Licence Sciences et Technologies

Module API

Approche impérative

Méthodes et outils algorithmiques

Juin 2011
Version 5

J. Barré,
P. Le Certen,
L. Ungaro

LICENCE SCIENCES ET TECHNIQUES

MÉTHODES ET OUTILS ALGORITHMIQUES
APPROCHE IMPÉRATIVE (MODULE API)

Version 5 - juin 2011

Jacques Barré, Pascale Le Certen, Lucien Ungaro

CHAPITRE 1 Introduction

Objectif du cours "algorithmique et programmation impérative"

Le but de ce cours est l'apprentissage des bases de l'algorithmique. L'algorithmique est la science des méthodes automatiques de calcul. Le terme "calcul" doit être compris au sens large, à savoir la manipulation d'informations de toutes sortes. Ce module s'intéresse aux algorithmes de style "impératif", c'est-à-dire construits à l'aide d'instructions qui agissent sur des données. Les instructions sont enchaînées au moyens de structures de contrôle usuelles : conditionnelle, itération et appel de fonction. Dans la première partie de ce cours, les types des données manipulées sont limités aux types primitifs simples (nombres entiers, nombres réels, caractères, booléens), aux structures, aux chaînes de caractères et aux tableaux. Les outils plus sophistiqués permettant l'invention de types de données abstraits seront étudiés dans une deuxième partie.
L'algorithmique est non seulement une science, mais aussi une technique. Le but de ce cours est de réaliser effectivement des programmes. Pour cela il faut connaître un langage de programmation. Nous avons choisi Java pour de nombreuses raisons : c'est un langage relativement simple qui permet cependant d'illustrer tous les concepts modernes de programmation. De plus c'est un langage très utilisé par les professionnels et sa distribution est gratuite.

1.1 Algorithmes, langages de programmation, programmes

objectif : *comprendre la notion élémentaire d'algorithme et avoir un aperçu des diverses sortes de langages de programmation.*

- Un ***algorithme*** est la description des opérations qui permettent d'accomplir une tâche. Cette tâche peut être un calcul, une transformation d'information, voire un procédé de fabrication. L'usage d'algorithmes est en fait assez répandu. Un modèle de tricot, une recette de cuisine... sont des algorithmes. Un algorithme doit être une description précise, mais pas nécessairement exprimé dans un langage prédéfini. Tous les moyens d'expression sont acceptables : langue naturelle, agrémentée de mathématiques, de croquis...
- Un ***langage de programmation*** est un langage rigoureux, compréhensible par une machine, qui permet d'exprimer ***sans aucune ambiguïté*** des algorithmes.
- Un ***programme*** est un algorithme exprimé dans un langage de programmation.

1.1.1 Langages de programmation

Il existe de nombreux langages de programmation qui se sont développés au cours des ans en fonction des besoins, de l'évolution des ordinateurs et des méthodes dans le domaine du "génie logiciel".

- Les langages les plus rudimentaires sont les ***langages machines***. Ils utilisent des instructions directement exécutées par l'ordinateur. Ils présentent de nombreux inconvénients :

  - ils sont difficiles à utiliser,

  - ils sont propres à un type de machine,

  - les programmes sont peu lisibles et compréhensibles par les personnes humaines.

- Pour faciliter l'écriture des programmes, on a développé des ***langages évolués*** :

  - ils aident à concevoir les algorithmes,

  - ils sont indépendants des machines utilisées,

  - ils sont plus faciles à comprendre et à vérifier.

  Pour pouvoir être exécutés sur une machine, ils nécessitent une phase de traduction par un ***compilateur***.

  On distingue deux grandes familles de langages évolués :

  - les langages ***déclaratifs*** qui résolvent un problème en utilisant des propriétés de sa solution,

  - les langages ***impératifs*** qui résolvent un problème en indiquant la succession des opérations à effectuer.

  Ce cours est consacré aux langages impératifs. Cependant, pour les situer dans un cadre plus général, nous donnerons quelques aperçus sur les langages déclaratifs.

1.1.2 Algorithmes en langages déclaratifs

Les langages déclaratifs résolvent un problème en utilisant certaines propriétés auxquelles la solution doit satisfaire.

On peut distinguer deux types de langages déclaratifs :

- les langages ***fonctionnels***,
- les langages ***relationnels***.

1.1.2.1 Langages fonctionnels

Un programme en langage fonctionnel utilise des définitions et des applications de ***fonctions.*** L'usage de définitions récursives de fonctions permet de résoudre une très grande classe de problèmes. Dans le cas d'un programme fonctionnel "pur", le résultat d'un programme est toujours le résultat de l'évaluation d'une fonction appliquée à des paramètres.

Comme exemple simple on peut considérer le calcul de la somme des nombres entiers de 1 à ***n*** : 1+2+3+...+**n**. Le résultat est une fonction du nombre entier ***n***. On peut appeler ce résultat ***Sigma(n)***. La fonction ***Sigma*** s'exprime facilement de manière récursive, c'est-à-dire en s'utilisant elle-même avec un paramètre "plus simple" que le sien :

En effet, on a ***Sigma(***1***)*** = 1

et pour ***n***>1, ***Sigma(n)*** = 1+2+...+***n***-1 +***n*** = ***Sigma(n-1)*** + ***n***

Dans un langage fonctionnel (hypothétique) cette fonction pourrait s'exprimer ainsi :

```
fonction Sigma(n) = si n=1 alors 1 sinon Sigma(n-1)+n
```

Si on désire calculer la somme des entiers de 1 à 4, il suffit d'appliquer cette fonction **`Sigma`** avec le nombre 4 en paramètre :

```
Sigma(4)
```

Le résultat affiché par l'exécution est alors : **`10`**

Il aura été calculé ainsi, par applications successives de la définition de la fonction **`Sigma`** :

```
Sigma(4) = si 4=1 alors 1 sinon Sigma(3)+4 = Sigma(3) +4
       = si 3=1 alors 1 sinon Sigma(2)+3 +4 = Sigma(2) +3+4
       = si 2=1 alors 1 sinon Sigma(1)+2 +3+4 = Sigma(1) + 2+3+4
       = si 1=1 alors 1 sinon Sigma(0)+1 +2+3+4 = 1 +2+3+4 = 10
```

Parmi les langages fonctionnels les plus réputés, on peut citer : LISP, SCHEME, CAML.

1.1.2.2 Langages relationnels

Les langages relationnels sont basés sur la logique et les techniques de preuves. Un programme est constitué de trois sortes de choses :

- Une collection de "faits", propriétés ou ***relations*** affirmées concernant certaines données. En reprenant l'exemple du calcul de la somme des nombres entiers de 1 à n, un tel fait serait :
  "***Sigma*** de 1 vaut 1".
- Des implications entre relations :
  "***Sigma*** de **n**-1 vaut **s**" et "***n*** > 1" implique que "***Sigma*** de ***n*** vaut ***s***+***n***".
- Une question à résoudre, par exemple : "que vaut ***Sigma*** de 4 ?".

Dans un langage relationnel, ce programme pourrait s'écrire :

règle 1 : **`Sigma(1,1).`**

règle 2 : **`Sigma(n,s+n) <- n>1 et Sigma(n-1,s)`**

question : **`Sigma(4,x)?`**

L'exécution du programme consiste, en utilisant les faits et les implications, à chercher les solutions s'il en existe. Dans l'exemple proposé, l'exécution trouve une solution : "**`x`**=**`10`**".

`Sigma(4,x)?` la règle **2** s'applique, et **`x`** vaut **`s+4`** à condition que **`Sigma(3,s)`**

`Sigma(3,s)?` la règle **2** s'applique, et **`s`** vaut **`t+3`** à condition que **`Sigma(2,t)`**

`Sigma(2,t)?` la règle **2** s'applique, et **`t`** vaut **`u+2`** à condition que **`Sigma(1,u)`**

`Sigma(1,u)?` la règle **1** s'applique, et **`u`** vaut **`1`**

donc **`t`** vaut **`3`**, donc **`s`** vaut **`6`**, donc **`x`** vaut **`10`**

Le problème a une solution **`x`**=**`10`**.

Le plus connu des langages relationnels s'appelle PROLOG.

1.1.3 Algorithmes en langages impératifs

Pour effectuer un calcul en langage impératif, on indique une succession de transformations d'état qui permet de passer d'un état initial à un état final contenant la solution.

Exemple simple d'algorithme :

calcul de la somme des nombres entiers de 1 à ***n*** :

$$1+2+3+...+\boldsymbol{n}.$$

Description de l'algorithme en style impératif :

- utiliser une mémoire ***i*** initialisée à 1 (pour énumérer les nombres 1, 2... ***n***)
- utiliser une mémoire ***somme*** initialisée à 0 (pour cumuler les valeurs successives de ***i***)
- tant que ***i*** est inférieur ou égal à ***n*** : ajouter ***i*** à ***somme***, ajouter 1 à ***i***
- le résultat est la valeur de ***somme***.

Exemple de programme :

En langage Java, voici la fonction ***Sigma*** qui calcule cette somme selon ce schéma :

```
int Sigma(int n){
// prérequis : n>=0
// résultat : la somme des nombres entiers de 1 à n
  int i=1; int somme=0;
  while(i<=n) {
      somme=somme+i;
      i=i+1;
  }
  return somme;
}
```

Exemple d'exécution de programme :

Voici comment se déroulera l'exécution de cette fonction pour n=4 :

lignes de programmes exécutées	*état des variables*
`int i=1; int somme=0;`	`somme=0, i=1`
`while(i<=n){somme=somme+i; i=i+1;}`	`1<=4` donc `somme=1, i=2`
`while(i<=n){somme=somme+i; i=i+1;}`	`2<=4` donc `somme=3, i=3`
`while(i<=n){somme=somme+i; i=i+1;}`	`3<=4` donc `somme=6, i=4`
`while(i<=n){somme=somme+i; i=i+1;}`	`4<=4` donc `somme=10, i=5`
`while(i<=n){}`	`5>4` donc `somme` et `i` inchangés
`return somme;`	résultat : `10`

1.2 Rigueur, syntaxe et sémantique

objectif : *comprendre la rigueur exigée par l'activité de programmation.*

La conception de programmes exige une ***rigueur absolue***.

Pour assurer cette rigueur, les langages de programmation ont une ***syntaxe*** et une ***sémantique*** précises :

- La ***syntaxe*** est l'ensemble des règles de construction que doivent respecter les textes des programmes.
- La ***sémantique*** est la signification des constructions du langage.

Un programme doit d'abord respecter la syntaxe, sinon on ne peut lui attribuer aucune signification.

Dans l'exemple précédent :

```
while(i<=n) {somme=somme+i; i=i+1;}
```

utilise la syntaxe correcte du langage Java pour exprimer une répétition d'opérations.

La signification donnée à cette construction est (informellement) :

"tant que ***i*** est inférieur ou égal à ***n*** : ajouter ***i*** à somme, ajouter 1 à ***i***"

En revanche, la forme :

```
while(i<=n) {somme=somme+i i=i+1}
```

comporte une ***erreur de syntaxe*** : le langage exige un "**`;`**" à la fin de chaque instruction élémentaire.

Un programme peut être correct au niveau syntaxique sans pour autant calculer ce que l'on désire. Il y a alors une ***erreur sémantique*** (ou ***erreur de conception***).

Ainsi la forme :

```
while(i<n) {somme=somme+i; i=i+1;}
```

est syntaxiquement correcte, mais cela ne calcule pas ce que l'on veut car cette version arrête le cumul sur ***n***-1 à cause de l'inégalité stricte utilisée dans le test "**`i<n`**".

Les notions de syntaxe et de sémantique se rencontrent dans tout langage. Comme exemple imagé, on peut considérer les phrases suivantes exprimées dans un langage de physique élémentaire :

- "liège plomb le" : cette phrase est une erreur de syntaxe, on ne peut lui attribuer aucune signification.
- "le liège flotte" : cette phrase est syntaxiquement correcte. Elle signifie que le liège flotte (il est moins dense que l'eau). De plus elle exprime une chose exacte. Elle est sémantiquement correcte (dans la mesure où l'exactitude est la valeur qui nous intéresse).
- "le plomb flotte" : cette phrase est syntaxiquement correcte. Elle signifie que le plomb flotte. Elle exprime une chose inexacte. On peut considérer que c'est une erreur sémantique.

Les erreurs de syntaxe ne sont jamais graves car un compilateur les signale avant toute exécution.

Les erreurs de conception sont plus difficiles à corriger car elles ne se manifestent pas toujours et elles peuvent nécessiter une phase de mise au point longue et fastidieuse.

1.3 Résolution d'un problème

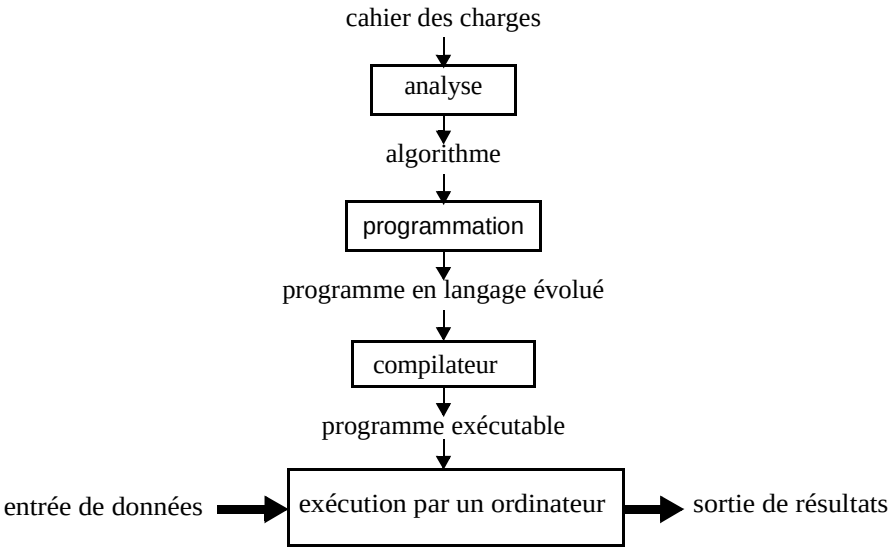
objectif : *comprendre la nécessité de bien analyser un problème avant de rédiger un programme censé le résoudre*

1.3.1 Phase d'analyse et phase de programmation

Pour éviter les erreurs de conception, on ne peut pas se lancer directement dans la programmation. On distingue deux phases principales dans la résolution d'un problème, une fois celui-ci énoncé clairement dans un "cahier des charges" :

- ***L'analyse*** : elle consiste à exprimer la résolution du problème sous une forme la moins ambiguë possible compréhensible par un être humain.
- ***La programmation*** : elle exprime cette solution en termes d'objets informatiques et d'instructions dans un langage de programmation.

La démarche pour résoudre un problème peut s'illustrer comme suit :



1.3.2 Méthodes d'analyse

Il existe plusieurs méthodes d'analyse, plus ou moins adaptées à la complexité et à la nature des problèmes.

1.3.2.1 Application des mathématiques :

Pour des problèmes qui consistent à "calculer" quelque chose, l'analyse consiste souvent à utiliser des résultats connus de mathématiques : équations, suites convergentes... De ce point de vue, l'algorithmique est une application directe des mathématiques, limitée à ce qui peut se calculer effectivement (mathématiques "constructives").

1.3.2.2 Décomposition fonctionnelle :

Pour des problèmes peu complexes qui ne s'appuient pas sur des mathématiques traditionnelles, on peut procéder par ***décomposition fonctionnelle*** : on décompose le problème P en sous problèmes

P_1, P_2, P_3... P_n que l'on cherche à résoudre séparément. La décomposition se poursuit jusqu'à l'obtention de sous-problèmes trivialement résolus. Les résolutions des sous-problèmes sont réalisées par des ***procédures*** que l'on regroupe généralement par thèmes dans des ***modules***.

Comme exemple imagé d'une telle décomposition, supposons que nous ayons à déterminer l'itinéraire routier pour aller de Rennes à Quimper. Une première analyse conduit à :

(P_1) Rennes → Ploërmel

(P_2) Ploërmel → Lorient

(P_3) Lorient → Quimper

Le sous-problème P_1 peut lui même être décomposé en :

(P_{11}) Rennes → Mordelles

(P_{12}) Mordelles → Plélan-le-Grand

(P_{13}) Plélan-le-Grand → Ploërmel

La décomposition se poursuit plus ou moins loin. Tout dépend des possibilités du langage utilisé et de procédures déjà existantes dans les bibliothèques.

1.3.2.3 Analyse avec des objets :

Pour des problèmes plus complexes, il faut utiliser une ***méthode d'analyse par objets*** qui consiste à ***spécifier de façon abstraite les données manipulées et les opérations qu'elles peuvent subir*** et ensuite à réaliser ces abstractions au moyen des types de données offerts par le langage ou que l'on sait réaliser.

1.3.3 Qualités d'un programme

Les qualités essentielles d'un programme sont en premier lieu d'être ***correct*** (faire ce que sa spécification a prévu) et en second lieu son ***efficacité*** (le faire rapidement et en utilisant le moins de ressources possibles).

Mais contrairement au simple exercice réalisé en travaux pratiques qui est abandonné aussitôt mis au point, un produit logiciel industriel peut être utilisé pendant plusieurs années. Au cours de ces années, l'équipe informatique sera amenée à en assurer la ***maintenance*** pour l'adapter aux nouvelles réglementations ou pour améliorer ses fonctionnalités.

Le temps passé à maintenir un logiciel est souvent plus important que celui qui a été consacré à son développement initial. C'est pourquoi, parmi les qualités d'un logiciel, on privilégie tout particulièrement la ***clarté de son code*** car elle facilite la maintenance. Cette qualité est favorisée en respectant des règles de programmation définies pour un projet et en évitant les bricolages plus ou moins "géniaux" compréhensibles de leur seul inventeur.

1.4 Exemple introductif

objectif : *être capable de concevoir des programmes très simples par analogie avec un exemple.*

Nous prendrons un exemple très simple.

Cahier des charges : calculer la surface et le volume de plusieurs sphères de rayons donnés.

1.4.1 Analyse

L'analyse est ici triviale : on applique des résultats de mathématiques. La surface et le volume des sphères sont des résultats bien connus : la surface est "$4\ \pi\ rayon^2$" et le volume "$4\ \pi\ rayon^3/3$".

Cependant, le problème posé consistant à faire ces calculs pour plusieurs sphères, il convient de définir deux ***fonctions*** *aire* et *volume* dépendant d'un paramètre qui est le *rayon.*

aire : réel → réel
aire(rayon) = 4 π rayon²

volume : réel → réel
volume(rayon) = 4 π rayon³/3

Ces fonctions pourront alors être sollicitées avec diverses valeurs du rayon.

Important : si un calcul correspond à une notion clairement identifiée, comme c'est le cas ici avec les notions de "surface de sphère" et de "volume de sphère" il est recommandé de le réaliser sous forme d'une fonction, même si ce calcul n'est pas destiné à être utilisé plusieurs fois.

1.4.2 Rédaction du programme

Le problème précédent peut être résolu en Java par le programme suivant :

```
class Sphere {  ← nom du programme

   static final double PI = 3.141592; ← déclaration de constante globale

   static double aire(double r) { ← définition de la fonction aire
    // résultat : l'aire d'une sphère de rayon r
        return 4*PI*r*r;
   }

   static double volume(double r) { ← définition de la fonction volume
    // résultat : le volume d'une sphère de rayon r
        return 4*PI*r*r*r / 3;
   }

   public static void main(String[] arg) { ← procédure principale
      System.out.println(aire(4.2));
      System.out.println (volume(4.2));
      System.out.println (aire(2)));
      System.out.println (volume(2));
   }
}
```

Explication des termes du programme :

`class Sphere { ... }` définit une ***classe*** appelée `Sphere` qui regroupe tout le programme. En Java, tout programme est une classe. Plus généralement, un programme peut être constitué de plusieurs classes : la classe est l'unité de programmation. Dans des exemples plus compliqués, les classes permettront de regrouper des fonctions concernant un même thème. Enfin, dans un style de programmation « par objets », les classes seront des modèles d'objets qui représentent les choses plus ou moins abstraites manipulées par une application.

Une classe comporte dans son intérieur des ***déclarations de données globales*** et des ***définitions de fonctions***.

Ici nous avons une déclaration de donnée globale :

```
static final double PI = 3.141592;
```

Cette déclaration donne le nom **PI** à la constante π bien connue. Le vocable **final** indique qu'il s'agit d'une constante, ce qui signifie qu'on n'a pas le droit de modifier la valeur associée au nom **PI**. Les données manipulées (constantes, variables, paramètres et résultats de fonctions) ont un ***type***. Le type d'une donnée caractérise le domaine auquel appartient sa valeur. Ici le vocable **double** indique que la valeur de **PI** est un nombre réel de grande précision[1]. Le vocable **static** signifie que la constante **PI** existe dès le début d'exécution du programme (cela peut sembler curieux d'avoir à le dire explicitement, mais c'est ainsi)[2].

Nous avons ensuite deux définitions de fonctions : **aire** et **volume**. Une définition de fonction est composée du nom de la fonction suivie de ses paramètres formels entre parenthèses. Le type du résultat de la fonction est indiqué devant le nom de la fonction et le type des paramètres est indiqué devant chaque nom de paramètre.

Ainsi :

```
static double aire(double r)
```

signifie que la fonction **aire** a un paramètre de type **double** appelé **r** et que son résultat est de type **double**. Le vocable **static** est obligatoire dans le cas d'une « simple » fonction[3].

Les textes compris entre "**//**" et la fin de la ligne sont des commentaires. Ils sont ignorés par le compilateur et servent à documenter le programme. Ici,

```
// résultat : l'aire d'une sphère de rayon r
```

est un ***commentaire de spécification***, qui indique la fonction (mathématique) réalisée par cette fonction (programmée).

Le résultat d'une fonction est obtenu en exécutant les ***instructions*** qui figurent dans le ***corps de la fonction***, entre accolades **{return 4*PI*r*r;}**. L'instruction **return** ***expression*** calcule l'expression et termine l'exécution en donnant comme valeur le résultat de ce calcul. Pour effectuer les calculs, on dispose des opérateurs usuels, ***** pour la multiplication, **/** pour la division.

Pour qu'un programme soit directement exécutable, il doit posséder une ***procédure principale*** :

```
public static void main(String[] arg)
```

C'est sur cette procédure que commencera l'exécution. Une procédure possède des paramètres et est composée d'instructions, mais contrairement à une fonction elle ne rend pas de résultat. Cela est indiqué par le vocable **void** devant le nom de la procédure.

En Java, la procédure principale doit s'appeler **main** et doit avoir un paramètre de type **String[]** (ce paramètre sert à capter des informations transmises au lancement de l'exécution, mais on s'en servira rarement). Une ***procédure*** ne rend pas de résultat au sens fonctionnel du mot. Autrement dit, étant donné une procédure **P**, son invocation ... **P()** ... ne signifie aucune valeur.

1. Le vocable **double** vient du terme "double précision" qui désigne cette sorte de représentation de nombres réels.
2. Java est un langage à objets et les classes servent souvent à programmer des modèles d'objets : dans ce cas les données déclarées sans ce vocable **static** sont propres à chaque objet créé selon ce modèle.
3. Sans le vocable **static** la fonction serait considérée comme une « méthode d'objet » destinée à s'appliquer sur un objet dont la classe serait le modèle, ce qui n'est pas le cas ici.

Pour être utile, elle doit produire un ***effet***. La notion d'effet est primordiale pour un langage impératif, alors que cette notion est quasiment inexistante pour un langage fonctionnel "pur"[1]. Les effets sont des actions perceptibles sur l'environnement du programme ou de la machine. Dans les cas simples, les effets sont de nature informationnelle tel que afficher des informations sur l'écran de l'ordinateur ou mémoriser des informations de façon rémanente dans des fichiers. Les effets peuvent également être de nature plus physique, tels que piloter un avion ou contrôler le freinage d'une voiture.

Dans l'exemple, l'instruction **`System.out.println(…);`** produit un effet. Elle ***affiche sur l'écran*** le résultat de l'évaluation de son paramètre. Ainsi la procédure principale a pour effet d'afficher successivement l'aire et le volume d'une sphère de rayon *4.2* puis l'aire et le volume d'une sphère de rayon *2*.

1.4.3 Compilation - exécution

Pour utiliser ce programme, il faut d'abord le ***compiler*** puis lancer son exécution. Dans le cas de Java, le programme précédent doit être placé dans un fichier appelé **`Sphere.java`**. Avec un système rudimentaire de développement de programmes, la compilation se fait en tapant la commande :

```
javac Sphere.java
```

S'il y a des erreurs de syntaxe, le compilateur **`javac`** (pour ***Java Compiler***) les signale. Il faut dans ce cas corriger ces erreurs et recommencer la compilation avant d'aller plus loin. Si le programme est exempt d'erreur de syntaxe, le compilateur produit un programme exécutable dans un fichier **`Sphere.class`**. Le texte du programme en langage évolué, **`Sphere.java`**, s'appelle "programme source". Le programme exécutable résultat de la compilation, **`Sphere.class`**, s'appelle "programme objet" ou encore "code objet".

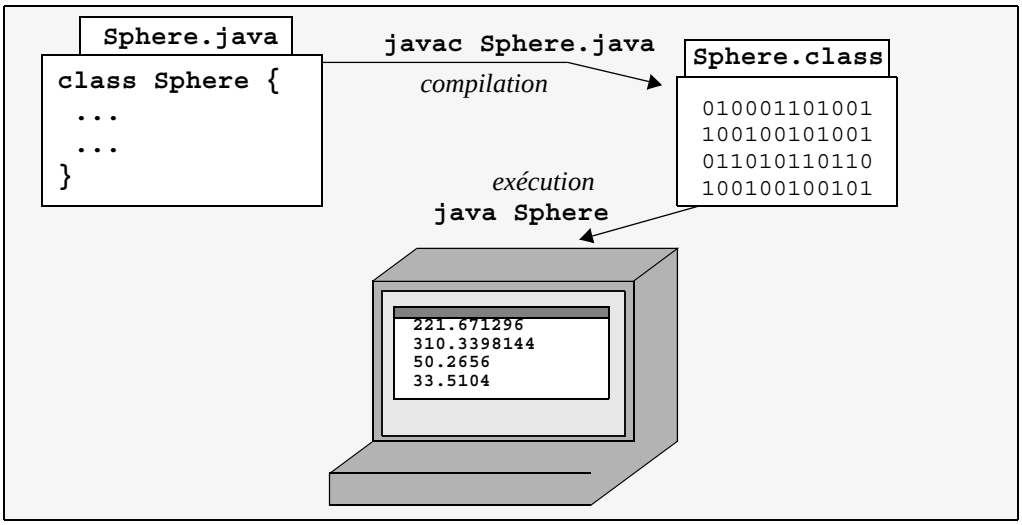
On peut alors lancer l'exécution en tapant la commande : **`java Sphere`**

L'exécution du programme provoque l'affichage suivant sur l'écran :

```
221.671296
310.3398144
50.2656
```

1. Le seul effet d'un programme fonctionnel pur est l'affichage du résultat de son évaluation.

```
33.5104
```


Sphere.java
class Sphere {
...
...
}
javac Sphere.java
compilation
Sphere.class
010001101001
100100101001
011010110110
100100100101
exécution
java Sphere
221.671296
310.3398144
50.2656
33.5104

QCM 1.1

Un algorithme est :

❏ 1 - Un programme d'ordinateur rédigé dans un langage de programmation précis.

❏ 2 - Une recette automatisable ("mécanisable") pour résoudre un problème.

❏ 3 - Une théorie physique.

❏ 4 - Une fonction mathématique très lentement croissante.

QCM 1.2

Une faute de syntaxe est :

❏ 1 - Une erreur grave qui se produit pendant l'exécution d'un programme.

❏ 2 - Un non respect des règles d'écriture d'un programme dans un langage de programmation précis.

❏ 3 - Une mauvaise méthode de résolution d'un problème.

❏ 4 - Bien plus grave et difficile à corriger qu'une faute de sémantique.

❏ 5 - Facile à corriger car indiquée et localisée par le compilateur.

QCM 1.3

On considère le programme suivant :

```
class TestFoisPi{

  static final double PI=3.141592;

  static double foisPi(int k) {
    return k*PI;
  }

  public static void main(String[] arg) {
    System.out.println(foisPi(10));
  }
}
```

Ce programme :

❏ 1 - Affiche : `3.141592`

❏ 2 - Affiche : `6.283184`

❏ 3 - Affiche : `31.41592`

❏ 4 - N'affiche rien.

❏ 5 - Est refusé par le compilateur car présentant une erreur de syntaxe.

❏ 3 - Retourne `31.41592`

Exercice 1.1 Installation

objectif : *prendre en main les outils de programmation* - ***indispensable pour pouvoir aller plus loin.***

Réaliser l'installation de l'environnement de travail en suivant les indications de l'annexe1 :

- Installation de Java
- Test de l'installation de Java
- Accès aux paquetages pour les exercices
- Installation d'Eclipse
- Usage d'Eclipse.

Exercice 1.2 Un petit programme simple

objectif : *comprendre le rôle des diverses rubriques de l'exemple du programme* **`Sphere`**

En s'inspirant du programme **`Sphere`** donné en exemple dans le chapitre précédent, rédiger un programme **`Parallelepipede`** qui calcule la surface et le volume de parallélépipèdes rectangles de diverses dimensions. Le programme doit afficher la surface et le volume de parallélépipèdes de dimensions 12×3×5 et 4×34×10.

Aide 1.2 Un petit programme simple

Il faut définir deux fonctions, **`aire`** et **`volume`**. Ces fonctions ont ici ***trois paramètres***, les dimensions du parallélogramme dans les trois directions de l'espace. Le programme total, que l'on pourra appeler **`Parallelepipede`** contiendra ces deux fonctions et la procédure principale **`main`** :

```
class Parallelepipede {

  static double aire(double lx, double ly, double lz) {
  // résultat : l'aire d'un parallélépipède
  // de dimensions lx, ly, lz
    ...
  }

  static double volume(double lx, double ly, double lz) {
  // résultat : le volume d'un parallélépipède
  // de dimensions lx, ly, lz
    ...
  }

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

CHAPITRE 2 Introduction aux types de données

objectif : *apprendre ce que sont les types de données et comprendre leur intérêt.*

Vous avez pu remarquer une différence dans les deux programmes donnés en exemple au chapitre précédent :

- le premier, qui calcule la somme des *n* premiers nombres entiers, utilise des données (**`i`**, **`n`** et **`somme`**) qualifiées par le vocable **`int`**,
- le second, qui calcule l'aire et le volume de sphères, utilise des données (**`PI`**, **`r`**, **`aire`**, **`volume`**) qualifiées par le vocable **`double`**.

Les vocables **`int`** et **`double`** indique les types des données manipulées. Pourquoi cette différence ? Est-elle nécessaire, utile, arbitraire ?

2.1 Notion de type

objectif : *comprendre ce qu'est un type de donnée.*

Les problèmes traités par l'informatique ne concernent pas uniquement des "nombres". Ils peuvent concerner des couleurs (bleu, vert, rouge...), des textes ou des choses encore plus exotiques comme des personnes, des voitures, des avions... Chaque nature de donnée est définie par un type.

Un ***type*** est caractérisé par :

- un domaine de valeurs,
- les opérations qui sont possibles sur ces valeurs.

Exemples :

nature	***exemple de valeurs***	***exemple d'opérations***	***type Java***
nombre entier	`12`	`5+7`	`int`
nombre réel	`3.14`	`6.28/2.0`	`double`
chaîne de caractères	`"bonjour"`	`"bon"+"jour"`	`String`
valeur logique (vrai ou faux)	`false`	`i>0 && i<3`	`boolean`

Un type sert à spécifier quelle sortes de données sont acceptées comme paramètres et rendues en résultat d'une opération. Par exemple, l'opération d'addition de nombres entiers, notée "**`+`**", admet deux opérandes entiers (type **`int`**) et rend en résultat une valeur entière (type **`int`** également). Autre exemple, l'opération de comparaison d'entiers, notée "**`>`**", admet deux opérandes de type **`int`** et rend en résultat une valeur de type **`boolean`** (*x*`>`y vaut **`true`** si *x>y*, **`false`** sinon).

2.2 Intérêt des types

objectif : *montrer l'intérêt d'être obligé d'indiquer le type les données et comment cette obligation devient un avantage en rendant les programmes plus lisibles et plus sûrs.*

2.2.1 Cohérence des données

Le principal intérêt des types est de permettre des vérifications de cohérence du programme avant l'exécution. Le compilateur vérifie que les paramètres des opérations sont de type convenable et que le résultat est utilisé de façon convenable. Certes la vérification de cohérence des types ne garantit pas qu'un programme est correct, c'est-à-dire qu'il réalise ce que l'on veut, mais elle rejette les programmes absurdes, sans qu'il soit besoin de les exécuter pour s'apercevoir de leur absurdité. Cela accélère donc la mise au point des programmes en faisant faire cette vérification préliminaire au compilateur.

On peut illustrer le rôle des types en prenant un exemple imagé. Considérons :

- le type **`Solide`**, représentant les matières ayant une forme propre,
- le type **`Liquide`**, représentant les matières tendant à s'écouler,
- et les opérations :
  `fondre(Solide aFondre)`, **`manger(Solide aManger)`**
  `bouillir(Liquide aBouillir)`, **`boire(Liquide aBoire)`**

Soient les déclarations suivantes :

```
Solide leBeurre; Solide leGruyere; Solide leCaoutchouc;

Liquide lHuile; Liquide lEau; Liquide leVin; Liquide leMazout;
```

L'instruction **`fondre(leVin)`** est refusée par le compilateur, car elle est absurde et on s'en aperçoit sans essayer d'exécuter cette opération.

En revanche, les instructions **`boire(lHuile)`**, **`boire(leMazout)`**, **`fondre(leBeurre)`** sont acceptées car les opérations mentionnées sont compatibles avec le type de leurs arguments. Ceci ne signifie pas que ces instructions sont nécessairement "correctes", par exemple **`boire(leMazout)`** est peut-être une erreur de programmation : l'usage des types ne garantit pas l'exactitude des programmes mais il permet au compilateur de détecter un grand nombre d'erreurs.

Exemple plus réaliste :

`12/"bonjour"` diviser un nombre par un texte n'a pas de sens,

```
double PI=3.1589;
(i<14)-PI
```
soustraire un nombre réel d'une valeur logique est absurde.

Ces expressions seront refusées par le compilateur.

2.2.2 Nécessité technologique de représenter correctement l'information

Le langage machine différencie déjà diverses sortes de données, par leurs représentations sur des paquets de bits et par les opérations offertes : nombres entiers, nombres réels... Ces différences sont inévitables : les nombres entiers sont des quantités "exactes", les nombres réels sont nécessairement de précision limitée...

2.2.3 Intérêt technologique de déterminer les besoins en mémoire

Un troisième intérêt des types, sans importance pour les aspects logiques des algorithmes, mais important pour l'efficacité des programmes, est de permettre au compilateur de savoir comment utiliser convenablement la mémoire de l'ordinateur. Une donnée occupe dans la mémoire une place qui dépend de sa nature. Par exemple un caractère est représenté sur un mot de 8 ou 16 bits (un ou deux octets), un nombre entier sur 32 bits, un réel en double précision sur 64 bits. La connaissance du type des données permet au compilateur d'utiliser juste la mémoire nécessaire.

2.3 Panorama des diverses catégories de types

objectif : *faire un tour d'horizon rapide sur les catégories de types qui seront étudiés par la suite.*

Le tableau suivant représente une classification des types offerts par la plupart des langages de programmation :

<table>
<tr><td rowspan="8">types</td><td rowspan="5">types primitifs</td><td rowspan="4">scalaires</td><td>entiers 12</td></tr>
<tr><td>réels 12.0</td></tr>
<tr><td>caractères 'w'</td></tr>
<tr><td>booléens false</td></tr>
<tr><td rowspan="2">composés</td><td>tableaux
0 12
1 2
2 25
3 12</td></tr>
<tr><td>chaînes "coucou"</td></tr>
<tr><td rowspan="3">types programmés</td><td>énumérés</td><td>Couleur = {bleu,blanc,rouge}</td></tr>
<tr><td>structures</td><td>Personne = <nom,age> ex. <toto,19></td></tr>
<tr><td>classe</td><td>class Voiture {
...
void accelerer() {...}
void freiner() {...}
}</td></tr>
</table>

Un certain nombre de types sont offerts par le langage. On les appelle ***types primitifs***. Parmi les types primitifs on peut distinguer :

- les ***types scalaires*** : nombres entiers, nombres réels, caractères, booléens...
- les ***types composés***, tableaux, chaînes de caractères... qui représentent des collections de données. Un ***tableau*** de *n* entiers correspond à peu près à l'idée mathématique d'un vecteur de *n* composantes entières. Une ***chaîne de caractères*** est une suite finie de caractères, souvent utilisée pour constituer des textes lisibles par un être humain.

Les langages offrent également au programmeur le moyen de définir de nouveaux types, qu'on peut appeler ***types programmés*** :

- Les plus simples des types programmés sont les ***types énumérés***. Un type énuméré est une collection finie de valeurs, sans autre propriété que l'égalité. Un exemple de tel type est un ensemble de couleurs, {*bleu*, *blanc*, *rouge*}.
- Une ***structure*** correspond à peu près à l'idée mathématique de produit cartésien de plusieurs domaines. Par exemple la description d'une personne se compose d'un nom, qui est une chaîne de caractères, et d'un âge, qui est une valeur entière.
- Dans un langage à objets comme Java, un type programmé peut se définir au moyen d'une ***classe***. Une telle classe joue le rôle de modèle pour décrire des objets de même nature : on y décrit comment ils sont représentés au moyen de types déjà existants et on y programme les opérations auxquelles ils participent. L'exemple suggéré dans le tableau concerne un type **`Voiture`** qui représente des voitures, dont l'état est caractérisé par une certaine vitesse et que l'on peut accélérer ou freiner.

Dans ce chapitre, nous nous bornerons à présenter les types primitifs scalaires. Les autres seront vus ultérieurement, une fois maîtrisée la rédaction de programmes simples.

2.4 Utilisations des types

objectif : *savoir déclarer des données, connaître l'influence des types sur la signification des symboles opératoires.*

2.4.1 Déclarations

Une ***déclaration*** consiste à donner un nom et indiquer le type d'une donnée ou d'une fonction. En Java, une déclaration de donnée a une des formes suivantes :

- Déclaration de donnée non initialisée :

  type nom **`;`**

  Une valeur sera attribuée ultérieurement par une instruction de la forme :

  nom **`=`** ***expression*** **`;`**

  Exemple : **`int i; ... i=12;`**

- Déclaration de donnée initialisée :

  type nom **`=`** ***expression*** **`;`**

  Exemple : **`int i=12;`**

- Déclaration de fonction et de paramètres de fonction :

  typeDuRésultat nomDeLaFonction **`(`** ***type nom*** **`,...) {...}`**

  les paramètres de la fonction sont déclarés dans les parenthèses à la suite du nom de la fonction, sous la forme ***type nom***. Ils ne sont jamais initialisés dans la déclaration, car leur valeurs sont fournies aux endroits du programme qui appelle la fonction.

  Exemple, une fonction qui rend en résultat $k \times \pi$:

