Formation Java : Rappels sur les tableaux. Structures de données.

Bernard Hugueney

Table des matières

1 Remarques préliminaires

Les programmes sont des constructions virtuelles : contrairement aux constructions matérielles de l'ingénierie classique qui résultent de l'assemblage d'objets physiques (e.g. engrenages), les programmes assemblent des instructions qui n'ont pas d'autre réalité physique que leur stockage en mémoire.

Cet aspect virtuel enlève toutes contraintes physiques (e.g. poids, résistance des matériaux, précision de réalisation) ne laissant que les contraintes de notre capacité à **comprendre** nos réalisations virtuelles malgré leur **complexité**.

En effet, les programmes réellement utiles sont souvent extrêmement complexes. Le noyau Linux est par exemple constitué de plus de 18 millions de lignes de codes, et beaucoup d'autres logiciels sont incroyablement complexes / "volumineux".

Considérant que l'intelligibilité des programmes est le problème fondamental à résoudre, il apparaît logique que les caractéristiques des langages et des pratiques de programmation soient destinées à permettre de limiter/gérer cette complexité.

Paradoxalement, on apprend à programmer en écrivant/lisant de minuscules programmes, pour des raisons évidentes. Il faut donc essayer d'imaginer que ces programmes ne sont que des fragments de "vrais" programmes : c'est-à-dire gros, complexes et sujets à des évolutions qu'il faut anticiper pour être capable de les implémenter sans compromettre l'intégrité logique du programme.

En conséquence, on ne jugera pas seulement un programme aux résultats de son exécution (résultats correct, c'est-à-dire conforme aux spécifications,

ou non) mais à sa **simplicité**, son **extensibilité** et ses **performances**. Dans cet ordre de priorité car il est plus facile de modifier (pour ajouter des fonctionnalités et/ou le rendre plus performant) un programme que l'on comprend que de modifier (pour le rendre plus compréhensible) un programme que l'on ne comprend pas complètement !

2 Conventions de nommage

Il faut bien sûr apporter tout d'abord un soin particulier au **sens** des noms que l'on choisi . En effet, c'est de la signification des noms (de variables, de fonctions, de classes) que va dépendre l'intelligibilité des programmes. Or c'est celle-ci qui est primordiale, car pour pourvoir écrire et modifier un programme, il faut pouvoir le comprendre !

Cependant, la forme des noms est aussi importante pour aider à lire un programme. Comme les programmes non triviaux sont le résultat d'un groupe d'auteurs, il est important d'avoir des *conventions* qui sont partagées par tou(te)s. Les conventions de nommage en Java sont les suivantes.

2.1 camelCase (ou CamelCase)

Un nom est composé d'un ou plusieurs mots que l'on accole alors en mettant la première lettre du mot accolé. On parle en anglais de camelCase (ou CamelCase) en référence aux bosses du chameau (*camel* en anglais).

2.1.1 NomsDeClasses

Les noms de *classes*[1] commencent par une majuscule :

- Personne, Student, Point,...
- BatimentAdministratif, FicheTechnique, ForeignStudent, FilledPolygon, ...
- LettreDeMission, ColoredFilledPolygon, AdministrativeTeamFactory, ...
- ...

2.1.2 autresNoms

Les noms de variables (arguments, variables locales, attributs d'instances ou de classes[2]), de fonctions/méthodes,... commencent par une minuscule :

1. et d'*interfaces*
2. Mais dans ce dernier cas, cf. LES_CONSTANTES_NOMMÉES

— i, j, k, nb, personne, student, point, ...
— batimentAdministratif, ficheTechnique, foreignStudent, filledPolygon, ...
— lettreDeMission, coloredFilledPolygon, administrativeTeamFactory, ...
— ...

2.2 LES_CONSTANTES_NOMMÉES

Dans le code d'un programme, certaines valeurs (numériques ou autres) sont constantes au cours de l'exécution du programme : la valeur de π, les composantes de différentes couleurs dans un espace colorimétrique (par exemple RVB),...

Ces constantes sont nommées en MAJUSCULES_SÉPARÉES_PAR_DES_SOULIGNÉS
— PI, BLACK,...
— LIGHT_GREY, DARK_BROWN,...
— NB_MAX_ELEMENTS,...

3 Décomposition en fonctions

3.1 Décomposition

Décomposer, c'est permettre de comprendre des sous-ensembles (*composants*) séparément du reste. En conséquence, il faut limiter au maximum (et donc si possible éliminer) les dépendances entre composants.

Ceci est indispensable à la maintenance évolutive des programmes. En effet si la modification d'un partie d'un logiciel (un "composant") nécessitait l'adaptation (et donc la modification !) d'autre parties, l'effet "boule de neige" de modifications nécessitant des modifications pourrait être sans fin !

De plus, l'impératif d'intelligibilité implique aussi qu'il faille pouvoir décomposer un logiciel en composants suffisamment simples pour que leur représentation mentale puisse "tenir" dans notre mémoire de travail.

3.2 Faire quelquechose vs Calculer un résultat

On peut conceptuellement distinguer les fonctions :
— qui font quelque chose (écrire à l'écran, dans un fichier ou une base de données, envoyer des données sur le réseau, mais aussi modifier l'état du programme,...)
— qui calculent quelque chose (le résultat d'opérations,...)

Bien sûr, ces ensembles ne sont pas disjoints et des fonctions peuvent à la fois avoir un effet (faire quelque chose) et un résultat (calculer quelque chose). De plus, chaque instruction exécutée par le(s) processeur(s) a forcément un effet et modifie l'état de l'ordinateur (en écrivant des 0 et des 1 en mémoire) et tout programme utile a forcément des effets (sinon l'ordinateur qui l'exécute n'est qu'un radiateur électrique hors de prix !). Néanmoins, c'est justement pour éviter de considérer les programmes au niveau des instructions machines que l'on a inventé les langages de programmation et c'est pour pouvoir considérer isolément des partie de programme avec des propriétés spécifiques (par exemple l'absence d'effets de bords) que l'on **décompose** celui-ci.

