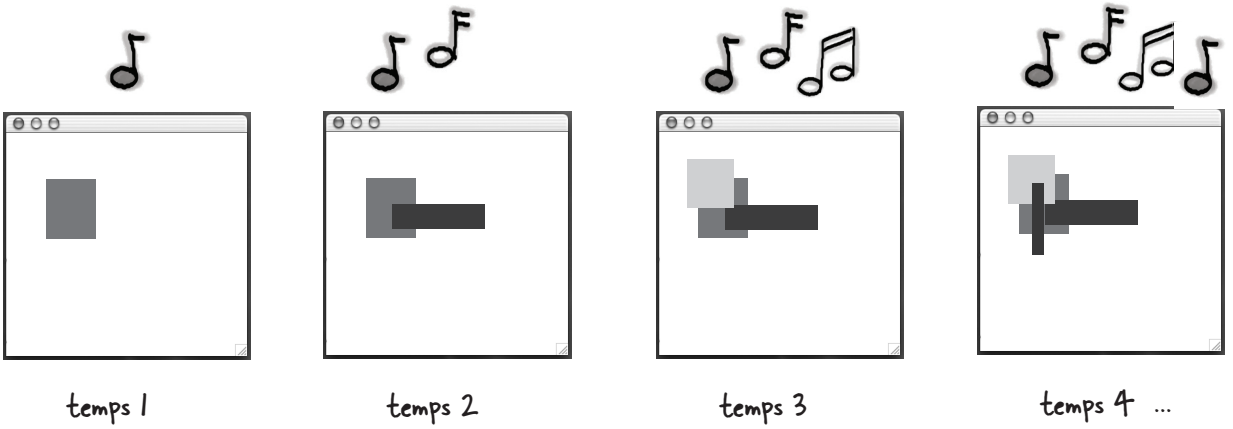
Comment modifieriez vous les coordonnées x et y pour produire les animations ci-dessous ? (Supposons que les incréments du premier exemple soient de 3 pixels.)





Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Recettes de code



Créons un clip musical. Nous allons utiliser des graphismes aléatoires générés par Java, qui suivront le rythme de la musique.

Ce faisant, nous enregistrerons (et écouterons) une nouvelle sorte d'événement, indépendant de l'interface graphique, déclenché par la musique elle-même.

Souvenez-vous que cette partie est entièrement optionnelle. Mais nous pensons que c'est bon pour vous. Et vous allez aimer ça. Et vous pourrez vous en servir pour impressionner vos collègues.

(Il est vrai qu'il faudrait qu'ils soient faciles à impressionner, mais quand-même...)

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Autres événements

Bon, ce ne sera peut-être pas exactement un clip vidéo, mais nous allons écrire des graphismes aléatoires à l'écran au rythme de la musique. En deux mots, ce programme écoute la musique et crée un rectangle aléatoire sur chaque temps.

Ceci va nous poser un certain nombre de questions. Jusqu'ici, nous n'avons écouté que les événements liés à l'IHM. Maintenant, nous devons nous occuper d'un type particulier d'événement MIDI. Mais le processus est virtuellement identique : on implémente une interface auditeur, on enregistre l'auditeur auprès une source d'événements, puis on s'installe confortablement en attendant que la source appelle le gestionnaire d'événements (la méthode définie dans l'interface).

La solution la plus simple serait d'écouter directement les événements MIDI : chaque fois que le séquenceur recevrait l'événement, notre code le recevrait aussi et pourrait tracer le rectangle. Mais… il y a un problème. Un bogue, en fait, qui va nous empêcher d'écouter les événements MIDI que nous créons (ceux de NOTE ON).

Nous allons donc devoir le contourner. Il existe un autre type d'événement MIDI que nous pouvons écouter, un événement contrôleur nommé, comme il se doit, ControllerEvent. Notre solution consiste à nous enregistrer auprès des ControllerEvents, puis à nous assurer qu'à chaque événement NOTE ON correspond un ControllerEvent qui se déclenche sur le même « temps ». Comment être sûr que le ControllerEvent est déclenché en même temps ? Nous l'ajoutons à la piste, tout comme les autres événements ! Autrement dit, notre séquence musicale se déroule comme ceci :

TEMPS 1 - NOTE ON, CONTROLLER EVENT

TEMPS 2 - NOTE OFF

TEMPS 3 - NOTE ON, CONTROLLER EVENT

TEMPS 4 - NOTE OFF

et ainsi de suite.

Mais avant de nous plonger dans le programme, simplifions un peu la création des messages et des événements MIDI, car il va nous en falloir beaucoup.

Ce que le programme doit faire :

1. Créer une série de messages/événements MIDI pour jouer des notes au hasard sur un piano (ou tout autre instrument de votre choix).
2. Enregistrer un auditeur pour les événements.
3. Lancer le séquenceur.
4. Chaque fois que le gestionnaire d'événements du séquenceur est appelé, dessiner un rectangle aléatoire sur le panneau et appeler repaint().

Nous allons le construire en trois itérations :

1. Version Un : Code qui simplifie la création et l'ajout des événements MIDI, car il nous en faudra beaucoup.
2. Version Deux : Enregistrer et écouter les événements, mais sans graphisme. Afficher un message pour chaque temps sur la ligne de commande.
4. Version Trois : Le vrai programme. Ajouter la partie graphique à la version deux.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Une façon plus simple de créer des messages / événements

Pour l'instant, créer des messages et ajouter des événements est une tâche fastidieuse. Pour chaque message, il faut en créer une instance (dans ce cas de ShortMessage), appeler setMessage(), créer un MidiEvent et ajouter l'événement à la piste. Dans le code du dernier chapitre, nous avons effectué chaque étape pour chaque message. Cela signifie huit lignes de code rien que pour jouer une note puis arrêter ! Quatre lignes pour l'événement NOTE ON et quatre lignes pour l'événement NOTE OFF.

```
ShortMessage a = new ShortMessage();
a.setMessage(144, 1, note, 100);
MidiEvent noteOn = new MidiEvent(a, 1);
track.add(noteOn);

ShortMessage b = new ShortMessage();
b.setMessage(128, 1, note, 100);
MidiEvent noteOff = new MidiEvent(b, 16);
track.add(noteOff);
```

Ce qu'il faut pour chaque événement :

1. Créer une instance de message

```
ShortMessage first = new ShortMessage();
```

2. Appeler setMessage() avec les instructions

```
first.setMessage(192, 1, instrument, 0)
```

3. Créer une instance de MidiEvent pour le message

```
MidiEvent noteOn = new MidiEvent(first, 1);
```

4. Ajouter l'événement à la piste

```
track.add(noteOn);
```

Construisons une méthode statique utilitaire qui crée un message et retourne un MidiEvent

Les quatre arguments du message.

Le "tick" indiquant QUAND ce message doit arriver.

```
public static MidiEvent makeEvent(int comd, int chan, int one, int two, int tick) {
  MidiEvent event = null;
  try {
     ShortMessage a = new ShortMessage();
     a.setMessage(comd, chan, one, two);
     event = new MidiEvent(a, tick);

  }catch(Exception e) { }
  return event;
}
```

Oh ! Une méthode avec cinq paramètres.

Créer le message et l'événement en utilisant les paramètres de la méthode.

Retourner l'événement (un MidiEvent chargé avec le message).

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Exemple: comment utiliser la nouvelle méthode statique makeEvent()

Pas de gestion des événements ici, ni de graphismes, juste une séquence de quinze notes qui montent une gamme. L'objectif de ce code est d'apprendre à utiliser notre nouvelle méthode makeEvent(). Le code des deux prochaines versions est considérablement réduit, simplement grâce à cette méthode.

```
import javax.sound.midi.*;

public class MiniMusicPlayer1 {

   public static void main(String[] args) {

      try {

         Sequencer sequencer = MidiSystem.getSequencer();
         sequencer.open();


         Sequence seq = new Sequence(Sequence.PPQ, 4);
         Track track = seq.createTrack();

      for (int i = 5; i < 61; i+= 4) {

         track.add(makeEvent(144,1,i,100,i));
         track.add(makeEvent(128,1,i,100,i + 2));

       } // fin de la boucle

         sequencer.setSequence(seq);
         sequencer.setTempoInBPM(220);
         sequencer.start();
      } catch (Exception ex) {ex.printStackTrace();}
  } // fin de la méthode main()


   public static MidiEvent makeEvent(int comd, int chan, int one, int two, int tick) {
         MidiEvent event = null;
         try {
           ShortMessage a = new ShortMessage();
           a.setMessage(comd, chan, one, two);
          event = new MidiEvent(a, tick);

         }catch(Exception e) { }
        return event;
      }
} // fin de la classe
```

Ne pas oublier d'importer le package.

Créer (et ouvrir) le séquenceur.

Créer une séquence et une piste.

Créer un groupe d'événements pour monter la gamme (de la note 5 à la note 61 sur le piano).

Appeler notre nouvelle méthode makeEvent() pour créer message et événement, puis ajouter le résultat (le MidiEvent retourné par makeEvent()) à la piste. Ce sont les paires NOTE ON (144) et NOTE OFF (128).

Lancer l'exécution.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Version deux : enregistrer et obtenir les ControllerEvents

```
import javax.sound.midi.*;
public class MiniMusicPlayer2 implements ControllerEventListener {

    public static void main(String[] args) {
        MiniMusicPlayer2 mini = new MiniMusicPlayer2();
        mini.go();
    }
  public void go() {

     try {
        Sequencer sequenceur = MidiSystem.getSequencer();
        sequencer.open();

        int[] mesEvenements = {127};
        sequenceur.addControllerEventListener(this, mesEvenements);

        Sequence seq = new Sequence(Sequence.PPQ, 4);
        Track piste = seq.createTrack();

        for (int i = 5; i < 60; i+= 4) {
           track.add(makeEvent(144,1,i,100,i));

           track.add(makeEvent(176,1,127,0,i));

           track.add(makeEvent(128,1,i,100,i + 2));
        } // fin de la boucle

        sequenceur.setSequence(seq);
        sequenceur.setTempoInBPM(220);
        sequenceur.start();
     } catch (Exception ex) {ex.printStackTrace();}
  } // fin de la méthode


   public void controlChange(ShortMessage event) {
        System.out.println("la");
   }

  public MidiEvent makeEvent(int comd, int chan, int one, int two, int tick) {
         MidiEvent event = null;
         try {
           ShortMessage a = new ShortMessage();
           a.setMessage(comd, chan, one, two);
           event = new MidiEvent(a, tick);

         }catch(Exception e) { }
        return event;
      }
} // fin de la classe
```

Nous devons écouter les ControllerEvents, donc nous implémentons l'interface auditeur.

S'enregistrer auprès du séquenceur. La méthode d'enregistrement accepte l'auditeur ET un tableau d'entiers représentant la liste d'événements voulus. Ici, nous n'en voulons qu'un, le N° 127.

Voilà comment nous suivons le rythme. Nous insérons notre PROPRE ControllerEvent (176 indique que le type de l'événement est ControllerEvent) avec un argument pour le numéro d'événement, 127. Cet événement ne fera RIEN! Il n'est là QUE pour que nous ayons un événement chaque fois qu'une note est jouée. Autrement dit, sa seule raison d'être est qu'un événement se déclenche que NOUS puissions écouter (nous ne pouvons pas écouter NOTE_ON/OFF). Notez que cet événement a lieu sur le MÊME temps que NOTE ON. Quand NOTE ON a lieu, nous le savons parce que NOTRE événement se déclenche en même temps.

Le gestionnaire d'événements (de l'interface ControllerEvent). À chaque événement, nous affichons "la" sur la ligne de commande.

Le code qui diffère des précédentes versions est sur fond gris. (Et cette fois il ne s'exécute pas entièrement dans main().)

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Version trois : dessiner au rythme de la musique

La version finale part de la version deux et ajoute les parties IHM. Nous construisons un cadre, ajoutons un panneau, et, chaque fois que nous recevons un événement, nous dessinons un nouveau rectangle et rafraîchissons l'écran. Le seul autre changement par rapport à la version deux est que les notes sont jouées au hasard au lieu de monter la gamme.

En termes de code, le changement le plus important (outre la construction de l'IHM) est que le panneau implémente ControllerEventListener et non le programme lui-même. En conséquence, quand le panneau (une classe interne) reçoit l'événement, il sait qu'il faut dessiner un rectangle.

La version complète se trouve page suivante.

La classe interne pour le panneau :

Le panneau est un auditeur.

```
class MonPanneau extends JPanel implements ControllerEventListener {

     boolean msg = false;

     public void controlChange(ShortMessage event) {
        msg = true;
        repaint();
     }

     public void paintComponent(Graphics g) {
      if (msg) {

        Graphics2D g2 = (Graphics2D) g;

        int r = (int) (Math.random() * 250);
        int gr = (int) (Math.random() * 250);
        int b = (int) (Math.random() * 250);

        g.setColor(new Color(r,gr,b));

        int ht = (int) ((Math.random() * 120) + 10);
        int width = (int) ((Math.random() * 120) + 10);
        int x = (int) ((Math.random() * 40) + 10);
        int y = (int) ((Math.random() * 40) + 10);
        g.fillRect(x,y,ht, width);
        msg = false;

      } // fin du if
    } // fin de la méthode
  }  // fin de la classe interne
```

Nous positionnons un indicateur à faux, et il ne sera vrai que si nous recevons un événement.

Nous avons un événement. Nous positionnons l'indicateur à vrai et nous appelons repaint().

Il nous faut un indicateur, parce que d'AUTRES événements pourraient déclencher repaint() et nous ne voulons dessiner QUE si c'est un ControllerEvent.

Le reste est du code qui génère une couleur aléatoire et dessine un rectangle semi-aléatoire.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

À vos crayons

Voici le listing complet de la Version Trois. Il est directement basé sur la Version Deux. Essayez de l'annoter vous-même, sans regarder les pages précédentes.

```
import javax.sound.midi.*;
import java.io.*;
import javax.swing.*;
import java.awt.*;

public class MiniMusicPlayer3 {

     static JFrame f = new JFrame("Mon nouveau clip vidéo");
     static MonPanneau ml;

     public static void main(String[] args) {
           MiniMusicPlayer3 mini = new MiniMusicPlayer3();
           mini.go();
     } // fin de la méthode


     public  void setUpGui() {
        ml = new MonPanneau();
        f.setContentPane(ml);
        f.setBounds(30,30, 300,300);
        f.setVisible(true);
    } // fin de la méthode

    public void go() {
       setUpGui();

       try {

          Sequencer sequenceur = MidiSystem.getSequencer();
          sequenceur.open();
          sequenceur.addControllerEventListener(ml, new int[] {127});
          Sequence seq = new Sequence(Sequence.PPQ, 4);
          Track piste= seq.createTrack();

          int r = 0;
          for (int i = 0; i < 60; i+= 4) {

             r = (int) ((Math.random() * 50) + 1);
             track.add(makeEvent(144,1,r,100,i));
             track.add(makeEvent(176,1,127,0,i));
             track.add(makeEvent(128,1,r,100,i + 2));
          } // end loop

          sequenceur.setSequence(seq);
          sequenceur.start();
          sequenceur.setTempoInBPM(120);
       } catch (Exception ex) {ex.printStackTrace();}
    } // fin de la méthode
```


Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

```
  public MidiEvent makeEvent(int comd, int chan, int one, int two, int tick) {
      MidiEvent event = null;
      try {
         ShortMessage a = new ShortMessage();
         a.setMessage(comd, chan, one, two);
          event = new MidiEvent(a, tick);

       }catch(Exception e) { }
       return event;
    } // fin de la méthode



 class MonPanneau extends JPanel implements ControllerEventListener {
      boolean msg = false;

      public void controlChange(ShortMessage event) {
         msg = true;
         repaint();
      }

      public void paintComponent(Graphics g) {
       if (msg) {

         Graphics2D g2 = (Graphics2D) g;

         int r = (int) (Math.random() * 250);
         int gr = (int) (Math.random() * 250);
         int b = (int) (Math.random() * 250);

         g.setColor(new Color(r,gr,b));

         int ht = (int) ((Math.random() * 120) + 10);
         int width = (int) ((Math.random() * 120) + 10);

         int x = (int) ((Math.random() * 40) + 10);
         int y = (int) ((Math.random() * 40) + 10);

         g.fillRect(x,y,ht, width);
         msg = false;

       } // fin du if
     } // fin de la méthode
   }  // fin de la classe interne

} // fin de la classe
```


Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Un groupe de gros bonnets Java en costume joue à « Qui suis-je ». Ils vous fournissent un indice et vous devez deviner qui ils sont. Nous supposerons qu'ils disent toujours la vérité. S'ils énoncent quelque chose qui pourrait être vrai de plusieurs d'entre eux, mentionnez tous ceux à qui la phrase s'applique. Remplissez les blancs jouxtant la phrase avec les noms d'un ou plusieurs participants.

Les participants de ce soir :

Il peut s'agir de n'importe laquelle des charmantes personnalités mentionnées dans ce chapitre !

Je tiens l'IHM dans mes mains. _____

Tout type d'événement en a une. _____

La méthode clé de l'auditeur. _____

Définit la taille du JFrame. _____

Vous lui ajoutez du code sans jamais l'appeler. _____

Quand l'utilisateur fait quelque chose, c'est un _____. _____

La plupart sont des sources. _____

Je renvoie des données à l'auditeur. _____

Une méthode addXxxListener( ) dit qu'un objet est une _____. _____

Comment un auditeur s'enregistre. _____

Méthode qui contient tout le code graphique. _____

Je suis généralement lié à une instance. _____

Le "g" de (Graphics g) est en réalité de ce type. _____

La méthode qui relance paintComponent( ). _____

Le package où résident les composants Swing. _____

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Vous ÊTES le compilateur

Le code Java de cette page représente un fichier source complet. Vous jouez le rôle du compilateur et vous devez déterminer s'il se compile ou non. Si oui, que fait-il ? Sinon, comment le corriger ?

```
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;

class BoutonInterne {

  JFrame cadre;
  JButton b;

  public static void main(String [] args) {
    BoutonInterne ihm = new BoutonInterne();
    ihm.go();
  }

  public void go() {
    cadre = new JFrame();
    cadre.setDefaultCloseOperation(
                  JFrame.EXIT_ON_CLOSE);

    b = new JButton("A");
    b.addActionListener();

    cadre.getContentPane().add(
                  BorderLayout.SOUTH, b);
    cadre.setSize(200,100);
    cadre.setVisible(true);
  }

  class BListener extends ActionListener {
    public void actionPerformed(ActionEvent e) {
      if (b.getText().equals("A")) {
        b.setText("B");
      } else {
        b.setText("A");
      }
    }
  }
}
```

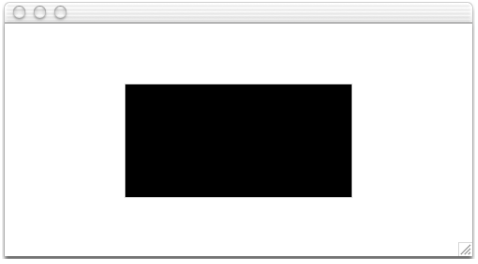

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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

L'étonnant rectangle bleu qui rétrécit.
Ce programme produit un rectangle bleu qui rétrécit, rétrécit… et finit par disparaître dans une étendue blanche.



```
import javax.swing.*;
import java.awt.*;
public class Animation {
    int x = 1;
    int y = 1;
    public static void main (String[] args) {
        Animation ihm = new Animation ();
        ihm.go();
    }
    public void go() {
        JFrame _________ = new JFrame();
        cadre.setDefaultCloseOperation(
                    JFrame.EXIT_ON_CLOSE);
        ____________________________________;
        _________.getContentPane().add(mP);
        ___________________________;
        _________.setVisible(true);
        for (int i=0; i<124; ______________) {
        ____________________;
        ____________________;
        try {
            Thread.sleep(50);
        } catch(Exception ex) { }
        }
    }
    class MonPanneau extends JPanel {
        public void paintComponent
                (Graphics _________) {
            ____________________________________;
            ____________________________________;
            ____________________________________;
            ____________________________________;
        }
    }
}
```

Note : Vous pouvez utiliser chaque fragment plus d'une fois !

g.fillRect(x,y,x-500,y-250)
g.fillRect(x,y,500-x*2,250-y*2)
g.fillRect(500-x*2,250-y*2,x,y)
g.fillRect(0,0,250,500)
g.fillRect(0,0,500,250)

x++
y++

i++
i++, y++
i++, y++, x++

Animation cadre = new Animation()
MonPanneau mP = new MonPanneau()
ContentPane mP = new ContentPane()

g.setColor(blue)
g.setColor(white)
g.setColor(Color.blue)
g.setColor(Color.white)

g
draw
cadre
panel

mP.paint()
dessin.repaint()
mP.repaint()

mP.setSize(500,270)
cadre.setSize(500,270)
panneau.setSize(500,270)

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Solutions des exercices

Qui suis-je ?

Je tiens l'IHM dans mes mains.	JFrame
Tout type d'événement en a une.	interface auditeur
La méthode clé de l'auditeur.	actionPerformed( )
Définit la taille du JFrame.	setSize( )
Vous lui ajoutez du code sans jamais l'appeler.	paintComponent( )
Quand l'utilisateur fait quelque chose, c'est un ____.	événement
Beaucoup sont des sources d'événements.	composants Swing
Je renvoie des données à l'auditeur.	objet event
Une méthode addXxxListener( ) dit qu'un objet est une ___.	source d'événement
Comment un auditeur s'enregistre.	addActionListener( )
La méthode qui contient tout le code graphique.	paintComponent( )
Je suis lié à une autre instance.	objet interne
Le «g» de (Graphics g), est en réalité de cet type.	Graphics2D
La méthode qui lance paintComponent( ).	repaint( )
Le package où résident les composants Swing.	javax.swing

Vous ÊTES le compilateur

Une fois ce code débogué, il crée une IHM avec un bouton qui bascule entre A et B quand on le clique.

```
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;

class BoutonInterne {

  JFrame cadre;
  JButton b;

  public static void main(String [] args) {
    BoutonInterne ihm = new BoutonInterne();
    ihm.go();
  }

  public void go() {
    cadre = new JFrame();
    cadre.setDefaultCloseOperation(
                    JFrame.EXIT_ON_CLOSE);
```

La méthode addActionListener( ) accepte une classe qui implémente l'interface ActionListener.

```
    b = new JButton("A");
    b.addActionListener( new BListener( ) );

    cadre.getContentPane().add(
                    BorderLayout.SOUTH, b);
    cadre.setSize(200,100);
    cadre.setVisible(true);
  }

  class BListener implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      if (b.getText().equals("A")) {
        b.setText("B");
      } else {
        b.setText("A");
      }
    }
  }
}
```

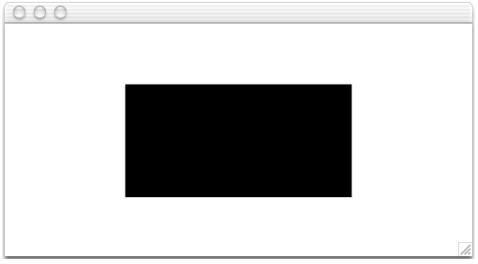
ActionListener est une interface. Les interfaces sont implémentées, non étendues.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

La piscine

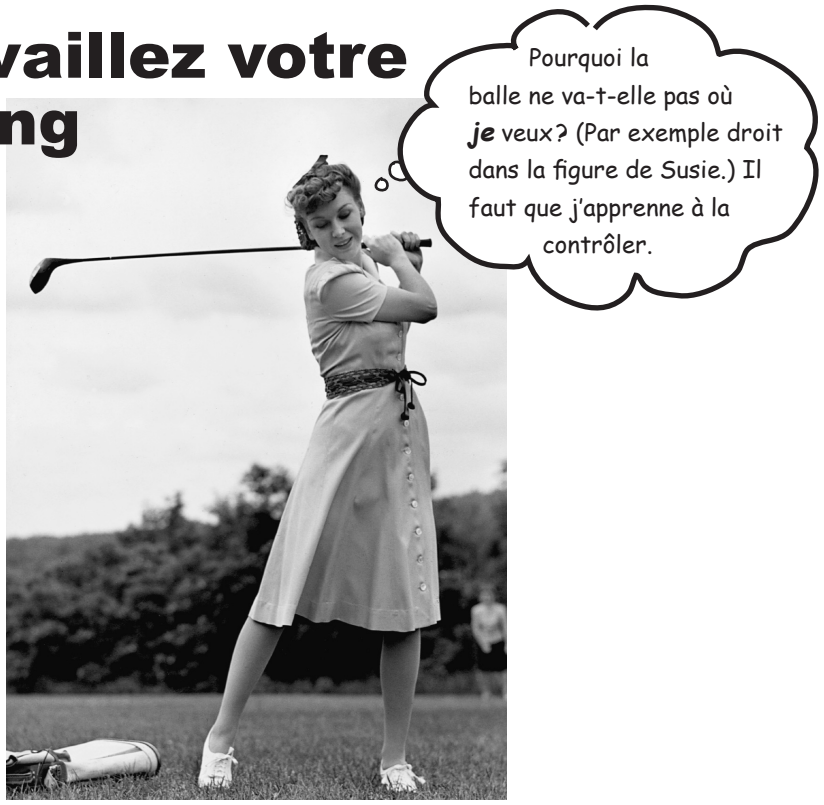
```
import javax.swing.*;
import java.awt.*;
public class Animation {
    int x = 1;
    int y = 1;
    public static void main (String[] args) {
       Animation ihm = new Animation ();
       ihm.go();
   }
   public void go() {
       JFrame cadre = new JFrame();
       cadre.setDefaultCloseOperation(
                    JFrame.EXIT_ON_CLOSE);
       MonPanneau mP = new MonPanneau();
       cadre.getContentPane().add(mP);
       cadre.setSize(500,270);
       cadre.setVisible(true);
       for (int i = 0; i < 124; i++,x++,y++ ) {
          x++;
          mP.repaint();
          try {
            Thread.sleep(50);
          } catch(Exception ex) { }
       }
   }
   class MonPanneau extends JPanel {
       public void paintComponent(Graphics g ) {
          g.setColor(Color.white);
          g.fillRect(0,0,500,250);
          g.setColor(Color.blue);
          g.fillRect(x,y,500-x*2,250-y*2);
       }
   }
}
```

L'étonnant rectangle bleu qui rétrécit



Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Travaillez votre Swing



Swing est facile. Tant que vous n'attachez *pas* d'importance à la façon dont les objets se placent à l'écran. Le code Swing *a l'air* facile, mais vous le compilez, vous l'exécutez et vous pensez « Hé, ce n'est pas *ça* qui est censé aller *ici* ». Ce qui *facilite* le *codage* est ce qui rend le *contrôle difficile* — les **gestionnaires d'agencement**. Les objets gestionnaires d'agencement contrôlent la taille et l'emplacement des widgets dans une interface graphique. Ils font une tonne de travail à votre place, mais vous n'aimerez pas toujours le résultat. Vous voulez deux boutons de la même taille, mais ce n'est pas le cas. Vous voulez un champ de texte de huit centimètres et il en fait vingt-deux. Ou trois. Et il est *sous* le bouton au lieu d'être *à côté*. Mais, avec un peu de patience, vous pouvez amener les gestionnaires d'agencement à se soumettre à votre volonté. Dans ce chapitre, nous allons travailler notre Swing. Outre les gestionnaires d'agencement, nous approfondirons les widgets. Nous en créerons, nous en afficherons (où *nous* voulons) et nous les utiliserons dans un programme. Ça ne s'annonce pas très bien pour Susie.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Les composants Swing

Composant est le terme correct pour ce que nous avons appelé un *widget*. Les éléments que vous placez dans une IHM. *Ceux que l'utilisateur voit et avec lesquels il interagit*. Champs de texte, listes déroulantes, boutons radio, etc., sont tous des composants et dérivent de **`javax.swing.JComponent`**.

Techniquement, un widget est un composant Swing. Presque tous les éléments d'une IHM dérivent de javax.swing.JComponent.

Les composants peuvent être imbriqués

Avec Swing, virtuellement tous les composants sont capables de contenir d'autres composants. Autrement dit, vous pouvez placer pratiquement *n'importe quoi sur n'importe quoi*. Mais, la plupart du temps, *vous ajouterez des* composants *interactifs* comme des boutons et des listes *à des composants d'arrière-plan* comme des cadres ou des panneaux. Même s'il est *possible* d'insérer par exemple un panneau dans un bouton, c'est une idée plutôt bizarre qui ne vous fera pas obtenir un prix d'ergonomie.

À l'exception de JFrame, la distinction entre composants *interactifs* et composants d'arrière-plan est artificielle. On utilise généralement un JPanel comme fond pour grouper d'autres composants, mais même un JPanel peut être interactif. Et vous pouvez utiliser ses événements, notamment les clics de souris et les appuis sur les touches.

Les quatre étapes de la création d'une IHM (révision)

1. Créer une fenêtre (un JFrame)

```
JFrame cadre = new JFrame();
```

2. Créer un composant (bouton, champ de texte, etc.)

```
JButton bouton = new JButton("cliquez-moi");
```

3. Ajouter le composant au cadre

```
cadre.getContentPane().add(BorderLayout.EAST, bouton);
```

4. L'afficher (lui donner une taille et le rendre visible)

```
cadre.setSize(300,300);
cadre.setVisible(true);
```

Placer des composants interactifs :

Dans des composants d'arrière-plan :

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Gestionnaires d'agencement

Un gestionnaire d'agencement est un objet Java associé à un composant particulier, presque toujours un composant d'*arrière-plan*. Il contrôle les composants contenus dans celui auquel il est associé. Autrement dit, si un cadre contient un panneau et que le panneau contient un bouton, le gestionnaire d'agencement du panneau contrôle la taille et le placement du bouton, tandis que celui du cadre contrôle la taille et le placement du panneau. En revanche, le bouton n'a pas besoin de gestionnaire d'agencement, parce qu'il ne contient rien d'autre.

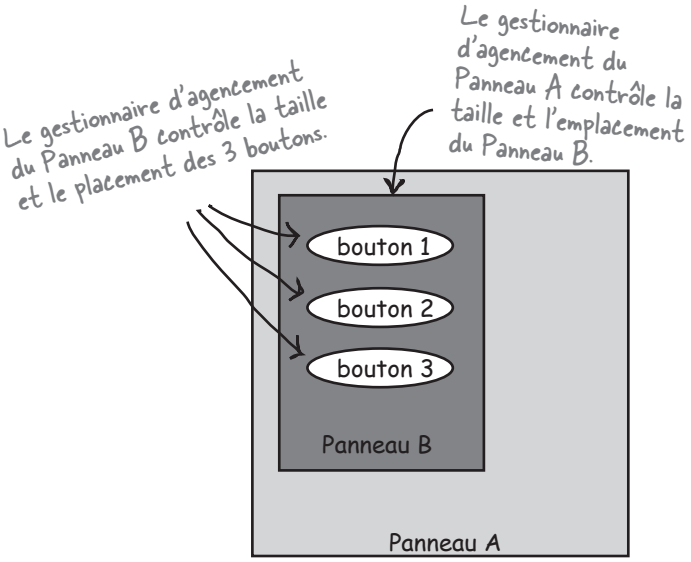
Si un panneau contient cinq composants, même si ces derniers ont leur propre gestionnaire d'agencement, c'est celui du panneau qui contrôle leur taille et leur placement. Si ces cinq éléments en contiennent d'*autres* à leur tour, ces *autres* composants sont placés selon le gestionnaire d'agencement de celui qui les contient.

Et nous parlons bien de *contenir*, comme dans « un panneau contient un bouton », parce que le bouton a été *ajouté* au panneau avec une instruction du genre :

```
monPanneau.add(bouton);
```

Il existe plusieurs sortes de gestionnaires d'agencement et chaque composant d'arrière-plan peut avoir le sien. Ils ont leurs propres politiques qu'il s'agit d'appliquer quand on construit une maquette. Par exemple, un gestionnaire d'agencement peut exiger que tous les composants soient de la même taille, alignés sur une grille, tandis qu'un autre laissera les composants choisir leur taille mais les empilera verticalement. Voici un exemple d'agencements imbriqués :

```
JPanel panneauA = new JPanel();
JPanel panneauB = new JPanel();
panneauB.add(new JButton("bouton 1"));
panneauB.add(new JButton("bouton 2"));
panneauB.add(new JButton("bouton 3"));
panneauA.add(panneauB);
```



Le gestionnaire d'agencement du panneau A n'a RIEN à dire sur les trois boutons. La hiérarchie n'a qu'un seul niveau. Le gestionnaire d'agencement de A ne contrôle que ce qui est ajouté directement à A, mais pas les composants imbriqués.

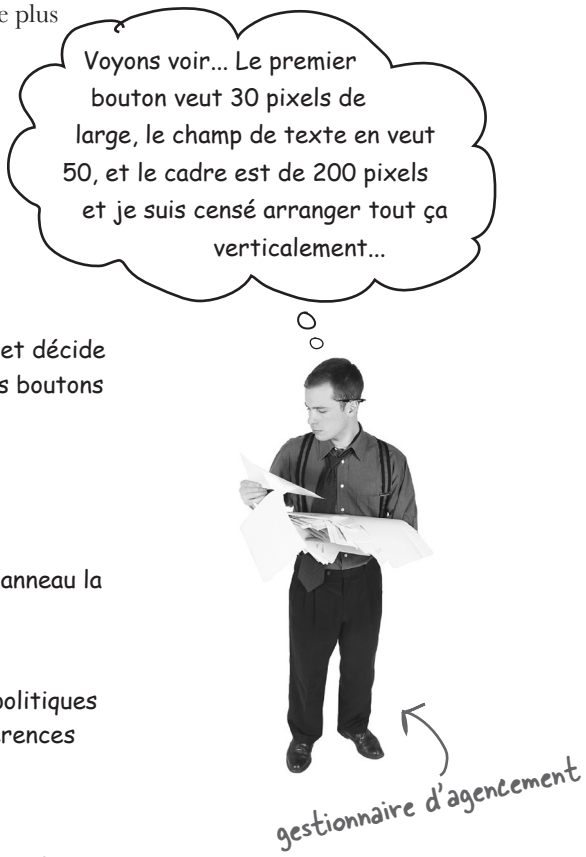
Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Comment le gestionnaire d'agencement décide-t-il ?

Différents gestionnaires d'agencement appliquent des politiques différentes pour organiser les composants (les aligner sur une grille, uniformiser leur taille, les empiler verticalement, etc.) Mais les composants agencés ont au moins un petit mot à dire. En général, le processus d'agencement d'un composant d'arrière-plan se déroule plus ou moins comme ceci :

Un scénario d'agencement :

1. On crée un panneau et on lui ajoute trois boutons.
2. Le gestionnaire d'agencement du panneau demande à chaque bouton la taille qu'il souhaite avoir.
3. Le gestionnaire d'agencement applique ses politiques et décide s'il doit respecter tout ou partie des préférences des boutons ou les ignorer.
4. On ajoute le panneau à un cadre.
5. Le gestionnaire d'agencement du cadre demande au panneau la taille qu'il préfère avoir.
6. Le gestionnaire d'agencement du cadre applique ses politiques et décide s'il doit respecter tout ou partie des préférences du panneau ou les ignorer.



Différents gestionnaires d'agencement ont différentes politiques

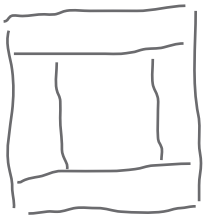
Certains gestionnaires d'agencement respectent les volontés des composants. Si le bouton veut une taille de 30 pixels sur 50, le gestionnaire la lui alloue. D'autres ne respectent qu'une partie des préférences des composants. Le bouton peut bien vouloir 30 pixels sur 50, il aura 30 pixels sur un nombre de pixels égal à la largeur du panneau d'arrière-plan. D'autres encore n'appliquent les préférences que du plus grand des composants et uniformise la taille des autres. Dans certains cas, le travail du gestionnaire d'agencement peut devenir très complexe. Mais, la plupart du temps, il est assez facile de deviner ce qu'il va probablement faire, une fois qu'on connaît ses politiques.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Les trois grands gestionnaires d'agencement : bordures, flots et boîtes.

BorderLayout

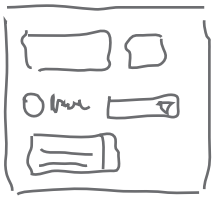
Un gestionnaire BorderLayout divise un composant d'arrière-plan en cinq régions. Vous ne pouvez ajouter qu'un composant par région à un fond contrôlé par BorderLayout. Les composants agencés par ce gestionnaire n'obtiennent généralement pas la taille voulue. **BorderLayout est le gestionnaire d'agencement par défaut pour un cadre !**



Un composant par région.

FlowLayout

Un gestionnaire FlowLayout se comporte un peu comme un traitement de texte où les mots seraient remplacés par des composants. La taille des composants est respectée, et ils sont agencés de gauche à droite, avec le « retour à la ligne » activé. Quand un composant ne tient pas dans le sens horizontal, il passe à la « ligne » suivante. **FlowLayout est le gestionnaire par défaut des panneaux !**



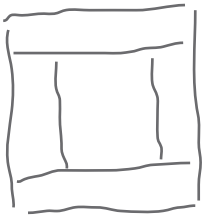
Les composants sont ajoutés de gauche à droite, et passent à la ligne si nécessaire.

BoxLayout

Un gestionnaire BoxLayout ressemble à FlowLayout : chaque composant a sa propre taille et ils sont placés dans l'ordre où ils ont été ajoutés. Mais, contrairement à FlowLayout, BoxLayout peut empiler les composants verticalement (ou les aligner horizontalement, ce qui est généralement inutile). Au lieu d'avoir un « retour à la ligne » automatique, vous pouvez insérer une sorte de « retour chariot » et **forcer** les composants à commencer sur une nouvelle ligne.

Les composants sont ajoutés de haut en bas, un par ligne.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>



BorderLayout connaît cinq régions : east, west, north, south et center

Ajoutons un bouton à la région EAST :

```
import javax.swing.*;
import java.awt.*;

public class Bouton1 {

    public static void main (String[] args) {
        Bouton1 ihm = new Bouton1();
        ihm.go();
    }

    public void go() {
        JFrame cadre = new JFrame();
        JButton bouton = new JButton("cliquez-moi");
        cadre.getContentPane().add(BorderLayout.EAST, bouton);
        cadre.setSize(200,200);
        cadre.setVisible(true);
    }
}
```

BorderLayout est dans le package java.awt.

Spécifier la région.

Comment BorderLayout a-t-il calculé cette taille de bouton ?

Quels sont les facteurs que le gestionnaire d'agencement a dû prendre en compte ?

Pourquoi le bouton n'est-il pas plus haut ou plus large ?

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Regardez ce qui se passe quand nous ajoutons des caractères au bouton...

Nous n'avons changé que le texte du bouton.

```
public void go() {
    JFrame cadre = new JFrame();
    JButton bouton = new JButton("cliquez avec modération");
    cadre.getContentPane().add(BorderLayout.EAST, bouton);
    cadre.setSize(200,200);
    cadre.setVisible(true);
}
```

Ce bouton a la hauteur voulue, mais non la largeur.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Essayons un bouton dans la région NORTH

```
public void go() {
    JFrame cadre = new JFrame();
    JButton bouton = new JButton("Je pense, donc je suis...");
    cadre.getContentPane().add(BorderLayout.NORTH, bouton);
    cadre.setSize(200,200);
    cadre.setVisible(true);
}
```

Le bouton a la hauteur qu'il veut avoir, mais il a la largeur du cadre.

Maintenant, essayons d'agrandir le bouton en hauteur

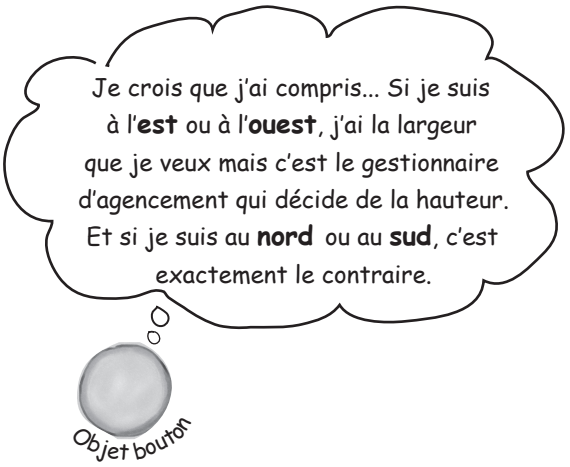
Comment procéder? le bouton est déjà aussi large que possible — il a la taille du cadre. Mais nous pouvons essayer d'augmenter sa hauteur en agrandissant la police.

```
public void go() {
    JFrame cadre = new JFrame();
    JButton bouton = new JButton("Cliquez ici!");
    Font bigFont = new Font("serif", Font.BOLD, 28);
    bouton.setFont(bigFont);
    cadre.getContentPane().add(BorderLayout.NORTH, bouton);
    cadre.setSize(200,200);
    cadre.setVisible(true);
}
```

Une police plus grosse forcera le cadre à allouer plus d'espace pour la hauteur du bouton.

La largeur reste la même, mais le bouton est plus haut. La région NORTH s'est allongée pour s'adapter à la nouvelle hauteur.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>



Mais que se passe-t-il au centre ?

La région centre obtient ce qui reste !

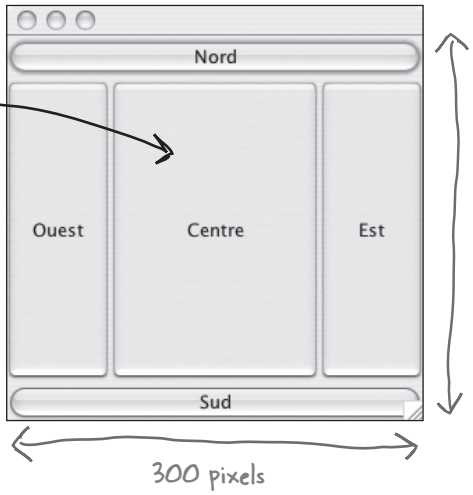
(Sauf dans un cas particulier que nous verrons plus tard)

```
public void go() {
    JFrame cadre = new JFrame();

    JButton east = new JButton("Est");
    JButton west = new JButton("Ouest");
    JButton north = new JButton("Nord");
    JButton south = new JButton("Sud");
    JButton center = new JButton("Centre");

    cadre.getContentPane().add(BorderLayout.EAST, east);
    cadre.getContentPane().add(BorderLayout.WEST, west);
    cadre.getContentPane().add(BorderLayout.NORTH, north);
    cadre.getContentPane().add(BorderLayout.SOUTH, south);
    cadre.getContentPane().add(BorderLayout.CENTER, center);

    cadre.setSize(300,300);
    cadre.setVisible(true);
}
```

Les composants du centre obtiennent ce qui reste, selon les dimensions du cadre (300 x 300 dans ce code).

Ceux de l'est et de l'ouest ont la largeur qu'ils demandent

Ceux du nord et du sud ont la hauteur qu'ils demandent.



Quand vous placez des composants au nord ou au sud, ils occupent toute la largeur du cadre. Ceux que vous placerez à l'est et à l'ouest ne seront pas aussi haut que si le nord et le sud étaient vides.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

FlowLayout s'occupe du flot des composants :

de gauche à droite, de haut en bas, dans l'ordre où ils ont été ajoutés.

Ajoutons un panneau à la région EAST :

Le gestionnaire d'agencement par défaut de JPanel est FlowLayout. Quand nous ajoutons un panneau à un cadre, la taille et le placement du panneau sont toujours contrôlés par BorderLayout. Mais tout ce qui est *dans* le *panneau* (autrement dit les composants ajoutés au panneau par l'appel de **panneau.add(unComposant)**) sont contrôlés par FlowLayout. Nous commencerons par insérer un panneau vide dans la région est du cadre, puis, dans les pages suivantes, nous ajouterons des composants au panneau.

Comme ce panneau ne contient rien, il ne demande pas beaucoup de largeur dans la région EAST.

```
import javax.swing.*;
import java.awt.*;

public class Panneau1 {

    public static void main (String[] args) {
        Panneau1 ihm = new Panneau1();
        ihm.go();
    }


    public void go() {
        JFrame cadre = new JFrame();
        JPanel panneau = new JPanel();
        panneau.setBackground(Color.darkGray);
        cadre.getContentPane().add(BorderLayout.EAST, panneau);
        cadre.setSize(200,200);
        cadre.setVisible(true);
    }
}
```

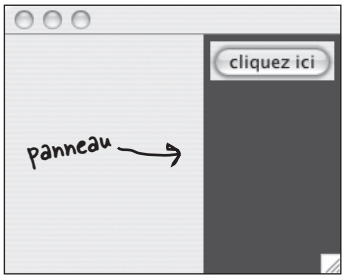
Choisir un gris foncé, pour voir où est le panneau dans le cadre.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Ajoutons un bouton au panneau

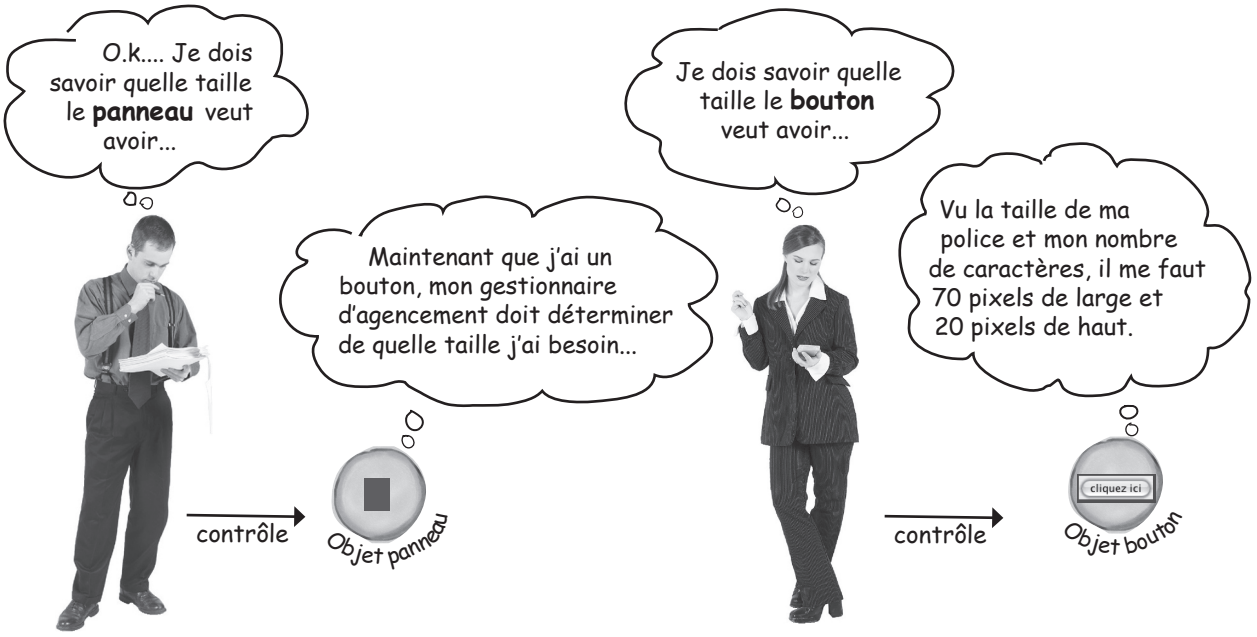
```
public void go() {
     JFrame cadre = new JFrame();
     JPanel panneau = new JPanel();
     panneau.setBackground(Color.darkGray);

     JButton bouton = new JButton("cliquez ici");

     panneau.add(bouton);
     cadre.getContentPane().add(BorderLayout.EAST, panneau);

     cadre.setSize(250,200);
     cadre.setVisible(true);
}
```

Ajouter le bouton au panneau et le panneau au cadre. Le gestionnaire d'agencement du panneau (FlowLayout) contrôle le bouton, et celui du cadre (BorderLayout) contrôle le panneau.



Le panneau s'est agrandi!

Et le bouton a la taille voulue dans les deux dimensions, parce que le panneau utilise FlowLayout, et que le bouton fait partie du panneau (pas du cadre).



Le gestionnaire d'agencement du cadre : **BorderLayout**

Le gestionnaire d'agencement du panneau : **FlowLayout**

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

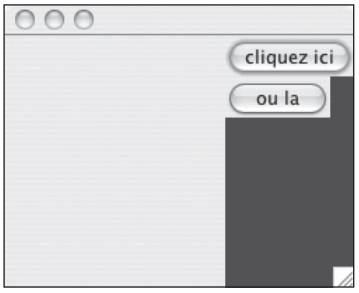
Que se passe-t-il si nous ajoutons DEUX boutons au panneau ?

```
public void go() {
    JFrame cadre = new JFrame();
    JPanel panneau = new JPanel();
    panneau.setBackground(Color.darkGray);

    JButton bouton = new JButton("cliquez ici");
    JButton boutonDeux = new JButton("ou la");

    panneau.add(bouton);
    panneau.add(boutonDeux);

    cadre.getContentPane().add(BorderLayout.EAST, panneau);
    cadre.setSize(250,200);
    cadre.setVisible(true);
}
```

Créer DEUX boutons.

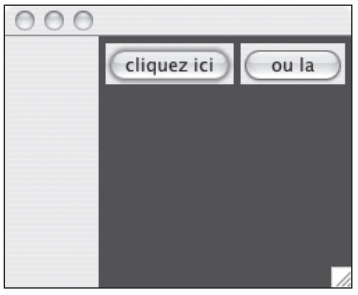
Ajouter les DEUX au panneau.

Ce que nous voulions :



Nous voulions empiler les boutons l'un au-dessus de l'autre.

Ce que nous avons :



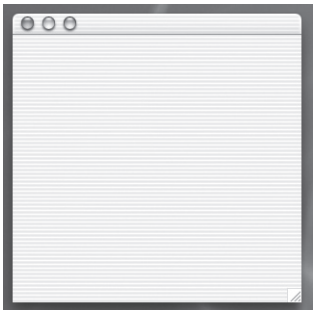
Le panneau s'est agrandi et les deux boutons sont côte à côte.

Remarquez que le bouton "ou la" est plus petit que le bouton "cliquez ici" ... C'est ainsi que FlowLayout fonctionne. Le bouton obtient ce dont il a besoin (et pas plus).

À vos crayons

Et si nous modifiions le code de la façon suivante, à quoi ressemblerait notre écran ?

```
JButton bouton = new JButton("cliquez ici");
JButton boutonDeux = new JButton("ou la");
JButton boutonTrois = new JButton("bof ?");
panneau.add(bouton);
panneau.add(boutonDeux);
panneau.add(boutonTrois);
```



Dessinez l'écran que vous pensez obtenir en exécutant le code de gauche.

(Et puis testez-le !)

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

BoxLayout à la rescousse !

Les composants restent empilés, même s'il y a assez de place pour les placer côte à côte.

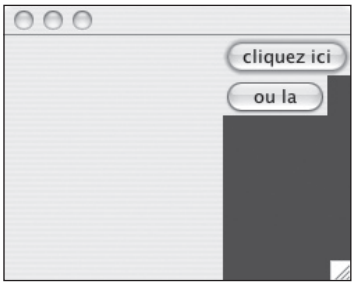
Contrairement à FlowLayout, BoxLayout peut forcer les composants à « aller à la ligne », même s'il y a assez de place pour eux horizontalement.

Mais vous devez abandonner le gestionnaire par défaut, FlowLayout, et adopter BoxLayout.

```
public void go() {
    JFrame cadre = new JFrame();
    JPanel panneau = new JPanel();
    panneau.setBackground(Color.darkGray);

    panneau.setLayout(new BoxLayout(panneau, BoxLayout.Y_AXIS));


    JButton bouton = new JButton("cliquez ici");
    JButton boutonDeux = new JButton("ou la");
    panneau.add(bouton);
    panneau.add(boutonDeux);
    cadre.getContentPane().add(BorderLayout.EAST, panneau);
    cadre.setSize(250,200);
    cadre.setVisible(true);
}
```

Le gestionnaire d'agencement est maintenant une nouvelle instance de BoxLayout.

Le constructeur de BoxLayout a besoin de savoir quel composant il agence (le panneau) et quel axe utiliser (Y_AXIS pour une pile verticale).



Remarquez que le panneau a rétréci, puisqu'il n'a plus besoin de faire de la place aux deux boutons horizontalement. Il a donc dit au cadre de calculer la taille du plus grand bouton, "cliquez ici".

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

il n'y a pas de Questions stupides

Q : Comment se fait-il qu'on ne puisse pas ajouter directement de composants à un cadre, comme on le fait pour un panneau ?

R : Un JFrame est spécial parce que c'est en quelque sorte une surface de contact. Alors que tous les autres composants Swing sont pur Java, un JFrame doit se connecter au système d'exploitation sous-jacent pour accéder à l'affichage. Considérez le panneau comme une couche 100 % Java qui réside ***au-dessus*** du JFrame. Ou bien imaginez que le JFrame est le cadre de la fenêtre et que le panneau est la... vitre. Vous voyez, le ***panneau*** de la fenêtre. Et vous pouvez même ***échanger*** le panneau avec votre propre JPanel, pour faire de votre JPanel le panneau de contenu du cadre, avec

```
monCadre.setContentPane(monPanneau);
```

Q : Est-ce que je peux changer le gestionnaire d'agencement du cadre ? Choisir FlowLayout au lieu de Border ?

R : La façon la plus simple consiste à créer un panneau, y insérer les composants de votre choix pour construire l'interface, puis à en faire le panneau de contenu du cadre avec le code précédent (au lieu d'utiliser le panneau par défaut).

Q : Et si je veux personnaliser la taille ? Est-ce qu'il y a une méthode setSize() pour les composants ?

R : Oui, il y a une méthode setSize(), mais les gestionnaires d'agencement l'ignorent. Il y a une distinction entre la *taille préférée* d'un composant et celle que vous voulez lui donner. La taille préférée est basée sur ce dont il a réellement *besoin* (il prend cette décision seul). Le gestionnaire d'agencement appelle la méthode getPreferredSize() du composant, et *cette* méthode *ne se soucie pas* de savoir si vous avez appelé setSize() auparavant.

Q : Est-ce que je peux désactiver les gestionnaires d'agencement et placer les composants où je veux ?

R : Oui Composant par composant, vous pouvez appeler **setLayout(null)** puis coder en dur les tailles et les dimensions. Mais, à la longue, il est presque toujours plus facile d'utiliser les gestionnaires d'agencement.

POINTS D'IMPACT

- Les gestionnaires d'agencement contrôlent la taille et le placement de composants imbriqués dans d'autres composants.
- Quand vous ajoutez un composant à un autre composant (appelé composant *d'arrière-plan*, sans que ce soit une distinction technique), le composant ajouté est contrôlé par le gestionnaire d'agencement du composant *d'arrière-plan*.
- Un gestionnaire d'agencement demande aux composants leur taille préférée avant de prendre une décision sur le placement. Selon sa politique, il peut respecter tout ou partie de leurs désirs ou les ignorer.
- BorderLayout permet d'ajouter un composant dans cinq régions. Vous devez spécifier la région lors de l'insertion du composant avec la syntaxe suivante : `add(BorderLayout.EAST, panneau);`
- Avec BorderLayout, les composants du nord et du sud ont leur hauteur préférée mais pas la largeur. À l'est et à l'ouest, ils ont leur largeur préférée mais pas la hauteur. Le composant du centre obtient ce qui reste (sauf si vous utilisez **`pack()`**).
- La méthode pack() utilise la taille préférée du composant du centre, puis elle détermine la taille du cadre en prenant le centre comme point de départ et construit le reste en se basant sur ce qui se trouve dans les autres régions.
- FlowLayout place les composants de gauche à droite et de haut en bas, dans l'ordre de leur insertion, et ne « va à la ligne » que lorsque les composants ne tiennent pas horizontalement.
- FlowLayout donne aux composants leur taille préférée dans les deux dimensions.
- BoxLayout permet d'empiler verticalement les composants, même s'il y a assez de place pour qu'ils soient côte à côte. Comme FlowLayout, BoxLayout utilise la taille préférée des composants dans les deux dimensions.
- BorderLayout est le gestionnaire par défaut des cadres et FlowLayout le gestionnaire par défaut des panneaux.
- Si vous voulez un autre gestionnaire d'agencement que FlowLayout, vous devez appeler **`setLayout()`** sur le panneau.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Jouons avec les composants Swing

Après avoir appris les bases des gestionnaires d'agencement, nous allons essayer quelques uns des composants les plus courants : champ de texte, zone de texte déroulante, case à cocher et liste. Nous n'allons pas vous réciter toute l'API, mais simplement fournir quelques points de départ.

JTextField

Constructeurs

```
JTextField champ = new JTextField(20);

JTextField champ = new JTextField("Votre nom");
```

20 signifie 20 colonnes, pas 20 pixels. Cette instruction définit la taille préférée du champ.

Utilisation

1. En extraire le texte

```
System.out.println(champ.getText());
```

2. Y placer du texte

```
champ.setText("abracadabra");
champ.setText("");
```

Ceci efface le contenu du champ.

3. Recevoir un ActionEvent quand l'utilisateur appuie sur la touche Entrée

```
champ.addActionListener(myActionListener);
```

Vous pouvez aussi écouter les événements de chaque touche si vous voulez vraiment être averti chaque fois que l'utilisateur appuie sur une touche.

4. Sélectionner (mettre en surbrillance) le texte du champ

```
champ.selectAll();
```

5. Replacer le curseur dans le champ (pour que l'utilisateur n'ait plus qu'à taper)

```
champ.requestFocus();
```


Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

JTextArea

Contrairement à JTextField, JTextArea peut contenir plusieurs lignes de texte. Il faut un peu de configuration pour en créer une, parce qu'elle ne sort pas toute prête de sa boîte avec des barres de défilement et un retour à la ligne. Pour rendre une JTextArea déroulante, il faut l'insérer dans un ScrollPane. Un ScrollPane est un objet qui adore défiler, et il s'occupera des besoins de la zone de texte en la matière.

Constructeur

```
JTextArea texte = new JTextArea(10,20);
```

10 signifie 10 lignes (définit la hauteur préférée)
20 signifie 20 colonnes (définit la largeur préférée)

Utilisation

① Créer uniquement un ascenseur

```
JScrollPane ascenseur = new JScrollPane(texte);
texte.setLineWrap(true);

ascenseur.setVerticalScrollBarPolicy(ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS);
ascenseur.setHorizontalScrollBarPolicy(ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER);

panneau.add(ascenseur);
```

Créer un JScrollPane lui transmettre la zone de texte qu'il fera dérouler.

Activer le retour à la ligne.

On ne défile que dans le sens vertical.

Important!! On transmet la zone de texte au JScrollPane (via son constructeur), puis on l'ajoute au panneau. On n'ajoute pas la zone de texte directement au panneau!

② Remplacer le texte de la zone

```
texte.setText("L'important, c'est la rose...");
```

③ Ajouter du texte à la zone

```
texte.append("crois-moi");
```

④ Sélectionner (mettre en surbrillance) le texte de la zone

```
texte.selectAll();
```

⑤ Replacer le curseur dans le champ (pour que l'utilisateur n'ait plus qu'à taper)

```
texte.requestFocus();
```


Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Exemple de JTextArea

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;


public class TextArea1 implements ActionListener {

    JTextArea texte;

    public static void main (String[] args) {
        TextArea1 ihm = new TextArea1();
        ihm.go();
    }


    public void go() {
        JFrame cadre = new JFrame();
        JPanel panneau = new JPanel();
        JButton bouton = new JButton("Cliquez-moi");
        bouton.addActionListener(this);
        texte = new JTextArea(10,20);
        texte.setLineWrap(true);

        JScrollPane ascenseur = new JScrollPane(texte);
        ascenseur.setVerticalScrollBarPolicy(ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS);
        ascenseur.setHorizontalScrollBarPolicy(ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER);

        panneau.add(ascenseur);

        cadre.getContentPane().add(BorderLayout.CENTER, panneau);
        cadre.getContentPane().add(BorderLayout.SOUTH, bouton);

        cadre.setSize(350,300);
        cadre.setVisible(true);
    }

    public void actionPerformed(ActionEvent ev) {
        texte.append("Clic ! \n ");
    }
}
```

Insérer un retour à la ligne après chaque clic de l'utilisateur, sinon vous obtiendrez ce résultat

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

JCheckBox

Constructeur

```
JCheckBox coche = new JCheckBox("Cochez moi");
```

Utilisation

1. Ecouter un événement (la case est cochée ou non)

```
coche.addItemListener(this);
```

2. Gérer l'événement (déterminer si elle est cochée ou non)

```
public void itemStateChanged(ItemEvent ev) {
   String ouiOuNon = "décochée";
   if (coche.isSelected()) ouiOuNon = "cochée";
   System.out.println("La coche est " + ouiOuNon);
}
```

3. La cocher ou la décocher dans le code

```
coche.setSelected(true);
coche.setSelected(false);
```

il n'y a pas de
Questions stupides

Q : Est-ce que les gestionnaires d'agencement ne nous donnent pas plus de soucis qu'ils n'en valent la peine ? Pourquoi se donner tout ce mal ? Pourquoi ne pas tout simplement coder en dur la taille et les coordonnées des composants ?

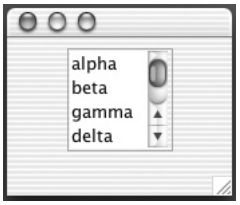
R : Obtenir exactement ce que vous voulez des gestionnaires d'agencement peut être un défi. Mais réfléchissez à tout ce qu'ils peuvent faire pour vous. Même la tâche apparemment simple qui consiste à organiser les composants à l'écran peut devenir complexe. Par exemple, le gestionnaire d'agencement s'occupe de les empêcher de se chevaucher. Autrement dit, il sait comment gérer l'espacement entre les composants (et par rapport aux bords du cadre). Naturellement, vous pouvez le faire vous-même. Mais que se passe-t-il si vous voulez qu'ils soient très serrés ? Vous pouvez bien les placer correctement à la main, mais cela risque de ne fonctionner que pour votre JVM !

Pourquoi ? Parce que les composants peuvent différer légèrement d'une plate-forme à l'autre, surtout s'ils en utilisent le « look and feel » natif. Des aspects subtils, comme l'arrondi des angles, peuvent varier de telle sorte que les composants seront parfaitement alignés sur une plate-forme, alors que sur l'autre ils vont soudain s'écraser les uns sur les autres.

Mais nous n'avons pas encore tout vu. Pensez à ce qui se passe quand l'utilisateur redimensionne la fenêtre ! Ou bien si votre interface est dynamique, avec des composants qui vont et viennent. Si vous deviez vous charger de réagencer tous les composant à chaque modification du contenu ou de l'arrière-plan... bonjour !

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

JList



Le constructeur d'une JList accepte un tableau de n'importe quel type de données, pas nécessairement String. Mais la liste affichera des chaînes de caractères.

Constructeur

```
String [] listEntries = {"alpha", "beta", "gamma", "delta",
                         "epsilon", "zeta", "eta", "theta "};

liste = new JList(listEntries);
```

Procéder comme pour JTextArea : créer un JScrollPane (et lui transmettre la liste), puis ajouter celui-ci (PAS la liste) au panneau.

Utilisation

1. Créer un ascenseur

```
JScrollPane ascenseur = new JScrollPane(liste);
ascenseur.setVerticalScrollBarPolicy(ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS);
ascenseur.setHorizontalScrollBarPolicy(ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER);

panneau.add(ascenseur);
```

2. Fixer le nombre de lignes à afficher avant de défiler

```
liste.setVisibleRowCount(4);
```

3. Empêcher l'utilisateur de sélectionner plus d'UN élément à la fois

```
liste.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
```

4. Ajouter un auditeur d'événements

```
liste.addListSelectionListener(this);
```

5. Gérer les événements (déterminer l'élément sélectionné)

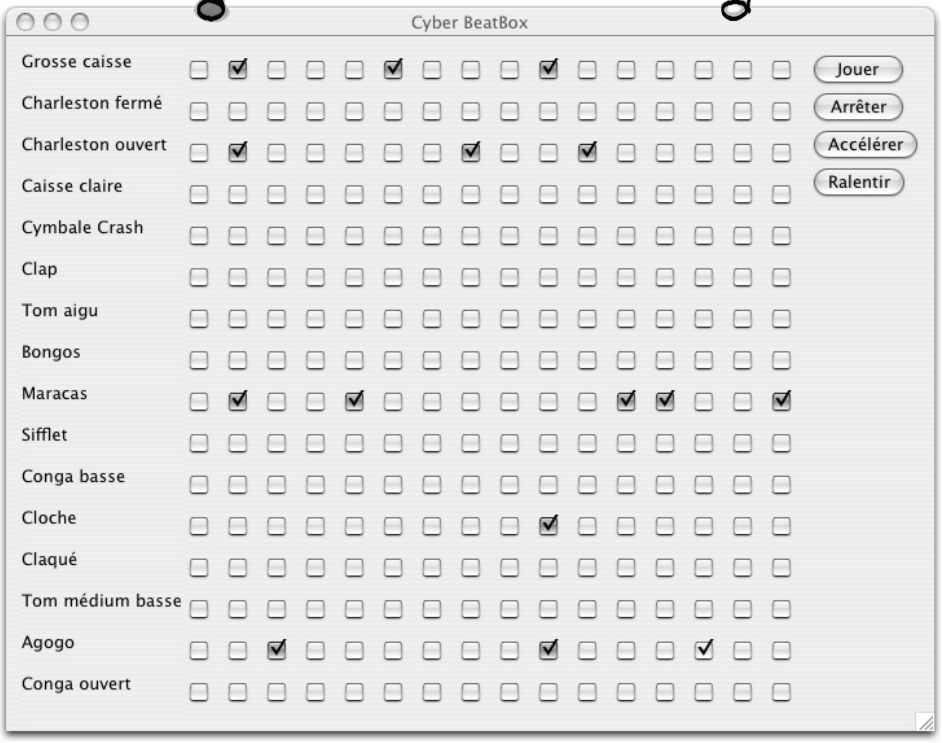
```
public void valueChanged(ListSelectionEvent lse) {
      if( !lse.getValueIsAdjusting()) {
        String selection = (String) liste.getSelectedValue();
        System.out.println(selection);
      }
   }
```

Si vous n'insérez pas ce test, vous recevrez l'événement DEUX fois.

getSelectedValue() retourne en réalité un Object. Une liste n'est pas limitée aux objets String.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Recettes de code



Cette partie est optionnelle. Nous allons créer l'interface graphique de la BeatBox. Au chapitre suivant, nous apprendrons comment sauvegarder et restaurer les motifs. Enfin, dans le chapitre consacré au réseau, nous transformerons la BeatBox en client de « discussion ».

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Créer la BeatBox

Voici le listing du code de cette version de la BeatBox, avec des boutons pour démarrer, arrêter et modifier le tempo. Il est complet et entièrement annoté, mais en voici une vue d'ensemble :

1. Construire une IHM avec 256 cases à cocher (JCheckBox) qui sont décochées au départ, 16 étiquettes (JLabel) pour les noms des instruments et quatre boutons.

2. Enregistrer un ActionListener pour chacun des quatre boutons. Nous n'avons pas besoin d'auditeurs pour les cases à cocher, parce que nous n'avons pas l'intention de modifier les sons dynamiquement, autrement dit chaque fois que l'utilisateur coche une case. En revanche, nous attendons qu'il clique le bouton « jouer », puis nous parcourons les 256 cases à cocher pour lire leur état et créer une piste MIDI.

3. Installer le système MIDI (comme au chapitre 11) : ouvrir un séquenceur, créer une séquence, puis créer une piste. Nous utilisons une méthode de Sequencer nouvelle en Java 5.0, setLoopCount( ). Celle-ci vous permet de spécifier combien de fois vous voulez qu'une séquence boucle. Nous utilisons également le facteur de tempo de la séquence pour accélérer ou ralentir ce dernier et maintenir le nouveau tempo d'une itération de la boucle à la suivante.

4. L'action réelle commence quand l'utilisateur clique sur « Jouer ». Le gestionnaire d'événements du bouton « Jouer » appelle la méthode construirePisteEtDemarrer(). Dans cette méthode, nous parcourons les 256 cases à cocher (une seule ligne à la fois, donc 16 temps pour un instrument) pour lire leur état, puis nous utilisons ces informations pour construire une piste MIDI (grâce à une méthode bien pratique, makeEvent(), que nous avons employée au chapitre précédent). Une fois la piste construite, nous lançons le séquenceur, qui joue le motif en boucle tant que l'utilisateur ne clique pas sur « Arrêter ».

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

```
import java.awt.*;
import javax.swing.*;
import javax.sound.midi.*;
import java.util.*;
import java.awt.event.*;


public class BeatBox {

   JPanel panneauPrincipal;
   ArrayList<JCheckBox> listeCases;
   Sequencer sequenceur;
   Sequence sequence;
   Track piste;
   JFrame leCadre;

   String[] nomsInstruments = {"Grosse caisse", "Charleston fermé",
      "Charleston ouvert","Caisse claire", "Cymbale Crash", "Clap",
      "Tom aigu", "Bongos", "Maracas", "Sifflet", "Conga basse",
      "Cloche", "Claqué", "Tom médium basse", "Agogo",
      "Conga ouvert"};
   int[] instruments = {35,42,46,38,49,39,50,60,70,72,64,56,58,47,67,63};


   public static void main (String[] args) {
       new BeatBox().construireIHM();
   }

   public void construireIHM() {
       leCadre = new JFrame("Cyber BeatBox");
       leCadre.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
       BorderLayout agencement = new BorderLayout();
       JPanel arrierePlan = new JPanel(agencement);
       arrierePlan.setBorder(BorderFactory.createEmptyBorder(10,10,10,10));

       listeCases = new ArrayList<JCheckBox>();
       Box boiteBoutons = new Box(BoxLayout.Y_AXIS);

       JButton start = new JButton("Jouer");
       start.addActionListener(new EcouteStart());
       boiteBoutons.add(start);

       JButton stop = new JButton("Arrêter");
       stop.addActionListener(new EcouteStop());
       boiteBoutons.add(stop);

       JButton plusVite = new JButton("Accélérer");
       plusVite.addActionListener(new EcoutePlusVite());
       boiteBoutons.add(plusVite);

       JButton moinsVite = new JButton("Ralentir");
```

Nous stockons les cases à cocher dans une ArrayList.

Les noms des instruments, un tableau de chaînes, pour construire les libellés de chaque ligne.

Les "touches" de la batterie. Le canal batterie ressemble à un piano, sauf que chaque "touche" est un instrument différent. Le nombre 35 est la touche de la grosse caisse, 42 est le charleston fermé, etc.

Une bordure vide nous permet de créer une marge entre les bords du panneau et les composants. Purement esthétique.

Rien de spécial ici : le code de l'IHM dont vous avez déjà vu la majeure partie.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

```
    moinsVite.addActionListener(new EcouteMoinsVite());
    boiteBoutons.add(moinsVite);

    Box boiteNoms = new Box(BoxLayout.Y_AXIS);
    for (int i = 0; i < 16; i++) {
       boiteNoms.add(new Label(nomsInstruments[i]));
    }

    arrierePlan.add(BorderLayout.EAST, boiteBoutons);
    arrierePlan.add(BorderLayout.WEST, boiteNoms);

    leCadre.getContentPane().add(arrierePlan);

    GridLayout grille = new GridLayout(16,16);
    grille.setVgap(1);
    grille.setHgap(2);
    panneauPrincipal = new JPanel(grille);
    arrierePlan.add(BorderLayout.CENTER, panneauPrincipal);

    for (int i = 0; i < 256; i++) {
        JCheckBox c = new JCheckBox();
        c.setSelected(false);
        listeCases.add(c);
        panneauPrincipal.add(c);
 } // fin de la boucle

    installerMidi();

    leCadre.setBounds(100,100,300,300);
    leCadre.pack();
    leCadre.setVisible(true);
 } // fin de la méthode


public void installerMidi() {
  try {
    sequenceur = MidiSystem.getSequencer();
    sequenceur.open();
    sequence = new Sequence(Sequence.PPQ,4);
    piste = sequence.createTrack();
    sequenceur.setTempoInBPM(120);

  } catch(Exception e) {e.printStackTrace();}
} // fin de la méthode
```

Encore du code pour l'IHM. Rien de remarquable.

Créer les cases à cocher et les initialiser à "false" (elles sont donc décochées) et les ajouter à l'ArrayList ET au panneau.

Le code d'installation habituel de MIDI pour obtenir le séquenceur, la séquence et la piste. Rien de spécial là non plus.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Voici le cœur de l'action : nous transformons l'état des cases à cocher en événements MIDI et nous les ajoutons à la piste.

```
public void construirePisteEtDemarrer() {
  int[] listePistes = null;

  sequence.deleteTrack(piste);
  piste = sequence.createTrack();

    for (int i = 0; i < 16; i++) {
      listePistes = new int[16];

      int touche = instruments[i];

      for (int j = 0; j < 16; j++ ) {

          JCheckBox jc = (JCheckBox) listeCases.get(j + (16*i));
          if ( jc.isSelected()) {
             listePistes[j] = touche;
          } else {
             listePistes[j] = 0;
          }
       } // fin de la boucle interne

       creerPistes(listePistes);
       piste.add(makeEvent(176,1,127,0,16));
   } // fin de la boucle externe

   piste.add(makeEvent(192,9,1,0,15));
   try {

      sequenceur.setSequence(sequence);
        sequenceur.setLoopCount(sequenceur.LOOP_CONTINUOUSLY);
       sequenceur.start();
       sequenceur.setTempoInBPM(120);
   } catch(Exception e) {e.printStackTrace();}
} // fin de la methode construirePisteEtDemarrer

public class EcouteStart implements ActionListener {
    public void actionPerformed(ActionEvent a) {
        construirePisteEtDemarrer();
    }
} // fin de la classe interne
```

Nous créons un tableau à 16 éléments qui contiendra les valeurs pour un instrument, pendant 16 temps. Si l'instrument est censé jouer sur un temps donné, la valeur de cet élément sera la touche. SINON, affecter zéro.

Se débarrasser de l'ancienne piste et en créer une nouvelle.

Répéter pour chacune des 16 LIGNES (Conga basse, etc.)

Affecter la "touche" qui représente l'instrument (Cloche, Charleston, etc.) Le tableau d'instruments contient les nombres MIDI pour chacun d'eux.

Répéter pour chaque TEMPS de la ligne.

La case à cocher de ce temps est-elle sélectionnée ? Si oui, placer la valeur dans le tableau (à l'emplacement qui représente ce temps). Si non, l'instrument n'est PAS censé jouer sur ce temps et nous affectons zéro.

Pour cet instrument et pour les 16 temps, créer les événements et les ajouter à la liste.

Nous voulons être toujours sûr qu'il EXISTE un événement sur le temps 16. Sinon, il manquerait un temps avant que la boucle ne recommence.

Vous permet de spécifier le nombre d'itérations de la boucle, ou, comme ici, une boucle sans fin.

Et MAINTENANT, JOUER !

Première des classes internes qui écoutent les boutons. Rien de spécial.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

```
public class EcouteStop implements ActionListener {
    public void actionPerformed(ActionEvent a) {
        sequenceur.stop();
    }
} // fin de la classe interne

public class EcoutePlusVite implements ActionListener {
    public void actionPerformed(ActionEvent a) {
        float facteurTempo = sequenceur.getTempoFactor();
        sequenceur.setTempoFactor((float)(facteurTempo * 1.03));
    }
 } // fin de la classe interne

 public class EcouteMoinsVite implements ActionListener {
     public void actionPerformed(ActionEvent a) {
        float facteurTempo = sequenceur.getTempoFactor();
        sequenceur.setTempoFactor((float)(facteurTempo * .97));
    }
} // fin de la classe interne
```

Les deux autres classes internes pour les boutons.

Modifie le tempo du séquenceur en utilisant le facteur spécifié. Le défaut étant 1.0, Nous l'ajustons de +/– 3 % par clic..

Création des événements pour un instrument à la fois et pour tous les temps. Il y a par exemple un int[] pour la caisse claire, et chaque indice du tableau contient la touche de cet instrument ou un zéro. Si c'est un zéro, l'instrument ne joue pas. Sinon, créer un événement et l'ajouter à la piste.