```
static double kFoisPI(int k) {return k*3.141592;}
```

2.4.2 Exemples d'opérations sur des données de types primitifs

2.4.2.1 Opérations sur les nombres "réels"

Les opérations d'addition (**+**), de soustraction (**-**), de multiplication (*****) et de division (**/**) sont traditionnelles. Il faut cependant se méfier : les types offerts par les langages de programmation ne sont que des ***approximations*** des nombres réels.

Considérons le programme suivant :

```
class Test {
  public static void main(String[] arg) {
     double x = 49.0;
     System.out.println(x*(1.0/x)==1.0);
  }
}
```

L'expression **x*(1.0/x)==1.0** teste si ***x /x*** est égal à 1.

Pour ***x***=49.0, le résultat affiché est **false** (***faux***), à cause de la précision limitée de la représentation des nombres réels (type **double**). Ainsi ***x*** multiplié par 1/***x*** ne donne pas toujours 1. Ce phénomène est similaire à ce qui se passe en "décimal" si on calcule $3 \times 1/3$ sur un nombre de chiffres limité : on obtient $3 \times 0{,}333333 = 0.999999$ au lieu de 1.

2.4.2.2 Opérations sur les nombres entiers

Les opération d'addition (**+**), de soustraction (**-**) et de multiplication (*****) sont les opérations usuelles. La division sur les nombres entiers conduit à deux opérations :

- le quotient entier de ***a*** par ***b*** (noté ***a/b***),
- le reste de la division de ***a*** par ***b*** (on dit également "***a*** modulo ***b***", noté ***a%b***).

Exemple :

14/3 vaut **4** et **14%3** vaut **2**

car 14 divisé par 3 donne un quotient entier égal à 4 et il reste 2.

2.4.2.3 Opérations sur les booléens

Le type booléen, **boolean** en Java, possède deux valeurs, ***vrai*** (**true**) et ***faux*** (**false**). Ces valeurs servent principalement à évaluer des conditions qui servent d'argument à des prises de décisions.

Les opérations de comparaisons numériques (inférieur **<**, supérieur **>**, inférieur ou égal **<=**, supérieur ou égal **>=**, égal **==**) fournissent des résultats booléens. Exemples :

3>5 vaut **false**, **5<=9** vaut **true**

Les opérations avec opérandes booléens sont le ***ET*** (**&&**) le ***OU*** (**||**) et la ***négation*** (**!**). Elles permettent d'exprimer des conditions complexes. Exemple, pour tester si un entier k est compris entre 5 et 9, bornes incluses, on peut utiliser l'expression : **(k>=5) && (k<=9)**

2.4.2.4 Types des expressions - conversions

Il faut être attentif et rigoureux lorsqu'on réalise des opérations numériques, car le même symbole opératoire, **`+`**, **`-`**, **`*`**, **`/`**, désigne des opérations différentes selon le type des opérandes. Les types des opérandes d'une opération se déduisent naturellement de la forme des expressions :

- les notations numériques pour les nombres réels se distinguent par la présence d'un point ou d'une notation dite "scientifique" : **`3.0`**, **`12E-3`** (pour 12×10^{-3}),
- les noms de données ont le type de leur déclaration :

  avec **`double x=7; ... x/2.0`** vaut 3.5 (division de nombre réels),

  avec **`int x=7; ... x/2`** vaut 3 (division de nombre entiers),
- la sollicitation d'une fonction a le type déclaré de son résultat :

  `kfoisPI(2)` est un nombre réel, de type **`double`**, qui vaut 6.283182.

Il existe des opérations qui permettent de "convertir" des données d'un type dans un autre type. Ces opérations s'appellent des ***conversions***. Elles réalisent des correspondances usuelles entre domaines de valeurs :

- la conversion d'un entier en réel fournit le réel qui représente la même "quantité",
- la conversion d'un réel en entier fournit la partie entière du nombre réel.

En Java une telle conversion se note : **`(`*type*`)`** ***expression***

Exemples :

`(int) 3.141592` vaut l'entier 3, de type **`int`**

`(double) 6` vaut le réel 6.0, de type **`double`**

En plus de ces règles strictes, la plupart des langages réalisent implicitement la conversion des entiers en réel dans les opérations notées +, -, *, / lorsque un des opérandes est explicitement réel. Par exemple :

`5.0/2` est équivalent à **`5.0/((double) 2)`** et vaut donc 2.5.

2.5 Description des types scalaires

objectif : *préciser le comportement des types primitifs scalaires de Java (ce paragraphe peut être sauté en première lecture. S'y reporter au fur et à mesure des besoins).*

2.5.1 Types primitifs scalaires

En Java offre les types primitifs scalaires suivants :

domaine représenté	*types Java*	*notations de valeurs*
nombres entiers	**`int, long, short, byte`**	**`12 -4567`**
nombres réels	**`double, float`**	**`3.141592 6.02E23`**
caractères	**`char`**	**`'a' 'z' '?' '\n'`**
valeurs logiques	**`boolean`**	**`true false`**

Il existe plusieurs types pour les nombres entiers : **`int`**, codés sur *32* bits (de -2^{31} à $2^{31}-1$), **`long`**, codés sur 64 bits (de -2^{63} à $2^{63}-1$), **`short`** sur 16 bits (de -2^{15} à $2^{15}-1$) et **`byte`** sur 8 bits (de *-128* à *127*). Nous utiliserons généralement le type **`int`**, bien qu'il ne permette pas de représenter des entiers très grands (à peu près de -2 milliards à +2 milliards).

Les valeurs entières sont notées classiquement, en décimal.

De même il existe deux types pour les nombres réels : **`float`**, codés sur *32* bits (nombres en simple précision) et **`double`**, codés sur *64* bits (nombres en double précision). Nous utiliserons le type **`double`**, car le type **`float`** est vraiment peu précis (même pas l'équivalent de 7 chiffres décimaux pour la mantisse).

Les valeurs réelles sont notées avec un point pour séparer la partie entière de la partie fractionnaire. La notation avec exposant **`6.02E23`** signifie 6.02×10^{23}.

Les caractères sont représentés par le type **`char`**. Les notations de valeurs utilisent l'apostrophe : **`'`***x*`'` signifie le caractère *x*. Pour les caractères dont l'écriture directe est impossible, on utilise une notation particulière : **`'\n'`** signifie le passage à la ligne, **`'\r'`** le retour en début de ligne, '**`\t'`** une tabulation...

Les tableaux suivants résument les opérateurs du langage. Pour chaque opérateur on a précisé son ***profil***, c'est-à-dire le type de ses paramètres et de son résultat. Certaines opérations et constantes sont notées au moyen d'un nom préfixé par **`Math`** : **`Math.abs`**, **`Math.sqrt`**, **`Math.PI`**... Ce sont des fonctions et constantes définies dans la classe **`Math`** de la bibliothèque standard de Java.

opérations avec arguments de type entier				
	notation	***profil***	***exemple***	***résultat***
addition	`+`	`int` × `int` → `int`	`4 + 12`	`16`
soustraction	`-`	`int` × `int` → `int`	`4 - 12`	`-8`
multiplication	`*`	`int` × `int` → `int`	`4 * 12`	`48`
changement de signe	`-`	`int` → `int`	`- 12`	`-12`
division entière	`/`	`int` × `int` → `int`	`23 / 4`	`5`
modulo (reste)	`%`	`int` × `int` → `int`	`23 % 4`	`3`
test d'égalité	`==`	`int` × `int` → `boolean`	`4 == 12`	`false`
test d'inégalité	`!=`	`int` × `int` → `boolean`	`4 != 12`	`true`
test de supériorité	`>`	`int` × `int` → `boolean`	`4 > 12`	`false`
test "supérieur ou égal"	`>=`	`int` × `int` → `boolean`	`4 >= 4`	`true`
test d'infériorité	`<`	`int` × `int` → `boolean`	`4 < 4`	`false`
test "inférieur ou égal"	`<=`	`int` × `int` → `boolean`	`4 <= 4`	`true`
valeur absolue	`Math.abs`	`int` → `int`	`Math.abs(-4)`	`4`

opérations avec arguments de type réel				
	notation	***profil***	***exemple***	***résultat***
addition	`+`	`double` × `double` → `double`	`4.0 + 12.7`	`16.7`
soustraction	`-`	`double` × `double` → `double`	`4.5 - 12`	`-7.5`
multiplication	`*`	`double` × `double` → `double`	`4E8 * 12E-3`	`48E5`
changement de signe	`-`	`double` → `double`	`- 12.0`	`-12.0`
division	`/`	`double` × `double` → `double`	`23.0 / 4.0`	`5.75`
test de supériorité	`>`	`double` × `double` → `boolean`	`4.5 > 4.0`	`true`
test d'infériorité	`<`	`double` × `double` → `boolean`	`4.5 < 4.0`	`false`
valeur absolue	`Math.abs`	`double` → `double`	`Math.abs(-4.5)`	`4.5`
racine carrée	`Math.sqrt`	`double` → `double`	`Math.sqrt(2.0)`	`1.414...`
sinus	`Math.sin`	`double` → `double`	`Math.sin(0.0)`	`0.0`
cosinus	`Math.cos`	`double` → `double`	`Math.cos(0.0)`	`1.0`
tangente	`Math.tan`	`double` → `double`	`Math.tan(0.0)`	`0.0`
logarithme népérien	`Math.log`	`double` → `double`	`Math.log(1.0)`	`0.0`
exponentielle	`Math.exp`	`double` → `double`	`Math.exp(0.0)`	`1.0`
constante π	`Math.PI`	→ `double`	`Math.PI`	`3.14159...`
constante e	`Math.E`	→ `double`	`Math.E`	`2.71828...`

Remarque : dans la plupart des langages de programmation les types réels admettent les comparaisons faisant intervenir l'égalité (=, ≥, ≤, ≠), mais il est préférable de ne pas les utiliser car c'est souvent un non-sens. À cause des erreurs d'arrondi, l'égalité de deux résultats réels est généralement le fruit du hasard.

opérations avec arguments de type caractère				
	notation	***profil***	***exemple***	***résultat***
test d'égalité	`==`	`char` × `char` → `boolean`	`'a' == 'w'`	`false`
test d'inégalité	`!=`	`char` × `char` → `boolean`	`'a' != 'w'`	`true`
test de supériorité	`>`	`char` × `char` → `boolean`	`'a' > 'w'`	`false`
test "supérieur ou égal"	`>=`	`char` × `char` → `boolean`	`'a' >= 'w'`	`false`
test d'infériorité	`<`	`char` × `char` → `boolean`	`'a' < 'w'`	`true`
test "inférieur ou égal"	`<=`	`char` × `char` → `boolean`	`'a' <= 'a'`	`true`

Les caractères peuvent être comparés. Une relation d'ordre est définie sur les caractères et elle respecte l'ordre alphabétique pour les lettres `'A'`... `'Z'`... `'a'`... `'z'` et l'ordre numérique usuel pour les chiffres `'0'`... `'9'`.

opérations avec arguments de type booléen				
	notation	***profil***	***exemple***	***résultat***
négation	`!`	`boolean` → `boolean`	`! false`	`true`
ou logique	`|`	`boolean` × `boolean` → `boolean`	`true | false`	`true`
ou logique conditionnel	`||`	`boolean` × `boolean` → `boolean`	`true || ???`	`true`
et logique	`&`	`boolean` × `boolean` → `boolean`	`false & true`	`false`
et logique conditionnel	`&&`	`boolean` × `boolean` → `boolean`	`false && ???`	`false`
test d'égalité	`==`	`boolean` × `boolean` → `boolean`	`false==false`	`true`
test d'inégalité	`!=`	`boolean` × `boolean` → `boolean`	`false!=false`	`false`

Le ***ou*** simple, noté '`|`', évalue toujours ses deux arguments, alors que le ***ou conditionnel***, noté '`||`', n'évalue pas le second argument si le premier est ***vrai*** (le premier argument détermine le résultat à ***vrai***). De même le ***et*** simple, '`&`', évalue ses deux arguments, alors que le ***et conditionnel***, '`&&`', n'évalue pas le second argument si le premier est ***faux*** (le premier argument détermine le résultat à ***faux***).

La différence est importante comme le montre l'exemple suivant :

```
x>0 & (3/x)<12
```

provoque une erreur à l'exécution si `x` vaut `0` car `(3/0)<12` est évalué et provoque l'erreur "tentative de division par 0".

En revanche

```
x>0 && (3/x)<12
```

ne provoque pas d'erreur si `x` vaut `0` car `x>0` est ***faux*** et le total vaut donc ***faux***.

2.5.2 Types énumérés

Java (à partir de la version 1.5) offre les types énumérés. Un tel type possède un ensemble fini de valeurs désignées par autant d'identificateurs. On peut par exemple définir le type `Couleur` comme un type énuméré possédant 3 valeurs notées `bleu`, `blanc` et `rouge` :

```
enum Couleur {bleu, blanc, rouge};
```

Les déclarations des données d'un type énuméré utilisent le nom du type, et les désignations de valeurs se font au moyen de la notation ***nomDuType*`.`*identificateur***. Le seul opérateur défini sur un type énuméré est le test d'égalité, noté "`==`".

Exemple :

- `Couleur milieuDuDrapeau=Couleur.blanc;` définit la couleur du "milieu du drapeau".
- L'expression de comparaison `milieuDuDrapeau==Couleur.blanc` vaut `true`,
- L'expression de comparaison `milieuDuDrapeau==Couleur.rouge` vaut `false`.

QCM 2.1

Soit l'expression : `2>=3 || 12%2==0`

- ❑ 1 - C'est une faute de syntaxe.
- ❑ 2 - Son résultat est de type **`int`**.
- ❑ 3 - Son résultat est de type **`boolean`**.
- ❑ 4 - Son résultat est 6.
- ❑ 5- Son résultat est ***vrai***.
- ❑ 6 - Son résultat est ***faux***.

QCM 2.2

Soit l'expression : `0<2<5`

- ❑ 1 - C'est une faute de syntaxe.
- ❑ 2 - Son résultat est de type **`int`**.
- ❑ 3 - Son résultat est de type **`boolean`**.
- ❑ 4 - Son résultat est 6.
- ❑ 5- Son résultat est ***vrai***.
- ❑ 6 - Son résultat est ***faux***.

QCM 2.3

Soit l'expression : `2/5`

- ❑ 1 - C'est une faute de syntaxe.
- ❑ 2 - Son résultat est de type **`int`**.
- ❑ 3 - Son résultat est de type **`double`**.
- ❑ 4 - Son résultat est 0.2.
- ❑ 5- Son résultat est 0.
- ❑ 6 - Cela provoque une erreur pendant l'exécution.

QCM 2.4

Soit l'expression : `(double) (2/5)`

- ❑ 1 - C'est une faute de syntaxe.
- ❑ 2 - Son résultat est de type **`int`**.
- ❑ 3 - Son résultat est de type **`double`**.
- ❑ 4 - Son résultat est 0.2.
- ❑ 5- Son résultat est 0.0.
- ❑ 6 - Cela provoque une erreur pendant l'exécution.

QCM 2.5

Soit l'expression : `((double) 2)/5`

- ❑ 1 - C'est une faute de syntaxe.
- ❑ 2 - Son résultat est de type **`int`**.
- ❑ 3 - Son résultat est de type **`double`**.
- ❑ 4 - Son résultat est 0.2.
- ❑ 5- Son résultat est 0.0.
- ❑ 6 - Cela provoque une erreur pendant l'exécution.

QCM 2.6

- ❑ 1 - Une donnée de type **`int`** peut être négative.
- ❑ 2 - Une donnée de type **`int`** peut valoir 1000000000 (un milliard).
- ❑ 3 - Une donnée de type **`int`** peut valoir 100000000000 (cent milliards).
- ❑ 4 - Une donnée de type **`long`** peut valoir 1000000000 (un milliard).
- ❑ 5- Une donnée de type **`long`** peut valoir 100000000000 (cent milliards).
- ❑ 6 - Une donnée de type **`int`** peut valoir 0.5 (un demi).
- ❑ 7 - Une donnée de type **`long`** peut valoir 0.5 (un demi).

QCM 2.7

- ❑ 1 - Une donnée de type **`double`** peut être négative.
- ❑ 2 - Une donnée de type **`double`** peut valoir 1000000000 (un milliard).
- ❑ 3 - Une donnée de type **`double`** peut valoir 100000000000 (cent milliards).
- ❑ 4 - Une donnée de type **`double`** peut valoir 0.5 (un demi).
- ❑ 5 - Une donnée de type **`double`** peut être nulle.

QCM 2.8

Considérons le type **`Couleur`** et les fonctions **`estChaude`** et **`estFoide`** ainsi définies :

```
enum Couleur{rouge,orange,jaune,vert,bleu,violet};

static boolean estChaude(Couleur c){
  return c==Couleur.rouge || c==Couleur.orange || c==Couleur.jaune;
}

static boolean estFroide(Couleur c){
  return !estChaude(c);
}
```

- 1 - L'expression **`estChaude(Couleur verte)`** est une erreur de syntaxe.
- 2 - L'expression **`estChaude(Couleur.verte)`** vaut **`true`** (vrai).
- 3 - L'expression **`estChaude(Couleur.verte)`** vaut **`false`** (faux).
- 4 - L'expression **`estChaude(Couleur.verte)`** vaut **`0`** (entier nul).
- 5 - L'expression **`estFroide(Couleur.verte)`** vaut **`false`** (faux).
- 6 - L'expression **`estFroide(Couleur.verte)`** vaut **`true`** (vrai).
- 7 - L'expression **`estChaude(Couleur.verte)`** est une erreur de syntaxe.
- 8- L'expression **`estChaude(Couleur.rouge)`** est une erreur de syntaxe.
- 9 - L'expression **`estChaude(Couleur.indigot)`** est une erreur de syntaxe.
- 10 - L'expression **`estChaude(Couleur.rouge)`** vaut **`true`** (vrai).

Exercice 2.1 Expressions, priorité des opérateurs

objectif : *se familiariser avec les types des opérandes et du résultat d'une expression, comprendre les erreurs de syntaxes provoquées par le non respect des types.*

L'expression `2+3*4` peut se comprendre de deux manières :

(2+3)*4 qui vaut 20 ou 2+(3*4) qui vaut 14

Pour éviter les ambiguïtés, les opérateurs en Java possèdent des *priorités*. L'opération la plus prioritaire est effectuée en premier.

Opérateurs par ordre de priorité décroissante :

`-` (moins unaire)

`*`, `/`,`%`(multiplication, division, reste)

`+`, `-` (addition, soustraction)

`==`,`!=`, `<`, `>`, `<=`, `>=` (comparaisons)

`&&` (*et* logique)

`||` (*ou* logique)

L'expression `2+3*4` se comprend donc 2+(3*4) et vaut 14.

En cas d'égalité des priorités, le calcul a lieu de gauche à droite.

Ainsi, `1-4-5` se comprend ((1- 4)-5) et vaut -8.

Si l'on veut contrôler ces priorités, il est nécessaire de parenthéser.

Par exemple, pour obtenir 20, il faut écrire `(2+3)*4`.

On considère les expressions suivantes :

(1) `4-3*2-1` *(2)* `-4-5` *(3)* `5-2<4*2`

(4) `2/3>0` *(5)* `4-5-3==2*-5/2+1`

(6) `3>2 && 1<4` *(7)* `6<7&&7<5+3` *(8)* `6<7<5+3`

1 - Parenthésage explicite :

- indiquer les équivalents totalement parenthésés de ces expressions,
- indiquer celles qui sont des erreurs de syntaxe (fautes de type),
- indiquer le résultat de celles qui sont correctes.

2 - Vérification :

Une façon simple de vérifier ce qui précède consiste à rédiger un programme de test (une simple procédure **`main`**). Voici le modèle, pour vérifier que **`2+3*4`** est équivalent à **`2+(3*4)`** :

```
class TestParenthesage {
  public static void main(String[] arg) {
    System.out.print"2+3*4 = ");
    System.out.println(2+3*4);
    System.out.print("équivalent : 2+(3*4) = ");
    System.out.println(2+(3*4));}
}
```

Ce modèle montre une façon systématique de réaliser un programme de test :

- **`System.out.println("2+3*4 = ")`** imprime le texte de la formule à tester,
- **`System.out.print(2+3*4)`** imprime le résultat de l'exécution la formule.

Ceci affiche : **`2+3*4 = 14`**

Certaines des formules proposées sont des erreurs de syntaxe. Essayer de comprendre le message d'erreur du compilateur, puis inhiber les lignes qui sont des erreurs en les transformant en commentaire. Il suffit pour cela de placer "**`//`**" en début de ligne :

```
// System.out.print(2*+/4-);
```

Exercice 2.2 Moyenne de trois nombres

objectif : *savoir choisir les types des données manipulées, faire des conversions si nécessaire.*

Première partie

Rédiger un programme **`TestMoyenne`** qui calcule la moyenne de 3 nombres entiers (de type **`int`**) pour les triplets de nombres suivants :

3, 1, 1

-12, 2, 12

4, 6,7

Attention : les résultats du calcul de la moyenne sont des nombres réels (de type **`double`**).

Deuxième partie

En Java, la procédure principale a pour en-tête :

```
public static void main(String[] arg)
```

Le paramètre **`arg`** est un tableau de chaînes de caractères (type **`String[]`**). Ce paramètre est un tableau dont les éléments valent les chaînes de caractères frappées à la suite du nom du programme lors de son lancement. Ceci permet de passer des paramètres au programme lors de son lancement. Si par exemple on lance le programme par la commande :

```
java TestMoyenne 12 53 6
```

`arg[0]` vaut **`"12"`**, **`arg[1]`** vaut **`"53"`** et **`arg[2]`** vaut **`"6"`**.

Les paramètres sont des chaînes de caractères. Pour obtenir les nombres réels correspondant, il faut utiliser la ***fonction de conversion*** offerte par la bibliothèque Java :

```
Integer.valueOf(s)
```

qui rend en résultat le nombre entier représenté par la chaîne de caractères **s**.

Exemple : **`Integer.valueOf("12")`** vaut le nombre entier 12.

Rajouter à la procédure principale de **`TestMoyenne`** le calcul et l'affichage de la moyenne de 3 nombres frappés à la suite de la commande de lancement.

Aide 2.2

La fonction **`moyenneDeTroisNombres`** a trois paramètres, les trois nombres entiers dont il faut calculer la moyenne.

```
class TestMoyenneDeTroisNombres {

  static double moyenneDeTroisNombres(int x, int y, int z){
  // résultat : moyenne de x, y et z
    ...
  }

  public static void main(String[] args) {
    System.out.print("moyenne de 3,1,1 = ");
    System.out.println(moyenneDeTroisNombres(3,1,1));
    ...
    ...
  }
}
```

Se méfier de la division entière : si on divise la somme de trois entiers par 3, la division effectuée est la division entière, ce qui n'est pas la moyenne. Penser à faire correctement une ***conversion*** d'entier en réel de type **`double`**.

CHAPITRE 3 Déclarations de données, fonctions, expressions, instructions

objectif : *connaître précisément les diverses rubriques d'un programme, les diverses sortes de données, les expressions et les instructions élémentaires.*

Les types spécifient la nature des données manipulées par les programmes. Nous allons voir maintenant ce qui permet de manipuler les données :

- les ***déclarations de données*** qui identifient les données,
- les ***instructions***. Nous nous limiterons dans ce chapitre à la définition de fonctions, l'appel de fonctions, la séquentialité et l'affichage d'un résultat.

3.1 Structure de programmes simples

Un programme simple possède la structure suivante :

```
class nom du programme {
    déclarations de données globales
    définitions de fonctions
    procédure principale
}
```

Les ***déclarations de données globales*** permettent de définir des données qui seront utilisables depuis tout le texte du programme, par opposition aux paramètres et données locales des diverses fonctions du programme qui ne sont utilisables que depuis le texte de ces fonctions.

Commentaires :

Pour faciliter la compréhension d'un programme, des commentaires peuvent être utiles et parfois nécessaires. Un commentaire est une explication destinée aux lecteurs humains, ignorée par la machine (le compilateur).

En Java il en existe de deux formes :

`//` *texte commentaire jusqu'à la fin de la ligne*

et

`/*` *commentaires*
sur un nombre quelconque
de lignes `*/`

La première forme est la plus agréable pour les commentaires qui spécifient ce que font les programmes.

La deuxième forme est souvent utilisée temporairement, en phase de mise au point de programmes, pour inhiber la prise en compte de certaines parties du programme.

3.2 Déclarations de données

3.2.1 Les diverses sortes de données

Un programme est constitué d'"instructions" qui agissent sur des "données". D'une façon très générale une donnée est quelque chose qui :

- joue un certain ***rôle***,
- a un ***nom*** (on dit aussi un ***identificateur***),
- possède une valeur d'un certain ***type***.

En plus de leur type, les données ont divers qualificatifs :

- Une donnée peut être ***globale*** ou ***locale*** :

  - Une donnée ***globale*** est utilisable depuis toutes les fonctions de la classe qui constitue le programme. Une déclaration de donnée globale apparaît au premier niveau de la classe, en dehors de toute fonction. De plus, une donnée globale peut être ***statique*** ou ***non statique***. Une donnée statique est créée dès le début d'exécution du programme. Dans le cas de programmes simples, les données globales seront toujours statiques. Elles sont précédées du vocable **`static`**

  - Une donnée ***locale*** n'est utilisable que depuis le texte de la fonction où elles est déclarée.

- Une donnée peut être une ***variable*** ou une ***constante*** :

  - La valeur associée à une ***variable*** peut être modifiée en cours d'exécution, au moyen d'une instruction d'affectation.

  - La valeur associée à une ***constante*** est fixée une fois pour toute à l'endroit de sa déclaration. Une déclaration de constante est précédée du vocable **`final`**.

Voici la forme générale des diverses déclarations de données que l'on peut faire dans un programme :

```
class leProgramme {
   ...
   static final type nom = valeur ; ← constante globale
   static type nom ; ← variable globale
   ...

   static void ppp(...) {← définition de fonction
      final type nom = valeur ; ← constante locale à la procédure ppp
      type nom; ← variable locale à la fonction ppp
      ...
   }

   public static void main(String[] z) { ← procédure principale
      final type nom = valeur ; ← constante locale à la procédure principale
      ...
   }
}
```

Exemple :

```
class leProgramme {
   ...
   static final double graviteTerrestre = 9.81; ← constante globale
   static final int nombreDeCartes = 32; ← constante globale
   ...

   public static void main(String[] z) {
      final int ageDuCapitaine = 53; ← constante locale
      int score; ← variable locale
   ...
   }
}
```

Remarques :

- Une déclaration locale n'a jamais le vocable **static**.
- l'usage des variables sera précisé ultérieurement. Leur principal intérêt concerne la réalisation des ***itérations***.
- Le besoin d'utiliser des variables statiques globales est très rare : il faut l'éviter.

3.2.2 Intérêt des déclarations de constantes

Les déclarations de constantes permettent, dans les instructions des programmes, d'utiliser les noms de ces constantes à la place des notations de valeurs. Ceci présente plusieurs avantages :

- Une plus grande ***lisibilité*** du programme. En effet un programme n'est pas seulement destiné à une machine, il doit aussi pouvoir être compris par des lecteurs humains, pour faciliter sa ***mise au point*** et ultérieurement sa ***maintenance***, c'est-à-dire permettre des modifications dues aux changements des besoins des utilisateurs du programme.

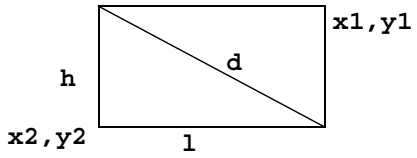
- Une facilité de ***modification*** en centralisant en un seul endroit la notation de valeur. Si la constante doit être modifiée, dans une version ultérieure, seule la déclaration de constante est à modifier.

```
public static void main(String[] z) {
   final int ageDuCapitaine = 53;← un seul endroit à modifier
   ...                                pour changer l'âge du capitaine
   ... ageDuCapitaine ... ageDuCapitaine ...
   ...
   ... ageDuCapitaine ...
}
```

- Les déclarations de constantes peuvent être utilisées également pour ***nommer des résultats intermédiaires*** et ainsi éviter des expressions trop grosses qui pourraient nuire à la lisibilité :

  soit un rectangle défini par les coordonnées de son coin supérieur droit (**x1**,**y1**) et de son coin inférieur gauche (**x2,y2**). On peut calculer la dimension de sa diagonale par :

  ```
  final double d = Math.sqrt((x1-x2)*(x1-x2)+(y1-y2)*(y1-y2));
  ```



  mais on préférera peut-être utiliser les notion intermédiaires de ***hauteur*** et de ***largeur*** :

  ```
  final double h = x1-x2;
  final double l = y1-y2;
  final double d = Math.sqrt(h*h+l*l);
  ```

- Enfin, les déclarations de constantes servent à ***calculer une seule fois*** une valeur qui doit être utilisée à plusieurs reprises :

  soit par exemple le calcul de la surface de base et du volume d'un parallélépipède. Le volume est lui-même obtenu en multipliant la surface de base par la hauteur. On veut calculer une fois la surface de base :

```
class Parallelepipede {
  public static void main(String[] arg) {
    int largeur=...; int longueur=...; int hauteur=...;

    final int surfaceDeBase=largeur*longueur;

    System.out.print("surface de base = ");
    System.out.println(surfaceDeBase);
    System.out.print("volume = ");
    System.out.println(surfaceDeBase*hauteur);
  }
}
```

3.3 Définitions de fonctions

objectif : *savoir définir une fonction, distinguer les fonctions qui réalisent des fonctions au sens mathématique et les procédures qui ont des effets.*

Une ***définition de fonction*** a la forme suivante[1] :

```
static typeDuRésultat nomDeLaFonction(type1 param1, type2 param2...) {
        déclarations locales et instructions
}
```

Le type du résultat est indiqué devant le nom de la fonction, le type et le nom de chaque paramètre sont indiqués entre parenthèses à la suite du nom de la fonction. Ces paramètres sont appelés ***paramètres formels***.

Si une fonction n'a pas de paramètre, il faut tout de même mettre les parenthèses :

```
static typeDuRésultat nomDeLaFonction() {...}
```

En langage impératif, il existe des fonctions qui ne rendent aucun résultat. On a l'habitude d'appeler ***procédure*** une telle fonction. Dans ce cas on utilise le vocable **`void`** à la place du type du résultat :

```
static void nomDeLaProcédure(type1 param1, type2 param2...) {
        déclarations locales et instructions
}
```

Une fonction est une "collection identifiée d'instructions qui fait quelque chose". Une fonction est destinée à être appelée. Cela consiste à donner des valeurs, appelées ***paramètres effectifs***, à la place des paramètres formels et à exécuter les instructions de la fonction en utilisant ces paramètres effectifs.

On peut distinguer trois catégories de fonctions :

les ***fonctions pures***, les ***procédures*** et les ***fonctions "impures"*** :

- Une ***fonction pure*** réalise une fonction au sens mathématique du mot. Elle calcule une valeur d'un certain type et la rend en résultat, sans plus. Par exemple la fonction **`sigma3`** qui calcule la somme de 3 nombres entiers :

```
static int sigma3(int a, int b, int c) {
// résultat : la somme de a, b et c
    return a+b+c;
}
```

1. Le vocable **`static`** signifie ici que la fonction est "simple", que ce n'est pas une "méthode d'objet" (voir plus loin).

Un appel de cette fonction pourra avoir la forme **`sigma(3,5,4)`** qui a pour résultat **`12`**. Un tel appel n'a pas d'effet, il signifie une valeur. Pour que l'appel serve à quelque chose, il faut nécessairement que la valeur soit utilisée, soit en donnant cette valeur à une variable du programme :

`int x = sigma3(3,5,4);...` donne la valeur *12* à **`x`**,

soit en l'utilisant pour calculer le paramètre effectif d'un opérateur ou d'un autre appel de fonction :

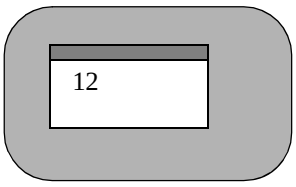
`... System.out.print(sigma3(3,5,4)+2));...` affiche "14" sur l'écran.

- Une ***procédure*** ne rend pas de résultat. On l'utilise pour les ***effets*** qu'elle produit. Parmi les effets on peut distinguer les interactions avec l'environnement, par exemple :
  - des affichages sur écran,
  - des acquisitions de données introduites par un utilisateur du programme,
  - des lectures et écritures de fichiers,
  - des actions sur des moteurs, des vannes...
  - des changements d'état de certains objets du programme.

  On peut prendre comme exemple la procédure **`afficheSommeDe3Nombres`** qui affiche sur l'écran la somme de trois nombres :

```
static void afficheSommeDe3Nombres(int a, int b, int c) {
// effet : affiche la somme de a, b et c
     System.out.print(a+b+c);
}
```

  Un appel possible de cette procédure est **`afficheSommeDe3Nombres(3,5,4)`** qui a pour effet d'afficher "12" sur l'écran :



- Une ***fonction "impure"*** rend un résultat, comme une fonction pure, mais elle ne réalise pas une simple fonction mathématique. Elle peut avoir un effet, comme une procédure, ou consulter l'environnement. On peut prendre pour exemple la "fonction" **`lectureNombreEntier`** qui rend en résultat le nombre entier frappé au clavier par l'utilisateur du programme :

  `static int lectureNombreEntier() {`*... lit un entier...*`}`

  Cette "fonction" attend que l'utilisateur frappe une suite de chiffres au clavier, et s'il frappe par exemple "**`176`**", elle rend l'entier **`176`** en résultat. Ceci ne réalise de toute évidence pas une fonction au sens mathématique, puisque bien que sans paramètre elle peut fournir des résultats différents à chaque appel.

Dans la suite, on utilisera le terme général de ***fonction*** pour désigner indifféremment une procédure, une fonction pure ou une fonction impure.

En Java, un même nom de fonction peut être utilisé pour des fonctions différentes, à condition que ces fonctions diffèrent par les types ou le nombre de paramètres. L'usage de ces fonctions n'est pas

ambigu, car on voit très bien, et le compilateur aussi, quels sont les types et le nombre des paramètres effectifs d'un appel et cela détermine quelle fonction est utilisée. Par exemple, on peut utiliser le même nom, **`maximum`**, pour la fonction qui rend en résultat le plus grand parmi deux caractères (selon l'ordre alphabétique) :

```
static char maximum(char c1, char c2) {
     if (c1>c2) {return c1;} else {return c2;}
     (si c1>c2 alors le résultat est c1, sinon le résultat est c2)
}
```

et pour la fonction qui rend en résultat le plus grand de deux nombres entiers :

```
static int maximum(int i, int j) {
     if (i>j) {return i;} else {return j;}
}
```

On appelle cela la "surcharge" des noms de fonction ("*overloading*" en anglais). Il ne faut pas en abuser, mais c'est souvent pratique. Cela évite d'inventer des noms différents pour des fonctions similaires sur des domaines différents. D'ailleurs les opérateurs du langage utilisent cette pratique, puisque par exemple les symboles `+`, `-`, `*`, `/`, `>`, `<`, `==`... servent à désigner les opérations usuelles, et cependant différentes, sur les entiers, les réels...

3.4 Commentaires de spécification d'une fonction

objectif : *savoir commenter sans ambiguïté les services rendus par une fonction. Apprendre à distinguer "résultats" et "effets". Savoir énoncer "ce que ça rend" ou " ce que ça fait" sans la moindre allusion à "comment c'est fait".*

Une fonction est destinée à des utilisateurs (éventuellement soi-même). Il est essentiel de documenter correctement ce qu'elle offre. Ceci se fait au moyen d'un commentaire de spécification. Un tel commentaire doit énoncer avec rigueur et sans ambiguïté ce que réalise la fonction. Il doit se limiter à dire ce que réalise la fonction et non pas "comment" elle le réalise car cela ne concerne pas l'utilisation de la fonction.

Il est bon que la présentation des commentaires de spécification soit toujours la même, cela améliore la lisibilité des programmes. Voici ce que nous proposons :

- On le place juste après l'en-tête de la fonction, en début du corps.
- Il est composé de trois rubriques, présentes uniquement si elles sont nécessaires :

  Prérequis : indique les conditions dans lesquelles l'appel est permis. C'est une formule logique concernant les paramètres et éventuellement l'environnement extérieur et l'état de certaines variables globales.

  Effet : indique les effets sur l'extérieur ou sur certaines variables globales.

  Résultat : indique quelle valeur est rendue en résultat, en fonction des paramètres et éventuellement de l'extérieur et de certaines variables globales.

Une fonction pure n'a pas de rubrique "effet" (elle n'offre qu'un résultat), une procédure n'a pas de rubrique "résultat" (elle n'offre qu'un effet) et une fonction impure a à la fois une rubrique "effet" et une rubrique "résultat".

Exemple de commentaires de spécification d'une fonction :

```
static double racineCarre(double x){
// prérequis : x>=0
// résultat : la racine carrée de x
  ...
}
```

Exemple de commentaires de spécification d'une procédure :

```
static void affichePersonne(String n, String p, int a){
// effet : affiche les caractéristiques d'une personne de nom n
// prénom p, âge a sous la forme "nom : n, prénom : p, âge : a ans"
  System.out.print("nom : "); System.out.print(n);
  System.out.print("prénom : "); System.out.print(p);
  System.out.print("âge : "); System.out.print(a);
  System.out.print(" ans");
}
```

3.5 Instructions et expressions

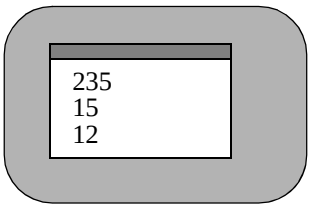
objectif : *savoir utiliser les instructions élémentaires.*

3.5.1 Séquences d'instructions

Les instructions figurent dans ce qu'on appelle le ***corps*** des fonctions. En Java, les instructions élémentaires sont systématiquement délimitées à droite par un "***;***". Une instruction indique quoi faire à l'ordinateur. La combinaison la plus simple est la combinaison séquentielle d'instructions : les instructions placées les unes à la suite des autres sont exécutées l'une après l'autre. Lors d'un appel de la fonction, la suite d'instructions ci-après est exécutée dans l'ordre *(1)* puis *(2)* puis *(3)* :

```
{
  System.out.println(235); (1)
  System.out.println(9+6); (2)
  System.out.println(12); (3)
}
```

L'exécution de ces instructions affiche donc successivement "235" puis "15" puis "12".



3.5.2 Instructions d'affichage à l'écran

En Java, les instructions d'affichage à l'écran sont :

`void System.out.print(`*diversType* **`x)`**
et **`void System.out.println(`***diversType* **`x)`**

La première forme, **`print`**, affiche simplement **`x`**, et la seconde, **`println`**, provoque en plus un passage au début de la ligne suivante après l'affichage de **`x`**.

Ce sont des procédures de la bibliothèque standard de Java. Elles admettent divers types pour le paramètre **`x`**. Il existe une procédure d'impression pour chaque type primitif scalaire. L'impression est faite selon le format usuel : notation décimale pour les entiers, notation avec point et exposant pour les réels, **`true`** et **`false`** pour les booléens...

Il existe également des procédures d'affichage d'un texte, c'est-à-dire d'une chaîne de caractères de longueur quelconque :