De façon générale (c'est-à-dire avec des exceptions), on préfèrera écrire et utiliser des fonctions qui calculent des résultats que des fonctions qui ont des effets. Le principal avantage de résultats sous forme de valeurs dans un programme est qu'il est possible de stocker et modifier ceux-ci ce qui permet de réutiliser de telles fonctions dans des contextes très variés.

3.3 Exemple de programme effectuant une conversion Fahrenheit → Celsius

Soit un programme destiné à permettre de convertir une température de l'échelle des degrés Fahrenheit à celle des degrés Celsius. Afin que ce programme soit utile pour différentes valeurs de températures, on veut pouvoir saisir la température et afficher le résultat de la conversion.

Les opérations nécessaires à la saisie qu clavier et à l'affichage à l'écran seront expliquées juste après, nous nous intéressons ici seulement à la décomposition du programme.

3.3.1 Modélisation des données

Java étant un langage typé, et même statiquement typé, chaque valeur doit avoir un type, et même un type indiqué explicitement lors de l'écriture du programme en Java. Ici, on veut manipuler des nombres à virgule pour lesquels des erreurs de représentations sont tolérables, donc on peut choisir un type primitif de représentation en virgule flottante. N'ayant pas de contraintes particulières en encombrement mémoire ou en performance (on convertit une seule valeur à la fois !), on peut choisir le type `double`.

3.3.2 Décomposition

Les étapes du programme sont les suivantes :

1. afficher le message d'invite demandant de saisir une température en Fahrenheit
2. lire le nombre à virgule saisi au clavier [3]
3. calculer le résultat
4. afficher le résultat

3.3.3 Degré 0 : pas de décomposition

Dans le cas où le seul objectif est le résultat de la conversion, c'est-à-dire lorsque le code ne fait pas partie d'un projet informatique destiné à évoluer [4], le plus immédiat est de ne coder que le strict minimum :

```
1  import java.util.Scanner;
2
3  public class FahrenheitToCelsius {
4      public static void main(String[] args){
5          System.out.println("Entrez la T° en F:");
6          Scanner sc= new Scanner(System.in);
7          double tF= sc.nextDouble();
8          System.out.println(""+tf" °F = "+((tF-32.)/1.8)+" °C");
9      }
10 }
```

Attention ! Si lordinateur est configuré "en Français", le programme attendra que les nombres à virgule saisis utilisent une **virgule** comme séparateur pour la partie décimale (par exemple "1,5") et l'utilisation d'un point (par exemple "1.5") provoquera une erreur. Et ce alors que le même programme utilisera un point comme séparateur pour la partie décimale lors de ses affichages !

3.3.4 Une fonction, mais laquelle ?

On envisage que le programme puisse évoluer pour :
— offir d'autres fonctionnalités
— fournir cette fonctionnalité de façon répétitive (pour que l'on ait pas à relancer le programme pour convertir plusieurs valeurs)

3. Dans un premier temps, on ignorera la possibilité d'une erreur de saisie et l'on ne se donnera pas la peine de valider celle-ci. Évidemment, un vrai programme doit valider toute donnée qu'il reçoit !

4. mais dans ce cas, là, on ne prendrait sans doute pas la peine d'utiliser le langage Java !

On peut alors vouloir définir une fonction (autre que le `main` correspondant au programme principal). Les deux possibilités à envisager en premier sont :

— une fonction qui lit la température en °F et écrit la température en °C :

```java
import java.util.Scanner;

public class FahrenheitToCelsius {
    public static void fToC(){
        System.out.println("Entrez la T° en F:");
        Scanner sc= new Scanner(System.in);
        double tF= sc.nextDouble();
        System.out.println(""+tf" °F = "+((tF-32.)/1.8)+" °C");

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

— une fonction qui prend en argument la température en °F et retourne la température en °C :

```java
import java.util.Scanner;

public class FahrenheitToCelsius {
    public static double fToC(double tempF){
        return ((tempF-32.)/1.8);
    }
    public static void main(String[] args){
        System.out.println("Entrez la T° en F:");
        Scanner sc= new Scanner(System.in);
        double tF= sc.nextDouble();
        System.out.println(""+tf" °F = "+fToC(tF)+" °C");
    }
}
```

On observe le lien entre Entrées/Sorties d'une part, et Arguments/Valeur de retour d'autre part.

L'avantage principal d'une fonction qui **calcule** un résultat et le retourne est qu'il est possible de faire ce que l'on veut avec : on pourrait alerter en cas de température dépassant un seuil, calculer les valeurs minimale, maximale ou moyenne sur une série de valeurs, etc.

3.3.5 Fonction auxiliaire

En fait, on peut aussi supposer que le fait de demander une valeur numérique (à virgule) pourra être une fonctionnalité élémentaire réutilisable

en dehors de la conversion entre échelles de température. Dans un vrai programme, cela pourrait (devrait !) aussi être l'occasion de valider la valeur saisie :

— qu'il s'agisse bien d'un nombre !

— que celui-ci fasse partie d'un intervalle de valeurs admissibles

On pourrait alors demander de répéter la saisie jusqu'à ce que la valeur soit valide (cf. §5.1.2).

Dans tous les cas, on pourra isoler la saisie dans une fonction :

