

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.

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 œuvres 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 informatiques 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.



	Cyber BeatBox															
Grosse caisse	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
Charleston fermé	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	
Charleston ouvert	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
Caisse claire	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
Cymbale crash	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
Clap	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
Tom aigu	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
Bongos	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
Maracas	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
Sifflet	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
Conga basse	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
Cloche	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
Claqué	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
Tom médium basse	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
Agogo	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
Conga ouvert	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	

Jouer Arrêter Accélérer Ralentir Envoyer

rythme dansant

Kevin0: rythme dansant
Léo0: rythme #2
Kevin1: rythme #2 revu
Chris0: rythme #2 encore plus dansant

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.

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

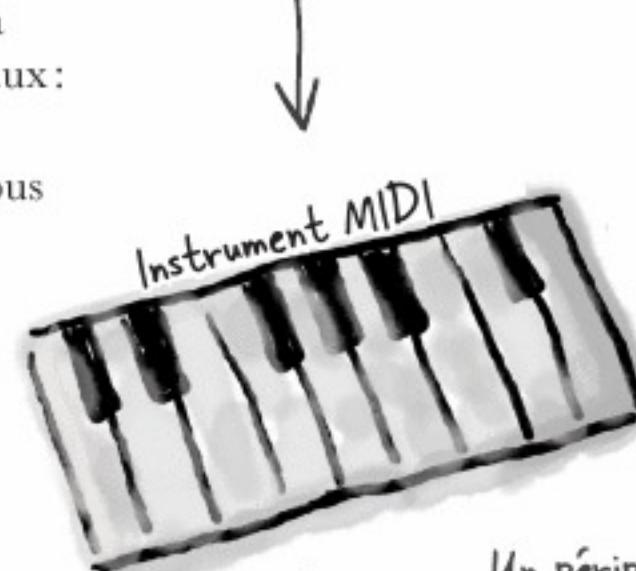
L'API JavaSound

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*.



Un fichier MIDI contient des informations sur la façon de jouer un morceau, mais il ne contient aucune donnée son. C'est comme une sorte de rouleau à musique pour un piano mécanique.

Un périphérique MIDI sait comment «lire» un fichier MIDI et restituer le son. Ce périphérique peut être un clavier ou toute autre sorte d'instrument. En général, un instrument MIDI peut jouer un TAS de sons différents (piano, batterie, violon, etc.), et les jouer tous en même temps. Un fichier MIDI n'est donc pas une partition pour un musicien isolé ; il contient les partitions de TOUS les musiciens jouant un morceau donné.

Il nous faut d'abord un séquenceur

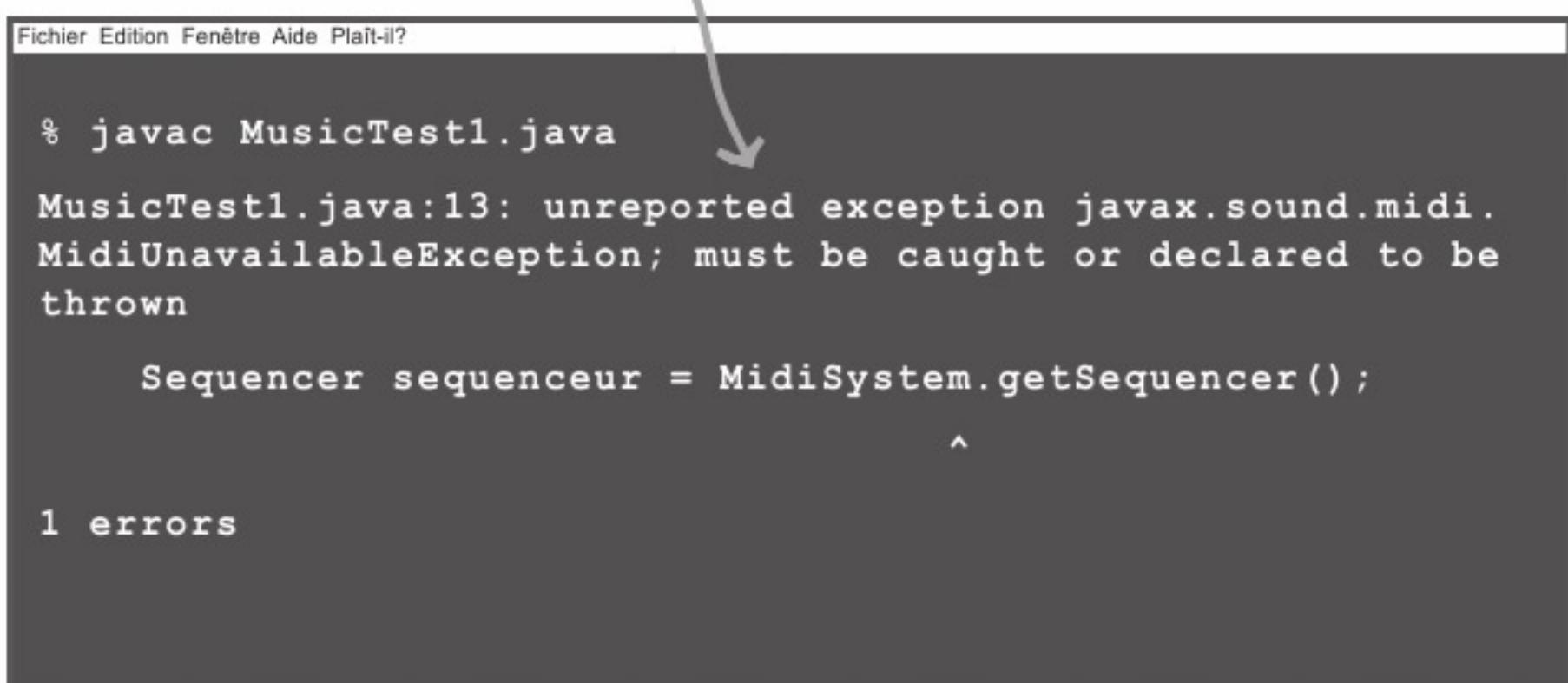
Avant de pouvoir jouer un son quelconque, il nous faut un objet Sequencer. Le séquenceur est l'objet qui reçoit toutes les données MIDI et les envoie aux bons instruments. C'est lui qui joue la musique. Un séquenceur peut faire beaucoup de choses différentes, mais, dans ce livre, nous l'utilisons strictement 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.*;           ← importation du package 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
```

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.



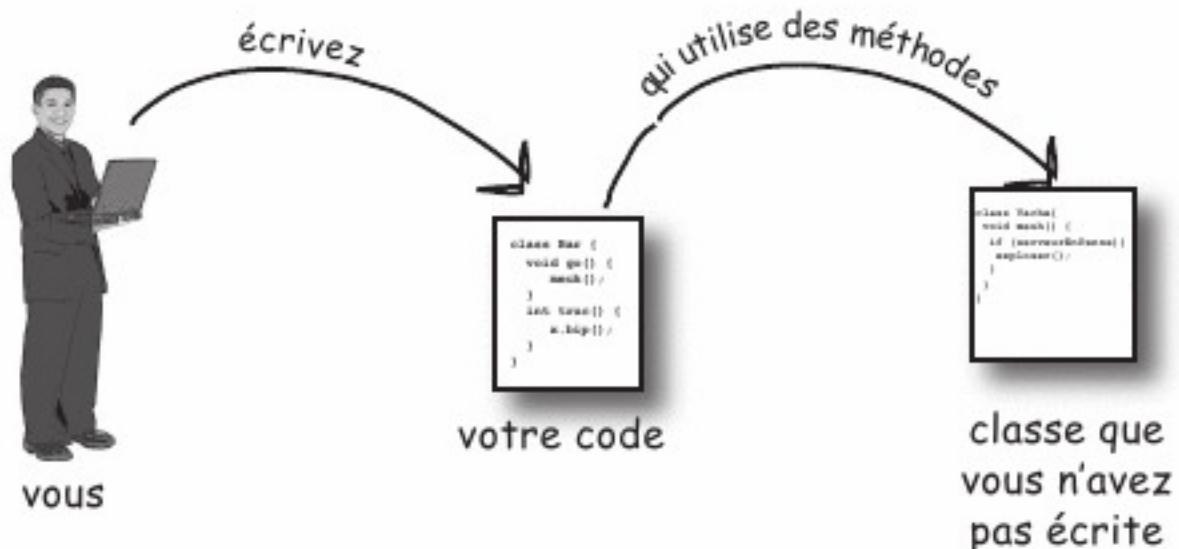
Fichier Edition Fenêtre Aide Plaît-il?

```
% 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
```

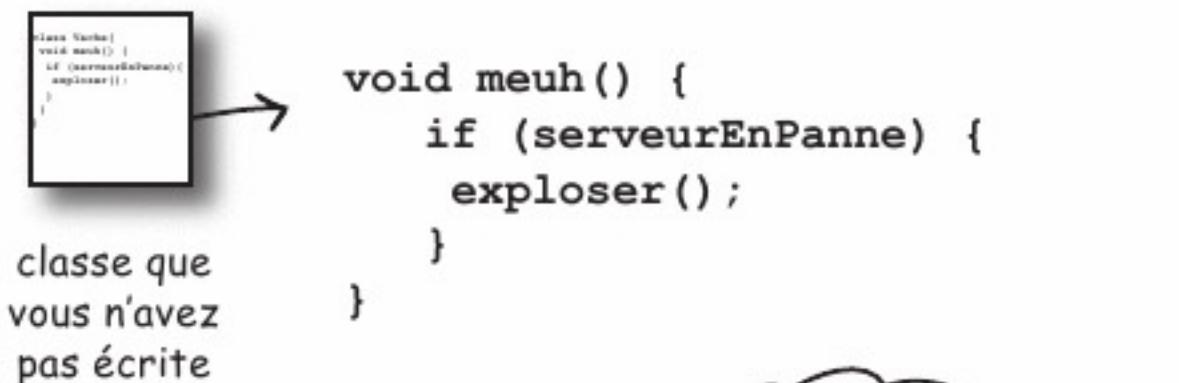
A screenshot of a terminal window. The title bar says "Fichier Edition Fenêtre Aide Plaît-il?". The command "% javac MusicTest1.java" is entered at the prompt. The output shows an error message: "MusicTest1.java:13: unreported exception javax.sound.midi.MidiUnavailableException; must be caught or declared to be thrown". Below this, the problematic line of code is shown: "Sequencer sequenceur = MidiSystem.getSequencer();". An arrow points from the explanatory text above to this line of code. The number "1 errors" is also visible at the bottom of the terminal window.

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.



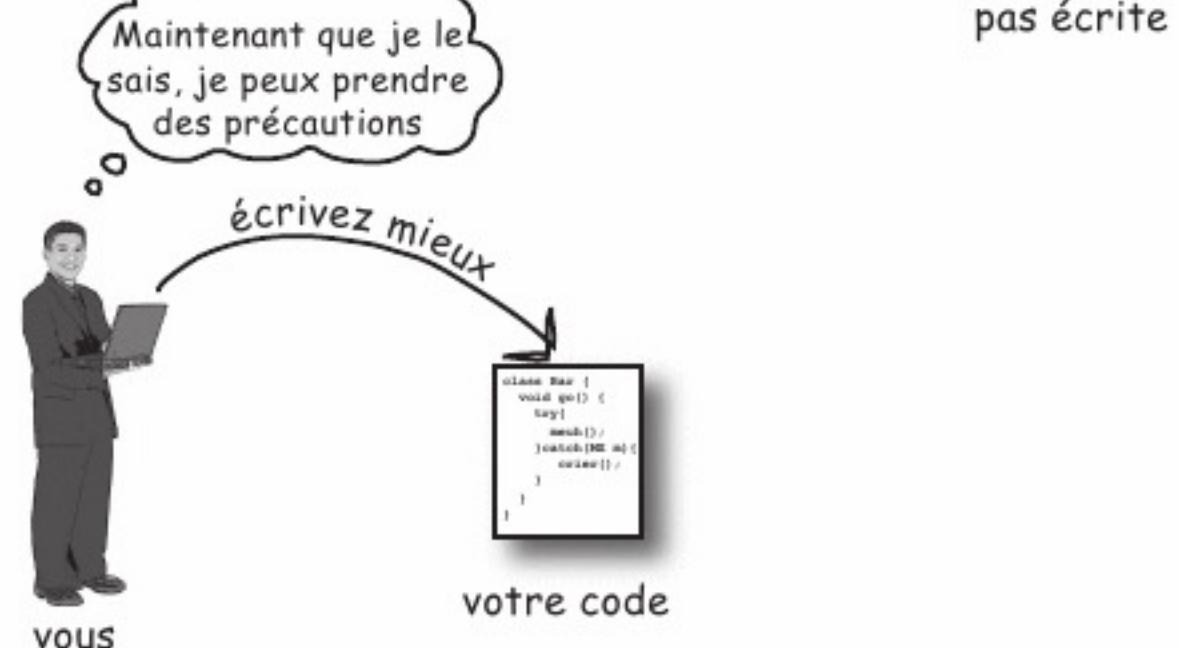
- ② Cette méthode fait quelque chose de risqué, qui peut ne pas fonctionner au moment de l'exécution



- 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.



quand il y a un risque d'erreur

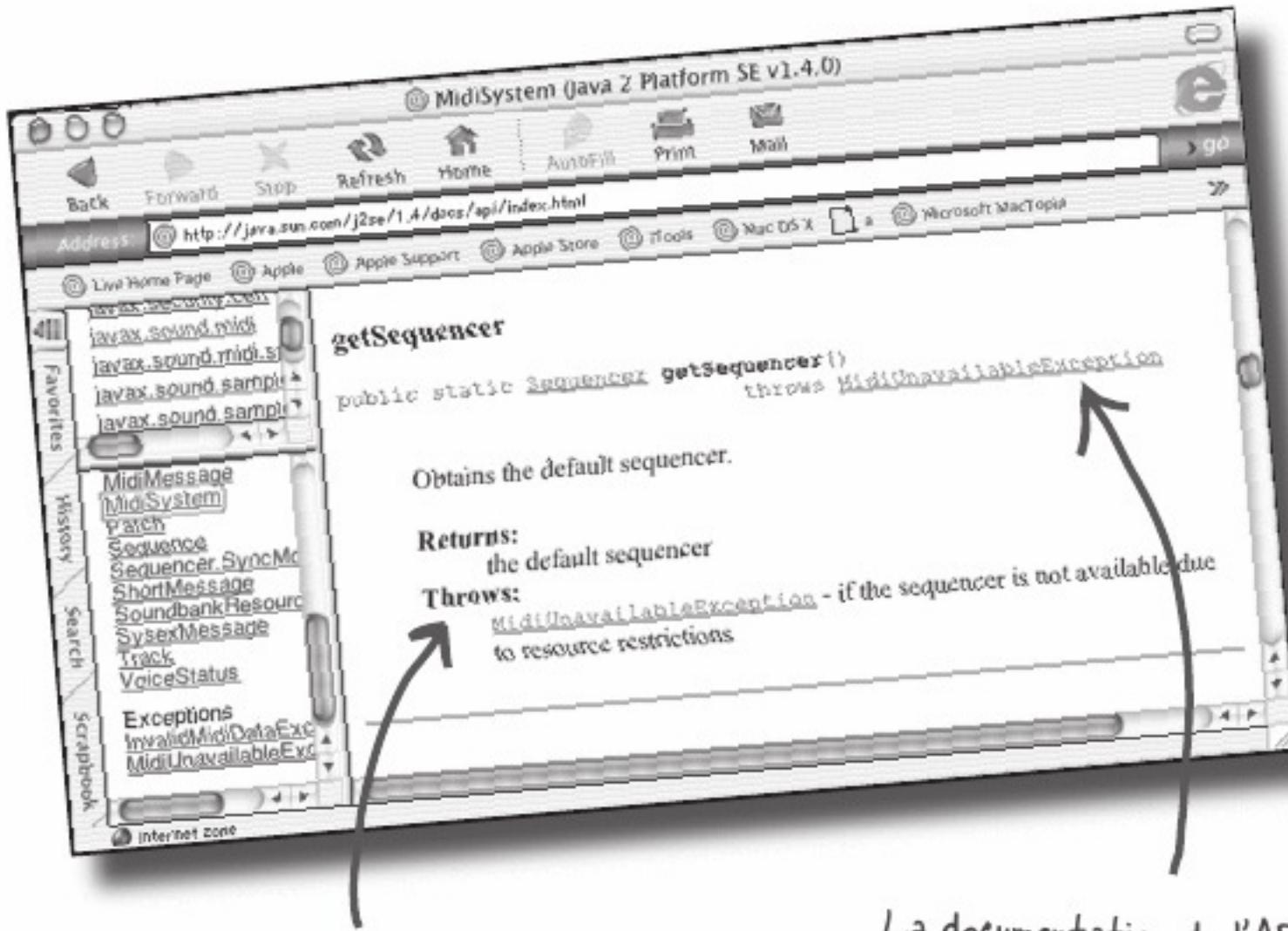
Les méthodes Java utilisent des *exceptions* pour dire au code appelant:

«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 vousappelez 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 lappelez.



Cette partie vous dit QUAND 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.

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 *saviez* 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.

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

public class MusicTest1 {
    public void jouer() {

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

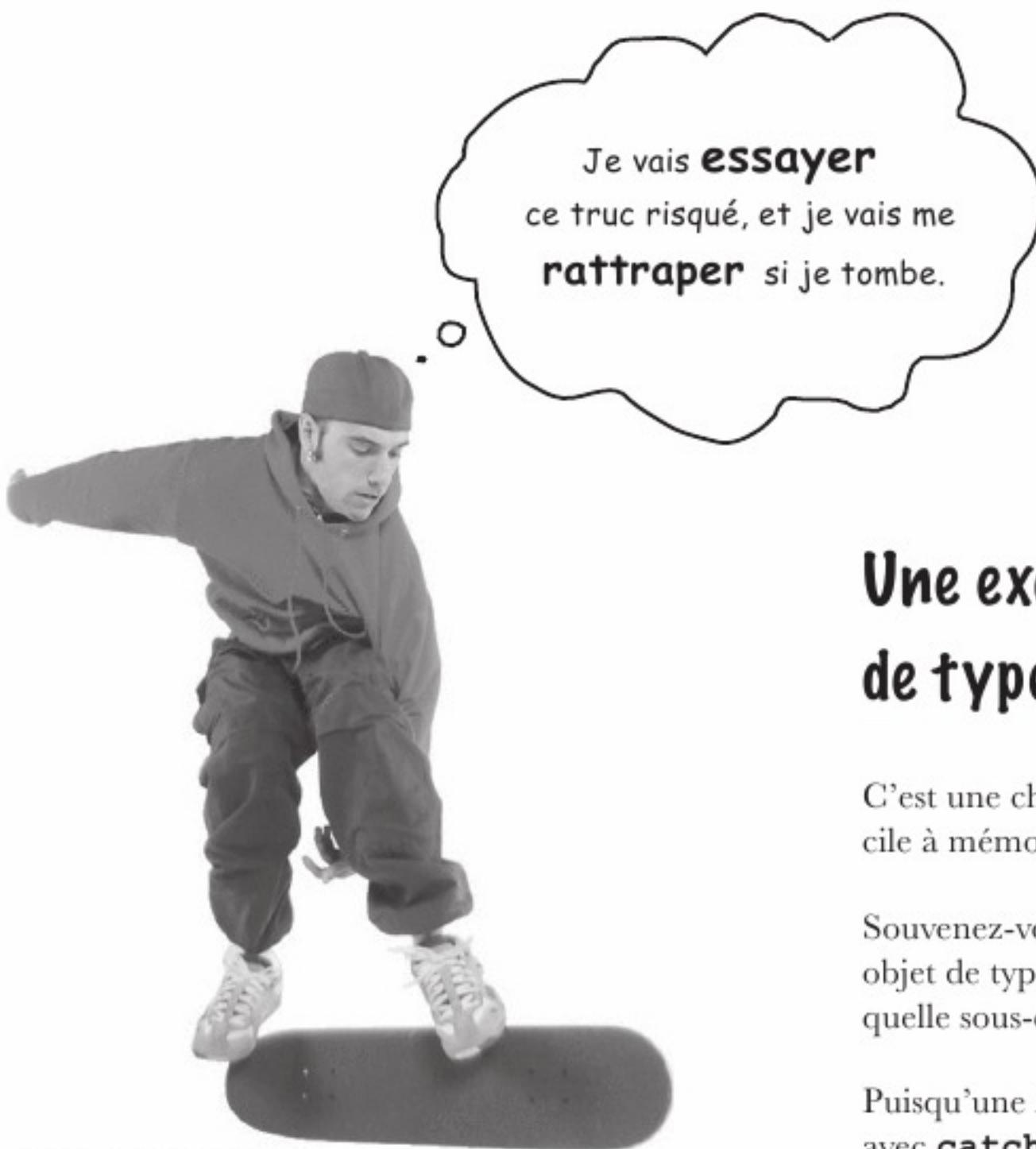
    public static void main(String[] args) {
        MusicTest1 mt = new MusicTest1();
        mt.play();
    } // fin de la méthode main()
} // fin de la classe
```

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,

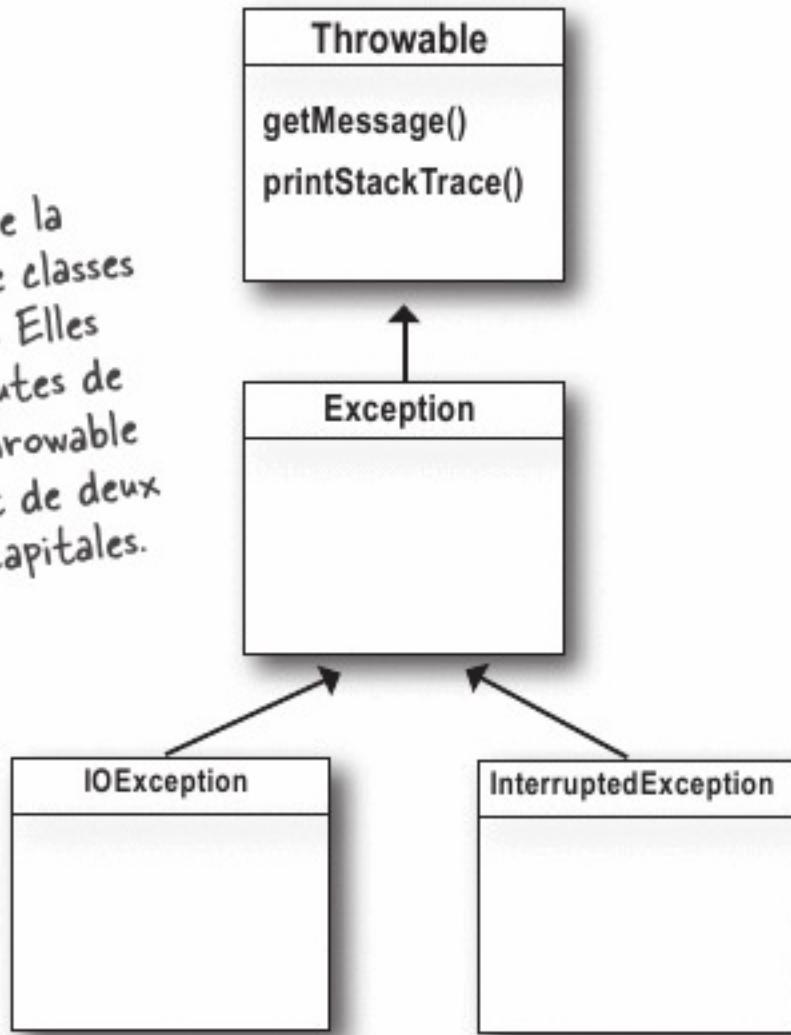
La vie est courte
(surtout sur le tas). Prenez
le risque. Essayez. Mais
au cas où les choses
tourneraient mal, faites bien
attention à intercepter tous
les problèmes avant que
l'enfer ne se déchaîne.

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().



N'essayez pas ceci chez vous.

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



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.

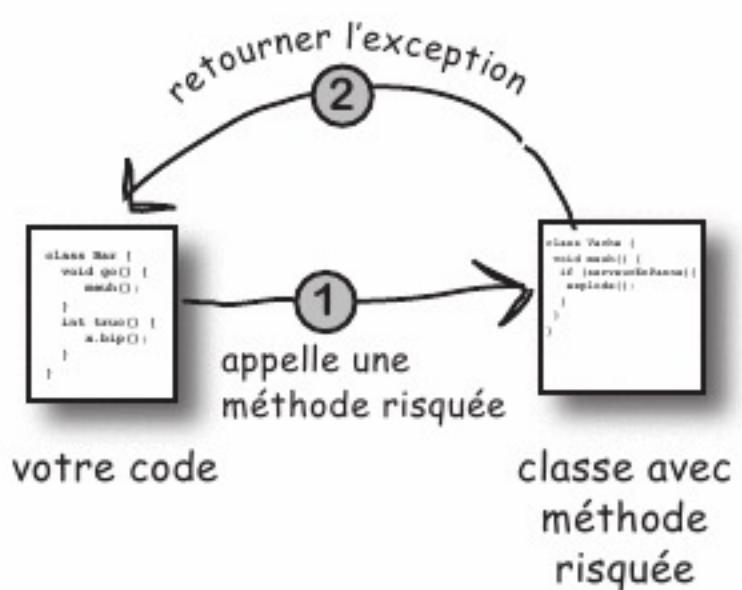
Ce que vous écrivez dans un bloc catch dépend de l'exception lancée. Si par exemple le serveur est en panne, vous pouvez utiliser le bloc catch pour essayer un autre serveur. Si un fichier est absent, vous pouvez demander à l'utilisateur de le chercher.

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:

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

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

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

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.

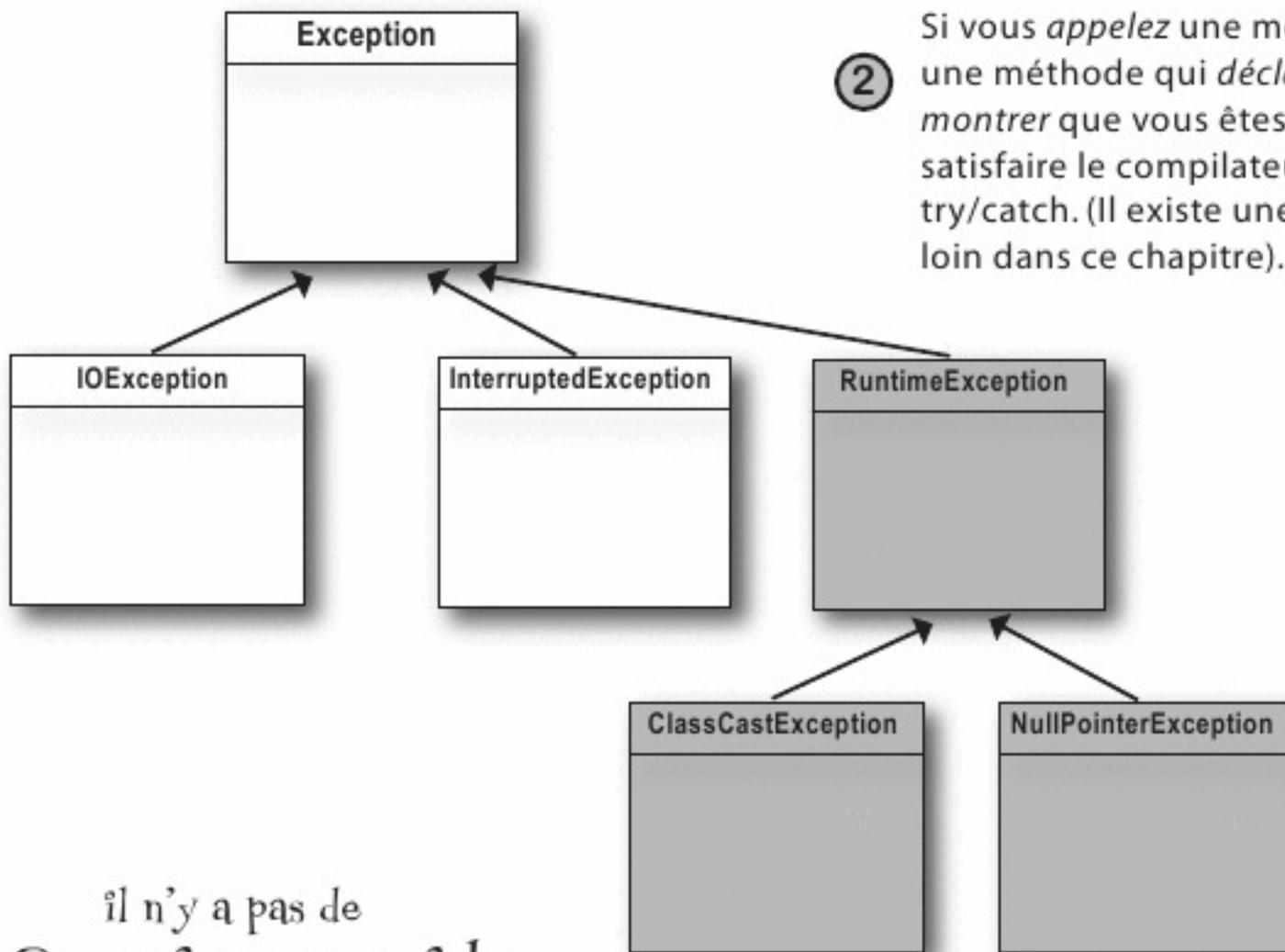
② Votre code qui appelle la méthode risquée:

```
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.

Le compilateur vérifie tout sauf les RuntimeExceptions.

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



il n'y a pas de
Questions stupides

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.

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.



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.



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

- vous connecter à un serveur distant
- 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

Problème possible

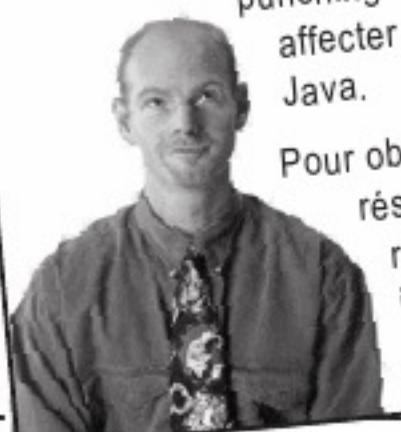
le serveur est en panne

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.

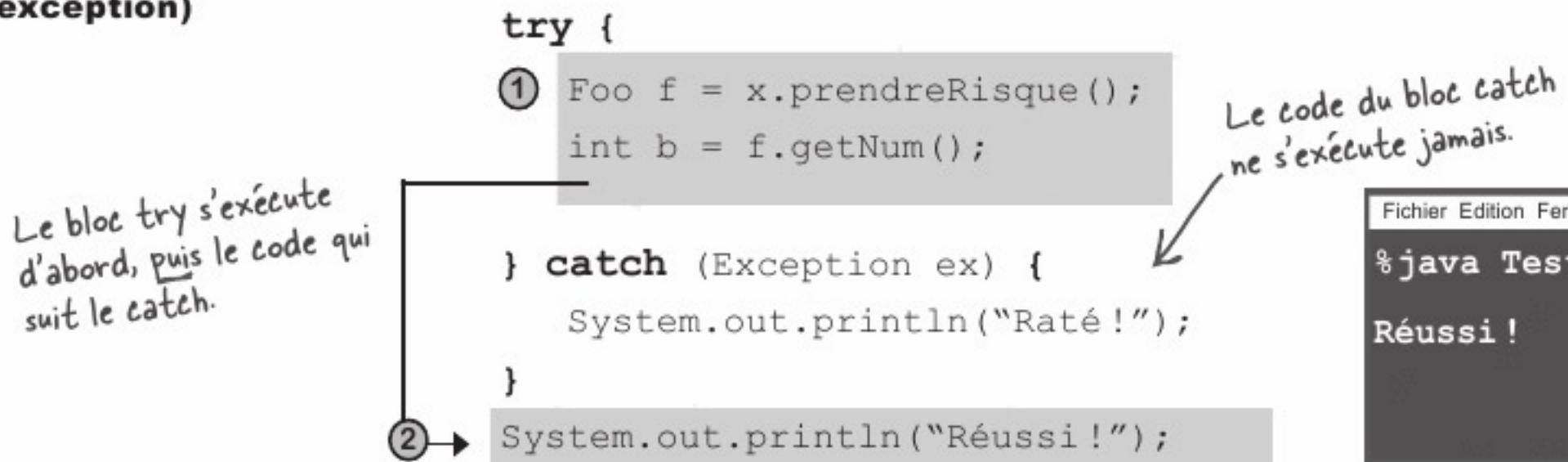


Contrôle de flot dans les blocs try/catch

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.

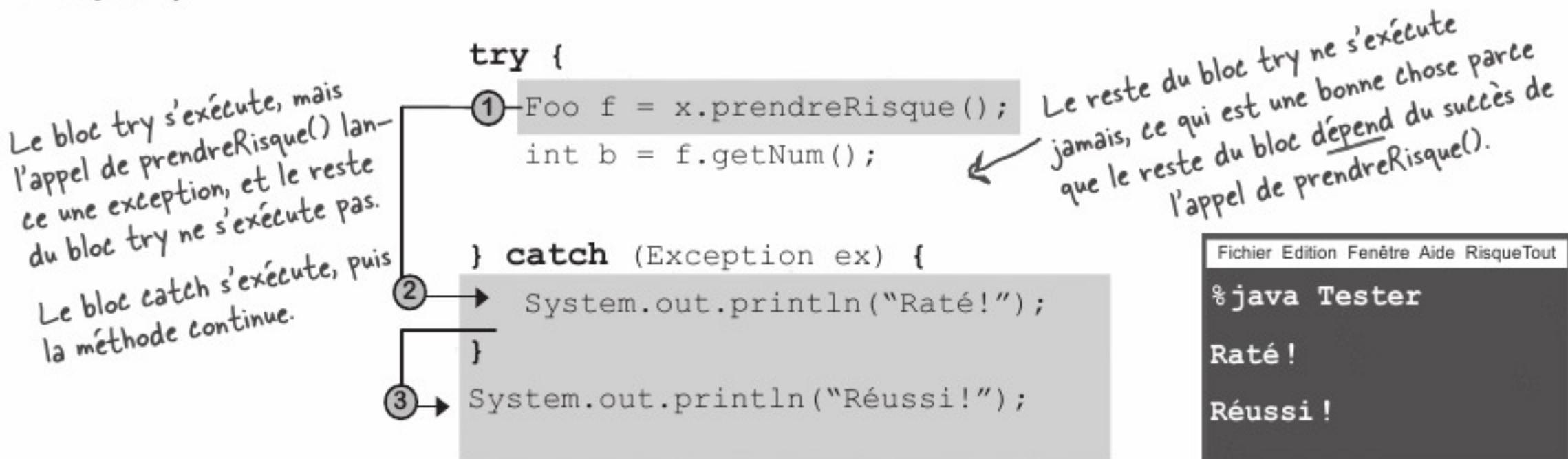
Si le bloc try réussit

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



Si le bloc try échoue

(`prendreRisque()` lance une exception)



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.



À vos crayons

Contrôle de flot

```
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;  
    }  
}
```

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 HorribleException étend Exeption.

Résultat quand test = "non"

Résultat quand test = "oui"

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

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
        }
    }
}
```



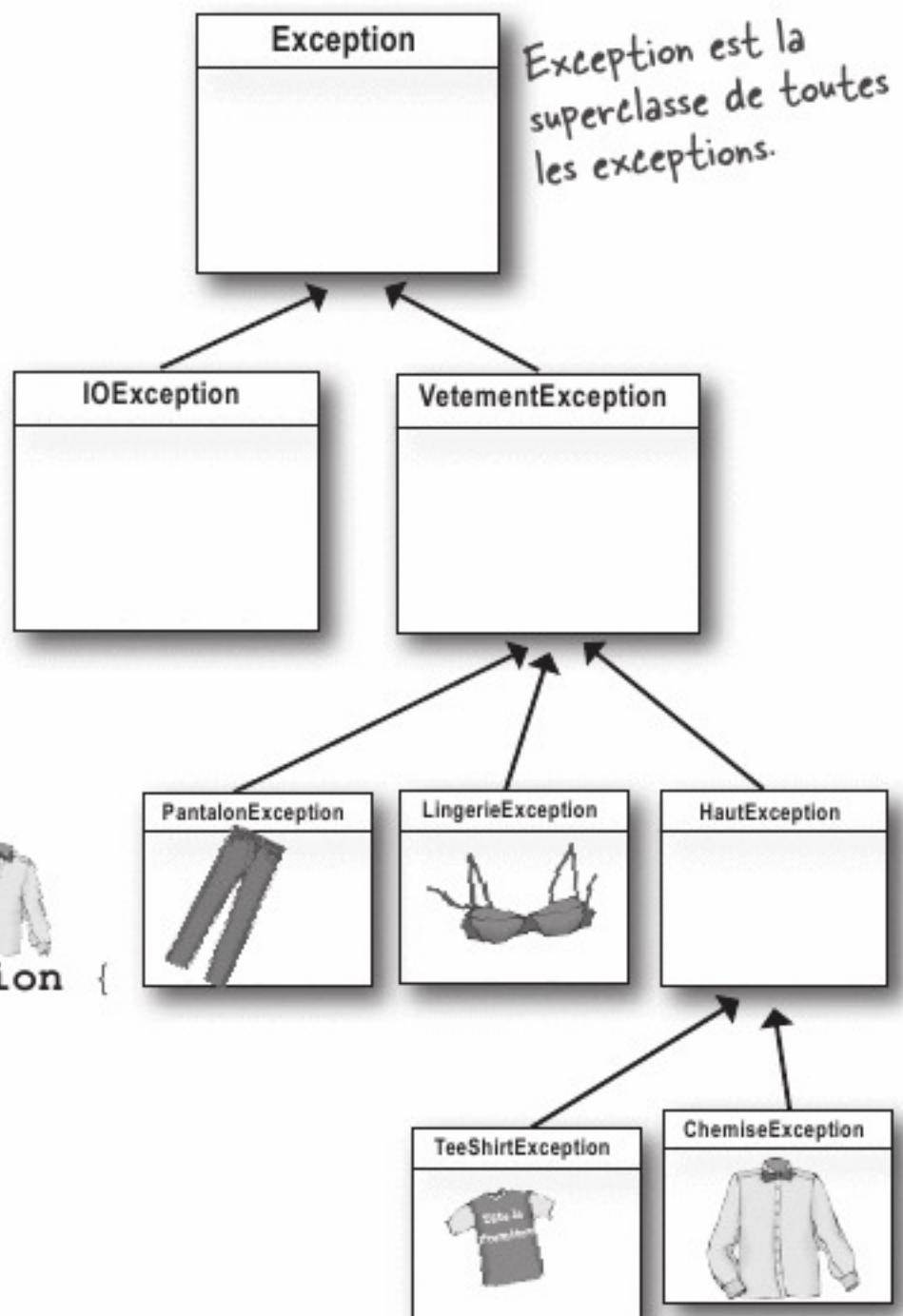
Si faireLaLessive() lance une PantalonException, elle atterrit dans le bloc catch de PantalonException.



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

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 VtementException {
```

Déclarer une VtementException vous permet de lancer n'importe quelle sous-classe de VtementException. 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(VtementException vex) {
    // code de récupération
}
```

Peut intercepter une sous-classe de VtementException quelconque.

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

Ne peut intercepter que TeeShirtException et ChemiseException

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.

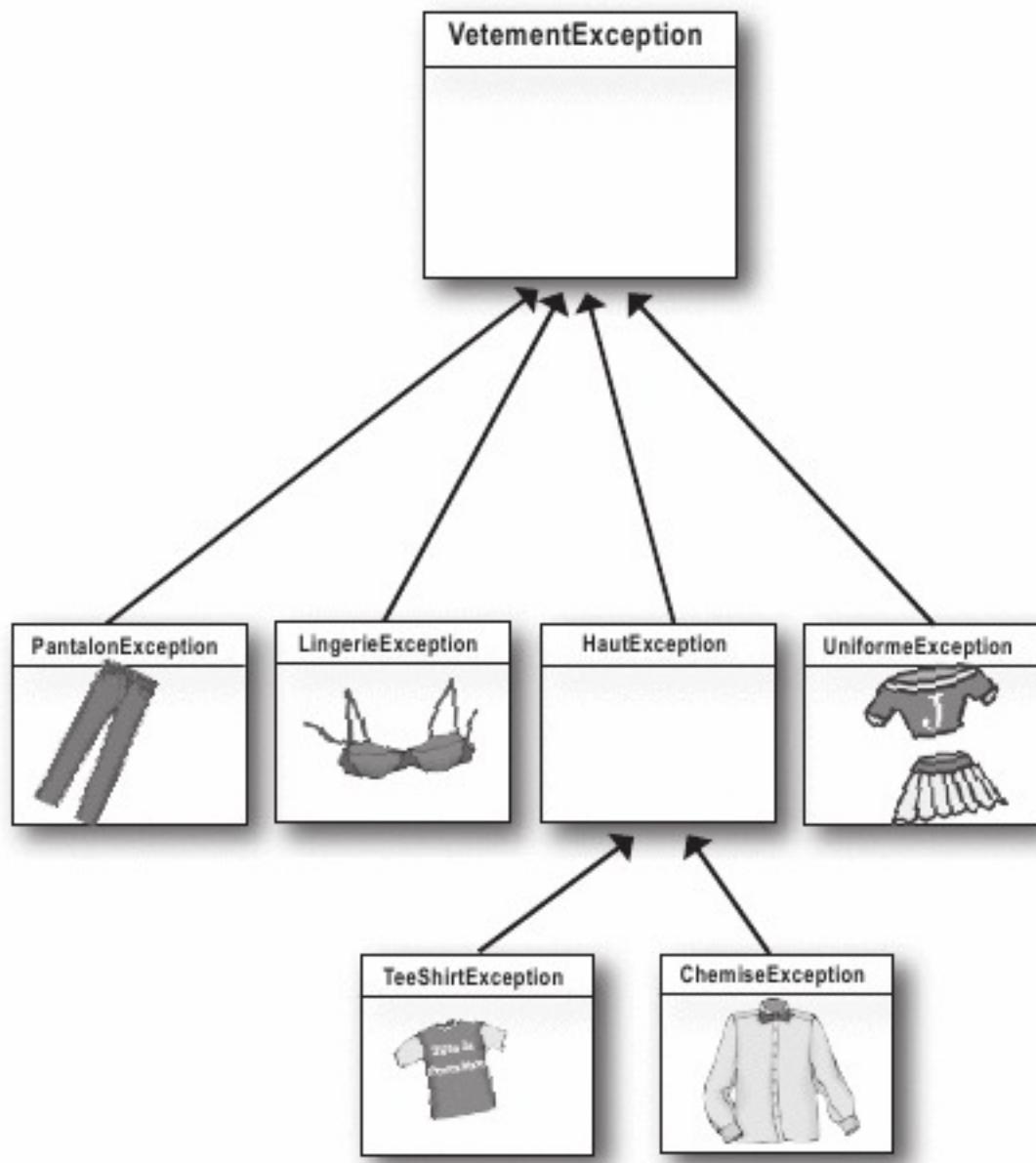
Si par exemple votre code traite différemment (ou récupère différemment) une TeeShirtException et une LingerieException, écrivez deux blocs catch distincts. Mais si vous traitez tous les autres types de VetementException de la même manière, ajoutez une clause catch avec une VetementException pour gérer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

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



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.

You 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.

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.

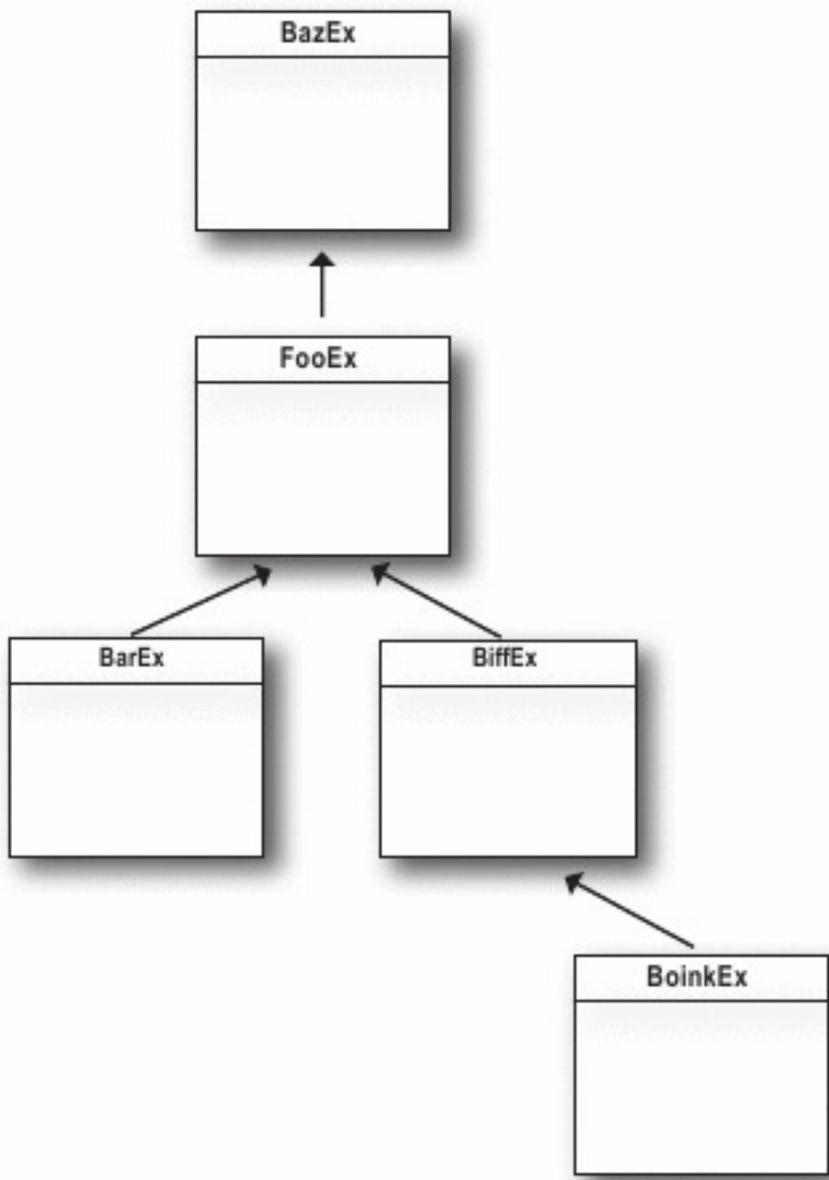
Vous pouvez placer HautException au-dessus de LingerieException, et personne n'y verra d'objection. Même si HautException est d'un type plus grand et peut intercepter d'autres types (ses propres sous-types) elle ne peut pas intercepter de LingerieException, donc il n'y a pas de problème.



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.



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.

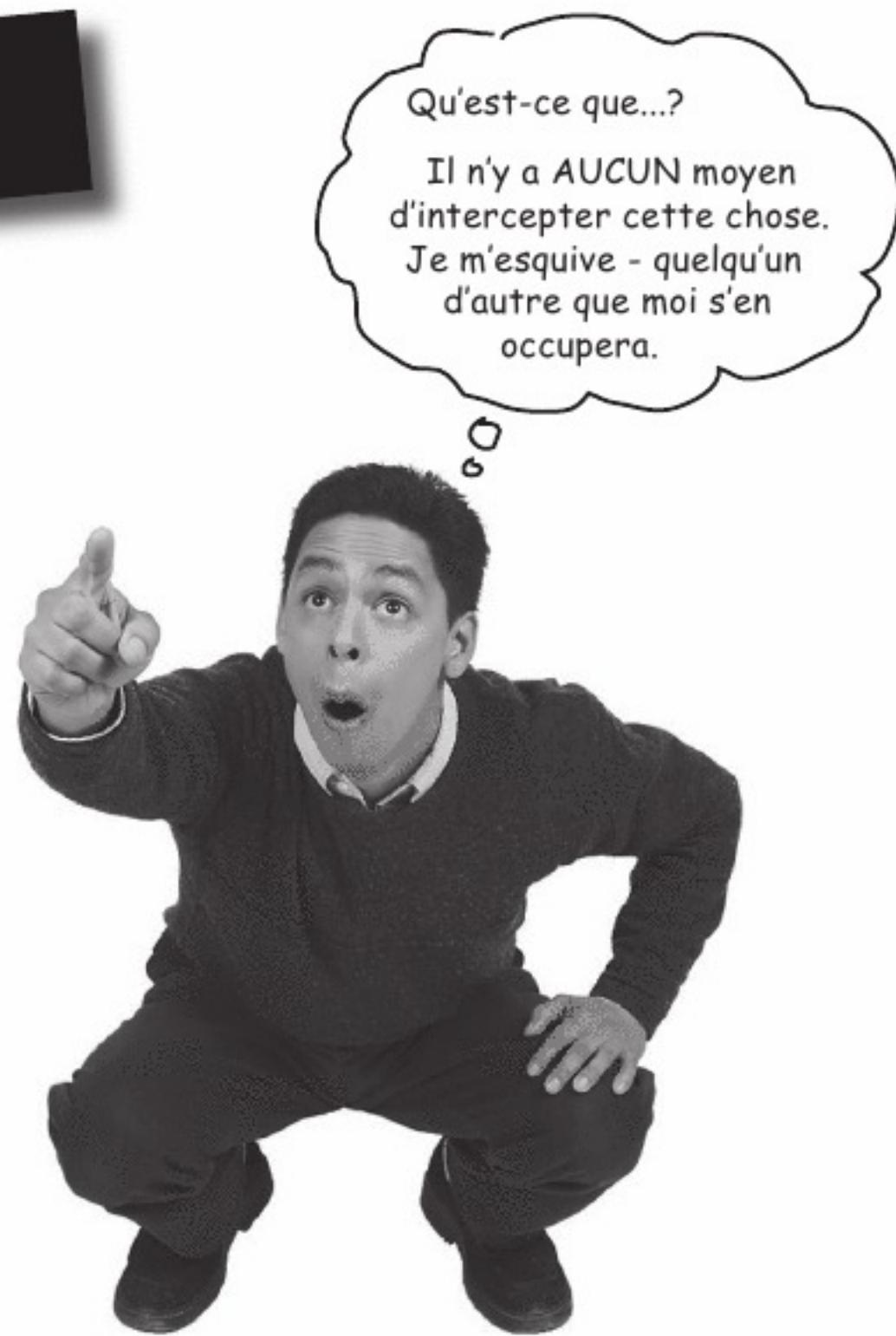
Quand vous appelez une méthode risquée, le compilateur doit savoir que vous êtes au courant. La plupart du temps, cela signifie «encapsuler» l'appel dans un bloc try/catch. Mais vous disposez d'une autre solution : l'esquiver et laisser la méthode appelée intercepter l'exception.

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 esquez 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 dépliée immédiatement, et l'exception est lancée à la méthode suivante dans la pile — la méthode *appelante*. Mais si la méthode appelante esque l'exception, elle n'a pas de clause catch pour l'intercepter. Elle est donc dépliée 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.

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 dépiler immédiatement et l'exception passe à... Qui?
Qui? 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.

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();
    }
}
```

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!

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.

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?

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.

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.

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 ».

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.

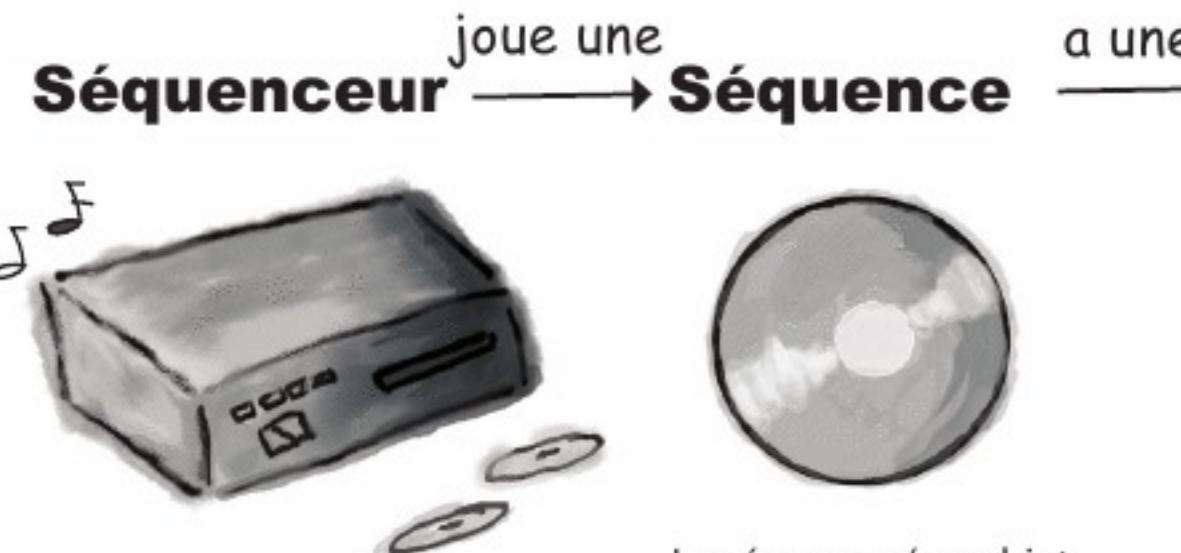
Mais si vous voulez en savoir plus sur JavaSound, tournez la page.

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.

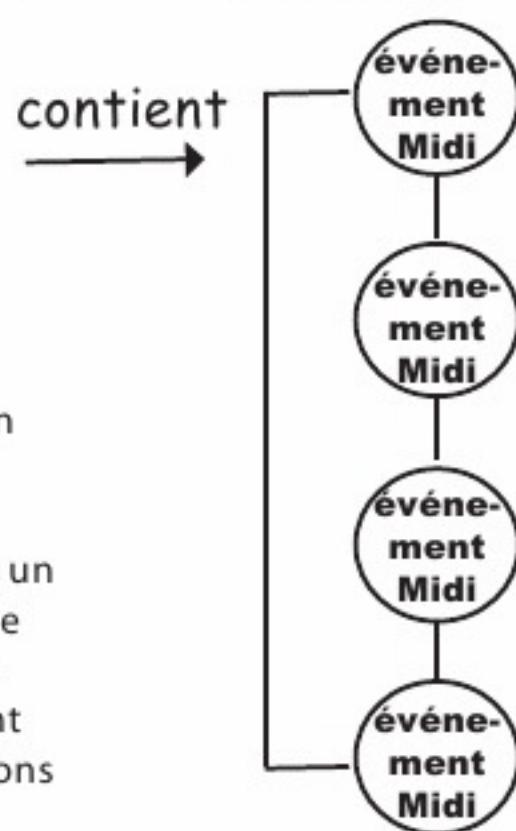


C'est le séquenceur (un objet Sequencer) qui restitue réellement le son. Vous pouvez le voir comme un **lecteur de CD**.

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**.

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.

Dans ce livre, nous n'avons besoin que d'une seule piste (un objet Track). Vous pouvez donc vous représenter 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 ».

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);
```

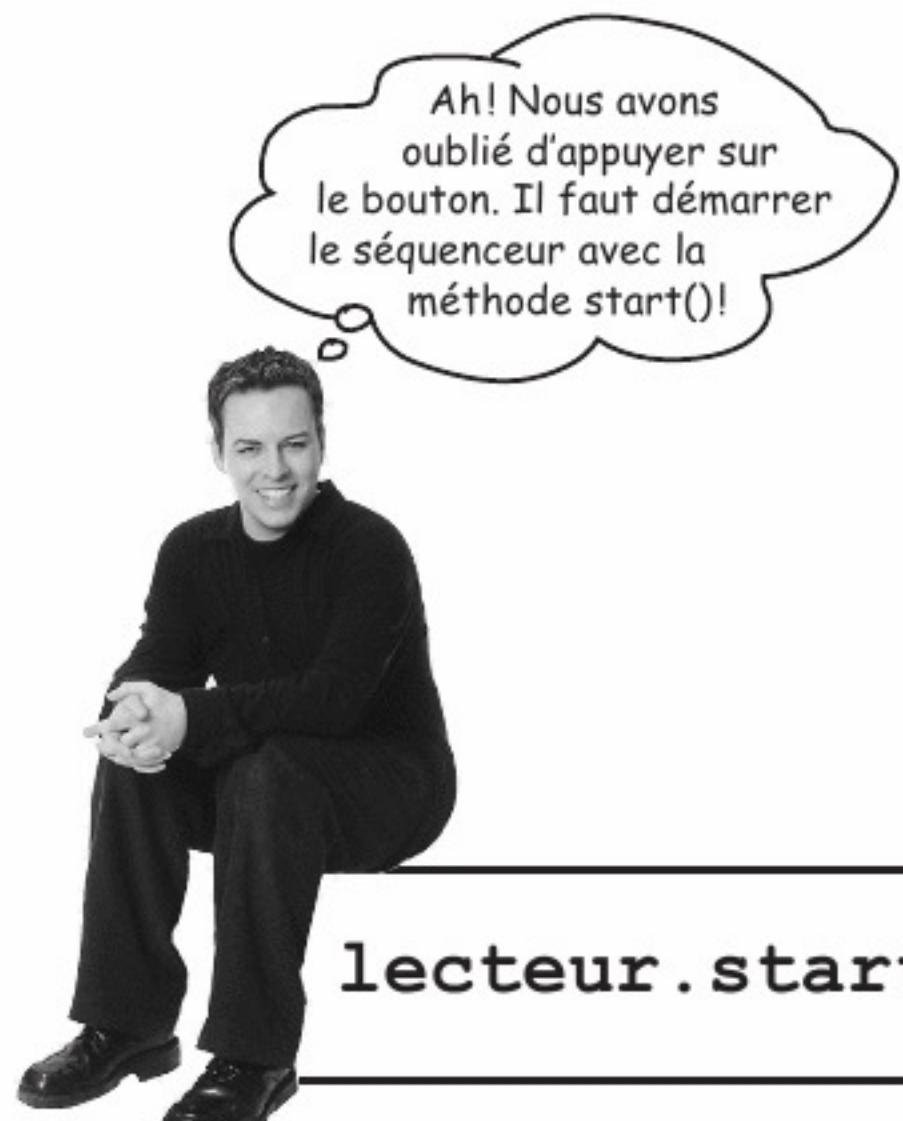
- ③ Demander à la Sequence de créer une piste piste de type **Track**

```
Track piste = seq.createTrack();
```

- ④ Remplir la piste d'événements MIDI – **MidiEvents**

– et transmettre la séquence au séquenceur

```
piste.add(monMidiEvent1);  
lecteur.setSequence(seq);
```



lecteur.start();

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.)

N'oubliez pas d'importer le package midi.

```

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.PPO, 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

```

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).

Demander de dormir pendant 20 secondes (le Thread MIDI a alors une chance de jouer)





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.

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.

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.

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.

Le Message dit quoi faire et le MidiEvent dit quand.

```
a.setMessage(144, 1, 44, 100);
```

① Type de message

144 signifie
NOTE ON



128 signifie
NOTE OFF

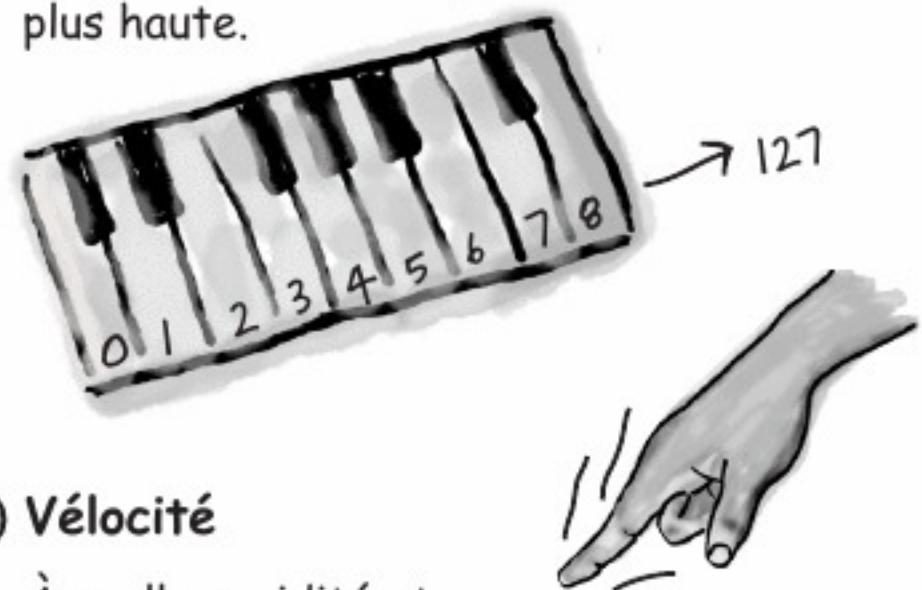


② Canal

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 la touche? 0 est si doux que vous n'entendrez probablement rien, mais 100 est une bonne valeur par défaut.