```
void System.out.print(String texte)
void System.out.println(String texte)
```

Le type chaîne de caractères, **`String`**, sera vu plus loin. Pour afficher des messages il suffit de savoir qu'une valeur de type chaîne de caractères se note au moyen de la chaîne elle-même entre doubles apostrophes, comme par exemple **`"coucou"`**.

Voici un exemple de programme qui utilise l'affichage de données de divers types :

```
class TestAffichage {
  public static void main(String[] arg) {
     System.out.print("la constante PI : "); ← affichage d'un texte
     System.out.println(Math.PI); ← affichage d'un réel
     System.out.print("la constante E : "); ← affichage d'un texte
     System.out.println(Math.E); ← affichage d'un réel
     System.out.print("le nombre de nains : "); ← affichage d'un texte
     System.out.println(7); ← affichage d'un entier
     System.out.print("la 1ère lettre de l'alphabet : "); ← un texte
     System.out.println('a'); ← affichage d'un caractère
     System.out.print("3 est-il plus grand que 8 ? : "); ← un texte
     System.out.println(3>8); ← affichage d'un booléen
  }
}
```

et voici son effet sur l'écran :

```
la constante PI : 3.141592653589793
la constante E : 2.718281828459045
le nombre de nains : 7
la 1ère lettre de l'alphabet : a
3 est-il plus grand que 8 ? : false
```

3.5.3 Expression de concaténation de chaînes de caractères

Les affichages se font en définitive toujours sous forme de chaînes de caractères. Si les fonctions d'affichage telles que **`System.out.print`** admettent tous les types primitifs du langage, c'est par l'intermédiaire de conversions "standard" en chaînes de caractères que cet affichage peut se faire.

Les chaînes de caractère, type **`String`** en Java, peuvent être élaborées par calcul, notamment grace à l'opération de ***concaténation***. La concaténation de deux chaînes ***c1*** et ***c2*** est la chaîne qui commence comme ***c1*** et se termine par ***c2***. Cette opération est notée **+** en Java (comme l'addition, mais c'est sans ambiguïté).

Ainsi :

```
"tant va l" + "a chruche à l'eau"
```

donne en résultat **`"tant va la chruche à l'eau"`**

De plus, si un des opérandes du symbole opératoire **+** est une chaîne de caractères, le symbole **+** est compris (par le compilateur) comme signifiant la concaténation et l'autre opérande est converti en chaîne de caractères selon la conversion standard. Ainsi on peut concaténer une chaîne de caractère avec tout type primitif.

Exemple :

```
int k = 12;
String s = "il y a " + k + "oeufs dans le panier";
```

La chaîne **`s`** vaut **`"il y a 12 oeufs dans le panier"`**.

En règle générale, il vaut mieux éviter de rédiger des "procédures qui affichent". Pour répondre aux besoins de communications d'un programme avec l'extérieurs, est bien préférable de rédiger des ***fonctions qui rendent des chaînes de caractère en résultat***. Ceci est très facile à réaliser grace à la concaténation.

Une telle fonction est d'usage plus général : si on veut réaliser un affichage, il suffit d'afficher le résultat de la fonction dans une procédure de niveau supérieur (dans le main par exemple), mais si on veut on peut en faire autre chose que de l'afficher.

3.5.4 Instruction de retour de fonction

Pour rendre son résultat, une fonction dispose de l'instruction de retour dont la forme générale est :

`return` ***expression*****`;`**

Exemple, fonction qui rend en résultat la moyenne de trois nombres :

```
static int moyenneDe3Nombres(int i, int j, int k) {
     return (i+j+k)/3;}
}
```

L'expression doit être du type du résultat de la fonction. Cette instruction provoque deux choses :

- elle évalue l'expression et donne sa valeur comme résultat de la fonction,
- elle termine l'exécution de la fonction.

Il peut y avoir plusieurs instructions de retour dans le corps d'une fonction, comme par exemple :

```
static int maximum(int i, int j) {
     if (i>j) {return i;} else {return j;}
}
```

si **i** est plus grand que **j**, c'est l'instruction **return i** qui est exécutée et termine en rendant en résultat la valeur de **i**, sinon c'est l'instruction **return j** qui est exécutée et termine en rendant en résultat la valeur de **j**.

On peut ne pas aimer cette multitude d'instructions **return** disséminées dans le corps d'une fonction. Pour faciliter la mise au point et la maintenance des programmes, on peut préférer une seule instruction **return** à la fin du texte de la fonction. Pour cela il suffit de donner un nom au résultat au moyen d'une variable temporaire[1], par exemple **resul** et de terminer la fonction par l'instruction **return resul**. Voici la fonction précédente programmée selon ce principe :

```
static int maximum(int i, int j) {
     int resul;
     if (i>j) {resul=i;} else {resul=j;}
     return resul;
}
```

Les procédures, sans résultat, peuvent également utiliser une instruction **return** sans expression accompagnatrice :

```
return ;
```

Le seul effet de cette instruction est de terminer l'exécution de la procédure. Pour des raisons de lisibilité et de maintenance, il vaut mieux éviter cette instruction, sauf dans des cas très simples, et laisser se terminer la procédure par la fin de son texte, auquel cas l'instruction **return** est inutile.

3.5.5 Expressions

3.5.5.1 Ce qu'est une expression

Un ***expression*** sert à exprimer le calcul d'une valeur. ***Une expression n'est pas une instruction***, elle apparaît dans une instruction pour spécifier une valeur qui intervient dans l'instruction. Nous venons de voir un exemple avec l'instruction **return** : dans **return a+b+c;** c'est "**return a+b+c;**" en entier qui est une instruction, "**a+b+c**" n'est qu'une expression dont l'évaluation, à savoir la somme de **a**, **b** et **c**, détermine la valeur rendue en résultat par ce **return**.

Les expressions sont construites de façon classique, à l'aide des opérateurs du langage, d'appels de fonctions, de notations de valeurs, de constantes et de variables, en utilisant les parenthèses pour construire des expressions aussi compliquées que l'on veut en combinant des expressions plus simples. Voici quelques exemples d'expressions, accompagnées de leur type et de leur valeur :

12 l'entier *12*
nombreDeCartes l'entier *32*
12+4 la somme de *12* et de *4*, à savoir l'entier *16*
maximum(4,maximum(8,2)) le maximum de *4*, *8* et *2*, à savoir *8*

Comme le montrent ces exemples, certaines expressions peuvent être très simples : une notation de valeur ou un nom de constante ou de variable constitue une expression.

1. L'utilisation des variables sera vue plus loin. Dans cet exemple la variable ne sert qu'à nommer un résultat de calcul pour s'en servir en un autre endroit que celui où il est calculé.

3.5.5.2 Où se rencontrent les expressions

Les expressions se rencontrent essentiellement en deux endroits :

- dans les rendus de résultat de fonction : **`return`** ***expression*****`;`**
- dans les appels de fonction, pour spécifier la valeur des paramètres effectifs : ... **`maximum(`*****expression*****$_1$`,`** ***expression*****$_2$`)`** ... par exemple **`maximum(12+3,14)`** qui vaut **`15`**.
-

3.5.5.3 Usage des parenthèses

Dans une expression compliquée, l'ordre d'évaluation des opérations peut être entièrement contrôlé grâce aux parenthèses :

`15-(3+2)` s'évalue en *10*
`(15-3)+2` s'évalue en *14*

On peut se dispenser de certaines parenthèses. Les opérateurs sont classés par priorité selon l'ordre suivant, en commençant par les plus prioritaires :

- **`-`** (moins unaire), **`!`** (négation),
- **`*`** (multiplication), **`/`** (division),
- **`+`** (addition), **`-`** (soustraction)
- **`>`**, **`<`**, **`>=`**, **`<=`**, **`==`**, **`!=`** (comparaisons)
- **`&`**, **`&&`** (et)
- |, || (ou)

En l'absence de parenthèses, les opérateurs sont évalués selon leur ordre de priorité et en cas d'égalité de priorité ils sont évalués en s'associant à gauche (c'est-à-dire "de gauche à droite"). Par exemple : **`15-3*6+2`** s'évalue comme (**`15-(3*6))+2`** c'est-à dire en **`-1`**.

Conseil : pour éviter les erreurs dues à un mauvais usage des priorités, ***il vaut mieux mettre des parenthèses***.

3.5.5.4 Différence essentielle entre impératif et fonctionnel

Pour bien appréhender la programmation impérative, il est essentiel de bien percevoir la différence entre ***expressions*** et ***instructions***. Une grande partie de la puissance mais aussi de la difficulté des langages impératifs vient de ce qu'ils combinent partout deux notions très différentes : les expressions qui ***ont une valeur*** et les instructions qui ***agissent***. Dans la vie courante et le langage courant, on rencontre également ces deux notions. Voici une expression : "***le bus numéro 12***", c'est une valeur qui en soi n'indique aucune action, ce n'est pas une instruction. Pour construire une instruction, il faut généralement rajouter un verbe d'action : "***prend*** *le bus numéro 12*" ou "***conduit*** *le bus numéro 12*" sont des instructions.

C'est la différence essentielle entre langages fonctionnels et langages impératifs : un langage fonctionnel est un langage uniquement composé d'expressions, c'est un langage sans instructions. Un langage fonctionnel "pur" ne permet de rien faire, car le "faire" n'est pas dans ses notions de base. Un langage fonctionnel ne permet que d'énoncer des valeurs (ce qui n'est déjà pas si mal, car ces valeurs peuvent être très difficiles à calculer). Pour rendre un langage fonctionnel utilisable, soit on lui rajoute "autour" un système interactif d'évaluation qui, ***au cours du temps***, lit des expressions, les évalue et affiche leurs résultats, soit on renonce à l'aspect "purement fonctionnel" en intégrant au langage, de façon toujours plus ou moins scabreuse, des moyens d'interaction avec l'environnement d'exécution, des entrées-sorties par exemple.

3.5.6 Appel de procédure - Appel de fonction

Une instruction d'appel de procédure provoque l'exécution des instructions de cette procédure. La forme générale est :

***nomDeLaProcédure*`(`*expression*$_1$`,` *expression*$_2$`...)``;`**

Les expressions sont évaluées, leurs valeurs sont attribuées aux paramètres formels de la procédure, puis les instructions de la procédure sont exécutées. Lorsque la procédure se termine, l'exécution se poursuit à la suite de cette instruction d'appel. Voici un exemple d'appels de procédures :

```
class TestAppelDeProcedure {

   Définition de procédure
   static void afficheSurfaceDeCarre(double cote) {
      System.out.print("la surface du carré de côté "); (p1)
      System.out.print(cote); (p2)
      System.out.print(" vaut "); (p3)
      System.out.println(cote*cote); (p4)
   }

   Programme principal qui appelle la procédure
   public static void main(String[] arg) {
      afficheSurfaceDeCarre(2.0); (a1)
      afficheSurfaceDeCarre(3.0); (a2)
   }
}
```

L'exécution commence par la procédure principale, **`main`**.

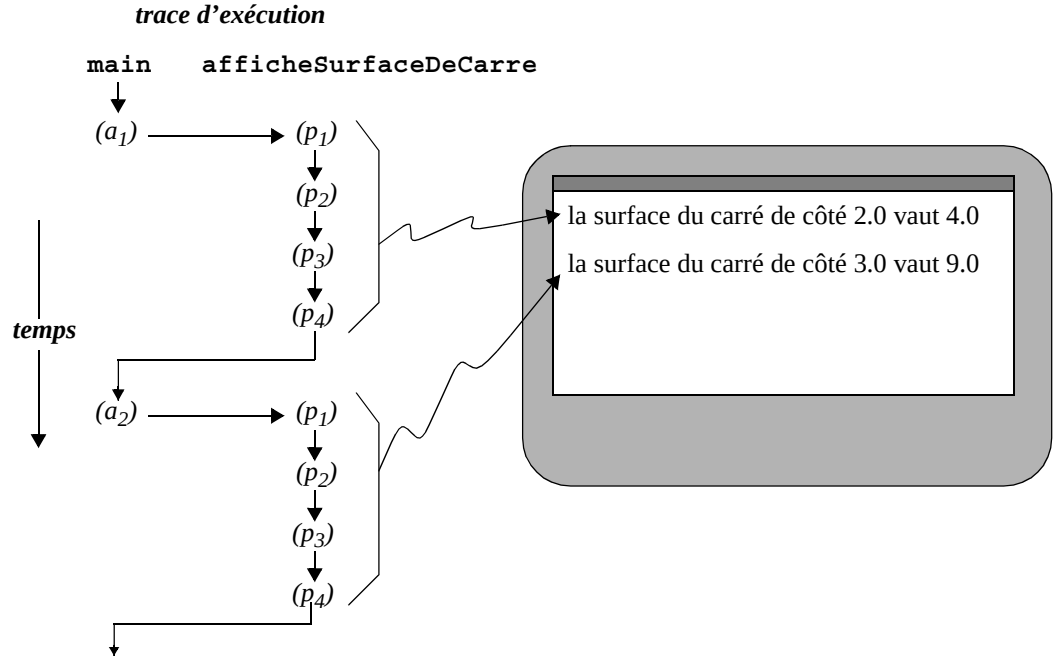
La première instruction exécutée *(a$_1$)* est un appel de la procédure **`afficheSurfaceDeCarre`** avec *2.0* pour paramètre effectif. Ceci a pour effet d'exécuter les instructions *(p$_1$)*, *(p$_2$)*, *(p$_3$)*, *(p$_4$)* du corps de la procédure qui affichent "la surface du carré de côté 2.0 vaut 4.0".

L'exécution retourne dans la procédure principale.

L'instruction *(a$_2$)* est exécutée. C'est un nouvel appel de la procédure **`afficheSurfaceDe-Carre`** avec *3.0* pour paramètre effectif. Ceci a pour effet d'exécuter les instructions *(p$_1$)*, *(p$_2$)*, *(p$_3$)*, *(p$_4$)* qui affichent "la surface du carré de côté 3.0 vaut 9.0".

On appelle ***trace d'exécution*** la séquence temporelle des exécutions d'instructions.

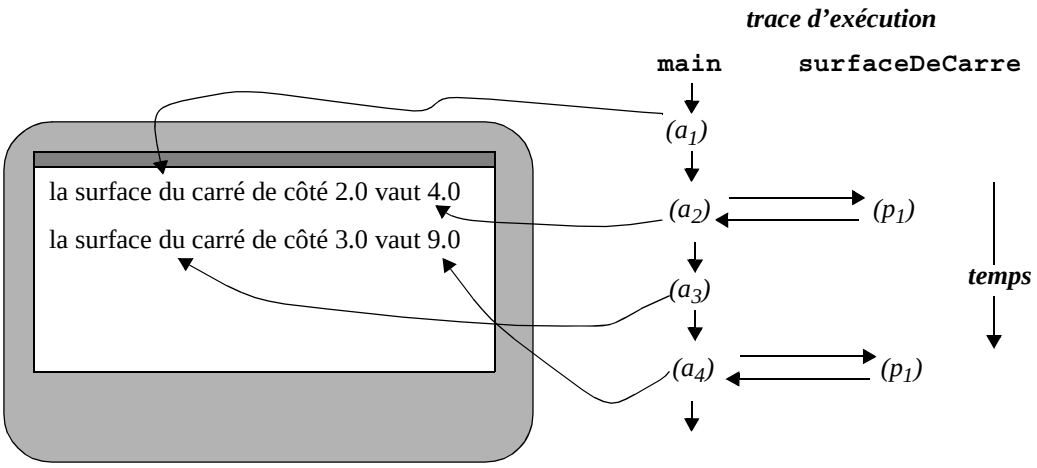
Le schéma suivant montre la trace d'exécution de l'exemple précédent et son effet sur l'écran.



Pour un appel de fonction il se passe à peu près la même chose. Cependant un appel de fonction ***n'est pas une instruction***, c'est une ***expression*** qui vaut le résultat de la fonction. Il faut donc faire apparaître l'appel en tant qu'expression dans une instruction qui exploite ce résultat. L'exemple suivant illustre l'appel de fonction. C'est un programme qui provoque un affichage identique au précédent, mais les actions d'écriture sont dans la procédure principale **`main`**, la fonction **`surfaceDeCarre`** ne faisant que calculer et rendre en résultat la surface d'un carré.

```
class TestAppelDeFonction {

   Définition de fonction
   static double surfaceDeCarre(double cote) {
      return cote*cote; (p1)
   }

   Programme principal qui appelle la fonction
   public static void main(String[] arg) {
      System.out.print("la surface du carré de côté 2.0 vaut "); (a1)
      System.out.println(surfaceDeCarre(2.0)); (a2)
      System.out.print("la surface du carré de côté 2.0 vaut "); (a3)
      System.out.println(surfaceDeCarre(3.0)); (a4)
   }
}
```

La trace d'exécution de ce programme est illustrée ci-dessous :



On remarquera ici une petite difficulté dans le dessin : l'appel de fonction **`surfaceDeCarre(2.0)`** est au sein même de l'instruction (a_2). Pendant que la fonction est exécutée, l'instruction (a_2) n'est pas terminée. Lorsque la fonction se termine, le flot de contrôle revient "dans" l'instruction (a_2) pour la terminer (faire l'impression de la valeur retournée ici).

3.5.7 Variables et instruction d'affectation

Contrairement à une constante, une ***variable*** peut prendre plusieurs valeurs successives pendant l'exécution du programme. Une déclaration de variable se présente sous la forme :

type nom ; ou bien ***type nom = valeurInitiale ;***

Le changement de valeur d'une variable se fait au moyen d'une ***instruction d'affectation***. En Java, la forme générale de cette instruction est :

variable = expression ;

Le résultat de l'expression doit être du type de la variable. Une variable ***doit être initialisée***, c'est à dire recevoir une valeur par affectation, avant d'être utilisée. Le compilateur Java le vérifie et refuse tout programme dans lequel une variable risque de ne pas être initialisée avant son utilisation.

Exemples :

`int k=2; ... k ...` utilisation de **`k`** ayant pour valeur 2

`k=12; ... k ...` utilisation de **`k`** ayant pour valeur 12

`k=9; ... k ...` utilisation de **`k`** ayant pour valeur 9

Remarque : le type n'est indiqué qu'une seule fois, à l'endroit de déclaration de la variable.

En toute rigueur, ***l'usage des variables n'est nécessaire que pour réaliser des itérations*** (voir plus loin). Mais le langage n'interdit pas de les utiliser dans d'autres circonstances, là où des constantes auraient suffi. Voici par exemple trois façons de calculer le maximum de trois nombres **`a`**, **`b`** et **`c`** en

disposant de la fonction **max** qui rend en résultat le maximum de deux nombres :

- version 1 : sans intermédiaire : `int maxABC = max(max(a,b),c);`
- version 2 : avec une constante pour nommer le résultat intermédiaire :
  `final int maxAB = max(a,b); int maxABC = max(maxAB,c);`
- version 3 : en utilisant une variable pour calculer "progressivement" le résultat :
  `int maxABC = max(a,b); int maxABC = max(maxABC,c);`
  cette dernière version est sans doute la moins "élégante" des trois.

3.5.8 Instructions de lecture du clavier

Les données d'un programme sont rarement fixées au moment de la rédaction du programme. Elles sont généralement fournies au moment de l'exécution du programme, par exemple introduites au clavier.

Le captage des données par le programme est fait par une ***opération de lecture***. En Java, une opération de lecture se présente comme un appel de fonction. Cette fonction attend que l'utilisateur introduise des données et rend en résultat les données introduites. La lecture se présente donc comme une expression.

Les données introduites peuvent être de divers types, c'est pourquoi il y a une fonction de lecture pour chaque type de données susceptibles d'être lues. Nous utiliserons les fonctions suivantes[1] :

`int Lecture.unEntier()` rend en résultat l'entier frappé au clavier

`double Lecture.unReel()` rend en résultat le nombre réel frappé au clavier

`char Lecture.unCar()` rend en résultat le caractère frappé au clavier

`String Lecture.chaine()` rend en résultat la chaîne des caractères frappés au clavier

L'utilisateur est censé introduire les données selon les formats usuels. Excepté pour la lecture d'un caractère, les données sont introduites sous forme d'une suite de caractères de longueur variable. Des délimiteurs doivent donc être définis pour séparer les données. Les séparateurs choisis sont l'espace et le passage à la ligne (`' '`, `'\r'` ou `'\n'`).

Voici par exemple une séquence d'instructions qui affecte aux variables **i**, **c**, **x** et **message** respectivement un entier, un caractère, un réel et une chaîne :

```
int i = Lecture.unEntier();
char c = Lecture.unCar();
double x = Lecture.unReel();
String message = Lecture.chaine();
```

Lorsque cette séquence est exécutée, si l'utilisateur frappe le texte suivant au clavier :

```
12  a6.02E23  coucou
```

i prend la valeur *12*, **c** le caractère `'a'`, **x** la valeur $6.02\ 10^{23}$ et **message** la valeur `"coucou"`.

1. Ces fonctions ne sont pas les fonctions standards de Java pour lire des données au clavier. Ce sont des fonctions plus faciles à utiliser introduites pour ce cours. Leurs définitions en termes des fonctions standards de Java sont données en annexe.

Pour disposer de ces fonctions de lecture, il faut placer en première ligne du fichier source du programme la directive :

```
import es.*;
```

Exemple : le programme suivant calcule et affiche le volume d'un parallélépipède rectangle dont la largeur, la longueur et la hauteur sont fournies au clavier.

```
import es.*;

class Parallelepipede {
  public static void main(String[] arg) {

    final int largeur = Lecture.unEntier();
    final int longueur = Lecture.unEntier();
    final int hauteur = Lecture.unEntier();
    final int surfaceDeBase=largeur*longueur;

    System.out.print("surface de base = ");
    System.out.println(surfaceDeBase);
    System.out.print("volume = ");
    System.out.println(surfaceDeBase*hauteur);
  }
}
```

Comme le montre cet exemple, des constantes peuvent être initialisées par des lectures. Pour alléger le texte, on utilise parfois des variables (abusivement, en omettant le vocable **`final`**) :

```
int largeur = Lecture.unEntier();
```

QCM 3.1 On considère le programme suivant :

```
import es.*;

class TestLectures{
  public static void main(String[] z){
    int x = 12;
    String s = "coucou";
    x = Lecture.unEntier();
    s = Lecture.chaine();
    System.out.print("x="+x); System.out.println(" s="+s);
  }
}
```

Au moment de l'exécution, l'utilisateur entre au clavier : **`56 toto`**

- ❑ 1 - Il ne se passe rien et le programme termine son exécution.
- ❑ 2 - Il ne se passe rien et le programme ne se termine pas.
- ❑ 3 - Il y a une erreur pendant l'exécution.
- ❑ 4 - Le programme affiche : **`x=56 s=toto`**
- ❑ 5 - Le programme affiche : **`x=12 s=coucou`**

Au moment de l'exécution, l'utilisateur entre au clavier : **`tutu toto`**

- ❑ 6 - Il ne se passe rien et le programme termine son exécution.
- ❑ 7 - Il ne se passe rien et le programme ne se termine pas.
- ❑ 8 - Il y a une erreur pendant l'exécution.
- ❑ 9 - Le programme affiche : **`x=tutu s=toto`**
- ❑ 10 - Le programme affiche : **`x=12 s=coucou`**

Au moment de l'exécution, l'utilisateur entre au clavier : **`56 95`**

- ❑ 11 - Il ne se passe rien et le programme termine son exécution.
- ❑ 12 - Il ne se passe rien et le programme ne se termine pas.
- ❑ 13 - Il y a une erreur pendant l'exécution.
- ❑ 14 - Le programme affiche : **`x=56 s=95`**
- ❑ 15 - Le programme affiche : **`x=12 s=coucou`**

Au moment de l'exécution, l'utilisateur n'entre rien au clavier.

- ❑ 16 - Il ne se passe rien et le programme termine son exécution.
- ❑ 17 - Il ne se passe rien et le programme ne se termine pas.
- ❑ 18 - Il y a une erreur pendant l'exécution.
- ❑ 19 - Le programme affiche : **`x=56 s=95`**
- ❑ 20 - Le programme affiche : **`x= s=`**

QCM 3.2

On considère le programme suivant :

```
class TestFonctions{

  static int x;

  static int plusUn(int k){
  // ???
    return k+1;
  }

  static void affichePlusUn(int x){
  // ???
    System.out.print(x+1);
  }

  static void ajouteUnAX(){
  // ???
    x=x+1;
  }

  public static void main(String[] z){
    x=12;
    System.out.print("x="+x);
    System.out.print(" 6+1="); affichePlusUn(6);
    System.out.print(" x="+x); ajouteUnAX();
    System.out.print(" x="+x);
    System.out.println(" 8+1="+plusUn(8));
  }
}
```

- ❑ 1 - Commentaire de spécification de **plusUn** : **// résultat : k+1**
- ❑ 2 - Commentaire de spécification de **plusUn** : **// effet : ajoute 1 à k**
- ❑ 3 - Commentaire de spécification de **plusUn** : **// effet : affiche k+1**
- ❑ 4 - Commentaire de spécification de **plusUn** : **// effet : ajoute 1 à x**

- ❑ 5 - Commentaire de spécification de **affichePlusUn** : **// résultat : x+1**
- ❑ 6 - Commentaire de spécification de **affichePlusUn** : **// effet : ajoute 1 à k**
- ❑ 7 - Commentaire de spécification de **affichePlusUn** : **// effet : affiche x+1**
- ❑ 8 - Commentaire de spécification de **affichePlusUn** : **// effet : ajoute 1 à x**

- ❑ 9 - Commentaire de spécification de **ajouteUnAX** : **// résultat : x+1**
- ❑ 10 - Commentaire de spécification de **ajouteUnAX** : **// effet : ajoute 1 à x**
- ❑ 11 - Commentaire de spécification de **ajouteUnAX** : **// effet : affiche x+1**

❏ 12 - La première instruction du main, **`x=12;`**, affecte la variable globale **`x`**.

❏ 13 - La première instruction du main, **`x=12;`**, donne la valeur 12 au paramètre **`x`** de la fonction **`affichePlusUn`**.

❏ 14 - L'instruction du main **`affichePlusUn(6);`** affiche 7.

❏ 15 - L'instruction du main **`affichePlusUn(6);`** donne la valeur 7 à la variable globale **`x`**.

❏ 16 - Le programme affiche **`x=12 6+1=7 x=7 x=8 8+1=9`**.

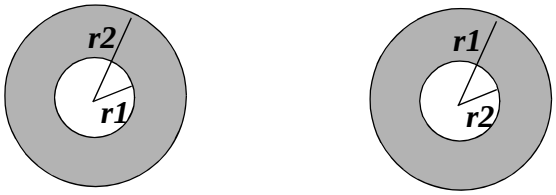
❏ 17 - Le programme affiche **`x=12 6+1=7 x=12 x=13 8+1=9`**.

❏ 18 - Le programme affiche **`x=12 6+1=7 x=13 x=14 8+1=9`**.

Exercice 3.1 Définition et appel de fonctions : surface d'une couronne

objectif : *savoir définir, combiner et utiliser des fonctions.*

On désire calculer la surface de couronnes délimitées par les cercles concentriques de rayons *r1* et *r2*. Les rayons sont quelconques : *r1* peut être supérieur, égal ou inférieur à *r2* :



Pour cela on décide de définir les fonctions :

- **`surfaceCercle`** : surface d'un cercle de rayon donné,
- **`surfaceCouronne`** : surface d'une couronne délimitée par deux cercles de rayon donnés.

Une surface ne doit pas être un nombre négatif. Pour assurer cela, la bibliothèque standard de Java offre la fonction :

`double Math.abs(double x)` : résultat, la valeur absolue de **`x`** (**`x`** si **`x`**≥0, **`-x`** sinon)

On pourra également utiliser avec profit la constante **`Math.PI`** offerte par la bibliothèque.

Rédiger un programme **`TestCouronne`** qui affiche la surface de la couronne délimitée par

- des cercles de rayons r1=2,3 et r2=4,1,
- des cercles de rayons r1=4,1 et r2=2,3,
- des cercles dont les rayons sont lus au clavier.

Exercice 3.2 Instructions de lecture et d'affichage

objectif : *savoir lire depuis le clavier des données de divers types*

Les chaînes de caractères sont de type **`String`** en Java. Par exemple voici comment se déclare et se lit une variable destinée à contenir le nom d'une personne :

```
String nom = Lecture.chaine();
```

Rédiger un programme qui lit au clavier :

- l'année en cours (un nombre entier, par exemple 2006),
- le nom et le prénom d'une personne (des chaînes de caractères),
- l'année de naissance de cette personne (un nombre entier),

et affiche un texte de la forme :

`Monsieur` *prénom nom*`,` **`vous avez`** *âge* **`ans`**

où *prénom*, *nom* et *âge* sont le nom, le prénom et l'âge à l'année en cours de la personne.

CHAPITRE 4 Nommage, portées d'identification, durées de vie, modularité

objectif : *savoir où et comment donner des noms aux éléments d'un programme, quand naissent et meurent les données.*

Le but de ce chapitre est d'apporter des précisions sur trois aspects importants :

- Le ***nommage*** : à quelle occasion on doit donner un nom à une chose (constante, variable, fonction...) et comment on nomme les choses.
- Les ***portées d'identification*** : comment les noms sont associés aux choses.
- Les ***durées de vie*** : quand naissent et meurent les choses pendant l'exécution.

4.1 Nommage

Les noms qui apparaissent dans les programmes pour désigner les choses s'appellent des ***identificateurs***. Ils doivent obéir à quelques règles simples :

- commencer par une lettre et se poursuivre par des lettres ou des chiffres,
- ne pas être identique à un vocable du langage tel que, en Java :

  `class`, **`int`**, **`boolean`**, **`if`**, **`else`**, **`while`**...

A ces règles de syntaxe s'ajoutent quelques conventions et conseils :

- Choisir les noms les plus signifiants possibles : des noms bien choisis complètent, voire remplacent avantageusement les commentaires. Il ne faut pas hésiter à utiliser des mots longs si nécessaire. Par exemple :

  ```
  final int graviteTerrestre = 9.81;
  ```

- Respecter les normes communément admises, surtout si elles ont fait leur preuve. La norme la plus répandue actuellement est :

  Les noms de classes commencent par une majuscule :

  ```
  class TestMoyenne
  ```

  Les noms de données et de fonctions commencent par une minuscule :

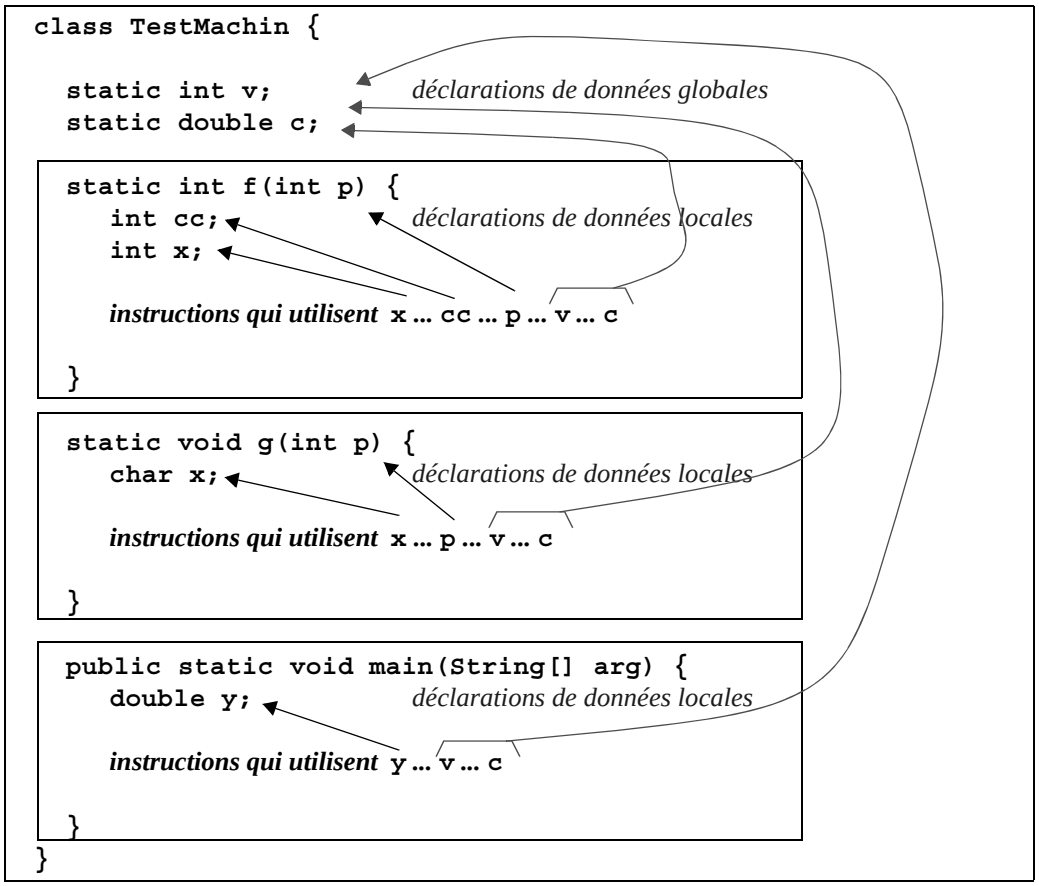
  ```
  String prenom;          int maximum(int i, int j){...}
  ```

  Pour les noms composés, les séparations se font au moyen de majuscules :

  ```
  ageDuCapitaine          moyenneDeTroisNombres
  ```

4.2 Portées d'identification

Le schéma suivant illustre la structure d'un programme simple doté de diverses déclarations de données :



Dans ce schéma, on a utilisé les mêmes noms pour désigner des choses différentes :

- une variable de la fonction **f** est appelée **x**,
- une variable de la fonction **g** est également appelée **x**.

Il n'y a pas de confusion possible grâce à la règle de ***portée des identifications*** : un identificateur introduit par une déclaration de donnée à l'intérieur d'une fonction désigne cette donnée seulement dans le corps de cette fonction. Ainsi, dans le corps de **f**, **x** désigne la variable de type **int** déclarée dans **f**, alors que dans le corps de **g**, **x** désigne la variable de type **char** déclarée dans **g**. Dans la procédure principale, **x** ne désignerait rien, et son emploi serait une erreur de syntaxe.

Il en est de même des identifications des paramètres des fonctions. Dans l'exemple, nous avons utilisé le même identificateur **p** pour désigner le paramètre de **f** et celui de **g**. Ces deux paramètres ne sont bien évidemment pas la même chose.

En revanche, l'identificateur **v** désigne, partout dans le programme, la variable déclarée

```
static int v;
```

au niveau global du programme (en dehors de toute fonction). Il en est de même de l'identificateur

c qui désigne partout la même donnée et des identificateurs de fonctions **f** et **g** qui sont utilisables partout pour désigner la fonction **f** et la procédure **g**.

Une donnée locale et une donnée globale peuvent porter le même nom. En cas d'homonymie, le nom désigne la donnée locale. Il vaut mieux éviter de telles homonymies car elles sont source d'erreurs.

4.3 Durées de vie

Les données déclarées au niveau global du programme, en dehors de toute fonction, et dotées du vocable **static**, telle que **static int v** dans l'exemple, s'appellent des ***variables globales***. Elles existent pendant toute l'exécution du programme. On dit que leur ***durée de vie*** s'étend du début à la fin de l'exécution du programme. Plus généralement c'est ce que signifie le vocable **static** : "qui existe en un seul exemplaire pendant toute l'exécution du programme"[1].

Les variables déclarées au sein d'une fonction, telle que **int x** dans la fonction **f**, **char x** dans la fonction **g** ou encore **double y** dans la procédure principale **main**, s'appellent des ***variables locales***. Elles existent uniquement pendant une exécution de la fonction dans laquelle elles sont déclarées. Leur durée de vie s'étend du début à la fin de l'exécution de cette procédure : elles sont créées au moment de l'appel de la fonction et sont détruites au retour.

Ceci signifie notamment qu'une variable locale n'a pas de valeur déterminée au début de l'exécution d'une fonction, alors qu'une variable globale a toujours une valeur déterminée, celle attribuée par sa dernière affectation.

Comme exemple simple d'usage de variables globales, on peut considérer le programme suivant qui imprime un texte avec un numéro de ligne en début de chaque ligne :

```
class Imprime LaCigaleEtLaFourmi {

   static int numeroDeLigne; // pour numéroter les lignes

  static void imprimeLigne(String ligne) {
    System.out.print('\n');
    System.out.print(numeroDeLigne);
    System.out.print("  "); System.out.print(ligne);
    numeroDeLigne=numeroDeLigne+1;
   }

   public static void main(String[] arg) {
      numeroDeLigne=1;
      imprimeLigne("La cigale ayant chanté");
      imprimeLigne("Tout l'été");
      imprimeLigne("Se trouva fort dépourvue");
      imprimeLigne("Quand la bise fut venue.");
   }
}
```

La variable **numeroDeLigne** est une variable globale. Elle est accessible depuis le programme principal qui l'initialise à 1, et par la procédure **imprimeLigne** qui imprime sa valeur en début de

1. Le vocable **static** qualifie donc les choses les "plus simples", qui existent tout le temps, en correspondance directe avec le texte de leur déclaration. Les choses "plus compliquées" telles que les variables locales de fonctions ou bien les composants d'objets (voir plus loin à propos des classes modèles d'objets) dont la durée de vie est égale à la durée d'exécution d'un appel de fonction ou à l'existence d'un objet, ne nécessitent ***aucun vocable particulier***.

ligne et l'incrémente de manière à y trouver une valeur augmentée de 1 lors de sa prochaine exécution. Ce programme imprime :

```
1  La cigale ayant chanté
2  Tout l'été
3  Se trouva fort dépourvue
4  Quand la bise fut venue.
```

Il ne faut pas abuser de l'usage de variables globales. Il est assez rare d'en avoir besoin. Il ne faut surtout pas les utiliser pour transmettre des arguments à une fonction : les paramètres sont faits pour cela. L'exemple précédent est à la limite d'un bon usage des variables globales. On peut à la place utiliser un paramètre supplémentaire **`numeroDeLigne`** pour indiquer le numéro de ligne à la procédure **`imprimeLigne`** :

```
class ImprimeLaCigaleEtLaFourmi {

  static void imprimeLigne(int numeroDeLigne, String ligne) {
    System.out.print("\n");
    System.out.print(numeroDeLigne);
    System.out.print("  "); System.out.print(ligne);
  }

  public static void main(String[] arg) {
     imprimeLigne(1,"La cigale ayant chanté");
     imprimeLigne(2,"Tout l'été");
     imprimeLigne(3,"Se trouva fort dépourvue");
     imprimeLigne(4,"Quand la bise fut venue.");
  }
}
```

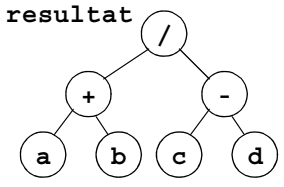
4.4 Identification temporaire - blocs d'instructions

On est souvent amené, au sein d'une fonction, à nommer un résultat intermédiaire. On a besoin de le faire si ce résultat intermédiaire doit être ***utilisé en plusieurs endroits*** du texte de la fonction. La syntaxe des expressions est suffisante si le résultat est simplement utilisé en un endroit. L'exemple suivant montre un calcul où une expression suffit et un calcul où une identification d'un résultat intermédiaire est utile :

calcul de *(a+b)/(c-d)*

```
resultat=(a+b)/(c-d)
```

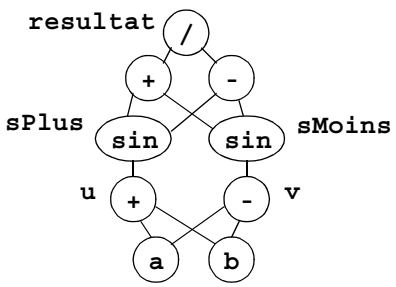
expression sous forme d'arbre



calcul de *(sin(a+b)+sin(a-b))/ (sin(a+b)-sin(a-b))*

```
double sPlus=sin(a+b);
double sMoins=sin(a-b);
resultat=(sPlus+sMoins)/(sPlus-sMoins);
```

expression sous forme de treillis



Si les résultats intermédiaires ne sont utilisés qu'en un seul endroit, c'est-à-dire si le calcul total s'exprime sous forme d'un arbre d'opérateurs et d'opérandes, on peut mettre directement son expression à l'endroit où il est utilisé, en tant que sous-expression. C'est le cas de l'exemple de gauche.

Si un résultat intermédiaire est utilisé en plusieurs endroits, on a intérêt, et c'est même parfois nécessaire, de nommer ce résultat intermédiaire et d'élaborer le calcul en plusieurs expressions distinctes. C'est le cas de l'exemple de droite. Le calcul total ne s'exprime pas sous forme d'un arbre mais sous forme d'un treillis d'opérateurs et d'opérandes.

Les langages de programmation permettent de déclarer des variables locales pour nommer ces résultats intermédiaires. Il faut ***déclarer ces variables le plus près possible des endroits d'utilisation***, et ne pas "polluer le programme" en les déclarant n'importe où. Il faut également, si c'est possible, ***leur donner une valeur à l'endroit de leur déclaration***. Ce sont des variables locales et leur durée de vie est donc le temps d'exécution de la fonction.

Rien n'interdit bien sûr de nommer un résultat intermédiaire dans le cas où il n'est utilisé qu'en un endroit. Cela peut même être plus clair, en évitant une expression peu lisible car trop longue ou en mettant en valeur ce résultat intermédiaire en le nommant de façon judicieuse. Un tel nommage remplace avantageusement certains commentaires. Mais il ne faut pas abuser des identifications inutiles. Des identifications intempestives attirent l'attention sur ce qui ne le mérite pas et dégradent la lisibilité en ne profitant pas du pouvoir expressif des expressions.

Java permet de placer des déclarations à peu près n'importe où dans le corps des fonctions : il faut en profiter.

Blocs d'instructions

De nombreux langages, Java en particulier, possèdent en plus une notion de ***bloc d'instructions***. Il s'agit simplement d'un regroupement d'instructions que l'on peut accompagner de déclarations. Ces déclarations sont alors locales au bloc : d'une part une variable ou constante ainsi déclarée a une durée de vie égale à la durée d'exécution du bloc et d'autre part l'identification introduite par la déclaration a une portée limitée à ce bloc.

... ***début de bloc*** entier *k*, réel *x*... instructions qui peuvent utiliser *k* et *x*... ***fin de bloc.***..

En Java, un bloc est délimité par des accolades "**`{...}`**", et tout ce qui est entre accolades constitue un bloc. On peut placer un bloc partout où on peut placer une instruction. Un des principaux intérêts du bloc est de limiter la portée d'identification des identificateurs et ainsi améliorer la lisibilité et la fiabilité des programmes. L'exemple suivant illustre l'usage d'un bloc en Java. Les variables **`u`**, **`v`**, **`sPlus`** et **`sMoins`** sont ici locales au bloc introduit.