```
1  import java.util.Scanner;
2
3  public class FahrenheitToCelsius {
4      public static double readDouble(String prompt){
5          System.out.println(prompt);
6          Scanner sc= new Scanner(System.in);
7          return sc.nextDouble();
8      }
9      public static double fToC(double tempF){
10         return ((tempF-32.)/1.8);
11     }
12     public static void main(String[] args){
13         double tF= readDouble("Entrez la T° en F:");
14         System.out.println(""+tf" °F = "+fToC(tF)+" °C");
15     }
16 }
```

Évidemment, dans un vrai programme, une fonction telle que `readDouble` aurait sa place dans une autre classe, pour des raisons évidentes d'organisation du code.

4 Tableaux

4.1 déclarations de tableaux

Un tableau d'éléments est un ensemble de `n` valeurs identifiées par un indice allant de `0` à `n-1`. Pour un tableau `ts`, on accède à la casse d'indice `i` avec la notation `ts[i]` et l'on peut lire (mais pas modifier) la taille de ce tableau avec le champ `length` : `ts.length`.

La déclaration d'une variable ou argument `ts` de type "tableau de `T`", où `T` est un type (n'importe lequel) est : `T[] ts` .

Comme "tableau de `T`" est un type, on peut donc exprimer un type "tableau de tableaux de `T`" directement de la façon suivante : `T[][] ts` .

En ajoutant des espaces pour indiquer visuellement le groupement : "`ts` est un tableau de tableaux de `T`" s'écrit (de droite à gauche !) : `T[] [] ts` .

Par exemple `int[][] ts` déclare `ts` comme étant de type tableau d'entiers.
On remarque que :

— la taille des tableaux (nombre de "cases") ne fait **pas** partie du type. Tous les tableaux contenant le même type d'éléments sont de même type quelque soit leur taille.

— la déclaration **ne crée pas de tableau** : il faut par ailleurs créer le tableau (avec `new`), en indiquant sa taille.

Pour le dernier point, il devient évident lorsque l'on réalise qu'en fait dire que "`ts` est un tableau de `T`" est un abus de langage. Pour être absolument exact, il faudrait dire que "`ts` est une *référence* sur un tableau de `T`".

4.2 types "références", identité et égalité

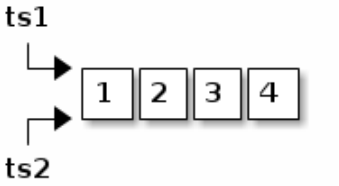
En Java, tous les types qui ne sont pas des types primitifs (`int`, `float`, `char`, `boolean`, etc.), c'est-à-dire :

— les tableaux (i.e. `int[][] ts`)

— les classes (i.e. `String str`)

sont en fait toujours manipulés à travers des *références* et non pas directement. Une référence, c'est ce qui permet d'accéder à une valeur stockée quelque part dans la mémoire de l'ordinateur. Deux références **égales** désignent donc des valeurs **identiques** (en fait, donc, une seule valeur à **un** endroit en mémoire).

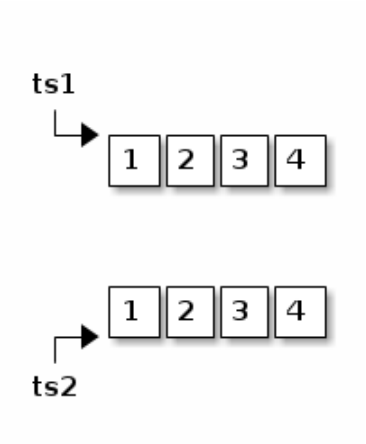
```
1  int [] ts1= {1, 2, 3, 4};
2  int [] ts2= ts1;
```



Si l'on veut des tableaux **égaux**, et non pas **identiques**, il faut faire une copie :

```
1  int [] ts1= {1, 2, 3, 4};
2  int [] ts2= new int[ts1.length];
```

```
3   ts2[0]= ts1[0];
4   ts2[1]= ts1[1];
5   ts2[2]= ts1[2];
6   ts1[3]= ts2[3];
```



En fait, on utilisera bien évidemment une boucle (`for`, cf §5.2) pour éviter les répétitions [5]

Les différences entre des valeurs **identiques** et des valeurs **égales** sont les suivantes :

— la modification d'une des valeurs identiques modifie aussi l'autre valeur, puisqu'en fait il s'agit d'une seule et même valeur.

— la copie d'une valeur pour obtenir une autre valeur, égale mais non identique à la première, est coûteuse en mémoire et en temps (et ceci de façon proportionnelle à la taille de l'objet ou du tableau à copier, cf. §9.1)

C'est parce que les types primitifs sont de petite taille qu'ils sont manipulés directement par valeur et donc copiés à chaque fois. En revanche, les objets et tableaux peuvent être de grande taille, d'où le fait qu'ils soient manipulés par références et que seules les références sont copiées, par exemple lors des passages d'arguments, comme le montre le code ci-après :

```
1   public class ExampleReferences {
2
```

5. ou même une fonction dédiée de la bibliothèque standard `java.lang.System.arraycopy` pour faire les copies, voir une autre fonction `java.util.Arrays.copyOf` pour faire à la fois la création du nouveau tableau et la copie des éléments.