```
public void creerPistes(int[] liste) {

   for (int i = 0; i < 16; i++) {
      int touche = liste[i];

      if (touche != 0) {
         piste.add(makeEvent(144,9,touche, 100, i));
         piste.add(makeEvent(128,9,touche, 100, i+1));
      }
   }
}
```

Créer les événements NOTE ON et NOTE OFF et les ajouter à la piste.

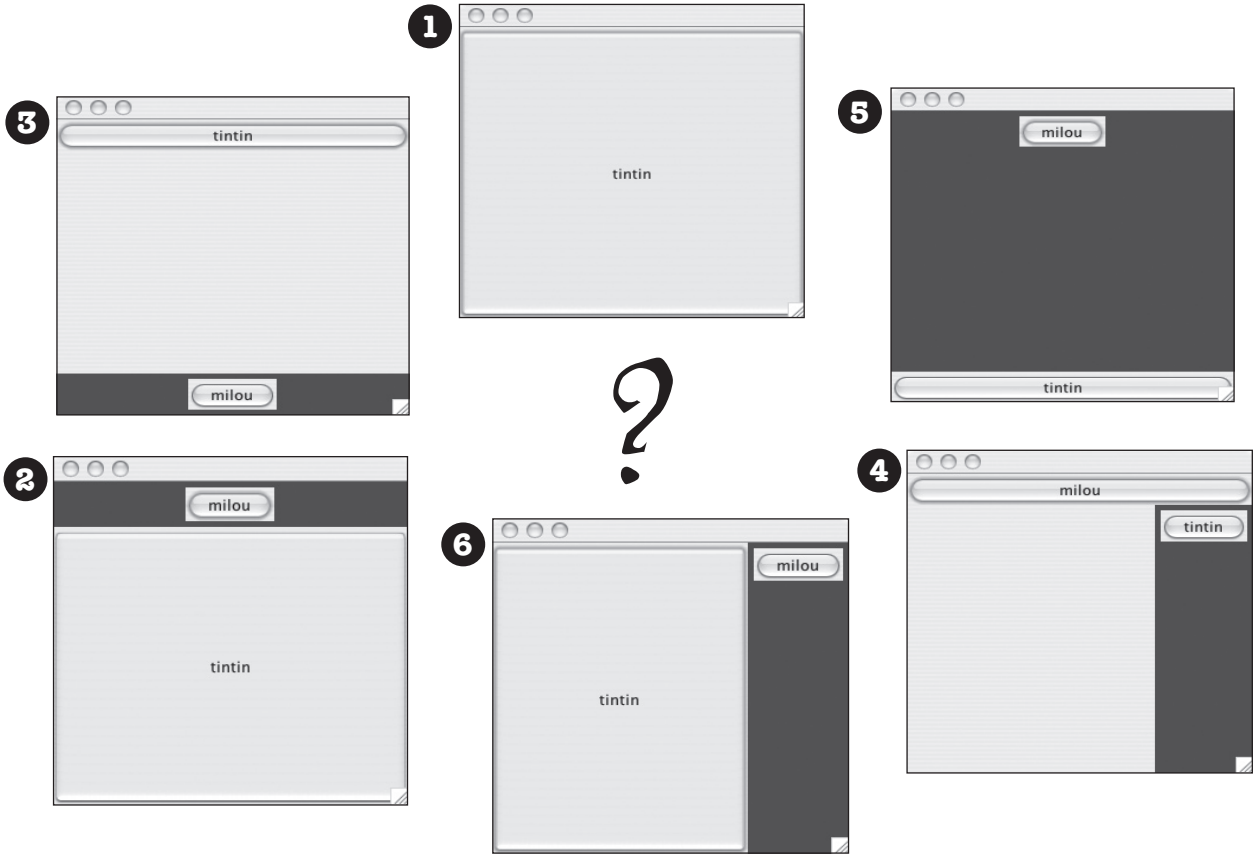
```
public  MidiEvent makeEvent(int comd, int chan, int one, int two, int tick) {
    MidiEvent evenement = null;
    try {
        ShortMessage a = new ShortMessage();
        a.setMessage(comd, chan, one, two);
        evenement = new MidiEvent(a, tick);

    } catch(Exception e) {e.printStackTrace(); }
    return evenement;
}

} // fin de la classe
```

La méthode utilitaire des Recettes de code du chapitre précédent. Rien de nouveau.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Quel code pour quel agencement ?

Cinq des six écrans ci-dessous ont été créés par les fragments de code de la page suivante. Appariez chaque fragment de code avec l'écran correspondant.



Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Fragments de code

A

```
JFrame cadre = new JFrame();
JPanel panneau = new JPanel();
panneau.setBackground(Color.darkGray);
JButton bouton = new JButton("tintin");
JButton boutonDeux = new JButton("milou");
cadre.getContentPane().add(BorderLayout.NORTH,panneau);
panneau.add(boutonDeux);
cadre.getContentPane().add(BorderLayout.CENTER,bouton);
```

B

```
JFrame cadre = new JFrame();
JPanel panneau = new JPanel();
panneau.setBackground(Color.darkGray);
JButton bouton = new JButton("tintin");
JButton boutonDeux = new JButton("milou");
panneau.add(boutonDeux);
cadre.getContentPane().add(BorderLayout.CENTER,bouton);
cadre.getContentPane().add(BorderLayout.EAST, panneau);
```

C

```
JFrame cadre = new JFrame();
JPanel panneau = new JPanel();
panneau.setBackground(Color.darkGray);
JButton bouton = new JButton("tintin");
JButton boutonDeux = new JButton("milou");
panneau.add(boutonDeux);
cadre.getContentPane().add(BorderLayout.CENTER,bouton);
```

D

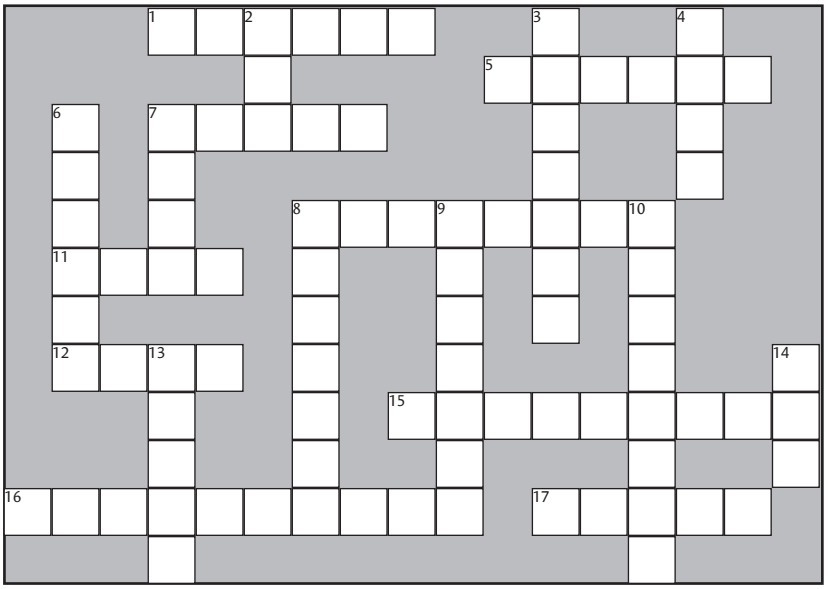
```
JFrame cadre = new JFrame();
JPanel panneau = new JPanel();
panneau.setBackground(Color.darkGray);
JButton bouton = new JButton("tintin");
JButton boutonDeux = new JButton("milou");
panneau.add(bouton);
cadre.getContentPane().add(BorderLayout.NORTH,boutonDeux);
cadre.getContentPane().add(BorderLayout.EAST, panneau);
```

E

```
JFrame cadre = new JFrame();
JPanel panneau = new JPanel();
panneau.setBackground(Color.darkGray);
JButton bouton = new JButton("tintin");
JButton boutonDeux = new JButton("milou");
cadre.getContentPane().add(BorderLayout.SOUTH,panneau);
panneau.add(boutonDeux);
cadre.getContentPane().add(BorderLayout.NORTH,bouton);
```


Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Mots-Croisés 7.0



Vous pouvez le faire.

Horizontalement

1. Sert à créer un panneau
5. Pour le milieu
7. Look Java
8. Pour créer des composants graphiques
11. Look Macintosh
12. Région orientale
15. Composant d'une boîte de dialogue
16. Ils se produisent
17. Région septentrionale

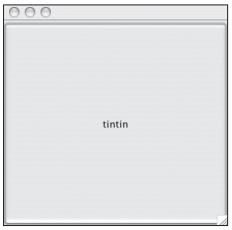
Verticalement

2. Abstract Windowing Toolkit
3. Fixe une taille
4. Région occidentale
6. Sert à créer un cadre
7. Donne le choix à l'utilisateur
8. Contraire de 3 vertical
9. Il en faut pour les textes
10. Mémorisera
13. Package indispensable pour les interfaces graphiques
14. X ou Y

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Solutions des exercices

1



C

```
JFrame cadre = new JFrame();
JPanel panneau = new JPanel();
panneau.setBackground(Color.darkGray);
JButton bouton = new JButton("tintin");
JButton boutonDeux = new JButton("milou");
panneau.add(boutonDeux);
cadre.getContentPane().add(BorderLayout.CENTER,bouton);
```

2

A

```
JFrame cadre = new JFrame();
JPanel panneau = new JPanel();
panneau.setBackground(Color.darkGray);
JButton bouton = new JButton("tintin");
JButton boutonDeux = new JButton("milou");
cadre.getContentPane().add(BorderLayout.NORTH,panneau);
panneau.add(boutonDeux);
cadre.getContentPane().add(BorderLayout.CENTER,bouton);
```

3

E

```
JFrame cadre = new JFrame();
JPanel panneau = new JPanel();
panneau.setBackground(Color.darkGray);
JButton bouton = new JButton("tintin");
JButton boutonDeux = new JButton("milou");
cadre.getContentPane().add(BorderLayout.SOUTH,panneau);
panneau.add(boutonDeux);
cadre.getContentPane().add(BorderLayout.NORTH,bouton);
```

4

D

```
JFrame cadre = new JFrame();
JPanel panneau = new JPanel();
panneau.setBackground(Color.darkGray);
JButton bouton = new JButton("tintin");
JButton boutonDeux = new JButton("milou");
panneau.add(bouton);
cadre.getContentPane().add(BorderLayout.NORTH,boutonDeux);
cadre.getContentPane().add(BorderLayout.EAST, panneau);
```

6

B

```
JFrame cadre = new JFrame();
JPanel panneau = new JPanel();
panneau.setBackground(Color.darkGray);
JButton bouton = new JButton("tintin");
JButton boutonDeux = new JButton("milou");
panneau.add(boutonDeux);
cadre.getContentPane().add(BorderLayout.CENTER,bouton);
cadre.getContentPane().add(BorderLayout.EAST, panneau);
```


Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Solution des mots-croisés

			1 J	P	2 A	N	E	L			3 S			4 W		
					W					5 C	E	N	T	E	R	
	6 J		7 M	E	T	A	L				T			S		
	F		E								S			T		
	R		N			8 G	R	A	9 P	H	I	C	10 S			
	11 A	Q	U	A		E			O		Z		T			
	M					T			L		E		O			
	12 E	A	13 S	T		S			I				C			14 A
			W			I		15 J	C	H	E	C	K	B	O	X
			I			Z			E				E			E
16 E	V	E	N	E	M	E	N	T	S		17 N	O	R	T	H	
			G										A			

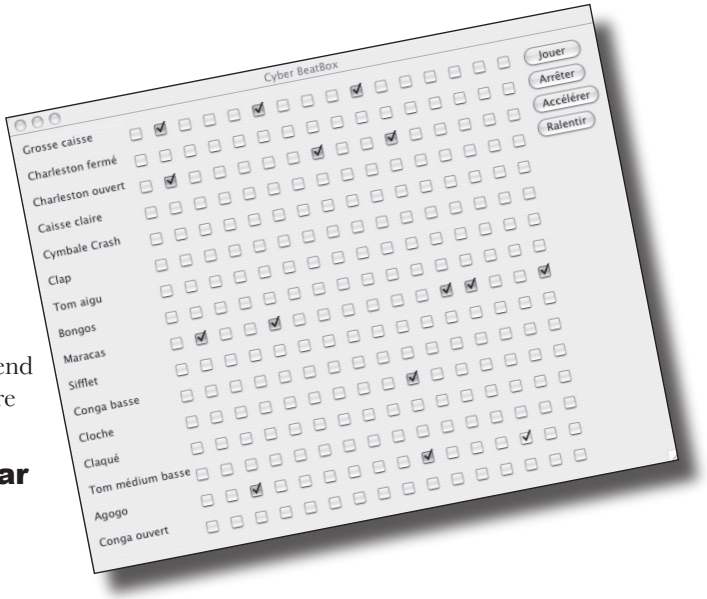
Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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. Mais que se passe-t-il quand c'est le moment de *sauvegarder* l'état d'un objet ? Si vous écrivez un jeu, vous avez besoin d'une fonction pour l'enregistrer et le restaurer. Si vous écrivez une application qui trace des graphiques, vous devez avoir une fonctionnalité pour les ouvrir et les enregistrer. Si votre programme doit sauvegarder un état, *vous pouvez suivre la voie la plus difficile*, interroger chaque objet, puis écrire laborieusement la valeur de chaque variable d'instance dans un fichier que vous créez. Ou bien **vous pouvez le faire simplement, à la mode OO** — en gelant, aplatissant, persistant, déshydratant l'objet, et en le reconstituant, regonflant, restaurant, réhydratant pour le récupérer. Mais c'est *parfois* encore un peu difficile, surtout quand le fichier que votre programme sauvegarde doit être lu par une application non-Java, aussi verrons-nous les deux procédés dans ce chapitre.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Capturez le rythme



Vous avez *créé* le motif parfait. Vous voulez le *conserver*. Vous pourriez vous emparer d'un morceau de papier et gribouiller dessus. À la place, vous cliquez sur le bouton ***Enregistrer*** (ou vous choisissez Enregistrer dans le menu Fichier). Puis vous choisissez un nom, sélectionnez un répertoire, et vous soupirez d'aise en sachant que votre chef d'oeuvre ne va pas disparaître en cas d'écran bleu.

Vous disposez de nombreuses options pour sauvegarder l'état de votre programme, et celle que vous choisirez dépend de la façon dont vous prévoyez de *l'utiliser*. Dans ce chapitre nous verrons ces différentes options.

Si vos données ne sont utilisées que par le programme Java qui les a générées

① Utilisez la sérialisation

Écrivez un fichier qui contient les objets compressés (sérialisés). Ensuite, votre programme lira les objets sérialisés dans ce fichier et les transformera en objets bien réels qui vivront sur le tas.

Si vos données doivent être utilisées par *d'autres* programmes :

② Écrivez un fichier texte brut

Écrivez un fichier avec des délimiteurs compatibles avec ces autres programmes. Par exemple un tableur ou une application de bases de données peut lire un fichier dont les champs sont séparés par des tabulations.

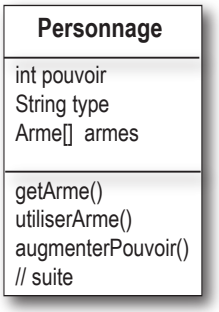
Ce ne sont pas les seules possibilités, bien entendu. Vous pouvez sauvegarder les données au format de votre choix. Vous pouvez par exemple écrire des octets au lieu d'écrire des caractères. Ou vous pouvez choisir un type primitif Java quelconque : il existe des méthodes pour écrire des entiers, des longs, des booléens, etc. Mais, indépendamment de la méthode que vous employez, les techniques d'E/S de base sont pratiquement toujours les mêmes : on écrit des données sur un support *quelconque*, et ce support est généralement un fichier ou un flot provenant d'une connexion réseau. La lecture des données est le processus inverse : on lit les données dans un fichier sur disque ou *via* une connexion réseau. Et, bien sûr, tout ce dont nous parlons dans cette partie concerne les fois où vous n'utilisez pas de base de données.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Sauvegarder un état

Imaginez que vous *ayez* par exemple un jeu d'aventure, et qu'il faille plus d'une séance pour le terminer. À mesure que le jeu progresse, les personnages deviennent plus forts, plus faibles, plus malins, etc., ils gagnent des armes (et ils en perdent). Vous ne voulez pas recommencer depuis le début chaque fois que vous lancez le jeu — il vous a fallu une éternité pour que vos personnages soient prêts à se livrer la plus spectaculaire des batailles. Il vous faut donc un moyen de *sauvegarder* leur état et de le restaurer lorsque vous reprendrez le jeu. Et comme vous êtes également le programmeur du jeu, vous voulez que ce soit aussi facile que possible (et à l'épreuve des utilisateurs inexpérimentés).

Imaginez que vous ayez trois personnages à sauvegarder...



Option un

① **Écrivez les trois objets sérialisés dans un fichier**

Créez un fichier et écrivez trois objets sérialisés. Si vous essayez de le lire comme du texte, il n'a aucun sens :

"ÌsrPersonnage "%gê8MÛIpouvoirzLjava/lang/String;[armest[Ljava/lang/String;xp2tlfur[Ljava.lang.String;≠"VÁÈ{Gxptarcepeepoussiere~»tTrolluq~tmainsnuesgrandehacheq~xtMagicienuq~tformulesinvisibilite

② Option deux

Écrivez un fichier texte

Créez un fichier et écrivez trois lignes de texte, une par personnage en séparant les éléments de l'état par des virgules :

50,Elfe,arc,épée,poussière
200,Troll,mains nues,grande hache
120,Magicien,formules,invisibilité

Le fichier sérialisé est beaucoup plus difficile à lire pour de humains, mais il est beaucoup plus facile (et sûr) pour votre programme de restaurer les fichiers de la sérialisation que de lire les valeurs des variables de l'objet qui auraient été sauvegardées dans un fichier texte. Par exemple, imaginez toutes les façons possibles de lire accidentellement le valeurs dan le mauvais ordre ! Le type pourrait devenir « poussière » au lieu de « Elfe », tandis que l'Elfe deviendrait une arme...

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Écrire un objet sérialisé dans un fichier

Voici les étapes pour sérialiser (sauvegarder) un objet. N'essayez pas de tout mémoriser tout de suite : nous allons bientôt entrer dans les détails.

Si le fichier « MonJeu.ser » n'existe pas, il sera créé automatiquement.

1 Créer un FileOutputStream

```
FileOutputStream fos = new FileOutputStream("MonJeu.ser");
```

Créer un objet FileOutputStream. FileOutputStream sait comment se connecter à un fichier (et en créer un).

2 Créer un ObjectOutputStream

```
ObjectOutputStream oos = new ObjectOutputStream(fos);
```

ObjectOutputStream vous permet d'écrire des objets, mais il ne peut pas se connecter directement à un fichier. On doit lui transmettre un « auxiliaire ». C'est ce qu'on appelle « chaîner » deux flots.

3 Écrire l'objet

```
oos.writeObject(personnageUn);
oos.writeObject(personnageDeux);
oos.writeObject(personnageTrois);
```

Sérialise les objets référencés par personnageUn, personnageDeux et personnageTrois, et les écrire dans le fichier « MonJeu.ser ».

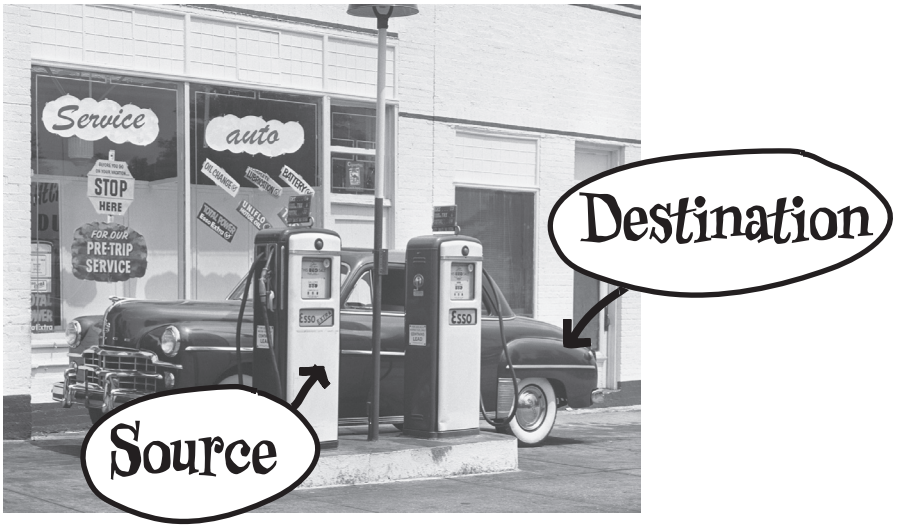
4 Fermer l'ObjectOutputStream

```
oos.close();
```

La fermeture du premier flot entraînant celle des autres, le FileOutputStream se ferme automatiquement (et le fichier aussi).

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Les données se déplacent en flots



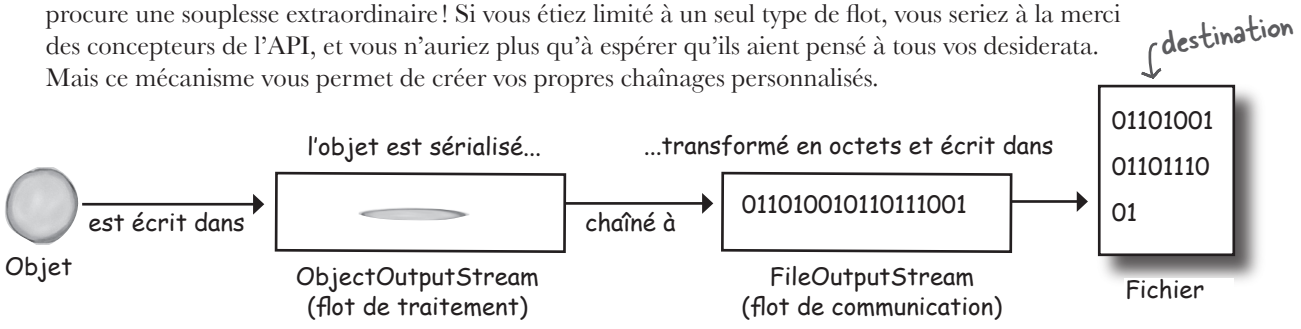
Les flots de communication représentent une connexion à une source ou une destination (fichier, socket, etc.) alors que les flots de traitement ne peuvent pas se connecter eux-mêmes et doivent être chaînés à un flot de communication.

L'API d'E/S Java comprend des flots de ***communication***, qui représentent des destinations et des sources telles que des fichiers ou des sockets réseau, et des flots de ***traitement*** qui ne fonctionnent que s'ils sont chaînés à d'autres flots.

Il faut souvent chaîner au moins deux flots pour faire quelque chose d'utile — *l'un* pour représenter la connexion et *l'autre* pour les appels de méthodes. Pourquoi deux ? Parce que les flots de communication sont habituellement de trop bas niveau. FileOutputStream, par exemple, un flot de communication, a des méthodes pour écrire des octets. Mais ce ne sont pas des octets que nous voulons écrire, ce sont des objets, et il nous faut donc un flot de traitement de plus haut niveau.

Bien. Pourquoi donc n'aurions-nous pas un seul flot qui ferait *exactement* ce que nous voulons ? Un flot qui nous permettrait d'écrire des objets et les convertirait en octets ? Pensez objet. Chaque classe ne fait qu'une seule chose mais la fait bien. FileOutputStream écrit des octets dans un fichier. ObjectOutputStreams transforme les objets en données pouvant être écrites dans un flot. Nous créons donc un FileOutputStream qui nous permet d'écrire dans un fichier, et nous connectons un ObjectOutputStream (un flot de traitement) au bout. Quand nous appelons writeObject() sur ObjectOutputStream, l'objet est aspiré dans le flot puis passe au FileOutputStream où il est finalement écrit sous forme d'octets dans un fichier.

La possibilité de mélanger différentes combinaisons de flots de communication et de traitement vous procure une souplesse extraordinaire ! Si vous étiez limité à un seul type de flot, vous seriez à la merci des concepteurs de l'API, et vous n'auriez plus qu'à espérer qu'ils aient pensé à tous vos desiderata. Mais ce mécanisme vous permet de créer vos propres chaînages personnalisés.



Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

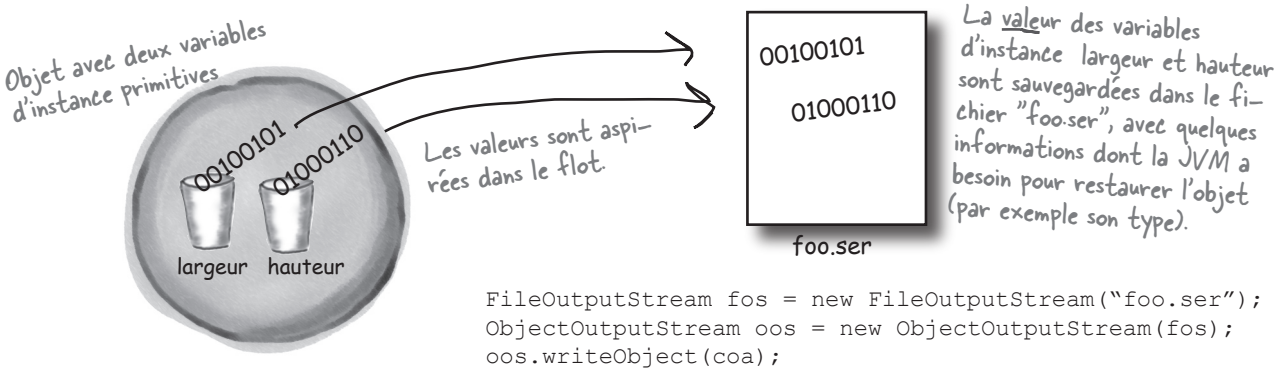
Qu'arrive-t-il réellement à un objet quand il est sérialisé ?

1 Objet sur le tas

Les objets sur le tas ont un état — la valeur de leurs variables d'instance. Ce sont ces valeurs qui différencient une instance de classe d'une autre instance de la même classe.

2 Objet sérialisé

La sérialisation **sauvegarde la valeur des variables d'instance**, pour qu'une instance identique (un objet) puisse être ramené à la vie sur le tas.




```
Foo coa = new Foo ();
coa.setLargeur(37);
coa.setHauteur(70);
```

```
FileOutputStream fos = new FileOutputStream("foo.ser");
ObjectOutputStream oos = new ObjectOutputStream(fos);
oos.writeObject(coa);
```

Créer un FileOutputStream qui se connecte au fichier "foo.ser", puis lui concaténer un ObjectOutputStream et dire à ce dernier d'écrire l'objet.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Mais qu'EST-ce exactement que l'état d'un objet ? Que faut-il sauvegarder ?

Maintenant, cela commence à devenir intéressant. Pas de problème pour sauvegarder les valeurs primitives 37 et 70. Mais que se passe-t-il si un objet a une variable d'instance qui est une *référence* à un objet ? Et si l'objet en a cinq ? Et si ces variables d'instance ont elles-mêmes des variables d'instance ?

Réfléchissez-y. Quelle est la partie d'un objet potentiellement unique ? Imaginez ce qu'il faut restaurer pour obtenir un objet identique à celui qui a été sauvegardé. Bien entendu, il aura un autre emplacement en mémoire, mais c'est sans importance. Tout ce qui nous intéresse, c'est d'avoir sur le tas un objet qui ait le même état que *celui qu'il* avait quand il a été sauvegardé.

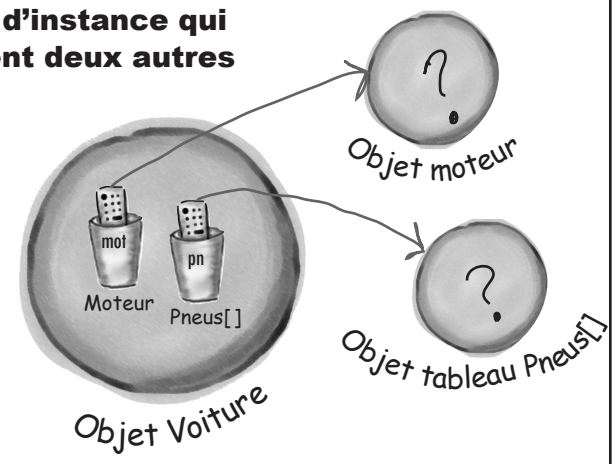
Gym du cerveau

Que doit-il se passer pour que l'objet Voiture soit sauvegardé de telle sorte qu'il puisse retrouver son état original ?

Demandez-vous ce dont vous avez besoin pour sauvegarder la Voiture — et comment procéder.

Et que se passe-t-il si un objet Moteur a une référence à un Carburateur ? Et qu'y a-t-il dans l'objet Pneus [] ?

L'objet Voiture a deux variables d'instance qui référencent deux autres objets.



Que faut-il pour sauvegarder un objet Voiture ?

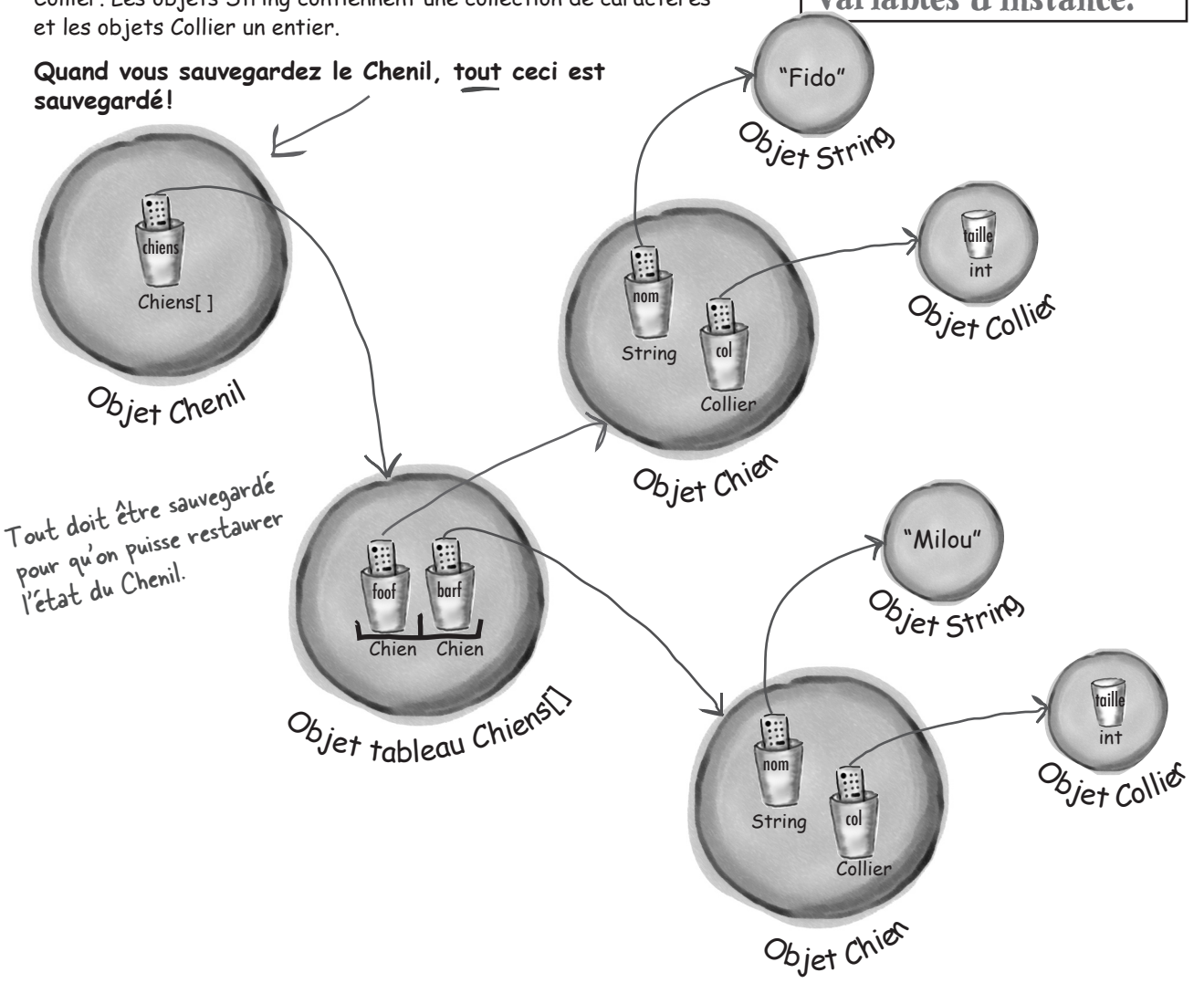
Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Quand un objet est sérialisé, tous les objets que ses variables d'instance référencent sont *également* sérialisés. Et tous les objets que *ces* objets référencent le sont aussi. Et tous les objets que *ces autres* objets référencent... et ce qui est génial, c'est que c'est automatique !

La sérialisation sauvegarde tout le graphe : d'abord le premier objet, puis tous les objets référencés par les variables d'instance.

Cet objet Chenil a une référence a un objet Chiens[] qui est un tableau. Chiens[] contient deux références à deux objets Chien. Chaque Chien contient une référence à un objet String et un objet Collier. Les objets String contiennent une collection de caractères et les objets Collier un entier.

Quand vous sauvegardez le Chenil, tout ceci est sauvegardé !



Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Si vous voulez que votre classe soit sérialisable, implémentez Serializable

On dit que Serializable est une interface de type *marqueur*, parce qu'elle n'a aucune méthode à implémenter et que sa seule finalité est de «marquer» la classe qui l'implémente, d'annoncer qu'elle est sérialisable. Autrement dit, les objets de ce type sont sauvegardables via le mécanisme de *sérialisation*. Si une superclasse est sérialisable, ses sous-classes le sont automatiquement, même si vous ne déclarez pas explicitement *implements Serializable*. (Les interfaces *fonctionnent* toujours ainsi. Si la superclasse «EST-UN» Serializable, la sous-classe l'est aussi.)

```
objectOutputStream.writeObject(maBoite);
```

Tout ce qui va ici DOIT implémenter Serializable, sinon l'exécution échoue.

```
import java.io.*;

public class Boite implements Serializable {

    private int largeur;
    private int hauteur;

    public void setLargeur(int l) {
       largeur = l;
    }

    public void setHauteur(int h) {
       hauteur = h;
    }

public static void main (String[] args) {

      Boite maBoite = new Boite();
      maBoite.setLargeur(50);
      maBoite.setHauteur(20);

      try {
         FileOutputStream fos = new FileOutputStream("foo.ser");
         ObjectOutputStream oos = new ObjectOutputStream(fos);
         oos.writeObject(maBoite);
         oos.close();
      } catch(Exception ex) {
          ex.printStackTrace();
      }
   }
}
```

Serializable étant dans le package java.io, vous devez l'importer.

Pas de méthodes à implémenter, mais "implements Serializable" dit à la JVM qu'il est possible de sérialiser les objets de ce type.

Ces deux valeurs seront sauvegardées.

Les opérations d'E/S peut lancer des exceptions.

Se connecter au fichier "foo.ser" s'il existe, sinon le créer.

Créer un ObjectOutputStream chaîné au flot de connexion.

Lui dire d'écrire l'objet.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Sérialisation = tout ou rien

Pouvez-vous imaginer ce qui se passerait si l'état de certains objets n'était pas sauvé correctement ?

Ouiiiin ! Ça me rend malade rien que d'y penser ! Imaginez que le chien revienne et qu'il ne pèse plus rien. Ou qu'il n'ait plus d'oreilles. Ou que son collier soit réduit à rien. C'est proprement intolérable !

Soit tout le graphe d'objets est sérialisé correctement, soit la sérialisation échoue.

Vous ne pouvez pas sérialiser un objet Mare si sa variable d'instance Canard refuse d'être sérialisée (elle n'implémente pas Serializable).

```
import java.io.*;

public class Mare implements Serializable {

    private Canard canard = new Canard();

     public static void main (String[] args) {
        Mare maMare = new Mare();
       try {
         FileOutputStream fos = new FileOutputStream("Mare.ser");
ObjectOutputStream oos = new ObjectOutputStream(fos);

         oos.writeObject(maMare);
         oos.close();

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

Les objets Mare sont sérialisables.

La classe Mare a une variable d'instance, un Canard.

Quand vous sérialisez maMare (un objet Mare) sa variable d'instance Canard est automatiquement sérialisée.

```
public class Canard{
   // code de Canard
}
```

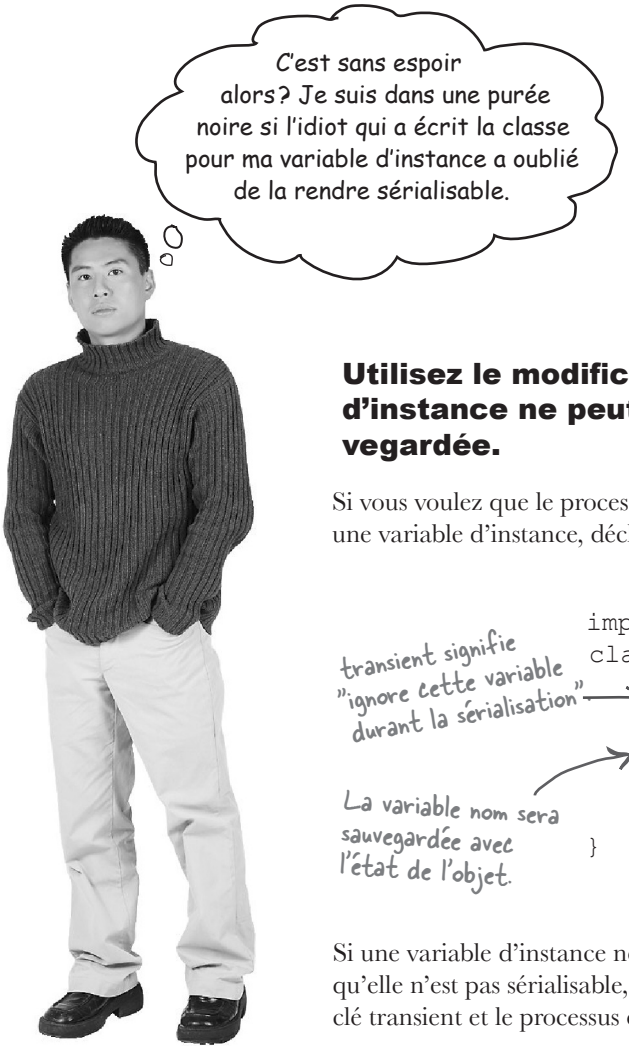
Aïe !! Canard n'est pas sérialisable ! Il n'implémente pas Serializable. Quand vous essayez de sérialiser un objet Mare, c'est l'échec, parce que sa variable d'instance Canard ne peut pas être sauvegardée.

Quand vous tentez d'exécuter la méthode main() de Mare :

Fichier Edition Fenêtre Aide Regrets

```
% java Mare
java.io.NotSerializableException: Canard
        at Mare.main(Mare.java:13)
```


Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>



Utilisez le modificateur transient si une variable d'instance ne peut pas (ou ne doit pas) être sauvegardée.

Si vous voulez que le processus de sérialisation ignore une variable d'instance, déclarez-la avec le mot-clé `transient`.

```
import java.net.*;
class Chat implements Serializable {
    transient String IDCourant;

    String nom;

    // suite du code
}
```

transient signifie "ignore cette variable durant la sérialisation"

La variable nom sera sauvegardée avec l'état de l'objet.

Si une variable d'instance ne peut pas être sauvegardée parce qu'elle n'est pas sérialisable, vous pouvez la marquer avec le mot-clé transient et le processus de sérialisation l'ignorera totalement.

Mais pourquoi une variable ne serait-elle pas sérialisable ? Ce peut être tout simplement parce que le concepteur de la classe a *oublié* de lui faire implémenter Serializable. Ou bien parce que l'objet demande des informations uniquement disponibles au moment de l'exécution et qui ne peuvent donc pas être sauvegardées. Même si de nombreux éléments des bibliothèques de classes Java sont sérialisables, les connexions réseau, les threads et les objets fichiers ne le sont pas. Ils dépendent tous à ce qui se passe au moment de l'exécution, et sont donc spécifiques. Autrement dit, ils sont instanciés d'une façon qui est unique à une exécution donnée de votre programme, sur une certaine plate-forme disposant d'une JVM particulière. Une fois le programme terminé, il n'y a plus moyen de ramener ces événements à la vie d'une façon qui ait un sens ; ils doivent être créés *ex nihilo* à chaque fois.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

il n'y a pas de
Questions stupides

Q : Puisque la sérialisation est si importante, pourquoi toutes les classes ne sont-elles pas sérialisables par défaut ? Si la classe Object implémentait Serializable, toutes les sous-classes seraient automatiquement sérialisables.

R : Même si la plupart des classes peuvent implémenter Serializable, vous avez toujours le choix. Et vous devez prendre la décision consciemment, classe par classe, d'activer la sérialisation en implémentant Serializable. Et d'abord, s'il y avait sérialisation par défaut, comment la désactiverait-on ? Les interfaces indiquent une fonctionnalité, pas une **absence** de fonctionnalité, et le modèle du polymorphisme ne fonctionnerait plus correctement si vous deviez dire « implements NonSerializable » pour annoncer que vous ne voulez pas sauvegarder quelque chose.

Q : Et pourquoi voudrais-je écrire une classe qui ne serait pas sérialisable ?

R : Il y a très peu de raisons, mais vous pourriez par exemple avoir une politique de sécurité qui interdirait de stocker des mots de passe. Ou bien un objet dont la sauvegarde n'a pas de sens, parce que ses principales variables d'instance ne sont pas elles-mêmes sérialisables, et qu'il n'y a donc aucune utilité à rendre *votre* classe sérialisable.

Q : Si j'ai une classe non sérialisable et qu'il n'y a pas de raison à cela (sauf que son concepteur a simplement oublié), est-ce que je peux sous-classer la « mauvaise » classe et rendre la sous-classe sérialisable ?

R : Oui ! Si la classe elle-même est sous-classable (non finale), vous pouvez créer une sous-classe sérialisable et simplement substituer la sous-classe partout où le supertype est attendu. (Le polymorphisme le permet, souvenez-vous.) Ce qui nous amène à une question intéressante : qu'est ce que ça **signifie** si la superclasse n'est pas sérialisable ?

Q : Vous l'avez dit : qu'est-ce que ça signifie d'avoir une sous-classe sérialisable et une superclasse non-sérialisable ?

R : Il faut d'abord comprendre ce qui se passe quand une classe est désérialisée (nous allons en parler bientôt). En un mot, quand un objet est désérialisé et que sa superclasse n'est **pas** sérialisable, le constructeur de la superclasse s'exécute comme si un nouvel objet de ce type était créé. S'il n'y a pas de raison valable pour qu'une classe ne soit pas sérialisable, créer une sous-classe sérialisable peut être une bonne solution.

Q : Ouaouh ! Je viens de réaliser quelque chose... si on déclare une variable « transient », la valeur de cette variable est ignorée durant la sérialisation. Mais que devient-elle ? On résout le problème de la variable d'instance non-sérialisable en la rendant temporaire, mais n'a-t-on pas BESOIN de cette variable quand on ressuscite l'objet ? Tout l'intérêt de la sérialisation est bien de préserver l'état des objets ?

R : Oui, c'est un problème, mais heureusement il a une solution. La variable reviendra à la vie avec une valeur **null**, indépendamment de celle qu'elle avait quand elle a été sauvegardée. Cela veut dire que le graphe d'objets connecté à cette variable d'instance ne sera pas sauvé. C'est un problème, bien évidemment, parce que vous avez probablement besoin d'une valeur non nulle.
Vous avez deux possibilités :
1) Quand l'objet est restauré, réinitialiser cette variable d'instance nulle à une valeur par défaut quelconque. Cela fonctionne si votre objet désérialisé ne dépend pas d'une valeur particulière. Par exemple, il peut être important que le Chien ait un Collier, mais, si tous les objets Collier sont identiques, peu importe que vous donniez au Chien ressuscité un Collier tout neuf. Personne ne verra la différence.
2) Si la valeur de la variable **est** importante (par exemple la couleur de chaque Collier est unique pour chaque Chien) vous devez mémoriser les valeurs du Collier et les utiliser quand le Chien sera désérialisé pour recréer un Collier identique à l'original.

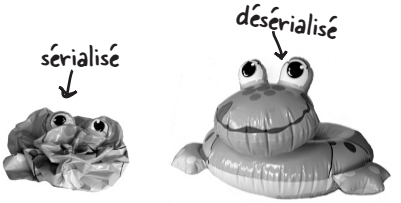
Q : Que se passe-t-il si deux objets du graphe sont le même objet ? Par exemple vous avez deux Chiens dans le chenil, mais ils ont tous deux une référence à l'objet Propriétaire. Est-ce que le propriétaire est sauvegardé deux fois ? J'espère que non.

R : Excellente question ! Le processus est suffisamment intelligent pour savoir quand deux objets du graphe sont un même objet. Dans ce cas, **un seul** des objets est sauvegardé. Durant la désérialisation, toutes les références à ce seul objet sont restaurées.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

La désérialisation restaure les objets

Tout l'intérêt de sérialiser un objet est de pouvoir le restaurer dans son état d'origine à une date ultérieure, lors d'une autre exécution de la JVM (qui ne sera peut-être même pas la JVM qui s'exécutait quand l'objet a été sérialisé). La désérialisation ressemble beaucoup à une sérialisation à rebours.



1 **Créer un FileInputStream**

```
FileInputStream fis = new FileInputStream("MonJeu.ser");
```

Si le fichier "MonJeu.ser" n'existe pas, vous aurez une exception.

Créer un objet FileInputStream. Il sait comment se connecter à un fichier existant.

2 **Créer un ObjectInputStream**

```
ObjectInputStream ois = new ObjectInputStream(fis);
```

ObjectInputStream vous permet de lire les objets, mais il ne peut pas se connecter directement à un fichier. Il doit être chaîné à un flot de connexion, dans ce cas un FileInputStream.

3 **Lire les objects**

```
Object un = ois.readObject();
Object deux = ois.readObject();
Object trois = ois.readObject();
```

Chaque fois que vous dites readObject(), vous extrayez l'objet suivant du flot. Vous les lisez donc dans l'ordre dans lequel ils ont été écrits. Vous obtiendrez une belle grosse exception si vous essayez de lire plus d'objets que vous n'en avez écrits.

4 **Convertir les objets**

```
Personnage elfe = (Personnage) un;
Personnage troll = (Personnage) deux;
Personnage magicien = (Personnage) trois;
```

La valeur de retour de readObject() est de type Object (comme pour ArrayList), et vous devez donc la reconvertir pour qu'elle retrouve son type réel.

5 **Fermer l'ObjectInputStream**

```
ois.close();
```

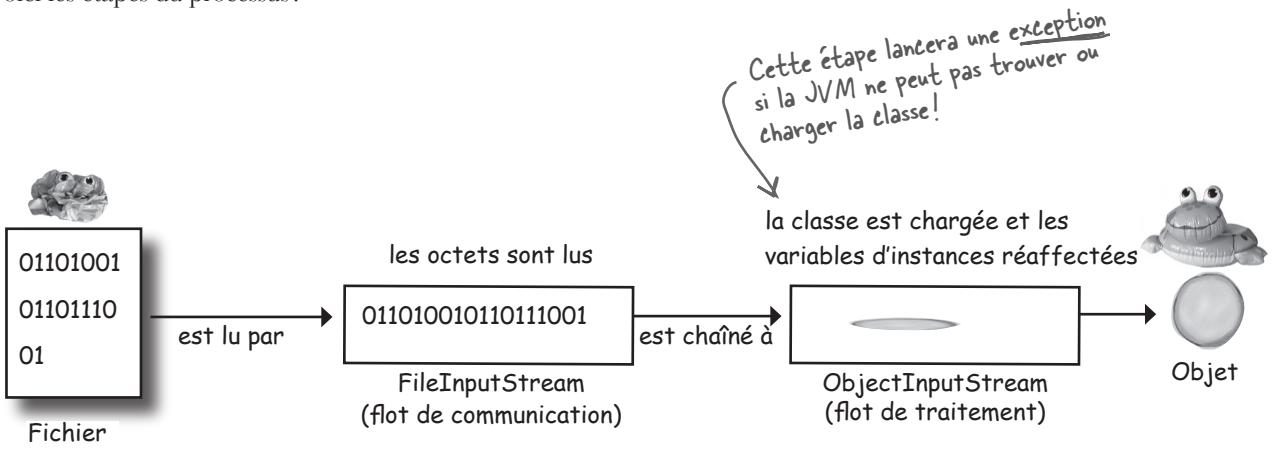
La fermeture du premier flot entraînant celle des autres, le FileInputStream se ferme automatiquement (et le fichier aussi).

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Que se passe-t-il durant la désérialisation ?

Quand un objet est désérialisé, la JVM tente de le ramener à la vie en créant sur le tas un nouvel objet qui ait le même état que l'objet sérialisé au moment où il a été sérialisé. Enfin, sauf pour les variables temporaires qui reviennent soit null (pour les références) soit sous forme de valeurs primitives par défaut.

Voici les étapes du processus :



1. L'objet est **lu** dans le flot.

2. La JVM détermine (en fonction des informations mémorisées avec l'objet sérialisé) le **type** de l'objet.

3. La JVM entreprend de **trouver et charger** la **classe** de l'objet. Si elle ne peut pas trouver et/ou charger la classe, elle lance une exception et la désérialisation échoue.

4. Elle alloue au nouvel objet de l'espace sur le tas, mais **le constructeur de l'objet sérialisé ne s'exécute PAS** ! De toute évidence, s'il s'exécutait, il restaurerait le « nouvel » état de l'objet, et ce n'est pas ce que nous voulons. Nous voulons que l'état restauré soit celui de l'objet *quand il a été sérialisé*, pas quand il a été créé.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

5. Si l'objet a une classe non-sérialisable quelque part dans son arbre généalogique, le **constructeur de cette classe non-sérialisable s'exécute** avec tous les autres constructeurs (même s'ils sont sérialisables). Une fois que le chaînage des constructeurs commence, il est impossible de l'arrêter : toutes les superclasses, à commencer par la première non-sérialisable, vont réinitialiser leur état.

6. Les **variables d'instance reçoivent les valeurs de l'état sérialisé**. Les variables temporaires reçoivent null pour les références et la valeur par défaut (0, false, etc.) pour les types primitifs.

il n'y a pas de Questions stupides

Q : Pourquoi la classe n'est-elle pas sauvegardée dans l'objet ? Cela éviterait de devoir trouver et charger la classe.

R : Oui, on aurait pu faire fonctionner la sérialisation de cette façon. Mais quel terrible gaspillage ce serait. Et si cela ne pose pas trop de difficultés quand on écrit dans un fichier sur un disque local, n'oubliez pas qu'on emploie également la sérialisation pour transmettre des objets sur une connexion réseau. Si la classe accompagnait chaque objet sérialisé (susceptible d'être transmis) , le problème de bande passante serait encore plus important qu'il ne l'est déjà.
Pour les objets sérialisés qui doivent transiter sur le réseau, il *existe* un mécanisme qui « estampille » l'objet avec l'URL de l'endroit où se trouve sa classe. C'est le principe de RMI (*Remote Method Invocation*) qui permet d'envoyer un objet sérialisé, comme argument de méthode par exemple : si la JVM qui reçoit l'appel de méthode ne possède pas la classe, elle utilise l'URL pour la chercher sur le réseau et la charger automatiquement. (Nous parlerons de RMI au chapitre 17.)

Q : Et les variables statiques ? Sont-elles sérialisées ?

R : Non. N'oubliez pas que statique signifie « une par classe » non « une par objet ». Les variables statiques ne sont pas sauvegardées. Quand l'objet sera désérialisé, il aura la variable statique que sa classe aura *à ce moment-là*. Moralité : ne faites pas dépendre les objets sérialisables d'une variable statique qui change dynamiquement ! Sa valeur pourrait être différente lorsque l'objet revient.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Sauvegarder et restaurer le personnages du jeu

```
import java.io.*;

public class TestSauvegarde {
   public static void main(String[] args) {
     Personnage un = new Personnage(50, "Elfe", new String[] {"arc", "épée", "poussière"});
     Personnage deux = new Personnage(200, "Troll", new String[] {"mains nues", "grande hache"});
     Personnage trois = new Personnage(120, "Magicien", new String[] {"formules", "invisibilité"});

     // imaginez du code qui pourrait modifier les valeurs de l'état des personnages

     try {
        ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("Jeu.ser"));
        oos.writeObject(un);
        oos.writeObject(deux);
        oos.writeObject(trois);
        oos.close();
    } catch(IOException ex) {
       ex.printStackTrace();
    }
    un = null;
    deux = null;
    trois = null;

    try {
      ObjectInputStream ois = new ObjectInputStream(new FileInputStream("Jeu.ser"));
       Personnage unBis = (Personnage) ois.readObject();
       Personnage deuxBis = (Personnage) ois.readObject();
       Personnage troisBis = (Personnage) ois.readObject();

      System.out.println("Type de Un : " + unBis.getType());
      System.out.println("Type de Deux : " + deuxBis.getType());
      System.out.println("Type de Trois : " + troisBis.getType());
    } catch(Exception ex) {
       ex.printStackTrace();
    }
  }
}
```

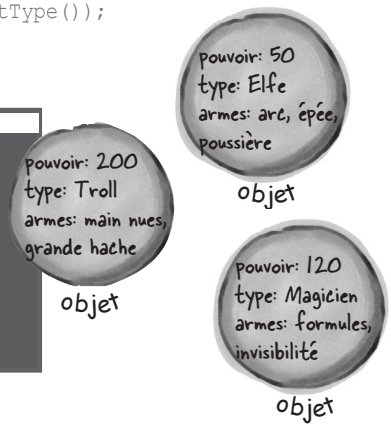
Créons quelques personnages

Initialisés à null : on ne peut pas accéder aux objets sur le tas.

Maintenant, relire le fichier

Vérifier que cela fonctionne

```
% java TestSauvegarde
Elfe
Troll
Magicien
```



Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

La classe Personnage

```
import java.io.*;

public class Personnage implements Serializable {
   int pouvoir;
   String type;
   String[] armes;

   public Personnage(int p, String t, String[] a) {
       pouvoir = p;
       type = t;
       armes = a;
   }

   public int getPouvoir() {
     return pouvoir;
   }

   public String getType() {
      return type;
   }

   public String getArmes() {
       String listeArmes = "";

       for (int i = 0; i < armes.length; i++) {
          listeArmes += armes[i] + " ";
       }
       return listeArmes;
   }
}
```

Voici une classe élémentaire qui ne sert qu'à tester la sérialisation. Il n'y a pas de vrai jeu : nous vous laissons le plaisir de l'écrire.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Sérialisation des objets

POINTS D'IMPACT

- On peut sauver l'état d'un objet en le sérialisant.
- Pour sérialiser un objet, il faut un ObjectOutputStream (dans le package java.io).
- Il existe des flots de communication et des flots de traitement.
- Les flots de communication représentent une connexion à une source ou une destination : un fichier, une socket réseau ou la console.
- Les flots de traitement ne peuvent pas se connecter à une source ou à une destination et doivent être chaînés à un flot de communication.
- Pour sérialiser un objet dans un fichier, créer un FileOutputStream et le chaîner à un ObjectOutputStream.
- Pour sérialiser un objet, appelez *writeObject(lObjet)* sur l'ObjectOutputStream. Vous n'avez pas besoin d'appeler de méthodes sur le FileOutputStream.
- Pour pouvoir être sérialisé, un objet doit implémenter l'interface Serializable. Si une superclasse implémente Serializable, la sous-classe sera automatiquement sérialisable même si elle ne déclare pas explicitement *implements Serializable*.
- Quand un objet est sérialisé, le graphe d'objets entier est sérialisé. Tous les objets référencés par les variables d'instance de l'objet sérialisé sont sérialisés, et tous les objets référencés par ces objets... et ainsi de suite.
- Si l'un des objets du graphe n'est pas sérialisable, une exception sera lancée à l'exécution, sauf si la variable d'instance référençant l'objet est ignorée.
- Marquez une variable d'instance avec le mot-clé *transient* si vous voulez que la sérialisation ignore cette variable. Elle sera restaurée avec la valeur null (pour les références) ou une valeur par défaut (pour les primitives).
- Durant la désérialisation, la classe de tous les objets du graphe doit être accessible à la JVM.
- Les objets sont lus (avec readObject()) dans l'ordre dans lequel ils ont été écrits à l'origine.
- Le type de retour de readObject() est Object, et les objets désérialisés doivent être reconvertis dans leur type réel.
- Les variables statiques ne sont pas sérialisées ! Sauvegarder la valeur d'une variable statique comme une partie de l'état d'un objet spécifique n'a pas de sens, puisque tous les objets de ce type partagent une seule valeur — celle de la classe.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Écrire du texte dans un fichier

Sérialiser des objets est la façon la plus simple de sauvegarder et de restaurer des données entre deux exécutions d'un programme Java. Mais vous devez parfois enregistrer des données dans un bon vieux fichier texte. Imaginez que votre programme Java doive écrire des données dans un simple fichier texte qu'un autre programme (pas nécessairement Java) doit pouvoir lire. Vous pourriez par exemple avoir une servlet (du code Java qui s'exécute sur un serveur Web) qui lit les données d'un formulaire que l'utilisateur a saisies dans un navigateur et les écrit dans un fichier texte que quelqu'un d'autre charge dans un tableur pour les analyser.

Écrire du texte (des données de type String) est similaire à l'écriture d'un objet, sauf qu'on écrit des chaînes de caractères et qu'on utilise un FileWriter au lieu d'un FileOutputStream (et qu'on ne le chaîne pas à un ObjectOutputStream).

Les données de nos personnages écrites sous forme lisible dans un fichier texte.

```
50,Elfe,arc, épée,poussière
200,Troll,mains
nues,grande hache
120,Magicien,formules,invisibilité
```

Pour écrire un objet sérialisé :

```
objectOutputStream.writeObject(unObjet);
```

Pour écrire du texte :

```
fileWriter.write("J'enregistre mon premier texte");
```

```
import java.io.*;

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

        try {
            FileWriter fw = new FileWriter("Foo.txt");

            fw.write("bonjour!");

            fw.close();

        } catch(IOException ex) {
            ex.printStackTrace();
        }
    }
}
```

Nous avons besoin du package java.io pour accéder à FileWriter.

Si le fichier "Foo.txt" n'existe pas, FileWriter le créera.

La méthode write() accepte un argument de type String.

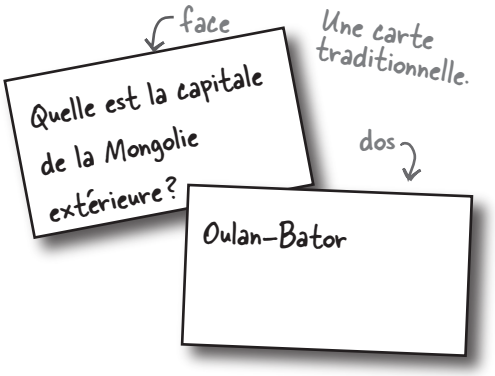
Fermer quand vous avez terminé!

TOUT le code d'E/S doit être dans un bloc try/catch. Tout ceci peut lancer une IOException!!

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Exemple de fichier texte: e-Flashcards

Les écoliers d'Amérique du Nord utilisent des «flashcards», des cartes-éclair avec une question d'un côté et la réponse de l'autre. Elles ne servent pas à grand chose pour comprendre une notion, mais elles sont imbattables quand il s'agit d'apprendre par coeur, quand il faut graver un fait dans sa mémoire. Et il n'y a pas mieux pour les jeux comme Trivial Pursuit™ ou Questions pour un champion.

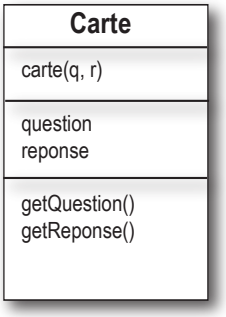


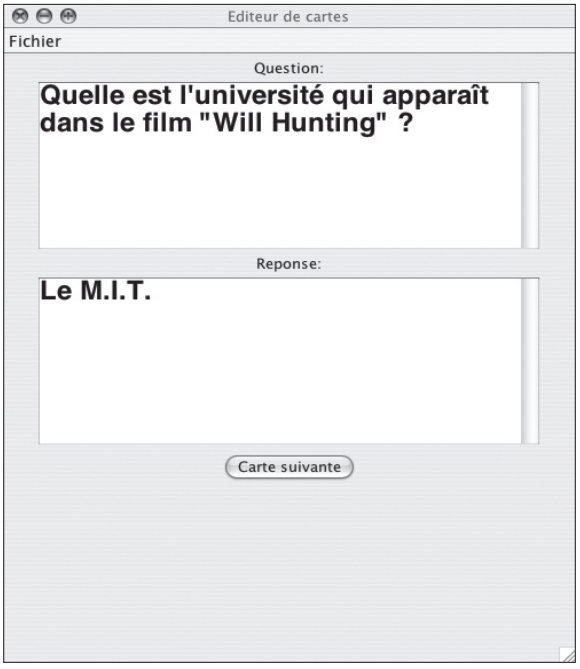
Nous allons en créer une version électronique qui a trois classes :

1) ***EditeurCartes***, un simple outil qui permet de créer et de sauvegarder des cartes.

2) ***QuizCartes***, un moteur de jeu qui charge une série de cartes et permet à l'utilisateur de jouer.

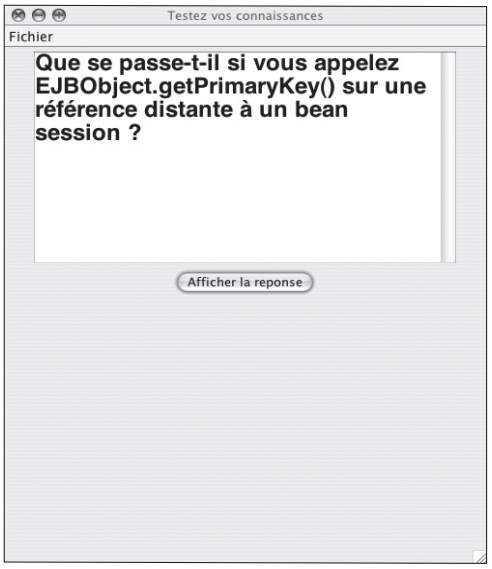
3) ***Carte***, une classe simple qui représente les données des cartes. Nous verrons le code de l'éditeur et du jeu, et vous devrez écrire la classe Carte vous-même en vous inspirant de ce diagramme.





EditeurCartes

Elle possède un menu Fichier avec une option «Enregistrer» pour sauvegarder une série de cartes dans un fichier texte.



QuizCartes

Elle possède un menu Fichier avec une option «Ouvrir» pour charger une série de cartes à partir d'un fichier texte.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

EditeurCartes (grandes lignes du code)

```
public class EditeurCartes {

  public void go() {
    // construit et affiche l'IHM
  }
```

Construit et affiche l'IHM, crée et enregistre les auditeurs d'événements.

Classe interne

```
  private class NouvelleCarte implements ActionListener {
    public void actionPerformed(ActionEvent ev) {
      //ajoute la carte courante à la liste et efface les zones de texte
    }
  }
```

Déclenché quand l'utilisateur clique sur "Carte suivante". Signifie que l'utilisateur veut enregistrer cette carte dans la liste et en commencer une nouvelle.

Classe interne

```
  private class MenuEnregistrer implements ActionListener {
    public void actionPerformed(ActionEvent ev) {
      // affiche une boîte de dialogue qui permet à l'utilisateur
      // de nommer et d'enregistrer le jeu de cartes
    }
  }
```

Déclenché quand l'utilisateur clique sur "Enregistrer" dans le menu Fichier, quand il veut sauvegarder la liste de cartes sous forme de série de questions (par exemple, "Grandes découvertes", "Cinéma italien" "Syntaxe Java", etc.)

Classe interne

```
  private class MenuNouveau ActionListener {
    public void actionPerformed(ActionEvent ev) {
      // efface la liste des cartes et les zones de texte
    }
  }
```

Déclenché par la sélection de "Nouveau" dans le menu Fichier parce que l'utilisateur veut commencer une nouvelle série (c'est pourquoi nous effaçons la liste et les zones de texte).

```
  private void sauveFichier (File fichier) {
    // parcourir la liste des cartes et écrire chacune d'elles dans un fichier texte
    // de façon analysable (autrement dit avec des séparateurs)
  }
}
```

Appelé par MenuEnregistrer, effectue l'écriture proprement dite.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

```
import java.util.*;
import java.awt.event.*;
import javax.swing.*;
import java.awt.*;
import java.io.*;

public class EditeurCartes {
    private JTextArea question;
    private JTextArea reponse;
    private ArrayList<Carte> listeCartes;
    private JFrame cadre;

    public static void main (String[] args) {
        EditeurCartes editeur = new EditeurCartes();
        editeur.go();
    }

    public void go() {
        // construction de l'IHM

        cadre = new JFrame("Éditeur de cartes");
        JPanel panneau = new JPanel();
        Font grossePolice = new Font("sanserif", Font.BOLD, 24);
        question = new JTextArea(6,20);
        question.setLineWrap(true);
        question.setWrapStyleWord(true);
        question.setFont(grossePolice);

        JScrollPane ascenseurQ = new JScrollPane(question);
        ascenseurQ.setVerticalScrollBarPolicy(ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS);
        ascenseurQ.setHorizontalScrollBarPolicy(ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER);

        reponse = new JTextArea(6,20);
        reponse.setLineWrap(true);
        reponse.setWrapStyleWord(true);
        reponse.setFont(grossePolice);

        JScrollPane ascenseurR = new JScrollPane(reponse);
        ascenseurR.setVerticalScrollBarPolicy(ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS);
        ascenseurR.setHorizontalScrollBarPolicy(ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER);

        JButton boutonSuivant = new JButton("Carte suivante");

        listeCartes = new ArrayList<Carte>();

        JLabel labelQ = new JLabel("Question :");
        JLabel labelR = new JLabel("Réponse :");
        panneau.add(labelQ);
        panneau.add(ascenseurQ);
        panneau.add(labelR);
        panneau.add(ascenseurR);
        panneau.add(boutonSuivant);
        boutonSuivant.addActionListener(new NouvelleCarte());
        JMenuBar barreMenus = new JMenuBar();
        JMenu menuFichier = new JMenu("Fichier");
        JMenuItem newMenuItem = new JMenuItem("Nouveau");
        JMenuItem saveMenuItem = new JMenuItem("Enregistrer");

        newMenuItem.addActionListener(new MenuNouveau());
        saveMenuItem.addActionListener(new MenuEnregistrer());
```

Cette page ne contient que le code de l'IHM. Rien de spécial, mais vous voudrez peut-être jeter un coup d'oeil sur le code des menus (JMenu, JMenuBar, JMenuItem).

Nous créons une barre de menus, puis un menu Fichier, et nous plaçons dans celui-ci les éléments "Nouveau" et "Enregistrer". Puis nous ajoutons le menu à la barre et nous l'intégrons au cadre. Les éléments de menu peuvent déclencher un ActionEvent.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

```
        menuFichier.add(newMenuItem);
        menuFichier.add(saveMenuItem);
        barreMenus.add(menuFichier);
        cadre.setJMenuBar(barreMenus);
        cadre.getContentPane().add(BorderLayout.CENTER, panneau);
        cadre.setSize(500,600);
        cadre.setVisible(true);
    }

    private class NouvelleCarte implements ActionListener {
        public void actionPerformed(ActionEvent ev) {

            Carte carte = new Carte(question.getText(), reponse.getText());
            listeCartes.add(carte);
            effacerCarte();
        }
    }

    private class MenuEnregistrer implements ActionListener {
        public void actionPerformed(ActionEvent ev) {

            Carte carte = new Carte(question.getText(), reponse.getText());
            listeCartes.add(carte);

            JFileChooser enregistrerFichier = new JFileChooser();
            enregistrerFichier.showSaveDialog(cadre);
            sauveFichier(enregistrerFichier.getSelectedFile());
        }
    }

    private class MenuNouveau implements ActionListener {
            public void actionPerformed(ActionEvent ev) {
            listeCartes.clear();
            effacerCarte();
        }
    }

    private void effacerCarte() {
        question.setText("");
        reponse.setText("");
        question.requestFocus();
        }

    private void sauveFichier(File fichier) {
        try {
            BufferedWriter bw = new BufferedWriter(new FileWriter(fichier));

            for(Carte carte:listeCartes) {
                bw.write(carte.getQuestion() + "/");
                bw.write(carte.getReponse() + "\n");
            }
          bw.close();

        } catch(IOException ex) {
            System.out.println("écriture impossible");
            ex.printStackTrace();
        }
    }
}
```

Affiche une boîte de dialogue et attend que l'utilisateur choisisse "Enregistrer". Toute la navigation et la sélection des fichiers est réalisée par le JFileChooser ! Rien de plus simple..

La méthode qui écrit réellement (appelée par le gestionnaire d'événements de MenuEnregistrer). L'argument est l'objet File que l'utilisateur enregistre. Nous parlerons de la classe File à la page suivante.

Nous chaînons un BufferedWriter à un nouveau FileWriter pour optimiser l'écriture. Nous en parlerons dans quelques pages.

Parcourir l'ArrayList de cartes et les écrire une par une, la question et la réponse séparées par un slash, puis ajouter un retour à la ligne ("\n").

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

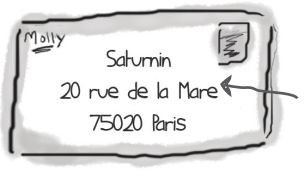
La classe java.io.File

La classe java.io.File *représente* un fichier sur le disque, mais elle ne représente pas le contenu du fichier. Vous pouvez voir un objet File plutôt comme le nom de chemin d'un fichier (ou même d'un *répertoire*) et non comme le Vrai Fichier lui-même. Par exemple, la classe File n'a pas de méthodes de lecture et d'écriture. Un objet File a une propriété TRÈS utile : il permet de représenter de façon beaucoup plus sûre un fichier qu'un simple nom de fichier String. La plupart des classes dont le constructeur accepte un nom de fichier String (comme FileWriter ou FileInputStream) acceptent aussi un objet File. Vous pouvez construire un objet File, vérifier que vous avez un chemin valide, etc. puis transmettre cet objet File à un FileWriter ou un FileInputStream.

Un objet File représente le nom et le chemin d'un fichier ou d'un répertoire sur le disque, par exemple :

/Utilisateurs/Kathy/Donnees/Jeu.txt

Mais il ne représente PAS les données contenues *dans* le fichier et ne vous permet pas d'y accéder !



Une adresse n'est PAS une maison ! Un objet File est comparable à une adresse... Il représente le nom et l'emplacement d'un fichier particulier, mais n'est pas le fichier lui-même.

Un objet File représente le nom de fichier "Jeu.txt".

Ce que vous pouvez faire avec un objet File :

① **Créer un objet représentant un fichier existant**

```
File f = new File("MonCode.txt");
```

② **Créer un répertoire**

```
File rep = new File("Chapitre7");
rep.mkdir();
```

③ **Lister le contenu d'un répertoire**

```
if (rep.isDirectory()) {
      String[] contenuRep = rep.list();
      for (int i = 0; i < contenuRep.length; i++) {
          System.out.println(contenuRep[i]);
      }
}
```

④ **Obtenir le chemin absolu d'un fichier ou d'un répertoire**

```
System.out.println(rep.getAbsolutePath());
```

⑤ **Supprimer un fichier ou un répertoire (retourne true en cas de succès)**

```
boolean supprime = f.delete();
```

Jeu.txt

50,Elfe,arc, épée,poussière
200,Troll,mains nues,grande hache
120,Magicien,formules,invisibilité

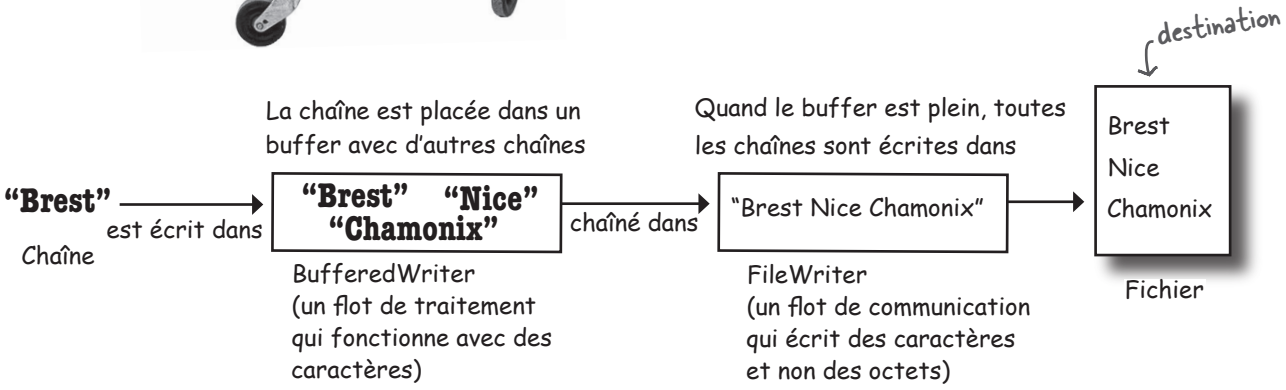
Un objet File ne représente PAS les données contenues dans le fichier (et ne vous permet pas d'y accéder directement) !

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

La beauté des buffers

Sans les buffers, ce serait comme de faire ses courses sans chariot. Vous devriez porter chaque article un par un jusqu'au parking, une boîte de petits pois ou un rouleau de papier toilette à la fois.

Les buffers vous fournissent un contenant temporaire pour grouper les données jusqu'à ce qu'il soit plein (comme le chariot). Avec un buffer, vous faites beaucoup moins de voyages.




```
BufferedWriter writer = new BufferedWriter(new FileWriter(unFichier));
```

Remarquez que nous n'avons même pas besoin de référence à l'objet FileWriter. La seule chose qui nous intéresse, c'est le BufferedWriter, parce que c'est celui sur lequel nous allons appeler les méthodes. Quand nous fermerons le BufferedWriter, il s'occupera du reste de la chaîne.

Ce qu'il y a de génial avec les buffers, c'est leur efficacité. Vous pouvez écrire dans un fichier en n'utilisant qu'un FileWriter et en appelant write(uneChaine), mais FileWriter écrit tout ce que vous lui transmettez dans le fichier à chaque fois. Vous préféreriez sans doute une meilleure solution, parce que tous ces accès disque prennent beaucoup plus de temps que la manipulation des données en mémoire. Si vous chaînez un BufferedWriter à un FileWriter, le BufferedWriter contiendra tout ce que vous écrivez. *Ce n'est que quand le buffer sera plein que le FileWriter sera réellement instruit d'écrire les données sur le disque.*

Si vous voulez envoyer des données *avant* que le buffer ne soit plein, c'est vous qui décidez. ***Il suffit de le vider***. L'appel de writer.flush() signifie : «envoie ***immédiatement*** tout ce qu'il y a dans le buffer!».

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Lire dans un fichier texte

Lire du texte dans un fichier est simple, mais cette fois nous utiliserons un objet File pour représenter le fichier, un FileReader pour lire et un BufferedReader pour optimiser la lecture.

Les lignes sont lues dans une boucle *while*, et la boucle se termine quand le résultat de readLine() est null. C'est la façon la plus courante de lire des données (pratiquement tout ce qui n'est pas un objet sérialisé) : lire dans une boucle while (un *test* while en réalité) et terminer quand il ne reste plus rien à lire (ce que nous savons parce que le résultat de notre méthode de lecture est null).

Fichier contenant deux lignes de texte

Combien font 2 + 2 ?/4
Combien font 20+22 ?/42

MonTexte.txt

Ne pas oublier d'importer ce package.

```
import java.io.*;

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

        try {
            File monFichier = new File("MonTexte.txt");
            FileReader fr = new FileReader(monFichier);

            BufferedReader br = new BufferedReader(fr);

            String ligne = null;

            while ((ligne = br.readLine()) != null) {
                System.out.println(ligne);
            }
            br.close();

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

Un FileReader est un flot de communication pour les caractères, qui se connecte à un fichier texte.

Chaîner le FileReader à un BufferedReader pour plus d'efficacité. Il ne retournera au fichier à lire que quand le buffer sera vide (parce que le programme aura tout lu).

Créer une variable de type String qui contiendra chaque ligne à mesure qu'elle est lue.

Ce bloc signifie "Lis une ligne de texte et affecte-la à la variable "ligne". Tant que la valeur de cette variable n'est pas null (parce qu'IL Y A quelque chose à lire) affiche la ligne que tu viens de lire".

Autrement dit : "Tant qu'il reste des lignes à lire, lis-les et affiche-les".

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Le jeu (grandes lignes du code)

```
public class QuizCartes{

  public void go() {
    // construire et afficher l'IHM
  }

  private class NouvelleCarte implements ActionListener {
    public void actionPerformed(ActionEvent ev) {
      // si c'est une question affiche la réponse, sinon affiche la question suivante
      // positionne un drapeau selon que c'est une question ou une réponse
    }
  }

  private class MenuOuvrir implements ActionListener {
    public void actionPerformed(ActionEvent ev) {
      // affiche une boîte de dialogue pour que l'utilisateur
      // puisse naviguer et choisir une série de cartes
    }
  }

  private void chargerFichier(File fichier) {
    // construit une ArrayList de cartes en les lisant dans un fichier texte
    // appelée par le gestionnaire d'événements de MenuOuvrir, lit le fichier ligne par ligne
    // et dit à la méthode creerCarte() de transformer la ligne en nouvelle carte
    // (une ligne du fichier contient la question et la réponse, séparées par un slash
  }


  private void creerCarte(String ligneAAnalyser) {
    // appelée par la méthode chargerFichier, lit une ligne dans le fichier texte et
    // la décompose en deux — question et réponse — puis crée un nouvel objet Carte
    // et l'ajoute à l'ArrayList nommée listeCartes
  }
}
```


Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

```
import java.util.*;
import java.awt.event.*;
import javax.swing.*;
import java.awt.*;
import java.io.*;

public class QuizCartes {

    private JTextArea affichage;
    private JTextArea reponse;
    private ArrayList<Carte> listeCartes;
    private Carte carteCourante;
    private int indiceCarteCourante;
    private JFrame cadre;
    private JButton boutonSuivant;
    private boolean reponseAffichee;


    public static void main (String[] args) {
       QuizCartes jeu = new QuizCartes();
       jeu.go();
    }

    public void go() {

        // construction de l'IHM

        cadre = new JFrame("Testez vos connaissances");
        JPanel panneauPrincipal = new JPanel();
        Font grossePolice = new Font("sanserif", Font.BOLD, 24);

        affichage = new JTextArea(10,20);
        affichage.setFont(grossePolice);

        affichage.setLineWrap(true);
        affichage.setEditable(false);

        JScrollPane ascenseurQ = new JScrollPane(affichage);
        ascenseurQ.setVerticalScrollBarPolicy(ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS);
        ascenseurQ.setHorizontalScrollBarPolicy(ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER);
        boutonSuivant = new JButton("Afficher une question");
        mainPanel.add(ascenseurQ);
        mainPanel.add(boutonSuivant);
        boutonSuivant.addActionListener(new NouvelleCarte());

        JMenuBar barreMenus = new JMenuBar();
        JMenu menuFichier = new JMenu("Fichier");
        JMenuItem elMenuOuvrir = new JMenuItem("Charger une série");
        elMenuOuvrir.addActionListener(new MenuOuvrir());
        menuFichier.add(elMenuOuvrir);
        barreMenus.add(menuFichier);
        cadre.setJMenuBar(barreMenus);
        cadre.getContentPane().add(BorderLayout.CENTER, panneauPrincipal);
        cadre.setSize(640,500);
        cadre.setVisible(true);

    } // fin de la méthode go
```

Seulement le code de l'IHM sur cette page. Rien de spécial.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

```
private class NouvelleCarte implements ActionListener {
             public void actionPerformed(ActionEvent ev) {
          if (reponseAffichee) {
             // montrer la réponse, ils ont vu la question
             affichage.setText(carteCourante.getReponse());
             boutonSuivant.setText("Carte suivante");
             reponseAffichee = false;
          } else {
              // afficher la question suivante
             if (indiceCarteCourante < listeCartes.size()) {

                afficherNouvelleCarte();

              } else {
                 // il n'y a plus de cartes !
                 affichage.setText("C'était la dernière carte");
                 boutonSuivant.setEnabled(false);
              }
          }
       }
    }

   private class MenuOuvrir implements ActionListener {
       public void actionPerformed(ActionEvent ev) {
           JFileChooser ouvrirFichier = new JFileChooser();
           ouvrirFichier.showOpenDialog(cadre);
           chargerFichier(ouvrirFichier.getSelectedFile());
       }
   }

   private void chargerFichier(File fichier) {

      listeCartes = new ArrayList<Carte>();
      try {
          BufferedReader jeu = new BufferedReader(new FileReader(fichier));
          String ligne = null;
          while ((ligne = jeu.readLine()) != null) {
             creerCarte(ligne);
          }
          jeu.close();

      } catch(Exception ex) {
          System.out.println("lecture du fichier impossible");
          ex.printStackTrace();
      }

     // on commence en affichant la première carte
     afficherNouvelleCarte();
   }

   private void creerCarte(String ligneAAnalyser) {
      String[] resultat = ligneAAnalyser.split("/");
      Carte carte = new Carte(resultat[0], resultat[1]);
      listeCartes.add(carte);
      System.out.println("une carte de plus");
   }

   private void afficherNouvelleCarte() {
        carteCourante = listeCartes.get(indiceCarteCourante);
        indiceCarteCourante++;
        affichage.setText(carteCourante.getQuestion());
        boutonSuivant.setText("Afficher la réponse");
        reponseAffichee = true;
   }
} // fin de la classe
```

Tester le drapeau booléen reponseAffichee pour savoir si nous avons une question appropriée et agir en conséquence.

Afficher la boîte de dialogue pour que l'utilisateur puisse naviguer et choisir un fichier.

Créer un BufferedReader chaîné à un nouveau FileReader, en passant au FileReader l'objet File que l'utilisateur a choisi dans le dialogue

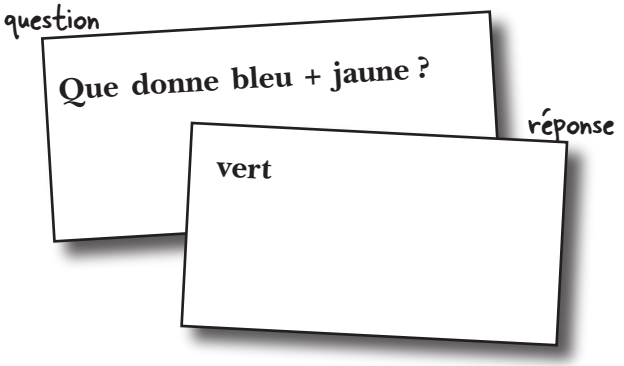
Lire une ligne à la fois en la transmettant à la méthode creerCarte() qui l'analyse, la transforme en carte et l'ajoute à l'ArrayList.

Chaque ligne de texte correspond à une seule carte, mais nous devons séparer la question de la réponse. Pour ce faire, nous utilisons la méthode String.split() pour scinder la ligne en deux parties. Nous allons voir la méthode split() page suivante.

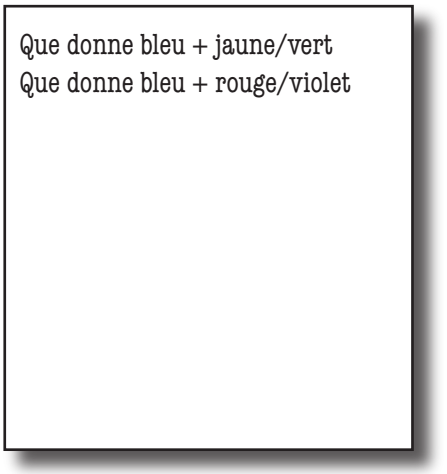
Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Analyser une chaîne avec split()

Imaginez une flashcard comme celle-ci :



Sauvegardée dans un fichier texte :



Comment séparez-vous la question et la réponse ?

Quand vous lisez le fichier, la question et la réponse sont tassées sur une seule ligne et séparées par un slash « / » (parce que c'est comme ça que nous avons écrit le fichier).

La méthode split() permet de décomposer les chaînes

La méthode split() dit « donnez moi un séparateur, je décomposerai la chaîne pour vous et je placerai les différentes parties dans un tableau ».

Dans l'application QuizCartes, c'est à cela que ressemble une ligne quand elle et lue dans le fichier.

```
String aTester = "Que donne bleu + jaune ?/vert";

String[] resultat = aTester.split("/");

for (String partie:result) {

   System.out.println(partie);

}
```

La méthode split() prend le slash et s'en sert pour décomposer la chaîne, ici en deux parties. (Note : split() est BEAUCOUP plus puissant que ce que nous en montrons ici. Elle peut réaliser des analyses extrêmement complexes, avec des filtres, des métacaractères, etc.)

Parcourir le tableau et afficher chaque partie. Dan cet exemple, il n'y en a que deux « Que donne bleu + jaune ? » et « vert ».

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

il n'y a pas de
Questions stupides

Q : J'ai regardé la documentation, et il y a environ cinq millions de classes dans le package java.io. Comment diable savoir lesquelles utiliser ?

R : L'API d'E/S utilise le concept modulaire de « chaînage » qui vous permet d'abouter flots de communication et flots de traitement (également nommé « filtres »). L'éventail de combinaisons est si vaste que vous pouvez obtenir exactement ce que vous voulez.

Le chaînage n'est pas nécessairement limité à deux niveaux : vous pouvez connecter ensemble autant de flots de traitement que vous en avez besoin.

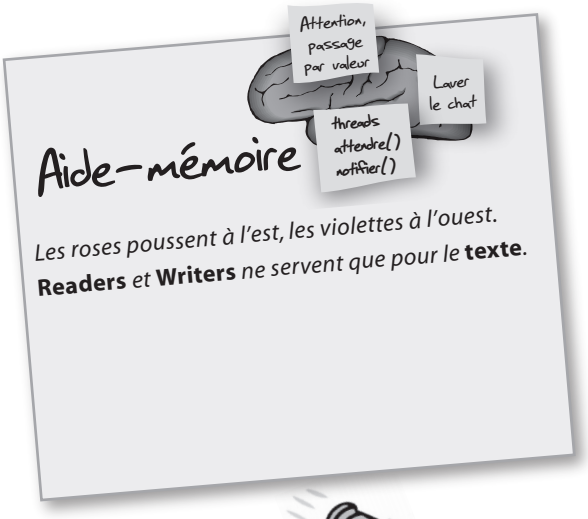
Mais, la plupart du temps, vous n'utiliserez qu'une poignée de classes, toujours les mêmes. Si vous écrivez des fichiers texte, vous n'aurez probablement besoin de rien d'autre que de BufferedReader et BufferedWriter (chaînés à FileReader et FileWriter). Si vous écrivez des objets sérialisés, vous utiliserez ObjectOutputStream et ObjectInputStream (chaînés à FileInputStream et FileOutputStream).

Autrement dit, nous avons déjà abordé 90 % de ce qui est généralement nécessaire pour les E/S Java.

Q : Et les nouvelles classes d'E/S nio ajoutées à la 1.4 ?

R : Les classes java.nio améliorent beaucoup les performances et exploitent mieux les capacités natives de la machine sur laquelle votre programme tourne. L'une des nouvelles caractéristiques essentielles de nio est le contrôle direct des buffers. Citons également les E/S non bloquantes, ce qui signifie que le code des E/S ne reste pas bloqué à attendre s'il n'y a rien à lire ni à écrire. Certaines des classes existantes (notamment FileInputStream et FileOutputStream) tirent parti en coulisse de ces nouvelles caractéristiques. Mais les classes sont plus compliquées à mettre en oeuvre, et si vous n'en avez pas **vraiment** besoin, vous pouvez vous en tenir aux versions plus simples que nous avons étudiées ici. De plus, si vous ne faites pas attention, NIO peut entraîner une **dégradation** des performances. Les E/S non-NIO conviennent vraisemblablement pour 90 % de ce que vous écrirez, surtout si vous débutez en Java.

Mais vous **pouvez** vous faciliter la vie en utilisant FileInputStream et en accédant à son **canal** via la méthode getChannel() (ajoutée à FileInputStream depuis Java 1.4).



POINTS D'IMPACT

- Pour écrire un fichier texte, commencez par un flot de communication FileWriter.
- Chaînez le FileWriter à un BufferedWriter pour augmenter l'efficacité.
- Un objet File représente un fichier et son nom de chemin, mais il ne représente pas le contenu réel du fichier.
- Avec un objet File, vous pouvez créer, traverser et supprimer des répertoires.
- La plupart des flots qui attendent un nom de fichier String peuvent accepter un objet File qui peut être plus sûr à utiliser.
- Pour lire un fichier texte, commencez par un flot de communication FileReader.
- Chaînez le FileReader à un BufferedReader pour augmenter l'efficacité.
- Pour analyser un fichier texte, il faut écrire celui-ci de manière à pouvoir reconnaître les différents éléments. Une approche courante consiste à utiliser un caractère de séparation.
- Utilisez la méthode String.split() pour décomposer une chaîne en plusieurs parties. Une chaîne ne contenant qu'un seul séparateur sera découpée en deux parties, une de chaque côté du séparateur. *Le séparateur lui-même ne compte pas.*

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Numéros de version : un gros problème pour la sérialisation

Vous avez constaté que les E/S Java sont vraiment très simples, surtout si vous vous en tenez aux combinaisons communication/traitement les plus courantes. Mais il y a un problème qui mérite attention.

Le contrôle des versions est crucial !

Si vous sérialisez un objet, vous devez disposer de la classe pour le désérialiser et l'utiliser. O.K., c'est évident. Mais il y a peut-être quelque chose de moins évident : que se passe-t-il si vous avez ***modifié la classe*** dans l'intervalle ? Aïe ! Imaginez que vous essayez de ramener un Chien à la vie et que l'une de ses variables d'instance (non temporaire) n'est plus un double mais une chaîne. Cela viole de façon outrageuse le principe de sécurité de type de Java. Mais ce n'est pas la seule modification qui peut provoquer une incompatibilité. Réfléchissez à ce qui suit :

Modifications qui peuvent empêcher la désérialisation :

Supprimer une variable d'instance.

Modifier le type d'une variable d'instance.

Transformer une variable d'instance en variable temporaire.

Modifier la position d'une classe dans la hiérarchie.

Transformer une classe sérialisable (quelle que soit sa place dans le graphe d'objets) en classe non sérialisable (en supprimant « implements Serializable » dans sa déclaration).

Transformer une variable d'instance en variable statique.

Modifications généralement sans problème :

Ajouter de nouvelles variables d'instance à la classe (quand les objets existants seront désérialisés, les variables d'instance qu'ils n'avaient pas quand ils ont été sérialisés auront une valeur par défaut).

Ajouter des classes à la hiérarchie d'héritage.

Supprimer des classes de la hiérarchie d'héritage.

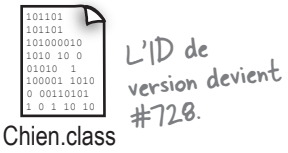
Modifier le niveau d'accès d'une variable d'instance n'a aucun effet sur la capacité de la désérialisation à affecter une valeur à la variable.

Transformer une variable d'instance temporaire en variable non temporaire (les variables autrefois temporaires des objets précédemment sérialisés auront simplement une valeur par défaut).

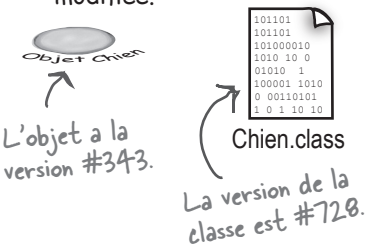
1. Vous écrivez une classe Chien.

2. Vous sérialisez un objet Chien en utilisant cette classe.

3. Vous modifiez la classe Chien.



4. Vous désérialisez l'objet Chien en utilisant la classe modifiée.



5. La désérialisation échoue. La JVM dit « On n'apprend pas à un vieux Chien à faire des grimaces ».

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Utiliser serialVersionUID

Chaque fois qu'un objet est sérialisé, il est «estampillé» (avec tous les autres objets du graphe) avec un ID dont le numéro correspond à sa classe. Cet ID se nomme serialVersionUID, et il est calculé à partir des informations sur la structure de la classe. Si la classe a été modifiée depuis la sérialisation quand l'objet est désérialisé, elle peut porter un serialVersionUID différent et la désérialisation échoue ! Mais vous pouvez contrôler ce processus.

Si vous pensez qu'il y a une possibilité QUELCONQUE que votre classe *évolue*, placez-y un ID de version.

Quand Java essaie de désérialiser un objet, il compare son serialVersionUID avec celui de la classe que la JVM utilise actuellement. Si par exemple une instance de Chien a été sérialisée avec l'ID 23 (en réalité un serialVersionUID est beaucoup plus long), la JVM va désérialiser l'objet en commençant par comparer son serialVersionUID avec celui de la classe Chien. Si les deux numéros ne correspondent pas, la JVM suppose que la classe n'est pas compatible avec l'objet sérialisé, et vous obtenez une exception.

La solution consiste donc à insérer un serialVersionUID dans votre classe. Si la classe évolue, le serialVersionUID demeure le même et la JVM dit «Parfait, la classe est compatible avec cet objet sérialisé», même si la classe a en fait changé.

Cela ne fonctionne que si vous êtes prudent dans vos modifications ! Autrement dit, vous prenez la responsabilité de tous les problèmes qui pourraient survenir quand un objet est ramené à la vie avec une version de classe plus récente.

Pour obtenir un serialVersionUID, utilisez l'outil en mode ligne de commande, serialver, qui accompagne le kit de développement Java.

```
Fichier Edition Fenêtre Aide serialKiller
% serialver Chien
Chien: static final long
serialVersionUID = -
5849794470654667210L;
```

Si vous pensez que votre classe pourrait évoluer après que quelqu'un a sérialisé les objets :

1. Utilisez l'outil serialver sur la ligne de commande pour obtenir un ID de version.