```
double resultat;
...
{
   double u=a+b; double v=a-b;
   double sPlus=sin u; double sMoins=sin v;
   resultat=(sPlus+sMoins)/(sPlus-sMoins);
}
...
```

Ces variables ne servent que d'intermédiaires de calcul pour la variable **`resultat`**. La présence d'un bloc est "rassurante" pour le lecteur : il n'a pas à se demander si **`u`**, **`v`**, **`sPlus`** et **`sMoins`** servent à autre chose dans le programme, car ces variables sont inconnues en dehors du bloc où elles sont déclarées.

4.5 Modularité

De façon générale, la ***modularité*** consiste à découper les “gros” programmes en morceaux, chaque morceau concernant un ***thème***.

La modularité intervient à plusieurs niveaux. En Java, on peut distinguer trois niveaux :

- Les ***fonctions*** : une fonction réalise un morceau de traitement bien spécifié (calcul d’une fonction, production de certains effets...)
- Les ***classes*** : un programme est généralement composé de plusieurs classes, chacune s’occupant d’un thème. Par exemple, la classe **`Math`** de la bibliothèque standard regroupe toutes les fonctions et constantes mathématiques usuelles : sinus, cosinus, exponentielle, π...
- Les ***paquetages*** (**`package`**) : un paquetage regroupe plusieurs classes concernant un même domaine. Par exemple, le paquetage **`io`** (input-output) de la bibliothèque standard regroupe les classes qui permettent de lire et d’écrire des données.

La modularité présente de nombreux avantages :

- Le découpage en petites unités améliore la lisibilité et la maintenance des programmes.
- Si ces unités concernent un thème bien identifié, avec des fonctionnalités bien spécifiées, elles sont généralement réutilisables dans de nombreuses applications.

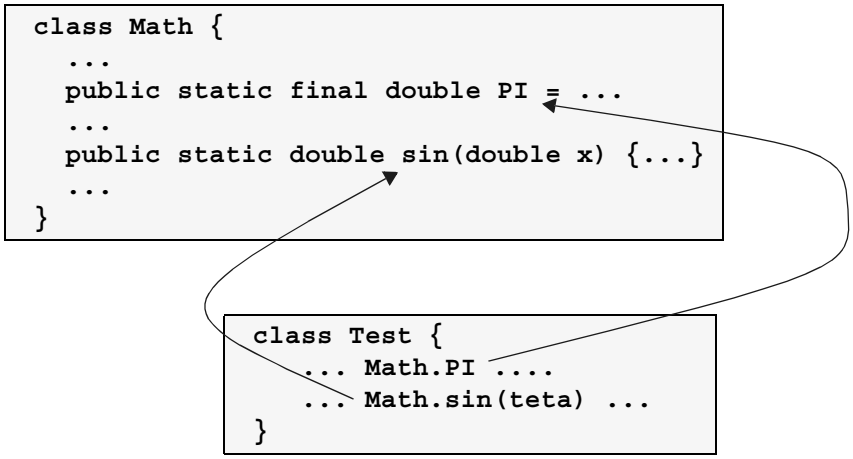
4.5.1 Découpage en classes

Les fonctions ou données déclarées dans une classe sont accessibles depuis les autres classes à condition d’être affublées du vocable **`public`**.

Si une fonction **`f`** ou une donnée **`x`** (statiques) sont déclarées dans une classe **`C`**, pour l’utiliser depuis une autre classe il faut utiliser une notation pointée :

`C.f()` ou **`C.x`**

C’est ainsi, par exemple, que l’on utilise les fonctions et données de la classe **`Math`** de la bibliothèque standard : **`Math.sin(x)`**, **`Math.PI`**...



Ou bien encore les fonctions de lectures depuis le clavier définies dans la classe **`Lecture`** du paquetage **`es`** offert pour les exercices : **`Lecture.chaine()`**, **`Lecture.unEntier()`** ...

4.5.2 Paquetages

En Java, les classes concernant un même domaine peuvent être regroupées en ***paquetages***. Les programmes sources des classes d'un paquetage appelé ***xxx*** commencent par la directive :

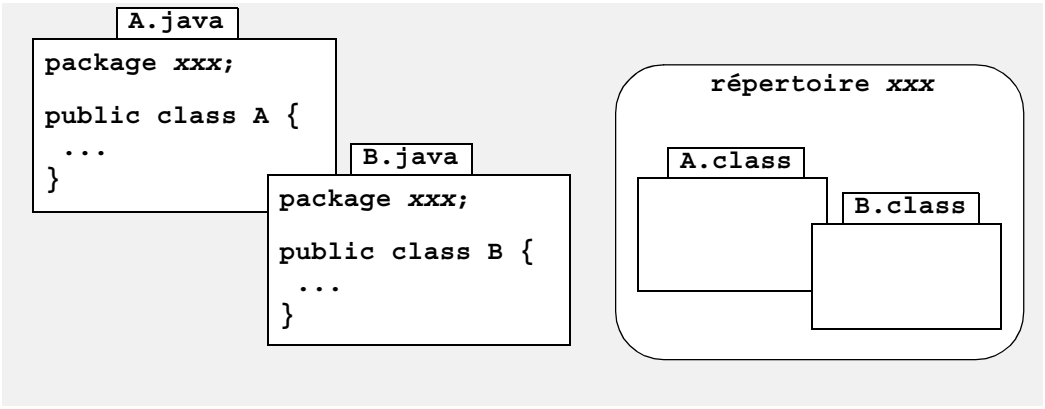
`package` ***`xxx;`***

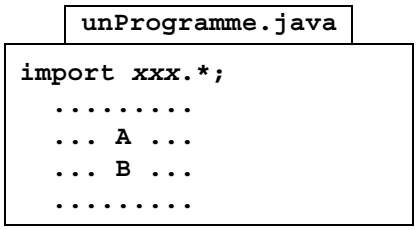
Les classes d'un paquetage doivent être placées dans un ***répertoire de même nom que le paquetage***. Au sein d'un paquetage, on a accès aux classes de ce paquetage et aux classes déclarées **`public`** des autres paquetages. Pour désigner une classe d'un autre paquetage, on utilise un nom absolu de la forme :

nom-du-paquetage **`.`** ***nom-de-classe***

Il est possible de désigner les classes par leur nom court, sans préciser le paquetage, en utilisant la directive **`import`** :

`import` ***`nom-de-paquetage.nom-de-classe;`*** pour utiliser une classe du paquetage
`import` ***`nom-de-paquetage.*;`*** pour pouvoir utiliser toutes les classes du paquetage.





Les paquetages permettent notamment de structurer les ***bibliothèques d'intérêt général***. La bibliothèque standard de Java est découpée en de nombreux paquetages, par exemple :

`java.io` : concerne les entrées et sorties, notamment pour accéder aux fichiers,

`java.awt` et **`javax.swing`** : concerne l'usage d'interfaces graphiques (fenêtres, boutons, zone de texte...)

`java.applet` : concerne la programmation d'***applets*** (programmes java téléchargés via le Web et exécutés sur la machine de l'utilisateur)

`java.net` : concerne les accès à un réseau,

etc...

Pour pouvoir utiliser les paquetages, le compilateur et l'exécuteur de Java doivent connaître les répertoires où se trouvent ces paquetages. Cela est indiqué par une ***variable d'environnement*** qui s'appelle **`CLASSPATH`** (Une variable d'environnement est une variable que l'on définit, par une commande, au niveau du système d'exploitation de l'ordinateur et qui sert à paramétrer les fonctions de base du système et les applications). La variable d'environnement **`CLASSPATH`** doit contenir la liste des répertoires qui contiennent les répertoires des paquetages, par exemple :

```
/udd/API/paquetagesMaison:.
```

Les noms des répertoires sont séparés par "**`:`**". Dans cet exemple, le répertoire **`/udd/API/paquetagesMaison`** contient les paquetages "maison" pour les travaux pratiques de ce cours et le symbole "**`.`**" signifie le répertoire courant.

L'endroit où se trouvent les classes à utiliser peut également être indiqué au moment de la compilation ou de l'exécution par une option ***-classpath*** :

`javac -classpath /udd/API/paquetagesMaison:. Prog.java` pour la compilation,

`java -classpath /udd/API/paquetagesMaison:. Prog` pour l'exécution.

QCM 4.1

A propos du choix des identificateurs, on peut affirmer :

❑ 1 - Il faut utiliser les noms les plus courts possibles car cela réduit la taille des programmes et les rend plus lisibles.

❑ 2 - Dans la norme usuelle, les noms de classe commencent par une majuscule et les noms des données et des fonctions commencent par une minuscule.

❑ 3 - Il faut utiliser les noms les plus signifiants possibles, même si cela conduit à des noms longs, car cela remplace avantageusement de nombreux commentaires et rend les programmes plus lisibles.

❑ 4 - On a le droit d'utiliser **`villeDeMilleHabitants`** comme identificateur.

❑ 5 - On a le droit d'utiliser **`ville De Mille Habitants`** comme identificateur.

❑ 6 - On a le droit d'utiliser **`villeDe1000habitants`** comme identificateur.

❑ 7 - On a le droit d'utiliser **`1000habitants`** comme identificateur.

❑ 8 - On a le droit d'utiliser **`classe`** comme identificateur.

❑ 9 - On a le droit d'utiliser **`class`** comme identificateur.

QCM 4.2

On considère le programme suivant :

```
class QCMLocalGlobal{

  static int x;

  static int plusK(int x, int k){
    int resul=x+k;
    return resul;
  }

  static void ajouteK(int k){
    int resul=x+k;
    x=resul;
  }

  public static void main(String[] z){
    x=56;
    int k=12;
    int troisPlusK = plusK(3,k);
    System.out.print("x="+x+" troisPlusK="+troisPlusK);
    ajouteK(10);
    System.out.println(" x="+x+" troisPlusK="+troisPlusK);
  }
}
```

❏ 1 - Dans le **main**, l'instruction **x=56;** donne la valeur 56 au paramètre **x** de la fonction **plusK**.

❏ 2 - Dans le **main**, l'instruction **x=56;** donne la valeur 56 à la variable globale **x**.

❏ 3 - Dans l'instruction **int resul=x+k;** de la procédure **ajouteK**, **x** désigne la variable locale **x** du **main**.

❏ 4 - Dans l'instruction **int resul=x+k;** de la procédure **ajouteK**, **x** désigne la variable globale **x**.

❏ 5 - L'identificateur **resul** désigne la même variable dans la fonction **plusK** et dans la procédure **ajouteK**.

❏ 6 - Ce programme affiche : **x=56 troisPlusK=15 x=66 troisPlusK=15**

❏ 7 - Ce programme affiche : **x=56 troisPlusK=68 x=13 troisPlusK=68**

QCM 4.3

❏ 1 - Une application est généralement constituée de plusieurs classes.

❏ 2 - Une application doit être constituée d'une seule classe.

❏ 3 - Il est bon de regrouper dans une même classe les choses concernant un même thème.

❏ 4 -la classe **Math** de la bibliothèque standard regroupe les fonctions et les constantes concernant les opérations numériques ("mathématiques").

❏ 5 - Il est impossible d'utiliser une fonction d'une classe ***A*** depuis une autre classe ***B***.

❏ 6 - Pour utiliser une chose statique appelée ***f*** définie dans une classe ***A*** depuis une autre classe ***B***, il faut la désigner par ***A*** **(** ***f*** **)** .

❏ 7 - Pour utiliser une chose statique appelée ***f*** définie dans dune classe ***A*** depuis une autre classe ***B***, il faut la désigner par ***A*** **.** ***f***.

QCM 4.4

❏ 1 - Un paquetage permet de regrouper des classes concernant un même domaine d'intérêt.

❏ 2 - L'usage de paquetages n'apporte rien quand à la modularité.

❏ 3 - L'usage de paquetages permet un niveau de modularité plus vaste que les classes.

❏ 4 - Les paquetages permettent de structurer les bibliothèques d'intérêt général.

❏ 5 - La variable d'environnement CLASSPATH indique (au compilateur ou à l'exécuteur de programmes java) dans quels répertoires se trouvent les classes et les paquetages à utiliser.

❏ 6 - L'option ***-classpath*** des commandes **javac** et **java** indique (au compilateur ou à l'exécuteur de programmes java) dans quels répertoires se trouvent les classes et les paquetages à utiliser.

❏ 7 - Pour utiliser les classes d'un paquetage **ppp**, on place généralement en début du texte source une directive **import ppp.*;**.

❏ 8 - Pour utiliser les classes d'un paquetage **ppp**, il est nécessaire de placer en début du texte source une directive **import ppp.*;**.

❏ 9 - Pour utiliser une classe **A** d'un paquetage **ppp**, on peut la désigner par **ppp.A**.

Exercice 4.1 Portées d'indentifications

objectif : *comprendre la portée des identifications*

Indiquer ce qu'affiche le programme suivant :

```
class TestPorteesDIdentification {

  static String texte = "alfred";

  static void imprimeTexte(){
    String texte="jules";
    System.out.println(texte);
    texte="dupont";
    System.out.println(texte);
  }

  public static void main(String[]arg){
    System.out.println(texte);
    imprimeTexte();
    System.out.println(texte);
  }
}
```

Exercice 4.2 Variables globales, variables locales

objectif : *comprendre les effets sur les variables globales*

On considère le programme suivant :

```
class TestLocalGlobal {

  static int i = 10; ← ligne A
  static int a = 4, b = 8;

  static int diff(int x, int y){
    int i; ← ligne B
    i = x-y;
    return i;
  }

  static int somDiff(int x, int y){
    i = x+y;
    return i + diff(x, y);
  }

  public static void main(String[]arg){
    int res;
    res=diff(a,b);System.out.println("i="+i);
    res=diff(i,a);System.out.println("i="+i);
    res=somDiff(a,b);System.out.println("i="+i);
    res = diff(b, b);System.out.println("i="+i);
  }
}
```

- Quelles sont les variables globales et les variables locales ?
- Quelles sont les valeurs affichées ?
- Que se passe-t-il si on enlève la ligne *B* ?
- Que se passe-t-il si on enlève la ligne *A* ?

Exercice 4.3 Usage de plusieurs classes

objectif : *savoir rédiger un programme composé de plusieurs classes*

On désire calculer le périmètre et l'aire de cercles et de rectangles. Pour raison de modularité, on décide de rédiger plusieurs classes :

- une classe **`Cercle`**, spécialisée dans les calculs concernant les cercles,
- une classe **`Rectangle`**, spécialisée dans les calculs concernant les rectangles.
- une classe **`TestModularite`** contenant la procédure principale qui teste les deux classes précédentes pour un cercle de rayon 12 et pour un rectangle de côtés 12 et 15.

Réaliser et tester cet ensemble de classes.

Remarque : on peut ici rédiger les sources des trois classes dans le même fichier, par exemple **`TestModularite.java`**. Il est cependant préférable (c'est obligatoire dans le cas général) de les placer dans des fichiers séparés : **`Cercle.java`** et **`Rectangle.java`**.

CHAPITRE 5 Instructions conditionnelles

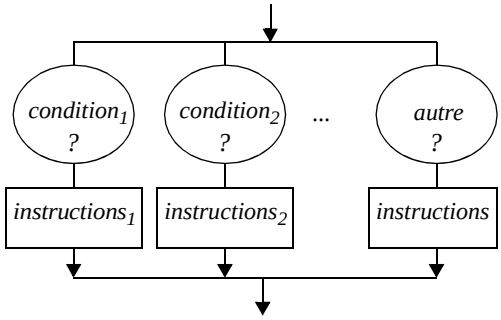
objectif : *savoir utiliser les instructions conditionnelles.*

5.1 Forme générale de l'instruction conditionnelle

objectif : *connaître la forme usuelle des instructions conditionnelles*

Les instructions conditionnelles permettent de choisir les instructions à exécuter en fonction de certains critères. La forme la plus répandue permet de choisir d'exécuter une action parmi un ensemble d'actions en fonction de critères indiqués par des expressions booléennes. Cela peut se représenter schématiquement par :

si *condition*$_1$ exécuter *instructions*$_1$
sinon si *condition*$_2$ exécuter *instructions*$_2$
sinon si *condition*$_3$ exécuter *instructions*$_3$
...
sinon exécuter *instructions*



Ici, ***condition*$_1$**, ***condition*$_2$**... sont des expressions à résultat booléen. Le lot d'instructions correspondant à la première de ces expressions dont le résultat est vrai est exécuté. Si toutes les expressions sont fausses, les instructions du dernier "sinon" sont exécutées.

En Java cela s'écrit :

```
if (condition1) { instructions1 }
else if (condition2) { instructions2 }
else if (condition3) { instructions3 }
...
else { instructions }
```

En Java, on n'exige pas que les conditions soient exclusives : une condition n'est testée que si les précédentes sont fausses. Le dernier cas, introduit par un simple "**`else`**", correspond au cas où aucune condition n'est vraie. Il est optionnel : si on ne le met pas, rien n'est exécuté si aucune condition n'est vraie. Les instructions de chacune des branches de la construction conditionnelle proposée sont encadrées d'accolades "**`{...}`**" et constituent donc un bloc. Ce n'est vraiment obligatoire que s'il y a une séquence de plusieurs instructions dans la branche, mais il vaut mieux mettre systématiquement les accolades, même pour une seule instruction, de façon à ne pas changer de forme si des modifications de programme obligent à rajouter des instructions.

Exemple 1 : fonction ***signe***

$$signe(n) = \begin{cases} -1 & \text{si } n<0 \\ 0 & \text{si } n=0 \\ +1 & \text{si } n>0 \end{cases}$$

Cette fonction peut s'écrire :

```
static int signe(int n) {
// résultat : -1, 0 ou +1 selon que n est <0, =0 ou >0
  int resul;
  if (n<0) { resul = -1; }
  else if (n==0) { resul = 0; }
  else /* n>0 */ { resul = 1; }
  return resul;
}
```

Toutes les conditions sont ici mutuellement exclusives. La dernière condition, **`n>0`**, est nécessairement vraie si les précédentes sont fausses. Il est recommandé d'indiquer la condition qu'on sait être vraie en commentaire, **`/* n>0 */`**, cela rend le programme plus facile à comprendre et à maintenir.

Remarque :

Java ne permettrait pas d'écrire la dernière branche de la conditionnelle avec un **`else if`**

```
else if (n>0) {resul=1;}
return resul;
```

car le compilateur n'admet pas qu'une variable puisse ne pas être initialisée. Certes, la condition **`n>0`** est certainement vraie si les précédentes sont fausses, mais le compilateur ne le sait pas (cette propriété fait partie de la sémantique du programme, et le compilateur ne sait appliquer que des règles de syntaxe). Effectivement, si aucune condition n'était vraie, la variable **`resul`** ne serait pas initialisée.

On peut se passer de la variable locale **`resul`** si on accepte les instructions **`return`** depuis le sein de la conditionnelle :

```
static int signe(int n) {
// résultat : -1, 0 ou +1 selon que n est <0, =0 ou >0
  if (n<0) { return -1; }
  else if (n==0) { return 0; }
  else /* n>0 */ { return 1; }
}
```

Ici encore, Java ne permettrait pas d'écrire la dernière branche de la conditionnelle

```
else if (n>0) { return 1; }
```

car le compilateur n'admet pas qu'une fonction qui doit rendre un résultat puisse se terminer sans instruction **`return`**.

Exemple 2 : fonction ***max***

$$max(a,b) = \begin{cases} a & \text{si } a \geq b \\ b & \text{si } a < b \end{cases}$$

En Java cela donne :

```
static int max(int a, int b) {
// résultat : le maximum de a et b
   if (a>=b) { return a; }
   else /* a<b */ { return b; }
}
```

Exemple 3 :

La fonction ***decodeMois*** rend en résultat le nom du mois de l'année dont le numéro est passé en paramètre. Le numéro du mois doit être un entier compris entre *1* et *12*. Le nom du mois est une chaîne de caractères de type **`String`**.

```
static String decodeMois(int numeroDeMois) {
// prérequis : numeroDeMois>=1 et numeroDeMois<=12
// résultat : le nom "en clair" du mois de numéro numeroDeMois
          if (numeroDeMois==1) { return "janvier";}
   else   if (numeroDeMois==2) { return "février";}
   else   if (numeroDeMois==3) { return "mars";}
   else   if (numeroDeMois==4) { return "avril";}
   else   if (numeroDeMois==5) { return "mai";}
   else   if (numeroDeMois==6) { return "juin";}
   else   if (numeroDeMois==7) { return "juillet";}
   else   if (numeroDeMois==8) { return "août";}
   else   if (numeroDeMois==9) { return "septembre";}
   else   if (numeroDeMois==10) { return "octobre";}
   else   if (numeroDeMois==11) { return "novembre";}
   else   /* numeroDeMois==12 */{ return "décembre";}
}
```

Remarque sur la notion de ***prérequis*** :

Dans la spécification de cette fonction, nous avons ***décidé*** de mettre en prérequis la nécessité que le numéro passé en paramètre soit un numéro de mois acceptable (compris entre 1 et 12). Ceci est clairement indiqué dans le commentaire de spécification :

```
// prérequis : numeroDeMois>=1 et numeroDeMois<=12
```

Un tel prérequis ***n'a pas à être vérifié*** par le programme de la fonction, mais ***doit être respecté*** par celui qui l'appelle. En d'autres termes : "tant pis pour l'utilisateur de la fonction s'il viole le prérequis, le résultat a le droit d'être n'importe quoi". Ici le résultat sera **`"décembre"`** dans ce cas, mais c'est un n'importe quoi comme un autre, et cela ne fait pas partie de la spécification de la fonction.

Instruction “if sans else” :

Dans un langage impératif, les branches d’une instruction conditionnelle sont des instructions, qui “font des choses”. Parfois il faut exécuter certaines instructions si une condition est vraie et ne ***rien faire*** si elle est fausse. Pour cela on peut utiliser une conditionnelle ***sans rubrique*** **`else`** :

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

Cette forme est strictement équivalente à :

```
if (condition) { instructions } else {/* rien */}
```

Exemple :

Voici un extrait de programme qui imprime la taille d’une personne et signale qu’elle est très grande si sa taille dépasse 2 mètres :

```
...
if (taillePersonne>2) {
  System.out.println("elle est très grande");
}
...
```

Remarque : cette forme de conditionnelle à une seule branche est typique des langages impératifs. On ne la trouve pas dans les langages fonctionnels, car dans ces langages la conditionnelle est une expression : elle signifie une “valeur”, et elle doit valoir quelque chose dans tous les cas.

5.2 Imbrication de conditionnelles

objectif : *savoir choisir entre des conditionnelles imbriquées ou une conditionnelle étalée*

La construction conditionnelle est elle-même une instruction. On peut donc utiliser une conditionnelle au sein d’une conditionnelle.

Considérons par exemple la fonction qui étant donné un numéro de mois donne le nombre de jours de ce mois. Pour simplifier, on ne s’intéresse pas aux années bissextiles. Le nombre de jours est donc :

- pour les mois de numéro inférieur à 8 (janvier... juillet), 30 jours pour les mois pairs, sauf 28 jours pour février, et 31 jours pour les mois impairs,
- pour les mois de numéro supérieur à 8 (août... décembre), 31 jours pour les mois pairs et 30 jours pour les mois impairs.

La fonction **`nombreDeJoursDuMois`** proposée ici réalise ce calcul en utilisant des conditionnelles imbriquées :

```
static int nombreDeJoursDuMois(int numeroDeMois) {
// résultat : nombre de jours du mois de numéro numeroDeMois
  if (numeroDeMois<8) { // cas janvier... juillet
    if (numeroDeMois%2 == 0) {
      if (numeroDeMois==2) { // cas spécial février
           return 28;
      }
      else { return 30; }
    }
    else /* numeroDeMois impair */ { return 31; }
  }
  else /* numeroDeMois>8 */ { // cas août... décembre
    if (numeroDeMois%2==0) { return 31; }
    else /* numeroDeMois impair */ { return 30; }
  }
}
```

Conditionnelle étalée :

On notera que l'utilisation de conditionnelles imbriquées ne facilite pas la lecture des algorithmes. Leur remplacement, quand c'est possible, par une conditionnelle étalée, quitte à complexifier les conditions, est souvent plus lisible. L'exemple précédent peut se réécrire sous la forme :

```
static int nombreDeJoursDuMois(int numeroDeMois) {
// résultat : nombre de jours du mois de numéro numeroDeMois
  if (numeroDeMois==2) { // cas particulier février
      return 28;
  }
  else if (numeroDeMois<8 & numeroDeMois%2==0) {
      // cas janvier... juillet, mois pair
      return 30;
  }
  else if (numeroDeMois<8 & numeroDeMois%2==1) {
      // cas janvier... juillet, mois impair
      return 31;
  }
  else if (numeroDeMois>=8 & numeroDeMois%2==0) {
      // cas août... décembre, mois pair
      return 31;
  }
  else /* (numeroDeMois>=8 & numeroDeMois%2==1) */ {
      //  cas août... décembre, mois impair
      return 30;
  }
}
```

5.3 Aiguillage

La plupart des langages de programmation offrent une forme particulière de conditionnelle, que l'on peut appeler ***aiguillage***, qui permet d'exécuter un bloc d'instructions choisi parmi plusieurs selon la valeur d'une expression de type scalaire "discret" : entier, caractère, type énuméré. La forme générale d'une telle construction est :

aiguillage selon la valeur de *expression*
soit *valeur*$_1$: exécuter *instructions*$_1$
soit *valeur*$_2$: exécuter *instructions*$_2$
soit *valeur*$_3$: exécuter *instructions*$_3$
...
autres cas : exécuter *instructions*

Cette construction n'est pas très générale car le critère du choix ne peut être que la valeur d'une expression scalaire et on n'a droit qu'à des constantes pour *valeur*$_1$, *valeur*$_2$... On peut se passer de cette construction car elle est strictement équivalente à :

v=expression
aiguillage selon la valeur de *v*
si *v=valeur*$_1$ exécuter *instructions*$_1$
sinon si *v=valeur*$_2$ exécuter *instructions*$_2$
sinon si *v=valeur*$_3$ exécuter *instructions*$_3$
...
sinon exécuter *instructions*

L'aiguillage est utilisé essentiellement pour des raisons de performance : le compilateur le traduit en une instruction de la machine qui, selon la valeur de l'expression, "saute" directement, en un coup, à la branche à exécuter.

En Java la forme de cette instruction est :

```
switch (expression) {
case valeur1: instructions1 break;
case valeur2 : instructions2 break;
case valeur3 : instructions3 break;
...
default  : instructions
}
```

Cette instruction n'est pas très heureuse à cause des "**`break;`**" qu'il faut placer en fin de chacune des branches. Si on ne met pas le "**`break;`**", au lieu de poursuivre l'exécution à la suite de l'aiguillage, la branche suivante est exécutée. Ceci est un héritage de mauvais goût du langage C.

À titre d'exemple, voici un programme qui reprend les fonctions précédentes, **`decodeMois`** et **`nombreDeJoursDuMois`** en utilisant l'aiguillage. La procédure principale de ce programme lit au clavier un numéro de mois et affiche à l'écran le nombre de jours du mois correspondant. On notera sur cet exemple que l'on peut regrouper les cas qui donnent lieu à un même traitement.

```
class TestNombreDeJoursDuMois {

   static int nombreDeJoursDuMois(int numeroDeMois) {
   // résultat : nombre de jours du mois de numéro numeroDeMois
   int resul;
      switch (numeroDeMois) {
      // janvier, mars, mai, juillet, août, octobre, décembre
      case 1 : case 3 : case 5 :
      case 7 : case 8 : case 10 : case 12 : resul=31; break;
      // avril, juin, septembre, novembre
      case 4 : case 6 : case 9 : case 11 : resul=30; break;
      // février
      case 2 : resul=28; break;
      default : resul=0;
      }
      return resul;
   }

   static String decodeMois(int numeroDeMois) {
   // résultat : le nom "en clair" du mois de numéro numeroDeMois
   // "mois inconnu" si numeroDeMois n'est pas compris
   // entre 1 et 12
   String resul;
      switch(numeroDeMois) {
      case  1 : resul= "janvier"; break;
      case  2 : resul= "février"; break;
      case  3 : resul= "mars"; break;
      case  4 : resul= "avril"; break;
      case  5 : resul= "mai"; break;
      case  6 : resul= "juin"; break;
      case  7 : resul= "juillet"; break;
      case  8 : resul= "août"; break;
      case  9 : resul= "septembre"; break;
      case 10 : resul= "octobre"; break;
      case 11 : resul= "novembre"; break;
      case 12 : resul= "décembre"; break;
      default : resul= "mois inconnu";
      }
      return resul;
   }

   public static void main(String[] arg) {
      System.out.print("numéroDeMois : ");
      int numMois=Lecture.unEntier();
      System.out.print("il y a ");
      System.out.print(nombreDeJoursDuMois(numMois));
      System.out.print(" jours en ");
      System.out.println(decodeMois(numMois));
   }
}
```

QCM 5.1

Considérons les fonctions :

```
static int choix(boolean cond,int k1,int k2){
// résultat : k1 si cond, k2 sinon
  if(cond){return k1;}
  else {return k2;}
}

static int f(int k1, int k2){
// résultat : ???
  return choix(k1>k2,k1,k2);
}

static int g(int k1){
// résultat : ???
  return choix(k1>=0,k1,-k1);
}

static int h(int k1, int k2){
// résultat : ???
  return choix(k1>k2,k1-k2,k2-k1);
}
```

- ❏ 1 - Le résultat de la fonction **`f(12,56)`** vaut **`44`**.
- ❏ 2 - Le résultat de la fonction **`f(56,12)`** vaut **`-44`**.
- ❏ 3 - Le résultat de la fonction **`f(12,56)`** vaut **`56`**.
- ❏ 4 - Le résultat de la fonction **`f`** est la somme de **`k1`** et **`k2`**.
- ❏ 5 - Le résultat de la fonction **`f`** est la différence de **`k1`** et **`k2`**.
- ❏ 6 - Le résultat de la fonction **`f`** est le maximum de **`k1`** et **`k2`**.
- ❏ 7 - Le résultat de la fonction **`f`** est la valeur absolue de la différence de **`k1`** et **`k2`**.
- ❏ 8 - Le résultat de la fonction **`g`** est **`k1`**.
- ❏ 9 - Le résultat de la fonction **`g`** est -1 si **`k1>=0`**, +1 sinon.
- ❏ 10 - Le résultat de la fonction **`g`** est la valeur absolue de **`k1`**.
- ❏ 11 - Le résultat de la fonction **`g`** est **`k1`** si **`k1>=0`**, **`-k1`** sinon.
- ❏ 12 - Le résultat de la fonction **`h(12,56)`** vaut **`44`**.
- ❏ 13 - Le résultat de la fonction **`h(56,12)`** vaut **`-44`**.
- ❏ 14 - Le résultat de la fonction **`h(12,56)`** vaut **`56`**.
- ❏ 15 - Le résultat de la fonction **`h`** est la somme de **`k1`** et **`k2`**.
- ❏ 16 - Le résultat de la fonction **`h`** est la différence de **`k1`** et **`k2`**.
- ❏ 17 - Le résultat de la fonction **`h`** est le maximum de **`k1`** et **`k2`**.
- ❏ 18 - Le résultat de la fonction **`h`** est la valeur absolue de la différence de **`k1`** et **`k2`**.

QCM 5.2

Considérons la fonction :

```
static String choix(int c, String s1, String s2, String s3){
// prérequis : ???
// résultat : ???
  if(c==1){return s1;}
  else if (c==2){return s2;}
  else /*c==3*/{return s3;}
}
```

❑ 1 - Il y a des erreurs de syntaxe.

❑ 2 - **choix(2,"alfred","toto","dupont")** vaut **"alfred"**.

❑ 3 - **choix(2,"alfred","toto","dupont")** provoque une erreur à l'exécution.

❑ 4 - **choix(2,"alfred","toto","dupont")** vaut **"toto"**.

❑ 5 - **choix(5,"alfred","toto","dupont")** provoque une erreur à l'exécution.

❑ 6 - **choix(5,"alfred","toto","dupont")** vaut **"dupont"**.

❑ 7 - Les commentaires de spécification :

```
// prérequis : c>=1 et c<=3
// résultat : s1, s2 ou s3 selon que c vaut 1, 2 ou 3
```

sont acceptables pour cette fonction.

❑ 8 - Les commentaires de spécification (sans prérequis) :

```
// résultat : s1, s2 ou s3 selon que c vaut 1, 2 ou 3
```

sont acceptables pour cette fonction.

❑ 9 - Les commentaires de spécification (sans prérequis) :

```
// résultat : s1 si c vaut 1, s2 si c vaut 2,
// s3 pour toute autre valeur de c
```

sont acceptables pour cette fonction.

❑ 10 - Les commentaires de spécification :

```
// prérequis : c>=1 et c<=3
// effet : affiche s1, s2 ou s3 selon que c vaut 1, 2 ou 3
```

sont acceptables pour cette fonction.

❑ 11 - La fonction **choix** pourrait être programmée en utilisant l'aiguillage suivant :

```
static String choix(int c, String s1, String s2, String s3){
  switch(c){
  case 1 : return s1;
  case 2 : return s2;
  default : return s3;
  }
}
```

❑ 12 - La fonction **choix** ne peut pas être programmée par l'aiguillage précédent car c'est une erreur de syntaxe.

❑ 13 - La fonction **choix** ne peut pas être programmée par l'aiguillage précédent car l'absence d'instructions **break** dans les branches rend le résultat incorrect.

QCM 5.3

Considérons la fonction :

```
static boolean comprisEntre(int k, int a, int b){
// prérequis : a<=b
// résultat : indique si a<=k<=b
  boolean resul;
  if(k<a){resul=false;}
  if(k>b){resul=false;}
  else {resul=true;}
  return resul;
}
```

❑ 1 - Il y a des erreurs de syntaxe.

❑ 2 - Cette fonction est incorrecte (le résultat ne correspond pas à la spécification).

❑ 3 - Cette fonction est correcte.

❑ 4 - **comprisEntre(2,14,56)** vaut **true**.

❑ 5 - **comprisEntre(2,14,56)** vaut **false**.

❑ 6 - **comprisEntre(62,14,56)** vaut **true**.

❑ 7 - **comprisEntre(62,14,56)** vaut **false**.

QCM 5.4

Considérons la fonction :