```
3       public static void f(int[] a, String b){
4           a[0]= 0; // changes the array received as argument
5       }
6
7       public static void main(String[] args){
8           int [] ts= {1, 2, 3, 4};
9           String str= "test";
10          f(ts, str);
11          System.out.println("ts[0]="+ts[0]);
12          g(ts, str);
13          System.out.println("ts[0]="+ts[0]);
14          System.out.println("str="+str);
15      }
16      public static void g(int[] c, String d){
17          c= new int[2];
18          d= "other";
19      }
20  }
```

Il faut donc bien faire attention au fait que des valeurs reçues par copie de références peuvent être modifiées dans une fonction appelée à travers la références, comme c'est le cas pour le tableau `ts` après l'appel de la fonction `f`. Heureusement, les chaînes de caractères, objets de la classe `String` sont *immutables*, c'est-à-dire qu'aucune des méthodes de la classe String ne permet de modifier l'objet. On peut donc passer des chaînes de caractères en argument de n'importe quelle fonction/méthode sans avoir à s'inquiéter que la chaîne soit modifiée.

Remarquons, dans la fonction `g`, que le fait de modifier la référence elle-même (par l'affectation d'une autre références vers une autre valeur) ne modifie pas la référence passée en argument.

4.3 Égalité de références, identité de valeurs

Il est important de comprendre que l'opérateur binaire == qui teste l'égalité est appliqué aux **références** pour les types références (tableaux et classes). Il ne teste donc pas si les valeurs sont **égales**, mais si elles sont **identiques**.

Deux valeurs sont identiques (il n'y a donc en fait qu'une seule valeur) si les références vers ces valeurs sont égales.

Pour tester si deux valeurs sont égales, sans être forcément identiques, il faut :

— pour les objets, utiliser la méthode `equals`
— pour les tableaux, tester l'égalité de chacune des cases (ce que fait la fonction java.util.Arrays.equals)

Attention ! Malgré son nom, la méthode `equals` teste par défaut l'identité pour une classe que l'on définit soi-même. En effet, l'égalité ne peut

qu'être spécifique à chaque classe et il faudra donc prendre soin de définir cette méthode soi-même. Pour les classes déjà existantes, et notamment la classe `String`, cette méthode a été spécifiée pour tester l'égalité, évidemment.

5 Rappels sur les boucles

On peut répéter l'exécution d'une ou plusieurs instructions à l'aide de *boucles*. En général, on répétera un ensemble d'instructions regroupées au sein d'un *bloc* délimité par des accolades, même si ce bloc ne contient qu'une seul instruction. Il faut faire particulièrement attention de ne pas mettre de `;` avant un bloc à répéter, car `;` est une instruction valide (qui ne fait rien) ! La répétition porterait donc sur cette seule instruction vide et non sur le bloc qui la suivrait.

5.1 Rappel sur les boucles `while`

Il y a deux formes de boucles `while`.

5.1.1 while(){}

La forme la plus élémentaire de répétition est la boucle `while` de la forme :

```
1  while(expressionBooleenne){
2    // bloc d'instructions à répéter
3    // tant que l'expression précédente booléenne est vraie
4  }
```

L'expression booléenne entre parenthèses après le mot clé `while` est souvent un test, mais ce peut être n'importe quoi ayant une valeur booléenne (une variable de type `boolean`, un appel de fonction retournant une valeur de type `boolean`, une combinaison de valeurs booléennes par des opérateurs logiques comme `||` ou/et `&&`, etc.).

Si l'on veut que la boucle prenne fin, il faut que l'exécution des instructions dans la boucle puisse modifier le résultat de l'évaluation de l'expression booléenne de condition de continuation, sinon celle-ci pour rester `true` et boucler éternellement.

Il faut noter que s'il l'expression vaut `false` initialement, le bloc n'est pas exécuté du tout et l'exécution passe directement après la fin du bloc.

5.1.2 do {} while() ;

Il arrive que l'on veille exécuter un bloc de code au moins une fois, et que ce soit seulement après une exécution du bloc que l'on puisse décider de recommencer ou non à exécuter ce bloc. Dans ce cas on utilise la forme suivante :

```
do{
  // bloc d'instructions à répéter
  // tant que l'expression booléenne suivante est vraie
}while (expressionBooleenne); // ne pas oublier le ;
```

5.2 for(; ;){}

Souvent, on utilise en fait un idiome de la forme :

```
{
    INITIALISATION;
    while(TEST de continuation){
        CODE;
        Mise À Jour pour itération suivante;
    }
}
```

Afin de rendre cet idiome plus lisible, on dispose de la boucle `for` qui permet de regrouper

— l'initialisation (ou les initialisations)
— le test de continuation
— la mise à jour (ou les mises à jours) pour l'itération suivante

```
for(INITIALISATION ; TEST de continuation; MAJ pour itération suivante){
    CODE;
}
```

Ce type de boucles est particulièrement utile pour parcourir toutes les valeurs d'un tableau à l'aide d'un indice.

— on initialise l'indice à 0
— on teste si l'indice est toujours valide (il ne l'est plus lorsqu'il vaut la taille du tableau, le dernier indice valide étant la taille du tableau -1)
— on passe à la case suivante en *incrémentant* d'indice

```java
for(int i=0; i != ts.length; i=i+1){
    // faire quelque chose avec ts[i]
}
```

On peut remplacer `i= i+1` par `++i` ou `i++` pour réaliser l'incrémentation [6].

```java
for(int i=0; i != ts.length; ++i){
    // faire quelque chose avec ts[i]
}
```

5.3 for(:){}

Si l'on veut traiter successivement chacun des éléments d'une *collection*, par exemple un tableau, il existe une deuxième forme de boucle `for` dédiée à cela :

```java
for(ElementType e : collection){
   // code utilisant e
}
```

La `collection` doit contenir des éléments de type `ElementType`. La variable `e` prend chacune des valeurs de la collection indiquée après les deux points. Par exemple le fragment de code suivant affiche successivement chacune des chaînes de `strs` :

```java
String[] strs={"chaîne 0", "chaîne 1", "chaîne 2"};

for(String s : strs){
    System.out.println(s);
}
```

6 Traitement d'un ensemble de valeurs

Soit un fragment de code (par exemple le corps d'une fonction) qui doit calculer un résultat dépendant d'un ensemble de valeurs (par exemple leur somme pour des nombres, ou leur concaténation pour des chaînes de caractères). Généralement, on ne peut pas calculer le résultat directement sur

6. La préincrémentation `++i` et la postincrémentation `i++` *font* la même chose (incrémenter `i`) mais ne *valent* pas la même chose : `++i` vaut la valeur **après** l'incrémentation et `i++` vaut la valeur **avant** incrémentation. Pareillement pour les opérateurs de prédécrémentation et postdécrémentation qui permettent d'écrire `--i` et `i--`.

l'ensemble, mais on peut directement mettre à jour un résultat partiel pour prendre en compte un nouvel élément.

La démarche est alors la suivante :

1. déclarer une valeur (par exemple `result`) avec le résultat correspondant au traitement d'un ensemble vide de valeurs (par exemple `0` pour une somme d'entiers, la chaîne vide `""` pour une concaténation de chaînes de caractères)
2. faire une boucle sur chacune des valeurs à traiter en mettant à jour le résultat pour prendre en compte cette valeur.
3. après la fin de la boucle (qui peut n'avoir jamais itéré si l'ensemble à traiter était vide), `result` contient donc le résultat final prenant en compte toutes les valeurs à traiter. Il peut être retourné le cas échéant.

6.1 Exemple de traitement complet des valeurs d'un tableau

Le cas le plus simple du traitement toujours intégral de toutes les valeurs d'un tableau peut donc être illustré par les fonctions suivantes :