```
Fichier Edition Fenêtre Aide SerialKiller
% serialver Chien
Chien: static final long
serialVersionUID = -
5849794470654667210L;
```

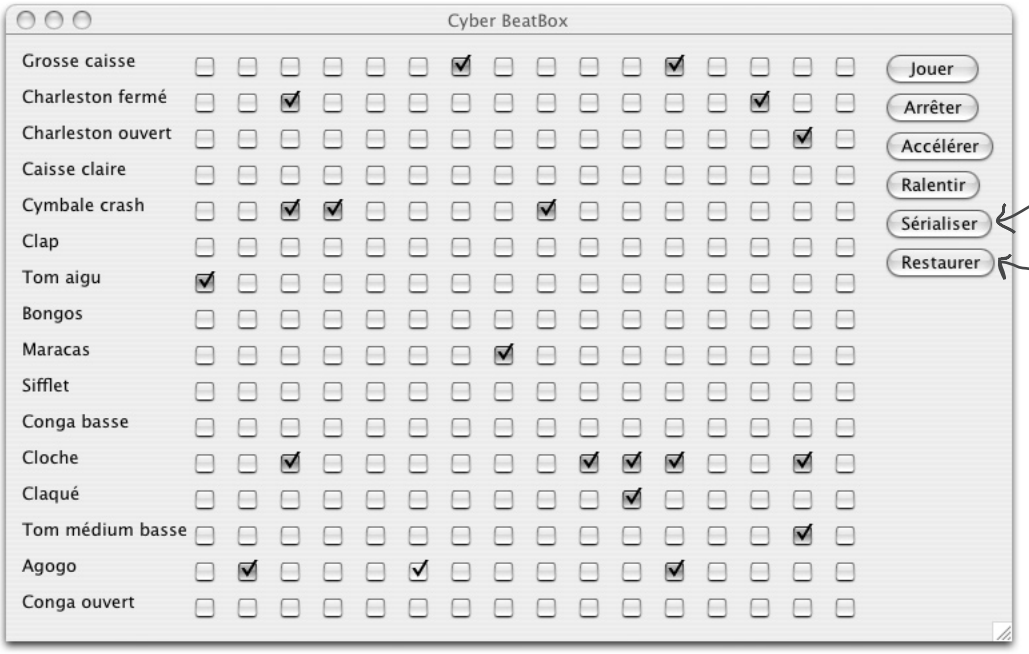
2. Collez le résultat dans votre classe.

```
public class Chien {

   static final long serialVersionUID =
                -5849794470654667210L;

   private String nom;
   private int taille;

   // code des méthodes
}
```

3. Quand vous modifierez la classe, veillez à prendre la responsabilité des conséquences de vos changements ! Vérifiez par exemple que votre nouvelle classe Chien peut gérer un vieux Chien désérialisé avec des valeurs par défaut pour les variables d'instance ajoutées à la classe *après* la sérialisation du Chien.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Recettes de code



Quand vous cliquez sur Sérialiser, le motif courant est sauvegardé.

"Restaurer" recharge le motif sauvegardé et réinitialise les cases à cocher.

Ajoutons à la BeatBox une fonction de sauvegarde et de restauration de votre motif favori.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Sauvegarder un motif de la BeatBox

Souvenez-vous : dans la BeatBox, un motif de batterie n'est rien d'autre qu'un groupe de cases à cocher. Quand c'est le moment de jouer la séquence, le code parcourt les cases à cocher pour déterminer quel son jouer sur chacun des seize temps. Pour sauvegarder un motif, il suffit donc de sauvegarder l'état des cases à cocher.

Nous pouvons créer un simple tableau de booléens qui contiendra l'état de chacune des 256 cases. Un tableau est sérialisable tant que les entités qu'il contient le sont, et nous n'aurons aucune difficulté à sauvegarder un tableau de booléens.

Pour recharger un motif, nous lisons cet unique tableau (en le désérialisant) et nous restaurons les cases à cocher. Vous avez déjà vu la majeure partie du programme dans la Recette de code dans laquelle nous avons construit l'IHM de la BeatBox, et nous n'étudierons dans ce chapitre que la partie sauvegarde et restauration.

Cette Recette de code nous prépare au prochain chapitre : au lieu d'écrire le motif dans un fichier, nous l'enverrons au serveur sur le réseau. Et au lieu de lire les motifs dans un fichier, nous les chargerons depuis le serveur chaque fois qu'un participant en envoie un.

Sérialiser un motif

Classe interne dans le code de la BeatBox.

```
public class EcouteEnvoi implements ActionListener {

    public void actionPerformed(ActionEvent a) {

        boolean[] etatCases = new boolean[256];

        for (int i = 0; i < 256; i++) {

            JCheckBox coche = (JCheckBox) listeCases.get(i);
            if (coche.isSelected()) {
               etatCases[i] = true;
            }
        }

        try {
           FileOutputStream fos = new FileOutputStream(new File("Cases.ser"));
           ObjectOutputStream oos = new ObjectOutputStream(fos);
           oos.writeObject(etatCases);
        } catch(Exception ex) {
            ex.printStackTrace();
        }

    } // fin de la méthode
} // fin de la classe interne
```

Tout se passe quand l'utilisateur clique sur le bouton et que l'ActionEvent se déclenche.

Créer un tableau de booléens qui contiendra l'état de chaque case.

Parcourir l'ArrayList des cases à cocher, lire l'état de chacune et l'ajouter au tableau de booléens.

Cette partie est vraiment du gâteau. Il suffit de sérialiser le tableau de booléens.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Restaurer un motif de la BeatBox

C'est pratiquement le processus inverse de la sauvegarde... lire le tableau de booléens et restaurer l'état des cases à cocher de l'IHM. Tout se passe quand l'utilisateur clique sur le bouton Restaurer.

Restaurer un motif

Autre classe interne à la classe BeatBox

```
public class EcouteLecture implements ActionListener {

    public void actionPerformed(ActionEvent a) {
       boolean[] etatCases = null;
       try {
           FileInputStream fis = new FileInputStream(new File("Cases.ser"));
           ObjectInputStream ois = new ObjectInputStream(fis);
           etatCases = (boolean[]) ois.readObject();

       } catch(Exception ex) {ex.printStackTrace();}


       for (int i = 0; i < 256; i++) {
          JCheckBox coche = (JCheckBox) listeCases.get(i);
          if (etatCases[i]) {
             coche.setSelected(true);
          } else {
             coche.setSelected(false);
           }
       }

       sequenceur.stop();
       construirePisteEtDemarrer();

      } // fin de la méthode
  } // fin de la classe interne
```

Lire le seul objet du fichier (le tableau de booléens) et le reconvertir en tableau de booléens (souvenez-vous que readObject() retourne une référence de type Object.

Maintenant, restaurer l'état de chaque case à cocher contenue dans l'ArrayList d'objets JCheckBox (listeCases).

Arrêter de jouer le morceau actuel et reconstruire la séquence en utilisant le nouvel été des cases à cocher de l'ArrayList.

À vos crayons

Cette version présente une énorme limite ! Quand vous cliquez sur le bouton Sérialiser, elle écrit automatiquement dans un fichier nommé Cases.ser (qui est créé s'il n'existe pas). Mais à chaque sauvegarde, le fichier est écrasé.

Améliorez les fonctions de sauvegarde et de restauration en incorporant une JFileChooser pour pouvoir nommer et sauver autant de motifs différents que vous en avez envie, et les restaurer à partir de *n'importe quel* fichier de motifs.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

À vos crayons

Seront-ils sauvés ?

Selon vous, quels sont les types ci-dessous qui sont sérialisables ? Sinon, pourquoi ? Pas de sens ? Risque de sécurité ? Spécifique à l'exécution courante de la JVM ? Essayez de trouver la bonne réponse sans consulter la documentation.

Type d'objet	Sérialisable ?	Si non, pourquoi ?
Object	Oui / Non	_____
String	Oui / Non	_____
File	Oui / Non	_____
Date	Oui / Non	_____
OutputStream	Oui / Non	_____
JFrame	Oui / Non	_____
Integer	Oui / Non	_____
System	Oui / Non	_____

LÉGAL OU NON ?

Entourez les fragments de code qui se compileront (en supposant qu'ils soient dans une classe légale).

```
FileReader fr = new FileReader();
BufferedReader br = new BufferedReader(fr);
```

```
FileOutputStream fos = new FileOutputStream(new File("Foo.ser"));
ObjectOutputStream oos = new ObjectOutputStream(fos);
```

```
BufferedReader br = new BufferedReader(new FileReader(fichier));
String ligne = null;
while ((ligne = br.readLine()) != null) {
    creerCarte(ligne);
}
```

```
ObjectInputStream ois = new ObjectInputStream(new FileOutputStream("Jeu.ser"));
Personnage unEncore = (Personnage) ois.readObject();
```


Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Ce chapitre nous a permis d'explorer le monde merveilleux des E/S Java. Déterminez si chacune des assertions suivantes est vraie ou fausse.

Vrai ou Faux

1. La sérialisation est appropriée quand des programmes non Java utiliseront les données sauvegardées.

2. On ne peut sauvegarder l'état des objets qu'en les sérialisant.

3. ObjectOutputStream est une classe qui sert à sauvegarder les objets sérialisés.

4. Les flots de traitement peuvent être utilisés seuls ou avec des flots de communication.

5. Un seul appel de writeObject() peut sauvegarder plusieurs objets.

6. Toutes les classes sont sérialisables par défaut.

7. Le modificateur transient permet de rendre les variables d'instance sérialisables.

8. Si une superclasse n'est pas sérialisable, la sous-classe ne peut pas l'être.

9. Quand les objets sont désérialisés, ils sont lus dans l'ordre inverse de leur écriture.

10. Quand un objet est désérialisé, son constructeur ne s'exécute pas.

11. La sérialisation et la sauvegarde dans un fichier texte peuvent lancer des exceptions.

12. On peut chaîner des BufferedWriters à des FileWriters.

13. Les objets File peuvent représenter des fichiers, mais pas des répertoires.

14. On ne peut pas forcer l'écriture d'un buffer tant qu'il n'est pas plein.

15. Les lectures et les écritures dans un fichier peuvent être bufferisées.

16. La méthode split() inclut les séparateurs dans le tableau résultant.

17. *Toute* modification d'une classe endommage les objets précédemment sérialisés de cette classe.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Le frigo

Celui-ci est assez délicat, et nous en avons fait un jeu au lieu d'un exercice. Réorganisez les fragments de code pour en faire un programme Java qui tourne et produit le résultat ci-dessous. Vous n'aurez peut être pas besoin de tous les fragments et certains serviront peut-être plusieurs fois !

```
class Donjons implements Serializable {
```

```
try {
```

```
FileOutputStream fos = new
  FileOutputStream("dj.ser");
```

```
short getZ() {
  return z;
```

```
e.printStackTrace();
```

```
oos.close();
```

```
ObjectInputStream ois = new
  ObjectInputStream(fis);
```

```
int getX() {
  return x;
```

```
System.out.println(d.getX()+d.getY()+d.getZ());
```

```
FileInputStream fis = new
  FileInputStream("dj.ser");
```

```
public int x = 3;
transient long y = 4;
private short z = 5;
```

```
long getY() {
  return y;
```

```
class DonjonsTest {
```

```
ois.close();
```

```
import java.io.*;
```

```
fos.writeObject(d);
```

```
} catch (Exception e) {
```

```
d = (Donjons) ois.readObject();
```

```
ObjectOutputStream oos = new
  ObjectOutputStream(fos);
```

```
oos.writeObject(d);
```

```
public static void main(String [] args) {
  Donjons d = new Donjons();
```

Fichier Edition Fenêtre Aide Torture

```
% java DonjonsTest
12
8
```


Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Solutions des exercices

1. La sérialisation est appropriée quand des programmes non Java utiliseront les données sauvegardées.	**Faux**
2. On ne peut sauvegarder l'état des objets qu'en les sérialisant.	**Faux**
3. ObjectOutputStream est une classe qui sert à sauvegarder les objets sérialisés.	**Vrai**
4. Les flots de traitement peuvent être utilisés seuls ou avec des flots de communication.	**Faux**
5. Un seul appel de writeObject() peut sauvegarder plusieurs objets.	**Vrai**
6. Toutes les classes sont sérialisables par défaut.	**Faux**
7. Le modificateur transient permet de rendre les variables d'instance sérialisables.	**Faux**
8. Si une superclasse n'est pas sérialisable, la sous-classe ne peut pas l'être.	**Faux**
9. Quand les objets sont désérialisés, ils sont lus dans l'ordre inverse de leur écriture.	**Faux**
10. Quand un objet est désérialisé, son constructeur ne s'exécute pas.	**Vrai**
11. La sérialisation et la sauvegarde dans un fichier texte peuvent lancer des exceptions.	**Vrai**
12. On peut chaîner des BufferedWriters à des FileWriters.	**Vrai**
13. Les objets File peuvent représenter des fichiers, mais pas des répertoires.	**Faux**
14. On ne peut pas forcer l'écriture d'un buffer tant qu'il n'est pas plein.	**Faux**
15. Les lectures et les écritures dans un fichier peuvent être bufferisées.	**Vrai**
16. La méthode split() inclut les séparateurs dans le tableau résultant.	**Faux**
17. *Toute* modification d'une classe endommage les objets précédemment sérialisés de cette classe.	**Faux**

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

```
import java.io.*;

class Donjons implements Serializable {
  public int x = 3;
  transient long y = 4;
  private short z = 5;
  int getX() {
    return x;
  }
  long getY() {
    return y;
  }
  short getZ() {
    return z;
  }
}

class DonjonsTest {
  public static void main(String [] args) {
    Donjons d = new Donjons();
    System.out.println(d.getX() + d.getY() + d.getZ());
    try {
      FileOutputStream fos = new FileOutputStream("dj.ser");
      ObjectOutputStream oos = new ObjectOutputStream(fos);
      oos.writeObject(d);
      oos.close();
      FileInputStream fis = new FileInputStream("dj.ser");
      ObjectInputStream ois = new ObjectInputStream(fis);
      d = (Donjons) ois.readObject();
      ois.close();
    } catch (Exception e) {
      e.printStackTrace();
    }
    System.out.println(d.getX() + d.getY() + d.getZ());
  }
}
```

Fichier Edition Fenêtre Aide Echappement

```
% java DonjonsTest
12
8
```


Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

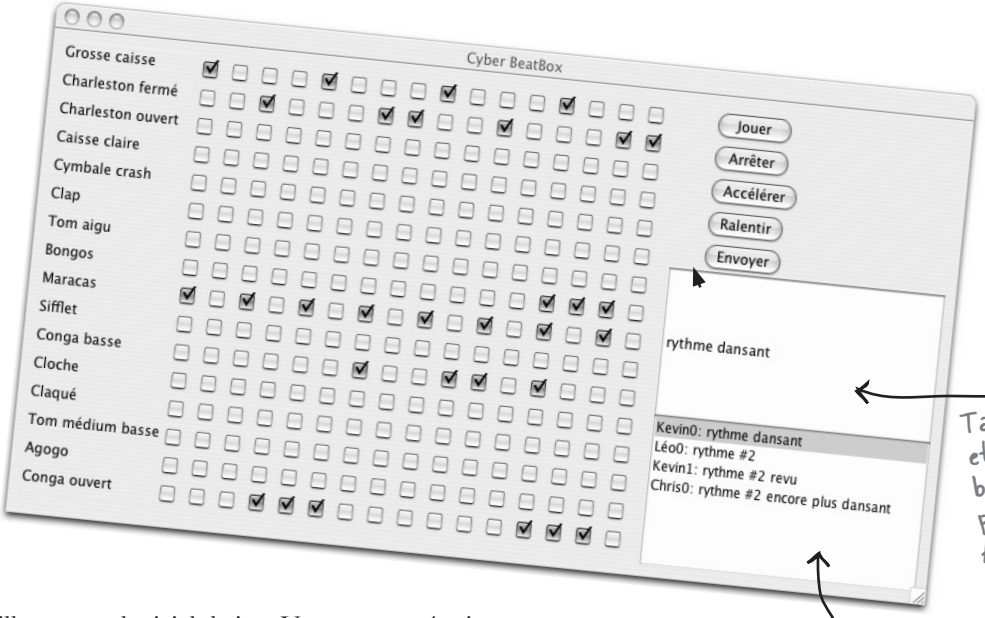
Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Établir une connexion

Connectez-vous au monde extérieur. Votre programme Java peut communiquer avec un autre programme résidant sur une autre machine. C'est facile. Tous les détails techniques de la connexion réseau sont pris en charge par la bibliothèque java.net. L'un des gros avantages de Java, c'est que l'émission et la réception de données sur un réseau fonctionne pratiquement comme les E/S avec un flot de communication légèrement différent en bout de chaîne. Si vous avez un BufferedReader, vous pouvez *lire*. Et le BufferedReader se moque totalement de savoir si les données viennent directement d'un fichier ou sont passées par un câble Ethernet. Dans ce chapitre, nous allons nous connecter au monde extérieur avec des sockets. Nous créerons des sockets *client*. Nous créerons des sockets serveur. Nous créerons des clients et des serveurs. Et nous les ferons communiquer. Avant la fin de ce chapitre, vous aurez un client de discussion *multithread* pleinement fonctionnel. Avons-nous dit *multithread* ? Oui, maintenant vous *allez* apprendre le secret de la communication : comment parler à Paul en écoutant Julie.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

BeatBox et discussion en temps réel

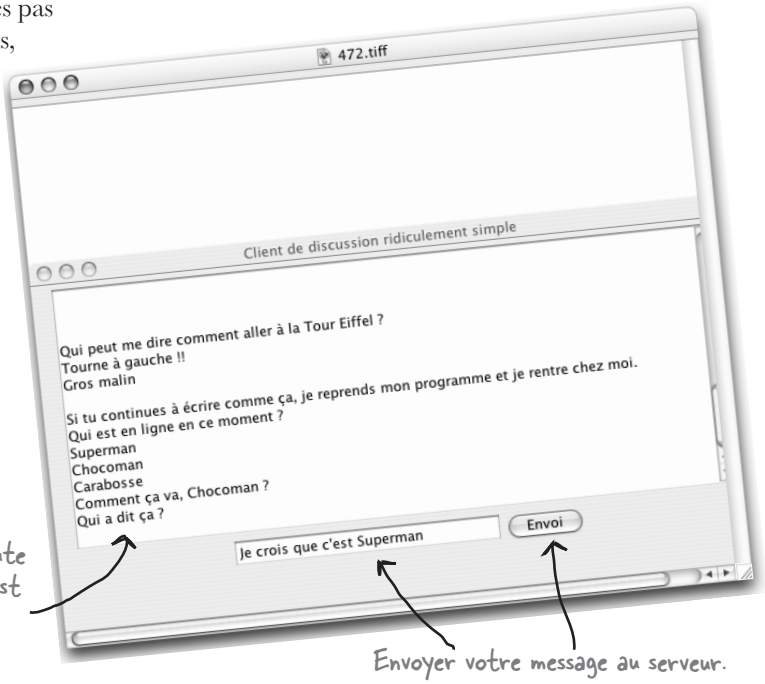


Tapez un message et cliquez sur le bouton Envoyer pour l'envoyer AVEC votre motif actuel.

Cliquer sur un message reçu charge le motif qui l'accompagne.

Vous travaillez sur un logiciel de jeu. Vous et votre équipe êtes en train de concevoir le son pour chaque partie du jeu. Avec une version «discussion» de la Beat Box, vous pouvez collaborer — vous pouvez envoyer un message accompagné d'un motif que chaque membre recevra. Vous n'êtes pas limité à la *lecture* des messages des autres participants, mais vous pouvez charger et *jouer* un motif rien qu'en cliquant dans la zone des messages entrants.

Dans ce chapitre, nous allons apprendre tout ce qui est nécessaire pour écrire un tel client de dicussion. Nous allons même apprendre deux ou trois choses sur la construction d'un *serveur*. Nous allons garder la version complète pour la Recette de code, mais nous allons commencer par écrire un client ridiculement simple qui envoie des messages texte et un serveur élémentaire qui en reçoit.



Vous pouvez avoir des conversations totalement authentiques et d'une haute tenue intellectuelle. Chaque message est envoyé à tous les participants.

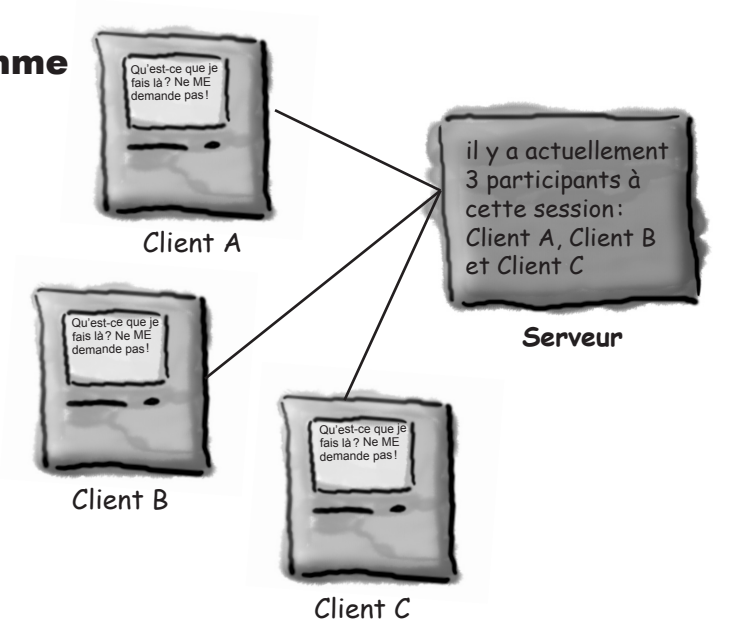
Envoyer votre message au serveur.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Vue d'ensemble du programme

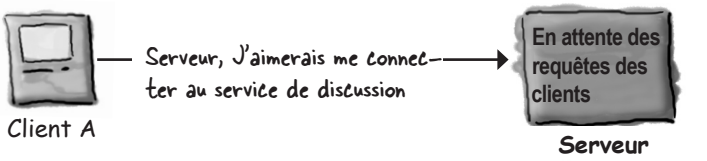
Le Client doit connaître le Serveur.

Le Serveur doit connaître TOUS les Clients.

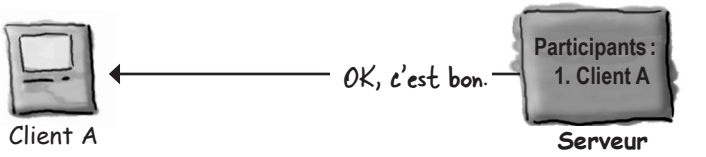


Comment ça marche :

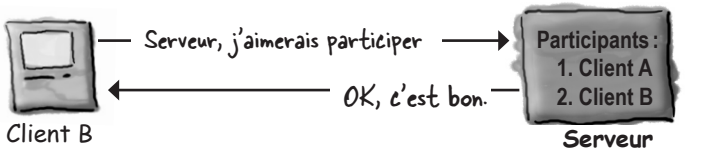
1. Le client se connecte au serveur



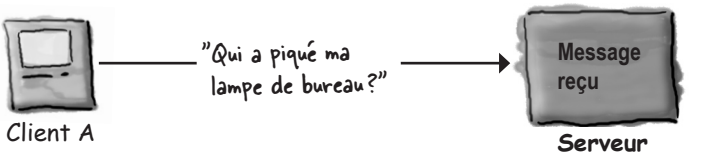
2. Le serveur établit une connexion et ajoute le client à la liste des participants



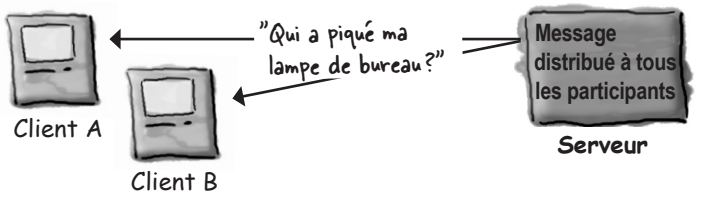
3. Un autre client se connecte



4. Le client A envoie un message au service



5. Le serveur distribue le message à TOUS les participants (dont l'émetteur d'origine)



Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Se connecter, émettre et recevoir

Nous avons trois choses à apprendre pour faire fonctionner le client :

1) Comment établir la **connexion** initiale entre le client et le serveur.

2) Comment **envoyer** des messages *au* serveur.

3) Comment **recevoir** des messages *du serveur.*

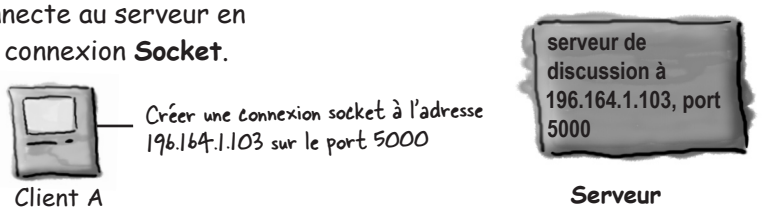
Il y a beaucoup de détails de bas niveau pour que ceci fonctionne. Mais nous avons de la chance, parce que le package réseau de l'API Java (java.net) en fait du gâteau pour les programmeurs. Vous travaillerez beaucoup plus sur les IHM que sur le réseau ou les E/S.

Mais ce n'est pas tout.

Tapi derrière le simple client de discussion, se trouve un problème auquel nous n'avons pas encore été confrontés dans ce livre : faire deux choses en même temps. L'établissement d'une connexion est une opération unique : elle réussit ou elle échoue. Après quoi, chaque participant veut pouvoir *émettre et recevoir des messages* **simultanément** (*via* le serveur). Hmmmm… voilà qui va demander un peu de réflexion, mais nous y viendrons dans quelques pages.

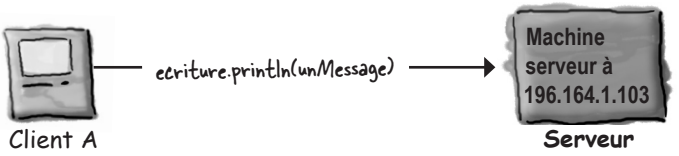
1 Connexion

Le client se connecte au serveur en établissant une connexion **Socket**.



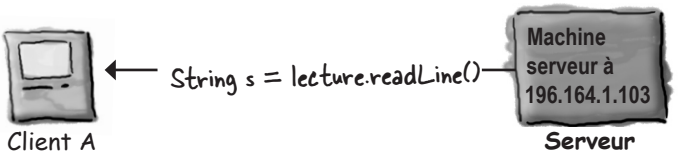
2 Émission

Le client **envoie** un message au serveur.



3 Réception

Le client **reçoit** un message du serveur.



Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Créer une connexion réseau

Pour nous connecter à une autre machine, il nous faut une connexion Socket. Une Socket (la classe java.net.Socket) est un objet qui représente une connexion réseau entre deux machines. Qu'est-ce qu'une connexion ? Une relation entre deux machines dans laquelle deux logiciels se rencontrent. Plus important encore, ces deux logiciels savent comment communiquer. Autrement dit comment échanger des bits.

Heureusement, nous ne nous occupons pas des détails de bas niveau, parce qu'ils sont gérés beaucoup plus bas, dans la «pile réseau». Si vous ne savez pas ce qu'est la «pile réseau», ne vous inquiétez pas. C'est juste une façon de considérer les couches que doivent traverser les informations (les bits) d'un programme Java qu'une JVM exécute sur un OS donné pour accéder à une autre machine et revenir, en passant par exemple par des câbles Ethernet. *Bien sûr, quelqu'un doit mettre les mains dans le cambouis. Mais ce n'est pas vous. Ce quelqu'un, c'est une combinaison d'un programme spécifique à l'OS et de l'*API réseau Java. Votre tâche est une tâche de très haut niveau, et elle est étonnamment simple. Prêt ?

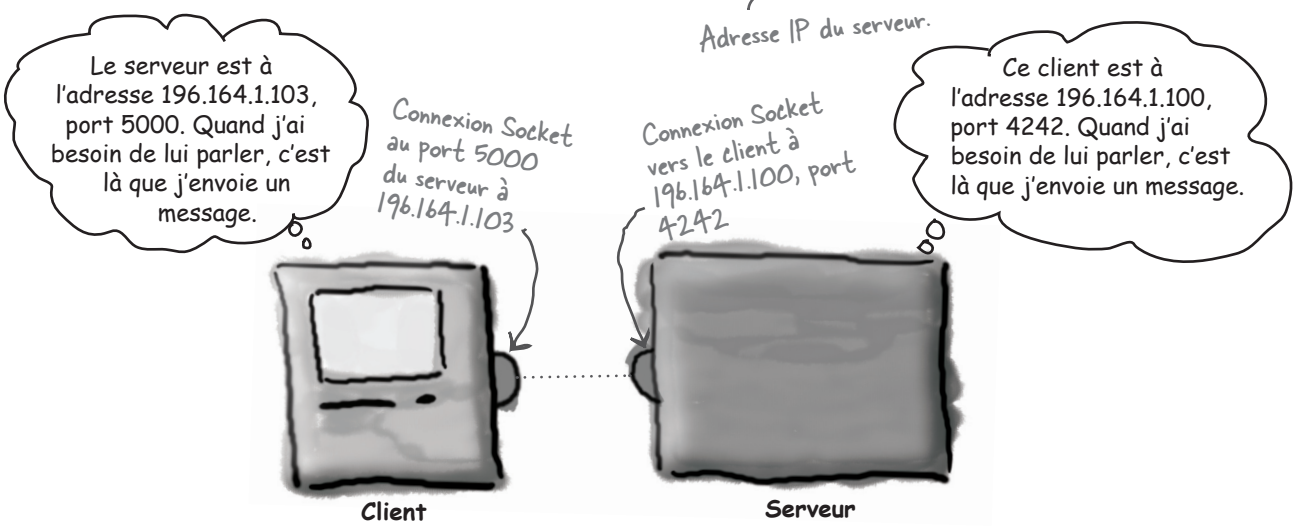
Pour créer une connexion, vous devez avoir deux informations sur le serveur : qui il est et sur quel port il s'exécute.

Autrement dit, son adresse IP et son numéro de port TCP.

Numéro de port TCP.

```
Socket socketDiscussion = new Socket("196.164.1.103", 5000);
```

Adresse IP du serveur.



Une connexion Socket signifie que deux machines possèdent des informations l'une sur l'autre, notamment leur emplacement sur le réseau (adresse IP) et leur numéro de port TCP.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Un port TCP n'est qu'un numéro. Un nombre de 16 bits qui identifie un programme spécifique sur le serveur.

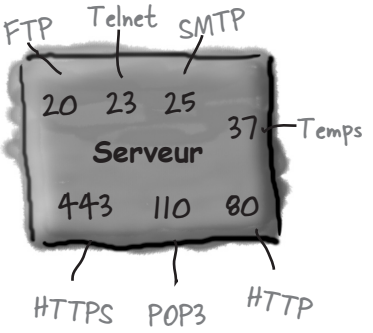
Votre serveur Web (HTTP) s'exécute sur le port 80. C'est un standard. Si vous avez un serveur Telnet, il s'exécute sur le port 23. Un serveur FTP ? 20. Un serveur de courrier POP3 ? 110. SMTP ? 25. Le serveur de temps est sur le port 37. Représentez-vous les numéros de port comme des identifiants uniques. Ils représentent une connexion logique à un logiciel particulier qui s'exécute sur le serveur. C'est tout. Vous pouvez désosser votre machine, vous ne trouverez pas de port TCP. D'abord, il y en a 65536 sur un serveur (de 0 à 65535). De toute évidence, aucun ne représente un emplacement auquel brancher un équipement physique. Ce ne sont que des nombres qui représentent une application.

Sans numéros de ports, le serveur n'aurait aucun moyen de savoir à quelle application un client veut se connecter. Et comme chaque application peut avoir son propre protocole, imaginez les problèmes que vous auriez sans ces identifiants. Que se passerait-il par exemple si votre navigateur web se connectait au serveur de courrier POP3 au lieu d'un serveur HTTP ? Le serveur de courrier ne saura pas analyser une requête ! Et même s'il savait le faire, il ne saurait pas comment y répondre.

Quand vous écrivez un programme serveur, vous incluez du code qui spécifie sur quel numéro de port vous voulez qu'il s'exécute (nous verrons comment faire en Java un peu plus loin dans ce chapitre). Dans le programme de discussion que nous écrivons, nous avons choisi 5000. Arbitrairement. Et parce qu'il répond à un critère : c'est un nombre compris entre 1024 et 65535. Pourquoi 1024 ? Parce que les nombres 0 à 1023 sont réservés pour les services du type dont nous venons de parler.

Et si vous écrivez des services (des programmes côté serveur) qui s'exécutent sur un réseau d'entreprise, vous devez vérifier avec votre administrateur système quels ports sont déjà occupés. Il vous dira par exemple que vous ne pouvez utiliser aucun numéro inférieur à 3000. En tous cas, si vous tenez à vos abattis, vous n'affecterez pas de numéros de ports à la légère. À moins qu'il ne s'agisse de votre réseau personnel, à la maison, auquel cas il suffit de demander à vos enfants.

Numéros de ports TCP réservés aux applications serveur courantes.



Un serveur peut avoir jusqu'à 65536 services différents qui s'exécutent, un par port.

Les numéros de ports TCP de 0 à 1023 sont réservés. Ne les utilisez pas dans vos propres programmes !*

Notre serveur de discussion utilise le port 5000. Nous avons juste choisi un nombre compris entre 1024 et 65535.

*En fait, vous *pouvez* les utiliser, mais l'administrateur système de votre entreprise vous tuera probablement.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

il n'y a pas de Questions stupides

Q : Comment connaît-on le numéro de port du serveur auquel on veut se connecter ?

R : Tout dépend si le programme est un service « bien connu ». Si vous essayez de vous connecter à un service bien connu comme ceux de la page précédente (HTTP, SMTP, FTP, etc.) vous pouvez le rechercher sur l'Internet (Google « Well-Known TCP Port ») ou encore demander à votre administrateur système préféré.

Mais si le programme n'en fait pas partie, celui ou celle qui a déployé le service doit vous le dire. Demandez-le lui. En général, si quelqu'un écrit un service réseau et veut que d'autres écrivent des clients, il publiera l'adresse IP, le numéro de port et le protocole du service. Si par exemple vous écrivez un client pour un serveur de jeu de go, vous pouvez visiter l'un des sites de serveurs de go pour trouver des informations sur la façon d'écrire un client particulier pour ce serveur.

Q : Peut-il y avoir plusieurs programmes s'exécutant sur un seul port ? Autrement dit, deux applications sur le même serveur peuvent-elles avoir le même numéro de port ?

R : Non ! Si vous essayez d'associer un programme à un port déjà occupé, vous obtiendrez une BindException. Associer un programme à un port signifie simplement lancer une application serveur et lui dire de s'exécuter sur un port donné. Nous y reviendrons quand nous passerons à la partie serveur de ce chapitre.

L'adresse IP est le centre commercial.

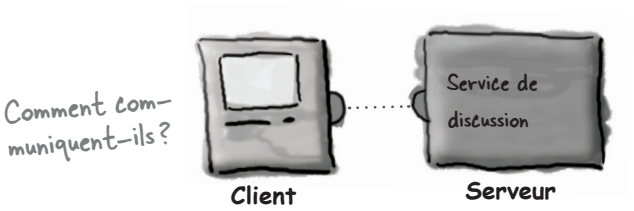
Spécifier l'adresse IP équivaut à spécifier un centre commercial ou une galerie marchande, par exemple "Le Forum des Halles".

Le numéro de port est une boutique spécifique.

Le numéro de port est comme une boutique particulière, par exemple, "la boutique de CD de Max".

Gym du cerveau

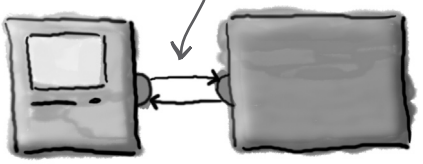
OK, vous avez une connexion Socket. Le client connaît l'adresse IP et le numéro de port TCP du serveur et inversement. Et maintenant ? Comment communiquez-vous sur cette connexion ? Autrement dit, comment transférez-vous des bits entre eux ? Imaginez le type de messages que votre client doit émettre et recevoir.



Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Pour lire des données sur une Socket, utilisez un BufferedReader

Pour communiquer sur une connexion Socket, on utilise des flots. De bon vieux flots d'E/S, comme ceux du chapitre précédent. L'une des caractéristiques les plus sympathiques de Java, c'est que peu lui importe en général ce à quoi un flot de traitement est connecté. Autrement dit, vous pouvez utiliser un BufferedReader exactement comme pour écrire dans un fichier. La seule différence, c'est que le flot de communication sous-jacent est maintenant un objet *Socket* et non un objet *File!*



1 Créer une connexion Socket au serveur

```
Socket socketDiscussion = new Socket("127.0.0.1", 5000);
```

Le numéro de port, que vous connaissez parce que nous vous avons DIT que 5000 était celui de notre serveur.

127.0.0.1 est l'adresse IP de "localhost", autrement dit l'hôte local sur laquelle ce code s'exécute. Vous pouvez l'utiliser pour tester votre client et votre serveur sur la même machine.

2 Créer un InputStreamReader chaîné au flot d'entrée (connexion) de la Socket

```
InputStreamReader stream = new InputStreamReader(socketDiscussion.getInputStream());
```

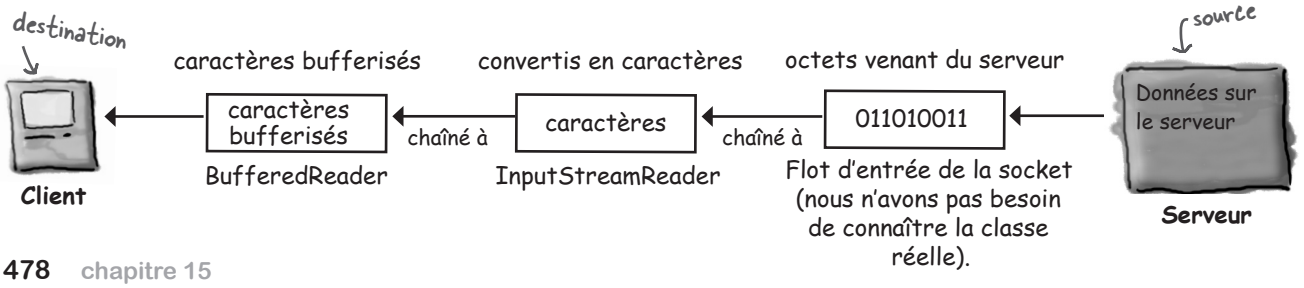
InputStreamReader est un "pont" entre un flot d'octets de bas niveau (comme celui qui vient de la Socket) et un flot de caractères de haut niveau (comme le BufferedReader qui termine le chaînage des flots).

Il suffit de DEMANDER à la socket un flot d'entrée! C'est un flot de communication de bas niveau, mais nous allons le chaîner à quelque chose de plus convivial.

3 Créer un BufferedReader et lire!

```
BufferedReader reader = new BufferedReader(stream);
String message = reader.readLine();
```

Chaîner le BufferedReader à l'InputStreamReader (qui a été chaîné au flot de communication que nous avons obtenu de la Socket.)



Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Pour écrire des données sur une Socket, utilisez un PrintWriter

Au dernier chapitre, nous n'avons pas utilisé un PrintWriter mais un BufferedWriter. Ici, nous avons le choix. Lorsqu'on écrit une chaîne à la fois, PrintWriter est la solution standard. Et vous allez reconnaître les deux méthodes clé de PrintWriter, print() et println() ! Exactement comme ce bon vieux System.out.

Cette partie est la même que celle de la page opposée — pour écrire au serveur, vous devez vous connecter.

1 Créer une connexion Socket au serveur

```
Socket socketDiscussion = new Socket("127.0.0.1", 5000);
```

2 Créer un PrintWriter chaîné au flot de sortie (connexion) de la Socket

```
PrintWriter writer = new PrintWriter(socketDiscussion.getOutputStream());
```

PrintWriter fonctionne comme un pont entre les caractères et les octets qu'il reçoit du flot de sortie de la Socket. En le chaînant à ce flot de sortie, nous pouvons écrire des chaînes de caractères sur la socket.

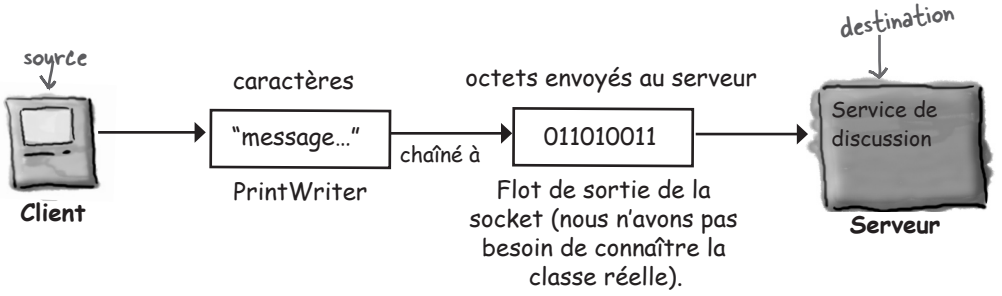
La Socket nous donne un flot de communication et nous le chaînons au PrintWriter en le passant à son constructeur.

3 Écrire (print) quelque chose

```
writer.println("mon message ");
writer.print("autre message");
```

println() ajoute un retour à la ligne à la fin de ce qu'il envoie.

print() n'ajoute pas de retour à la ligne.



Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Le conseil du jour (côté client)

Avant de commencer à construire l'application de discussion, commençons par quelque chose d'un peu plus simple. Docteur Bonconseil est un service qui vous donne des conseils pratiques et inspirés pour vous réconforter pendant ces longues journées de codage.

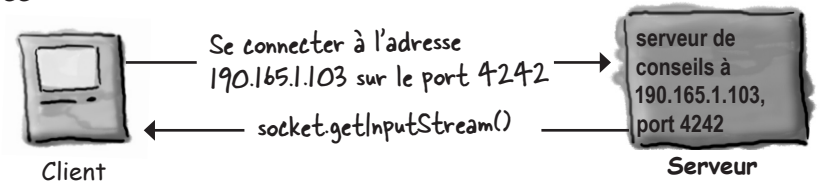
Nous allons construire pour le programme Docteur Bonconseil un client qui retire un message du serveur chaque fois qu'il se connecte.

Qu'attendez-vous ? Qui sait quelles opportunités vous avez manquées sans cette application.

Docteur Bonconseil

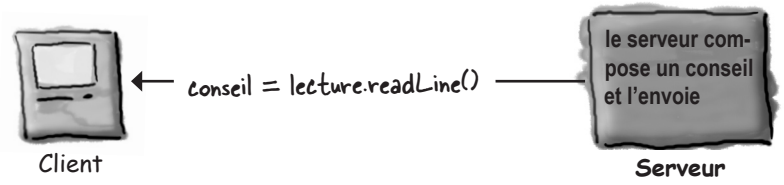
1 Se connecter

Le client se connecte au serveur et obtient un flot d'entrée



2 Lire

Le client lit un message du serveur



Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Le code de ConseilDuJourClient

Ce programme crée un objet Socket, puis un BufferedReader (avec l'aide d'autres flots), il lit une seule ligne de l'application serveur (qui s'exécute sur le port 4242).

```
import java.io.*;
import java.net.*;

public class ConseilDuJourClient {

   public void go() {
       try {
           Socket s = new Socket("127.0.0.1", 4242);

           InputStreamReader isr = new InputStreamReader(s.getInputStream());
           BufferedReader lecture = new BufferedReader(isr);



           String conseil = lecture.readLine();
           System.out.println("Le conseil du jour: " + conseil);

           lecture.close();

        } catch(IOException ex) {
           ex.printStackTrace();
        }
     }

   public static void main(String[] args) {
       ConseilDuJourClient client = new ConseilDuJourClient();
       client.go();
   }
}
```

La classe Socket est dans java.net

Prévenir les risques.

Créer une connexion Socket au programme qui s'exécute sur le port 4242, sur l'hôte sur lequel ce même code s'exécute ('localhost').

Chaîner un BufferedReader à un InputStreamReader au flot d'entrée de la Socket.

Cette méthode readLine() est EXACTEMENT la même que pour un BufferedReader chaîné à un objet FILE. Autrement dit, dès que vous appelez une méthode de BufferedReader, peu importe la provenance des caractères.

Ferme TOUS les flots.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

À vos crayons

Testez ce que vous avez mémorisé sur les flots et les classes qui permettent de lire et d'écrire sur une socket. Essayez de ne pas regarder la page suivante !

Pour ***lire*** du texte sur une socket :

Client

Représentez la chaîne de flots que le client utilise pour lire sur le serveur.

source

Serveur

Pour ***envoyer*** du texte à une socket :

Client

Représentez la chaîne de flots que le client utilise pour envoyer quelque chose au serveur.

destination

Serveur

À vos crayons

Remplissez les blancs :

Quelles sont les deux informations que le client doit connaître pour se connecter à un serveur ? _____ _____

Quels sont les numéros de ports TCP réservés pour les « well-known services » comme HTTP et FTP ? _____

VRAI ou FAUX : La plage de numéros de ports réservés valides peut être représenté par un short ? _____

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Écrire un serveur simple

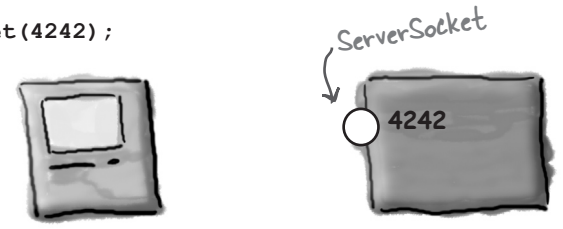
Que faut-il donc pour écrire une application serveur ? Uniquement une paire de sockets. Oui, une paire, deux sockets. Une ServerSocket qui attend les requêtes du client (quand un client crée une nouvelle Socket()) et une bonne vieille socket Socket pour communiquer avec le client.

Comment ça marche

1. L'application crée une ServerSocket, sur un port donné

```
ServerSocket serverSock = new ServerSocket(4242);
```

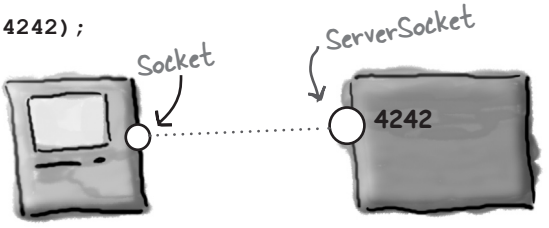
Cette ligne lance l'application qui écoute les requêtes entrantes du client sur le port 4242.



2. Le client crée une connexion Socket à l'application

```
Socket sock = new Socket("190.165.1.103", 4242);
```

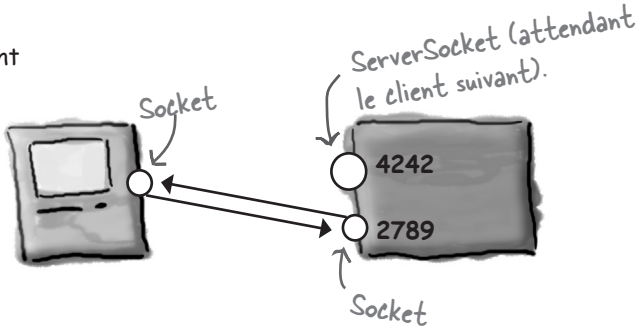
Le client connaît l'adresse IP et le numéro de port (publiés ou fournis par la personne qui a configuré l'exécution du service sur ce port).



3. Le serveur crée une nouvelle Socket pour communiquer avec ce client

```
Socket sock = serverSock.accept();
```

La méthode accept() bloque (ne fait rien) en attendant la connexion Socket du client. Quand un client essaie enfin de se connecter, elle envoie une bonne vieille Socket (sur un port *différent*) qui sait comment communiquer avec le client (elle connaît l'adresse IP et le numéro de port du *client*). Comme elle est sur un port différent, la ServerSocket peut continuer à attendre d'autres clients.



Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Le code de ConseilDuJourServeur

Ce programme crée une ServerSocket et attend les requêtes des clients. Quand il en reçoit une (parce que le client a dit new Socket() pour cette application), le serveur crée une nouvelle connexion Socket à ce client. Puis il crée un PrintWriter (en utilisant le flot de sortie de la socket) et envoie un message au client.

(N'oubliez pas que c'est l'éditeur de code qui gère les retours à la ligne. N'appuyez jamais sur la touche Entrée au milieu d'une chaîne de caractères.

```
import java.io.*;
import java.net.*;
```

N'oubliez pas les importations.

```
public class ConseilDuJourServeur {
```

Les conseils viennent de ce tableau.

```
   String[] listeConseils = {"Mangez moins de pizzas.", "Super, ce jean moulant. Non,
ça ne grossit PAS.", "Un seul mot: hallucinant", "Soyez honnête pour une fois. Dites à
votre patron ce que vous pensez *vraiment*.", "Vous pourriez aller chez le coiffeur."};

   public void go() {

       try {
           ServerSocket serverSock = new ServerSocket(4242);
```

Cette application "écoute" les requêtes des clients sur le port 4242 de la machine sur laquelle ce code s'exécute.

Le serveur exécute une boucle sans fin, attendant (et servant) les requêtes des clients).

```
           while(true) {

              Socket sock = serverSock.accept();
```

La méthode accept() bloque jusqu'à réception d'une requête, puis elle retourne une Socket (sur un port anonyme) pour communiquer avec le client.

```
              PrintWriter ecriture = new PrintWriter(sock.getOutputStream());
              String conseil = getConseil();
              ecriture.println(conseil);
              ecriture.close();
              System.out.println(conseil);
           }

       } catch(IOException ex) {
           ex.printStackTrace();
       }
   } // fin de la méthode go()
```

Maintenant, nous utilisons la connexion Socket au client pour créer un PrintWriter et lui envoyer un conseil (println()). Puis nous fermons la Socket parce que nous en avons terminé avec ce client.

```
   private String getConseil() {
      int random = (int) (Math.random() * listeConseils.length);
      return listeConseils[random];
   }

   public static void main(String[] args) {
       ConseilDuJourServeur serveur = new ConseilDuJourServeur();
       serveur.go();
   }
}
```


Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Gym du cerveau

Comment le serveur sait-il communiquer avec le client ?

Le client connaît l'adresse IP et le numéro de port du serveur, mais comment le serveur peut-il créer une connexion Socket au client (et des flots d'entrées/sorties) ?

Pensez à quand / comment / où le serveur prend connaissance du client.

il n'y a pas de Questions stupides

Q : Le code du serveur de conseils de la page précédente présente une limite TRÈS sérieuse — on dirait qu'il ne sait gérer qu'un client à la fois !

R : Oui, c'est vrai. Il ne peut accepter d'autre requête tant qu'il n'a pas fini avec le client courant et exécuté l'itération suivante de la boucle sans fin (là où il s'arrête à l'appel de accept() jusqu'à l'arrivée d'une requête, après quoi il crée une Socket vers le nouveau client puis relance le processus).

Q : Laissez-moi reformuler le problème : comment créer un serveur qui gère plusieurs clients simultanément ??? Ce code ne marcherait jamais avec un serveur de discussion par exemple.

R : Ah, c'est très simple, vraiment. On utilise des threads séparés, et on passe chaque nouvelle Socket client à un nouveau thread. C'est précisément ce que nous allons apprendre maintenant!

POINTS D'IMPACT

- Les applications client et serveur communiquent sur une connexion Socket.
- Une Socket représente une connexion entre deux applications qui peuvent (ou non) s'exécuter sur deux machines physiques différentes.
- Un client doit connaître l'adresse IP (ou le nom de domaine) du serveur et le numéro de port TCP de l'application.
- Un numéro de port TCP est un nombre de 16 bits non signé affecté à un service spécifique. Les numéros de ports permettent à différents clients de se connecter à la même machine, mais communiquent avec différentes applications s'exécutant sur cette machine.
- Les numéros de ports de 0 à 1023 sont réservés pour les « well-known services », notamment HTTP, FTP, SMTP, etc.
- Un client se connecte à un serveur en créant une socket :
  `Socket s = new Socket("127.0.0.1", 4200);`
- Une fois connecté, un client peut obtenir des flots d'E/S de la socket. Ce sont des flots de communication de bas niveau.
  `sock.getInputStream();`
- Pour lire les données du serveur, créez un BufferedReader, chaîné à un InputStreamReader, qui est chaîné à son tour au flot d'entrée de la Socket.
- InputStreamReader est un « pont » qui lit des octets et les transforme en caractères. Il sert principalement de chaînon intermédiaire entre le BufferedReader (haut niveau) et le flot d'entrée de la socket (bas niveau)
- Pour envoyer du texte au serveur, créez un PrintWriter chaîné directement au flux de sortie de la Socket. Appelez ensuite la méthode print() ou la méthode println().
- Les serveurs utilisent une ServerSocket qui attend les requêtes des clients sur un numéro de port particulier.
- Quand une ServerSocket reçoit une requête, elle l'accepte en créant une connexion Socket avec le client.

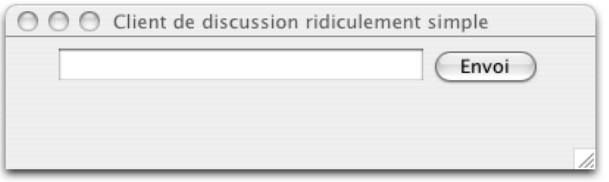
Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Écrire un client de discussion

Nous allons créer l'application client de discussion en deux phases. Tout d'abord, nous allons créer une version qui envoie des messages au serveur mais ne lit pas les messages des autres participants (une interprétation excitante et mystérieuse du concept de salon de dicussion).

Puis nous écrirons une version plus complète qui pourra émettre et recevoir des messages.

Version un : envoi seulement



Tapez un message, puis cliquez sur « Envoi » pour l'envoyer au serveur. Comme nous ne RECEVRONS pas de messages du serveur dans cette version, la zone de texte ne défile pas.

Plan du code

```
public class SimpleClientDiscussionA {

   JTextField sortants;
   PrintWriter ecriture;
   Socket sock;

   public void go() {
        // créer l'IHM et enregistrer un auditeur pour le bouton Envoi
        // appeler la méthode installerReseau()
   }

   private void installerReseau() {
        // créer un objet Socket puis un objet PrintWriter
        // affecter le PrintWriter à la variable d'instance ecriture
   }

   public class EcouteEnvoi implements ActionListener {
      public void actionPerformed(ActionEvent ev) {
         // lire le texte dans le champ de texte et
         // l'envoyer au serveur en utilisant ecriture (un PrintWriter)
      }
   }  // fin de la classe interne EcouteEnvoi

} // fin de la classe externe
```


Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

```
import java.io.*;
import java.net.*;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
```

importations pour les flots d'E/S (java.io), la Socket (java.net) et les classes de l'IHM.

```
public class SimpleClientDiscussionA {

    JTextField sortants;
    PrintWriter ecriture;
    Socket sock;

    public void go() {
        JFrame cadre = new JFrame("Client de discussion ridiculement simple");
        JPanel panneau = new JPanel();
        sortants = new JTextField(20);
        JButton boutonEnvoi = new JButton("Envoi");
        boutonEnvoi.addActionListener(new EcouteEnvoi());
        panneau.add(sortants);
        panneau.add(boutonEnvoi);
        cadre.getContentPane().add(BorderLayout.CENTER, panneau);
        installerReseau();
        cadre.setSize(400,500);
        cadre.setVisible(true);
    } // fin de la méthode go()
```

Construire l'interface. Rien de nouveau et rien de lié aux E/S ni au réseau.

```
    private void installerReseau() {
        try {
           sock = new Socket("127.0.0.1", 5000);
           ecriture = new PrintWriter(sock.getOutputStream());
           System.out.println("Connexion établie");
        } catch(IOException ex) {
           ex.printStackTrace();
        }
    } // fin de la méthode installerReseau()
```

Nous utilisons localhost pour que vous puissiez tester client et serveur sur une seule machine.

C'est là que nous créons la Socket et le PrintWriter (appelé par la méthode go() juste avant l'affichage de l'IHM).

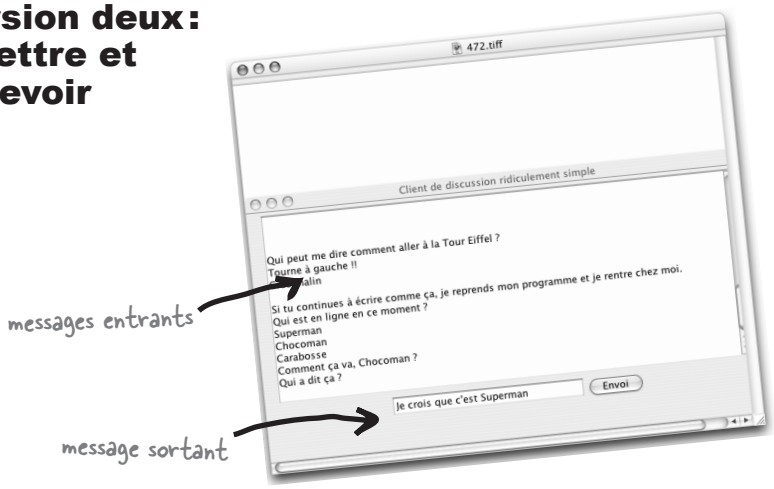
```
   public class EcouteEnvoi implements ActionListener {
      public void actionPerformed(ActionEvent ev) {
          try {
             ecriture.println(sortants.getText());
             ecriture.flush();

          } catch(Exception ex) {
             ex.printStackTrace();
          }
          sortants.setText("");
          sortants.requestFocus();
      }
    }  // fin de la classe interne EcouteEnvoi

    public static void main(String[] args) {
       new SimpleClientDiscussionA().go();
    }
 } // fin de la classe externe
```

Maintenant, l'écriture réelle. Souvenez-vous que le flot d'écriture est chaîné au flot d'entrée de la Socket : le résultat de chaque instruction println() va directement au serveur.

Si vous voulez essayer maintenant, tapez le code prêt à l'emploi du serveur listé en fin de chapitre. Lancez d'abord le serveur dans une nouvelle fenêtre. Puis ouvrez-en une autre pour lancer ce client.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Version deux : émettre et recevoir



Dès que le serveur reçoit un message, il l'envoie à tous les clients participants. Quand un client envoie un message, il n'apparaît pas dans la zone d'affichage des messages entrants tant que le serveur ne l'a pas envoyé à tout le monde.

Question : *COMMENT* recevez-vous des messages du serveur ?

Ce devrait être facile. Quand vous installez la partie réseau, vous créez également un flot d'entrée (probablement un BufferedReader). Puis vous lisez les messages avec readLine().

Question plus difficile : *QUAND* recevez-vous les messages du serveur ?

Réfléchissez-y. Quelles sont les solutions ?

1. **Option un : Scruter le serveur toutes les 20 secondes.**

   Pour : Eh bien, c'est faisable.

   Contre : Comment le serveur sait-il ce que vous avez lu et ce que vous n'avez pas lu ? Il devrait stocker les messages au lieu de les distribuer puis de les oublier. Et pourquoi 20 secondes ? Un tel délai affecte l'utilisabilité. Mais si vous réduisez ce délai, vous risquez d'accéder au serveur pour rien. Inefficace.

2. **Option deux : Lire quelque chose sur le serveur chaque fois que l'utilisateur envoie un message.**

   Pour : Faisable, très facile.

   Contre : Stupide. Pourquoi choisir un moment aussi arbitraire pour lire les messages ? Et si l'utilisateur est un traînard et n'envoie rien ?

3. **Option trois : Lire les messages dès qu'ils sont envoyés par le serveur.**

   Pour : Très efficace, meilleure utilisabilité.

   Contre : Comment faites vous pour faire deux choses en même temps ? Où placeriez-vous ce code ? Il vous faudrait une boucle quelque part qui attendrait en permanence les messages du serveur. Mais où irait-elle ? une fois que vous avez lancé l'IHM, rien ne se passe tant qu'un événement n'est pas déclenché par un de ses composants.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

En Java, vous POUVEZ vraiment marcher et mâcher du chewing-gum en même temps.

Vous savez maintenant que nous allons choisir l'option 3.

Il nous faut quelque chose qui s'exécute en permanence et qui vérifie si l'utilisateur envoie des messages *sans pour autant interrompre l'interaction de l'utilisateur avec l'IHM !* Quelque chose qui s'exécute en coulisse et reste à l'écoute pendant que l'utilisateur tape allégrement de nouveaux messages ou parcourt les messages entrants.

Autrement dit, nous avons besoin d'un nouveau fil d'exécution, d'une pile séparée : d'un nouveau *thread*.

Nous voulons que tout ce que nous avons écrit dans la version un fonctionne de la même manière, tandis qu'un nouveau processus s'exécutera en parallèle, lira les données en provenance du serveur et les affichera dans la zone des messages entrants.

Enfin, pas vraiment. À moins que vous n'ayez une machine multiprocesseur, chaque nouveau thread Java n'est pas réellement un processus séparé. Mais il en a toutes les apparences.

Multithreading en Java.

Le multithreading est intégré au langage Java. Et la création d'un nouveau thread est simple comme bonjour :

```
Thread t = new Thread();
t.start();
```

C'est tout. En créant un nouvel objet Thread, vous avez créé un fil d'exécution séparé, qui possède sa propre pile.

Sauf qu'il y a un problème.

En réalité, ce thread ne fait rien. C'est pourquoi il « meurt » virtuellement à l'instant même où il naît. Et quand un thread meurt, sa nouvelle pile disparaît. Fin de l'épisode.

Il nous manque donc un composant clé — la tâche du thread. Autrement dit, il nous faut la portion de code que vous voulez faire exécuter par un thread séparé.

En Java, le multithreading signifie que nous devons nous occuper du *thread* et de la *tâche* qui est *exécutée* par le thread. Et nous devons également nous intéresser à la classe Thread *qui fait partie de* java.lang. (Rappelez-vous, java.lang est ce package que vous importez implicitement et qui contient les classes les plus fondamentales pour le langage, notamment String et System.)

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Java a plusieurs threads mais une seule classe Thread

Nous écrivons *thread* avec un petit « t » et **Thread** avec un grand « T ». Quand vous lisez *thread*, nous parlons d'un fil d'exécution séparé, autrement dit d'une pile d'appels distincte. Quand vous lisez **Thread**, pensez à la convention de nommage de Java. Qu'est-ce qui commence par une majuscule en Java ? Les noms des classes et des interfaces. En l'occurrence, **Thread** est une classe du package java.lang. Un objet **Thread** représente un *thread :* vous créez un objet de type **Thread** chaque fois que vous voulez lancer un nouveau fil d'exécution.

Un thread est un « fil d'exécution » séparé. Autrement dit une pile d'appels distincte.

Thread est une classe Java class qui représente un thread.

Pour créer un thread, créez un Thread.

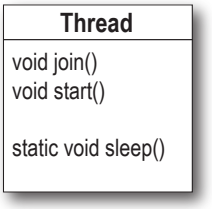
thread

thread principal

autre thread lancé par le code

Un thread (avec un petit « t ») est un fil d'exécution séparé, ce qui implique une pile d'appels distincte. Toute application Java lance un thread principal — celui qui place la méthode main() au sommet de la pile. La JVM est responsable de lancer le thread principal (et les autres threads qu'elle choisit de créer, y compris celui du ramasse-miettes). En tant que programmeur, vous pouvez écrire du code pour lancer vos propres threads.

Thread



La classe java.lang.Thread

Thread (avec un grand « T ») est une classe qui représente un fil d'exécution. Elle possède des méthodes permettant de lancer un thread, de joindre deux threads et de mettre un thread en sommeil. (Ce ne sont pas les seules méthodes, mais celles-ci sont capitales, et nous allons les utiliser maintenant).

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Que signifie avoir plusieurs piles d'appels ?

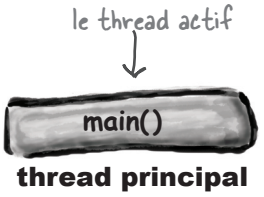
Avec plusieurs piles d'appels, on a l'*impression* que plusieurs choses se passent en même temps. En réalité, seul un vrai système multiprocesseur peut faire plusieurs choses en même temps, mais les threads Java permettent de faire semblant. Autrement dit, l'exécution passe d'une pile à l'autre si rapidement que vous avez le sentiment que les threads s'exécutent tous en même temps. Souvenez-vous : Java n'est rien d'autre qu'un processus qui s'exécute au-dessus de l'OS sous-jacent. Mais une fois que Java prend son tour, qu'est-ce que la JVM *exécute réellement* ? Quel est le code binaire qui s'exécute ? Celui qui est au sommet de la pile courante ! Et il ne faut pas plus de 100 millisecondes pour que le code qui s'exécute à un instant donné passe à une autre méthode sur une pile *différente*.

L'une des attributions d'un thread est de conserver la trace de quelle instruction (quelle méthode) est en cours sur sa pile.

Le déroulement est le suivant :

1 **La JVM appelle la méthode main().**

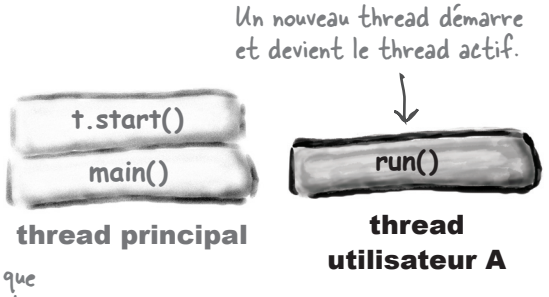
```
public static void main(String[] args) {
...
}
```



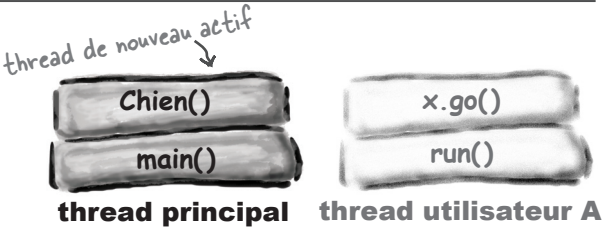
2 **main() lance un nouveau thread. Le thread principal est temporairement gelé pendant que l'autre commence à s'exécuter.**

```
Runnable r = new MonThreadTache();
Thread t = new Thread(r);
t.start();
Chien c = new Chien();
```

Vous allez apprendre ce que ça signifie dans un moment...



3 **La JVM permute entre le nouveau thread (thread utilisateur A) et le thread d'origine jusqu'à ce que les deux se terminent.**



Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Comment lancer un nouveau thread :

1 Créer un objet Runnable (la tâche du thread)

```
Runnable threadTache = new MonRunnable();
```

Runnable est une interface que vous allez découvrir page suivante. Vous allez écrire une classe qui implémente Runnable et c'est dans cette classe que vous allez définir le travail que le thread va effectuer. Autrement dit, la méthode qui sera exécutée depuis la nouvelle pile d'appels du thread.

2 Créer un objet Thread (le travailleur) et lui passer l'objet Runnable (la tâche)

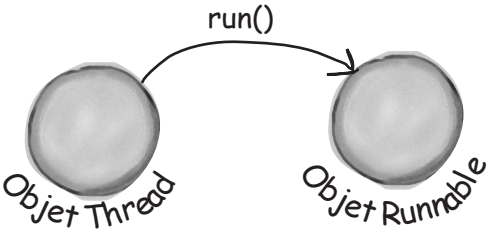
```
Thread monThread = new Thread(threadTache);
```

On passe l'objet Runnable au constructeur du Thread. Le nouvel objet Thread sait ainsi quelle méthode placer en bas de la nouvelle pile d'appels — la méthode run() de Runnable.

3 Lancer le thread

```
monThread.start();
```

Rien ne se passe tant que vous n'appelez pas la méthode start() du thread. C'est à ce moment-là que vous cessez d'avoir simplement une instance de Thread pour avoir un nouveau fil d'exécution. Quand le nouveau thread démarre, il prend la méthode run() de l'objet Runnable et la place en bas de la nouvelle pile d'appels.



Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Tout thread a un travail à faire. une méthode à placer sur la nouvelle pile.

Runnable est à un thread ce qu'un travail est à un travailleur. L'objet Runnable est le travail que le thread est censé effectuer.

Un objet Runnable contient la méthode qui va aller en base de la pile d'appels du thread : run().

Un objet Thread a besoin d'un travail. Un travail que le thread exécutera quand il sera lancé. Ce travail est en réalité la première méthode qui va sur la pile du nouveau thread, et elle doit toujours ressembler à ceci :

```
public void run() {
   // code à exécuter par le nouveau thread
}
```

L'interface Runnable ne définit qu'une seule méthode : public void run(). (Souvenez-vous : puisque c'est une interface, la méthode est publique indépendamment de la façon dont vous la tapez.)

Mais comment le thread sait-il quelle méthode placer en base de la pile ? Parce que Runnable définit un contrat. Parce que Runnable est une interface. Le travail d'un thread peut être défini dans toute classe qui implémente Runnable. Le thread ne se soucie que d'une chose : que vous passiez au constructeur du Thread un objet d'une classe qui implémente Runnable.

Quand vous passez un Runnable à un constructeur de Thread, vous fournissez en réalité au Thread un moyen d'obtenir une méthode run(). Vous donnez au thread un travail à faire.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Pour donner un travail à votre thread, implémentez l'interface Runnable

Runnable étant dans le package java.lang, vous n'avez pas besoin de l'importer.

```
public class MonRunnable implements Runnable {

    public void run() {
        go();
    }

    public void go() {
        fairePlus();
    }

    public void fairePlus() {
        System.out.println("sommet de la pile");
    }
}
```

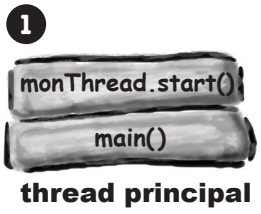
2

Runnable n'a qu'une seule méthode à implémenter : public void run() (sans arguments). C'est là que vous placez le TRAVAIL que le thread doit exécuter. C'est la méthode qui va en bas de la nouvelle pile.

```
class TestThread {

    public static void main (String[] args) {

        Runnable threadTache = new MonRunnable();
        Thread monThread = new Thread(threadTache);

    1   monThread.start();

        System.out.println("retour à main()()");
    }
}
```

Transmettre la nouvelle instance de Runnable au constructeur du nouveau Thread. Cela indique au thread quelle méthode placer en bas de la nouvelle pile. Autrement dit, la première méthode que le nouveau thread exécutera

Vous n'avez pas de nouveau fil d'exécution tant que vous n'appelez pas start() sur l'instance de Thread. Un thread n'est pas vraiment un thread tant qu'il n'est pas lancé. Auparavant, ce n'est qu'une instance de Thread, comme tout autre objet, mais le thread n'est pas fonctionnel.

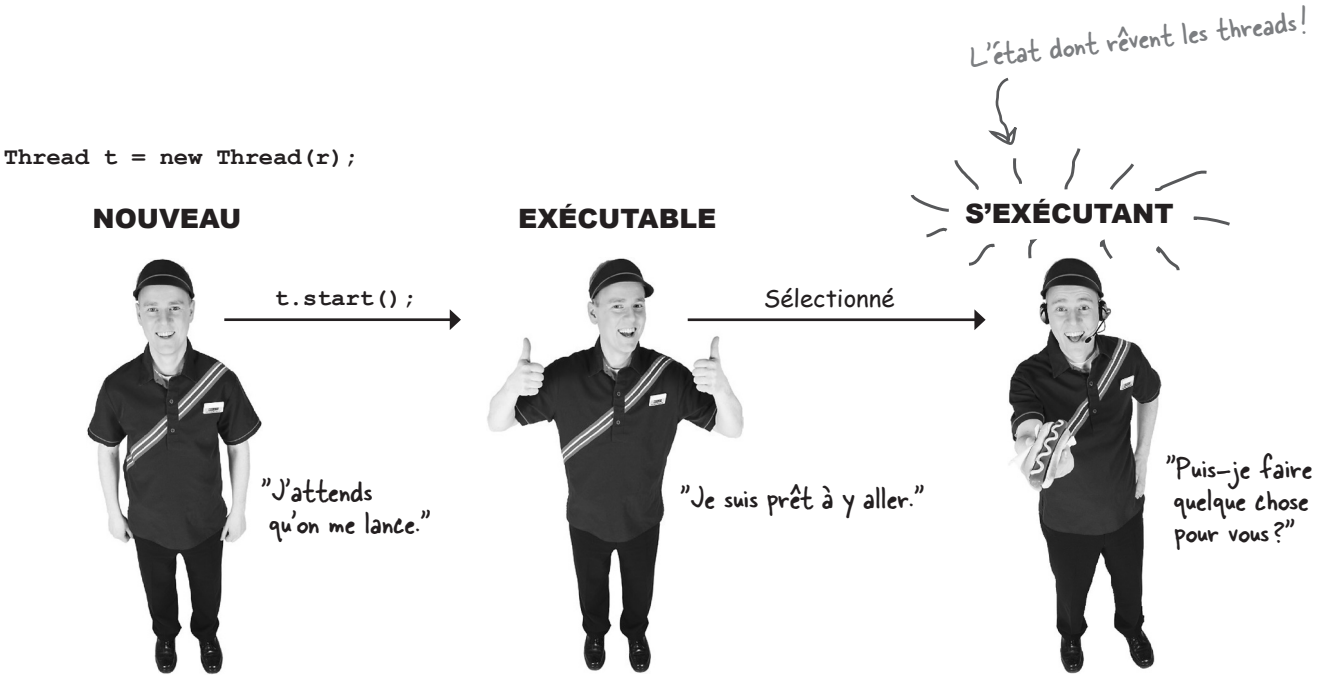


Gym du cerveau

Selon vous, quel sera le résultat de l'exécution de la classe TestThread ? (nous le saurons dans quelques pages)

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Les trois états d'un nouveau thread



`Thread t = new Thread(r);`

Une instance de Thread a été créée mais elle n'est pas lancée. Autrement dit, il y a un *objet* Thread, mais pas de *fil d'exécution*.

`t.start();`

Quand vous lancez le thread, il passe à l'état exécutable (runnable). Cela signifie que le thread est prêt et attend la chance de sa vie : être sélectionné pour l'exécution. À ce stade, ce thread a une nouvelle pile d'appels.

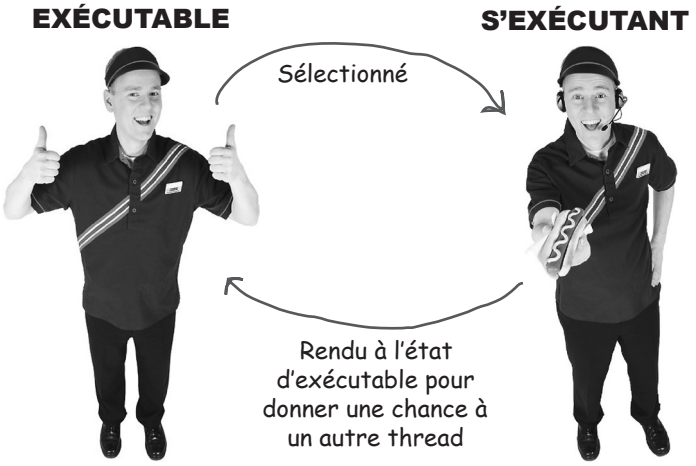
Voilà l'état dont rêvent tous les threads ! Être l'élu. LE thread qui s'exécute. Seul l'ordonnanceur de threads de la JVM peut prendre cette décision. Vous pouvez parfois *influencer* cette décision, mais vous ne pouvez pas forcer un thread exécutable à s'exécuter. Quand un thread s'exécute, ce thread (et SEULEMENT ce thread) a une pile d'appels active, et c'est la méthode qui est au sommet de la pile qui s'exécute.

Mais ce n'est pas tout. Une fois que le thread devient exécutable, il peut permuter entre les deux états, être exécutable, s'exécuter… et avoir un état supplémentaire : *temporairement non exécutable* (alias « bloqué »).

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Une boucle type entre deux états

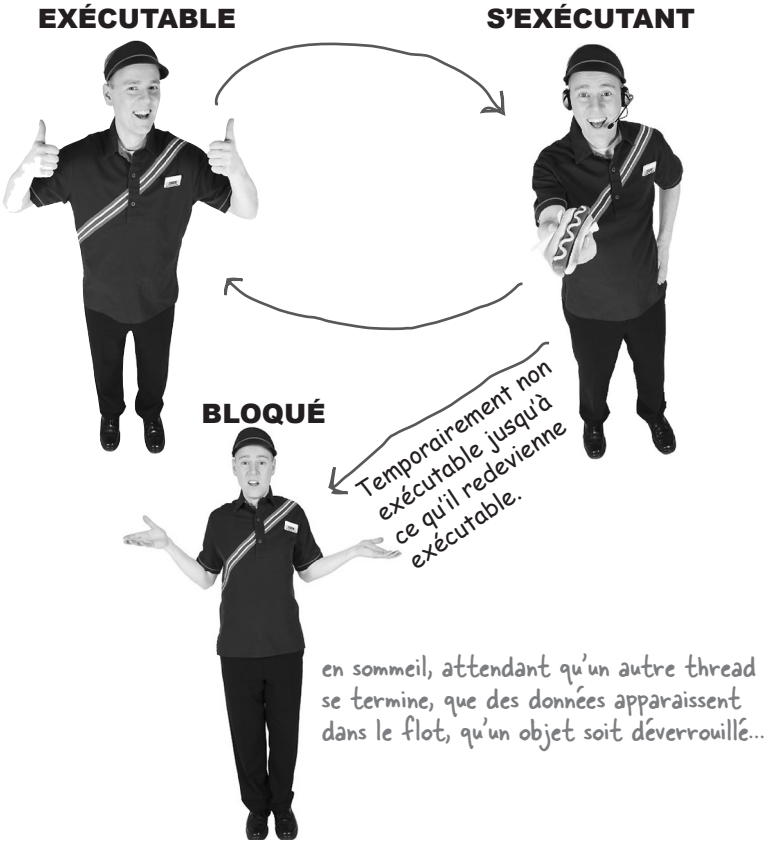
En général, un thread oscille entre les deux états (il est exécutable ou il s'exécute), tandis que la JVM les sélectionne à tour de rôle, en en sélectionnant un pour l'exécution puis le stoppant pour qu'un autre thread ait sa chance.



Un thread peut être rendu temporairement non exécutable

L'ordonnanceur peut bloquer un thread qui s'exécute pour une variété de raisons. Par exemple, le thread peut exécuter du code qui lit dans le flot d'entrée d'une Socket mais il n'y a pas de données à lire. L'ordonnanceur arrête donc le thread jusqu'à ce que des données deviennent disponibles. Ou bien le code peut avoir dit au thread de se mettre en sommeil (sleep()). Ou encore le thread peut être en attente parce qu'il a essayé d'appeler une méthode sur un objet mais que cet objet a été «verrouillé». Dans ce cas, il ne peut pas continuer tant que l'objet n'a pas été déverrouillé par le thread dans lequel il se trouve.

Toutes ces situations (et bien d'autres) peuvent faire qu'un thread devient temporairement non exécutable.



Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

L'ordonnanceur de threads

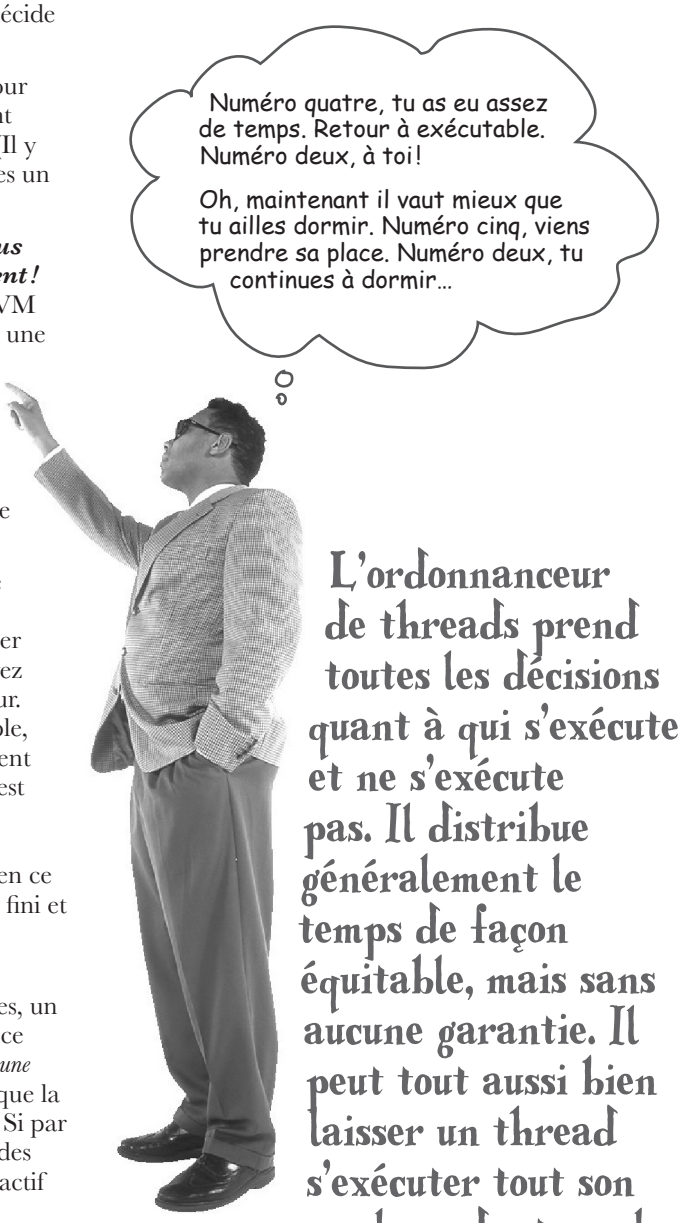
L'ordonnanceur prend toutes les décisions sur l'état des threads : lesquels sont exécutables, lequel s'exécute et quand (et dans quelles circonstances) un thread cesse de s'exécuter. Il détermine le thread qui doit être actif, pendant combien de temps il doit l'être, et que faire des threads quand il décide qu'ils ne s'exécutent plus.

Vous ne contrôlez pas ce processus. Il n'y a pas d'API pour appeler des méthodes sur l'ordonnanceur. Plus important encore, il n'y a aucune garantie sur l'ordonnancement ! (Il y a quelques « quasi-garanties », mais elles sont elles-mêmes un peu floues.)

Moralité : ***ne comptez pas sur l'ordonnanceur si vous voulez que votre programme s'exécute correctement !*** Les implémentations de l'ordonnanceur varient d'une JVM à l'autre, et deux exécutions d'un même programme sur une même machine peuvent vous donner des résultats différents. L'une des pires erreurs qu'un programmeur débutant en Java puisse commettre consiste à tester un programme multithread sur une seule machine, puis de supposer que l'ordonnanceur fonctionnera toujours de la même manière, indépendamment de la plate-forme sur laquelle le programme s'exécute.

Mais qu'est-ce que ça signifie en termes d'indépendance du code par rapport à la plate-forme. Cela signifie que, pour qu'un programme Java multithread puisse s'exécuter correctement sur n'importe quelle machine, vous ne devez pas vous appuyer sur le comportement de l'ordonnanceur. Autrement dit, vous ne pouvez pas dépendre, par exemple, de la décision de l'ordonnanceur d'accorder équitablement du temps d'exécution aux différents threads. Même si c'est hautement improbable aujourd'hui, votre programme pourrait se retrouver à tourner sur une JVM dont l'ordonnanceur dirait « O.K. thread cinq, tu es actif, et, en ce qui me concerne, tu peux rester là jusqu'à ce que tu aies fini et que ta méthode run() se termine ».

Le secret quasi universel est le *sommeil*. Oui, le sommeil. Mettre en *sommeil*, ne serait-ce que quelques millisecondes, un thread qui s'exécute le force à s'arrêter et donne sa chance à un autre thread. La méthode sleep() du thread fournit *une* garantie : un thread en sommeil ne s'exécutera pas tant que la durée programmée pour son sommeil n'aura *pas* expiré. Si par exemple vous dites à votre thread de dormir deux secondes (2 000 millisecondes), il ne redeviendra jamais le thread actif tant que les deux secondes ne seront pas écoulées.



L'ordonnanceur de threads prend toutes les décisions quant à qui s'exécute et ne s'exécute pas. Il distribue généralement le temps de façon équitable, mais sans aucune garantie. Il peut tout aussi bien laisser un thread s'exécuter tout son saoul pendant que les autres « dépérissent ».

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Un exemple de l'imprévisibilité de l'ordonnanceur…

L'exécution de ce code sur une machine…

```
public class MonRunnable implements Runnable {

   public void run() {
      go();
   }

   public void go() {
      fairePlus();
   }

   public void fairePlus() {
      System.out.println("sommet de la pile");
   }
}


class TestThread {

   public static void main (String[] args) {

      Runnable threadTache = new MonRunnable();
      Thread monThread = new Thread(threadTache);

      monThread.start();

      System.out.println("retour à main()()");
   }
}
```

…a produit ce résultat

Fichier Edition Fenêtre Aide ChoisissezMoi

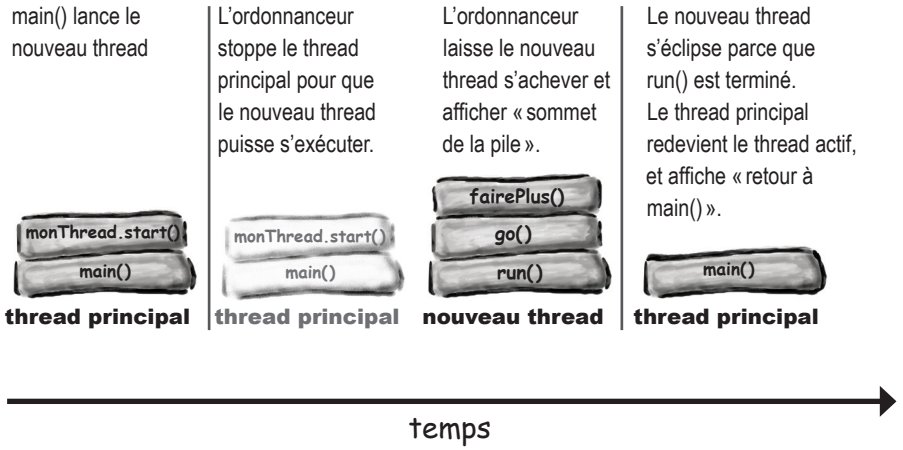
```
% java TestThread
retour à main()
sommet de la pile
% java TestThread
sommet de la pile
retour à main()
% java TestThread
sommet de la pile
retour à main()
% java TestThread
sommet de la pile
retour à main()
% java TestThread
sommet de la pile
retour à main()
% java TestThread
sommet de la pile
retour à main()
% java TestThread
retour à main()
sommet de la pile
```

Remarquez comment l'ordre change de façon aléatoire. C'est parfois le nouveau thread qui termine le premier et parfois le thread principal.

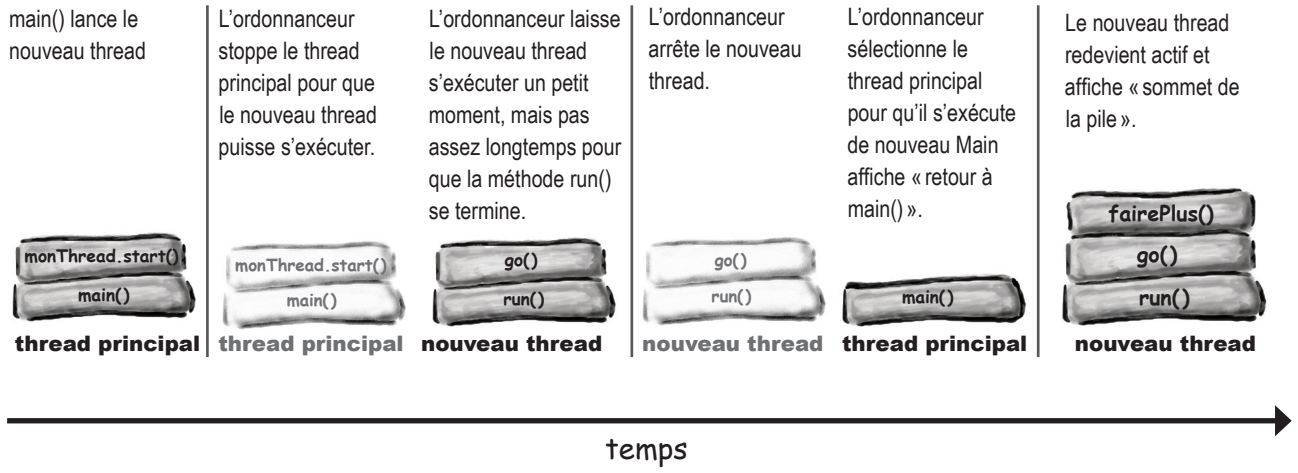
Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Pourquoi les résultats sont-ils différents?

Parfois, le code s'exécute comme ceci:



Et parfois il s'exécute comme cela:



Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

il n'y a pas de Questions stupides

Q : J'ai vu des exemples dans lesquels on n'utilise pas d'implémentation séparée de Runnable, mais où on crée simplement une sous-classe de Thread et on redéfinit la méthode run() de Thread. De cette façon, on appelle le constructeur de Thread sans argument quand on crée le nouveau thread :

Thread t = new Thread(); // non Runnable

R : Oui, c'est *possible*, mais réfléchissez-y en termes d'objet. Quel est l'objectif du sous-classement ? N'oubliez pas que nous parlons ici de deux notions différentes — le Thread et la tâche du thread. D'un point de vue objet, ce sont deux activités très distinctes qui appartiennent à deux classes séparées. La seule raison de vouloir sous-classer/étendre la classe Thread, c'est de créer un nouveau type plus spécifique de Thread. Autrement dit, si vous vous représentez le Thread comme le travailleur, ne créez pas de sous-classe de Thread à moins que vous n'ayez besoin d'un comportement de travailleur spécifique. Mais s'il ne vous faut qu'une nouvelle tâche à exécuter par un Thread/travailleur, implémentez Runnable dans une classe séparée, spécifique à la tâche et non au travailleur.

Il s'agit là d'un problème de conception, et non de performance ou de langage. Il est parfaitement légal de sous-classer Thread et de redéfinir la méthode run(), mais c'est rarement une bonne idée.

Q : Peut-on réutiliser un objet Thread ? Peut on lui donner une nouvelle tâche et le relancer en appelant de nouveau start() ?

R : Non. Une fois que la méthode run() d'un thread est terminée, il est impossible de le relancer. En fait, à ce stade, le thread passe à un état dont nous n'avons pas encore parlé — il est ***mort***. Il a fini sa méthode run() et ne sera jamais relancé. L'objet Thread est peut-être encore sur le tas, comme un objet vivant sur lequel vous pourriez appeler d'autres méthodes, mais il a perdu pour toujours ce qui fait de lui un thread. Autrement dit, il ne possède plus de pile d'appels distincte et l'objet Thread n'est plus un *thread*. Ce n'est plus qu'un objet comme les autres objets.

Il existe des modèles de conception pour créer un pool de threads que vous pouvez utiliser pour exécuter différentes tâches. Mais on ne ressuscite pas un thread mort.

POINTS D'IMPACT

- En Java, un thread avec un « t » minuscule est un fil d'exécution séparée.
- Tout thread Java possède sa propre pile d'appels.
- Un Thread avec un « T » majuscule est la classe java.lang.Thread. Un objet Thread représente un thread, un fil d'exécution.
- Un Thread a besoin d'un travail à faire. Le travail d'un Thread est une instance de classe qui implémente l'interface Runnable.
- L'interface Runnable n'a qu'une seule méthode, run(). C'est la méthode qui sera placée en bas de la nouvelle pile d'appels. Autrement dit, la première méthode qui s'exécutera dans le nouveau thread .
- Pour lancer un nouveau thread, vous devez passer un objet Runnable au constructeur de Thread.
- Un thread est NOUVEAU quand vous avez instancié un objet Thread mais que vous n'avez pas encore appelé start().
- Quand vous lancez un thread (en appelant la méthode start() de l'objet Thread), une nouvelle pile est créée, avec la méthode run() de Runnable en bas de la pile. Le thread est maintenant EXÉCUTABLE, et attend d'être sélectionné.
- Un thread s'EXÉCUTE quand l'ordonnanceur de la JVM l'a choisi pour être le thread actif. Sur une machine monoprocesseur, il ne peut y avoir qu'un thread actif à la fois.
- Parfois, un thread peut être BLOQUÉ (temporairement non exécutable), parce qu'il attend des données, parce qu'il est en sommeil ou parce qu'il attend le déverrouillage d'un objet.
- L'ordonnancement des threads n'est pas garanti fonctionner de façon uniforme, et vous ne pouvez pas être certain que les threads prennent leur tour. Vous pouvez influencer ce processus en mettant périodiquement vos threads en sommeil.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Mettre un thread en sommeil

L'une des meilleures façons d'aider vos threads à prendre leur tour est de les mettre périodiquement en sommeil. Il suffit d'appeler la méthode statique sleep() et de lui transmettre la durée de sommeil en millisecondes.

Par exemple :

```
Thread.sleep(2000);
```

stoppe l'exécution du thread et l'empêche d'être exécutable pendant deux secondes. Il ne peut pas redevenir le thread actif tant qu'au moins deux secondes en se sont pas écoulées.

Malheureusement, la méthode sleep() lance une InterruptedException, une exception vérifiée par le compilateur, et tous les appels de sleep() doivent être encapsulés dans un bloc try/catch (ou déclarés). Un appel de sleep() ressemble donc à ceci :

```
try {
   Thread.sleep(2000);
} catch(InterruptedException ex) {
    ex.printStackTrace();
}
```

Le sommeil de votre thread ne sera probablement *jamais* interrompu : l'exception figure dans l'API pour prendre en charge un mécanisme de communications entre threads que pratiquement personne n'utilise dans le monde réel. Mais comme vous devez quand même obéir à la loi « gérer ou déclarer », vous devez vous habituer à placer vos appels de sleep() dans un bloc try/catch.

Vous savez maintenant que votre thread ne se réveillera pas *avant* la fin de la période spécifiée, mais est-il possible qu'il se réveille après l'expiration du « timer » ? Oui et non. Mais cela n'a pas de réelle importance, ***parce qu'un thread qui se réveille revient toujours à l'état exécutable !*** Il ne se réveille pas automatiquement au bout de la durée définie pour devenir le thread actif. Quand il se réveille, il est de nouveau à la merci de l'ordonnanceur. Maintenant, dans les applications qui ne nécessitent pas de timing parfait et qui n'ont que quelques threads, on peut avoir l'impression que le thread se réveille et reprend son exécution immédiatement après les 2000 millisecondes que vous avez spécifiées. Mais ne pariez pas dessus.

Mettez votre thread en sommeil si vous voulez que les autres aient une chance de s'exécuter.

Quand le thread se réveille, il redevient exécutable et attend que l'ordonnanceur le choisisse à nouveau.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Utiliser sleep() pour rendre notre programme plus prévisible

Vous souvenez-vous de l'exemple précédent dans lequel nous ne cessions d'avoir des résultats différents à chaque exécution du code ? Regardez de nouveau la page 478 et étudiez le code et l'échantillon de résultat. Parfois main() doit attendre que le nouveau thread termine (et affiche « sommet de la pile »), alors que d'autres fois l'exécution du nouveau thread est stoppée avant qu'il ne termine, ce qui permet au thread principal de revenir et d'afficher « retour à main() ». Comment corriger ce problème ? Arrêtez de lire un moment et posez-vous cette question : « Où pouvez-vous placer un appel de sleep() pour être sûr que « retour à main() » s'affichera toujours avant « sommet de la pile » ?

Nous attendrons pendant que vous cherchez une réponse (il y en a plusieurs possibles).

Avez-vous trouvé ?

Voici ce que nous voulons : que les instructions print s'exécutent dans un ordre cohérent :

Fichier Edition Fenêtre Aide Sommeil

```
% java TestThread
retour à main()
sommet de la pile
% java TestThread
retour à main()
sommet de la pile
% java TestThread
retour à main()
sommet de la pile
% java TestThread
retour à main()
sommet de la pile
% java TestThread
retour à main()
sommet de la pile
```

```
public class MonRunnable implements Runnable {

   public void run() {
      go();
   }

   public void go() {

      try {
        Thread.sleep(2000);
      } catch(InterruptedException ex) {
        ex.printStackTrace();
      }

      fairePlus();
   }

   public void fairePlus() {
      System.out.println("sommet de la pile");
   }
}

class TestThread {
   public static void main (String[] args) {
      Runnable laTache = new MonRunnable();
      Thread t = new Thread(laTache);
      t.start();
      System.out.println("retour à main()");
   }
}
```

Appeler sleep() ici forcera le nouveau thread à quitter son état de thread actif !

Le thread principal va redevenir le thread actif et afficher 'retour à main()'. Puis il y aura une pause (d'environ deux secondes) avant que nous ne parvenions à cette ligne qui appelle fairePlus() et affiche 'sommet de la pile'.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Créer et lancer *deux* threads

Les threads ont des noms. Vous pouvez leur donner le nom de votre choix ou accepter leur nom par défaut. Mais ce qui est génial, c'est que vous pouvez utiliser ces noms pour savoir quel thread s'exécute. L'exemple suivant lance deux threads. Les deux ont la même tâche : parcourir une boucle et afficher le nom du thread en cours d'exécution à chaque itération.

