Chapitre 4

Comment programmer

Nous venons de voir dans le chapitre précédent un ensemble d'instructions : l'affectation, la conditionnelle, le boucle for, la boucle while. S'il est relativement aisé de comprendre un code Java à ce point du cours, il reste une grosse difficulté : concevoir un programme. Quelles variables faut-il définir ? Quelles instructions faut-il employer ? Comment les assembler ?

Dans ce chapitre, nous vous présentons une démarche pour répondre à ces questions et vous aider à concevoir un programme.

4.1 Construction du programme

La première chose à faire quand on est confronté à un nouveau problème est d'essayer de voir s'il ne ressemblerait pas par hasard à un autre problème que nous avons déjà résolu. Dans ce cas, il ne reste plus qu'à reproduire l'algorithme en l'adaptant.

Le plus difficile est quand nous sommes confrontés à un problème pour la première fois. De plus, vous êtes en grande majorité en cours d'acquisition de la logique de programmation qui n'est pas forcément naturelle. Il faut alors utiliser une méthode pour arriver à concevoir un nouveau programme.

La méthode présentée ci-dessous peut paraître longue et fastidieuse, et il est très tentant de brûler les étapes... Foncer tête baissée dans votre éditeur de texte ou votre IDE préféré pour écrire un nouveau programme tant que vous n'êtes pas suffisamment aguerri à la programmation, vous mènera bien souvent à l'échec. Des heures à tourner en rond et à ne pas savoir par quel bout prendre le problème. Appliquer la méthode avec sérieux et soin, vous mènera beaucoup plus certainement à une solution. Après, au fil de votre acquisition d'expérience et au fur et à mesure que votre bibliothèque de solutions s'enrichira, vous pourrez de plus en plus souvent coder directement en Java. Mais pour l'instant, c'est hors de question à quelques rares exceptions près.

Étape 1 : Entrées et sorties

La première chose à faire consiste à déterminer les données nécessaires à la réalisation du calcul et le ou les résultats attendus. Pour ce faire, il faut lire attentivement le sujet. A chaque donnée et chaque résultat il faudra lui donner un nom et fixer le type de sa valeur.

Exemple

Nous allons utiliser tout au long de ce chapitre un exemple. Il est volontairement choisi un peu complexe par rapport à notre niveau actuel pour, d'une part, couvrir l'ensemble de la démarche et,

d'autre part, vous montrer qu'une approche méthodique permet de résoudre des problèmes assez complexes.

Son énoncé est le suivant :

Problème : *Écrire un programme simulant un virus informatique. Ce programme demandera à l'utilisateur un entier : le nombre de fois qu'il devra écrire un message à l'écran. Chacun de ces messages sera pris au hasard dans une liste que vous établirez. Sur la même ligne que le message et à sa suite, pour simuler la visualisation de l'avancement de l'opération en cours, le programme affichera à l'écran un nouveau point toutes les demi-secondes entre 5 et 15 fois. Le nombre de fois qu'un point affiché sera aussi pris au hasard. Ensuite, il affichera les texte* `"Fait"` *avant de passer à la ligne suivante et afficher le message suivant.*

Pour vous aider, tout d'abord, l'instruction suivante permet de tirer un nombre au hasard compris entre `p` *et* `n-1+p`*, bornes comprises, et de le mettre dans la variable* `alea` *supposée de type* `int` *:*

```
alea = (int) (n *Math.random ()) + p;
```

Bien entendu, vous pouvez remplacer `n` *et* `p` *par n'importe quelle valeur selon vos besoins.*

Explications : `Math.random ()` *est une méthode faisant partie de l'API Java. Elle retourne un* `double` *aléatoire entre 0, borne comprise, et 1, borne non comprise. En multipliant son résultat par* `n` *on obtient un* `double` *dans l'intervalle* `0..n`*, 0 compris et* `n` *non compris. On convertit ce résultat en* `int` *ce qui a pour effet de tronquer le nombre à la virgule sans l'arrondir. En ajoutant* `p` *au résultat, on obtient un nombre compris entre* `p` *et* `n-1+p`*.*

Ensuite, la méthode `Thread.sleep(long temps)` *permet d'interrompre momentanément l'exécution du programme. Le programme reprendra son exécution au bout de* `temps` *millisecondes. Par exemple,*

```
Thread.sleep(2000);
```

interrompra l'exécution du programme pendant 2 secondes.