```
static int valeurDeCarte(String c){
// prérequis : c vaut "valet", "dame", "roi" ou "as"
// résultat : valeur de la carte de nom c
// (respectivement 1, 2, 3 et 10)
  switch(c){
  case "valet" : return 1;
  case "dame" : return 2;
  case "roi" : return 3;
  default : return 10;
  }
}
```

❑ 1 - Il y a des erreurs de syntaxe.

❑ 2 - Cette fonction peut provoquer une erreur lors de l'exécution.

❑ 3 - Cette fonction peut rendre un résultat incorrect (résultat qui ne correspond pas à la spécification).

❑ 4 - Cette fonction est correcte.

Exercice 5.1 Maximum de trois nombres

objectif : *savoir utiliser la conditionnelle.*

Rédiger et tester les fonctions suivantes :

1 - Fonction **`max`** qui calcule le plus grand de deux entiers donnés en paramètres.

2 - Fonction “maximum de trois nombres” qui calcule le plus grand de trois entiers donnés en paramètres. En réaliser deux versions :

- version1 : **`maxDeTroisNombresV1`**, en utilisant la fonction **`max`**,
- version2 : **`maxDeTroisNombresV2`**, directement, sans utiliser la fonction **`max`**.

Exercice 5.2 Médiane

Rédiger une fonction qui calcule la médiane de trois entiers distincts, c’est-à-dire celui qui est “encadré” par les deux autres.

Exemple : la médiane de 8, 1, 4 est 4.

Exercice 5.3 Equation du second degré

objectif : *savoir analyser tous les cas pertinents sur les données et maîtriser l’imbrication de conditionnelles.*

Rédiger un programme qui affiche la ou les solutions d’une équation du second degré :

$\boldsymbol{a}x^2+\boldsymbol{b}x+\boldsymbol{c}=0$

Attention : faire une analyse sérieuse. Les nombres ***a***,***b***,***c*** sont des nombres ***quelconques***. Ils peuvent notamment être nuls.

Conseil : rédiger une ***fonction*** qui rende en résultat une chaîne de caractères signifiant “en clair” le nombre et la nature des solutions.

Aide 5.1 Maximum de trois nombres

Pour **`maxDeTroisNombresV1`** :

utiliser le fait que ***maximum(a,b,c) = max(a,max(b,c))***

Pour **`maxDeTroisNombresV2`** :

on peut utiliser une conditionnelle étalée à trois branches dont les branches correspondent aux cas respectifs où le maximum est ***a***, ***b*** ou ***c***.

Aide 5.2 Médiane

Analyser le problème selon les trois résultats possibles (**`a`**, **`b`** ou **`c`**), ce qui conduit à une conditionnelle étalée avec les trois conditions qui expriment respectivement que **`a`**, **`b`** ou **`c`** est la médiane.

Aide 5.3 Equation du second degré

Nous avons choisi de réaliser une fonction solution qui reçoit en paramètres les trois coefficients **`a`**, **`b`** et **`c`** et rend en résultat une chaîne de caractère qui signifie en clair le nombre et la nature des solutions : **`static String solution(double a, double b, double c)`**

```
class TestEquationSecondDegre {

  static String solution(double a, double b, double c) {
  // résultat : les solutions en clair de l'équation ax2+bx+c=0
    ...
  }

  public static void main(String[] arg) {
    System.out.println("4X2 - 8X + 4 : "+solution(4,-8,4));
    ...
  }
}
```

L'analyse des divers cas est la suivante :

Pour a=0 : si $b \neq 0$, la solution est $-c/b$

si b=0 : si c=0, tout nombre est solution (quelque soit x, 0=0)

si $c \neq 0$, il n'y a aucune solution (quelque soit x, $c \neq 0$)

pour $c \neq 0$: la forme du résultat dépend du "déterminant" : $delta = b^2-4ac$

si $delta > 0$, il y a deux solutions qui sont des nombres réels :

$$\frac{-b+\sqrt{delta}}{2a} \quad \text{et} \quad \frac{-b-\sqrt{delta}}{2a}$$

si $delta$=0, il y a une solution "double" qui est un nombre réel : $-b/2a$

si $delta > 0$, il y a deux solutions qui sont des nombres complexes :

$$\frac{-b+i\sqrt{-delta}}{2a} \quad \text{et} \quad \frac{-b+i\sqrt{-delta}}{2a}$$

CHAPITRE 6 Récursivité et itération

6.1 Exprimer la répétition

objectif : *comprendre deux façons différentes de répéter des traitements :*
la ***récursivité*** *: exprime le résultat cherché comme solution d'équations,*
*l'****itération*** *: exprime le résultat cherché comme élément d'une suite de valeurs.*

La programmation atteint sa puissance en répétant un bloc d'instructions un nombre de fois non connu à l'avance. La répétition est en effet le seul moyen d'exprimer de façon finie des quantités de calculs qui dépendent des données traitées.

Il y a deux façons d'exprimer la répétition : la ***récursivité*** et ***l'itération***. Ces deux moyens sont très différents, sauf dans quelques cas particuliers. Il est impossible de dire si l'un est mieux adapté que l'autre, ou si l'un est plus facile que l'autre. Cela dépend du problème à résoudre et également du goût et des habitudes de celui qui programme. Pour bien saisir la différence entre les deux techniques, nous prendrons un exemple très simple : le calcul de la somme des nombres entiers de 0 à n :

somme(n) = 0 + 1 + 2 +...+n

Nous faisons ici semblant d'ignorer que le résultat peut s'obtenir par la formule *n(n+1)/2*.

6.1.1 Méthode récursive

La méthode récursive est basée sur un ***système d'équations***. On cherche des équations qui expriment la fonction à calculer pour diverses formes de l'argument en terme de la même fonction appliquée à des arguments "plus simples". Pour notre exemple, l'encadré suivant suggère d'exprimer ***somme(n)*** au moyen de ***somme(n-1)*** :

$$somme(n) = \underbrace{0 + 1 + 2 + \ldots + n}_{somme(n-1)}$$

Cette formulation n'est valable que pour n>0. 0n se pose le problème pour n=0, et bien évidemment la réponse est 0. Ces cas particuliers jouent un rôle important : on les appelle les "cas de base". Ce sont les cas pour lesquels le calcul de la solution ne fait pas intervenir la fonction elle-même. Il est impératif d'avoir de tels cas, sinon les équations ne conduisent pas à un algorithme. Sans les cas de base, les équations "tournent en rond" et ne permettent pas de calculer la fonction en un nombre fini

d'étapes. Ici, nous avons donc les deux équations :

- ***somme(n) = n+ somme(n-1) pour n>0***
- ***somme(0) = 0***

Ces équations conduisent à la définition récursive suivante de la fonction ***somme*** :

```
static int somme(int n) {
// prérequis : n>=0
// résultat : la somme des entiers de 0 à n
  if (n>0) {return n+somme(n-1);}
  else /*n==0*/ {return 0;}
}
```

Remarque : les langages de programmation fonctionnels n'ont que ce moyen de calcul répétitif.

6.1.2 Méthode itérative

La méthode itérative est basée sur une ***suite récurrente***. On cherche une suite définie par récurrence (c'est-à-dire dont le terme de rang *k* s'obtient par une formule utilisant le terme de rang *k-1*).

Pour la fonction ***somme*** prise en exemple, on envisage la suite :

$$S_0 = 0,\quad S_1 = 0 + 1,\quad S_2 = 0 + 1 + 2,\ldots\quad S_n = 0 + 1 + 2 \ldots + n$$

qui, de toute évidence, peut être définie par la récurrence :

- $S_0 = 0,$
- $S_i = S_{i-1} + i \quad pour\ i>0$

On a alors : $somme(n) = S_n$

Le passage de la suite au programme se fait ainsi :

- on utilise une ***variable***, **`s`** ici, destinée à valoir ***successivement*** les valeurs de la suite,
- on initialise cette variable à la première valeur de la suite, ***0*** ici,
- on ***répète***, en Java au moyen d'une instruction **`while`**, une instruction d'affectation de cette variable qui fait passer sa valeur du terme de rang *i-1* au terme de rang *i*, et ce jusqu'à obtention du terme qui nous satisfait, le terme de rang *n* ici.

Pour savoir si on a obtenu le terme de rang *n*, il faut dans cet exemple une variable auxiliaire, **`i`**, qui vaut en permanence le rang du terme calculé dans **`s`**. La variable **`i`** est initialisée à *0* et incrémentée à chaque répétition de l'itération.

Cela conduit à la définition itérative suivante de la fonction ***somme*** :

```
static int somme(int n) {
// prérequis : n>=0
// résultat : la somme des entiers de 0 à n
 int s=0; int i=0;
 while(i<n) {s=s+i; i=i+1;}
 return s;
}
```

6.2 Récursivité

objectif : *savoir rédiger des fonctions simples de manière récursive.*

6.2.1 Exemples de définitions récursives de fonctions

Exemple 1 : fonction factorielle

La fonction ***factorielle*** s'applique à un entier strictement positif *n* et donne en résultat le nombre entier produit des entiers de *1* à *n* :

factorielle(n) = 1×2×...×n (on la note généralement ***n!***)

La factorielle satisfait aux équations :

$$factorielle(n) = \begin{cases} 1 & \text{si } n = 1 \\ n \times factorielle(n-1) & \text{si } n > 1 \end{cases}$$

Ceci conduit à la programmation récursive suivante :

```
static int factorielle(int n) {
// prérequis : n>0
// résultat : la factorielle de n
   if (n==1) {return 1;}
   else {return n*factorielle(n-1);}
}
```

Exemple 2 : fonction pgcd

La fonction ***pgcd*** rend comme résultat le plus grand commun diviseur de deux nombres entiers strictement positifs *a* et *b*. Le calcul récursif peut-être établi à partir des équations bien connues suivantes :

$$pgcd(a,b) = \begin{cases} a & \text{si } a=b \\ pgcd(a-b,b) & \text{si } a>b \\ pgcd(a,b-a) & \text{si } a<b \end{cases}$$

Ce qui conduit à la définition récursive :

```
static int pgcd(int a, int b) {
// prérequis : a>0 et b>0
// résultat : le pgcd de a et b
   if (a==b) {return a;}
   else if (a>b) {return pgcd(a-b,b);}
   else /* a<b */ {return pgcd(a,b-a);}
}
```

Ces exemples montrent que le passage des équations au programme récursif est systématique. Chaque équation donne lieu à une branche de conditionnelle :

- dont la condition teste l'appartenance des paramètres au domaine de validité de l'équation,
- dont le corps est une instruction **`return`** ayant pour expression le second membre de l'équation.

6.3 Terminaison d'un algorithme récursif

objectif : *savoir justifier qu'un algorithme récursif termine en un temps fini.*

Un calcul récursif est la traduction directe d'équations que l'on sait ***mathématiquement correctes***. Il s'ensuit que ***si la fonction calcule un résultat, il est nécessairement correct***. Encore faut-il que le résultat soit effectivement calculé. Pour cela il faut s'assurer que le processus d'évaluation se ***termine***, c'est-à-dire que la fonction ne va pas se rappeler indéfiniment.

Dans la fonction **`factorielle`**, les paramètres des appels successifs sont *n*, *n-1*, *n-2*... jusqu'à *1* pour lequel il n'y a pas d'appel récursif, le résultat de la fonction valant alors *1*.

La terminaison est le principal point, voire pratiquement le seul, à s'assurer pour garantir l'exactitude d'un calcul récursif. L'autre facteur, la validité des équations, est un problème de "mathématique", pas vraiment d'informatique. C'est d'ailleurs cette facilité de programmation et cette sécurité quant à la validité des résultats qui font l'attrait de la programmation récursive. La garantie de terminaison n'est cependant pas toujours un problème trivial. Ainsi le système suivant d'équations pour la factorielle n'est pas faux, mais il ne conduit pas à un algorithme récursif qui termine :

$$factorielle(n) = \begin{cases} 1 & \text{si } n = 1 \\ \dfrac{factorielle(n+1)}{n+1} & \text{si } n > 1 \end{cases}$$

Pour s'assurer de la terminaison, il faut considérer la ou les suites possibles d'appels récursifs et vérifier que cette suite atteint en un nombre fini de termes les cas pour lesquels il n'y a pas d'appel récursif, appelés "cas de base".

Un ***moyen systématique et formel*** de s'en assurer est de trouver une ***fonction de terminaison***. C'est une fonction (mathématique) qui a les propriétés suivantes :

- elle ***dépend des arguments*** de la fonction définie récursivement,
- son résultat est un ***entier positif ou nul***,
- son application aux arguments des appels récursifs a une valeur ***strictement inférieure*** à son application aux arguments de la fonction définie récursivement.

Dans le cas de **`factorielle`**, on peut considérer la fonction de terminaison :

fonctionDeTerminaison(n) = n

Elle satisfait bien aux critères ci-dessus :

- elle est positive (par prérequis sur l'argument de factorielle, ***n*** est positif),
- l'argument des appels récursifs, ***n-1*** ici, est strictement inférieur à ***n***.

Pour la fonction **`pgcd`**, trouver une fonction de terminaison est moins trivial. On peut prendre :

fonctionDeTerminaison(a,b) = a+b

- Cette fonction est bien positive, par prérequis sur les arguments ***a*** et ***b***.
- dans le cas ***a>b*** : l'appel récursif se fait avec ***a-b*** et ***b*** et ***a-b+b = a*** est bien strictement inférieur à ***a+b*** puisque ***b*** est >0.
- dans le cas ***a<b*** : l'appel récursif se fait avec ***a*** et ***b-a*** et ***a+b-a = b*** est bien strictement inférieur à ***a+b*** puisque ***a*** est >0.

L'algorithme termine donc.

La notion de fonction de terminaison sert à prouver la terminaison d'un calcul récursif. La conception d'un algorithme est souvent une activité intuitive, basée sur des analogies et des expériences acquises et concevoir un algorithme n'est pas la même chose que de le prouver. On fait rarement une preuve formelle des algorithmes, mais il est tout de même essentiel de savoir que cela existe, que tout programme correct ne marche pas par hasard. Il est satisfaisant pour l'esprit de savoir que, si on veut, on peut faire de telles preuves.

6.4 Généralités sur l'Itération

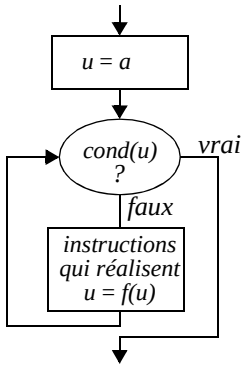
objectif : *savoir rédiger des fonctions simples de manière itérative.*

6.4.1 Forme générale d'une itération

Une itération calcule un résultat sous la forme d'un terme d'une suite récurrente u_0, u_1, u_2..., qui satisfait à une certaine condition ***cond***. Les termes successifs de la suite n'ont pas à être conservés. Il suffit d'utiliser ***une seule variable*** pour conserver la valeur de l'***élément courant de la suite***. À chaque étape, l'élément suivant est calculé et affecté à la variable.

Le schéma de calcul itératif est donc :

u = a
tant que *cond(u) est fausse* faire *u = f(u)*



En Java, la forme générale de l'itération appelée "boucle tant-que"[1] est :

```
while (non condition d'arrêt) { instructions }
```

L'exécution de cette boucle consiste à exécuter les instructions entre accolades, appelées ***corps de la boucle***, tant que la condition est vraie.

L'exemple précédent de calcul d'une suite récurrente u_i avec une formule de récurrence exprimable par une fonction `f` s'écrit donc :

```
u = a;
while (!cond(u)) {u=f(u);}
```

1. "while loop" en anglais.

6.4.2 Premier exemple de calcul itératif : racine carrée

Pour calculer la racine carrée d'un nombre réel ***a*** on peut utiliser la relation de récurrence suivante (formule d'Héron d'Alexandrie) :

$$u_0 = 1$$
$$u_i = (u_{i-1} + a / u_{i-1}) / 2 \text{ pour } i > 0$$

La valeur cherchée est la limite de u_i pour **i** tendant vers l'infini. En informatique, on n'aime pas beaucoup l'infini. On se contentera d'un résultat ayant une précision suffisante, disons à 10^{-3} près. On cherche donc le premier terme de la suite qui satisfait à la condition :

$$| u_i^2 - a | < 0.001$$

On a maintenant tous les éléments pour construire le programme itératif de la fonction **`racineCarree`** :

- une variable **`u`**, de type **`double`**, sera utilisée, initialisée à ***1***, le premier terme de la suite,
- la condition d'arrêt de l'itération sera $| u^2 - a | < 0.001$,
- le corps de l'itération réalisera **`(u + a/u)/2`**..

Ce qui donne :

```
static double racineCarree(double a) {
// prérequis : a>=0
// résultat : racine carrée de a à 0.001 près
  double u = a;
  while (!(Math.abs(u*u-a)<0.001)) {u = (u + a/u)/2;}
  return u;
}
```

6.4.3 Itération sur une composition de plusieurs variables

objectif : *savoir raisonner sur la suite de valeurs de plusieurs variables.*

Dans l'exemple précédent, l'itération calcule une suite de valeurs simples, u_i, chaque u_i étant un simple nombre réel. Dans le cas général, la suite récurrente concerne une ***composition de valeurs*** :

$$<u, v, w...>_i \text{ que l'on peut également noter } <u_i, v_i, w_i...>$$

Les éléments de la suite et donc leurs composants sont fonction de l'élément précédent :

$$<u_i, v_i, w_i...> = F(<u_{i-1}, v_{i-1}, w_{i-1}...>)$$

soit encore :

$$u_i = f(u_{i-1}, v_{i-1}, w_{i-1}...)$$
$$v_i = g(u_{i-1}, v_{i-1}, w_{i-1}...)$$
$$w_i = h(u_{i-1}, v_{i-1}, w_{i-1}...)$$
$$...$$

L'itération qui calcule les éléments de la suite est construite comme précédemment. La seule petite complication provient de la multiplicité des composants des éléments de la suite. Il faut ***une variable par composant***. Ces variables sont appelées ***variables d'état de la boucle***. La forme de l'itération doit être :

u = a, v = b, w = c
tant que ***cond(u,v,w)*** **est fausse faire** $<u, v, w> = <f(u, v, w), g(u, v, w), h(u, v, w)>$

6.4.3.1 Exemple : fonction factorielle

A partir de ***factorielle(1)=1*** et, pour ***i***>1, ***factorielle(i) = i×factorielle(i-1)***, nous constatons que l'on peut calculer ***factorielle(n)*** au moyen de la suite :

$fac_1 = 1$, $fac_2 = 2 \times 1 = 2 \times fac_1$, $fac_3 = 3 \times 2 \times 1 = 3 \times fac_2$,... $fac_i = i \times fac_{i-1}$

Mais cette suite de valeurs n'est pas récurrente par elle même. Il manque le "***i***". Le "***i***" doit lui aussi être défini par récurrence. Nous sommes amenés à introduire une composante supplémentaire, que nous appellerons ***k***, dont la suite des valeurs sera les "***i***". Voici la suite des valeurs de ***k***, ainsi que celles de ***fac*** réécrites en faisant intervenir ***k*** :

i :	1	2	3	4	5	...	i
k_i :	**1**	**2**	**3**	**4**	**5**	...	$k_{i-1}+1$
fac_i :	**1**	2×1	3×2	4×6	5×24	...	$(k_{i-1}+1) \times fac_{i-1}$

Le résultat, ***factorielle(n)***, est la valeur de fac_i pour $k_i = n$.

Cette analyse conduit au programme suivant :

```
static int factorielle(int n){
// prérequis : n>0 et n pas trop grand
// résultat : factorielle de n
  int k=1; int fac=1;
  while(k<n){
    fac=(k+1)*fac;
    k=k+1;
  }
  return fac;
}
```

Important : développer des scénarios

Cet exemple montre l'intérêt de développer un ou plusieurs ***scénarios***, c'est-à-dire des exemples de de valeurs des suites récurrentes envisagées. C'est ce que nous avons fait pour les suites ***k*** et ***fac***. C'est une bonne méthode pour trouver les "bonnes" formules de récurrence. On peut facilement se tromper sinon.

6.4.3.2 Autre exemple de calculs itératifs : pgcd

Comme exemple d'itération portant sur une composition de valeurs, considérons le calcul du plus grand commun diviseur. À partir des équations :

$$pgcd(a,b) = \begin{cases} a & \text{si } a=b \\ pgcd(a-b,b) & \text{si } a>b \\ pgcd(a,b-a) & \text{si } a<b \end{cases}$$

on imagine assez facilement une suite de couples $\langle u_i, v_i \rangle$ dont les composants convergent vers ***pgcd(a,b)***. Il s'agit de la suite initialisée à $\langle a,b \rangle$ et dont l'élément suivant s'obtient en retranchant le plus petit composant du plus grand. Par exemple, pour ***a=21*** et ***b=28***, la suite $\langle u_i, v_i \rangle$ est :

$$\langle 21,28 \rangle, \langle 21,7 \rangle, \langle 14,7 \rangle, \langle 7,7 \rangle$$

Le ***pgcd*** est ***7***. L'élément qui nous intéresse dans cette suite se reconnaît au fait que $u_i = v_i$. La condi-

tion d'arrêt est donc ***u=v***.

L'itération correspondante a la forme suivante :

u* = *a*, *v* = *b
tant que *u*≠*v* faire <*u*, *v*> = <*u*-*v*, *v*> si *u*>*v* , <*v*, *v*-*u*> sinon

ou encore, puisqu'on constate ici qu'un seul composant ***u*** ou ***v*** est modifié à chaque étape :

u* = *a*, *v* = *b
tant que *u*≠*v* faire si *u*>*v* : u = *u*-*v*
sinon : *v* = *v*-*u*

On s'est ramené à cette seconde forme, car les langages de programmation usuels n'ont pas, c'est dommage, l'affectation "multiple" telle que "***u,v* = *expression***". On est obligé de le décomposer en "***u* = *expression*$_1$ et *v* = *expression*$_2$**".

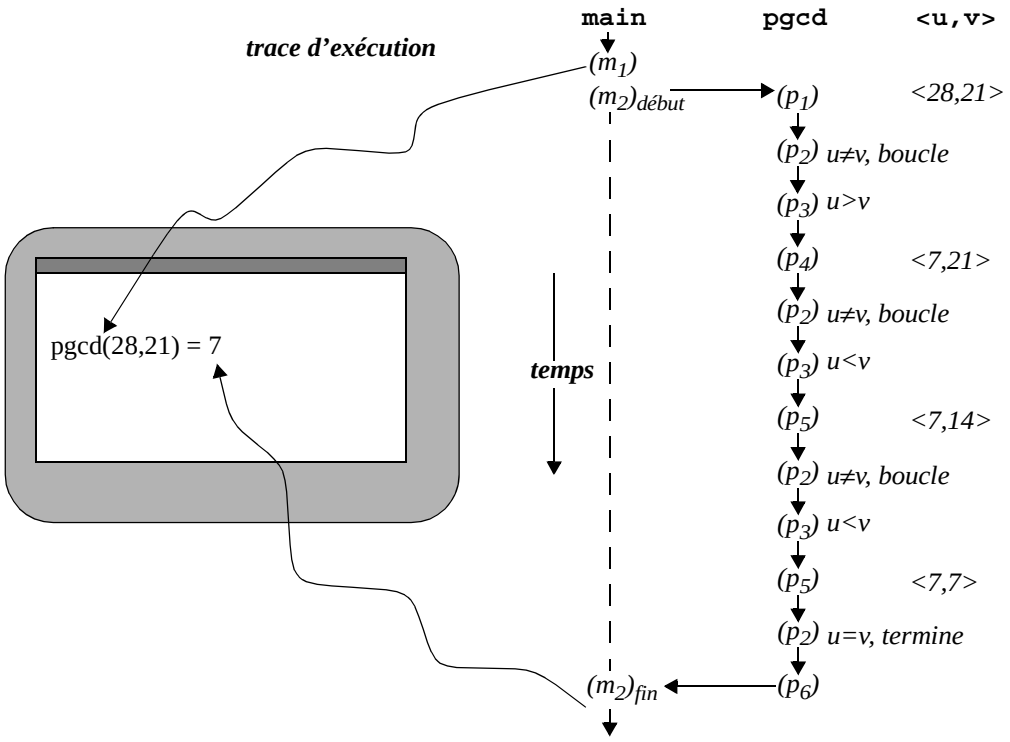
La fonction **`pgcd`** s'écrit alors :

```
static int pgcd(int a, int b) {
// prérequis : a>0 et b>0
// résultat : le pgcd de a et b
   int u=a; int v=b;
   while (u!=v) {
      if (u>v) {u=u-v;}
      else /*u<v*/ {v=v-u;}
   }
   return u;
}
```

6.4.4 Trace d'exécution d'une itération

Pour illustrer ce qui se passe lors de l'exécution d'une itération, nous allons indiquer la séquence d'instructions exécutées au cours du temps, accompagnée des valeurs successives des variables, dans le cas du calcul de ***pgcd(21,28)***. Le programme complet est :

```
class TestPgcd {
   static int pgcd(int a, int b) {
      int u=a; int v=b;   (p1)
      while (u!=v) {      (p2), test
         if (u>v) {       (p3), test
            u=u-v; (p4)
         }
         else /*u<v*/ {
            v=v-u; (p5)
         }
      }
      return u;           (p6)
   }

   public static void main(String[] arg) {
      System.out.print("pgcd(28,21) = "); (m1)
      System.out.println(pgcd(28,21)); (m2)
   }
}
```



6.4.5 Petits ennuis dus à l'absence d'affectation multiple

objectif : *savoir contourner les problèmes dus à la séquentialité des affectations de plusieurs variables.*

Les affectations sont séquentielles et il faut en tenir compte pour affecter correctement plusieurs variables qui représentent les composants des éléments de la suite calculée par la boucle.

Pour réaliser : $\langle u,v\rangle_i = F(\langle u,v\rangle_{i-1})$, c'est à dire $u_i = f(u_{i-1},v_{i-1})$ **et** $v_i = g(u_{i-1},v_{i-1})$

on est amené à effectuer une affectation que l'on peut noter : *u, v* reçoivent *f(u,v), g(u,v)*

mais l'affectation multiple n'étant généralement pas offerte par les langages, on est tenté de la remplacer par plusieurs affectations simples, ce qui est généralement incorrect si on utilise les mêmes expressions :

```
...
u = f(u,v);
v = g(u,v);
...
```

incorrect, ne réalise pas "<u, v> reçoivent *f(u,v), g(u,v)"*

La séquence ci-dessus réalise en fait : ***u, v = f(u,v), g(f(u,v),v)*** puisque la première affectation à *u* est déjà faite au moment où on affecte *v.*

Un moyen systématique et clair de contourner ce petit problème est d'utiliser, quand c'est vraiment nécessaire, des variables intermédiaires pour désigner de façon temporaire le futur état des compo-

sants. Par exemple, si **u** et **v** sont des variables de type **double** :

```
...
double uFutur = f(u,v);
double vFutur = g(u,v);          réalise "<u, v> reçoivent f(u,v), g(u,v)"
u=uFutur; v=vFutur;
...
```

Comme exemple, on peut reprendre le calcul du ***pgcd***, à partir des équations :

$$pgcd(a,b) = \begin{cases} a & \text{si } a=b \\ pgcd(max(a,b) - min(a,b), min(a,b)) & \text{si } a \neq b \end{cases}$$

Ces équations sont strictement équivalentes aux précédentes. C'est une autre façon de dire, au moyen des fonctions ***min*** et ***max***, que l'on peut remplacer le plus grand des deux nombres par la différence du plus grand et du plus petit. L'itération correspondante est :

u* = *a*, *v* = *b
tant que *u*≠*v* faire <*u, v*> = <max(*u,v*)-*min(u,v), min(u,v)*>

Affectation multiple qui, selon le principe précédent, peut se traduire en :

```
static int pgcd(int a, int b) {
  int u=a; int v=b;
  while (u!=v) {
    int uFutur=max(u,v)-min(u,v); int vFutur=min(u,v);
    u=uFutur; v=vFutur;
  }
  return u;
}
```

ou encore, en introduisant une variable **minuv** pour éviter de calculer deux fois **min(u,v)** :

```
static int pgcd(int a, int b) {
  int u=a; int v=b;
  while (u!=v) {
    int minuv=min(u,v); int uFutur=max(u,v)-minuv;
    u=uFutur; v=minuv;
  }
  return u;
}
```

6.5 Notions d'invariant de boucle et de fonction de terminaison

objectif : *prouver qu'une itération termine en un temps fini et calcule bien ce qui est prévu.*

6.5.1 Construction d'une itération

Deux problèmes se posent au concepteur d'un algorithme itératif :

- d'une part ***inventer*** l'itération,
- d'autre part s'assurer de l'***exactitude*** de l'itération, c'est-à-dire être sûr qu'elle calcule le bon résultat dans tous les cas.

Pour l'invention, il n'y a pas de "voie royale". C'est souvent l'expérience acquise et l'analogie avec

des problèmes que l'on a déjà résolus qui conduisent à inventer l'itération. La méthode consiste à chercher une suite récurrente qui conduise à la solution, et ceci se fait par essai-erreur, en exhibant des exemples de telles suites.

Pour l'exactitude, il existe deux notions importantes qui sont l'***invariant de boucle*** et la ***fonction de terminaison***. Ces notions seront vues aux paragraphes suivants.

Nous allons illustrer la construction d'une itération sur un exemple simple. Soit à réaliser une fonction ***nbChiffres*** égale au nombre de chiffres significatifs de l'écriture décimale d'un entier strictement positif.

Par exemple : ***nbChiffres(***1998***)*** = 4

Comment peut s'obtenir ce résultat 4 ?

On peut considérer que un chiffre vient du "8" et trois chiffres viennent de "199". Une piste :

nbChiffres(1998***)*** = ***nbChiffres(***199***)*** + 1 = 3+1 = 4

On peut généraliser ceci en :

pour $n < 10$, ***nbChiffres(n)*** = 1

pour $n \geq 10$, ***nbChiffres(n)*** = ***NbChiffres(n***/10***)***+1

Remarque :

On aurait pu partir sur une "mauvaise idée" en considérant, ce qui n'est pas faux, que le résultat 4 s'obtient en prenant 1 chiffre de poids fort, le "1", et 3 chiffres de poids faibles "998". On s'aperçoit que c'est une "mauvaise idée" car on n'arrive pas à exprimer simplement comment extraire ce "1" et ce "998" de "1998" avec les opérations dont on dispose.

À ce stade nous avons des équations dont on pourrait tirer un ***algorithme récursif*** évident. Mais pour un ***algorithme itératif***, il faut trouver une suite récurrente qui conduise au résultat.

Les équations nous suggèrent de décomposer le nombre en son chiffre de poids faible (8 dans l'exemple) et le reste constitué des chiffres à voir (199 ici). Chaque chiffre extrait doit être comptabilisé pour fournir le résultat cherché. Ces réflexions nous conduisent à considérer la suite de paires de nombres

$$<resteAVoir_i, compteur_i>$$

où $resteAVoir_i$ est le nombre formé des chiffres non encore comptés et $compteur_i$ est le nombre de chiffres déjà comptés. Voici le scénario pour ***n***=1998 :

$resteAVoir_i$:	1998	199	19	1	0
$compteur_i$:	0	1	2	3	4

$compteur_i$ vaut le résultat cherché quand $resteAVoir_i$ vaut 0

Il ne faut pas hésiter à développer un ou plusieurs exemples de suites comme ci-dessus. Ceci permet de s'assurer de la validité de l'idée et surtout de fixer les valeurs initiales de la suite, les formules de récurrence et la condition d'arrêt. Ici on a :

- **valeurs initiales :** $resteAVoir_0 = n$, $compteur_0 = 0$
- **formules de récurrence :** $resteAVoir_{i+1} = resteAVoir_i/10$, $compteur_{i+1} = compteur_i + 1$
- **condition d'arrêt :** $resteAVoir_i = 0$

L'algorithme itératif s'en déduit aussitôt :

- Il comporte deux variables, **resteAVoir** et **compteur**, initialisées respectivement à **n** et *0*,
- les formules de récurrence donnent les affectations du corps de la boucle,
- la condition d'arrêt est **resteAVoir==0**.

```
static int nbChiffres(int n) {
// prérequis : n>0
// résultat : nombre de chiffres de l'écriture décimale de n
  int compteur=0; int resteAVoir=n;
  while (resteAVoir>0) {
    compteur=compteur+1; resteAVoir=resteAVoir/10;
  }
  return compteur;
}
```

6.5.2 Invariant de boucle

Maintenant que nous avons découvert un algorithme itératif, comment s'assurer formellement qu'il est correct ? la méthode consiste à trouver un ***invariant de boucle***. Un invariant de boucle est une propriété (un prédicat) fonction des variables de la boucle et qui doit :

- ***être vrai avant la boucle***, après les initialisations,
- ***être maintenu vrai par les instructions du corps de boucle*** (la condition d'arrêt n'étant pas vérifiée),
- et enfin, pour être "intéressant", il faut que l'invariant de boucle ***accompagné de la condition d'arrêt assurent logiquement que le résultat est correct***.

Pour l'itération de la fonction **nbChiffres**, l'invariant de boucle est :

compteur + *nbChiffres*(resteAVoir) = *nbChiffres*(n)

- Cette propriété est vraie après les initialisations puisque après **compteur=0** et **resteAVoir=n** on a bien évidemment ***0*** **+** ***nbChiffres*****(n) = *nbChiffres*(n)**
- elle reste vraie à chaque exécution du corps de boucle puisque, pour **resteAVoir**>0 les mathématiques nous assurent que (propriété bien connue...) :

  nbChiffres**(resteAVoir) = *nbChiffres*(resteAVoir/10)+1**

  et donc si on tient compte de l'effet de l'affectation **compteur=compteur+1**, le bilan est de conserver la valeur de la somme **compteur + *nbChiffres*(resteAVoir)**

Lorsque la condition d'arrêt devient vraie, le résultat est atteint. En effet :

si **resteAVoir**=0, ***nbChiffres*****(resteAvoir)** =0, et donc **compteur**=***nbChiffres*****(n)**

Analogie du citron pressé

Toute itération ressemble à la fabrication d'un jus de citron.

Les données de l'algorithme sont le citron entier, le résultat attendu est le jus du citron. Pour faire un citron pressé :

Il faut un citron et un récipient destiné à recueillir le jus : ce sont les variables de la boucle.

Il faut aussi un presse citron qui à chaque tour de main extrait un peu du jus du citron : c'est le corps de la boucle.

On s'arrête de presser le citron quand il est devenu sec : c'est la condition d'arrêt.

Dans cette analogie l'invariant est la "conservation du jus" : à tout moment la somme du jus dans le récipient et du jus qui reste dans le citron est égale au jus initialement contenu dans le citron.

Dans la fonction **`nbChiffres`**, la variable **`resteAVoir`** correspond au citron et la variable **`compteur`** correspond au récipient qui recueille le jus. La condition d'arrêt "citron sec" est traduite par **`resteAVoir`**==0. Le pressage du citron correspond aux affectations du corps de boucle.

Les affectations du corps de boucle :

Pour pouvoir justifier que le corps de boucle respecte l'invariant, il faut savoir comment tenir compte des affectations de variables. Dans les cas simples cela peut être considéré comme intuitif et évident. Cependant voici comment s'y prendre plus formellement :

pour une affectation **`x =`** ***expression*****`;`**

il suffit de montrer que l'invariant reste vrai si on ***substitue*** dans son texte les occurrences de **`x`** par "***expression***", avec comme hypothèses :

- que l'invariant est vrai,
- que la condition de sortie est fausse,
- que les conditions des branches de conditionnelles pour lesquelles ces affectations ont lieu sont vraies.

Dans l'exemple précédent, les affectations sont :

```
compteur=compteur+1; resteAVoir=resteAVoir/10;
```

Il faut donc substituer dans l'invariant :

`compteur` par **`compteur+1`** et **`resteAVoir`** par **`resteAVoir/10`**

"**`compteur+1 +`** ***nbChiffres*****`(resteAVoir/10) =`** ***nbChiffres*** **`(n)`**"

est-ce vrai ? oui puisque les mathématiques nous disent que, sous réserve que **`resteAVoir`**>0, ***nbChiffres*****`(resteAVoir/10) =`** ***nbChiffres*****`(resteAVoir)-1`**

et donc l'invariant substitué redevient :

"**`compteur +`** ***nbChiffres*****`(resteAVoir) =`** ***nbChiffres*** **`(n)`**"

ce qui est vrai par hypothèse que l'invariant est vrai.

Ici nous avons du substituer ***deux*** variables (**`compteur`** et **`resteAVoir`**). Ceci n'est directement possible que si les affectations sont indépendantes (la nouvelle valeur de **`compteur`** ne dépend que

de **`compteur`**, et celle de **`resteAVoir`** ne dépend que de **`resteAVoir`**). Dans des cas d'affectations interdépendantes, il faut tenir compte de la séquentialité de ces affectations et corriger les expressions substituées en conséquence.

Ainsi, grâce à l'invariant mis en évidence, on vient de prouver que ***si la boucle termine***, alors le résultat est correct. Mais cela ne suffit pas. Encore faut-il s'assurer que la boucle termine.

6.5.3 Fonction de terminaison et notion de complexité

Pour s'assurer qu'une itération termine, il faut exhiber une ***fonction de terminaison***. C'est une notion voisine de celle rencontrée pour la programmation récursive :

- c'est une ***fonction des variables de la boucle***,
- son résultat est un ***entier positif ou nul***,
- elle ***décroît strictement*** à chaque exécution du corps de la boucle.

Dans l'exemple de **`nbChiffres`**, on peut choisir pour fonction de terminaison :

fonctionDeTerminaison = **`resteAVoir`**

en effet **`resteAVoir`** est un entier, toujours >=0, et il décroît strictement puisque divisé par 10 à chaque exécution du corps de boucle. Attention à la subtilité : **`resteAVoir`** décroît strictement car la condition de l'itération, **`resteAVoir>0`**, nous assure qu'il n'est pas nul (sinon la décroissance ne serait pas stricte, car 0/10 n'est pas strictement inférieur à 0). Il faut se méfier des "fausses fonctions de terminaison", qui ne décroissent pas toujours ***strictement*** : de tels programmes sont susceptibles de boucler indéfiniment.

Notion de complexité :

Il est évident que le nombre d'exécutions du corps de boucle est toujours inférieur à la fonction de terminaison appliquée à l'état initial. Ceci permet d'avoir une idée des ***performances en temps d'exécution*** d'un algorithme itératif. Si on veut approcher "au mieux" le temps d'exécution, il faut exhiber une fonction de terminaison aussi "petite" que possible.

Dans le cas de la fonction **`nbChiffres`** on peut prendre la fonction suivante :

autreFonctionDeTerminaison = ***nbChiffres(*****`resteAVoir`*****)***

C'est de toute évidence une fonction de terminaison, elle décroît de 1 à chaque itération car il est bien connu que **`n/10`** a un chiffre décimal de moins que **`n`** (pour **`n>0)`**.

Cette fonction de terminaison est bien meilleure que la précédente. Elle montre que le nombre d'exécution du corps de boucle sera inférieur ou égal à ***nbChiffres(n)***, c'est-à-dire à $\boldsymbol{log_{10}(n)+1}$.

On dit que la ***complexité*** (temporelle) de l'algorithme est ***O(log n)***.

Plus généralement, on dit que la complexité est ***O(f(n))*** si le temps d'exécution est inférieur à ***k.f(n)***, où ***k*** est une constante arbitraire qui abstrait notamment les facteurs technologiques.

6.5.4 Autre exemple d'invariant de boucle et de fonction de terminaison

Nous considérerons ici un exemple un peu plus complexe, le programme itératif de la fonction **pgcd**.