```
public class ExecThreads implements Runnable {

    public static void main(String[] args) {
        ExecThreads exec = new ExecThreads();
        Thread alpha = new Thread(exec);
        Thread beta = new Thread(exec);
        alpha.setName("Le thread alpha");
        beta.setName("Le thread beta");
        alpha.start();
        beta.start();
    }

    public void run() {
        for (int i = 0; i < 25; i++) {
            String nom = Thread.currentThread().getName();
            System.out.println(nom + " s'exécute");
        }
    }
}
```

Créer une instance de Runnable.

Créer deux threads en passant le même Runnable (la même tâche – nous verrons ce qui se passe avec deux threads et un seul Runnable dans quelques pages.

Nommer les threads.

Lancer les threads.

Chaque thread parcourt cette boucle et affiche son nom à chaque itération.

Une partie du résultat quand la boucle s'exécute 25 fois.

```
Fichier  Edition  Fenêtre  Aide  Alphabet
Le thread alpha s'exécute
Le thread alpha s'exécute
Le thread alpha s'exécute
Le thread beta s'exécute
Le thread alpha s'exécute
Le thread beta s'exécute
Le thread beta s'exécute
Le thread beta s'exécute
Le thread beta s'exécute
Le thread beta s'exécute
Le thread beta s'exécute
Le thread beta s'exécute
Le thread beta s'exécute
Le thread beta s'exécute
Le thread beta s'exécute
Le thread beta s'exécute
Le thread beta s'exécute
Le thread beta s'exécute
Le thread beta s'exécute
Le thread alpha s'exécute
```

Que va-t-il se passer ?

Les threads vont-il s'exécuter à tour de rôle ? Allez-vous voir les noms des threads s'alterner ? Combien de fois vont-ils permuter ? À chaque itération ? Au bout de cinq itérations ?

Vous connaissez déjà la réponse : *nous n'en savons rien !* Cela dépend de l'ordonnanceur. Et avec un autre OS, une autre JVM et/ou une autre UC, vous pourriez obtenir des résultats très différents.

Sous Mac OS X 10.2 (Jaguar), avec cinq itérations ou moins, le thread alpha s'exécute jusqu'au bout puis le thread beta s'exécute jusqu'au bout. Très cohérent. Pas garanti, mais très cohérent.

Mais avec 25 itérations ou plus, l'exécution est plus incertaine. Le thread alpha peut ne pas terminer toutes les 25 itérations, parce que l'ordonnanceur va le stopper pour donner une chance au thread beta.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Hum, oui. Il EXISTE une face cachée. Les threads peuvent causer des « problèmes » d'accès concurrents.

Les problèmes de concurrence engendrent des problèmes d'intégrité. Les problèmes d'intégrité engendrent des corruptions de données. Les corruptions de données engendrent la peur… Vous connaissez la suite.

Tout cela débouche sur un scénario potentiellement mortel : deux ou plusieurs threads accèdent aux *données* d'un même objet. Autrement dit, des méthodes s'exécutant sur deux piles différentes appellent toutes deux par exemple les méthodes set et get d'un seul objet sur le tas.

C'est toute l'histoire de la main droite qui ne sait pas ce que fait la main gauche. Deux threads, de la façon la plus insouciante du monde, exécutent allégrement leurs méthodes, chacun d'eux étant persuadé qu'il est le Seul Vrai Thread. Le seul qui compte. Après tout, quand un thread ne s'exécute pas et qu'il n'est qu'exécutable (ou bloqué) il est quasiment inconscient. Quand il redevient le thread actif, il ne se souvient même pas qu'il a été stoppé.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Problème de couple

Ce mariage peut-il être sauvé ?

Ce soir, un numéro très spécial du show du Dr Max

[Transcription de l'épisode 42]

Bienvenue au show du docteur Max.

Notre sujet d'aujourd'hui tourne autour des deux principales raisons pour lesquelles les couples se séparent — les finances et le sommeil.

Le couple de ce soir, Sylvie et Bruno, partage un lit et un compte en banque. Mais pas pour longtemps si nous ne trouvons pas de solution. Le problème ? Il est classique : deux personnes et un seul compte.

Sylvie et Bruno : victimes du "problème du compte joint".

Voici la description de Sylvie :

« Bruno et moi nous sommes mis d'accord pour que le compte en banque ne soit jamais à découvert. Nous avons adopté une procédure. Chaque fois que l'un de nous veut retirer de l'argent, il *doit* vérifier le solde du compte *avant* d'effectuer le retrait. Cela semblait si simple. Mais tout d'un coup voilà que nos chèques sont sans provision et que nous recevons des agios à payer !

Je pensais que ce n'était pas possible. J'étais convaincue que la procédure était sûre. Mais il s'est passé *quelque chose* :

Bruno avait besoin de 50 €. Il vérifie le solde du compte et il voit qu'il reste 100 €. Pas de problème. Il dit qu'il va aller retirer l'argent. ***Mais avant ça, il s'endort !***

Bruno s'endort après avoir vérifié le solde mais avant d'avoir effectué le retrait. Quand il se réveille, il termine immédiatement la transaction sans revérifier le solde.

Et c'est là que j'entre en scène. Pendant que Bruno dort, *je* veux retirer 100 €. Je vérifie le solde, il est de 100 € (parce que Bruno dort encore et qu'il n'a rien retiré) et je me dis « pas de problème ». J'effectue le retrait… toujours pas de problème. Mais quand Bruno se réveille, il termine *son* retrait, et nous voilà à découvert ! Il ne se souvient même pas qu'il s'est endormi et il ne pense pas à revérifier le solde avant de terminer sa transaction. Il faut nous aider, docteur Max ! »

Y a-t-il une solution ? Sont-ils condamnés ? Nous ne pouvons pas empêcher Bruno de s'endormir, mais pouvons-nous empêcher Sylvie de toucher au compte en banque tant qu'il n'est pas réveillé ?

Prenez un moment pour y réfléchir, on se retrouve après la pub.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

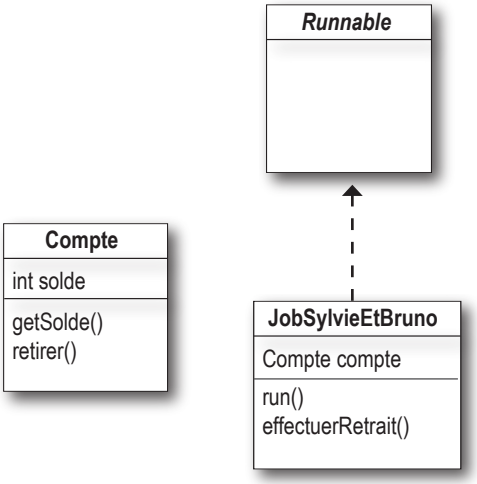
Le problème de Sylvie et Bruno, codé

L'exemple suivant montre ce qui peut se passer quand deux threads (Sylvie et Bruno) partagent un même objet (le compte en banque).

Le code comprend deux classes, Compte et JobSylvieEtBruno. La classe JobSylvieEtBruno implémente Runnable et représente le comportement que Sylvie et Bruno ont tous deux — vérifier le code et effectuer des retraits. Mais, naturellement, chaque thread s'endort entre le moment où il vérifie le compte et celui où il retire réellement l'argent.

La classe JobSylvieEtBruno a une variable d'instance de type Compte qui représente leur compte commun.

Voici comment le code fonctionne :



1 On crée une instance de JobSylvieEtBruno

La classe JobSylvieEtBruno est le Runnable (le travail à effectuer), et puisque Sylvie et Bruno font tous deux la même chose (vérifier le solde et retirer de l'argent), nous n'en créons qu'une seule instance.

```
JobSylvieEtBruno laTache = new JobSylvieEtBruno();
```

2 On crée deux threads avec le même Runnable
(l'instance de JobSylvieEtBruno)

```
Thread un = new Thread(laTache);
Thread deux  = new Thread(laTache);
```

3 On nomme les threads et on les lance

```
un.setName("Bruno");
deux.setName("Sylvie");
un.start();
deux.start();
```

4 Et on regarde les deux threads exécuter la méthode run()
(vérifier le solde)

L'un des threads représente Bruno et l'autre Sylvie. Les deux vérifient le solde en permanence puis effectuent un retrait, mais seulement si le compte est provisionné !

```
if (compte.getSolde() >= montant) {
   try {
      Thread.sleep(500);
   } catch(InterruptedException ex) {ex.printStackTrace(); }
}
```

Dans la méthode run(), faire exactement ce que Sylvie et Bruno feraient — vérifier le solde, et, s'il y a assez d'argent, effectuer le retrait.

Cette procédure devrait éviter que le compte soit à découvert.

Sauf que... Sylvie et Bruno s'endorment toujours après avoir vérifié le solde mais avant d'avoir terminé le retrait.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

L'exemple de Sylvie et Bruno

```
class Compte {
   private int solde = 100;

   public int getSolde() {
      return solde;
   }
   public void retirer(int montant) {
      solde = solde - montant;
   }
}

 public class JobSylvieEtBruno implements Runnable {

   private Compte compte = new Compte();

   public static void main (String [] args) {
      JobSylvieEtBruno laTache = new JobSylvieEtBruno();
      Thread un = new Thread(laTache);
      Thread deux = new Thread(laTache);
      un.setName("Bruno");
      deux.setName("Sylvie");
      un.start();
      deux.start();
    }

  public void run() {
   for (int x = 0; x < 10; x++) {
      effectuerRetrait(10);
      if (compte.getSolde() < 0) {
        System.out.println("Découvert!");
      }
    }
 }

  private void effectuerRetrait(int montant) {
     if (compte.getSolde() >= montant) {
     System.out.println(Thread.currentThread().getName() + " va retirer");
      try {
        System.out.println(Thread.currentThread().getName() + " va dormir");
        Thread.sleep(500);
      } catch(InterruptedException ex) {ex.printStackTrace(); }
      System.out.println(Thread.currentThread().getName() + " se réveille");
      compte.retirer(montant);
      System.out.println(Thread.currentThread().getName() + " termine le retrait");
     }
    else {
      System.out.println("Désolé, pas assez pour " + Thread.currentThread().getName());
   }
 }
```

Le compte commence avec un solde de 100 €.

Il n'y a qu'UNE instance de JobSylvieEtBruno. Autrement dit UNE SEULE instance du compte. Les deux threads vont accéder à ce seul compte.

Instanciation du Runnable (la tâche).

On crée deux threads en donnant à chacun la même tâche. Les deux threads vont donc accéder à la seule variable d'instance, compte, de la classe du Runnable.

Dans la méthode run(), un thread s'exécute en boucle et tente un retrait à chaque itération. Après le retrait, il revérifie le compte pour voir si le compte n'est pas à découvert.

On vérifie le solde. S'il n'y a pas assez d'argent, on affiche simplement un message. S'il y en a ASSEZ, on va dormir, tout comme Bruno, puis on se réveille et on termine la transaction.

Nous avons inséré quelques instructions print pour pouvoir observer le déroulement de l'exécution.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Fichier Edition Fenêtre Aide Visa

```
Bruno va retirer
Bruno va dormir
Sylvie se réveille
Sylvie termine le retrait
Sylvie va retirer
Sylvie va dormir
Bruno se réveille
Bruno termine le retrait
Bruno va retirer
Bruno va dormir
Sylvie se réveille
Sylvie termine le retrait
Sylvie va retirer
Sylvie va dormir
Bruno se réveille
Bruno termine le retrait
Bruno va retirer
Bruno va dormir
Sylvie se réveille
Sylvie termine le retrait
Désolé, pas assez pour Sylvie
Désolé, pas assez pour Sylvie
Désolé, pas assez pour Sylvie
Désolé, pas assez pour Sylvie
Désolé, pas assez pour Sylvie
Bruno se réveille
Bruno termine le retrait
Découvert !
Désolé, pas assez pour Bruno
Découvert !
Désolé, pas assez pour Bruno
Découvert !
Désolé, pas assez pour Bruno
Découvert !
```

Que s'est-il donc passé ?

La méthode effectuerRetrait() vérifie toujours le code avant de retirer de l'argent, mais le compte est quand même à découvert.

Voici un scénario :

Bruno consulte le solde, constate qu'il y a assez d'argent puis s'endort.

Pendant ce temps, Sylvie arrive et vérifie le solde. Elle constate également qu'il y a assez d'argent. Elle n'imagine pas que Bruno va se réveiller et terminer le retrait.

Sylvie s'endort.

Bruno se réveille et termine sa transaction.

Sylvie se réveille et termine SA transaction. Patatras ! Entre le moment où elle a vérifié le solde et celui où elle a terminé le retrait, Bruno s'est réveillé et a retiré de l'argent du compte.

Le résultat de la vérification de Sylvie n'est pas valide, parce que Bruno a déjà vérifié et qu'il est toujours au milieu d'une transaction.

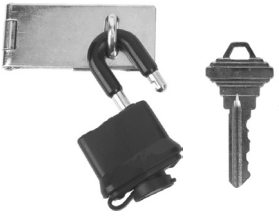
Il faut empêcher Sylvie d'accéder au compte tant que Bruno ne s'est pas réveillé et n'a pas fini sa transaction. Et inversement.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Il faut verrouiller l'accès au compte !

Voici comment le verrou fonctionne :

1. On associe un verrou à la transaction (vérification du solde et retrait). Il n'y a qu'une clé et elle reste sur le verrou jusqu'à ce que quelqu'un veuille accéder au compte.



La transaction est déverrouillée quand personne n'utilise le compte.

2. Quand Bruno veut accéder au compte, il ferme le verrou et met la clé dans sa poche. Maintenant, plus personne ne peut toucher au compte, puisque la clé a disparu.

Quand Bruno veut accéder au compte, il ferme le verrou et empoche la clé.

3. **Bruno garde la clé dans sa poche jusqu'à la fin de la transaction.** Comme c'est la seule clé, Sylvie n'a pas accès au compte tant que Bruno n'a pas ouvert le verrou et remis la clé.

   Maintenant, même si Bruno s'endort après avoir consulté le solde, il a la garantie que le solde sera le même quand il se réveillera parce qu'il a gardé la clé pendant qu'il dormait !

Quand Bruno a terminé, il ouvre le verrou et remet la clé. La clé est maintenant disponible pour Sylvie (ou de nouveau pour Bruno) et le compte est accessible.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

La méthode effectuerRetrait() doit s'exécuter comme une entité atomique

Une fois qu'un thread est entré dans la méthode effectuerRetrait(), nous devons nous assurer *qu'il peut terminer d'exécuter la méthode* avant qu'un autre thread puisse y entrer.

Autrement dit, nous devons faire en sorte que le thread qui a vérifié le solde du compte puis s'est endormi ait la garantie de pouvoir se réveiller et terminer la transaction avant que tout autre thread ne puisse consulter le solde !

On utilise le modificateur **synchronized** pour qu'un seul thread à la fois puisse accéder à la méthode.

C'est comme ça que vous allez pouvoir protéger votre compte en banque ! Vous ne verrouillez pas le code lui-même, mais vous verrouillez la méthode qui réalise la transaction bancaire. De cette façon, un thread doit exécuter la transaction de bout en bout même s'il s'endort au beau milieu de la méthode !

Mais si on ne verrouille pas le compte, qu'est-ce qu'on verrouille exactement ? Est-ce la méthode ? L'objet Runnable ? Le thread lui-même ?

Nous verrons cela page suivante. En termes de code, c'est extrêmement simple : il suffit d'ajouter le modificateur synchronized à la déclaration de la méthode :

Le mot-clé `synchronized` signifie qu'un thread a besoin d'une clé pour accéder au code synchronisé.

Pour protéger vos *données* (comme le compte en banque), synchronisez les *méthodes* qui ont une action sur ces données.

```
private synchronized void effectuerRetrait(int montant) {

    if (compte.getSolde() >= montant) {
       System.out.println(Thread.currentThread().getName() + " va retirer");
       try {
          System.out.println(Thread.currentThread().getName() + " va dormir");
          Thread.sleep(500);
       } catch(InterruptedException ex) {ex.printStackTrace(); }
       System.out.println(Thread.currentThread().getName() + " se réveille");
       compte.retirer(montant);
       System.out.println(Thread.currentThread().getName() + " termine le retrait");
    } else {
       System.out.println("Désolé, pas assez pour " + Thread.currentThread().getName());
    }
}
```

(Note pour les lecteurs férus de physique. Oui, la convention qui consiste à employer le mot 'atomique' ne reflète pas toute cette histoire de particules subatomiques. Pensez à Newton, pas à Einstein, quand vous entendez ce mot dans un contexte de threads ou de transactions. Hé, ce n'est pas NOTRE convention. Si NOUS avions le pouvoir, nous appliquerions le principe d'incertitude d'Heisenberg à pratiquement tout ce qui a trait aux threads.)

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Utiliser les verrous des objets

Tout objet possède un verrou. La plupart du temps, le verrou est ouvert et on peut imaginer qu'ils ont une clé virtuelle. Les verrous n'entrent en scène que quand il existe des méthodes synchronisées. Quand un objet a une ou plusieurs méthodes synchronisées, un ***thread ne peut entrer dans ces méthodes que s'il a la clé du verrou de l'objet !***

Les verrous ne sont pas associés à une *méthode* mais à un *objet*. Si un objet a deux méthodes synchronisées, cela ne signifie pas simplement que deux threads ne peuvent pas entrer dans la même méthode. Cela signifie qu'ils ne peuvent entrer dans aucune des méthodes synchronisées.

Réfléchissez. Si vous avez plusieurs méthodes qui peuvent agir sur les variables d'instance d'un objet, toutes ces méthodes doivent être protégées avec synchronized.

Le but de la synchronisation est de protéger toutes les données critiques. Mais n'oubliez pas : vous ne protégez pas les données elles-même, vous synchronisez les méthodes qui accèdent à ces données.

Que se passe-t-il donc quand un thread exécute sa pile d'appels (en commençant par sa méthode run()) et qu'il tombe soudain sur une méthode synchronisée ? Il reconnaît qu'il a besoin de la clé de l'objet avant de pouvoir entrer dans la méthode. Il cherche la clé (ce processus est géré par la JVM et il n'y a pas d'API permettant d'accéder aux verrous des objets). Si la clé est disponible, le thread s'en saisit et entre dans la méthode.

Dès lors, le thread se cramponne à la clé comme si sa vie en dépendait. Il ne la lâchera pas tant qu'il n'aura pas terminé la méthode synchronisée. Tant qu'il est en possession de la clé, aucun autre thread ne peut entrer dans les méthodes synchronisées de cet objet, parce qu'il n'y a qu'une clé et qu'elle est indisponible.

Tout objet Java a un verrou. Ce verrou n'a qu'une clé.

La plupart du temps, le verrou est ouvert et personne ne s'en soucie.

Mais si un objet a des méthodes synchronisées, le thread ne peut entrer dedans QUE si la clé est disponible, autrement dit que si aucun autre thread ne s'est pas déjà emparé de cette seule clé.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Le terrible problème des mises à jour perdues

Voici un autre problème de concurrence classique qui nous vient du monde des bases de données. Il est étroitement lié à l'histoire de Sylvie et Bruno, et nous utiliserons cet exemple pour illustrer quelques points supplémentaires.

La mise à jour perdue tourne autour d'un processus :

Étape 1 : Obtenir le solde du compte.

```
int i = solde;
```

étape 2 : Ajouter 1 au solde.

```
solde = i + 1;
```

Pour la démonstration, l'astuce consiste à forcer l'ordinateur à modifier le solde du compte en deux étapes. Dans le monde réel, il vous suffirait d'une seule instruction : **solde++;**

Mais si nous imposons deux étapes, le problème du processus non atomique devient clair. Imaginez donc qu'au lieu du banal « lire le solde et ajouter 1 », les deux étapes (ou plus) de cette méthode soient beaucoup plus complexes et qu'une seule instruction ne suffise pas.

Dans le problème de la mise à jour perdue, nous avons deux threads qui essaient tous deux d'incrémenter le solde. Jetez un coup d'œil à ce code, puis nous verrons comment se pose le problème réel :

```
class TestSync implements Runnable {

    private int solde;

    public void run() {
       for(int i = 0; i < 50; i++) {
        incrementer();
        System.out.println("le solde est de  " + solde);
      }
    }

    public void incrementer() {
       int i = solde;
       solde = i + 1;
    }
 }

public class TestSyncTest {
   public static void main (String[] args) {
     TestSync tache = new TestSync();
     Thread a = new Thread(tache);
     Thread b = new Thread(tache);
     a.start();
     b.start();
   }
}
```

Chaque thread s'exécute 50 fois, incrémentant le solde à chaque itération.

Voici le point capital ! Nous incrémentons le solde en ajoutant 1 à la valeur du solde AU MOMENT OÙ NOUS LA LISONS et non en ajoutant 1 à sa valeur COURANTE.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Exécutons ce code...

① Le thread A s'exécute pendant un certain temps

Placer la valeur du solde dans la variable i.
Le solde est de 0, donc i vaut 0.
Affecter à solde le résultat de i + 1.
Le solde vaut maintenant 1.
Placer la valeur du solde dans la variable i.
Le solde est de 1, donc i vaut maintenant 1.
Affecter à solde le résultat de i + 1.
Le solde vaut maintenant 2.

② Le thread B s'exécute pendant un certain temps

Placer la valeur du solde dans la variable i.
Le solde vaut 2, donc i vaut maintenant 2.
Affecter à solde le résultat de i + 1.
Le solde vaut maintenant 3.
Placer la valeur du solde dans la variable i.
Le solde est de 3, donc i vaut maintenant 3.

*[Maintenant, le thread B retourne à l'état exécutable **avant** qu'il n'ait positionné la valeur du solde à 4]*

③ Le thread A reprend là où il s'est arrêté

Placer la valeur du solde dans la variable i.
Le solde vaut 3, donc i vaut maintenant 3.
Affecter à solde le résultat de i + 1.
Le solde vaut maintenant 4.
Placer la valeur du solde dans la variable i.
Le solde vaut 4, donc i vaut maintenant 4.
Affecter à solde le résultat de i + 1.
Le solde vaut maintenant 5.

④ Le thread B reprend exactement là où il s'est arrêté!

Affecter à solde le résultat de i + 1.
Le solde vaut maintenant **4.**

Aïe!!

Le thread A a actualisé le solde à 5, mais B revient et saute par-dessus la mise à jour de A comme si elle n'avait jamais existé.

Nous avons perdu les mises à jour du thread A! Le thread B a «lu» la valeur du solde. Quand B s'est réveillé, il a continué comme s'il n'avait jamais sauté une mesure.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Rendez la méthode incrémenter() atomique synchronisez-la!

La synchronisation de la méthode incrementer() résout le problème de la mise à jour perdue, parce qu'elle fusionne les deux étapes de la méthode en une seule unité indivisible.

```
public synchronized void incrementer() {
    int i = solde;
    solde = i + 1;
}
```

Une fois qu'un thread entre dans la méthode, nous devons veiller à ce que toutes les étapes de celles-ci soient terminées (en un seul processus atomique) avant qu'aucun autre thread ne puisse y entrer.

---

il n'y a pas de

Questions stupides

Q : Est-ce que ce ne serait pas une bonne idée de tout synchroniser, histoire d'être tranquille avec les threads ?

R : Non, ce n'est pas une bonne idée. La synchronisation n'est pas gratuite. D'abord, une méthode synchronisée a un coût. Autrement dit, quand le code tombera sur une méthode synchronisée, il y aura une baisse de performance (même si elle passe généralement inaperçue) jusqu'à ce que la question de la disponibilité de la clé soit résolue.

Deuxièmement, le programme peut être ralenti parce que la synchronisation réduit les accès concurrents. Autrement dit, une méthode synchronisée force les autres threads à faire la queue et à attendre leur tour. Ce n'est pas nécessairement un problème, mais vous devez en tenir compte.

Troisièmement, et c'est bien plus inquiétant, la synchronisation des méthodes peut provoquer un verrou mortel (voir page 516) !

Une bonne règle empirique consiste à synchroniser le strict minimum. En fait, vous pouvez adopter une granularité plus fine que celle de la méthode. Nous ne le faisons pas dans ce livre, mais vous pouvez synchroniser au niveau d'une ou plusieurs instructions et non au niveau de la méthode entière.

```
public void go() {
    faireQqch();

    synchronized(this) {
        codeCritique();
        autreCodeCritique();
    }
}
```

La méthode faireQqch() n'a pas besoin d'être synchronisée et nous nous abstenons.

Maintenant, seuls ces deux appels de méthodes sont groupés en une seule unité atomique. Quand vous utilisez synchronized DANS une méthode et non dans sa déclaration, vous devez passer en argument l'objet dont le thread doit posséder la clé. Même s'il y a d'autres façons de procéder, on passe presque toujours l'objet courant (this). C'est le même objet qui serait verrouillé si toute la méthode était synchronisée.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

① Le thread A s'exécute pendant un certain temps

Essayer d'entrer dans la méthode incrementer().

Comme elle est synchronisée, **prendre la clé** de l'objet.
Mettre la valeur du solde dans la variable i.
Le solde vaut 0, donc i vaut 0.
Affecter à solde le résultat de i + 1.
Le solde vaut maintenant 1.
Rendre la clé (il a terminé la méthode incrementer()).
Réentrer dans la méthode incrementer() et **prendre la clé**.
Mettre la valeur du solde dans la variable i.
Le solde vaut 1, donc i vaut 1.

[maintenant le thread A retourne à l'état exécutable, mais comme il n'a pas terminé la méthode, il garde la clé]

② Le thread B est sélectionné

Essayer d'entrer dans la méthode incrementer(). Comme la méthode est synchronisée, il faut la clé.

La clé n'est pas disponible.

[le thread B est envoyé en salle d'attente]

③ Le thread A reprend là où il s'est arrêté (souvenez-vous qu'il a toujours la clé)

Affecter à solde le résultat de i + 1.
Le solde vaut maintenant 2.
Rendre la clé.

[maintenant le thread A retourne à l'état exécutable, mais comme il a terminé la méthode, il ne conserve PAS la clé]

④ Le thread B est sélectionné

Essayer d'entrer dans la méthode incrementer(). Comme la méthode est synchronisée, il faut la clé.

Cette fois, la clé est disponible. La prendre.
Mettre la valeur du solde dans la variable i.

[continuer…]

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

L'aspect mortel de la synchronisation

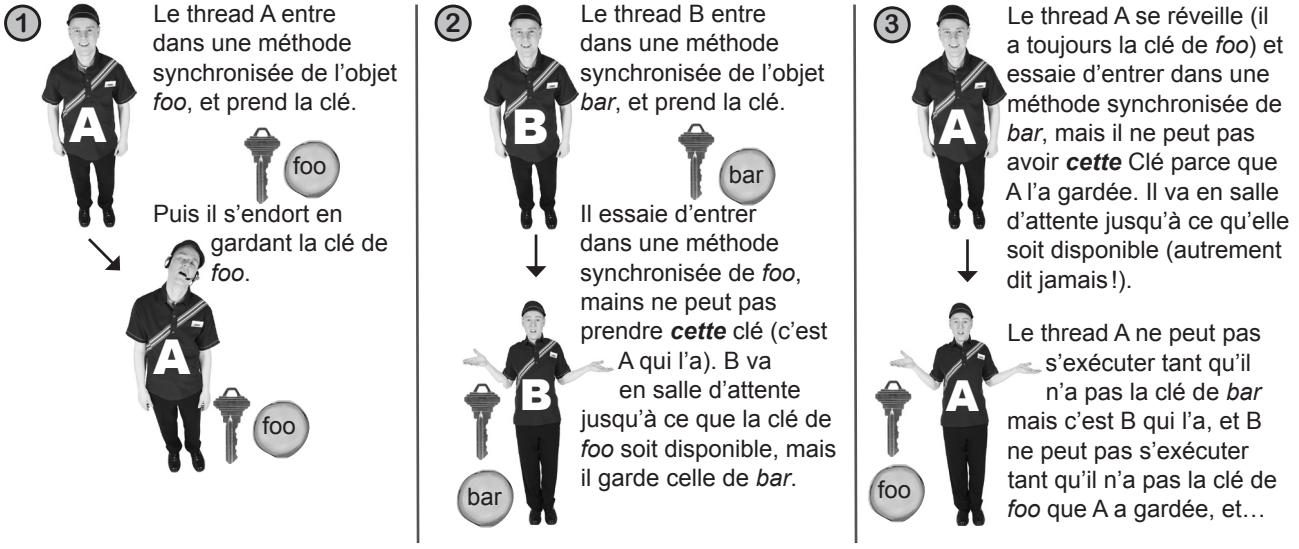
Soyez prudent quand vous synchronisez des méthodes, car il n'y a rien de pire pour un programme qu'un verrou mortel. Un verrou mortel est un phénomène qui se produit quand deux threads détiennent chacun la clé dont l'autre a besoin. Comme il n'existe aucun moyen de sortir de ce scénario, les deux threads se contentent de s'arrêter là et d'attendre… d'attendre… d'attendre encore…

Si vous connaissez bien les serveurs de bases de données ou d'autres applications, vous avez peut-être reconnu le problème : les bases de données disposent souvent d'un mécanisme de verrouillage qui ressemble quelque peu à la synchronisation. Mais un vrai système de gestion de transactions peut parfois résoudre un verrou mortel. Il peut supposer par exemple que le problème est survenu parce que deux transactions mettent trop longtemps à se terminer. Mais, à la différence de Java, le serveur d'applications peut effectuer ce qu'on appelle un *rollback* et restaurer l'état de la transaction (la partie atomique) à ce qu'il était avant que cette transaction ne commence.

Java n'a pas de mécanisme pour gérer les verrous mortels et ne se rend même pas compte qu'ils se produisent. Il vous appartient donc de concevoir votre programme avec soin. Si vous devez écrire beaucoup de code multithread, vous trouverez dans *Java Threads* de Scott Oaks et Henry Wong des astuces de conception pour éviter les verrous mortels. L'une des plus courantes consiste à faire attention à l'ordre dans lequel les threads sont lancés.

Pour créer un verrou mortel, il suffit de deux objets et de deux threads.

Un scénario simple :



1. Le thread A entre dans une méthode synchronisée de l'objet *foo*, et prend la clé. Puis il s'endort en gardant la clé de *foo*.
2. Le thread B entre dans une méthode synchronisée de l'objet *bar*, et prend la clé. Il essaie d'entrer dans une méthode synchronisée de *foo*, mains ne peut pas prendre ***cette*** clé (c'est A qui l'a). B va en salle d'attente jusqu'à ce que la clé de *foo* soit disponible, mais il garde celle de *bar*.
3. Le thread A se réveille (il a toujours la clé de *foo*) et essaie d'entrer dans une méthode synchronisée de *bar*, mais il ne peut pas avoir ***cette*** Clé parce que A l'a gardée. Il va en salle d'attente jusqu'à ce qu'elle soit disponible (autrement dit jamais !). Le thread A ne peut pas s'exécuter tant qu'il n'a pas la clé de *bar* mais c'est B qui l'a, et B ne peut pas s'exécuter tant qu'il n'a pas la clé de *foo* que A a gardée, et…

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

POINTS D'IMPACT

- La méthode statique Thread.sleep() force un thread à cesser de s'exécuter pendant au moins la durée transmise en argument. Thread.sleep(200) met un thread en sommeil pendant 200 millisecondes.
- Comme la méthode sleep() lance une exception vérifiée par le compilateur (InterruptedException), tous les appels de sleep() doivent être placés dans un bloc try/catch ou déclarés.
- Vous pouvez utiliser sleep() pour essayer de donner à tous les threads une chance de s'exécuter, mais il n'existe aucune garantie quant à l'ordre réel d'exécution des threads. Dans la plupart des cas, des appels de sleep() correctement programmés devraient permettre à vos threads d'alterner gentiment.
- Vous pouvez nommer un thread au moyen de la méthode (autre surprise) setName(). Tous les threads ont un nom par défaut, mais les nommer explicitement vous aide à en conserver la trace, en particulier si vous déboguez avec des instructions print.
- Vous pouvez rencontrer de graves problèmes si deux threads ou plus accèdent au même objet sur le tas.
- Deux threads ou plus accédant au même objet peuvent provoquer une corruption de données, par exemple si l'un d'eux cesse de s'exécuter au beau milieu de la manipulation de l'état critique d'un objet.
- Pour sécuriser vos objets, définissez les instructions qui doivent être traitées comme un processus atomique. Autrement dit, décidez quelles sont les méthodes qui doivent se terminer avant qu'un autre thread n'entre dans la même méthode du même objet.
- Employez le mot-clé **`synchronized`** pour modifier une déclaration de méthode quand vous voulez empêcher deux threads d'entrer simultanément dans cette méthode.
- Tout objet a un verrou unique et ce verrou n'a qu'une seule clé. La plupart du temps, nous ne nous occupons pas de ce verrou. Les verrous n'entrent en scène que lorsque l'objet a des méthodes synchronisées.
- Quand un thread entreprend d'entrer dans une méthode synchronisée, il doit « prendre » la clé de l'objet (l'objet dont il essaie d'exécuter la méthode). Si la clé est indisponible (parce qu'un autre thread la détient déjà), le thread va dans une sorte de « salle d'attente » jusqu'à ce qu'il puisse récupérer la clé.
- Même si un objet possède plus d'une méthode synchronisée, il n'a toujours qu'une clé. Une fois un thread entré dans une des ces méthodes, aucun autre ne peut y entrer. Cette restriction vous permet de protéger vos données en synchronisant toutes les méthodes qui les manipulent.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

SimpleClientDiscussion revu et corrigé

Au début de ce chapitre, nous avons construit un programme SimpleClientDiscussion qui pouvait envoyer des messages sortants au serveur mais ne pouvait rien recevoir. Vous vous en souvenez ? C'est la raison pour laquelle nous sommes rentrés dans toute cette histoire de threads, parce que nous avions besoin de faire deux choses en même temps : envoyer des messages au serveur (par l'intermédiaire de l'IHM) tout en recevant simultanément des messages et en les affichant dans la zone de texte.

```
import java.io.*;
import java.net.*;
import java.util.*;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class SimpleClientDiscussion {

    JTextArea entrants;
    JTextField sortants;
    BufferedReader lecture;
    PrintWriter ecriture;
    Socket sock;

    public static void main(String[] args) {
       SimpleClientDiscussion client = new SimpleClientDiscussion();
       client.go();
    }

    public void go() {

        JFrame cadre = new JFrame("Client de discussion ridiculement simple");
        JPanel panneau = new JPanel();
        entrants = new JTextArea(15,50);
        entrants.setLineWrap(true);
        entrants.setWrapStyleWord(true);
        entrants.setEditable(false);
        JScrollPane zoneTexte = new JScrollPane(entrants);
        zoneTexte.setVerticalScrollBarPolicy(ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS);
        zoneTexte.setHorizontalScrollBarPolicy(ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER);
        sortants = new JTextField(20);
        JButton boutonEnvoi = new JButton("Envoi");
        boutonEnvoi.addActionListener(new EcouteBoutonEnvoi());
        panneau.add(zoneTexte);
        panneau.add(sortants);
        panneau.add(boutonEnvoi);
        installerReseau();

        Thread threadLecture = new Thread(new LectureEntrants());
        threadLecture.start();

        cadre.getContentPane().add(BorderLayout.CENTER, panneau);
        cadre.setSize(400,500);
        cadre.setVisible(true);

    } // fin de la méthode go()
```

Oui, ce chapitre a VRAIMENT une fin. Mais nous n'y sommes pas encore...

Nous construisons l'IHM comme d'habitude. Rien de particulièrement nouveau, sauf la partie grisée où nous lançons le nouveau thread de "lecture".

Nous lançons un nouveau thread, dont le Runnable (la tâche) sera une nouvelle classe interne. Cette tâche consiste à lire le flot émis par la socket du serveur et à afficher les messages entrants dans la zone de texte.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

```
private void installerReseau() {

      try {
         sock = new Socket("127.0.0.1", 5000);
         InputStreamReader isr = new InputStreamReader(sock.getInputStream());
         lecture = new BufferedReader(isr);
         ecriture = new PrintWriter(sock.getOutputStream());
         System.out.println("Connexion établie");
      } catch(IOException ex) {
         ex.printStackTrace();
      }
   } // fin de la méthode installerReseau()
```

Nous utilisons la socket pour les flots d'entrée et de sortie. Nous utilisions déjà le flot de sortie pour émettre vers le serveur, mais nous allons maintenant utiliser le flot d'entrée pour que le nouveau thread de lecture puisse lire les messages provenant du serveur.

```
public class EcouteBoutonEnvoi implements ActionListener {
   public void actionPerformed(ActionEvent ev) {
       try {
          ecriture.println(sortants.getText());
          ecriture.flush();

       } catch(Exception ex) {
          ex.printStackTrace();
       }
       sortants.setText("");
       sortants.requestFocus();
   }
 }  // fin de la classe interne
```

Rien de nouveau ici. Quand l'utilisateur clique sur Envoi, cette méthode envoie le contenu du champ de texte au serveur.

```
public class LectureEntrants implements Runnable {
   public void run() {
         String message;
         try {

           while ((message = lecture.readLine()) != null) {
              System.out.println("lire " + message);
              entrants.append(message + "\n");

            } // fin de la boucle while
         } catch(Exception ex) {ex.printStackTrace();}
      } // fin de la méthode run()
 } // fin de la classe interne

} // fin de la classe externe
```

Voici ce que fait le thread!!

Dans la méthode run(), il reste dans la boucle (tant que ce qu'il reçoit du serveur n'est pas null), lisant une ligne à la fois et ajoutant chaque ligne à la zone de texte (avec un retour à la ligne).

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Code prêt à l'emploi

Un serveur de discussion très très simple

Vous pouvez utiliser ce serveur pour les deux versions du client. Toutes les réserves d'usage jamais alléguées s'appliquent ici. Pour encombrer le moins possible le code, nous avons omis bon nombre de blocs dont vous auriez besoin pour en faire un vrai serveur. Autrement dit, il fonctionne mais il y a au moins une centaine de façons de le planter. Si vous voulez faire un exercice vraiment efficace quand vous aurez fini ce livre, reprenez ce programme et essayez de le rendre plus robuste.

Un autre exercice possible que vous pouvez faire immédiatement consiste à annoter le code vous-même. Vous le comprendrez beaucoup mieux si vous essayez de découvrir ce qui se passe que si nous vous l'expliquions. Mais encore une fois, c'est du code prêt à l'emploi et vous n'avez pas vraiment besoin d'y comprendre quoi que ce soit. Sa seule finalité est de prendre en charge le client.

Pour exécuter le client, il vous faut deux fenêtres d'exécution. Lancez ce serveur dans la première, puis lancez le client dans la seconde.

```
import java.io.*;
import java.net.*;
import java.util.*;

public class ServeurTresSimple {

    ArrayList clientOutputStreams;

    public class GestionClient implements Runnable {
         BufferedReader lecture;
         Socket sock;

         public GestionClient(Socket socketClient) {
           try {
             sock = socketClient;
             InputStreamReader isr = new InputStreamReader(sock.getInputStream());
             lecture = new BufferedReader(isr);

           } catch(Exception ex) {ex.printStackTrace();}
          } // fin du constructeur

        public void run() {
           String message;
           try {
             while ((message = lecture.readLine()) != null) {
                System.out.println("read " + message);
                afficherATous(message);

              } // fin de la boucle while
           } catch(Exception ex) {ex.printStackTrace();}
        } // fin de la méthode run()
    } // fin de la classe interne
```


Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

```
    public static void main (String[] args) {
         new ServeurTresSimple().go();
    }

    public void go() {
       clientOutputStreams = new ArrayList();
       try {
         ServerSocket serverSock = new ServerSocket(5000);

         while(true) {
            Socket socketClient = serverSock.accept();
            PrintWriter ecriture = new PrintWriter(socketClient.getOutputStream());
            clientOutputStreams.add(ecriture);

            Thread t = new Thread(new GestionClient(socketClient));
            t.start();
            System.out.println("connexion établie");
         }

       } catch(Exception ex) {
         ex.printStackTrace();
       }
   } // fin de la méthode go()

   public void afficherATous(String message) {

      Iterator it = clientOutputStreams.iterator();
      while(it.hasNext()) {
         try {
            PrintWriter ecriture = (PrintWriter) it.next();
            ecriture.println(message);
            ecriture.flush();
         }  catch(Exception ex) {
              ex.printStackTrace();
         }

      } // fin de la boucle while

   } // fin de la méthode afficherATous
} // fin de la classe
```


Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

il n'y a pas de
Questions stupides

Q : Et la protection de l'état des variables statiques ? Si on a des méthodes statiques qui modifient l'état des variables statiques, est-ce qu'on peut toujours utiliser la synchronisation ?

R : Oui ! N'oubliez pas que les méthodes statiques sont associées à la classe et non à une instance individuelle de la classe. On peut donc alors se demander à quel objet appartiendrait le verrou utilisé dans le cas d'une méthode statique. Après tout, il n'y a peut être même pas d'instances de cette classe. Heureusement, tout comme chaque objet a son propre verrou, chaque classe chargée possède un verrou. Si vous avez trois objets Chien sur le tas, vous avez donc un total de quatre verrous associés à Chien. Trois d'entre eux appartiennent aux instances de Chien et le dernier à la classe Chien elle-même. Quand vous synchronisez une méthode statique, Java utilise le verrou de la classe. Si vous synchronisez deux méthodes statiques dans une même classe, un thread aura besoin du verrou de la classe pour entrer dans l'une quelconque des deux méthodes.

Q : Qu'est-ce que les priorités des threads ? J'ai entendu dire que c'était une façon de contrôler l'ordonnancement.

R : Les priorités pourraient vous aider à influencer l'ordonnanceur, mais elles n'offrent toujours aucune garantie. Ce ne sont que des valeurs numériques qui indiquent à l'ordonnanceur (à supposer qu'il s'en soucie) l'importance qu'un thread donné revêt pour vous. En général, il rétrogradera un thread moins prioritaire si un thread plus prioritaire devient soudain exécutable. Mais… encore une fois, répétez avec moi « il n'y a aucune garantie ». Nous vous recommandons de ne recourir aux priorités que pour des questions de performance, mais de ne jamais, jamais compter dessus pour que votre programme s'exécute correctement.

Q : Pourquoi ne pas simplement synchroniser les méthodes get et set de la classe qui contient les données que vous voulez protéger ? Par exemple, pourquoi ne pas juste synchroniser la méthode getSolde() et retirer() de la classe Compte au lieu de synchroniser la méthode effectuerRetrait() de la classe du Runnable ?

R : En réalité, nous aurions dû synchroniser ces méthodes pour empêcher d'autres threads d'y accéder d'une autre manière. Nous nous en sommes dispensés dans notre exemple parce qu'aucun autre code n'accédait au compte.

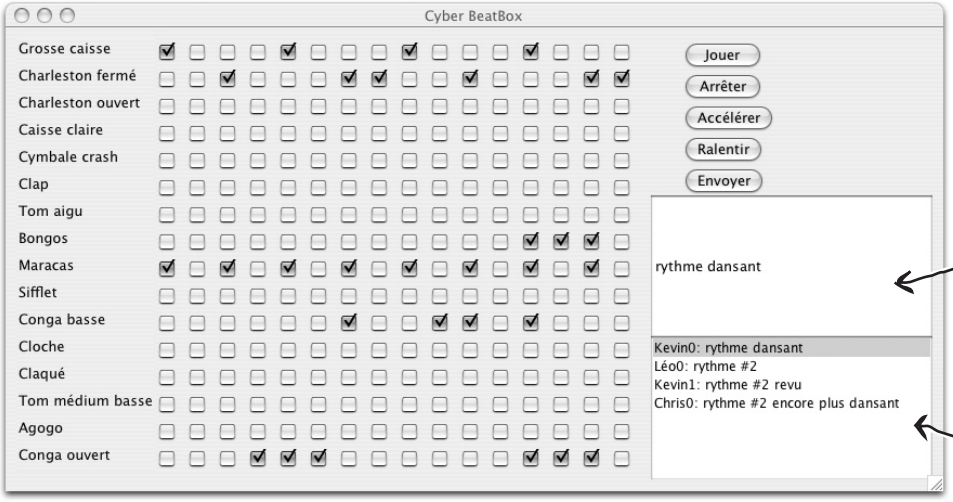
Mais synchroniser les méthodes get et set (ou dans ce cas de getSolde() et de retirer()) ne suffit pas. N'oubliez pas que l'objectif de la synchronisation est de rendre ATOMIQUE une portion de code spécifique. Autrement dit, ce ne sont pas seulement les méthode individuelles qui nous imposent, ce sont les méthodes qui nécessitent ***plusieurs étapes*** ! Réfléchissez. Si nous n'avions pas synchronisé effectuerRetrait(), Bruno aurait vérifié le solde (en appelant getSolde()), puis il serait sorti immédiatement de la méthode et aurait rendu la clé !

Naturellement, il se serait emparé de nouveau de la clé après s'être réveillé pour pouvoir appeler la méthode retirer() synchronisée, mais le problème que nous avions avant la synchronisation reste entier ! Si Bruno vérifie le solde et va dormir, Sylvie peut toujours vérifier le solde avant que Bruno n'ait une chance de se réveiller et de terminer son retrait.

La synchronisation de toutes les méthodes d'accès est donc probablement une bonne idée pour empêcher d'autres threads d'intervenir, mais vous devez toujours synchroniser les méthodes qui doivent s'exécuter comme une seule unité atomique.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Recettes de code



Quand vous cliquez sur "Envoyer", votre message est envoyé aux autres participants avec votre motif.

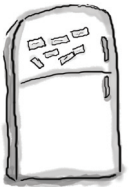
Les messages entrants des autres joueurs. Cliquez sur l'un d'eux pour charger le motif qui l'accompagne puis sur "Jouer" pour le jouer.

Voici la dernière version de la BeatBox !

Elle se connecte à un simple MusicServer pour que vous puissiez échanger des motifs avec d'autres clients.

Le code étant assez long, le listing complet se trouve à l'annexe A.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>



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 !

```
public class TestThreads {
```

```
class Accum {
```

```
class ThreadUn
```

```
class ThreadDeux
```

```
Fichier Edition Fenêtre Aide Filer
% java TestThreads
un 98098
deux 98099
```

Question bonus : Pourquoi pensez-vous que nous ayons utilisé les modificateurs de cette façon dans la classe Accum ?

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Le frigo (suite)

```
Accum a = Accum.getAccum();
```

```
Thread un = new Thread(t1);
```

```
System.out.println("deux "+ a.getCompte());
```

```
} catch(InterruptedException ex) { }
```

```
ThreadDeux t2 = new ThreadDeux();
```

```
try {
```

```
return compteur;
```

```
compteur += add;
```

```
Thread deux = new Thread(t2);
```

```
implements Runnable {
```

```
un.start();
```

```
Accum a = Accum.getAccum();
```

```
Thread.sleep(50);
```

```
public static Accum getAccum() {
```

```
} catch(InterruptedException ex) { }
```

```
private static Accum a = new Accum();
```

```
private int compteur = 0;
```

```
public void run() {
```

```
a.majCompteur(1);
```

```
Thread.sleep(50);
```

```
for(int x=0; x < 99; x++) {
```

```
implements Runnable {
```

```
a.majCompteur(1000);
```

```
public int getCompte() {
```

```
return a;
```

```
System.out.println("un "+ a.getCompte());
```

```
public void majCompteur(int add) {
```

```
for(int x=0; x < 98; x++) {
```

```
deux.start();
```

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

```
try {
```

```
public void run() {
```

```
private Accum() { }
```

```
ThreadUn t1 = new ThreadUn();
```


Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Solutions des exercices

```
public class TestThreads {
  public static void main(String [] args) {
    ThreadUn t1 = new ThreadUn();
    ThreadDeux t2 = new ThreadDeux();
    Thread un = new Thread(t1);
    Thread deux = new Thread(t2);
    un.start();
    deux.start();
  }
}

class Accum {
  private static Accum a = new Accum();
  private int compteur = 0;

  private Accum() { }

  public static Accum getAccum() {
    return a;
  }

  public void majCompteur(int add) {
    compteur += add;
  }

  public int getCompte() {
    return compteur;
  }
}

class ThreadUn implements Runnable {
  Accum a = Accum.getAccum();
  public void run() {
    for(int x=0; x < 98; x++) {
      a.majCompteur(1000);
      try {
        Thread.sleep(50);
      } catch(InterruptedException ex) { }
    }
    System.out.println("un "+a.getCompte());
  }
}
```

Création d'une instance de Accum.

Constructeur privé.

Les threads de deux classes différentes mettent à jour le même objet d'une troisième classe, parce qu'ils accèdent tous deux à une même instance de Accum. La ligne de code :

```
private static Accum a = new Accum( );
```

crée une instance statique de Accum (n'oubliez pas que statique signifie une par classe) et le constructeur privé signifie que personne d'autre ne peut créer d'objet Accum. Ces deux techniques (constructeur privé et méthode get statique) utilisées ensemble créent ce qu'on appelle un « Singleton » : un pattern OO qui permet de limiter le nombre d'instances d'un objet pouvant exister dans une application. (Cette instance est généralement unique, mais vous pouvez appliquer le pattern (le motif) pour restreindre la création d'instances à votre guise.)

```
class ThreadDeux implements Runnable {
  Accum a = Accum.getAccum();
  public void run() {
    for(int x=0; x < 99; x++) {
      a.majCompteur(1);
      try {
        Thread.sleep(50);
      } catch(InterruptedException ex) { }
    }
    System.out.println("deux "+a.getCompte());
  }
}
```


Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Sas d'embrouilles

La clé du mystère

En chemin pour rejoindre l'équipe de développement embarquée, Sarah fit une pause devant la baie pour contempler le lever du soleil sur l'Océan indien. Même si la salle de conférence du vaisseau engendrait indéniablement la claustrophobie, le spectacle du croissant bleu et blanc dispersant la nuit sur la planète la remplit d'admiration mêlée d'effroi.

La réunion de ce matin-là était consacrée au système de contrôle des sas de l'orbiteur. À mesure que les dernières phases de construction touchaient à leur fin, le nombre de sorties dans l'espace devait augmenter de manière spectaculaire et le trafic empruntant les sas était élevé dans les deux sens. «Bonjour Sarah», dit Léo, «tu tombes à pic. Nous allions juste commencer la revue de conception détaillée».

«Comme vous le savez tous», commença Léo, «chaque sas est équipé des deux côtés de terminaux graphiques à l'épreuve de l'espace. Chaque fois qu'un éclaireur sortira du vaisseau ou y rentrera, il utilisera ces terminaux pour lancer les procédures d'ouverture et de fermeture du sas». Sarah Hocha la tête. «Léo, peux-tu nous dire quelles sont les séquences de méthodes pour la sortie et la rentrée ?». Léo se leva et flotta jusqu'au tableau blanc, «D'abord, voici le pseudocode des méthodes pour la sortie.» Il se mit à écrire rapidement au tableau.

```
sequenceSortieSasOrbiteur()

  verifierEtatBaie();

  pressuriserSas();

  ouvrirPorteInterieure();

  confirmerOccupationSas();

  fermerPorteInterieure();

  decompresserSas();

  ouvrirPorteExterieure();

  confirmerSasVide();

  fermerPorteExterieure();
```

«Pour garantir que la séquence ne sera pas interrompue, nous avons synchronisé toutes les méthodes appelées par sequenceSortieSasOrbiteur()», expliqua Léo. «Nous détesterions voir un éclaireur rentrant au vaisseau surprenant par inadvertance un de ses copains avec son pantalon de scaphandre baissé !».

Tout le monde gloussa tandis que Léo effaçait le tableau, mais quelque chose tracassait Sarah et cela fit tilt quand il se mit à griffonner le pseudo-code de la séquence de rentrée. «Attends une minute Léo !», cria-t-elle. «Je crois que nous avons un gros défaut dans la conception de la séquence de sortie. Revoyons-la, le problème pourrait être critique !».

Pourquoi Sarah a-t-elle interrompu la réunion ? Que soupçonne-t-elle ?

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Que savait Sarah ?

Sarah s'était rendu compte que, pour empêcher toute interruption de la séquence de sortie, il fallait synchroniser la méthode `sequenceSortieSasOrbiteur()`. En l'état actuel de la conception, il était possible à un éclaireur rejoignant le vaisseau de la bloquer. Le thread correspondant ne serait pas interrompu au milieu d'un des appels de méthodes de bas niveau, mais il pouvait l'être entre ces appels.

Sarah savait que la totalité de la séquence devait s'exécuter comme une seule entité atomique. Si on synchronisait la méthode `sequenceSortieSasOrbiteur()` absolument plus rien ne pourrait l'interrompre.

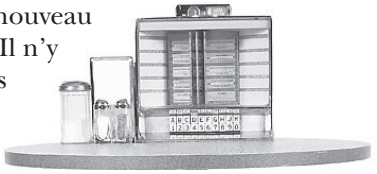
Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Structures de données

Le tri est un jeu d'enfants en Java. Vous avez tous les outils pour collecter et manipuler vos données sans devoir écrire vos propres algorithmes de tri (à moins qu'en ce moment vous ne soyez en train de lire ceci dans votre cours d'introduction à l'informatique, auquel cas, croyez-nous, vous ALLEZ écrire des programmes de tri pendant que nous nous contenterons d'appeler des méthodes de l'API Java). Le framework Java Collections dispose d'une structure de données qui devrait virtuellement répondre à tous vos besoins. Vous voulez une liste dans laquelle il est facile d'insérer ? Trouver quelque chose par son nom ? Créer une liste qui élimine automatiquement tous les doublons ? Trier vos collègues selon le nombre de fois qu'ils vous ont tapé dans le dos ? Trier vos animaux domestiques selon le nombre de tours qu'ils ont appris ? Tout est là...

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Suivre la popularité des morceaux sur votre juke-box



Félicitations pour votre nouveau travail — la gestion du nouveau système de juke-box automatisé au restaurant de Bruno. Il n'y a pas de Java dans le juke-box lui-même, mais chaque fois que quelqu'un joue un morceau, les données de celui-ci sont ajoutées à la fin d'un simple fichier texte.

Votre tâche consiste à gérer les données pour suivre la popularité des morceaux, générer des rapports et manipuler les programmes de disques. Vous n'allez pas écrire toute l'application — d'autres développeurs sont également impliqués — mais vous êtes responsable de la gestion et du tri des données dans l'application Java. Et comme Bruno a quelque chose contre les bases de données, cette collection réside entièrement en mémoire. Vous ne disposez que du fichier dans lequel le juke-box continue à insérer des données. Votre travail est de le récupérer.

Vous avez déjà déterminé comment lire et analyser le fichier, et jusqu'ici vous stockez les données dans une ArrayList.

ListeMorceaux.txt

```
Pink Moon/Nick Drake
Somersault/Zero 7
Shiva Moon/Prem Joshua
Circles/BT
Deep Channel/Afro Celts
Passenger/Headmix
Listen/Tahiti 80
```

Voici le fichier dans lequel le juke-box écrit. Votre code doit lire ce fichier puis manipuler les données des morceaux.

Difficulté n° 1
Trier les morceaux par ordre alphabétique

Vous avez une liste de morceaux dans un fichier, dans lequel chaque ligne représente un morceau et où le titre et le nom de l'artiste sont séparés par une barre oblique. Il devrait donc être simple d'analyser la ligne et de placer tous les morceaux dans une ArrayList.

Comme votre patron ne s'intéresse qu'aux titres des morceaux, vous pouvez vous contenter pour l'instant d'une liste qui ne contient que ceux-ci.

Mais vous constatez que la liste n'est pas dans l'ordre alphabétique... Que pouvez-vous faire ?

Vous savez que, dans une ArrayList, les éléments restent dans l'ordre dans lequel ils ont été insérés : l'ArrayList ne s'occupera donc pas de les classer dans l'ordre alphabétique, à moins que... Peut-être y a-t-il une méthode sort() dans la classe ArrayList ?

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Voici votre code existant, sans le tri :

```
import java.util.*;
import java.io.*;

public class Jukebox1 {

    ArrayList<String> listeMorceaux = new ArrayList<String>();

    public static void main(String[] args) {
        new Jukebox1().go();
    }

    public void go() {
        getMorceaux();
        System.out.println(listeMorceaux);
    }

    void getMorceaux() {
        try {
            File fichier = new File("ListeMorceaux.txt");
            BufferedReader br = new BufferedReader(new FileReader(fichier));
            String ligne =  null;
            while ((ligne= br.readLine()) != null) {
                ajouterMorceau(ligne);
            }

        } catch(Exception ex) {
            ex.printStackTrace();
        }
    }

    void ajouterMorceau(String ligneAAnalyser) {
        String[] param = ligneAAnalyser.split("/");
        listeMorceaux.add(param[0]);
    }

}
```

Nous allons stocker les titres des morceaux dans une ArrayList de chaînes de caractères.

La méthode qui commence à charger le fichier puis imprime le contenu de listeMorceaux.

Rien de spécial ici... On lit le fichier et on appelle la méthode ajouterMorceau() pour chaque ligne.

La méthode ajouterMorceau fonctionne exactement comme le QuizCartes dans le chapitre sur les E/S : on décompose la ligne (qui contient le titre et l'artiste) en deux parties (paramètres) avec la méthode split().

Comme on ne veut que le titre du morceau, on n'ajoute que la première partie à listeMorceaux (l'ArrayList).

```
Fichier Edition Fenêtre Aide Danser
%java Jukebox1
[Pink Moon, Somersault,
Shiva Moon, Circles,
Deep Channel, Passenger,
Listen]
```

La listeMorceaux s'affiche. Les morceaux apparaissent selon l'ordre dans lequel ils ont été ajoutés dans l'ArrayList (le même que dans le fichier texte d'origine).

De toute évidence, ce n'est PAS l'ordre alphabétique !

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Mais la classe ArrayList n'a PAS de méthode sort()!

Quand on lit la documentation d'ArrayList, on ne trouve aucune méthode relative aux tris. Remonter dans la hiérarchie d'héritage ne nous aide pas non plus : il est clair qu'il est impossible d'appeler une méthode de tri sur l'***ArrayList.***

ArrayList (Java 2 Platform SE 5.0)

http://java.sun.com/j2se/1.5.0/docs/api/index.html

java docs api

ArrayList (Java 2 Platform S...

java.nio.channels
java.nio.channels.spi
java.nio.charset
java.nio.charset.spi
java.rmi
java.rmi.activation
java.rmi.dgc
java.rmi.registry
java.rmi.server
java.security
java.security.acl
java.security.cert
java.security.interfaces
java.security.spec
java.sql
java.text
java.util
java.util.concurrent
java.util.concurrent.atomic
java.util.concurrent.locks
java.util.jar
java.util.logging
java.util.prefs
java.util.regex
java.util.zip
javax.accessibility
javax.activity
javax.crypto
javax.crypto.interfaces
javax.crypto.spec
javax.imageio
javax.imageio.event

BAD_POLICY_VALUE
BAD_QOS
BAD_TYPECODE
BadAttributeValueExpException
BadBinaryOpValueExpException
BadKind
BadLocationException
BadPaddingException
BadStringOperationException
BandCombineOp
BandedSampleModel
BaseRowSet
BasicArrowButton
BasicAttribute
BasicAttributes
BasicBorders
BasicBorders.ButtonBorder
BasicBorders.FieldBorder
BasicBorders.MarginBorder
BasicBorders.MenuBarBorder
BasicBorders.RadioButtonBorder
BasicBorders.RolloverButtonBorder
BasicBorders.SplitPaneBorder
BasicBorders.ToggleButtonBorder
BasicButtonListener
BasicButtonUI
BasicCheckBoxMenuItemUI
BasicCheckBoxUI
BasicColorChooserUI

Method Summary

boolean	add(E o) Appends the specified element to the end of this list.
void	add(int index, E element) Inserts the specified element at the specified position in this list.
boolean	addAll(Collection<? extends E> c) Appends all of the elements in the specified Collection to the end of this list, in the order that they are returned by the specified Collection's Iterator.
boolean	addAll(int index, Collection<? extends E> c) Inserts all of the elements in the specified Collection into this list, starting at the specified position.
void	clear() Removes all of the elements from this list.
Object	clone() Returns a shallow copy of this ArrayList instance.
boolean	contains(Object elem) Returns true if this list contains the specified element.
void	ensureCapacity(int minCapacity) Increases the capacity of this ArrayList instance, if necessary, to ensure that it can hold at least the number of elements specified by the minimum capacity argument.
E	get(int index) Returns the element at the specified position in this list.
int	indexOf(Object elem) Searches for the first occurence of the given argument, testing for eq
boolean	isEmpty() Tests if this list has no elements.
int	lastIndexOf(Object elem) Returns the index of the last occurrence of the specified object in t
E	remove(int index) Removes the element at the specified position in this list.
boolean	remove(Object o) Removes a single instance of the specified element from this list,
protected void	removeRange(int fromIndex, int toIndex) Removes from this List all of the elements whose index is betwee
E	set(int index, E element) Replaces the element at the specified position in this list with th
int	size() Returns the number of elements in this list.
Object[]	toArray() Returns an array containing all of the elements in this list in th
<T> T[]	toArray(T[] a) Returns an array containing all of the elements in this list in th
void	trimToSize() Trims the capacity of this ArrayList instance to be the list's

Methods inherited from class java.util.AbstractList

equals, hashCode, iterator, listIterator, listIterator, subLi

ArrayList a une foule de méthodes, mais rien qui ressemble à quelque chose qui puisse trier...

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

ArrayList n'est pas la seule collection

Même si c'est l'ArrayList que vous utiliserez le plus souvent, il y en a d'autres pour les occasions spéciales. En voici quelques unes :

N'essayez pas de tout retenir tout de suite. Nous entrerons dans les détails un peu plus loin.

- **TreeSet**
  Conserve les éléments triés et empêche les doublons.

- **HashMap**
  Permet de stocker les éléments et d'y accéder sous forme de paires nom/valeur.

- **LinkedList**
  Conçue pour obtenir de meilleures performances quand on insère ou supprime des éléments au milieu de la collection. (En pratique, une ArrayList suffit généralement.)

- **HashSet**
  Empêche les doublons dans la collection et permet de trouver rapidement un élément.

- **LinkedHashMap**
  Comme une HashMap ordinaire, sauf qu'elle peut se souvenir de l'ordre dans lequel les éléments (paires nom/valeur) ont été insérés, et qu'on peut la configurer pour se souvenir de l'ordre dans lequel on y a accédé en dernier.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Vous *pourriez* utiliser un TreeSet... ou vous pourriez utiliser la méthode Collections.sort()

Si vous placez toutes les chaînes de caractères (les titres des morceaux) dans un **TreeSet** au lieu d'une ArrayList, elles atterriront automatiquement à la bonne place, triées dans l'ordre alphabétique. Chaque fois que vous afficherez la liste, les éléments seront toujours dans cet ordre.

Et c'est génial quand vous avez besoin d'un *ensemble* (nous parlerons des ensembles dans quelques minutes) ou quand vous savez que la liste doit toujours rester triée dans l'ordre de l'alphabet.

En revanche, si vous n'avez pas besoin que la liste reste triée, TreeSet pourrait être plus coûteux que nécessaire : ***chaque fois que vous insérez dans un TreeSet, celui-ci doit prendre le temps de déterminer à quel endroit de l'arbre le nouvel élément doit aller.*** Avec les ArrayList, les insertions peuvent être extrêmement rapides, parce que le nouvel élément va tout simplement

java.util.Collections

```
public static void copy(List destination, List source)
public static List emptyList()
public static void fill(List listeARemplir, Object objetPourRemplir)
public static int frequency(Collection c, Object o)
public static void reverse(List liste)
public static void rotate(List list,e int distance)
public static void shuffle(List liste)
public static void sort(List liste)
public static bo[illegible]laceAll(List liste, Object ancVal, Object nouvVal)
// beaucoup d[illegible]
```

Mmmm... Il y a BIEN une méthode sort() dans la classe Collections. Elle accepte une List, et comme ArrayList implémente l'interface List, ArrayList EST-UNE List. Grâce au polymorphisme, vous pouvez transmettre une ArrayList à une méthode déclarée pour accepter une List.

Q : Mais on PEUT ajouter quelque chose à une ArrayList à un indice spécifique et non simplement à la fin. Il y a une méthode add() surchargée qui prend en paramètre un entier avec l'élément à ajouter. Est-ce que ce ne serait pas plus lent que d'ajouter à la fin ?

R : Oui, c'est plus lent d'insérer quelque chose dans une ArrayList autre part qu'à la fin. Utiliser la méthode surchargée add(indice, element) ne fonctionne donc pas aussi rapidement que l'appel de add(element) qui place l'élément ajouté à la fin. Mais dans la plupart des cas d'utilisation d'ArrayList, vous n'avez pas besoin de placer quelque chose à un indice donné.

Q : Je vois qu'il y a une classe LinkedList. Est-ce qu'elle ne conviendrait pas mieux pour insérer quelque chose en milieu de liste ? Au moins si je me souviens bien de mon cours de structures de données à la fac...

R : Oui, bien vu. La classe LinkedList peut être plus rapide quand vous insérez ou supprimez quelque chose au milieu. Mais dans la plupart des applications, la différence entre les insertions au milieu d'une LinkedList et d'une ArrayList est généralement négligeable, à moins que vous n'ayez affaire à un nombre gigantesque d'éléments. Nous aborderons LinkedList dans quelques minutes.

Note : ceci n'est PAS la vraie documentation de la classe Collections. Nous l'avons simplifiée en omettant les informations sur les types génériques (que vous verrez dans quelques pages).

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Ajouter Collections.sort() au code du Jukebox

```
import java.util.*;
import java.io.*;

public class Jukebox1 {

    ArrayList<String> listeMorceaux = new ArrayList<String>();

    public static void main(String[] args) {
        new Jukebox1().go();
    }

    public void go() {
        getMorceaux();
        System.out.println(listeMorceaux);
        Collections.sort(listeMorceaux);
        System.out.println(listeMorceaux);
    }

    void getMorceaux() {
        try {
            File fichier = new File("ListeMorceaux.txt");
            BufferedReader br = new BufferedReader(new FileReader(fichier));
            String ligne =  null;
            while ((ligne= br.readLine()) != null) {
                ajouterMorceau(ligne);
            }

        } catch(Exception ex) {
            ex.printStackTrace();
        }
    }

    void ajouterMorceau(String ligneAAnalyser) {
        String[] param = ligneAAnalyser.split("/");
        listeMorceaux.add(param[0]);
    }
}
```

La méthode Collections.sort() trie une liste de chaînes de caractères par ordre alphabétique.

Appeler la méthode statique Collections sort(), puis afficher de nouveau la liste. Le second affichage est dans l'ordre alphabétique!

```
Fichier Edition Fenêtre Aide Chanter
%java Jukebox1

[Pink Moon, Somersault, Shiva Moon, Circles, Deep
Channel, Passenger, Listen]

[Circles, Deep Channel, Listen, Passenger, Pink
Moon, Shiva Moon, Somersault]
```

Avant l'appel de sort().

Après l'appel de sort().

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Mais maintenant, il vous faut des objets Morceau, pas de simples chaînes.

Maintenant, votre patron veut de vraies instances de la classe Morceau dans la liste, et non des chaînes, afin que chaque morceau puisse avoir plus de données. Comme le nouveau jukebox produit plus d'informations, le même fichier doit avoir quatre paramètres au lieu de deux.

La classe Morceau est vraiment simple, avec une seule caractéristique intéressante — la méthode toString() redéfinie. Souvenez-vous : la méthode toString() étant définie dans la classe Object, toutes les classes Java en héritent. Et puisque toString() est appelée sur un objet quand il est affiché (System.out.println(unObjet)), vous devez la redéfinir pour afficher quelque chose de plus lisible que le code d'identifiant unique par défaut. Quand vous afficherez une liste, toString() sera appelée sur chaque objet.

ListeMorceauxPlus.txt

```
Pink Moon/Nick Drake/5/80
Somersault/Zero 7/4/84
Shiva Moon/Prem Joshua/6/120
Circles/BT/5/110
Deep Channel/Afro Celts/4/120
Passenger/Headmix/4/100
Listen/Tahiti 80/5/90
```

Le nouveau fichier contient quatre attributs au lieu de deux. Et comme nous les voulons TOUS dans notre liste, nous devons créer une classe Morceau avec des variables d'instance pour ces quatre attributs.

```
class Morceau {
    String titre;
    String artiste;
    String classement;
    String bpm;

    Morceau(String t, String a, String c, String b) {
        titre = t;
        artiste = a;
        classement = c;
        bpm = b;
    }

    public String getTitre() {
        return titre;
    }

    public String getArtiste() {
        return artiste;
    }

    public String getClassement() {
        return classement;
    }

    public String getBpm() {
        return bpm;
    }

    public String toString() {
        return titre;
    }
}
```

Quatre variables d'instance pour les quatre attributs du morceau dans le fichier.

Les variables sont toutes initialisées dans le constructeur quand le nouveau Morceau est créé.

Les méthodes get pour les quatre attributs.

Nous redéfinissons toString() parce que nous voulons que System.out.println(unObjetMorceau) affiche le titre. System.out.println(uneListeDeMorceaux) appellerait la méthode toString() de CHAQUE élément de la liste.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Modifier le code du Jukebox pour utiliser des Morceaux et non des chaînes

Votre code ne change pas beaucoup : le code des E/S est le même et celui de l'analyse est identique (String.split()), sauf qu'il y aura cette fois quatre paramètres pour chaque morceau/ligne, et que les quatre seront utilisés pour créer un nouvel objet Morceau. Et, bien sûr, l'ArrayList sera de type <Morceau> au lieu de <String>.

Créer une ArrayList d'objets de type Morceau et non de type String.

```
import java.util.*;
import java.io.*;

public class Jukebox3 {

    ArrayList<Morceau> listeMorceaux = new ArrayList<Morceau>();
    public static void main(String[] args) {
        new Jukebox3().go();
    }
    public void go() {
        getMorceaux();
        System.out.println(listeMorceaux);
        Collections.sort(listeMorceaux);
        System.out.println(listeMorceaux);
    }
    void getMorceaux() {
        try {
           File fichier = new File("ListeMorceauxPlus.txt");
           BufferedReader br = new BufferedReader(new FileReader(fichier));
           String ligne =  null;
           while ((ligne= br.readLine()) != null) {
              ajouterMorceau(ligne);
           }
        } catch(Exception ex) {
           ex.printStackTrace();
        }
    }

    void ajouterMorceau(String ligneAAnalyser) {
        String[] param = ligneAAnalyser.split("/");

        Morceau morceauSuivant = new Morceau(param[0], param[1], param[2], param[3]);
        listeMorceaux.add(morceauSuivant);
    }
}
```

Créer un nouvel objet Morceau en utilisant les quatre paramètres (autrement dit le quatrième bout d'information du fichier pour cette ligne), puis ajouter le Morceau à la liste.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Cela ne compile pas !

Quelque chose ne va pas... La documentation de la classe Collections montre clairement qu'il y a une méthode sort() , qui accepte une List.

ArrayList est-une List, parce qu'ArrayList implémente l'interface List, et donc... cela devrait fonctionner.

Mais çela ne fonctionne pas !

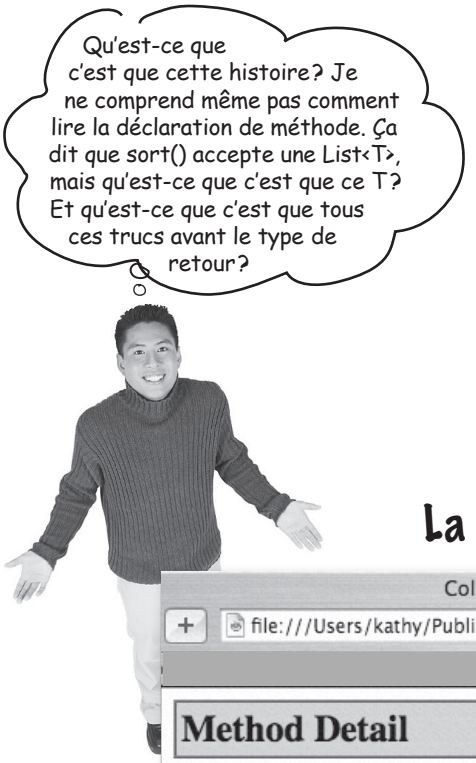
Si le compilateur dit qu'il ne trouve pas de méthode sort() qui accepte une ArrayList<Morceau>, c'est peut-être qu'il n'aime pas une ArrayList d'objets Morceau ? L'ArrayList<String> ne le dérangeait pas, alors où est la différence importante entre Morceau et String ? Qu'est-ce qui fait échouer le compilateur ?