Attention ! Pour pouvoir utiliser `Thread.sleep`*, il faut ajouter une directive* `throws InterruptedException` *avant l'accolade sur la ligne de déclaration du programme principal :*

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

Pour la liste des messages, prévoyez en quelques-uns du genre `"Reformatage du disque dur"` *ou* `"Suppression de vos fichiers"`*. Il ne vous restera plus qu'à afficher l'un d'entre-eux en fonction d'un nombre tiré au hasard.*

Que faut-il donner au programme comme données pour faire le calcul ? *En lisant l'énoncé, il y a un certain nombre de données. D'une part le nombre de messages à afficher qui doit être lu au clavier. Nous prendrons un* `int : nbMessages`*. En ce qui concerne les messages, ils seront définis directement dans le code Java.*

Quel est le résultat attendu ? *Le résultat attendu est une suite de messages affichés à l'écran. Autant que l'utilisateur en a demandé. Chacun de ces messages sera suivi par des points et la ligne sera terminée par le mot* `Fait`*. Pour tout ceci, il s'agit de valeurs prédéfinies dans le programme et qui seront affichées directement. Il n'y a pas nécessité de définir des variables.*

Étape 2 : Trouver une méthode de calcul ou comprendre celle proposée

Dans les exercices qui vous sont proposés, s'il ne s'agit ni d'un problème simple ni d'un problème bien connu, une ou plusieurs indications sur la méthode de résolution vous sont indiquées. Si ce n'est pas le cas, vérifiez bien que ce « *nouveau* » problème ne s'approche pas d'un autre déjà vu ou d'un problème bien connu.

Lisez bien l'énoncé ! Une fois que vous vous êtes assuré de bien comprendre le problème (la réponse à donner en fonction des entrées) et les indications, prenez un exemple de données et essayez d'y appliquer à la main la méthode de résolution proposée. Refaites ceci avec de nouveaux exemples jusqu'à ce que vous soyez apte à le faire « *comme un automate* », sans avoir ni à réfléchir ni à inventer.

Exemple

Est-ce que le problème est connu ? *Sur l'exemple abordé, il ne s'agit pas vraiment d'un problème bien connu, mais il reste relativement simple à comprendre.*

Construisons un jeu d'exemples. *Choisissons une liste de messages dans un premier temps. Nous en prenons 6 par exemple :*

- `"Reformatage du disque dur."`
- `"Effacement de tous les fichiers."`
- `"Destruction de la mémoire."`
- `"Vidage de la base de registres."`
- `"Recopie du virus dans tous les fichiers."`
- `"Envoi du virus à tous vos contacts."`

Nous allons construire maintenant un tableau avec l'entrée `nbMessages` *et un exemple de résultat attendu*

`nbMessages`	Ecran	remarques
4	Recopie du virus dans tous les fichiers.....Fait Destruction de la mémoire.....Fait Effacement de tous les fichiers.....Fait Destruction de la mémoire.....Fait	Le nombre de points est variable. Il y a une pause de 500 ms entre chaque point
6	Envoi du virus à tous vos contacts.....Fait Recopie du virus dans tous les fichiers.....Fait Effacement de tous les fichiers.....Fait Destruction de la mémoire.....Fait Recopie du virus dans tous les fichiers.....Fait Vidage de la base de registres.....Fait	

TABLE 4.1 – jeu d'exemples

Certes avec seulement 6 messages différents, il y a de fortes chances que le même message sorte plusieurs fois au cours de la même exécution du programme, mais en en augmentant le nombre, cela arrivera moins fréquemment. Pour en augmenter le nombre il faudrait utiliser une structure de données que nous n'avons pas encore abordée. Nous laisserons cette seconde solution pour plus tard... Nous nous contenterons alors de ces 6 messages différents.

Étape 3 : Cas particuliers

Il faut toujours envisager tous les cas y compris les cas erronés et ceux qui ne peuvent soit-disant jamais arriver. Ce n'est pas toujours faisable suivant la complexité de la structure de données et du programme, mais il faut au moins essayer d'en envisager un maximum.

Exemple

Le seul cas particulier est celui où l'utilisateur entre une valeur plus petite ou égale à 0. Dans ce cas, on peut faire plusieurs choix :

- *le programme affiche une erreur et sort sans rien écrire d'autre ;*
- *le programme affiche une erreur et redemande une valeur correcte ;*
- *le programme ne fait rien.*

D'autres choix sont encore possibles, mais nous ne pourrons les envisager que plus tard. Pour des raisons de simplification, ici, nous allons prendre la troisième option.

Étape 4 : Quelles instructions ?

Tant que vous n'avez pas une expérience suffisante et même si cette étape peut poser question, c'est très probablement celle qui vous débloquera pour la suite de l'élaboration du programme. Elle consiste à essayer de repérer quelles instructions structurées nous seront nécessaires.

Nous avons à notre disposition plusieurs types d'instructions. Si la déclaration de variables, l'affectation et l'affichage de résultats à l'écran sont simples à mettre en œuvre, il n'en est pas de même pour les instructions structurées : la conditionnelle (`if ... else ...`), la boucle tant que (`while...`) et sa variante la boucle répéter...tant que (`do...while...`), et la boucle pour tout... (`for...`). Revoyons à quoi servent ces instructions et voyons quand y penser.