```
static int pgcd(int a, int b) {
  int u=a; int v=b;
  while (u!=v) {
    if (u>v) {u=u-v;}
    else /*u<v*/ {v=v-u;}
  }
  return u;
}
```

Invariant proposé : ***pgcd(u,v) = pgcd(a,b)***

En effet, les initialisations l'assurent trivialement, et le corps de boucle le conserve :

pour le cas **u>v :** l'affectation **u=u-v** nous conduit à substituer **u-v** à **u** dans l'invariant :

"***pgcd(u-v,v) = pgcd(a,b)***"

par la propriété (mathématique) du ***pgcd*** : pour **u>v**, ***pgcd(u-v,v) = pgcd(u,v)***

donc l'invariant substitué est équivalent à l'invariant initial, vrai par hypothèse.

Le raisonnement est analogue pour le cas **u<v**.

Pour fonction de terminaison on peut proposer **|u+v|**. C'est bien positif ou nul et cela décroît strictement à chaque itération.

6.6 Itérations sur des entrées de données

objectif : *savoir raisonner sur une séquence de données lues depuis l'extérieur.*

On utilise parfois les itérations pour effectuer un traitement sur une séquence de données introduites depuis l'extérieur au moment de l'exécution, notamment des données frappées au clavier par l'utilisateur du programme. Par exemple pour calculer la somme ou la moyenne d'une suite de nombres. Le programme lit un élément de la séquence à chaque étape de l'itération. Une convention doit être choisie pour déterminer la fin de la séquence. On peut choisir d'indiquer la fin par une valeur particulière, qui ne peut pas apparaître dans la séquence, appelée ***marque de fin***. Par exemple pour une séquence de nombres positifs, la marque de fin pourra être *-1*.

Le programme suivant calcule la somme d'une séquence d'entiers frappée au clavier :

```
class SommeSequenceEntree {

   static final int marqueFin=-1;

   public static void main(String[] arg) {
      int somme=0;
      int nombreLu=Lecture.unEntier();
      while (nombreLu!=marqueFin) {
         somme=somme+nombreLu;
         nombreLu=Lecture.unEntier();
      }
      System.out.print("somme="); System.out.println(somme);
   }
}
```

L'itération utilise deux variables d'état : **somme**, qui sert à cumuler les valeurs lues, et **nombreLu** qui reçoit à chaque étape un nouvel élément de la séquence.

Suite associée à la séquence d'entrée :

Le programme précédent est suffisamment simple pour avoir été conçu directement (avec un peu d'habitude). Cependant de tels programmes itératifs qui lisent des données peuvent être conçus en utilisant la méthodologie des suites récurrentes. Il suffit simplement d'introduire la suite des valeurs de la séquence d'entrée. C'est une suite qui bien évidemment n'est pas calculée par une formule de récurrence, elle est à considérer comme "donnée".

Pour le problème précédent du calcul de la somme, la séquence d'entrée doit être formalisée par une suite $nombreLu_0$, $nombreLu_1$, $nombreLu_2$... suite finie dont le dernier élément est *-1*. La récurrence qui permet de calculer la somme est :

$somme_0 = 0$
$somme_i = somme_{i-1} + nombreLu_{i-1}$ pour $i>0$ et tel que $nombreLu_j \neq -1$ pour tout $j<i$

Par exemple, pour la séquence d'entrée : *6 4 8 2 5 -1*,

la suite récurrente $<somme, nombreLu>_i$ est :

i :	0	1	2	3	4	5
$nombreLu_i$:	6	4	8	2	5	-1
$somme_i$:	0	6	10	18	20	25

On retrouve bien la même traduction en boucle que pour une suite récurrente entièrement définie par calcul, la seule différence est que le passage de $nombreLu_{i-1}$ à $nombreLu_i$ se fait au moyen de l'instruction de lecture :

```
nombreLu = Lecture.unEntier();
```

Autre exemple d'itération sur entrée de données :

Soit une séquence de nombres entiers positifs ou nuls introduite au clavier et terminée par la marque de fin "-1". On veut écrire un programme qui indique le nombre de zéros rencontrés dans cette séquence.

L'analyse consiste à chercher une relation de récurrence utilisant les éléments de la séquence donnée, appelons-les $nombreLu_i$, et une autre suite, appelons-la *nbZeros*, telle que $nbZeros_i$ contienne le nombre d'occurrences de 0 parmi $nombreLu_0 ... nombreLu_{i-1}$.

Exemple :

i :	0	1	2	3	4	5	6
$nombreLu_i$:	6	0	8	2	0	7	-1
$nbZeros_i$:	0	0	1	1	1	2	2

La relation de récurrence est :

$$nZeros_i = \begin{cases} nbZeros_{i-1} + 1 & \text{si } nombreLu_{i-1} = 0 \\ nbZeros_{i-1} & \text{si } nombreLu_{i-1} \neq 0 \end{cases}$$

Les variables de la boucle sont donc **nbZeros** et **nombreLu**. L'initialisation de **nbZeros** est 0 et celle de **nombreLu** est faite par une première lecture avant la boucle, comme c'est souvent le cas pour une boucle qui traite une séquence de données lues. La condition d'arrêt est bien évidemment **nombreLu==marqueFin**. Le programme peut s'écrire :

```
class NombreDeZeros {

   static final int marqueFin=-1;

   public static void main(String[] arg) {
      int nbZeros=0;
      int nombreLu=Lecture.unEntier();
      while (nombreLu!=marqueFin) {
         if (nombreLu==0) {nbZeros=nbZeros+1;}
         nombreLu=Lecture.unEntier();
      }
      System.out.print("nombre de zéros = ");
      System.out.println(nbZeros);
   }
}
```

6.7 Boucle pour faire *n* fois

On a souvent à réaliser une itération qui consiste simplement à répéter *n* fois un traitement, *n* étant connu avant de commencer l'itération.

Il peut s'agir de calculer le $n^{\text{ième}}$ élément d'une suite récurrente :

u = a
pour *i=1* à *n* faire *u = f(u)* après l'itération, *u* vaut $f^{(n)}(a)$, *n* compositions d'applications de *f*.

Il peut également s'agir de traiter *n* données introduites par lecture, par exemple faire la somme de

12 valeurs entières lues au clavier :

somme = 0
pour *i=1* à *12* faire *nombreLu* = lecture puis *somme = somme+nombreLu*

Un autre usage fréquent de cette sorte d'itération concerne le traitement des éléments d'un tableau. Nous aurons l'occasion de voir cela dans un chapitre ultérieur.

On peut bien évidemment réaliser ce genre d'itération en utilisant la forme générale déjà vue. Il suffit d'introduire dans la récurrence l'élément supplémentaire *i*, de façon à s'arrêter lorsque *i=n*.

Prenons comme exemple le calcul de $f^{(n)}(a)$:

u = a, i = 0
tant que *i<n* faire *u = f(u), i = i+1*

qui génère bien la suite d'états $<u,i> = <a,0> <f(a),1> <f^{(2)}(a), 2> ... <f^{(n)}(a), n>$

ou, autre exemple, le calcul de la somme de *12* nombres lus au clavier :

somme = 0, i = 0
tant que *i<12* faire
nombreLu* = =lecture puis *somme = somme+nombreLu, i = i+1

Cependant il est plus clair d'utiliser la forme particulière bien adaptée à cette sorte d'itération appelée "***boucle pour***".

La syntaxe de cette boucle en Java est :

```
for (int i=0; i<n; i=i+1) { instructions }
```

Cette forme est strictement équivalente à :

```
{ int i=0; while (i<n) { instructions   i=i+1;} }
```

Remarque : la construction **`for`** ouvre un bloc qui permet d'avoir une variable **`int i`** locale à ce bloc, qui n'a d'existence que pendant la durée d'exécution de l'itération. Nous avons nommé **`i`** cette variable, mais bien sûr tout autre identificateur est acceptable.

On peut ainsi programmer les deux exemples précédents :

Calcul de $f^{n}(a)$:

```
u=a; for (int i=0; i<n; i=i+1) {u=f(u);}
```

Somme de 12 nombres lus au clavier :

```
somme=0;
for (int i=0; i<12; i=i+1) {
  int nombreLu=Lecture.unEntier();
  somme=somme+nombreLu;
}
```

Instructions d'incrémentation et décrémentation : *i*++, *i*--

Le besoin d'incrémenter (ajouter 1) ou de décrémenter (retrancher 1) une variable étant très fréquent, Java, comme C et C++, possède une forme abrégée pour réaliser ces opérations :

***i*++;** est à peu près[1] équivalent à ***i*=*i*+1;**
***i*--;** est à peu près équivalent à ***i*=*i*-1;**

Avec cette notation pour l'incrémentation, la boucle précédente s'écrit donc :

```
for (int i=0; i<12; i++) {...}
```

Tout le monde utilise ces formes abrégées : elles sont pratiques et améliorent la lisibilité si on les utilise ***sans abus***, c'est à dire juste pour incrémenter ou décrémenter une variable qui joue un rôle de compteur.

Forme générale de la boucle pour

La boucle **for** de Java est plus générale que la forme préconisée ci-dessus. La forme générale est :

```
for (initialisation; condition; progression) { instructions }
```

dans laquelle :

- ***initialisation*** est une instruction quelconque destinée à initialiser l'état du contrôle de l'itération,
- ***condition*** est une expression booléenne qui conditionne la continuation de l'itération,
- ***progression*** est une instruction quelconque destinée à faire progresser l'état du contrôle de l'itération.

La sémantique exacte est donnée par la forme strictement équivalente utilisant un **while** :

```
{
  initialisation;
  while (condition) { instructions   progression;}
}
```

On déconseille d'abuser de cette forme générale. Elle n'est ni plus courte ni plus puissante que la forme **while**. C'est un simple chamboulement textuel des rubriques de l'itération.

1. Les différences sont que dans la forme ***i*++** "***i***" n'est évaluée qu'une fois, ce qui peut conduire à un résultat différent si ***i*** est une expression de désignation compliquée, et en tant qu'expression "***i*++**" vaut ***i*** avant incrémentation alors que "***i*=*i*+1**" vaut la valeur de ***i*** après que l'incrémentation soit faite.

6.8 Boucles à résultat polymorphe

objectif : *concevoir une itération dont le résultat peut prendre plusieurs formes.*

On a parfois besoin de calculer par une itération un résultat qui peut prendre plusieurs formes, c'est-à-dire qui ne s'exprime pas de façon naturelle par une valeur ni par une composition de valeurs de types déterminés. Le résultat peut être d'un type ou d'un autre type selon les données. Un tel résultat qui peut avoir plusieurs formes, donc plusieurs "types", est dit *polymorphe*.

Exemple de boucle à résultat polymorphe :

On recherche dans une séquence de nombres le premier nombre ayant une certaine propriété, par exemple le premier nombre pair. S'il n'y a aucun nombre pair dans la séquence, le résultat est de la forme "pas de nombre pair", s'il y a effectivement un nombre pair, le résultat est de la forme "le premier nombre pair vaut telle valeur".

Pour programmer une telle itération, une méthode consiste à utiliser une variable auxiliaire pour indiquer la forme effective du résultat obtenu. Cette variable est d'un type énuméré dont chaque valeur correspond à une des formes du résultat. Dans notre exemple, cette variable, qu'on pourra appeler ***etatRecherche***, aura 3 valeurs possibles : **{*nonDécidé, absent, présent*}**. La valeur initiale de la variable ***etatRecherche*** est ***nonDécidé***, ce qui signifie que la séquence de données lues jusqu'à présent ne permet pas de décider du résultat. Dans notre exemple, ***etatRecherche*** vaut ***nonDécidé*** tant que l'on n'a pas rencontré de nombre pair ni la marque de fin.

Pour la séquence de données 3 56 7 9 8 -1, la suite des états des variables de boucle sera :

i :	*0*	*1*	*2*
$étatRecherche_i$:	*nonDécidé*	*nonDécidé*	*présent*
$nombreLu_i$	*3*	*56*	*56*

Le résultat est la paire *étatRecherche=présent, nombreLu*=56.

Pour la séquence de données 1 3 5 7 -1, la suite des états des variables de boucle sera :

i :	*0*	*1*	*2*	*3*	*4*	*5*
$étatRecherche_i$:	*nonDécidé*	*nonDécidé*	*nonDécidé*	*nonDécidé*	*nonDécidé*	*absent*
$nombreLu_i$:	*1*	*3*	*5*	*7*	*-1*	*-1*

Le résultat est *étatRecherche=absent*. La variable *nombreLu* n'a pas de valeur significative (elle vaut -1, mais c'est sans importance).

La généralisation de ces scénarios conduit au programme suivant :

```
class ChercheNombrePair {

  enum Etat {nonDecide,absent,present}

  public static void main(String[] arg) {
    Etat etatRecherche=Etat.nonDecide;
    int nombreLu=Lecture.unEntier();
    while (etatRecherche==Etat.nonDecide) {
       if (nombreLu==-1) {etatRecherche=Etat.absent;}
       else if (nombreLu%2==0) {etatRecherche=Etat.present;}
       else {nombreLu=Lecture.unEntier();}
    }
    switch (etatRecherche){
    case absent : System.out.println("aucun nombre pair"); break;
    case present : System.out.print("premier nombre pair = ");
        System.out.println(nombreLu);
    }
  }
}
```

Remarque : pour citer une valeur de type énuméré en Java, il faut généralement préfixer l'identificateur de la valeur par le nom du type (**`Etat.nonDecide`**) sauf pour l'usage dans un **`switch`** où on indique simplement l'identificateur de la valeur (**`case absent :`**). Le type de l'argument du **`switch`** indique sans ambiguïté de quel type énuméré il s'agit.

Dans le cas relativement fréquent d'une boucle de "recherche", comme c'est le cas ici, on peut utiliser une programmation "ad-hoc" qui utilise une variable booléenne initialisée à faux et qui est affectée à vrai quand ce que l'on cherche est trouvé.

La condition d'arrêt de la boucle est alors "les données sont toutes visitées ***ou*** on a trouvé". Pour l'exemple précédent, en appelant **`present`** cette variable booléenne, cela donne :

```
class ChercheNombrePair {
  public static void main(String[] arg) {
    boolean present=false;
    int nombreLu=Lecture.unEntier();
    while (!present && nombreLu!=-1) {
       if (nombreLu%2==0) {present=true;}
       else {nombreLu=Lecture.unEntier();}
    }
    if (present) {
      System.out.print("premier nombre pair = ");
      System.out.println(nombreLu);
    }
    else /*!present*/{
      System.out.println("aucun nombre pair");
    }
  }
}
```

Remarque sur les conditions complexes

On a souvent besoin d'exprimer des conditions d'arrêt compliquées, qui ne s'expriment pas par un simple test d'égalité ou autre. C'est le cas dans l'exemple précédent : "les données sont toutes visitées ***ou*** on a trouvé". Une telle condition s'exprime en utilisant les opérateurs logiques usuels :

le "et" (**`&&`**), le "ou" (`||`) et la négation (**`!`**)

Il est impératif de savoir manipuler ces opérations logiques. Par exemple, si ***cond*** est une condition d'arrêt d'une itération, la condition à indiquer dans le **`while`** est sa négation **`!`*****cond***. Il faut donc connaître les lois logiques qui permettent d'exprimer de différentes façons la négation d'une condition complexe. Ce sont les lois de ***De Morgan*** :

non(A ou B) = non A et non B ***non(A et B) = non A ou non B***

soit en Java :

`!`***(A || B) = !A && !B*** **`!`*****(A && B) = !A || !B***

C'est pourquoi dans l'exemple précédent, la condition du **`while`** a été écrite :

```
!present && nombreLu!=-1
```

on aurait pu aussi bien l'écrire (question de goût) :

```
!(present || nombreLu=-1)
```

QCM 6.1

Considérons les fonctions :

```
public static double p1(double x, int k){
// prérequis : k>=0
// résultat : ???
  double resul=1;
  int i=0;
  while(i<k){resul=resul*x; i++;}
  return resul;
}

public static double p2(double x, int k){
// prérequis : k>=0
// résultat : ???
  if(k==0) {return 1;}
  else {return x*p2(x,k-1);}
}
```

- ❑ 1 - Les fonctions **p1** et **p2** sont équivalentes (rendent des résultats identiques).
- ❑ 2 - Les fonctions **p1** et **p2** ne sont pas équivalentes (ne rendent pas des résultats identiques).
- ❑ 3 - La fonction **p1** est récursive.
- ❑ 4 - La fonction **p2** est récursive.
- ❑ 5 - La fonction **p1** est itérative.
- ❑ 6 - La fonction **p2** est itérative.
- ❑ 7 - **p1(10,5)** vaut 50.
- ❑ 8 - **p2(10,5)** vaut 50.
- ❑ 9 - **p1(10,5)** vaut 100000.
- ❑ 10 - **p2(10,5)** vaut 100000.
- ❑ 11 - **p1(2,5)** vaut 10.
- ❑ 12 - **p2(2,5)** vaut 64.
- ❑ 13 - **p1(1,5)** vaut 5.
- ❑ 14 - **p2(1,5)** vaut 1.
- ❑ 15 - **p1(-1,5)** vaut -5.
- ❑ 16 - **p2(1,5)** vaut -1.
- ❑ 17 - La fonction **p1** rend en résultat **x** fois **k**.
- ❑ 18 - La fonction **p1** rend en résultat **x** à la puissance **k** (**x** multiplié **k** fois par lui-même).
- ❑ 19 - La fonction **p1** ne termine pas toujours.
- ❑ 20 - La fonction **p2** rend en résultat **x** fois **k**.
- ❑ 21 - La fonction **p2** rend en résultat **x** à la puissance **k** (**x** multiplié **k** fois par lui-même).
- ❑ 22 - La fonction **p2** ne termine pas toujours.

QCM 6.2

Considérons la fonction :

```
static int log(int b,int n){
// prérequis : b>=2, n>0
// résultat : la partie entière du logarithme base b de n
  int resul=0; int nn=n;
  while(nn!=1){
    nn=nn/b; resul++;
  }
  return resul;
}
```

Rappel : la partie entière du logarithme base ***b*** de ***n*** est égal au nombre de chiffres significatifs de l'écriture en base ***b*** de ***n*** moins 1. On doit avoir par exemple log(10,3675)=3, car $1000 \leq 3675 < 10000$.

- ❑ 1 - **`log(10,10000)`** rend 4.
- ❑ 2 - **`log(10,10000)`** rend 1.
- ❑ 3 - **`log(10,10000)`** ne termine pas.
- ❑ 4 - **`log(2,4)`** rend 2.
- ❑ 5 - **`log(2,32)`** rend 5.
- ❑ 6 - **`log(2,32)`** ne termine pas.
- ❑ 7 - **`log(10,3675)`** rend 3.
- ❑ 8 - **`log(10,3675)`** ne termine pas.
- ❑ 9 - Cette fonction est correcte (satisfait à sa spécification).
- ❑ 10 - Cette fonction est incorrecte car ne termine pas toujours.

Exercices

objectif : *le but des 3 exercices qui suivent est de*

- *résoudre un problème de manière récursive en trouvant de bonnes équations,*
- *résoudre un problème de manière itérative en imaginant une suite récurrente et en testant cette suite sur des exemples,*
- *savoir prouver qu'une itération est correcte en exhibant un invariant et une fonction de terminaison.*

Dans les trois exercices qui suivent, on s'intéresse aux chiffres de l'écriture décimale d'un nombre entier. Par exemple l'entier ***n***=3827 est représenté en décimal par la suite de chiffres :

7 2 8 3

Ici, un chiffre décimal sera simplement un entier compris entre 0 et 9.

Le premier chiffre est le chiffre de poids faible, ou chiffre de rang 0. Il s'obtient comme reste de la division du nombre par 10 : n%10 = 7

Les chiffres restants sont les chiffres du quotient du nombre par 10 : n/10=382.

Ceci nous permet d'utiliser un nombre entier comme une suite de données, ses chiffres décimaux. C'est un petit subterfuge qui nous permet de nous passer provisoirement de structures de données mieux adaptées telles que les tableaux, les chaînes de caractères, les listes...

Exercice 6.1 Récursivité - itération : $i^{ème}$ chiffre décimal d'un nombre

Étant donné un entier *n>=0*, on souhaite trouver le ***$i^{ème}$ chiffre*** de son écriture décimale. Les chiffres sont numérotés à partir de 0 depuis les poids faibles. Si *i* est plus grand que le rang du plus fort chiffre significatif, le résultat sera 0.

Exemple : pour *n*=245

`chiffre(`*`0`***`,`***`n`***`)`** vaut 5,

`chiffre(`*`1`***`,`***`n`***`)`** vaut 4,

`chiffre(`*`2`***`,`***`n`***`)`** vaut 2,

`chiffre(`*`3`***`,`***`n`***`)`** vaut 0...

Rappels : le chiffre décimal de rang 0 de **`n`** est le reste de la division de **`n`** par 10 ("*n* modulo 10", **`n%10`** en java). les autres sont ceux du quotient entier de *n* par 10 (**`n/10`** en Java).

Exemple : **`245%10`** vaut ***5***, **`245/10`** vaut ***24***

- Rédiger la fonction **`chiffre`** de manière ***récursive***. On commencera par indiquer les équations qui justifient cette définition récursive.
- Rédiger la fonction **`chiffre`** de manière ***itérative***. On commencera par envisager une suite qui conduise à la solution et on testera cette suite sur quelques scénarios.
- Justifier l'exactitude de l'algorithme itératif au moyen d'un invariant de boucle.

Exercice 6.2 Récursivité - itération : somme des chiffres décimaux d'un nombre

On considère la fonction ***SommeChiffres*** qui vaut la somme des chiffres de l'écriture décimale d'un entier positif ou nul.

exemple : ***SommeChiffres(***582***)*** = 2 + 8 + 5 = 15

- Rédiger cette fonction de manière récursive (fonction **`SommeChiffresRec`**). On commencera par indiquer les équations qui justifient cette définition récursive.
- Rédiger cette fonction de manière itérative (fonction **`SommeChiffresIter`**). On commencera par envisager une suite qui conduise à la solution et on testera cette suite sur quelques scénarios.
- Indiquer l'invariant de boucle et une fonction de terminaison de la version itérative.

Exercice 6.3 Récursivité - itération : inclusion des chiffres décimaux

Étant donnés deux entiers *m* et *n*, on souhaite déterminer si l'écriture décimale de *m* "contient" celle de *n*. On suppose que *m* >= 0, *n* >= 0. On suppose également que $m < 2^{31}-1$, de sorte qu'il peut être représenté par une valeur de type **`int`** en Java.

Exemple : si *m* = 9**41**3 et *n* = **41** alors **`contient(`***m***`,`***n***`)`** est vrai.

Pour réaliser cette fonction, on a besoin d'une fonction auxiliaire plus simple, **`termine(`***m***`,`***n***`)`** qui vaut ***vrai*** si l'écriture décimale de *n* est égale à la fin de celle de *m* et ***faux*** sinon. Exemples :

`termine(`941**`,`**41**`)`** est vrai, **`termine(`**9418**`,`**41**`)`** est faux.

Version récursive :

- Rédiger la fonction **`termine`** de manière récursive.
- Rédiger la fonction **`contient`** de manière récursive.
- Rédiger une procédure principale qui teste **`contient`** et **`termine`** sur des exemples pertinents.

Version itérative :

- Refaire la même chose en programmant les fonctions **`termine`** et **`contient`** de manière itérative.
- Indiquer les invariants de boucles qui justifient l'exactitude de ces itérations.

Exercice 6.4 Itération : plus grand commun diviseur

objectif : *savoir gérer la malencontreuse séquentialité des affectations dans un corps de boucle.*

Rédiger une fonction qui calcule le *pgcd* de deux nombres *a* et *b* les strictement positifs en utilisant les propriétés suivantes :

avec *a*>*b* : si le reste de la division de *a* par *b* est nul, *b* est le pgcd, sinon le pgcd de *a* et *b* est égal au pgcd de *b* et du reste. Exemple :

pgcd(310,21) = pgcd(21,12) = pgcd(12,9) = pgcd(9,3). Ici *3* divise *9*, le pgcd est donc *3*.

Exercice 6.5 Itération : Résolution d'une équation

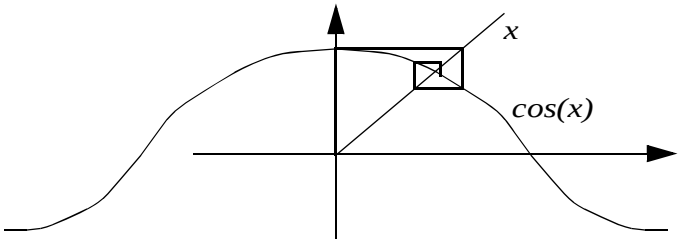
objectif : *savoir utiliser des identifications auxiliaires pour éviter de calculer plusieurs fois la même chose.*

On se propose de trouver une solution de l'équation ***x = cos(x)*** entre 0 et $\pi/2$.

Pour cela on peut calculer les termes de la suite :

$x_0 = 0$, $x_{i+1} = cos(x_i)$ pour $i>0$

Cette suite converge vers la solution, comme le suggère le schéma suivant :



Rédiger la fonction qui calcule cette solution à epsilon près, epsilon étant passé en paramètre.

Pour calculer le cosinus, on dispose de la fonction : **`double Math.cos(double x)`**.

Exercice 6.6 Itération : calcul d'une valeur approchée de π

objectif : *savoir transformer une itération pour la rendre plus efficace en utilisant des variables auxiliaires contenant les valeurs des expressions coûteuses à calculer.*

On peut calculer une valeur approchée de π à l'aide de la série :

$$\pi = \lim_{n \to \infty} 4 \times \sum_{i=0}^{n} \frac{-1^i}{2i+1} = 4 \times \left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \ldots\right)$$

Les termes de la série étant alternativement positifs et négatifs, et strictement décroissants en valeur absolue, la série converge par valeurs alternativement supérieures et inférieures, et le n-ième terme est un majorant de l'erreur commise en arrêtant le calcul au rang *n*.

Rédiger et tester une fonction **`calculPi`** calculant π à epsilon près, epsilon étant donné en paramètre.

Première version :

Dans cette version on calculera chaque terme de la série comme l'indique la formule $\frac{-1^i}{2i+1}$ en calculant -1^i au moyen de la fonction **`Math.pow(-1,i)`** dont le résultat est un nombre réel.

Seconde version :

Dans cette version, on cherchera à éviter les calculs de -1^i (alternance des signes) et de $2i+1$ en les remplaçant par des opérations jugées moins coûteuses.

Exercice 6.7 Boucle pour : fonction puissance

objectif : *savoir utiliser la boucle pour.*

Rédiger une fonction qui calcule a^n (a à la puissance n, a nombre réel et n nombre entier positif ou nul), en utilisant la multiplication.

Exercice 6.8 Récurrence du second ordre : suite de Fibonacci

objectif : *savoir transformer une récurrence du second ordre en récurrence du premier ordre.*

La suite de Fibonacci est définie par la relation suivante :

$fibo(0) = 0$

$fibo(1) = 1$

$fibo(n) = fibo(n\text{-}2) + fibo(n\text{-}1)$ pour $n > 1$

On se propose de rédiger une fonction **`int fibo(int n)`** qui calcule le $n^{\text{ième}}$ terme de cette suite.

La définition utilise une ***récurrence du second ordre*** (le rang n dépend des rangs n-1 et n-2). Nous savons calculer par une itération les termes d'une récurrence du premier ordre. Comment faire pour le second ordre (ou tout ordre supérieur) ?

Principe général : ***on se ramène à une récurrence du premier ordre***, mais ***vectorielle***. Ici cela devient :

$$\begin{bmatrix} fiboNMoins1_i \\ fiboN_i \end{bmatrix} = \begin{bmatrix} fiboN_{i-1} \\ fiboNMoins1_{i-1} + fiboN_{i-1} \end{bmatrix}$$

Appliquer ce principe pour programmer **`fibo`** de manière itérative.

objectif : *Les 5 exercices qui suivent concernent des itérations sur entrées de données. Le but principal est de placer convenablement les instructions de lecture.*

Exercice 6.9 Itération sur entrées de données : moyenne

Rédiger un programme qui lit une suite de nombres entiers positifs, terminée par -1 et affiche la moyenne de ces nombres.

Exemple : 7 3 0 4 -1

la moyenne est 3.5

Exercice 6.10 Itération sur entrées de données : maximum d'une séquence de nombres

On entre au clavier une suite non vide d'entiers positifs terminée par la marque -1. Rédiger un programme qui lit ces nombres et calcule le maximum de la suite.

Exemple : 7 3 0 4 1 8 23 0 9 -1

le maximum est 23

Exercice 6.11 Itération sur entrées de données : plus longue sous suite constante

On entre au clavier une suite non vide d'entiers positifs terminée par la marque -1. Rédiger un programme qui lit ces nombres et calcule la longueur de la plus longue sous-suite constante.

Exemple : 7 5 5 4 1 1 1 1 8 8 4 -1

longueur de la plus longue sous-suite constante = 4 (suite de quatre 1).

Exercice 6.12 Nombre de "le"

Rédiger un programme qui affiche le nombre de paires de caractères successifs **"le"** dans un texte lu au clavier et terminé par un point.

Exercice 6.13 Nombre de mots

Rédiger un programme qui détermine et affiche le nombre de mots contenus dans une phrase terminée par un point. On considère que le caractère séparateur de mots est le caractère espace.

Aide 6.1 Récursivité - itération : i^{ème} chiffre décimal d'un nombre

Version récursive : utiliser les équations suivantes

- pour ***i***=0, ***chiffre(i,n) = n*%10**
- pour ***i***>0, ***chiffre(i,n)=chiffre(i-1,n*/10*)***

Version itérative :

Voici un exemple de scénario, pour la recherche du chiffre de rang 3 dans 24593 :

ii	3	2	1	0
nn	2**4**593	2**4**59	2**4**5	2**4**

Aide 6.2 Récursivité - itération : somme des chiffres décimaux d'un nombre

Version récursive : utiliser les équations suivantes

- Pour n=0, la somme des chiffres est nulle.
- Pour n>0, la somme des chiffres est la somme du chiffre de poids faible et de la somme des chiffres du nombre restant une fois enlevé le chiffre de poids faible. Exemple :

sommeChiffres(285) = 5 + sommeCchiffres(28)

Nous avons donc les équations :

- pour ***n***=0, ***sommeChiffres(n)*** = 0
- pour ***n***>0, ***sommeChiffres(n)*** = ***n%10 + sommeChiffres(n/10)***

Version itérative :

L'idée consiste à cumuler les chiffres de ***n*** en commençant par les poids faibles.

Exemple pour n=285 :

somme	0	5	13	15
resteAVoir	285	28	2	0

Aide 6.3 Récursivité - itération : inclusion des chiffres décimaux

Version récursive de `termine` :

- si ***n***==0, ***termine(m,n)*** est ***vrai*** (par convention).
- si ***n***>0 et les chiffres de poids faibles de ***n*** et ***m*** sont différents, ***termine(n,m)*** est ***faux***.
  Exemple : ***termine(***457**6**, 123***)*** est ***faux***
- si ***n***>0 et les chiffres de poids faibles de n et m sont égaux, ***termine(m,n)*** est ***vrai*** si et seulement si le nombre constitués des poids plus forts de ***n*** termine le nombre constitué des poids plus forts de ***m***.
  Exemples : ***termine(***4123, 123***)*** est ***vrai***
  termine(4523, 123***)*** est ***faux***

Ce qui donne les équations :

- si ***n***==0, ***termine(m,n)***=***vrai***
- si ***n***>0 et ***n%***10≠***m%***10, ***termine(m,n)***=***faux***
- si ***n***>0 et ***n%***10≠***m%***10, ***termine(m,n)***= ***termine(m***/10**,n**/10***)***

Version récursive de `contient` :

- si ***n***>***m***, ***contient(m,n)*** est ***faux***.
  Exemple : ***contient(***87, 463***)*** est ***faux***
- si ***n***≤***m*** et ***termine(m,n)*** est ***vrai***, ***contient(n,m)*** est ***vrai***.
  Exemple : ***contient(***4**123,** 123***)*** est ***vrai***
- si ***n***≤***m*** et ***termine(m,n)*** est ***faux***, ***contient(m,n)*** vaut la même chose que ***contient(m***/10**,n)***
  Exemples : ***contient (***2**423**56, 423***)*** = ***contient (***2**423**5, 423***)*** = ***contient (***2**423**, 423***)*** = ***vrai***
  contient(24585,423***)*** = ***contient(***2758,423***)*** = ***contient(***275,423***)*** = ***faux***

Ce qui donne les équations :

- si ***n***>***m***, ***contient(m,n)*** = ***faux***.
- si ***n***≤***m*** et ***termine(m,n)***, ***contient(m,n)***=***vrai***.
- si ***n***≤***m*** et ***!termine(m,n)*** est ***faux***, ***contient(m,n)***=***contient(m***/10**,n)***

Ces équations conduisent facilement aux versions récursives de **`termine`** et de **`contient`**.

Version itérative de `termine` :

Le principe consiste à comparer les chiffres de ***n*** et ***m*** à partir des poids faibles.

Exemple de scénario pour lequel ***termine(m,n)***=***vrai*** :

mm	94**1**	9**4**	9
nn	4**1**	**4**	0

Le résultat ***vrai*** est atteint quand ***nn***=0, ce qui signifie que l'on a comparé avec succès tous les chiffres de ***n*** avec ceux de ***m***.

Exemple de scénario pour lequel ***termine(m,n)***=***faux*** :

mm 9481 948

nn 41 4

Le résultat ***faux*** est atteint quand les chiffres de poids faible de ***nn*** et ***mm*** sont différents.

Version itérative de `contient` :

Le principe consiste à tester si les chiffres de `n` se rencontrent quelque part à l'intérieur des chiffres de `m`.

Exemple de scénario pour lequel ***contient(m,n)***=***vrai*** :

mm 9413 941

`n` 41 **41** --> résultat ***vrai*** car `termine(mm,n)`

Exemple de scénario pour lequel ***contient(m,n)***=***faux*** :

mm 9413 941 94 9

`n` 43 43 43 43 --> résultat ***faux*** car ***mm*****`<n`**

Aide 6.4 Itération : plus grand commun diviseur

Les propriétés indiquées nous suggèrent une suite de deux composantes, ***aa*** et ***bb***, avec ***aa***≥***bb***. Il faut se méfier que ***aa*** soit bien supérieur à ***bb*** initialement. Pour cela il doit être initialisé avec la plus grande valeur parmi `a` et `b`, et ***bb*** doit être initialisée avec l'autre valeur. Le scénario suivant illustre le calcule du pgcd de 21 et 390 :

aa 390 21 12 9 3

bb 21 12 9 3 0

Le pgcd est 3. Il est atteint quand ***bb***=0.

Aide 6.9 Itération sur entrées de données : moyenne

La moyenne est la somme divisée par le nombre de nombres sommés. On peut difficilement calculer la moyenne de façon itérative. Il faut calculer la somme et le nombre de nombres, puis après l'itération calculer la moyenne. Exemple :

nombreLu : 7 4 0 3 -1

somme : 0 7 11 11 14

nbNombres : 0 1 2 3 4

puis : moyenne = 14/4 = 3.5

Aide 6.10 Itération sur entrées de données : maximum d'une séquence de nombres

En plus de la suite des valeurs lues, il faut avoir le maximum des donnée lues depuis le début de la séquence. Le résultat est bien évidemment ce maximum une fois lue toute la séquence.

Exemple :

nombreLu :	2	5	1	6	3	2	8	3	4	-1
max :	2	5	5	6	6	6	8	8	8	

Aide 6.11 Itération sur entrées de données : plus longue sous-suite constante

Nous envisageons une suite composée :

- du nombre lu : ***nombreLu***,
- de la valeur de la sous-suite constante courante : ***valSousSuite***,
- de la longueur courante de la sous-suite constante courante : ***longueurSousSuite***,
- de la longueur maximum des sous-suites rencontrées : ***longueurMax***.

Voici un exemple de scénario :

nombreLu :	7	5	5	4	1	1	1	1	8	8	4	-1
valSousSuite :	7	5	5	4	1	1	1	1	8	8	4	
longueurSousSuite :	1	1	2	1	1	2	3	4	1	2	1	
longueurMax :	1	1	2	2	2	2	3	4	4	4	4	

Aide 6.12 Nombre de "le"

L'idée générale est d'avoir en permanence deux caractères successifs du texte, ***car1*** et ***car2*** et de les comparer avec `'l'` suivi de `'e'` :

car1	t	y	l	e	l	u	l	e	.
car2	y	l	e	l	u	l	e	.	
nombreDeLe	0	0	1	1	1	1	2	2	

Remarque : Il faut initialiser ***car1*** et ***car2*** par deux lectures avant l'itération.

Aide 6.13 Nombre de mots

Quelques précisions : nous acceptons les textes de 0 caractère, et dans ce cas le nombre de mots et 0. Les mots peuvent être séparés par un nombre quelconque d'espaces.

On peut partir du principe que ce qui caractérise un mot est la "fin" du mot. Une fin de mot est caractérisée une paire de caractères dont le premier **`car1`** n'est pas un espace et le second **`car2`** est un espace :

```
le  corbeau    et le     renard     .
  1        2     3              4        4 mots
```

Il ne faut pas oublier le cas où le dernier mot serait terminé par le point final :

```
le  corbeau    et le     renard.
  1        2     3              4        4 mots également
```

Il y a un mot de plus à la fin si **`car1`** n'est pas un espace après avoir lu le point final.

CHAPITRE 7 Type énuméré, type structure, notion de référence

objectif : *comprendre l'intérêt de définir de nouveaux types de données, spécifiques à un domaine d'applications. Nous verrons dans ce chapitre les types énumérés et les structures. Les structures au sens strict ne sont pratiquement plus utilisées dans les langages modernes : elles sont avantageusement remplacées par les classes qui permettent d'assurer l'abstraction, notion essentielle pour la compréhension et la fiabilité des logiciels. Les types classe seront vus dans un chapitre ultérieur.*

La simplicité des exemples choisis jusqu'à présent a permis de représenter les données traitées par les algorithmes en utilisant uniquement les types prédéfinis mis à notre disposition : entier, réel, caractère, etc... Il n'en n'est pas de même lorsque les données à représenter sont plus complexes.

Exemples :

- On veut représenter les cartes d'un jeu de 32 cartes. Il faut notamment représenter la force de chaque carte, ***sept***, ***huit***, ***neuf***, ***dix***, ***valet***, ***dame***, ***roi***, ***as***, ainsi que sa couleur, ***trèfle***, ***carreau***, ***cœur***, ***pique***. On peut évidemment imaginer de représenter ces valeurs par des entiers, mais les algorithmes deviennent alors totalement illisibles.
- On veut représenter les clients d'une société de vente. Aucun des types prédéfinis n'est adapté à représenter par un seul objet, un ***nom***, une ***adresse*** et un ***compte***.

Dans chacun de ces cas, il va falloir ***inventer un nouveau type de données***. Dans ce chapitre nous allons étudier les plus simples des types programmés : les types énumérés et les types structures.

7.1 Type énuméré

objectif : *approfondir l'usage des types énumérés.*

Comme nous l'avons déjà vu, un type énuméré est défini par l'énumération de ses valeurs. Chaque valeur est notée au moyen d'un identificateur. Par exemple, pour la couleur et la force d'une carte on peut définir en Java les types énumérés :

```
enum CouleurDeCarte {trefle, carreau, coeur, pique}
enum ForceDeCarte {sept, huit, neuf, dix, valet, dame, roi, as}
```

Un type énuméré offre peu de propriétés : dans les cas usuels, on n'a besoin que du test d'égalité. Dans certains cas, on peut vouloir une relation d'ordre, par exemple pour la force d'une carte, on peut souhaiter l'ordre `sept<huit<neuf<dix<valet<dame<roi<as`.

La relation d'ordre en Java est définie par l'ordre d'énumération des identificateurs de valeurs. La

comparaison se fait au moyen de la fonction **compareTo**. La syntaxe d'utilisation de cette fonction est particulière : ***a*** et ***b*** étant des expressions d'un même type énuméré,

***a*.compareTo(*b*)**

rend en résultat un entier <0 si ***a*** est inférieur à ***b***, 0 si ***a*** est égal à ***b*** et >0 si ***a*** est supérieur à ***b***.

Exemple : la fonction suivante indique si une couleur est "rouge" (carreau ou cœur) :