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



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) {  
        MiniMusicCmdLine 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
```

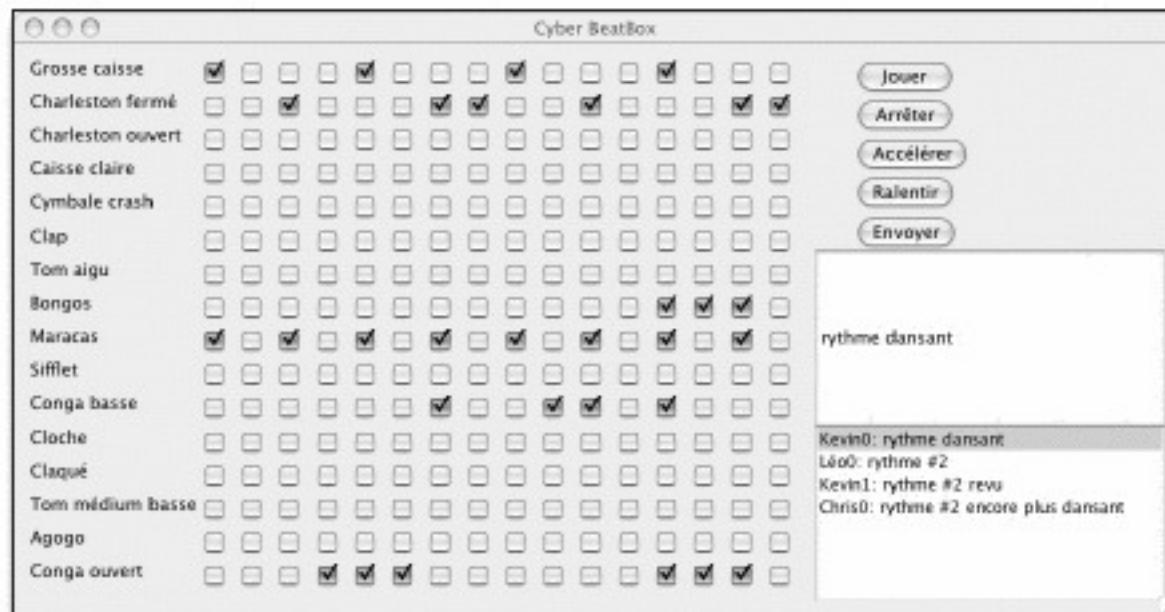
Exécutez le programme avec deux arguments de 0 à 127. Essayez ceux-ci pour commencer:

```
Fichier Edition Fenêtre Aide Jouer  
%java MiniMusicCmdLine 102 30  
%java MiniMusicCmdLine 80 20  
%java MiniMusicCmdLine 40 70
```

Suite du programme des Recettes de code

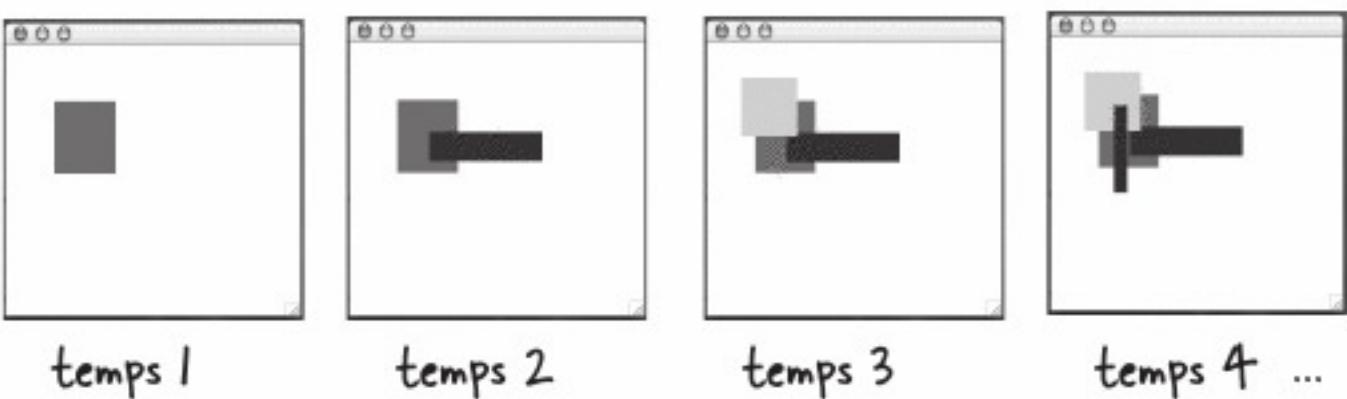
Chapitre 15: le but final

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).



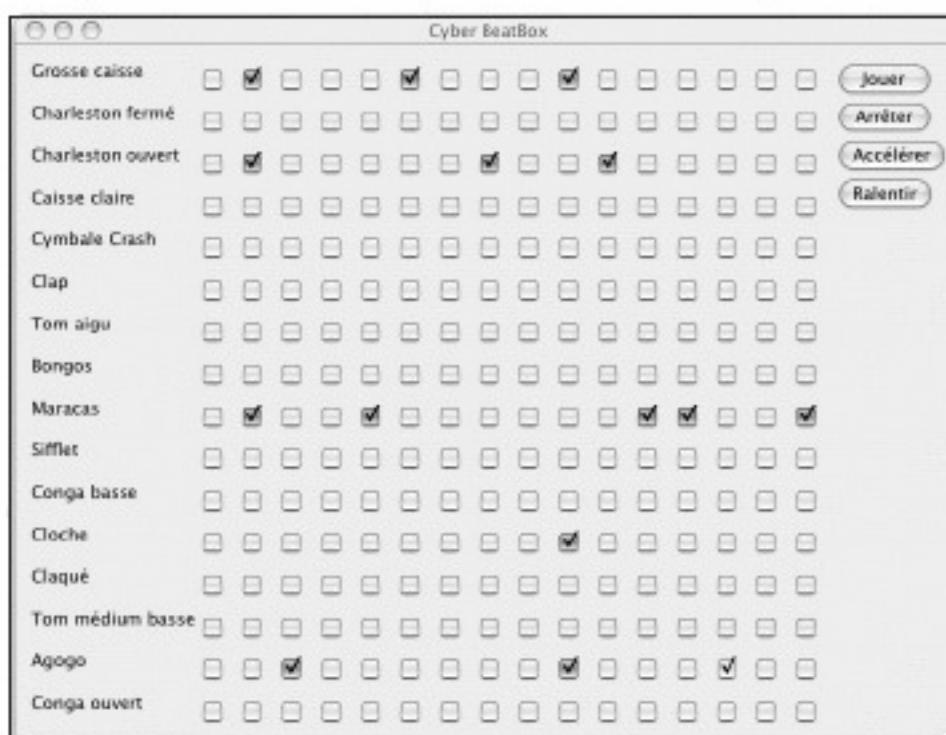
Chapitre 12: événements MIDI

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).



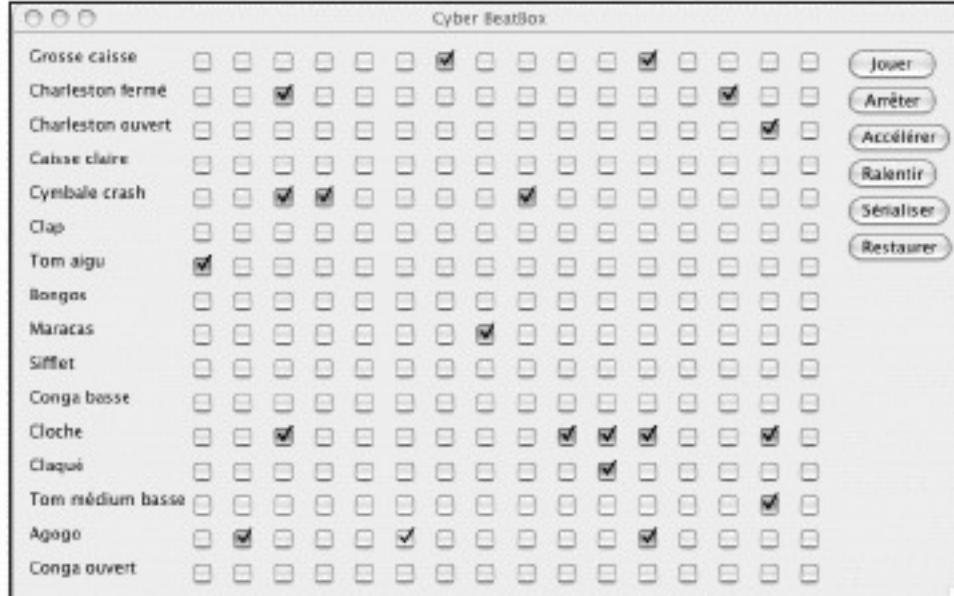
Chapitre 13: BeatBox autonome

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.





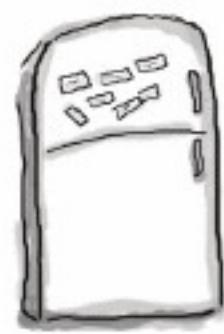
Dans ce chapitre, nous avons exploré le merveilleux monde des exceptions. Vous devez maintenant déterminer si les affirmations suivantes sont vraies ou fausses.

Thumbs up **Vrai ou Faux** Thumbs down

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.



Exercice



Le frigo

gestion des exceptions

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];

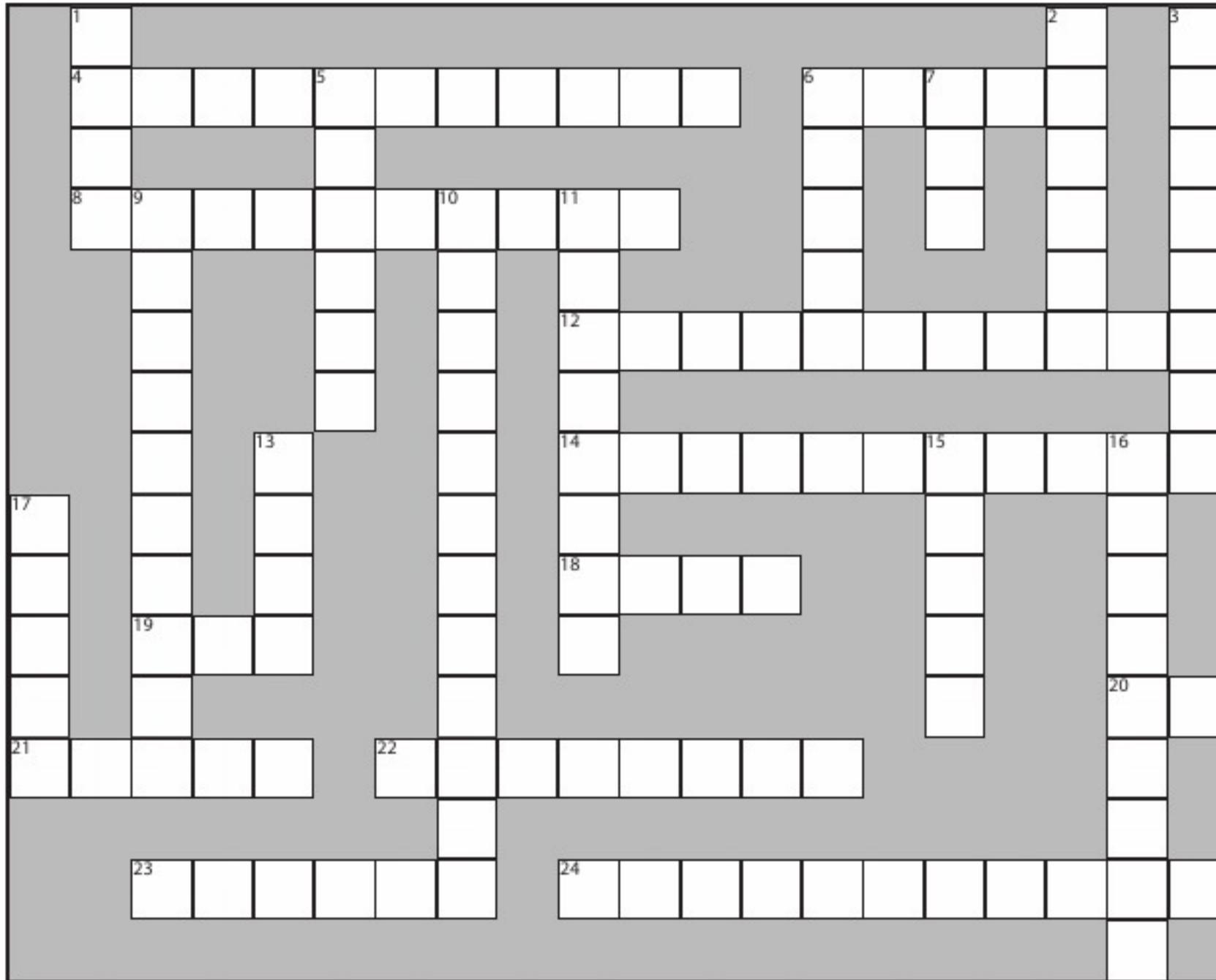
Fichier Edition Fenêtre Aide Lancer

% java ExTestDrive non
laver

% java ExTestDrive oui
lancer



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



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");
    }
}
```

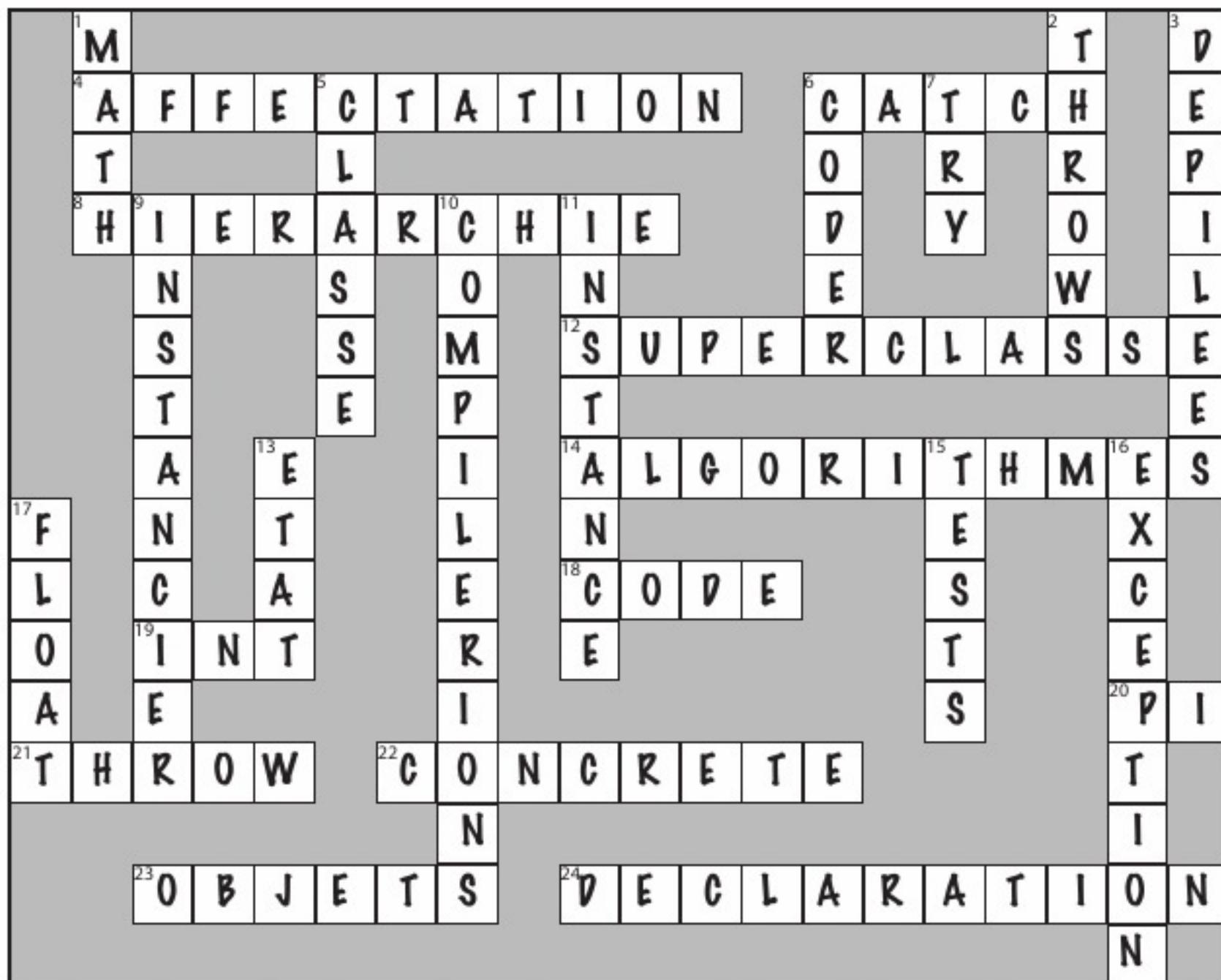
Fichier Edition Fenêtre Aide Laver

% java ExTestDrive non
laver

% java ExTestDrive oui
lancer



Solution des mots-croisés



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.

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 :



Un JFrame avec une barre de menus et deux « widgets » (un bouton et un bouton radio).

Si je vois encore une application en mode ligne de commande, vous êtes viré.



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);`

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.

Votre première interface graphique: un bouton dans un cadre

```

import javax.swing.*; ← Ne pas oublier d'importer le
                        package swing.

public class SimpleIHM1 {
    public static void main (String[] args) {
        JFrame cadre = new JFrame(); ← Créer un cadre et un bouton.
        JButton bouton = new JButton("cliquez-moi"); (Vous pouvez passer au constructeur
                                                le texte que vous voulez voir sur le
                                                bouton.)

        cadre.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE); ← Cette ligne termine le programme dès que
                                                vous fermez la fenêtre. Si vous l'oubliez, elle
                                                restera indéfiniment affichée à l'écran.

        cadre.getContentPane().add(bouton); ← Ajouter le bouton au panneau.

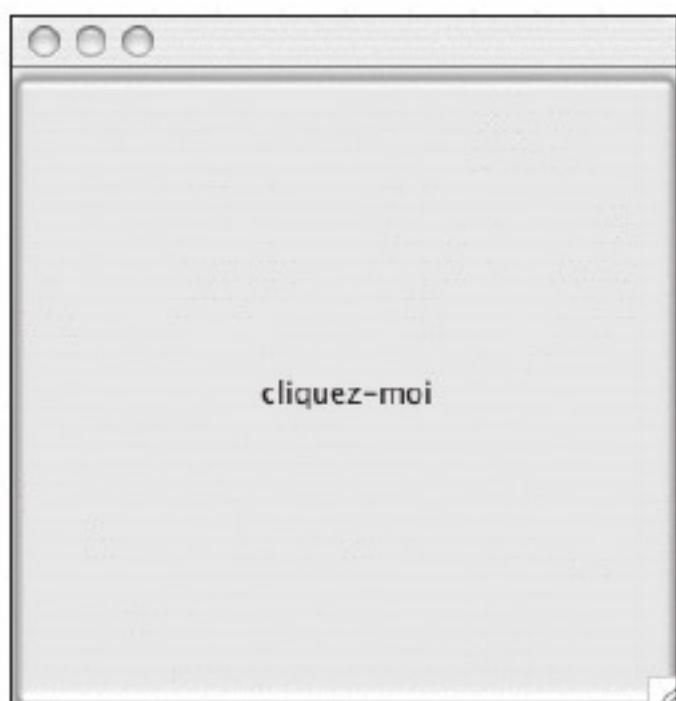
        cadre.setSize(300,300); ← Fixer la taille du cadre en pixels.

        cadre.setVisible(true); ← 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).

il n'y a pas de
Questions stupides

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!



**Quand l'utilisateur clique,
nous voulons le savoir.**

**Nous nous intéressons à un
événement: l'action de l'utili-
sateur sur le bouton.**

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.

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!");
}
```

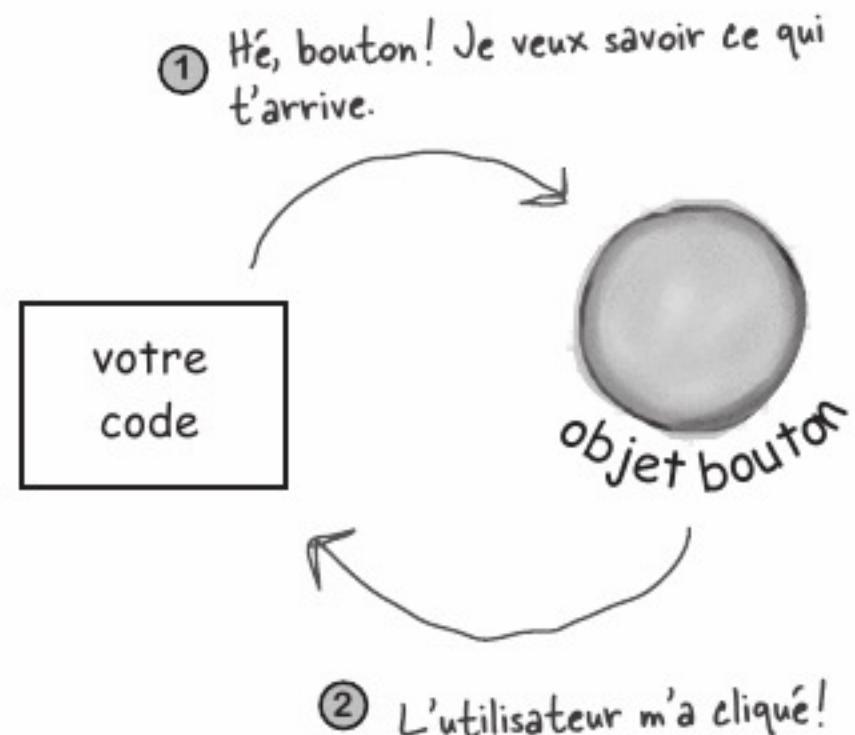
Et maintenant? Comment saurons-nous quand cette méthode doit s'exécuter? ***Et comment saurons-nous que le bouton a été cliqué?***

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?

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).

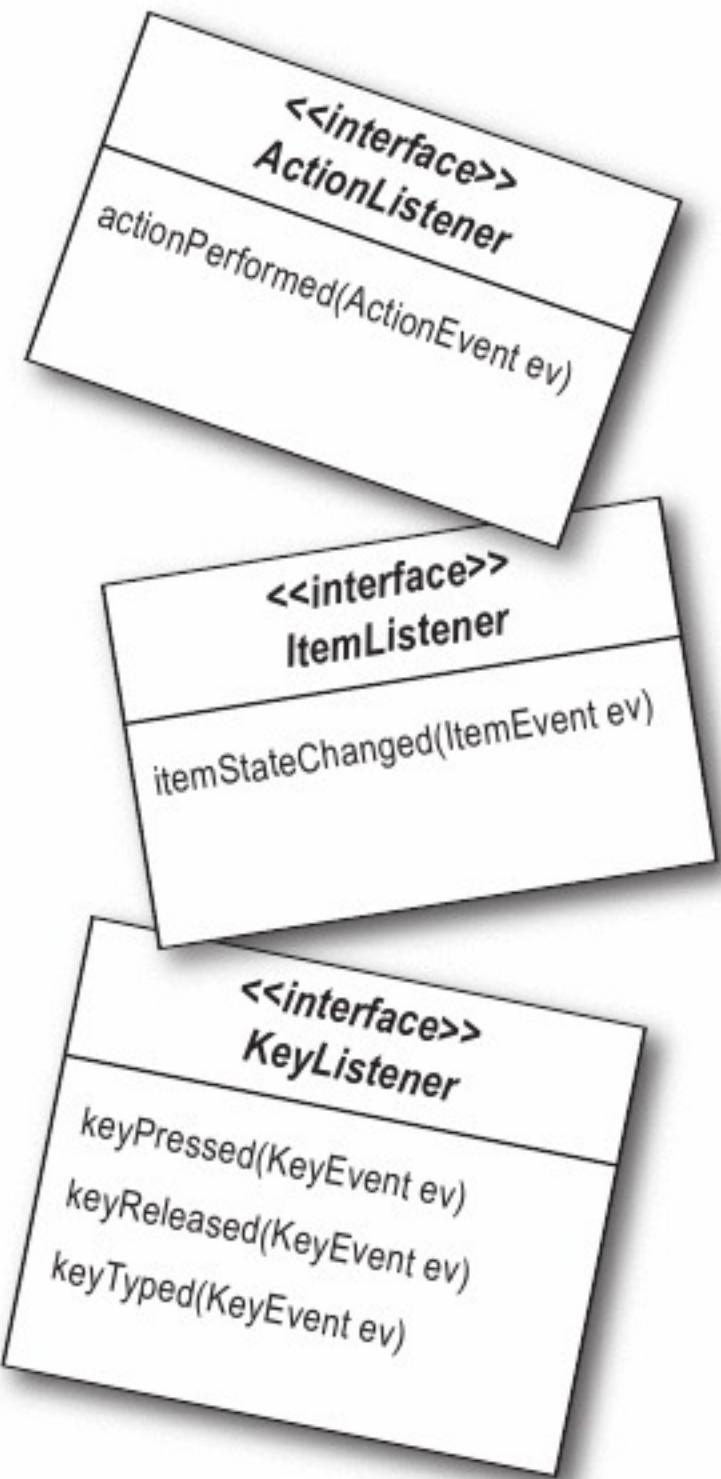
Les composants Swing sont des sources d'événements. Dans la terminologie Java, une source d'événement est un objet qui transforme les actions de l'utilisateur (cliquer un bouton, appuyer sur une touche, fermer une fenêtre) en événements. Et, comme virtuellement tout en Java, un événement est représenté 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*.

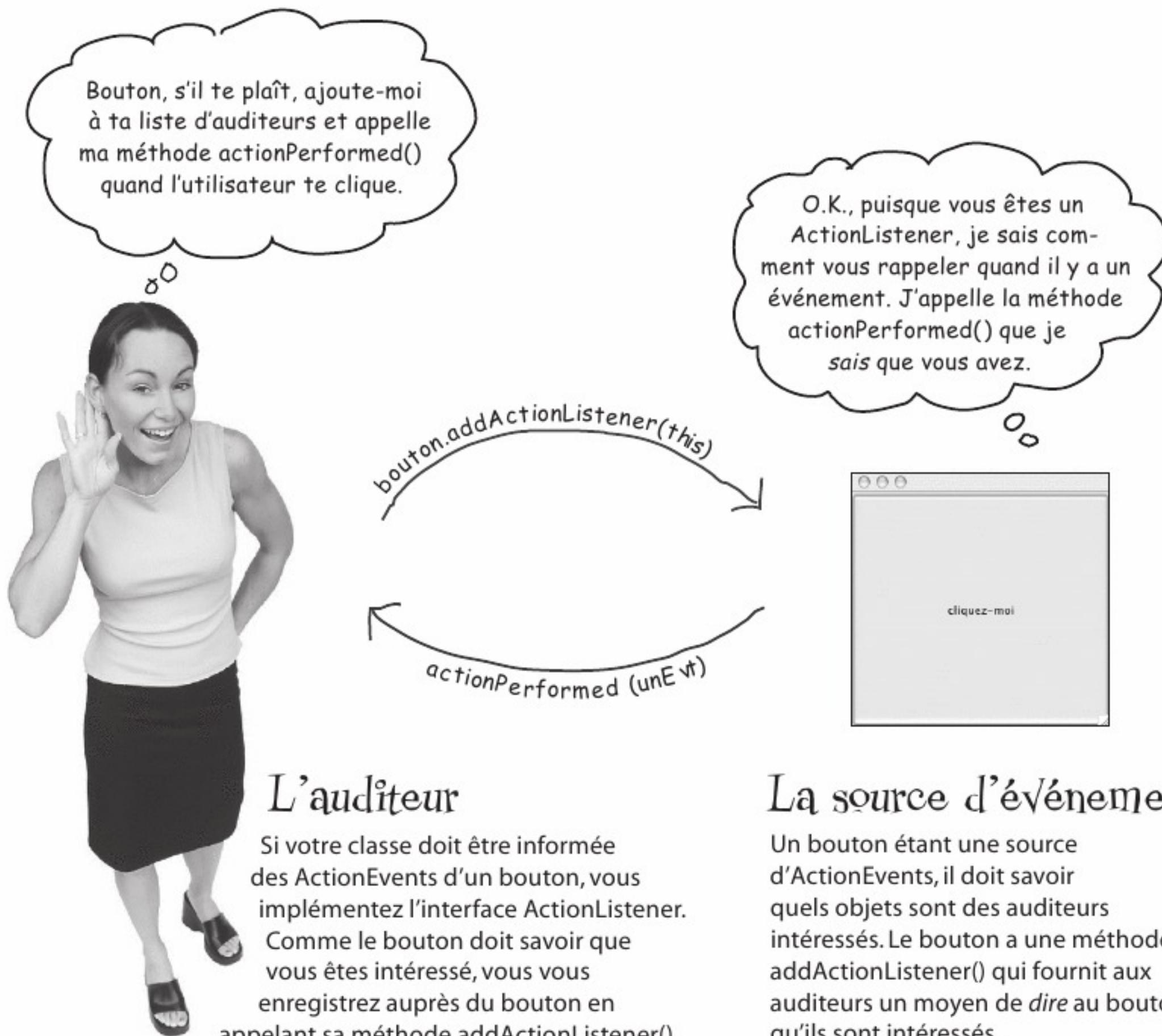
Chaque type d'événement a une interface auditeur qui lui correspond. Si vous voulez un `MouseEvent`, implémentez l'interface `MouseListener`. Un `WindowEvent`? Implémentez `WindowListener`. Vous voyez l'idée. Et rappelez-vous les règles des interfaces — pour implémenter une interface, vous *déclarez* 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.



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.

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.*; ← Importer le package qui contient ActionListener et ActionEvent.

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");
        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!");
    }
}

  ① public class SimpleIhm1B implements ActionListener { ← Implémenter l'interface. Cette instruction dit "Une instance de SimpleIhm1B EST-UN ActionListener". (Le bouton enverra des événements uniquement aux implémentateurs de l'ActionListener).
  ② bouton.addActionListener(this); ← 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!!
  ③ public void actionPerformed(ActionEvent event) { ← Implémenter la méthode actionPerformed() de l'interface ActionListener. C'est cette méthode qui gère réellement l'événement!
  bouton.setText("J'ai été cliqué!"); ← 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.
}

```

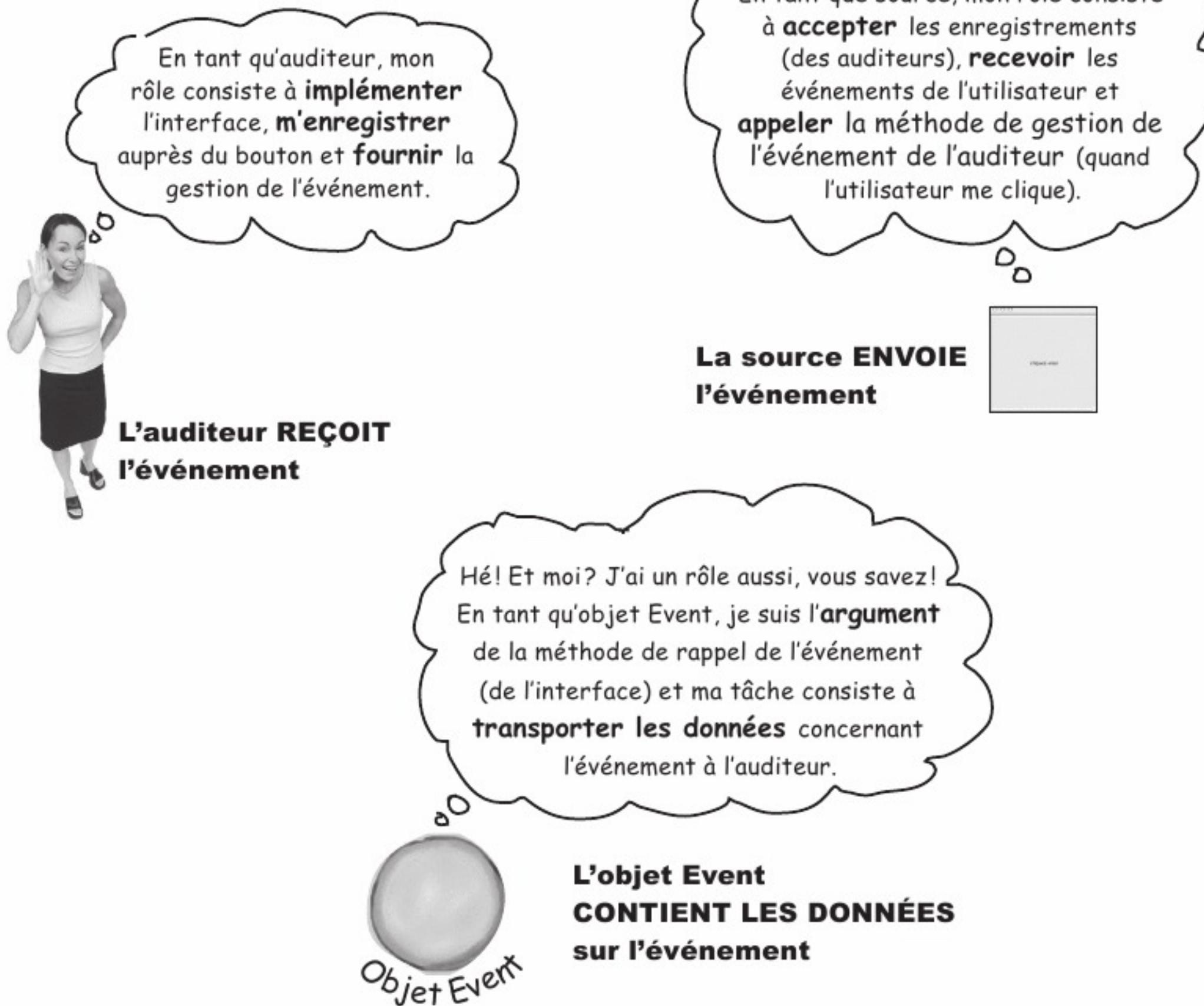
Auditeurs, sources et événements

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.)



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

case à cocher

champ de texte

liste déroulante

bouton

boîte de dialogue

bouton radio

élément de menu

Méthodes

windowClosing()

actionPerformed()

itemStateChanged()

mousePressed()

keyTyped()

mouseExited()

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.

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

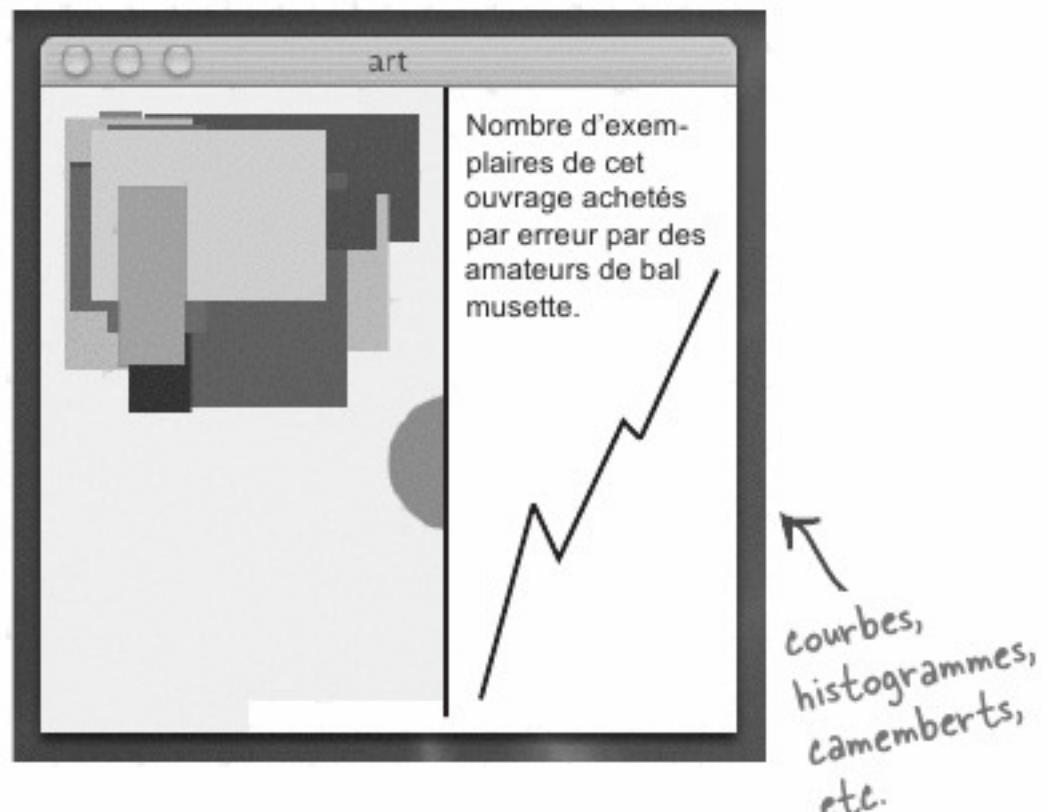
Utiliser un objet graphics pour «peindre» des formes.

```
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.

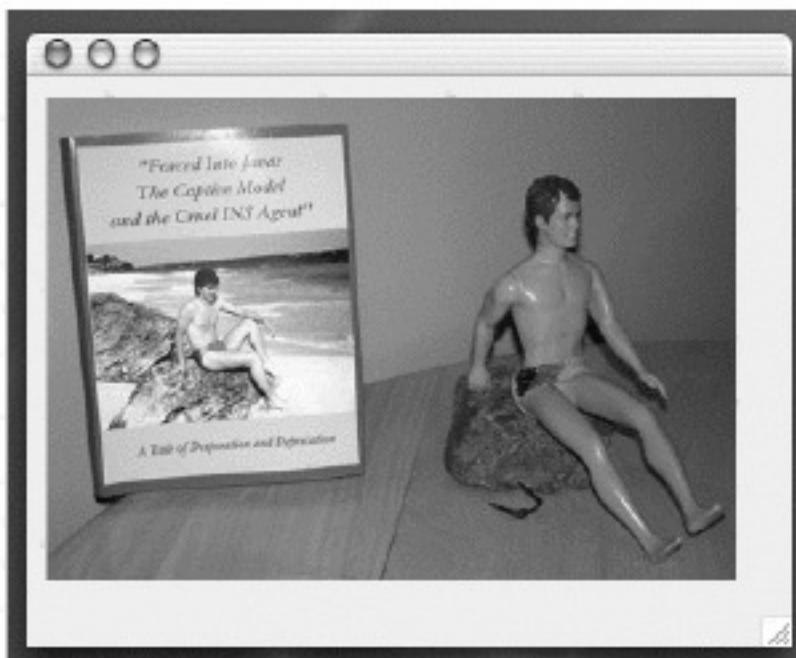
*art, jeux,
simulations,
etc.*



③ Placer un JPEG sur un widget

Vous pouvez placer vos propres images sur un widget.

```
graphics.drawImage(monImage,10,10,this);
```



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.

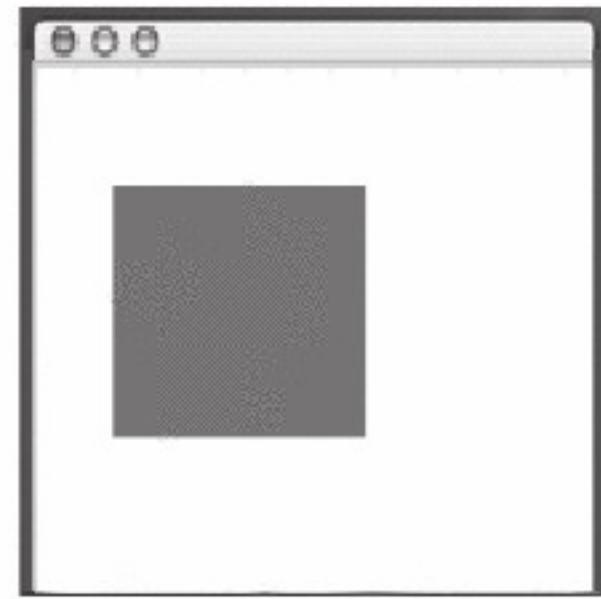
C'est vraiment du gâteau.

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'appeliez 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 *demandez* 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).

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);

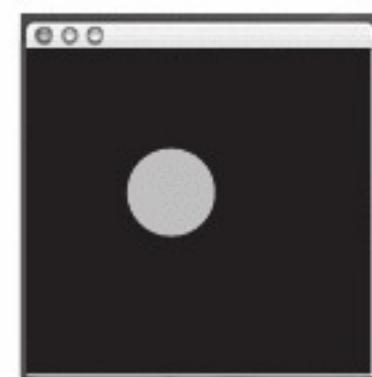
    Color randomColor = new Color(rouge, vert, bleu);
    g.setColor(randomColor);
    g.fillOval(70, 70, 100, 100);
}
```

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.



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 :

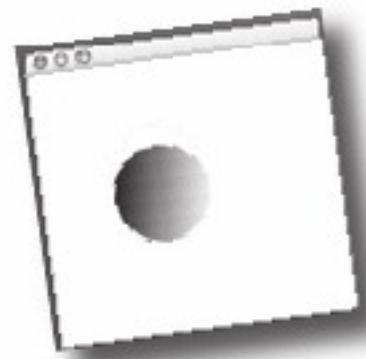
```
Graphics2D g2d = (Graphics2D) g;
```

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.)

Parce que la vie est trop courte pour peindre un cercle d'une seule couleur quand un mélange de nuances vous attend.



```
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);
}
```

Objet Graphics2D déguisé en simple objet Graphics.

Convertissez-le pour pouvoir appeler une méthode de Graphics2D que Graphics ne possède pas.

↑ ↑ ↑ ↑
couleur de départ coordonnées de départ couordonnées 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 !



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 partie 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;
```

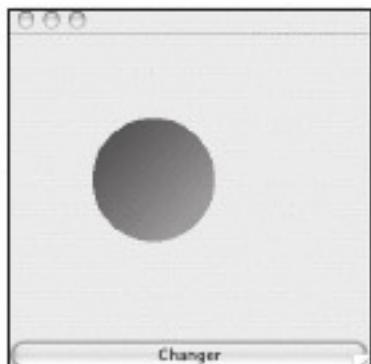
Nous savons gérer un événement.

Nous savons dessiner.

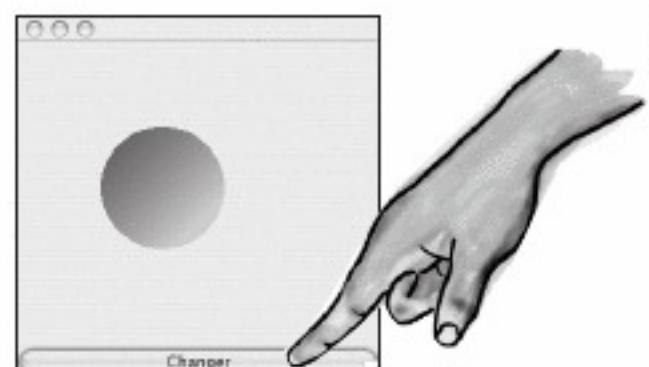
Mais pouvons-nous dessiner quand nous recevons un événement?

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 :

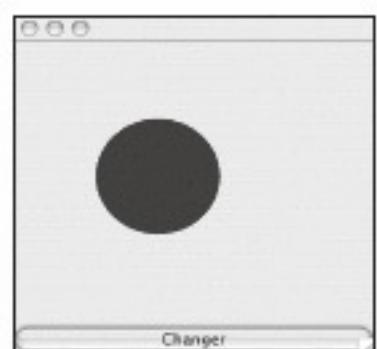
Lancer l'application



- 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 Voilà! Une nouvelle couleur est affichée, parce que paintComponent() s'exécute de nouveau et remplit le cercle avec une couleur aléatoire.



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 :

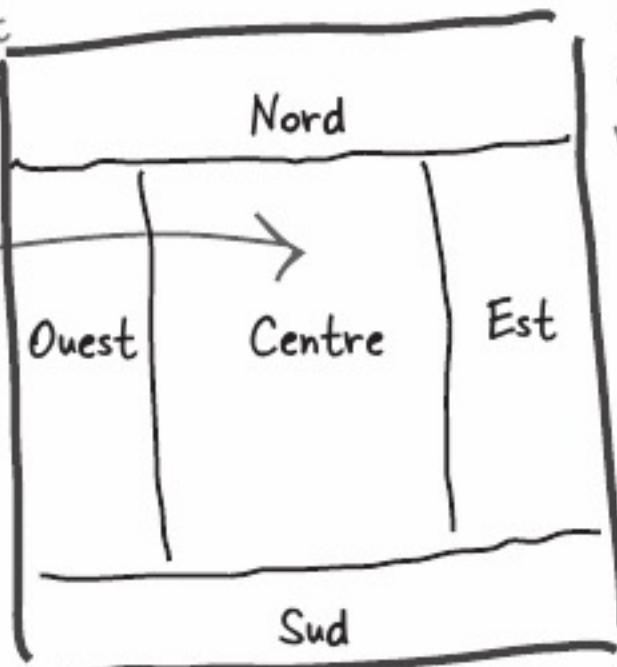
```
cadre.getContentPane().add(bouton);
```

Ce n'est pas vraiment ainsi que vous êtes censé procéder (une méthode à un seul argument).

```
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.

région par défaut



Méthode préférable (et généralement obligatoire) pour ajouter un panneau au cadre. Toujours spécifier Où (dans quelle région) vous voulez placer le widget.

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.

Le cercle change de couleur chaque fois que vous cliquez sur le bouton.

```

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();
    }
}

```

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!

```

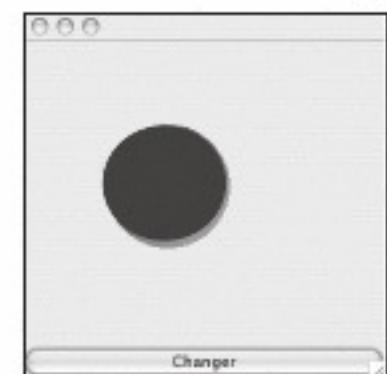
class MonPanneau extends JPanel {

    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.

Le Panneau personnalisé (instance de MonPanneau) est dans la région "CENTER" du cadre.



Le bouton est dans la région "SOUTH" du cadre.

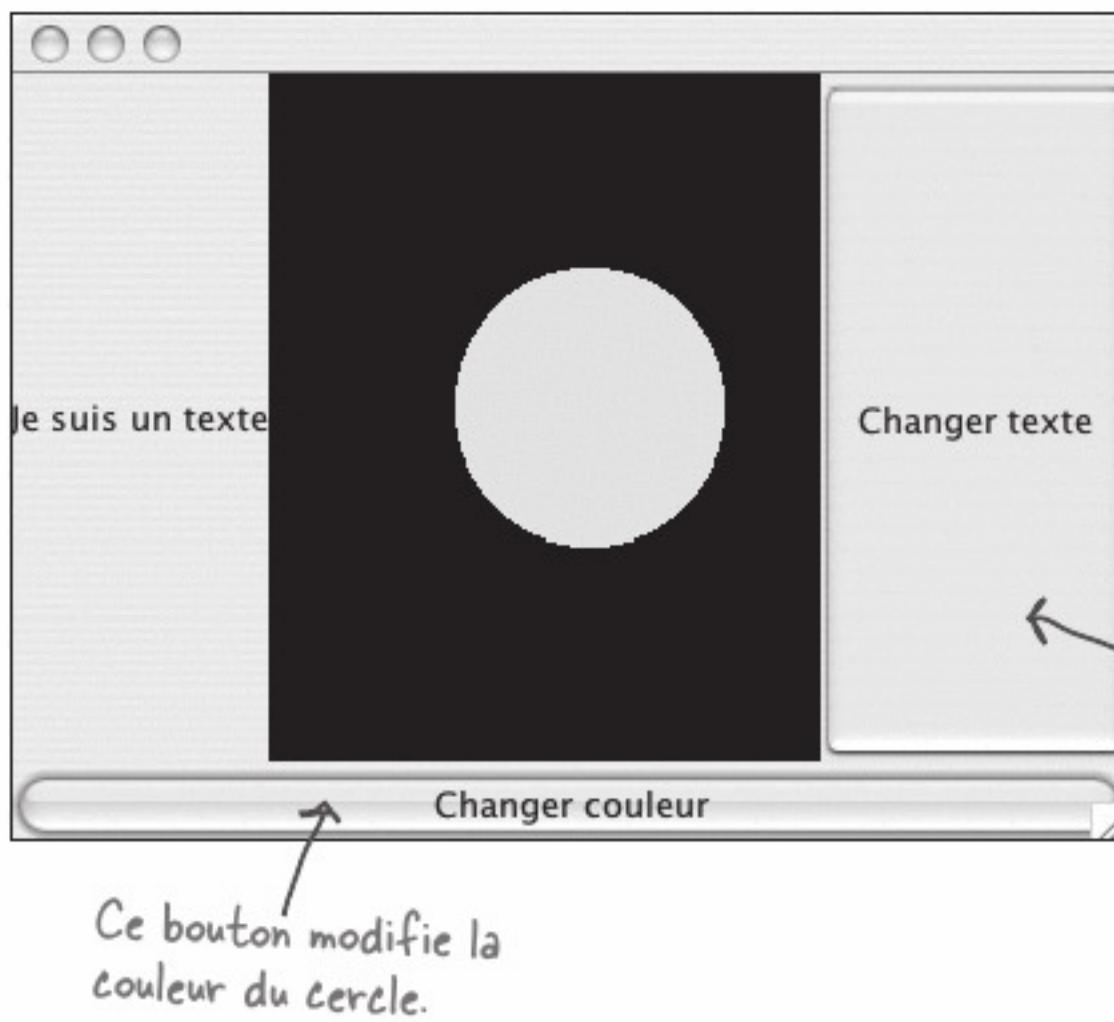
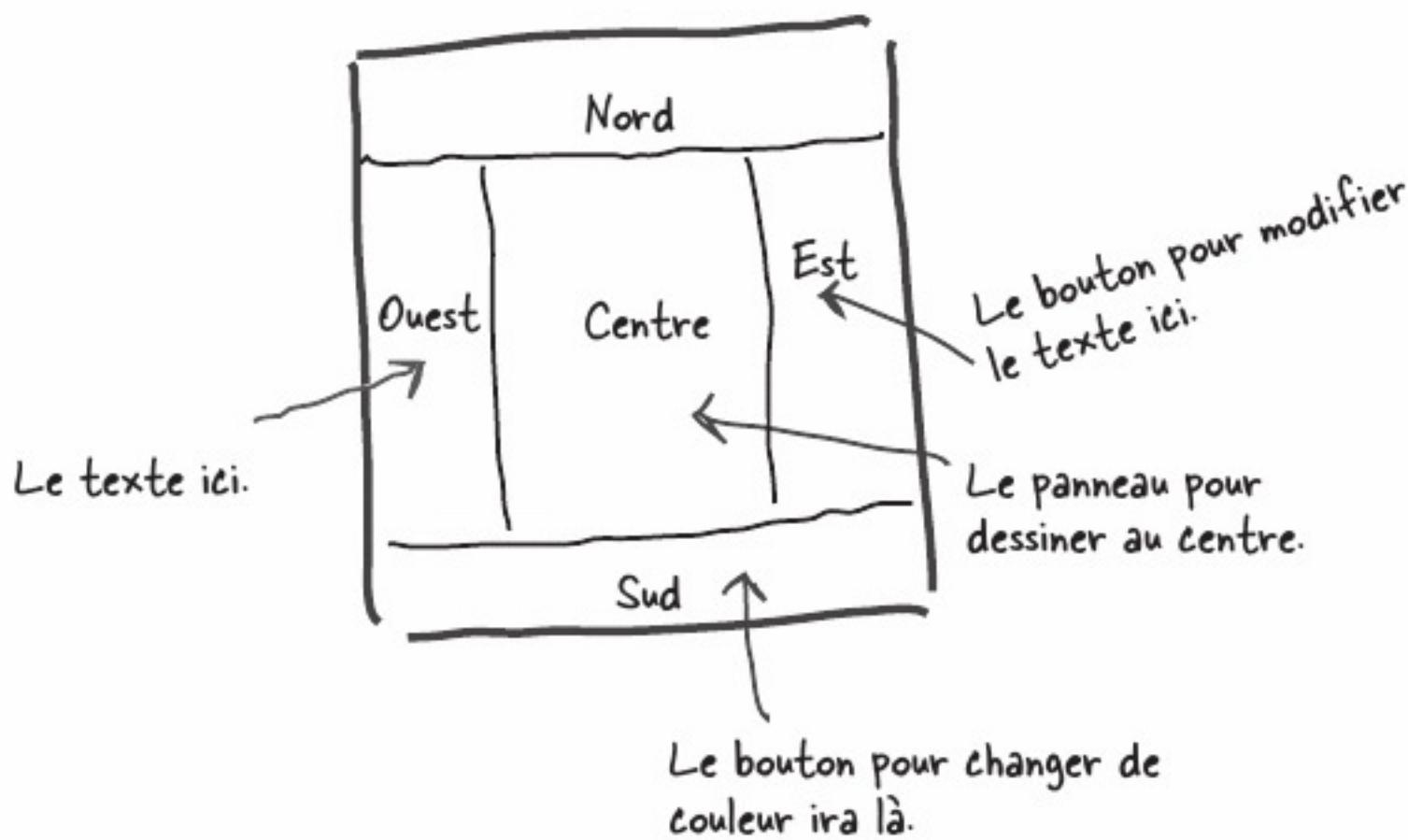
Ajouter l'auditeur (this) au bouton.

Ajouter les deux widgets (bouton et panneau) aux deux régions du cadre.

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.

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); ← Enregistrer le même auditeur
        boutonTexte.addActionListener(this); ← auprès des deux boutons.
        // suite du code ...
    }

    public void actionPerformed(ActionEvent event) {
        if (event.getSource() == boutonCouleur) {
            cadre.repaint(); ← Tester l'objet event pour
        } else {                                déterminer quel bouton l'a
            etiquette.setText("Ouille!");         déclenché, puis décider quoi faire
        }
    }
}
```

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.

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!



Ne serait-ce pas merveilleux si on pouvait avoir deux classes auditeurs différentes, mais qu'elles puissent accéder aux variables d'instance de la classe principale, presque comme si elles appartenaient à l'autre classe. Comme ça, on aurait le meilleur des deux mondes. Oui, ce serait la solution rêvée. Mais ce n'est qu'un fantasme...

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

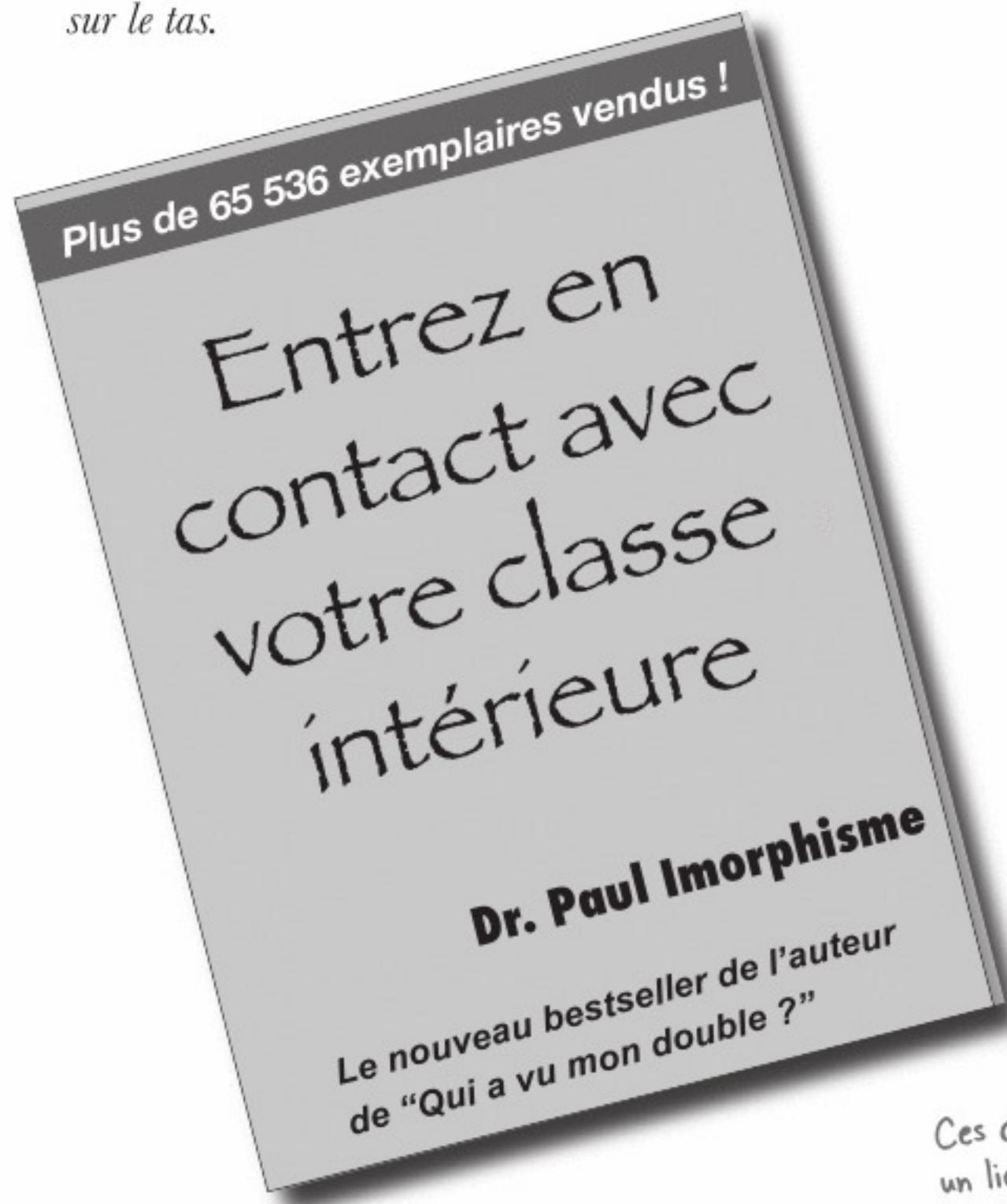
```
class MaClasseExterne {  
  
    private int x;  
  
    class MaClasseInterne {  
        void go() {  
            x = 42; ← "x" est utilisé comme si c'était une variable  
            }           de la classe interne!  
        } // fin de la classe interne  
    } // fin de la classe externe
```

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 ?

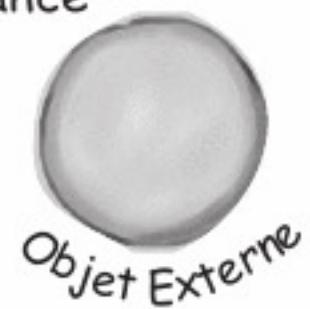
Une instance arbitraire quelconque de la classe interne peut-elle accéder aux variables et aux méthodes de n'importe quelle instance de la classe externe? **Non!**

*Un objet **interne** doit être lié à un objet spécifique **externe** sur le tas.*

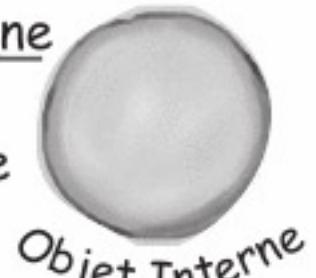


Un objet interne partage un lien particulier avec un objet externe. 

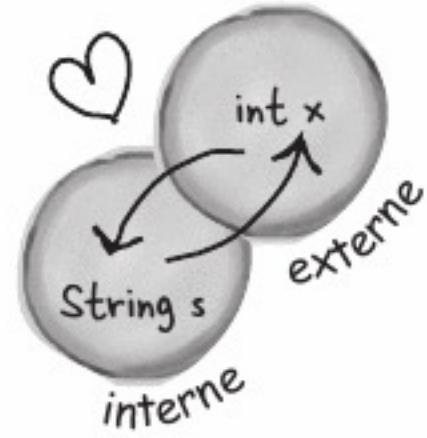
- ① Créer une instance de la classe externe.



- 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.

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;
```

La classe externe a une variable d'instance "x" privée.

```
    Mon Interne interne = new MonInterne();
```

Créer une instance de la classe interne.

```
    public void faireQqch() {
```

```
        interne.go();
```

Appeler une méthode de la classe interne.

```
    class MonInterne {
```

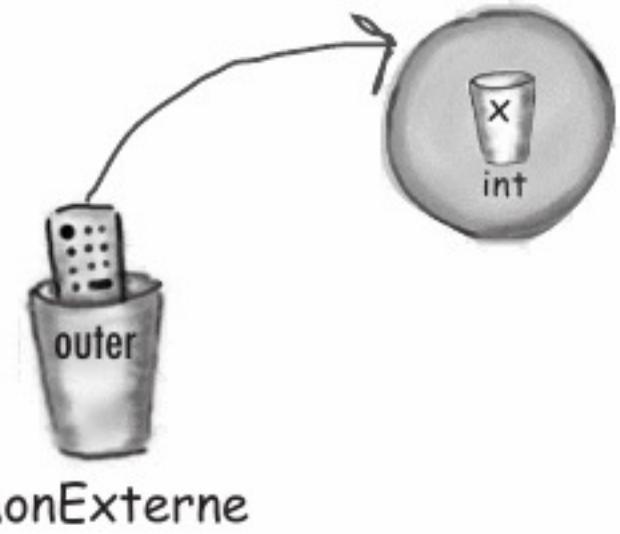
```
        void go() {
```

```
            x = 42;
```

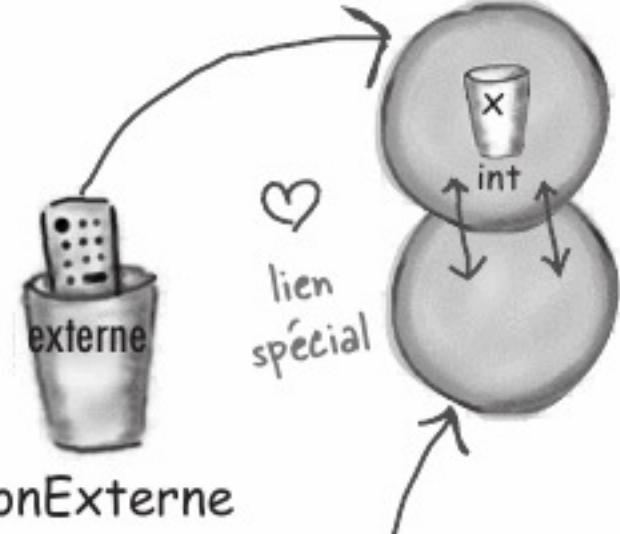
La méthode de la classe interne accède à la variable d'instance "x", comme si "x" appartenait à la classe interne.

```
    } // fin de la classe interne
```

```
} // fin de la classe externe
```



MonExterne



MonExterne



MonInterne

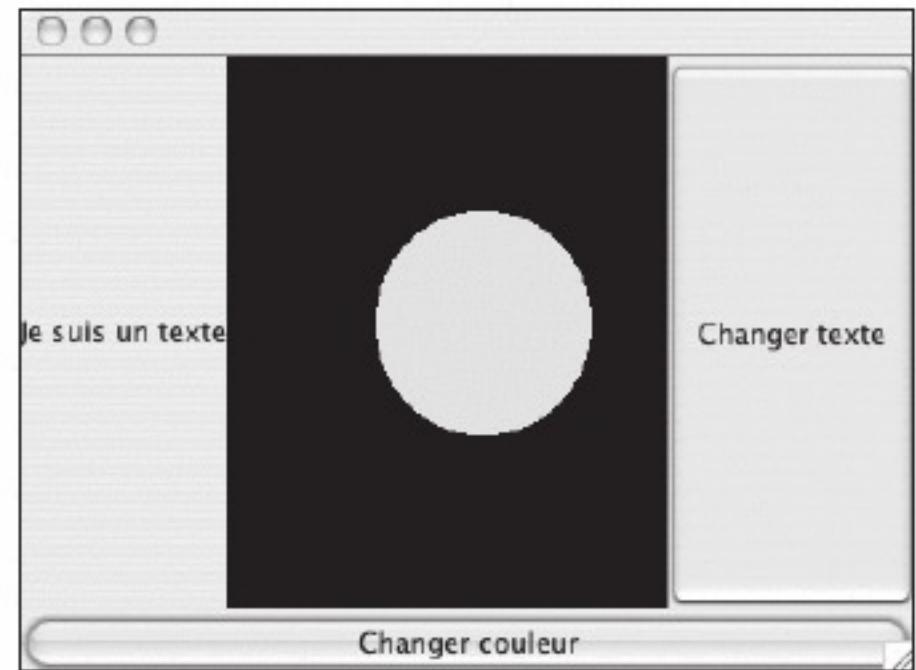
Aparté

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();
    }
}
```

Maintenant, nous pouvons créer deux boutons

```
public class DeuxBoutons { ← La classe principale  
n'implémente plus  
ActionListener.  
  
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());
```

Au lieu de passer (this) à la méthode d'enregistrement de l'auditeur du bouton, passer une nouvelle instance de la classe appropriée.

```
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);
```

```
}
```

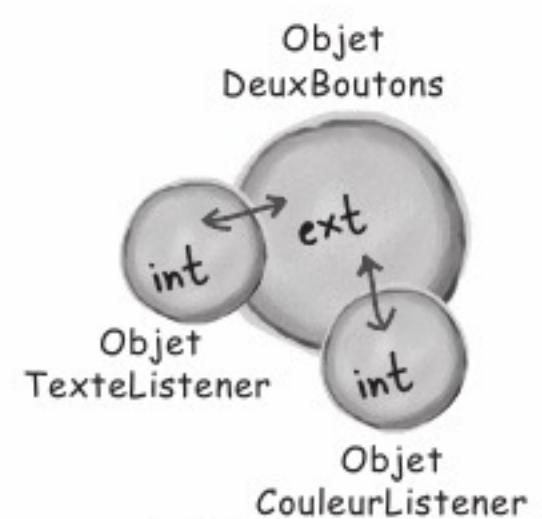
```
class TexteListener implements ActionListener {  
    public void actionPerformed(ActionEvent event) {  
        etiquette.setText("Aie!");  
    } // fin de classe interne
```

La classe interne connaît "etiquette"

Maintenant, nous avons DEUX ActionListeners dans une seule classe.

```
class CouleurListener implements ActionListener {  
    public void actionPerformed(ActionEvent event) {  
        cadre.repaint(); ←  
    } // fin de classe interne
```

La classe interne utilise la variable d'instance "cadre" de référence explicite à l'objet de la classe externe sans avoir



```
}
```



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'implementations *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 implementations 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 ?

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... hmmmm. 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 entretez 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.



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

- Dessiner un objet à une position (x,y) donnée

```
g.fillOval(20, 50, 100, 100);
```

 20 pixels de la gauche,
50 pixels du haut.

- 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)

- 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.)

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 :

Le code complet de l'animation

```

import javax.swing.*;
import java.awt.*;

public class SimpleAnimation {
    int x = 70;
    int y = 70; } ← Créer deux variables d'instance dans la classe principale de l'IHM
    pour représenter les coordonnées x et y du cercle.

    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); } ← Rien de nouveau ici. Créer les
        widgets et les placer dans le cadre.

        L'action est ici! for (int i = 0; i < 130; i++) { Répéter 130 fois.
            x++;
            y++; ← Incrémenter les coordonnées
            x et y.
            panneau.repaint (); ← Dire au panneau de se repeindre lui-même (pour afficher
            le cercle au nouvel emplacement).
            try {
                Thread.sleep (50); ← Ralentir un peu, sinon il bougera si vite
            } catch (Exception ex) { }
        } // fin de la méthode go()

        Maintenant, class MonPanneau extends JPanel {
        c'est une
        classe interne. public void paintComponent (Graphics g) {
            g.setColor (Color.green);
            g.fillOval (x,y,40,40); } ← Utiliser les coordonnées x et y continuellement
            mises à jour de la classe externe.
        }
    } // fin de la classe interne
} // fin de la classe externe

```

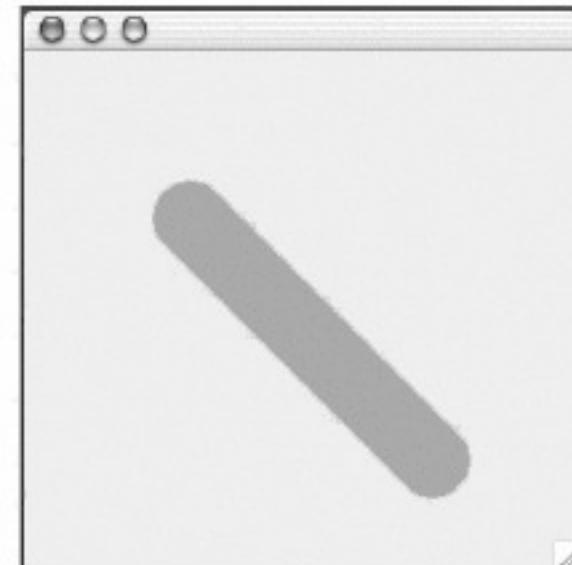
Hum. Il n'a pas bougé... il a bavé.

Où est l'erreur?

Il y a juste un petit défaut dans la méthode `paintComponent()`.

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.



Ce n'est pas vraiment le résultat recherché!

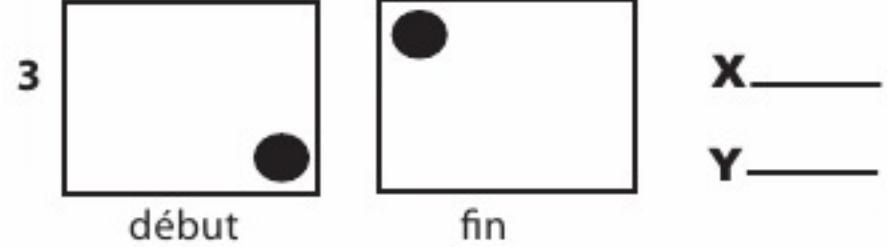
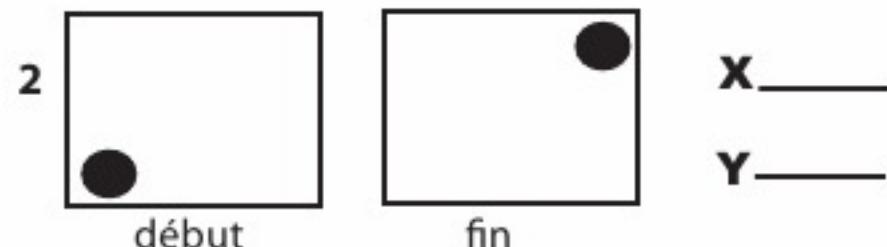
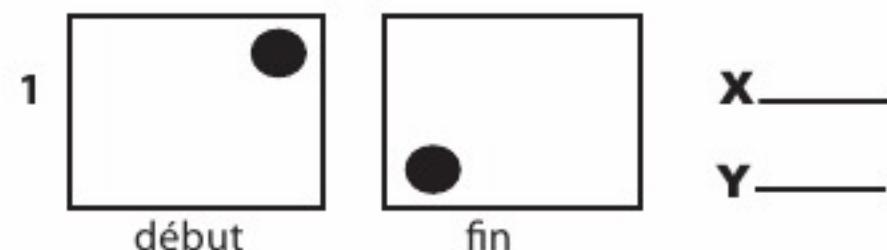
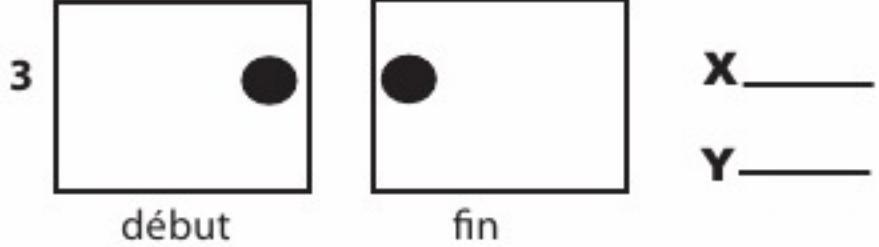
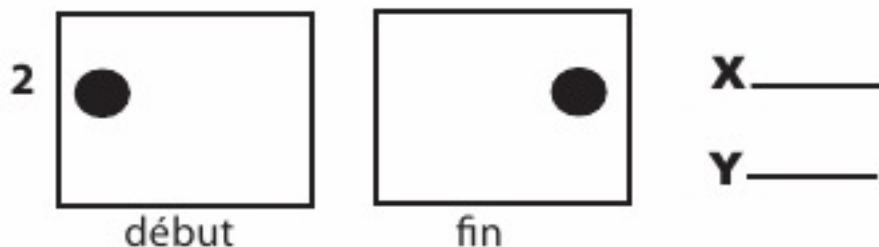
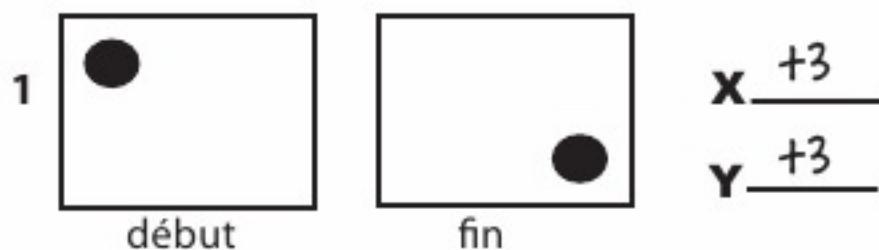
```
public void paintComponent(Graphics g) {
    g.setColor(Color.white);
    g.fillRect(0,0,this.getWidth(), this.getHeight());
    g.setColor(Color.green);
    g.fillOval(x,y,40,40);
}
```

getWidth() et getHeight() sont des méthodes héritées de JPanel.

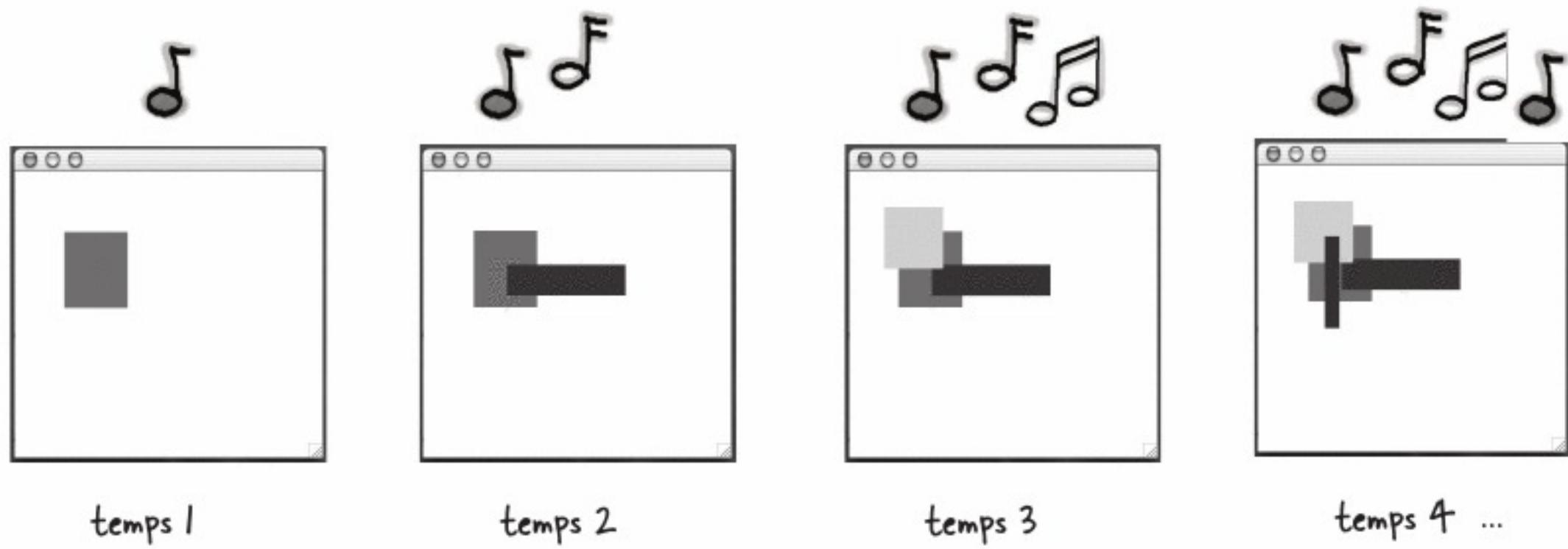


À vos crayons (optionnel, juste pour le plaisir) _____

Comment modifieriez vous les coordonnées x et y pour produire les animations ci-dessous?
(Supposons que les incrémentations du premier exemple soient de 3 pixels.)



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...)

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.

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 Créez une instance de message

```
ShortMessage first = new ShortMessage();
```

② Appeler setMessage() avec les instructions

```
first.setMessage(192, 1, instrument, 0)
```

③ Créez 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

```
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; ↗ Retourner l'événement (un MidiEvent chargé avec le message).
}
```

Les quatre arguments du message.

Le "tick" indiquant QUAND ce message doit arriver.

Créer le message et l'événement en utilisant les paramètres de la méthode.

Oh! Une méthode avec cinq paramètres.

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.*; ← Ne pas oublier d'importer le package.

public class MiniMusicPlayer1 {
    public static void main(String[] args) {

        try {

            Sequencer sequencer = MidiSystem.getSequencer(); ← Créer (et ouvrir) le séquenceur.
            sequencer.open();

            Sequence seq = new Sequence(Sequence.PPQ, 4); ← Créer une séquence et une piste.
            Track track = seq.createTrack(); ←

            for (int i = 5; i < 61; i += 4) { ← Créer un groupe d'événements pour monter la
                gamme (de la note 5 à la note b1 sur le piano).
                track.add(makeEvent(144, 1, i, 100, i));
                track.add(makeEvent(128, 1, i, 100, i + 2)); Appeler notre nouvelle méthode makeEvent()
                // fin de la boucle                                         Pour créer message et événement, puis ajouter
                sequencer.setSequence(seq);                                le résultat (le MidiEvent retourné par
                sequencer.setTempoInBPM(220); }                                makeEvent()) à la piste. Ce sont les paires
                sequencer.start(); }                                         NOTE ON (144) et NOTE OFF (128).
            } 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
}

```

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();
            sequenceur.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
}

```

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.