La conditionnelle

L'instruction conditionnelle permet d'exécuter des codes spécifiques en fonction d'une ou plusieurs conditions. Parfois on entend parler de **"boucle" conditionnelle** mais **ce terme est à bannir au plus vite car il est inapproprié et est source de confusion**. En effet, dans le mot **boucle**, il y a la notion de **répétition du même calcul**, opération **pour laquelle l'instruction conditionnelle n'est absolument pas conçue**. En effet, le `if` permet d'effectuer un code spécifique en fonction d'une ou plusieurs conditions mais ne permet en aucun cas de répéter plusieurs fois ce code. Pour ces raisons, nous n'emploierons que l'expression **instruction conditionnelle** ou nous parlerons du `if`.

Quand faut-il penser à l'instruction conditionnelle ? il faut y penser dès que, suivant une ou plusieurs conditions, il y a une ou plusieurs actions spécifiques à réaliser. Cependant, il ne faut pas confondre avec une boucle `while` où il y a aussi une condition mais elle est associée avec une notion de répétition du calcul. Ici, pas de notion de reprise du calcul.

Il y a essentiellement 4 formes possibles et toutes ces formes sont envisageables. Juste différents cas avec des calculs spécifiques. Il suffit de choisir celle qui est adaptée à votre problème :

un cas particulier et rien d'autre

Cette première forme se détecte lorsque nous sommes amenés à dire *si ... alors ...* sans rien préciser d'autre. Pas de sinon ou, au mieux, un sinon rien. Un `if` sans `else` conviendra. Ceci nous donne un squelette Java de la forme :

```
if (condition) {
  instructions
}
```

Si cela ne suffit pas à formuler votre problème, envisagez une des autres formes ci-dessous.

un calcul spécifique et un cas général ou un cas par défaut

Cette deuxième forme est à adopter si votre problème peut s'exprimer de la forme *si ... alors ... sinon ...*. Un `if` avec cette fois-ci un `else` conviendra. Soit un squelette Java de la forme :

```
if (condition) {
  instructions
} else {
  instructions
}
```

devrait convenir. Si ce n'est pas le cas, envisagez une des autres formes.

plusieurs cas spécifiques sans cas général ni cas par défaut

Cette troisième forme s'énonce par *si ... alors ... sinon si ... ...* avec un ou plusieurs *sinon si* mais pas de *sinon* final. Un `if` avec un ou plusieurs `else if`, autant que nécessaire, conviendra. Soit un squelette de la forme :

```
if (condition) {
  instructions
} else if (condition) {
  instructions
} ...
```

devrait convenir avec autant de `if` apparaissant que de cas différents à envisager. Si ce n'est pas le cas, envisagez une autre forme.

plusieurs cas spécifiques avec un cas général ou un cas par défaut

Cette dernière forme s'énonce par *si ... alors ... sinon si ... ... sinon ...* avec plusieurs *sinon si* et un *sinon* final. Un `if` avec un ou plusieurs `else if` puis un `else` final conviendra. Soit un squelette de la forme :

```
if (condition) {
  instructions
} else if (condition) {
  instructions
} else if ... {
  ...
} else {
  instructions
}
```

devrait convenir avec, comme pour le cas précédent, autant de `else if` que nécessaire. Le `else` est toujours en dernier. Si ce n'est pas le cas et si aucune des formes précédentes ne convient, posez-vous la question de savoir si c'est bien une conditionnelle qu'il vous faut.

Exemple

Reprenons notre exemple. À ce point de notre progression sur le problème posé, il est temps de se poser la question de savoir si une ou plusieurs conditionnelles sont nécessaires. En reprenant l'énoncé, on peut y lire :

> Chacun de ces messages sera pris au hasard dans une liste.

Par ailleurs, nous avons établi une liste de 6 messages comme il nous a été demandé, et il nous a été expliqué comment tirer un nombre au hasard par l'instruction suivante :

```
alea = (int) (n *Math.random ()) + p;
```

Avec ces deux éléments, il nous est possible de générer un nombre compris entre 0 et 5 avec `n` *valant 6 et* `p` *valant 0, puis, si* `alea` *vaut 0 on affiche le premier message, sinon si* `alea` *vaut 1 on affiche le deuxième et ainsi de suite jusqu'au cinquième message (*`alea` *valant 4 dans ce dernier cas) sinon on affiche le sixième. Ceci correspond à la quatrième forme présentée ci-dessus. On peut d'ores et déjà affirmer qu'il nous faudra une instruction conditionnelle et exprimer cette partie du problème sous la forme du pseudo-code qui suit.*

```
alea = (int) (6 *Math.random ());
si alea == 0 alors
  écrire le premier message
sinon si alea == 1 alors
  écrire le deuxième message
sinon si alea == 2 alors
  écrire le troisième message
sinon si alea == 3 alors
  écrire le quatrième message
sinon si alea == 4 alors
  écrire le cinquième message
sinon
  écrire le sixième message
fin si
```

En complément, il faudra bien sûr ne pas oublier de déclarer la variable `alea`*.*

Les boucles

Une fois que nous avons déterminé s'il y a une ou plusieurs instructions conditionnelles dans le programme, il est temps de se poser la question de savoir s'il y a une ou plusieurs boucles puis de savoir de quel type de boucles nous aurons besoin.