```
Fichier  Edition  Fenêtre  Aide  LaPoisse
%javac Jukebox3.java
Jukebox3.java:15: cannot find symbol
symbol  : method sort(java.util.
ArrayList<Morceau>)
location: class java.util.Collections
                Collections.sort(listeMorceaux);
                           ^
1 error
```

Et, bien sûr, vous vous êtes probablement déjà demandé « Quel serait le critère de tri ? ». Comment la méthode sort() pourrait-elle savoir ce qui fait qu'un Morceau et supérieur ou inférieur à un autre ? À l'évidence, si vous voulez que ce soit le titre qui détermine la façon dont les morceaux sont triés, il vous faut un moyen de dire à la méthode qu'elle doit utiliser cette information, et non, par exemple, le nombre de battements par minute.

Nous allons entrer dans les détails dans quelques pages, mais voyons tout d'abord pourquoi le compilateur ne nous laisse même pas transmettre une ArrayList de type Morceau à la méthode sort().

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>



La déclaration de la méthode sort()

Collections (Java 2 Platform SE 5.0)

file:///Users/kathy/Public/docs/api/index.html

Google

Method Detail

sort

```
public static <T extends Comparable<? super T>> void sort(List<T> list)
```

Sorts the specified list into ascending order, according to the *natural ordering* of its elements. All elements in the list must implement the `Comparable` interface. Furthermore, all elements in the list must be *mutually comparable* (that is, `e1.compareTo(e2)` must not throw a `ClassCastException` for any elements `e1` and `e2` in the list).

Dans la doc de l'API (en cherchant la classe java.util.Collections et en défilant jusqu'à la méthode sort()), on constate que celle-ci est déclarée de façon... *étrange*. Ou du moins différemment de ce que nous avons vu jusqu'ici.

C'est parce que la méthode sort() (ainsi que d'autres éléments du framework de collections Java) utilise énormément les *génériques*. Chaque fois que vous voyez quelque chose entre chevrons dans du code source Java ou dans la documentation, cela veut dire génériques — une fonctionnalité ajoutée à Java 5.0. On dirait donc que nous allons devoir apprendre comment interpréter la documentation avant de comprendre pourquoi on peut trier des objets String dans une ArrayList, mais pas des objets Morceau.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Génériques = fiabilité du typage

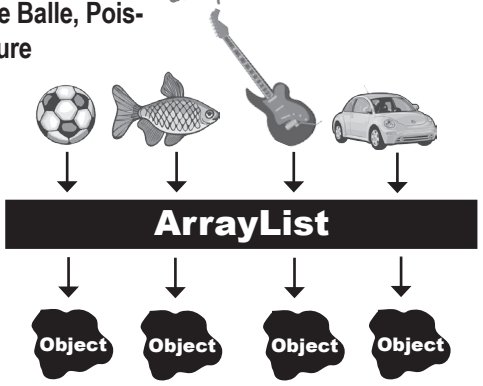
Nous ne le répéterons pas — *virtuellement tout le code que vous écrire et qui utilise des génériques sera du code lié aux collections.* Même s'il existe d'autres façons de les utiliser, le principal intérêt des génériques est qu'ils vous permettent d'écrire des collections typées de façon fiable. Autrement dit, du code qui oblige le compilateur à vous arrêter si vous mettez un Chien dans une liste de Canards.

Avant les génériques (ce qui veut dire avant Java 5.0), le compilateur se moquait totalement de ce que vous mettiez dans une collection, parce que tout ce que les collections contenaient était de type Object. Vous pouviez placer n'importe quoi dans une ArrayList : c'était comme si toutes les ArrayLists étaient déclarées en tant que ArrayList<Object>.

Avant les génériques, il n'y avait pas moyen de déclarer le type d'une ArrayList, et sa méthode add() acceptait donc un type Object.

SANS génériques

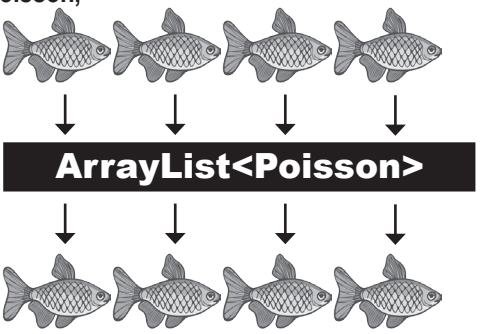
Les objets ENTRENT sous forme de référence de type Balle, Poisson, Guitare et Voiture



Et SORTENT sous forme de référence de type Object

Avec les génériques

Les objets ENTRENT sous forme de référence de type Poisson,



Et SORTENT sous forme de référence de type Poisson

Avec les génériques, vous pouvez créer des collections au typage sûr, qui permettent d'intercepter plus de problèmes au moment de la compilation au lieu d'avoir des erreurs d'exécution.

Sans les génériques, le compilateur vous aurait laissé allégrement mettre une Citrouille dans une ArrayList censée ne contenir que des Chats.

Maintenant, avec les génériques, vous ne pouvez placer que des objets Poisson dans une ArrayList<Poisson>, et les objets sortent donc comme des références de Poisson. Plus besoin d'avoir peur que quelqu'un y mettre une Volkswagen ou que ce qui en ressort ne soit pas vraiment convertible en référence de Poisson.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Apprendre les génériques

Sur les dizaines de choses que vous pourriez apprendre sur les génériques, il n'y en a vraiment que trois qui comptent pour la plupart des programmeurs :

1 Créer des instances de *classes* « générifiées » (comme ArrayList)

Quand vous créez une ArrayList, vous devez spécifier le type des objets que vous autorisez dans la liste, comme pour les bons vieux tableaux.

```
new ArrayList<Morceau>()
```

2 Déclarer et affecter des *variables* de types génériques

Comment le polymorphisme fonctionne-t-il réellement avec des types génériques ? Si vous avez une variable de référence ArrayList<Animal>, pouvez-vous lui affecter une ArrayList<Chien> ? Et une référence List<Animal> ? Pouvez-vous lui affecter une ArrayList<Animal> ? Nous allons voir...

```
List<Morceau> listeMorceaux =
      new ArrayList<Morceau>()
```

3 Déclarer (et invoquer) des *méthodes* qui acceptent des types génériques

Si vous avez une méthode qui accepte en paramètre, par exemple, une ArrayList d'objets Animal, qu'est-ce que cela veut dire réellement ? Pouvez-vous également lui transmettre une ArrayList d'objets Chien ? Nous verrons un certain nombre de problème de polymorphisme subtils et délicats qui sont très différents de la façon dont vous écrivez des méthodes qui acceptent de bons vieux tableaux.

(Ce point est en fait identique au point 2, mais cela vous montre l'importance que nous lui accordons.)

```
void truc(List<Morceau>
list)

x.truc(listeMorceaux)
```

Q : Mais est-ce que je ne dois pas aussi apprendre à créer mes PROPRES classes génériques ? Que se passe-t-il si je veux créer un type de classe qui permet à celui qui l'instancie de décider le type d'objets qu'elle utilisera ?

R : Il y a assez peu de chances que vous le fassiez. Réfléchissez : les concepteurs de l'API ont créé toute une bibliothèque de classes collections qui constituent la plupart des structures de données dont vous avez besoin. Virtuellement les seules classes qui ont réellement besoin d'être génériques sont les classes collections, autrement dit, les classes conçues pour contenir d'autres éléments. De plus, vous voulez que les programmeurs les utilisent pour spécifier de quel type sont ces éléments quand ils déclarent et instancient la classe collection.

Oui, vous pourriez vouloir créer des classes génériques, mais c'est l'exception et nous n'en parlerons pas ici. (Mais ce dont nous parlerons vous permettra de le faire.)

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Utiliser les CLASSES génériques

Comme ArrayList est notre type « générifiée » le plus utilisé, nous allons commencer par regarder sa documentation. Les deux endroits à observer dans une classe « générifiée » sont :

1) La déclaration de la classe.

3) Les déclarations des méthodes qui vous permettent d'ajouter des éléments.

Représentez-vous le « E » comme remplaçant le « type d'élément que vous voulez que cette collection contienne et retourne ». (E signifie Element.)

Comprendre la documentation d'ArrayList

(Ou : quelle est la vraie signification du « E » ?)

Le « E » remplace le VRAI type que vous utilisez et créez une ArrayList

ArrayList étant une sous-classe d'AbstractList, tout type que vous spécifiez pour l'ArrayList est automatiquement utilisé comme type de l'AbstractList.

```
public class ArrayList<E> extends AbstractList<E> implements List<E>  ... {

      public boolean add(E o)

      // reste du code
}
```

Voici la partie importante ! Quel que soit « E », il détermine le type d'objets que vous êtes autorisé à ajouter dans l'ArrayList.

Le type (la valeur de <E>) devient également celui de l'interface List.

Le « E » représente le type utilisé pour créer une instance d'ArrayList. Quand vous voyez un « E » dans la documentation de la classe ArrayList, vous pouvez effectuer mentalement une recherche/remplacement et lui substituer le <type> que vous utiliserez pour instancier ArrayList.

En conséquence, new ArrayList<Morceau> signifie que « E » devient « Morceau » dans toute déclaration de variable ou de méthode qui utilise « E ».

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Utiliser les paramètres de type avec ArrayList

Ce code :

```
ArrayList<String> cetteListe = new ArrayList<String>
```

signifie que l'ArrayList :

```
public class ArrayList<E> extends AbstractList<E> ... {

      public boolean add(E o)
      // reste du code
}
```

est traitée par le compilateur comme :

```
public class ArrayList<String> extends AbstractList<String>... {

      public boolean add(String o)
      // reste du code
}
```

Autrement dit, le « E » est remplacé par le type réel (également appelé paramètre de type) que vous utilisez quand vous créez l'ArrayList. C'est pourquoi la méthode add() d'ArrayList ne vous laisse ajouter que des objets d'un type de référence compatible avec le type de « E ». Si vous créez une ArrayList**<String>**, la méthode add() devient soudain **add(String o)**. Si vous créez une ArrayList de type Chien, la méthode add() devient subitement **add(Chien o)**.

Q : « E » est-elle la seule lettre qu'on puisse employer ? Parce que j'ai vu un « T » dans la documentation de sort()....

R : Vous pouvez employer tout ce qui est un identifiant légal en Java, autrement dit tout ce qui peut servir pour un nom de méthode ou de variable. Mais on utilise une seule lettre par convention (et mieux vaut procéder ainsi). Une autre convention consiste à employer « T », à moins d'écrire spécifiquement une classe collection, auquel cas on utilise « E » pour représenter « le type de l'élément que la collection contiendra ».

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Utiliser des MÉTHODES génériques

Dire qu'une classe est générique signifie que sa *déclaration* comprend un paramètre de type. Dire qu'une méthode est générique signifie que sa signature contient un paramètre de type.

On peut utiliser les paramètres de type dans une méthode de différentes manières :

1 Utiliser un paramètre de type défini dans la déclaration de la classe

```
public class ArrayList<E> extends AbstractList<E> ... {
    public boolean add(E o)
```

Vous pouvez utiliser le « E » ici UNIQUEMENT parce qu'il est déjà défini dans la classe.

Quand vous déclarez un paramètre de type dans la classe, vous pouvez l'utiliser partout où vous utiliseriez un vrai type de classe ou d'interface. Le type déclaré dans l'argument de la méthode est remplacé par celui que vous utilisez quand vous instanciez la classe.

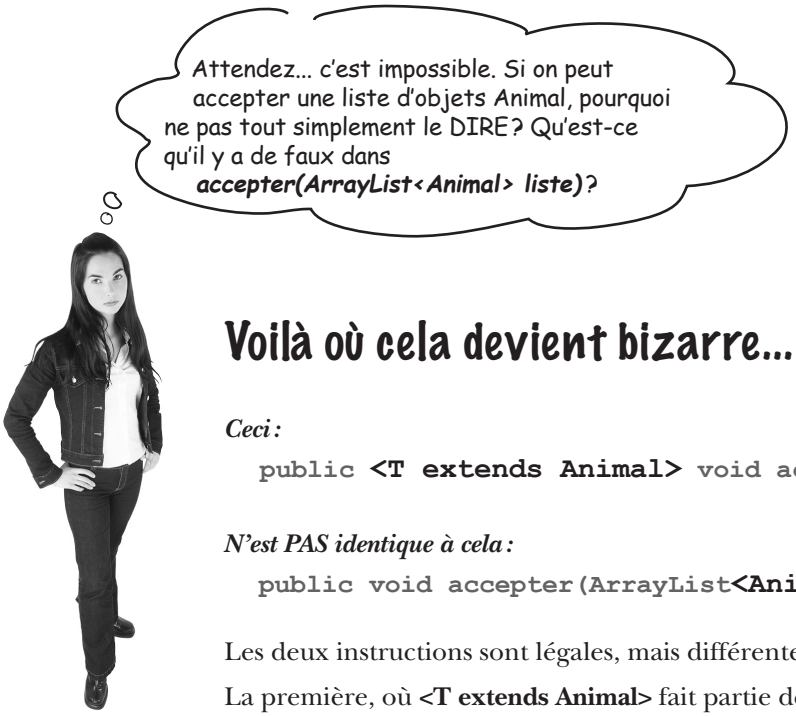
2 Utiliser un paramètre de type qui n'a PAS été défini dans la déclaration de la classe

```
public <T extends Animal> void accepter(ArrayList<T> liste)
```

Ici, on peut utiliser <T> parce qu'on a introduit « T » avant dans la déclaration de la méthode.

Si la classe elle-même n'utilise pas de paramètre de type, vous pouvez toujours en spécifier un pour une méthode, en le déclarant dans un espace vraiment inhabituel (mais disponible) : avant le type de retour. Cette méthode dit que T peut être « n'importe quel type d'Animal ».

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>



Voilà où cela devient bizarre...

Ceci :

```
public <T extends Animal> void accepter(ArrayList<T> liste)
```

N'est PAS identique à cela :

```
public void accepter(ArrayList<Animal> liste)
```

Les deux instructions sont légales, mais différentes !

La première, où **<T extends Animal>** fait partie de la déclaration de la méthode, signifie que toute ArrayList déclarée d'un type qui est Animal ou un des sous-types d'Animal (comme Chien ou Chat) est légale. Vous pouvez donc invoquer la méthode du haut en utilisant ArrayList<Chien>, ArrayList<Chat> ou ArrayList<Animal>.

Mais... celle du bas, où l'argument de la méthode est (ArrayList<Animal> liste) signifie que *seule* une ArrayList<Animal> est légale. Autrement dit, si la première version accepte une ArrayList de n'importe quel type d'Animal (Animal, Chien, Chat, etc.), la seconde n'accepte qu'une ArrayList de type Animal ; ni ArrayList<Chien> ni ArrayList<Chat>, mais seulement ArrayList<Animal>.

Et oui, on dirait que cela va à l'encontre du polymorphisme. Mais tout deviendra clair quand nous reverrons ceci en détail à la fin du chapitre. Souvenez-vous que nous en parlons uniquement parce que nous cherchons toujours un moyen de trier cette ListeMorceaux, ce qui nous a conduits à consulter l'API et à trouver cette méthode sort() , qui a cette drôle de déclaration de type générique.

Pour l'instant, contentez-vous de savoir que la syntaxe de la version du haut est légale, et qu'elle signifie que vous pouvez transmettre un objet ArrayList instancié comme un Animal ou un sous-type quelconque d'Animal.

Revenons maintenant à notre méthode sort() ...

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Rappelez-vous où nous en étions

```
Fichier Edition Fenêtre Aide LaPoisse
%javac Jukebox3.java
Jukebox3.java:15: cannot find symbol
symbol  : method sort(java.util.
ArrayList<Morceau>)
location: class java.util.Collections
                Collections.sort(listeMorceaux);
                                ^
1 error
```

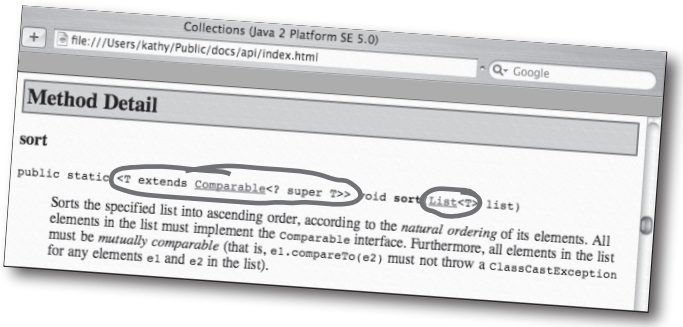
```
import java.util.*;
import java.io.*;

public class Jukebox3 {
    ArrayList<Morceau> listeMorceaux = new ArrayList<Morceau>();
    public static void main(String[] args) {
        new Jukebox3().go();
    }
    public void go() {
        getMorceaux();
        System.out.println(listeMorceaux);
        Collections.sort(listeMorceaux);
        System.out.println(listeMorceaux);
    }
    void getMorceaux() {
        try {
           File fichier = new File("ListeMorceauxPlus.txt");
           BufferedReader br = new BufferedReader(new FileReader(fichier));
           String ligne =  null;
           while ((ligne= br.readLine()) != null) {
              ajouterMorceau(ligne);
           }
        } catch(Exception ex) {
           ex.printStackTrace();
        }
    }
    void ajouterMorceau(String ligneAAnalyser) {
        String[] param = ligneAAnalyser.split("/");
        Morceau morceauSuivant = new Morceau(param[0], param[1], param[2], param[3]);
        listeMorceaux.add(morceauSuivant);
    }
}
```

Voilà où le bât blesse! Tout allait bien avec une ArrayList<String>, mais dès que nous essayons de trier une ArrayList<Morceau>, rien ne va plus.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Revoyons la méthode sort()

Nous revoici, essayant de lire la documentation de la méthode sort() pour comprendre pourquoi nous pouvions trier une liste d'objets String, mais pas une liste d'objets Morceau. Et on dirait que la réponse est...



La méthode sort() n'accepte que des listes d'objets Comparable.

Comme Morceau n'est PAS un sous-type de Comparable, on ne peut pas trier la liste de Morceaux.

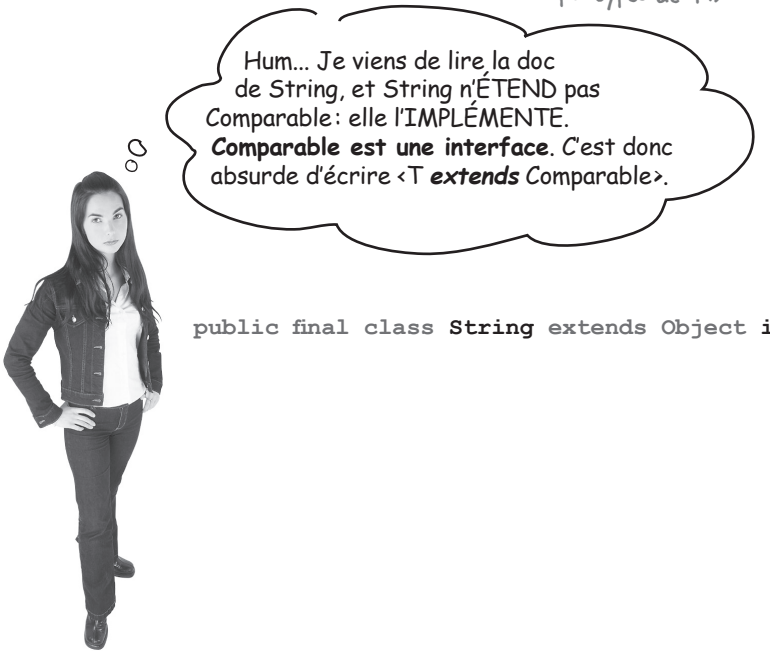
Du moins pas encore...

```
public static <T extends Comparable<? super T>> void sort(List<T> list)
```

Ceci veut dire « Quel que soit "T", il doit être de type Comparable."

(Ignorez ceci pour le moment. Si vous ne le pouvez pas, cela signifie juste que le paramètre de type pour Comparable doit être de type T ou d'un des supertypes de T.)

Vous ne pouvez transmettre qu'une List (ou un sous-type de List, comme ArrayList) qui utilise un type paramétré qui « étend Comparable ».




```
public final class String extends Object implements Serializable,
                                                    Comparable<String>, CharSequence
```


Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Avec les génériques, « extends » signifie « étend ou implémente »

Les concepteurs de Java devaient vous fournir un moyen d'imposer une contrainte à un type paramétré, pour que vous puissiez le restreindre à, par exemple, seulement des sous-classes d'Animal. Mais vous devez également pouvoir contraindre un type pour n'autoriser que les classes qui implémentent une interface donnée. Voici donc une situation où l'on a besoin d'une syntaxe qui fonctionne dans les deux situations — héritage et implémentation. Autrement dit qui fonctionne à la fois pour **extends** et pour **implements**.

Et le mot gagnant est... **extends**. Mais il signifie réellement « est-une » et fonctionne indépendamment du fait que le type de droite soit une interface ou une classe.

Avec les génériques, le mot-clé « extends » signifie en réalité « est-une » et fonctionne pour les classes ET les interfaces.

Comme Comparable est une interface, ceci signifie RÉELLEMENT que T doit être un type qui implémente l'interface Comparable.

```
public static <T extends Comparable<? super T>> void sort(List<T> list)
```

Peu importe que ce qui se trouve à droite soit une classe ou une interface... Vous écrirez toujours « extends ».

Q : Pourquoi n'ont-ils pas inventé un nouveau mot-clé, « is » ?

R : Ajouter un nouveau mot-clé au langage n'est PAS une mince affaire, parce que cela empêcherait le code Java que vous avez écrit dans une précédente version de fonctionner. Réfléchissez : vous pourriez avoir utilisé une variable nommée « is » (ce que nous faisons dans ce livre pour représenter un Input Stream). Et comme vous n'avez pas le droit d'utiliser des mots-clés comme identifiants, cela signifierait que tout le code que vous avez écrit avant que ce ne soit un mot réservé cesserait de fonctionner. Donc chaque fois que les ingénieurs de Sun ont une chance de réutiliser un mot-clé existant, comme ils l'ont fait pour « extends », ils choisissent généralement cette solution. Mais parfois ils n'ont pas le choix...

Quelques nouveaux mots-clés (très peu) ont été ajoutés au langage, comme ***assert*** en Java 1.4 et ***enum*** en Java 5.0 (nous verrons enum dans l'annexe). Cela peut endommager votre code, mais vous avez parfois l'option de compiler et d'exécuter avec une version plus récente de Java en spécifiant qu'elle doit se comporter comme si c'était une version plus ancienne. Pour ce faire, vous appelez le compilateur ou la JVM sur la ligne de commande avec une option spéciale pour dire « Oui, oui, je SAIS que c'est Java 1.4, mais fais semblant que c'est 1.3, parce que j'utilise dans mon code une variable nommée *assert* et que je l'ai écrit quand vous avez dit qu'il n'y avait pas de problème ! ».

(Pour savoir de quelles options vous disposez, entrez *javac* (pour le compilateur) ou *java* (pour la JVM) sur une ligne de commandes, sans rien d'autre après. Vous devriez voir une liste d'options possibles. Vous en saurez plus sur ces options dans le chapitre sur le déploiement.)

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Nous savons enfin ce qui ne va pas...
La classe Morceau doit implémenter Comparable

Nous ne pouvons passer l'ArrayList<Morceau> à la méthode sort() que si la classe Morceau implémente Comparable, puisque c'est ainsi que la méthode a été déclarée. Un rapide coup d'œil sur l'API montre que l'interface Comparable est très simple, avec une seule méthode à implémenter :

java.lang.Comparable

```
public interface Comparable<T> {
   int compareTo(T o);
}
```

Et voici ce que dit la documentation de compareTo() :

Retourne :
un entier négatif, zéro ou un entier positif selon que cet objet est inférieur, égal ou supérieur à l'objet spécifié.

On dirait que la méthode compareTo() va être appelée sur un objet Morceau, passant à ce Morceau une référence à un Morceau différent. Le morceau qui exécute la méthode compareTo() doit déterminer si le Morceau qu'il a reçu doit être placé plus haut, plus bas ou au même endroit dans la liste.

Votre tâche principale consiste maintenant à déterminer ce qui fait qu'un morceau est supérieur à un autre, puis à implémenter la méthode compareTo() en fonction. Un nombre négatif (n'importe lequel) indique que le morceau reçu est supérieur au Morceau qui exécute la méthode. Un nombre positif indique que le Morceau qui exécute la méthode est supérieur au Morceau passé à la méthode compareTo(). Zéro signifie que les deux sont égaux (au moins pour les besoins du tri... cela ne signifie pas nécessairement qu'il s'agit du même objet). Vous pourriez par exemple avoir deux morceaux portant le même titre.

Mais c'est un problème très différent, que nous verrons plus tard...).

Voilà la grande question : qu'est-ce qui rend *un* morceau inférieur, égal ou supérieur à un *autre* morceau ?

Vous ne pouvez pas implémenter l'interface Comparable tant que vous n'avez pas pris cette décision.

À vos crayons

Écrivez votre idée et le pseudocode (ou mieux le VRAI code) pour implémenter la méthode compareTo() afin qu'elle trie les objets Morceau par titre.

Indication : si vous êtes sur la bonne piste, moins de trois lignes de code devraient suffire !

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

La nouvelle classe Morceau, améliorée et comparable

Comme nous avons décidé que nous voulions trier par titre, nous implémentons la méthode compareTo() pour qu'elle compare le titre du Morceau transmis à la méthode à celui sur lequel elle a été invoquée. Autrement dit, le morceau qui exécute la méthode doit décider quel est le résultat de la comparaison de son titre au titre passé en paramètre à la méthode.

Mmmm... Nous savons que la classe doit connaître l'ordre alphabétique, puisque la méthode sort() fonctionnait sur une liste de type String. Comme nous savons aussi que String a une méthode compareTo(), pourquoi donc ne pas l'appeler ? Ainsi, nous pouvons laisser un titre se comparer à un autre et nous n'avons pas besoin d'écrire l'algorithme de tri !

Généralement, cela correspond... Nous spécifions le type auquel on peut comparer la classe qui implémente.

Cela veut dire que les objets Morceau peuvent être comparés à d'autres objets Morceau pour les besoins du tri.

```
class Morceau implements Comparable<Morceau> {
   String titre;
   String artiste;
   String classement;
   String bpm;

   public int compareTo(Morceau m) {
       return titre.compareTo(m.getTitre());
   }

   Morceau(String t, String a, String c, String b) {
      titre = t;
      artiste = a;
      classement = c;
      bpm = b;
   }

   public String getTitre() {
       return titre;
   }

   public String getArtiste() {
      return artiste;
   }

   public String getClassement() {
      return classement;
   }

  public String getBpm() {
    return bpm;
  }

  public String toString() {
     return titre;
  }
}
```

La méthode sort() transmet un Morceau à compareTo() pour voir comment il se compare à celui sur lequel la méthode a été invoquée.

Simple ! Il suffit de déléguer aux objets String qui contiennent les titres, puisque nous savons que String a une méthode compareTo().

Cette fois, cela fonctionne. Le code affiche la liste, puis appelle sort() qui classe les morceaux dans l'ordre alphabétique des titres.

Fichier Edition Fenêtre Aide Ambiance

```
%java Jukebox3

[Pink Moon, Somersault, Shiva Moon, Circles, Deep
Channel, Passenger, Listen]

[Circles, Deep Channel, Listen, Passenger, Pink
Moon, Shiva Moon, Somersault]
```


Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Nous pouvons trier la liste, mais...

Il y a un nouveau problème. Bruno veut pouvoir trier la liste de morceaux de deux façons différentes : une par titre et une par artiste !

Mais quand on rend un élément de collection comparable (en implémentant l'interface Comparable), on n'a qu'une seule possibilité d'implémenter la méthode compareTo(). Que pouvez-vous donc faire ?

Il y a une solution horrible : utiliser une variable indicateur dans la classe Morceau, puis faire un test « if » dans compareTo() et donner un résultat différent selon que l'indicateur est positionné pour utiliser le titre ou l'artiste dans la comparaison.

Mais c'est une solution affreuse et fragile, et il y a bien mieux. Quelque chose qui a été intégré à l'API à cette fin — quand vous voulez pouvoir trier les mêmes objets sur plusieurs critères

Regardons de nouveau la doc de la classe Collections. Il y a une seconde méthode sort(), et elle accepte un Comparateur.

Collections (Java 2 Platform SE 5.0)

file:///Users/kathy/Public/docs/api/index.html

Google

Jellyvision, Inc. | Collections ...form SE 5.0) | Caffeinated ...d Brain Day | Brand Noise | Diva Marketing

static <K,V> Map<K,V>	singletonMap(K key, V value) Returns an immutable map, mapping only the specified key to the specified value.
static ...uper T>> void	sort(List<T> list) Sorts the specified list into ascending order, according to the *natural ordering* of its elements.
static ...<T> void	sort(List<T> list, Comparator<? super T> c) Sorts the specified list according to the order induced by the specified comparator.

La méthode sort() est surchargée pour accepter quelque chose nommé un Comparateur.

Note perso : voir comment faire pour obtenir/créer un Comparateur qui peut comparer et ordonner les morceaux par artiste et non par titre...

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Utiliser un Comparateur personnalisé

Un élément d'une liste peut se comparer à un autre élément de son propre type d'une seule façon, en utilisant sa méthode compareTo(). Mais un Comparateur est externe au type d'élément que vous comparez : c'est une classe séparée. Vous pouvez donc en créer autant que vous voulez ! Vous voulez trier par artiste ? Créez un ComparateurArtiste. Trier sur les battements par minute ? Créez un ComparateurBPM.

Il suffit ensuite d'appeler la méthode sort() surchargée qui accepte la List et le Comparateur qui vont l'aider à mettre les objets dans l'ordre.

La méthode sort() qui accepte un Comparateur va l'utiliser à la place de la méthode compareTo() de l'élément pour ordonner les éléments. Autrement dit, si votre méthode sort() reçoit un Comparateur, elle n'appellera même pas la méthode compareTo() des éléments de la liste. À la place, la méthode sort() appellera la **méthode** compare() sur le Comparateur.

Voici donc les règles :

- **L'appel de la méthode à un seul argument sort(List o) signifie que c'est la méthode compareTo() de l'élément qui détermine l'ordre. Les éléments de la liste DOIVENT donc implémenter l'interface Comparable.**

- **Avec l'appel de sort(List o, Comparator c), la méthode compareTo() de l'élément ne sera PAS appelée et la méthode compare() du Comparateur sera utilisée à la place. Cela signifie que les éléments de la liste n'ont PAS besoin d'implémenter interface Comparable.**

java.util.Comparator

```
public interface Comparator<T> {
   int compare(T o1, T o2);
}
```

Si vous passez un Comparateur à la méthode sort(), l'ordre de tri est déterminé par le Comparateur, non par la méthode compareTo() de l'élément.

Q : Est-ce que ça veut dire que si l'on a une classe qui n'implémente pas Comparable, et qu'on n'a pas le code source, on peut toujours trier des éléments en utilisant un Comparateur ?

R : Exactement. L'autre solution (quand elle est possible) serait de sous-classer l'élément et de faire en sorte que la sous-classe implémente Comparable.

Q : Mais pourquoi toutes les classes n'implémentent-elles pas Comparable ?

R : Croyez-vous vraiment que tout puisse être ordonné ? Si vous avez des types d'élément qui ne se prêtent à aucun ordre naturel, vous induiriez en erreur les autres programmeurs si vous implémentiez Comparable. Et vous ne prenez pas un énorme risque en n'implémentant pas Comparable, puisqu'un programmeur peut comparer ce qu'il veut en créant son propre Comparateur.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Actualiser le Jukebox avec un Comparateur

Nous avons fait trois nouvelles choses dans ce code :

1) Créer une classe interne qui implémente Comparator (et donc la ***méthode c***ompare() qui fait le travail précédemment effectué par ***compareTo()***).

2) Créer une instance de la classe interne Comparator.

3) Appeler la méthode sort() surchargée, en lui transmettant à la fois la liste de morceaux et l'instance de Comparator.

Note : nous avons également mis à jour la méthode toString() de la classe Morceau pour qu'elle affiche le titre et l'artiste. (Elle affiche ***titre: artiste,*** indépendamment de la façon dont la liste est triée.)

```
import java.util.*;
import java.io.*;

public class Jukebox5 {
   ArrayList<Morceau> listeMorceaux = new ArrayList<Morceau>();
   public static void main(String[] args) {
      new Jukebox5().go();
   }

   class ComparateurArtiste implements Comparator<Morceau> {
     public int compare(Morceau un, Morceau deux) {
         return un.getArtiste().compareTo(deux.getArtiste());
     }
   }

   public void go() {
     getMorceaux();
     System.out.println(listeMorceaux);
     Collections.sort(listeMorceaux);
     System.out.println(listeMorceaux);

     ComparateurArtiste  compareArtiste = new ComparateurArtiste();
     Collections.sort(listeMorceaux, compareArtiste);

     System.out.println(listeMorceaux);
   }

   void getMorceaux() {
       // code d'E/S ici
   }

   void ajouterMorceau(String ligneAAnalyser) {
       // analyser la ligne
       // et l'ajouter à la liste
   }
}
```

Créer une nouvelle classe interne qui implémente Comparator (notez que son paramètre de type correspond à celui que nous allons comparer — en l'occurrence des objets Morceau.)

Ceci devient une chaîne (l'artiste)

Nous laissons les variables String (pour l'artiste) faire la comparaison, puisque les objets String savent déjà se classer alphabétiquement.

Créer une instance de la classe interne Comparator.

Appeler sort(), en lui passant la liste et une référence au nouvel objet Comparator.

Note: nous avons fait du titre le critère de tri par défaut en faisant utiliser les titres par la méthode compareTo() de Morceau. Une autre façon de procéder serait d'implémenter le tri par titre et le tri par artiste comme deux classes Comparateur internes, Morceau n'implémentant pas Comparable du tout. Dans ce cas, nous utiliserions toujours la version à deux arguments de Collections.sort().

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

À vos crayons

Rétro-ingénierie

Supposez que ce code existe dan un seul fichier. Votre tâche consiste à remplir les blancs pour que le programme affiche le résultat ci-dessous.

Note : les réponses sont en fin de chapitre.

```
import ________________;

public class TriMontagnes {

  LinkedList_______________ mtgn = new LinkedList_____________();

  class ComparateurNoms ____________________________________ {
    public int compare(Montagne un, Montagne deux) {

      return ____________________________;
    }
  }
  class ComparateurHauteurs ________________________________ {
    public int compare(Montagne un, Montagne deux) {

      return (___________________________);
    }
  }
  public static void main(String [] args) {
    new TriMontagnes().go();
  }
  public void go() {
    mtgn.add(new Montagne("Cervin", 4482));
    mtgn.add(new Montagne("MontBlanc", 4807));
    mtgn.add(new Montagne("JungFrau", 4168));
    mtgn.add(new Montagne("Pelvoux", 3955));

    System.out.println("par ordre d'insertion:\n" + mtgn);
    ComparateurNoms cn = new ComparateurNoms();

    ________________________________;
    System.out.println("par nom:\n" + mtgn);
    ComparateurHauteurs ch = new ComparateurHauteurs();

    ________________________________;
    System.out.println("par hauteur:\n" + mtgn);
  }
}

class Montagne {

  ________________;
  ________________;

  ______________________ {
    ________________;
    ________________;
  }
  ________________________ {
    ________________________________;
  }
}
```

Résultat :

```
Fichier Edition  Fenêtre  Aide CeluiCiPourBob
%java TriMontagnes
dans l'ordre d'insertion:
[Cervin 4482, MontBlanc 4807, JungFrau 4168, Pelvoux 3955]
par nom:
[Cervin 4482, JungFrau 4168, MontBlanc 4807 Pelvoux 3955]
par hauteur:
[MontBlanc 4807, Cervin 4482, JungFrau 4168 Pelvoux 3955]
```


Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

À vos crayons

Remplissez les blancs

Pour chacune des questions ci-dessous, remplissez le blanc avec l'un des mots de la liste des réponses possibles. Les réponses sont en fin de chapitre.

Réponses possibles :

Comparator,
Comparable,
compareTo( ),
compare( ),
oui,
non

Étant donné l'instruction compilable suivante :

```
Collections.sort(monArrayList);
```

1. Que doit implémenter la classe des objets stockés dans `monArrayList` ? _____
2. Quelle méthode la classe des objets stockés dans `monArrayList` doit-elle implémenter ? _____
3. La classe des objets stockés dans mon `monArrayList` peut-elle implémenter à la fois `Comparator` ET `Comparable` ? _____

Étant donné l'instruction compilable suivante :

```
Collections.sort(monArrayList, maComparaison);
```

4. La classe des objets stockés dans `monArrayList` peut-elle implémenter `Comparable` ? _____
5. La classe des objets stockés dans `monArrayList` peut-elle implémenter `Comparator` ? _____
6. La classe des objets stockés dans `monArrayList` doit-elle implémenter `Comparable` ? _____
7. La classe des objets stockés dans `monArrayList` doit-elle implémenter `Comparator` ? _____
8. Que doit implémenter la classe de l'objet `maComparaison` ? _____
9. Quelle méthode la classe de l'objet `maComparaison` doit-elle implémenter ? _____

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Aïe! Le tri fonctionne, mais maintenant nous avons des doublons...

Le tri fonctionne parfaitement, maintenant que nous savons trier à la fois sur le titre (avec la méthode compareTo() de l'objet Morceau) et l'*artiste* (avec la méthode compare() du Comparateur). Mais il y a un nouveau problème que nous n'avions pas détecté dans l'échantillon de test du fichier texte du juke-box : la liste triée contient des doublons.

Or, le juke-box continue à écrire dans le fichier sans s'occuper de savoir si le même morceau a déjà été joué (et donc écrit dans le fichier texte). Le fichier texte ListeMorceauxPlus.txt est un historique complet de chaque morceau qui a été joué et peut contenir le même plusieurs fois.

Fichier Edition Fenêtre Aide TropDeNotes

```
%java Jukebox4

[Pink Moon: Nick Drake, Somersault: Zero 7, Shiva Moon: Prem
Joshua, Circles: BT, Deep Channel: Afro Celts, Passenger:
Headmix, Listen: Tahiti 80, Listen: Tahiti 80, Listen: Tahiti
80, Circles: BT]

[Circles: BT, Circles: BT, Deep Channel: Afro Celts, Listen:
Tahiti 80, Listen: Tahiti 80, Listen: Tahiti 80, Passenger:
Headmix, Pink Moon: Nick Drake, Shiva Moon: Prem Joshua,
Somersault: Zero 7]

[Deep Channel: Afro Celts, Circles: BT, Circles: BT, Passenger:
Headmix, Pink Moon: Nick Drake, Shiva Moon: Prem Joshua, Listen:
Tahiti 80, Listen: Tahiti 80, Listen: Tahiti 80, Somersault:
Zero 7]
```

Avant le tri.

Après le tri utilisant la méthode compareTo() de Morceau (tri par titre).

Après le tri utilisant ComparateurArtiste (tri par artiste).

ListeMorceauxPlus.txt

```
Pink Moon/Nick Drake/5/80
Somersault/Zero 7/4/84
Shiva Moon/Prem Joshua/6/120
Circles/BT/5/110
Deep Channel/Afro Celts/4/120
Passenger/Headmix/4/100
Listen/Tahiti 80/5/90
Listen/Tahiti 80/5/90
Listen/Tahiti 80/5/90
Circles/BT/5/110
```

Le fichier texte ListeMorceauxPlus contient maintenant des doublons, parce que le juke-box continue à écrire chaque morceau joué, dans l'ordre. Quelqu'un a décidé de jouer « Listen » trois fois d'affilée, suivi de « Circles », un morceau qui a déjà été joué.

Nous ne pouvons pas changer la façon d'écrire dans le fichier parce que nous aurons parfois besoin de toutes ces informations. Il faut donc modifier le code Java.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

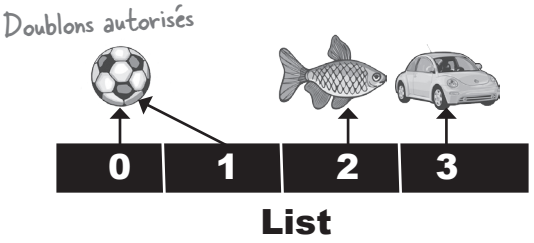
Il nous faut un Set au lieu d'une List

Dans l'API Collection, on trouve trois interfaces principales : **List**, **Set** et **Map**. ArrayList est une **List**, mais on dirait que ***Set*** est exactement ce qu'il nous faut.

- **LIST, quand la *séquence* compte**

  Des collections qui reconnaissent les ***indices.***

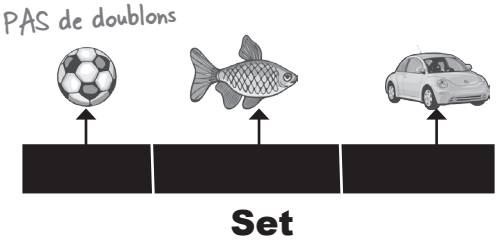
  Une List sait où se trouve quelque chose dans la liste. Vous pouvez avoir plusieurs éléments référençant le même objet.



- **SET, quand l'*unicité* compte**

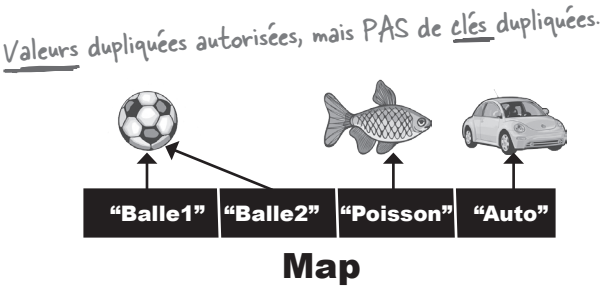
  Des collections qui ***interdisent les doublons.***

  Un Set sait qu'il y a déjà quelque chose dans la collection. Vous ne pouvez pas avoir plus d'un élément référençant le même objet (ou plus d'un élément référençant deux objets considérés comme égaux — nous verrons ce que signifie l'égalité des objets dans un moment).



- **MAP, pour *trouver quelque chose par clé***

  Des collections qui utilisent des ***paires clé-valeur.***
  Une Map connaît la valeur associée à une clé donnée. Vous pouvez avoir deux clés qui référencent la même valeur, mais vous ne pouvez pas avoir de clés dupliquées. Même si les clés sont généralement de type String (par exemple pour créer des listes de propriétés nom-valeur), une clé peut être un objet.

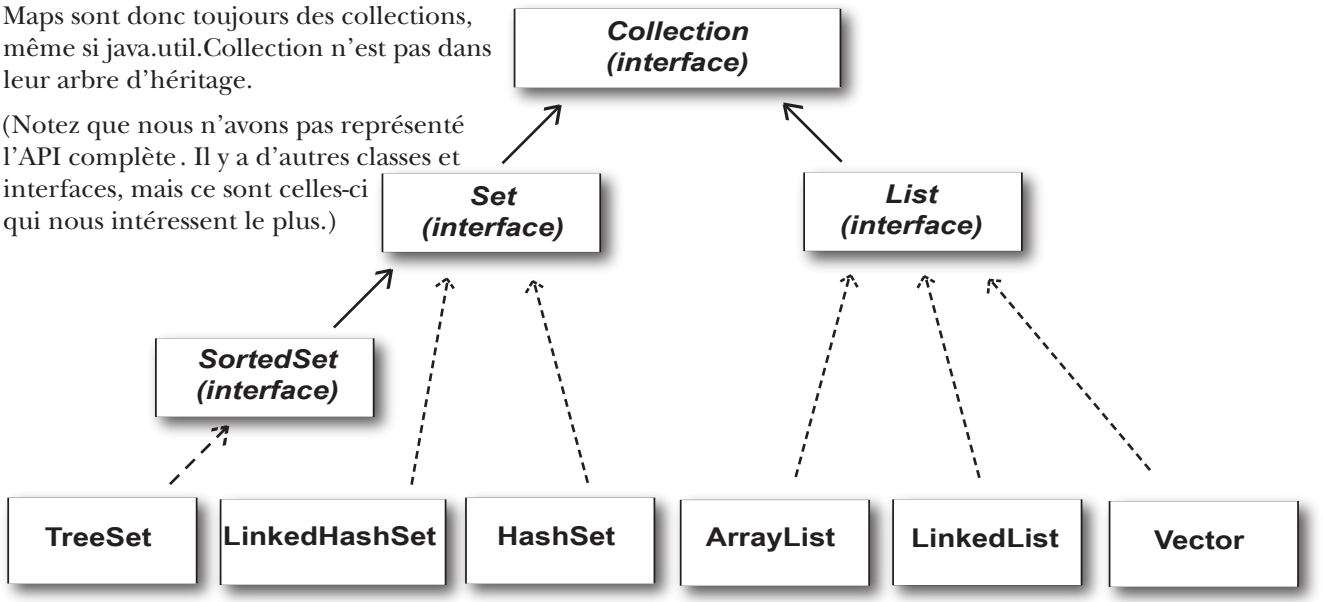


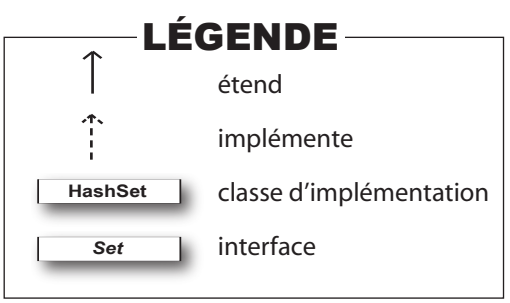
Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

L'API Collection (partielle)

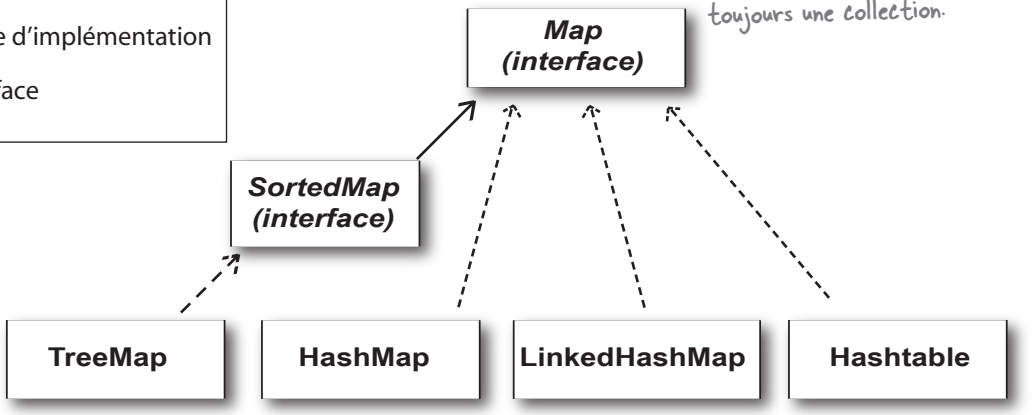
Remarquez que l'interface Map n'étend pas l'interface Collection, mais on la considère comme faisant partie du « Framework Collections » (alias API Collection). Les Maps sont donc toujours des collections, même si java.util.Collection n'est pas dans leur arbre d'héritage.

(Notez que nous n'avons pas représenté l'API complète. Il y a d'autres classes et interfaces, mais ce sont celles-ci qui nous intéressent le plus.)





Les Maps n'étendent pas java.util.Collection, mais on considère qu'elles font partie du « framework collections » de Java. Une Map est donc toujours une collection.



Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Utiliser un HashSet à la place d'ArrayList

Nous avons modifié le Jukebox pour placer les morceaux dans un HashSet. (Note : nous avons omis une partie du code, mais vous pouvez le recopier sur une ancienne version. Et pour que le résultat soit plus facile à lire, nous sommes revenus à l'ancienne version de la méthode toString(), pour qu'elle n'affiche que le titre au lieu du titre et de l'artiste.)

```
import java.util.*;
import java.io.*;

public class Jukebox6 {
   ArrayList<Morceau> listeMorceaux = new ArrayList<Morceau>();
   // méthode main, etc.

   public void go() {
      getMorceaux();
      System.out.println(listeMorceaux);
      Collections.sort(listeMorceaux);
      System.out.println(listeMorceaux);
      HashSet<Morceau> hashMorceau = new HashSet<Morceau>();
      hashMorceau.addAll(listeMorceaux);
      System.out.println(hashMorceau);
   }
   // méthodes getMorceaux() et ajouterMorceau()
}
```

Comme nous n'avons pas modifié getMorceaux(), elle continue à placer les morceaux dans une ArrayList

Ici nous créons un HashSet paramétré pour contenir des Morceaux.

HashSet a une méthode addAll() qui peut prendre une autre collection et l'utiliser pour remplir le HashSet. C'est comme si nous avions ajouté chaque morceau un à un (mais en beaucoup plus simple).

Fichier Edition Fenêtre Aide GetBetterMusic

```
%java Jukebox6

[Pink Moon, Somersault, Shiva Moon, Circles, Deep Channel,
Passenger, Listen, Listen, Listen, Circles]

[Circles, Circles, Deep Channel, Listen, Listen, Listen,
Passenger, Pink Moon, Shiva Moon, Somersault]

[Pink Moon, Listen, Shiva Moon, Circles, Listen, Deep Channel,
Passenger, Circles, Listen, Somersault]
```

Avant de trier l'ArrayList.

Après avoir trié l'ArrayList (par titre).

Après l'avoir placé dans un HashSet, et affiché ce dernier (nous n'avons pas rappelé sort()).

Le Set n'a servi à rien !!
Nous avons toujours des doublons !

(Et l'ordre du tri a été perdu quand nous avons mis la liste dans le HashSet, mais nous nous en inquiéterons plus tard...)

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Qu'est-ce qui rend deux objets égaux ?

La première question à se poser est : que sont deux doublons d'une référence de Morceau ? On doit les considérer comme égaux. S'agit-il juste de deux références au même objet ou de deux objets distincts qui portent le même titre ?

Cela ouvre une question fondamentale : *égalité des références* vs. *égalité des objets*.

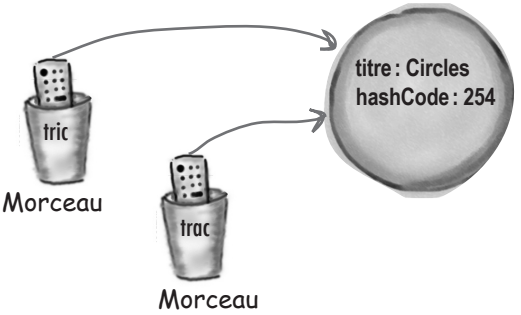
Si deux objets *tric* et *trac* sont égaux, *tric.equals(trac)* doit être vrai, et *tric* et *trac* doivent retourner la même valeur de *hashCode()*. Pour qu'un Set traite deux objets comme des doublons, vous devez redéfinir le méthode hashCode() et equals() héritées de la classe Object, afin que deux objets différents soient considérés comme égaux.

➤ Égalité des références

Deux références, un seul objet sur le tas.

Deux références qui renvoient au même objet sur le tas sont égales. Point. Si vous appelez la méthode **hashCode()** sur les deux références, vous obtiendrez le même résultat. Si vous ne redéfinissez pas hashCode(), le comportement par défaut (n'oubliez pas que vous en avez hérité de la classe Object) est que chaque objet aura un numéro unique (la plupart des versions de Java affectent un code de hachage en fonction de l'adresse mémoire de l'objet sur le tas, si bien que deux objets ne peuvent avoir le même).

Si vous voulez savoir si deux *références* renvoient au même objet, utilisez l'opérateur == , qui (souvenez-vous) compare les bits de deux variables. Si les deux références pointent sur le même objet, les bits seront identiques.



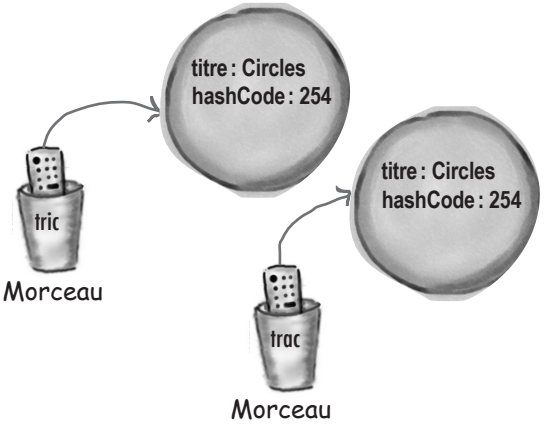

```
if (tric == trac) {
    // les deux références désignent
    // le même objet sur le tas
}
```

➤ Égalité des objets

Deux références, deux objets sur le tas, mais les objets sont considérés comme *équivalents*.

Si vous voulez traiter deux objets Morceau différents comme égaux (par exemple si vous avez décidé que deux morceaux sont le même parce que leur variable *titre* correspond), vous devez redéfinir *à la fois* la méthode **hashCode()** et la méthode **equals()** héritées de la classe Object.

Comme on l'a vu ci-dessus, si vous ne redéfinissez *pas* hashCode(), son comportement par défaut (hérité d'Object) consistera à attribuer à chaque objet un code de hachage unique. Il faut donc redéfinir hashCode() pour être sûr que deux objets équivalents retournent le même code. Mais il faut aussi redéfinir equals() pour que si vous l'appelez sur *l'un* des objets en passant l'autre objet, elle retourne toujours ***vrai***.




```
if (tric.equals(trac) && tric.hashCode() == trac.hashCode())
{
    // les deux références désignent soit un seul objet
    // soit deux objets qui sont égaux
}
```


Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Comment un HashSet détecte les doublons : hashCode() et equals()

Quand vous placez un objet dans un HashSet, il utilise le code de hachage de l'objet pour déterminer où il va le placer dans le Set. Mais il compare également ce code de hachage à celui de tous les autres objets présents dans le HashSet. Si aucun code ne correspond, le HashSet présume que ce nouvel objet n'est pas un doublon.

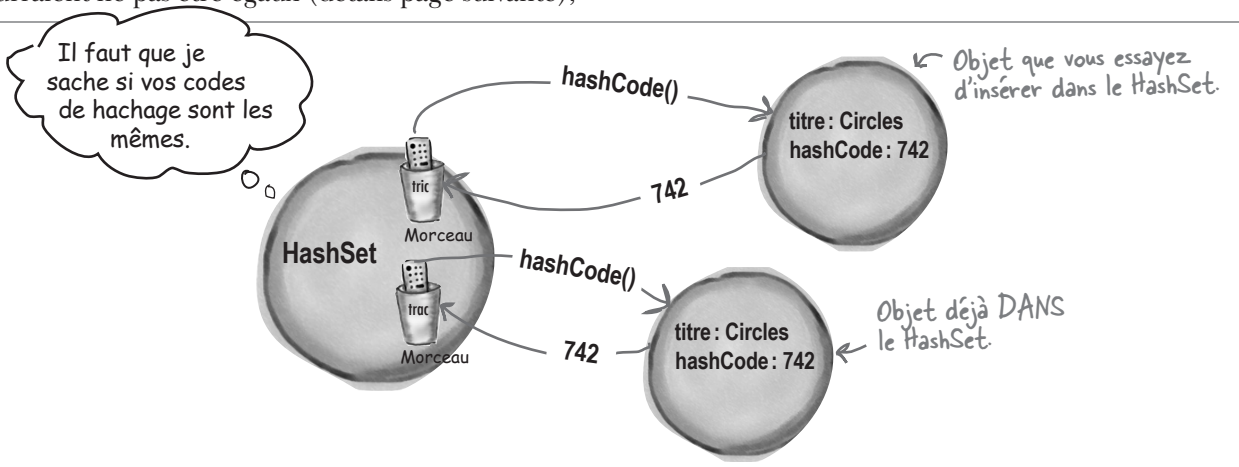
Autrement dit, si les codes de hachage sont différents, le HashSet suppose qu'il est impossible que les deux objets soient égaux !

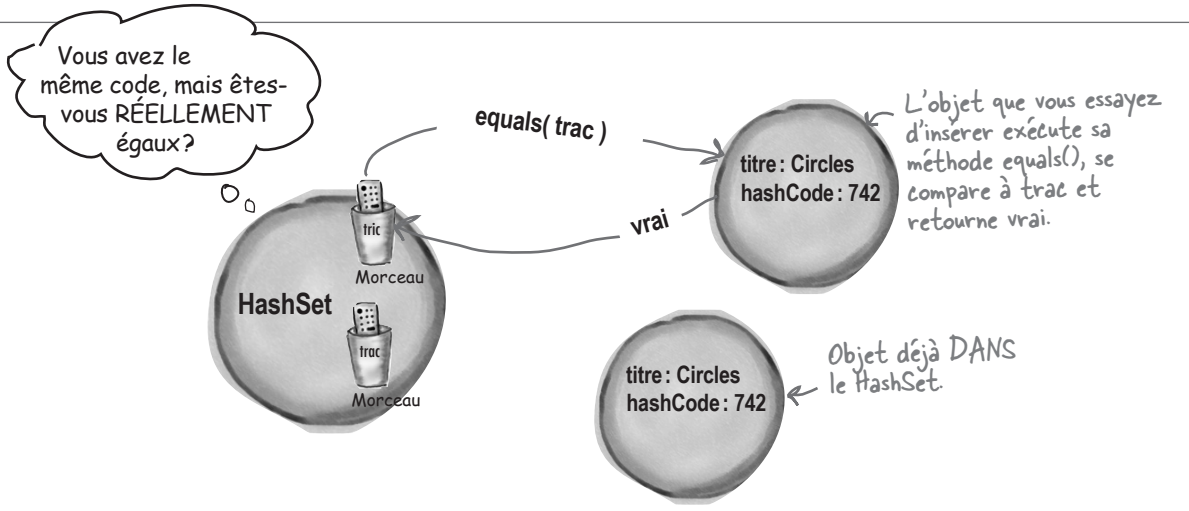
Vous devez donc redéfinir hashCode() pour être sûr que les deux objets ont la même valeur.

Mais comme deux objets ayant le même code pourraient ne pas être égaux (détails page suivante), si le HashSet trouve le même code pour deux objets — un que vous insérez et un déjà présent — il va appeler la méthode equals() de l'un des objets pour voir si ces deux objets sont réellement égaux.

Et s'ils sont égaux, le HashSet sait que l'objet que vous essayez d'insérer est un doublon de quelque chose qui est déjà dans la collection, et l'insertion n'a pas lieu.

Aucune exception n'est déclenchée, mais la méthode add() du HashSet retourne un booléen pour vous dire (si cela vous intéresse) si le nouvel objet a été ajouté. Si la méthode add() retourne faux, vous savez que ce nouvel objet était un doublon d'un objet déjà existant.





Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

La classe Morceau avec les méthodes hashCode() et equals() redéfinies

```
class Morceau implements Comparable<Morceau>{
   String titre;
   String artiste;
   String classement;
   String bpm;

  public boolean equals(Object unMorceau) {
     Morceau m = (Morceau) unMorceau;
     return getTitre().equals(m.getTitre());
  }

  public int hashCode() {
      return titre.hashCode();
  }

   public int compareTo(Morceau m) {
      return titre.compareTo(m.getTitre());
  }

   Morceau(String t, String a, String c, String b) {
      titre = t;
      artiste = a;
      classement = c;
      bpm = b;
   }

   public String getTitre() {
       return titre;
   }

   public String getArtiste() {
      return artiste;
   }

   public String getClassement()
      return classement;
   }

  public String getBpm() {
    return bpm;
  }

  public String toString() {
     return titre;
  }
}
```

Le HashSet (ou tout autre code appelant cette méthode) transmet un nouveau morceau.

SUPER nouvelle : ce titre est de type String, qui a une méthode equals() redéfinie. Il suffit donc de demander si un titre est égal au titre de l'autre morceau.

Pareil ici... La classe a une méthode hashCode() redéfinie. Vous pouvez donc juste retourner le résultat de l'appel de hashCode() sur le titre. Remarquez que hashCode() et equals() utilisent la MEME variable d'instance.

Maintenant, c'est bon ! Aucun doublon quand on affiche le HashSet. Mais nous n'avons pas appelé sort(), et quand nous avons placé l'ArrayList dans le HashSet, celui-ci n'a pas conservé l'ordre du tri.

Fichier Edition Fenêtre Aide RebootWindows

```
%java Jukebox6

[Pink Moon, Somersault, Shiva Moon, Circles,
Deep Channel, Passenger, Listen, Listen,
Listen, Circles]

[Circles, Circles, Deep Channel, Listen,
Listen, Listen, Passenger, Pink Moon, Shiva
Moon, Somersault]

[Pink Moon, Listen, Shiva Moon, Circles,
Deep Channel, Passenger, Somersault]
```


Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Lois Java pour hashCode() et equals()

La documentation de la classe Object édicte les règles que vous DEVEZ suivre :

- **Si deux objets sont égaux, ils DOIVENT avoir le même code de hachage.**
- **Si deux objets sont égaux, l'appel de equals() sur l'un des objets DOIT retourner vrai. Autrement dit, si (a.equals(b)) alors (b.equals(a)).**
- **Si deux objets ont le même code de hachage, ils ne sont PAS nécessairement égaux. Mais s'ils sont égaux, ils DOIVENT avoir le même code.**
- **Si vous redéfinissez equals(), vous DEVEZ redéfinir hashCode().**
- **Par défaut, hashCode() génère un entier unique pour chaque objet sur le tas. Si vous ne redéfinissez pas hashCode() dans une classe, deux objets de ce type ne pourront JAMAIS être considérés comme égaux.**
- **Par défaut, la méthode equals() effectue une comparaison ==. Autrement dit, elle teste si les deux références pointent sur un seul objet sur le tas. Si vous ne redéfinissez pas equals() dans une classe, deux objets ne pourront JAMAIS être considérés comme égaux, puisque les références à deux objets distincts contiendront toujours un motif de bits différent.**

a.equals(b)* implique que *a.hashCode() == b.hashCode()

Mais *a.hashCode() == b.hashCode()* n'implique PAS nécessairement *a.equals(b)*

il n'y a pas de

Questions stupides

Q : Comment se fait-il que deux codes de hachage puissent être identiques si les objets ne sont pas égaux ?

R : HashSet utilise des codes de hachage pour stocker les éléments d'une façon qui permet des accès beaucoup plus rapides. Si vous essayez de trouver un objet dans une ArrayList en lui donnant une copie de l'objet (et non la valeur de son indice), l'ArrayList doit commencer à chercher au début et examiner chaque élément de la liste pour voir s'il correspond. Mais un HashSet peut trouver un objet beaucoup plus vite, parce qu'il utilise le code de hachage comme une sorte d'étiquette apposée sur la « case » où il a stocké l'élément. Si vous dites « Je veux que tu me trouves l'objet qui est exactement comme celui-ci », le HashSet lit le code de la copie du Morceau que vous avez spécifié (par exemple, 742) et dit « Oh, je sais exactement où est stocké l'objet ayant le code 742 », puis il va directement à la case 742. Vous en apprendriez plus dans un cours d'informatique, mais cela suffit pour savoir utiliser un HashSet efficacement. En réalité, le développement de ce genre d'algorithme a fait l'objet de plus d'une thèse de doctorat, mais nous n'avons pas l'intention d'aller aussi loin dans ce livre. L'important est que ces codes peuvent être les mêmes sans nécessairement garantir que les objets sont égaux, parce que l'algorithme de hachage utilisé dans la méthode hashCode() pourrait retourner la même valeur pour plusieurs objets. Et oui, cela signifie que plusieurs objets atterriraient dans la même case du HashSet (parce que chaque case représente un seul code de hachage), mais ce n'est pas la fin du monde. Cela pourrait signifier que le HashSet est un petit peu moins efficace (ou qu'il contient un nombre gigantesque d'éléments), mais si le HashSet trouve plusieurs objets dans la même case, il appellera simplement la méthode equals() pour voir s'il y a une correspondance parfaite. Autrement dit, on peut parfois utiliser les codes de hachage pour restreindre la recherche, mais, pour trouver la correspondance exacte, le HashSet doit toujours prendre tous les objets dans la case qui porte le même code puis appeler equals() sur chacun pour voir si l'objet qu'il recherche s'y trouve.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Et si nous voulons que la collection reste triée, nous avons TreeSet

TreeSet est similaire à HashSet au sens où il empêche les doublons. Mais il conserve également la liste triée. Il fonctionne exactement comme la méthode sort() : si vous créez un TreeSet en utilisant le constructeur sans arguments, le TreeSet utilise la méthode compareTo() de chaque objet pour trier. Mais vous avez la possibilité de transmettre un Comparateur au constructeur du TreeSet, pour que celui-ci l'utilise à la place. TreeSet a un inconvénient : si vous n'avez pas besoin du tri, vous continuez à payer pour, avec une légère perte de performance. Mais vous constaterez probablement qu'elle est pratiquement indétectable dans la plupart des cas.

```
import java.util.*;
import java.io.*;
public class Jukebox8 {
   ArrayList<Morceau> listeMorceaux = new ArrayList<Morceau>();
   int val;

   public static void main(String[] args) {
      new Jukebox8().go();
   }

   public void go() {
      getMorceaux();
      System.out.println(listeMorceaux);
      Collections.sort(listeMorceaux);
      System.out.println(listeMorceaux);
      TreeSet<Morceau> hashMorceau = new TreeSet<Morceau>();
      hashMorceau.addAll(listeMorceaux);
      System.out.println(hashMorceau);
   }

   void getMorceaux() {
      try {
         File fichier = new File("ListeMorceauxPlus.txt");
         BufferedReader br = new BufferedReader(new FileReader(fichier));
         String ligne =  null;
         while ((ligne= br.readLine()) != null) {
              ajouterMorceau(ligne);
         }

      } catch(Exception ex) {
         ex.printStackTrace();
      }
   }

   void ajouterMorceau(String ligneAAnalyser) {
      String[] param = ligneAAnalyser.split("/");
      Morceau morceauSuivant = new Morceaux(param[0], param[1], param[2], param[3]);
      listeMorceaux.add(morceauSuivant);
   }
}
```

Nous instancions un TreeSet au lieu d'un HashSet. L'appel du constructeur sans arguments signifie que le TreeSet utilisera la méthode compareTo() de l'objet Morceau pour trier.

(Nous aurions pu transmettre un Comparateur.)

Nous ajoutons tous les morceaux du HashSet avec addAll(). (Nous aurions pu aussi les insérer individuellement en utilisant hashMorceau.add(), tout comme nous les avons ajoutés dans l'ArrayList.)

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Ce que vous DEVEZ savoir sur TreeSet...

TreeSet a l'air facile, mais vérifiez que vous comprenez vraiment ce qu'il vous faut pour l'utiliser. Nous trouvons cela si important que nous en avons fait un exercice pour que vous n'ayez pas besoin d'y réfléchir. Ne tournez PAS la page avant de l'avoir terminé. *Nous sommes sérieux.*

Regardez ce code. Lisez-le attentivement, puis répondez aux questions. (Note : ce code ne contient aucune erreur de syntaxe.)

```
import java.util.*;

public class TestTree {
   public static void main (String[] args) {
      new TestTree().go();
   }

   public void go() {
     Livre l1 = new Livre("La vie des chats");
     Livre l2 = new Livre("Remixez votre corps");
     Livre l3 = new Livre("Collections, mode d'emploi");

     TreeSet<Livre> tree = new TreeSet<Livre>();
     tree.add(l1);
     tree.add(l2);
     tree.add(l3);
     System.out.println(tree);
   }
}

class Livre {
   String titre;
   public Livre(String t) {
     titre = t;
   }
}
```

1). Quel est le résultat quand vous compilez ce code ?

2). S'il se compile, quel est le résultat quand vous exécutez la classe TestTree ?

3). S'il y a un problème (de compilation ou d'exécution) comment le résolvez-vous ?

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Les éléments d'un TreeSet DOIVENT être comparables

TreeSet ne peut pas lire dans la tête du programmeur pour savoir comment il faut trier les objets. Vous devez lui dire comment faire.

Pour utiliser un TreeSet, l'une de ces conditions doit être vraie :

- **Les éléments de la liste doivent être d'un type qui implémente *Comparable***

Comme la classe Livre de la page précédente n'implémentait pas Comparable, elle échouait à l'exécution. Réfléchissez : la seule raison d'être dans la vie du pauvre TreeSet est de conserver vos éléments triés, et, répétons-le, il n'a aucune idée de la façon de trier des objets Livre ! Le code se compile, parce que la méthode add() de TreeSet n'accepte pas un type Comparable, mais le type que vous avez spécifié quand vous avez créé le TreeSet. Autrement dit, si vous écrivez new TreeSet<Livre>(), la méthode add() se résume à add(Livre). Et rien ne dit que la classe Livre doive implémenter Comparable ! mais il échoue lors de l'exécution quand vous ajoutez le deuxième élément. C'est la première fois que l'ensemble essaie d'appeler la méthode compareTo() de l'un des objets... et n'y parvient pas.

```
class Livre implements Comparable {
   String titre;
   public Livre(String t) {
     titre = t;
   }
   public int compareTo(Object l) {
     Livre livre = (Livre) l;
     return (titre.compareTo(livre.titre));
   }
}
```

OU

- **Vous utilisez le constructeur surchargé du TreeSet qui accepte un *Comparateur***

TreeSet fonctionne beaucoup comme la méthode sort() : vous avez le choix d'utiliser la méthode compareTo() de l'élément, à supposer que son type implémente l'interface Comparable, OU d'utiliser un Comparateur sur mesure qui sait comment trier les éléments. Pour utiliser ce Comparateur, vous appelez le constructeur de TreeSet qui accepte un Comparateur.

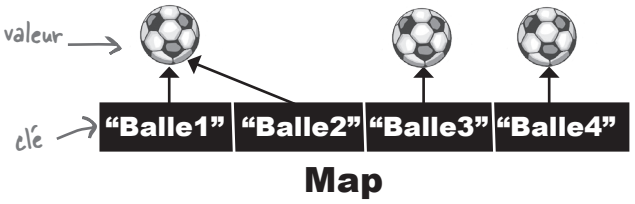
```
public class ComparateurLivres
             implements Comparator<Livre>{
  public int compare(Livre un, Livre deux) {
     return (un.titre.compareTo(deux.titre));
  }
}

class Test {
     public void go() {
     Livre l1 = new Livre("La vie des chats");
     Livre l2 = new Livre("Remixez votre corps");
     Livre l3 = new Livre("Collections, mode d'emploi");
     ComparateurLivres cLivres = new ComparateurLivres();
     TreeSet<Livre> tree = new TreeSet<Livre>(cLivres);
     tree.add(new Livre("La vie des chats");
     tree.add(new Livre("Collections, mode d'emploi");
     tree.add(new Livre("Remixez votre corps");
     System.out.println(tree);
  }
}
```


Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Après les classes List et Set, utilisons une Map

List et Set sont très intéressantes, mais il arrive qu'une Map soit la meilleure collection (pas Collection avec un grand C, souvenez-vous que Map fait partie des collections Java mais qu'elle n'implémente pas l'interface Collection).

Imaginez que vous vouliez une collection pour créer une liste de propriétés : vous fournissez un nom et elle retourne la valeur associée à ce nom. Même si les clés seront souvent des Strings, elles peuvent être un objet Java quelconque (ou, *via* l'autoboxing, un type primitif).



Chaque élément d'une Map se compose en réalité de DEUX objets : une *clé* et une *valeur*.
Les *valeurs* peuvent être dupliquées, mais PAS les *clés*.

Exemple de Map

```
import java.util.*;

public class TestMap {

   public static void main(String[] args) {

     HashMap<String, Integer> scores = new HashMap<String, Integer>();

     scores.put("Kathy", 42);
     scores.put("Bert", 343);
     scores.put("Skyler", 420);

      System.out.println(scores);
      System.out.println(scores.get("Bert"));
   }
}
```

HashMap nécessite DEUX paramètres de type — un pour la clé et un pour la valeur.

Au lieu de add(), utilisez put() qui bien sûr accepte maintenant deux arguments (clé, valeur).

La méthode get() accepte une clé et retourne une valeur (en l'occurrence un Integer).

Fichier Edition Fenêtre Aide OùSuisJe

```
%java TestMap

{Skyler=420, Bert=343, Kathy=42}
343
```

Quand vous affichez une Map, elle retourne clé=valeur entre accolades { } au lieu des crochets [ ] que vous voyez quand vous affichez une classe List ou Set.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Revenons aux génériques

Souvenez-vous : au début de ce chapitre, nous avons dit que les méthodes qui acceptaient en arguments des types génériques pouvaient être... *bizarres*. Nous voulons dire bizarre du point de vue du polymorphisme. Si vous commencez à avoir une impression d'étrangeté, continuez à lire : il va falloir quelques pages pour vous raconter toute l'histoire.

Commençons par un rappel sur la façon dont les arguments d'un tableau fonctionnent de façon polymorphe, puis nous verrons comment faire de même avec les listes génériques. Le code ci-dessous se compile et s'exécute sans erreurs.

Si l'argument d'une méthode est un tableau d'Animaux, elle acceptera également un tableau de n'importe quel sous-type d'Animal.

Autrement dit, si une méthode est déclarée :

void truc(Animal[] a) { }

En supposant que Chien dérive d'Animal, vous pouvez appeler aussi bien :

truc(unTableauDAnimaux);
truc(unTableauDeChiens;

Fonctionnement des *tableaux* ordinaires :

```
import java.util.*;

public class TestGeneriques1 {
   public static void main(String[] args) {
      new TestGeneriques1().go();
   }

   public void go() {
      Animal[] animaux = {new Chien(), new Chat(), new Chien()};
      Chien[] chiens = {new Chien(), new Chien(), new Chien()};
      accepterAnimaux(animaux);
      accepterAnimaux(chiens);
   }

  public void accepterAnimaux(Animal[] animaux) {
      for(Animal a: animaux) {
        a.manger();
      }
   }
}
```

Déclarer et créer un tableau de type Animal, qui contient à la fois des chiens et des chats.

Déclarer et créer un tableau de type Chien, qui ne contient que des chiens (le compilateur ne vous laissera pas y mettre un chat).

Appeler accepterAnimaux(), en utilisant les deux types comme arguments...

Soulignons un point crucial : la méthode accepterAnimaux() accepte un Animal[] ou un Chien[], puisque Chien EST-UN Animal. Polymorphisme en action.

Souvenez-vous : on ne peut appeler QUE que les méthodes déclarées dans le type Animal, puisque le paramètre animaux est de type tableau d'Animal, et nous n'avons pas sous-typé. (Et pourquoi sous-typer ? Ce tableau pourrait bien contenir des Chiens et des Chats.)

```
abstract class Animal {
   void manger() {
     System.out.println("l'animal mange");
   }
}
class Chien extends Animal {
   void aboyer() { }
}
class Chat extends Animal {
   void miauler() { }
}
```

La hiérarchie de classes simplifiée d'Animal

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Utiliser des arguments polymorphes et des génériques

Nous avons vu le fonctionnement des tableaux, mais cela va-t-il fonctionner de la même manière quand nous allons passer d'un tableau à une ArrayList ? Cela semble raisonnable, n'est-ce pas ?

Essayons d'abord seulement avec l'ArrayList Animal. Nous avons juste un peu modifié la méthode go() :

Ne transmettre que ArrayList<Animal>

Simple transformation de Animal[] en ArrayList<Animal>.

```
public void go() {
    ArrayList<Animal> animaux = new ArrayList<Animal>();
    animaux.add(new Chien());
    animaux.add(new Chat());
    animaux.add(new Chien());

    accepterAnimaux(animaux);
  }
```

Nous devons les ajouter un par un, puisqu'il n'y a pas de syntaxe abrégée comme pour la création des tableaux.

C'est le même code, sauf que la variable « animaux » renvoie à une ArrayList et non à un tableau.

```
public void accepterAnimaux(ArrayList<Animal> animaux) {
    for(Animal a: animaux) {
    a.manger();
   }
  }
```

La même méthode accepte maintenant une ArrayList au lieu d'un tableau, mais tout le reste est identique. Souvenez-vous que la syntaxe de la boucle for fonctionne aussi bien pour les tableaux que pour les collections.

Le code se compile et s'exécute parfaitement

Fichier Edition Fenêtre Aide RonRon

```
%java TestGeneriques2

l'animal mange
l'animal mange
l'animal mange
```


Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Mais cela va-t-il fonctionner avec ArrayList<Chien>?

En raison du polymorphisme, le compilateur nous permet de passer un tableau de Chiens à une méthode ayant un argument acceptant un tableau Animal. Pas de problème. On peut aussi transmettre une ArrayList<Animal> à une méthode dont l'argument est une ArrayList<Animal>. La grande question est donc : l'argument ArrayList<Animal> acceptera-t-il une ArrayList<Chien>? Si cela fonctionne avec les tableaux, pourquoi pas ici ?

Ne transmettre que ArrayList<Chien>

```
public void go() {
   ArrayList<Animal> animaux = new ArrayList<Animal>();
   animaux.add(new Chien());
   animaux.add(new Chat());
   animaux.add(new Chien());
   accepterAnimaux(animaux);

   ArrayList<Chien> chiens = new ArrayList<Chien>();
   chiens.add(new Chien());
   chiens.add(new Chien());
   accepterAnimaux(chiens);
}
```

Nous savons que cette ligne fonctionnait

Créons une ArrayList de Chiens et plaçons-y deux chiens.

Ceci va-t-il fonctionner maintenant que nous avons transformé le tableau en ArrayList?

```
public void accepterAnimaux(ArrayList<Animal> animaux) {
     for(Animal a: animaux) {
     a.manger();
    }
 }
```

Quand nous compilons :

```
Fichier Edition Fenêtre Aide LesChatsSontPlusMalins
%java TestGeneriques3

TestGeneriques3.java:21: accepterAnimaux(java.util.
ArrayList<Animal>) in TestGeneriques3 cannot be applied
to (java.util.ArrayList<Chien>)
     accepterAnimaux(chiens);
     ^
1 error
```

Cela semblait si bien... mais rien ne va plus...

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Et je suis censée être d'accord ? Cela bousille totalement ma simulation où le programme vétérinaire prend une liste de n'importe quel type d'animal, pour qu'un élevage de chiens puisse envoyer une liste de chiens et un élevage de chats une liste de chats... Et maintenant vous me dites que c'est impossible si je remplace les tableaux par des collections ?

Que se passerait-il si l'on pouvait...

Imaginez que le compilateur accepte. Il vous permet donc de transmettre une ArrayList<Chien> à une méthode déclarée :

```
public void accepterAnimaux(ArrayList<Animal> animaux) {
   for(Animal a: animaux) {
      a.manger();
   }
}
```

Il n'y a rien de dangereux *a priori* dans cette méthode, d'accord ? Après tout, tout l'intérêt du polymorphisme est que tout ce qu'un Animal peut faire (en l'occurrence la méthode manger()), un Chien peut le faire aussi. Alors, où est le problème si l'on appelle la méthode manger() sur chaque référence de Chien ?

Rien. Absolument rien.

Ce code ne pose aucun problème. Mais imaginez *celui-ci* :

```
public void accepterAnimaux(ArrayList<Animal> animaux) {
   animaux.add(new Chat());
}
```

Ouille !! Nous venons de fourrer un Chat dans une ArrayList réservée aux Chiens.

Voilà le problème. Il n'y a sans aucun doute rien de mal à ajouter un Chat dans une ArrayList<Animal>, et c'est tout l'intérêt d'une ArrayList d'un supertype comme Animal : vous pouvez placer tous les types d'animaux dans une seule ArrayList.

Mais si vous avez transmis une ArrayList de Chiens — qui n'est censée contenir QUE des Chiens — à cette méthode qui accepte une ArrayList<Animal>, vous vous retrouvez soudain avec un Chat dans la liste de Chiens. Le compilateur sait que s'il vous laissait passer une ArrayList de Chiens à la méthode, quelqu'un pourrait au moment de l'exécution ajouter un Chat à votre liste de Chiens. Et le compilateur ne vous laissera pas prendre le risque.

Si vous déclarez qu'une méthode accepte une ArrayList<Animal>, elle n'accepte QUE ArrayList<Animal>, mais ni ArrayList<Chien> ni ArrayList<Chat>.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Attendez une minute... Si c'est la raison pour laquelle on ne nous laisse pas passer une ArrayList de chiens à une méthode qui accepte une ArrayList d'animaux — pour vous empêcher de mettre un chat dans ce qui est en fait une liste de chiens, pourquoi peut-on le faire avec les tableaux? N'a-t-on pas le même problème avec les tableaux? Ne peut-on pas ajouter un objet Chat à un tableau Chien[]?

Les types des tableaux sont vérifiés lors de l'exécution, mais les types des collections le sont à la compilation.

Disons que vous ajoutez un Chat a un tableau déclaré Chien[] (un tableau qui a été transmis à un argument de méthode déclaré comme Animal[], ce qui est une affectation parfaitement légale pour un tableau).

```
public void go() {
   Chien[] chiens = {new Chien(), new Chien(), new Chien()};
   accepterAnimaux(chiens);
}

public void accepterAnimaux(Animal[] animaux) {
   animaux[0] = new Chat();
}
```

Nous plaçons un Chat dans un tableau de Chiens. Le compilateur l'autorise : comme il sait que vous auriez pu transmettre un tableau de type Chat ou de type Animal à la méthode, il pense que c'est O.K.

Le code se compile, mais quand nous l'exécutons :

Fichier Edition Fenêtre Aide LesChatsSontPlusMalins

```
%java TestGeneriques1
Exception in thread "main" java.lang.ArrayStoreException:
Chat
        at TestGeneriques1.accepterAnimaux(TestGeneriques1.
java:16)
        at TestGeneriques1.go(TestGeneriques1.java:12)
        at TestGeneriques1.main(TestGeneriques1.java:5)
```

Ouaouh ! Au moins la JVM l'a arrêté.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Les jokers à la rescousse

Cela semble inhabituel, mais il existe un moyen de créer un argument de méthode qui accepte une ArrayList de n'importe quel sous-type d'Animal. Le plus simple consiste à utiliser un joker, ajouté à Java explicitement pour cette raison.