```

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;
}

```

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().)

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:

```

class MonPanneau extends JPanel implements ControllerEventListener {
    boolean msg = false; ← Le panneau est un auditeur.

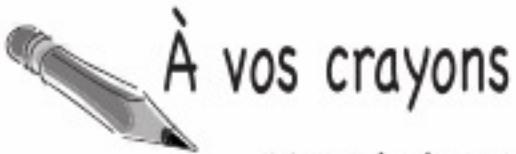
    public void controlChange(ShortMessage event) {
        msg = true; ← Nous positionnons un indicateur à faux, et il ne
                     sera vrai que si nous recevons un événement.
        repaint(); ← Nous avons un événement. Nous positionnons l'indicateur à vrai
                     et nous appelons repaint().
    }

    public void paintComponent(Graphics g) {
        if (msg) { ← Il nous faut un indicateur, parce que d'AUTRES événements pourraient déclencher
                     repaint() et nous ne voulons dessiner QUE si c'est un ControllerEvent.
            Graphics2D g2 = (Graphics2D) g;

            int r = (int) (Math.random() * 250);
            int gr = (int) (Math.random() * 250);
            int b = (int) (Math.random() * 250); ← Le reste est du code qui génère
            g.setColor(new Color(r, gr, b));           une couleur aléatoire et dessine un
                                                       rectangle semi-aléatoire.

            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

```



À vos crayons

```

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
}

```

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.

```
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
```



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.



```

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");
            }
        }
    }
}

```

You 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 ?





La piscine



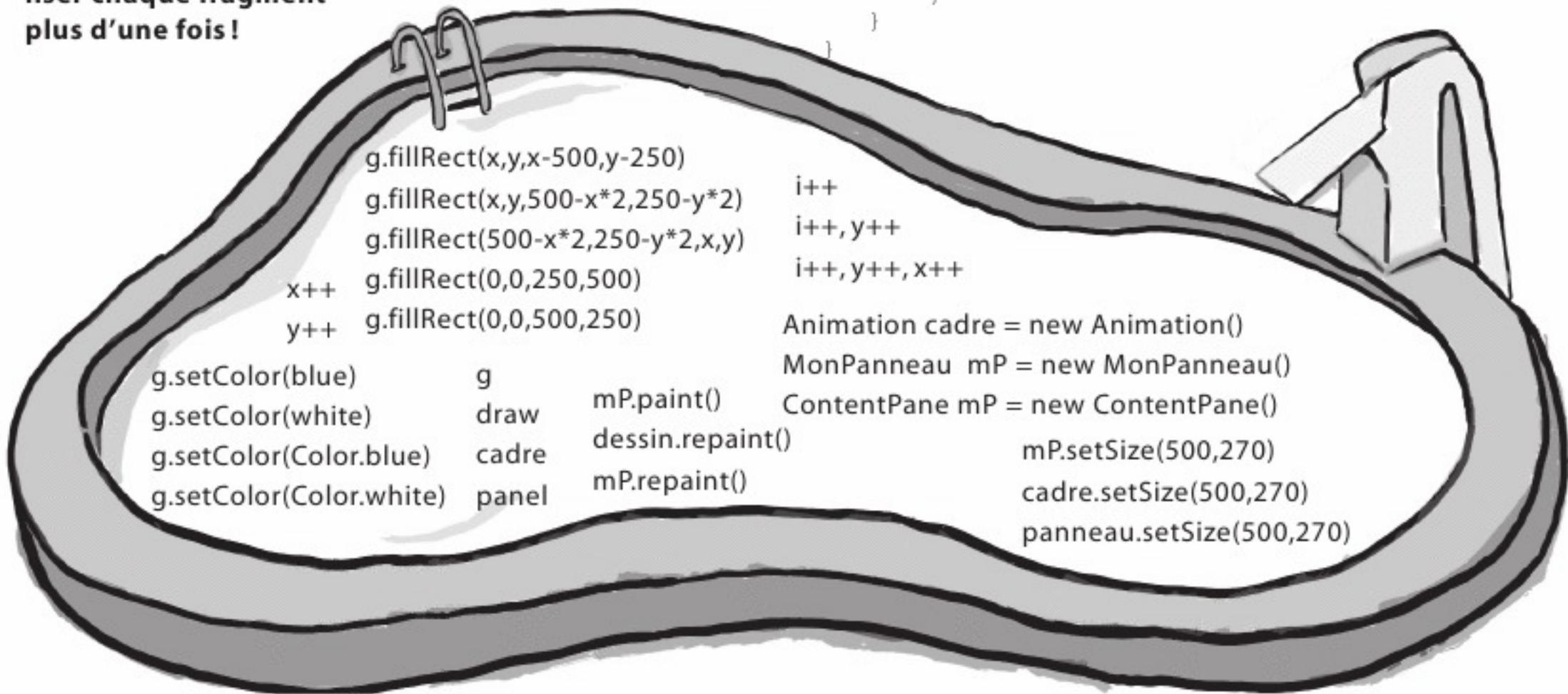
Votre **tâche** consiste à prendre des fragments de code dans la piscine et à les remettre dans le programme à la place des blancs. Vous **pouvez** utiliser le même fragment plusieurs fois, et vous n'avez pas besoin de les utiliser tous. Votre **but** est de créer une classe qui se compile, s'exécute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



Note : Vous pouvez utiliser chaque fragment plus d'une fois !



```

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 _____) {
                _____;
                _____;
                _____;
                _____;
            }
        }
    }
}

```

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**

objet event

Je renvoie des données à l'auditeur.

source d'événement

Une méthode addXxxListener() dit qu'un objet est une ____.

addActionListener()

Comment un auditeur s'enregistre.

paintComponent()

La méthode qui contient tout le code graphique.

objet interne

Je suis lié à une autre instance.

Graphics2D

Le «g» de (Graphics g), est en réalité de cet type.

repaint()

La méthode qui lance paintComponent().

javax.swing

Le package où résident les composants Swing.

Vous ÊTES le compilateur

```
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);
}
```

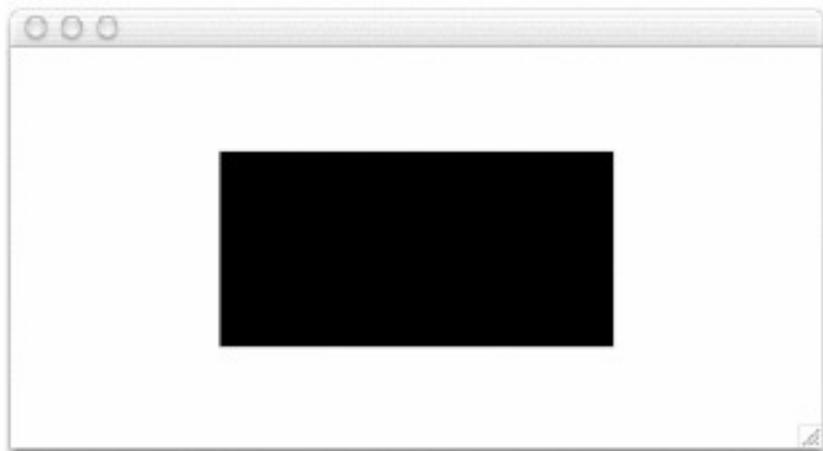
```
class BListener implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        if (b.getText().equals("A")) {
            b.setText("B");
        } else {
            b.setText("A");
        }
    }
}
```

ActionListener est une interface. Les interfaces sont implémentées, non étendues.



La piscine

L'étonnant rectangle bleu qui rétrécit



```
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);
        }
    }
}
```

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.

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`.

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);
```

Techniquement,
un widget est un
composant Swing.
Presque tous les
éléments d'une IHM
dérivent de javax.
swing.JComponent.

Placer des composants interactifs: Dans des composants d'arrière-plan:



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.

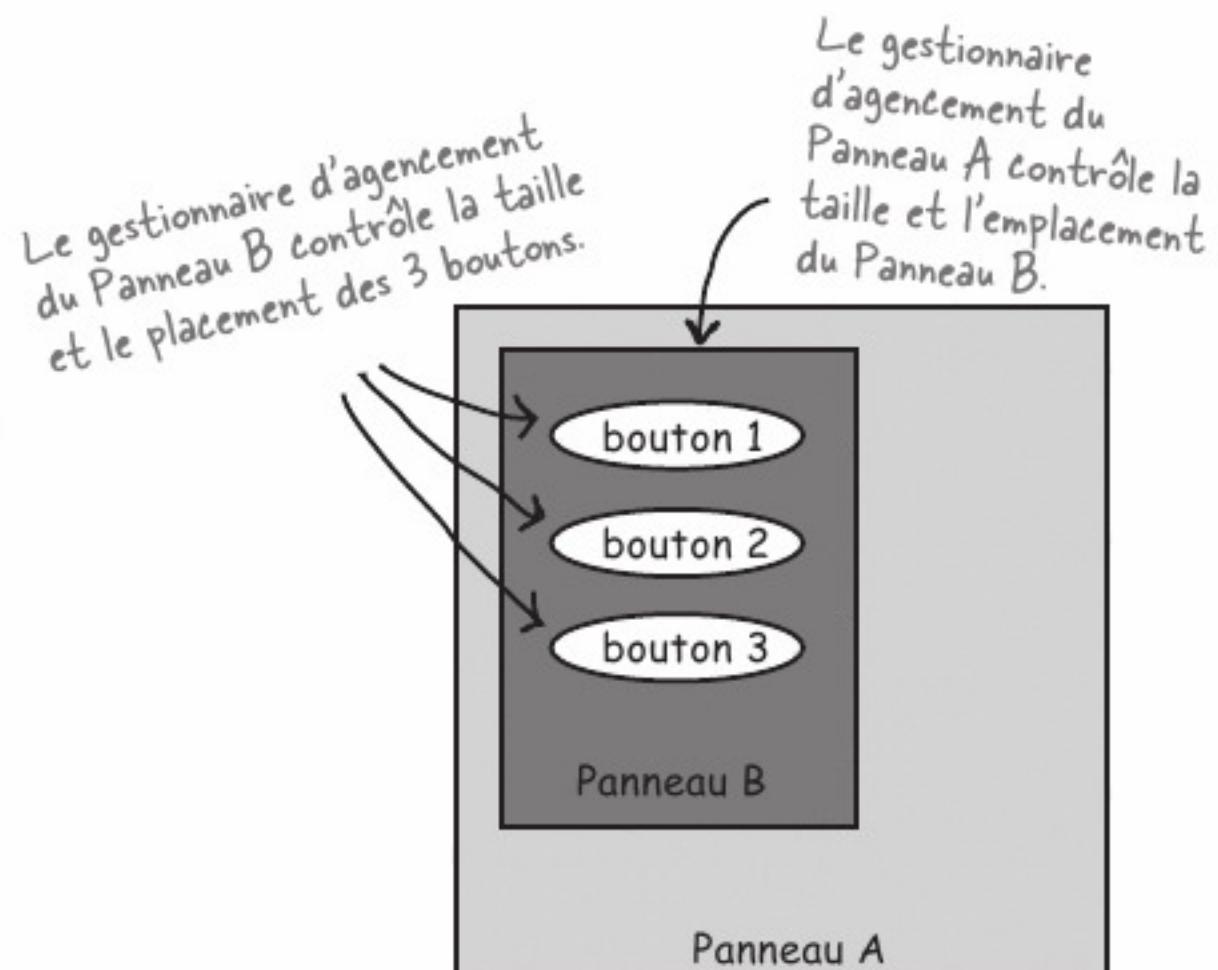
Si un panneau contient cinq composants, même si ces derniers ont leur propre gestionnaire d'agencement, c'est celui du panneau qui contrôle leur taille et leur placement. Si ces cinq éléments en contiennent d'*autres* à leur tour, ces *autres* composants sont placés selon le gestionnaire d'agencement 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.

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.

Voyons voir... Le premier bouton veut 30 pixels de large, le champ de texte en veut 50, et le cadre est de 200 pixels et je suis censé arranger tout ça verticalement...



gestionnaire d'agencement

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.

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!**



FlowLayout

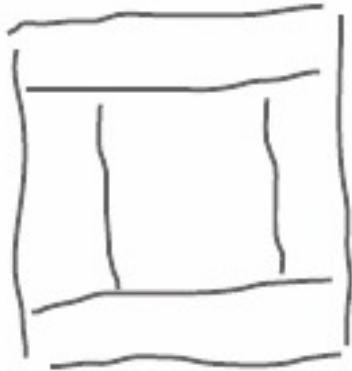
Un gestionnaire FlowLayout se comporte un peu comme un traitement de texte où les mots seraient remplacés par des composants. La taille des composants est respectée, et ils sont agencés de gauche à droite, avec le «retour à la ligne» activé. Quand un composant ne tient pas dans le sens horizontal, il passe à la «ligne» suivante. **FlowLayout est le gestionnaire par défaut des panneaux!**



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.





BorderLayout connaît cinq régions: east, west, north, south et center

Ajoutons un bouton à la région EAST:

```
import javax.swing.*;
import java.awt.*; ← BorderLayout est dans le package 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);
    }
}
```

↑
Spécifier la région.



MUSCLEZ VOS NEURONES

Comment BorderLayout a-t-il calculé cette taille de bouton?

Quels sont les facteurs que le gestionnaire d'agencement a dû prendre en compte?

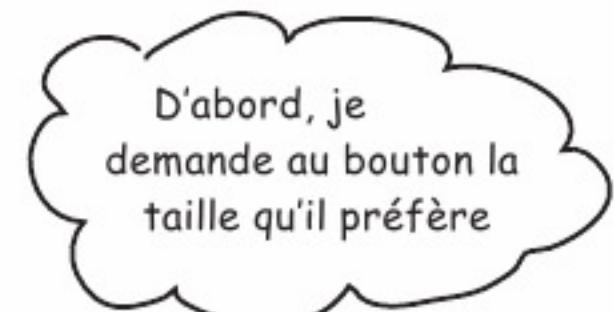
Pourquoi le bouton n'est-il pas plus haut ou plus large?



Regardez ce qui se passe quand nous ajoutons des caractères au 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);
}
```

Nous n'avons changé que le texte du bouton.



Comme il est dans la région EAST et que je suis un BorderLayout, je vais respecter sa largeur. Mais je me moque de ses préférences en hauteur; il sera de la hauteur du cadre, parce que c'est ma politique.



Ce bouton a la hauteur voulue, mais non la largeur.



La prochaine fois, j'irai avec FlowLayout. Il me donne TOUT ce que je veux.



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.



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();

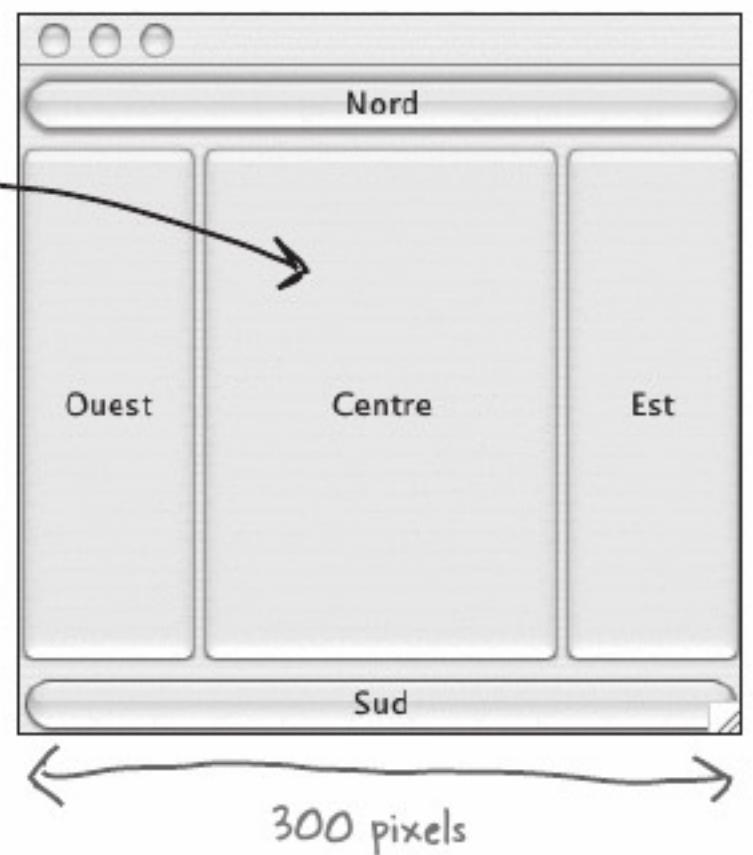
    JButton east = new JButton("Est");
    JButton west = new JButton("Ouest");
    JButton north = new JButton("Nord");
    JButton south = new JButton("Sud");
    JButton center = new JButton("Centre");

    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.



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.

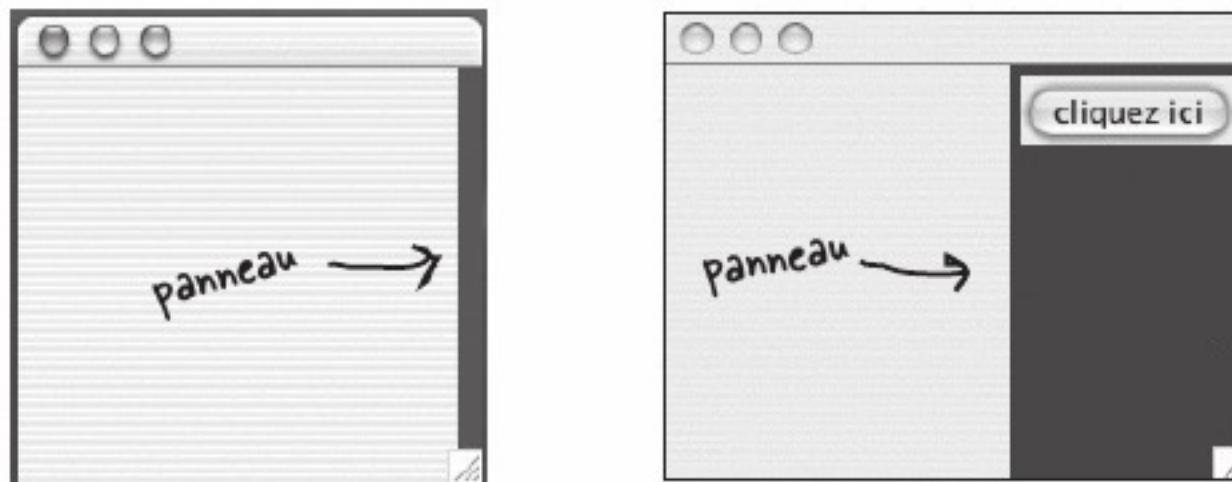
Ajoutons un bouton au panneau

```
public void go() {
    JFrame cadre = new JFrame();
    JPanel panneau = new JPanel();
    panneau.setBackground(Color.darkGray);

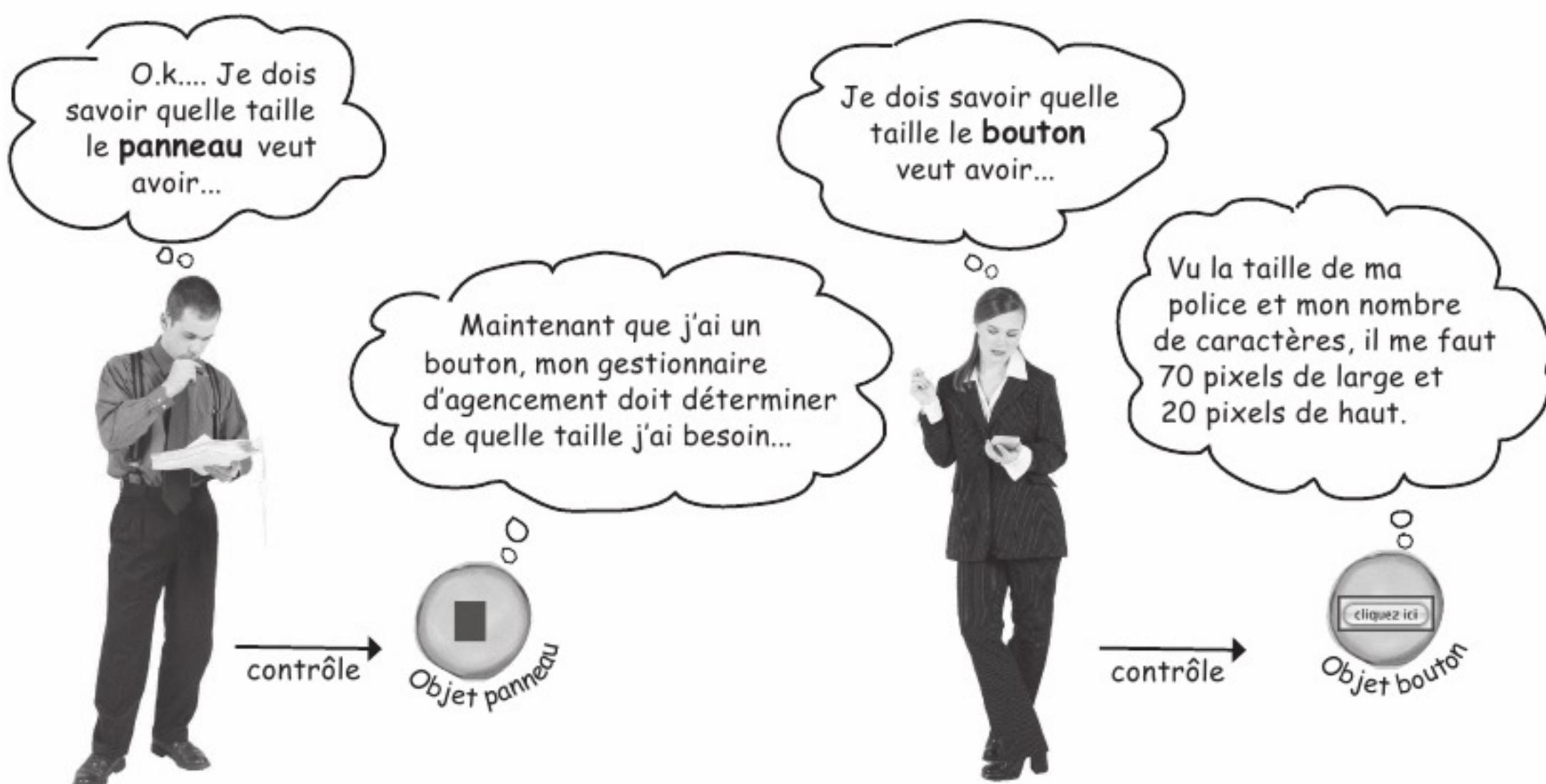
    JButton bouton = new JButton("cliquez ici");
    panneau.add(bouton);
    cadre.getContentPane().add(BorderLayout.EAST, panneau);

    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**

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"); ← Créer DEUX boutons.
    JButton boutonDeux = new JButton("ou la"); ←

    panneau.add(bouton); ← Ajouter les DEUX au panneau.
    panneau.add(boutonDeux);

    cadre.getContentPane().add(BorderLayout.EAST, panneau);
    cadre.setSize(250, 200);
    cadre.setVisible(true);
}

```

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 côté à côté.

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 modifions 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!)



BoxLayout à la rescouisse!

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".

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.getContentPane() = 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 côté à côté. 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.

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();
```

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 JScrollPane. Un JScrollPane 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); // Activer le retour à la ligne.
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.

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();
```

Exemple de JTextArea

```

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class TextAreal implements ActionListener {
    JTextArea texte;

    public static void main (String[] args) {
        TextAreal ihm = new TextAreal();
        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






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

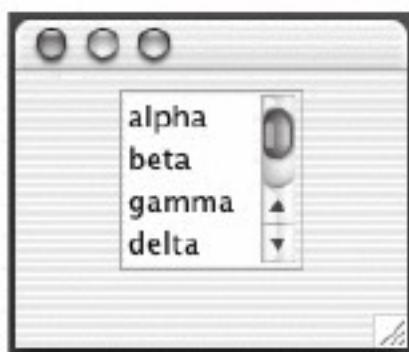
Q : Est-ce que les gestionnaires d'agencement ne nous donnent pas plus de soucis qu'ils n'en valent la peine ? Pourquoi se donner tout ce mal ? Pourquoi ne pas tout simplement coder en dur la taille et les coordonnées des composants ?

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 !

JList



Constructeur

```
String [] listEntries = {"alpha", "beta", "gamma", "delta",
                        "epsilon", "zeta", "eta", "theta"};  
  
liste = new JList(listEntries);
```

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.

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);
```

Procéder comme pour JTextArea : créer un JScrollPane (et lui transmettre la liste), puis ajouter celui-ci (PAS la liste) au panneau.

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);
```

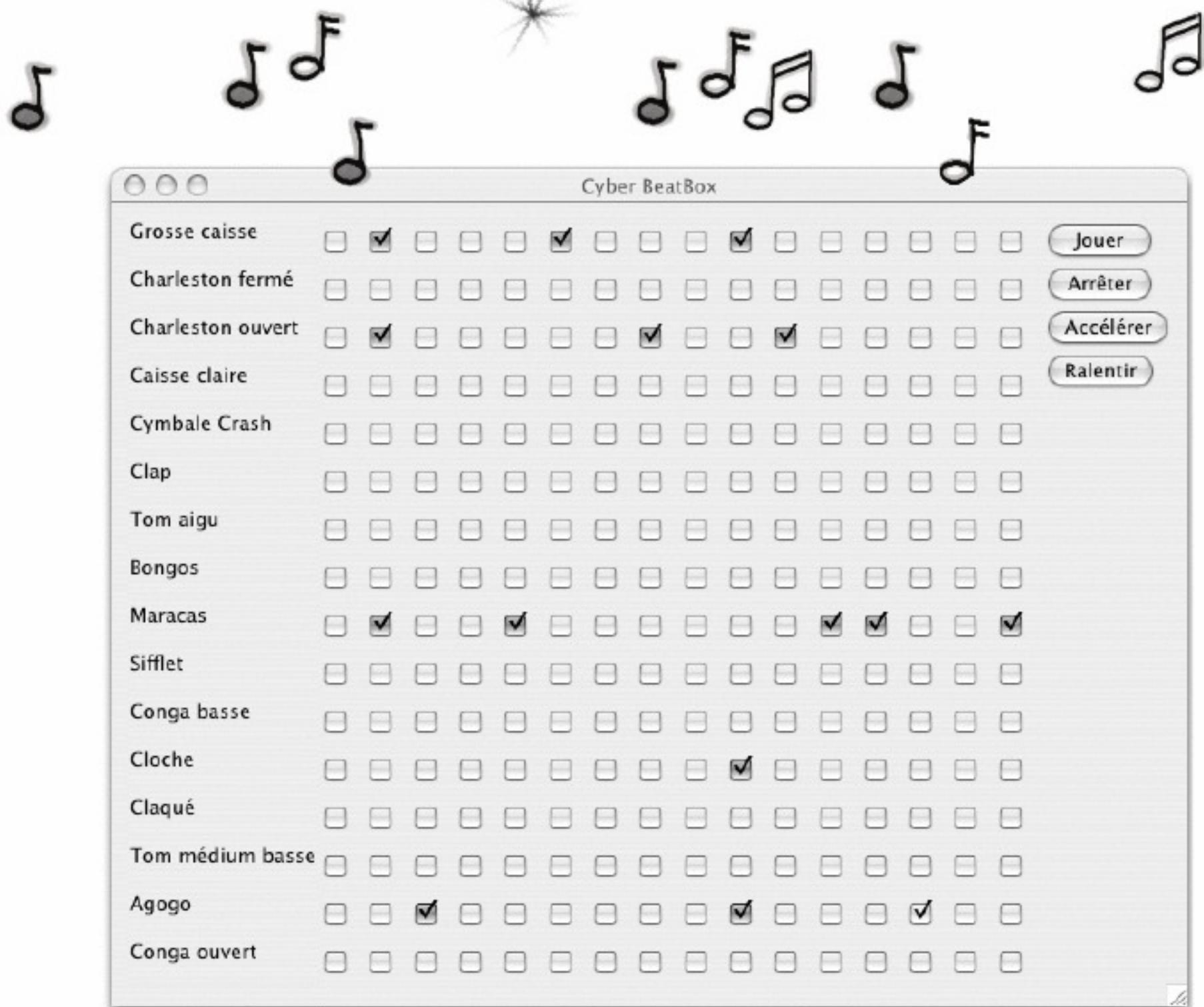
⑤ Gérer les événements (déterminer l'élément sélectionné)

```
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.

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 ».

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».

le code de la BeatBox

```
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; ← Nous stockons les cases à cocher
    Sequencer sequenceur;           dans une ArrayList.
    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)); ← 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.

        listeCases = new ArrayList<JCheckBox>();
        Box boiteBoutons = new Box(BoxLayout.Y_AXIS); ← Une bordure vide nous permet de créer une marge entre les bords du panneau et les composants. Purement esthétique.

        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");
        moinsVite.addActionListener(new EcouteMoinsVite());
        boiteBoutons.add(moinsVite);

        arrierePlan.add(listeCases);
        arrierePlan.add(boiteBoutons);
        leCadre.add(arrierePlan);
        leCadre.pack();
        leCadre.setVisible(true);
    }
}
```

Rien de spécial ici: le code de l'IHM dont vous avez déjà vu la majeure partie.

```

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
} // 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'HTML.
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.

le code de la BeatBox

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 méthode 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.

```

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.

Créer les événements NOTE ON et NOTE OFF et les ajouter à la piste.

La méthode utilitaire des Recettes de code du chapitre précédent. Rien de nouveau.

```

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));
        }
    }
}

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

```



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.

The image contains six numbered screen mockups (1 through 6) and a large question mark in the center, all set against a white background. Each screen has a standard window title bar at the top and a dark footer bar at the bottom. The screens are arranged as follows:

- 1**: A single large light gray rectangular area occupies most of the screen, with the word "tintin" centered within it.
- 2**: A dark gray horizontal bar at the top contains the word "milou". Below this, a light gray rectangular area contains the word "tintin".
- 3**: A light gray rectangular area occupies most of the screen, with the word "milou" centered at the bottom.
- 4**: A light gray rectangular area occupies most of the screen, with the word "tintin" centered at the top. A dark gray vertical bar on the right side contains the word "milou".
- 5**: A dark gray rectangular area occupies most of the screen, with the word "milou" centered at the top.
- 6**: A light gray rectangular area occupies most of the screen, with the word "tintin" centered at the bottom. A dark gray vertical bar on the right side contains the word "milou".

A large black question mark is positioned centrally between screens 1 and 4.

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

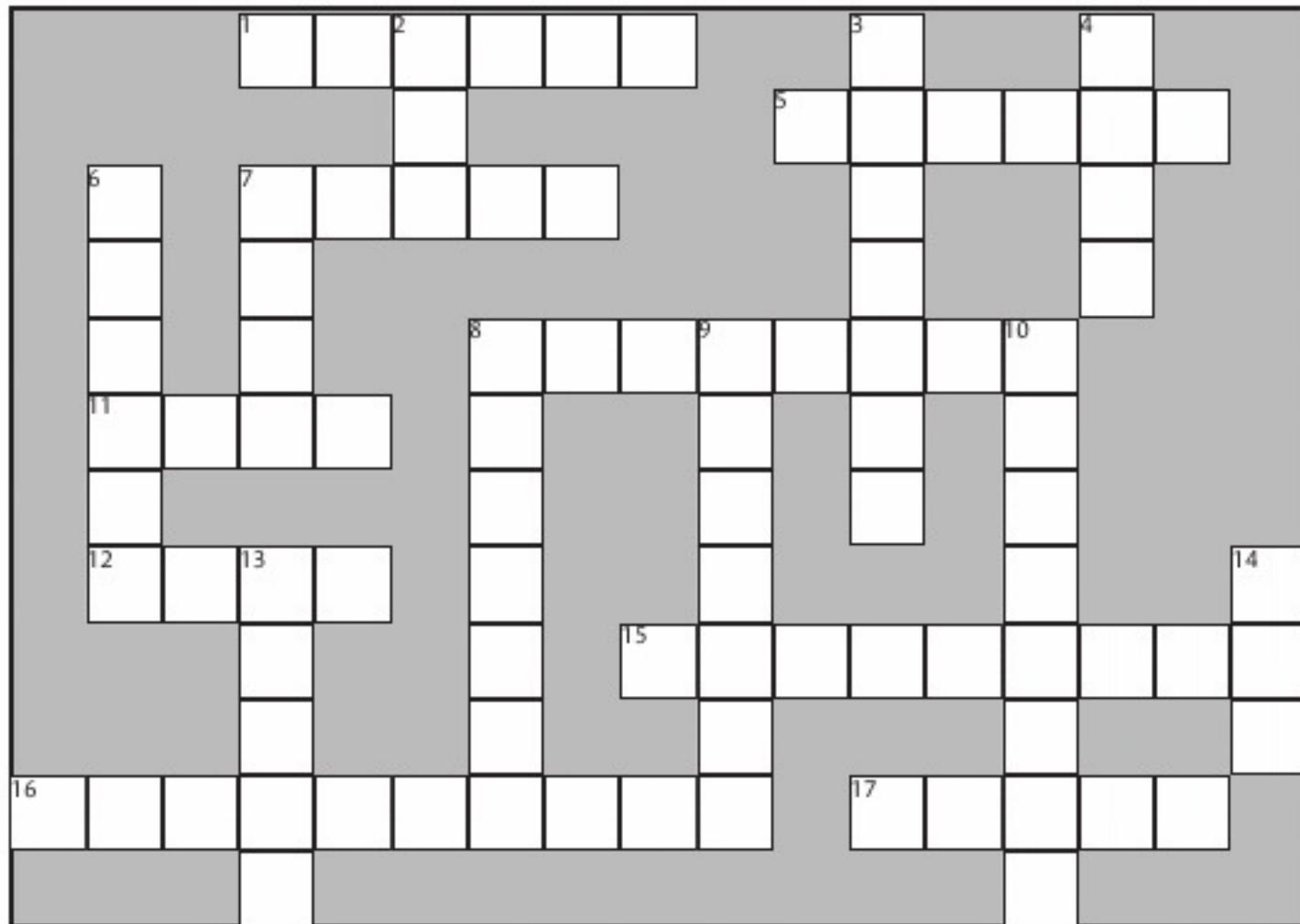
```
JFrame cadre = new JFrame();
 JPanel panneau = new JPanel();
 panneau.setBackground(Color.darkGray);
 JButton bouton = new JButton("tintin");
 JButton boutonDeux = new JButton("milou");
 panneau.add(bouton);
 cadre.getContentPane().add(BorderLayout.NORTH,boutonDeux);
 cadre.getContentPane().add(BorderLayout.EAST, 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);
```



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



Solutions des exercices

1**2****3****4****6**

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);
```

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);
```

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);
```

D

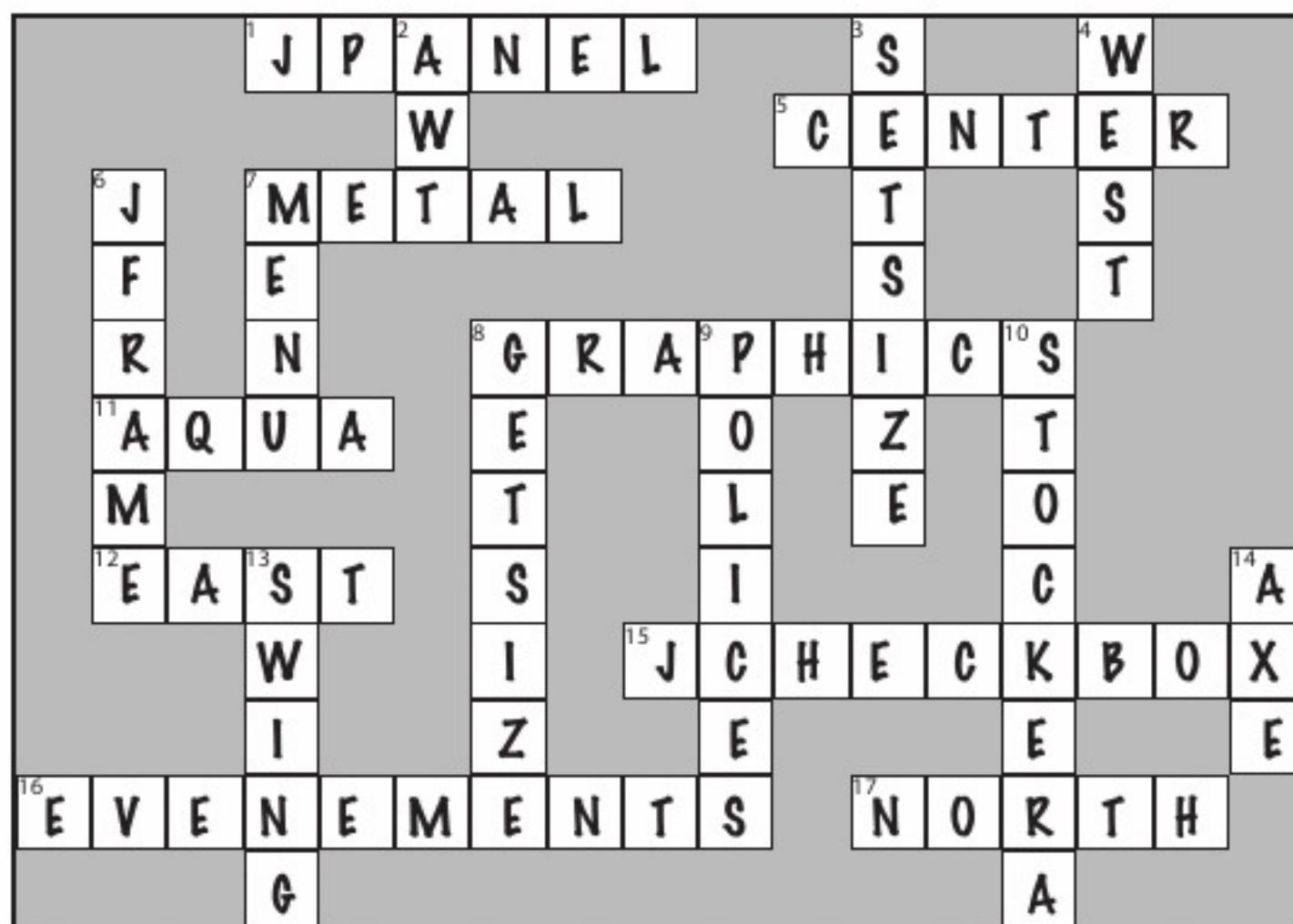
```
JFrame cadre = new JFrame();
 JPanel panneau = new JPanel();
 panneau.setBackground(Color.darkGray);
 JButton bouton = new JButton("tintin");
 JButton boutonDeux = new JButton("milou");
 panneau.add(bouton);
 cadre.getContentPane().add(BorderLayout.NORTH,boutonDeux);
 cadre.getContentPane().add(BorderLayout.EAST, panneau);
```

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);
```



Solution des mots-croisés



Sauvegarder les objets



Si je dois lire encore UN dossier plein de données, je crois que je vais le tuer. Il *sait* que je peux mémoriser des objets entiers, et vous croyez qu'il me laisserait faire? **NON**, ce serait trop facile. Bon, on verra bien ce qui va se passer quand je vais...

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, aplatisant, 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.

Capturez le rythme

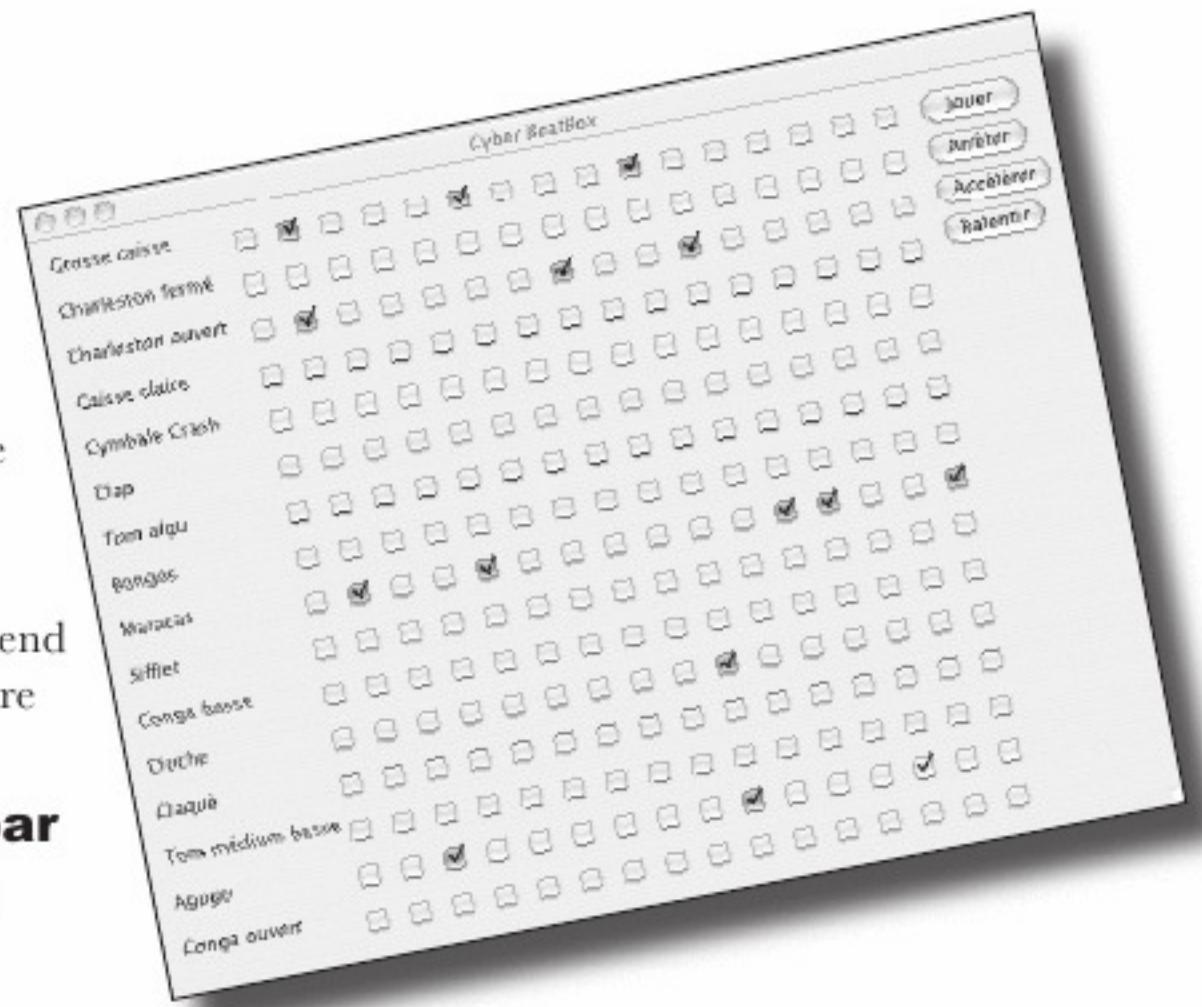
Vous avez *créé* le motif parfait. Vous voulez le *conserver*. Vous pourriez vous emparer d'un morceau de papier et gribouiller dessus. À la place, vous cliquez sur le bouton **Enregistrer** (ou vous choisissez Enregistrer dans le menu Fichier). Puis vous choisissez un nom, sélectionnez un répertoire, 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.

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).

Option un

① Écrivez les trois objets serialisés 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 :

"IsrPersonnage "%gê8MÛIpouvoirzLjava/lang/String;[armest[Ljava/lang/String;xp2tlfur[Ljava/lang/String;≠“VÀÈ{Gxptarcepeepoussiere~»tTrolluq~tmainsnuesgrandehacheq~xtMagicienuq~tformulesinvisibilite

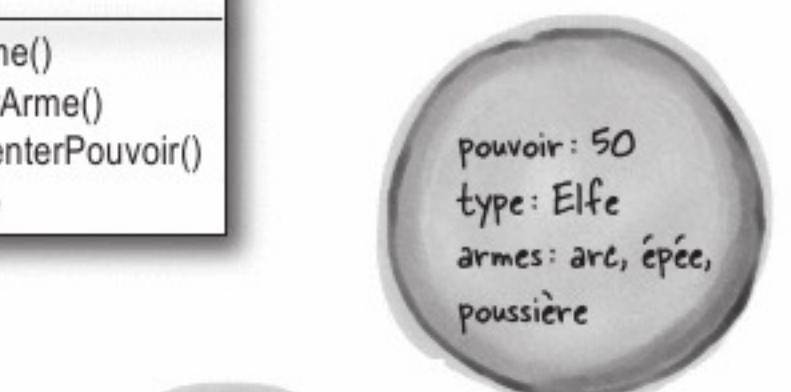
② Option deux

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

```
Personnage  
int pouvoir  
String type  
Arme[] armes  
  
getArme()  
utiliserArme()  
augmenterPouvoir()  
// suite
```



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 dans le mauvais ordre! Le type pourrait devenir « poussière » au lieu de « Elfe », tandis que l'Elfe deviendrait une arme...

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.

1 Créer un FileOutputStream

```
FileOutputStream fos = new FileOutputStream("MonJeu.ser");
```

Si le fichier « MonJeu.ser » n'existe pas, il sera créé automatiquement.

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);
```

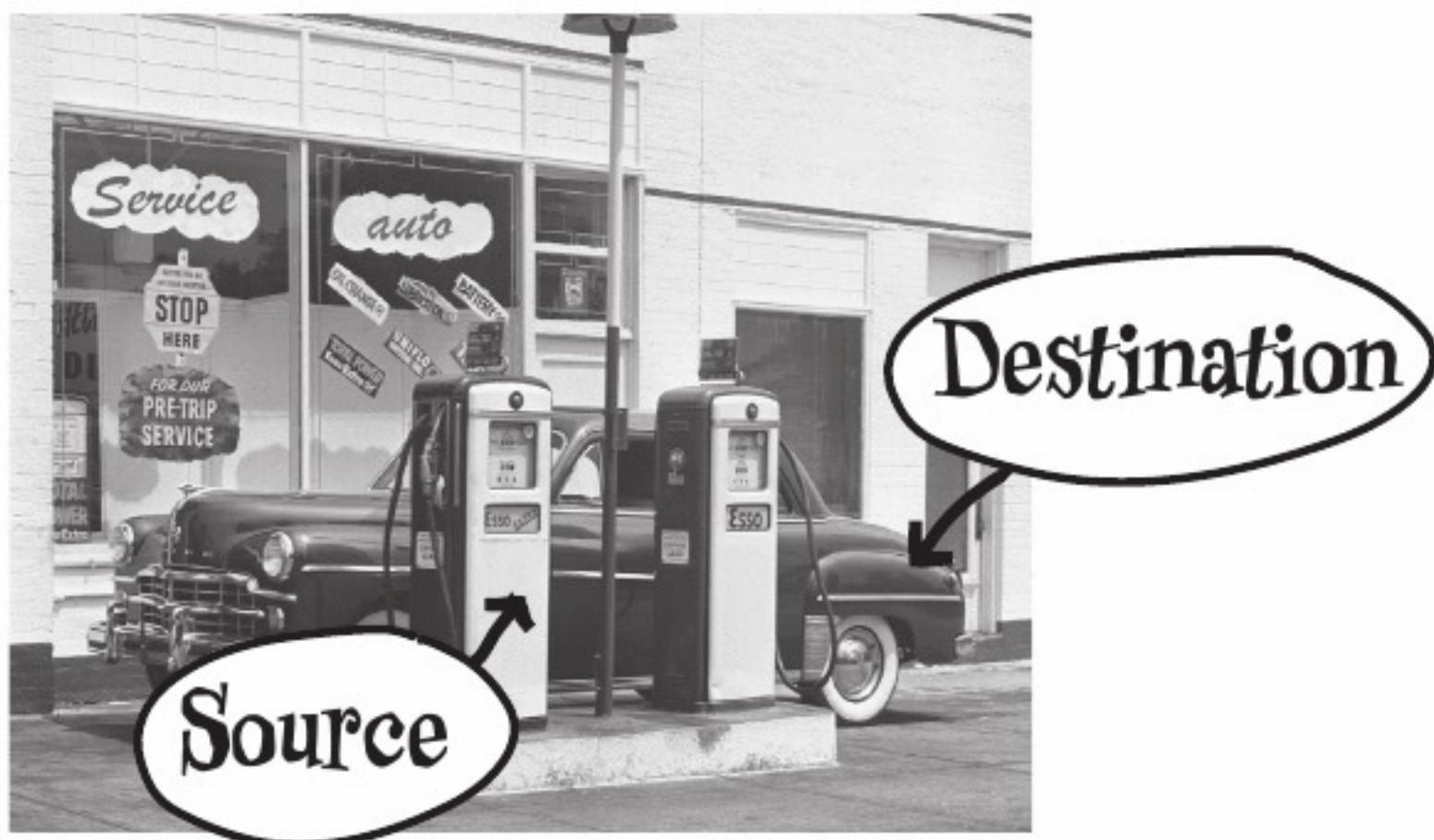
Sérialise les objets référencés par personnageUn, personnageDeux et personnageTrois, et les écrire dans le fichier « MonJeu.ser ».

4 Fermer l'ObjectOutputStream

```
oos.close();
```

La fermeture du premier flot entraînant celle des autres, le FileOutputStream se ferme automatiquement (et le fichier aussi).

Les données se déplacent en flots



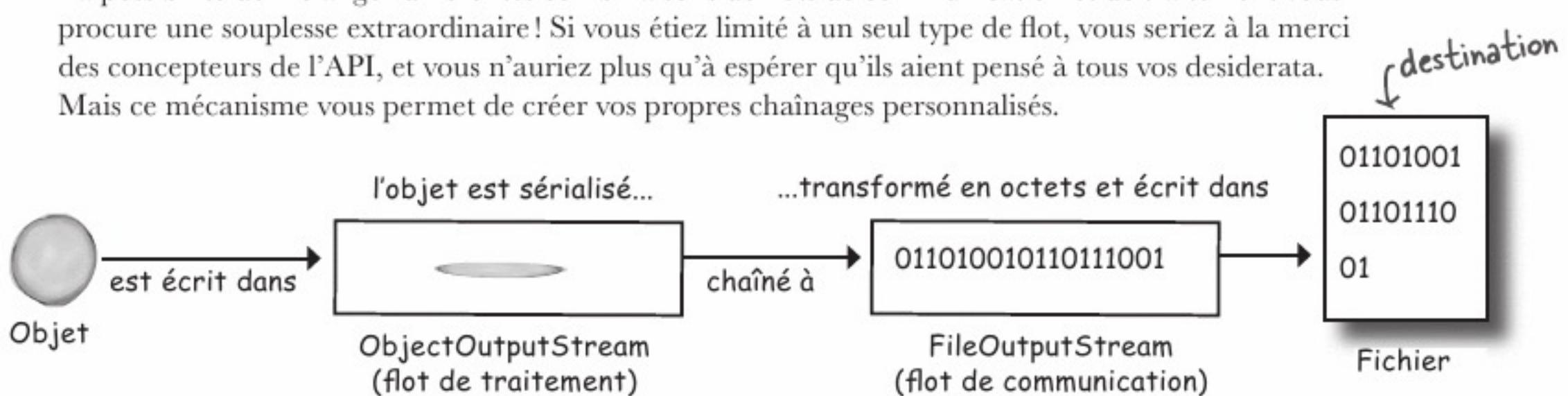
Les flots de communication représentent une connexion à une source ou une destination (fichier, socket, etc.) alors que les flots de traitement ne peuvent pas se connecter eux-mêmes et doivent être chaînés à un flot de communication.

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. ObjectOutputStream 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 désiderata. Mais ce mécanisme vous permet de créer vos propres chaînages personnalisés.

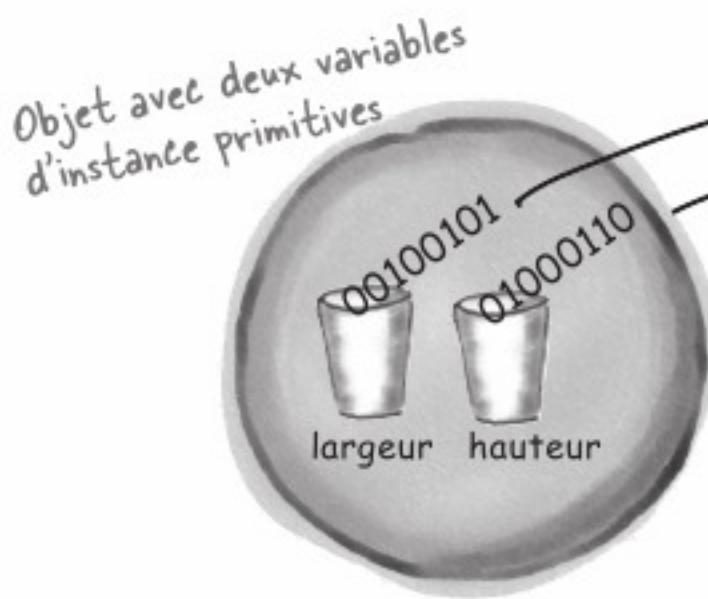


Qu'arrive-t-il réellement à un objet quand il est sérialisé ?

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 diffèrent entre une instance de classe d'une autre instance de la même classe.

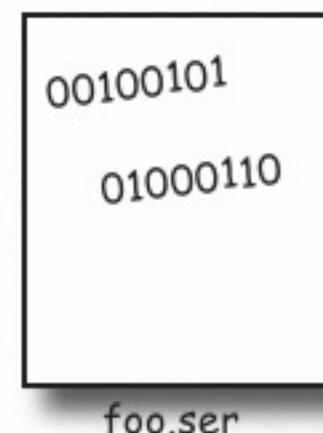


```
Foo coa = new Foo ();
coa.setLargeur(37);
coa.setHauteur(70);
```

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.



```
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.

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.



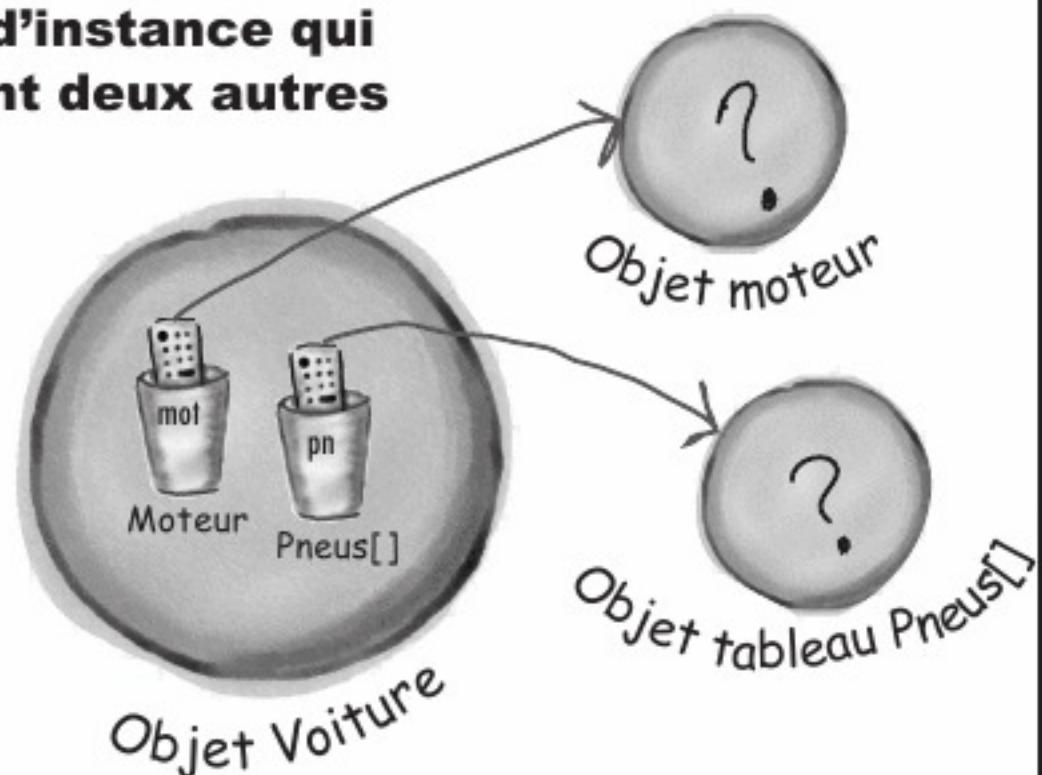
Gym du cerveau

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 réfèrentent deux autres objets.



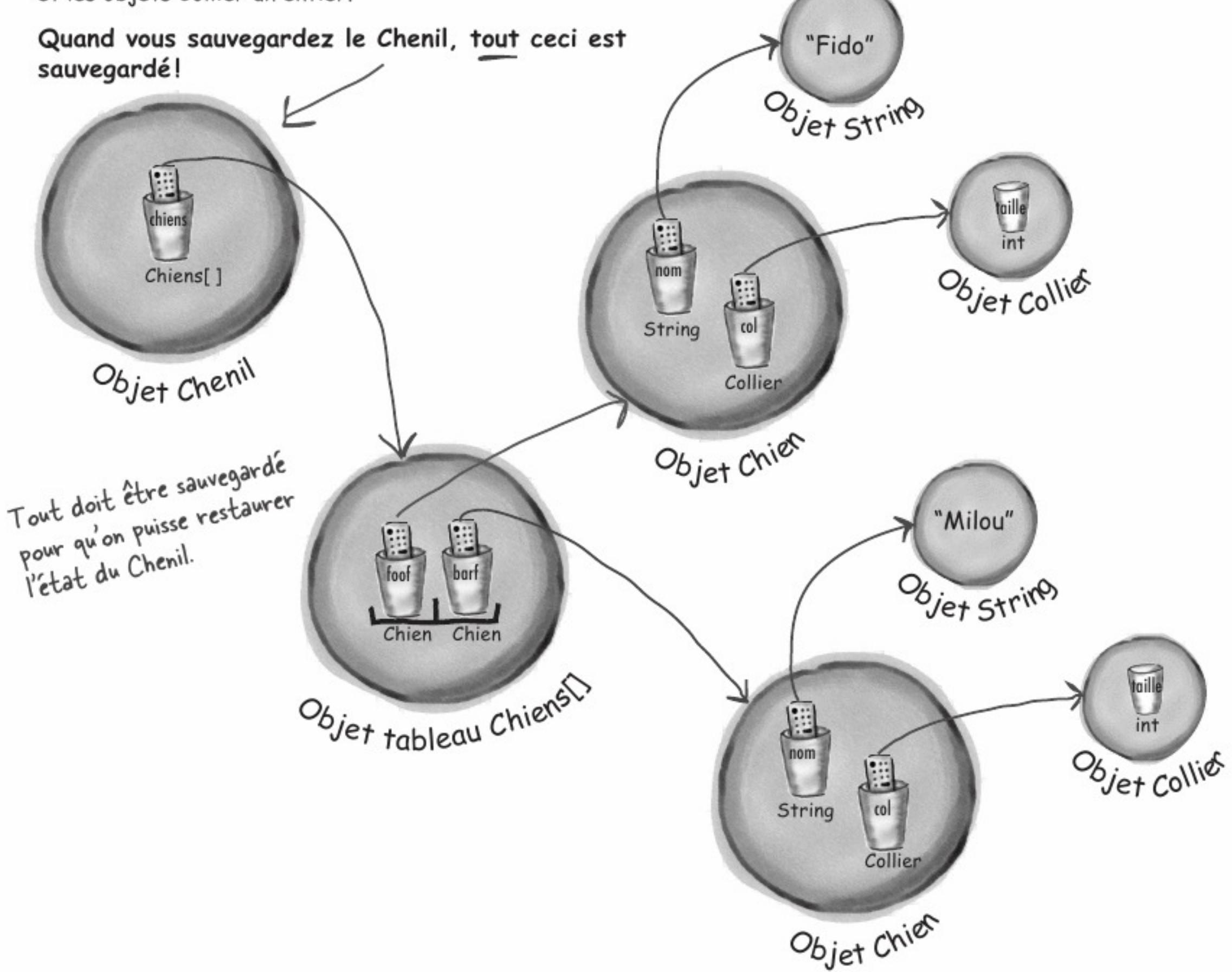
Que faut-il pour sauvegarder un objet Voiture?

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!

Cet objet Chenil a une référence à 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.

La sérialisation sauvegarde tout le graphe : d'abord le premier objet, puis tous les objets référencés par les variables d'instance.

Quand vous sauvegardez le Chenil, tout ceci est sauvegardé !



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.*;
```

Serializable étant dans le package java.io,
vous devez l'importer.

```
public class Boite implements Serializable {
```

Pas de méthodes à implémenter, mais
"implements Serializable" dit à la JVM qu'il
est possible de sérialiser les objets de ce type.

```
private int largeur;  
private int hauteur;
```

Ces deux valeurs seront sauvegardées.

```
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 {
```

Les opérations d'E/S peut lancer des exceptions.

```
    FileOutputStream fos = new FileOutputStream("foo.ser");  
    ObjectOutputStream oos = new ObjectOutputStream(fos);  
    oos.writeObject(maBoite);  
    oos.close();  
} catch(Exception ex) {  
    ex.printStackTrace();  
}
```

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.

```
}
```

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();
        }
    }
}
```

```
public class Canard{
    // code de Canard
}
```

Aie!! 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.

Les objets Mare sont sérialisables.

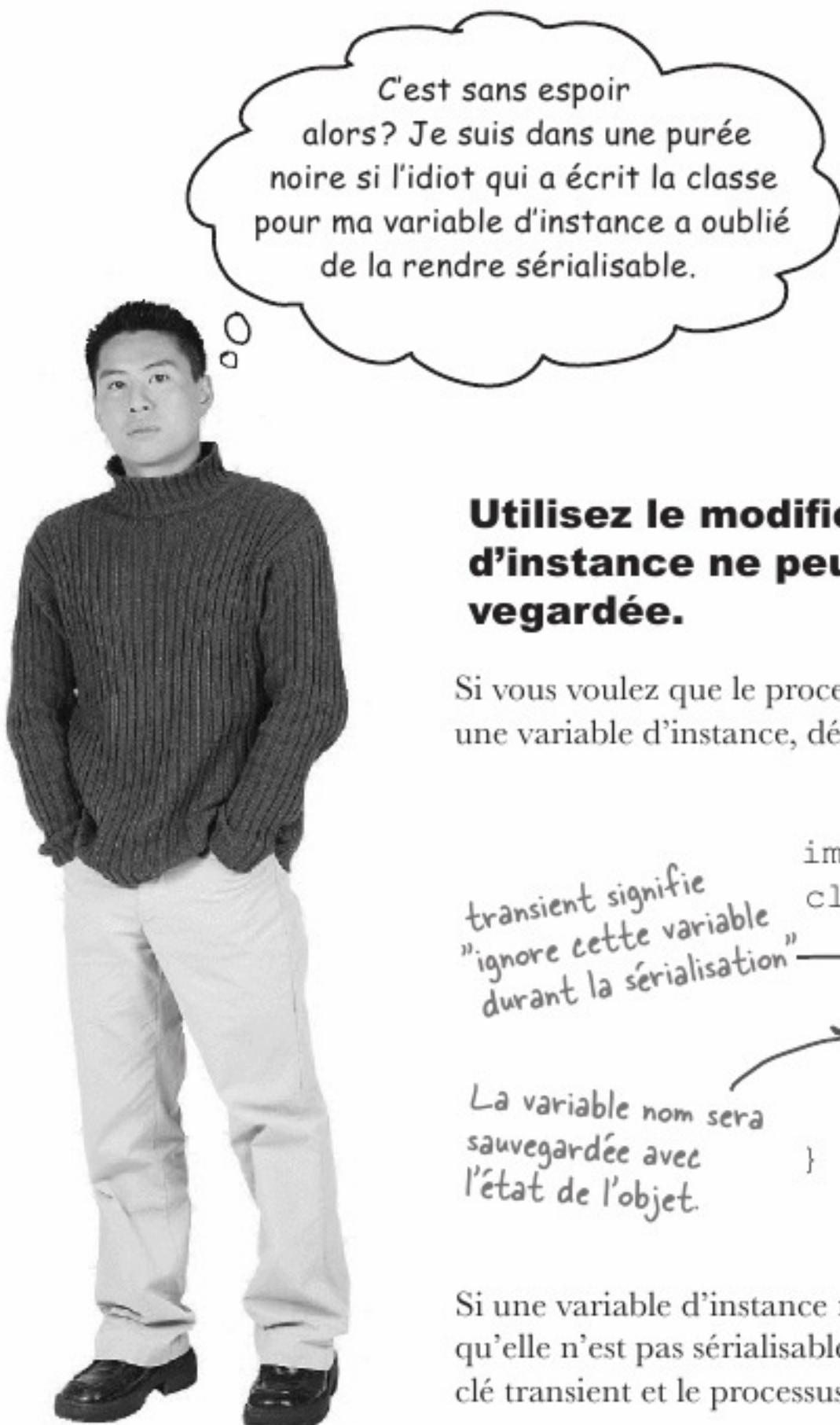
La classe Mare a une variable d'instance, un Canard.

Quand vous sérialisez maMare (un objet Mare) sa variable d'instance Canard est automatiquement sérialisée.

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)
```



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" → transient String IDCourant;

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.

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 déserialisée (nous allons en parler bientôt). En un mot, quand un objet est déserialisé 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 :

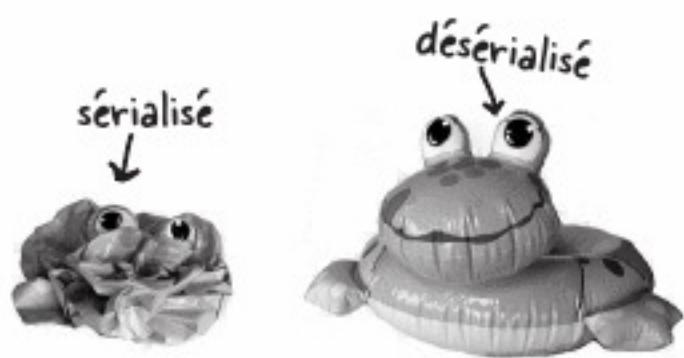
- 1) Quand l'objet est restauré, réinitialiser cette variable d'instance nulle à une valeur par défaut quelconque. Cela fonctionne si votre objet déserialisé ne dépend pas d'une valeur particulière. Par exemple, il peut être important que le Chien ait un Collier, mais, si tous les objets Collier sont identiques, peu importe que vous donnez 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 déserialisé 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 déserialisation, toutes les références à ce seul objet sont restaurées.

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");
```

Créer un objet FileInputStream connecter à un fichier existant. Il sait comment se

Si le fichier "MonJeu.ser" n'existe pas, vous aurez une exception.

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 objets

```
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 reconvertis pour qu'elle retrouve son type réel.