```
static boolean estRouge(CouleurDeCarte c) {
// résultat : indique si c est rouge
  return c==CouleurDeCarte.carreau || c==CouleurDeCarte.coeur;
}
```

On aurait pu également l'écrire, en utilisant **compareTo** :

```
static boolean estRouge(CouleurDeCarte c) {
// résultat : indique si c est rouge
  return   c.compareTo(CouleurDeCarte.carreau)==0
                       || c.compareTo(CouleurDeCarte.coeur)==0;
}
```

Type énuméré et entrées-sorties

Les types énumérés sont utilisés pour améliorer la lisibilité des programmes. Il peuvent également servir à communiquer avec l'extérieur, au moyen de lectures et d'écritures. Les langages de programmation offrent rarement la possibilité de réaliser des opérations d'entrées sorties "en clair" de valeurs d'un type énuméré. En revanche, Java le permet. La forme "en clair" des valeurs d'un type énuméré sont simplement les chaînes de caractères de leurs identificateurs.

Il existe pour chaque type énuméré deux fonctions de conversion :

- L'appel de fonction ***x*.toString()** rend en résultat la chaîne de caractères correspondant à la valeur de ***x***. Exemple :

  CouleurDeCarte c = CouleurDeCarte.trefle;

  System.out.println(c.toString()); affiche "**trefle**"

  On peut écrire plus directement :

  System.out.println(c); affiche "**trefle**"

  car dans ce cas le compilateur rajoute automatiquement le **.toString()**.

- L'appel de fonction ***nomDuTypeEnuméré*.valueOf(*s*)** rend en résultat la valeur de type énuméré ayant pour identificateur la chaîne de caractères ***s***. Exemple :

  CouleurDeCarte c = CouleurDeCarte.valueOf("trefle");

  donne à **c** la valeur **CouleurDeCarte.trefle**.

  On peut ainsi obtenir facilement des valeurs de type énuméré par lecture au clavier :

  CouleurDeCarte couleurChoisie;

  couleurChoisie = CouleurDeCarte.valueOf(Lecture.chaine());

  Si la chaîne de caractères ne correspond à aucun identificateur de valeur du type énuméré, la fonction **valueOf** provoque une erreur (une "exception" de type **IllegalArgumentException**).

7.2 Type structure

objectif : *apprendre à caractériser des données complexes à l'aide de composantes plus simples.*

Un type structure permet de représenter des données ayant plusieurs caractéristiques appartenant chacune à un domaine de valeurs, donc d'un certain type. Cela correspond à peu près à l'idée mathématique du produit cartésien de plusieurs domaines. Comme exemple on peut considérer :

- Un ***point*** dans un plan : ses caractéristiques sont l'abscisse et l'ordonnée. Le type de l'***abscisse*** est un nombre réel et celui de l'***ordonnée*** également.
- Une ***carte à jouer*** : ses caractéristiques sont la force et la couleur. La ***force*** est de type énuméré ForceDeCarte={*sept*, *huit*, *neuf*, *dix*, *valet*, *dame*, *roi*, *as*} et la ***couleur*** de type énuméré CouleurDeCarte= {*trèfle*, *carreau*, *cœur*, *pique*}.
- Une ***personne*** : ses caractéristiques sont le ***nom***, l'***âge*** et le ***sexe***. Le nom est de type chaîne de caractères, l'âge de type entier et le sexe de type énuméré Sexe={*masculin*, *féminin*}.

7.2.1 Définition d'un type structure

Une structure se présente donc comme une collection de rubriques, appelées ***champs***, ou encore ***composants***. Chaque composant possède un type et est identifié par un nom :

structure ***Point*** = < réel x, réel y >
structure ***CarteAJouer*** = < ForceDeCarte force, CouleurDeCarte couleur >
structure ***Personne*** = < ChaîneDeCaractères nom, entier âge, Sexe sexe >

Des valeurs de ces types pourraient être :

Point origine = <0,0>, Point p = <1.0, 12.45>
CarteAJouer mistigri = <*valet*, *trèfle*>
Personne toto = <"toto", 14, *masculin*>

En Java, les types "structure" se programment comme un cas particulier d'objets de type classe qui sera décrit au chapitre 10. On peut définir ainsi les types précédents :

```
class Point {
  public double x; public double y;
}

class CarteAJouer {
  public ForceDeCarte force; public CouleurDeCarte couleur;
}

class Personne {
  public String nom; public int age; public Sexe sonSexe;
}
```

Le vocable **`public`** signifie que ces composants sont accessibles en dehors du texte de la classe.

Notion de constructeur

Lorsqu'on crée un objet de type structure, on a fréquemment besoin de lui donner une valeur. On verra plus loin qu'on peut modifier par affectation les champs d'une structure, mais il est préférable de prévoir l'initialisation globale de tous les champs dès la création. Cela donne dans tous les cas une meilleure lisibilité et c'est même nécessaire pour des structures dont les membres seraient des constantes, c'est-à-dire qu'on ne peut plus modifier une fois l'objet créé.

En Java, cette initialisation se fait au moyen d'un ***constructeur*** que l'on définit dans le texte de la classe. Un constructeur se présente un peu comme une procédure. Il doit impérativement porter le même nom que la classe. Il a des paramètres, mais ne rend aucun résultat : son rôle est d'initialiser l'objet au moment de sa création. Dans le cas d'une structure, il est bon de programmer un constructeur qui a comme paramètres les valeurs initiales de tous les champs de la structure (on appelle parfois cela le "constructeur trivial"). En reprenant les exemples précédents, cela donne :

```
class Point {
   public double x; public double y;
   public Point(double sonX, double sonY) {x=sonX; y=sonY;}
}

class CarteAJouer {
   public ForceDeCarte force; public CouleurDeCarte couleur;
   public CarteAJouer(ForceDeCarte f, CouleurDeCarte c) {
      force=f; couleur=c;
   }
}

class Personne {
   public String nom; public int age; public Sexe sonSexe;
   public Personne(String n, int a, Sexe s) {
      nom=n; age=a; sexe=s;
   }
}
```

En Java, la création d'une structure se fait au moyen de l'opération **new** suivie du constructeur doté de paramètres effectifs. Avec ces constructeurs, on pourra créer des structures initialisées en utilisant les formes suivantes :

```
Point origine = new Point(0,0); Point p = new Point(1.0,12.45);
CarteAjouer mistigri =
       new CarteAJouer(ForceDeCarte.valet, CouleurDeCarte.trefle);
Personne toto = new Personne("toto",14,Sexe.masculin);
```

Les déclarations précédentes supposent les définitions suivantes de types énumérés :

```
enum CouleurDeCarte {trefle, carreau, coeur, pique}
enum ForceDeCarte {sept, huit, neuf, dix, valet, dame, roi, as}
enum Sexe {masculin, feminin}
```

7.2.2 Création de structure - désignation par référence

objectif : *comprendre la création dynamique d'objets et la désignation par référence..*

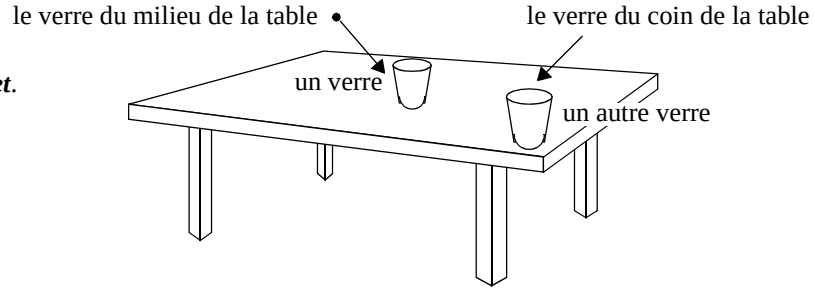
En Java, comme dans d'autres langages à objets, les structures sont un cas particulier d'objet et les objets sont ***obligatoirement désignés par référence***.

7.2.2.1 Première notion d'objet

Un objet est quelque chose qui nécessite d'être créé et qui est doté d'une identité. De façon intuitive, un objet est comme "une vraie chose". Par exemple, sur une table se trouvent deux verres :

- Ils n'ont pas toujours existé, ils ont un jour été ***créés***.
- Ils ont des caractéristiques qui constituent leur "valeur" : une hauteur, un diamètre, une couleur... Ils ont tous deux 10 cm de haut, 5 cm de diamètre et sont transparents. Mais bien que possédant les même caractéristiques, bien qu'ils soient ***indiscernables par leur valeur***, il ne s'agit ***pas du même*** verre : il y a le verre du milieu de la table et le verre du coin de la table. Ils ont chacun une ***identité***.

Un verre est un objet.



De ce point de vue, la notion d'objet s'oppose à la notion de valeur. Une "valeur" est un élément d'un ensemble, au sens mathématique. Par exemple ***12 est une valeur***, c'est un élément de l'ensemble des entiers. 12 n'a jamais été "créé" (du moins pas au sens de la fabrication d'un objet physique). Il n'a pas d'identité : son existence est une existence "mathématique", elle ne se situe ni dans le temps, ni dans l'espace. ***12 n'est pas un objet***.

7.2.2.2 Notion de référence

Une ***référence*** est une ***valeur qui désigne un objet***. On dit parfois une "***adresse***". Dans l'exemple des deux verres, "le verre du milieu de la table" et "le verre du coin de la table" sont des références. Elles désignent respectivement le verre du milieu de la table et le verre du coin de la table.

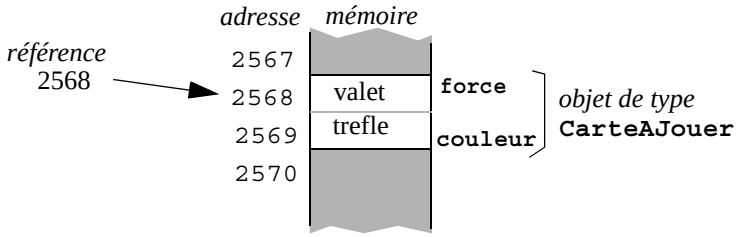
Une référence est en quelque-sorte le "nom" d'un objet, mais contrairement aux identificateurs que nous avons déjà utilisés, les références sont des valeurs manipulables par les programmes : on va pouvoir en passer en paramètre, en rendre en résultat, en mémoriser dans des variables... Une référence est un "nom interne", en ce sens qu'il constitue une donnée au même titre qu'une valeur entière par exemple. Par contre un identificateur est un "nom externe" qui n'a pas d'existence objective pendant l'exécution.

Il existe une valeur particulière de type référence qui ne désigne aucun objet. C'est la "référence à rien", encore appelée "référence nulle". En Java elle se note :

`null` : référence qui ne désigne rien

Du point de vue de la mise en œuvre dans un ordinateur, un objet est représenté par une succession

de mots de mémoire qui contiennent les divers composants de l'objet. Chaque mot de la mémoire de l'ordinateur est désigné par un numéro qu'on appelle son ***adresse***. La référence à un objet est simplement l'adresse où est rangé l'objet en mémoire.



7.2.2.3 Création d'un objet

Un ***expression de création*** permet de créer des objets. Par exemple, l'expression

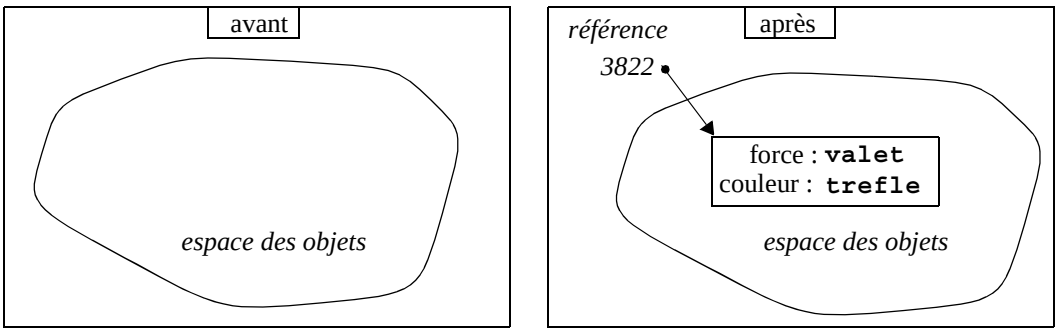
créer une CarteAJouer = < valet, trèfle >

crée un nouvel objet de type ***CarteAJouer*** dont la force est ***valet*** et la couleur ***trèfle***. Cette expression rend en résultat la référence qui désigne l'objet créé.

En Java, les objets sont toujours créés par une expression de création dont l'opérateur est **`new`**. L'expression précédente s'écrit :

```
new CarteAJouer(ForceDeCarte.valet,CouleurDeCarte.trefle)
```

Son effet est illustré par le schéma suivant :



De façon concrète et proche de la réalisation en machine, de la place mémoire est réservée pour le nouvel objet, dans "l'espace des objets", le constructeur est exécuté pour initialiser l'objet et l'adresse mémoire de l'objet, qui constitue la référence à l'objet, est rendue en résultat (3822 dans l'exemple de la figure).

La forme générale de la création est :

`new` ***nomDeLaClasse*** (***paramètres du constructeur***)

7.2.2.4 Déclaration de références

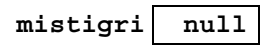
En général, lorsqu'on crée un objet, il faut ***capter sa référence*** pour ensuite utiliser l'objet. On peut le faire en déclarant des variables, des constantes ou des paramètres de type référence. Une déclaration de référence à objet de type classe ***T*** s'écrit :

T nom ;

Par exemple :

`CarteAJouer mistigri;` variable de type référence à un objet de type **`CarteAJouer`**.

Ceci est une déclaration sans initialisation explicite. En Java, une telle variable est initialisée à **`null`**, référence qui ne désigne rien.

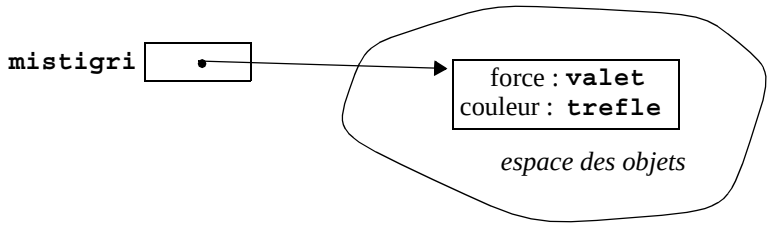


On peut ensuite affecter cette variable, par exemple pour capter la référence à un objet créé :

```
mistigri =
       new CarteAJouer(ForceDeCarte.valet,CouleurDeCarte.trefle);
```

On peut également, comme pour toute déclaration de variable, l'initialiser à l'endroit de sa déclaration :

```
CarteAJouer mistigri =
       new CarteAJouer(ForceDeCarte.valet,CouleurDeCarte.trefle);
```



On peut également déclarer des constantes de type référence, au moyen de l'attribut **`final`** :

`final` ***T nom*** **`=`** ***expression*****`;`**

Par exemple :

```
final CarteAJouer mistigri = new CarteAJouer(valet,trefle);
```

Dans ce cas on est sûr que l'identificateur désignera toujours le même objet.

On peut aussi, comme avec tout autre type, avoir des paramètres de type référence et des résultats de fonction de type référence. Nous en aurons quelques exemples dans le paragraphe suivant.

7.2.3 Sélection de champ

Les champs d'une structure sont accessibles au moyen d'une opération appelée ***sélection de champ***. Etant donnée une référence à objet ***obj*** qui possède un champ de nom ***c***, l'expression :

obj**`.`***c*

désigne le champ ***c*** de ***obj***.

En tant qu'expression, elle vaut ce que vaut le champ : **`mistigri.force`** vaut **`valet`**.

Champs variables - champs constants

- Un champ déclaré sans aucun autre vocable peut s'utiliser comme une ***variable*** que l'on peut affecter :

  `mistigri.couleur=pique;` modifie la couleur de l'objet désigné par **`mistigri`**.

- Si on désire qu'un champ ne soit pas modifiable, il suffit de le déclarer avec le vocable **`final`**. Le champ ainsi déclaré se comporte comme une ***constante***. Il reçoit sa valeur par le ***constructeur***, au moment de la création de la structure, et cette valeur est définitive :

```
class CarteAJouer {
   public final ForceDeCarte force;
  public final CouleurDeCarte couleur;
   public CarteAJouer(ForceDeCarte f, CouleurDeCarte c) {
      force=f; couleur=c;
   }
}
```

Avec cette définition de **`CarteAjouer`**, la force et la couleur d'une carte ne sont pas modifiables. avec

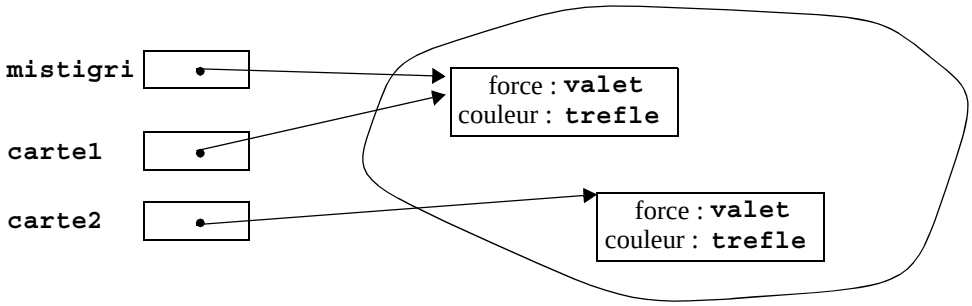
```
mistigri =
     new CarteAJouer(ForceDeCarte.valet,CouleurDeCarte.trefle);
```

la carte désignée par mistigri est à tout jamais le valet de trèfle.

7.2.4 Comparaison des références - comparaison des valeurs

La comparaison d'égalité est définie pour les références aux objets. C'est le seul opérateur de comparaison applicable entre références. Deux références sont égales si elles désignent le même objet.

En Java la comparaison des références se note '**`==`**'.



Il faut bien remarquer qu'une telle comparaison rend ***vrai*** si et seulement si les deux références sont les mêmes, c'est-à-dire désignent le même objet. Elle rend faux si les références désignent des objets différents, même s'ils ont même valeur. Par exemple, avec la situation illustrée ci-dessus :

`mistigri==carte1` vaut **`true`**
`mistigri==carte2` vaut **`false`**

bien que les objets désignés par **`mistigri`** et **`carte2`** aient la même valeur. L'opérateur **`==`** sur les objets ne compare pas les "valeurs" mais seulement les références. Si on veut comparer les "valeurs", il faut le programmer, le langage ne l'offre pas.

Voici comment programmer la comparaison d'égalité de valeur pour le type **`CarteAJouer`**, en considérant que deux cartes sont égales si et seulement si elles ont même force et même couleur. Cette comparaison, fréquente pour des structures, s'appelle une comparaison "champ par champ". Une convention très répandue est d'appeler **`equals`** la fonction de comparaison d'égalité de valeur entre objets.

```
static boolean equals(CarteAJouer c1, CarteAJouer c2) {
// résultat : indique si les c1 et c2 ont même valeur
   return (c1.force==c2.force) && (c1.couleur==c2.couleur);
}
```

7.2.5 Références en paramètre et en résultat

Un paramètre ou un résultat de type référence se note simplement en indiquant le type d'objet référencé. Prenons comme exemple une fonction **laGagnante** qui, étant données deux cartes à jouer rend en résultat la carte gagnante. Les paramètres sont deux références à des cartes et le résultat est également une référence à une carte. Le résultat est la référence à la carte la plus forte, selon la force des cartes, puis en cas d'égalité des forces, selon la couleur des cartes. En cas d'égalité des cartes, c'est-à-dire si les deux cartes ont même force et même couleur, le résultat est la référence **null**.

```
static CarteAJouer laGagnante(CarteAJouer c1,CarteAJouer c2) {
// résultat : la plus forte des cartes c1 et c2
// null si les cartes sont de forces égales
   if (c1.force.compareTo(c2.force)>0) {return c1;}
   else if (c1.force.compareTo(c2.force)<0) {return c2;}
   else /* forces égales, on compare les couleurs */ {
      if (c1.couleur.compareTo(c2.couleur)>0) {return c1;}
      else if (c1.couleur.compareTo(c2.couleur)<0) {return c2;}
      else /* carte égales */ {return null;}
   }
}
```

7.2.6 Exemple récapitulatif

Le programme suivant gère un jeu de bataille simplifié entre deux joueurs. À chaque coup chaque joueur fournit la force et la couleur de sa carte, en clair au clavier. Si une carte est gagnante, le joueur correspondant marque un point supplémentaire. En cas d'égalité des cartes, aucun joueur ne marque de point. La partie dure 5 coups et le score final est affiché.

```
class JeuDeBataille {

  static CarteAJouer laGagnante(CarteAJouer c1,CarteAJouer c2) {
  // résultat : la plus forte des cartes c1 et c2
  // null si les cartes sont de forces égales
    if (c1.force.compareTo(c2.force)>0) {return c1;}
    else if (c1.force.compareTo(c2.force)<0) {return c2;}
    else /* forces égales, on compare les couleurs */ {
      if (c1.couleur.compareTo(c2.couleur)>0) {return c1;}
      else if (c1.couleur.compareTo(c2.couleur)<0) {return c2;}
      else /* carte égales */ {return null;}
    }
  }
                                                          //...
```

```
   public static void main(String[] arg) {
      int pointsDuJoueur1=0;
      int pointsDuJoueur2=0;
      for(int i=0; i<5; i++) {
         System.out.print("joueur 1 : ");
         CarteAJouer carte1 = new CarteAJouer(
                        ForceDeCarte.valueOf(Lecture.chaine()),
                        CouleurDeCarte.valueOf(Lecture.chaine()));
         System.out.print("joueur 2 : ");
         CarteAJouer carte2 = new CarteAJouer(
                        ForceDeCarte.valueOf(Lecture.chaine()),
                        CouleurDeCarte.valueOf(Lecture.chaine()));
         CarteAJouer carteGagnante = laGagnante(carte1,carte2);
         if (carteGagnante==carte1) {
            System.out.println("le joueur 1 marque un point");
            pointsDuJoueur1++;
         }
         else if (carteGagnante==carte2) {
            System.out.println("le joueur 2 marque un point");
            pointsDuJoueur2++;
         }
         else /*carteGagnante==null*/ {
            System.out.println("personne ne marque");
         }
      }
      System.out.println("score final");
      System.out.print("joueur 1 :");
      System.out.print(pointsDuJoueur1);
      System.out.print(" joueur 2 :");
      System.out.print(pointsDuJoueur2);
      System.out.println();
   }
}
```

Remarque : dans ce programme, deux nouvelles cartes sont créées par l'instruction **new** à chaque étape du jeu. Cela peut sembler coûteux en utilisation de la mémoire. Il ne faut pas trop se tracasser de cela car le langage utilisé, Java, dispose d'un mécanisme de gestion de mémoire qui récupère la mémoire occupée par les objets qui ne servent plus, ce qui est le cas des anciennes cartes lorsqu'on en crée des nouvelles à chaque itération. Ce mécanisme de récupération de mémoire s'appelle un ***ramasse-miettes***, ou encore un "***garbage collector***" en anglais.

QCM 7.1

On souhaite définir une structure qui regroupe les caractéristiques de chaque employé d'une entreprise. Ces caractéristiques sont le ***nom***, le ***prénom***, et la ***nature du poste*** occupé. Les natures de poste sont ***technique***, ***administratif*** ou ***commercial***.

La nature du poste est une valeur du type énuméré ainsi défini :

```
enum NatureDePoste{technique, administratif, commercial}
```

❑ 1 - Une bonne définition de la structure envisagée est :

```
class Employe{
  public static String nom;
  public static String prenom;
  public static NatureDePoste emploi;
  public Employe(String n, String p, NatureDePoste e){
    nom=n; prenom=p; emploi=e;
  }
}
```

❑ 2 - Une bonne définition de la structure envisagée est :

```
class Employe{
  public String nom;
  public String prenom;
  public NatureDePoste emploi;
  public Employe(String n, String p, NatureDePoste e){
    nom=n; prenom=p; emploi=e;
  }
}
```

❑ 3 - Pour créer un objet de type **Employe** et capter sa référence par l'identificateur **durand**, il faut écrire :

```
Employe durand =
      new Employe("Durand","Alfred",NatureDePoste.commercial);
```

❑ 4 - Pour créer un objet de type **Employe** et capter sa référence par l'identificateur **durand**, il faut écrire :

```
Employe("Durand","Alfred",NatureDePoste.commercial) durand;
```

❑ 5- Avec la (bonne) définition de la classe **Employe**, et la (bonne) création de **durand**, le nom de **durand** s'obtient par l'expression :

```
durand.nom
```

❑ 6- Avec la (bonne) définition de la classe **Employe**, il est impossible de modifier la nature du poste d'un employé.

❑ 7 - Avec la (bonne) définition de la classe **Employe**, on peut modifier la nature du poste d'un employé. Pour attribuer à durand la nature de poste ***administratif***, il suffit d'exécuter :

```
durand=Employe(administratif);
```

❑ 8 - Avec la (bonne) définition de la classe **Employe**, on peut modifier la nature du poste d'un employé. Pour attribuer à durand la nature de poste ***administratif***, il suffit d'exécuter :

```
durand.emploi=NatureDePoste.administratif;
```

QCM 7.2

La structure suivante décrit les caractéristiques d'un produit vendu dans un magasin :

```
class DescriptifProduit{// descriptif d'un produit
  public String nomProduit;
  public String nomFournisseur;
  public double prixDAchat;
  public double tauxTVA;
  public double margeBeneficiaire;
  public DescriptifProduit(String nP,String nF,
                           double pA,double tva,double m){ ...
  }
}
```

❏ 1 - Pour créer le descriptif d'une lessive, on peut écrire :

```
new DescriptifProduit("lessiveMachin","machin",12,19.6,0.2)
```

❏ 2 - Pour créer le descriptif d'une lessive, on peut écrire :

```
DescriptifProduit
    lessiveMachin("lessiveMachin","machin",12,19.6,0.2)
```

❏ 3 - Pour créer le descriptif d'une lessive et capter sa référence par la variable **lessiveMachin** on peut écrire :

```
DescriptifProduit
("lessiveMachin","machin",12,19.6,0.2) lessiveMachin;
```

❏ 4 - Pour créer le descriptif d'une lessive et capter sa référence par la variable **lessiveMachin** on peut écrire :

```
DescriptifProduit lessiveMachin =
     new DescriptifProduit("lessiveMachin","machin",12,19.6,0.2);
```

❏ 5 - Le nom du fournisseur de **lessiveMachin** s'obtient par l'expression :

```
nomFournisseur(lessiveMachin)
```

❏ 6 - Le nom du fournisseur de **lessiveMachin** s'obtient par l'expression :

```
lessiveMachin.nomFournisseur
```

❏ 7 - la fonction suivante :

```
static double prixDeVente(DescriptifProduit d){
  double prixHorsTaxe = prixDAchat*(1+margeBeneficiaire);
  return prixHorsTaxe*(1+tauxTVA);
}
```

est syntaxiquement correcte et rend en résultat le prix de vente du produit décrit par **d**.

❏ 8 - la fonction suivante :

```
static double prixDeVente(DescriptifProduit d){
  double prixHorsTaxe = d.prixDAchat*(1+d.margeBeneficiaire);
  return prixHorsTaxe*(1+d.tauxTVA);
}
```

est syntaxiquement correcte et rend en résultat le prix de vente du produit décrit par **d**.

Exercice 7.1 Représentation des points du plan au moyen d'une structure

objectif : *savoir utiliser les champs d'une structure et créer un résultat de type structure.*

Un point du plan peut être représenté par ses coordonnées cartésiennes (x, y) dans un repère arbitraire. Définir une telle structure **`Point`** et rédiger les fonctions suivantes :

- Distance entre deux points :

```
static double distance(Point a, Point b)
// résultat : la distance entre a et b
```

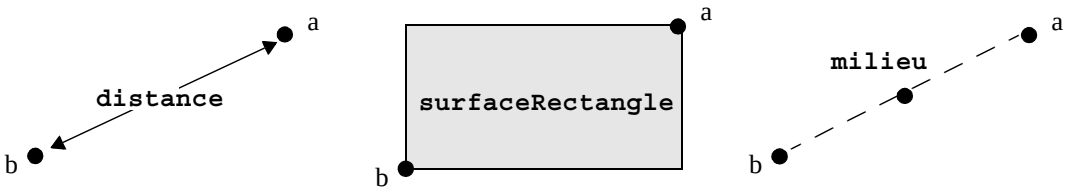
- Surface du rectangle défini par deux coins :

```
static double surfaceRectangle(Point a, Point b)
// résultat : la surface du rectangle défini par les deux coins
// diagonalement opposés a et b
```

- Point milieu de deux points :

```
static Point milieu(Point a, Point b)
// résultat : le point milieu entre a et b
```

Tester ces fonctions sur les points **`a`**=(5,4) et **`b`**=(1,2).



Vérifier qu'on a bien les égalités :

`distance(a,b)` = 2×**`distance(a,milieu(a,b))`**

`surfaceRectangle(a,b)` = 4×**`surfaceRectangle(a,milieu(a,b))`**

Exercice 7.2 Représentation des nombres complexes au moyen d'une structure

objectif : *savoir utiliser les champs d'une structure, créer un résultat de type structure, définir une constante de type structure.*

Un nombre complexe peut être représenté au moyen d'une structure à deux champs, **`x`** (partie réelle) et **`y`** (partie imaginaire).

Définir cette représentation au moyen d'une classe **`Complexe`**.

- Rédiger la fonction **`add`** qui rend en résultat la somme de deux complexes.
- Rédiger la fonction **`mul`** qui rend en résultat le produit de deux complexes.
- Rédiger une fonction

  `String toString(Complexe z)`

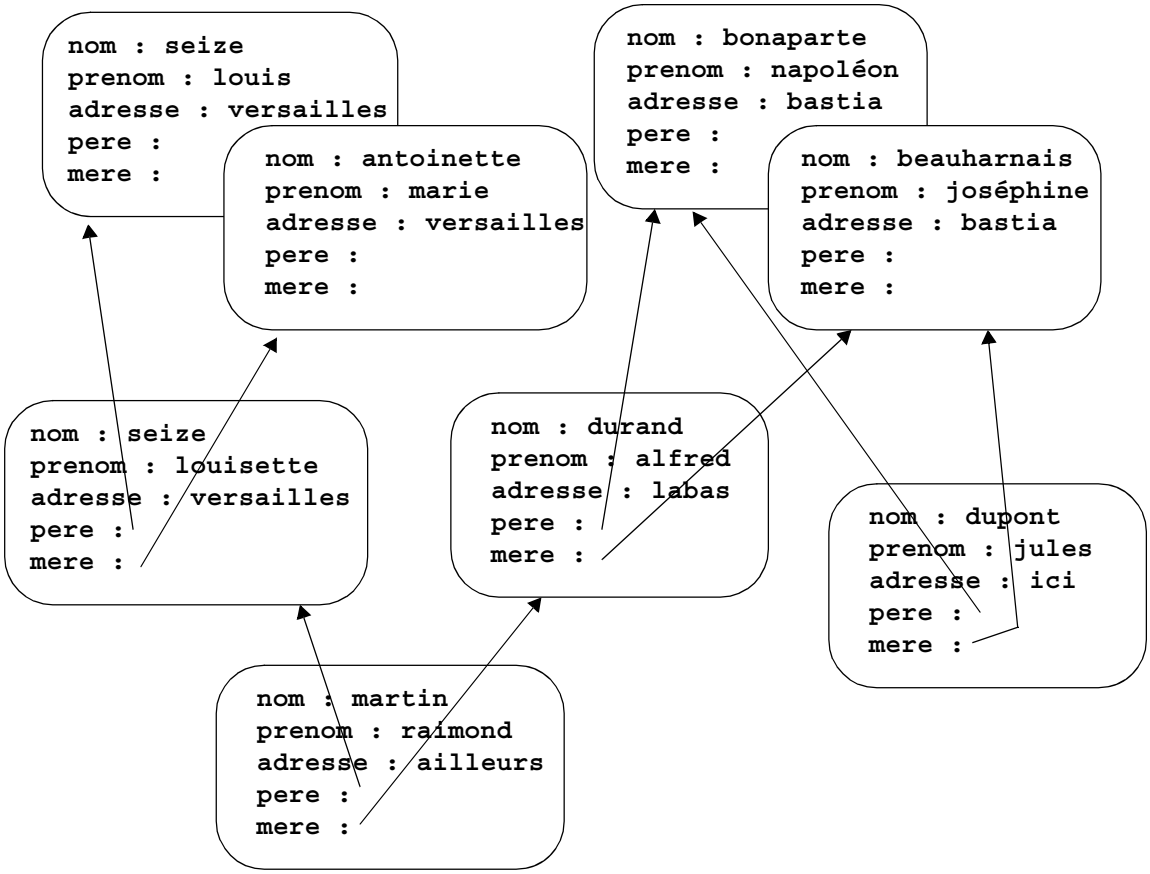
  qui rend en résultat la chaîne de caractères de la forme "*x*`+`*y*`.i`" qui dénote le nombre complexe **`z`** sous la forme usuelle. On fera en sorte de rendre simplement "*x*" pour "*x*`+0.i`", "*y*`.i`" pour "`0+`*y*`.i`", "*x*`+i`" pour "*x*`+1.i`", "`i`" pour "`0+i`" et "`0`" pour "`0+0.i`".
- Rédiger une procédure principale qui lit deux réels x et y, crée le nombre complexe *z*=x+y.***i*** et calcule et affiche le nombre complexe *z.(z.(z*+1*)*+1*)*. Remarque : il faudra définir le complexe "1" (alias "`1+0.i`") pour le faire participer aux opérations.

Exercice 7.3 Représentation des liens de parenté entre personnes

objectif : *comprendre la sémantique et l'intérêt de la désignation par référence.*

On désire représenter des liens de parenté entre personnes. Une personne possède un ***nom***, un ***prénom***, une ***adresse***, un ***père*** et une ***mère***. Le nom, le prénom et l'adresse sont des chaînes de caractères. Le père et la mère sont des personnes.

Le type **`Personne`** qui représente les personnes est ce qu'on appelle un ***type récursif*** : sa définition s'utilise elle-même. La désignation par référence permet de réaliser ceci facilement. Un objet de type **`Personne`** possédera deux champs, **`pere`** et **`mere`**, qui seront des références aux objets **`Personne`** qui représentent le père et la mère. Le schéma suivant illustre une collection de personnes avec leurs liens de parenté :



1 - Rédiger la classe **`Personne`** qui représente de telles personnes. On prévoira deux constructeurs :

- un constructeur avec les 5 paramètres correspondant aux 5 attributs d'une personne,
- un constructeur avec seulement les 3 paramètres nom, prénom et adresse, pour créer une personne sans père ni mère (ou dont on ignore qui est le père et la mère : c'est les cas de "louis seize", "marie antoinette", "napoléon bonaparte" et "joséphine beauharnais" dans l'exemple).

2 - Rédiger les fonctions :

```
static boolean sontDesFreres(Personne p1, Personne p2)
// résultat : indique si p1 et p2 sont frères
// (ont même père et même mère)

static boolean p1EstGrandPereDep2(Personne p1, Personne p2)
// résultat : indique si p1 est grand-père de p2
// (est le pere du pere ou de la mere)
```

3 - Rédiger une procédure principale qui :

- crée le réseau de personnes illustré précédemment,
- teste si “jules dupont” et “alfred durant” sont frères, teste si “jules dupont” et “louis seize” sont frères, teste si “louis seize” est un grand-père de “raimond martin” et teste si “napoléon bonaparte” est un grand-père de “jules dupont”,
- affiche l’adresse de “napoléon bonaparte”,
- modifie l’adresse du grand-père paternel de “raimond martin” et affiche l’adresse de “napoléon bonaparte” (prévoir le résultat de cet affichage).

Aide 7.1 Représentation des points du plan au moyen d'une structure

La représentation d'un point possède deux composantes, l'abscisse **x** et l'ordonnée **y**. L'abscisse d'un points **a** s'obtient par **a.x** et son ordonnées par **a.y**.

La distance entre deux points ***a*** et ***b*** de coordonnées respectives ***(a.x, a.y)*** et ***(b.x, b.y)*** est donnée par la formule de Pythagore-Euclide :

$$\sqrt{(a \cdot x - b \cdot x)^2 + (a \cdot y - b \cdot y)^2}$$

Les points étant représentés par une structure **Point** dotée d'un constructeur habituel, pour créer un point de coordonnées ***(5,4)***, il faut exécuter la déclaration/instruction :

```
Point a = new Point(5,4);
```

La création est également sollicitée dans la fonction **milieu**, pour créer la représentation du point milieu.

Aide 7.2 Représentation des nombres complexes au moyen d'une structure

Les nombres complexes seront représentés par une structure **Complexe** à 2 champs, **x** et **y**, dotée d'un constructeur usuel. Une telle structure représente le nombre complexe **x**+**y.*i***.

Les fonctions **add**, **mul** et **toString** seront rédigées dans une classe **TestStructureComplexe** dont le main est le programme de test. Voici sa forme générale :

```
import es.*;

class TestStructureComplexe {

  static Complexe add(Complexe z1, Complexe z2) {
  // résultat : somme de z1 et z2
    ...
   }

  static Complexe mul(Complexe z1, Complexe z2) {
  // résultat : produit de z1 et z2
    ...
  }

  static String toString(Complexe z){
  // résultat : la chaîne de caractère qui figure z en clair
    ...
  }

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

Les fonctions **`add`** et **`mul`** nécessitent la création du résultat. Ces création utilisent le constructeur, en appliquant les formules classiques de l'addition et de la multiplication des nombres complexes :

pour l'addition : $(x_1+y_1.i) +(x_2+y_2.i) = x_1+x_2 + (y_1+y_2).i$

pour la multiplication : $(x_1+y_1.i)\times(x_2+y_2.i) = x_1.x_2 - y_1.y_2 +(x_1.y_2 + y_1.x_2).i$

La fonction d'addition par exemple est de la forme :

```
static Complexe add(Complexe z1, Complexe z2) {
// résultat : somme de z1 et z2
  ...
}
```

Ne pas oublier que les composantes de **`z1`**, par exemple, s'obtiennent par **`z1.x`** et **`z1.y`**.

Pour la fonction **`toString`**, il est conseillé dans un premier temps d'en faire une version simple qui rend en résultat une chaîne de caractères de la forme unique **`x+y*i`**, quelques soient **`x`** et **`y`**, nuls ou non, valant 1 ou non... Cette fonction est importante pour visualiser le résultat des tests.

Dans la procédure principale de test, nous devons utiliser le nombre complexe "1", qui n'est autre que 1+0.*i*. Nous devons définir pour cela la constante complexe **`UN`** :

```
final Complexe UN = new Complexe(1,0);
```

Aide 7.3 Représentation des liens de parenté entre personnes

```
class Personne {
  public String nom;
  public String prenom;
  public String adresse;
  public Personne pere;
  public Personne mere;

  public Personne(String n, String p, String a,
                  Personne sonPere, Personne saMere){
  // personne de nom n, prenom p, adresse a,
  // père sonPere et mère saMere
    nom=n; prenom=p; adresse=a; pere=sonPere; mere=saMere;
  }

  public Personne(String n, String p, String a){
  // personne de nom n, prenom p, adresse a, sans père ni mère
    nom=n; prenom=p; adresse=a; pere=null; mere=null;
  }
}
```

Les champs **`pere`** et **`mere`** sont déclaré de type **`Personne`**, ce qui, en Java, en fait des ***références*** à des objets de type **`Personne`**.