```
public void accepterAnimaux(ArrayList<? extends Animal> animaux) {
   for(Animal a: animaux) {
      a.manger();
   }
}
```

Souvenez-vous : le mot-clé « extends » signifie étend OU implémente en fonction du type. Si vous voulez une ArrayList de types qui implémentent l'interface AnimalDomestique, vous la déclarez :

ArrayList<? extends AnimalDomestique>

« Et alors ? » dites-vous. « Où est la différence ? N'avons-nous pas le même problème ? La méthode ci-dessus ne fait rien de dangereux — on appelle une méthode que tout sous-type d'Animal est garantie d'avoir — mais quelqu'un ne pourrait-il pas ajouter un Chat à la liste d'*animaux*, même si c'est en réalité une ArrayList<Chien>? Et puisqu'il n'y a pas de vérification à l'exécution, en quoi cela diffère-t-il de la déclarer sans le joker ? »

Et vous auriez raison de vous étonner. La réponse est NON. Quand vous employez le joker <?> dans votre déclaration, le compilateur ne vous laisse pas insérer dans la liste !

Quand vous utilisez un joker dans l'argument de votre méthode, le compilateur vous EMPÊCHE de faire quoi que ce soit qui puisse endommager la liste référencée par le paramètre.

Vous pouvez toujours invoquer des méthodes sur les éléments de la liste, mais vous ne pouvez pas ajouter d'éléments.

Autrement dit, vous pouvez faire ce que vous voulez avec les éléments de la liste, mais vous ne pouvez pas en insérer. L'exécution est donc sûre, parce que le compilateur ne vous laisse rien faire qui produirait une horreur à l'exécution.

Ceci est donc correct dans accepterAnimaux():

```
for(Animal a: animaux) {
   a.manger();
}
```

Mais CELA ne se compilera pas :

```
animaux.add(new Chat());
```


Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Autre syntaxe pour faire la même chose

Vous vous souvenez probablement que, quand nous avons étudié la méthode sort(), elle utilisait un type générique, mais avec un format inhabituel où le paramètre de type était déclaré avant le type de retour. C'est une autre façon de déclarer le paramètre de type, mais les résultats sont identiques.

Ceci :

```
public <T extends Animal> void accepter(ArrayList<T> liste)
```

Fait la même chose que :

```
public void accepter(ArrayList<? extends Animal> liste)
```

il n'y a pas de
Questions stupides

Q : Si les deux font la même chose, pourquoi préférer une syntaxe à l'autre ?

R : Cela dépend. Voulez-vous utiliser "T" ailleurs ou non ? Par exemple, que se passe-t-il si vous voulez que la méthode ait deux arguments — dont les deux sont des listes d'un type qui dérive d'Animal ? Dans ce cas, il est plus efficace de ne déclarer le paramètre de type qu'une fois :

```
public <T extends Animal> void accepter(ArrayList<T> un, ArrayList<T> deux)
```

Au lieu de taper :

```
public void accepter(ArrayList<? extends Animal> un,
                     ArrayList<? extends Animal> deux)
```


Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Vous ÊTES le compilateur, avancé

Votre tâche consiste à jouer le rôle du compilateur et à déterminer lesquelles de ces instructions se compileraient. Mais comme une partie de ce code n'a pas été abordée dans ce chapitre, vous devez trouver les réponses d'après ce que vous AVEZ appris, en appliquant les « règles » à ces nouvelles situations. Vous devrez peut-être deviner dans certains cas, mais l'objectif est de donner une réponse raisonnable en vous appuyant sur ce que vous savez déjà.

(Note : supposez que ce code se trouve dans une classe et une méthode valides.)

Compile ?

- ☐ `ArrayList<Chien> chiens1 = new ArrayList<Animal>();`
- ☐ `ArrayList<Animal> animaux1 = new ArrayList<Chien>();`
- ☐ `List<Animal> liste = new ArrayList<Animal>();`
- ☐ `ArrayList<Chien> chiens = new ArrayList<Chien>();`
- ☐ `ArrayList<Animal> animaux = chiens;`
- ☐ `List<Chien> listeDeChiens = chiens;`
- ☐ `ArrayList<Object> objets = new ArrayList<Object>();`
- ☐ `List<Object> listeDObjets = objets;`
- ☐ `ArrayList<Object> objs = new ArrayList<Chien>();`

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Rétro-ingénierie : Solution

```
import java.util.*;

public class TriMontagnes {

  LinkedList<Montagne> mtgn = new LinkedList<Montagne>();

  class ComparateurNoms implements Comparator <Montagne> {
    public int compare(Montagne un, Montagne deux) {
      return un.nom.compareTo(deux.nom);
    }
  }
  class ComparateurHauteurs implements Comparator <Montagne> {
    public int compare(Montagne un, Montagne deux) {
      return (deux.hauteur - un.hauteur);
    }
  }
  public static void main(String [] args) {
    new TriMontagnes().go();
  }
  public void go() {
    mtgn.add(new Montagne("Cervin", 4482));
    mtgn.add(new Montagne("MontBlanc", 4807));
    mtgn.add(new Montagne("JungFrau", 4168));
    mtgn.add(new Montagne("Pelvoux", 3955));

    System.out.println("dans l'ordre d'insertion:\n" + mtgn);
    ComparateurNoms cn = new ComparateurNoms();

    Collections.sort(mtgn, cn);
    System.out.println("par nom:\n" + mtgn);
    ComparateurHauteurs ch = new ComparateurHauteurs();

    Collections.sort(mtgn, ch);
    System.out.println("par hauteur:\n" + mtgn);
  }
}

class Montagne {
  String nom;
  int hauteur;

  Montagne(String n, int h) {
    nom = n;
    hauteur = h;
  }
  public String toString( ) {
    return nom + " " + hauteur;
  }
}
```

Aviez-vous remarqué que la liste des hauteurs était dans l'ordre DÉCROISSANT ?)

Résultat :

```
Fichier Edition Fenêtre Aide CeluiCiPourBob
%java TriMontagnes
dans l'ordre d'insertion:
[Cervin 4482, MontBlanc 4807, JungFrau 4168, Pelvoux 3955]
par nom:
[Cervin 4482, JungFrau 4168, MontBlanc 4807 Pelvoux 3955]
par hauteur:
[MontBlanc 4807, Cervin 4482, JungFrau 4168 Pelvoux 3955]
```


Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Solution de l'exercice

Réponses possibles :

Comparator,

Comparable,

compareTo( ),

compare( ),

oui,

non

Étant donné l'instruction compilable suivante :

```
Collections.sort(monArrayList);
```

1. Que doit implémenter la classe des objets stockés dans `monArrayList` ? **Comparable**
2. Quelle méthode la classe des objets stockés dans `monArrayList` doit-elle implémenter ? **compareTo( )**
3. La classe des objets stockés dans mon `monArrayList` peut-elle implémenter à la fois `Comparator` ET `Comparable` ? **oui**

Étant donné l'instruction compilable suivante :

```
Collections.sort(monArrayList, maComparaison);
```

4. La classe des objets stockés dans `monArrayList` peut-elle implémenter `Comparable` ? **oui**
5. La classe des objets stockés dans `monArrayList` peut-elle implémenter `Comparator` ? **oui**
6. La classe des objets stockés dans `monArrayList` doit-elle implémenter `Comparable` ? **non**
7. La classe des objets stockés dans `monArrayList` doit-elle implémenter `Comparator` ? **non**
8. Que doit implémenter la classe de l'objet `maComparaison` ? **Comparator**
9. Quelle méthode la classe de l'objet `maComparaison` doit-elle implémenter ? **compare( )**

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Vous ÊTES le compilateur : Solution

Compile ?

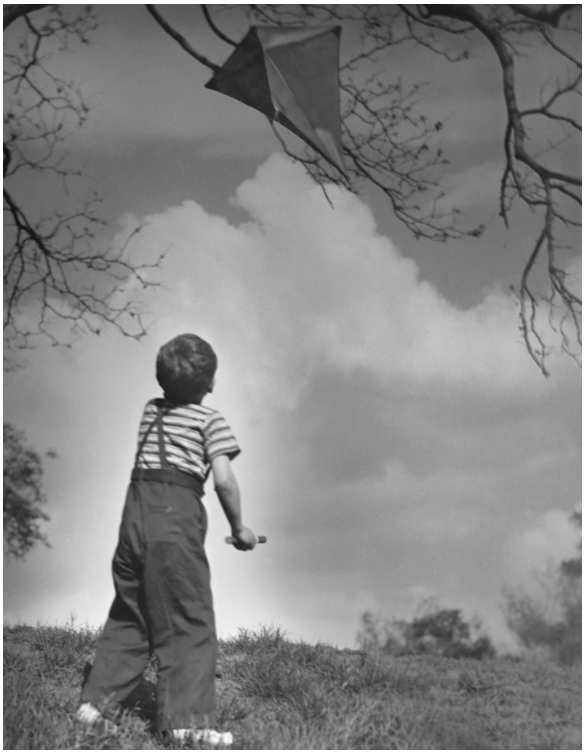
- [ ] `ArrayList<Chien> chiens = new ArrayList<Animal>();`
- [ ] `ArrayList<Animal> animaux1 = new ArrayList<Chien>();`
- [x] `List<Animal> liste = new ArrayList<Animal>();`
- [ ] `ArrayList<Chien> chiens = new ArrayList<Chien>();`
- [ ] `ArrayList<Animal> animaux = chiens;`
- [x] `List<Chien> listeDeChiens = chiens;`
- [x] `ArrayList<Object> objets = new ArrayList<Object>();`
- [x] `List<Object> ListeDObjets = objets;`
- [ ] `ArrayList<Object> objs = new ArrayList<Chien>();`

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

17 packages, archives jar et déploiement

Déployez votre code



Il est temps de lâcher prise. Vous avez écrit votre code. Vous avez testé votre code. Vous avez peaufiné votre code. Vous avez dit à tout le monde que ce serait génial de 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 ? *Comment* allez-vous le communiquer aux utilisateurs ? Et qu'allez-vous *exactement* leur donner ? Et si vous ne savez même pas qui ils sont ? Dans ces deux derniers chapitres, nous allons voir comment organiser, packager et déployer votre code Java. Nous parlerons des options de déploiement local, semi-local et à distance, notamment des jars exécutables, de Java Web Start, de RMI et des servlets. Dans ce chapitre, nous nous consacrerons majoritairement à l'organisation et au packaging — tout ce que vous devez savoir indépendamment de votre choix final. Dans le dernier chapitre, nous terminerons par l'un des aspects les plus cools de Java. Décontractez-vous. Ce n'est pas qu'un au revoir. Il y a toujours la maintenance…

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

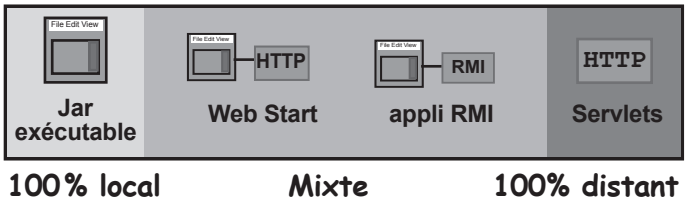
Déployer votre application

Qu'est-ce exactement qu'une application Java ? Quand vous en avez terminé avec le développement, qu'allez-vous livrer ? Il est fort possible que vos utilisateurs finaux ne possèdent pas un système identique au vôtre. Plus important encore, ils n'ont pas votre application. C'est donc le moment de mettre en forme votre programme pour pouvoir le déployer dans le monde extérieur. Dans ce chapitre, nous étudierons les déploiements locaux, notamment les jars exécutables, et la technologie mi-locale mi-distante nommée Java Web Start. Le suivant sera consacré aux solutions à distance, telles que RMI et les servlets.

Un programme Java est un ensemble de classes. C'est le résultat de votre développement.

La vraie question est : que faire de ces classes quand vous aurez terminé ?

Options de déploiement



1. **Local**
   Toute l'application s'exécute sur la machine de l'utilisateur final. C'est un programme autonome, probablement doté d'une interface graphique, déployé sous forme de JAR exécutable (nous verrons JAR dans quelques pages).

2. **Mixte**
   L'application est distribuée. La portion client s'exécute sur le système local qui se connecte à un serveur sur lequel s'exécute le reste du programme.

3. **Distant**
   Toute l'application Java s'exécute sur le serveur. Le client accède au système par des moyens non-Java, probablement un navigateur web.

Mais avant de rentrer vraiment dans les questions de déploiement, revenons en arrière et voyons ce qui se passe lorsque vous avez fini de programmer votre application et que vous voulez simplement extraire les fichiers classe pour les communiquer à l'utilisateur. Qu'y a-t-il réellement dans ce répertoire de travail ?

Gym du cerveau

Quels avantages et quels inconvénients y a-t-il à livrer votre programme Java sous forme d'application locale autonome s'exécutant sur l'ordinateur de l'utilisateur final ?

Quels avantages et quels inconvénients y a-t-il à livrer votre programme Java sous forme de système basé sur le Web, l'utilisateur interagissant avec un navigateur et le code s'exécutant sous forme de servlets sur le serveur ?

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Imaginez ce scénario...

Paul se met joyeusement au travail sur les dernières finitions de son nouveau super-programme Java. Après des semaines passées en mode « encore une petite compilation et c'est fini », cette fois il a réellement terminé. L'application comporte une interface graphique relativement complexe, mais comme le plus gros est constitué de code Swing, il n'a écrit lui-même que neuf classes.

Le moment de livrer le programme au client est enfin arrivé. Il s'imagine qu'il va lui suffire de copier les neuf fichiers de classes, puisque l'API Java est déjà installée sur la machine du client. Il commence par lister le répertoire dans lequel tous ses fichiers se trouvent...

Ouaouh ! Quelque chose d'étrange s'est produit. Au lieu de 18 fichiers (neuf fichiers source et neuf fichiers compilés), il en voit 31, dont la plupart portent des noms très bizarres comme :

```
Compte$FileListener.class
Graphique$SaveListener.class
```

et ainsi de suite. Il a complètement oublié que le compilateur devait générer des fichiers distincts pour toutes les classes internes des auditeurs d'événements qu'il a créés, et c'est ce que sont tous ces fichiers aux noms exotiques.

Maintenant, il doit extraire soigneusement tous les fichiers de classes dont il a besoin. S'il en oublie ne serait-ce qu'un, son programme ne tournera pas. Mais l'opération s'annonce délicate : il ne veut pas envoyer accidentellement un fichier source à son client, or tout est dans le même répertoire et c'est un vrai capharnaüm.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Séparez le code source des fichiers classe

Un unique répertoire rempli de code source et de fichiers classe en vrac est un vrai gâchis. Paul aurait dû organiser ses fichiers dès le départ et séparer le code source du code compilé. Autrement dit veiller à ce que ses fichiers classe n'atterrissent pas dans le même répertoire que son code source.

La solution consiste à combiner une structure de répertoires et l'option `-d` du compilateur.

Il existe des dizaines de façons d'organiser ses fichiers, et votre entreprise a peut-être une procédure spécifique que vous devrez appliquer. Nous vous recommandons toutefois un schéma d'organisation qui est pratiquement devenu un standard.

Dans ce schéma, vous créez un répertoire «projet» dans lequel vous créez un sous-répertoire nommé **source** et un sous-répertoire nommé **classes**. Vous commencez par sauvegarder votre code source (les fichiers .java) dans le répertoire **source**. L'astuce consiste ensuite à compiler votre code de telle sorte que le résultat (les fichiers .class) se retrouvent dans le répertoire classes.

Et le compilateur dispose pour ce faire d'une option sympathique : l'option **`-d`**.

Compiler avec l'option `-d` (directory)

```
%cd MonProjet/source
%javac  -d ../classes  MonAppli.java
```

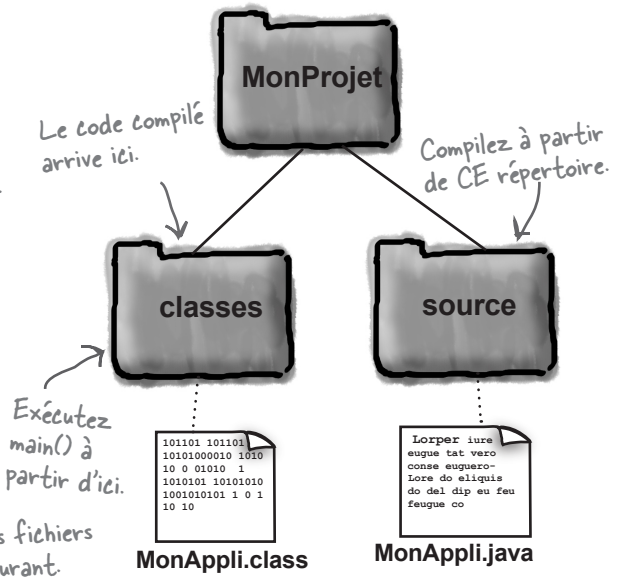
Cette option dit au compilateur de placer le code compilé (les fichiers .class) dans le répertoire "classes", au même niveau de l'arborescence que le répertoire courant en passant par le répertoire supérieur.

Le dernier élément est toujours le nom du fichier .java à compiler.

Avec l'option **`-d`**, *vous décidez dans quel répertoire placer le code compilé* au lieu d'accepter le comportement par défaut dans lequel les fichiers .class se retrouvent dans le même répertoire que le code source. Pour compiler tous les fichiers .java du répertoire source, tapez :

```
%javac  -d ../classes  *.java
```

*.java compile TOUS les fichiers source du répertoire courant.



Faire tourner le code

```
%cd MonProjet/classes
%java Mini
```

Exécute le programme depuis le répertoire "classes".

(Note : dans tout ce chapitre, nous partons du principe que le répertoire courant (le répertoire ".") est dans votre classpath. Si vous avez défini explicitement cette variable d'environnement, vérifiez qu'elle contient ".")

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Placez votre code Java dans un fichier JAR

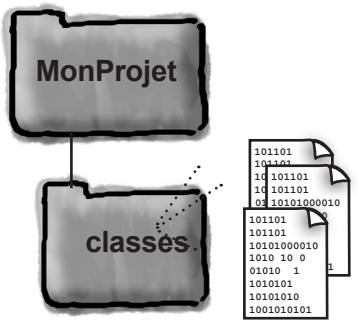
JAR signifie **J**ava **AR**chive. Le format d'un fichier JAR est basé sur pkzip. Il vous permet de grouper toutes vos classes : au lieu de présenter à votre client 28 fichiers classe, vous ne lui livrez qu'un seul fichier JAR. Si vous connaissez la commande tar sous UNIX, vous reconnaîtrez les commandes de l'outil jar. (Note : quand nous écrivons JAR en capitales, nous parlons du fichier d'archive. Quand nous employons des minuscules, nous désignons l'*outil* nommé *jar* qui sert à créer des fichiers JAR.)

Question : Que fait le client de ce JAR ? Où trouve-t-il le programme ?

Réponse : vous rendez le JAR ***exécutable***.

Un JAR exécutable signifie que l'utilisateur n'a pas besoin d'extraire les fichiers classe avant d'exécuter le programme. Il peut lancer l'application en laissant les dits fichiers dans le JAR. L'astuce consiste à placer dans le JAR un fichier ***manifeste*** qui contiendra les informations sur les autres fichiers. Pour qu'un JAR soit exécutable, le manifeste doit indiquer à la JVM *dans quelle classe se trouve la méthode main() !*

Créer un JAR exécutable



1 Vérifiez que tous vos fichiers classe sont dans le répertoire classes

Nous verrons que c'est un peu plus complexe dans quelques pages. Pour l'instant, contentez-vous de placer tout le code compilé dans le répertoire nommé « classes ».

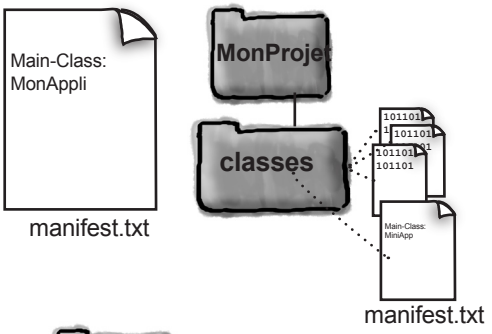
2 Créez un fichier manifest.txt qui déclare dans quelle classe se trouve main()

Créez un fichier texte nommé manifest.txt qui ne contient qu'une ligne :

```
Main-Class: MonAppli
```

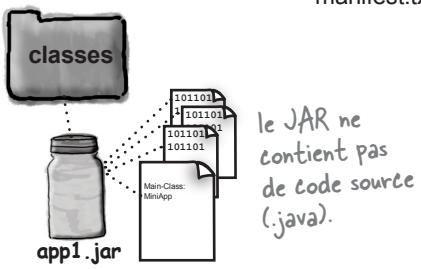
← Pas d'extension .class

Appuyez sur la touche Entrée après avoir tapé la ligne Main-Class ou votre manifeste risque de ne pas fonctionner correctement. Placez le manifeste dans le répertoire « classes ».

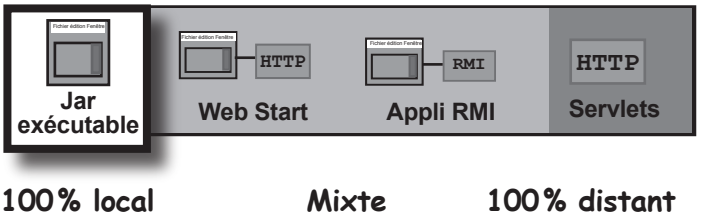


3 Exécutez l'outil jar pour créer un fichier JAR qui contient toutes les classes plus le manifeste.

```
%cd MiniProject/classes
%jar -cvmf manifest.txt app1.jar *.class
OR
%jar -cvmf manifest.txt app1.jar MonAppli.class
```



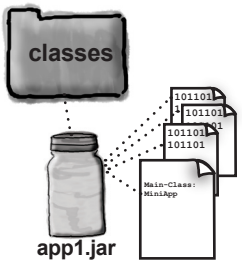
Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>



Près de 100% des applications Java locales sont déployées sous forme de JAR exécutable.

Exécuter le JAR

Java (la JVM) est capable de charger une classe depuis un JAR et d'appeler la méthode main() de cette classe. En fait, l'application entière peut rester dans le JAR. Une fois le coup d'envoi donné (que la méthode main() commence à s'exécuter), la JVM se moque de l'endroit d'où viennent les classes du moment qu'elle peut les trouver. Et l'un des endroits dans lesquels la JVM recherche un chemin d'accès aux fichiers JAR est la variable CLASSPATH. Si elle trouve un JAR, la JVM regarde dedans quand elle a besoin de charger une classe.




```
%cd MonProjet/classes

%java -jar app1.jar
```

Puisque la JVM doit « voir » le JAR, son chemin doit être dans votre CLASSPATH. La meilleure façon de rendre votre JAR visible consiste à le placer dans votre répertoire courant.

L'option –jar dit à la JVM qu'il s'agit d'un JAR et non d'une classe.

La JVM cherche dans ce JAR un manifeste avec une entrée Main–Class. Si elle n'en trouve pas, Vous obtenez une erreur d'exécution.

Selon la configuration de votre système d'exploitation, il se peut même que vous puissiez lancer un fichier JAR rien qu'en cliquant dessus. C'est possible sous la plupart des versions de Windows et sous Mac OS X. En général, vous pouvez également sélectionner le JAR et choisir la commande « Ouvrir avec... » (ou son équivalent dans votre système d'exploitation).

il n'y a pas de

Questions stupides

Q : Pourquoi ne pas compacter un répertoire entier ?

R : La JVM inspecte le JAR et s'attend à trouver ce dont elle a besoin là et pas ailleurs. Elle ne va pas se mettre à fouiller dans les répertoires, sauf si la classe fait partie d'un package. Et même alors, la JVM ne regarde que dans les répertoires qui correspondent à l'instruction package.

Q : Vous pouvez répéter ?

R : Vous ne pouvez pas mettre vos fichiers classe dans un répertoire arbitraire et en faire un JAR. Mais si vos classes appartiennent à des packages, vous pouvez faire un JAR qui contient toute la structure des répertoires. En fait, vous *devez* le faire. Décontractez-vous, c'est le sujet de la page suivante.

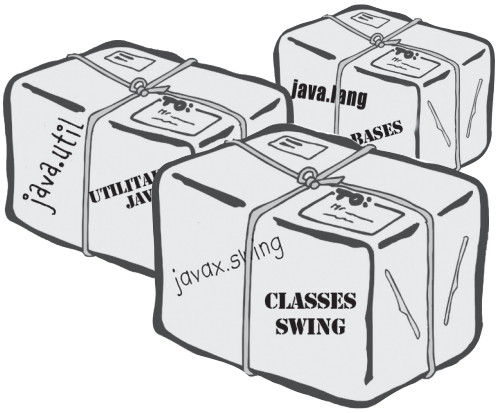
Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Placez vos classes dans des packages!

Vous avez donc écrit des fichiers classe bien réutilisables et vous les avez publiés dans votre bibliothèque de développement interne à l'intention des autres programmeurs. Tandis que vous vous délectez du plaisir d'avoir livré l'un des meilleurs exemples (à votre humble opinion) de programme OO jamais conçu, vous recevez un coup de fil complètement paniqué. Deux de vos classes portent le même nom que les classes que Fred vient d'insérer dans la bibliothèque. Et l'enfer est en train de se déchaîner, parce que les ambiguïtés et les collisions de nommage sont en train de bousiller tout le développement.

Et tout cela parce que vous n'avez pas utilisé de packages! Enfin, vous avez bien utilisé des packages, au sens où vous avez utilisé des classes de l'API qui sont bien sûr dans des packages. Mais vous n'avez pas mis vos propres classes dans des packages, et, dans le monde réel, c'est une grossière erreur.

Nous allons juste un peu modifier la structure que nous avons vue dans les pages précédentes, placer les classes dans un package et créer un JAR avec ce package. Faites très attention, car les détails sont subtils et la plus minuscule déviation peut empêcher votre code de se compiler et/ou de s'exécuter.



Structure des packages de l'API Java pour :

java.text.NumberFormat

java.util.ArrayList

java.awt.FlowLayout

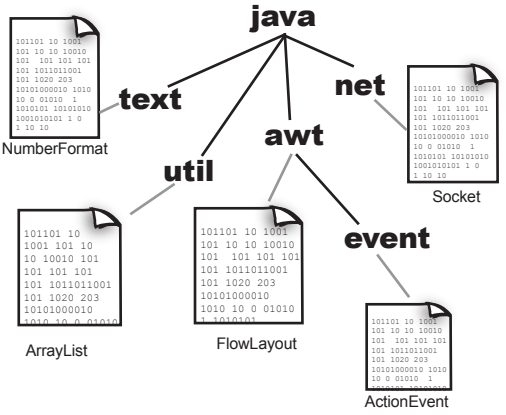
java.awt.event.ActionEvent

java.net.Socket

Les packages empêchent les conflits de nommage

Même si les packages ne servent pas seulement à empêcher les conflits de nommage, c'est l'une de leurs caractéristiques clé. Imaginez que vous écriviez une classe nommée Client, une classe nommée Compte et une classe nommée Panier. Il est fort probable que la moitié des développeurs travaillant dans le commerce électronique aient déjà écrit des classes homonymes. Dans un monde orienté objet, c'est une situation dangereuse. Si une partie de l'approche OO consiste à écrire des composants réutilisables, les développeurs doivent pouvoir assembler des composants provenant de différentes sources pour construire quelque chose de nouveau. Vos composants doivent être capable de «jouer avec les autres», y compris ceux que vous n'avez pas écrits ou dont vous ne connaissez même pas l'existence.

Souvenez-vous du chapitre 6. Nous avons vu qu'un nom de package était similaire à un nom de classe complet, qu'on nomme techniquement *nom pleinement qualifié*. La classe ArrayList est en réalité ***java.util.ArrayList***, JButton est ***javax.swing.JButton*** et Socket est ***java.net.Socket***. Remarquez que deux de ces classes, ArrayList et Socket, ont toutes deux *java* pour «prénom». Autrement dit, la première partie de leur nom complet est «java». Pensez qu'une structure de packages doit être hiérarchisée et organisez vos classes en conséquence.



Cette figure ne vous rappelle-t-elle pas quelque chose? Ne ressemble-t-elle pas furieusement à une arborescence de répertoires?

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Les packages peuvent empêcher les conflits de nommage, mais seulement si vous choisissez un nom de package garanti unique. La meilleure façon de procéder consiste à employer comme préfixe votre nom de domaine inversé.

Empêcher les conflits de noms de packages

Placer votre classe dans un package réduit les risques de conflits de nommage avec d'autres classes, mais qu'est-ce qui empêche deux programmeurs de donner le même nom à leurs *packages* ? Autrement dit, qu'est-ce qui les empêche d'avoir tous les deux une classe nommée Compte et de la placer dans une classe nommé shopping.clients ? Dans ce cas, les deux classes auraient toujours le même nom :

shopping.clients.Compte

Sun recommande vivement une convention de nommage qui réduit beaucoup ce risque — ajouter devant chaque nom de classe votre nom de domaine inversé. Souvenez-vous que les noms de domaine sont garantis uniques. Deux personnes différentes peuvent s'appeler Barthélemy Durand, deux domaines différents ne peuvent pas être nommés ceci.com.

fr.jtlp.LivreJava

nom du package

nom de la classe

Noms de domaine inversés

fr.jtlp.projets.LivreJava

L'initiale des noms de classes est toujours une majuscule.

Commencez le nom de package par votre nom de domaine inversé séparé par un point (.), puis ajoutez votre propre structure.

Si projets.LivreJava peut être un nom répandu, ajouter fr.oreilly signifie que nous n'avons plus à nous soucier que des développeurs internes.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Pour placer votre code dans un package :

① Choisissez un nom de package

Dans notre exemple, nous utilisons com.jtlp. Le nom de la classe étant ExercicePackage, le nom complet de la classe est **maintenant fr.jtlp.ExercicePackage**.

② Placez une instruction package dans la classe

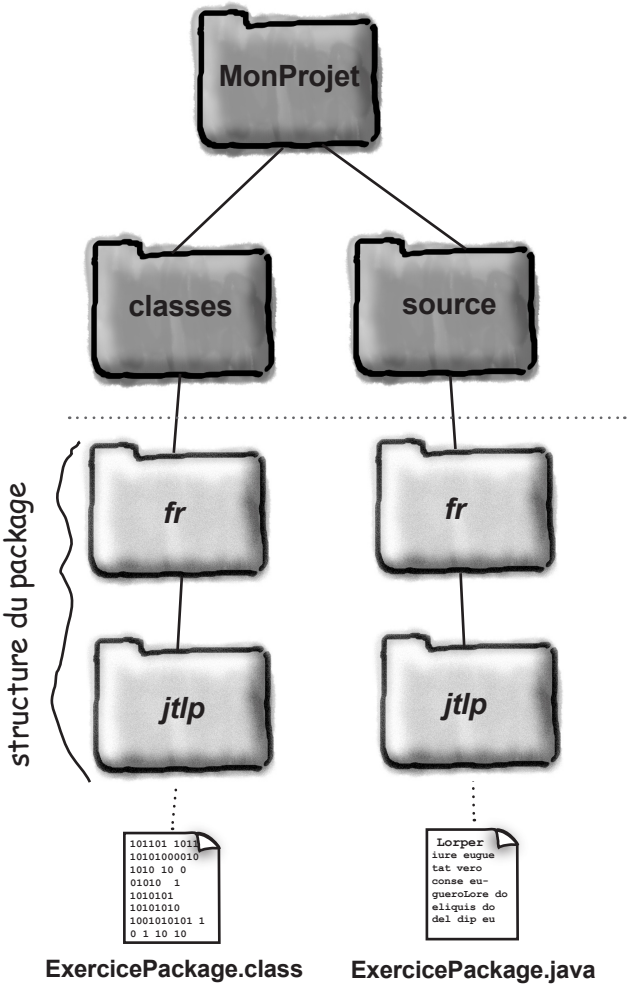
Ce doit être la première instruction du fichier source, avant toute instruction import. Il ne peut y en avoir qu'une par fichier source, donc toutes les **classes d'un fichier source doivent être dans le même package**. Bien entendu, ceci s'applique aussi aux classes internes.

```
package fr.jtlp;

import javax.swing.*;

public class ExercicePackage {
   // code qui change la vie
}
```

③ Concevez une structure de répertoires

Il ne suffit pas de *dire* que votre classe est dans un package en insérant simplement une instruction package dans le code. Votre classe n'est pas *vraiment* dans un package tant qu'elle n'est pas dans une structure de répertoires. Si le nom complet de la classe est com.jtlp. ExercicePackage, vous ***devez*** placer le code source de ExercicePackage dans un répertoire nommé **jtlp** qui ***doit*** lui-même être dans un répertoire nommé **fr**.
Il est *possible* de compiler sans cela, mais vous pouvez nous faire confiance — cela n'en vaut pas la peine en comparaison des problèmes qui s'ensuivraient. Placez le code source qui correspond à la structure du package et vous éviterez bien des migraines en aval.

Vous devez placer une classe dans une structure de répertoire qui corresponde à la hiérarchie du package.



Mettez en place une structure de répertoires homogène. L'arborescence des sources et des classes doit correspondre.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Compiler et exécuter avec des packages

Quand votre classe se trouve dans un package, la compilation et l'exécution sont un peu plus délicates. La principale difficulté est que le compilateur et la JVM doivent être capables de trouver votre classe et toutes les autres classes qu'elle utilise. Pour celles de l'API noyau, ce n'est jamais un problème. Java sait toujours où se trouvent ses propres composants. Mais en ce qui concerne vos classes, la solution consistant à compiler depuis le répertoire dans lequel se trouvent les fichiers source ne fonctionnera pas (ou du moins pas de façon *fiable*). Mais si vous vous conformez à la structure décrite dans cette page, la réussite est garantie. Il existe d'autres façons de procéder, mais c'est celle que nous trouvons la plus fiable et la plus facile à appliquer systématiquement.

Compiler avec l'option `-d` (directory)

```
%cd MonProjet/source
```

Restez dans le répertoire source! Ne descendez PAS dans le répertoire où se trouve le fichier .java!

```
%javac -d ../classes fr/jtlp/ExercicePackage.java
```

L'option -d dit au compilateur de placer le code compilé (les fichiers classe) dans le répertoire classes, dans la bonne structure de package!! Oui, elle sait comment faire.

Maintenant, vous devez spécifier le CHEMIN du fichier source.

Pour compiler tous les fichiers .java du package fr.jtlp :

```
%javac -d ../classes fr/jtlp/*.java
```

Compile tous les fichiers source (.java) de ce répertoire.

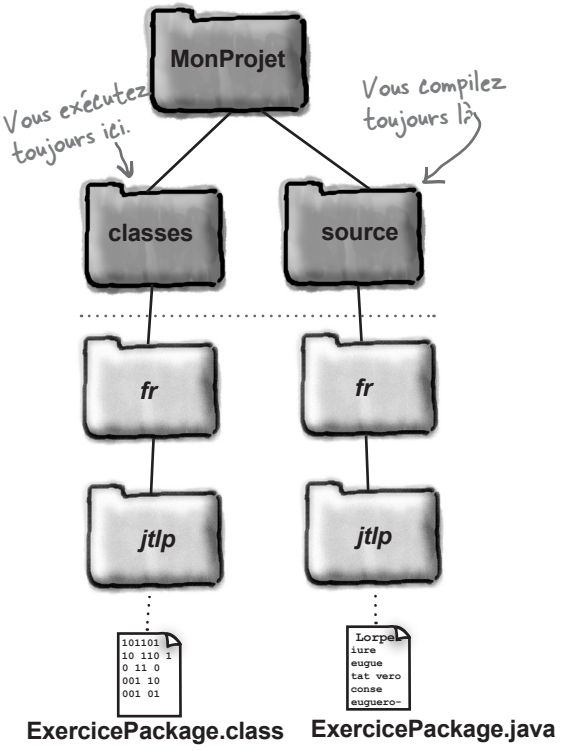
Exécuter le code

```
%cd MonProjet/classes
```

Exécutez votre programme depuis le répertoire "classes".

```
%java fr.jtlp.ExercicePackage
```

Vous DEVEZ fournir le nom complet de la classe! La JVM recherchera immédiatement dans son répertoire courant (classes) et s'attendra à trouver un répertoire nommé com, dans lequel elle pense trouver un répertoire nommé jtlp qui doit contenir la classe. Si la classe est dans le répertoire "fr", ou même dans "classes", cela ne marche pas!



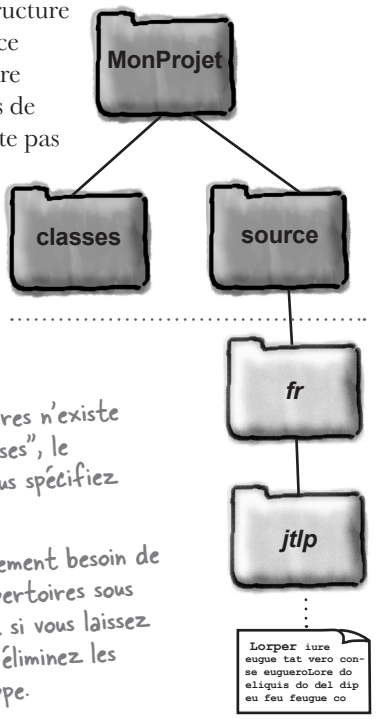
Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

L'option -d est vraiment cool

L'option **-d** est fantastique. Non seulement elle permet de placer automatiquement les fichiers compilés dans un autre répertoire que les fichiers source, mais elle sait également comment insérer la classe dans la bonne structure de répertoires.

Mais ce n'est pas tout !

Disons que vous avez une bonne structure de répertoires pour votre code source mais que vous n'avez pas de structure correspondante pour les classes. Pas de problème ! L'option -d ne se contente pas de *mettre* les classes dans le bon répertoire, elle *crée* les répertoires s'ils n'existent pas.



ExercicePackage.java

Si la structure de répertoires n'existe pas sous le répertoire "classes", le compilateur la créera si vous spécifiez l'option -d.

Vous n'avez donc pas réellement besoin de créer physiquement les répertoires sous le répertoire "classes". Et si vous laissez faire le compilateur, vous éliminez les risques de fautes de frappe.

L'option -d dit au compilateur « Place la classe dans la structure de répertoires du package, en utilisant la classe spécifiée après l'option -d comme racine. Mais... si les répertoires n'existent pas, crée les d'abord *puis* mets la classe au bon endroit ! »

il n'y a pas de Questions stupides

Q : J'ai essayé de me positionner dans le répertoire de ma classe principale, et maintenant la JVM me dit qu'elle ne peut pas trouver la classe ! Pourtant elle est LÀ, dans le répertoire courant !

R : Une fois que votre classe est dans un package, vous ne pouvez pas l'appeler par son nom « court ». Vous DEVEZ spécifier sur la ligne de commande le nom pleinement qualifié de la classe dont vous voulez exécuter la méthode main(). Mais comme ce nom contient la structure du *package*, Java insiste pour que la classe soit dans la structure de *répertoires* correspondante. Si vous tapez sur la ligne de commande :

```
%java com.foo.Livre
```

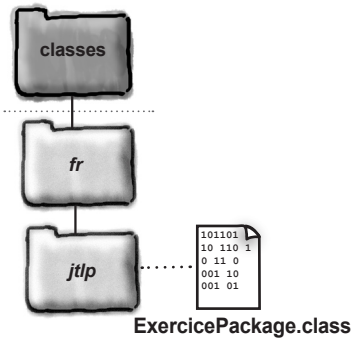
la JVM cherchera dans le répertoire courant (et le reste du classpath) un répertoire nommé « com ». ***Elle ne cherchera pas une classe nommée Livre tant qu'elle n'aura pas trouvé un répertoire nommé « com » contenant un sous-répertoire nommé « foo »***. Ce n'est qu'alors que la JVM reconnaîtra qu'elle a trouvé la bonne classe Livre. Si elle trouve une classe Livre n'importe où ailleurs, elle suppose que la classe n'est pas dans la bonne structure, même si elle y est ! Elle ne va pas s'amuser à remonter dans l'arborescence et dire « Oh, je vois qu'au-dessus de nous il y a un répertoire nommé com, donc ce doit être le bon package… ».

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

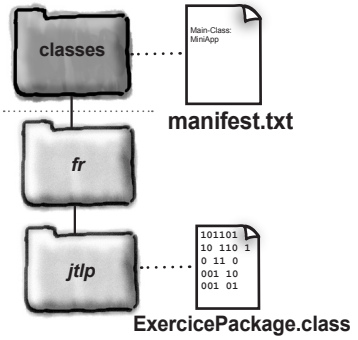
Créer un JAR exécutable avec des packages

Quand votre classe est dans un package, la structure de répertoires doit être dans le JAR ! Vous ne pouvez pas vous contenter de fourrer vos classes dans le JAR comme nous l'avons fait pour les pré-packages. Et vous devez veiller à ne pas inclure d'autres répertoires au-dessus de votre package. Le premier répertoire de votre package (habituellement com) doit être le premier répertoire du JAR ! Si vous placez accidentellement un répertoire *au-dessus* du package (par exemple le répertoire « classes »), le JAR ne fonctionnera pas correctement.

Créer un JAR exécutable



1 Vérifiez que tous vos fichiers classes sont dans la bonne structure de package, sous le répertoire classes.

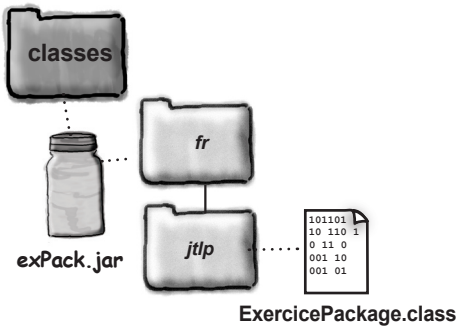


2 Créez un fichier manifest.txt qui spécifie quelle classe contient la méthode main(), et veillez à utiliser le nom complet de la classe!

Le fichier texte nommé manifest.txt ne contient qu'une seule ligne :

```
Main-Class: fr.jtlp.ExercicePackage
```

Placez le fichier manifeste dans le répertoire « classes ».



3 Exécutez l'outil jar pour créer un fichier JAR qui contiendra les répertoires du package plus le manifeste.

Le seul élément que vous devez inclure est le répertoire « fr », et le package entier (avec toutes les classes) sera placé dans le JAR.

```
%cd MonProjet/classes

%jar -cvmf  manifest.txt  exPack.jar  fr
```

Vous ne spécifiez que le répertoire fr! Tout le reste s'y retrouvera automatiquement!

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Mais où est passé le manifeste?

Pourquoi ne pas regarder dans le JAR et le chercher? En mode ligne de commande, l'outil jar peut faire bien plus que créer et exécuter un JAR. Vous pouvez extraire le contenu du JAR (tout comme pour un fichier zip ou un fichier tar).

Imaginez que vous ayez placé votre fichier exPack.jar dans un répertoire nommé Alibaba.

Commandes jar pour lister et extraire

① Lister le contenu d'un JAR

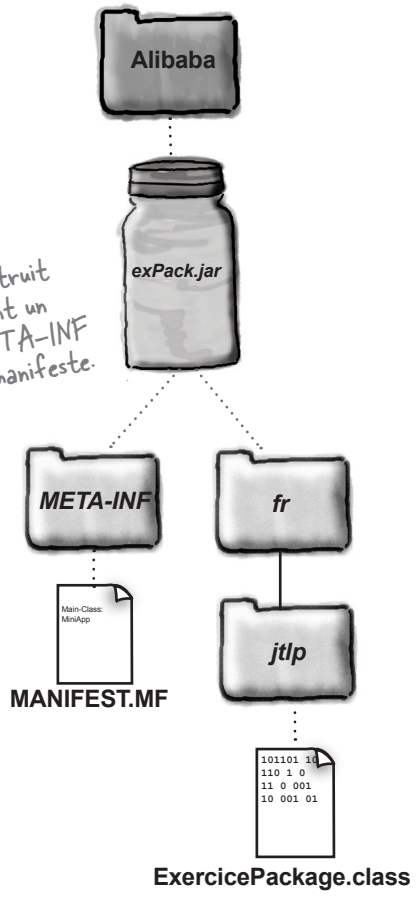
```
% jar -tf exPack.jar
```

-tf signifie "Table File", autrement dit « affiche-moi un tableau du fichier JAR ».

```
Fichier Edition Fenêtre Aide SésameOuvreToi
% cd Alibaba
% jar -tf exPack.jar
META-INF/
META-INF/MANIFEST.MF
com/
com/jtlp/
com/jtlp/ExercicePackage
.class
```

Nous plaçons le fichier JAR dans un répertoire nommé Alibaba.

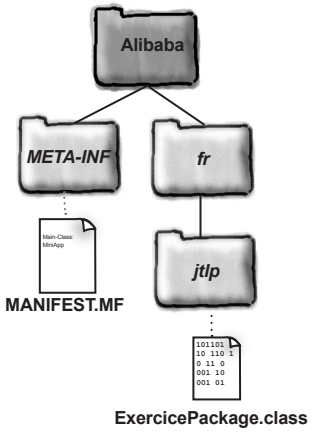
L'outil jar construit automatiquement un répertoire META-INF et y place le manifeste.



② Extraire le contenu d'un JAR

```
% cd Alibaba
% jar -xf exPack.jar
```

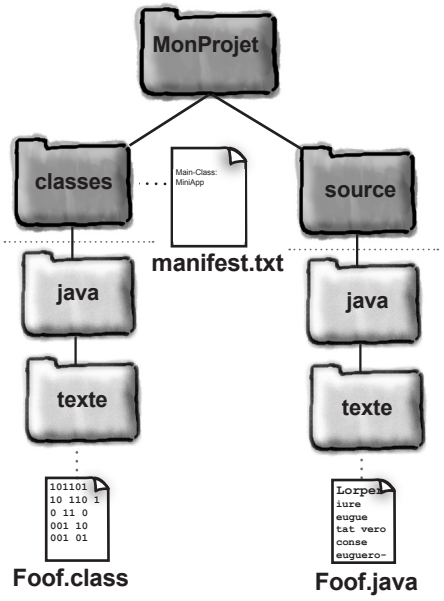
-xf signifie "eXtract File" et fonctionne comme si vous décompressiez un fichier zip ou tar. Si vous extrayez exPack.jar, vous verrez un répertoire META-INF et le répertoire com dans votre répertoire courant.



META-INF signifie « méta-information». L'outil jar crée le répertoire META-INF et le fichier MANIFEST.MF. Il lit également le contenu de votre fichier manifest.txt et l'insère dans le fichier MANIFEST.MF. Ce n'est donc pas *votre fichier* manifeste qui va dans le JAR, mais c'est son *contenu* qui est placé dans le « vrai» manifeste (MANIFEST.MF).

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

À vos crayons



Étant donné la structure de répertoires ci-contre, que devez-vous taper sur la ligne de commande pour compiler, exécuter le programme, créer un JAR, et exécuter le JAR ? Supposez que nous appliquions le standard selon lequel la structure de répertoires du package commence juste en dessous des répertoires *source* et *classes*. Autrement dit, les répertoires *source* et *classes* ne font pas partie du package.

Compiler :

```
%cd source
%javac _____
```

Exécuter :

```
%cd _____
%java _____
```

Créer le JAR :

```
%cd _____
% _____
```

Exécuter le JAR :

```
%cd _____
% _____
```

Question bonus : qu'est-ce qui ne va pas dans le nom du package ?

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

il n'y a pas de
Questions stupides

Q : Que se passe-t-il si on essaie de lancer un JAR exécutable et que l'utilisateur final n'a pas installé java ?

R : Rien du tout, puisqu'un programme Java ne peut pas tourner en l'absence de JVM. L'utilisateur final doit installer Java.

Q : Comment puis-je installer Java sur la machine de l'utilisateur final ?

R : Dans l'idéal, vous pouvez créer un assistant d'installation et le distribuer avec votre application. Plusieurs fournisseurs proposent des programmes d'installation qui vont du plus simple au plus puissant. Un tel programme peut par exemple détecter si l'utilisateur final dispose de la bonne version de Java, et, dans le cas contraire, installer et configurer Java avant d'installer votre application. Installshield, InstallAnywhere et DeployDirector offrent tous trois ce type de solution.

Un autre aspect sympathique des programmes d'installation est que vous pouvez même créer un CD-ROM de déploiement contenant des installateurs pour les principales plates-formes Java, et donc d'avoir un unique CD. Si par exemple l'utilisateur travaille sous Solaris, c'est la version Solaris de Java qui est installée. Sous Windows, c'est la version Windows, etc. Si vous disposez du budget, c'est de loin la meilleure façon de permettre à vos utilisateurs finaux d'installer et de configurer la bonne version de Java.

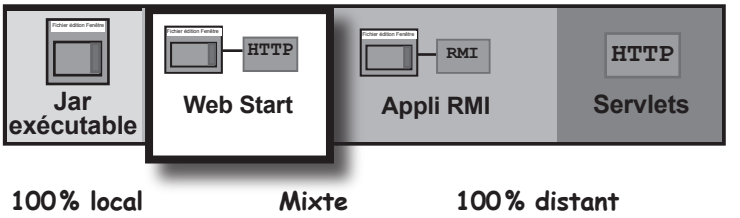
POINTS D'IMPACT

- Organisez votre projet pour que votre code source et vos fichiers classe soient dans le même répertoire.
- Une structure d'organisation standard consiste à créer un répertoire *projet* puis à y placer un répertoire *classes* et un répertoire *source*.
- Organiser vos classes en packages empêche les collisions de nommage avec d'autres classes si vous placez votre nom de domaine inversé avant le nom des classes.
- Pour insérer une classe dans un package, placez une instruction package au début du fichier source, avant toutes les instructions import :
  `package com.supergenial;`
- Pour être dans un package, une classe doit être dans *une structure de répertoires qui correspond exactement à la structure du package.* Pour une classe com.supergenial.Foo, la classe Foo doit être dans un répertoire nommé *supergenial* qui est dans un répertoire nommé *com*.
- Pour que votre classe compilée se retrouve dans la bonne structure de répertoires sous le répertoire *classes*, utilisez l'option de compilation `-d`:
  ```
  % cd source
  % javac -d ../classes com/supergenial/Foo.java
  ```
- Pour exécuter votre code, positionnez-vous dans le répertoire classes et spécifiez le nom complet de la classe :
  ```
  % cd classes
  % java com.supergenial.Foo
  ```
- Vous pouvez grouper vos classes dans des fichiers JAR (Java ARchive). JAR est basé sur le format pkzip.
- Vous pouvez rendre un JAR exécutable en y plaçant un manifeste qui spécifie la classe qui contient la méthode main(). Pour créer un fichier manifeste, créer un fichier texte d'une seule ligne, par exemple) :
  `Main-Class: com.supergenial.Foo`
- N'oubliez pas d'appuyer sur la touche Entrée à la fin de la ligne Main-Class ou votre manifeste risque de ne pas fonctionner.
- Pour créer un fichier JAR, tapez :
  `jar -cvfm manifest.txt MonJar.jar com`
- Toute la structure de répertoires du package (et *seulement* les répertoires du package) doivent se retrouver immédiatement dans le JAR.
- Pour lancer un fichier JAR exécutable, tapez :
  `java -jar MonJar.jar`

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Les JAR exécutables sont intéressants, mais quel rêve ce serait s'il y avait un moyen de créer un client riche autonome en mode graphique que l'on pourrait distribuer sur le web. Ça éviterait de graver et de distribuer tous ces CD-ROM. Et ne serait-ce pas tout simplement merveilleux si le programme s'actualisait tout seul, en ne remplaçant que les composants qui ont changé? Les clients seraient à jour en permanence et on n'aurait plus jamais besoin de livrer de nouvelles versions...

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>



Java Web Start

Avec Java Web Start (JWS), votre application est initialement lancée depuis un navigateur web (d'où le nom de *Web Start*) mais elle s'exécute sous forme d'application autonome (enfin *presque*) sans les contraintes du navigateur. Et une fois téléchargée sur la machine de l'utilisateur final (la première fois que celui-ci clique sur le lien qui déclenche le téléchargement), elle y reste.

Java Web Start est, entre autres, un petit programme Java qui réside sur la machine cliente et fonctionne de façon très comparable à un plug-in (comme par exemple Adobe Acrobat Reader qui s'ouvre quand votre navigateur accède à un fichier .pdf). Ce programme est le gestionnaire d'applications JWS, et sa principale finalité est de gérer le téléchargement, la mise à jour et le lancement (l'exécution) de vos applications JWS.

Quand JWS télécharge votre application (un JAR exécutable), il invoque la méthode main() de celle-ci. Après quoi l'utilisateur final peut lancer le programme directement depuis le gestionnaire d'applications JWS sans repasser par le lien de la page web.

Mais il y a mieux encore. La caractéristique la plus étonnante de JWS est sa capacité à détecter si une portion quelconque, si petite soit-elle, de l'application (par exemple, un seul fichier classe) a changé sur le serveur, et à télécharger et intégrer le code mis à jour *sans* la moindre intervention de l'utilisateur final.

Bien sûr, il reste un problème : comment l'utilisateur se procure-t-il Java et Java Web Start ? Il a besoin des deux Java pour exécuter l'application et Java Web Start (une petite application Java *par elle-même*) pour l'extraire et la lancer. Eh bien le problème est résolu. Vous pouvez faire en sorte que les utilisateurs finaux qui ne possèdent pas JWS peuvent le télécharger sur le site de Sun. Et s'ils ont JWS mais que leur version de Java est périmée (parce que vous avez spécifié dans votre application JWS qu'il fallait une version spécifique de Java), ils peuvent également télécharger Java 2 Standard Edition sur leur machine.

Enfin, JWS est simple à utiliser. Vous pouvez servir une application JWS pratiquement comme tout autre type de ressource web, comme une bonne vieille page HTML ou une image JPEG. Vous publiez une page web (HTML) avec un lien vers votre application JWS et vous êtes prêt. Finalement, votre application JWS n'est pas beaucoup plus qu'un JAR exécutable que l'utilisateur final peut télécharger sur l'Internet.

Les utilisateurs finaux lancent une application Java Web Start en cliquant sur un lien dans une page web. Mais une fois téléchargée, elle s'exécute en dehors du navigateur, tout comme n'importe quelle autre application Java autonome. En réalité, une application JWS n'est autre qu'un JAR exécutable distribué sur le Web.

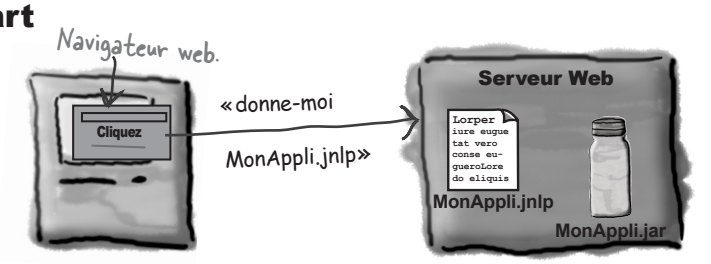
Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Comment marche Java Web Start

1. **Le client clique dans une page web sur un lien vers votre application JWS (un fichier .jnlp).**

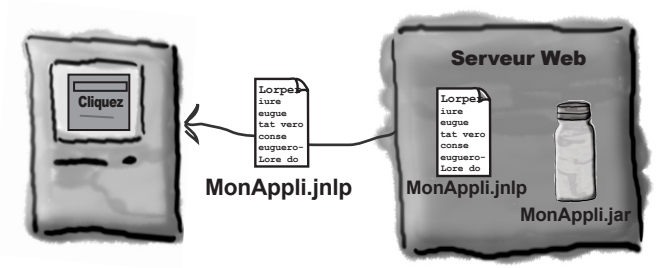
   Le lien dans la page Web

   ```
   <a href="MonAppli.jnlp">Cliquez</a>
   ```

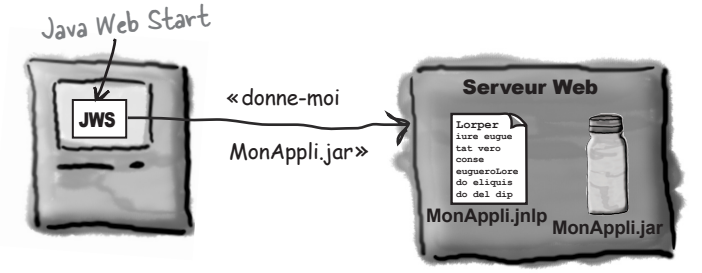


2. **Le serveur web (HTTP) reçoit la requête et renvoie un fichier .jnlp (ce n'est PAS le JAR).**

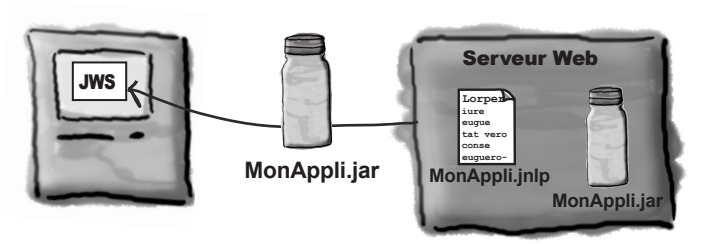
   Le fichier .jnlp est un document XML qui spécifie le nom du fichier JAR exécutable de l'application.



3. **Le navigateur lance Java Web Start (une petite application côté client). Le gestionnaire d'applications JWS lit le fichier .jnlp et demande au serveur le fichier MonAppli.jar.**



4. **Le serveur web «sert» le fichier .jar demandé.**



5. **Java Web Start obtient le JAR et lance l'application en appelant la méthode main( ) spécifiée (comme un JAR exécutable).**

   La prochaine fois que l'utilisateur voudra exécuter ce programme, il pourra ouvrir l'application Java Web Start et lancer votre application sans même être en ligne.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Le fichier .jnlp

Pour créer une application Java Web Start, il vous faut un fichier .jnlp (Java Network Launch Protocol) qui décrit votre application. C'est ce fichier que lit l'application JWS et qu'elle utilise pour trouver le JAR et lancer le programme (en appelant la méthode main() du JAR). Un fichier .jnlp est un simple document XML. Vous pouvez y placer différents éléments, mais il contient au minimum ceci :

```
<?xml version="1.0" encoding="utf-8"?>

<jnlp spec="0.2 1.0"
      codebase="http://127.0.0.1/~kathy"
      href="MonAppli.jnlp">

   <information>
      <title>kathy App</title>
      <vendor>Wickedly Smart</vendor>
      <homepage href="index.html"/>
      <description>Head First WebStart demo</description>
      <icon href="kathys.gif"/>
      <offline-allowed/>
   </information>

   <resources>
      <j2se version="1.3+"/>
      <jar href="MonAppli.jar"/>
   </resources>

   <application-desc main-class="HelloWebStart"/>
</jnlp>
```

C'est dans la balise "codebase" que vous spécifiez la "racine" de votre programme JWS sur le serveur. Comme nous testons sur la machine locale, nous utilisons l'adresse de loopback 127.0.0.1. Pour accéder au serveur web, ce serait par exemple "http://www.supergenial.com".

Ceci indique l'emplacement du fichier .jnlp par rapport à "codebase". Dans cet exemple, nous voyons que MonAppli.jnlp se trouve dans le répertoire racine du serveur web et qu'il n'est pas imbriqué dans un autre répertoire.

N'oubliez pas d'inclure toutes ces balises ou votre application risque de ne pas fonctionner correctement! les balises "information" sont utilisées par JWS, principalement pour l'affichage quand l'utilisateur veut relancer une application déjà téléchargée.

Cette ligne signifie que l'utilisateur peut exécuter votre programme sans être connecté à l'Internet. Mais s'il n'est pas en ligne, la mise à jour automatique ne fonctionnera pas.

Cette ligne spécifie que l'application nécessite au moins Java 1.3.

Le nom du JAR exécutable! Vous pouvez également avoir d'autres fichiers JAR qui contiennent d'autres classes ou même des sons ou des images utilisés par votre application.

Comme l'entrée Main-Class du manifeste, cette ligne indique quelle classe du JAR contient la méthode main().

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Étapes de la création et du déploiement d'une application Java Web Start

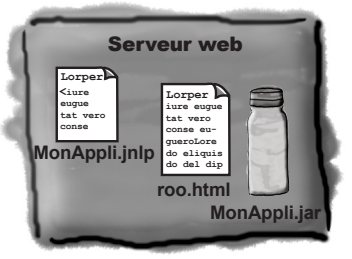
1. Créez un JAR exécutable pour votre application.

MonAppli.jar

2. Écrivez un fichier .jnlp.

MonAppli.jnlp

3. Placez les fichiers JAR et .jnlp sur votre serveur web.



4. Ajoutez un nouveau type MIME à votre serveur web.

`application/x-java-jnlp-file`

Cela permet au serveur d'envoyer le fichier .jnlp avec l'en-tête correct. Quand le navigateur recevra le fichier .jnlp, il saura ce que c'est et comment lancer le JWS.

5. Créez une page web contenant un lien vers votre fichier .jnlp

MyJWSApp.html

```
<HTML>
  <BODY>
    <a href="MonAppli22.jnlp">Lancer mon application</a>
  </BODY>
</HTML>
```


Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

L'œuf et la poule

Regardez la sequence d'événements ci-dessous et placez-les dans l'ordre dans lequel ils se produisent dans une application JWS.

- le navigateur web lance le gestionnaire d'applications JWS
- le serveur web envoie un JAR au gestionnaire JWS
- le gestionnaire JWS demande le fichier JAR
- le serveur web envoie un fichier .jnlp au navigateur
- le gestionnaire JWS invoque la méthode main() du JAR
- l'utilisateur clique sur un lien dans une page web
- le navigateur demande un fichier .jnlp au serveur web

1.

2.

3.

4.

5.

6.

7.

il n'y a pas de Questions stupides

Q : En quoi Java Web Start diffère-t-il d'une applet ?

R : Les applets ne peuvent pas s'exécuter en dehors d'un navigateur web. Une applet n'est pas simplement chargée à partir d'une page web : elle fait partie de la page. Autrement dit, le navigateur considère l'applet comme un fichier JPEG ou toute autre ressource et il utilise soit un plug-in Java soit sa propre console Java intégrée (beaucoup plus courante aujourd'hui) pour l'exécuter. Les applets n'ont pas le même niveau de fonctionnalité. Par exemple, elles ne sont pas mises à jour automatiquement, et elles doivent toujours être lancées à partir du navigateur. En revanche, une fois une application JWS téléchargée, l'utilisateur n'a même pas besoin d'un navigateur pour la relancer localement. Il lui suffit de lancer JWS et de l'utiliser pour exécuter l'application déjà téléchargée.

Q : Quelles sont les restrictions de sécurité de JWS ?

R : Les applications JWS présentent plusieurs limitations, dont l'impossibilité de lire et d'écrire sur le disque dur de l'utilisateur. Mais... JWS a sa propre API et une boîte de dialogue qui permet d'ouvrir et de sauvegarder des fichiers dans une zone restreinte du disque avec la permission de l'utilisateur.

POINTS D'IMPACT

- La technologie Java Web Start vous permet de déployer une application cliente autonome à partir du Web.
- Java Web Start (JWS) comporte un gestionnaire d'applications qui doit être installé sur le client (avec Java).
- Une application JWS a deux composants : un JAR exécutable et un fichier .jnlp.
- Un fichier .jnlp est un simple document XML qui décrit votre application JWS. Des balises spécifient le nom et l'emplacement du JAR, et le nom de la classe qui contient la méthode main().
- Quand un navigateur reçoit un fichier jnlp du serveur (parce que l'utilisateur a cliqué sur le lien qui pointe dessus), il lance le gestionnaire d'applications JWS.
- Le gestionnaire JWS lit le fichier .jnlp et demande le JAR exécutable au serveur web.
- Quand JWS reçoit le JAR, il invoque la méthode main() spécifiée dans le fichier jnlp.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Dans ce chapitre, nous avons étudié les packages, le déploiement et JWS. Vous devez déterminer si chacune des instructions ci-dessous est vraie ou fausse.

Vrai ou Faux

1. Le compilateur Java a une option, -d, qui vous permet de décider où placer vos fichiers .class.

2. Un JAR est un répertoire standard dans lequel vos fichiers .class doivent résider.

3. Pour produire une archive JAR, vous devez créer un fichier nommé jar.mf.

4. Dans un JAR, le fichier de support déclare quelle classe contient la méthode main().

5. Les fichiers JAR doivent être dézipés avant que la JVM ne puisse utiliser les classes.

6. En mode ligne de commande, on invoque un JAR à l'aide de l'option -arch.

7. La structure d'un package est représentée de façon significative et hiérarchisée.

8. L'emploi du nom de domaine de votre entreprise est déconseillé pour nommer les packages.

9. Différentes classes d'un même fichier source peuvent appartenir à des packages différents.

10. Pour compiler les classes d'un package, l'option -p est vivement recommandée.

11. Pour compiler les classes d'un package, le nom complet doit refléter l'arborescence des répertoires.

12. Un usage judicieux de l'option -d peut contribuer à garantir l'absence de fautes de frappe.

13. L'extraction d'un JAR contenant des packages créera un répertoire nommé meta-inf.

14. L'extraction d'un JAR contenant des packages créera un fichier nommé manifest.mf.

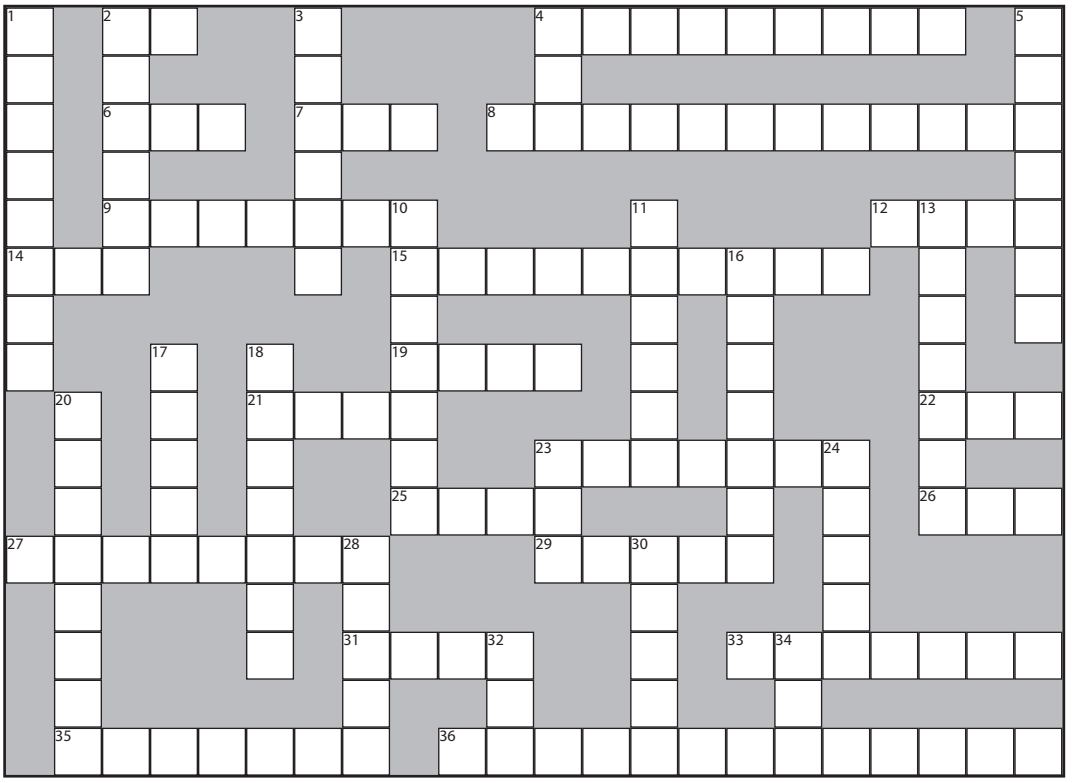
15. Le gestionnaire d'applications JWS s'exécute depuis un navigateur.

16. Les applications JWS nécessitent un fichier .nlp (Network Launch Protocol) pour fonctionner.

17. La méthode main() d'une application JWS est spécifiée dans son fichier JAR.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Mots-Croisés 7.0



Horizontalement

2. Acronyme orienté objet
4. Marque la fugacité
6. Archive Java
7. Méthode de thread
8 Empêche les threads de s'emmêler
9. Elles ont du caractère
12. Début de bloc conditionnel
14. Accesseur
15. Peut tourner
19. main() est public static ____
21. Non définie
22. Documentation très abrégée
23. Pour se connecter
25. Type
26. Sujet du prochain chapitre
29. Conditionnements
29. Essais
31 On écoute dessus
33. Début de méthode agile
35. Pas un client
36. L'un des principes OO

Verticalement

1. Ils vont en boîte
2. LA superclasse !
3. Type pour textes
4. Précède le catch
5. Composants d'IHM
10. S'exécute sur un serveur
11. Sans instances
13. Faire ce que fait la méthode get
16. Servent de tampon
17. Sort prématurément
18. Enveloppe d'entiers
20. Pleins de 1 et de 0
23. Mutateur
24. Court mais entier
28. Appelle la méthode de la superclasse
30. Met en sommeil
32. Prend la tangente
34. Format de JNLP

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Solutions des exercices

1. l'utilisateur clique sur un lien dans une page web
2. le navigateur demande un fichier .jnlp au serveur web
3. le serveur web envoie un fichier .jnlp au navigateur
4. le navigateur web lance le gestionnaire d'applications JWS
5. le gestionnaire JWS demande le fichier JAR
6. le serveur web envoie un JAR au gestionnaire JWS
7. le gestionnaire JWS invoque la méthode main() du JAR

Vrai 1. Le compilateur Java a une option, -d, qui vous permet de décider où placer vos fichiers .class.

Faux 2. Un JAR est un répertoire standard dans lequel vos fichiers .class doivent résider.

Faux 3. Pour produire une archive JAR, vous devez créer un fichier nommé jar.mf.

Vrai 4. Dans un JAR, le fichier de support déclare quelle classe contient la méthode main().

Faux 5. Les fichiers JAR doivent être dézipés avant que la JVM ne puisse utiliser les classes.

Faux 6. En mode ligne de commande, on invoque un JAR à l'aide de l'option -arch.

Vrai 7. La structure d'un package est représentée de façon significative et hiérarchisée.

Faux 8. L'emploi du nom de domaine de votre entreprise est déconseillé pour nommer les packages.

Faux 9. Différentes classes d'un même fichier source peuvent appartenir à des packages différents.

Faux 10. Pour compiler les classes d'un package, l'option -p est vivement recommandée.

Vrai 11. Pour compiler les classes d'un package, le nom complet doit refléter l'arborescence des répertoires.

Vrai 12. Un usage judicieux de l'option -d peut contribuer à garantir l'absence de fautes de frappe.

Vrai 13. L'extraction d'un JAR contenant des packages créera un répertoire nommé meta-inf.

Vrai 14. L'extraction d'un JAR contenant des packages créera un fichier nommé manifest.mf.

Faux 15. Le gestionnaire d'applications JWS s'exécute depuis un navigateur.

Faux 16. Les applications JWS nécessitent un fichier .nlp (*Network Launch Protocol*) pour fonctionner.

Faux 17. La méthode main() d'une application JWS est spécifiée dans son fichier JAR.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

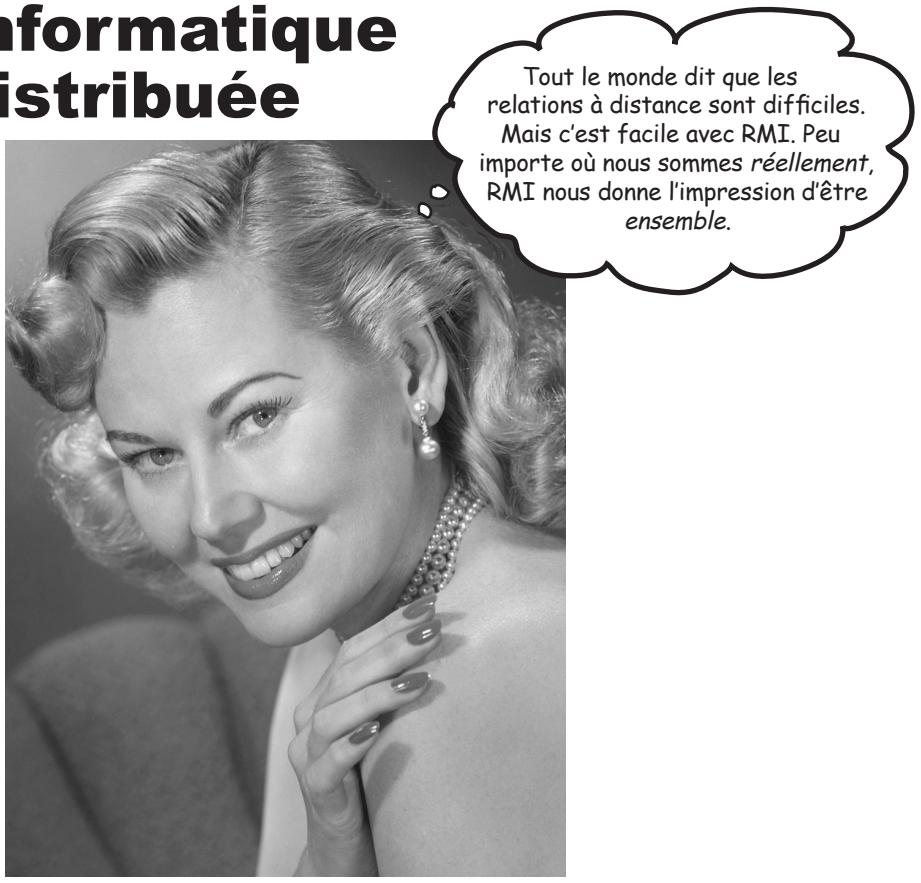
Mots-Croisés 7.0

1 D		2 O	O			3 S				4 T	R	A	N	S	I	E	N	T			5 W
I		B				T				R											I
A		6 J	A	R		7 R	U	N		8 S	Y	N	C	H	R	O	N	I	Z	E	D
L		E				I															G
O		9 C	H	A	I	N	E	10 S					11 S					12 C	13 A	S	E
14 G	E	T				G		15 E	X	E	C	U	T	A	16 B	L	E		C		T
U								R					A		U				C		S
E			17 B		18 I			19 V	O	I	D		T		F				E		
	20 B		R		21 N	U	L	L					I		F				22 D	O	C
	I		E		T			E			23 S	O	C	K	E	T	24 S		E		
	N		A		E			25 T	Y	P	E				R		H		26 R	M	I
27 P	A	C	K	A	G	E	28 S				29 T	E	30 S	T	S		O				
	I				E		U						L				R				
	R				R		31 P	O	R	32 T			E		33 E	34 X	T	R	E	M	E
	E						E			A			E			M					
	35 S	E	R	V	E	U	R		36 E	N	C	A	P	S	U	L	A	T	I	O	N

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

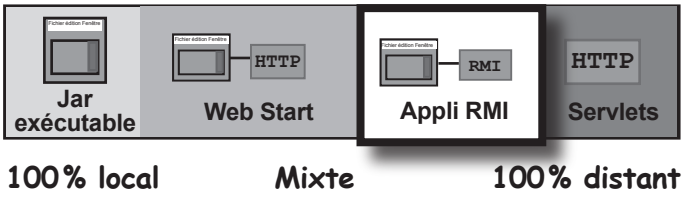
Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Informatique distribuée



Être à distance n'est pas forcément négatif. Bien sûr, les choses *sont* plus faciles quand toutes les parties de votre application sont centralisées, avec un seul tas et une seule JVM pour régir le tout. Mais ce n'est pas toujours possible. Ni désirable. Que se passe-t-il si votre programme a besoin de beaucoup de puissance de calcul mais que vos utilisateurs finaux ont un petit équipement apathique ? Ou si votre application lit dans une base de données, mais que seul le code résidant sur votre serveur peut accéder à la base pour des raisons de sécurité ? Imaginez un gros dorsal de commerce électronique qui doit s'exécuter au sein d'un système de gestion de transactions. Une partie de votre application *doit* parfois s'exécuter sur un serveur, alors que l'autre partie (généralement un client) doit être sur une *autre* machine. Dans ce chapitre, nous allons découvrir une technologie Java étonnamment simple : RMI (*Remote Method Invocation)*. Nous allons également jeter un bref coup d'œil sur les Servlets, les EJB (*Enterprise Java Beans*) et Jini, et voir comment EJB et Jini *dépendent* de RMI. Nous terminerons le livre par l'un des programmes les plus intéressants que vous puissiez écrire en Java, un *navigateur de services universels.*

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

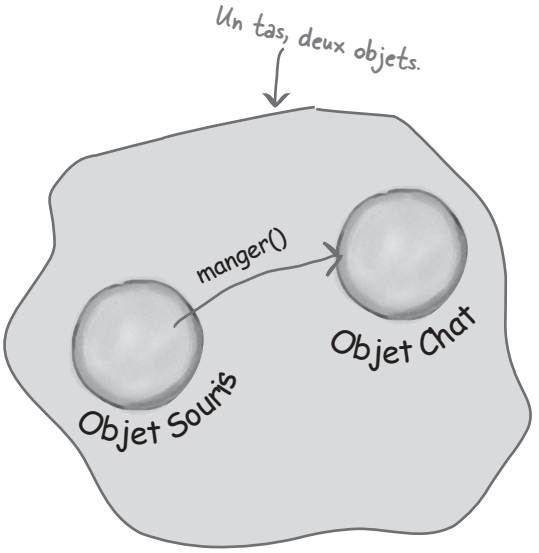


Les appels de méthodes sont toujours entre deux objets sur le même tas.

Jusqu'ici, toutes les méthodes que nous avons appelées concernaient un objet s'excutant dans la même machine virtuelle que l'appelant. Autrement dit, l'objet appelant et l'objet appelé (celui sur lequel nous appelons la méthode) résident sur le même tas.

```
class Souris{
    void go() {
        Chat c = new Chat();
        c.manger();
    }
    public static void main (String[] args) {
       Souris s = new Souris();
       s.go();
    }
}
```

Dans le code ci-dessus, nous savons que l'instance de Souris référencée par s et l'objet Chat référencé par c sont sur le même tas, exécutés par la même JVM. Souvenez-vous que la JVM a la responsabilité de placer des bits dans la variable référence qui représente la façon d'accéder à un objet sur le tas. La JVM sait toujours où se trouve chaque objet et comment le récupérer. Mais elle ne peut connaître les références que dans son propre tas ! Une JVM qui s'exécute sur une machine ne sait rien de l'espace de tas d'une JVM s'exécutant sur une autre machine. En fait, une JVM qui s'exécute sur une machine ne sait même pas ce qui se passe dans une autre JVM qui s'exécute sur la même machine. Cela ne fait aucune différence que les deux JVM soient sur des machines physiques différentes ou sur la même. Tout ce qui compte, c'est que ce sont deux invocations différentes de la JVM.



Dans la plupart des applications, quand un objet appelle une méthode sur un autre, les deux objets sont sur le même tas. Autrement dit, ils s'exécutent dans la même JVM.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Et si vous voulez appeler une méthode sur un objet qui s'exécute sur une autre machine?

Nous savons comment transmettre des informations à distance — avec des sockets et des E/S. Nous ouvrons une connexion Socket avec l'autre machine, nous obtenons un flot de sortie (un OutputStream) et nous y écrivons des données.

Mais si nous voulons *appeler une méthode* sur un objet qui réside sur l'autre machine... dans une autre JVM? Naturellement, nous pourrions toujours construire notre propre protocole. Quand nous enverrions des données à une ServerSocket, le serveur pourrait les analyser, comprendre ce que vous avez voulu dire, faire le travail et renvoyer le résultat dans un autre flot. Mais quelle galère. Imaginez comme ce serait agréable de pouvoir se contenter d'obtenir une référence à l'objet distant et d'appeler une méthode.

Imaginez deux ordinateurs...

Petit

Gros

Gros a quelque chose que Petit veut. De la puissance de calcul.

Petit veut envoyer des données à Gros, pour que Gros puisse faire le travail de force.

Petit veut simplement appeler une méthode...

```
double calculerAvecLaBase(CalcNombres nombres)
```

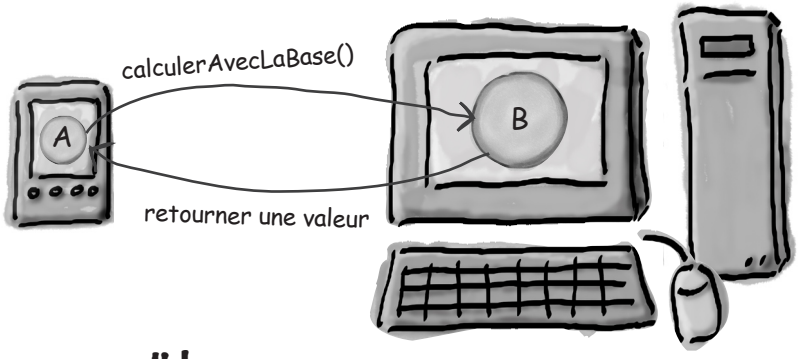
et obtenir le résultat.

Mais comment Petit obtient-il une référence à un objet qui réside sur Gros?

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

L'objet A qui s'exécute sur Petit veut appeler une méthode sur un objet B qui s'exécute sur Gros.

Question : comment amener un objet résidant sur une machine (tas différent, JVM différente) à appeler une méthode sur une autre machine ?



Mais ce n'est pas possible.

Enfin pas directement, en tous cas. Vous ne pouvez pas obtenir de référence à un objet qui se trouve sur un autre tas. Si vous écrivez :

Chien c = ???

Ce que c référence doit être dans le même espace de tas que le code qui exécute l'instruction.

Mais imaginez que vous vouliez écrire du code qui utilise des sockets et des E/S pour communiquer votre intention (un appel de méthode sur un objet résidant sur une autre machine), et que ça ressemble à un appel de méthode local.

Autrement dit, vous voulez provoquer un appel de méthode sur un objet *distant* (le tas étant ailleurs), mais avec du code qui vous *permette* de faire semblant d'appeler une méthode sur un objet local. La facilité d'un bon vieil appel de méthode de tous les jours mais avec la puissance d'un appel de méthode distant. Voilà notre objectif.

C'est ce que RMI (Remote Method Invocation) vous offre !

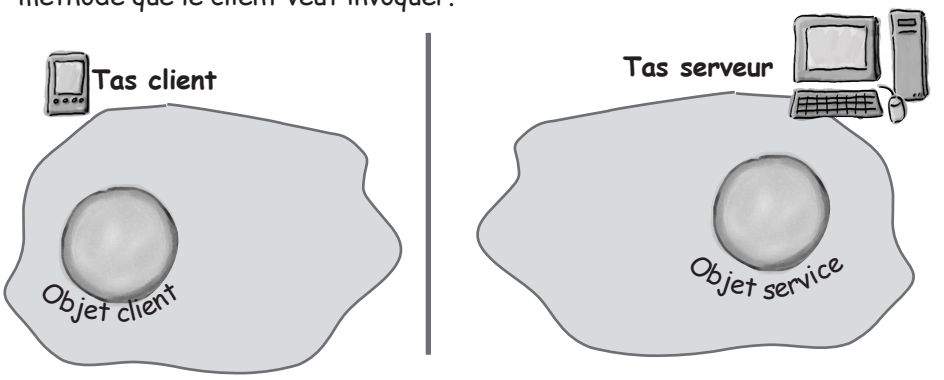
Mais revenons en arrière et imaginons comment vous concevriez un appel de méthode distant si vous l'écriviez vous-même. Comprendre ce que vous devriez construire vous aidera à comprendre comment RMI fonctionne.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

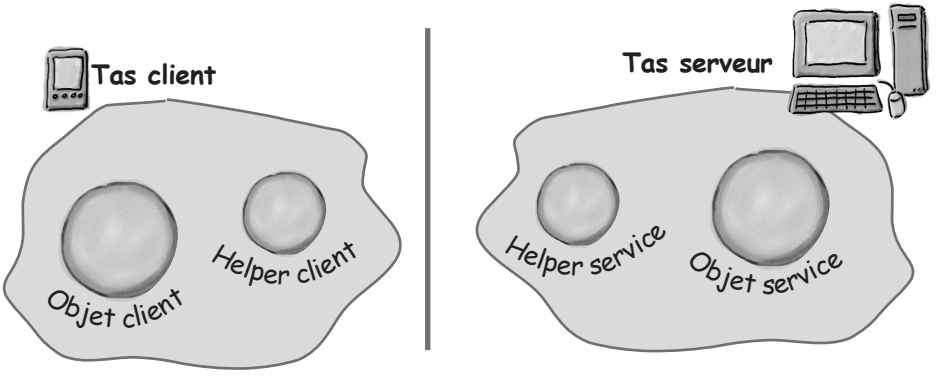
Concevoir des appels de méthodes distants

Créer quatre éléments : un serveur, un client, un helper serveur et un helper client (les aides serveur et clients).

1. Créer les applications client et serveur. L'application serveur est le **service distant** qui a un objet avec la méthode que le client veut invoquer.



2. Créer deux «helpers» client et serveur. Ils géreront tous les détails réseau et E/S de bas niveau pour que le client et le service puissent faire semblant d'être sur le même tas.



Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Le rôle des «helpers»

Les «helpers» sont les objets qui assurent réellement la communication. Ils permettent au client de se comporter comme s'il appelait une méthode sur un objet local. En fait, c'est bien le cas. le client appelle une méthode sur le helper client *comme si ce dernier était le service réel. Le helper client est un proxy, un substitut de l'Objet Réel.*

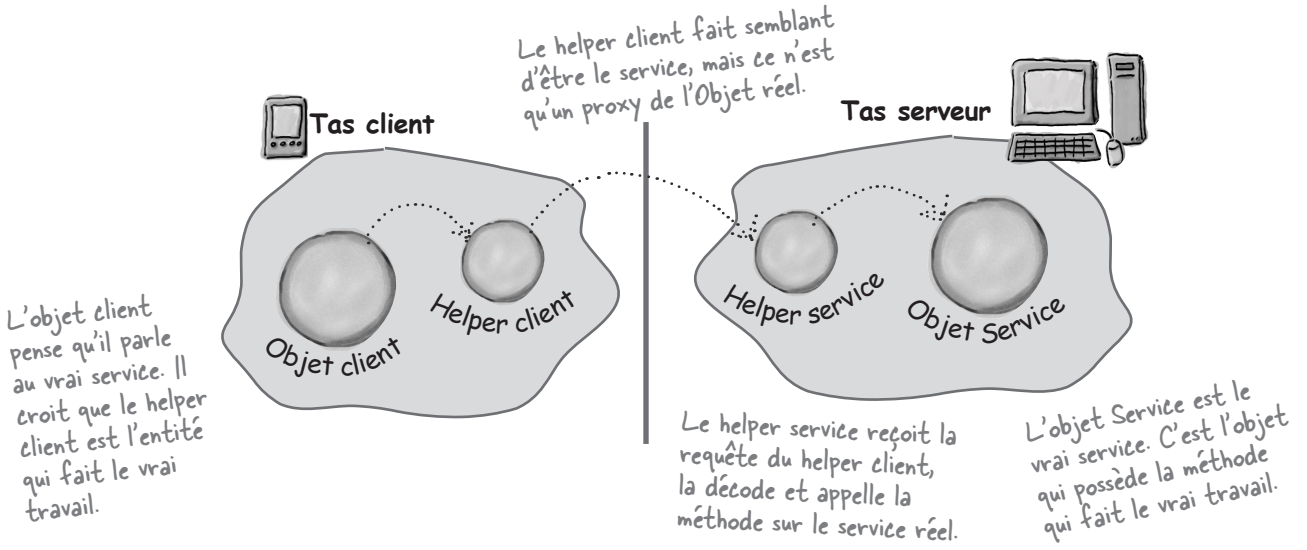
En d'autres termes, l'objet client pense qu'il appelle une méthode sur le service distant, parce que le helper client fait semblant d'être l'objet service. ***Il prétend être l'entité qui détient la méthode que le client veut appeler !***

Mais le helper client n'est pas réellement le service distant. Même si le helper se comporte comme si c'était vrai (parce qu'il a la même méthode que celle que le service annonce), il ne possède en réalité aucune des méthodes auxquelles le client s'attend. En revanche, il contacte le serveur, transfère les informations sur l'appel de méthode (nom de la méthode, arguments, etc.) et attend un retour du serveur.

Côté serveur, le helper service reçoit la requête du helper client (via une connexion Socket), décode les informations sur l'appel puis appelle la vraie méthode sur l'objet service réel. Pour l'objet service, l'appel est donc local. Il vient de son helper, non d'un client distant.

Le helper service reçoit du service la valeur de retour, l'encode et la renvoie (sur le flot de sortie d'une Socket) au helper client. Le helper client décode l'information et retourne la valeur à l'objet client.

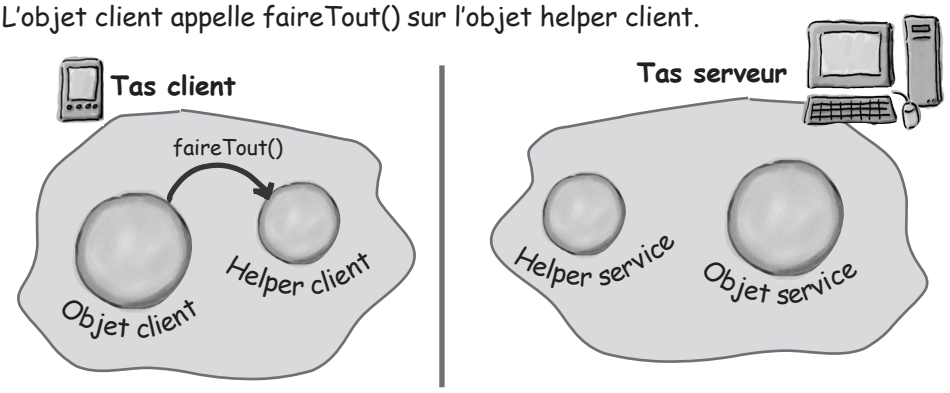
Votre objet client se comporte comme s'il effectuait des appels de méthodes distants. En réalité, il appelle des méthodes sur un objet «proxy» local qui gère les détails de bas niveau des sockets et des flots d'E/S.



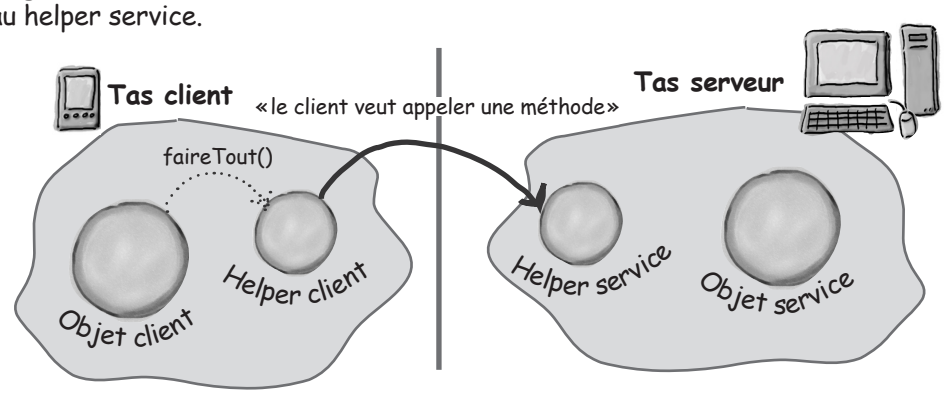
Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Déroulement de l'appel de méthode

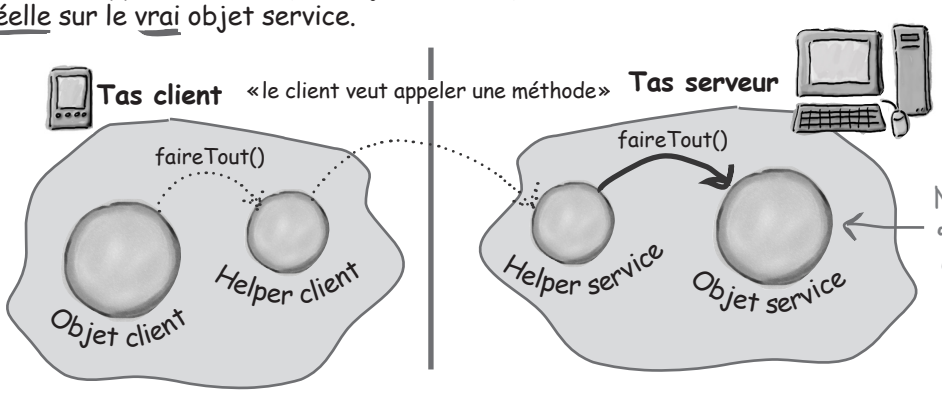
1. L'objet client appelle faireTout() sur l'objet helper client.



2. Le helper client encode les informations sur l'appel (arguments, nom de méthode, etc.) et l'envoie sur le réseau au helper service.



3. Le helper service décode les informations, détermine quelle méthode appeler (et sur quel objet) et invoque la méthode réelle sur le vrai objet service.



N'oubliez pas que c'est l'objet qui possède la méthode RÉELLE, celui qui fait le vrai travail!

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Java RMI vous donne les objets helpers client et service!

En Java, RMI construit les objets helper client et service à votre place et sait même comment faire pour que le helper client ressemble au vrai service. Autrement dit, RMI sait comment donner à l'objet helper client les méthodes que vous voulez appeler sur le service distant.

Plus encore, RMI fournit toute l'infrastructure d'exécution, y compris un service de recherche qui permet au client de trouver le helper client (le proxy du service réel) et d'y accéder.

Avec RMI, vous n'écrivez vous-même aucun code pour le réseau ou les E/S. Le client appelle les méthodes distantes (celles du service réel) exactement comme il appellerait normalement des méthodes sur des objets s'exécutant sur la JVM locale du client.

Enfin presque.

Il existe une différence entre les appels RMI et les appels de méthodes en local (normaux). N'oubliez pas que même si le client a l'impression que l'appel de méthode est local, le helper client transmet cet appel sur le réseau. Il faut donc des méthodes pour le réseau et les E/S. Et que savons-nous des méthodes pour le réseau et les E/S ?

Elles sont risquées !

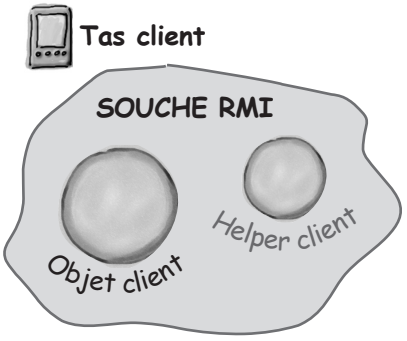
Elles lancent des tonnes d'exceptions.

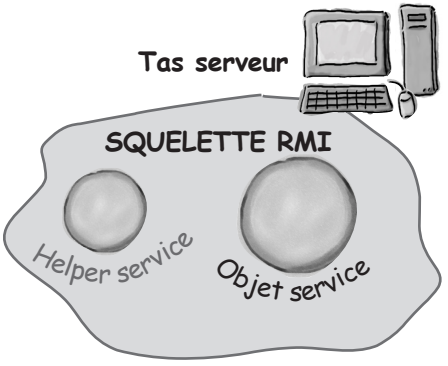
Le client doit donc reconnaître le risque. Quand il appelle une méthode distante, même si ce n'est pour lui qu'un appel local à l'objet proxy, il doit reconnaître que cet appel implique au bout du compte des sockets et des flots. L'appel d'origine est local, mais le proxy le transforme en appel *distant*. Un appel distant signifie simplement qu'une méthode est invoquée sur un objet qui s'exécute sur une autre JVM. La façon dont les informations sur l'appel sont transférées d'une JVM à l'autre dépend du protocole utilisé par les proxys.

RMI vous donne le choix entre deux protocoles : JRMP ou IIOP. JRMP est le protocole « natif » de RMI, celui qui a été conçu pour les appels distants Java-à-Java. En revanche, IIOP est le protocole de CORBA (Common Object Request Broker Architecture) et autorise des appels distants entre des entités qui ne sont pas forcément des objets Java. CORBA est généralement beaucoup plus difficile à mettre en œuvre que RMI : si l'autre extrémité n'est pas écrite en Java, il faut des quantités pharamineuses de traductions et de conversions.

Heureusement, seul Java-à-Java nous intéresse ici, et nous nous en tiendrons à ce bon vieux RMI remarquablement facile.

Pour RMI, le helper client est une « souche » et le helper serveur un « squelette ».





Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

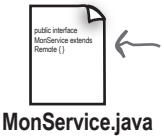
Créer le service distant

Voici une **vue d'ensemble** des cinq étapes de la création d'un service distant (qui s'exécute sur le serveur). Ne vous inquiétez pas, chacune d'elles est expliquée en détail dans les pages qui suivent.

Étape un :

Créer une **Interface distante**

L'interface distante définit les méthodes qu'un client peut appeler à distance. C'est elle que le client utilisera comme type polymorphe pour votre service. À la fois la souche et le service réel l'implémenteront !



Cette interface définit les méthodes distantes que vous voulez que les clients appellent.

Étape deux :

Créer une **Implémentation distante**

C'est la classe qui exécute le vrai travail. Elle contient l'implémentation réelle des méthodes définies dans l'interface distante. C'est l'objet sur lequel le client va appeler les méthodes.

Le service réel. La classe qui contient les méthodes qui font le vrai travail. Elle implémente l'interface distante.

Étape trois :

Générer les **souches** et les **squelettes** avec rmic

Ce sont les « helpers » client et serveur. Vous n'avez pas besoin de créer ces classes ni même de regarder le code source qui les génère. Tout est géré automatiquement quand vous exécutez l'outil rmic qui accompagne votre kit de développement Java.

L'exécution de rmic avec le nom de la classe qui implémente le service...