5 Fermer l'ObjectInputStream

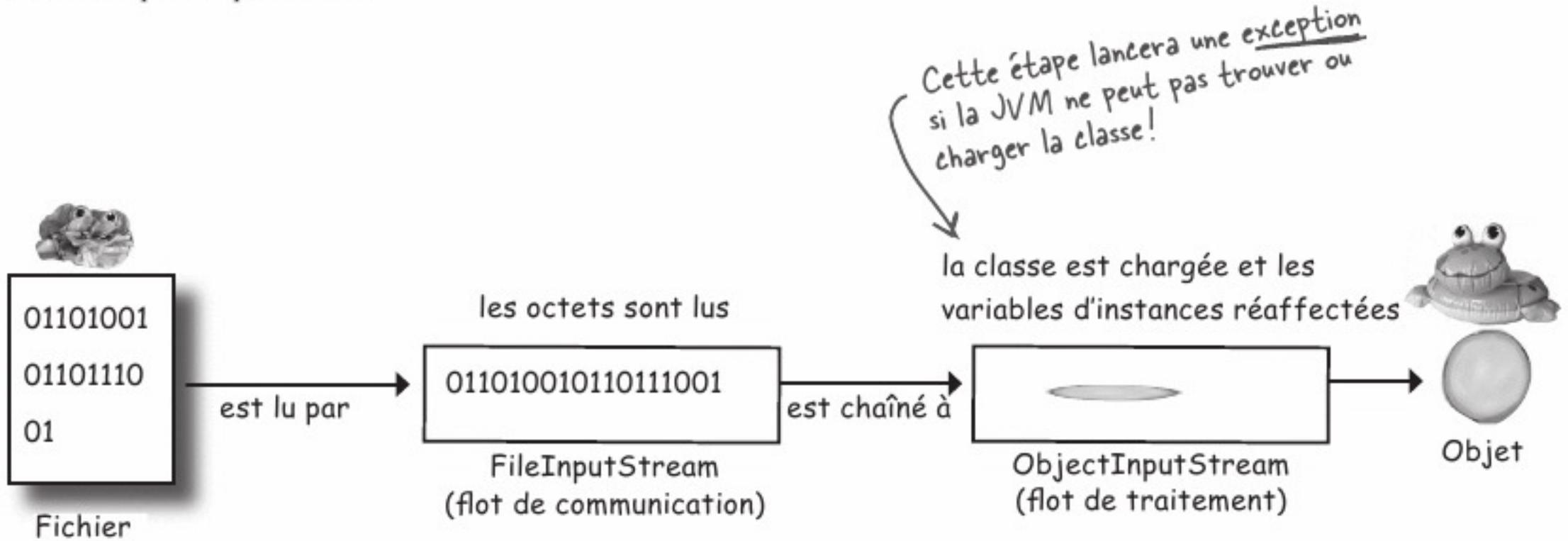
```
ois.close();
```

La fermeture du premier flot entraînant celle des autres, le FileInputStream se ferme automatiquement (et le fichier aussi).

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.

- 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 déserialisé, 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.

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; ← Initialisés à null: on ne peut pas accéder aux objets sur le tas.
    trois = null;
}

```

Créons quelques personnages

```

try {
    ObjectInputStream ois = new ObjectInputStream(new FileInputStream("Jeu.ser"));
    Personnage unBis = (Personnage) ois.readObject();
    Personnage deuxBis = (Personnage) ois.readObject();
    Personnage troisBis = (Personnage) ois.readObject();
}

```

Maintenant, relire le fichier

```

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();
}
}

```

Vérifier que cela fonctionne

```

% java TestSauvegarde
Elfe
Troll
Magicien

```



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.

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(IObjet)` 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 déserialisés 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.

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 tableau 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,epée,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.*; // Nous avons besoin du package java.io pour accéder à FileWriter.
```

```
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();
        }
    }
}
```

TOUT le code d'E/S doit être dans un bloc try/catch. Tout ceci peut lancer une IOException!!

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!

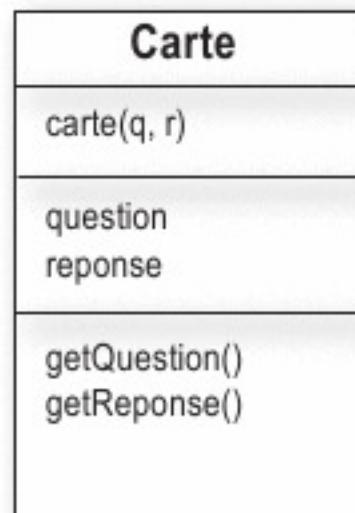
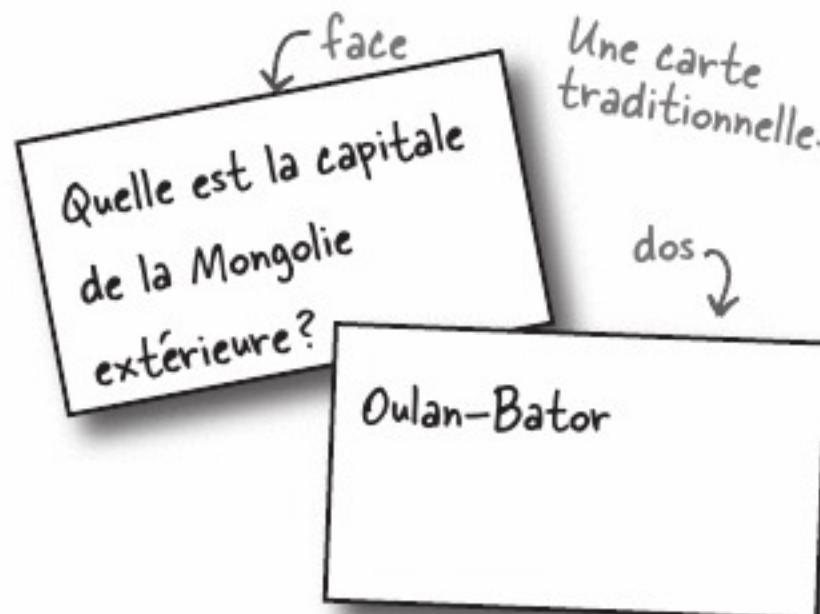
écrire dans un fichier texte

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.

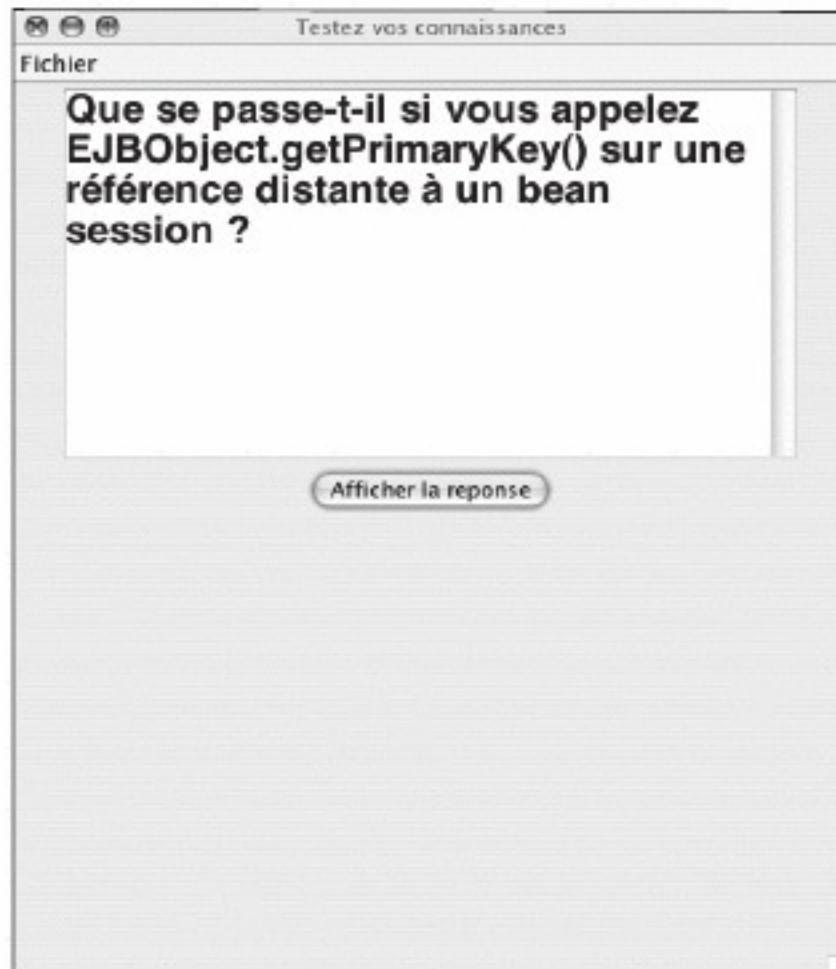
Nous allons en créer une version électronique qui a trois classes :

- 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.

EditeurCartes (grandes lignes du code)

```

public class EditeurCartes {

    public void go() {
        // construit et affiche l'IHM
    }
}

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
    }
}

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
    }
}

Classe interne
private class MenuNouveau ActionListener {
    public void actionPerformed(ActionEvent ev) {
        // efface la liste des cartes 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)
}

```

Construit et affiche l'IHM, crée et enregistre les auditeurs d'événements.

Déclenché quand l'utilisateur clique sur "Carte suivante". Signifie que l'utilisateur veut enregistrer cette carte dans la liste et en commencer une nouvelle.

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.)

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).

Appelé par MenuEnregistrer, effectue l'écriture proprement dite.

EditeurCartes : le code

```
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 newItem = new JMenuItem("Nouveau");
        JMenuItem saveItem = new JMenuItem("Enregistrer");

        newItem.addActionListener(new MenuNouveau());
        saveItem.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).
    }
}
```

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.

```

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").

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`.

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());
```

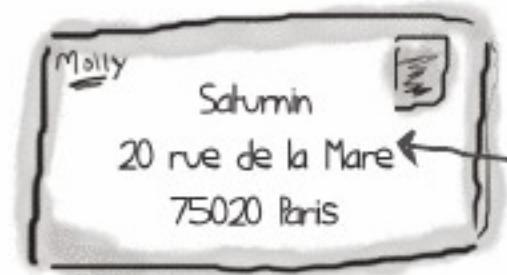
⑤ Supprimer un fichier ou un répertoire (retourne true en cas de succès)

```
boolean supprime = f.delete();
```

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 !



Un objet File représente le nom de fichier "Jeu.txt".

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.

Jeu.txt

```
50,Elfe,arc,epée,poussière
200,Troll,mains nues,grande
hache
120,Magicien,formules,invisibilité
```

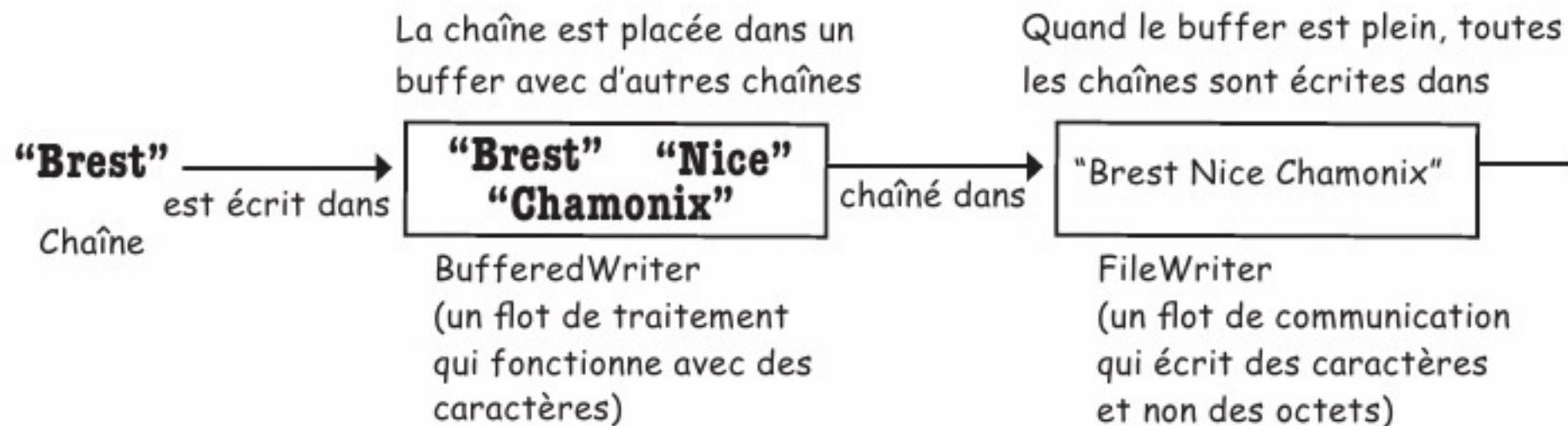
↑
Un objet File ne représente PAS les données contenues dans le fichier (et ne vous permet pas d'y accéder directement)!

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));
```

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!».

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.

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

`import java.io.*;`

Ne pas oublier d'importer ce package.

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

`try {`

`File monFichier = new File("MonTexte.txt");`
`FileReader fr = new FileReader(monFichier);`

MonTexte.txt

`BufferedReader br = new BufferedReader(fr);`

Un FileReader est un flot de communication pour les caractères, qui se connecte à un fichier texte.

Créer une variable de type String qui contiendra chaque ligne à mesure qu'elle est lue.

`String ligne = null;`

Chainer 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).

`while ((ligne = br.readLine()) != null) {`

`System.out.println(ligne);`

`}`

`br.close();`

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".

`}`

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
    }
}

```

QuizCartes : le code

```
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.

```

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.

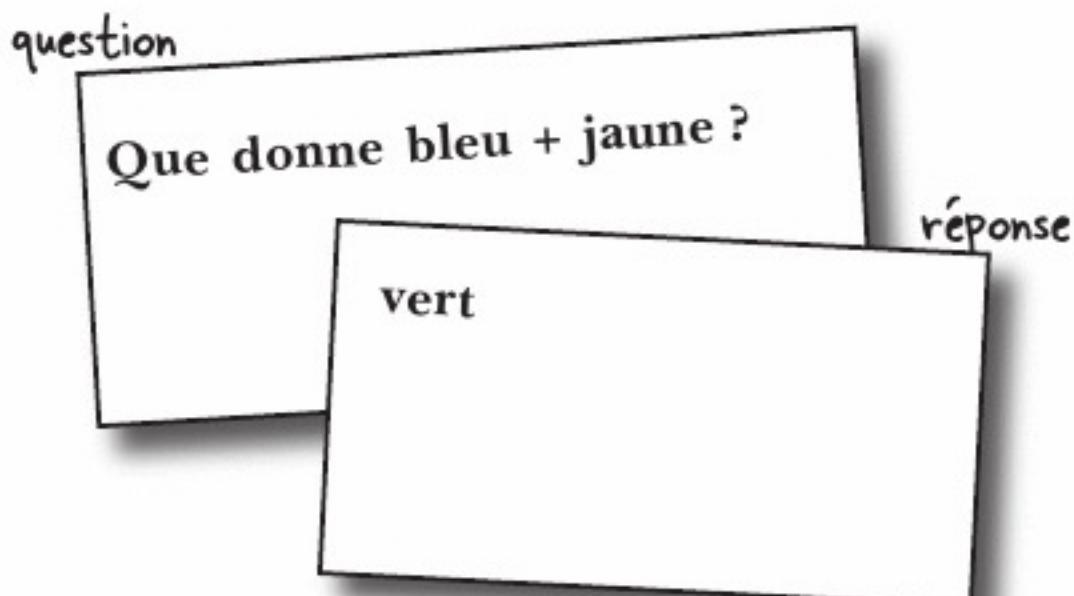
Créer un BufferedReader chaîné à un nouveau FileReader, en passant au FileReader l'objet File que l'utilisateur a choisi dans le dialogue

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.

Analyser une chaîne avec split()

Imaginez une flashcard comme celle-ci:



Sauvegardée dans un fichier texte:

Que donne bleu + jaune/vert
Que donne bleu + rouge/violet

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 ».



```
String aTester = "Que donne bleu + jaune ?/vert";
```

```
String[] resultat = aTester.split("/");
```

```
for (String partie:result) {
```

```
    System.out.println(partie);
```

```
}
```

Dans l'application QuizCartes, c'est à cela que ressemble une ligne quand elle est lue dans le fichier.

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. Dans cet exemple, il n'y a que deux « Que donne bleu + jaune ? » et « vert ».

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.*

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 déserialiser 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 déserialisés, 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.

```
101101  
101101  
10101000010  
1010 10 0  
01010 1  
1010101  
10101010  
1001010101
```

Chien.class

La classe a l'ID de version #343.

- ② Vous sérialisez un objet Chien en utilisant cette classe.



- ③ Vous modifiez la classe Chien.

```
101101  
101101  
101000010  
1010 10 0  
01010 1  
100001 1010  
0 00110101  
1 0 1 10 10
```

Chien.class

L'ID de version devient #728.

- 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 ».

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 déserialisé, elle peut porter un serialVersionUID différent et la déserialisation 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 déserialiser 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 déserialiser 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 {

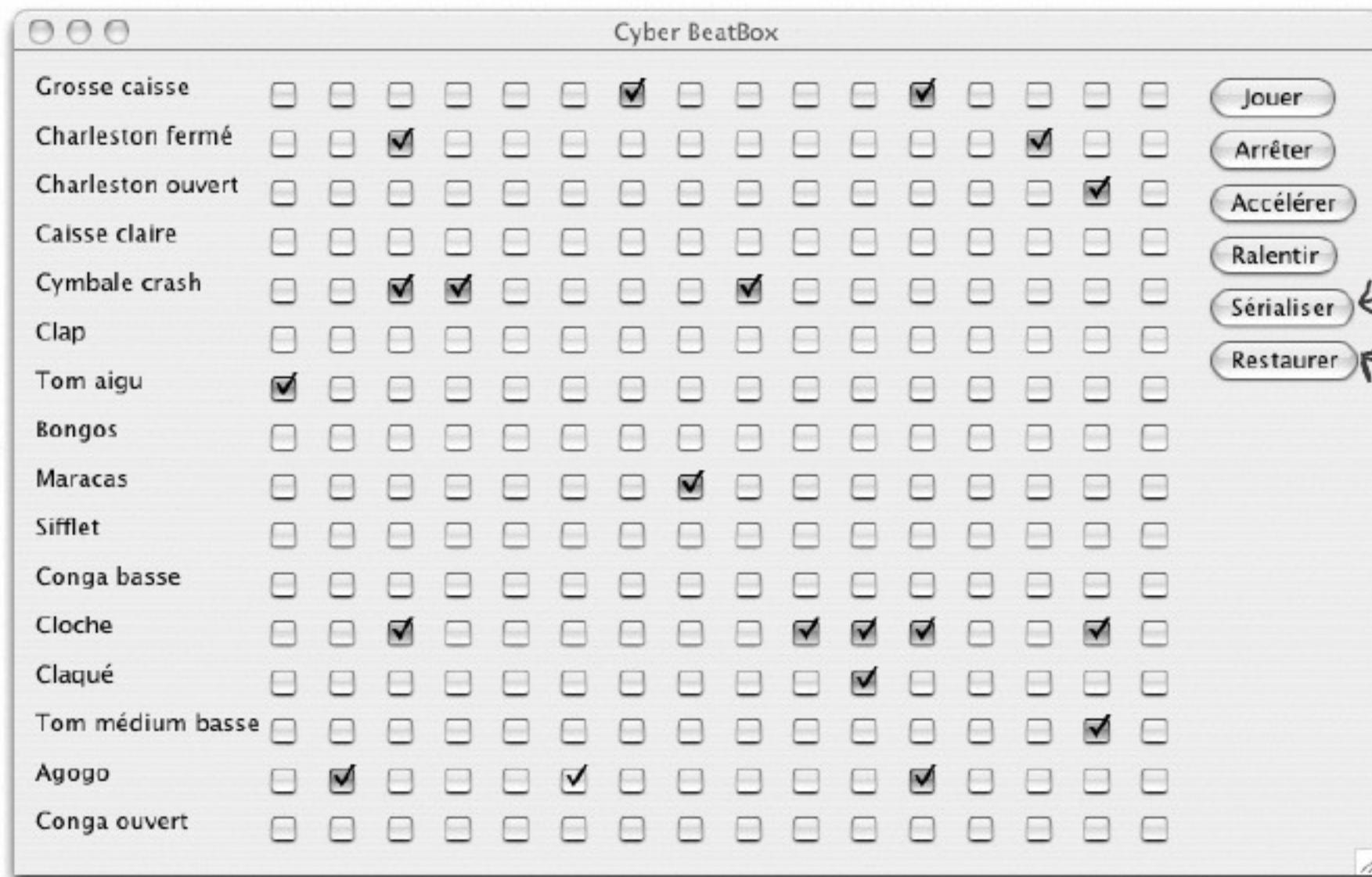
    static final long serialVersionUID =
        -5849794470654667210L;

    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 déserialisé avec des valeurs par défaut pour les variables d'instance ajoutées à la classe après la sérialisation du Chien.

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.

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.

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(); ← Lire le seul objet du fichier (le tableau de booléens) et le reconvertis en tableau de booléens (souvenez-vous que readObject() retourne une référence de type Object).  
        } 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
```

Maintenant, restaurer l'état de chaque case à cocher contenue dans l'ArrayList d'objets JCheckBox (listeCases).

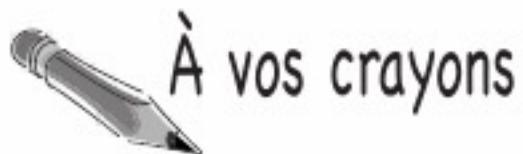
Arrêter de jouer le morceau actuel et reconstruire la séquence en utilisant le nouvel état des cases à cocher de l'ArrayList.



À vos crayons

Cette version présente une énorme limite ! Quand vous cliquez sur le bouton Srialiser, 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.



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 FileInputStream("Jeu.ser"));
Personnage unEncore = (Personnage) ois.readObject();
```

SERRER
←
À DROITE

exercice: vrai ou faux



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 BufferedWriter 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.



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 {
    public int x = 3;
    transient long y = 4;
    private short z = 5;

    try {
        FileOutputStream fos = new
            FileOutputStream("dj.ser");
        ObjectOutputStream oos = new
            ObjectOutputStream(fos);
        d = new Donjons();
        oos.writeObject(d);
        oos.close();
    } catch (Exception e) {
        e.printStackTrace();
    }
}

class DonjonsTest {
    import java.io.*;
    public static void main(String [] args) {
        Donjons d = new Donjons();
        System.out.println(d.getX() + d.getY() + d.getZ());
    }
}

public class Donjons {
    short getZ() {
        return z;
    }

    int getX() {
        return x;
    }

    long getY() {
        return y;
    }
}

```

Fichier Edition Fenêtre Aide Torture

```
% java DonjonsTest
12
8
```

```

ObjectOutputStream oos = new
    ObjectOutputStream(fos);
oos.writeObject(d);

public static void main(String [] args) {
    Donjons d = new Donjons();
}
```



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 BufferedWriter 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**



Ouf, on en est aux réponses. Je commençais à en avoir assez de ce chapitre.



```
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

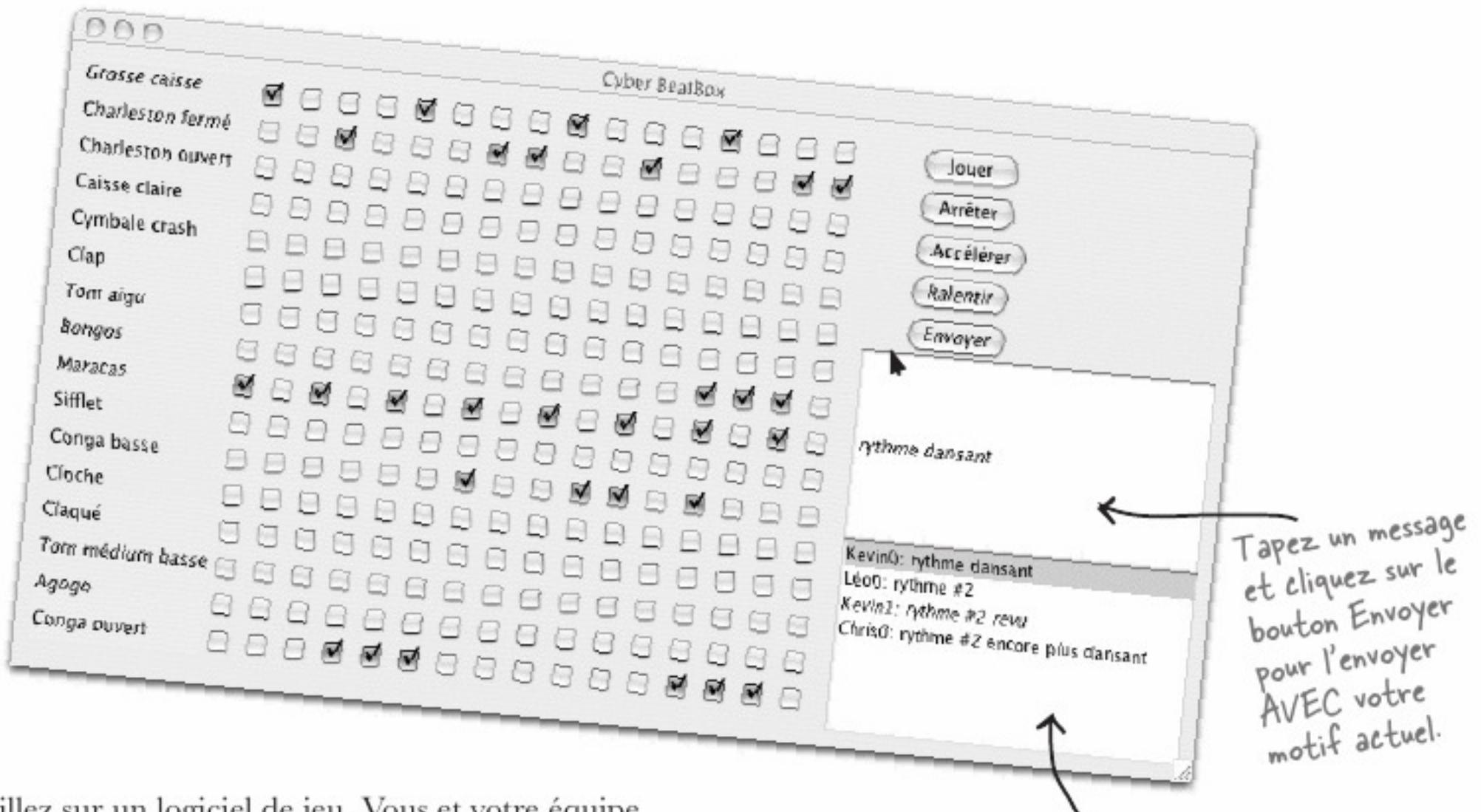
```
% java DonjonsTest  
12  
8
```


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.

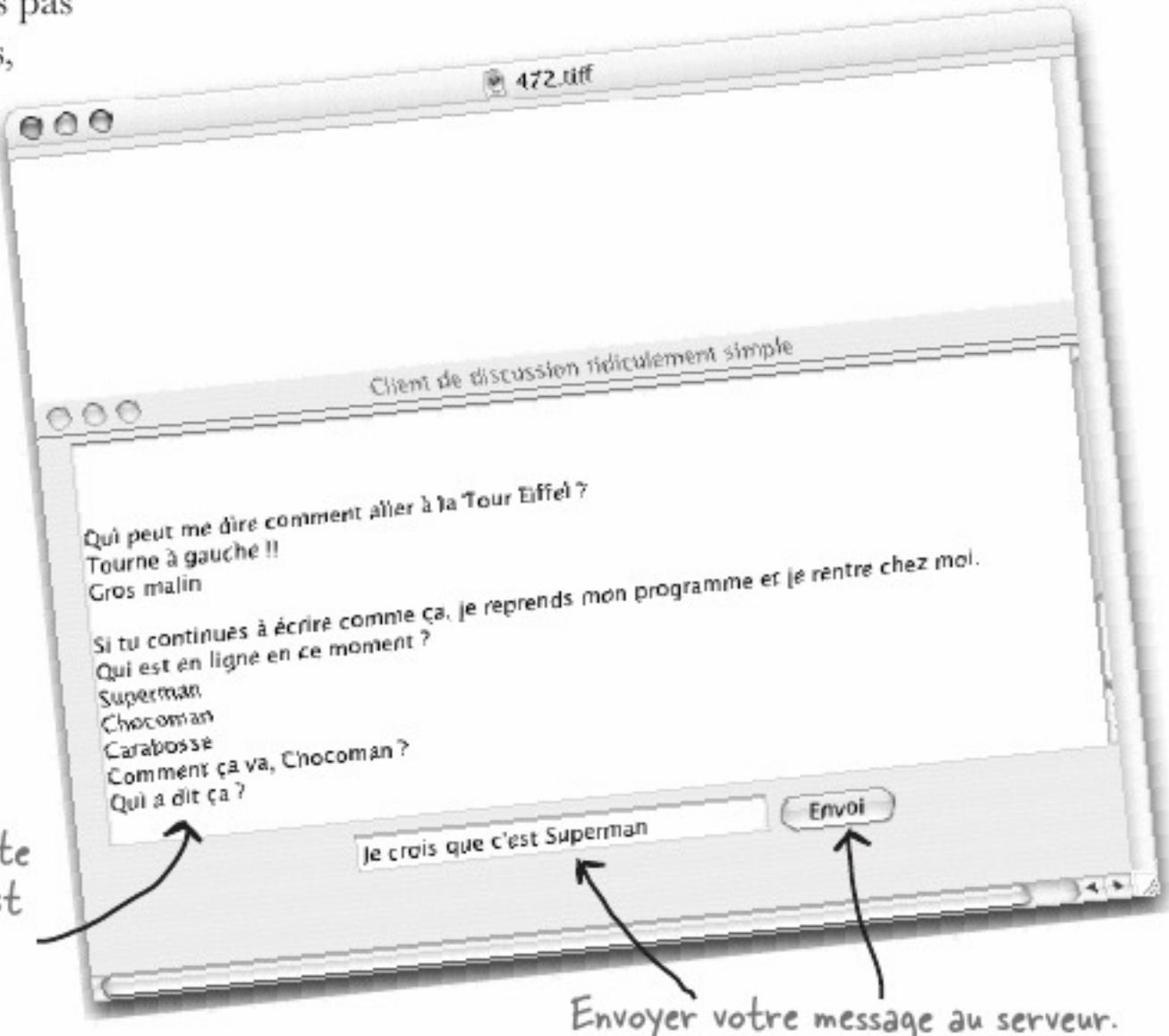
BeatBox et discussion en temps réel



Vous travaillez sur un logiciel de jeu. Vous et votre quipe tes 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 accompagn d'un motif que chaque membre recevra. Vous n'tes pas limit  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 ncessaire pour crire un tel client de discussion. Nous allons mme apprendre deux ou trois choses sur la construction d'un *serveur*. Nous allons garder la version complte pour la Recette de code, mais nous allons commencer par crire un client ridiculement simple qui envoie des messages texte et un serveur lmentaire qui en reoit.

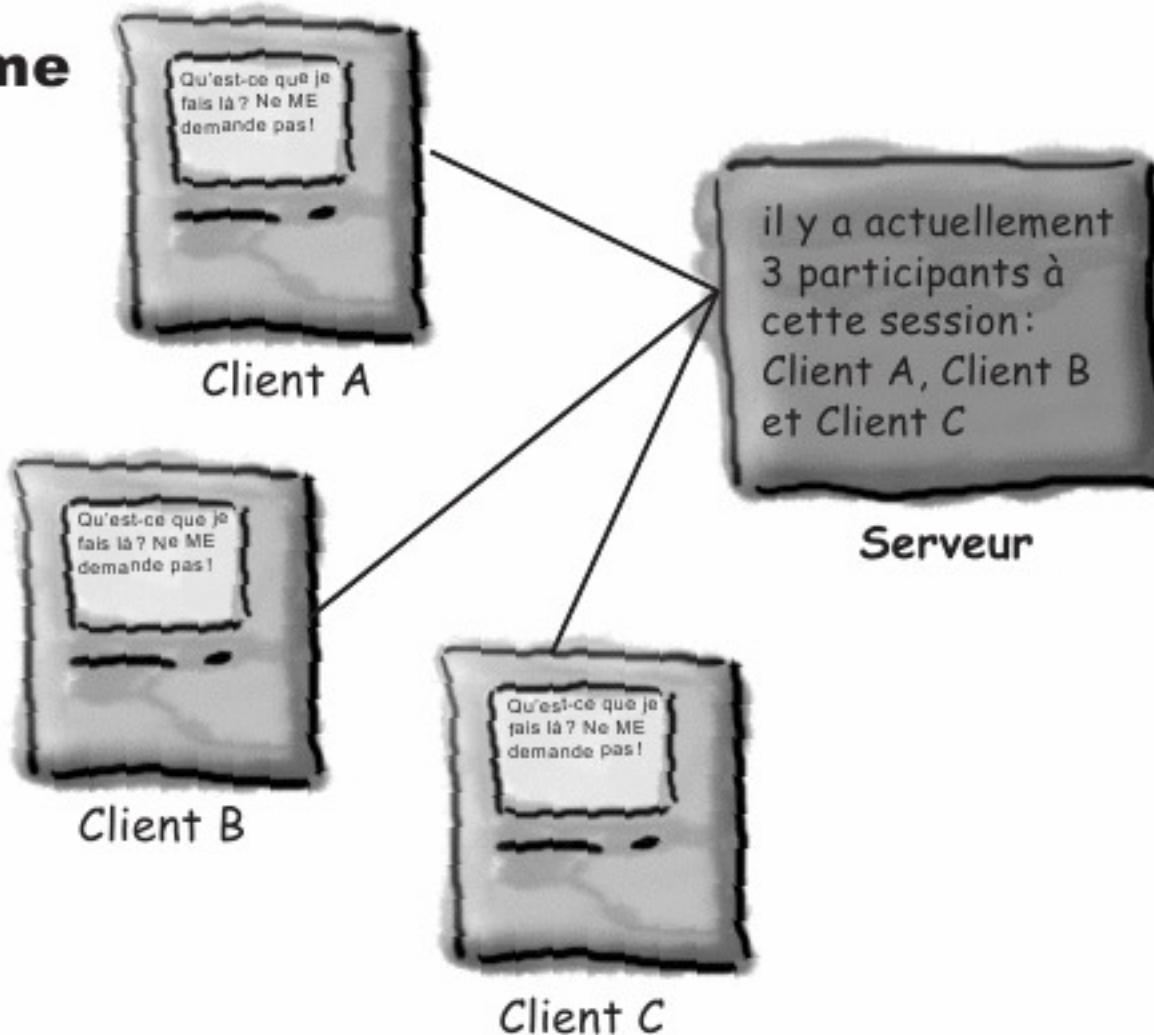
Vous pouvez avoir des conversations totalement authentiques et d'une haute tenue intellectuelle. Chaque message est envoy  tous les participants.



Vue d'ensemble du programme

Le Client doit connaître le Serveur.

Le Serveur doit connaître TOUS les Clients.

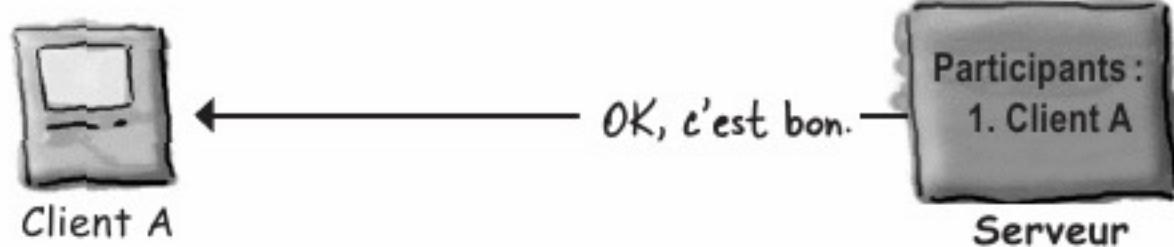


Comment ça marche :

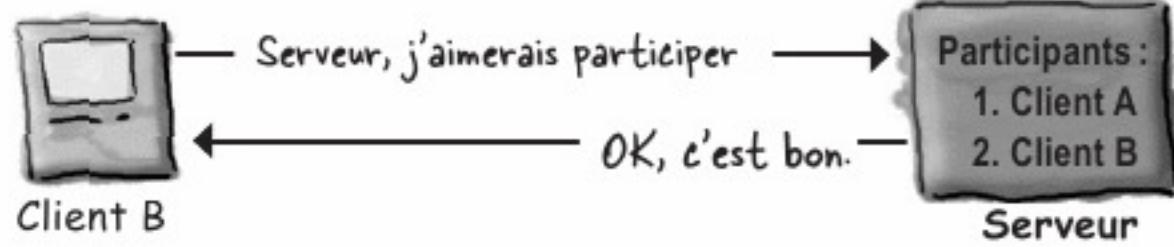
- 1 Le client se connecte au serveur



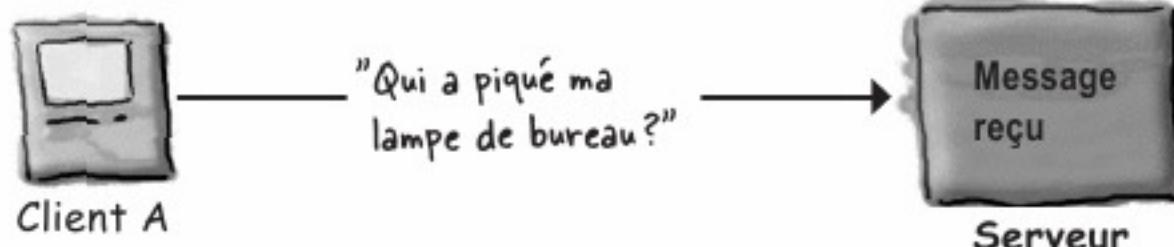
- 2 Le serveur établit une connexion et ajoute le client à la liste des participants



- 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)



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). Hmm... 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**.



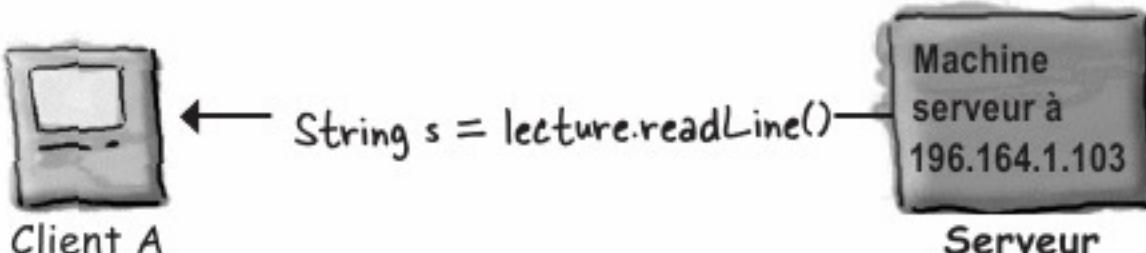
2 Émission

Le client **envoie** un message au serveur.



3 Réception

Le client **reçoit** un message du serveur.



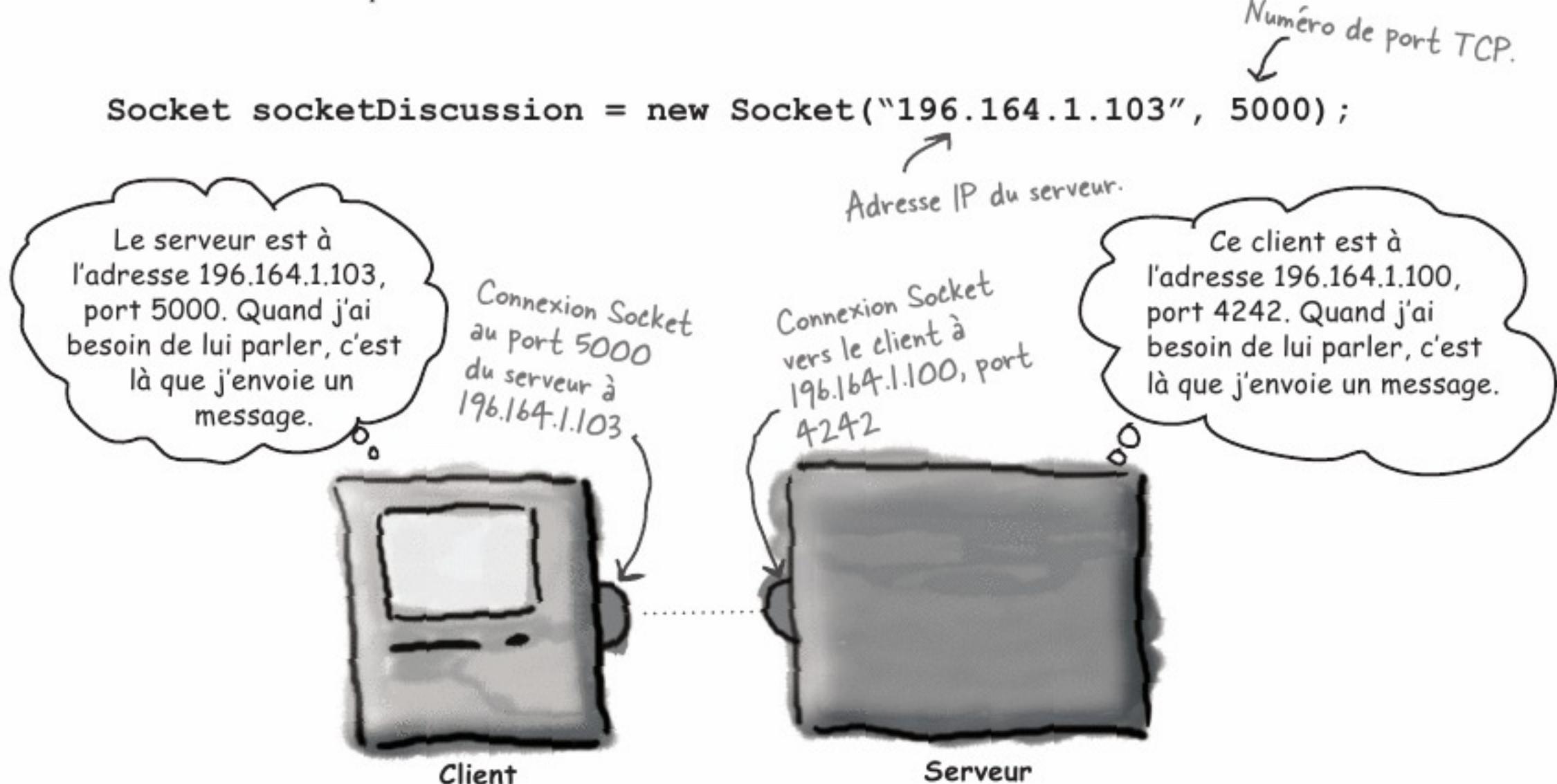
Créer une connexion réseau

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.



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.

Un port TCP n'est qu'un numéro. Un nombre de 16 bits qui identifie un programme spécifique sur le serveur.

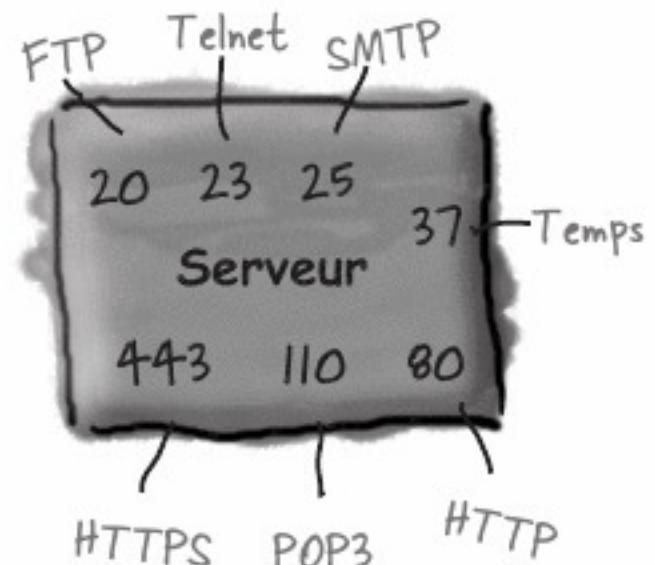
Votre serveur Web (HTTP) s'exécute sur le port 80. C'est un standard. Si vous avez un serveur Telnet, il s'exécute sur le port 23. Un serveur FTP ? 20. Un serveur de courrier POP3 ? 110. SMTP ? 25. Le serveur de temps est sur le port 37. Représentez-vous les numéros de port comme des identifiants uniques. 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.

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.

Le numéro de port est une boutique spécifique.



Spécifier l'adresse IP équivaut à spécifier un centre commercial ou une galerie marchande, par exemple "Le Forum des Halles".

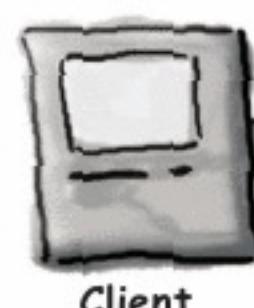
Le numéro de port est comme une boutique particulière, par exemple, "la boutique de CD de Max".



Gym du cerveau

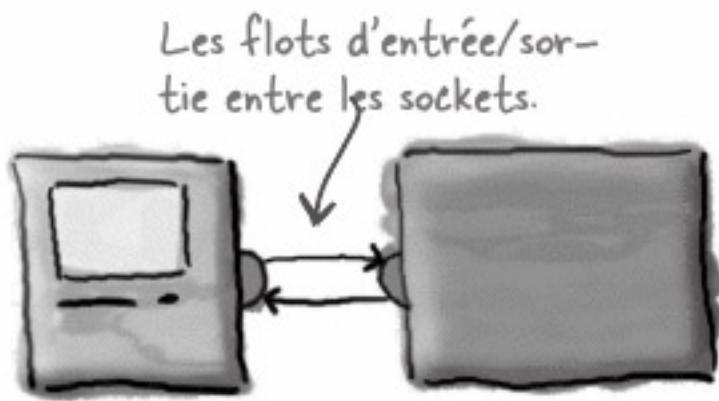
OK, vous avez une connexion Socket. Le client connaît l'adresse IP et le numéro de port TCP du serveur et inversement. Et maintenant ? Comment communiquiez-vous sur cette connexion ? Autrement dit, comment transférez-vous des bits entre eux ? Imaginez le type de messages que votre client doit émettre et recevoir.

Comment com-
muniquent-ils ?



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 connaîtsez parce que nous 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());
```

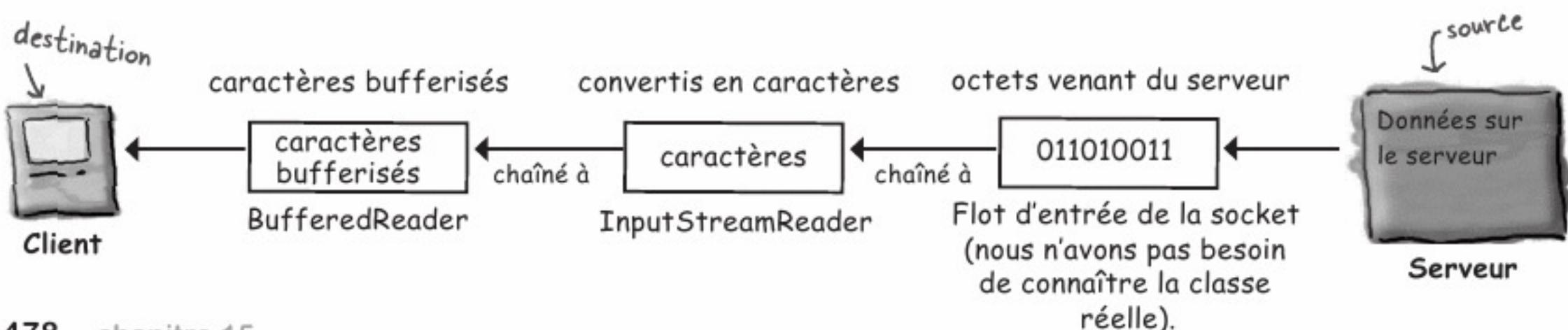
InputStreamReader est un "pont" entre un flot d'octets de bas niveau (comme celui qui vient de la Socket) et un flot de caractères de haut niveau (comme le BufferedReader qui termine le chaînage des flots).

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.

Chainer le BufferedReader à l'InputStreamReader (qui a été chaîné au flot de communication que nous avons obtenu de la Socket.)

3 Créer un BufferedReader et lire!

```
BufferedReader reader = new BufferedReader(stream);
String message = reader.readLine();
```



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.

1 Créer une connexion Socket au serveur

```
Socket socketDiscussion = new Socket("127.0.0.1", 5000);
```

Cette partie est la même que celle de la page opposée -- pour écrire au serveur, vous devez vous connecter.

2 Créer un PrintWriter chaîné au flot de sortie (connexion) de la Socket

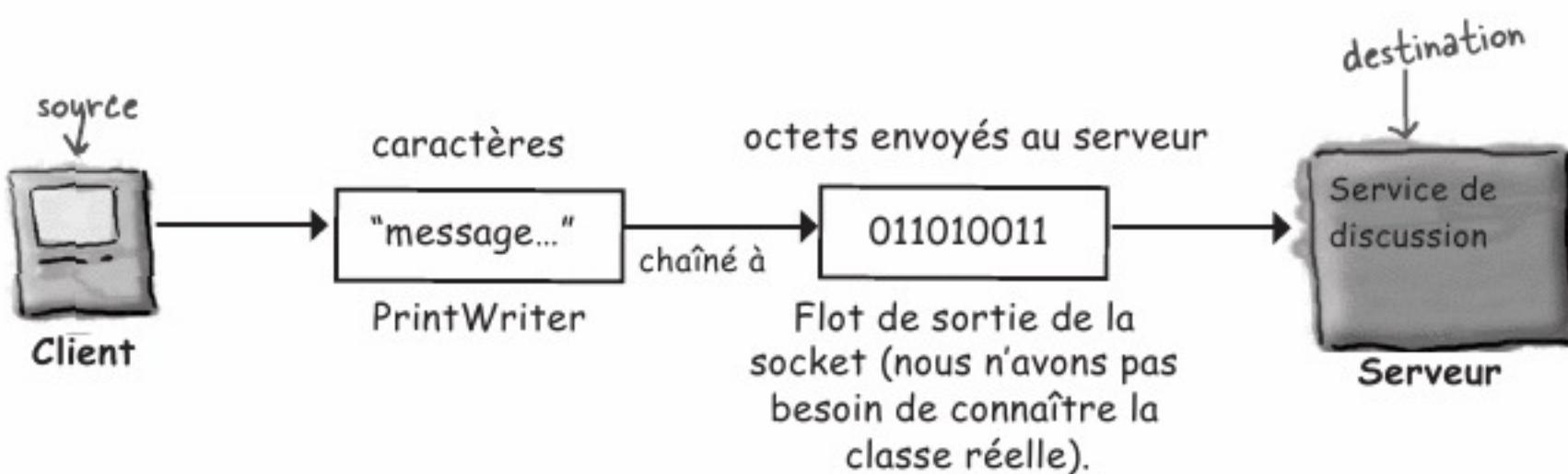
```
PrintWriter writer = new PrintWriter(socketDiscussion.getOutputStream());
```

↑
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"); ← println() ajoute un retour à la ligne à la fin de ce qu'il envoie.  
writer.print("autre message"); ← print() n'ajoute pas de retour à la ligne.
```



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.

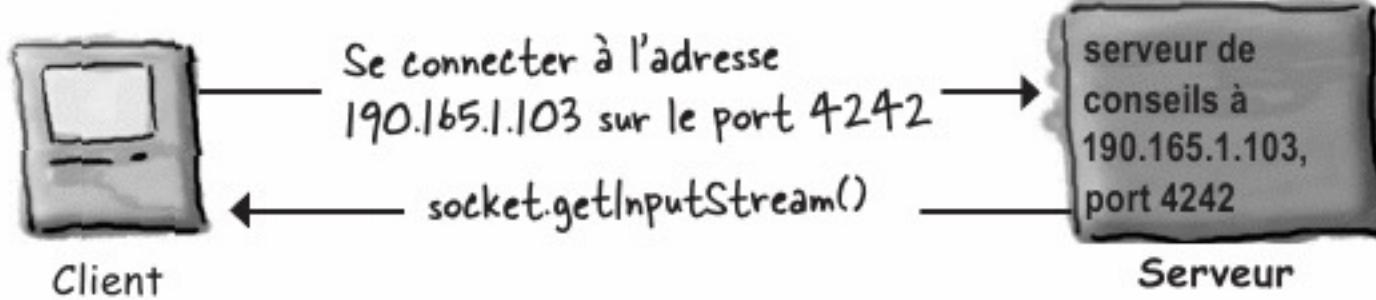
Qu'attendez-vous? Qui sait quelles opportunités vous avez manquées sans cette application.



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



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.*; ← La classe Socket est dans java.net

public class ConseilDuJourClient {
    public void go() {
        try { ← Prévenir les risques.
            Socket s = new Socket("127.0.0.1", 4242);
            Créer une connexion Socket au programme qui
            s'exécute sur le port 4242, sur l'hôte sur
            lequel ce même code s'exécute ('localhost').

            InputStreamReader isr = new InputStreamReader(s.getInputStream());
            BufferedReader lecture = new BufferedReader(isr);
            ← Chaîner un BufferedReader
            à un InputStreamReader au
            flot d'entrée de la Socket.

            String conseil = lecture.readLine(); ←
            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.

            System.out.println("Le conseil du jour: " + conseil);
            lecture.close(); ← Ferme TOUS les flots.

        } catch(IOException ex) {
            ex.printStackTrace();
        }
    }

    public static void main(String[] args) {
        ConseilDuJourClient client = new ConseilDuJourClient();
        client.go();
    }
}

```



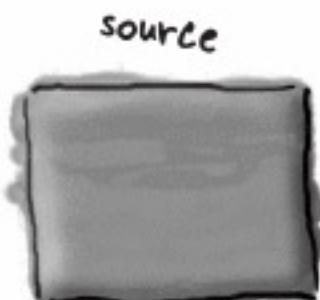
À vos crayons

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 :



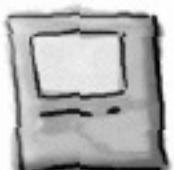
Client



Serveur

Représentez la chaîne de flots que le client utilise pour lire sur le serveur.

Pour **envoyer** du texte à une socket :



Client



Serveur

Représentez la chaîne de flots que le client utilise pour envoyer quelque chose au 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 ? _____

Écrire un serveur simple

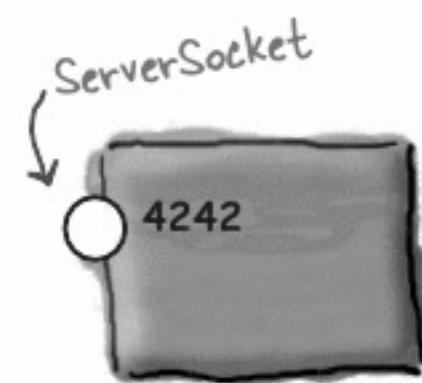
Que faut-il donc pour écrire une application serveur ? Uniquement une paire de sockets. Oui, une paire, deux sockets. Une ServerSocket qui attend les requêtes du client (quand un client crée une nouvelle Socket()) et une bonne vieille socket Socket pour communiquer avec le client.

Comment ça marche

- 1 L'application crée une ServerSocket, sur un port donné

```
ServerSocket serverSock = new ServerSocket(4242);
```

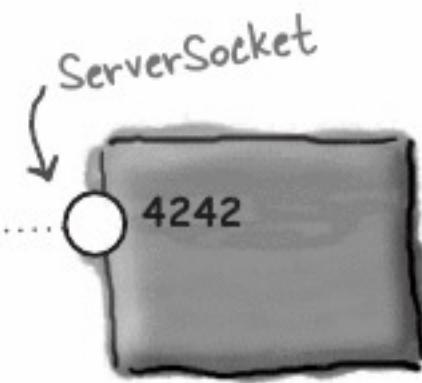
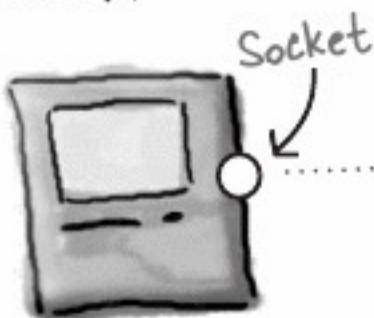
Cette ligne lance l'application qui écoute les requêtes entrantes du client sur le port 4242.



- 2 Le client crée une connexion Socket à l'application

```
Socket sock = new Socket("190.165.1.103", 4242);
```

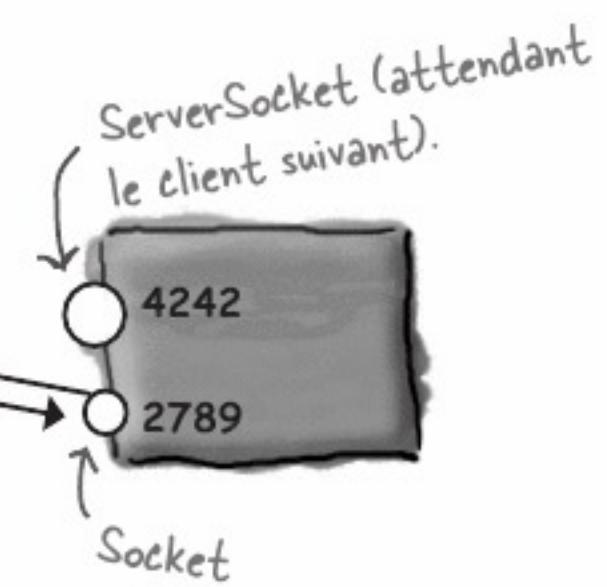
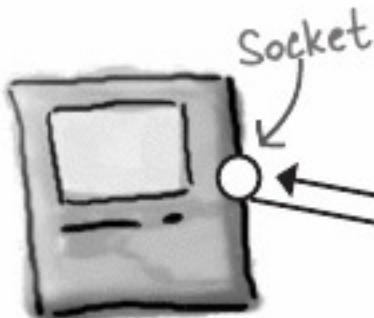
Le client connaît l'adresse IP et le numéro de port (publiés ou fournis par la personne qui a configuré l'exécution du service sur ce port).



- 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.



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 {
    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);
            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()

    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();
    }
}

```

Les conseils viennent de ce tableau.

Cette application "écoute" les requêtes des clients sur le port 4242 de la machine sur laquelle ce code s'exécute.

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.



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

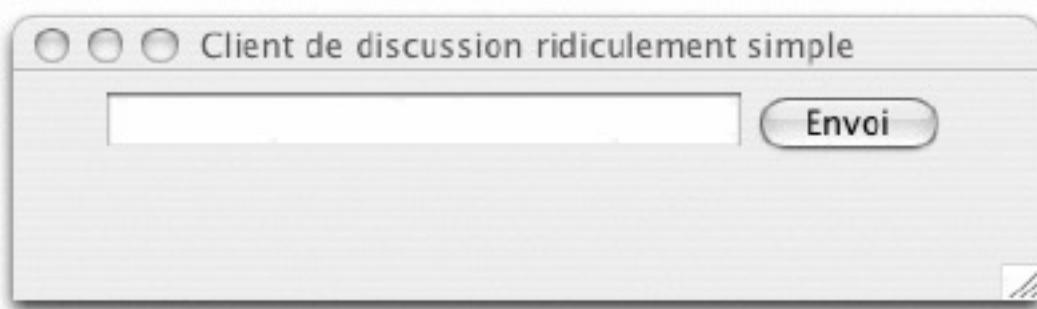
- Les applications client et serveur communiquent sur une connexion Socket.
- Une Socket représente une connexion entre deux applications qui peuvent (ou non) s'exécuter sur deux machines physiques différentes.
- Un client doit connaître l'adresse IP (ou le nom de domaine) du serveur et le numéro de 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.

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 discussion).

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
```

```

import java.io.*;
import java.net.*;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

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()

    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()

    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

```

importations pour les flots d'E/S (java.io), la Socket (java.net) et les classes de l'HTML.

Construire l'interface. Rien de nouveau et rien de lié aux E/S ni au réseau.

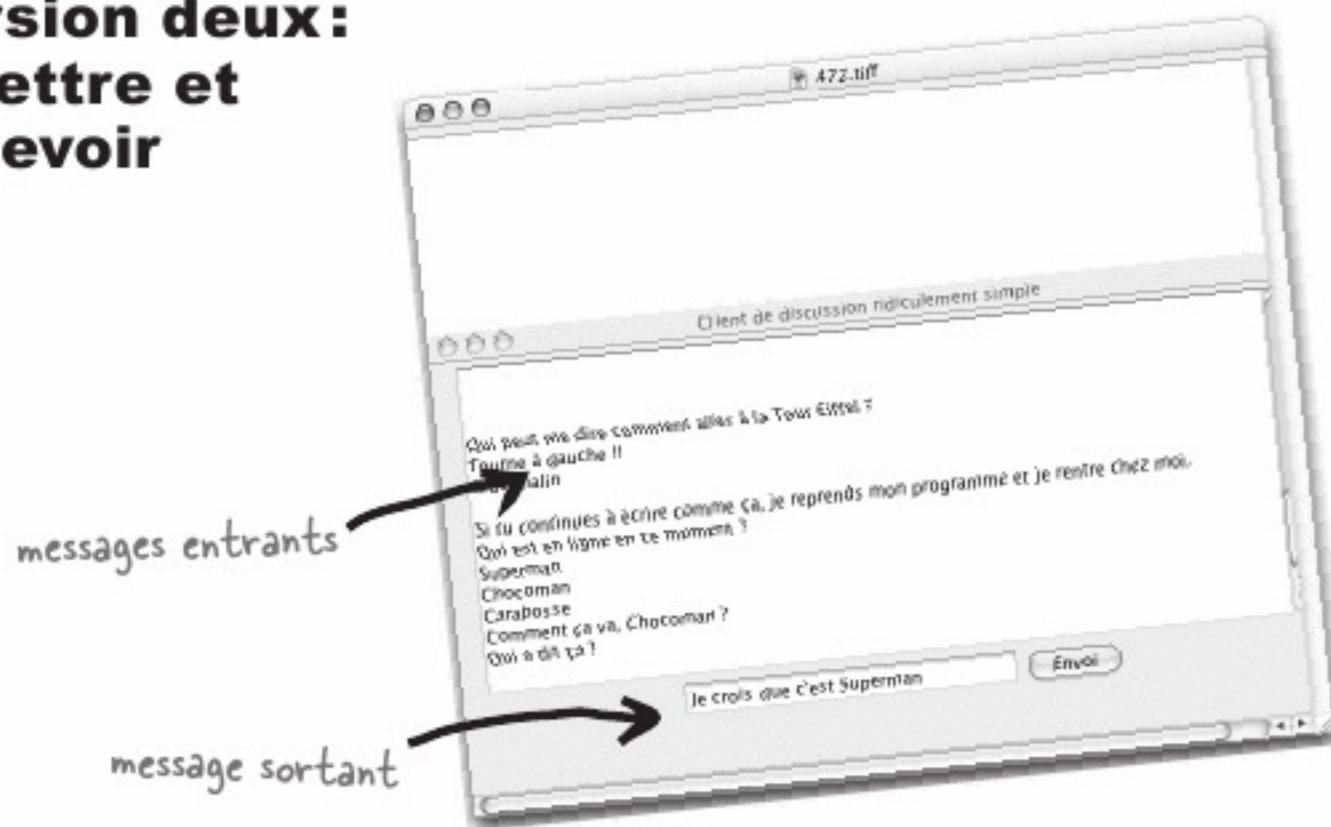
Nous utilisons localhost pour que vous puissiez tester client et serveur sur une seule machine.

C'est là que nous créons la Socket et le PrintWriter (appelé par la méthode go() juste avant l'affichage de l'HTML).

Maintenant, l'écriture réelle. Souvenez-vous que le flux d'écriture est lié au flux 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.

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 trainard 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.



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.

En Java, vous POUVEZ vraiment marcher et mâcher du chewing-gum en même temps.

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.)

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.

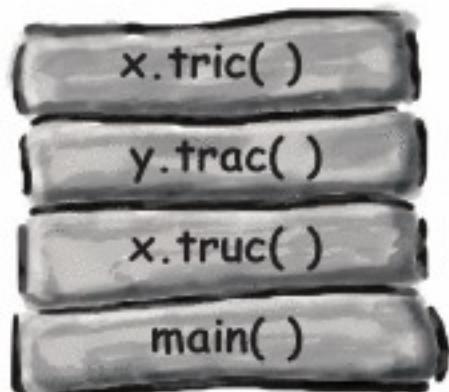
Un **thread** est un «fil d'exécution» séparé.

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

Thread

Thread
<code>void join()</code>
<code>void start()</code>

`static void sleep()`

La classe `java.lang.Thread`

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 (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).

Que signifie avoir plusieurs piles d'appels ?

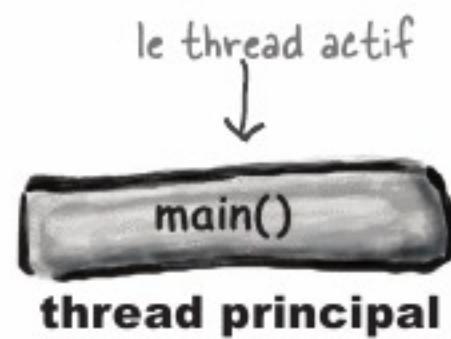
Avec plusieurs piles d'appels, on a l'*impression* que plusieurs choses se passent en même temps. En réalité, seul un vrai système multiprocesseur peut faire plusieurs choses en même temps, mais les threads Java permettent de faire semblant. Autrement dit, l'exécution passe d'une pile à l'autre si rapidement 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 *diffrerente*.

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().

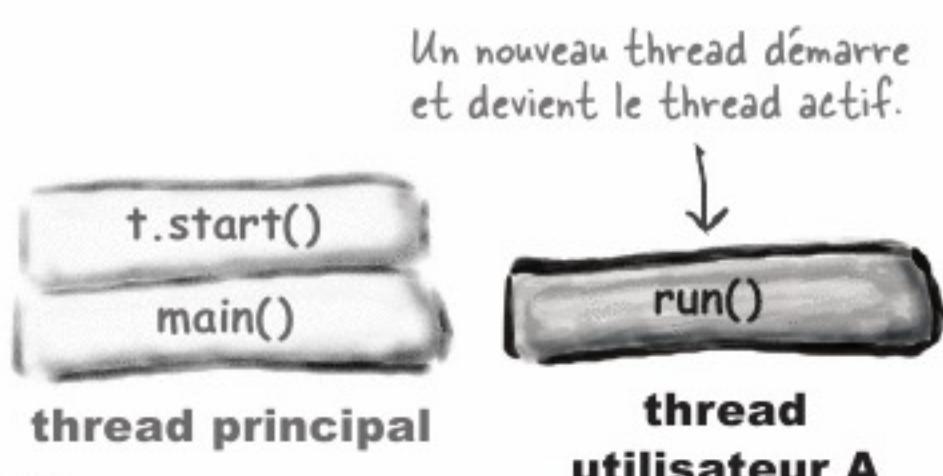
```
public static void main(String[] args) {  
    ...  
}
```



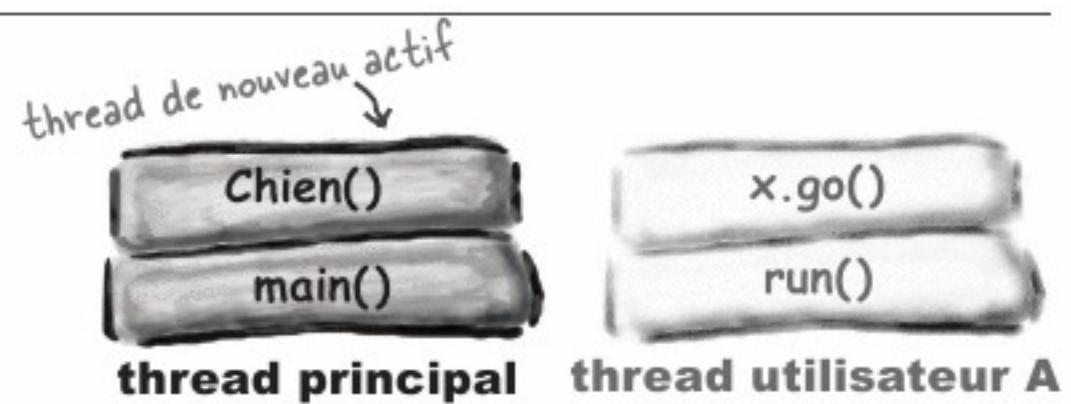
2 main() lance un nouveau thread. Le thread principal est temporairement gelé pendant que l'autre commence à s'exécuter.

```
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.