```
1  public class ProcessAllElements{
2
3      public static int sum(int[] vs){
4          int result=0;
5          for(inti=0; i != vs.length; ++i){
6              result= result + vs[i];
7          }
8          return result;
9      }
10     public static int concatenateln(String[] vs){
11         int result="";
12         for(inti=0; i != vs.length; ++i){
13             result= result + "\n" + vs[i];
14         }
15         return result;
16     }
17     public static void main(String[] args){
18         int[] ts={1,2,3,4};
19         System.out.println("somme :"+sum(ts));
20         String[] strs={"ligne 0", "ligne 1", "ligne 2"};
21         System.out.println("concatenation:"+concatenateln(strs));
22     }
23 }
```

6.2 Exemple de traitement avec *early exit* des valeurs d'un tableau

Dans certains cas, on a pas forcément besoin de traiter tous les éléments et l'on peut interrompre le traitement avant la fin (*early exit* en anglais). Par exemple, si l'on veut retourner une valeur booléenne indiquant si le tableau contient une valeur strictement négative. En effet, dès que l'on trouve une telle valeur, on peut interrompre le parcours et gagner en temps d'exécution.

La version naïve qui parcours inutilement toujours tous les éléments est la suivante :

```
1  public class EarlyExits{
2
3      public static double[] readDoubles(String prompt){
4          System.out.println(prompt);
5          System.out.println("Combien de valeurs ?");
6          Scanner sc= new Scanner(System.in);
7          int n= sc.nextInt();
8          System.out.println("Entrez les "+ n +" valeurs:")
9              double [] res= new double[n];
10         for(int i=0; i != res.length; ++i){// i=i+1
11             res[i]= sc.nextDouble();
12         }
13         return res;
14     }
15
16     // no early exit
17     public static boolean containsNegativeValues(double[] vs){
18         boolean result= false;
19         for(int i=0; i != vs.length; ++i){
20             if(vs[i]<0){
21                 result= true;
22             }// else we must not set result to false !
23         }
24         return result;
25     }
26
27     public static void main(String[] args){
28         double[] ts= readDoubles("Saisissez les températures:");
29         if(containsNegativeValue(ts)){
30             System.out.println("Il y a une ou plusieurs valeur(s) négative(s) !");
31         }
32     }
33
34 }
```

A priori, on peut éviter les tests inutiles (dès que l'on trouve une valeur négative) de (au moins !) trois façons différentes.

6.2.1 Utiliser une instruction `break;` pour sortir de la boucle à l'intérieur de celle-ci

```
1  // early exit with break
2  public static boolean containsNegativeValues(double[] vs){
3      boolean result= false;
4      for(int i=0; i != vs.length; ++i){
5          if(vs[i]<0){
6              result= true;
7              break;
8          }// else we must not set result to false !
9      }
10     return result;
11 }
```

6.2.2 Faire directement le `return` dans la boucle pour sortir directement de la fonction (et donc a fortiori de la boucle !)

```
1  // early exit with return
2  public static boolean containsNegativeValues(double[] vs){
3      for(int i=0; i != vs.length; ++i){
4          if(vs[i]<0){
5              return true;
6          }// else we must not return early with false !
7      }
8      return false;// if we get here no value was < 0
9  }
```

6.2.3 Modifier la condition de continuation pour prendre aussi en compte cette condition de terminaison.

```
1  // early exit with modified condition
2  public static boolean containsNegativeValues(double[] vs){
3      boolean result= false;
4      for(int i=0; (result == false) && (i != vs.length); ++i){
5          if(vs[i]<0){
6              result= true;
7          }// else we must not set result to false !
8      }
9      return result;
10 }
```

On préférera une version qui modifie la condition de continuation, car ainsi celle-ci "ne ment pas" et indique bien tous les cas dans lesquels on sort de la boucle sans qu'il soit besoin de lire toutes les instructions de celle-ci.

On peut écrire une version plus élégante de l'expression booléenne avec l'opérateur booléen de négation : `!`. On peut se convaincre qu'elle est équiva-

lente en calculant les valeurs de `result==false` et `!result` pour toutes les valeurs de `result` (il n'y en a que deux : `true` et `false`).

```
// early exit with modified condition (more elegant)
public static boolean containsNegativeValues(double[] vs){
    boolean result= false;
    for(int i=0; !result && (i != vs.length); ++i){
        if(vs[i]<0){
            result= true;
        }// else we must not set result to false !
    }
    return result;
}
```

7 Tableaux : limitations

7.1 Taille fixe

La principale limitation, et la plus évidente, est qu'il faut connaître à l'avance la taille du tableau. Ainsi dans la fonction `readDoubles` du programme `FahrenheitToCelsius`, on doit commencer par demander le nombre de valeurs qui seront saisies par l'utilisateur/trice.

On voudrait pouvoir ajouter des éléments au fur et à mesure, ne serait-ce que parce qu'il est parfois impossible de savoir à l'avance combien il y en aura. Par exemple, on pourrait vouloir stocker des informations sur chaque tour de jeu d'une partie jusqu'à ce qu'un des joueurs gagne.

Il est possible d' "ajouter une valeur à un tableau" de n éléments en créant un nouveau tableau de $n+1$ cases et en recopiant les n cases avant d'ajouter la $n+1^{ème}$ valeur.