Ce qu'il faut regarder pour répondre à cette question, c'est s'il y a une notion de répétition d'actions ou un ensemble de données sur lequel effectuer certaines actions. Si, en décrivant la solution, on entend des termes comme *pour tout*, *pour chaque*, *faire n fois*, *pour ... allant de ... à ...*, *répéter jusqu'à*, *répéter tant que*, *refaire*, *reprendre*, etc., nous pouvons nous attendre à avoir une ou plusieurs boucles dans le programme. Ce sont ces indicateurs qui vont nous inciter à mettre une boucle et dans un second temps, en fonction des termes employés, à choisir le type de boucle.

Exemple

Reprenons notre exemple et essayons de voir s'il y a une ou plusieurs boucles. Commençons par relire l'énoncé.

> Ce programme demandera à l'utilisateur un entier : le nombre de fois qu'il devra écrire un message à l'écran.

A priori, il va falloir afficher un message à l'écran un certain nombre de fois. Le nombre de fois est variable d'une exécution à l'autre, il faudra alors impérativement une boucle. Continuons la lecture.

> Chacun de ces messages sera pris au hasard dans une liste.

Nous avons déjà traité cette partie. Elle a donné lieu à l'introduction d'une conditionnelle. Il n'y a pas vraiment de notion d'itération ici sauf qu'il faudra le faire pour chacun des messages à afficher. Un élément à garder en mémoire pour plus tard quand nous agencerons les instructions...

> Sur la même ligne que le message et à sa suite, pour simuler la visualisation de l'avancement de l'opération en cours, le programme affichera à l'écran un nouveau point toutes les demi-secondes entre 5 et 15 fois.

De nouveau une action à répéter un certain nombre de fois. Faut-il une deuxième boucle ou la précédente suffira ? Apparemment, cette répétition n'est pas la même que la précédente. La seule relation avec la précédente est qu'il faudra le faire pour chacun des messages mais notons cela pour l'instant. Il sera temps de s'en préoccuper quand nous organiserons les instructions. Pour l'instant, écrivons qu'il nous faudra une seconde boucle. Continuons la lecture de l'énoncé.

> Le nombre de fois sera aussi pris au hasard. Ensuite, il affichera `"Fait"` avant de passer au message suivant.

On parle de nouveau d'un nombre de fois... Mais ! Attention ! Il s'agit du nombre de fois qu'il faut afficher un point, action que nous avons déjà traitée. Donc, pas de nouvelle boucle cette fois-ci. Ensuite, il ne s'agit que d'afficher un texte à l'écran. Pas de nouvelle boucle non plus, même s'il faudra le faire pour chaque message. La boucle sur les messages permettra de répéter cet affichage.

Dans la suite de l'énoncé, nous ne trouvons que quelques conseils ou indications pour résoudre certains sous-problèmes. En vérifiant bien, il n'y a pas de nouvelle notion de boucle de toute évidence.

En résumé, il y aura une conditionnelle et deux boucles. Nous avons à notre disposition deux types de boucles avec une variante pour la boucle tant que. Il reste à déterminer de quel type sera chacune des boucles.

Choix des boucles

Bien entendu, si notre analyse précédente nous a amenés à conclure qu'il n'y aura pas de boucle dans le programme, il ne sera pas nécessaire de faire ce choix.

Dans le cas où il y a une ou plusieurs boucles dans le programme en cours d'étude, pour chacune d'elle, il faut déterminer quel type de boucle conviendra le mieux : un `while` ou un `for` ? Certes, il est possible de se contenter d'un seul des deux types : toute boucle `while` peut être écrite à l'aide d'une boucle `for` et, réciproquement, toute boucle `for` peut être écrite à partir d'une boucle `while`. Mais, dans certains cas, la boucle `while` sera mieux adaptée alors que la boucle `for` le sera dans d'autres.

On choisira de préférence :

— une boucle `for` quand on peut calculer le nombre de tours de boucle qu'il faudra faire au moment d'y entrer ;

— une boucle while quand le nombre de tours n'est pas calculable à l'entrée de la boucle. Dans ce second cas, il y a toujours une condition associée à la boucle : soit une condition d'arrêt (*répéter jusqu'à*) soit une condition de reprise (*répéter tant que*).