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);
```

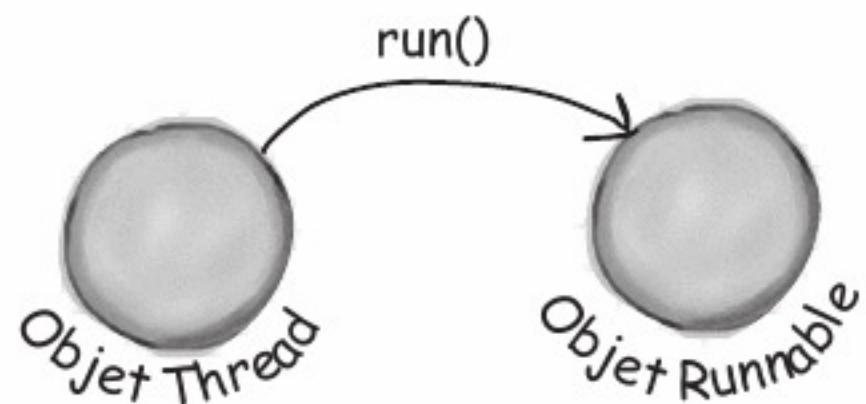
On passe l'objet Runnable au constructeur du Thread. Le nouvel objet Thread sait ainsi quelle méthode placer en bas de la nouvelle pile d'appels — la méthode run() de Runnable.



3 Lancer le thread

```
monThread.start();
```

Rien ne se passe tant que vous nappelez 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.



Tout thread a un travail à faire. une méthode à placer sur la nouvelle pile.



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
}
```

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.

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().

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.)

Pour donner un travail à votre thread, implémentez l'interface Runnable

```
public class MonRunnable implements Runnable {
    public void run() {
        go();
    }

    public void go() {
        fairePlus();
    }

    public void fairePlus() {
        System.out.println("sommet de la pile");
    }
}
```

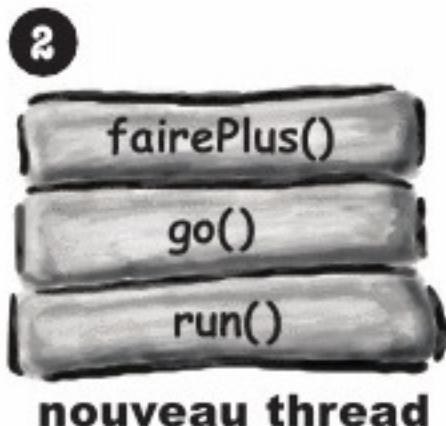
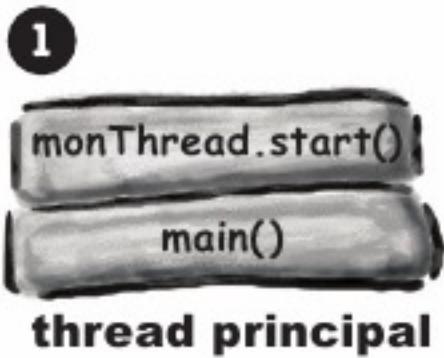
Runnable étant dans le package java.lang, vous n'avez pas besoin de l'importer.

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);
        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'appeliez 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)

Les trois états d'un nouveau thread

```
Thread t = new Thread(r);
```

NOUVEAU



`t.start();`

EXÉCUTABLE



Sélectionné

L'état dont rêvent les threads!

S'EXÉCUTANT



"Puis-je faire quelque chose pour vous?"

```
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();`

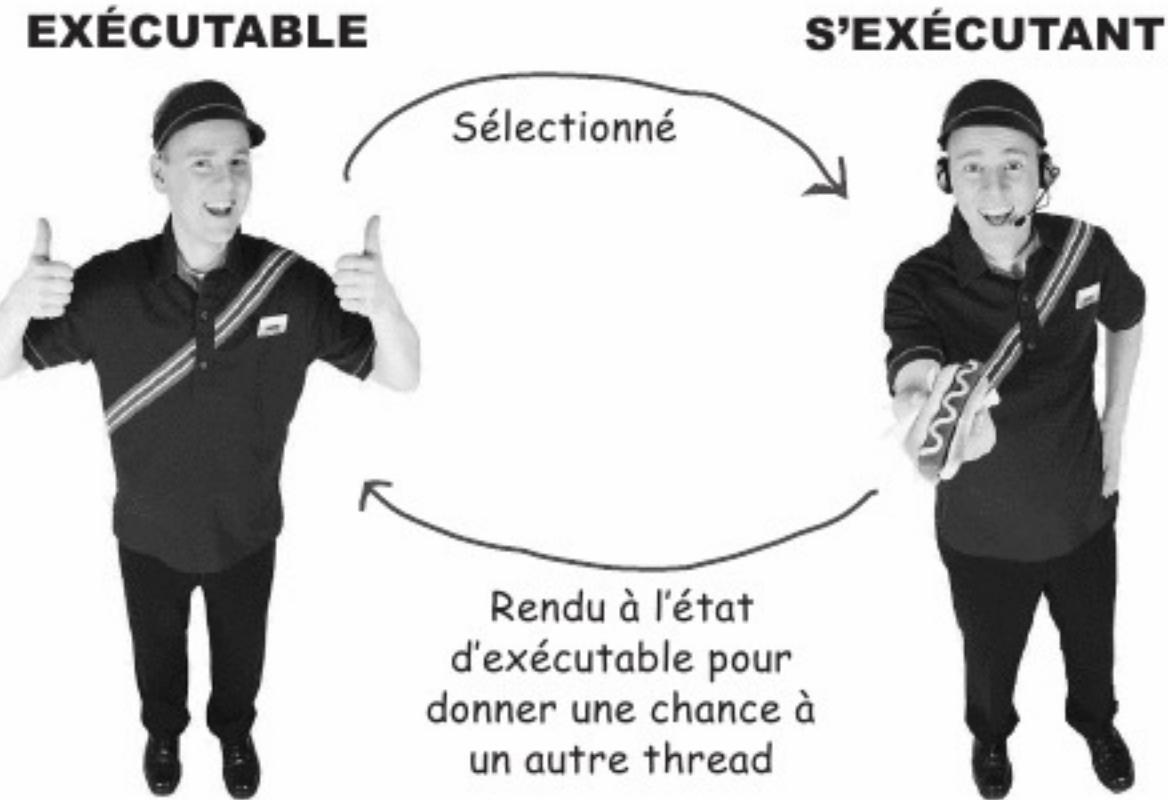
Quand vous lancez le thread, il passe à l'état exécutable (runnable). Cela signifie que le thread est prêt et attend la chance de sa vie : être sélectionné pour l'exécution. À ce stade, ce thread a une nouvelle pile d'appels.

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 permute entre les deux états, être exécutable, s'exécuter... et avoir un état supplémentaire: temporairement non exécutable (alias « bloqué »).

Une boucle type entre deux états

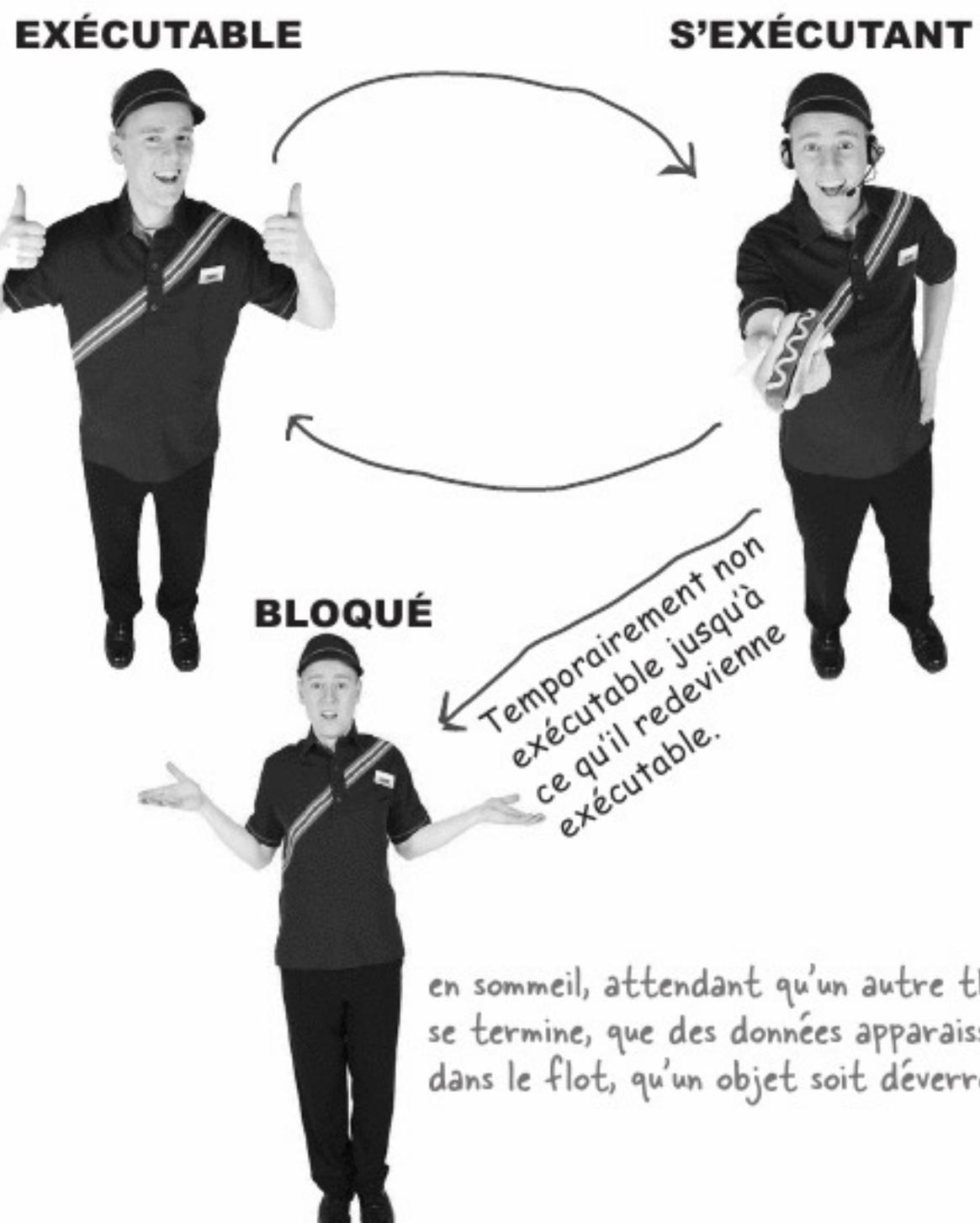
En général, un thread oscille entre les deux états (il est exécutable ou il s'exécute), tandis que la JVM les sélectionne à tour de rôle, en en sélectionnant un pour l'exécution puis le stoppant pour qu'un autre thread ait sa chance.



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.



L'ordonnanceur de threads

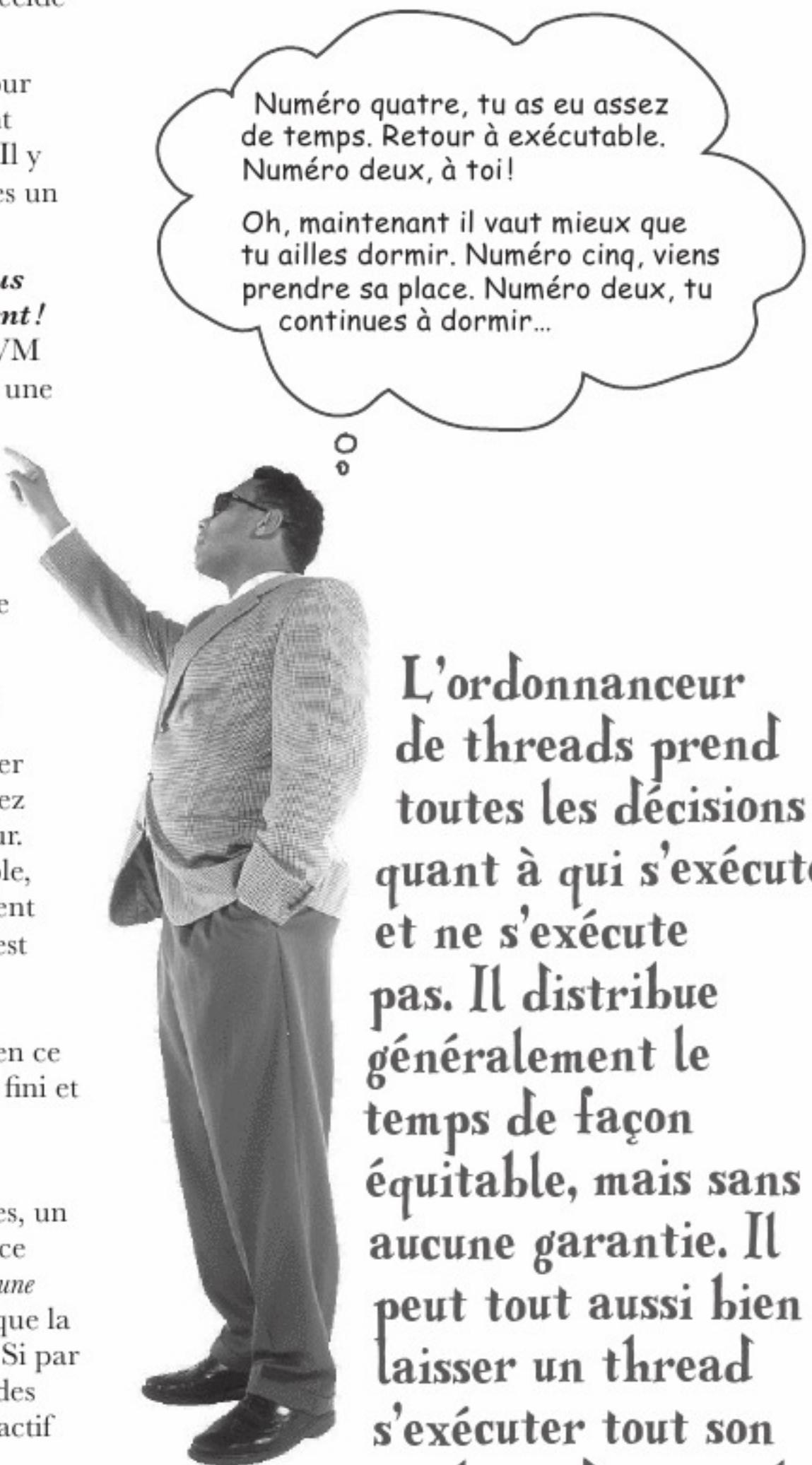
L'ordonnanceur prend toutes les décisions sur l'état des threads : lesquels sont exécutables, lequel s'exécute et quand (et dans quelles circonstances) un thread cesse de s'exécuter. Il détermine le thread qui doit être actif, pendant combien de temps il doit l'être, et que faire des threads quand il 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».

Un exemple de l'imprévisibilité de l'ordonnanceur...

L'exécution de ce code sur une machine...

... a produit ce résultat

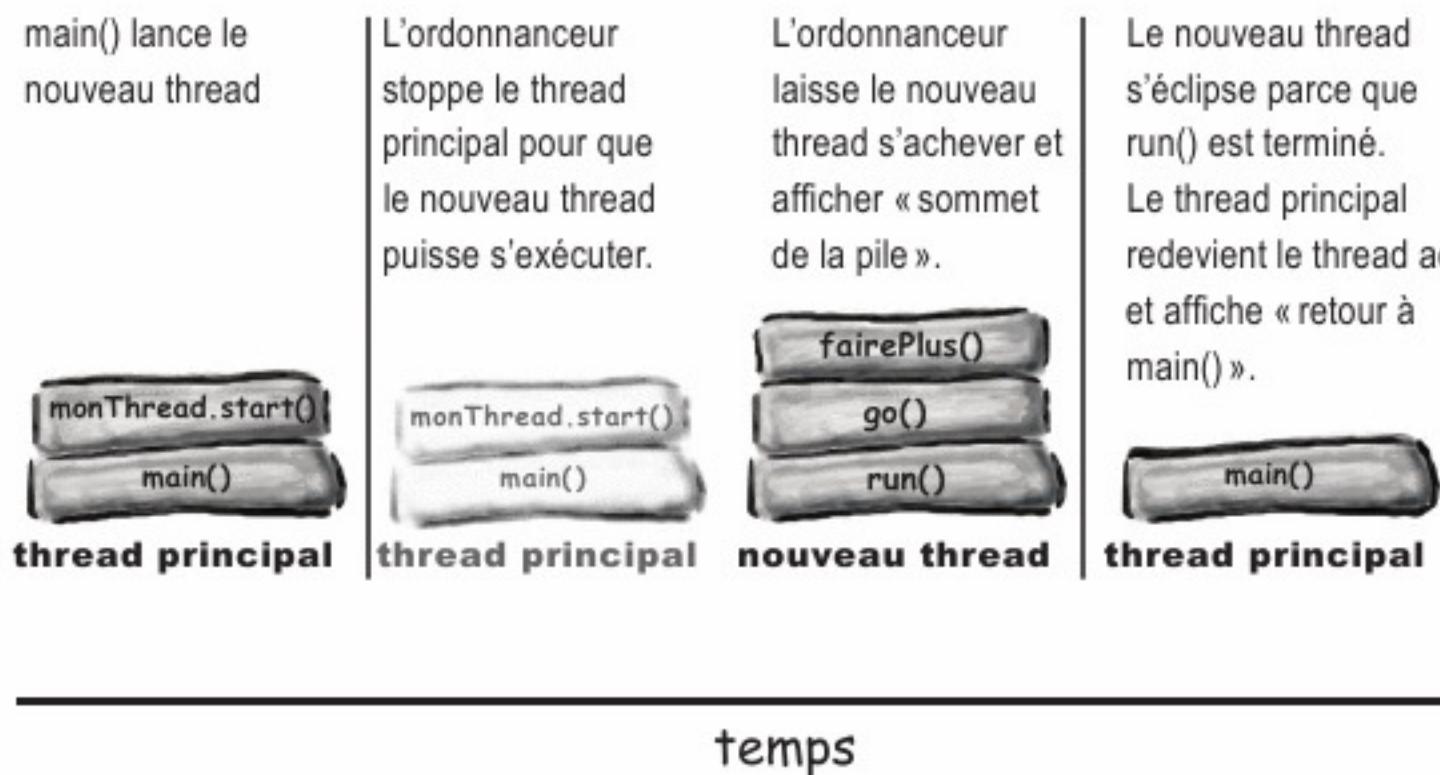
```
public class MonRunnable implements Runnable {  
  
    public void run() {  
        go();  
    }  
  
    public void go() {  
        fairePlus();  
    }  
  
    public void fairePlus() {  
        System.out.println("sommet de la pile");  
    }  
}  
  
class TestThread {  
  
    public static void main (String[] args) {  
  
        Runnable threadTache = new MonRunnable();  
        Thread monThread = new Thread(threadTache);  
  
        monThread.start();  
  
        System.out.println("retour à main() ()");  
    }  
}
```

Remarquez comment l'ordre change de façon aléatoire. C'est parfois le nouveau thread qui termine le premier et parfois le thread principal.

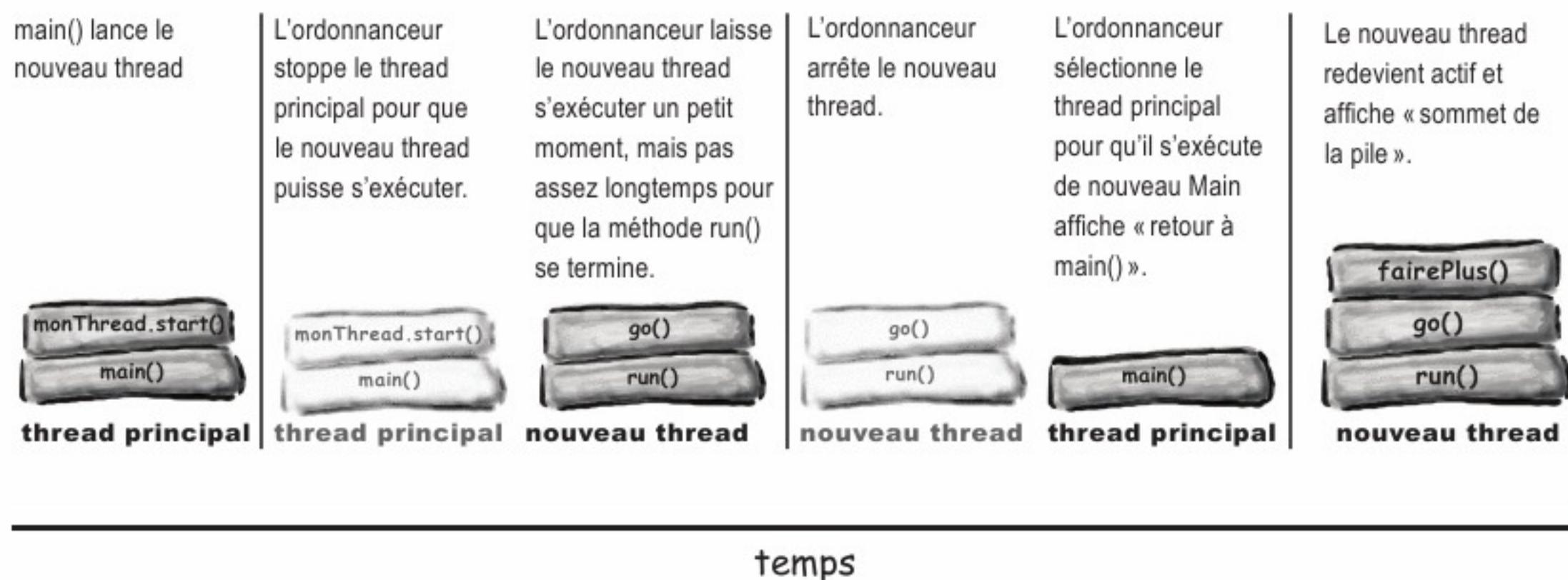
```
Fichier Edition Fenêtre Aide ChoisissezMoi  
  
% java TestThread  
retour à main()  
sommet de la pile  
% java TestThread  
sommet de la pile  
retour à main()  
% java TestThread  
sommet de la pile  
retour à main()  
% java TestThread  
sommet de la pile  
retour à main()  
% java TestThread  
sommet de la pile  
retour à main()  
% java TestThread  
sommet de la pile  
retour à main()  
% java TestThread  
sommet de la pile  
retour à main()  
% java TestThread  
sommet de la pile  
retour à main()
```

Pourquoi les résultats sont-ils différents?

Parfois, le code s'exécute comme ceci:



Et parfois il s'exécute comme cela:



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 exécutabile), 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.

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 parlez 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.

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?

```
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()");
    }
}
```

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
```

Appeler sleep() ici forceira 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'.

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.

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.

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 alpha s'exécute

```



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.

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.

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 !**

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.



Sylvie et Bruno : victimes du "problème du compte joint".



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.

Le problème de Sylvie et Bruno, codé

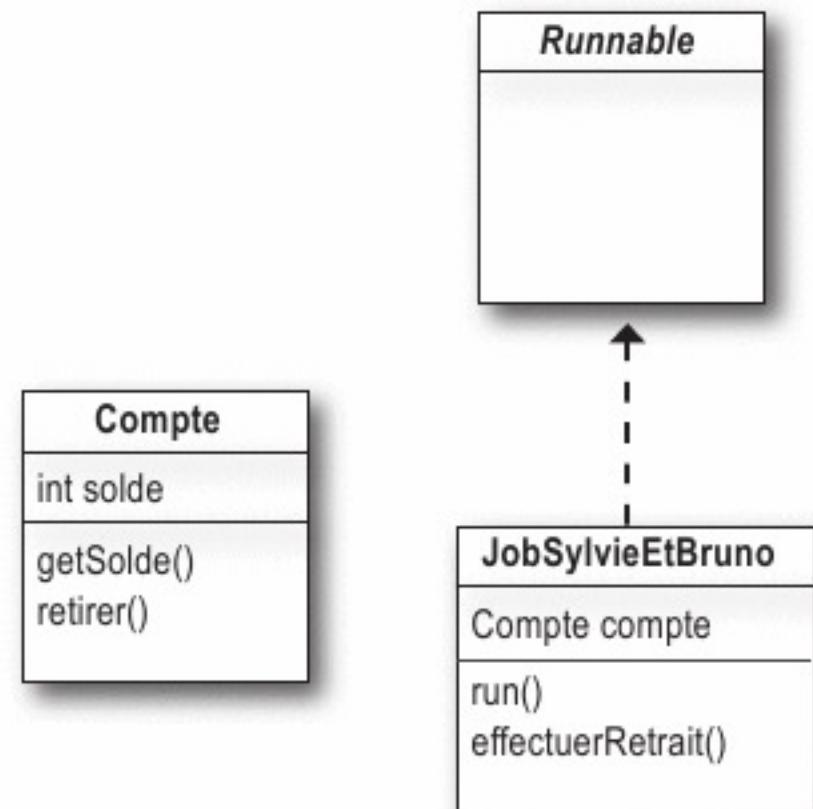
L'exemple suivant montre ce qui peut se passer quand deux threads (Sylvie et Bruno) partagent un même objet (le compte en banque).

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();
```

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.

② On crée deux threads avec le même Runnable (l'instance de JobSylvieEtBruno)

```
Thread un = new Thread(laTache);
Thread deux = new Thread(laTache);
```

Cette procédure devrait éviter que le compte soit à découvert.

③ On nomme les threads et on les lance

```
un.setName("Bruno");
deux.setName("Sylvie");
un.start();
deux.start();
```

Sauf que... Sylvie et Bruno s'endorment toujours après avoir vérifié le solde mais avant d'avoir terminé le retrait.

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();}
}
```

L'exemple de Sylvie et Bruno

```

class Compte {
    private int solde = 100; ← Le compte commence avec
                                un solde de 100€.

    public int getSolde() {
        return solde;
    }

    public void retirer(int montant) {
        solde = solde - montant;
    }
}

public class JobSylvieEtBruno implements Runnable { ← Il n'y a qu'UNE instance de JobSylvieEtBruno.
    private Compte compte = new Compte(); ← Autrement dit UNE SEULE instance du compte.
                                                Les deux threads vont accéder à ce seul compte.

    public static void main (String [] args) {
        JobSylvieEtBruno laTache = new JobSylvieEtBruno(); ← Instanciation du Runnable (la tâche).
        Thread un = new Thread(laTache); ←
        Thread deux = new Thread(laTache); ← On crée deux threads en donnant à chacun la même tâche.
        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());
        }
    }
}

```

Nous avons inséré quelques instructions print pour pouvoir observer le déroulement de l'exécution.

Sylvie et Bruno: le résultat

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
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. Pataugas ! 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.

Il faut verrouiller l'accès au compte!

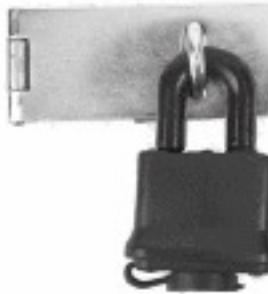
Voici comment le verrou fonctionne :

- 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.

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 :

```
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());
    }
}
```



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.

(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.)

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.

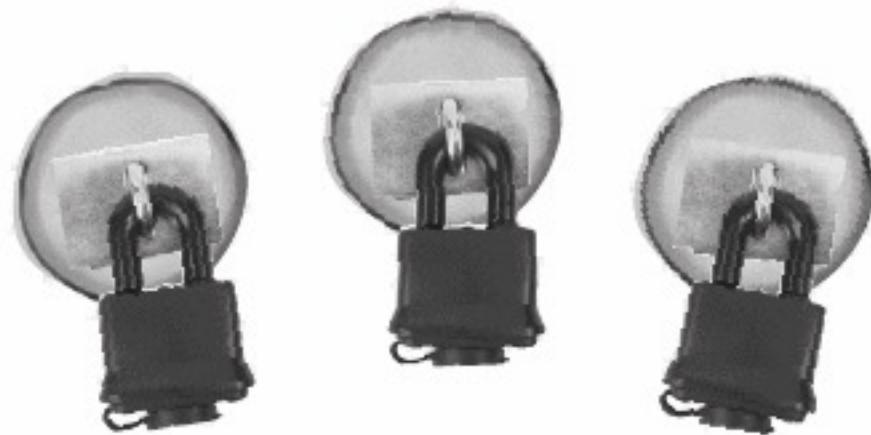
Le but de la synchronisation est de protéger toutes les données critiques. Mais n'oubliez pas : vous ne protégez pas les données elles-mêmes, vous synchronisez les méthodes qui accèdent à ces données.

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.



Hé, cette méthode prendreLArgent() est synchronisée. Il me faut la clé de cet objet pour pouvoir entrer...



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.

la synchronisation est importante

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++) { ← Chaque thread s'exécute 50 fois, incrémentant le solde à chaque itération.  
            incrementer();  
            System.out.println("le solde est de " + solde);  
        }  
    }  
  
    public void incrementer() {  
        int i = solde; ← 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.  
        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();  
        }  
    }  
}
```

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.

Aie!!

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.

Rendez la méthode incrementer() 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.

La méthode faireQqch() n'a pas besoin d'être synchronisée et nous nous abstenons.

```
public void go() {
    faireQqch();
    synchronized(this) {
        codeCritique();
        autreCodeCritique();
    }
}
```

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.

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...]

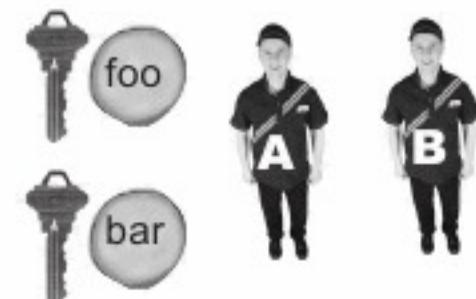
L'aspect mortel de la synchronisation

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...

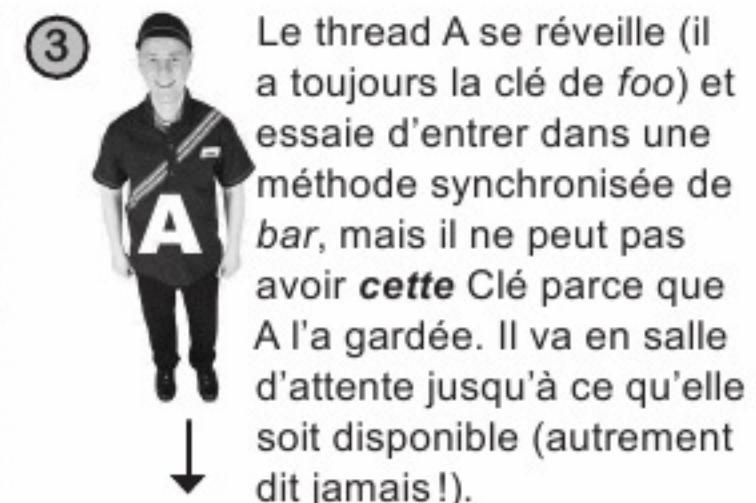
Si vous connaissez bien les serveurs de bases de données ou d'autres applications, vous avez peut-être reconnu le problème : les bases de données disposent souvent d'un mécanisme de verrouillage qui ressemble quelque peu à la synchronisation. Mais un vrai système de gestion de transactions peut parfois 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 :



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...



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.

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.

```

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

```

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).

} // fin de la classe externe



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.

```
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
```

Pour exécuter le client, il vous faut deux fenêtres d'exécution. Lancez ce serveur dans la première, puis lancez le client dans la seconde.

```
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
```

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.

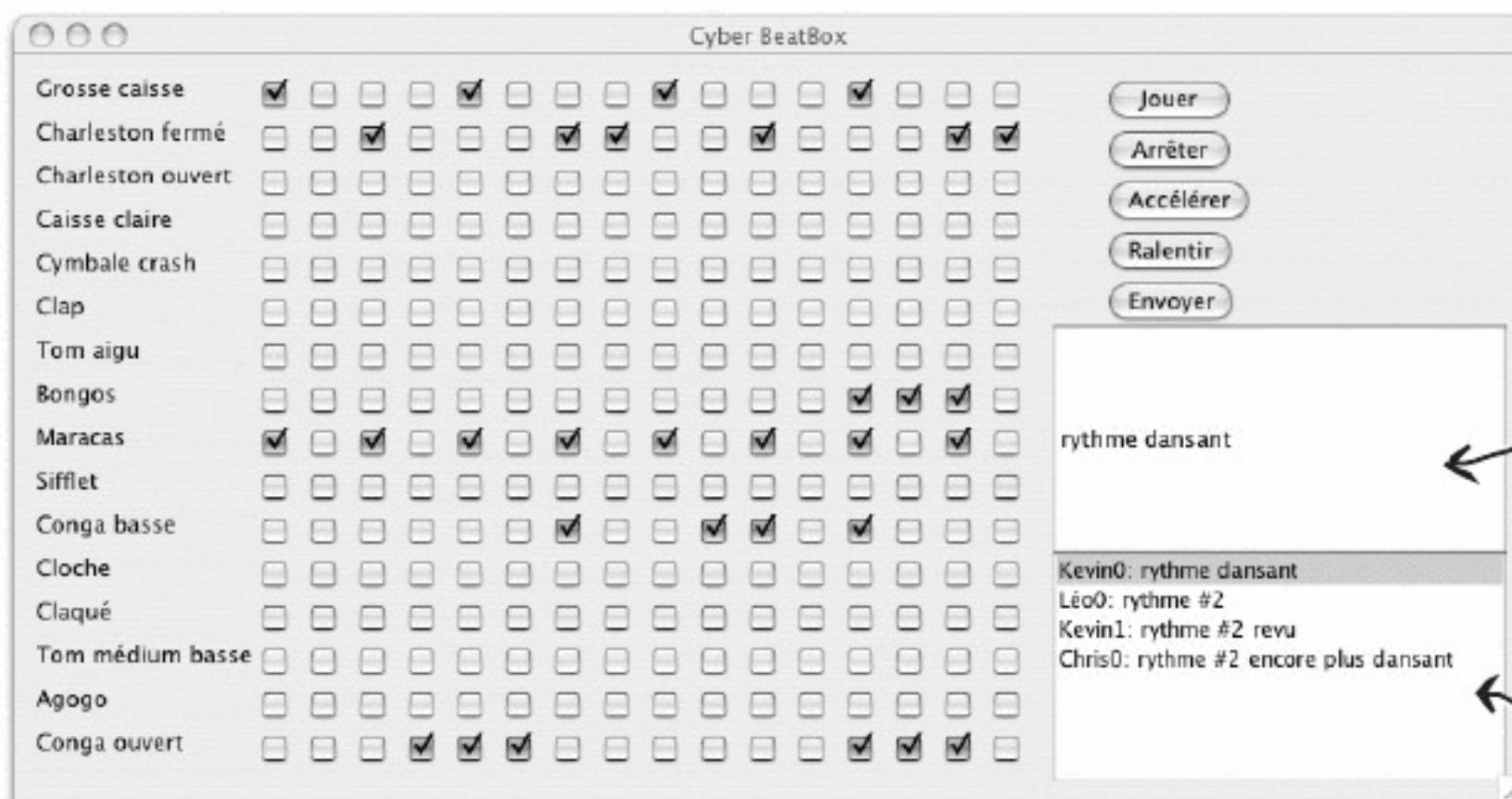
Q : Pourquoi ne pas simplement synchroniser les méthodes get et set de la classe qui contient les données que vous voulez protéger ? Par exemple, pourquoi ne pas juste synchroniser la méthode getSolde() et retirer() de la classe Compte au lieu de synchroniser la méthode effectuerRetrait() de la classe 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 méthodes 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.

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 "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.

exercice : le frigo



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
```

Question bonus : Pourquoi pensez-vous que nous ayons utilisé les modificateurs de cette façon dans la classe Accum ?

Fichier Edition Fenêtre Aide Filer

```
% java TestThreads
un 98098
deux 98099
```

Le frigo (suite)

```
Thread un = new Thread(t1);
```

```
} catch(InterruptedException ex) { }
```

```
Thread deux = new Thread(t2);
```

```
Accum a = Accum.getAccum();
```

```
public static Accum getAccum() {
```

```
private int compteur = 0;
```

```
a.majCompteur(1);
```

```
for(int x=0; x < 99; x++) {
```

```
public int getCompte() {
```

```
public void majCompteur(int add) {
```

```
for(int x=0; x < 98; x++) { deux.start();
```

```
try {
```

```
public void run() {
```

```
private Accum() { }
```

```
Accum a = Accum.getAccum();
```

```
System.out.println("deux "+ a.getCompte());
```

```
ThreadDeux t2 = new ThreadDeux();
```

```
try {
```

```
return compteur;
```

```
compteur += add;
```

```
implements Runnable {
```

```
un.start();
```

```
Thread.sleep(50);
```

```
} catch(InterruptedException ex) { }
```

```
private static Accum a = new Accum();
```

```
public void run() {
```

```
Thread.sleep(50);
```

```
implements Runnable {
```

```
a.majCompteur(1000);
```

```
return a;
```

```
System.out.println("un "+ a.getCompte());
```

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

```
ThreadUn t1 = new ThreadUn();
```

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(); ← Creation d'une instance de Accum.  
    private int compteur = 0;  
  
    private Accum() {} ← Constructeur privé.  
  
    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());  
    }  
}
```



Solutions des exercices

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());  
    }  
}
```



Sas d'embrouilles

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 clé du mystère



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 hochla 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?

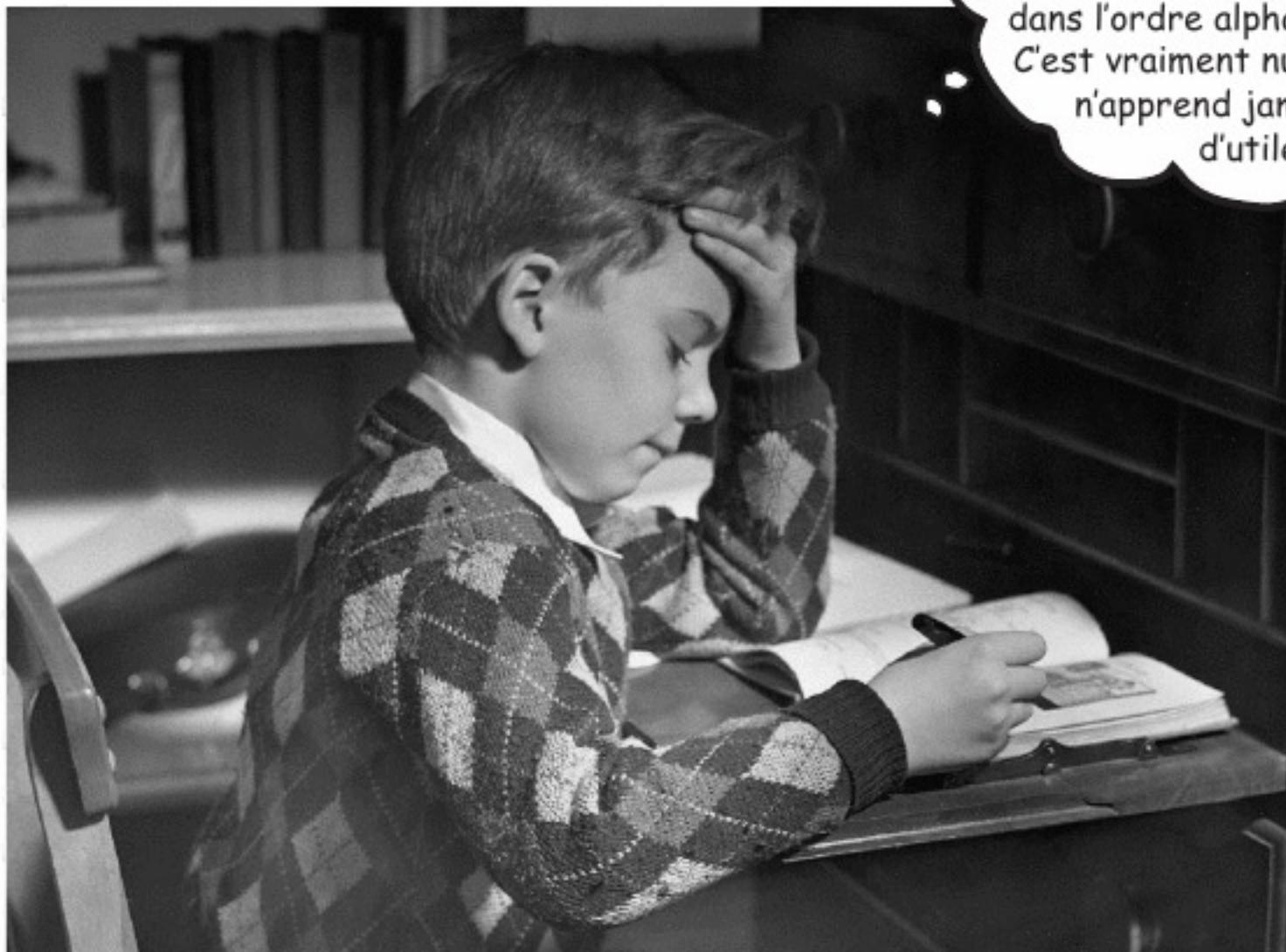


Que savait Sarah?

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.

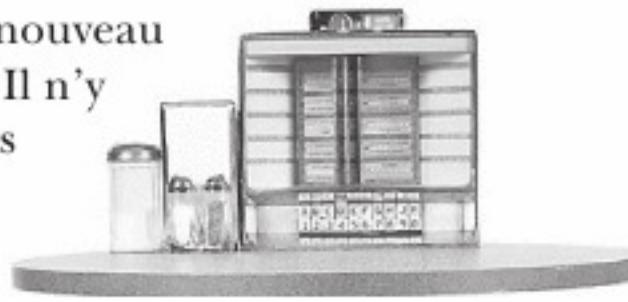
Structures de données



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...

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` ?

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 Danse
%java Jukebox1
[Pink Moon, Somersault,
Shiva Moon, Circles,
Deep Channel, Passenger,
Listen]

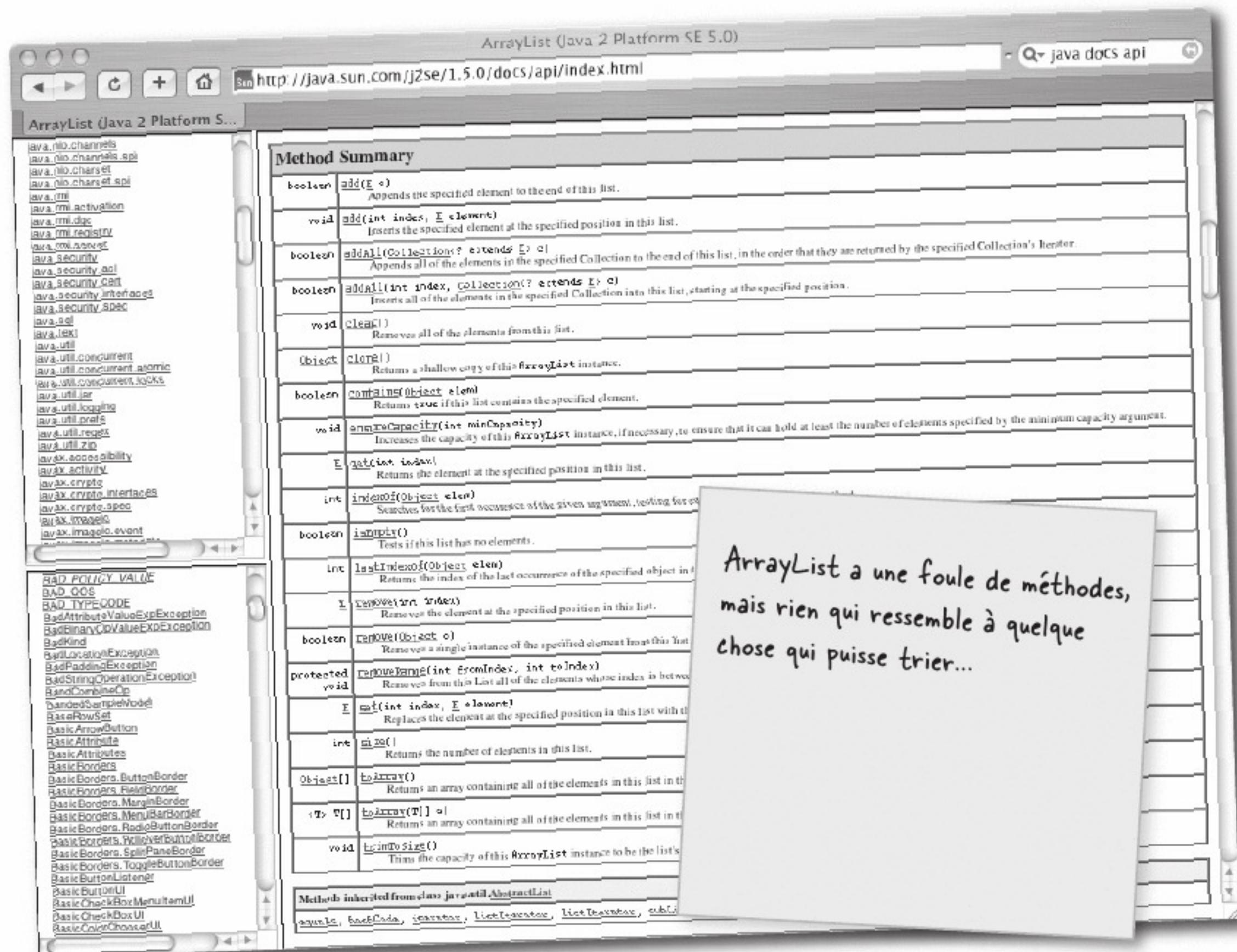
```

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!

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*.





Je vois bien une classe collection nommée TreeSet... et la doc dit que les données peuvent être triées. Je me demande si je ne pourrais pas utiliser TreeSet au lieu d'ArrayList...

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.

You 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

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.

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 boolean swap(List list, int placeAll, Object ancVal, Object nouvVal)
// beaucoup de choses supplémentaires...
```

Mmmmm... 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**.

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).

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]);
    }
}

```

```

%java Jukebox1

[Pink Moon, Somersault, Shiva Moon, Circles, Deep
Channel, Passenger, Listen]

[Circles, Deep Channel, Listen, Passenger, Pink
Moon, Shiva Moon, Somersault]

```

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!

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.

```
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)` appellera la méthode `toString()` de CHAQUE élément de la liste.

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.

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>`.

```

import java.util.*;
import java.io.*;

public class Jukebox3 {
    ArrayList<Morceau> listeMorceaux = new ArrayList<Morceau>(); //Créer une ArrayList d'objets de type
    public static void main(String[] args) { //Morceau et non de type String.
        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 une ArrayList d'objets de type Morceau et non de type String.

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.

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 cela 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 est 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().



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`.

Génériques = fiabilité du typage

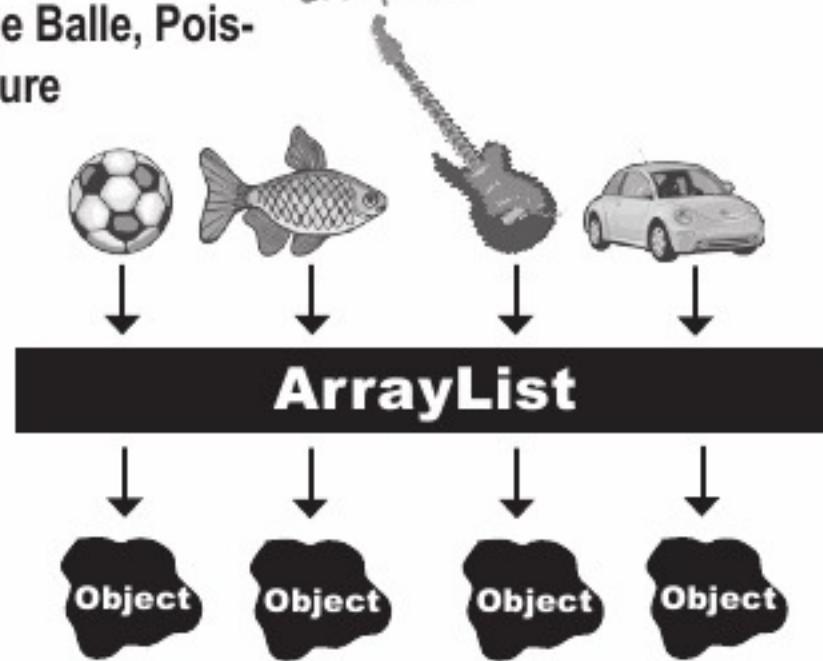
Nous ne le répéterons pas — *virtuellement tout le code que vous écrivez et qui utilise des génériques sera du code lié aux collections.* Même s'il existe d'autres façons de les utiliser, le principal intérêt des génériques est qu'ils vous permettent d'écrire des collections typées de façon fiable. Autrement 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>.

SANS génériques

Les objets ENTRENT sous forme de référence de type Balle, Poisson, Guitare et Voiture

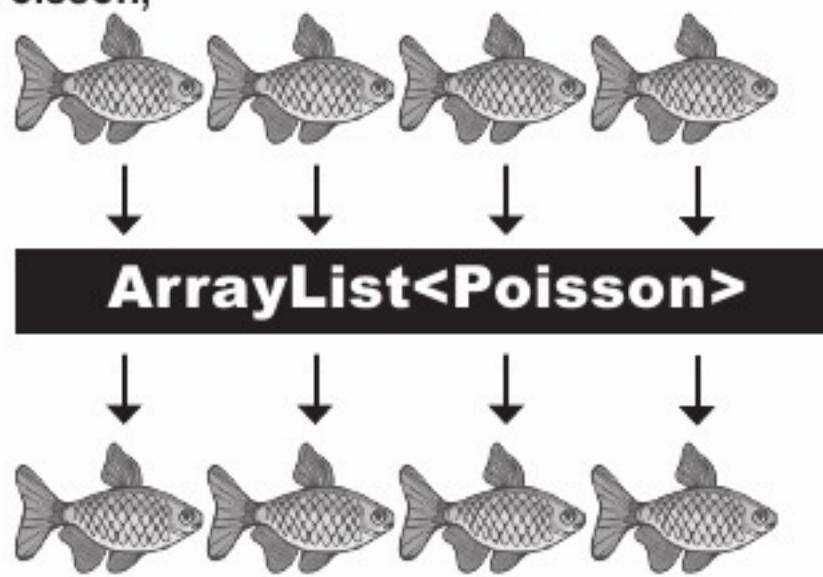
Avant les génériques, il n'y avait pas moyen de déclarer le type d'une ArrayList, et sa méthode add() acceptait donc un type Object.



Et SORVENT sous forme de référence de type Object

Avec les génériques

Les objets ENTRENT sous forme de référence de type Poisson,



Et SORVENT 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 mette une Volkswagen ou que ce qui en ressort ne soit pas vraiment convertible en référence de Poisson.

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.

```
void truc(List<Morceau>  
list)  
  
x.truc(listeMorceaux)
```

(Ce point est en fait identique au point 2, mais cela vous montre l'importance que nous lui accordons.)

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.)

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.

Comprendre la documentation d'ArrayList (Ou : quelle est la vraie signification du « E »?)

```
public class ArrayList<E> extends AbstractList<E> implements List<E> ... {  
  
    public boolean add(E o)  
}  
  
Voici la partie importante! Quel que soit  
« E », il détermine le type d'objets que vous  
êtes autorisé à ajouter dans l'ArrayList.  
  
// reste du code
```

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.

Le type (la valeur de <E>) devient également celui de l'interface List.

Représentez-vous le « E » comme remplaçant le « type d'élément que vous voulez que cette collection contienne et retourne ». (E signifie Element.)

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 ».

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 ».

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)
```

Si la classe elle-même n'utilise pas de paramètre de type, vous pouvez toujours en spécifier un pour une méthode, en la 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 ».

Ici, on peut utiliser <T> parce qu'on a introduit <T> avant dans la déclaration de la méthode.



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() ...



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);
    }
}
```

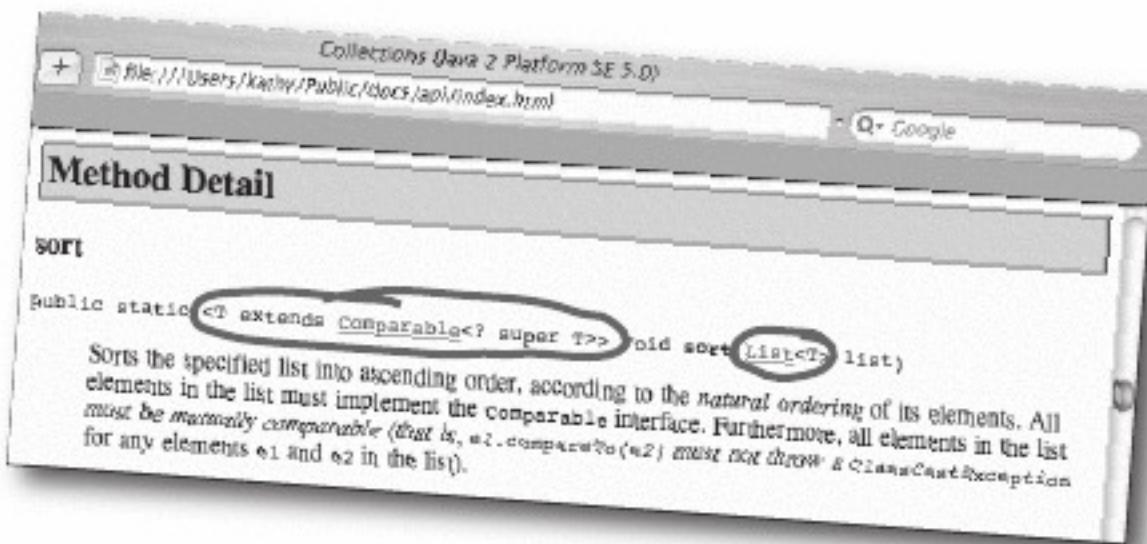
Voilà où le bâton blesse ! Tout allait bien avec une ArrayList<String>, mais dès que nous essayons de trier une ArrayList<Morceau>, rien ne va plus.

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 paramètre qui « étend Comparable ».

Hum... Je viens de lire la doc de String, et String n'ÉTEND pas Comparable: elle l'IMPLÉMENTE. Comparable est une interface. C'est donc absurde d'écrire <T extends Comparable>.



```
public final class String extends Object implements Serializable,
    Comparable<String>, CharSequence
```

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.

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 écrivez toujours «`extends`».

Avec les génériques, le mot-clé «`extends`» signifie en réalité «est-une» et fonctionne pour les classes ET les interfaces.

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 `InputStream`). 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.)

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 !

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.

```
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;
    }
}
```

Cela veut dire que les objets Morceau peuvent être comparés à d'autres objets Morceau pour les besoins du tri.

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]
```

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.

Ce n'est pas suffisant.
Parfois, je veux trier par artiste et non par titre.



Collections (Java 2 Platform SE 5.0)

file:///Users/kathy/Public/docs/api/index.html

Jellyvision, Inc. Collections ...form SE 5.0) Caffeinated ...d Brain Day Brand Noise Diva Marketing

<code><K, V> static Map<K, V></code>	<u>singletonMap(K key, V value)</u> Returns an immutable map, mapping only the specified key to the specified value.
<code>super T>> static void</code>	<u>sort(List<T> list)</u> Sorts the specified list into ascending order, according to the <i>natural ordering</i> of its elements.
<code><T> static void</code>	<u>sort(List<T> list, Comparator<? super T> c)</u> 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...

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.

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.

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 : 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.

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 compare()* 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());
        }
    } Ceci devient une chaîne (l'artiste)
}

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.)

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()*.

exercice sur les collections

```
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 {
    _____;
    _____;
    _____ {
        _____;
        _____;
    }
    _____;
    _____ {
        _____;
        _____;
    }
}
}
```



À vos crayons

Rétro-ingénierie

Supposez que ce code existe dans 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.

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]
```



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 ArrayList 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? _____

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 TrapDeNotes
%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] ← Avant le tri.

[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] ← Après le tri
utilisant la méthode
compareTo() de
Morceau (tri par
titre).

[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] ← 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.

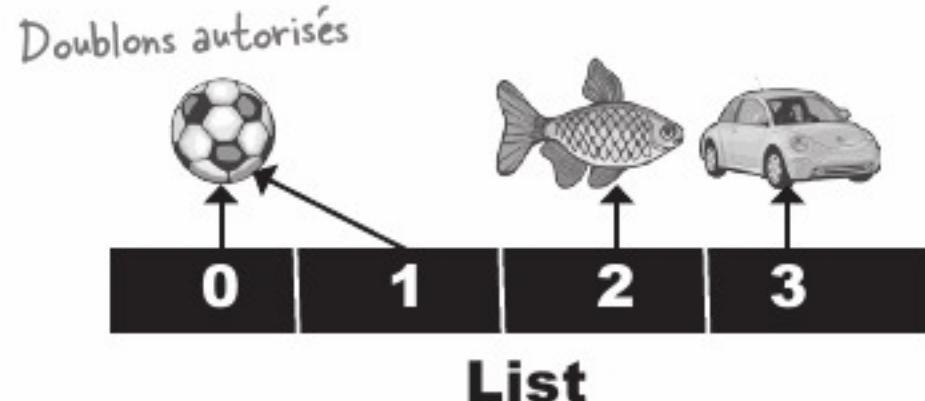
Il nous faut un Set au lieu d'une List

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*.

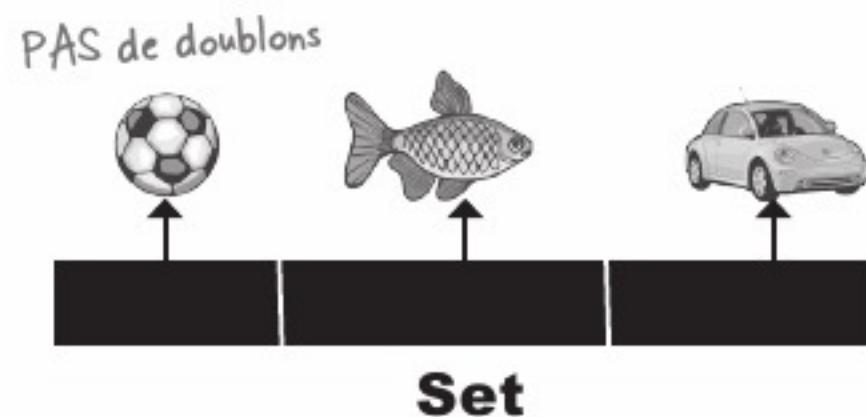
Une List sait où se trouve quelque chose dans la liste. Vous pouvez avoir plusieurs éléments référençant le même objet.



► 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).

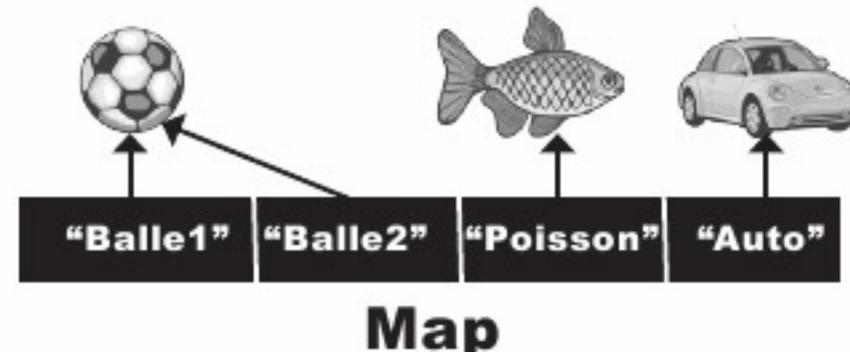


► MAP, pour trouver quelque chose par clé

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.

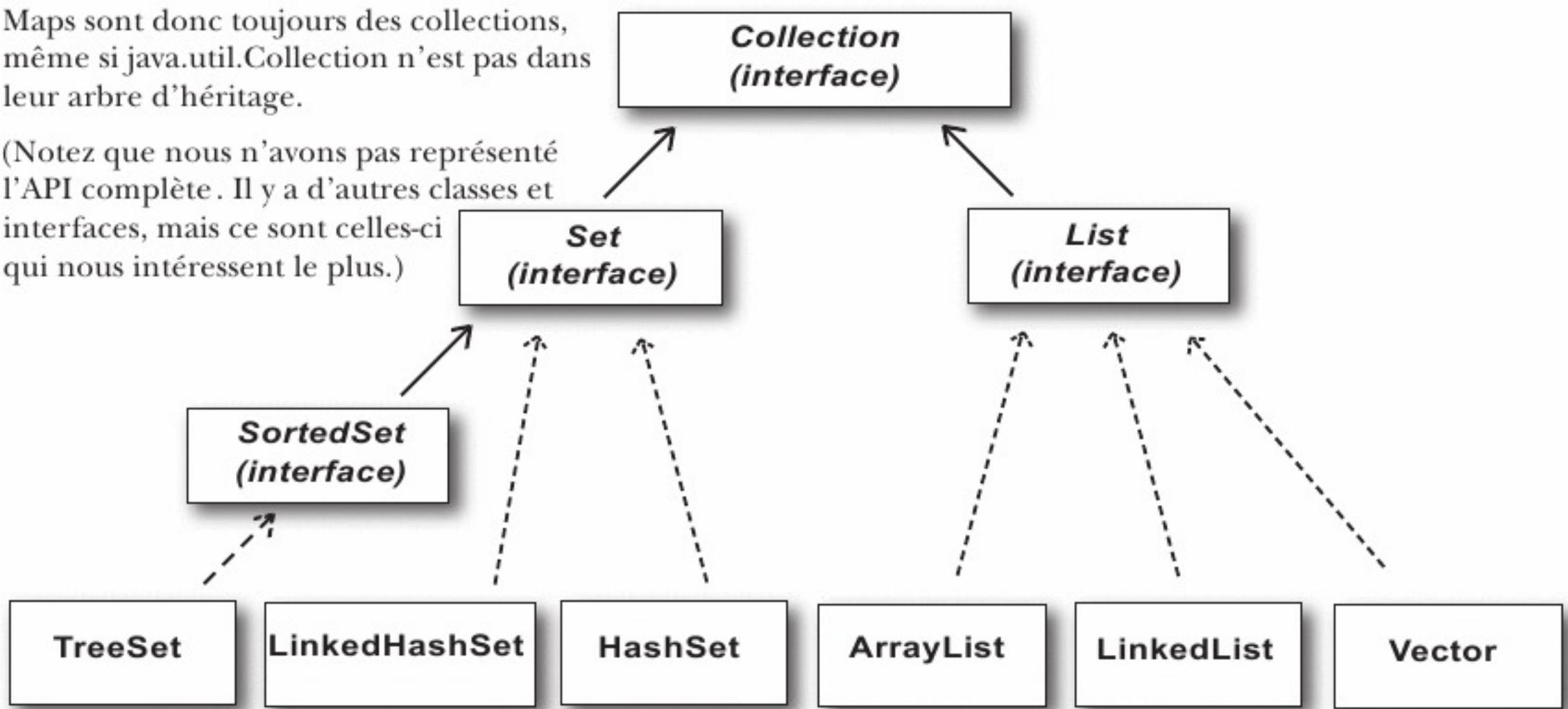
Valeurs dupliquées autorisées, mais PAS de clés dupliquées.



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.

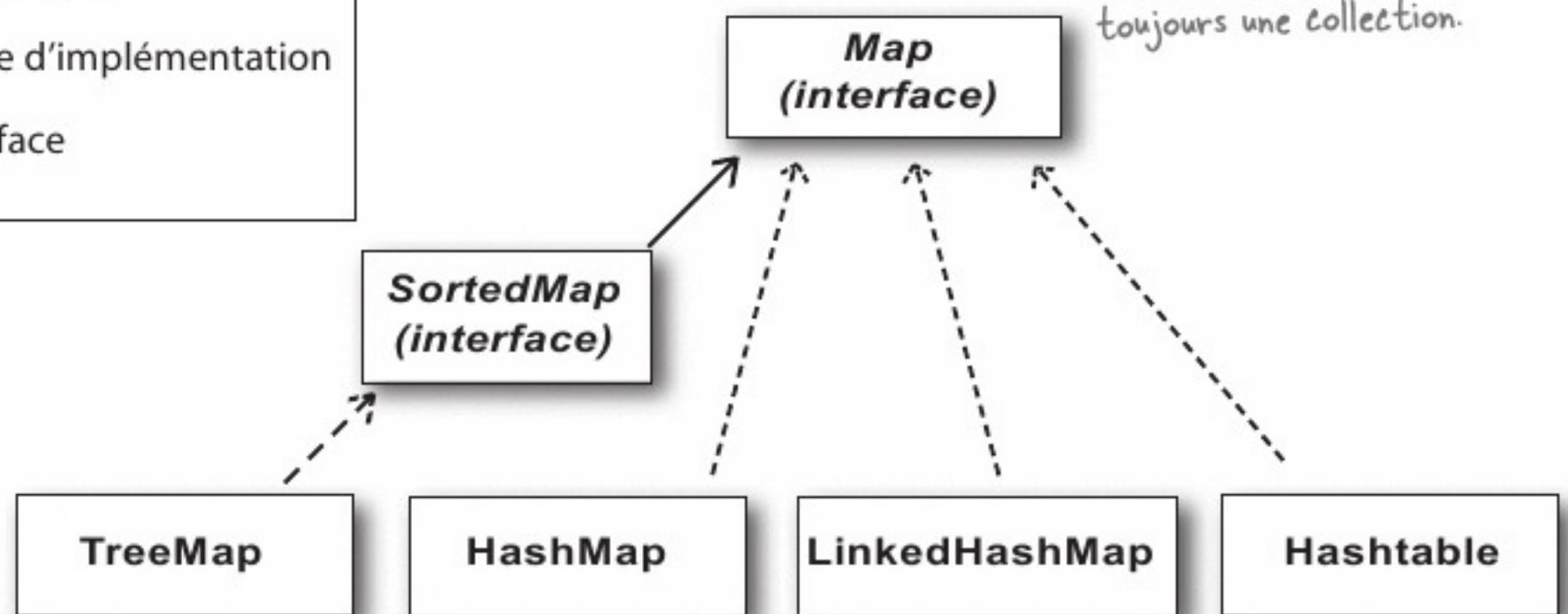
(Notez que nous n'avons pas représenté l'API complète. Il y a d'autres classes et interfaces, mais ce sont celles-ci qui nous intéressent le plus.)



LÉGENDE

	étend
	implémente
	classe d'implémentation
	interface

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.



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() { ← Comme nous n'avons pas modifié getMorceaux(), elle
        getMorceaux(); ← continue à placer les morceaux dans une ArrayList

        System.out.println(listeMorceaux);
        Collections.sort(listeMorceaux); ← Ici nous créons un HashSet paramétré
        System.out.println(listeMorceaux); ← pour contenir des Morceaux.

        HashSet<Morceau> hashMorceau = new HashSet<Morceau>(); ←

        hashMorceau.addAll(listeMorceaux); ← HashSet a une méthode addAll() qui peut prendre
        System.out.println(hashMorceau); ← une autre collection et l'utiliser pour remplir le
        } ← HashSet. C'est comme si nous avions ajouté chaque
        // méthodes getMorceaux() et ajouterMorceau()
    }
}
```

```
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).

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...)

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*.

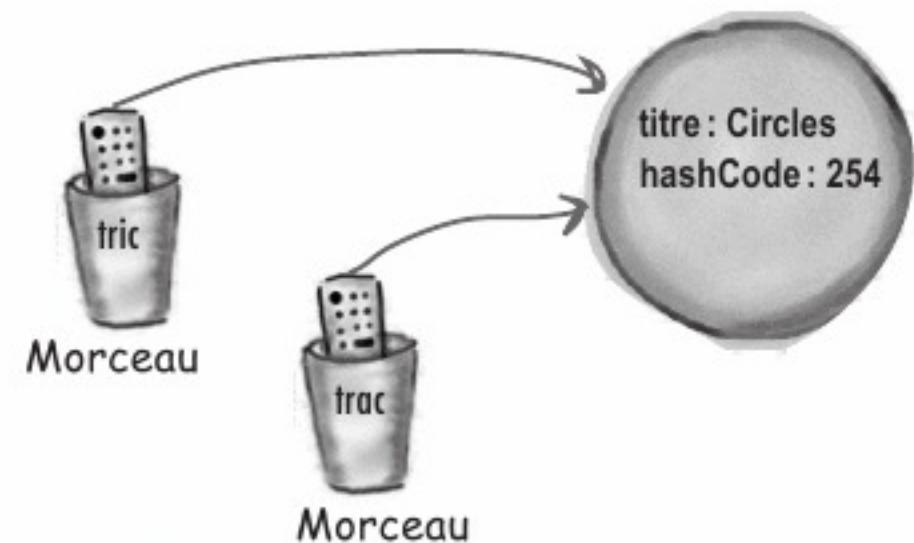
► Égalité des références

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.

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 la méthode `hashCode()` et `equals()` héritées de la classe `Object`, afin que deux objets différents soient considérés comme égaux.