```
Fichier Edition Fenêtre Aide Manger
%rmic MonServiceImpl
```

... génère deux nouvelles classes.

Étape quatre :

Lancer le **registre RMI** (rmiregistry)

Le registre est comparable aux pages blanches d'un annuaire téléphonique. C'est là que l'utilisateur va chercher le proxy (l'objet souche client).

```
Fichier Edition Fenêtre Aide Boire
%rmiregistry
```

Exécutez cette commande dans une fenêtre d'exécution séparée.

Étape cinq :

Lancer le **service distant**

Vous devez mettre en route et exécuter l'objet service. Votre classe d'implémentation du service crée une instance du service et l'enregistre dans le registre RMI. C'est ce qui permet aux clients d'y accéder.

```
Fichier Edition Fenêtre Aide FaireLaFete
%java MonServiceImpl
```


Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Étape un : créer une interface distante

MonService.java

1 Étendre java.rmi.Remote

Remote est une interface «marqueur», ce qui veut dire qu'elle n'a pas de méthodes. Mais comme elle a une signification spéciale pour RMI, vous devez appliquer cette règle. Remarquez que nous avons écrit «extends». Une interface a le droit d'étendre une autre interface.

```
public interface MonService extends Remote {
```

Votre interface doit annoncer qu'elle sert aux appels de méthodes distantes. Une interface ne peut rien implémenter, mais elle peut étendre d'autres interfaces.

2 Déclarer que toutes les méthodes lancent une RemoteException

L'interface distante est celle que le client utilise comme type polymorphe pour le service. Autrement dit, le client invoque les méthodes sur une entité qui implémente l'interface distante. Cette entité est bien sûr la souche, et comme la souche s'occupe du réseau et des E/S, toutes sortes de catastrophes peuvent survenir. Le client doit reconnaître l'existence de ces risques en gérant ou en déclarant les exceptions distantes. Si les méthodes d'une interface déclarent des exceptions, tout code appelant des méthodes sur une référence de ce type (le type de l'interface) doit gérer ou déclarer les exceptions.

```
import java.rmi.*;

public interface MonService extends Remote {
    public String direBonjour() throws RemoteException;
}
```

L'interface Remote est dans java.rmi

Chaque appel de méthode distante est considéré comme 'risqué'. Déclarer RemoteException dans chaque méthode oblige le client à faire attention et à reconnaître qu'un incident peut se produire.

3 Vérifier qu'arguments et valeurs de retour sont de type primitif ou sérialisables

Les arguments et les valeurs de retour d'une méthode distante doivent être soit de type primitif soit sérialisables. Tout argument d'une méthode distante doit être encodé et transmis sur le réseau, et c'est ce que fait la sérialisation. Pareil pour les valeurs de retour. Si vous employez des types primitifs, des Strings et la majorité des types de l'API (y compris les tableaux et les collections), tout va bien. Si vous utilisez vos propres types, vérifiez simplement que vos classes implémentent Serializable.

```
public String direBonjour() throws RemoteException;
```

Comme le serveur va utiliser le réseau pour renvoyer cette valeur de retour au client, elle doit être sérialisable. C'est de cette façon que les valeurs de retour et les arguments sont encodés et transmis.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Étape deux : créer une implémentation distante

MonServiceImpl.java

1 Implémenter l'interface Remote

Votre service doit implémenter l'interface distante — celle qui contient les méthodes que votre client va appeler.

```
public class MonServiceImpl extends UnicastRemoteObject implements MonService {
    public String direBonjour() {
        return "Le serveur dit 'ohé'";
    }
    // reste du code
}
```

Le compilateur vérifiera que vous avez implémenté toutes les méthodes de l'interface que vous implémentez. Dans ce cas, il n'y en a qu'une.

2 Étendre UnicastRemoteObject

Pour pouvoir fonctionner comme service distant, votre objet doit avoir une fonctionnalité associée au fait d'« être distant ». La façon la plus simple consiste à étendre UnicastRemoteObject (qui se trouve dans le package java.rmi.server) et de laisser cette classe (votre superclasse) faire le travail à votre place.

```
public class MonServiceImpl extends UnicastRemoteObject implements MonService
{
```

3 Écrire un constructeur sans argument qui déclare une RemoteException

Votre nouvelle superclasse, UnicastRemoteObject, a un petit problème : son constructeur lance une RemoteException. La seule façon de le résoudre est de déclarer un constructeur pour votre implémentation distante, afin de disposer d'un endroit pour déclarer la RemoteException. Souvenez-vous : quand une classe est instanciée, le constructeur de sa superclasse est toujours appelé. Si le constructeur de votre superclasse lance une exception, vous n'avez pas le choix : vous devez déclarer que votre constructeur lance aussi une exception.

```
public MonServiceImpl() throws RemoteException { }
```

Vous n'avez pas besoin de passer quoi que ce soit au constructeur. Il sert simplement à déclarer que le constructeur de la superclasse lance une exception.

4 Enregistrer le service dans le registre RMI

Maintenant que vous avez un service distant, vous devez le mettre à la disposition des clients. Pour ce faire, vous l'instanciez et vous le placez dans le registre (qui doit s'exécuter sinon cette ligne de code échoue). Quand vous enregistrez l'implémentation, c'est en réalité la souche que RMI place dans le registre, puisque c'est d'elle dont le client a vraiment besoin. Enregistrez votre service en appelant la méthode statique rebind() de la classe java.rmi.Naming.

```
try {
    MonService service = new MonServiceImpl();
    Naming.rebind("Bonjour distant", service);
} catch(Exception ex) {...}
```

Donnez à votre service un nom (qui permettra aux clients de chercher dans le registre) et enregistrez-le. Quand vous liez l'objet service, RMI permute le service et la souche et place celle-ci dans le registre.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Étape trois : générer des souches et des squelettes

1 Exécuter rmic avec le nom de la classe de l'implémentation distante (pas de l'interface)

L'outil rmic, qui fait partie du SDK Java, lit l'implémentation du service et crée deux nouvelles classes, la souche et le squelette. Par convention, il leur donne le nom de l'implémentation distante, en ajoutant _Stub ou _Skeleton à la fin. rmic offre d'autres possibilités, notamment de ne pas générer de squelette, d'examiner le code source de ces classes et même d'employer IIOP comme protocole. Nous procédons ici comme on le fait généralement. Les classes se retrouveront dans le répertoire courant (celui dans lequel vous vous êtes positionné avec cd). Souvenez-vous que rmic doit pouvoir voir la classe de l'implémentation. Vous exécuterez donc probablement rmic depuis le répertoire dans lequel l'implémentation distante se trouve. (Pour simplifier, nous avons délibérément exclu ici l'emploi de packages. Dans le monde réel, vous devriez tenir compte des arborescences de répertoire et des noms pleinement qualifiés.)

Remarquez qu'on n'utilise pas l'extension .class, seulement le nom de la classe.

```
Fichier Edition Fenêtre Aide Comment?
%rmic MonServiceImpl
```

rmic génère deux nouvelles classes.

MonServiceImpl_Stub.class

MonServiceImpl_Skel.class

Étape quatre : exécuter rmiregistry

1 Ouvrir une fenêtre et lancer rmiregistry

Vérifiez que vous partez bien d'un répertoire duquel vous avez accès à vos classes. La façon la plus simple de procéder consiste à partir du répertoire « classes ».

```
Fichier Edition Fenêtre Aide Comment?
%rmiregistry
```

Étape cinq : démarrer le service

1 Ouvrir une autre fenêtre et lancer le service

Vous pouvez le lancer depuis la méthode main() de l'implémentation distante ou d'une classe « lanceur » séparée. Dans ce simple exemple, nous avons placé le code de démarrage dans la classe de l'implémentation, dans une méthode main() qui instancie l'objet et l'enregistre dans le registre RMI.

```
Fichier Edition Fenêtre Aide Comment?
%java MonServiceImpl
```


Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Code complet côté serveur

L'interface distante :

```
import java.rmi.*;

public interface MonService extends Remote {

    public String direBonjour() throws RemoteException;
}
```

RemoteException et Remote sont dans le package java.rmi.

Votre interface DOIT étendre java.rmi.Remote

Toutes vos méthodes distantes doivent déclarer une RemoteException.

Le service distant (l'implémentation) :

```
import java.rmi.*;
import java.rmi.server.*;

public class MonServiceImpl extends UnicastRemoteObject implements MonService {

   public String direBonjour() {
       return "Le serveur dit 'ohé'";
   }

   public MonServiceImpl() throws RemoteException { }

   public static void main (String[] args) {

       try {
           MonService service = new MonServiceImpl();
           Naming.rebind("Bonjour distant", service);
       } catch(Exception ex) {
           ex.printStackTrace();
       }
   }
}
```

UnicastRemoteObject est dans le package java.rmi.server.

étendre UnicastRemoteObject est la façon la plus facile de créer un objet distant.

Vous DEVEZ implémenter votre interface distante !!

Vous devez bien sûr implémenter toutes les méthodes de l'interface. Mais remarquez que vous n'avez PAS à déclarer la RemoteException.

Le constructeur de la superclasse (pour UnicastRemoteObject) déclarant une exception, VOUS devez écrire un constructeur, parce que cela signifie que votre constructeur appelle un code risqué (son superconstructeur).

Créer l'objet distant, puis le « lier » au registre RMi grâce à la méthode statique Naming.rebind(). Le nom sous lequel vous l'enregistrez est celui que les clients devront chercher dans le registre.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Comment le client accède-t-il à l'objet souche ?

Le client doit accéder à l'objet souche, puisque c'est sur cette entité qu'il va appeler des méthodes. C'est là que le registre RMI entre en scène. Le client effectue une « recherche », comme s'il consultait les pages blanches de l'annuaire du téléphone, et dit en substance « Voici un nom et je voudrais la souche qui va avec ce nom ».

lookup() est une méthode statique de la classe Naming.

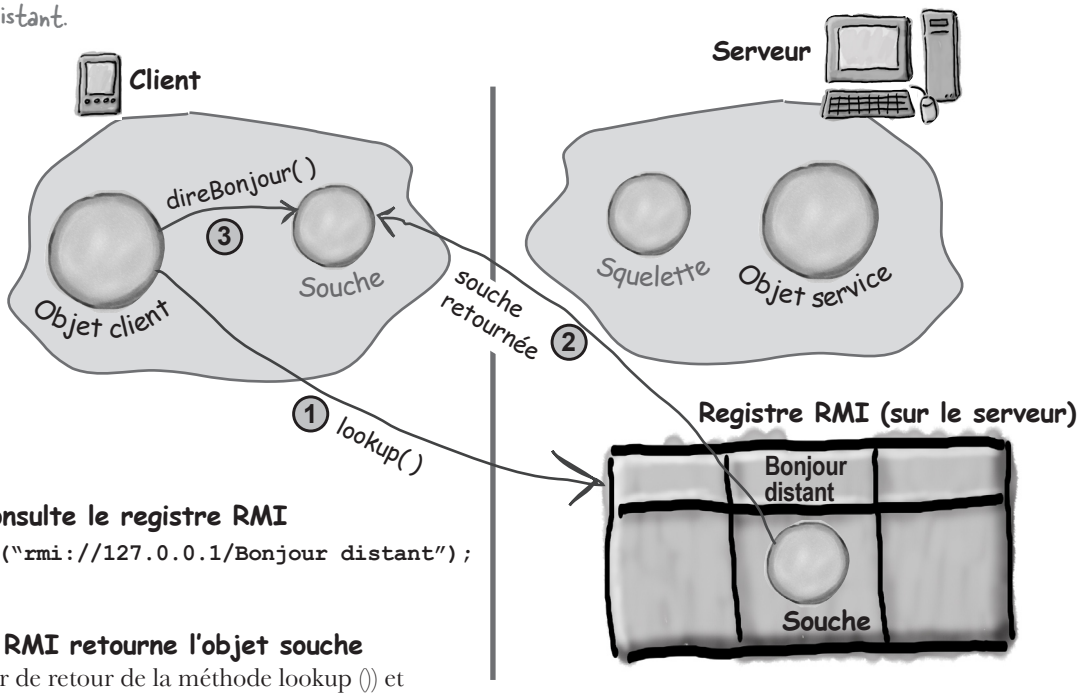
Ici le nom sous lequel le service a été enregistré.

```
MonService service = (MonService) Naming.lookup("rmi://127.0.0.1/Bonjour distant");
```

Le client utilise toujours l'implémentation distante comme type du service. De fait, le client n'a jamais besoin de connaître le vrai nom de classe du service distant.

Vous devez transtyper, puisque la méthode lookup() retourne un objet de type Object.

Ici votre nom de machine ou votre adresse IP.



1 Le client consulte le registre RMI

```
Naming.lookup("rmi://127.0.0.1/Bonjour distant");
```

2 Le registre RMI retourne l'objet souche
(comme valeur de retour de la méthode lookup ()) et RMI le désérialise automatiquement. La classe de la souche (que rmic a générée) DOIT être sur le client ou la souche ne sera pas désérialisée.

3 Le client invoque une méthode sur la souche comme si elle ÉTAIT le service réel

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Comment le client a-t-il obtenu la souche?

Et maintenant, voilà la question intéressante. D'une manière ou d'une autre, le client doit avoir la classe de la souche (que vous avez générée auparavant avec rmic) au moment où il effectue la recherche, sinon la souche n'est pas désérialisée sur le client et toute l'opération échoue. Sur un système simple, vous pouvez vous contenter de placer cette classe sur le client.

Mais il existe un moyen beaucoup plus pratique, bien qu'il dépasse la portée de ce livre. Juste au cas où vous seriez intéressé, ce moyen s'appelle le « téléchargement de classe dynamique ». Dans le téléchargement dynamique, l'objet souche (en fait n'importe quel objet sérialisé) est « estampillé » avec une URL qui indique au système RMI du client où trouver le fichier .class de cet objet. Puis, au cours du processus de désérialisation, si RMI ne trouve pas la classe localement, il utilise cette URL et émet un appel HTTP Get pour extraire le fichier. Il vous faut donc un simple serveur web pour servir les fichiers .class et vous devez également modifier certains paramètres de sécurité sur le client. Le téléchargement dynamique implique encore deux ou trois éléments quelque peu délicats, mais c'est là l'idée globale.

Code complet côté client

```
import java.rmi.*;

public class MonClientDistant {
  public static void main (String[] args) {
        new MonClientDistant().go();
   }

  public void go() {

    try {
       MonService service = (MonService) Naming.lookup("rmi://127.0.0.1/Bonjour distant");

        String s = service.direBonjour();

        System.out.println(s);
     } catch(Exception ex) {
         ex.printStackTrace();
     }
   }
}
```

La classe Naming (pour la recherche dans le registre) est dans le package java.rmi.

L'objet retourné est de type Object: n'oubliez pas le transtypage.

Vous avez besoin de l'adresse IP ou du nom de machine.

Et du nom utilisé pour « lien » le service.

Cela a tout l'air d'un bon vieil appel de méthode ordinaire! (Sauf qu'il doit reconnaître la RemoteException).

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

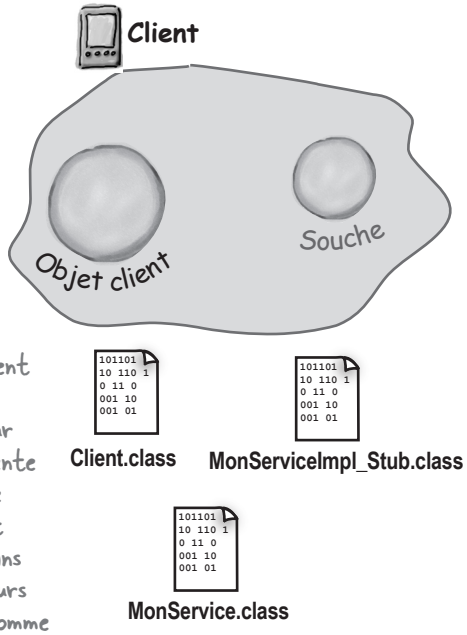
Vérifiez que chaque machine dispose des classes dont elle a besoin

Les trois principales erreurs que les programmeurs commettent avec RMI sont les suivantes :

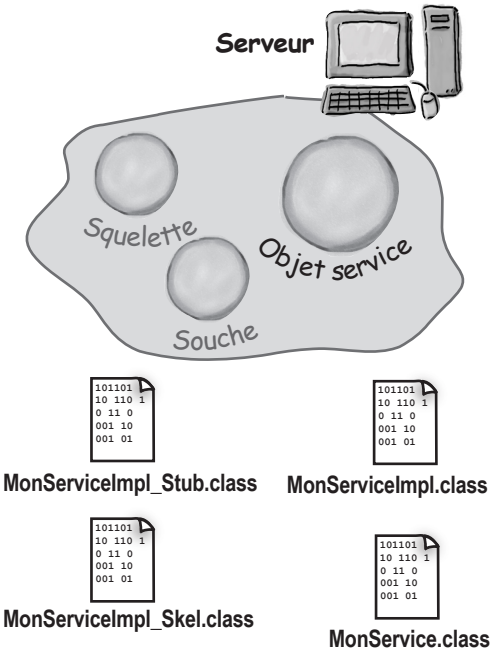
1) Oublier de lancer rmiregistry avant de démarrer le service distant (quand vous enregistrez le service en appelant Naming.rebind(), rmiregistry doit être en train de s'exécuter !).

2) Oublier de rendre les arguments et les types de retour sérialisables (l'erreur n'est pas détectée par le compilateur et vous ne vous en rendrez pas compte avant l'exécution).

3) Oublier de communiquer la classe de la souche au client.



N'oubliez pas que le client utilise l'interface pour appeler les méthodes sur la souche. La JVM cliente a besoin de la classe de la souche, mais le client ne s'y réfère jamais dans le code. Il utilise toujours l'interface distante, comme si cette dernière ÉTAIT l'objet distant réel.



Le serveur a besoin à la fois des classes de la souche et du squelette, ainsi que celles du service et de l'interface distante. La classe de la souche est nécessaire : n'oubliez pas que la souche est substituée au service réel quand celui-ci est lié au registre RMI.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

À vos crayons

L'œuf et la poule

Regardez la séquence d'événements ci-dessous, et remettez-les dans l'ordre dans lequel ils se produisent dans une application Java RMI.

1.

2.

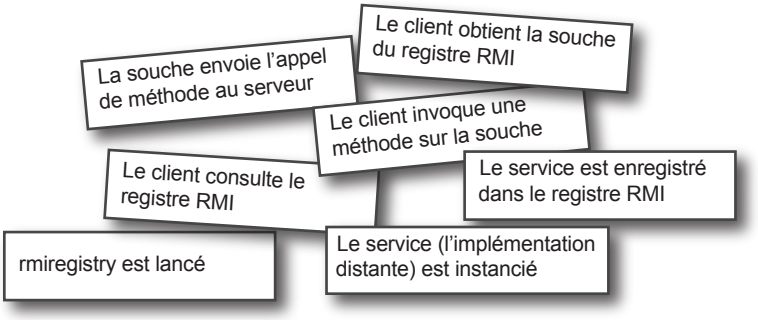
3.

4.

5.

6.

7.

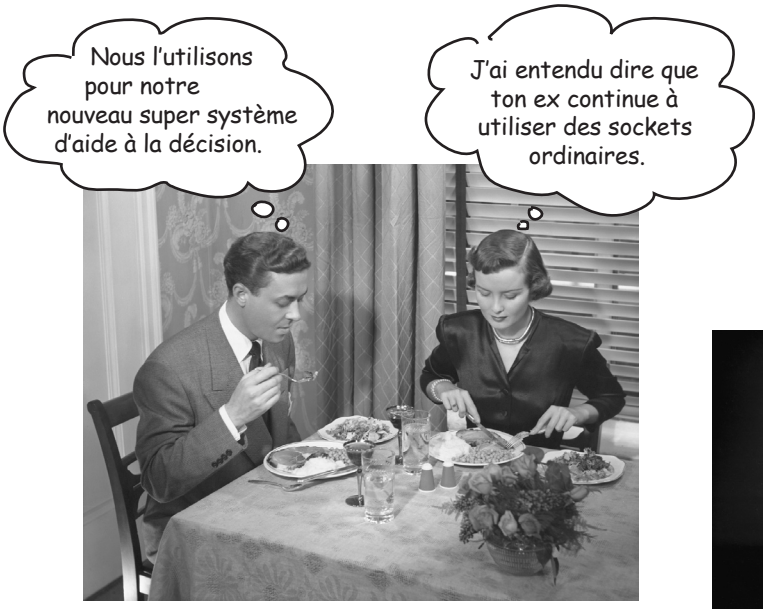


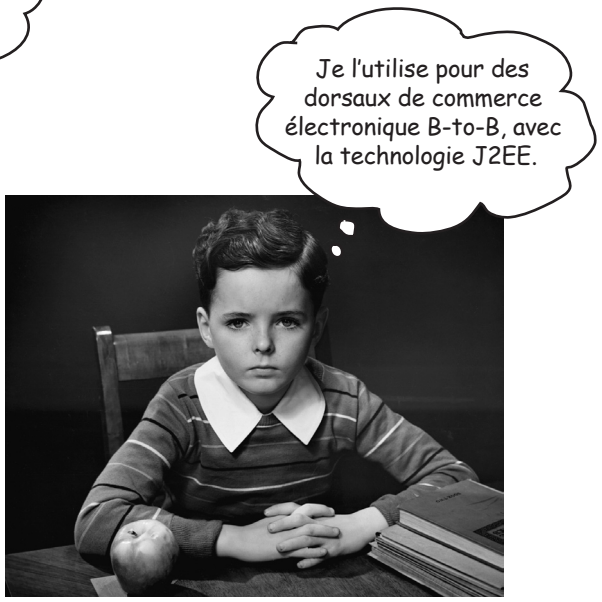
POINTS D'IMPACT

- Un objet résidant sur un tas ne peut pas obtenir une référence normale à un objet résidant sur un tas différent (s'exécutant sur une JVM différente).
- RMI (*Remote Method Invocation*) donne l'impression que vous appelez une méthode sur un objet distant (s'exécutant sur une autre JVM), mais ce n'est pas le cas.
- Quand un client appelle une méthode sur un objet distant, il appelle en réalité la méthode sur un proxy de l'objet distant. On nomme ce proxy une « souche ».
- Une souche est un objet helper côté client qui se charge des détails réseau de bas niveau (sockets, flots, sérialisation, etc.) en encodant et transmettant les appels de méthode au serveur.
- Pour construire un service distant (autrement dit un objet sur lequel un client distant pourra en fin de compte appeler des méthodes), vous devez commencer par une interface distante.
- Une interface distante doit étendre l'interface java.rmi.Remote et toutes les méthodes doivent déclarer une RemoteException.
- Votre service distant implémente l'interface distante.
- Votre service distant doit étendre UnicastRemoteObject. (Il existe d'autres techniques pour créer un objet distant, mais étendre UnicastRemoteObject est la plus simple).
- La classe de votre service distant doit avoir un constructeur qui déclare une RemoteException (parce que le constructeur de la superclasse en déclare une).
- Votre service distant doit être instancié et l'objet enregistré dans le registre RMI.
- Pour enregistrer un service distant, utilisez la méthode statique Naming.rebind("NomService", instanceService);
- Le registre RMI doit s'exécuter sur la même machine que le service distant avant que vous n'essayiez d'enregistrer votre objet distant.
- Le client recherche votre service distant via la méthode statique Naming.lookup("rmi://NomMachine/NomService");
- Presque tout ce qui est lié à RMI peut lancer une RemoteException (vérifiée par le compilateur), notamment la recherche ou l'enregistrement d'un service et tous les appels de méthode d'un client à la souche.

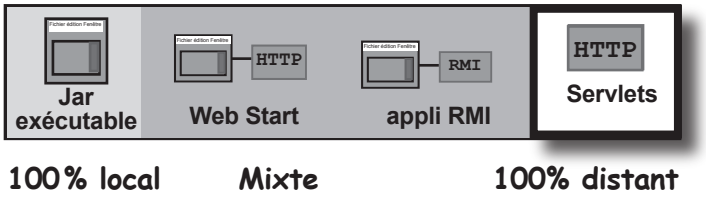
Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Oui, mais qui *utilise* réellement RMI ?





Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>



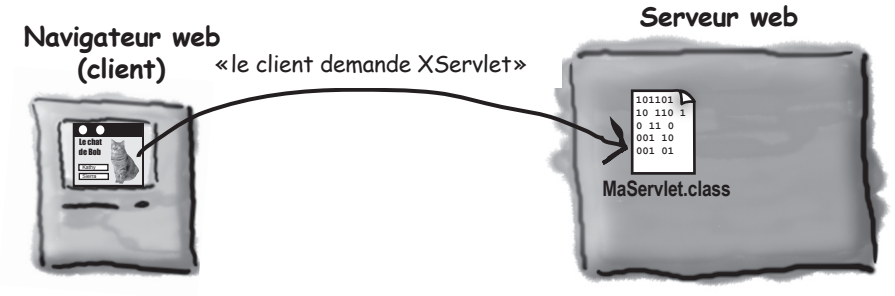
Et les servlets?

Les servlets sont des programmes Java qui s'exécutent sur (et avec) un serveur HTTP. Quand un client utilise un navigateur web pour interagir avec une page web, une requête est renvoyée au serveur web. Si la requête a besoin de l'aide d'une servlet Java, le serveur exécute (ou appelle, si la servlet s'exécute déjà) le code de la servlet. Ce code n'est rien d'autre qu'un programme côté serveur qui exécute la requête du client (par exemple sauvegarder des informations dans un fichier texte ou dans une base de données sur le serveur). Si les scripts CGI écrits en Perl vous sont familiers, vous savez exactement de quoi nous parlons. Les développeurs web emploient des scripts CGI ou des servlets pour faire pratiquement n'importe quoi, de l'envoi d'une information soumise par l'utilisateur à l'exécution d'un forum de discussion d'un site web.

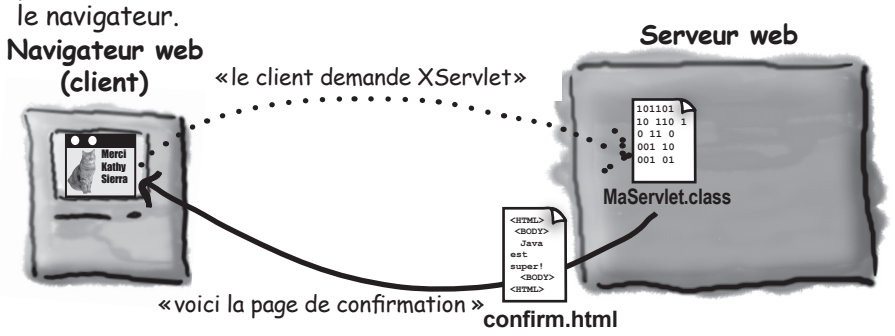
Et même les servlets peuvent utiliser RMI!

L'usage de loin le plus courant de la technologie J2EE consiste à mélanger des servlets et des EJB, les servlets étant les clients de l'EJB. Dans ce cas, *les servlets communiquent avec les EJB via RMI*. (Bien que la façon d'utiliser RMI avec les EJB diffère un peu du processus que nous venons d'étudier.)

1. Le client remplit un formulaire d'enregistrement et clique sur Envoi. Le serveur HTTP (le serveur web) lit la requête, constate qu'elle est destinée à une servlet et transmet la requête à celle-ci.

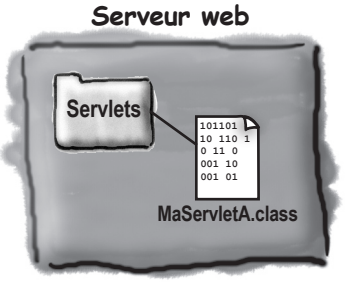


2. La servlet (le code Java) s'exécute, insère les données dans la base, compose une page web (avec des informations personnalisées) et la retourne au client où elle s'affiche dans le navigateur.



Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Étapes pour créer et exécuter une servlet



1 Déterminez où placer vos servlets.

Dans ces exemples, nous supposerons que vous avez déjà un serveur web, qu'il s'exécute et qu'il est configuré pour prendre en charge des servlets. Le plus important est de déterminer à quel endroit exact placer les fichiers classe des servlets pour que votre serveur les «voie». Si vous avez un site web hébergé par un FAI, le service d'hébergement peut vous indiquer où placer vos servlets, tout comme il vous dira où placer vos scripts CGI.

2 Procurez-vous servlets.jar et ajoutez-le à votre classpath

Les servlets ne faisant pas partie des bibliothèques standard Java, vous avez besoin des classes qui se trouvent dans le fichier servlets.jar. Vous pouvez les télécharger sur le site java.sun.com ou les obtenir sur un serveur web équipé pour Java (comme Apache Tomcat, sur le site apache.org). En l'absence de ces classes, vous ne pourrez pas compiler vos servlets.

3 Écrivez une classe en étendant HttpServlet

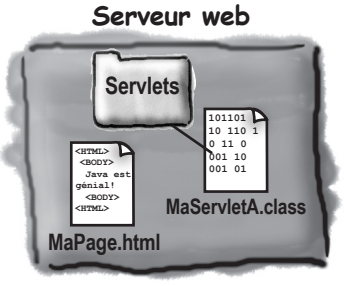
Une servlet n'est rien d'autre qu'une classe Java qui étend HttpServlet (dans le package javax.servlet.http). Il existe d'autres types de servlets, mais c'est HttpServlet qui nous intéressera la plupart du temps.

```
public class MaServletA extends HttpServlet { ... }
```

4 Écrivez une page HTML qui appelle votre servlet

Quand l'utilisateur cliquera sur un lien qui pointe vers votre servlet, le serveur trouvera la servlet et invoquera la méthode appropriée en fonction de la commande HTTP (GET, POST, etc.).

```
<a href="servlets/MaServletA">C'est la plus bath des servlets.</a>
```



5 Mettre la servlet et la page HTML à disposition du serveur

Maintenant, vous pouvez entrer l'adresse sur votre navigateur et appeler la servlet depuis cette page html.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Une servlet très simple

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class MaServletA extends HttpServlet {

    public void doGet (HttpServletRequest request, HttpServletResponse response)
                                    throws  ServletException, IOException  {

        response.setContentType("text/html");

        PrintWriter out = response.getWriter();

        String message = "Si vous lisez ceci, ça a marché!";

        out.println("<HTML><BODY>");
        out.println("<H1>" + message + "</H1>");
        out.println("</BODY></HTML>");
        out.close();
    }
}
```

Outre io, nous devons importer deux des packages de servlets. N'oubliez pas qu'ils ne font PAS partie des bibliothèques standard Java et que vous devez les télécharger.

La plupart des servlets "normales" étendront HttpServlet, puis redéfiniront une ou plusieurs méthodes.

Redéfinir la méthode doGet pour les simples messages HTTP GET.

Le serveur web appelle cette méthode, vous passe la requête du client (vous pouvez en extraire des données) et un objet qui vous servira à envoyer la réponse au client (une page).

Ceci indique au navigateur quel type de contenu repartira du serveur.

L'objet response fournit un flot de sortie qui permet d'écrire les informations sur le serveur.

Ce que nous écrivons ici est une page HTML! La page est transmise via le serveur au navigateur, tout comme n'importe quelle page HTML, même si elle n'a jamais existé auparavant. Autrement dit, il n'existe nulle part de fichier qui contient ces lignes.

Page HTML avec un lien vers cette servlet

```
<HTML>
  <BODY>
     <a href="servlets/MaServletA">C'est la plus bath des servlets.</a>
  </BODY>
</HTML>
```

Cliquez sur le lien pour déclencher la servlet.

L'aspect de la nouvelle page web :

C'est la plus bath des servlets

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

POINTS D'IMPACT

- Les servlets sont des classes Java qui s'exécutent entièrement sur un serveur HTTP (web).
- Les servlets servent à exécuter sur le serveur du code qui répond à une interaction du client avec une page web. Par exemple, si un client soumet des informations dans un formulaire, la servlet peut les traiter, les insérer dans une base de données et renvoyer au client une page de confirmation personnalisée.
- Pour compiler une servlet, il vous faut les packages contenus dans le fichier servlets.jar. Ces classes ne faisant pas partie des bibliothèques standard Java, vous devez télécharger servlets.jar sur le site java.sun.com ou les obtenir d'un serveur web prenant en charge les servlets. (La bibliothèque Servlet et comprise dans JZEE.)
- Pour exécuter une servlet, vous devez disposer d'un serveur web compatible, par exemple le serveur Tomcat d'apache.org.
- Votre servlet doit être placée à un endroit spécifique de votre serveur web, et vous devez le déterminer avant d'essayer de l'exécuter. Si votre site web est hébergé par un FAI qui prend en charge les servlets, celui-ci vous indiquera dans quel répertoire les placer.
- Une servlet type étend HttpServlet et en redéfinit une ou plusieurs méthodes, comme doGet() et doPost().
- Le serveur web lance la servlet et appelle la méthode appropriée (doGet(), etc.) en fonction de la requête du client.
- La servlet peut renvoyer une réponse en obtenant un flot de sortie PrintWriter du paramètre response de la méthode doGet().
- La servlet « écrit » une page HTML complète avec des balises.

il n'y a pas de

Questions stupides

Q : Qu'est-ce que JSP, et quel est le rapport avec les servlets ?

R : JSP est l'acronyme de Java Server Pages. À la fin, le serveur web transforme une page JSP en servlet, mais la différence entre une servlet et une page JSP est ce que VOUS (le développeur) allez créer. Avec une servlet, vous écrivez une *classe* Java qui contient du code *HTML* dans les instructions de sortie (si vous retournez une page HTML au client). Avec une JSP, c'est l'inverse —vous écrivez une page *HTML* qui contient du *code* Java !

Cette technologie vous permet d'avoir des pages web dynamiques : vous écrivez une page HTML normale, mais vous y encapsulez du code Java (et d'autres balises qui « déclenchent » le code Java) qui est traité au moment de l'exécution. Autrement dit, une partie de la page est personnalisée à l'exécution quand le code Java s'exécute.

Le principal avantage de JSP sur les servlets ordinaires est qu'il est beaucoup plus facile d'écrire la partie HTML d'une servlet sous forme de page JSP que d'insérer du HTML dans la réponse de la servlet. Représentez-vous une page HTML raisonnablement complexe, puis imaginez que vous la formatez avec des instructions println. La torture !

Mais les JSP sont inutiles dans bien des applications, parce que la servlet ne renvoie pas de réponse dynamique, ou que la page HTML est suffisamment simple. Et si votre serveur web prend en charge les servlets mais pas JSP, vous êtes coincé.

Un autre avantage de JSP est que vous pouvez fractionner le travail entre les programmeurs Java qui écrivent les servlets et les développeurs web qui écrivent les pages JSP. Du moins en théorie. En pratique, toute personne écrivant des JSP devra quand même apprendre suffisamment de Java (outre les balises). Il est donc irréaliste de penser qu'un concepteur de pages HTML va se mettre à pondre par magie des JSP. Par bonheur, on commence à trouver des outils qui permettent aux concepteurs de pages web de créer des JSP sans devoir écrire le code *ex nihilo*.

Q : C'est tout ce que vous allez dire sur les servlets ? Après tout ce discours sur RMI ?

R : Oui. RMI fait partie du langage, Java et toutes les classes sont dans les bibliothèques standard. Les servlets et JSP ne font *pas* partie du langage. Elles sont considérées comme des *extensions standard*. Vous pouvez exécuter RMI sur toute JVM récente, mais les servlets et JSP nécessitent un serveur web configuré *ad hoc* avec un « conteneur » de servlets. C'est notre façon de dire que ces sujets dépassent la portée de ce livre. Mais vous saurez tout sur les servlets et JSP en lisant *Servlets Java – guide du programmeur* et *JavaServer Pages*.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Rien que pour le plaisir, transformons le Phrase-O-Matic en servlet

Maintenant que nous vous avons dit que nous ne parlerions plus des servlets, nous ne résistons pas au plaisir de servletiser (oui, on *peut* en faire un verbe) le Phrase-O-Matic du chapitre 1. Une servlet n'est rien d'autre que du code Java. Et un code Java peut appeler un autre code Java. Une servlet est donc libre d'appeler une méthode du Phrase-O-Matic. Il suffit de déposer la classe Phrase-O-Matic dans le même répertoire que votre servlet, et vous voilà parti. (Le code du Phrase-O-Matic se trouve à la page suivante.)

```
import java.io.*;

import javax.servlet.*;
import javax.servlet.http.*;

public class KathyServlet extends HttpServlet {
  public void doGet (HttpServletRequest request, HttpServletResponse response)
                                          throws ServletException, IOException  {

        String titre = "PhraseOMatic a généré la phrase suivante.";

        response.setContentType("text/html");
        PrintWriter out = response.getWriter();

        out.println("<HTML><HEAD><TITLE>");
        out.println("PhraseOmatic");
        out.println("</TITLE></HEAD><BODY>");
        out.println("<H1>" + titre + "</H1>");
        out.println("<P>" + PhraseOMatic.creerPhrase());
        out.println("<P><a href=\"KathyServlet\">générer une autre phrase</a></p>");
        out.println("</BODY></HTML>");

        out.close();
    }
}
```

Vous voyez? Votre servlet peut appeler les méthodes d'une autre classe. En l'occurrence, nous appelons la méthode statique creerPhrase() de la classe PhraseOMatic (page suivante).

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Phrase-O-Matic, façon servlet

Voici une version qui diffère légèrement du code du chapitre 1. Dans l'original, tout se passait dans la méthode main() et il fallait réexécuter chaque fois le programme pour générer une nouvelle phrase en mode ligne de commande. Dans cette version, le code retourne simplement une chaîne de caractères (la phrase) quand on appelle la méthode statique creerPhrase(). Cela permet d'appeler la méthode depuis n'importe quel autre code et de récupérer une chaîne qui contient la phrase aléatoire.

Notez que ces longues affectations des tableaux String[] sont victimes du traitement de texte — ne tapez pas les tirets ! Contentez-vous de saisir le code et laissez votre éditeur de texte aller à la ligne. Et, quoi qu'il arrive, n'appuyez jamais sur la touche Entrée au milieu d'une chaîne (autrement dit entre les guillemets).

```
public class PhraseOMatic {
   public static String creerPhrase() {

     // construire trois listes de mots
   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"};

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

    // générer trois nombres aléatoires pour extraire un mot de chaque liste
   int rand1 = (int) (Math.random() * longueurListeUn);
   int rand2 = (int) (Math.random() * longueurListeDeux);
   int rand3 = (int) (Math.random() * longueurListeTrois);

    // construire une phrase
    String phrase = listeUn[rand1] + " " + listeDeux[rand2] + " " + listeTrois[rand3];

    // et l'afficher
    return ("Il nous faut une " + phrase);
  }
}
```


Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Enterprise JavaBeans : RMI gonflé aux stéroïdes

RMI est extra pour écrire et exécuter des services distants. Mais la technologie n'est pas suffisante à elle seule pour des sites du type Amazon ou eBay. Pour une grosse application d'entreprise mortellement sérieuse, il vous faut quelque chose de plus. Quelque chose qui soit capable de gérer les transactions, les problèmes d'accès concurrents (quand un gazillion de personnes visitent votre site en même temps pour acheter vos croquettes pour chat biologiques), la sécurité (pour que personne ne puisse accéder à votre programme de paie) et les bases de données. Pour ce faire, il vous faut un *serveur d'applications d'entreprise*.

En Java, cela signifie un serveur J2EE (*Java 2 Enterprise Edition*). Un serveur J2EE comprend un serveur web et un serveur EJB (*Enterprise JavaBeans*), pour que vous puissiez développer une application qui contienne à la fois des servlets et des EJB. Comme les servlets, EJB dépasse de loin la portée de cet ouvrage et il n'y a aucun moyen de vous présenter « juste un petit » exemple de code, mais nous allons jeter un bref coup d'œil sur son fonctionnement. Pour une approche détaillée d'EJB, lisez le guide de formation à la certification (en anglais), *Head First EJB*.)

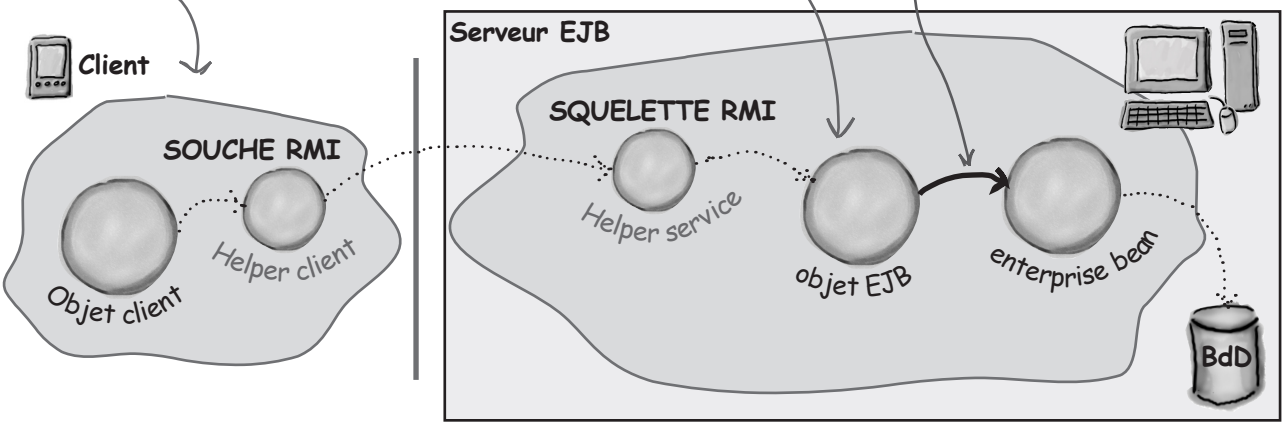
Un serveur EJB ajoute un ensemble de services que RMI n'assure pas : transactions, sécurité, concurrence, gestion des bases de données et réseau.

Un serveur EJB s'interpose au milieu d'un appel RMI et embarque tous les services.

Ce client pourrait être N'IMPORTE QUOI, mais le client EJB type est une servlet s'exécutant sur le même serveur J2EE.

Voilà où le serveur EJB entre en scène ! L'objet EJB intercepte les appels au bean (le bean contient la logique métier) et embarqye tous les services fournis par le serveur EJB (sécurité, transactions, etc.)

L'objet bean n'est pas directement accessible par le client ! Seul le serveur converse réellement avec le bean, ce qui lui permet de dire par exemple "Tiens ! Ce client n'a pas le droit d'appeler cette méthode..." Presque tout ce que vous demandez au serveur EJB se passe ICI, là où le serveur s'interpose.



Ce n'est qu'une petite partie du paysage EJB.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Et pour finir en beauté... un peu de Jini

Nous adorons Jini. Nous pensons que Jini est presque ce qu'il y a de mieux dans Java. Si EJB c'est RMI gonflé aux stéroïdes, Jini c'est RMI avec des ailes. Du pur bonheur Java. Comme pour EJB, nous n'entrerons pas ici dans les détails, mais si vous connaissez RMI vous avez fait les trois quarts du chemin. En termes de technologie en tous cas. En termes d'état d'esprit, il est temps de faire un grand bond. Non, il est temps de s'envoler.

Jini utilise RMI (ainsi qu'éventuellement d'autres protocoles), mais vous offre quelques fonctionnalités capitales, notamment :

La découverte adaptative

Les réseaux autoréparateurs

Avec RMI, souvenez-vous, le client doit connaître le nom et l'emplacement du service distant. Le code qui effectue la recherche contient l'adresse IP ou le nom de machine du service distant (parce que c'est là que le registre RMI s'exécute) *et le nom logique sous lequel le* service a été enregistré.

Mais avec Jini, le client n'a besoin que d'une seule information : *l'**interface** que le service implémente !* C'est tout.

Mais comment trouve-t-on un service ? L'astuce tourne autour des services de recherche de Jini. Ces services sont de loin beaucoup plus puissants et plus souples que le registre RMI. Tout d'abord, ils s'annoncent eux-mêmes sur le réseau automatiquement. Quand un service de recherche arrive en ligne, il émet un message sur le réseau (avec IP multicast) qui dit « Je suis là, si quelqu'un est intéressé ».

Mais ce n'est pas tout. Disons que vous (le client) arrivez en ligne après que le service de recherche s'est déjà annoncé. *Vous pouvez envoyer un* message sur le réseau qui demande « Y a-t-il des services de recherche quelque part ? ».

Sauf que ce n'est pas réellement le service de recherche qui vous intéresse. Ce qui vous intéresse vraiment, ce sont les services qui sont enregistrés. Des services distants RMI, d'autres objets Java sérialisés et même des équipements comme des imprimantes, des caméras et des percolateurs.

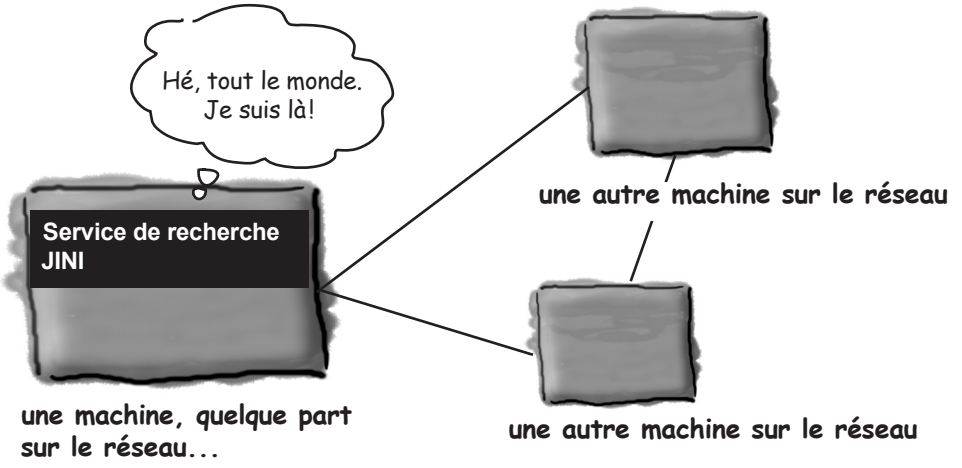
Mais voici le *nec plus ultra :* quand un service arrive en ligne, il découvre dynamiquement tous les services de recherche Jini du réseau et s'enregistre auprès d'eux en envoyant un objet sérialisé qui sera placé dans le service de recherche. Cet objet sérialisé peut être la souche d'un service RMI distant, un pilote de périphérique en réseau ou même le service lui-même qui (une fois que vous l'avez obtenu sur votre recherche) s'exécute localement sur votre machine. Et au lieu de s'enregistrer par son nom, le service fournit celui de l'interface qu'il implémente.

Une fois que vous (le client) avez une référence à un service de recherche, vous pouvez dire à ce service « Hé, est-ce que tu aurais quelque chose qui implémente CalculatriceScientifique ? ». Le service de recherche vérifie alors la liste des interfaces enregistrées. S'il trouve une correspondance, il vous répond « Oui, j'ai quelque chose qui implémente cette interface. Voici l'objet que CalculatriceScientifique a enregistré auprès de moi ».

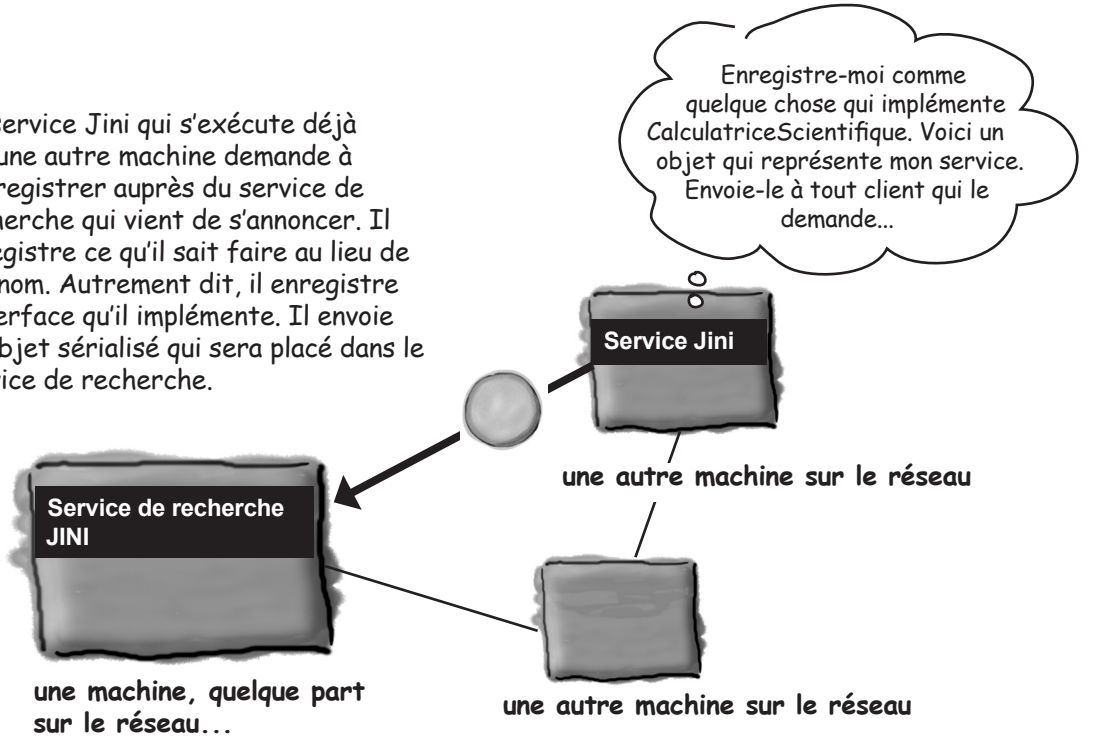
Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

La découverte adaptative en action

1. Le service de recherche Jini est lancé quelque part sur le réseau et s'annonce en utilisant IP multicast.



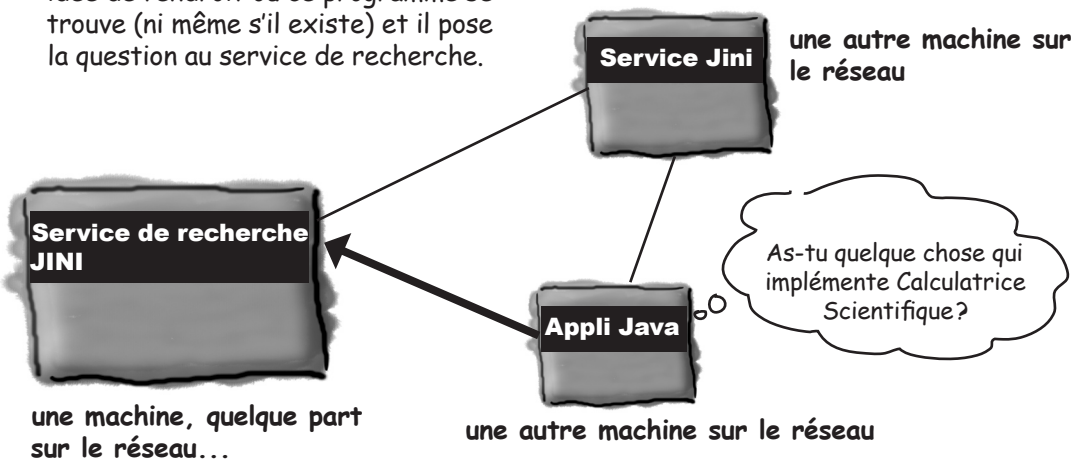
2. Un service Jini qui s'exécute déjà sur une autre machine demande à s'enregistrer auprès du service de recherche qui vient de s'annoncer. Il enregistre ce qu'il sait faire au lieu de son nom. Autrement dit, il enregistre l'interface qu'il implémente. Il envoie un objet sérialisé qui sera placé dans le service de recherche.



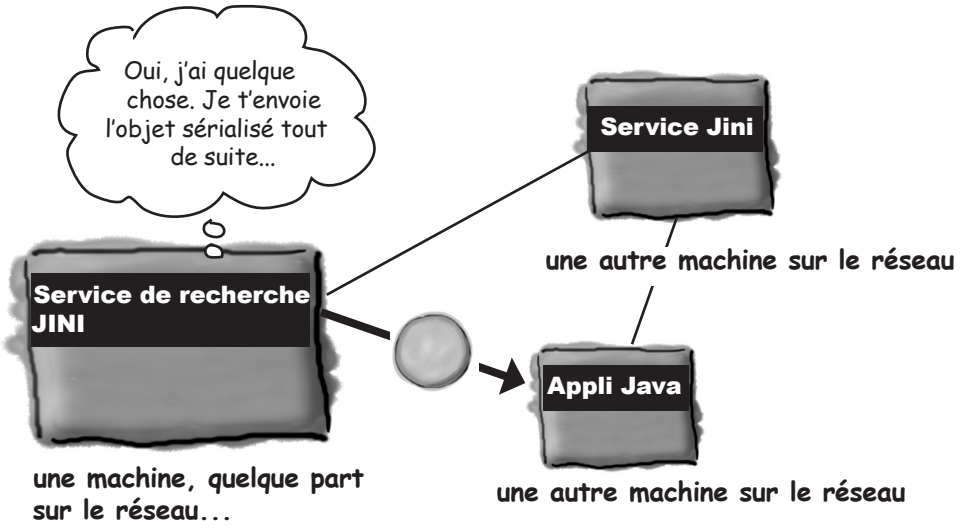
Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Découverte adaptative en action, suite...

3 Un client sur le réseau veut quelque chose qui implémente l'interface CalculatriceScientifique. Il n'a aucune idée de l'endroit où ce programme se trouve (ni même s'il existe) et il pose la question au service de recherche.



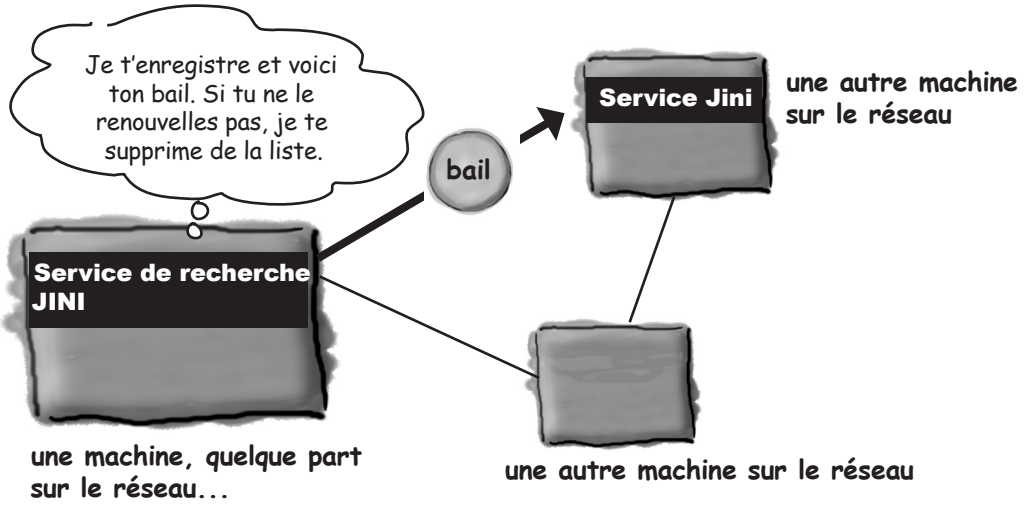
4 Le service de recherche répond, puisqu'il a enregistré une Interface CalculatriceScientifique.



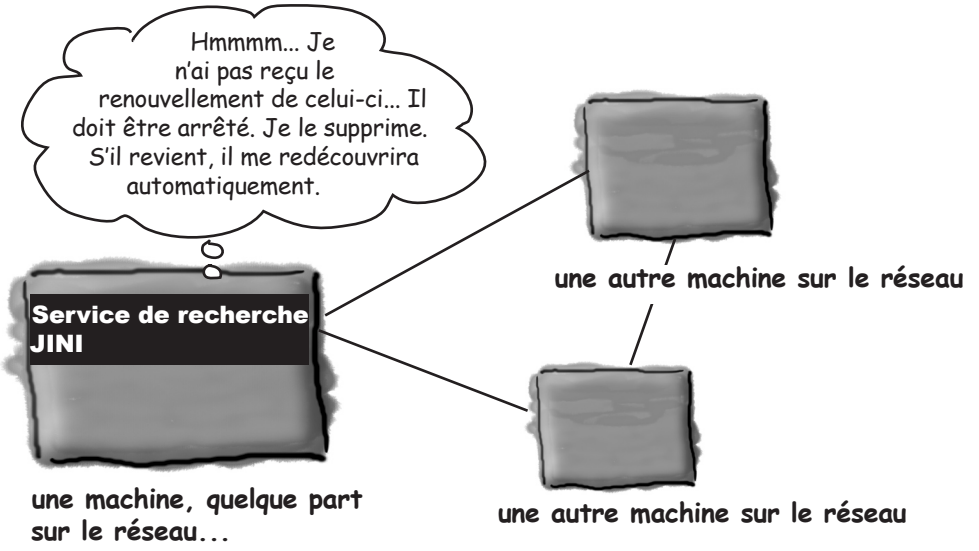
Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Réseau autoréparateur en action

1. Un service Jini a demandé à s'enregistrer auprès du service de recherche. Le service de recherche répond avec un « bail ». Le service nouvellement enregistré doit renouveler sans cesse son bail, sinon le service de recherche suppose que le service est hors ligne. Le service de recherche veut présenter en permanence au reste du réseau une image exacte des services disponibles.



2. Si le service est hors ligne (quelqu'un l'a arrêté), il ne renouvelle pas son bail avec le service de recherche. Ce dernier l'efface de la liste.



Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

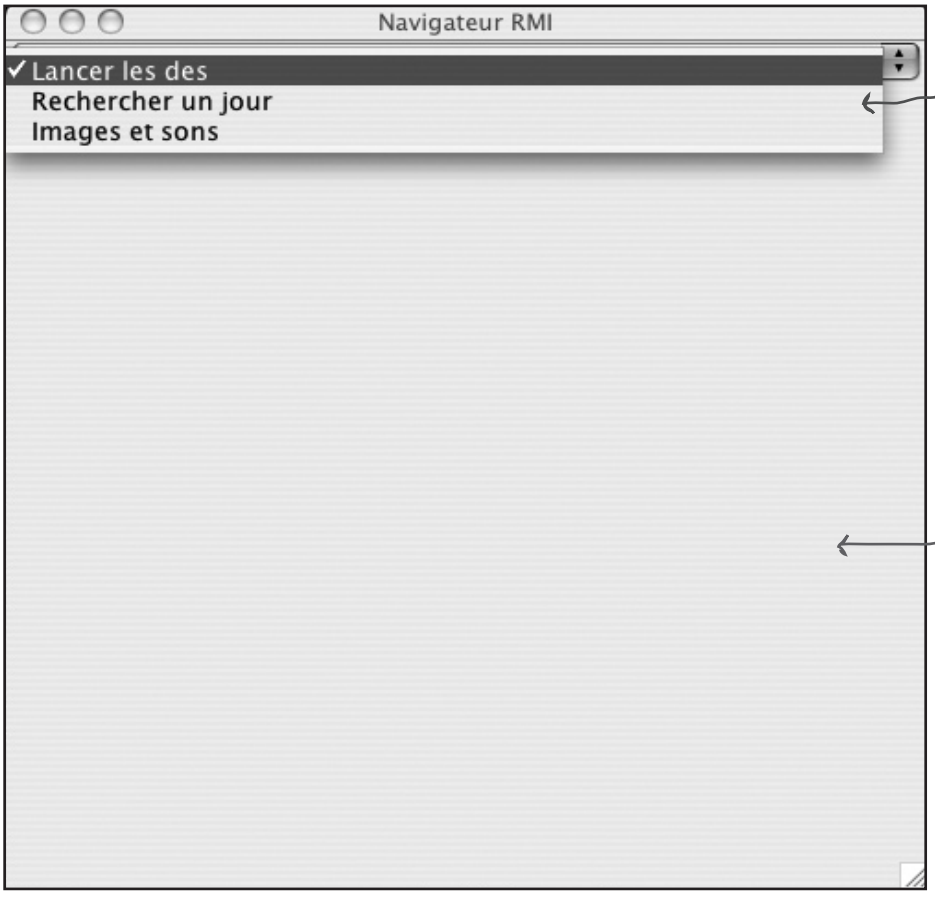
Projet final : le navigateur de services universels

Nous allons créer quelque chose qui n'est pas basé sur Jini mais qui pourrait facilement l'être. Cela vous donnera le parfum de Jini tout en utilisant uniquement RMI. En fait, la principale différence entre notre application et une application Jini réside dans la façon dont on accède au service. Au lieu du service de recherche Jini qui s'annonce automatiquement lui-même et réside n'importe où sur le réseau, nous utilisons le registre RMI qui doit s'exécuter sur la même machine que le service distant et qui ne s'annonce pas automatiquement.

Et au lieu que notre service s'enregistre automatiquement auprès du service de recherche, *nous devons le placer dans le* registre RMI (avec Naming.rebind()).

Mais une fois que le client a trouvé le service dans le registre RMI, le reste de l'application est presque identique à la façon dont nous procédons en Jini. (Il ne manque pratiquement que le mécanisme du « bail » qui nous permettrait d'avoir un réseau autoréparateur au cas où l'un des services tomberait.)

Le navigateur de services universels ressemble à un navigateur web spécialisé, sauf qu'au lieu de pages HTML il télécharge et affiche des IHM Java interactives que nous appelons *services universels*.



Choisissez un service dans la liste. Le service distant RMI possède une méthode getListeServices() qui retourne cette liste.

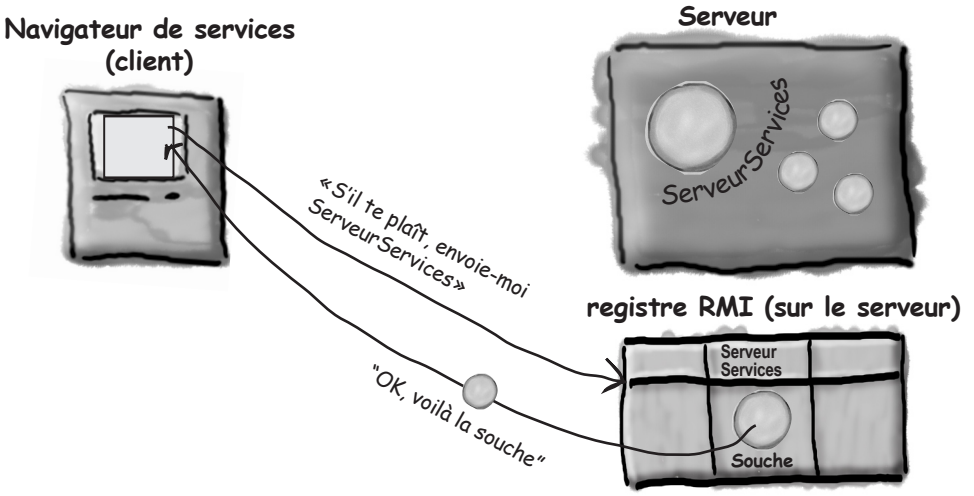
Quand l'utilisateur en sélectionne un, le client demande au service RMI distant de lui afficher le service réel (LanceDes, QuelJour, etc.).

Quand on sélectionne un service, il s'affiche ici.

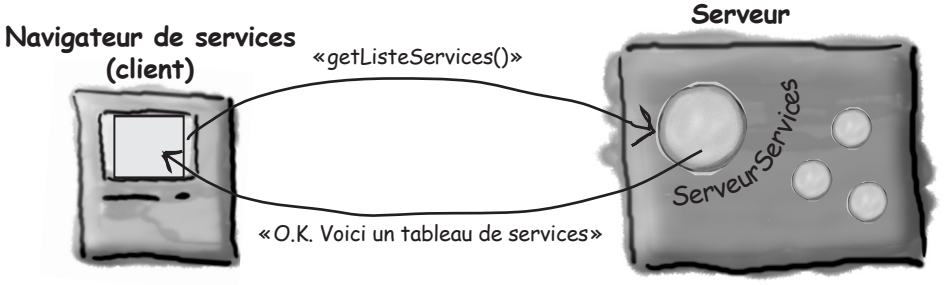
Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Comment ça marche :

1. Le client démarre, cherche dans le registre RMI le service nommé ServeurServices et récupère la souche.



2. Le lient appelle getListeServices() sur la souche. ServeurServices retourne un tableau de services.

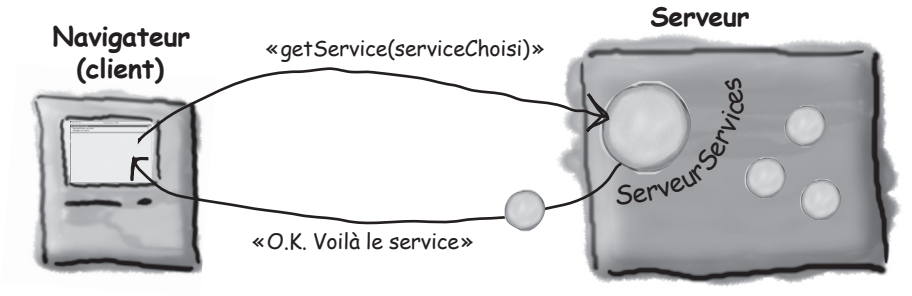


3. Le client affiche la liste des services dans une IHM.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Comment ça marche, suite...

4 L'utilisateur sélectionne un service dans la liste et le client appelle la méthode getService() sur le service distant. Le service distant retourne un objet sérialisé : c'est le service réel qui s'exécutera dans le navigateur client.



5 Le client appelle la méthode getIHM() sur l'objet service sérialisé qu'il vient de recevoir du service distant. L'IHM de ce service est affiché dans le navigateur et l'utilisateur peut interagir avec lui localement. Dès lors, nous n'avons plus besoin du service distant à moins que l'utilisateur ne décide de sélectionner un autre service.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Les classes et interfaces :

1. **L'interface ServeurServices implémente Remote**
Une bonne vieille interface distante RMI pour le service distant (celui qui possède la méthode pour obtenir la liste des services et retourner le service sélectionné).

2. **La classe ServeurServicesImpl implémente ServeurServices**
Le vrai service RMI distant (qui étend UnicastRemoteObject). Sa tâche consiste à instancier et mémoriser tous les services et à enregistrer le serveur lui-même (ServeurServicesImpl) dans le registre RMI.

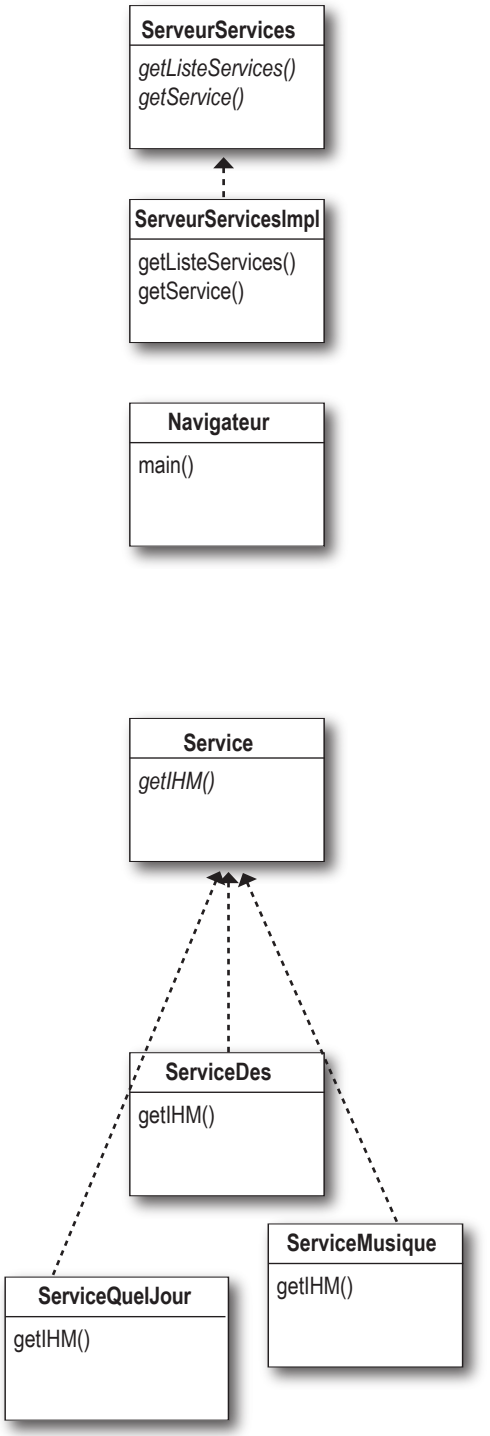
3. **La classe Navigateur**
Le client. Elle construit une IHM très simple, effectue une recherche dans le registre RMI pour obtenir la souche de ServeurServices, puis appelle dessus une méthode distante pour extraire la liste de services qui sera affichée dans l'IHM.

4. **L'interface Service**
C'est la clé de tout le reste. Cette interface très simple ne possède qu'une seule méthode, getIHM(). Tout service retourné au client doit implémenter cette interface. C'est ce qui rend le programme UNIVERSEL ! En implémentant cette interface, un service est accessible même si le client n'a aucune idée de la ou des classes qui constituent ce service. Tout ce que le client sait, c'est que l'entité qu'il reçoit doit implémenter l'interface Service et DOIT donc avoir une méthode getIHM().
Le client reçoit un objet sérialisé pour avoir appelé la méthode getService(serviceChoisi) sur la souche, et il ne dit qu'une chose à cet objet : « Je ne sais pas qui tu es, mais comme je SAIS que tu implémentes l'interface Service, je sais que je peux appeler getIHM(). Et puisque getIHM() retourne un JPanel, je vais me contenter de le placer dans l'interface graphique du navigateur et commencer à interagir avec ! »

5. **La classe ServiceDes implémente Service**
Vous n'avez pas de dés et vous en avez besoin ? Ce service vous permet de lancer de 1 à 6 dés où que vous soyez.

6. **La classe ServiceMusique implémente Service**
Vous souvenez vous de ce fabuleux petit programme de « vidéo musique » des premières Recettes de code ? Nous l'avons transformé en service et vous pouvez jouer avec sans arrêt jusqu'à ce que vos invités finissent par partir.

7. **La classe ServiceQuelJour implémente Service**
Êtes-vous né un vendredi ? Tapez votre date de naissance pour le savoir.



Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

L'interface ServeurServices (l'interface distante)

```
import java.rmi.*;

public interface ServeurServices extends Remote {

    Object[] getListeServices() throws RemoteException;

    Service getService(Object cleService) throws RemoteException;
}
```

Une interface distante RMI normale qui définit les deux méthodes que le service distant possédera.

L'interface Service (que les services implémentent)

```
import javax.swing.*;
import java.io.*;

public interface Service extends Serializable {
    public JPanel getIHM();
}
```

Une bonne vieille interface (autrement dit non distante) qui définit la seule et unique méthode que tout service doit posséder – getIHM().
L'interface étendant Serializable, toute classe implémentant l'interface Service sera automatiquement sérialisable.

C'est une obligation, parce que les services transitent sur le réseau quand le client a appelé getService() sur le ServeurServices distant.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

La class ServeurServicesImpl (l'implementation distante)

```
import java.rmi.*;
import java.util.*;
import java.rmi.server.*;

public class ServeurServicesImpl extends UnicastRemoteObject implements ServeurServices  {

    HashMap listeServices;

    public ServeurServicesImpl() throws RemoteException {
       installerServices();
    }

    private void installerServices() {
       listeServices = new HashMap();
       listeServices.put("Lancer les dés", new ServiceDes());
       listeServices.put("Jour de la semaine", new ServiceQuelJour());
       listeServices.put("Vidéo musique", new ServiceMusique());
    }

    public Object[] getListeServices() {
       System.out.println("à distance");
       return listeServices.keySet().toArray();

    }

    public Service getService(Object cleService) throws RemoteException {
       Service leService = (Service) listeServices.get(cleService);
       return leService;
    }

    public static void main (String[] args) {
       try {
         Naming.rebind("ServeurServices", new ServeurServicesImpl());
       } catch(Exception ex) { }
       System.out.println("Le service distant s'exécute");
    }
}
```

Une implémentation RMI normale.

Les services seront stockés dans une collection HashMap. Au lieu de placer UN objet dans la collection, vous en placez DEUX : une clé (comme une String) et une valeur (ce que vous voulez). (Voir l'annexe B pour les détails sur HashMap.)

Quand le constructeur est appelé, initialiser les vrais services (ServiceDes, ServiceMusique, etc.).

Créer les services (les vrais objets Service) et les placer dans la HashMap, avec un nom de type String (pour la clé).

Le client effectue cet appel pour obtenir une liste de services à afficher dans le navigateur. Nous lui envoyons un tableau de type Object (même s'il contient des chaînes) qui ne contient que les CLÉS qui sont dans la HashMap. Nous n'envoyons pas d'objet Service à moins que le client ne le demande en appelant getService().

Le client appelle cette méthode après que l'utilisateur a sélectionné un service dans la liste affichée (qu'il a obtenue avec l'appel de méthode ci-dessus). Ce code utilise la clé (la même que celle qui a été envoyée à l'origine au client) pour extraire le service correspondant de la HashMap.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

La classe Navigateur (le client)

```
import java.awt.*;
import javax.swing.*;
import java.rmi.*;
import java.awt.event.*;


public class Navigateur {

   JPanel panneau;
   JComboBox listeServices;
   ServeurServices serveur;


   public void construireIHM() {
      JFrame cadre = new JFrame("RMI navigateur");
      panneau = new JPanel();
      cadre.getContentPane().add(BorderLayout.CENTER, panneau);

      Object[] services = getListeServices();

      listeServices = new JComboBox(services);

      cadre.getContentPane().add(BorderLayout.NORTH, listeServices);

      listeServices.addActionListener(new EcouteMaListe());

      cadre.setSize(500,500);
      cadre.setVisible(true);

  }

   void chargerService(Object selectionService) {
       try {
          Service svc = serveur.getService(selectionService);

          panneau.removeAll();
          panneau.add(svc.getIHM());
          panneau.validate();
          panneau.repaint();
       } catch(Exception ex) {
          ex.printStackTrace();
       }
   }
```

Cette méthode consulte le registre RMI, obtient la souche et appelle getListeServices(). (La vraie méthode se trouve page suivante.)

Ajouter les services (un tableau d'Objects) à la JComboBox (la liste affichée). La JComboBox sait comment créer du texte affichable à partir de chaque élément du tableau.

Voici où nous ajoutons le vrai service à l'IHM après que l'utilisateur en a choisi un. (Cette méthode est appelée par l'auditeur d'événements sur la JComboBox). Nous appelons getService() sur le serveur distant (la souche de ServeurServices) et nous lui passons la chaîne qui a été affichée dans la liste (la MEME que celle que nous avons originellement reçue du serveur quand nous avons appelé getListeServices()). Le serveur retourne le vrai service (sérialisé) qui est désérialisé automatiquement (grâce à RMI), puis nous appelons simplement getIHM() sur le service et ajoutons le résultat (un JPanel) au panneau du navigateur.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

```
   Object[] getListeServices() {
      Object obj = null;
      Object[] services = null;

      try {

         obj = Naming.lookup("rmi://127.0.0.1/ServeurServices");

      }
      catch(Exception ex) {
       ex.printStackTrace();
      }
      serveur = (ServeurServices) obj;


      try {

        services = serveur.getListeServices();

      } catch(Exception ex) {
         ex.printStackTrace();
      }
      return services;

   }

   class EcouteMaListe implements ActionListener {
      public void actionPerformed(ActionEvent ev) {

          Object selection =  listeServices.getSelectedItem();
          chargerService(selection);
      }
    }

   public static void main(String[] args) {
      new Navigateur().construireIHM();
   }
}
```

Effectuer la recherche RMI et obtenir la souche.

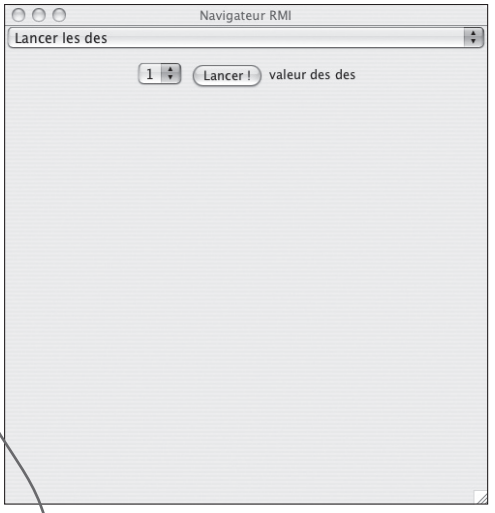
Convertir la souche dans le type de l'interface distante pour pouvoir appeler getListeServices().

getListeServices() nous donne le tableau d'Objects que nous affichons dans la JComboBox pour que l'utilisateur puisse choisir un service.

Si nous en sommes là, cela veut dire que l'utilisateur a effectué une sélection dans la liste de la JComboBox. Nous lisons donc la sélection et nous chargeons le service approprié. (Voir page précédente la méthode chargerService () qui demande au serveur le service qui correspond à la sélection.)

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

La classe ServiceDes (un service universel, implémente Service)

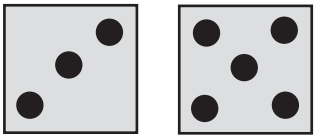



```
import javax.swing.*;
import java.awt.event.*;
import java.io.*;

public class ServiceDes implements Service {

    JLabel label;
    JComboBox nbDes;



    public JPanel getIHM() {
       JPanel panneau = new JPanel();
       JButton bouton = new JButton("Lancer!");
       String[] choix = {"1", "2", "3", "4", "5"};
       nbDes = new JComboBox(choix);
       label = new JLabel("valeur des dés");
       bouton.addActionListener(new EcouteLancer());
       panneau.add(nbDes);
       panneau.add(bouton);
       panneau.add(label);
       return panneau;
    }

   public class EcouteLancer implements ActionListener {
      public void actionPerformed(ActionEvent ev) {
         // lancer les dés
         String resultat = "";
         String selection = (String)  nbDes.getSelectedItem();
         int nbDesALancer = Integer.parseInt(selection);
         for (int i = 0; i < nbDesALancer; i++) {
            int r = (int) ((Math.random() * 6) + 1);
            resultat += (" " + r);
         }
        label.setText(resultat);

      }
    }
}
```

Voici LA méthode importante! La méthode de l'interface Service -- celle que le client va appeler quand ce service sera sélectionné et chargé. Vous pouvez faire tout ce que vous voulez dans la méthode getIHM(), tant que vous retournez un JPanel, pour qu'elle construise l'interface graphique.