Un autre critère de choix équivalent :

— quand on utilise des termes de la forme *pour tout*, *pour chaque*, *faire n fois*, *pour ... allant de ... à ...*, *répéter ... fois*, *refaire ... fois*, pour décrire la boucle, on choisira une boucle `for` ;

— quand ce sont plutôt des termes contenant *tant que* ou *jusqu'à* suivis d'une condition, on choisira une boucle `while`. Après, un *while* ou un *do ... while* ? Pour l'instant, nous ne chercherons pas à les différencier. Votre future expérience vous guidera dans votre choix.

Exemple

Toujours sur notre exemple de faux virus, nous avons deux boucles.

La première consiste à afficher du texte un certain nombre de fois. Au moment où nous concevons le programme, nous ne connaissons pas ce nombre de fois puisqu'il ne sera précisé par l'utilisateur qu'au moment de l'exécution. Notons que, quelque soit le problème, c'est presque toujours le cas : on ne connait pas le nombre de tours de boucle à faire au moment de la conception du programme. Par contre, au moment de d'entrer dans la boucle, l'utilisateur aura indiqué combien de messages à écrire. C'est ce moment là qui est important. Il faudra alors préférer une boucle `for`*.*

La seconde consiste à afficher un certain nombre de points avec une temporisation entre deux points. Ce nombre n'est pas connu maintenant mais le sera au moment d'entrer dans la boucle. Donc, là-aussi une boucle `for`*.*

Étape 5 : Agencement des instructions

Maintenant, il nous faut organiser les instructions que nous avons identifiées dans l'étape précédente les unes par rapport aux autres. Bien entendu, s'il n'y en a pas plus d'une, il n'y a rien à faire de plus. Nous pouvons passer à l'étape suivante.

Dans le cas de plusieurs instructions parmi la conditionnelle et les boucles, il faut travailler deux instructions par deux instructions. Dans ce cas, nous avons 4 dispositions possibles :

— la première avant la seconde,
— la première après la seconde,
— la première dans la seconde et
— la première contenant la seconde.

Cas de deux conditionnelles Ce cas ne pose pas de gros problèmes en général. Il est facile de voir si l'une doit être avant, après ou dans l'autre. Tout de même, dans tous les cas, demandez-vous s'il ne serait pas possible de n'en faire qu'une. Le code obtenu n'en sera que plus simple.

Cas d'une conditionnelle et d'une boucle Si nous avons une conditionnelle et une boucle, quel que soit le type de la boucle, la conditionnelle peut être placée avant, après ou dans la boucle, ou, encore, elle peut contenir la boucle. Notez que cette dernière solution peut amener à reproduire la boucle plusieurs fois. Il faut essayer d'éviter au maximum ce dernier choix en utilisant au besoin des variables locales pour essayer de transformer la conditionnelle et de la placer avant la boucle. Cependant, ceci n'est pas toujours faisable.

Pour déterminer quel agencement convient le mieux, il faut se poser les questions suivantes :

- Est-ce que la valeur de la ou des conditions du `if` est susceptible de varier au fil des tours de boucle et est-ce qu'en conséquence l'ensemble des instructions à exécuter peut varier à chaque tour de la boucle ?
  - si oui, placer le `if` dans la boucle
  - si non passer à la question suivante
- Est-ce que la condition porte sur le résultat final de la boucle ?
  - si oui, placer le `if` après la boucle
  - si non passer à la question suivante
- Est-ce que le calcul à faire dans la boucle varie en fonction de la condition mais ne change pas d'un tour à l'autre ?
  - Est-ce qu'il ne s'agit que d'une ou plusieurs valeurs utilisées dans le calcul qui changent mais pas le calcul en lui-même (calcul paramétré) ?
    - si oui, pour chaque paramètre du calcul, utiliser une variable, placer le `if` avant la boucle et y initialiser ses paramètres.
    - si non passer à la question suivante
  - Est-ce qu'il est nécessaire d'optimiser le temps de calcul à tout prix (ce cas est rare) ?
    - si oui, mettre la boucle dans la conditionnelle en la répétant dans chaque cas aux modifications nécessaires près.
    - si non placer le `if` dans la boucle.

Cas de deux boucles Dans le cas de deux boucles, il n'y a que deux possibilités : l'une après l'autre ou l'une dans l'autre. Pour savoir laquelle des deux options prendre, il suffit de regarder si l'une d'entre-elles doit être effectuée pour chaque tour de l'autre. Si tel est le cas, la première devra être placée à l'intérieur de la seconde. Dans le cas contraire, on essaiera de les placer consécutivement.

Exemple

Reprenons notre exemple. Nous avons une conditionnelle et deux boucles `for`. Comment les assembler ? Notons que le type de la boucle importe peu dans cette étape.

Nous avons une première boucle `for` sur le nombre de messages à afficher et une condition sur une variable `alea` permet de choisir le message. De toute évidence, si on veut que le message change à chaque fois, il va falloir tirer un nouveau nombre `alea` au hasard pour chaque affichage de message et, selon le résultat obtenu, afficher le message correspondant. C'est le premier cas. Donc, le `if` sera dans cette boucle `for`.