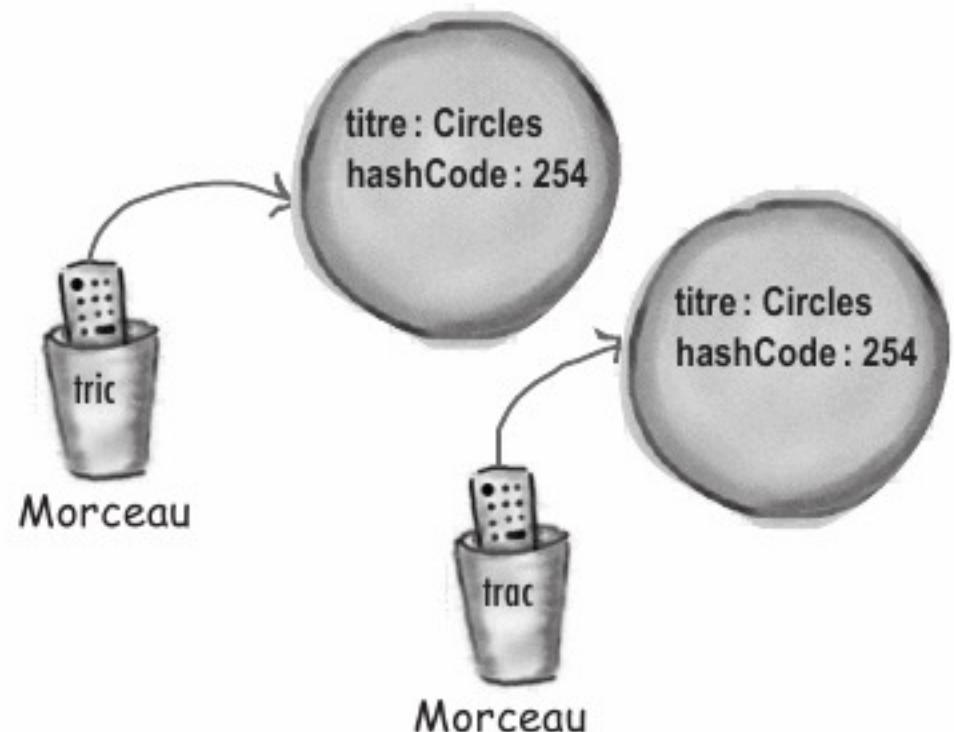
```
if (tric == trac) {
    // les deux références désignent
    // le même objet sur le tas
}
```

► 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 lappelez 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
}
```

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 !

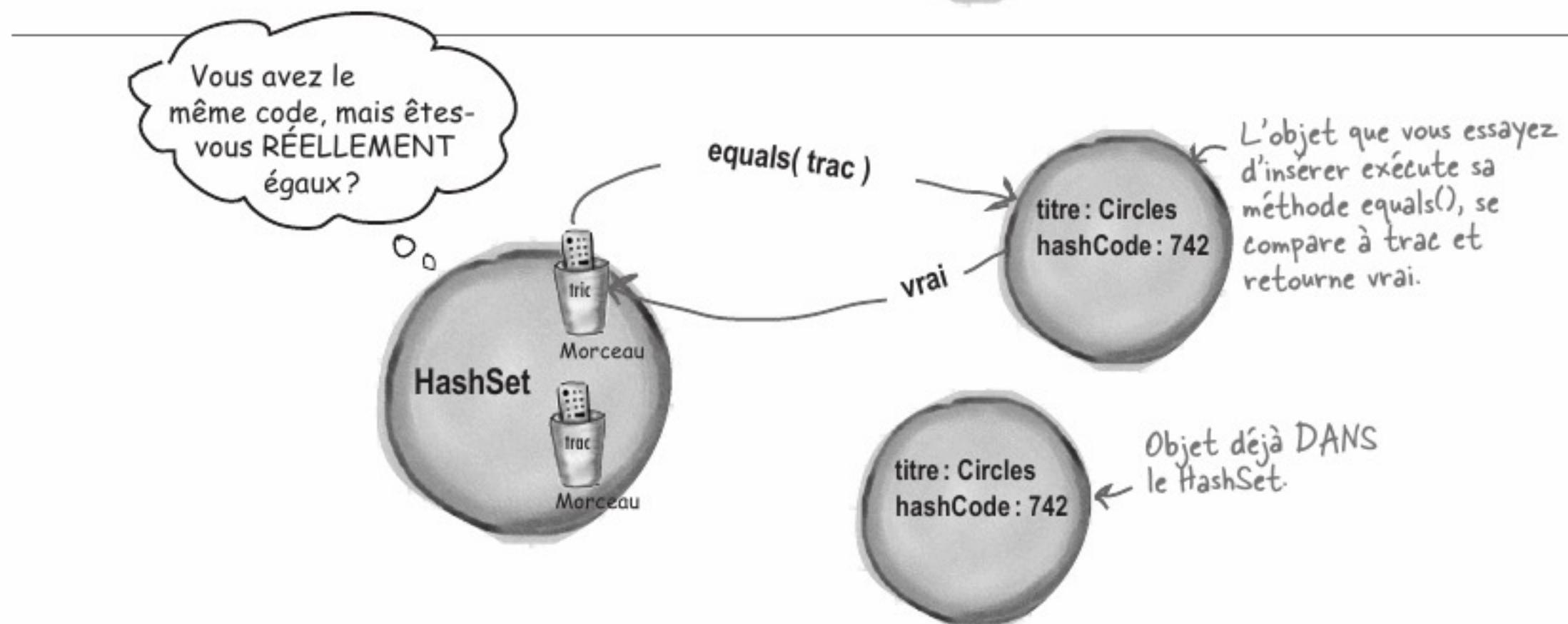
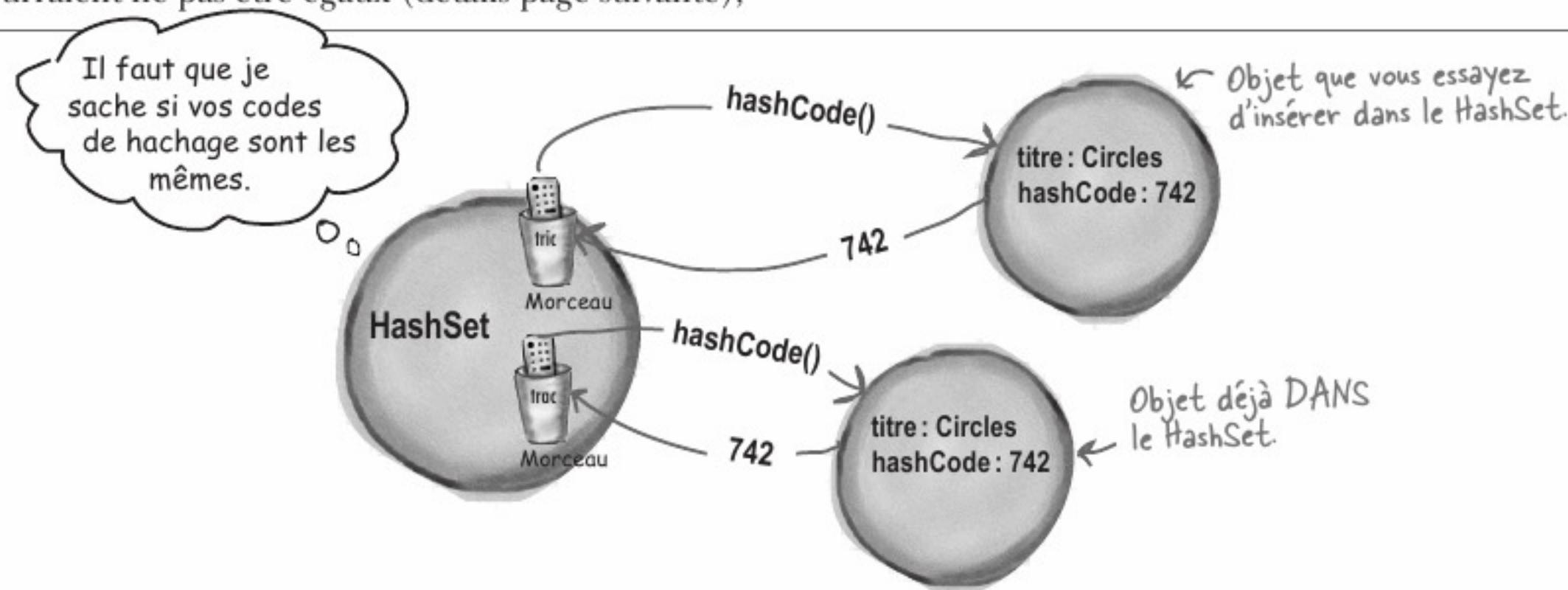
Vous devez donc redéfinir hashCode() pour être sûr que les deux objets ont la même valeur.

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.



redéfinir hashCode() et equals()

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()); ← 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.
    }

    public int hashCode() {
        return titre.hashCode(); ← Pareil ici... La classe a une méthode hashCode() redéfinie.
    }

    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.

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 MÊME 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]
```

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.

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 indéetectable 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.)

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.*



À vos crayons

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 ?

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.

OU

- Vous utilisez le constructeur surchargé du TreeSet qui accepte un Comparateur

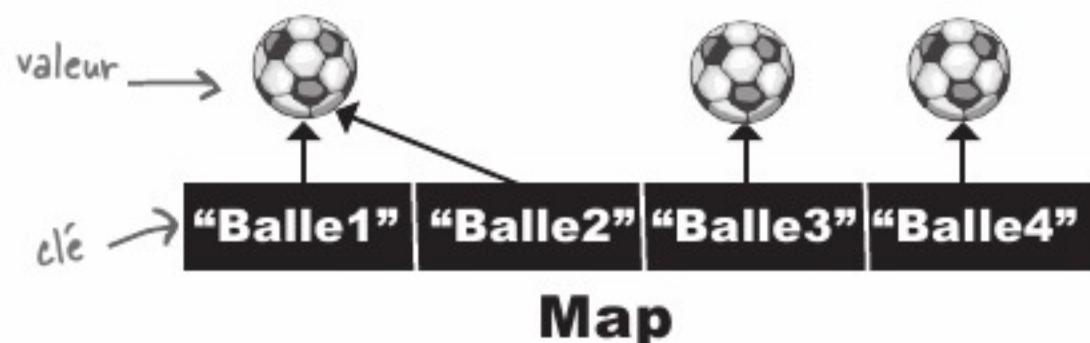
TreeSet fonctionne beaucoup comme la méthode sort() : vous avez le choix d'utiliser la méthode compareTo() de l'élément, à supposer que son type implémente l'interface Comparable, OU d'utiliser un Comparateur sur mesure qui sait comment trier les éléments. Pour utiliser ce Comparateur, vous appelez le constructeur de TreeSet qui accepte un Comparateur.

```
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));  
    }  
  
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);  
    }  
}
```

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); ← Au lieu de add(), utilisez put() qui bien sûr
        scores.put("Skyler", 420); accepte maintenant deux arguments (clé, valeur).

        System.out.println(scores);
        System.out.println(scores.get("Bert")); ← La méthode get() accepte une clé et retourne
    }                                         une valeur (en l'occurrence un Integer).
}
```

HashMap nécessite DEUX paramètres de type – un pour la clé et un pour la valeur.

```
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.

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.

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); ← Appeler accepterAnimaux(), en
                                utilisant les deux types comme
                                arguments...
    }

    public void accepterAnimaux(Animal[] animaux) {
        for(Animal a: animaux) {
            a.manger();
        }
    } ← 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.)
}

```

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).

← Soulignons un point crucial : la méthode accepterAnimaux() accepte un Animal[] ou un Chien[], puisque Chien EST-UN Animal. Polymorphisme en action.

```
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

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);**

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>

```
public void go() {
    ArrayList<Animal> animaux = new ArrayList<Animal>();
    animaux.add(new Chien());
    animaux.add(new Chat()); ← Nous devons les ajouter un par un, puisqu'il n'y a pas de
    animaux.add(new Chien());   syntaxe abrégée comme pour la création des tableaux.

    accepterAnimaux(animaux); ← 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();
    }
}
```

Simple transformation de Animal[] en ArrayList<Animal>.

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
```

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?

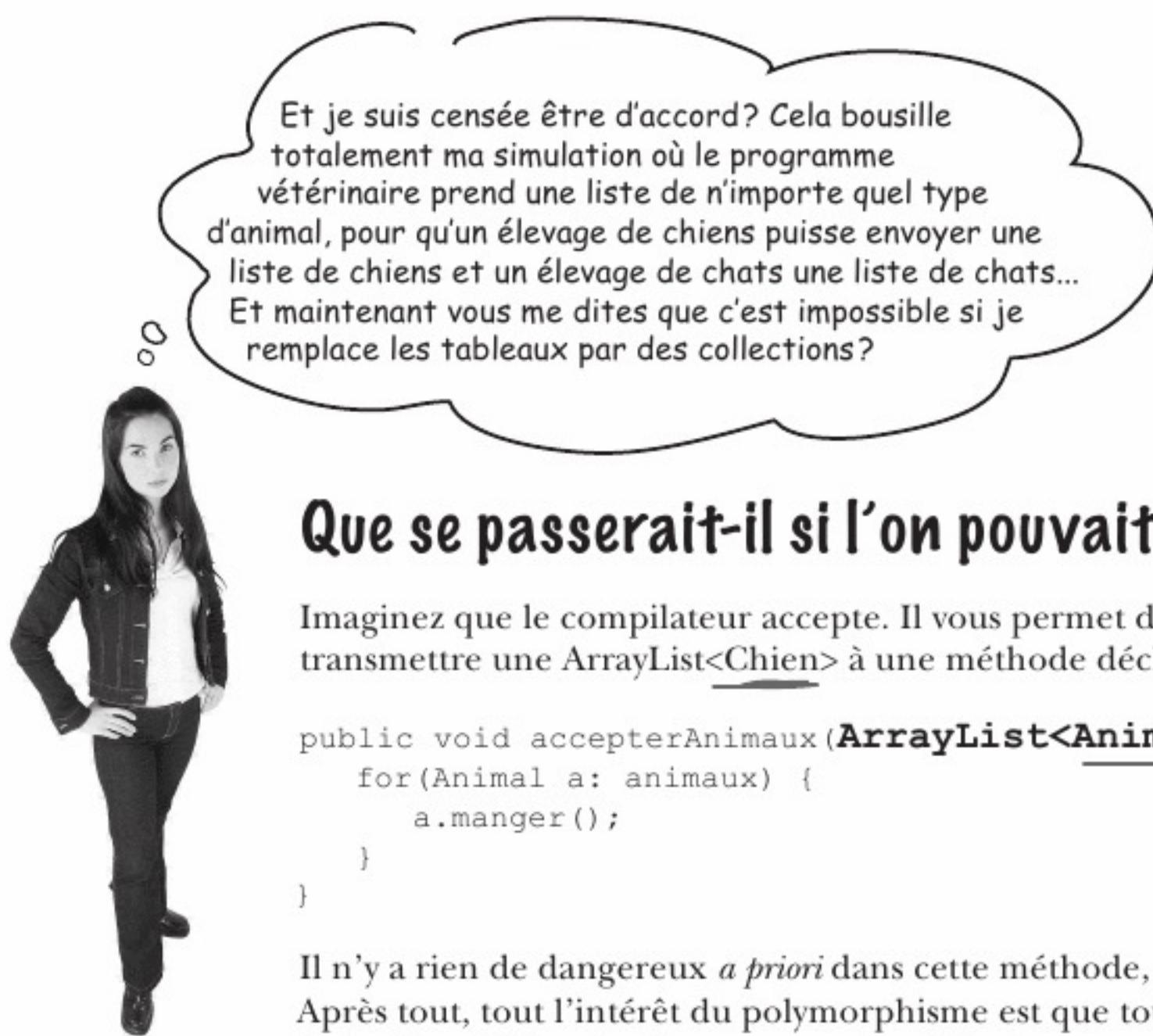
Ne transmettre que ArrayList<Chien>

```
public void go() {  
    ArrayList<Animal> animaux = new ArrayList<Animal>();  
    animaux.add(new Chien());  
    animaux.add(new Chat());  
    animaux.add(new Chien());  
    accepterAnimaux(animaux); ← Nous savons que cette ligne fonctionnait  
  
    ArrayList<Chien> chiens = new ArrayList<Chien>();  
    chiens.add(new Chien());  
    chiens.add(new Chien());  
    accepterAnimaux(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...



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>.



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 à 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 Édition 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.

Ne serait-ce pas merveilleux s'il y avait un moyen d'utiliser des types polymorphes comme arguments des méthodes, pour que mon programme vétérinaire puisse accepter des listes de Chiens et des listes de Chats? Je pourrai alors parcourir les listes et appeler leur méthode vacciner(), mais le typage serait toujours fiable et je ne pourrais pas insérer un Chat dans la liste des Chiens. Mais je suppose que ce n'est qu'un fantasme...



Les jokers à la rescouuse

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();
    }
}
```

«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!

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>`

Quand vous utilisez un joker dans l'argument de votre méthode, le compilateur vous EMPÈCHE 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());
```

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)
```



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>();`

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 DECROISSANT?



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]

```

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()



Vous ÊTES le compilateur : Solution

Compile ?

- `ArrayList<Chien> chiens = 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>();`

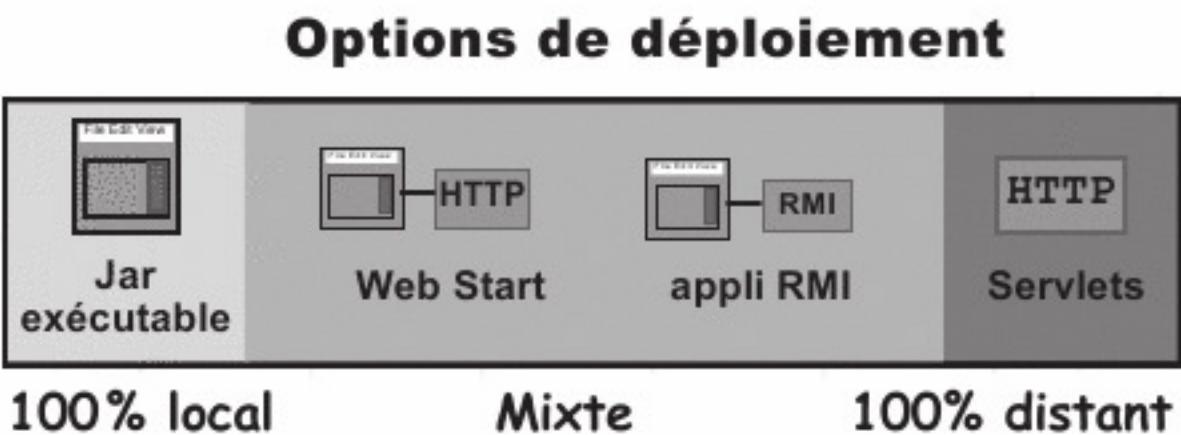
Déployez votre code



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...

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.



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 ?

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 ?



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 ?



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.

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.

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 :

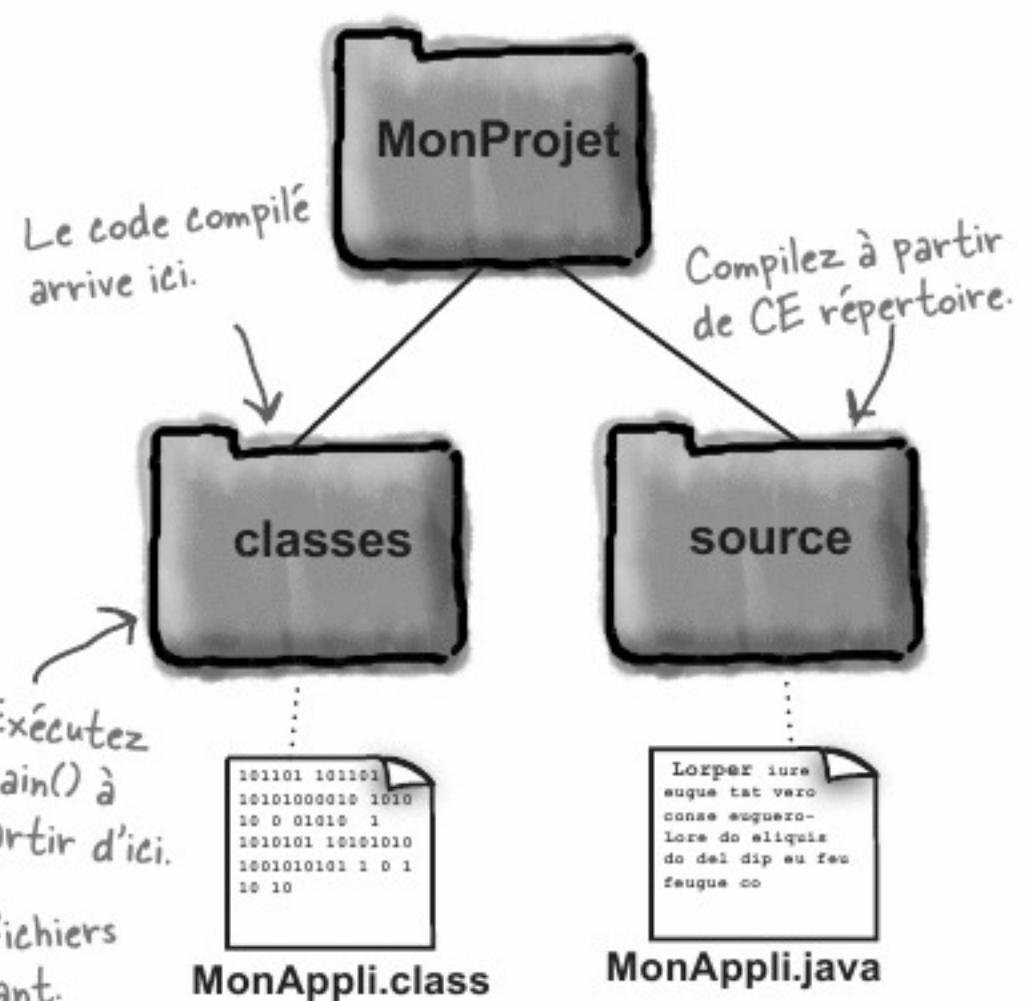
```
%javac -d ../classes *.java
```

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 ".")

Placez votre code Java dans un fichier JAR



JAR signifie Java ARchive. 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 ?

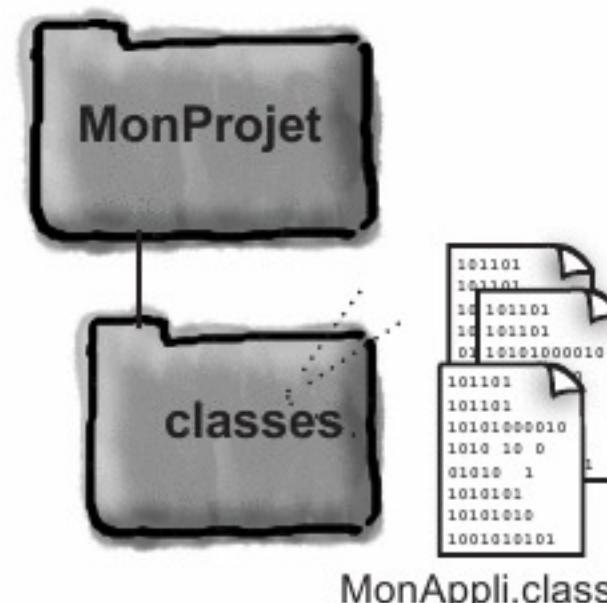
Réponse : vous rendez le JAR **exécutable**.

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 ».

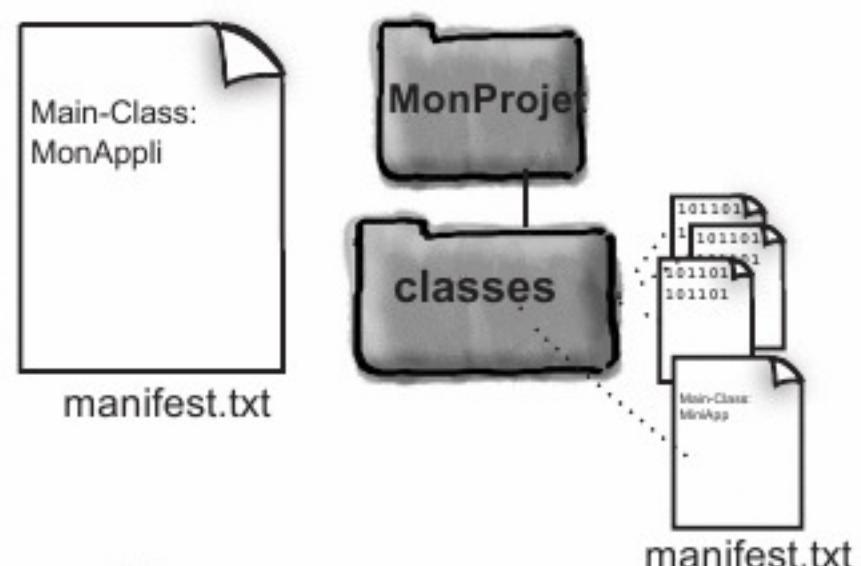


② Créez un fichier manifest.txt qui déclare dans quelle classe se trouve main()

Créez un fichier texte nommé manifest.txt qui ne contient qu'une ligne :

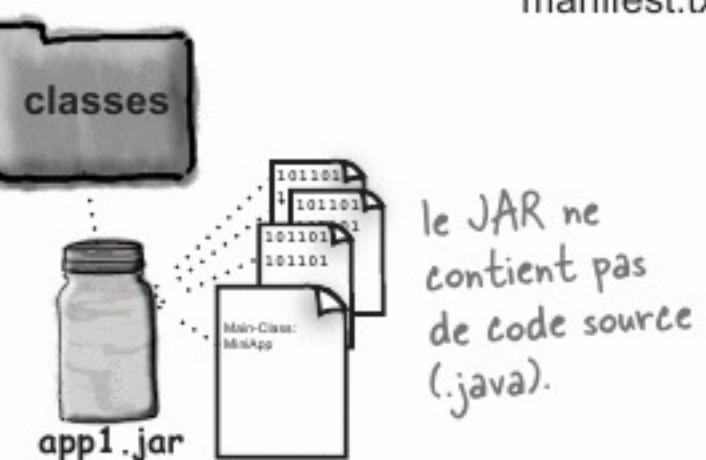
Main-Class: MonAppli ← Pas d'extension .class

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 ».

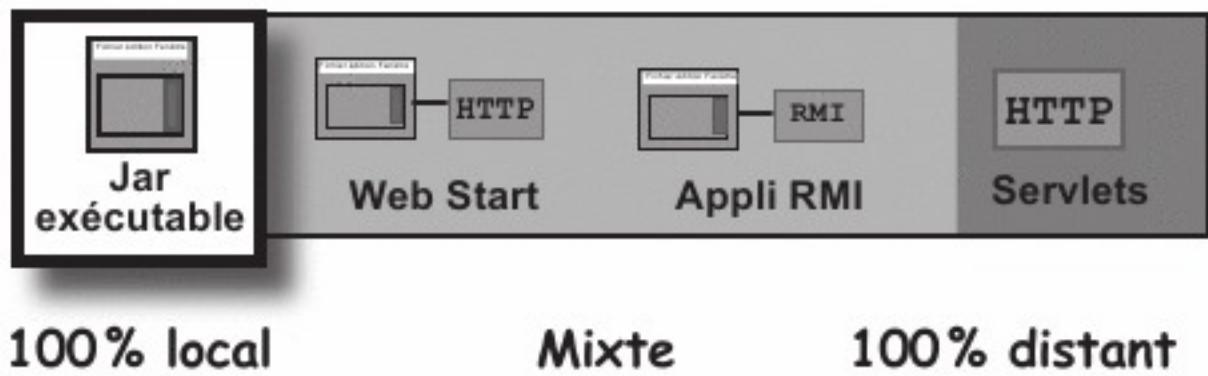


③ Exécutez l'outil jar pour créer un fichier JAR qui contient toutes les classes plus le manifeste.

```
%cd MiniProject/classes
%jar -cvmf manifest.txt app1.jar *.class
OR
%jar -cvmf manifest.txt app1.jar MonAppli.class
```

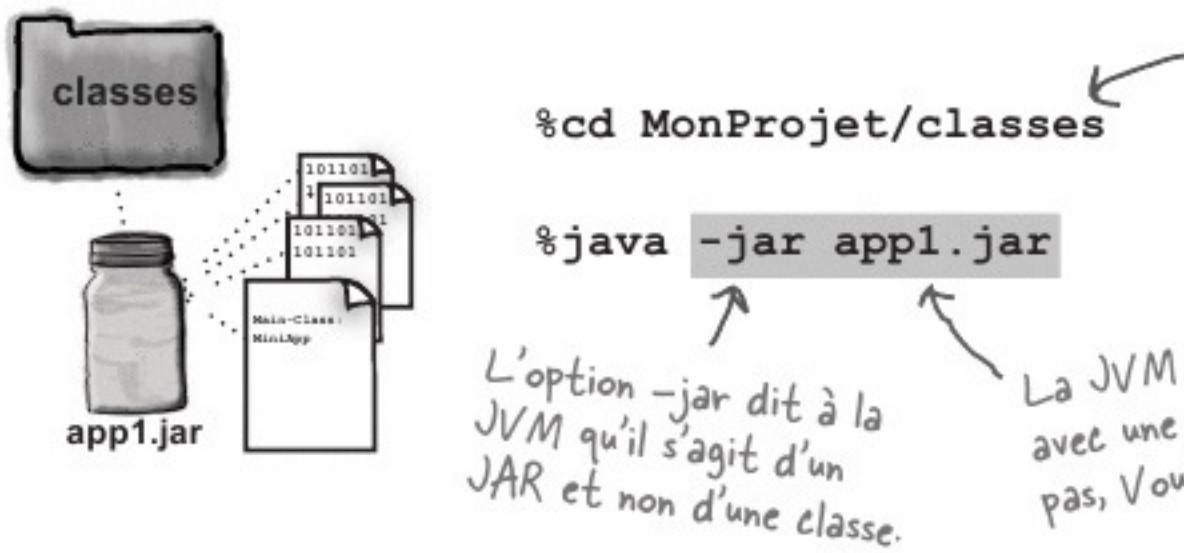


JAR exécutable



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.



Près de 100 % des applications Java locales sont déployées sous forme de JAR exécutable.

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.

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.

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.

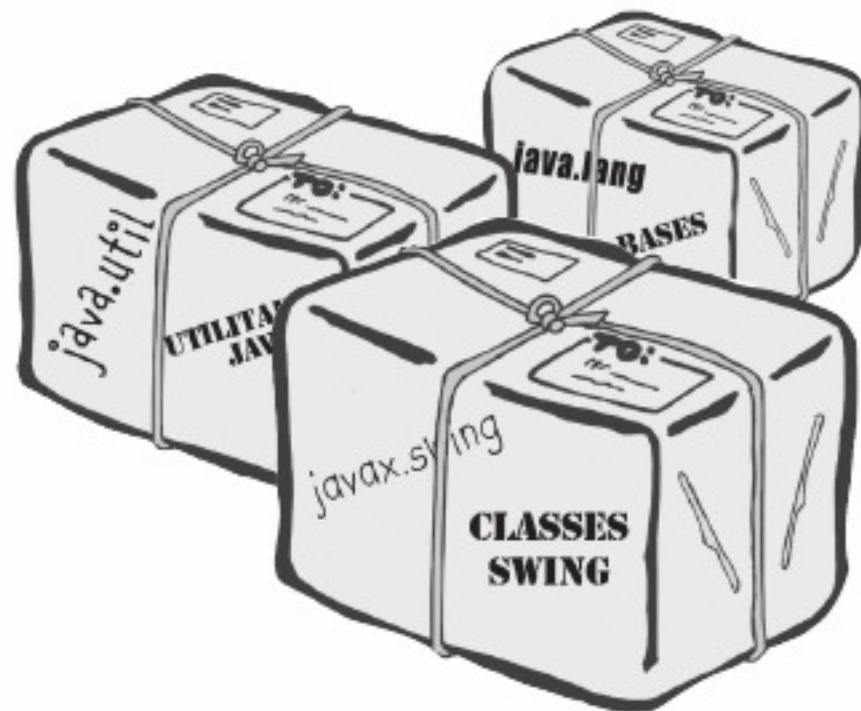
Les packages empêchent les conflits de nommage

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***.

ArrayList, JButton est ***javax.swing.JButton*** et Socket est ***java.net.Socket***.

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.



Structure des packages de l'API Java pour :

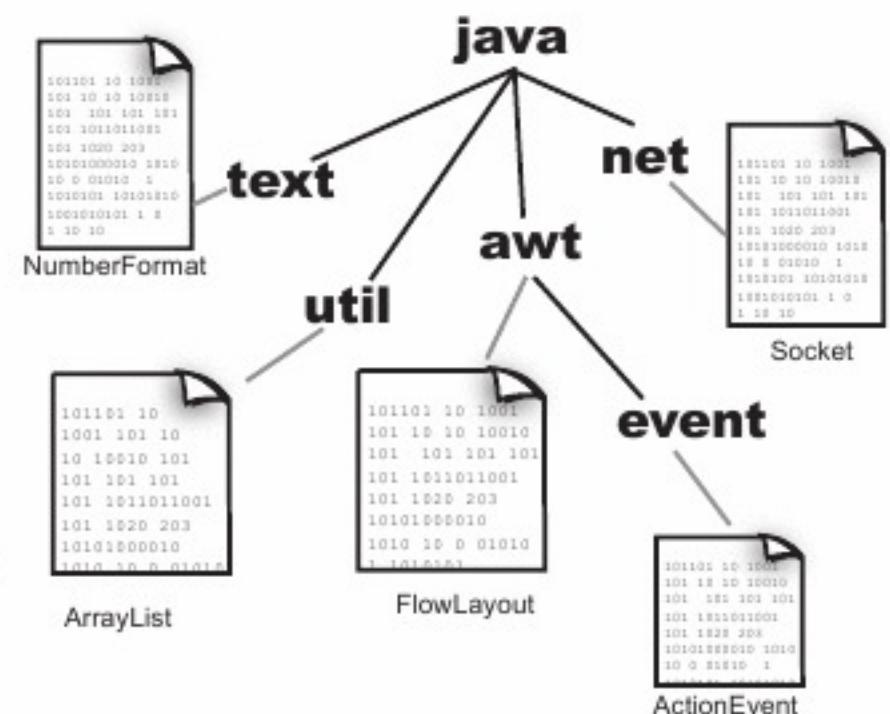
java.text.NumberFormat

java.util.ArrayList

java.awt.FlowLayout

java.awt.event.ActionEvent

java.net.Socket



Cette figure ne vous rappelle-t-elle pas quelque chose? Ne ressemble-t-elle pas furieusement à une arborescence de répertoires?



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 nommée 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 Barthélémy Durand, deux domaines différents ne peuvent pas être nommés ceci.com.

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.



Noms de domaine inversés

fr.jtlp.projets.LivreJava

Commencez le nom de package par votre nom de domaine inversé séparé par un point (.), puis ajoutez votre propre structure.

L'initiale des noms de classes est toujours une majuscule.

Si projets.LivreJava peut être un nom répandu, ajouter fr.oreilly signifie que nous n'avons plus à nous soucier que des développeurs internes.

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.*;

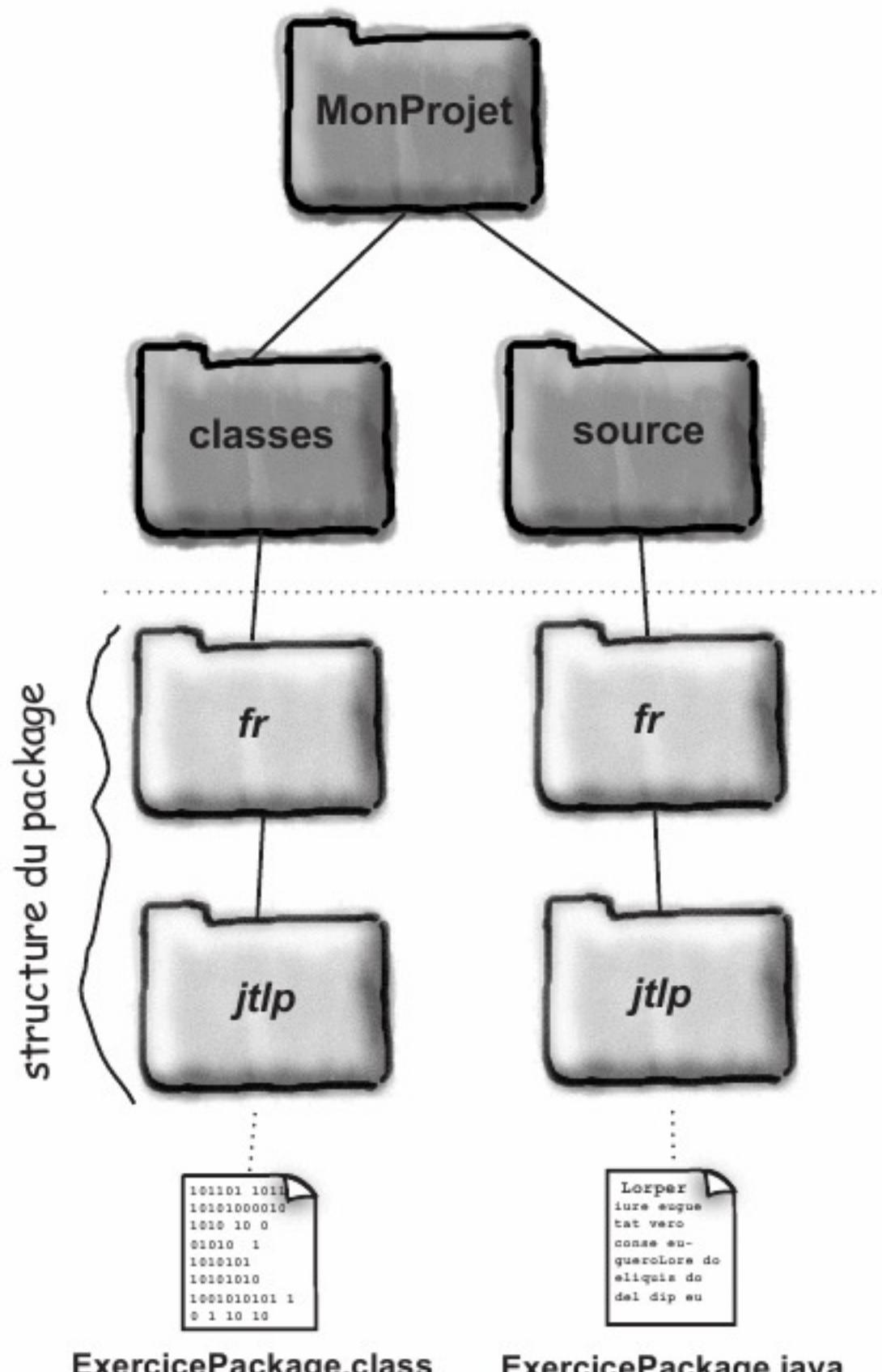
public class ExercicePackage {
    // code qui change la vie
}
```

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.

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`

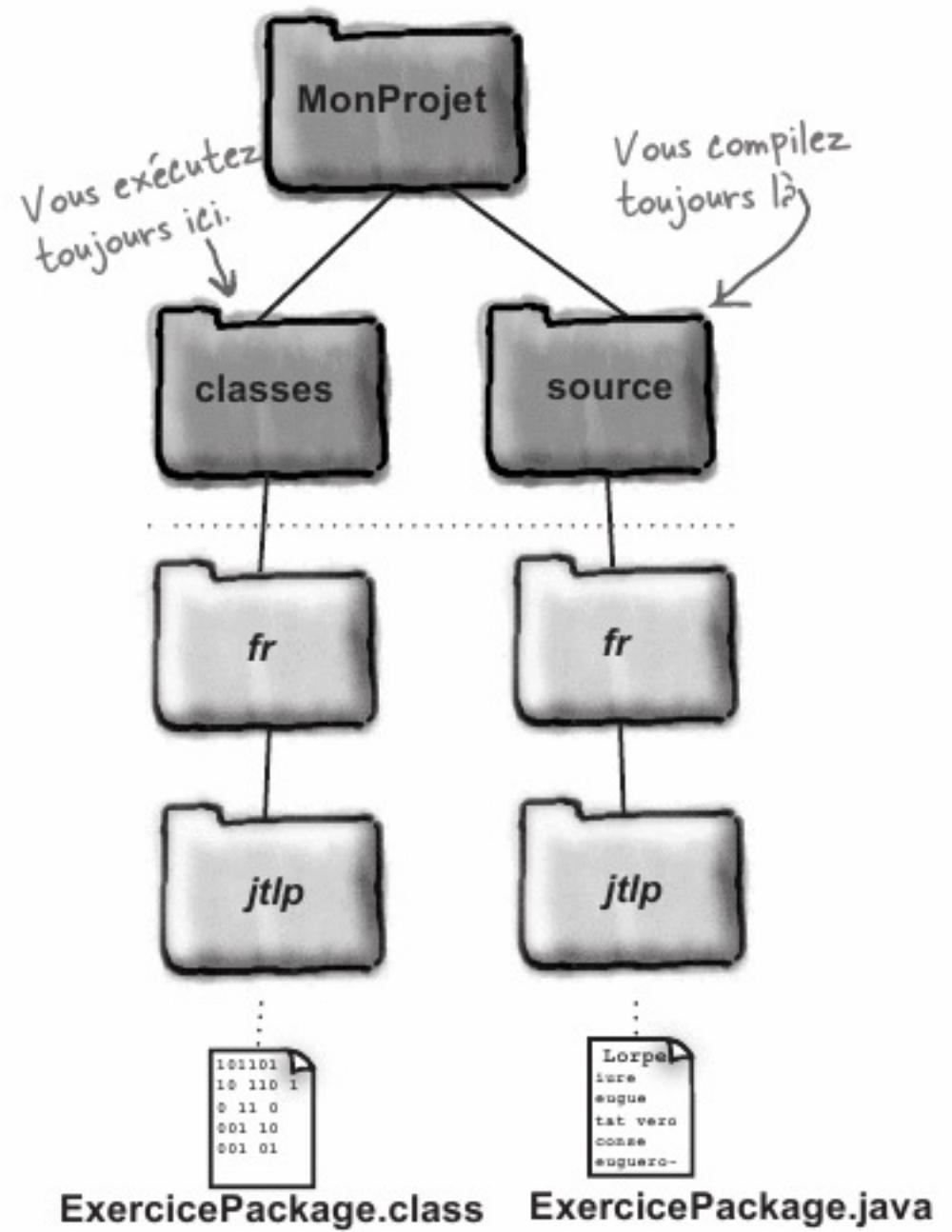
Compile tous les fichiers source (.java) de ce répertoire.

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!

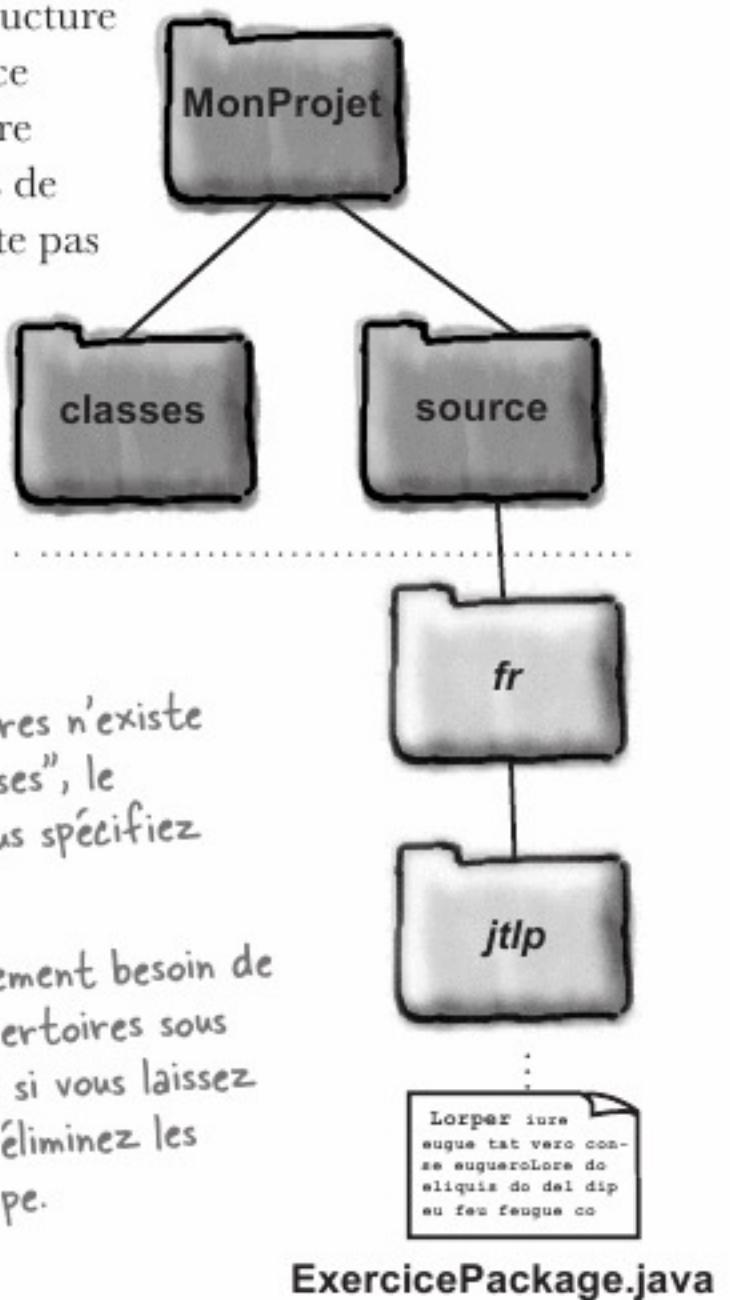


L'option **-d** est vraiment cool

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.



*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... ».

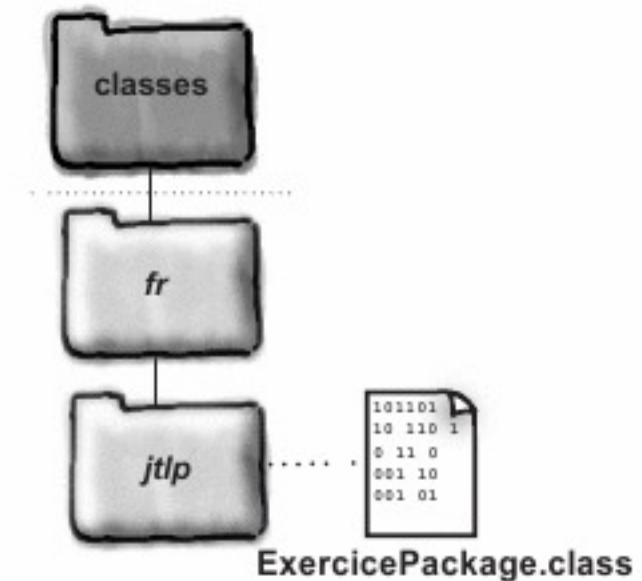
Créer un JAR exécutable avec des packages



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

- Vérifiez que tous vos fichiers classes sont dans la bonne structure de package, sous le répertoire classes.

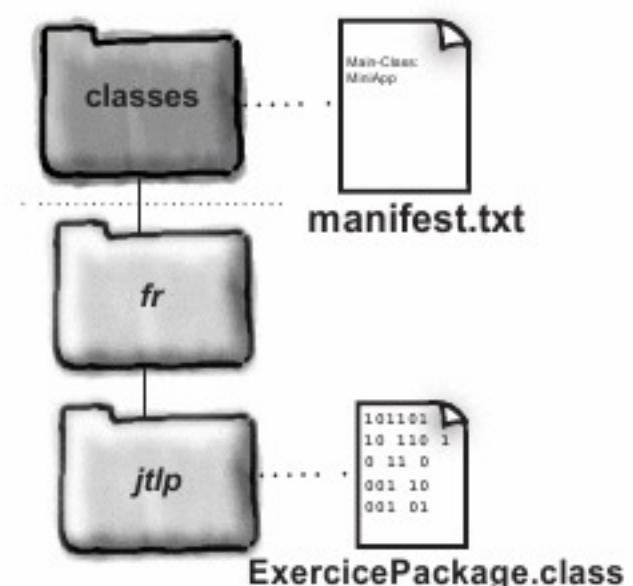


- Créez un fichier manifest.txt qui spécifie quelle classe contient la méthode main(), et veillez à utiliser le nom complet de la classe !

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».



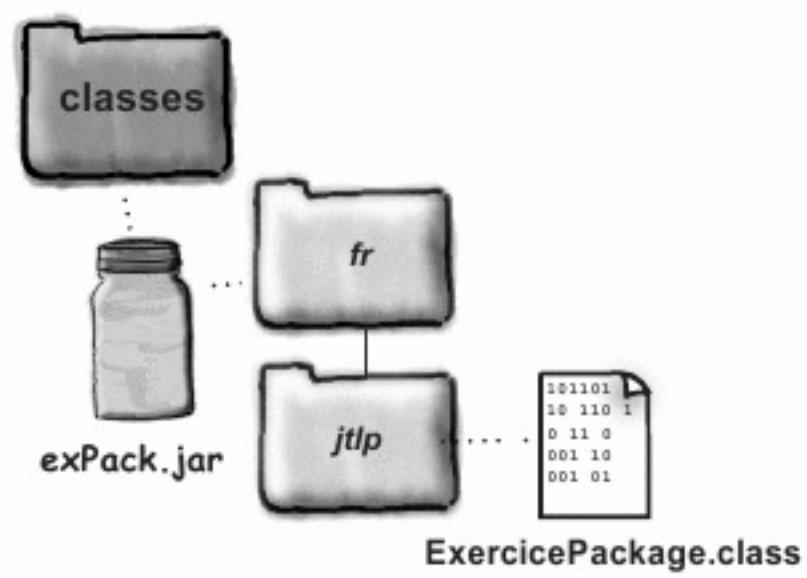
- Exécutez l'outil jar pour créer un fichier JAR qui contiendra les répertoires du package plus le manifeste.

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 !



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
```

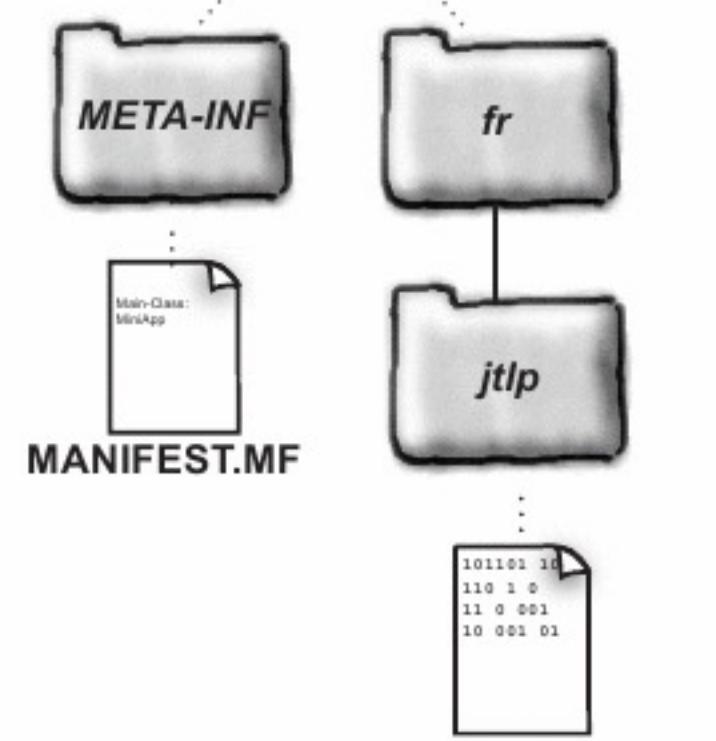
-tf signifie "Table File", autrement dit « affiche-moi un tableau du fichier JAR ».

```
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.



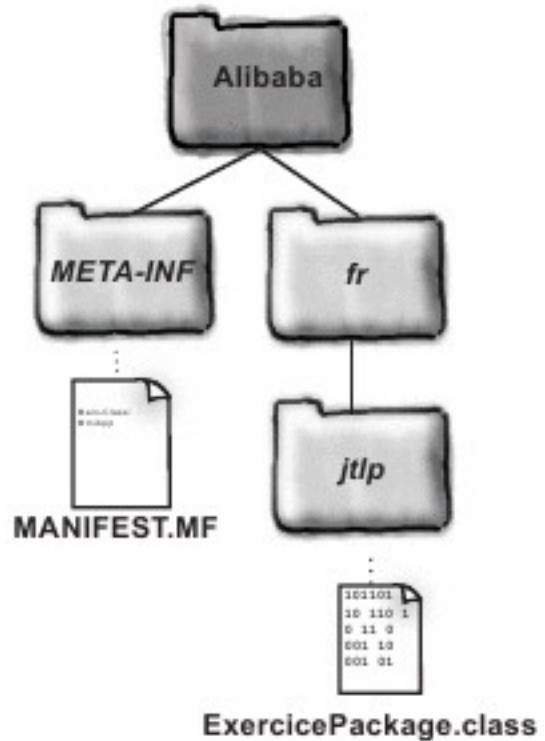
L'outil jar construit automatiquement un répertoire META-INF et y place le manifeste.



② Extraire le contenu d'un JAR

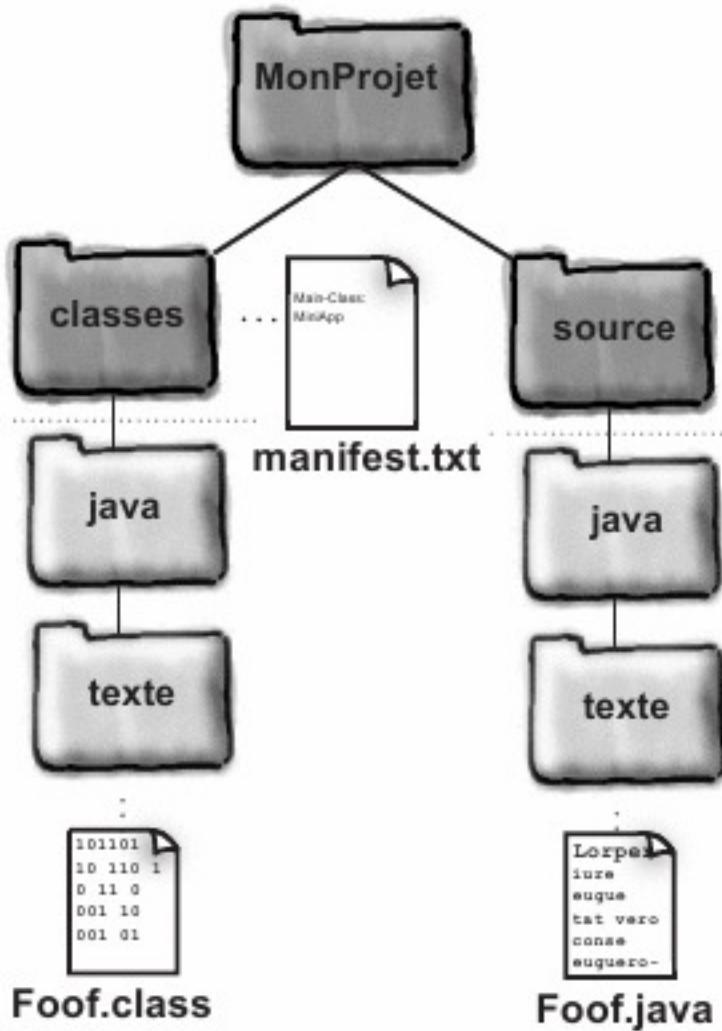
```
% cd Alibaba
% jar -xf exPack.jar
```

-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 « métainformation ». 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).

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 ?

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 :

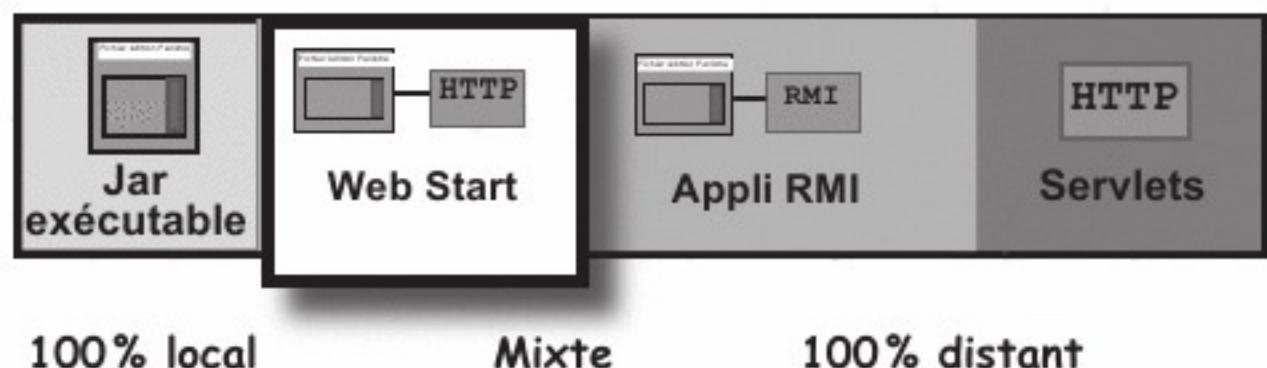

```
jar -cvfm manifest.txt MonJar.jar com
```
- Toute la structure de répertoires du package (et seulement les répertoires du package) doivent se retrouver immédiatement dans le JAR.
- Pour lancer un fichier JAR exécutable, tapez :


```
java -jar MonJar.jar
```

quel rêve ce serait...

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...





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.

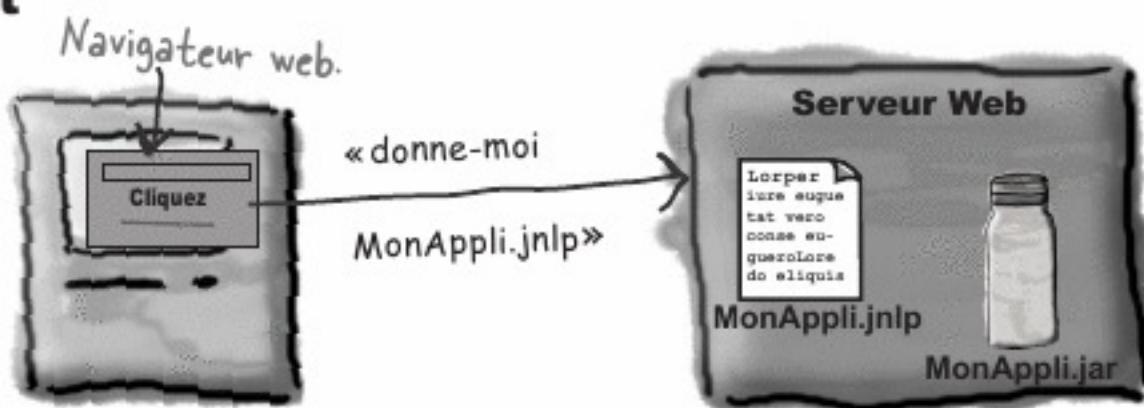
Les utilisateurs finaux lancent une application Java Web Start en cliquant sur un lien dans une page web. Mais une fois téléchargée, elle s'exécute en dehors du navigateur, tout comme n'importe quelle autre application Java autonome. En réalité, une application JWS n'est autre qu'un JAR exécutable distribué sur le Web.

Comment marche Java Web Start

- ① Le client clique dans une page web sur un lien vers votre application JWS (un fichier .jnlp).

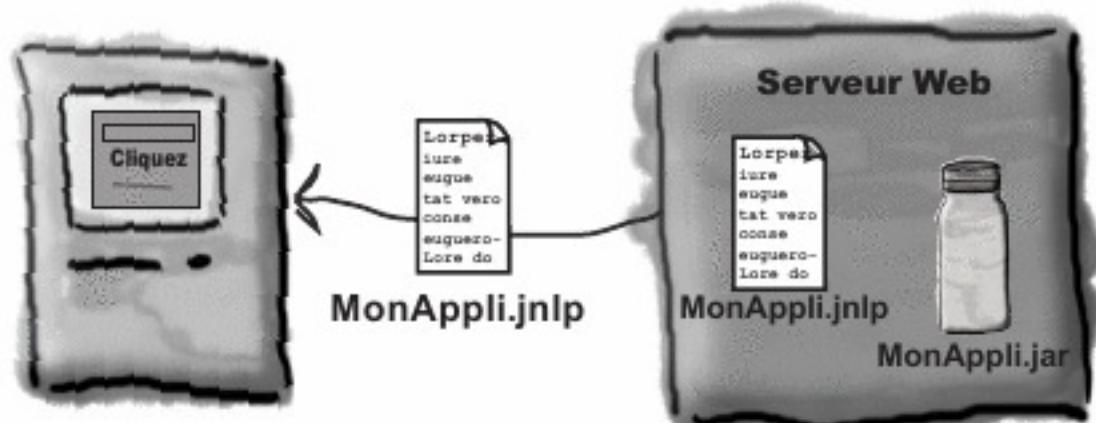
Le lien dans la page Web

```
<a href="MonAppli.jnlp">Cliquez</a>
```

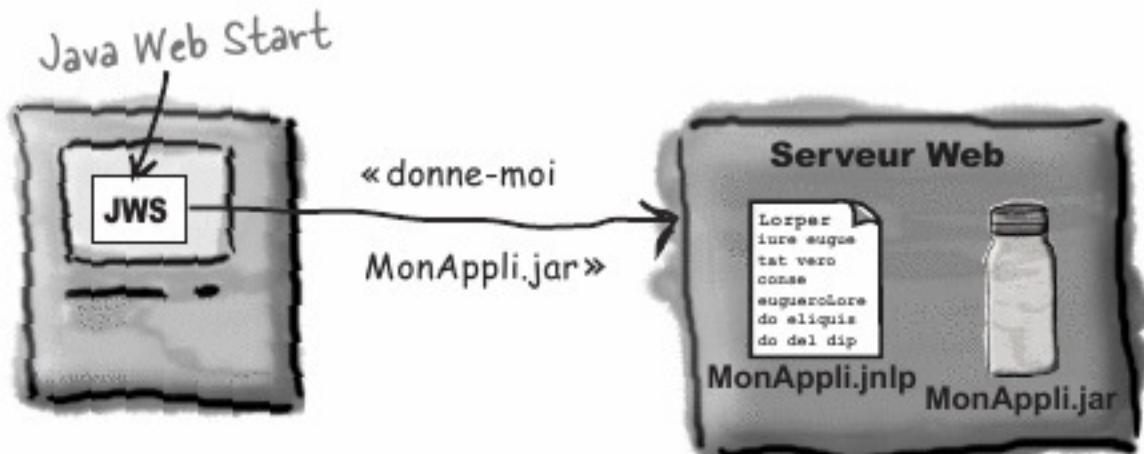


- ② Le serveur web (HTTP) reçoit la requête et renvoie un fichier .jnlp (ce n'est PAS le JAR).

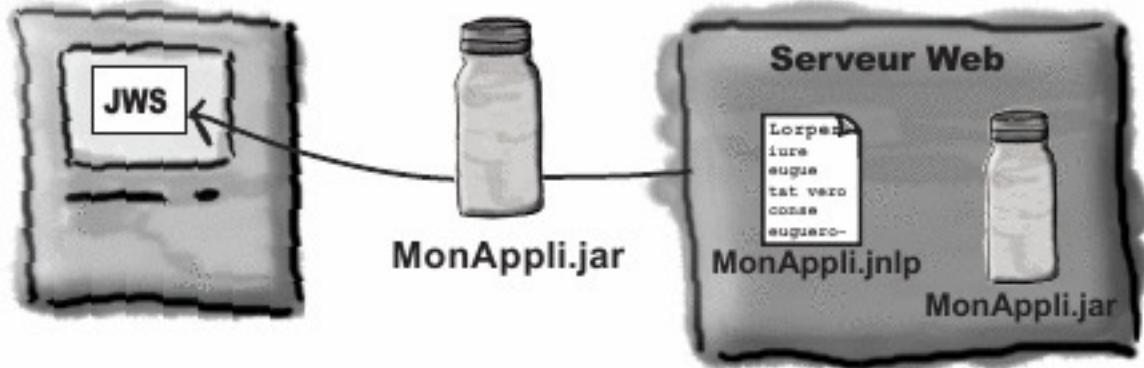
Le fichier .jnlp est un document XML qui spécifie le nom du fichier JAR exécutable de l'application.



- 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.



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"
```

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".

```
<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/>
```

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.

```
</information>
```

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.

```
<resources>
```

```
  <j2se version="1.3+/">
```

Cette ligne spécifie que l'application nécessite au moins Java 1.3.

```
  <jar href="MonAppli.jar"/>
```

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.

```
</resources>
```

```
<application-desc main-class="HelloWebStart"/>
```

```
</jnlp>
```

Comme l'entrée Main-Class du manifeste, cette ligne indique quelle classe du JAR contient la méthode main().

Étapes de la création et du déploiement d'une application Java Web Start

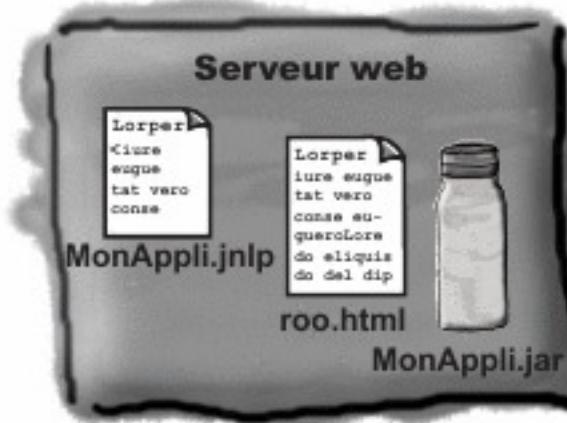
- ① Créez un JAR exécutable pour votre application.



- ② Écrivez un fichier .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.

Serveur web
configurer
le type MIME

- ⑤ Créez une page web contenant un lien vers votre fichier .jnlp

```
<HTML>
  <BODY>
    <a href="MonAppli22.jnlp">Lancer mon application</a>
  </BODY>
</HTML>
```





L'œuf et la poule



Regardez la séquence d'événements ci-dessous et placez-les dans l'ordre dans lequel ils se produisent dans une application JWS.



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.

packages, archives jar et déploiement

1.

2.

3.

4.

5.

6.

7.



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.

exercice: vrai ou faux



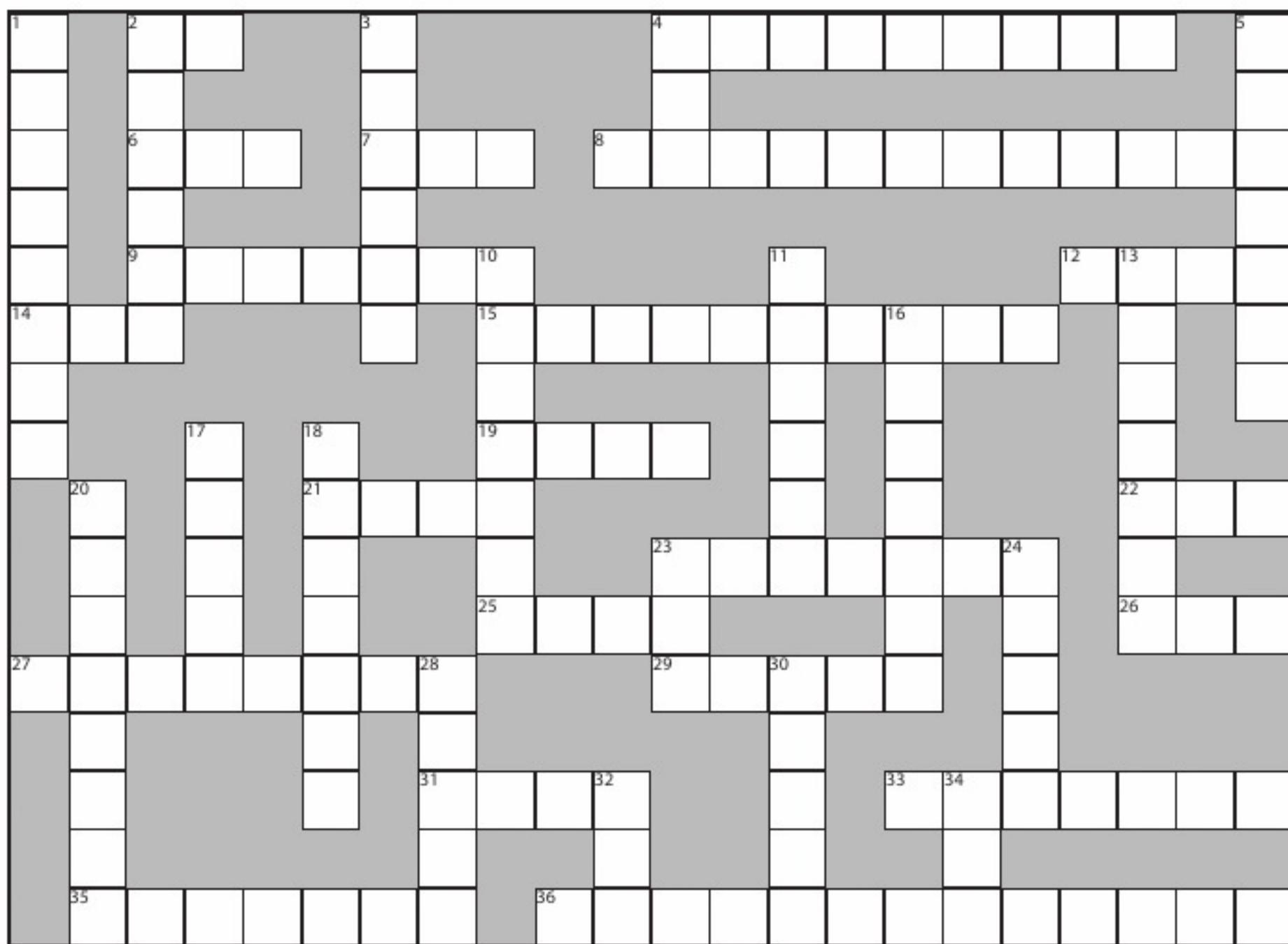
Dans ce chapitre, nous avons étudié les packages, le déploiement et JWS. Vous devez déterminer si chacune des instructions ci-dessous est vraie ou fausse.