```
public static int[] add(int[] xs, int x){
    int[] result= int[xs.length+1];
    for(int i=0; i != xs.length; ++i){
        result[i]= xs[i];
    }
    result[xs.length]= x;
    return result;
}
```

L'ajout en début de tableau, ou à n'importe quelle position, est laissé en exercice.

Suivant le même principe, il est aussi possible d'enlever un élément, en début, en fin ou à n'importe quelle position :

```java
// returns the removed element. Should throw an IndexOutOfBoundsException
// if the index of the element to remove is <0 or > length
public static int[] remove(int[] xs, int toRemoveIdx){
    int res= new int[xs.length-1];
    for(int src=0, dest=0; (src != xs.length) && (dest != res.length); ++src){
        if(src != toRemove){
            res[dest]= xs[src];
            ++dest;
        }
    }
    return res;
}
```

Cet exemple permet de montrer que la partie initialisation de la boucle `for` peut contenir plusieurs initialisations (mais on ne peut déclarer que des variables du même type), séparées par des virgules `,`.[7]

De même, on aurait pu l'écrire de façon à montrer qu'il est possible d'avoir plusieurs mises à jour pour l'itération suivante en incrémentant toujours `dest` et en le décrémentant aussi lorsque `src` est égal à `toRemove`, de façon à laisser `dest` alors inchangé :

```java
// returns the removed element. Should throw an IndexOutOfBoundsException
// if the index of the element to remove is <0 or > length
public static int[] remove(int[] xs, int toRemoveIdx){
    int res= new int[xs.length-1];
    for(int src=0, dest=0; (src != xs.length) && (dest != res.length); ++src, ++dest){
        if(src != toRemove){
            res[dest]= xs[src];
        }else{
            --dest;
        }
    }
    return res;
}
```

Ces solutions ne sont pas vraiment satisfaisantes, pour des raisons expliquées en §9. La solution plus générale consistera à utiliser d'autres *structures de données* pour ajouter et enlever des valeurs facilement et *efficacement*.

7.2 Solution au déplacement de valeurs dans un tableau : échanges

On peut aussi vouloir déplacer des valeurs dans un tableau, par exemple pour mettre la valeur minimale d'un tableau en début de tableau, ou pour trier complètement un tableau.

7. On aurait aussi pu écrire `res[dest++] = xs[src]`, mais la concision peut devenir un défaut lorsqu'elle est poussée à l'excès.

Comme on vient de le voir, on ne peut pas directement (efficacement) enlever une valeur puis l'insérer ensuite à la place désirée. Cependant, il est possible de "déplacer" efficacement une valeur d'un tableau en effectuant un échange avec une autre case du tableau.

7.2.1 Échange de deux valeurs

Soit deux variables `a` et `b`, quelle séquence d'instructions permet d'échanger leurs valeurs ? Le fragment de code ci-après ne permet pas d'effectuer l'échange à cause de l'enchaînement des affectations :

```
1  int a= 0;
2  int b= 1;
3
4  a= b;
5  b= a;
6
7  System.out.println("a= "+a+" et b= "+ b);// a= 1 et b= 1 !
```

En effet, c'est la nouvelle valeur de `a`, après que l'on a exécuté `a= b;`, qui est affectée à `b`.

La solution est d'utiliser une troisième variable (temporaire), pour permettre l'échange :

```
1  int a= 0;
2  int b= 1;
3
4  int tmp= a;
5  a= b;
6  b= tmp;
7
8  System.out.println("a= "+a+" et b= "+ b);// a= 1 et b= 0
```

Ceci devient intéressant pour échanger les valeurs de deux cases d'un tableau :

```
1  public static void swap(int[] xs, int i, int j){
2      int tmp= xs[i];
3      xs[i]= xs[j];
4      xs[j]= tmp;
5  }
```

L'utilisation de cette fonction pour mettre en début ou fin de tableau la valeur la plus petite ou la plus grande, voire pour trier complètement un tableau, est laissée en exercice.

8 Exemple pratique des limitations des tableaux

On veut récupérer deux listes de noms (un nom par ligne, avec répétitions) et effectuer les opérations suivantes :

- récupérer chacune des listes à partir de l'URL du fichier, en implémentant la fonction `readNames`
- calculer l'intersection, c'est-à-dire l'ensemble des noms (sans doublons) sont à la fois dans les deux listes, en implémentant la fonction `intersection`
- trouver quel est le nom le plus commun (celui ayant le plus de répétitions) dans l'une des listes, en implémentant la fonction `mostCommon`

Le code ci-après est une ébauche à compléter. Seule la fonction `readNames` est partiellement implémentée : elle se contente d'afficher chaque nom (ligne) du fichier distant au lieu de les stocker pour pouvoir les retourner en valeur de retour.