Nous avons toujours la première boucle `for` sur le nombre de messages à afficher et la boucle d'affichage des points avec temporisation. Là encore, nous avons noté qu'il faut afficher ces points pour chaque affichage de message. Ainsi la boucle `for` sur les points se situera dans la boucle `for` sur les messages. Ainsi, la boucle sur les messages contiendra à la fois un `if` et un `for`. Mais comment organiser ces deux instructions ?

En regardant plus précisément, il faut d'abord afficher le message tiré au hasard, ensuite les points et, enfin, afficher le mot "Fait" en passant ensuite à la ligne. De là, nous en déduisons que le `if` doit être placé avant le `for` et qu'il y aura encore un affichage à faire à la suite du `for`. Nous en arrivons à la structure de programme suivante qui n'était pas forcément évidente à la lecture de l'énoncé :

```
lecture du nombre de messages
for sur les messages {
  calcul de la variable alea
```

```
    if de choix du message à afficher
    for sur les points
    affichage complémentaire
  }
```

Étape 6 : Formalisation de l'algorithme

Maintenant que nous connaissons la structure du programme, il ne reste plus qu'à formaliser l'algorithme. Pour ceux qui sont le plus à l'aise, il sera possible d'omettre cette étape pour coder directement le programme en Java.

En fait, une fois que nous avons décidé quelles instructions seront nécessaires, puis que nous les avons organisées, l'algorithme est pratiquement écrit comme vu dans l'exemple ci-dessus. On peut affiner cet algorithme en essayant d'être plus précis. En faisant un bilan de ce que nous venons de faire :

- nous avons déterminé lors de l'étape 1 les entrées et les sorties ;
- lors des étapes 2 et 3 nous avons construit un jeu de données envisageant, à la fois les cas simples et les cas particuliers.
- les étapes suivantes nous ont amenés à un schéma de programme.

Il ne nous reste plus qu'à assembler tout ceci, déterminer les variables locales supplémentaires et à affiner l'expression du calcul.

Exemple

A l'étape 1, nous avons déterminé qu'il n'y a qu'une seule entrée : le nombre de messages nommé `nbMessages` *de type* `int`*. Dans cette même étape, nous aurons en sortie une suite de messages sélectionnés dans un ensemble. Il n'y aura pas besoin de variable particulière pour ces messages. Nous avons construit un jeu d'exemple lors de l'étape 2 et déterminé le comportement du programme pour les cas particuliers.*

A l'étape 4, nous avons déterminé que nous tirons au hasard un entier pour choisir un des messages. Le résultat est mémorisé dans une variable `alea` *qui devra être de type* `int`*. Dans la même étape, plus loin, nous avons une première boucle* `for` *sur le nombre de messages. Il nous faudra une variable aussi. Appelons la* `n`*. Elle sera aussi de type* `int` *et sera déclarée dans la boucle* `for`*. Ensuite, il nous faudra aussi tirer au hasard un nombre entre 5 et 15. Il nous faut de nouveau une variable de type* `int` *pour mémoriser ce nombre. Nous l'appellerons* `tempo`*. un nouvel indice de boucle que nous appellerons* `k` *de type* `int` *sera nécessaire pour la seconde boucle* `for`*. Celle comptant le nombre de points à afficher. Comme pour* `n`*, cette variable sera déclarée dans la boucle.*

Maintenant nous pouvons formaliser l'algorithme :

```
Entrée
  nbMessages un entier : le nombre de messages à afficher
Sortie
  autant de messages que demandé
Variables locales
  alea un entier : tirage au sort du message
  n un entier : compteur du nombre de messages affichés
  tempo un entier : nombre de points à afficher
  k un entier : compteur du nombre de points affichés
```

```
Début
  écrire "Nombre␣de␣messages␣:␣"
  lire nbMessages

  pour n allant de 1 à nbMessages
    alea = un nombre entier pris au hasard entre 0 et 5
    si alea == 0
      écrire "Je␣reformate␣le␣disque␣dur."
    sinon si alea == 1
      écrire "Effacement␣de␣tous␣les␣fichiers."
    sinon si alea == 2
      écrire "Destruction␣de␣la␣mémoire."
    sinon si alea == 3
      écrire "Vidage␣de␣la␣base␣de␣registres."
    sinon si alea == 4
      écrire "Recopie␣du␣virus␣dans␣tous␣les␣fichiers."
    sinon
      écrire "Envoi␣du␣virus␣à␣tous␣vos␣contacts."
    fin si

    tempo = un nombre entier pris au hasard entre 0 et 5
    pour k allant de 1 à tempo
      écrire un point
      temporiser 500 ms
    fin pour
    écrire "Fait"
  fin pour
fin
```