拇指 Vrai ou Faux 拇指

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.



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

Verticalement

- 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
- 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



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

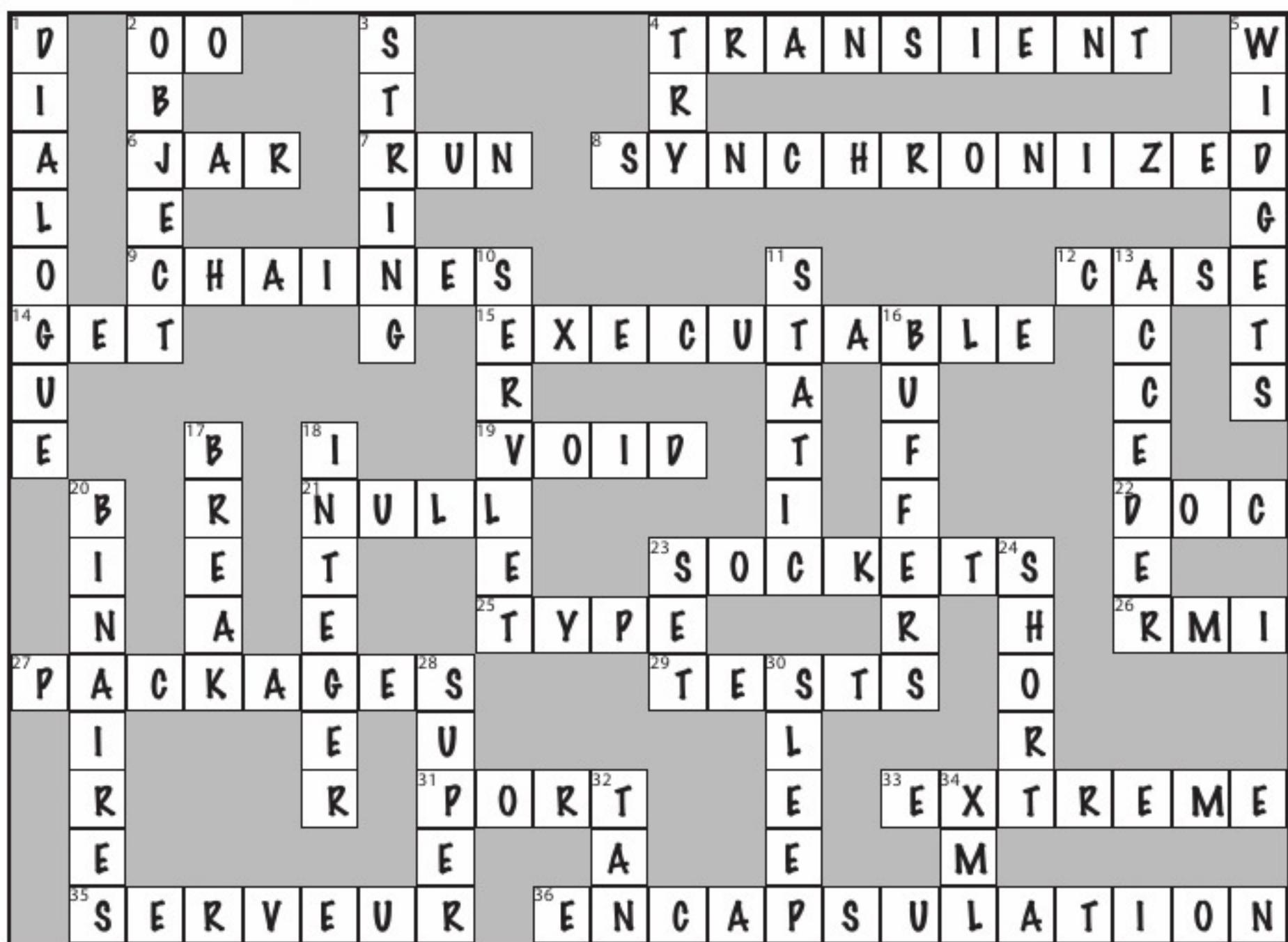
16. Les applications JWS nécessitent un fichier .nlp (*Network Launch Protocol*) pour fonctionner.

Faux

17. La méthode main() d'une application JWS est spécifiée dans son fichier JAR.



Mots-Croisés 7.0



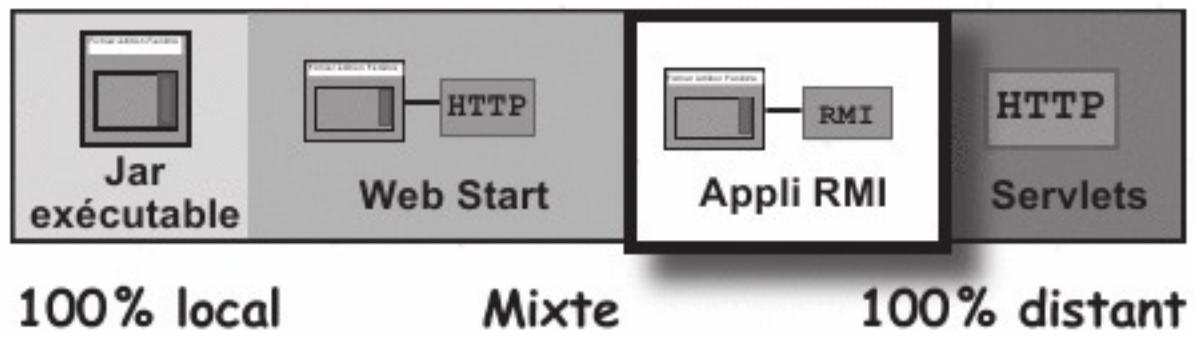
Informatique distribuée



Tout le monde dit que les relations à distance sont difficiles. Mais c'est facile avec RMI. Peu importe où nous sommes réellement, RMI nous donne l'impression d'être ensemble.

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*.

combien y a-t-il de tas?

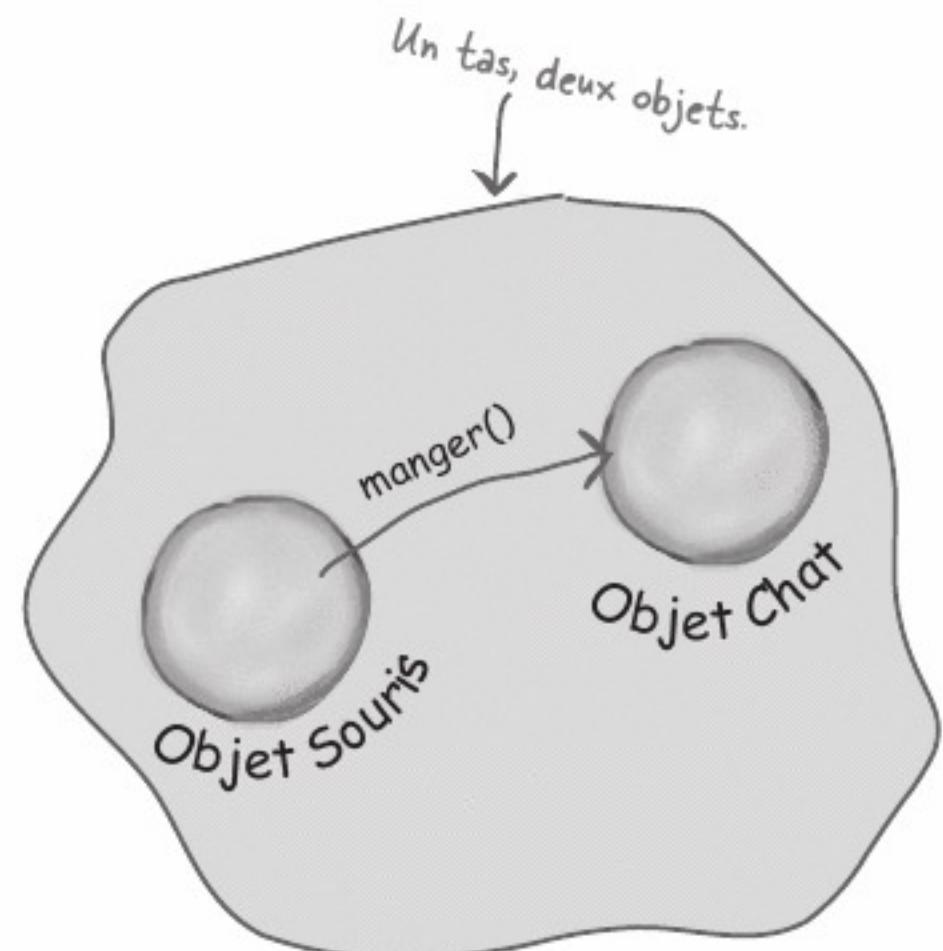


Les appels de méthodes sont toujours entre deux objets sur le même tas.

Jusqu'ici, toutes les méthodes que nous avons appelées concernaient un objet s'exécutant 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.



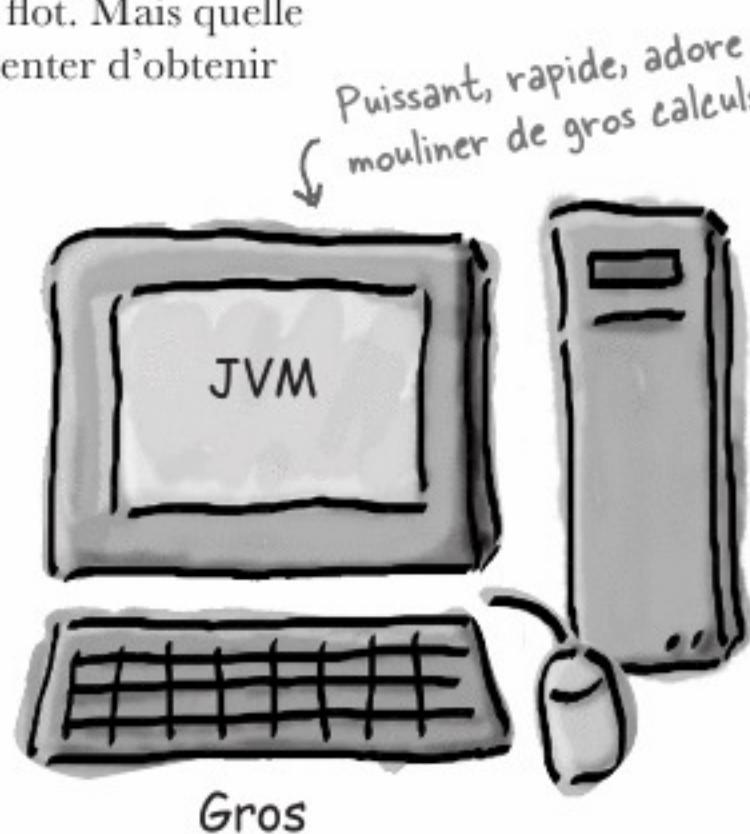
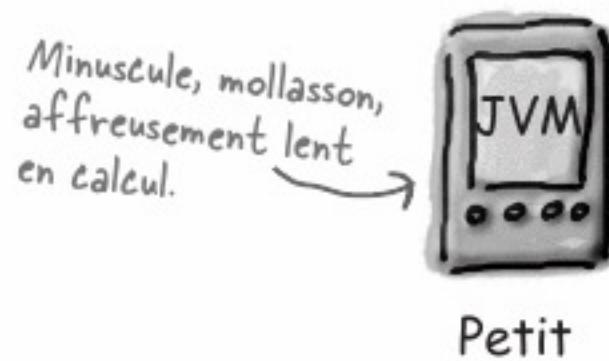
Dans la plupart des applications, quand un objet appelle une méthode sur un autre, les deux objets sont sur le même tas. Autrement dit, ils s'exécutent dans la même JVM.

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...



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)
```

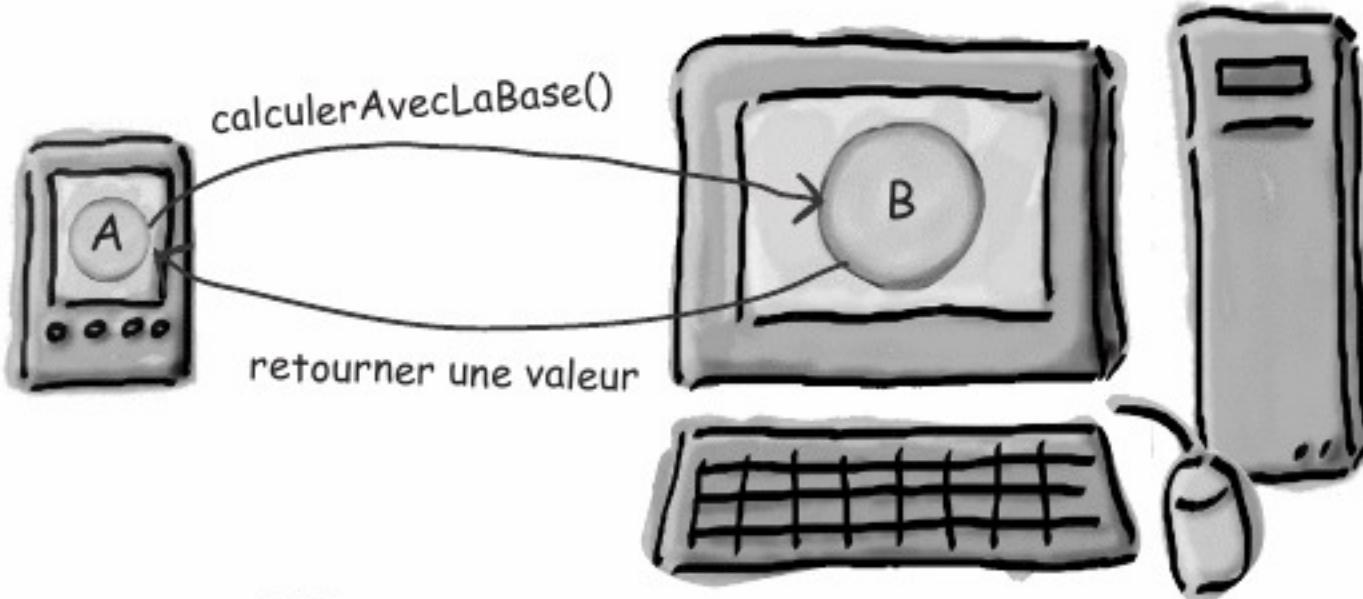
et obtenir le résultat.

Mais comment Petit obtient-il une référence à un objet qui réside sur Gros?

deux objets, deux tas

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 différent) 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 vieux appel de méthode de tous les jours mais avec la puissance d'un appel de méthode distant. Voilà notre objectif.

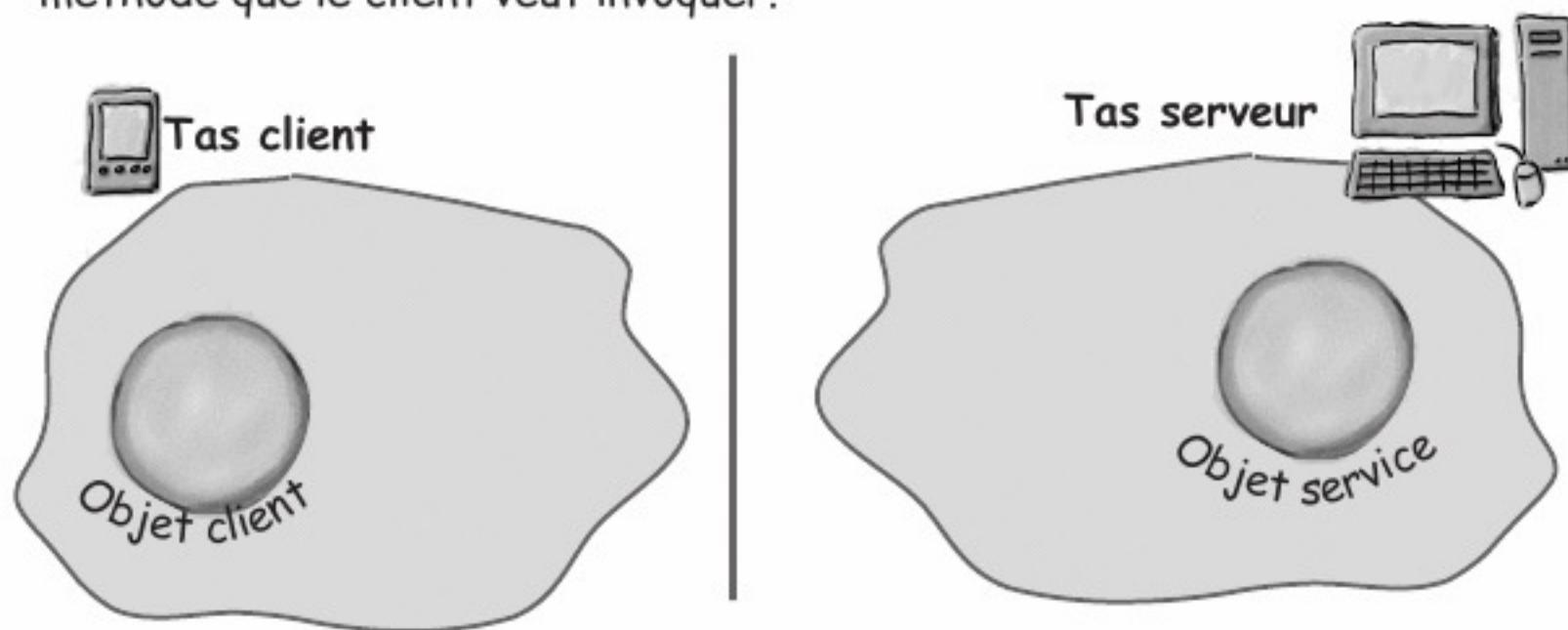
C'est ce que RMI (Remote Method Invocation) vous offre !

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.

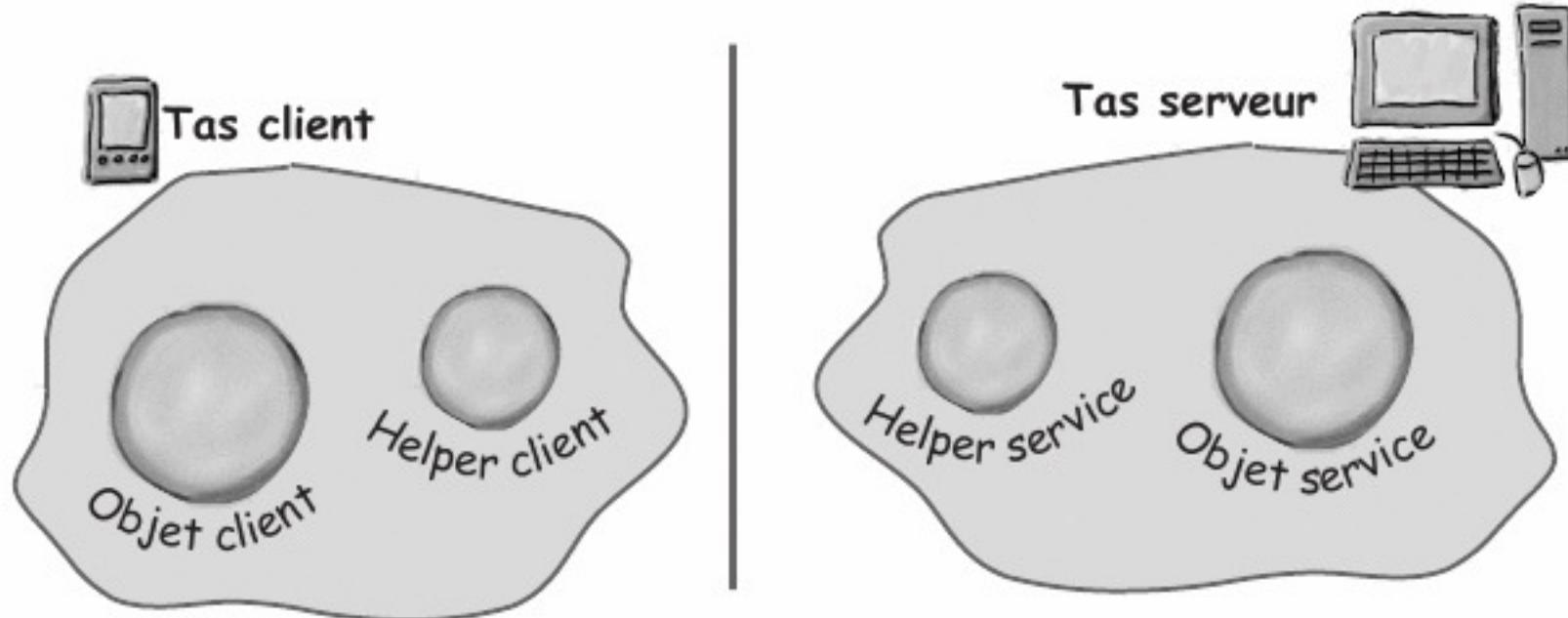
Concevoir des appels de méthodes distants

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.



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.*

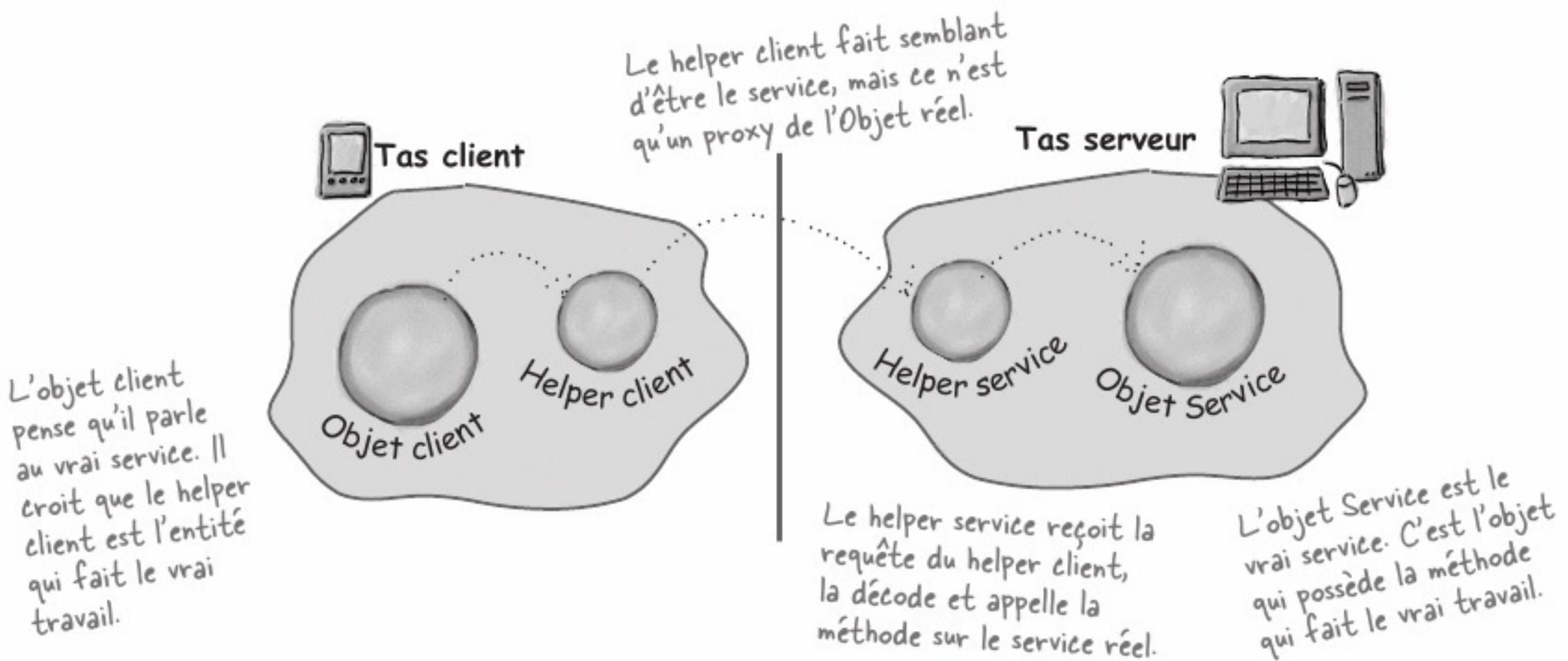
En d'autres termes, l'objet client pense qu'il appelle une méthode sur le service distant, parce que le helper client fait semblant d'être l'objet service. ***Il prétend être l'entité qui détient la méthode que le client veut appeler !***

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.

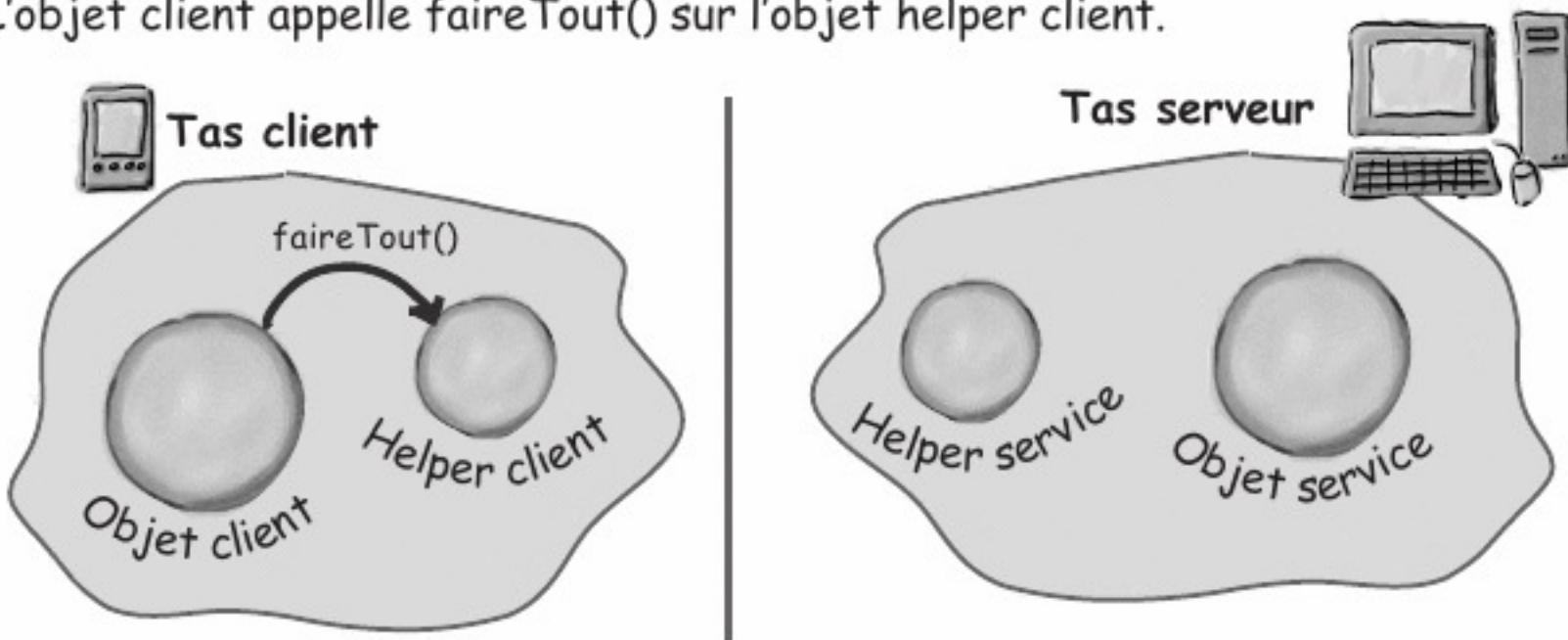
Le helper service reçoit du service la valeur de retour, l'encode et la renvoie (sur le flot de sortie d'une Socket) au helper client. Le helper client décode l'information et retourne la valeur à l'objet client.

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.

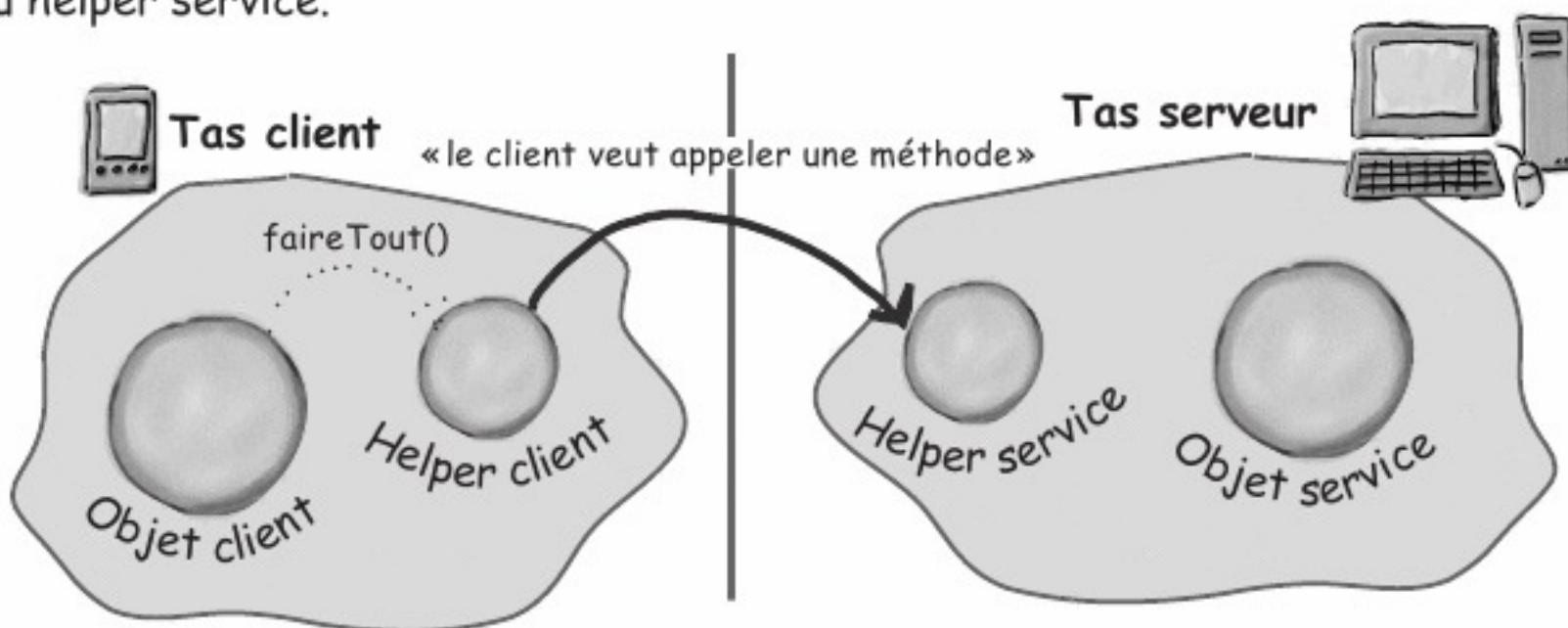


Déroulement de l'appel de méthode

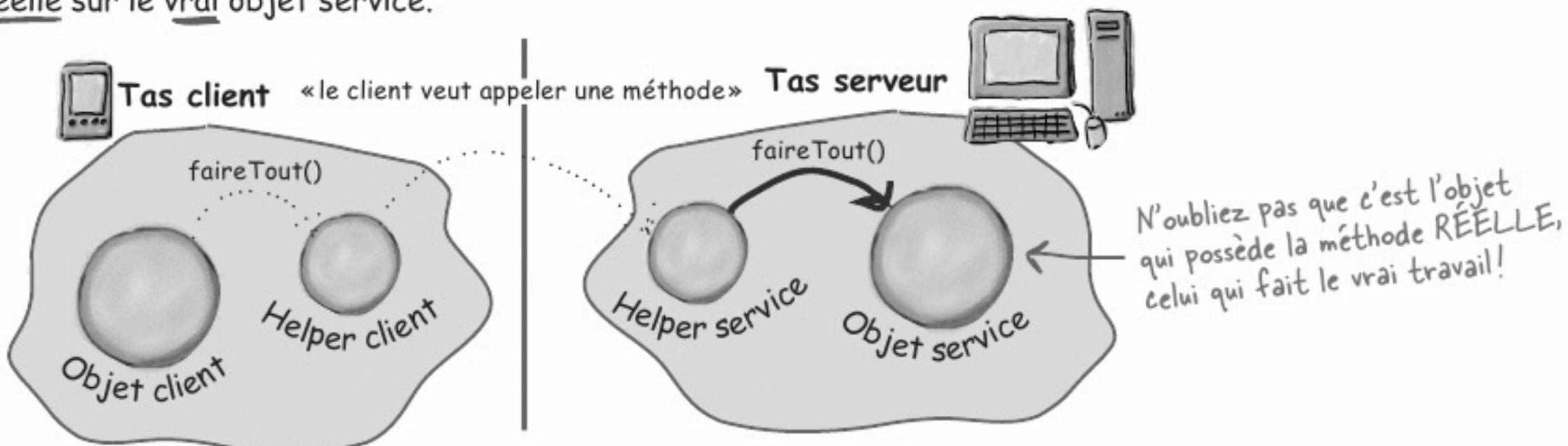
- ① L'objet client appelle faireTout() sur l'objet helper client.



- 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.



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!

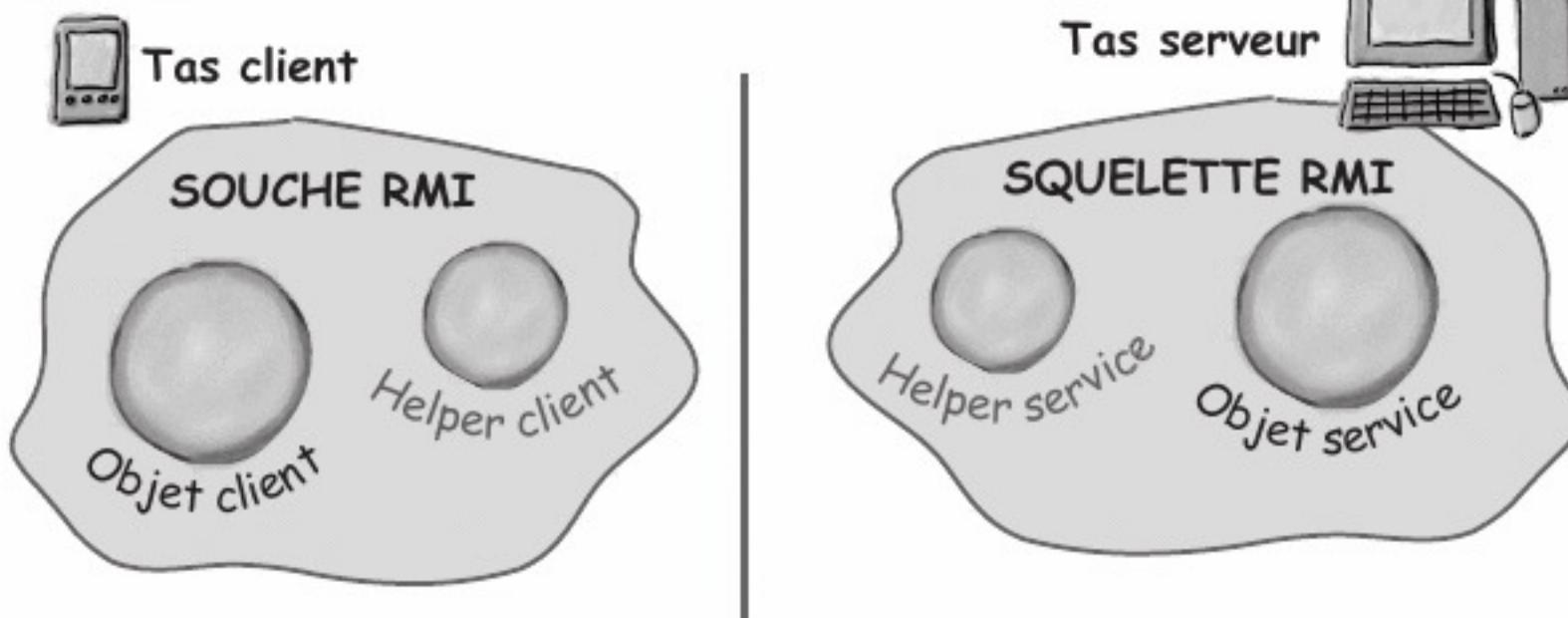
Elles lancent des tonnes d'exceptions.

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 ».



Créer le service distant

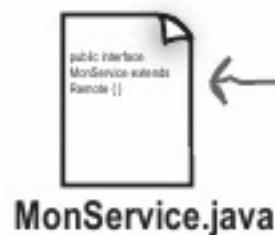
Voici une **vue d'ensemble** des cinq étapes de la création d'un service distant (qui s'exécute sur le serveur). Ne vous inquiétez pas, chacune d'elles est expliquée en détail dans les pages qui suivent.



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...

... génère deux nouvelles classes.

```
Fichier Edition Fenêtre Aide Manger
%rmic MonServiceImpl
```



MonServiceImpl_Stub.class



MonServiceImpl_Skel.class

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
```

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.*; ← L'interface Remote est dans java.rmi
```

```
public interface MonService extends Remote {  
    public String direBonjour() throws RemoteException;  
}
```

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.

Étape deux: créer une implémentation distante

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 implementez. Dans ce cas, il n'y en a qu'une.



MonServiceImpl.java

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.

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.

rmic génère deux nouvelles classes.

```
Fichier Edition Fenêtre Aide Comment?
% rmic MonServiceImpl
```

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

Code complet côté serveur



L'interface distante:

```
import java.rmi.*;      ← RemoteException et Remote
                        sont dans le package java.rmi.

public interface MonService extends Remote { ← Votre interface DOTT
    étendre java.rmi.Remote

    public String direBonjour() throws RemoteException;
}

} ← Toutes vos méthodes distantes doivent déclarer une RemoteException.
```

Le service distant (l'implémentation):

```
import java.rmi.*;      ← UnicastRemoteObject est dans le package java.rmi.server.
import java.rmi.server.*; ← étendre UnicastRemoteObject est la façon
                           la plus facile de créer un objet distant.

public class MonServiceImpl extends UnicastRemoteObject implements MonService { ← Vous DEVEZ implémenter
                                                                           votre interface distante!!

    public String direBonjour() { ← Vous devez bien sûr implémenter
        return "Le serveur dit 'ohé'"; toutes les méthodes de l'interface.
    }                                Mais remarquez que vous n'avez
                                      PAS à déclarer la RemoteException.

    public MonServiceImpl() throws 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).

    public static void main (String[] args) {
        try {
            MonService service = new MonServiceImpl(); ←
            Naming.rebind("Bonjour distant", service);
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }
}
```

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.

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 ».

```
MonService service = (MonService) Naming.lookup("rmi://127.0.0.1/Bonjour distant");
```

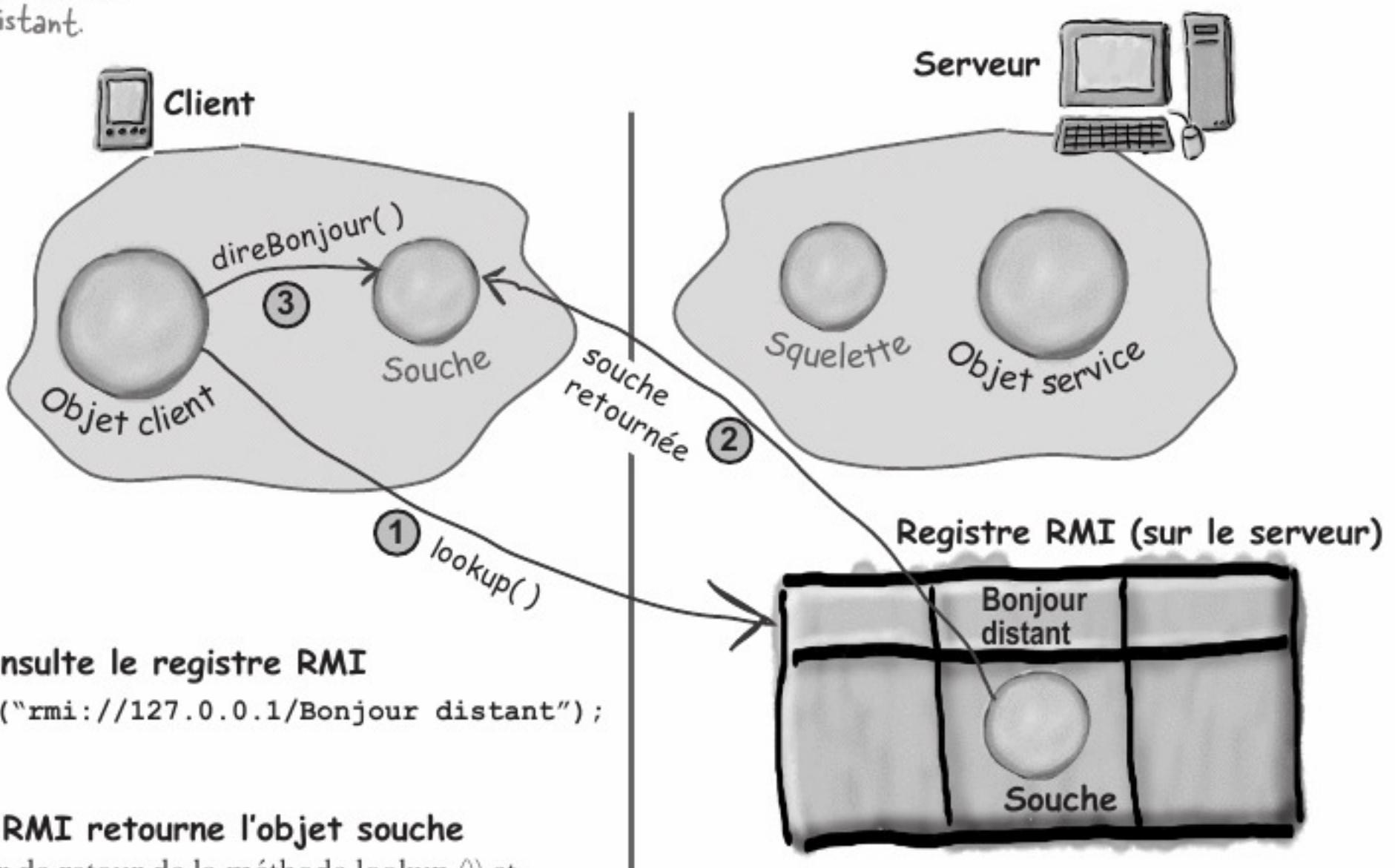
lookup() est une méthode statique de la classe Naming.

Ici le nom sous lequel le service a été enregistré.

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.



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 déserialisée 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 serialisé) 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 déserialisation, 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.*;           ← La classe Naming (pour la recherche dans
                            le registre) est dans le package 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");
            ↑
            ↑
            Vous avez besoin de
            l'adresse IP ou du
            nom de machine.   Et du nom utilisé pour
                            << lien >> le service.

            String s = service.direBonjour();
            System.out.println(s);
        } catch(Exception ex) {
            ex.printStackTrace();
        }
    }
}
```

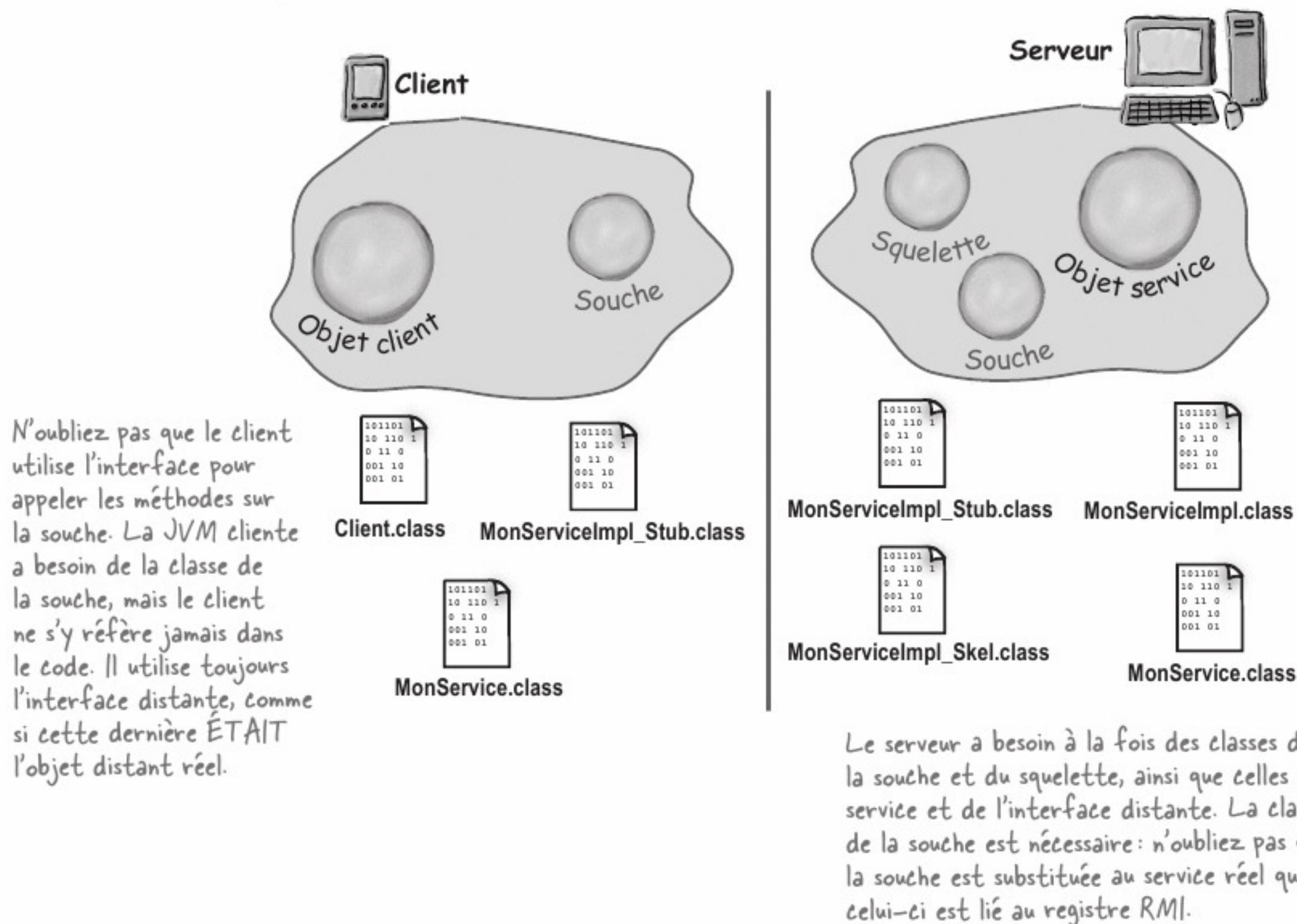
← L'objet retourné est de type Object:
n'oubliez pas le transtypage.

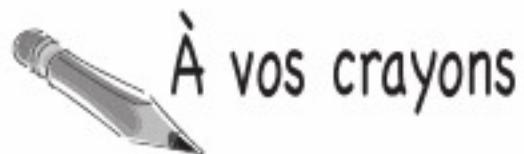
↑ Cela a tout l'air d'un bon
vieil appel de méthode
ordinaire! (Sauf qu'il
doit reconnaître la
RemoteException).

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 rendez pas compte avant l'exécution).
- 3) Oublier de communiquer la classe de la souche au client.

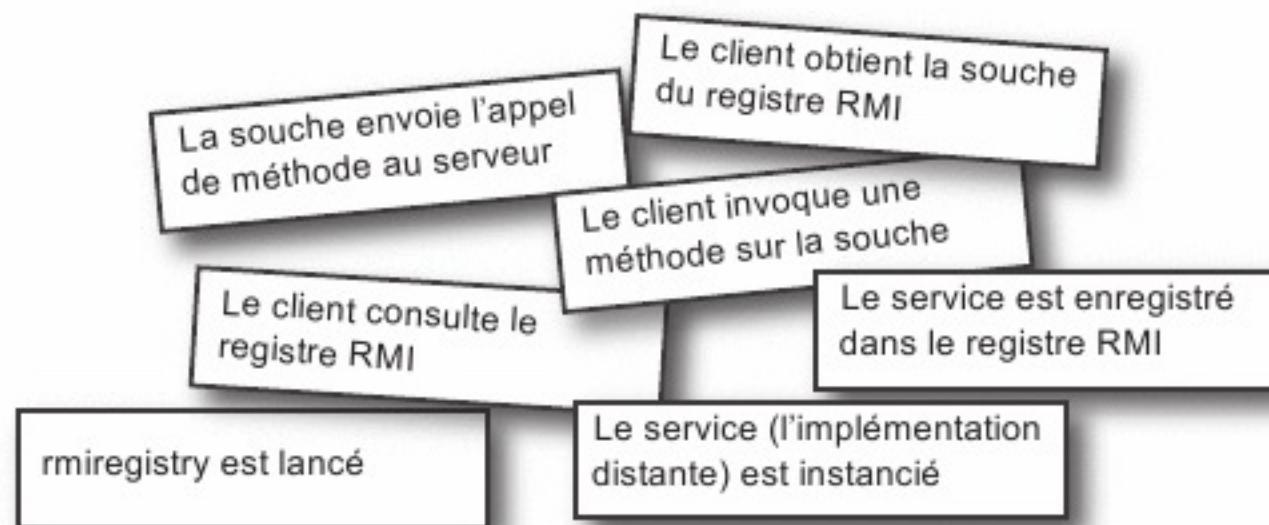




L'œuf et la poule



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



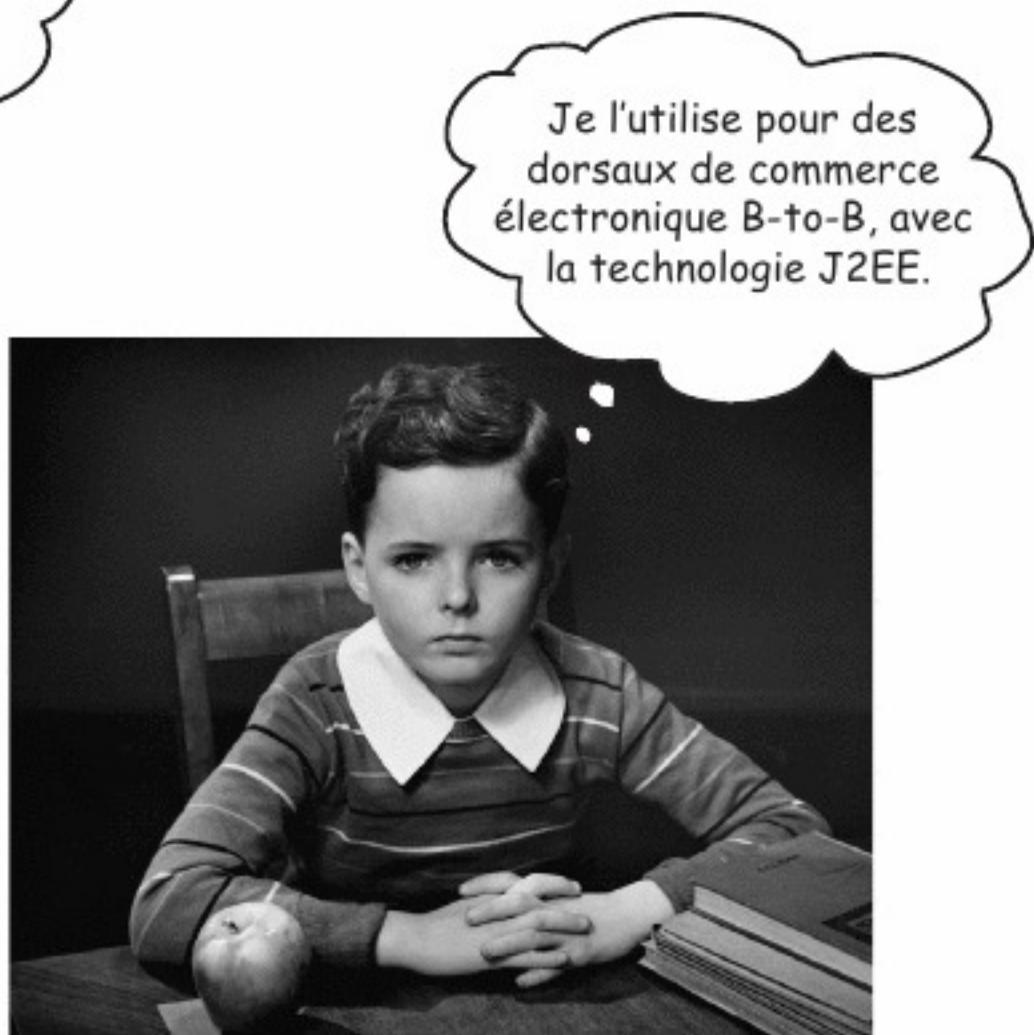
- Un objet résidant sur un tas ne peut pas obtenir une référence normale à un objet résidant sur un tas différent (s'exécutant sur une JVM différente).
- RMI (*Remote Method Invocation*) donne l'impression que vous appelez une méthode sur un objet distant (s'exécutant sur une autre JVM), mais ce n'est 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.

Oui, mais qui utilise réellement RMI?

Nous l'utilisons pour notre nouveau super système d'aide à la décision.



J'ai entendu dire que ton ex continue à utiliser des sockets ordinaires.



Je l'utilise pour des dorsaux de commerce électronique B-to-B, avec la technologie J2EE.

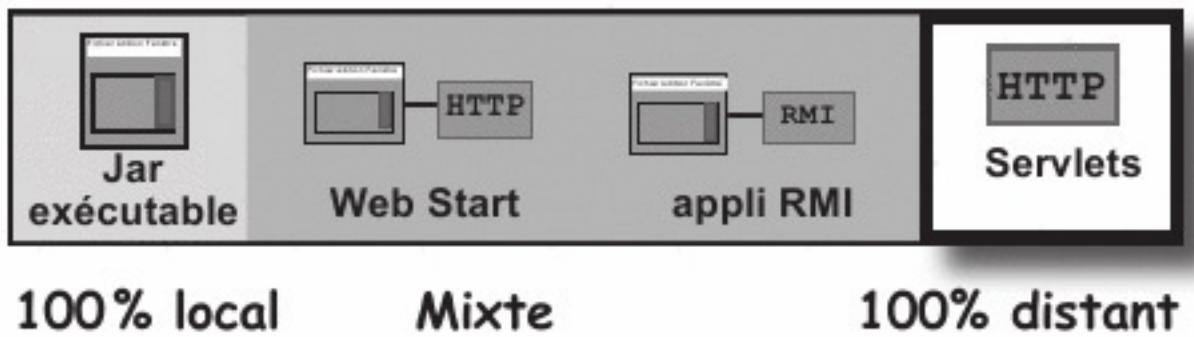
Nous avons un système de réservation hôtelière à base d'EJB. Et EJB utilise RMI!



Je ne peux même pas imaginer la vie sans notre système domotique Jini en réseau!



Moi aussi! Comment peut-on s'en passer? J'adore RMI pour nous avoir donné Jini.



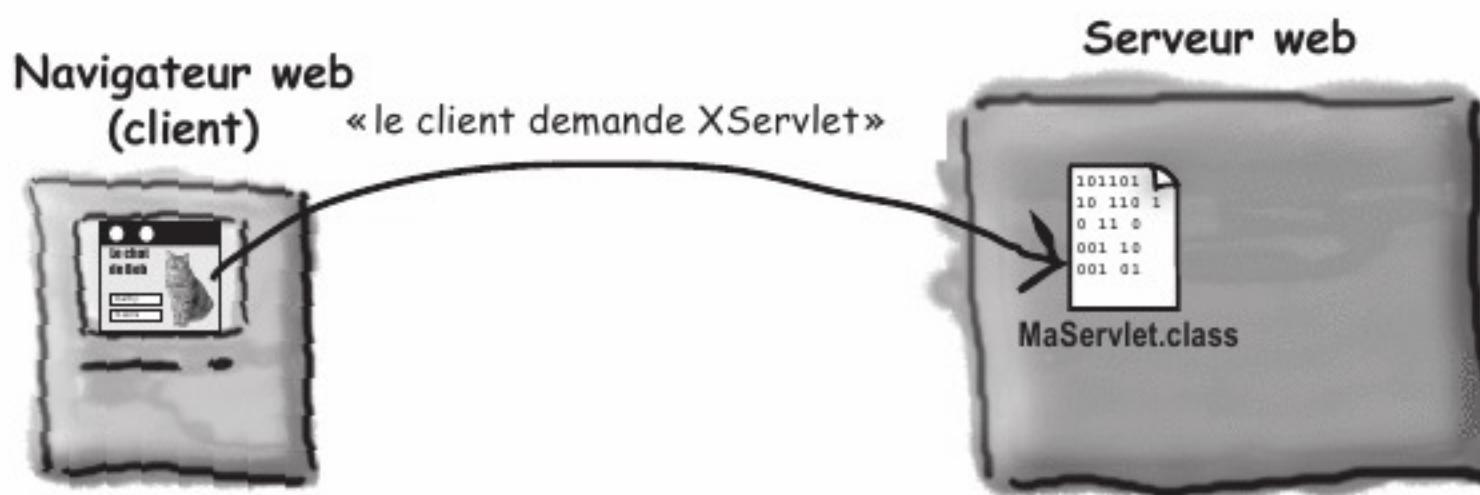
Et les servlets?

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.

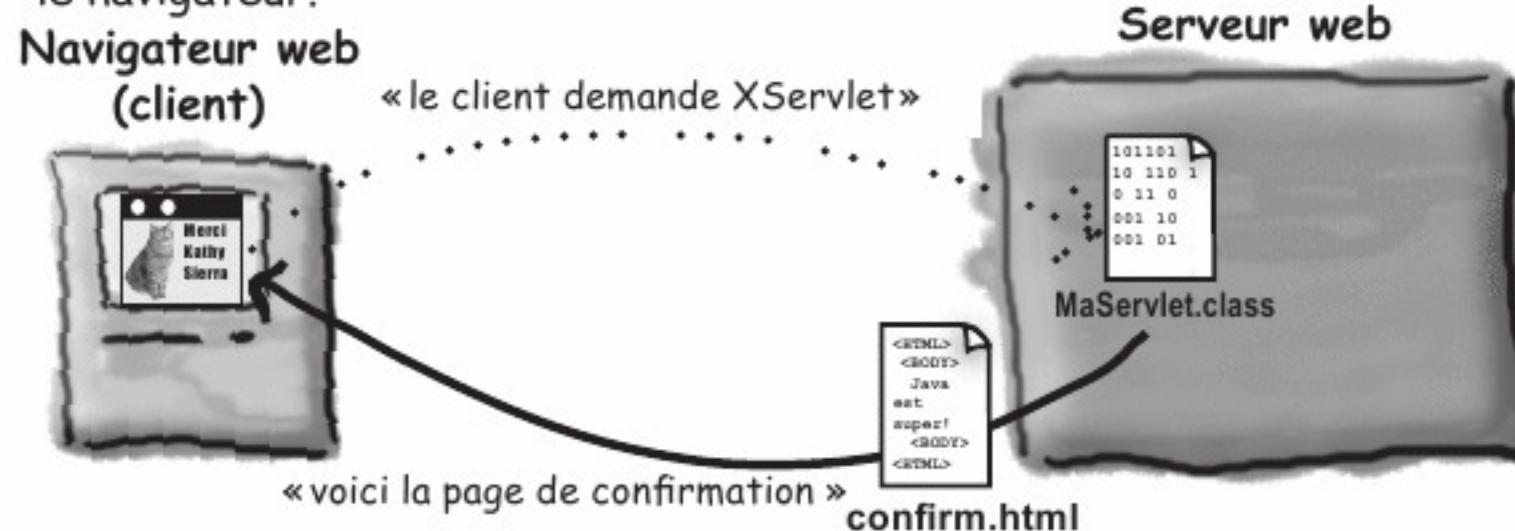
Et même les servlets peuvent utiliser RMI!

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.



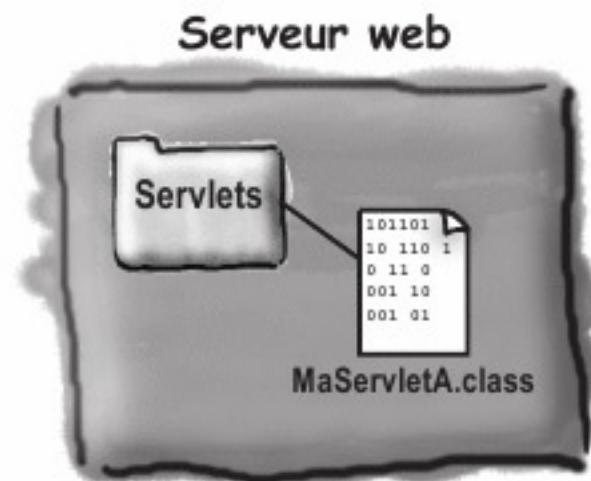
- ② La servlet (le code Java) s'exécute, insère les données dans la base, compose une page web (avec des informations personnalisées) et la retourne au client où elle s'affiche dans le navigateur.



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.



MaServletA.class

```
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.).

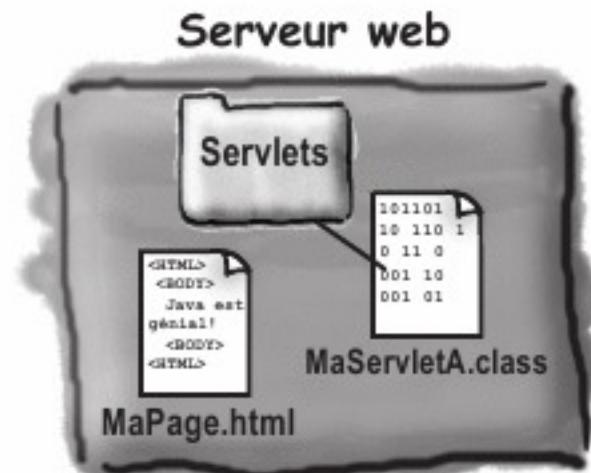


MaPage.html

```
<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.



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

```

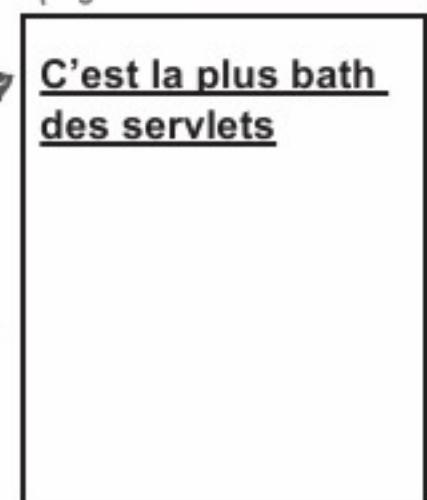
<HTML>
<BODY>
    <a href="servlets/MaServletA">C'est la plus bath des servlets.</a>
</BODY>
</HTML>

```

Cliquez sur le lien pour déclencher la servlet.

C'est la plus bath des servlets

L'aspect de la nouvelle page web:





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` est 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*.

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 servleter (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.)



Essayez
mon nouveau Phrase-
O-Matic amélioré pour
le web et vous serez aussi
beau parleur que votre
patron ou les gars du
marketing.

```
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).

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];

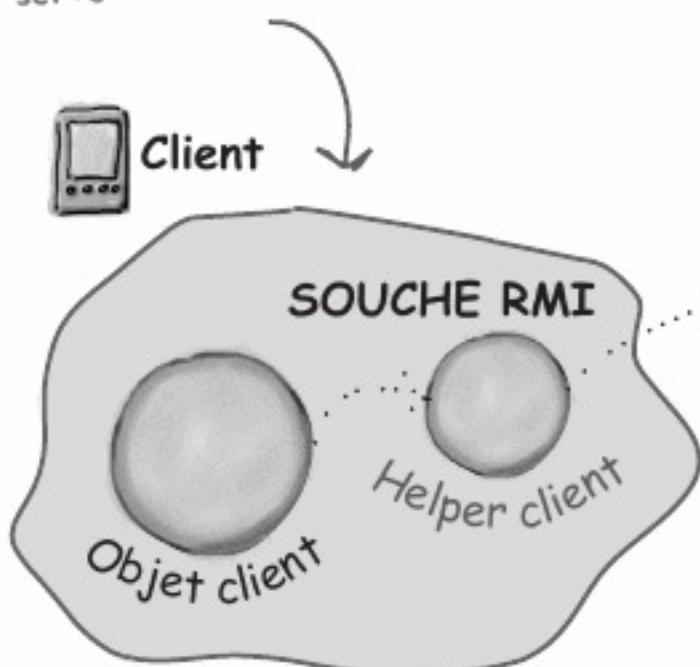
        // et l'afficher
        return ("Il nous faut une " + phrase);
    }
}
```

Enterprise JavaBeans: RMI gonflé aux stéroïdes

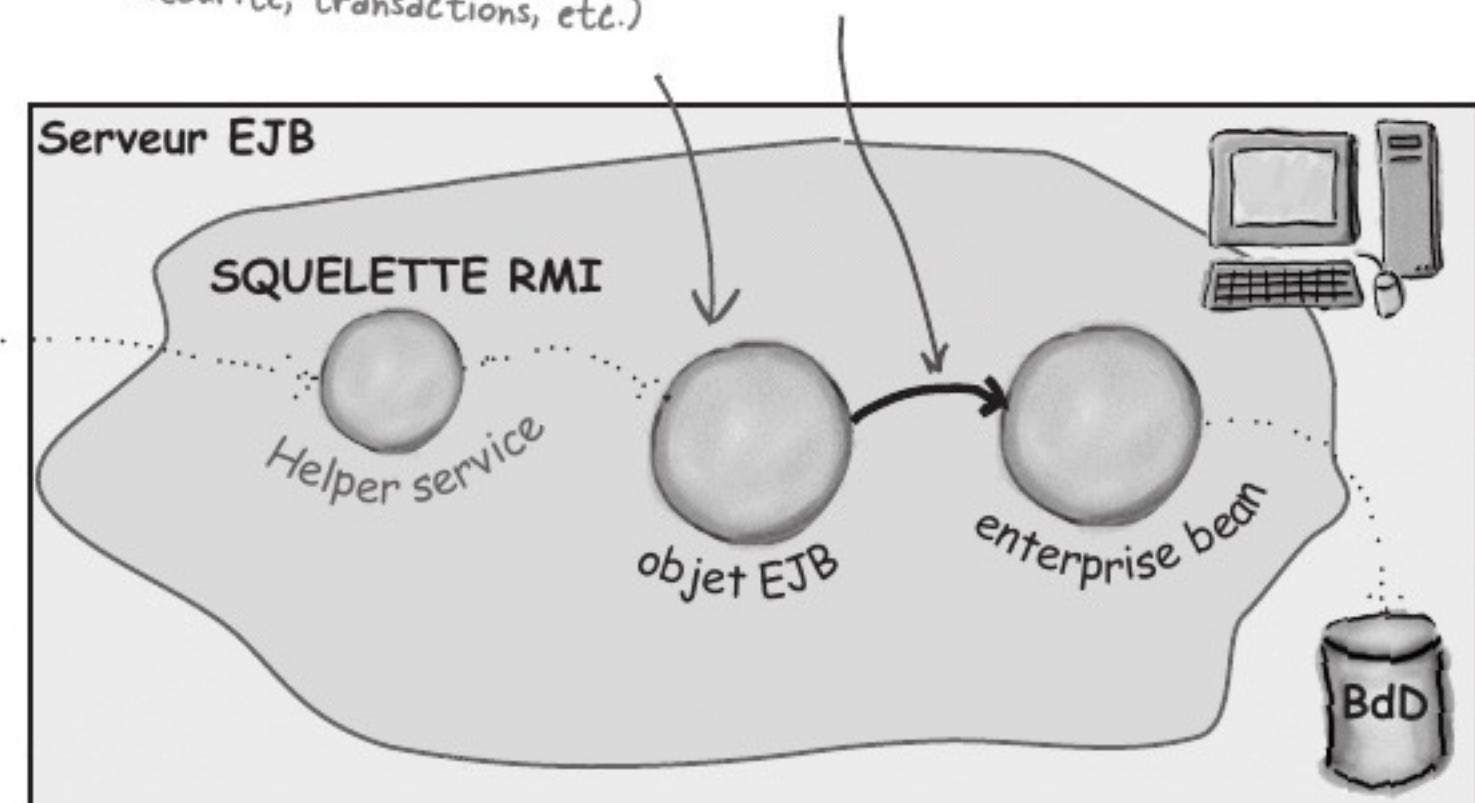
RMI est extra pour écrire et exécuter des services distants. Mais la technologie n'est pas suffisante à elle seule pour des sites du type Amazon ou eBay. Pour une grosse application d'entreprise mortellement sérieuse, il vous faut quelque chose de plus. Quelque chose qui soit capable de gérer les transactions, 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*.)