```
1  import java.io.FileReader;
2  import java.io.BufferedReader;
3
4  import java.io.FileNotFoundException;
5  import java.io.IOException;
6
7  import java.io.InputStreamReader;
8  import java.net.URL;
9
10 public class ExampleDataStructures {
11     public static String[] readNames(String filename) throws FileNotFoundException, IOException
12     {
13         URL url= new URL(filename);
14         try(BufferedReader br = new BufferedReader(new InputStreamReader(url.openStream()))) {
15                 for(String line = br.readLine(); line != null; line= br.readLine()){
16                     System.out.println(line);
17                 }
18             }
19         return null;
20     }
21     public static String[] intersection(String [] arr1, String[] arr2){
22         return null;
23     }
24     public static String mostCommon(String [] arr){
25         return null;
26 }
27
28     public static void main(String[] args){
29         try{
30             String base="https://raw.githubusercontent.com/bhugueney/II.2407/master/docs/Data/";
31             String[] names1= readNames(base+"liste-des-gares-1.txt");
32             String[] names2= readNames(base+"liste-des-gares-2.txt");
33             String[] names1and2= intersection(names1, names2);
34             String mostCommonName1= mostCommon(names1);
```

```
35              System.out.println(mostCommonName1);
36          }catch(FileNotFoundException e){
37              System.err.println(e);
38          }
39          catch(IOException e){
40              System.err.println(e);
41          }
42      }
43  }
```

9 Introduction à la complexité algorithmique

Comme on l'a vu en §7.1 , et comme on a vu le mettre en pratique en §8, on peut ajouter des éléments un par un en créant à chaque fois un nouveau tableau et en recopiant le contenu de l'ancien. Cela peut cependant poser des problèmes d'efficacité.

9.1 Complexité algorithmique linéaire

En effet, plus le nombre d'éléments du tableau est grand, plus l'ajout d'un seul élément va prendre de temps puisqu'il faut tous les recopier. On dit que l'ajout d'un élément est alors de *complexité algorithmique linéaire*. La *complexité algorithmique* désigne le nombre d'étapes en fonction du nombre d'éléments à traiter. Dans le cas de l'ajout (ou du retrait) d'un élément, on a donc une complexité *linéaire* notée $O(n)$: traiter deux fois plus d'éléments demandera deux fois plus de temps, en traiter 10 fois plus demandera 10 fois plus de temps, etc. C'est le cas le plus classique lorsqu'il faut faire un traitement élémentaire (i.e. de complexité algorithmique constante, cf. infra.) sur chacun des éléments [8].

9.2 Complexité algorithmique constante

En revanche, accéder simplement (en lecture ou en écriture) à n'importe quelle case d'un tableau, en exemple en écrivant `ts[i]` pour accéder à la case d'indice `i` du tableau `ts` se fait une seule "étape", quel que soit le nombre de cases du tableau. On parle alors de complexité algorithmique *constante*, notée $O(1)$. Évidemment, c'est le cas idéal.

8. ou sur une proportion constante de ceux-ci, par exemple la moitié, ou le tiers.

9.3 Complexité algorithmique quadratique

Dans le cas de l'ajout successivement de n éléments, on a vu que chacun des ajouts était de complexité algorithmique linéaire ($O(n)$), mais il faudra répéter n fois l'opération d'ajout. Au total, on aura donc $1 + 2 + \cdots + n$ opérations, c'est-à-dire $\sum_{i=1}^{n} i = \frac{(n+1)^2}{2}$ opération. Ce qui compte, c'est qu'intuitivement on réalise que l'on fait de l'ordre de n fois n opérations. Il s'agit donc d'une complexité algorithmique *quadratique* notée $O(n^2)$. Traiter 2 fois plus d'éléments demandera 4 fois plus de temps, traiter mille fois plus d'éléments demandera un million de fois plus de temps. On réalise qu'il ne sera pas possible de traiter ainsi des millions d'éléments !

9.4 Complexité algorithmique logarithmique

Lorsque l'on cherche un mot dans un dictionnaire, on ne va pas commencer par le premier mot pour les regarder ensuite les uns après les autres jusqu'à trouver le mot cherché car cela serait beaucoup trop long ! On effectue une recherche *dichotomique* en utilisant le fait que les mots du dictionnaire sont dans l'ordre alphabétique. On peut commencer par ouvrir un dictionnaire au milieu et regarder sur le mot au milieu [9]. Selon que ce mot soit situé avant ou après le mot rechercher, on peut éliminer la moitié des mots. On peut répéter l'opération sur la moité restante, puis sur le quart, etc. Le nombre d'opérations, et donc le temps nécessaire, est alors lié au logarithme du nombre d'éléments. On parle de complexité algorithmique *logarithmique*, notée $O(\log n)$. Traiter deux fois plus d'éléments ne demandera qu'une étape de plus, en traiter quatre fois plus demandera deux étapes de plus, en traiter 1024 fois plus demandera 10 étapes de plus, traiter un million de fois plus d'éléments ne demandera que 20 étapes de plus. C'est une complexité algorithmique correspondant à des algorithmes très efficaces qui traitent une proportion des éléments (ici la moitié, mais ce pourrait être le quart, le dixième ou le centième) en une seule étape.

10 Structures de données

Si les tableaux permettent de répondre aux problèmes vus en §8, ils ne permettent pas de le faire de façon efficace. En effet, la lecture des données est en $O(n^2)$, l'intersection faite naïvement serait elle aussi en $O(n^2)$ et de même pour la recherche de l'élément le plus commun (pour chaque nom, on parcourt

9. la *médiane* de l'ensemble trié dans l'ordre alphabétique

un ensemble de noms). Ceci ne permettrait pas de traiter efficacement un grand nombre de données.

Pour résoudre ces problèmes, on va utiliser des *structures de données* adaptées. Ce seront des objets, instances de classes spécialement conçues à cet effet. Comme java est statiquement typé, on voudra pouvoir déclarer le type d'éléments contenus dans les structures de données. Pour cela, on utilise des classes dites *génériques* paramétrées par le type contenu. Ainsi, `ArrayList<String>` désigne une classe qui permet de stocker des objets de classe `String`, alors que `ArrayList<Integer>` désigne une classe qui permet de stocker des objets de classe `Integer`. On ne peut pas paramétrer avec des types primitifs mais seulement avec des types référence (tableaux ou classes, donc), c'est pourquoi on ne pourrait pas avoir `ArrayList<int>` et l'on doit utiliser la classe correspondante (ici `Integer`).

Pour les structures de données comme pour toutes les (nombreuses !) classes de bibliothèques en java, il n'est bien sûr pas utile de chercher à connaître par cœur toutes les méthodes. Il faut savoir quelles classes existent et à quoi elles servent pour pouvoir se référer à la documentation.

Il est possible d'accéder à chacun des éléments de ces structures de données en tant que *collections* avec la boucle `for(:)` telle que vue en §5.3.

10.1 Liste

Le fait de pouvoir ajouter et enlever des éléments efficacement correspond au concept de liste[10] et la bibliothèque standard Java propose une implémentation sous la forme de la classe java.util.ArrayList.

On peut notamment :

— créer une liste vide en appelant le constructeur sans argument avec `new`

— ajouter un élément avec la méthode add

— convertir la liste en tableau d'éléments avec la méthode toArray. À noter que cette méthode prend un tableau en argument.

L'implémentation de `readNames` devient donc simplement :