***Une remarque** : lors de l'étape 3 nous avons identifié le cas où l'utilisateur entre une valeur négative ou nulle pour `nbMessages`. Nous avons choisi de ne rien faire dans cette éventualité. Dans tous les langages de programmation, le `pour tout n allant de 1 à nbMessages` se traduira par une boucle `for` qui, dans le cas qui nous intéresse, sort immédiatement sans jamais exécuter son corps. C'est ce que nous avions prévu comme comportement !*

Étape 7 : Écriture du code source

L'algorithme étant écrit, l'étape suivante consiste à le coder dans le langage de programmation choisi. Comme tous les langages de programmation issus de l'impératif comme Java, C, C++, VBA, etc. contiennent le même jeu d'instructions à la syntaxe près, si nous avons pris soin de ne pas introduire d'instructions ou d'expressions spécifiques à un langage précis, celui-ci peut être traduit dans n'importe quel langage évolué. Un point positif supplémentaire jouant en faveur de l'écriture d'un algorithme avant le codage. Comme ce cours porte sur le langage Java, nous le traduirons dans ce langage mais nous pourrions le traduire aussi simplement dans d'autres langages de programmation.

En général, si l'algorithme a été bien conçu, il ne s'agit que d'une simple transcription sans grosse difficulté. Si vous avez à trop réfléchir dans cette étape, la cause est très probablement un manque de précision dans l'algorithme - une idée exprimée de mnière trop floue bien souvent. Lors de la

traduction, il faut veiller à ce que le programme reflète au maximum l'algorithme. Il ne doit pas y avoir de "prise d'initiative" lors de cette phase. Chaque ligne du programme doit correspondre à une ligne de l'algorithme. Ceci afin de faciliter le débogage du code par vous ou tout autre personne ultérieurement.

Exemple

Nous avons pris soin d'écrire l'algorithme le plus précisément possible afin qu'il reste facilement compréhensible par une autre personne tout en collant au mieux au jeu d'instructions des langages de programmation modernes issus de l'impératif. Il en résulte que la traduction se fera presque mot à mot. Voici le code obtenu où l'algorithme a été mis en commentaire pour mettre en évidence cette similarité.

```
  public static void main(String[] args) throws InterruptedException {
// Entrée
//  nbMessages un entier : le nombre de messages à afficher
    int nbMessages;

// Sortie
//  autant de messages que demandé
//  *********pas de ligne de code correspondant

// Variables locales
//  alea un entier : tirage au sort du message
    int alea;

//  n un entier : compteur du nombre de messages affichés
//  *********sera déclaré dans la boucle

//  tempo un entier : nombre de points à afficher
    int tempo;

//  k un entier : compteur du nombre de points affichés
//  *********sera déclaré dans la boucle

// Début
//  écrire "Nombre de messages : "
    System.out.print("Nombre de messages : ");

//  lire nbMessages
    nbMessages = Terminal.lireInt();

//  pour n allant de 1 à nbMessages
    for (int n = 1; n <= nbMessages; n = n+1) {

//   alea = un nombre entier pris au hasard entre 0 et 5
      alea = (int) (Math.random() *6);
```

```
//   si alea == 0
       if (alea == 0) {
//    écrire "Je reformate le disque dur."
         System.out.print("Je␣reformate␣le␣disque␣dur.");
//   sinon si alea == 1
       } else if (alea == 1) {
//    écrire "Effacement de tous les fichiers."
         System.out.print("Effacement␣de␣tous␣les␣fichiers.");
//   sinon si alea == 2
       } else if (alea == 2) {
//   écrire "Destruction de la mémoire."
         System.out.print("Destruction␣de␣la␣mémoire.");
//   sinon si alea == 3
       } else if (alea == 3) {
//    écrire "Vidage de la base de registres."
         System.out.print("Vidage␣de␣la␣base␣de␣registres.");
//   sinon si alea == 4
       } else if (alea == 4) {
//    écrire "Recopie du virus dans tous les fichiers."
         System.out.print(
              "Recopie␣du␣virus␣dans␣tous␣les␣fichiers.");
//   sinon
       } else {
//    écrire "Envoi du virus à tous vos contacts."
         System.out.print("Envoi␣du␣virus␣à␣tous␣vos␣contacts.");
//   fin si
       }

//   tempo = un nombre entier pris au hasard entre 0 et 5
       tempo = (int) (Math.random() *11) + 5;

//   pour k allant de 1 à tempo
       for (int k = 1; k <= tempo; k++) {
//    écrire un point
         System.out.print(".");
//    temporiser 500 ms
         Thread.sleep(500);
//   fin pour
       }
//   écrire "Fait"
       System.out.println("Fait");
//  fin pour
     }
     System.out.println("Fini␣!␣Bonne␣réinstallation␣du␣système␣!");
// fin
   }
```

4.2 En résumé

Lors de la réalisation d'un exercice de programmation, il est tentant d'écrire directement le code dans le langage de programmation choisi. Or, un débutant aura beaucoup de mal à aboutir de la sorte à un programme opérationnel. On a des difficultés à comprendre ou à imaginer la méthode ; on ne sait pas par quel bout commencer ; on mélange un peu tout ; etc. C'est pourquoi il vaut mieux procéder méthodiquement même si la démarche proposée peut sembler lourde. Après, au fil de l'acquisition d'expérience, les différentes étapes deviendront de plus en plus faciles et même, vous arriverez à coder des problèmes de moins en moins simples directement en Java.