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 embarque tous les services fournis par le serveur EJB (sécurité, transactions, etc.)



Ce n'est qu'une petite partie du paysage 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.

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.

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 serialisés et même des équipements comme des imprimantes, des caméras et des percolateurs.

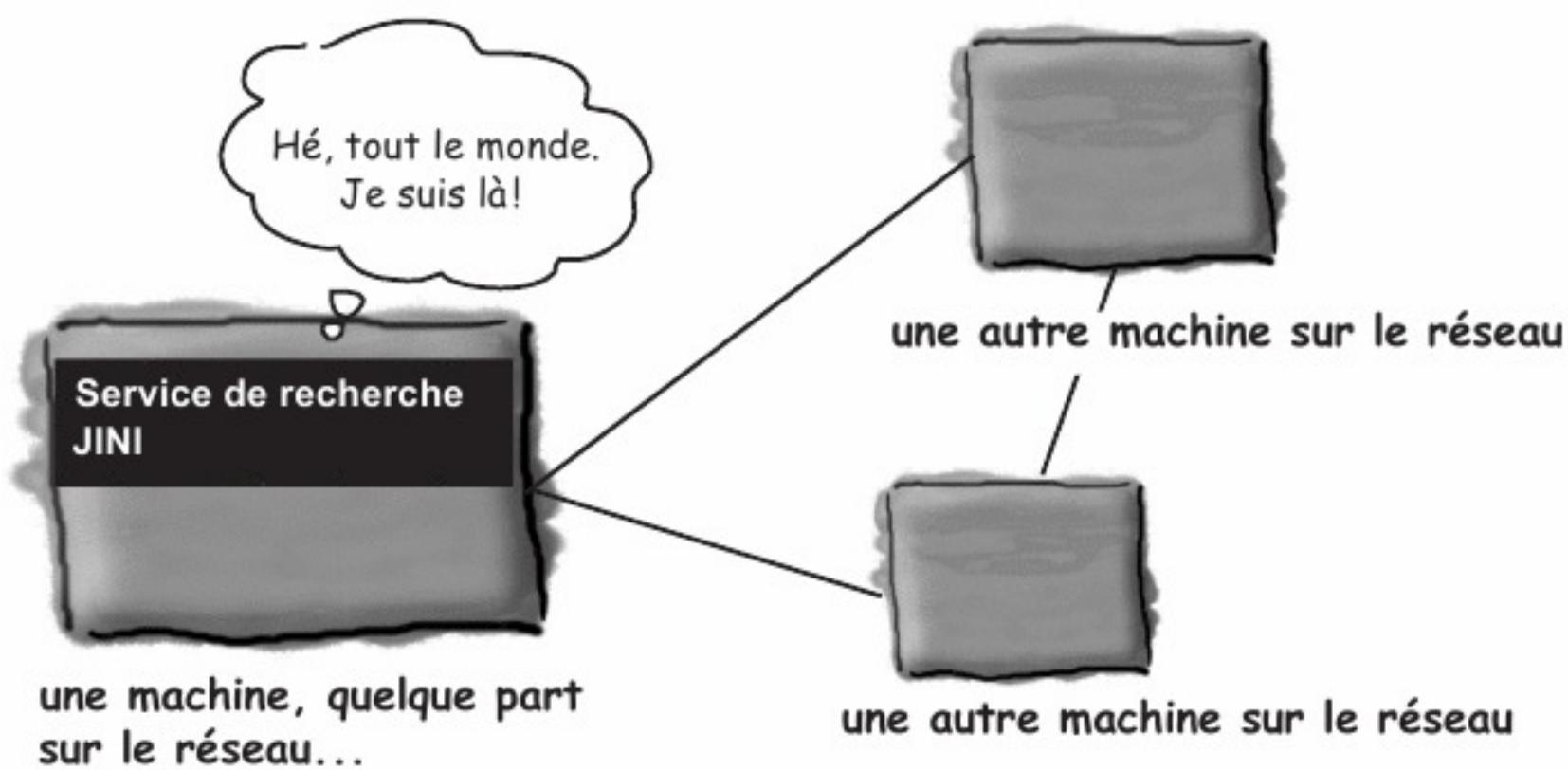
Mais voici le *nec plus ultra* : quand un service arrive en ligne, il découvre dynamiquement tous les services de recherche Jini du réseau et s'enregistre auprès d'eux en envoyant un objet serialisé qui sera placé dans le service de recherche. Cet objet serialisé peut être la souche d'un service RMI 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».

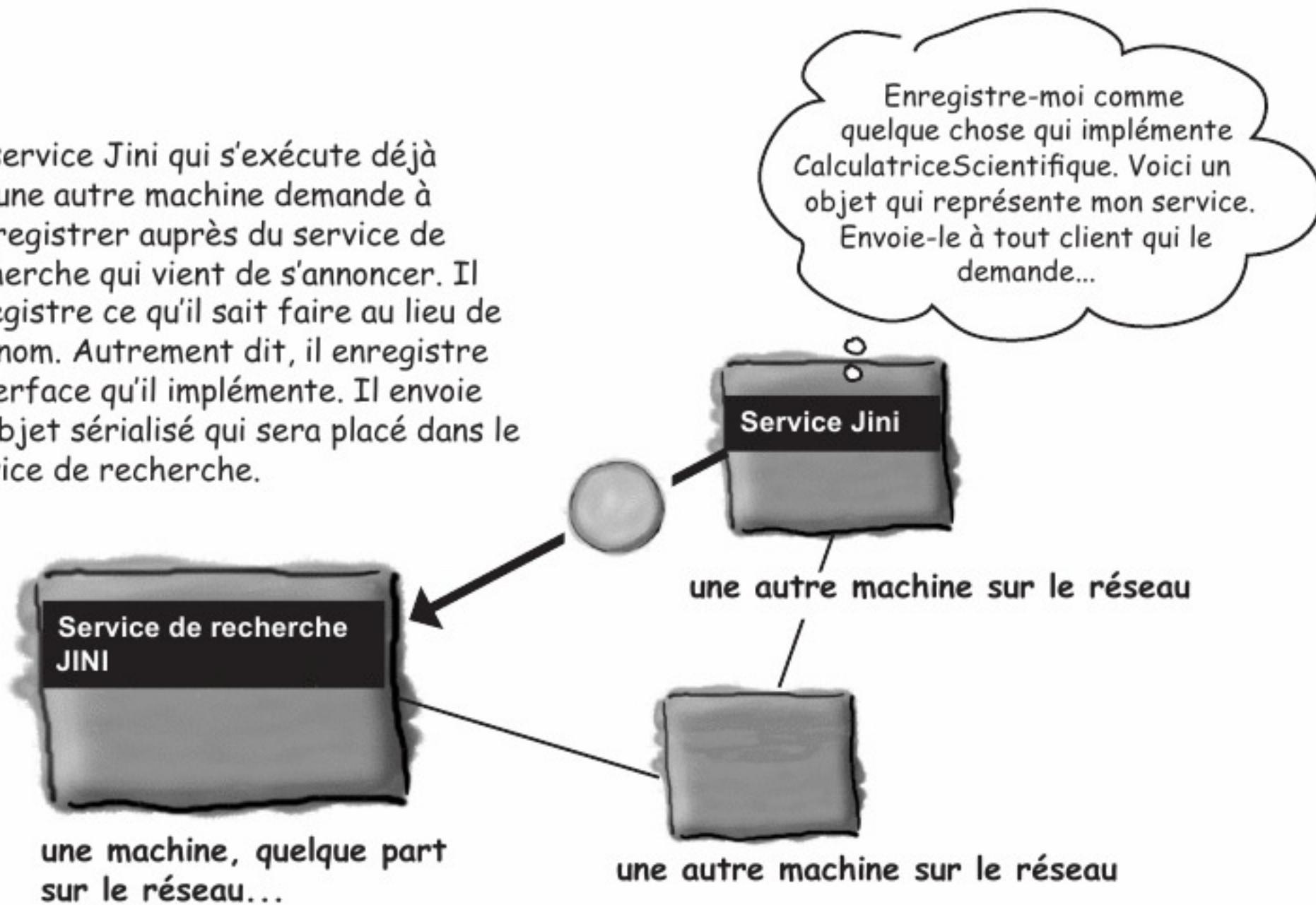


La découverte adaptative en action

- ① Le service de recherche Jini est lancé quelque part sur le réseau et s'annonce en utilisant IP multicast.

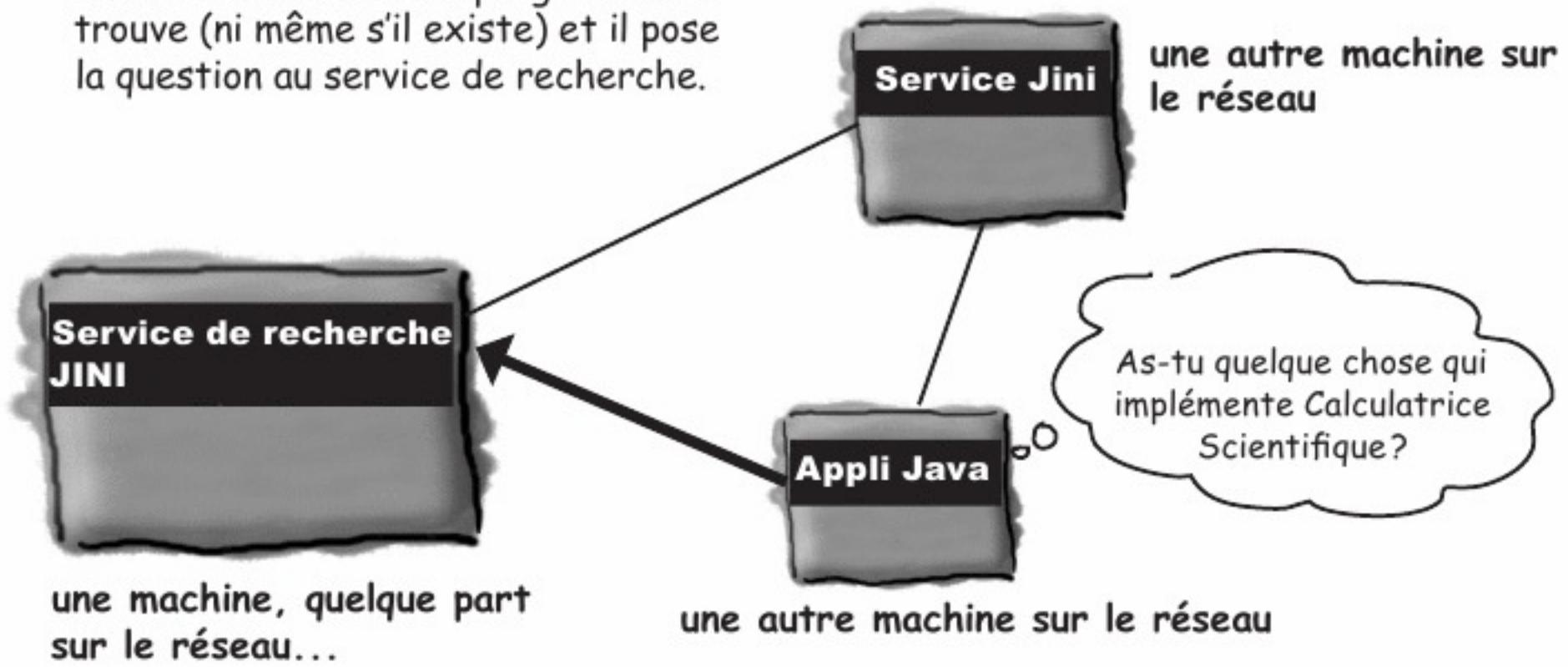


- ② Un service Jini qui s'exécute déjà sur une autre machine demande à s'enregistrer auprès du service de recherche qui vient de s'annoncer. Il enregistre ce qu'il sait faire au lieu de son nom. Autrement dit, il enregistre l'interface qu'il implémente. Il envoie un objet sérialisé qui sera placé dans le service de recherche.

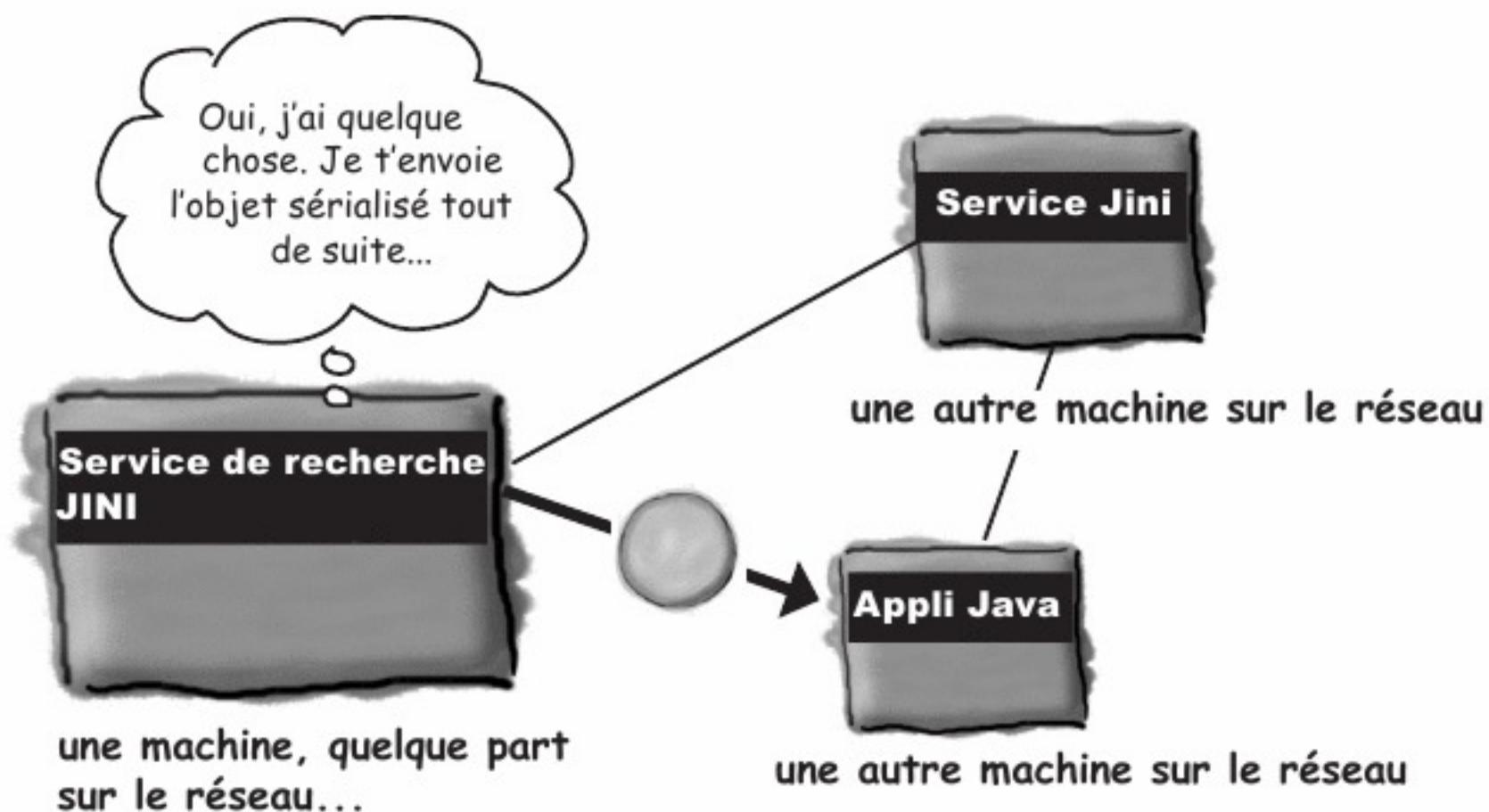


Découverte adaptative en action, suite...

- 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.

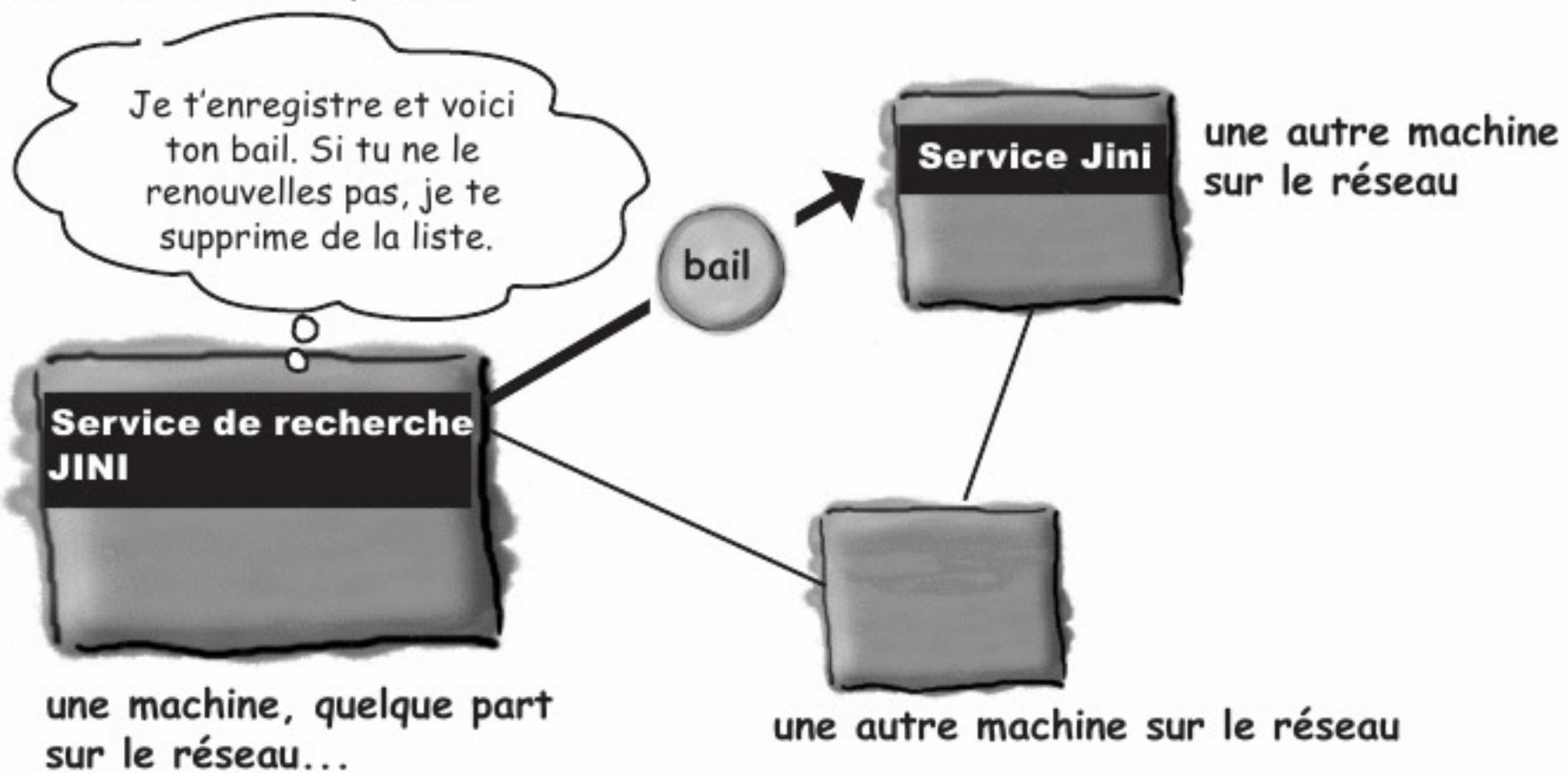


- ④ Le service de recherche répond, puisqu'il a enregistré une Interface `CalculatriceScientifique`.

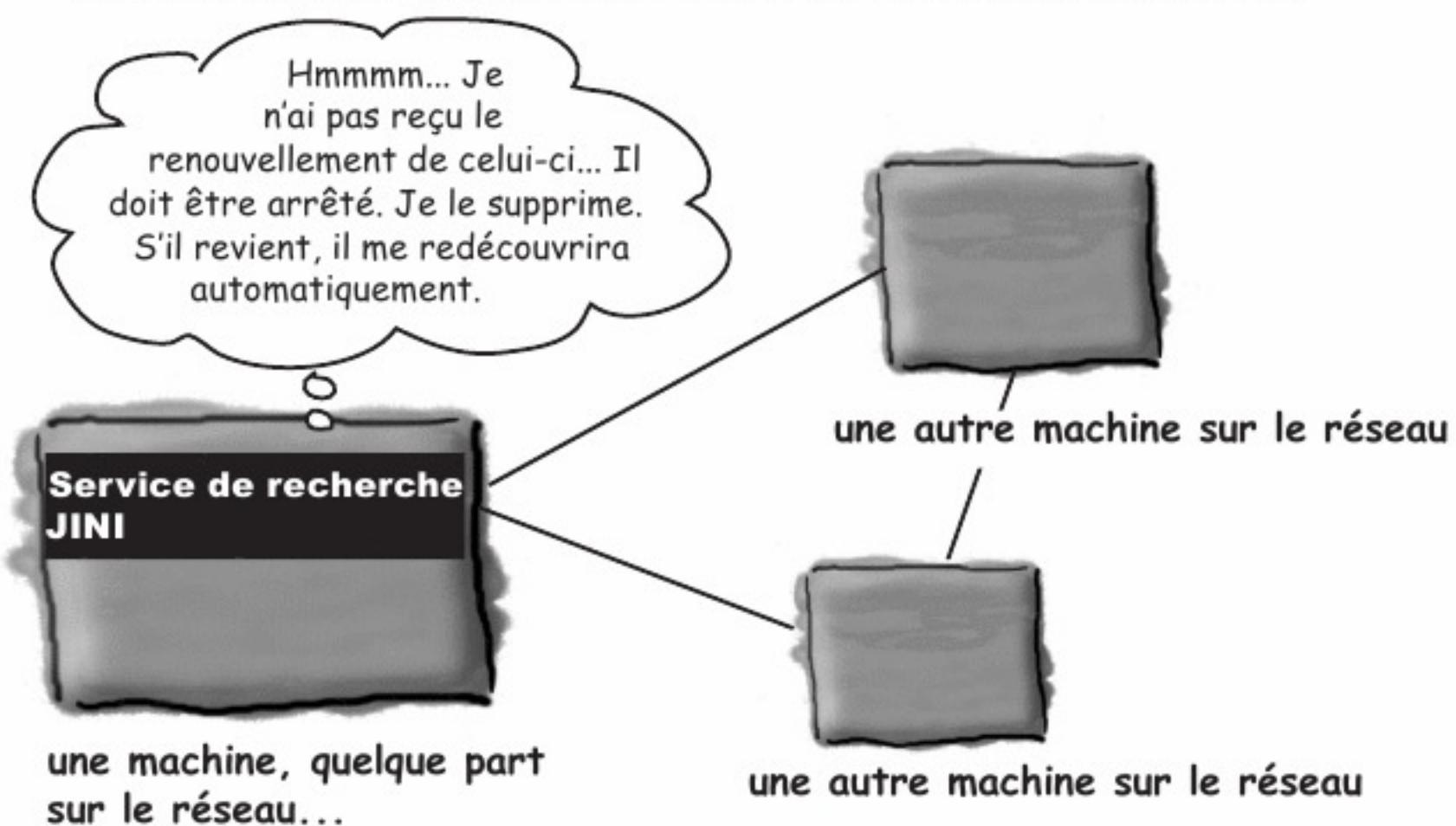


Réseau autoréparateur en action

- 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.



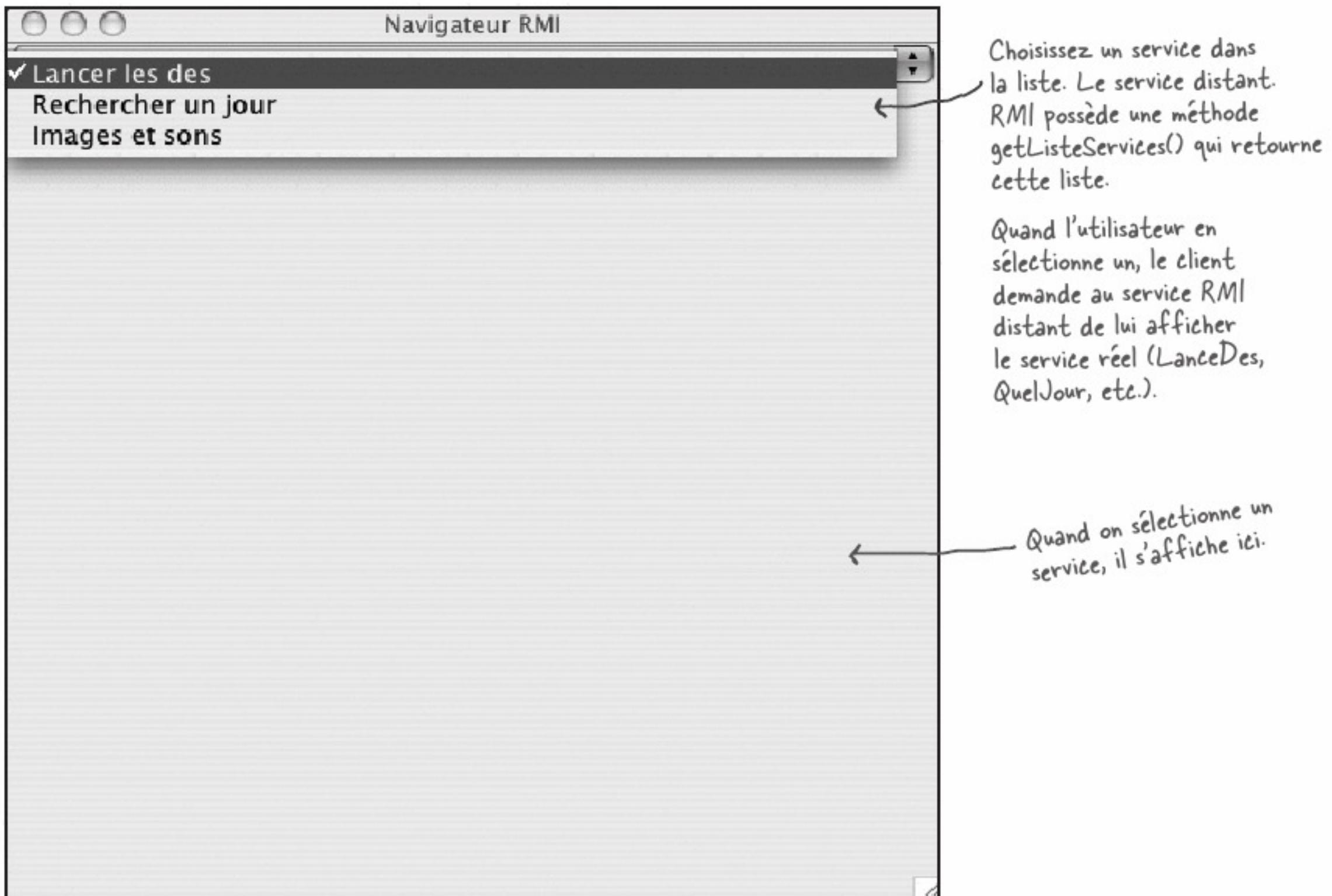
Projet final: le navigateur de services universels

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()`).

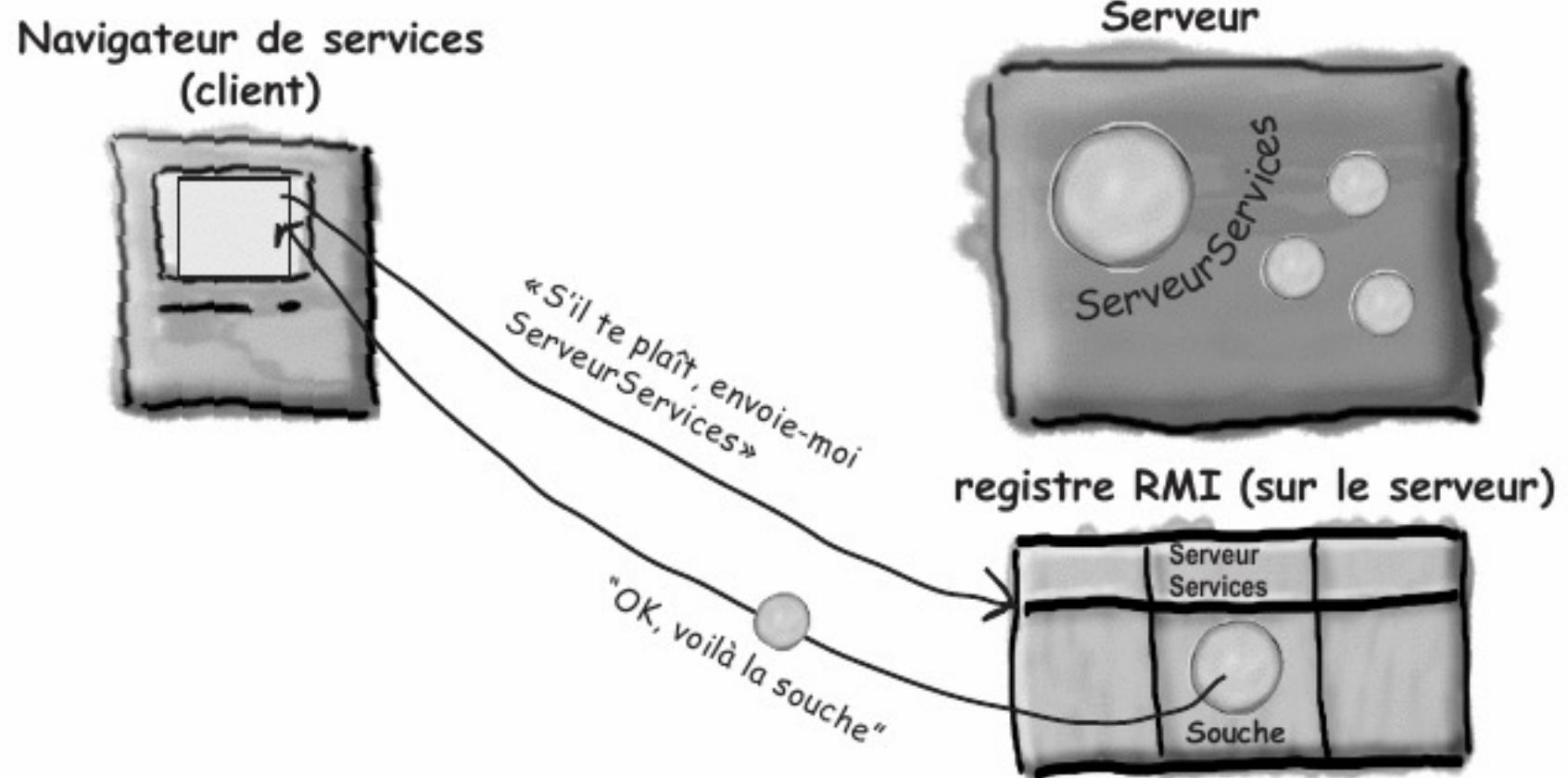
Mais une fois que le client a trouvé le service dans le registre RMI, le reste de l'application est presque identique à la façon dont nous procédons en Jini. (Il ne manque pratiquement que le mécanisme du «bail» qui nous permettrait d'avoir un réseau autoréparateur au cas où l'un des services tomberait.)

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*.

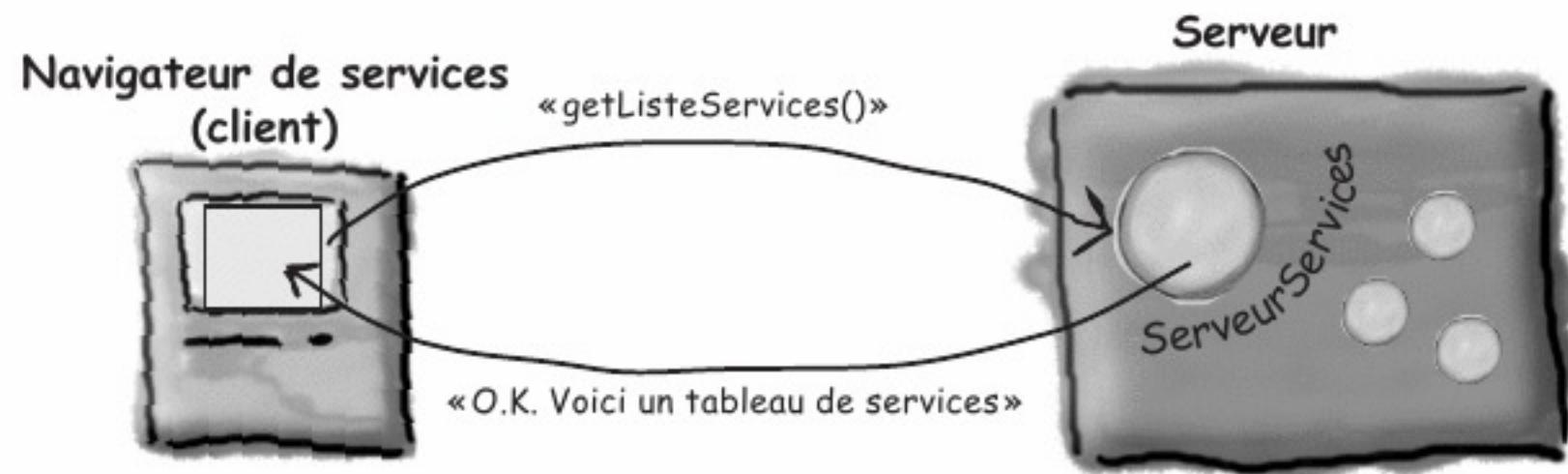


Comment ça marche :

- ① Le client démarre, cherche dans le registre RMI le service nommé ServeurServices et récupère la souche.



- ② Le client appelle getListeServices() sur la souche. ServeurServices retourne un tableau de services.

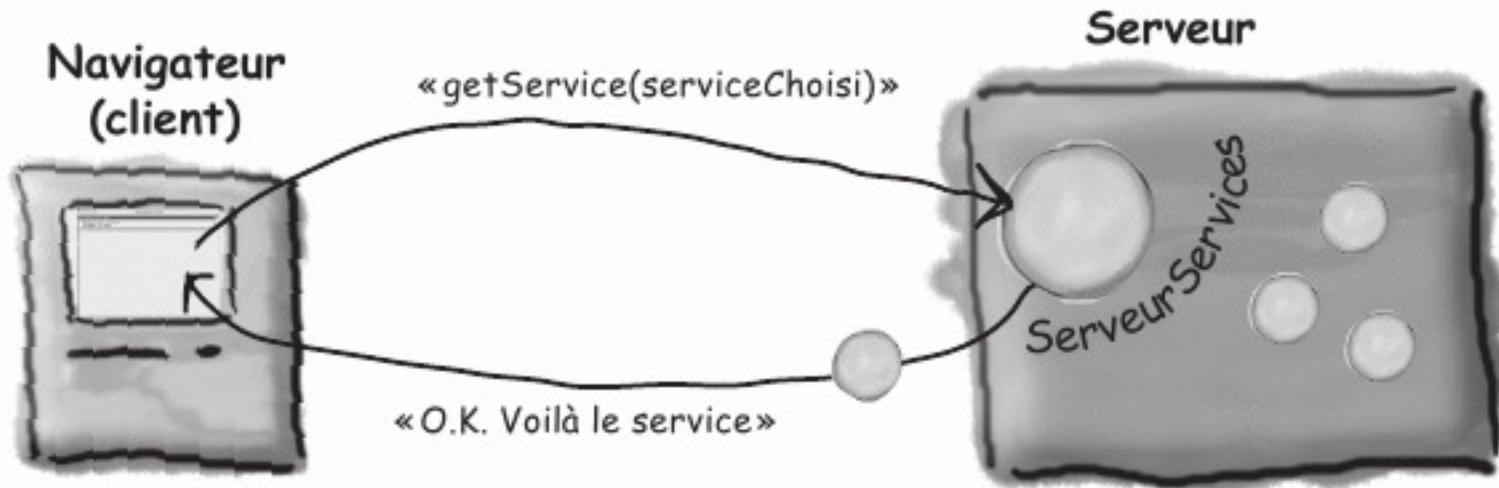


- ③ Le client affiche la liste des services dans une IHM.

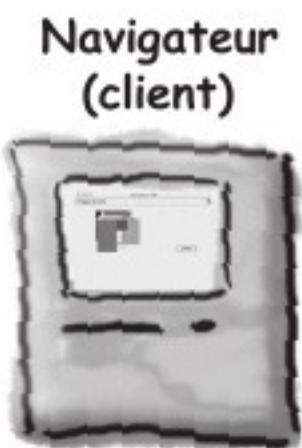


Comment ça marche, suite...

- 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.



Les classes et interfaces :

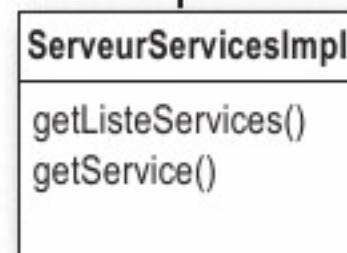
① L'interface ServeurServices implémente Remote

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).



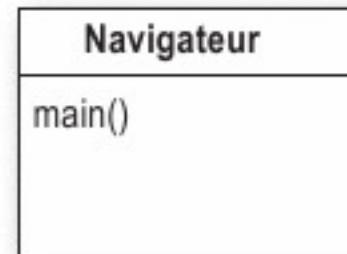
② La classe ServeurServicesImpl implémente ServeurServices

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

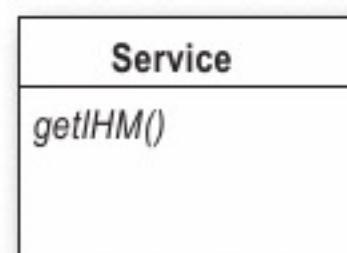
Le client. Elle construit une IHM très simple, effectue une recherche dans le registre RMI pour obtenir la souche de ServeurServices, puis appelle dessus une méthode distante pour extraire la liste de services qui sera affichée dans l'IHM.



④ L'interface Service

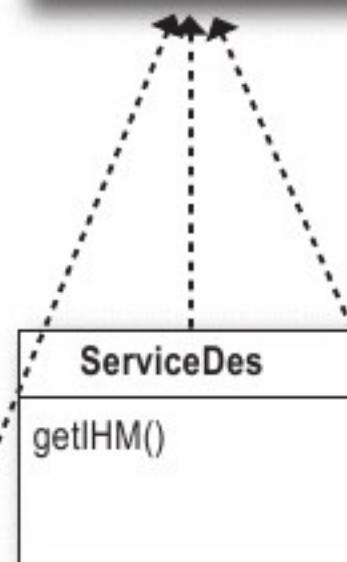
C'est la clé de tout le reste. Cette interface très simple ne possède qu'une seule méthode, getIHM(). Tout service retourné au client doit implémenter cette interface. C'est ce qui rend le programme UNIVERSEL ! En implementant cette interface, un service est accessible même si le client n'a aucune idée de la ou des classes qui constituent ce service. Tout ce que le client sait, c'est que l'entité qu'il reçoit doit implementer l'interface Service et DOIT donc avoir une méthode getIHM().

Le client reçoit un objet serialisé 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 implementes 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 ! »



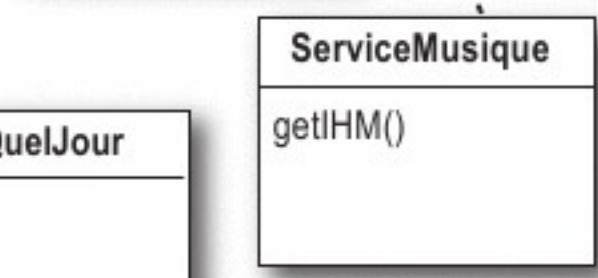
⑤ La classe ServiceDes implémente Service

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.



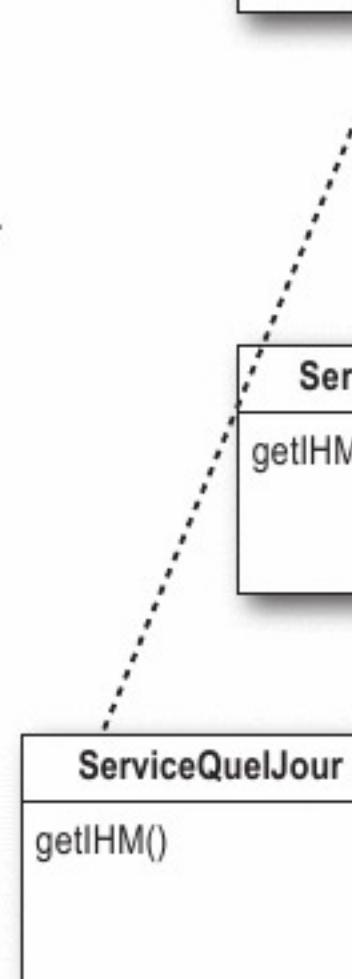
⑥ La classe ServiceMusique implémente Service

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.



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 implementant 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.

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; 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.)

    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");
    }
}

```

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.

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(); ← Cette méthode consulte le registre RMI, obtient
                                                la souche et appelle getListeServices(). (La vraie
                                                méthode se trouve page suivante.)
        listeServices = new JComboBox(services); ← 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.

        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();
        }
    }
}

```

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 MÊME que celle que nous avons originellement reçue du serveur quand nous avons appelé getListeServices()). Le serveur retourne le vrai service (sérialisé) qui est déserialisé automatiquement (grâce à RMI), puis nous appelons simplement getIHM() sur le service et ajoutons le résultat (un JPanel) au panneau du navigateur.

```

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.)

La classe ServiceDes (un service universel, implémente Service)

```

import javax.swing.*;
import java.awt.event.*;
import java.io.*;

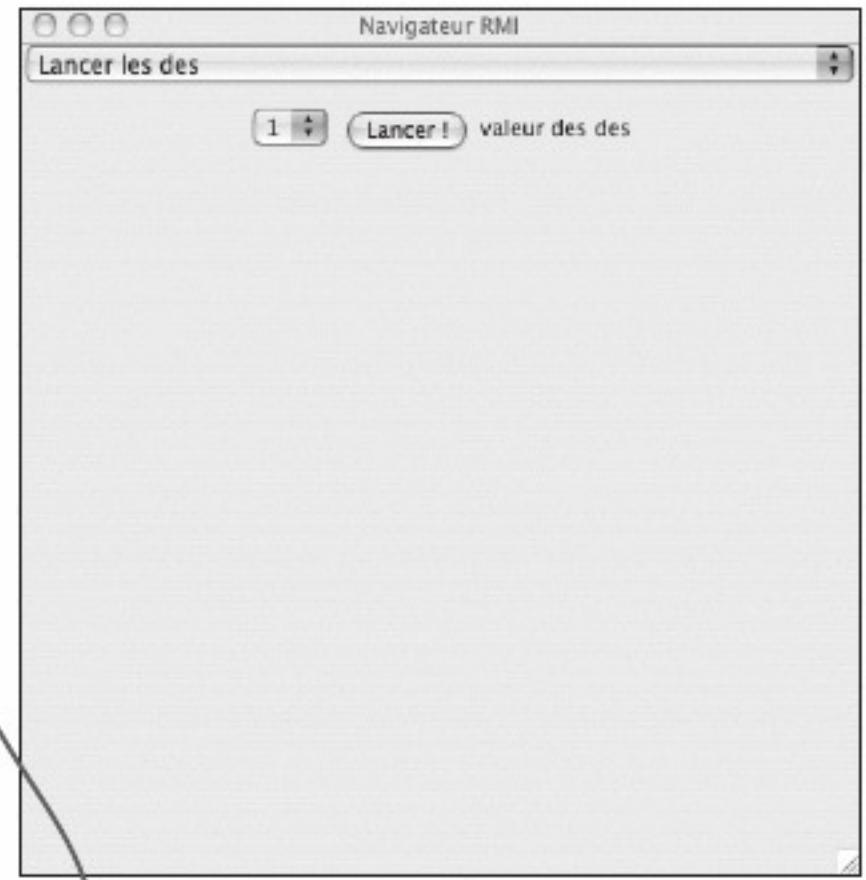
public class ServiceDes implements Service {

    JLabel label;
    JComboBox nbDes;

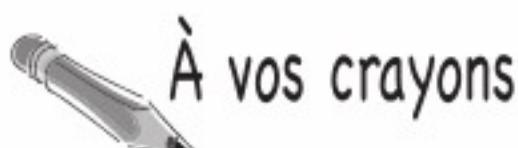
    public JPanel getIHM() {
        JPanel panneau = new JPanel();
        JButton bouton = new JButton("Lancer !");
        String[] choix = {"1", "2", "3", "4", "5"};
        nbDes = new JComboBox(choix);
        label = new JLabel("valeur des dés");
        bouton.addActionListener(new EcouteLancer());
        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);
        }
    }
}

```

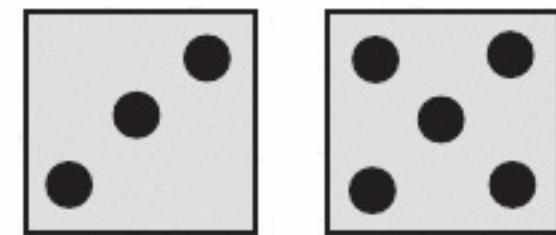


Voici LA méthode importante! La méthode de l'interface Service -- celle que le client va appeler quand ce service sera sélectionné et chargé. Vous pouvez faire tout ce que vous voulez dans la méthode getIHM(), tant que vous retournez un JPanel, pour qu'elle construise l'interface graphique.



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.



La classe ServiceMusique (un service universel, implémente Service)

```

import javax.sound.midi.*;
import java.io.*;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

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) {

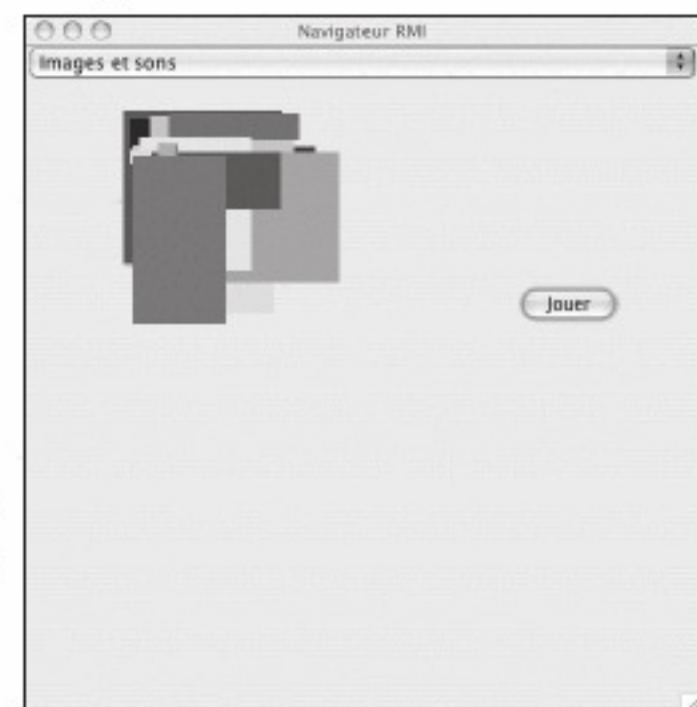
                    int rNum = (int) ((Math.random() * 50) + 1);
                    if (rNum < 38) { // ne faire que si num < 38 (75% du temps)
                        piste.add(makeEvent(144, 1, rNum, 100, i));
                        piste.add(makeEvent(176, 1, 127, 0, i));
                        piste.add(makeEvent(128, 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.



ServiceMusique : le code

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;
    }
}
```

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.

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);
        }
    }
}

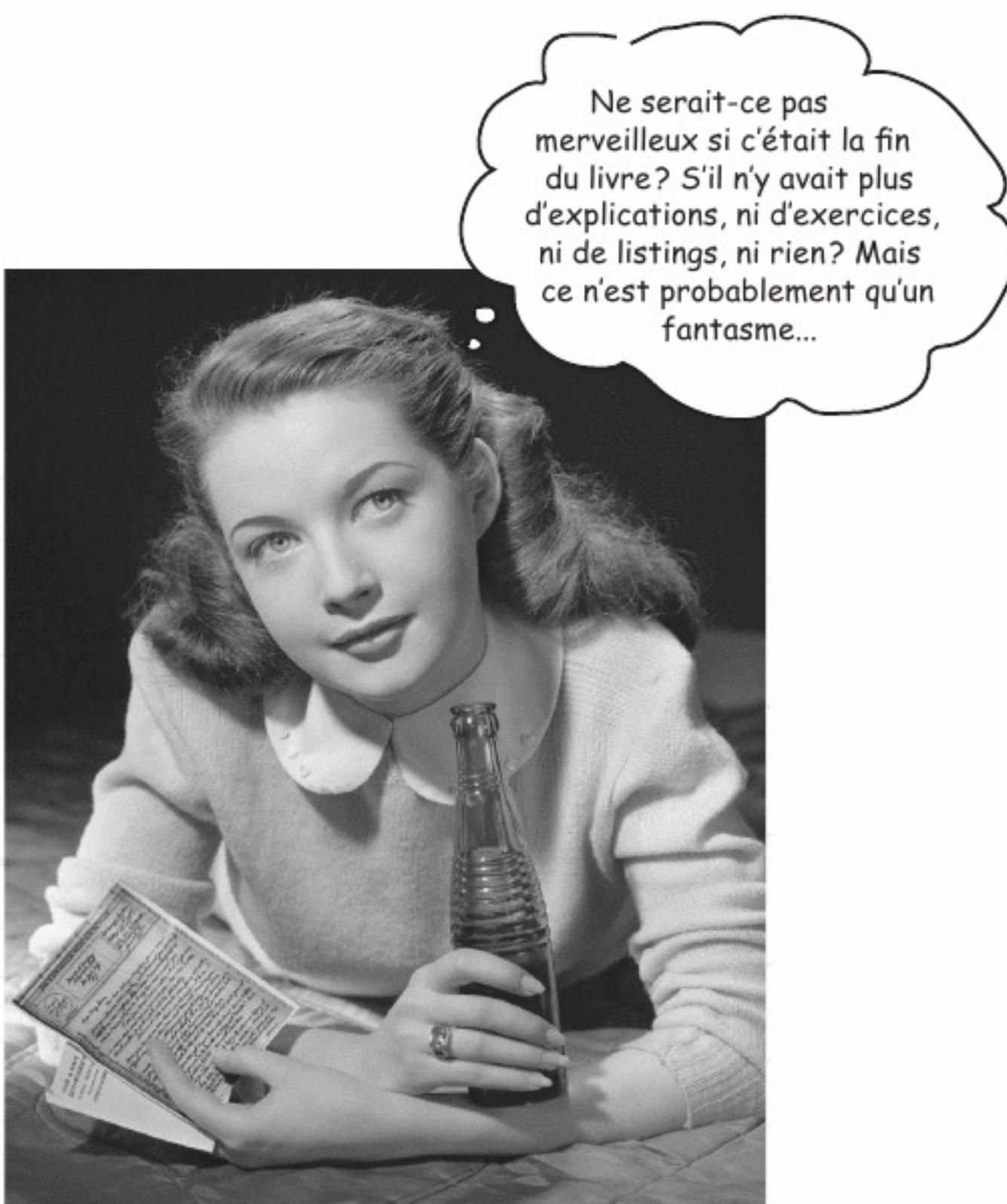
```

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.



la fin... enfin presque



Ne serait-ce pas
merveilleux si c'était la fin
du livre? S'il n'y avait plus
d'explications, ni d'exercices,
ni de listings, ni rien? Mais
ce n'est probablement qu'un
fantasme...

Félicitations! Vous êtes arrivé au bout.

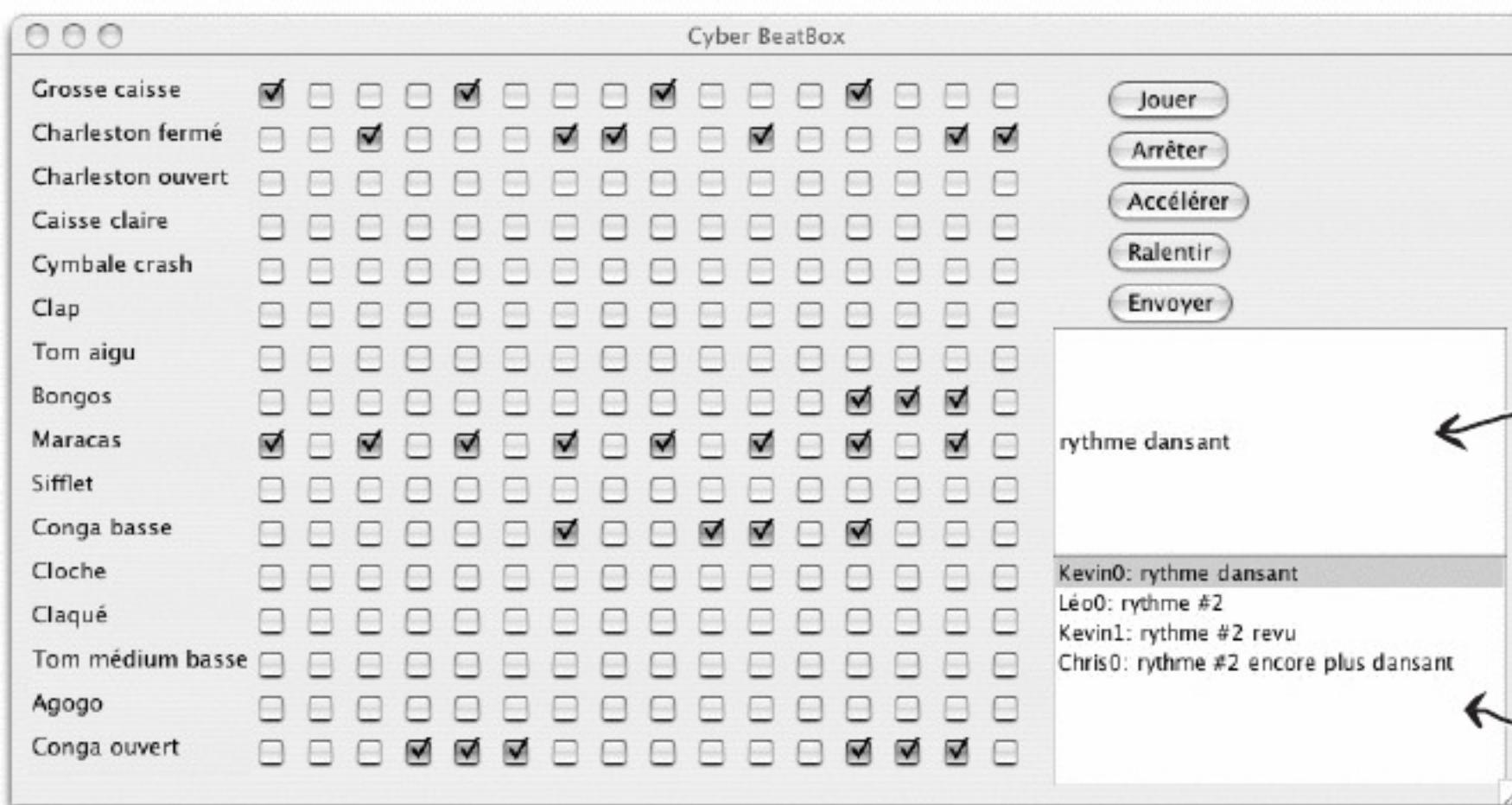
Bien sûr, il reste les deux annexes.

Et l'index.

Et puis il y a le site web.

Il n'y a vraiment aucune issue.

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.

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};
```

```

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 LectureDistant());
        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.

BeatBox: le code complet

```
listeEntrants = new JList();
listeEntrants.addListSelectionListener(new EcouteSelection());
listeEntrants.setSelectionMode(ListSelectionMode.SINGLE_SELECTION);
JScrollPane laListe = new JScrollPane(listeEntrants);
boiteBoutons.add(laListe);
listeEntrants.setListData(listeDonnees); // pas de données pour commencer
```



```
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()
```

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.

Rien d'autre de nouveau dans cette page.

Obtenir le séquenceur, créer une séquence puis créer une piste.

```

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.

BeatBox: le code complet

```
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 flux 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. A 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.

```

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 serialisés: 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.

BeatBox : le code complet

```
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.

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 réseau et les threads. En réalité, la seule différence est que le serveur reçoit puis renvoie deux objets serialisés et non une chaîne de caractères (même si l'un des deux objets serialisés 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
```

BeatBox : le code complet

```
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);

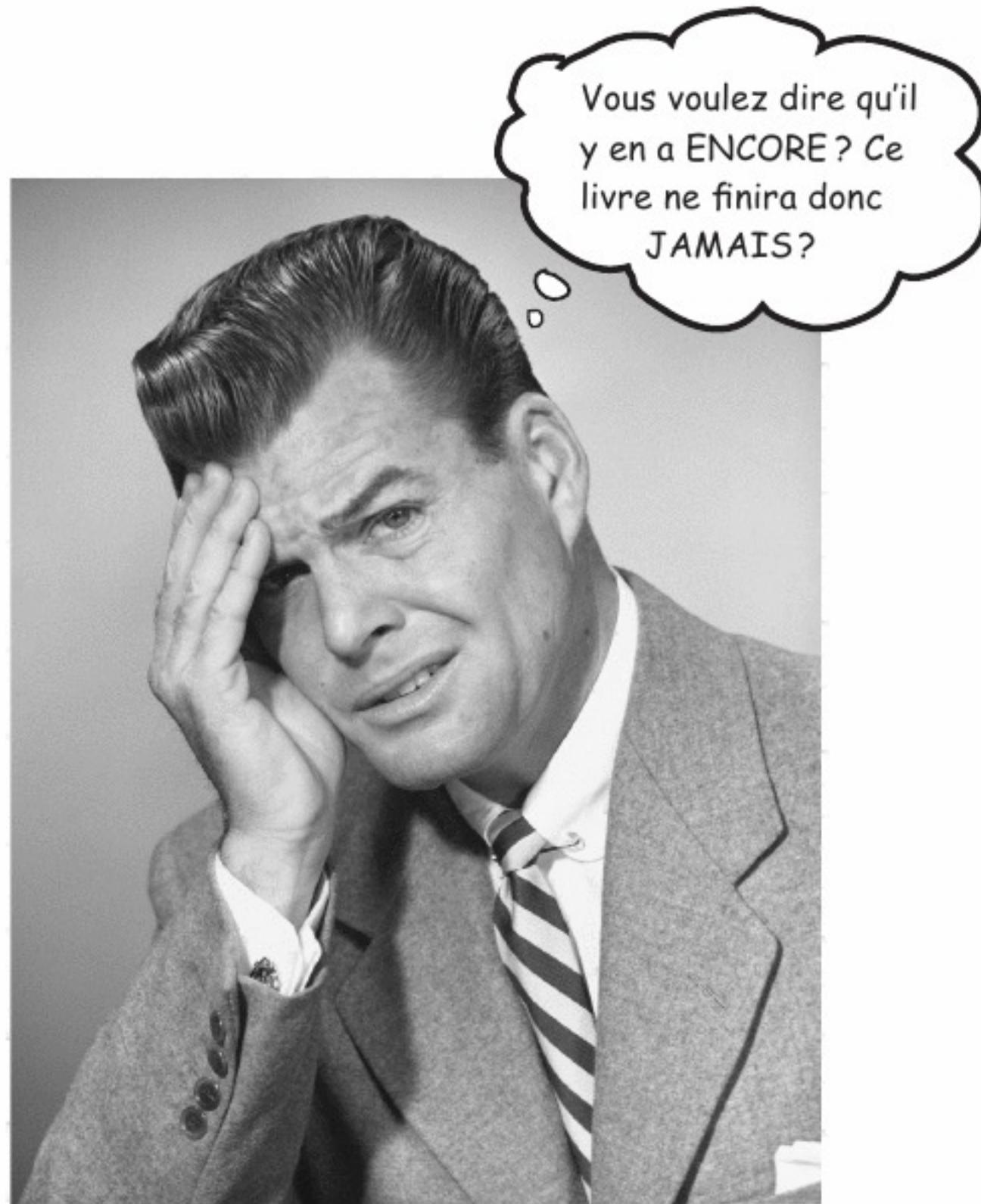
            Thread t = new Thread(new GestionClient(socketClient));
            t.start();
            System.out.println("connexion obtenue");
        }
    }catch(Exception ex) {
        ex.printStackTrace();
    }
} // fin de go()

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
```

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!).

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
```

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 !



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.

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() { ← Début de la méthode.
    int x = 0; ← Variable locale dont la portée est la méthode.
    for(int y = 0; y < 5; y++) { ← Début d'un bloc for : la portée
        x = x + y; ← Pas de problème, x et y sont toutes deux dans la portée.
    } ← Fin de la boucle for.

    x = x * y; ← 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.

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.

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 lié 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).

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 apercevez 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 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, add ActionListener().

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");
        cadre.getContentPane().add(bouton);
        // bouton.addActionListener(quitListener);
    }
}

// bouton.addActionListener(new ActionListener() {
//     public void actionPerformed(ActionEvent ev) {
//         System.exit(0);
//     }
//});
```

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()>>.

Nous avons créé un cadre, ajouté un bouton, et maintenant, nous voulons enregistrer un auditeur d'action auprès du bouton. Sauf que nous n'avons jamais créé 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()).

Mais au lieu de transmettre une référence à un objet, nous mettons... toute la définition de la classe !! Autrement dit, nous écrivons la classe qui implémente ActionListener A L'ENDROIT MEME OÙ NOUS EN AVONS BESOIN. Cette syntaxe crée aussi automatiquement une instance de la classe.

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.

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 concep-tions 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.

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 **StringBuilder** au lieu de **StringBuffer**, 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

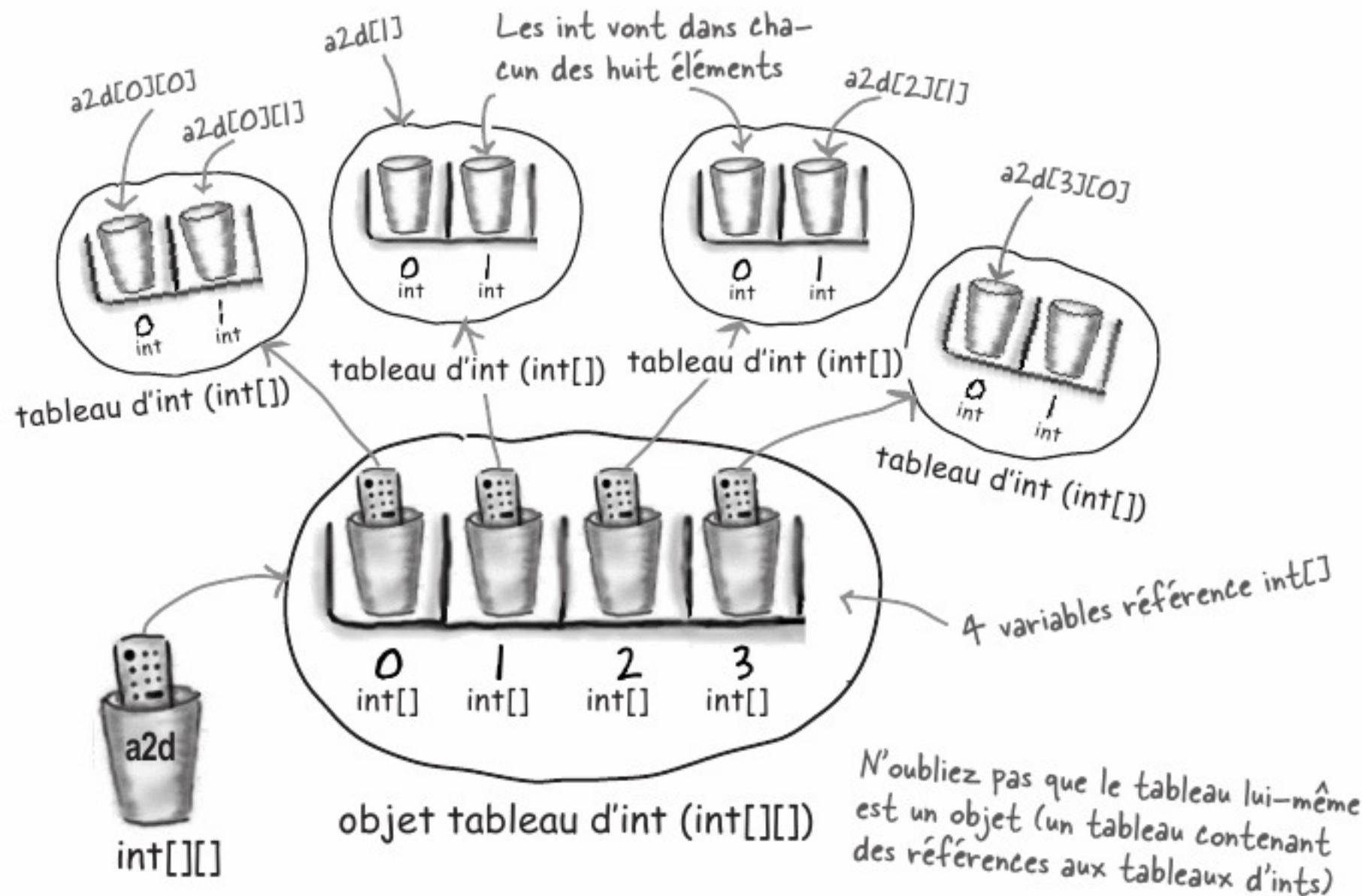
Note : StringBxxxx = StringBuffer ou StringBuilder, selon la classe employée .

2 Tableaux multidimensionnels

Dans la plupart des langages, si vous créez par exemple un tableau bidimensionnel de 4×2 , 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 2×3 : `int[][] x = { { 2,3,4 }, { 7,8,9 } };`
- Créer un tableau bidimensionnel aux dimensions irrégulières:

```
int[][] y = new int [2][];
y[0] = new int [3];
y[1] = new int [5];
```

// crée le premier tableau de 2 éléments
 // crée le premier sous-tableau de 3 éléments
 // crée le deuxième sous-tableau de 5 éléments

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...

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
}

Inutile de se soucier de la valeur de cette variable
```

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.

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; ← Affectation d'une valeur énumérée à une variable.

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 ! ");
}
```

Les deux fonctionnent bien!
«Bravo» s'affiche.

Pop Quiz! Quel est le résultat?

Réponse: va plus loin encore!

#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'occasion";
    }
}

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() est 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'occasion
%
```

Remarquez que la méthode chante() de base n'est appelée que lorsque une valeur n'a pas de corps de classe spécifique aux constantes.



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 calculateRoute() 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 ?



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.

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

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
BoxLayout 411
break (instruction) 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 paramétrés 137
 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

- 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 désrialisation 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 désrialisation 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

l'index

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

fichiers
 classe File 452
 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

finals
 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

F

fiabilité de type 540
 et génériques 540

G

génériques 540, 542, 568–574
 méthodes 544
 jokers 574

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

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

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

equals() 209, 561
 ramasse-miettes 260–263
 tableaux 59, 60, 83
 vie et mort 258–263
 verrou 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

O

Object (classe)
 définition 208–216
 equals() 561
 hashCode() 561
 ObjectOutputStream 432, 437
 objets
 comparaison 209
 création 55, 240–256
 définition 55
 égalité 560

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

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

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
 déserialisation 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

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

transtypepage
 explicite (primitifs) 117
 explicite (référence) 216
 implicite (primitifs) 117
 transtypepage 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

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 fantôme 109

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

Ceci n'est pas un
au revoir

Visitez le site

digitbooks.fr/catalogue/9782815000079.html

Vous ne connaissez pas le site
web? Vous y trouverez notamment
le code du livre. Dites bonjour aux
auteurs pour nous.