```
1  public static String[] readNames(String filename) throws FileNotFoundException, IOException
2  {
3      ArrayList<String> res= new ArrayList<String>();
4      URL url= new URL(filename);
5      try(BufferedReader br = new BufferedReader(new InputStreamReader(url.openStream()))) {
```

10. les concepts sont exprimés en Java par des *interfaces*, ici `java.util.List`. Les relations entre classes et interfaces seront vues plus tard avec la notion d'héritage en Programmation Orientée Objet.

```
            for(String line = br.readLine(); line != null; line= br.readLine()){
                res.add(line);
            }
        }
    return res.toArray(new String[res.size()]);
}
```

Bien sûr, il faudra au préalable avoir importé la classe au début du fichier avec :

```
import java.util.ArrayList;
```

10.2 Ensemble

Pour calculer une intersection, il faut tester si les élément d'un ensemble appartiennent à un autre ensemble, et pour éviter les doublons, il faut aussi pouvoir détecter si l'ensemble des résultats contient déjà la valeur à y mettre.

On comprend qu'il serait possible de faire la recherche plus efficacement si les données sont triées [11], mais se pose alors la question du maintient efficace de l'ordre trié si l'on ajoute des éléments.

Le concept répondant à ce besoin est celui d'*ensemble* (`Set` en anglais) et la bibliothèque standard Java propose une implémentation sous la forme de la classe java.util.HashSet.

Cette classe permet notamment de :

- créer un ensemble vide en appelant le constructeur sans argument avec `new`
- ajouter un élément avec la méthode add
- tester l'appartenance d'une valeur avec la méthode contains
- convertir la liste en tableau d'éléments avec la méthode toArray. À noter que cette méthode prend un tableau en argument.

Ce n'est évidemment pas un hasard si les méthodes qui font la même chose que pour la liste portent le même nom.

L'implémentation d'`intersection` devient donc simplement :

```
public static String[] intersection(String [] arr1, String[] arr2){
    HashSet<String> set1= new HashSet<String>();// could be better
    for(int i=0; i != arr1.length; ++i){
        set1.add(arr1[i]);
    }
    HashSet<String> res= new HashSet<String>();// could be better
    for(int i=0; i != arr2.length; ++i){
```

11. ou s'il y a un index

```
8           if (set1.contains(arr2[i])){
9               res.add(arr2[i]);
10          }
11      }
12      return res.toArray(new String[res.size()]);
13  }
```

En ayant au préalable importé la classe `java.util.HashSet`.

Attention ! Pour savoir si un ensemble contient déjà une valeur, c'est (logiquement) l'**égalité** et non l'identité qui est prise en compte !

10.3 Table d'association

Le cas de la recherche du nom le plus fréquent est un peu plus complexe, car pour chaque nom on voudra compter combien de fois il apparaît et donc **associer** un compter (entier) à chaque nom (chaîne de caractères). Le concept qui permet d'exprimer (associer une valeur à une clé) cela est celui de *table d'association* (`Map` en anglais, avec un type `Value` associé à un type `Key`) et la bibliothèque standard Java propose une implémentation avec la classe java.util.HashMap.

Cette classe générique est paramétrée par deux types (le type des clés et le type des valeurs), et l'on utilisera donc `HashMap<String, Integer>`.

Cette classe permet notamment de :

- créer une table d'association vide en appelant le constructeur sans argument avec `new`
- tester si la table contient une valeur associée à une certaine clé avec la méthode containsKey
- associer une valeur à une clé avec la méthode put
- retourner la valeur associée à une clé donnée avec la méthode get
- retourner l'ensemble (donc une *collection*) de couples *(clé, valeur)* stockés dans la table avec la méthode entrySet. Ces couples sont de type `Map.Entry<String, Integer>` qui permet d'appeler sur l'élément les méthodes :
  - getKey pour récupérer la clé du couple
  - getValue pour récupérer la valeur associée à la clé du couple

On implémente donc facilement la fonction `mostCommon` en deux temps :

1. on parcourt le tableau en renseignant une table de d'association qui tient à jour le nombre d'occurrence de chaque nom
2. on parcourt l'ensemble des couple $(nom, nb\ d'occurrences)$ pour trouver le nom qui a le plus grand nombre d'occurrences.

```java
    public static String mostCommon(String [] arr){
        HashMap<String, Integer> tmp= new HashMap<String,Integer>();// TODO
        for(int i=0; i != arr.length; ++i){
            if (tmp.containsKey(arr[i])){
                tmp.put(arr[i], new Integer(tmp.get(arr[i]).intValue()+1));
            }else{
                tmp.put(arr[i], new Integer(1));
            }
        }
        String res=null;
        int max=-1;
        for(Map.Entry<String, Integer> entry : tmp.entrySet()) {
            String name = entry.getKey();
            int value = entry.getValue().intValue();
            if(value > max){
                res= name;
                max=value;
            }
        }
        return res;
}
```

Cela après avoir importé les classes `java.util.HashMap` et `java.util.Map` (cette dernière pour pouvoir utiliser `Map.Entry`).