La démarche est la suivante :

Entrées et sorties
: Déterminer les entrées et les sorties du programme.

Trouver une méthode de calcul ou comprendre celle proposée
: En se dotant d'un jeu d'exemples, il faut réussir à résoudre le problème en agissant comme un automate, sans avoir à réfléchir.

Cas particuliers
: Envisager les cas particuliers en essayant d'être le plus exhaustif possible.

Quelles instructions ?
: En analysant la méthode de résolution, faire l'inventaire des instructions structurées qui seront nécessaires.

Agencement des instructions
: En prenant deux par deux les instructions trouvées à l'étape précédente, il faut déterminer leur agencement (avant, après, dedans, autour) pour aboutir à un schéma de programme.

Formalisation de l'algorithme
: Résumer et formaliser vos remarques sous forme d'algorithme.

Écriture du code source
: Traduire le plus fidèlement possible l'algorithme dans le langage de programmation choisi.

Bien sûr, chaque fois que nous écrirons du code intermédiaire, nous vérifierons que le code conçu réponde aux attentes en simulant l'exécution non seulement sur le jeu d'exemples défini à la deuxième étape mais aussi sur les cas particuliers identifiés à la troisième. La dernière étape accomplie, on testera le programme sur les mêmes exemples et cas particuliers.

4.3 Pièges classiques

Il est fréquent qu'un débutant en programmation confonde l'instruction conditionnelle (`if`) et la boucle tant que (`while`). On trouve plusieurs cas de figure :

`else` à la suite d'un `while`
: dans ce cas, l'erreur est vite repérée : vous obtiendrez un message d'erreur à la compilation ou, sous un IDE comme Eclipse, le `else` sera souligné en rouge. Rappelez-vous que la boucle `while` n'a pas de `else`. C'est le code qui suit le corps de la boucle qui est exécuté dès que la condition de la boucle n'est plus vérifiée. Pour corriger ce problème il y a deux possibilités en fonction de ce que l'on souhaite faire :

- **le code du `while` est vraiment à répéter plusieurs fois**. Il s'agit alors d'une mauvaise compréhension du fonctionnement de la boucle `while` et surtout de sa syntaxe. Dites-vous bien qu'il n'y a pas de `else` à la suite d'un `while` ! Pour corriger ce problème, il suffit bien souvent de supprimer le `else` tout en laissant les instructions qu'il contient.
- **le corps du `while` n'est pas à répéter mais il ne faut l'exécuter que lorsque la condition est vérifiée**. Dans ce cas, il s'agit plus d'une confusion entre le `while` et le `if`. Le `while` permet de répéter un code tant qu'une condition est vérifiée. Il y a répétition des instructions. Le `if` permet de réaliser des calculs différents en fonction de conditions. Il n'y a pas de répétition du code. Pour résoudre ce problème, il suffit de remplacer le mot-clé `while` par `if`.

utilisation du `while` à la place du `if`

ici et en l'absence de `else`, le compilateur ne détectera pas d'erreur. Ce n'est uniquement lors de l'exécution de votre programme que vous vous apercevrez que ce dernier "boucle" à l'infini : il n'affiche rien et ne vous redonne pas la main. C'est normal. Vous attendant à ce que le corps du `while` ne soit exécuté qu'une seule fois, vous ne vous êtes pas soucié de changer la valeur des variables utilisées dans la condition de la boucle et il n'y a pas lieu de le faire car il ne faut pas boucler sur ces instructions. Il s'agit de nouveau d'une confusion entre les deux instructions (`while` et `if`). La correction à apporter ne consiste qu'à remplacer le mot-clé `while` par `if`.

utilisation du `if` à la place du `while`

comme pour le cas précédent, il n'y a pas d'erreur à la compilation. Ce n'est que lors de l'exécution du programme que vous vous apercevez d'un mauvais fonctionnement de votre programme : il ne fait qu'une seule fois les instructions et même, parfois, il ne les exécute pas. Il s'agit là encore d'une confusion entre les deux instructions bien souvent associée à l'utilisation de l'expression boucle `if` qui, rappelons-le, est absolument à bannir. Un simple remplacement du mot-clé `if` par `while` suffit à résoudre ce problème.