À vos crayons

Réfléchissez à des façons d'améliorer ServiceDes. Une suggestion : en utilisant ce que vous avez appris aux chapitres 12 et 13, vous pouvez matérialiser les dés. Dessinez un rectangle pour chaque dé, puis le nombre approprié de cercles correspondant à chaque lancer.



Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

La classe ServiceMusique (un service universel, implémente Service)

```
import javax.sound.midi.*;
import java.io.*;
import javax.swing.*;
import java.awt.*;
import java.awt.evenement.*;

public class ServiceMusique implements Service {

    MonPanneauDessin monPanneau;

    public JPanel getIHM() {
       JPanel panneau = new JPanel();
       monPanneau = new MonPanneauDessin();
       JButton boutonJouer = new JButton("Jouer");
       boutonJouer.addActionListener(new EcouteJouer());
       panneau.add(monPanneau);
       panneau.add(boutonJouer);
       return panneau;
    }

    public class EcouteJouer implements ActionListener {
      public void actionPerformed(ActionEvent ev) {

        try {

         Sequencer sequenceur = MidiSystem.getSequencer();
         sequenceur.open();

         sequenceur.addControllerEventListener(monPanneau, new int[] {127});
         Sequence seq = new Sequence(Sequence.PPQ, 4);
         Track piste = seq.createTrack();

         for (int i = 0; i < 100; i+= 4) {

            int  rNum = (int) ((Math.random() * 50) + 1);
            if (rNum < 38) {  // ne faire que si num <38 (75% du temps)
              piste.add(makeEvent(144,1,rNum,100,i));
              piste.add(makeEvent(176,1,127,0,i));
              piste.add(makeEvent(128,1,rNum,100,i + 2));
            }
         } // fin de la boucle

         sequenceur.setSequence(seq);
         sequenceur.start();
         sequenceur.setTempoInBPM(220);
      } catch (Exception ex) {ex.printStackTrace();}

    } // fin de actionPerformed()
   } // fin de la classe interne
```

La méthode de Service! Elle affiche simplement un bouton et le service de dessin (où les rectangles seront dessinés).

Cette partie étant identique aux Recettes de code du chapitre 12, nous ne l'annoterons pas de nouveau

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

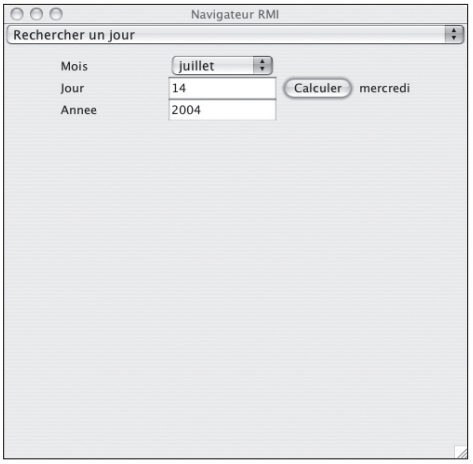
La classe ServiceMusique, suite...

```
public MidiEvent makeEvent(int comd, int chan, int one, int two, int tick) {
     MidiEvent evenement = null;
        try {
           ShortMessage a = new ShortMessage();
           a.setMessage(comd, chan, one, two);
           evenement = new MidiEvent(a, tick);

         }catch(Exception e) { }
     return evenement;
     }


class MonPanneauDessin extends JPanel implements ControllerEventListener {

     // nous ne dessinons que si nous avons un evenement
     boolean msg = false;

     public void controlChange(ShortMessage evenement) {
        msg = true;
        repaint();
    }

     public Dimension getPreferredSize() {
       return new Dimension(300,300);
     }

     public void paintComponent(Graphics g) {
        if (msg) {

          Graphics2D g2 = (Graphics2D) g;

          int r = (int) (Math.random() * 250);
          int gr = (int) (Math.random() * 250);
          int b = (int) (Math.random() * 250);

          g.setColor(new Color(r,gr,b));

          int ht = (int) ((Math.random() * 120) + 10);
          int lg = (int) ((Math.random() * 120) + 10);

          int x = (int) ((Math.random() * 40) + 10);
          int y =  (int) ((Math.random() * 40) + 10);

          g.fillRect(x,y,ht, lg);
          msg = false;

       } // fin du if
    } // fin de la méthode
  }  // fin de la classe interne
} // fin de la classe
```

Rien de nouveau dans cette page. Vous avez déjà tout vu dans les Recettes de code. Si vous voulez un autre exercice, essayez d'annoter ce code vous même, puis comparez-le à celui du chapitre 12.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

La classe ServiceQuelJour (un service universel, implémente Service)

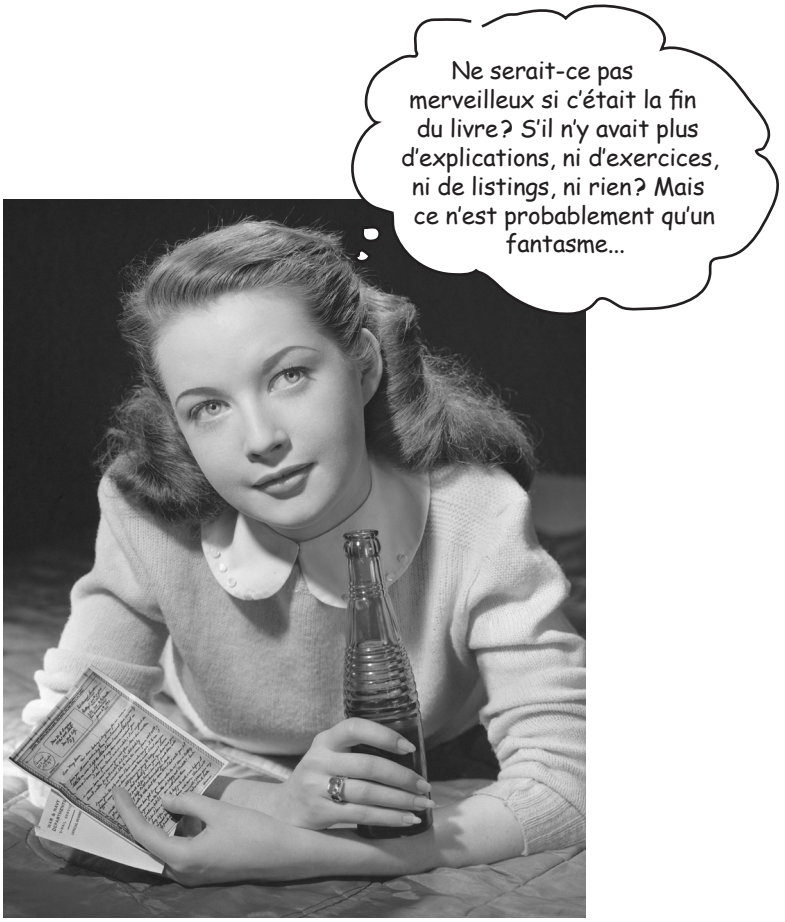



```
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
import java.io.*;
import java.util.*;
import java.text.*;

public class ServiceQuelJour implements Service {

    JLabel resultat;
    JComboBox mois;
    JTextField jour;
    JTextField annee;

    public JPanel getIHM() {
       JPanel panneau = new JPanel();
       JButton bouton = new JButton("Calculer");
       bouton.addActionListener(new EcouteCalculer());
       resultat = new JLabel("la date apparaît ici");
       DateFormatSymbols donneesDate = new DateFormatSymbols();
       mois = new JComboBox(donneesDate.getMonths());
       jour = new JTextField(8);
       annee = new JTextField(8);
       JPanel panneauSaisie = new JPanel(new GridLayout(3,2));
       panneauSaisie.add(new JLabel("Mois"));
       panneauSaisie.add(mois);
       panneauSaisie.add(new JLabel("Jour"));
       panneauSaisie.add(jour);
       panneauSaisie.add(new JLabel("Année"));
       panneauSaisie.add(annee);
       panneau.add(panneauSaisie);
       panneau.add(bouton);
       panneau.add(resultat);
       return panneau;
    }

    public class EcouteCalculer implements ActionListener {
       public void actionPerformed(ActionEvent ev) {
            int numMois = mois.getSelectedIndex();
            int numJour = Integer.parseInt(jour.getText());
            int numAnnee = Integer.parseInt(annee.getText());
            Calendar c = Calendar.getInstance();
            c.set(Calendar.MONTH, numMois);
            c.set(Calendar.DAY_OF_MONTH, numJour);
            c.set(Calendar.YEAR, numAnnee);
            Date date = c.getTime();
            String quelJour = (new SimpleDateFormat("EEEE")).format(date);
            resultat.setText(quelJour);
       }
    }
}
```

La méthode de l'interface Service qui construit l'IHM.

Reportez-vous au chapitre 10 si vous avez besoin de revoir comment le formatage des dates et des nombres fonctionne. Mais ce code est légèrement différent parce qu'il utilise la classe Calendar. De plus, SimpleDateFormat nous permet de spécifier un motif pour l'affichage de la date.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

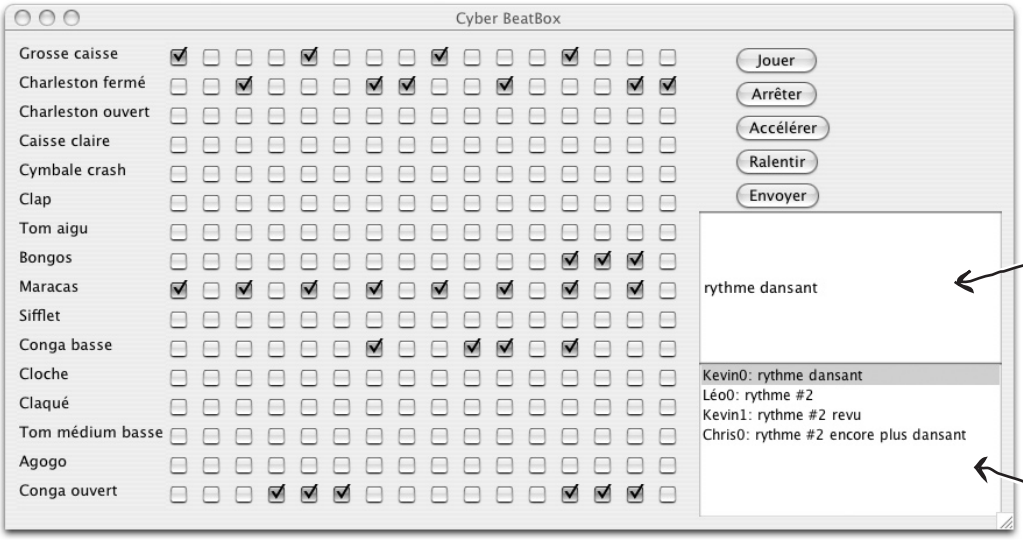


Félicitations !
Vous êtes arrivé au bout.

Bien sûr, il reste les deux annexes.
Et l'index.
Et puis il y a le site web.
Il n'y a vraiment aucune issue.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Annexe A :
Dernière recette de code



Votre message est envoyé aux autres joueurs avec votre motif quand vous cliquez sur le bouton Envoyer.

Les messages entrants des autres joueurs. Cliquez sur l'un d'eux pour charger le motif, puis sur le bouton Jouer pour le jouer.

Enfin, la version complète de la BeatBox !

Elle se connecte à un simple serveur (ServeurMusique) pour que vous puissiez échanger des motifs et des messages avec les autres clients.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

BeatBox : le client

La majeure partie de ce code étant identique à celui des Recettes de code des chapitres précédents, nous ne l'annoterons pas entièrement. Les nouveautés sont les suivantes :

IHM - nous ajoutons deux nouveaux composants pour la zone de texte qui affiche les messages entrants (une liste déroulante) et le champ de texte.

RÉSEAU - comme notre simple client de discussion du chapitre 15, la BeatBox se connecte maintenant à un serveur et utilise un flot d'entrée et un flot de sortie.

THREADS - toujours comme dans le client de discussion, nous lançons un « lecteur » qui écoute en permanence les messages provenant du serveur. Mais au lieu de texte, les messages entrant contiennent DEUX objets : le message (une String) et l'ArrayList sérialisé (l'entité qui contient l'état de toutes les cases à cocher.)

```
import java.awt.*;
import javax.swing.*;
import java.io.*;
import javax.sound.midi.*;
import java.util.*;
import java.awt.event.*;
import java.net.*;
import javax.swing.event.*;

public class BeatBoxFinal {

  JFrame leCadre;
  JPanel panneauPrincipal;
  JList listeEntrants;
  JTextField messageUtil;
  ArrayList<JCheckBox> listeCases;
  int numSuivant;
  Vector<String> listeDonnees = new Vector<String>();
  String nomUtilisateur;
  ObjectOutputStream out;
  ObjectInputStream in;
  HashMap<String, boolean[]> mapSequences = new HashMap<String, boolean[]>();

  Sequencer sequenceur;
  Sequence sequence;
  Sequence maSequence = null;
  Track piste;

  String[] nomsInstruments = {"Grosse caisse", "Charleston fermé", "Charleston ouvert",
  "Caisse claire", "Cymbale crash", "Clap", "Tom aigu", "Bongos", "Maracas", "Sifflet",
  "Conga basse", "Cloche", "Claqué", "Tom médium basse", "Agogo", "Conga ouvert"};

  int[] instruments = {35,42,46,38,49,39,50,60,70,72,64,56,58,47,67,63};
```


Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

```
public static void main (String[] args) {
    new BeatBoxFinal().connecter(args[0]); // args[0] est votre nom d'utilisateur
  }
  public void connecter(String name) {
    nomUtilisateur = name;
    // ouvrir la connexion au serveur
    try {
      Socket sock = new Socket("127.0.0.1", 4242);
      out = new ObjectOutputStream(sock.getOutputStream());
      in = new ObjectInputStream(sock.getInputStream());
      Thread distant = new Thread(new LectureDistante());
      distant.start();
    } catch(Exception ex) {
      System.out.println("Connexion impossible : vous devrez jouer seul.");
    }

    installerMidi();
    construireIHM();
  } // fin de la méthode connecter()

  public void construireIHM() {

    leCadre = new JFrame("Cyber BeatBox");
    BorderLayout agencement = new BorderLayout();
    JPanel arrierePlan = new JPanel(agencement);
    arrierePlan.setBorder(BorderFactory.createEmptyBorder(10,10,10,10));

    listeCases = new ArrayList<JCheckBox>();
    Box boiteBoutons = new Box(BoxLayout.Y_AXIS);
    JButton start = new JButton("Jouer");
    start.addActionListener(new EcouteStart());
    boiteBoutons.add(start);

    JButton stop = new JButton("Arrêter");
    stop.addActionListener(new EcouteStop());
    boiteBoutons.add(stop);

    JButton upTempo = new JButton("Accélérer");
    upTempo.addActionListener(new EcoutePlusVite());
    boiteBoutons.add(upTempo);

    JButton downTempo = new JButton("Ralentir");
    downTempo.addActionListener(new EcouteMoinsVite());
    boiteBoutons.add(downTempo);

    JButton sendIt = new JButton("Envoyer");
    sendIt.addActionListener(new EcouteEnvoi());
    boiteBoutons.add(sendIt);

    messageUtil = new JTextField();
    boiteBoutons.add(messageUtil);
```

Ajout d'un argument passé sur la ligne de commande : votre nom d'utilisateur.

Exemple : java BeatBoxFinal Zazie

Rien de nouveau... installer le réseau et les E/S, puis créer (et lancer) le thread "lecteur".

Le code de l'IHM. Rien de nouveau.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

```
  listeEntrants = new JList();
  listeEntrants.addListSelectionListener(new EcouteSelection());
  listeEntrants.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
  JScrollPane laListe = new JScrollPane(listeEntrants);
  boiteBoutons.add(laListe);
  listeEntrants.setListData(listeDonnees); // pas de données pour commencer
```

JList est un composant que nous n'avons pas encore utilisé. C'est là que les messages entrants sont affichés. Mais, contrairement à une discussion normale où vous vous contentez de REGARDER les messages, vous pouvez en CHOISIR un dans la liste pour charger et jouer le motif attaché.

```
  Box boiteNoms = new Box(BoxLayout.Y_AXIS);
  for (int i = 0; i < 16; i++) {
  boiteNoms.add(new Label(nomsInstruments[i]));
  }

  arrierePlan.add(BorderLayout.EAST, boiteBoutons);
  arrierePlan.add(BorderLayout.WEST, boiteNoms);

  leCadre.getContentPane().add(arrierePlan);
  GridLayout grille = new GridLayout(16,16);
  grille.setVgap(1);
  grille.setHgap(2);
  panneauPrincipal = new JPanel(grille);
  arrierePlan.add(BorderLayout.CENTER, panneauPrincipal);

  for (int i = 0; i < 256; i++) {
    JCheckBox c = new JCheckBox();
    c.setSelected(false);
    listeCases.add(c);
    panneauPrincipal.add(c);
  } // fin de la boucle

  leCadre.setBounds(50,50,300,300);
  leCadre.pack();
  leCadre.setVisible(true);

} // fin de construireIHM()

public void installerMidi() {
  try {
  sequenceur = MidiSystem.getSequencer();
  sequenceur.open();
  sequence = new Sequence(Sequence.PPQ,4);
  piste = sequence.createTrack();
  sequenceur.setTempoInBPM(120);
  } catch(Exception e) {e.printStackTrace();}

} // fin de installerMidi()
```

Rien d'autre de nouveau dans cette page.

Obtenir le séquenceur, créer une séquence puis créer une piste.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

```
public void construirePisteEtDemarrer() {
    ArrayList<Integer> trackList = null; // contiendra les instruments
    sequence.deleteTrack(piste);
    piste = sequence.createTrack();

    for (int i = 0; i < 16; i++) {
      trackList = new ArrayList<Integer>();

    for (int j = 0; j < 16; j++) {
      JCheckBox jc = (JCheckBox) listeCases.get(j + (16*i));
      if (jc.isSelected()) {
        int key = instruments[i];
        trackList.add(new Integer(key));
      } else {
        trackList.add(null); // cet emplacement doit être vide
      }
    } // fin de la boucle interne
    creerPistes(trackList);
    } // fin de la boucle externe
    piste.add(creerEvenement(192,9,1,0,15)); // pour avoir 16 temps
    try {
      sequenceur.setSequence(sequence);
      sequenceur.setLoopCount(sequenceur.LOOP_CONTINUOUSLY);
      sequenceur.start();
      sequenceur.setTempoInBPM(120);
    } catch(Exception e) {e.printStackTrace();}
  } // fin de la méthode

  public class EcouteStart implements ActionListener {
    public void actionPerformed(ActionEvent a) {
      construirePisteEtDemarrer();
    } // fin de actionPerformed()
  } // fin de la classe interne

  public class EcouteStop implements ActionListener {
    public void actionPerformed(ActionEvent a) {
      sequenceur.stop();
    } // fin de actionPerformed()
  } // fin de la classe interne

  public class EcoutePlusVite implements ActionListener {
    public void actionPerformed(ActionEvent a) {
      float tempoFactor = sequenceur.getTempoFactor();
      sequenceur.setTempoFactor((float)(tempoFactor * 1.03));
    } // fin de actionPerformed()
  } // fin de la classe interne
```

Construire une piste en parcourant les cases pour lire leur état et les associer à un instrument (et créer le MidiEvent correspondant). Ce code est plutôt complexe, mais comme il est EXACTEMENT identique à celui des chapitres précédents, vous pouvez vous reporter aux précédentes Recettes de code pour revoir toutes les explications.

Les auditeurs de l'interface. Exactement comme dans les précédents chapitres.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

```
public class EcouteMoinsVite implements ActionListener {
   public void actionPerformed(ActionEvent a) {
     float tempoFactor = sequenceur.getTempoFactor();
     sequenceur.setTempoFactor((float)(tempoFactor * .97));
   }
 }

 public class EcouteEnvoi implements ActionListener {
   public void actionPerformed(ActionEvent a) {
     // créer une ArrayList de l'ETAT des cases
     boolean[] etatCases = new boolean[256];
   for (int i = 0; i < 256; i++) {
     JCheckBox coche = (JCheckBox) listeCases.get(i);
     if (coche.isSelected()) {
       etatCases[i] = true;
     }
   } // fin de la boucle

   String messageAEnvoyer = null;
     try {
       out.writeObject(nomUtilisateur + numSuivant++ + ": " + messageUtil.getText());
       out.writeObject(etatCases);
     } catch(Exception ex) {
       System.out.println("Désolé : envoi au serveur impossible.");
     }
     messageUtil.setText("");
   } // fin de actionPerformed()
 } // fin de la classe interne

 public class EcouteSelection implements ListSelectionListener {
   public void valueChanged(ListSelectionEvent le) {
     if (!le.getValueIsAdjusting()) {
       String selected = (String) listeEntrants.getSelectedValue();
       if (selected != null) {
         boolean[] selectedState = (boolean[]) mapSequences.get(selected);
         changerSequence(selectedState);
         sequenceur.stop();
         construirePisteEtDemarrer();
       }
     }
   } // fin de valueChanged()
 } // fin de la classe interne

 public class LectureDistante implements Runnable {
```

Ceci est nouveau... Ce code est très semblable à celui du client de discussion, excepté qu'au lieu d'envoyer un message de type String nous sérialisons deux objets (le message et le motif) et nous les écrivons dans le flot de sortie de la socket (vers le serveur).

Autre nouveauté : un ListSelectionListener qui nous informe que l'utilisateur a sélectionné un message dans la liste. À ce moment-là, nous chargeons IMMÉDIATEMENT le motif associé (il est dans la HashMap nommée mapSequences) et nous commençons à le jouer. Il y a quelques tests if parce que l'obtention des ListSelectionEvents n'est pas évidente.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

```
  boolean[] etatCases = null;
  String nomAAfficher = null;
  Object obj = null;
  public void run() {
    try {
      while((obj=in.readObject()) != null) {
        System.out.println("reçu un objet du serveur");
        System.out.println(obj.getClass());
        String nomAAfficher = (String) obj;
        etatCases = (boolean[]) in.readObject();
        mapSequences.put(nomAAfficher, etatCases);
        listeDonnees.add(nomAAfficher);
        listeEntrants.setListData(listeDonnees);
      } // fin du while
    } catch(Exception ex) {ex.printStackTrace();}
  } // fin de run()
} // fin de la classe interne

public class EcouteLaMienne implements ActionListener {
  public void actionPerformed(ActionEvent a) {
    if (maSequence != null) {
      sequence = maSequence; // restaure l'original
    }
  } // fin de actionPerformed()
} // fin de la classe interne

public void changerSequence(boolean[] etatCases) {
  for (int i = 0; i < 256; i++) {
    JCheckBox coche = (JCheckBox) listeCases.get(i);
      if (etatCases[i]) {
        coche.setSelected(true);
      } else {
        coche.setSelected(false);
      }
  } // fin de la boucle
} // fin de changerSequence()

public void creerPistes(ArrayList liste) {
  Iterator it = liste.iterator();
  for (int i = 0; i < 16; i++) {
    Integer num = (Integer) it.next();
    if (num != null) {
      int numTouche = num.intValue();
      piste.add(creerEvenement(144,9,numTouche, 100, i));
      piste.add(creerEvenement(128,9,numTouche,100, i + 1));
    }
  } // fin de la boucle
} // fin de creerPistes()
```

Voilà la tâche du thread – lire les données provenant du serveur. Dans ce code, les « données » seront toujours deux objets sérialisés : le message et le motif (l'ArrayList qui contient les états des cases).

Quand un message arrive, nous lisons (désérialisons) les deux objets (le message et l'ArrayList de valeurs booléennes qui représentent l'état des cases) et nous les ajoutons à la JList. L'insertion dans une JList est un processus en deux étapes : mémoriser les données dans un Vector (une ancienne version d'ArrayList) puis dire à la JList d'utiliser ce Vector comme source de ce qu'elle va afficher dans la liste.

Cette méthode est appelée quand l'utilisateur sélectionne un élément de la liste. Nous remplaçons IMMEDIATEMENT le motif courant par celui qui a été sélectionné

Tout le code MIDI est exactement identique à celui de la précédente version.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

```
public MidiEvent creerEvenement(int comd, int can, int un, int deux, int tic) {
    MidiEvent evenement = null;
    try {
      ShortMessage a = new ShortMessage();
      a.setMessage(comd, can, un, deux);
      evenement = new MidiEvent(a, tic);
    }catch(Exception e) { }
    return evenement;
  } // fin de creerEvenement()
} // fin de la classe
```

Rien de nouveau. Strictement identique à la version précédente.

À vos crayons

De quelle façon pourriez-vous améliorer ce programme ?

Voici quelques idées pour vous mettre sur la voie :

1) Une fois que vous effectuez une sélection, le motif courant est perdu. Si vous étiez en train de travailler sur ce motif (ou d'en modifier un), vous n'avez pas de chance. Vous pourriez afficher une boîte de dialogue qui demanderait à l'utilisateur s'il veut enregistrer le motif courant.

2) Si vous ne fournissez pas d'argument sur la ligne de commande, vous obtenez une exception au moment de l'exécution ! Placez du code dans la méthode main() pour vérifier qu'un argument a été passé. Si l'utilisateur oublie l'argument, utilisez-en un par défaut ou affichez un message qui lui demande de relancer l'exécution en fournissant son nom d'utilisateur.

3) Il pourrait être agréable d'avoir une fonctionnalité qui vous permette de cliquer sur un bouton et de générer un motif aléatoire. Vous pourriez tomber sur un motif qui vous plaise vraiment. Mieux encore, vous pourriez offrir la possibilité de télécharger des motifs « classiques » de jazz, de rock, de reggae, etc., que l'utilisateur pourrait modifier.

Vous pouvez trouver des motifs existants sur le site web.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

BeatBox : le serveur

La majeure partie de ce code est identique à celui du serveur de discussion que nous avons créé dans le chapitre sur le résau et les threads. En réalité, la seule différence est que le serveur reçoit puis renvoie deux objets sérialisés et non une chaîne de caractères (même si l'un des deux objets sérialisés en est une).

```
import java.io.*;
import java.net.*;
import java.util.*;

public class ServeurMusique {

   ArrayList<ObjectOutputStream> flotsSortieClients;

   public static void main (String[] args) {
      new ServeurMusique().go();
   }

   public class GestionClient implements Runnable {

      ObjectInputStream in;
      Socket socketClient;

      public GestionClient(Socket socket) {
         try {
            socketClient = socket;
            in = new ObjectInputStream(socketClient.getInputStream());
         } catch(Exception ex) {ex.printStackTrace();}
      } // fin du constructeur

      public void run() {
            Object o2 = null;
            Object o1 = null;
         try {
            while ((o1 = in.readObject()) != null) {
            o2 = in.readObject();
            System.out.println("lecture de deux objets");
      afficherATous(o1, o2);
            } // fin du while
         } catch(Exception ex) {ex.printStackTrace();}
      } // fin de run()
   } // fin de la classe interne
```


Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

```
      public void go() {
      flotsSortieClients = new ArrayList<ObjectOutputStream>();

         try {
            ServerSocket socketServeur = new ServerSocket(4242);
         while(true) {
            Socket socketClient = socketServeur.accept();
            ObjectOutputStream out = new ObjectOutputStream(socketClient.getOutputStream());
            flotsSortieClients.add(out);

            Thread t = new Thread(new GestionClient(socketClient));
            t.start();
               System.out.println("connexion obtenue");
            }
         }catch(Exception ex) {
            ex.printStackTrace();
         }
      } // fin de go()

   public void afficherATous(Object un, Object deux) {
      Iterator it = flotsSortieClients.iterator();
      while(it.hasNext()) {

         try {
            ObjectOutputStream out = (ObjectOutputStream) it.next();
            out.writeObject(un);
            out.writeObject(deux);
         }catch(Exception ex) {ex.printStackTrace();}
      }
   } // fin de afficherATous()
} // fin de la classe
```


Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Annexe B

Dix sujets importants qui ne tenaient pas dans le livre...

Nous avons abordé de nombreux sujets et vous en avez pratiquement terminé avec ce livre. Vous nous manquerez. Mais avant de vous laisser partir, nous aurions scrupule à vous voir vous aventurer dans JavaLand sans un peu plus de préparation. Il est impossible de faire entrer tout ce que vous devriez savoir dans cette annexe relativement restreinte. En fait, nous *avions* tout d'abord inclus tout ce que vous devez savoir de Java (non abordé dans les chapitres) en réduisant au maximum la taille des caractères. Tout tenait mais personne ne pouvait le lire. Nous en avons donc supprimé la plus grande partie et gardé les meilleurs morceaux pour ce Top Ten.

Cette annexe *est* la vraie fin de ce livre. À part l'index (un must !).

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

10 Opérateurs bit à bit

Quelle est leur importance?

Nous avons vu qu'il y avait 8 bits dans un byte, 16 bits dans un short et ainsi de suite. Vous aurez peut être l'occasion de devoir activer ou désactiver des bits un par un. Si par exemple vous vous retrouvez à écrire du code pour votre nouveau grille-pain compatible Java, vous risquez de vous apercevoir que certains réglages doivent être contrôlés au niveau du bit en raison des sévères limitations de la mémoire. Pour plus de facilité, dans les commentaires, nous ne montrons que les 8 derniers bits.

NON bit à bit : ~

Cet opérateur inverse tous les bits d'un type primitif.

```
int x = 10; //       les bits sont 00001010
x = ~x  // les bits sont maintenant 11110101
```

Les trois opérateurs suivants comparent deux valeurs primitives bit par bit et retournent le résultat de la comparaison. Pour les décrire nous nous appuierons sur l'exemple suivant :

```
int x = 10;    // les bits sont 00001010
int y =  6;    // les bits sont 00000110
```

ET bit à bit : &

Cet opérateur retourne une valeur dont les bits ne sont à 1 que si les deux bits d'origine sont à 1 :

```
int a = x & y;  // les bits sont 00000010
```

OU bit à bit : |

Cet opérateur retourne une valeur dont les bits ne sont à 1 que si l'un au moins des deux bits d'origine est à 1 :

```
int a = x | y;  // les bits sont 00001110
```

XOR (OU exclusif) bit à bit : ^

Cet opérateur retourne une valeur dont les bits ne sont à 1 que si exactement un des bits d'origine est à 1 :

```
int a = x ^ y;  // les bits sont 00001100
```

Les opérateurs de décalage

Ces opérateurs acceptent une seule valeur primitive entière et décalent (font glisser) tous ses bits dans une direction donnée. Si vous voulez dépoussiérer vos connaissances sur la numération en base 2, vous vous rendrez compte que décaler les bits à *gauche multiplie* un nombre par une puissance de deux et que le décalage à *droite* le *divise* par une puissance de deux.

Nous utiliserons l'exemple suivant pour les trois prochains opérateurs :

```
int x = -11;   //   les bits sont 11110101
```

D'accord, d'accord, c'est l'explication sur le stockage des nombres négatifs et le complément à deux la plus courte du monde. Souvenez-vous que le chiffre le plus à gauche d'un nombre négatif est le ***bit de signe***. En Java, le bit de signe d'un nombre négatif est toujours positionné à 1 et celui d'un nombre positif est toujours à 0. Java utilise la formule du *complément* à deux pour stocker les nombres négatifs. Pour changer le signe d'un nombre avec le complément à deux, inversez tous les bits puis ajoutez 1 (en prenant comme exemple un byte, cela voudrait dire ajouter 00000001 à la valeur inversée).

Décalage à droite : >>

Cet opérateur décale à droite tous les bits d'un nombre donné et remplit les bits de gauche avec la valeur d'origine du bit le plus à gauche. **Le bit de signe *ne* change pas :**

```
int y = x >> 2;  // les bits sont 11111101
```

Décalage à droite non signé : >>>

Cet opérateur est exactement identique au précédent SAUF qu'il remplit TOUJOURS les bits de gauche avec des zéros. **Le bit de signe *peut* changer** :

```
int y = x >>> 2;  // les bits sont 00111101
```

Décalage à gauche : <<

Exactement comme l'opérateur de décalage à droite non signé mais dans l'autre sens. Les bits de droite sont remplis par des zéros. **Le bit de signe *peut* changer.**

```
int y = x << 2;  // les bits sont 11010100
```


Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

9 Immuabilité

Quelle est l'importance de l'immuabilité des chaînes ?

Quand la taille de vos programmes Java commencera à prendre de l'importance, vous finirez inévitablement par avoir des quantités d'objets de type String. Pour des raisons de sécurité et pour économiser de la mémoire (n'oubliez-pas que ces programmes peuvent tourner sur de minuscules téléphones mobiles), les chaînes sont immuables en Java. Cela signifie que quand vous écrivez :

```
String s = "0";
for (int x = 1; x < 10; x++) {
  s = s + x;
}
```

vous créez en réalité dix objets String (dont les valeurs sont "0", "01", "012"... "0123456789"). À la fin, *s* pointe sur la chaîne qui vaut "0123456789", mais les dix objets existent toujours !

Chaque fois que vous créez un nouvel objet String, la JVM le place dans une zone particulière de la mémoire nommée «pool de Strings». Si une chaîne de même valeur s'y trouve déjà, la JVM ne la duplique pas et votre variable de référence pointe maintenant sur l'entrée existante. La JVM fonctionne ainsi parce que les chaînes sont immuables : une variable de référence ne peut pas modifier la valeur d'une chaîne, même à partir d'une autre variable référençant la même chaîne.

L'autre problème avec le pool de Strings, c'est que le ramasse-miettes ne s'en occupe pas. Dans notre exemple, sauf si vous créez plus tard par coïncidence une chaîne nommée "01234", les neuf premières chaînes créées dans notre boucle *for* restent là à gaspiller de la mémoire.

Où est l'économie de mémoire ?

Eh bien, si vous ne faites pas attention, *il n'y en a pas* ! Mais si vous comprenez comment l'immuabilité fonctionne, vous pouvez parfois en tirer parti. En revanche, si vous devez effectuer de nombreuses manipulations de chaînes (concaténations, etc.). Il y a une autre classe, StringBuilder, qui convient mieux à cet effet. Nous reparlerons de StringBuilder plus loin.

Quelle est l'importance de l'immuabilité des enveloppes ?

Dans le chapitre sur les maths, nous avons évoqué les deux principales utilisations des classes enveloppes :

- Envelopper une valeur primitive pour qu'elle puisse faire semblant d'être un objet.
- Utiliser les méthodes utilitaires statiques (par exemple, Integer.parseInt()).

N'oubliez pas que quand vous créez un objet enveloppe comme ceci :

```
Integer enveloppe = new Integer(42);
```

c'est terminé pour cet objet. Sa valeur sera toujours 42.

Il n'y a pas de méthode set pour les objets enveloppes. Vous pouvez bien sûr faire en sorte qu'une enveloppe référence un autre objet, mais dans ce cas vous aurez deux objets. Une fois un objet enveloppe créé, il n'existe aucun moyen de modifier sa valeur !

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

8 Assertions

Nous n'avons pas beaucoup parlé de la façon de déboguer votre programme pendant le développement. Nous pensons qu'il est préférable d'apprendre Java en mode ligne de commande, comme nous l'avons fait tout au long de ce livre. Une fois que vous serez un pro, vous déciderez peut-être de travailler avec un IDE* et vous disposerez d'autres outils de débogage. Au bon vieux temps, quand un programmeur Java voulait déboguer son code, il le parsemait d'instructions System.out.println( ) qui affichaient la valeur courante des variables et des messages «J'en suis là» pour permettre de vérifier que le flot de contrôle fonctionnait correctement. (C'est le cas de la Recette de code du chapitre 6.) Puis, une fois le programme au point, il retirait toutes ces instructions System.out.println( ). C'était une procédure fastidieuse et sujette à l'erreur. Mais, depuis Java 1.4 (et 5.0), le débogage est devenu beaucoup plus facile. Pourquoi ?

Assertions

Les assertions sont comme des instructions System.out.println( ) gonflées aux stéroïdes. Ajoutez-les à votre code comme vous le feriez avec des instructions println. Le compilateur Java 5.0 suppose que vous allez compiler des fichiers source compatibles 5.0. En conséquence, à partir de Java 5.0, la compilation avec assertions est activée par défaut.

Lors de l'exécution, si vous ne faites rien, les instructions assert que vous avez insérées dans votre code seront ignorées par la JVM et ne ralentiront pas votre programme. Mais si vous demandez à la JVM *d'activer* vos assertions, elles vous aideront à déboguer sans modifier une ligne de code !

Certains se sont plaints de devoir laisser des instructions assert dans leur code de production, mais cette technique peut se montrer précieuse quand votre code est déjà déployé sur le terrain. Si votre client a des problèmes, vous pouvez lui dire d'exécuter le programme en activant les assertions et de vous communiquer le résultat. Si vous aviez supprimé les assertions du code déployé, vous n'auriez pas cette possibilité. Et il n'y a pratiquement aucun inconvénient : les assertions inactives étant totalement ignorées par la JVM, il n'y a aucune dégradation de performances à craindre.

Comment activer les assertions

Ajoutez ces instructions à votre code chaque fois que vous pensez qu'une condition doit être vraie. Par exemple :

```
assert (hauteur > 0);

// si vrai, continuer normalement

// si faux, lancer une AssertionError
```

Vous pouvez augmenter les informations sur l'état de la pile d'appelants en écrivant :

```
assert (hauteur > 0) : "hauteur= " +
hauteur + " poids = " + poids;
```

L'expression qui suit les deux points peut être n'importe quelle expression Java légale dont la valeur s'avère non nulle. Mais, quoi qu'il arrive, ne créez aucune ***assertion qui modifie l'état d'un objet ! Si vous le faisiez, l'activation des*** assertions lors de l'exécution pourrait modifier le comportement de votre programme.

Compiler et exécuter avec des assertions

Pour *compiler* avec des assertions :

```
javac  JeuTestDrive.java
```

(Notez qu'aucune option n'est nécessaire.)

Pour *exécuter* avec des assertions :

```
java -ea JeuTestDrive
```

* Un IDE est un environnement de développement intégré. Eclipse, JBuilder de Borland ou l'outil *open source* NetBeans (netbeans.org) sont des IDE.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

7 Portée de bloc

Au chapitre 9, nous avons indiqué que les variables locales ne vivaient que tant que la méthode dans laquelle elles étaient déclarées étaient sur la pile. Mais certaines variables peuvent avoir une durée de vie encore plus courte. À l'intérieur d'une méthode, nous créons souvent des blocs de code. C'est ce que nous avons fait tout au long de ce livre, mais sans en parler explicitement en termes de blocs. En général, les blocs de code se trouvent dans une méthode et sont délimités par des accolades{ }. Parmi les blocs de code que vous reconnaîtrez facilement, citons les instructions itératives (*for*, *while*) et conditionnelles (*if* par exemple).

Prenons un exemple :

```
void faireQqch() {
    int x = 0;
    for(int y = 0; y < 5; y++) {

        x = x + y;
    }

    x = x * y;
}
```

Début de la méthode.

Variable locale dont la portée est la méthode.

Début d'un bloc for : la portée de y est limitée à la boucle !

Pas de problème, x et y sont toutes deux dans la portée.

Fin de la boucle for.

Ouille ! Compilation impossible ! y a cessé d'être visible ! (Faites attention : ceci n'est pas vrai dans tous les langages de programmation.)

Fin de la méthode : x aussi a cessé d'être visible.

Dans cet exemple, *y* est une variable de bloc, déclarée dans un bloc, et *y* sort de la portée dès la fin de la boucle for. Vos programmes Java seront plus faciles à déboguer et à étendre si vous employez des variables locales à la place des variables d'instance et des variables de blocs au lieu de variables locales chaque fois que vous en aurez la possibilité. Comme le compilateur s'assure que vous n'essayez pas d'utiliser une variable locale qui n'est plus dans la portée, vous ne risquez pas que le programme s'effondre à l'exécution.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

6 Invocations liées

Lorsque ce livre abordait certains sujets complexes, nous avons essayé de conserver une syntaxe aussi nette et aussi lisible que possible. Il existe toutefois de nombreux raccourcis légaux en Java, et vous en rencontrerez certainement, surtout si vous êtes amené à lire beaucoup de code que vous n'avez pas écrit vous-même. L'une des constructions les plus courantes auxquelles vous serez confronté est l'invocation liée. En voici un exemple :

```
StringBuffer sb = new StringBuffer("canard");
sb = sb.delete(3,6).insert(2,"anca").deleteCharAt(1);
System.out.println("sb = " + sb);
// le résultat est sb = cancan
```

Que se passe-t-il donc dans la deuxième ligne de code ? L'exemple est un peu forcé, admettons-le, mais vous devez apprendre à déchiffrer ce genre d'instructions.

1 - Travaillez de gauche à droite.

2 - Trouvez le résultat de l'appel de méthode le plus à gauche, en l'occurrence `sb.delete(3,6)`. Si vous consultez la documentation sur StringBuffer dans l'API, vous verrez que la méthode `delete()` retourne un objet StringBuffer. Le résultat de l'exécution de la méthode `delete()` est un objet StringBuffer dont la valeur est « can ».

3 - La méthode suivante, `insert()`, est appelée sur l'objet StringBuffer nouvellement créé, "can". Le résultat de cet appel de méthode est également un objet StringBuffer (bien qu'il n'ait pas besoin d'avoir le même type que la valeur de retour de la méthode précédente) et voilà ce qui se passe : l'objet retourné est utilisé pour appeler la méthode suivante (celle de droite). En théorie, vous pouvez lier autant de méthodes que vous le voulez dans une seule instruction (même s'il est rare d'en rencontrer plus de trois). Sans la liaison, la deuxième ligne de code de notre exemple serait plus lisible et aurait l'aspect suivant :

```
sb = sb.delete(3,6);
sb = sb.insert(2,"anca");
sb = sb.deleteCharAt(1);
```

Mais voici un exemple plus courant et plus utile. Vous l'avez certes déjà vu mais il vaut mieux en reparler. Il s'agit du cas où votre méthode main() a besoin d'appeler une méthode d'instance de la classe principale, sans que vous ayez besoin de conserver une *référence* à l'instance de la classe. Autrement dit, main() ne doit créer l'instance que pour pouvoir en appeler une des méthodes.

```
class Abradacabra{
   public static void main(String [] args) {
     new Abradacabra().go();
   }
   void go() {
    // ce que nous voulons VRAIMENT...
   }
}
```

Nous voulons appeler go(). Mais comme l'instance d'Abradacabra ne nous intéresse pas, nous ne nous soucions pas d'affecter le nouvel objet Abradacabra à une référence.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

5 Classes imbriquées anonymes et statiques

Il existe de nombreuses formes de classes imbriquées

À la section sur la gestion des événements dans une IHM, nous avons commencé à utiliser des classes imbriquées (internes) pour implémenter des interfaces auditeurs d'événements. C'est la forme de classe interne la plus courante, la plus pratique et la plus lisible : la classe est simplement imbriquée entre les accolades d'une autre classe. N'oubliez pas que cela signifie que vous avez besoin d'une instance de la classe externe pour avoir une instance de la classe interne, parce que la classe interne est un membre de la classe externe qui la contient.

Mais il y a d'autres formes de classes internes, notamment *statiques* et *anonymes*. Nous n'entrerons pas dans les détails, mais nous ne voulons pas que vous soyez surpris par une syntaxe étrange quand vous la verrez dans le code de quelqu'un d'autre. En effet, parmi tout ce que vous pouvez virtuellement faire en Java, rien ne semble produire du code à l'aspect aussi bizarre que des classes internes anonymes. Mais commençons par quelque chose de plus simple : les classes statiques imbriquées.

Classes statiques imbriquées

Vous savez déjà ce que veut dire statique : quelque chose liée à la classe et, non à une instance donnée. Une classe statique imbriquée ressemble aux classes non statiques que nous avons utilisées pour l'auditeur d'événements, sauf qu'elles ont précédées du mot-clé `static`.

```
public class TricExterne {
   static class TracInterne {
      void leDire() {
         System.out.println("méthode d'une classe statique interne");
      }
   }
}

class Test {
   public static void main (String[] args) {
      TricExterne.TracInterne trac = new TricExterne.TracInterne();
      trac.leDire();
   }
}
```

Une classe statique imbriquée n'est rien d'autre qu'une classe contenue dans une autre et précédée du modificateur static.

Comme une classe statique interne est... statique, vous n'utilisez pas d'instance de la classe externe. Vous indiquez juste le nom de la classe, de la même manière que vous invoquez des méthodes statiques ou que vous accédez à des variables statiques.

Les classes statiques internes ressemblent plus aux classes non imbriquées ordinaires, au sens où elles n'entretiennent pas de relation particulière avec un objet externe. Mais comme ces classes statiques sont toujours considérées comme des membres de la classe externe, elles ont toujours accès aux membres privés de cette classe externe... mais seulement à ceux qui sont également *statiques*. Comme la classe statique interne n'est pas liée à une instance de la classe externe, elle n'a aucun moyen particulier d'accéder aux méthodes et aux variables non statiques (d'instance).

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

5 Classes imbriquées anonymes et statiques, suite

La différence entre *imbriquée* et *interne*

Toute classe Java définie dans la portée d'une autre classe se nomme une classe imbriquée. Peu importe qu'elle soit anonyme, statique, normale ou autre. Si elle est dans une autre classe, on la considère techniquement comme une classe imbriquée. Mais on appelle souvent les classes imbriquées non statiques des classes internes, comme nous l'avons fait jusqu'ici dans ce livre. Conclusion : toutes les classes internes sont des classes imbriquées, mais toutes les classes imbriquées ne sont pas des classes internes.

Classes internes anonymes

Imaginez que vous écrivez le code d'une IHM et que vous vous aperceviez soudain que vous avez besoin d'une instance d'une classe qui implémente ActionListener. Mais vous constatez que vous n'avez pas d'instance d'ActionListener. Puis vous vous rendez compte que vous n'avez jamais écrit de classe pour cet auditeur. À ce stade, vous avez deux possibilités :

1) Écrire une classe dans votre programme, comme nous l'avons fait dans le nôtre, puis l'instancier et transmettre cette instance à la méthode d'enregistrement de l'événement du bouton, addActionListener().

OU

2) Créer une classe interne *anonyme* et l'instancier, à la volée, juste à temps. ***Littéralement à l'endroit précis où vous avez besoin de l'objet***. Oui, vous créez la classe et l'instance à l'emplacement où vous fourniriez normalement l'instance. Réfléchissez-y un moment : cela signifie que vous transmettez toute une *classe* là où vous passeriez d'ordinaire l'*instance* en argument à une méthode !

```
import java.awt.event.*;
import javax.swing.*;
public class TestAnonyme {
   public static void main (String[] args) {

      JFrame cadre = new JFrame();
      JButton bouton = new JButton("cliquez")
      frame.getContentPane().add(bouton);
      // bouton.addActionListener(quitListener);

      bouton.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent ev) {
          System.exit(0);
        }
      });

   }
}
```

Nous avons crée un cadre, ajouté un bouton, et maintenant, nous voulons enregistrer un auditeur d'action auprès du bouton. Sauf que nous n'avons jamais crée de classe qui implémente l'interface ActionListener...

Normalement, nous passerions une référence à une instance d'une classe interne... une classe interne qui implémente ActionListener (et la méthode actionPerformed()).

Cette instruction

se termine là!

Remarquez que nous écrivons « new ActionListener() » bien que ActionListener soit une interface et qu'on ne puisse pas en CRÉER une instance! Mais cette syntaxe signifie en réalité « créer une nouvelle classe (sans nom) qui implémente l'interface ActionListener, et, tant que nous y sommes, voici l'implémentation de la méthode de l'interface, actionPerformed() ».

Mais au lieu de transmettre une référence à un objet, nous transmettons... toute la définition de la classe!! Autrement dit, nous écrivons la classe qui implémente ActionListener À L'ENDROIT MÊME OÙ NOUS EN AVONS BESOIN. Cette syntaxe crée aussi automatiquement une instance de la classe.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

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

Java dispose de quatre niveaux d'accès et de trois modificateurs. Pourquoi seulement trois modificateurs ? Parce que le package est le niveau d'accès par défaut et qu'il ne nécessite pas de modificateur.

Niveaux d'accès (du plus permissif au plus restrictif)

public ← Signifie que la totalité du code peut accéder à l'élément public (par "élément" nous entendons une classe, une variable, une méthode, un constructeur, etc.).

protected ← protected fonctionne comme package (le code situé dans le même package a accès), SAUF qu'il permet également aux sous-classes extérieures au package d'hériter de l'élément protégé.

default ← Signifie que seul le code appartenant au même package de la classe ayant l'élément par défaut peut accéder à l'élément.

private ← Signifie que seul le code de la même classe peut accéder à l'élément privé. Souvenez-vous que cela signifie privé pour la classe, non privé pour l'objet. Un Chien peut voir les éléments privés d'un autre Chien, mais pas un Chat.

Modificateurs d'accès

```
public
protected
private
```

La plupart du temps, vous n'utiliserez que les niveaux public et private.

public

Utilisez public pour les classes, les constantes (variables statiques finales) et les méthodes que vous exposez au reste du code (par exemple les méthodes get et set), ainsi que la plupart des constructeurs.

private

Utilisez private pour pratiquement toutes les variables d'instance et pour les méthodes que vous ne voulez pas voir appelées par un code externe (autrement dit les méthodes utilisées par les méthodes publiques de votre classe).

Même si vous aurez rarement l'occasion d'utiliser les deux autres (protected et default), vous devez les connaître parce que vous les rencontrerez dans d'autres programmes que les vôtres.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

4 Niveaux d'accès et modificateurs d'accès (suite)

default et protected

default

Les niveaux d'accès default et protected sont tous deux liés aux packages. Le niveau package est simple — il signifie que seul le *code du même package* peut y accéder. Par exemple une classe qui a ce niveau d'accès (autrement dit une classe qui n'est pas explicitement déclarée *public*) n'est accessible que par les classes qui appartiennent au même package qu'elle.

Mais que signifie en réalité accéder à une classe ? Un code qui n'a pas accès à une classe n'est en aucun cas autorisé à l'utiliser. Par exemple, si vous n'avez pas accès à une classe en raison d'une restriction d'accès, vous n'avez pas le droit de l'instancier ni même de la déclarer comme type d'une variable, d'un argument ou d'une valeur de retour. Elle ne peut absolument pas figurer dans votre code ! Si vous le faites, le compilateur se rebiffera.

Réfléchissez aux implications. Une classe accessible par défaut possédant des méthodes publiques signifie que ces méthodes ne sont pas publiques du tout. Vous ne pouvez pas accéder à une méthode si vous ne pouvez pas voir la classe.

Mais pourquoi vouloir limiter l'accès au code appartement au même package ? En général, les packages sont des groupes de classes conçues pour fonctionner ensemble. Il semble donc logique que les classes d'un même package aient besoin d'accéder au code des autres, tandis que seul un petit nombre de classes et de méthodes sera exposé au monde extérieur (autrement dit au code externe à ce package).

C'est cela, l'accès par défaut. C'est très simple. Si un élément est accessible par défaut (ce qui, souvenez-vous, signifie aucun modificateur d'accès explicite !), seul le code se trouvant dans le même package que cet élément (classe, variable, méthode, classe interne) peut y accéder.

Mais alors, à quoi sert *protected* ?

protected

L'accès protected est presque identique à l'accès par défaut à une exception près : il permet aux sous-classes d'hériter de l'élément protégé si ces sous-classes n'appartiennent pas au package de la superclasse qu'elles étendent. C'est tout. Voilà ce que ce niveau d'accès vous offre : la capacité d'avoir des sous-classes en dehors du package de la superclasse qui continuent à hériter des éléments de la classe, notamment des méthodes et des constructeurs.

La plupart des développeurs ne voient guère de raisons d'utiliser protected, mais certaines conceptions l'emploient et vous trouverez peut-être un jour que c'est exactement ce qu'il vous faut. L'un des aspects intéressants de protected est que — contrairement aux autres niveaux d'accès — il ne s'applique qu'à *l'héritage*. Si une sous-classe externe au package possède une *référence* à une instance de la superclasse et que la superclasse a par exemple une méthode protégée, elle ne peut pas accéder à cette méthode par l'intermédiaire de cette référence ! Elle ne peut y accéder que si elle en *hérite*. Autrement dit, la sous-classe externe au package n'accède pas à la méthode protégée, mais elle la possède par le biais de l'héritage.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

3 Méthodes de String et de StringBuffer/StringBuilder

Deux des classes les plus couramment utilisées dans l'API Java sont String et StringBuffer (nous avons vu au point 9 que les Strings sont immuables : un StringBuffer/StringBuilder peut donc être beaucoup plus efficace si vous manipulez une String). À partir de Java 5.0, vous devez utiliser String***Builder*** au lieu de String***Buffer***, à moins que vos manipulations n'impliquent des threads, ce qui n'est pas courant. Voici un bref aperçu des principales méthodes de ces classes :

String et StringBuffer/StringBuilder possèdent les méthodes suivantes :

```
char charAt(int indice);                    // retourne le caractère occupant la position donnée
int length();                               // retourne la longueur de la chaîne
String substring(int debut, int fin);       // extrait une partie de la chaîne
String toString();                          // retourne la valeur de la chaîne
```

Pour concaténer des chaînes :

```
String concat(string);                      // pour la classe String
String append(string);                      // pour les classes StringBuffer et StringBuilder
```

Méthodes de la classe String :

```
String replace(char ancien, char nouveau);  // remplace toutes les occurrences d'un caractère
String substring(int debut, int fin);       // extrait une portion d'une chaîne
char [] toCharArray();                      // convertit en tableau de caractères
String toLowerCase();                       // convertit tous les caractères en minuscules
String toUpperCase();                       // convertit tous les caractères en majuscules
String trim();                              // supprime les espaces des extrémités
String valueOf(char []) ;                   // transforme un tableau de caractères en chaîne
String valueOf(int i) ;                     // transforme une valeur primitive en chaîne
                                            // d'autres types primitifs sont pris en charge
```

Méthodes des classes StringBuffer et StringBuilder :

```
StringBxxxx delete(int debut, int fin);                    // supprime une portion de chaîne
StringBxxxx insert(int decalage, primitif ou un char []);  // insère un élément
StringBxxxx replace(int debut, int fin, String s);         // remplace une partie par une chaîne
StringBxxxx reverse();                                     // inverse le contenu du SB
void setCharAt(int indice, char ch);                       // remplace un caractère donné
```

Note : StringBxxxx = StringBuffer ou StringBuilder, selon la classe employée .

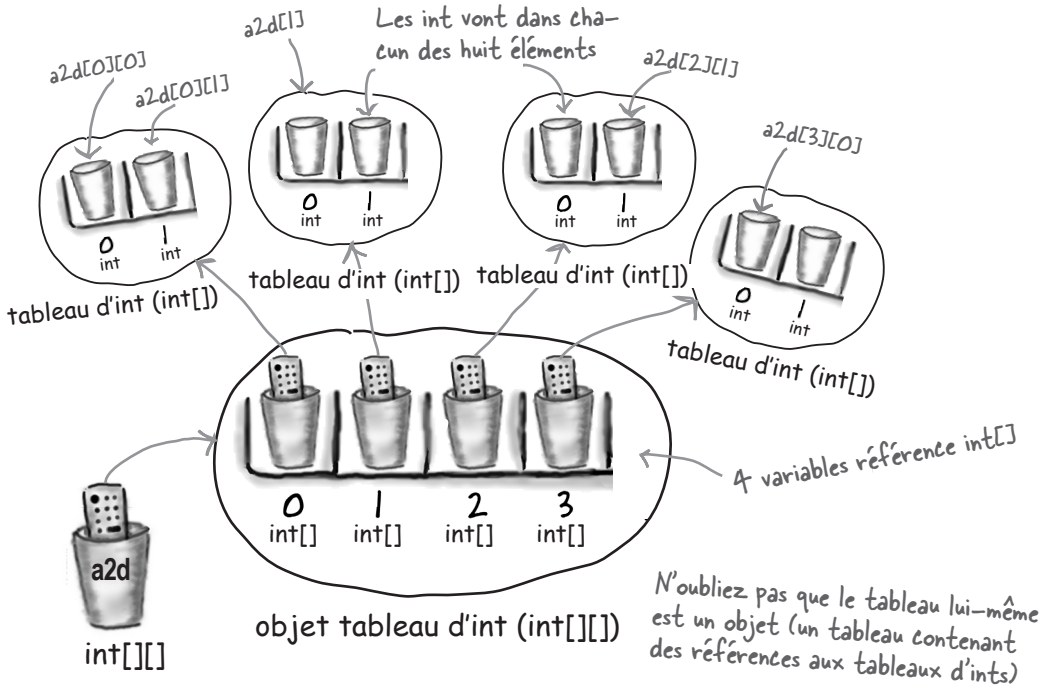
Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

2 Tableaux multidimensionnels

Dans la plupart des langages, si vous créez par exemple un tableau bidimensionnel de 4 x 2, vous visualisez un rectangle de 4 éléments sur 2 éléments, soit au total 8 éléments. Mais en Java, un tel tableau serait en réalité formé de *5* tableaux liés ensemble ! Un tableau bidimensionnel Java est simplement un tableau de tableaux. (Un tableau tridimensionnel est un tableau de tableaux de tableaux, mais nous vous laisserons le plaisir de le découvrir.) Voici comment ça marche.

```
int[][] a2d  = new int [4][2];
```

La JVM crée un tableau de 4 éléments. *Chacun de ces quatre* éléments est en réalité une variable référence qui pointe sur un tableau d'int (nouvellement créé) de 2 éléments.



Travailler avec des tableaux multidimensionnels

- Accéder au deuxième élément du troisième tableau : `int x = a2d[2][1]; // n'oubliez pas, on commence à 0 !`
- Créer une référence unidimensionnelle à l'un des sous-tableaux : `int[] copy = a2d[1];`
- Raccourci pour initialiser un tableau de 2 x 3 : `int[][] x = { { 2,3,4 }, { 7,8,9 } };`
- Créer un tableau bidimensionnel aux dimensions irrégulières :

```
int[][] y = new int [2][];      //  crée le premier tableau de 2 éléments
y[0] = new int [3];              // crée le premier sous-tableau de 3 éléments
y[1] = new int [5];              // crée le deuxième sous-tableau de 5 éléments
```


Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Et le sujet numéro 1, qui ne rentrait pas dans le livre...

1 Énumérations (alias Types énumérés ou Enums)

Nous avons parlé des constantes qui sont définies dans l'API, **`JFrame.EXIT_ON_CLOSE`**, par exemple. Vous pouvez également créer vos propres constantes en déclarant une variable **`static final`**. Mais vous voudrez parfois créer un ensemble de constantes pour représenter les seules valeurs valides d'une variable. On dit couramment que cet ensemble de valeurs valides est une *énumération*. Avant Java 5.0, la création d'une énumération était une cotte mal taillée. Depuis Java 5.0, vous pouvez créer des énumérations à part entière qui feront l'envie de tous vos amis qui utilisent des versions antérieures.

Qui est dans le groupe ?

Disons que vous êtes en train de créer un site web pour votre groupe favori et que vous voulez être certain que tous les commentaires soient adressés à un membre du groupe donné.

L'ancienne façon de simuler une énumération :

```
public static final int PAUL = 1;
public static final int JOEL = 2;
public static final int YANN = 3;

// plus loin dans le code

if (membreChoisi == PAUL) {
  // exécuter les méthodes relatives à Paul
}
```

Parvenu à ce stade, on espère que membreChoisi a une valeur valide !

La bonne nouvelle à propos de cette technique, c'est qu'elle rend le code VRAIMENT facile à lire. Autre bonne nouvelle : vous ne pourrez jamais changer la valeur de la fausse énumération que vous avez créée ; `PAUL` vaudra toujours `1`. La mauvaise, c'est qu'il n'y a aucun moyen satisfaisant ni facile d'être certain que la valeur de `membreChoisi` sera toujours `1`, `2` ou `3`. Si un fragment de code bien caché affecte `812` à `membreChoisi`, il est plus que probable que votre programme plantera...

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

1 Énumérations, suite

La même situation en utilisant une authentique énumération Java 5.0. Celle-ci est élémentaire, mais la plupart ne sont pas beaucoup plus compliquées.

Nouvelle énumération officielle :

```
public enum Membres { PAUL, JOEL, YANN };
public Membres membreChoisi;

// plus loin dan le code

if (membreChoisi == Membres.PAUL) {
  // exécuter les méthodes relatives à PAUL
}
```

On dirait bien une simple définition de classe, n'est-ce pas ? Eh bien oui, les énumérations SONT des classes spéciales. Ici, nous avons créé un nouveau énuméré nommé « Membres ».

La variable membreChoisi est de type Membres et ne peut prendre QUE les valeurs PAUL, JOEL ou YANN.

Inutile de se soucier de la valeur de cette variable

La syntaxe pour se référer à une « instance » d'une énumération.

Votre énumération étend java.lang.Enum

Quand vous créez une énumération, vous créez une nouvelle classe et vous ***étendez implicitement `java.lang.Enum`***. Vous pouvez la déclarer comme une classe isolée, dans son propre fichier source, ou comme un membre d'une autre classe.

Utiliser « if » et « switch » avec des énumérations

Avec l'énumération que nous venons de créer, nous pouvons écrire des branchements dans notre code en utilisant l'instruction `if` ou `switch`. Notez également qu'on peut en comparer des instances avec == ou avec la méthode `.equals()`. Généralement, on considère que == est un style plus correct.

```
Membres n = Membres.JOEL;
if (n.equals(Membres.PAUL)) System.out.println("Paul!");
if (n == Membres.JOEL) System.out.println("Bravo");

Membres siNom = Membres.YANN;
switch (siNom) {
  case PAUL: System.out.print("super ");
  case YANN: System.out.print("va plus loin ");
  case JOEL: System.out.println("encore ! ");
}
```

Affectation d'une valeur énumérée à une variable.

Les deux fonctionnent bien ! « Bravo » s'affiche.

Pop Quiz ! Quel est le résultat ?

Réponse : va plus loin encore !

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

#1 Énumérations, fin

Une version réellement contournée d'une énumération similaire

Vous pouvez ajouter une foule de choses à votre énumération, notamment un constructeur, des méthodes, des variables et quelque chose qu'on qualifie de corps de classe spécifique aux constantes. Ils ne sont pas courants, mais vous pourriez en rencontrer :

```
public class MusiqueEnum {

  enum Noms {
    PAUL("première guitare") { public String chante() {
                        return "plaintivement"; }
                    },
    JOEL("guitare rythmique") { public String chante() {
                          return "d'une voix rauque"; }
                      },
    YANN("basse");

    private String instrument;

    Noms(String instrument) {
      this.instrument = instrument;
    }
    public String getInstrument() {
      return this.instrument;
    }
    public String chante() {
      return "à l'occcasion";
    }
  }

  public static void main(String [] args) {
    for (Noms n : Noms.values()) {
      System.out.print(n);
      System.out.print(", instrument: "+ n.getInstrument());
      System.out.println(", chante: " + n.chante());
    }
  }
}
```

Argument transmis au constructeur déclaré après.

Voici le « corps de classe spécifique aux constantes ». Imaginez qu'il redéfinissent la méthode de base (en l'occurrence la méthode chante()), si chante() et appelée sur une variable dont la valeur est PAUL ou JOEL.

Voici le constructeur de l'énumération. Il s'exécute une fois pour chaque valeur déclarée (dans ce cas, il s'exécute trois fois).

Ces méthodes ont appelées à partir de main().

Toute énumération possède une méthode values() qui est généralement utilisée dans une boucle for comme ici.

```
Fichier Edition Fenêtre Aide Gratter
%java MusiqueEnum

PAUL, instrument: première guitare, chante: plaintivement
JOEL, instrument: guitare rythmique, chante: d'une voix rauque
YANN, instrument: basse, chante: à l'occcasion
%
```

Remarquez que la méthode chante() de base n'est appelée que lorsque une valeur n'a pas de corps de classe spécifique aux constantes.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

La clé du mystère

Un long voyage de retour

Jacques Long, le capitaine du Téméraire avait reçu du quartier général une transmission urgente et top secrète. Le message contenait trente codes de navigation au chiffrement prétendument à toute épreuve dont le Téméraire aurait besoin pour calculer sa route de retour à travers les secteurs ennemis. Originaires d'une galaxie voisine, les Hackariens avaient mis au point un rayon diabolique destiné à brouiller les codes et capable de créer des objets fantômes sur le tas du seul et unique ordinateur de bord du Téméraire. En outre, le rayon étranger pouvait modifier des variables de références valides afin qu'elles pointent sur ces objets fantômes. La seule défense dont disposait l'équipage du Téméraire contre le maléfique rayon hackarien consistait à exécuter en ligne un antivirus encapsulable dans le code Java 1.4 dernier cri du vaisseau.

Le capitaine Long donna à l'enseigne Martin les instructions suivantes pour traiter ces codes d'importance stratégique :

« Placez les cinq premiers dans un tableau de type CleParsec. Placez les 25 codes suivants dans un tableau bidimensionnel de cinq sur cinq de type CleQuadrant. Passez ces deux tableaux à la méthode calculerRoute() de la classe publique finale Navigation. Une fois l'objet route retourné, exécutez l'antivirus en ligne sur toutes les variables référence du programme puis exécutez le programme SimNav et apportez-moi les résultats. »

Quelques minutes plus tard, l'enseigne Martin revint avec les résultats de SimNav. « Résultat de SimNav prêt pour la revue, monsieur », déclara l'enseigne Martin. « Bien », répondit le capitaine, « expliquez-moi ce que vous avez fait ». « Oui monsieur ! », rétorqua l'enseigne. « J'ai d'abord déclaré et construit un tableau de type CleParsec avec le code suivant : CleParsec [] p = new CleParsec[5]; Puis j'ai déclaré et construit un tableau de type CleQuadrant avec le code suivant : CleQuadrant [] [] q = new CleQuadrant [5] [5];. Ensuite j'ai chargé les cinq premiers codes dans le tableau CleParsec avec une boucle for, et les 25 derniers dans le tableau CleQuadrant avec des boucles for imbriquées. Enfin, j'ai exécuté l'antivirus sur chacune des 32 variables références, une pour le tableau CleParsec et cinq pour ses éléments, puis une pour le tableau CleQuadrant et 25 pour ses éléments. Une fois que l'antivirus a eu signalé qu'il ne détectait aucun problème j'ai exécuté SimNav et repassé l'antivirus par acquis de conscience… Monsieur ! »

Le capitaine Long jeta à l'enseigne un long regard glacial et dit calmement « Enseigne, vous êtes consigné dans vos quartiers pour avoir mis en danger la sécurité de ce bâtiment. Je ne veux plus vous revoir sur ce pont tant que votre Java ne sera pas au point ! Lieutenant Boole, remplacez l'enseigne et faites ce travail correctement ! ».

Pourquoi le capitaine a-t-il consigné l'enseigne dans ses quartiers ?

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Solution de la Clé du mystère

Un long voyage de retour

Le capitaine Long savait que les tableaux multidimensionnels Java sont en réalité des tableaux de tableaux. Le tableau de CleQuadrant nommé «q» nécessitait 31 variables référence pour qu'on puisse accéder à tous ses composants :

1 - variable pour «q»

5 - variables pour q[0] - q[4]

25 - variables pour q[0][0] - q[4][4]

L'enseigne avait oublié les variables référence pour les cinq tableaux unidimensionnels imbriqués dans le tableau «q». N'importe laquelle de ces cinq variables aurait pu être corrompue par le rayon hackarien et le test de l'enseigne n'aurait jamais révélé le problème.

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Index

Symboles

&, &&, |. || (opérateurs booléens) 151, 660

&, <<, >>, >>>, ^, |, ~ (opérateurs bit à bit) 660

. (opérateur point) 36
    référence 54

+ (opérateur de concaténation de chaînes) 17

++ -- (incrémentation/décrémentation) 105, 115

<, <=, ==, !=, >, >= (opérateurs de comparaison) 86, 114, 151

<, <=, ==, >, >= (opérateurs de comparaison) 11

A

aboyez autrement 73

abstraites (classes) 200–210
    modificateurs de classes 200

abstraites (méthodes)
    déclaration 203

accès
    et héritage 180
    modificateurs de classes 667
    modificateurs de méthodes 81, 667
    modificateurs de variables 81, 667

accesseurs et mutateurs. *Voir* get et setters

accolades 10

ActionListener (interface) 358, 358–361

addActionListener() 359–361

adresses IP. *Voir* réseau

Aeron™ 28

affectations
    types primitifs 52
    variables références 55, 57, 83

analyse d'un entier. *Voir* enveloppes

analyse d'un texte avec String.split() 458

animation 382–385

annexe A 649–658
    BeatBox final : le client 650
    BeatBox final : le serveur 657

annexe B
    assertions 662
    immuabilité 661
    invocations liées 664
    manipulation des bits 660
    méthodes de String et StringBuffer/Builder 669
    niveaux d'accès et modificateurs 667
    portée de bloc 663
    tableaux multidimensionnels 670

API 154–155, 158–160

arguments
    des méthodes 74, 76, 78
    polymorphes 187

ArrayList 132, 133–138, 156, 208, 558
    API 532
    ArrayList<Object> 211–213
    autoboxing 288–289
    transtypage 229

ArrayList 532

assertions 662

astuces métacognitives 33, 108, 325

atomicité 510–512. *Voir aussi* threads

audio. *Voir* midi

auditeurs
    interfaces 358–361

A-UN 177–181

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

autoboxing 288–291
affectations 291
et opérateurs 291

B

baignoire 177
BeatBox 316, 347, 472. *Voir* aussi annexe A
bière 14
bit à bit (opérateurs) 660
boolean 51
booléennes (expressions) 11, 114
BorderLayout 370–371, 401, 407
boucles, 10
break 105
for 105
while 115
BoxLayout411
break (intruction) 105
BufferedReader 454, 478
BufferedWriter 453
buffers 453, 454
byte 51
bytecode 2

C

café 51
Calendar
méthodes 305
Calendar 303–305
catch 338
champ de texte (JTextField) 413
char 51
classes
abstraites 200–210
concrètes 200–210
conception 34, 41, 79
finales 283
noms pleinement qualifiés 154–155, 157
classes internes
explication 376–386
événements 379
trinité de classes internes 381
clé du mystère. *Voir* jeux
client/serveur 473
code prêt à l'emploi 112, 152–153, 520
collections 137, 533, 558
API 558
ArrayList 137
ArrayList<Object> 211–213
Collections.sort() 534, 539
HashMap 533
HashSet 533
LinkedHashMap 533
LinkedList 533
List 557
Map 557, 567
Set 557
TreeSet 533
types paramétrés137
Collections.sort() 534, 539
Comparator 551
compare() 553
Comparable 547, 566
et TreeSet 566
compareTo() 549
Comparator 551, 566
et TreeSet 566
compare() 553
comparer avec == 86
compareTo() 549
compilateur 2, 18
javac -d 590
comportement 73

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

compte chèques. *Voir* Sylvie et Bruno

concaténation 17

concrètes (classes) 200–210

conditionnelles (expressions) 10, 11, 13

conseil du jour
  client 480
  serveur 484

constantes 282

constantes d'IHM
  ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER 415
  ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS 415

constructeurs 240
  chaînage 250–256
  surchargés 256
  de la superclasse 250–256

contrats 190–191, 218

contrôle de flot
  exceptions 326

coulez un point com 96–112, 139–150

coupe-circuit (opérateurs) 151

D

dates
  Calendar 303
  formatage 301
  GregorianCalendar 303
  java.util.Date 303
  méthodes 305

déclaration de variables 50
  d'instance 84
  primitives 51
  références 54

déclarations
  variables d'instance 50
  exceptions 335–336

défaut (accès par) 668

défaut (valeur par) 84

défilement (JScrollPane) 414

déploiement 582, 608

désérialisation 441. *Voir* aussi sérialisation

devinettes 38

discussion (client) 486
  avec threads 518

discussion (serveur) 520

distant (contrôle) 54, 57

distante (interface). Voir RMI

docteur 169

double 51

E

E/S
  avec sockets 478
  BufferedReader 454, 478
  BufferedWriter 453
  buffers 453
  désérialisation 441
  FileInputStream 441
  FileOutputStream 432
  FileWriter 447
  flots 433, 437
  InputStreamReader 478
  ObjectInputStream 441
  ObjectOutputStream 432, 437
  sérialisation 432, 434–439, 437, 446, 460

écriture. Voir E/S

égalité 560
  et hashCode() 561

encapsulation 79–82
  avantages

énumérations 671–672

enums 671–672

enveloppes 287
  autoboxing 288–289
  Integer.parseInt() 104, 106, 117
  utilitaires de conversion 292

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

equals()
classe Object 209
définition 209

EST-UN 177–181, 251

événements
gestion 357–361
objets événements 361
interfaces auditeurs 358–361
classes internes 379
sources 359–361

exceptions 320, 325, 338

catch 321, 338
compilation vs exécution 324
contrôle de flot 326
déclaration 335–336
esquive 335–336
exceptions distantes 616
exceptions multiples 329, 330, 332
finally 327
gestion ou déclaration 337
lancement 323–326
multiples 329, 332
propagation 335–336
try 321, 338
vérification par le compilateur 324

exercices
le frigo 20, 43, 64, 119, 312, 349, 467, 524–525
pot de miel 267
quel gestionnaire d'agencement ? 424
quelle est la déclaration ? 231
quelle est l'image ? 230
qui suis-je ? 45, 89, 394
vous êtes... 88, 118, 266, 310, 395
vrai ou faux 311, 348, 466, 602

expressions booléennes 11, 114

Extreme Programming 101

F

fiabilité de type 540
et génériques 540

fichiers
classe File452
écriture 432, 447
lecture 441, 454
structure des fichiers source 7

fichiers source
structure 7

File (classe) 452

File 452

FileInputStream 441. *Voir aussi* E/S

FileOutputStream 432

FileReader 454. *Voir aussi* E/S

FileWriter 447

fille rêveuse
classes internes 375
Java Web Start 596

fin du livre 648

finales
classes 189, 283
méthodes 189, 283
variables statiques 282
variables 282, 283

finally 327

float 51

flots 433. *Voir aussi* E/S

FlowLayout 403, 408–410

for 105

formatage
arguments 300
dates 301–302
nombres 294–295
printf() 294
spécificateurs de format 295–296
String.format() 294

G

génériques 540, 542, 568–574
méthodes 544
jokers 574

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

gestionnaires d'agencement 401–412
BorderLayout 370–371, 403, 407
BoxLayout 403, 411
FlowLayout 403, 408–410

get et set 79

girafe 50

graphes d'objets 436, 438

graphismes 364–366. *Voir aussi* IHM
classe Graphics2D 366
objet Graphics 364

GregorianCalendar 303

H

hashCode() 561

HashMap 533, 558

HashSet 533, 558

Hashtable 558

héritage
animaux 170–175
définition 31, 166–192
et classes abstraites 201
ET-UN 214, 251
mot-clé super 228
multiple 223

I

if (instruction) 13

if-else 13

IHM
animation 382–385
BorderLayout 370–371, 401, 407
boutons 405
BoxLayout 403, 411
cadres 400
composants 354, 363–368, 400
constantes 415
défilement (JScrollPane) 414
définition 354, 400
FlowLayout 403, 408
gestion des événements 357–361, 379
gestionnaires d'agencement 401–412
graphismes 363–367
ImageIcon 365
interface auditeurs 358–361
JButton 400
JLabel 400
JPanel 400, 401
JTextArea 414
JTextField 413
méthodes 364-365
Swing 354
widgets

immuabilité des chaînes

implements 224

import (instruction)155, 157

imports statiques 307

incrémentation 105

initialisation
types primitifs 84
variables d'instance 84
variables statiques 281

InputStreamReader 478

instanciation. *Voir* objets

int 50
type primitif 51

Integer. *Voir* enveloppes

interfaces
définition 219–227
implémentation 224, 437
implémentation de plusieurs interfaces 226
java.io.Serializable 437
sérialisation 437

invocations liées 664

invocations de méthode virtuelles 175

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

J

J2EE 631

JAR (fichiers)
avec Java Web Start 598
commandes de base 593
exécutables 585–586, 592
exécution 586, 592
manifeste 585
outil 593

JAR exécutables 585–586, 586
avec packages 592, 592–593

Java 5, 6

Java en concentré 158–159

Java Web Start 597–601
fichiers jnlp 598, 599

javac. *Voir* compilateur

JCheckBox 416

jeux
la clé du mystère 92, 527, 674
mots-croisés 22, 120, 162, 350, 426, 603
la piscine 24, 44, 65, 91, 194, 232, 396

Jini 632–635

JNLP 598
fichiers jnlp 598, 599

jokers 574

JPEG 365

JVM 2, 18

JWS. *Voir* Java Web Start

L

lapin 50

lingerie, exceptions 329

LinkedHashMap 533, 558

LinkedHashSet 558

LinkedList 533, 558

List 557

littérales, affectation de valeurs
type primitifs 52

locales
variables 85, 236, 236–238, 258–263

long 51

losange de la mort 223

M

main() 9, 38

manifeste 585

Map 557, 567

Math
méthodes 274–278, 286
random() 111

mémoire
ramasse-miettes 260–263

méthodes
abstraites 203
arguments 74, 76, 78
arguments génériques 544
explication 34, 78
finales 283
redéfinition 32, 167–192
retour 75, 78
statiques 274–278
sur la pile 237
surcharge 191

méthodes d'IHM
drawImage() 365
fillOval() 365
fillRect() 364
gradientPaint(). *Voir aussi* IHM
paintComponent() 364
setColor() 364
setFont() 406

midi 317, 340–346, 387–390

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

mises à jour perdues. *Voir* threads
mnémoniques 53, 87, 157, 179, 227, 278
modificateurs
  de classes 200
  de méthodes 203
mots réservés 53
mots-clés 53
musique. *Voir* midi
mystère. *Voir* jeux

N

navigateur de services universel 636–648
new 55
nombres
  formatage 294–295
nommage 53. *Voir aussi* RMI
  classes et interfaces 154–155, 157
  collisions 587
  packages 587
noms pleinement qualifiés 154, 157
  packages 587
null
  référence 262

O

Object (classe)
  définition 208–216
  equals() 561
  hashCode() 561
ObjectOutputStream 432, 437
objets
  comparaison 209
  création 55, 240–256
  définition 55
  égalité 560
  equals() 209, 561
  ramasse-miettes 260–263
  tableaux 59, 60, 83
  vie et mort 258–263
  verrous 509
objets abandonnés. *Voir* ramasse-miettes
objets bizarres 200
OO
  A-UN 177–181
  conception 34, 41, 79, 166–191
  contrats 190–191, 218
  EST-UN 177–181, 251
  héritage 166–192
  interfaces 219–227
  losange de la mort 223
  polymorphisme 183, 183–191, 206–217
  redéfinition 167–192
  superclasse 251–256
  surcharge 191
opérateurs
  bit à bit 660
  comparaison 151
  conditionnels 11
  coupe-circuit 151
  décalage 660
  décrémentation 115
  et autoboxing 291
  incrémentation 105, 115
  logiques 151
ordonnancement des threads 496–498

P

packages 154–155, 157, 587–593
  organisation du code 589
  structure des répertoires 589
paintComponent() 364–368
paramètres. *Voir* arguments
passage par copie. *Voir* passage par valeur
passage par valeur 77

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

phrase-o-matic 16

pile
  et tas 236
  méthodes sur 237
  portée 236
  threads 490
  trace 323

piscine. *Voir* jeux

point (opérateur) 54

polices 406

polymorphisme 183–191
  arguments et types de retour 187
  classes abstraites 206–217
  et exceptions 330
  références de type Object 211–213

portée
  variables 236–238, 258–263
  de bloc 663

ports 475

pré-code 99–102

primitifs. *Voir* types primitifs

printf() 294

PrintWriter 479

private
  modificateur d'accès 81

protected 668

public
  modificateur d'accès 81, 668

Q

QuizCartes 448, 448–451

R

ramasse-miettes, 40
  annulation des références 58
  objets éligibles 260–263
  réaffectation de références 58
  tas 57, 58

random() 111

recettes de code
  BeatBox, sauvegarde et restauration 462
  BeatBox, version finale. *Voir* annexe A
  création d'une IHM 418
  jouer un son 339
  musique et graphismes 386

redéfinition
  définition 32, 167–192
  polymorphisme. *Voir* polymorphisme

redéfinition des méthodes 563

références d'objets 54, 56
  affectation 55, 262
  annulation 262
  comparaison 86
  égalité 560
  polymorphisme 185–186
  transtypage 216

registre RMI 615, 617, 620

répertoires
  packages 589
  servlets 626

réseau
  explication 473
  ports 475
  sockets 475

risque 319–336

RMI
  client 620, 622
  compilateur 618
  définition 614–622
  exceptions distantes 616
  implémentations distantes 615, 617
  interfaces distantes 615, 616

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

RMI *(suite)*
Jini. Voir aussi Jini
Naming.lookup() 620
Naming.rebind().
navigateur de services universel 636–648
registre 615, 617, 620
rmic 618
squelettes 618
souches 618
UnicastRemoteObject 617

rmic. *Voir* RMI

run()
redéfinition dans l'interface Runnable 494

Runnable 492
état d'un thread 495
run() 493, 494
threads 493

S

séquenceur midi 340–346

sérialisation 434–439, 446
désérialisation 460
écriture 432
graphes d'objets 436
interface 437
lecture. *Voir* E/S
ObjectInputStream. *Voir* E/S
objectOutputStream 432
objets 460
restauration 460. *Voir aussi* E/S
sauvegarde 432
serialVersionUID 461
transient 439
versionnage 460, 461

Server
sockets 483. *Voir aussi* sockets

servlets 625–627

Set 557
importance de equals() 561
importance de hashCode() 561

short 51

sleep() 501–503

snowboard 214

sockets
adresses 475
création 478
définition 475
E/S 478
écriture 479
lecture 478
ports 475
serveur 483
TCP/IP 475

son 317, 340

souche. *Voir* RMI

sous-classes 31, 166–192

spécificateurs
d'arguments 300
de format 295, 298

squelettes. *Voir* RMI

statiques
imports statiques 307
initialiseurs 282
méthodes 274–278
méthodes de la classe Math 274–278
types énumérés 671
variables 282

String
analyse 458
concaténation 17
méthodes 669
String.format() 294–297
String.split() 458
tableaux 17

StringBuffer/StringBuilder
méthodes 669

super 31, 228

superclasse 166–192, 214–217, 228

super-constructeur 250–256

surcharge 191
constructeurs 256

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Swing. *Voir* IHM

Sylvie et Bruno 505–506

synchronized
méthodes synchronisées 510. *Voir aussi* threads

syntaxe 10, 12

System.out.print() 13

T

tableaux 17, 59, 135
affectation 59
attribut length 17
création 60
de types primitifs 59
déclarations 59
d'objets 60, 83
multidimensionnels 670
vs ArrayList 134–137

tableaux multidimensionnels 670

tas
explication 40, 57, 236–238
ramasse-miettes 40, 57, 58

TCP, ports 475

tests
Extreme Programming 101

texte
analyse avec String.split() 458 458
lecture dans un fichier. *Voir aussi* E/S
écriture dans un fichier 447

Thread.sleep() 501–503

threads
définition 489–515
démarrage 492
en sommeil 501–503
états 495, 496
imprévisibilité 498–499
mises à jour perdues 512–514
ordonnancement 496, 496–498
pile 490–491
résumé 500, 517
run() 493, 494
Runnable 492, 493, 494
sleep() 501–503
start() 492
Sylvie et Bruno 505–507
synchronisation 510–512
verrous 509
verrous mortels 516

throw
exceptions 323–326
throws 323–326

transient 439

transtypage
explicite (primitifs) 117
explicite (référence) 216
implicite (primitifs) 117
transtypage 216

TreeMap 558

TreeSet 533, 558, 564–566, 566

tri
Collections.sort() 534, 539, 547
Comparator 551, 553
interface Comparable 547, 549
TreeSet 564–566

try
blocs 321, 326
try 321

type 50
paramètres de 137, 542, 544

types de retour
définition 75
polymorphes 187
valeurs 78

types paramétrés 137

types primitifs 53
autoboxing 288–289
boolean 51
byte 51
char 51
double 51
float 51

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

types primitifs *(suite)*
int 51
intervalles 51
opérateur == 86
short 51
transtypage explicite
type 51

V

variables
affectation 52, 262
annulations 262
déclaration 50, 54, 84, 236–238
locales 85, 236–238
primitives 51, 52
références 54, 55, 56, 185–186
scope 236–238
statiques. *Voir* statiques

variables d'instance
définition 34, 73
déclaration 84
valeur par défaut 84
initialisation 84
durée de vie et portée 258–263
vs. variables locales 236–238, 239
vs. variables statiques 277

variables référence. *Voir* références d'objets

verrou mortel 516

verrous
objets 509
threads 509

Vieilles charrues 30

ville fantôme109

W

while 11, 115

widgets
JButton 354, 405
JCheckBox 416
JFrame 354, 400, 401
JList 417
JPanel 400, 401
JScrollPane 414, 417
JTextArea 414
JTextField 413

Z

zone de texte (JTextArea) 414

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>

Ceci n'est pas un au revoir

Vous ne connaissez pas le site web ? Vous y trouverez notamment le code du livre. Dites bonjour aux auteurs pour nous.

Visitez le site

digitbooks**.fr/catalogue/**9782815000079**.html**

Propriété de Andre MAFFEI <andre.maffei@yahoo.fr>