Le constructeur pour des personnes sans père ni mère initialise les champs **`pere`** et **`mere`** à **`null`**, référence qui ne désigne rien.

Les fonctions **sontDesFreres** et **p1EstGrandPereDep2** sont à rédiger dans la classe **TestParents** qui contient également la procédure principale de test. Voici sa forme générale :

```
class TestParents {

  static boolean sontDesFreres(Personne p1, Personne p2){
  // résultat : indique si p1 et p2 sont frères
  // (ont même père et même mère)
    ...
  }

  static boolean p1EstGrandPereDep2(Personne p1, Personne p2){
  // résultat : indique si p1 est grand-père de p2
  // (est le père du père ou de la mère)
    ....
  }

  public static void main(String[] z){
    Personne louisSeize =
             new Personne("seize","louis","versailles");
    ...
  }
}
```

Pour les fonctions **sontDesFreres** et **p1EstGrandPereDep2** il faut cependant se poser la question de quel est le résultat si certaines des personnes concernées n'ont pas de père ou de mère. Il semble naturel de considérer que pour avoir un frère, il faut avoir des parents, et pour avoir un grand père, il faut avoir un père ou une mère.

CHAPITRE 8 Chaînes de caractères

objectif : *connaître les principales fonctionnalités offertes par les chaînes de caractères : comparer, rechercher, construire des chaînes de caractères, accéder aux caractères d'une chaîne.*

8.1 Type chaîne de caractères

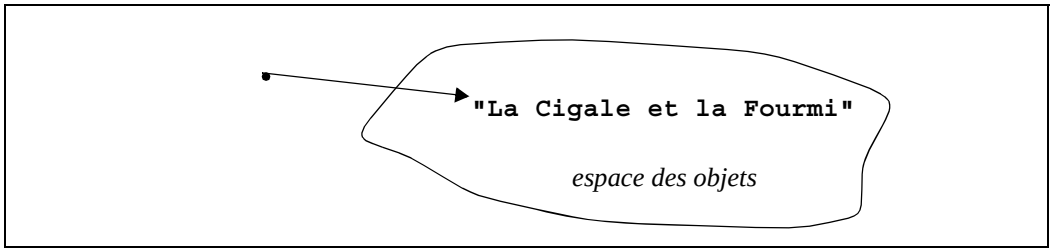
Le type ***chaîne de caractères*** permet de représenter des séquences de caractères de longueur quelconque. Les chaînes de caractères servent notamment à constituer des textes pour communiquer avec les utilisateurs des programmes.

En Java, les chaînes de caractères sont réalisées par le type **`String`**. Chaque chaîne de caractères est un ***objet***. Ceci implique notamment que les chaînes sont ***toujours désignées par référence***.

Une notation de valeur de chaîne se note entre doubles apostrophes :

```
"La Cigale et la Fourmi"
```

Une telle notation provoque la création, en mémoire, d'un objet qui vaut cette chaîne et elle rend en résultat la référence à cet objet (l'adresse de cet objet).

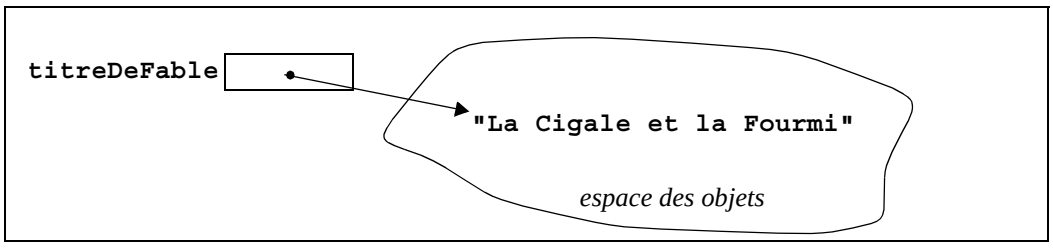


Par ailleurs, une déclaration de variable de type chaîne s'écrit :

```
String titreDeFable;
```

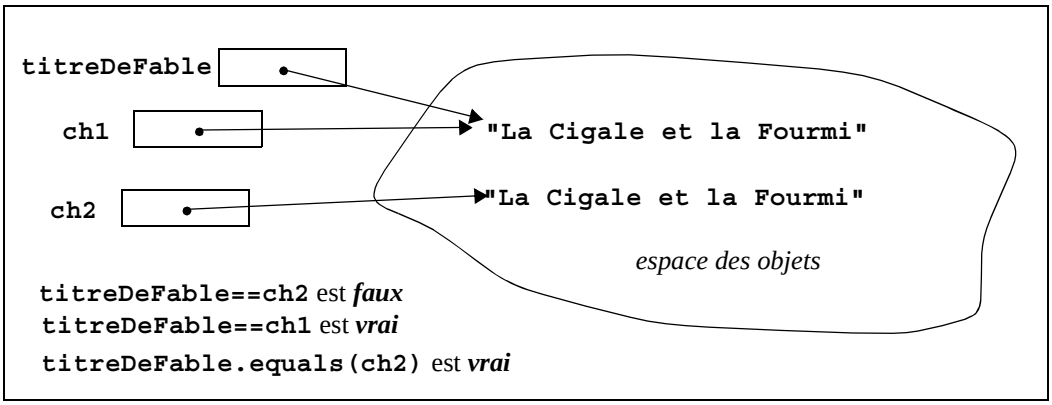
On peut alors affecter à une telle variable la référence à un objet chaîne de caractères, par exemple :

```
titreDeFable = "La Cigale et la Fourmi";
```

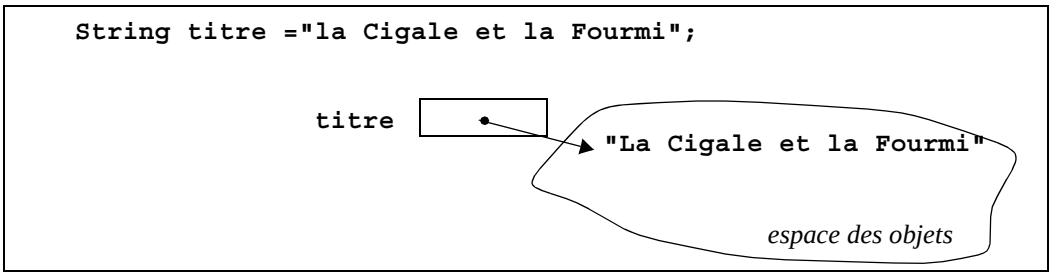


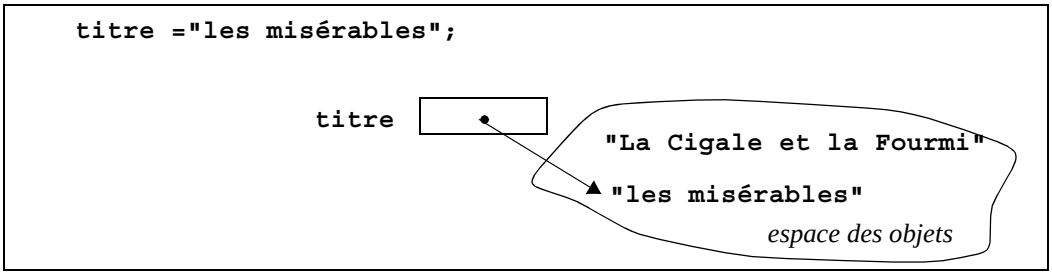
Le fait que les chaînes soient des objets, donc désignés par référence, oblige à prendre certaines précautions pour comparer des chaînes. L'opérateur de comparaison "`==`" compare les références et non les valeurs. Dans l'exemple suivant, `titreDeFable` et `ch2` désignent des chaînes de même valeur `"La Cigale et la Fourmi"`. Cependant "`titreDeFable==ch2`" est ***faux***, car `titreDeFable` et `ch2` désignent des objets différents. Pour comparer la valeur de deux chaînes, Java fournit la fonction `.equals()`. Ici `titreDeFable.equals(ch2)` rend ***vrai***.

Ce point est une source fréquente d'erreur, car "`titreDeFable==ch2`" n'est pas une faute de syntaxe, c'est officiellement la comparaison des références (qui est utile, mais dont l'utilité est extrêmement rare dans le cas de chaînes de caractères[1]).



Les chaîne de caractères de type `String` sont des objets ***non modifiables***. Une fois créée, un objet de type String aura toujours la même valeur. Ceci n'empêche pas, bien évidemment, d'avoir des variables de type `String`, mais ce qui varie c'est leur valeur qui est une référence. Lorsqu'on affect un identificateur de type `String`, on lui fait simplement désigner une autre chaîne :





1. Ce qui est critiquable ici ce n'est pas la notion de référence qui est inséparable de la notion d'objet, mais c'est le choix que les chaînes de type `String` soient des objets, alors que de toute évidence on voudrait que ce soient des valeurs.

8.2 Quelques opérations disponibles sur les chaînes de caractères

Chaîne vide :

La chaîne vide est la chaîne de longueur nulle. Elle ne contient aucun caractère. Elle se note **`""`**.

Concaténation :

***s*₁ `+` *s*₂** : concaténation de ***s*₁** et ***s*₂**.

Par exemple **`"La Cigale " + "et la Fourmi"`** rend en résultat un nouvel objet chaîne dont la valeur est **`"La Cigale et la Fourmi"`** (plus exactement, crée un nouvel objet chaîne et rend en résultat la référence à cet objet).

Autre exemple : avec **`titreDeFable`** désignant la chaîne **`"La Cigale et la Fourmi"`** :

```
titreDeFable + " est une fable de La Fontaine"
```

désigne une chaîne de valeur :

```
"La Cigale et la Fourmi est une fable de La Fontaine".
```

Longueur :

s*`.length()`** : nombre de caractères de la chaîne ***s. C'est une valeur entière de type **`int`**.

Par exemple **`"coucou".length()`** vaut 6, **`"".length()`** vaut 0,

et avec "**`String ch="coucou";`**" **`ch.length()`** vaut 6.

Le i^{ème} caractère :

s*`.charAt(`*i*`)`** : rend en résultat la valeur du ***i^{ème} caractère de ***s***. Les caractères sont numérotés à partir de 0 en commençant par la gauche. ***i*** est une expression entière dont la valeur doit être comprise entre 0 et **`s.length()-1`** (bornes incluses).

Exemples :

`"La Cigale et la Fourmi".charAt(0)` vaut **`'L'`**
`"La Cigale et la Fourmi".charAt(4)` vaut **`'i'`**
avec "**`String ch="coucou";`**" **`ch.charAt(2)`** vaut **`'u'`**

Comparaison selon l'ordre alphabétique :

s1*`.compareTo(`*s2*`)`** : rend en résultat un entier <0 si ***s1 est "avant" ***s2*** par ordre alphabétique, 0 si ***s1*** est égale à ***s2*** et >0 si ***s1*** est "après" ***s2*** par ordre alphabétique.

Sous-chaîne

s*.substing(*i*,*j*)** : rend en résultat la sous-chaîne de ***s comprise entre l'indice ***i*** (compris) et l'indice ***j*** (exclu). ***j*** doit être supérieur ou égal à ***i***.

Exemple :

"bonjour".substring(2,6) vaut **"njou"**

"bonjour".substring(5,5) vaut **""** (chaîne vide)

Position d'un caractère dans une chaîne

s*.indexOf(*c*)** : rend en résultat si ***c apparaît dans ***s***, l'indice de sa première occurrence, et rend -1 si ***c*** n'apparaît pas dans ***s***.

Exemples :

"bonjour".indexOf('j') vaut 3

"bonjour".indexOf('d') vaut -1

Position d'une sous-chaîne dans une chaîne

s1*.indexOf(*s2*)** : rend en résultat si ***s2 apparaît dans ***s1***, l'indice de sa première occurrence, et rend -1 si ***s2*** n'apparaît pas dans ***s1***.

s1*.indexOf(*s2*,*i*)** : rend en résultat si ***s2 apparaît dans ***s1*** à partir de la position i, l'indice de sa première occurrence à partir de la position ***i***, et rend -1 si ***s2*** n'apparaît pas dans ***s1*** à partir de la position ***i***.

Exemples :

"bonjour".indexOf("jo") vaut 3

"bonjour bonsoir".indexOf("bon") vaut 0 (position du **"bon"** de **"bonjour"**)

"bonjour bonsoir".indexOf("bon",2) vaut 8 (position du **"bon"** de **"bonsoir"**)

Conversions en chaînes de caractères

Les chaînes de caractères servent essentiellement à communiquer "en clair" de l'information avec des utilisateurs humains. C'est pourquoi tous les types de base possèdent une conversion "standard" en chaîne de caractères. Par exemple, la conversion de l'entier **12** est **"12"**, celle du réel **3.14** est **"3.14"**, celle du booléen **false** est **"false"** et celle du caractère **'z'** est **"z"**.

L'opérateur de concaténation, noté '**+**', admet comme opérande n'importe quel type de base, à condition qu'un de ses deux arguments soit explicitement une chaîne. Dans ce cas, le compilateur réalise implicitement la conversion standard de cet opérande en chaîne avant de faire la concaténation.

Exemples :

"toto a " + 12 + " ans" signifie la même chose que **"toto a " + "12" + " ans"** c'est-à-dire **"toto a 12 ans"**.

Ceci est très utilisé pour afficher joliment un résultat. Par exemple, **ageDeToto** étant un entier, plutôt que de programmer :

```
System.out.print("toto a ");
System.out.print(ageDeToto);
System.out.println(" ans");
```

Il est plus concis de programmer :

```
System.out.println("toto a " + ageDeToto + " ans");
```

8.3 Exemple de manipulations de chaînes de caractères

Nous prendrons comme exemple la mise en lettres majuscules d'un texte composé de caractères quelconques. Il s'agit de fabriquer un nouveau texte similaire, mais où toutes les lettres ont été remplacées par la lettre majuscule correspondante. Exemple :

La mise en lettres majuscules de :

`"les 24 heures du Mans"`

est :

`"LES 24 HEURES DU MANS"`

Principe :

Nous allons rédiger une fonction

`static String majuscule(String txt)`

dont le résultat est le texte résultant du remplacement dans **`txt`** des lettres minuscules par les lettres majuscules correspondantes.

Le résultat est une chaîne de caractère ***nouvelle***, qu'il faut donc créer dans le corps de cette fonction. On la construit caractère par caractère, à l'aide de la concaténation. Ce résultat, **`resul`**, est initialisé à la chaîne vide et pour chaque caractère de **`txt`** on lui concatène le caractère majuscule correspondant.

Ceci fait naître le besoin d'une fonction auxiliaire :

`static char majuscule(char c)`

qui rend en résultat la majuscule correspondant au caractère **`c`**. Nous convenons que le résultat de cette fonction est **`c`** lui même si **`c`** n'est pas une lettre minuscule.

Pour la transformation minuscule/majuscule, une technique efficace et pratique consiste à utiliser deux chaînes de caractères :

la chaîne **`minuscule`** : **`"aàbcçdeéèêfghiïjklmnoôpqrstuùûüvwxyz"`**

et la chaîne **`majuscule`**, qui contient en chaque position ***k*** la majuscule correspondant à la minuscule de position ***k*** : **`"AABCCDEEEEFGHIIJKLMNOOPQRSTUUUUVWXYZ"`**

Par soucis de généralité nous avons prévu les lettres accentuées (du Français) ainsi que le "ç". Pour simplifier, avons décidé que la majuscule correspondante serait sans accent ni cédille :

```
class TestTexteMajuscule{

  static final String
              minuscules = "aàbcçdeéèêfghiïjklmnoôpqrstuùûüvwxyz";
  static final String
              majuscules = "AABCCDEEEEFGHIIJKLMNOOPQRSTUUUUVWXYZ";

  static char majuscule(char c){
  // résultat : la majuscule correspondant à c
  // (le résultat est c si c n'est pas une lettre minuscule)
    int k=minuscules.indexOf(c);
    if(k==-1){return c;}
    else{return majuscules.charAt(k);}
  }

  static String texteMajuscule(String txt){
  // résultat : texte résultant de txt par remplacement de toute
  // lettre par la majuscule correspondante
    String resul="";
    for(int i=0;i<txt.length(); i++){
      resul=resul+majuscule(txt.charAt(i));
    }
    return resul;
  }

  public static void main(String[] z){
    System.out.println(texteMajuscule("les 24 heures du Mans"));
  }
}
```

QCM 8.1

On considère les déclaration et créations suivantes de chaînes de caractères :

```
String s1 = "tant va la";
String s2 = "cruche à l'eau";
String s3 = s1+s2;
String s4 = s1+" "+s2;
String s5 = s1+""+s2;
```

❑ 1 - L'expression **`s3=="tant va la cruche à l'eau"`** vaut **`true`**.

❑ 2 - L'expression **`s4=="tant va la cruche à l'eau"`** vaut **`true`**.

❑ 3 - L'expression **`s5=="tant va la cruche à l'eau"`** vaut **`true`**.

❑ 4 - L'expression **`s3.equals("tant va la cruche à l'eau")`** vaut **`true`**.

❑ 5 - L'expression **`s4.equals("tant va la cruche à l'eau")`** vaut **`true`**.

❑ 6 - L'expression **`s5.equals("tant va la cruche à l'eau")`** vaut **`true`**.

❑ 7 - L'expression **`s1.substring(0,3)=="tan"`** vaut **`true`**.

❑ 8 - L'expression **`s1.substring(0,3)=="tant"`** vaut **`true`**.

❑ 9 - L'expression **`s1.substring(0,3).equals("tan")`** vaut **`true`**.

❑ 10 - L'expression **`s1.substring(0,3).equals("tant")`** vaut **`true`**.

QCM 8.2

Soit la déclaration et création de chaîne :

```
String s = "abcd";
```

❑ 1 - La longueur de **`s`** est donnée par l'expression **`s.length`**.

❑ 2 - La longueur de **`s`** est donnée par l'expression **`s.length()`**.

❑ 3 - La longueur de **`s`** est donnée par l'expression **`length(s)`**.

❑ 4 - L'expression **`s.charAt(2)`** vaut le caractère **`'b'`**.

❑ 5 - L'expression **`s.charAt(2)`** vaut le caractère **`'c'`**.

❑ 6 - L'objet chaîne de caractère désigné par **`s`** est modifiable.

❑ 7 - L'instruction **`s.charAt(2)='x';`** transforme la chaîne désignée par **`s`** en **`"axcd"`**.

❑ 8 - L'instruction **`s.charAt(2)='x';`** transforme la chaîne désignée par **`s`** en **`"abxd"`**.

❑ 9 - **`s.charAt(2)='x';`** est une erreur de syntaxe.

❑ 10 - L'instruction **`s=s.substring(0,2)+'x'+s.substring(3,4);`** crée la chaîne **`"axcd"`** et la fait désigner par **`s`**.

❑ 11 - L'instruction **`s=s.substring(0,2)+'x'+s.substring(3,4);`** crée la chaîne **`"abxd"`** et la fait désigner par **`s`**.

❑ 12 - **`s=s.substring(0,2)+'x'+s.substring(3,4);`** est une erreur de syntaxe.

❑ 13 - **`s=s.substring(0,2)+'x'+s.substring(3,4);`** provoque une erreur à l'exécution.

Exercice 8.1 Comptage d'un caractère dans une chaîne

objectif : *savoir parcourir une chaîne de caractères et accéder à ses caractères.*

Rédiger et tester la fonction :

```
static int nombreDe(char c, String s)
```

qui rend en résultat le nombre d'occurrences du caractère **c** dans la chaîne **s**.

Exemples :

```
nombreDe('a',"toto") = 0
nombreDe('o',"toto") = 2
```

Exercice 8.2 Recherche d'une sous-chaîne

objectif : *savoir comparer les caractères de deux chaînes, inventer une fonction auxiliaire, se méfier des dépassement d'indices au sein d'une chaîne.*

Sans utiliser la fonction **indexOf**, rédiger et tester la fonction :

```
static int indiceDeSousChaine(String s1, String s2)
```

qui, si **s1** est une sous-chaîne de **s2**, rend en résultat l'indice de la première occurrence de **s1** dans **s2**, sinon -1.

Exemples :

```
indiceDeSousChaine("toto","0+0 = la tête à toto") = 16
indiceDeSousChaine("zozo","0+0 = la tête à toto") = -1
indiceDeSousChaine("0 =","0+0 = la tête à toto") = 2
indiceDeSousChaine("","0+0 = la tête à toto") = 0
```

Remarque : penser à rédiger une fonction auxiliaire.

Exercice 8.3 Interprétation d'une chaîne de chiffres décimaux en un entier

objectif : *savoir utiliser une chaîne auxiliaire comme réservoir d'information (ici, la suite des chiffres* `"0123456789"`*)*

Le but de cet exercice est de réaliser la conversion traditionnelle d'une chaîne caractères composée uniquement de chiffre décimaux en l'entier représenté en décimal par cette chaîne. rédiger et tester la fonction :

```
static int interpretationDecimale(String s)
// prérequis : s ne contient que des chiffres décimaux
// et l'interprétation décimale de s est < 2**31
// résultat : l'interprétation décimale de s
```

Le prérequis "interprétation décimale de **s** < 2^{31}" sert à garantir que le résultat soit représentable sur un **int**.

Exemple :

interpretationDecimale("1703") vaut 1703

On conviendra que l'interprétation décimale de la chaîne vide (aucun chiffres) vaut 0 (convention raisonnable et pratique).

Aide 8.1 Comptage d'un caractère dans une chaîne

Il suffit de parcourir la chaîne de caractère, au moyen d'une boucle

```
for (int i;i<s.length;i++)
```

pour incrémenter un compteur (**resul**) chaque fois que le caractère en position **i** est égal à **c**.

Aide 8.2 Recherche d'une sous-chaîne

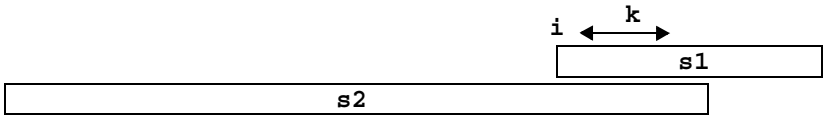
L'idée générale consiste à chercher pour chaque position **i** de **s2**, la chaîne **s1** est présente à partir de cette position. Cela fait apparaître le besoin d'une fonction auxiliaire :

```
static boolean occurrence(int i, String s1, String s2)
```

qui indique si **s1** est présente en position **i** dans **s2**.

Cette fonction compare les caractères de **s1** aux caractères de **s2** à partir de la position **i**. Ceci est réalisé par une itération qui utilise **k** comme indice de caractère dans **s1** et **i+k** comme indice de caractère dans **s2**. La boucle s'arrête à la première différence rencontrée. En sortie de boucle, **s1** est dans **s2** si et seulement si tous les caractères de **s1** ont été comparés avec succès, c'est-à-dire si **k** est égal à la longueur de **s1**.

Attention : un indice de caractère dans une chaîne doit être positif ou nul et strictement inférieurs à la longueur de la chaîne. Se méfier des cas où à partir de la position i les caractères de **s2** sont en nombre inférieur à la taille de **s1** :



Aide 8.3 Interprétation d'une chaîne de chiffres décimaux en un entier

L'interprétation décimale de la chaîne de chiffres $c_n c_{n-1}...c_1 c_0$ est :

$$[c_n].10^n+[c_{n-1}].10^{n-1}+...[c_1].10+[c_0]$$

dans cette formulation, $[c_n]$ dénote la valeur numérique attribuée au chiffre c. Cela fait apparaître le besoin d'obtenir la valeur numérique d'un chiffre décimal. Nous rédigeons donc une fonction auxiliaire :

```
static int valeurDecimale(char c){
// prérequis : c est un chiffre décimal
// résultat : la valeur numérique de c
```

Un moyen simple de réaliser cette fonction est d'utiliser l'indice du chiffre dans la chaîne **"0123456789"**. Par exemple l'indice de **'3'** dans cette chaîne est l'entier 3, justement ce que l'on désire.

Pour évaluer la somme $[c_n].10^n+[c_{n-1}].10^{n-1}+...[c_1].10+[c_0]$, on peut utiliser les puissances de 10. On peut également évaluer cette formule sous la forme :

$$(...([c_n].10+[c_{n-1}]).10+...[c_1]).10+[c_0]$$

(Cette forme d'évaluation d'un "polynôme" est connue sous l'appellation "schéma de Horner").

CHAPITRE 9 Tableaux

objectif : *savoir utiliser les tableaux pour organiser des collections de données*

9.1 Intérêt des tableaux

Un tableau regroupe une ***collection de données de même type***, chaque élément étant ***désigné à l'aide d'un indice***. Les indices sont les nombres entiers d'un intervalle.

La figure suivant illustre un tableau de 5 nombres réels. Les éléments sont indicés de 0 à 4.

T :		
	0	3.97
	1	1.23
	2	0.56
	3	1.23
	4	7.90

Si le tableau s'appelle ***T***, l'expression ***T[2]*** désigne la donnée d'indice 2 qui vaut 0.56.

L'usage d'un tableau est justifié dans les cas suivants :

- Lorsque la quantité de données est important : il n'est pas question de rédiger un programme qui définisse autant de noms individuels pour manipuler ces données.
- Lorsqu'on doit déterminer ***par calcul*** quels éléments de la collection sont utilisés.

Exemple 1

Un morceau de musique numérique est constitué d'échantillons d'amplitude du signal sonore. Il faut environ 20 000 échantillons par seconde d'enregistrement. Chaque échantillon étant un nombre entier, il faut 2 400 000 valeurs entières pour représenter 2 minutes de musique. Pour appliquer certains traitements à un tel enregistrement, on pourra le conserver dans un tableau :

tableau d'entiers [0...2 399 999] morceauDeMusique

Ce tableau est tel que ***morceauDeMusique[t]*** désigne l'échantillon de signal sonore mesuré à l'intervalle de temps numéro ***t***.

Exemple 2

Si on a fréquemment besoin d'obtenir le nom d'un mois à partir d'un numéro de mois, on pourra regrouper les noms de mois, des chaînes de caractères, dans un tableau de 12 éléments indicé de 1 à 12.

tableau de chaînes de caractères [1...12] nomDuMois

Avec un tel tableau, ***nomDuMois[3]*** désignera le nom du 3ème mois, et vaudra sans doute "***mars***".

L'exemple 1 est typique de l'usage d'un tableau pour une grosse quantité de données. Mais la fonction de correspondance $t \rightarrow$ ***morceauDeMusique[t]*** est également importante pour les traitements sur le son. Dans l'exemple 2 la collection de valeurs est relativement petite. Le tableau est utilisé

pour établir une correspondance entre entiers et éléments, $n \rightarrow$ ***nomDuMois[n]***.

9.2 Création et nommage des tableaux en Java

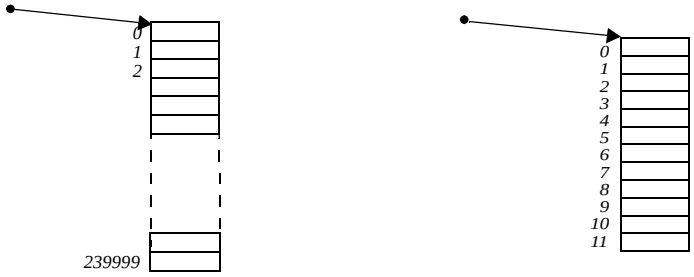
objectif : *savoir créer et déclarer des tableaux en Java. Comprendre les particularités des tableaux en Java : ils sont toujours créée par* **`new`** *et désignés par référence.*

En Java les tableaux sont obligatoirement créés par **`new`** et désignés par référence (ce n'est pas le cas dans d'autres langages).

`new` ***TypeDesEléments*** **`[`** ***taille*** **`]`**

crée un tableau de taille indiquée par ***taille*** dont les éléments sont de type ***TypeDesEléments***. Cette expression rend en résultat la référence au tableau créé.

Exemples : **`new int[240000]`** tableau de 240 000 entiers
`new String[12]` tableau de 12 chaînes de caractères



Par ailleurs, on peut déclarer des variables, constantes, paramètres ou résultats de fonctions de type "tableau", c'est-à-dire de type référence à un tableau.

La forme générale d'une déclaration de variable référence à tableau est :

TypeDesEléments **`[]`** ***nomDeTableau*** **`;`**

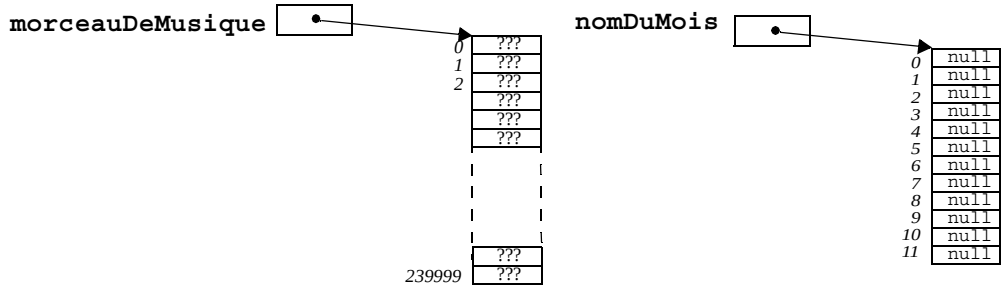
Exemples :

`int[] morceauDeMusique;` référence à un tableau d'entiers appelé **`morceauDeMusique`**
`String[] nomDuMois;` référence à un tableau de chaînes de caractères appelé **`nomDuMois`**

La déclaration de tableau n'indique pas la taille du tableau. C'est normal car la taille est décidée au moment de la création du tableau et figure donc dans l'expression de création **`new`** ***nomDeType*** **`[`** ***taille*** **`]`** . La déclaration seule ne crée pas de tableau. Dans les exemples ci-dessus, les

variables **morceauDeMusique** et **nomDuMois** sont initialisées à la référence **null** qui ne désigne rien. Le captage dans une variable de la référence à un tableau se fait par une affectation :

```
morceauDeMusique = new int[240000];
nomDuMois = new String[12];
```



Quand c'est possible, on a intérêt à regrouper création et déclaration :

```
int[] morceauDeMusique = new int[240000];
String[] nomDuMois = new String[12];
```

Les éléments d'un tableau en Java sont des ***variables*** du type indiqué pour les éléments : **int**, **char**, **double**, **boolean**... pour les types de base et ***référence*** pour les objets de type classe, chaînes de caractères et tableaux. Lorsque le tableau est créé, les valeurs des éléments ne sont pas initialisées dans le cas d'éléments d'un type de base, et sont initialisées à **null** dans le cas d'éléments de type référence.

La ***taille*** d'un tableau peut être obtenue au moyen de l'attribut **length** : la taille du tableau **T** s'obtient par l'expression **T.length**.

Remarque : la taille d'un tableau peut être égale à 0, et cela est utile dans certains cas particuliers.

9.3 Accès aux éléments de tableau

En Java, les tableaux sont toujours indicés à partir de 0. L'élément ***i*** d'un tableau ***tab*** se note :

tab **[** ***i*** **]**

i est une expression entière dont la valeur doit être comprise entre 0 et ***taille-1*** (bornes incluses). Si au moment de l'exécution ***i*** n'est pas dans cet intervalle, cela provoque une erreur "accès hors des bornes d'un tableau" ("Array Out Of Bound Exception").

L'indiciation à partir de 0 est rarement gênante. C'est par exemple "naturel" pour l'exemple **morceauDeMusique**, car on a coutume de prendre l'origine des temps à 0, donc **morceauDeMusique[t]** représente bien l'échantillon au temps **t**. C'est moins naturel pour l'exemple **nomDuMois** car on a l'habitude de numéroter les mois à partir de 1. Dans ce cas il faut appliquer une correction. Pour désigner le nom du $k^{\text{ième}}$ mois il faut utiliser l'expression :

nomDuMois[*k*-1] nom du $k^{\text{ième}}$ mois pour $k \in [1...12]$

Un élément de tableau, ***tab*** **[** ***i*** **]** se comporte comme une variable. Il peut donc être affecté :

tab **[** ***i*** **]** **=** ***expression*** **;**

Par exemple voici les affectations qui initialisent le tableau des noms de mois :

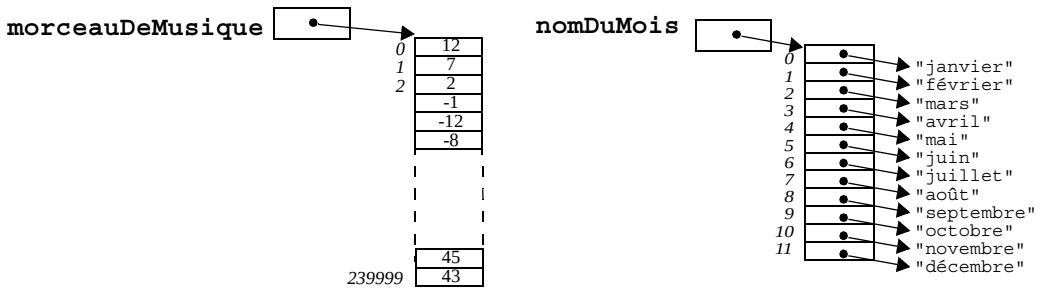
```
nomDuMois[0] = "janvier";
nomDuMois[1] = "février";
nomDuMois[2] = "mars";
...
nomDuMois[11] = "décembre";
```

Pour un tableau de taille importante comme le morceau de musique, l'initialisation se fait généralement au moyen d'une boucle qui parcourt tous les éléments au moyen d'une variable entière. Le tableau pourra être initialisé avec des valeurs lues dans un fichier :

```
for (int i=0; i<240000; i++) {
   morceauDeMusique[i] = fichierDeMusique.lireUnEntier();
}
```

Dans cet exemple, la fonction **`fichierDeMusique.lireUnEntier()`** est supposée lire à chaque appel une nouvelle valeur entière depuis le fichier **`fichierDeMusique`**.

Après ces initialisations, les tableaux contiennent enfin des valeurs exploitables :



9.4 Conséquences de la désignation par référence

Un conséquence importante de la désignation des tableaux par référence est la suivante : Lorsqu'une fonction admet un tableau en paramètre, c'est la référence au tableau paramètre effectif qui est transmise lors d'un appel (et non une copie de ce tableau). Si la fonction modifie des élément du tableau, c'est le tableau passé en paramètre qui est modifié. Ceci permet donc de programmer des ***procédures*** qui ***transforment*** des tableaux.

Exemple :

Considérons la procédure suivante, **`metAZero`**, qui met à zéro l'élément d'indice **`k`** d'un tableau de nombres réels.

```
static void metAZero(double[] T, int k){
// prérequis : k>=0 et k<T.length
// effet : met à zéro T[k]
    T[k]=0;
}
```

Si on l'utilise ainsi :

```
double[] T1 = {4,12,67,1};
metAZero(T1,2);
```

T1 se trouve modifié. Il vaut **{4,12,0,1}**

L'appel a eu un ***effet***. C'est d'ailleurs ce que dit la spécification de cette procédure.

Si on ne désire pas d'effet, mais obtenir un nouveau tableau à partir d'un tableau passé en paramètre, la fonction doit créer le nouveau tableau et ne pas modifier celui passé en paramètre. C'est le cas de la fonction suivante, **miseAZero**, qui est sans effet et rend en résultat un nouveau tableau obtenu à partir de **T** en remplaçant par 0 la valeur de l'élément **k**.

```
static double[] miseAZero(double[] T, int k){
// prérequis : k>=0 et k<T.length
// résultat : nouveau tableau ayant zéro en position k
// et les même valeurs que T dans les autres positions
  double[] resul = new double[T.length];
  for(int i=0; i<T.length; i++){
    resul[i]=T[i];
  }
  resul[k]=0;
  return resul;
}
```

Ici, nous avons créé au sein de la fonction un nouveau tableau **resul**, de même taille que **T**. Puis nous avons copié **T** dans resul (on appelle cela "cloner" le tableau **T**). Ensuite nous avons mis à 0 l'élément **k** de **resul**. Et enfin nous avons rendu **resul** en résultat.

9.5 Déclarations de tableaux initialisés

objectif : *savoir utiliser des tableaux initialisés comme jeu de test ou comme intermédiaire de codage.*

Java permet, lors de la déclaration d'un tableau, de créer le tableau et ses éléments. La forme générale est :

***typeDesEléments*`[]` *nomDeTableau* `=` {*e0, e2... ek*}`;`

Exemples :

```
int[] lesDixPremiersEntiers = {0,1,2,3,4,5,6,7,8,9};
char[] chiffresDecimaux = {'0','1','2','3','4','5',6','7','8','9'};
```

Ceci est très utile, notamment en deux circonstances :

- Pour fournir des ***tests reproductibles*** :

  C'est une très mauvaise idée que de tester des fonctions en introduisant les données par des lectures au clavier. Il est préférable de rédiger des programmes de test qui contiennent toutes les données dans la procédure principale. Ainsi les tests sont facilement reproductibles. Dans les cas de données qui sont des collections, un tableau initialisé convient parfaitement. Exemple : soit à tester une fonction **`max`** qui rend en résultat le maximum d'un tableau d'entiers.

```
class TestMax {

  static int max(int[] T){
  // prérequis : T.length > 0
  // (le maximun d'un domaine vide n'est pas défini)
  // résultat : le maximum des éléments de T
    int resul=T[0];
    for (int i=1; i<T.length; i++){
      if(T[i]>resul) {resul=T[i];}
    }
    return resul;
  }

  public static void main(String[] z){
  // programme de test de la fonction max
    int[]T1 = {12,56,4,89,-23,0,56};
    System.out.println("le max de T1 = " + max(T1));
    int[]T2 = {1,6,0,92};
    System.out.println("le max de T2 = " + max(T2));
  }
}
```

- Pour réaliser facilement des ***encodages*** ou ***décodages*** de données.

  Comme exemple on peut considérer la fonction suivante qui fournit le nom en clair d'un mois à partir de son numéro :

```
static String nomDuMois(int n){
// prérequis : n>=1 et n<=12
// résultat : le nom du mois numéro n
  final String[] mois = {
    "janvier","février","mars","avril","mai","juin","juillet",
    "août","septembre","octobre","novembre","décembre"};
  return mois[n-1];
}
```

9.6 Exemples d'utilisation de tableaux

Exemple 1

Le programme suivant utilise un tableau **population** pour mémoriser une collection de personnes. Une personne est décrite par la structure **Personne**, composée du nom, de l'adresse et de l'âge de la personne. Le programme utilise une boucle pour initialiser le tableau (population de 5 personnes), puis cherche et affiche les caractéristiques d'une personne de nom donné. Si aucune personne de ce nom ne figure dans le tableau, le programme affiche "aucune personne de ce nom".

```
class Personne {
  public String nom;
  public String adresse;
  public int age;
  public Personne(String sonNom, String sonAdresse, int sonAge) {
    nom=sonNom; adresse=sonAdresse; age=sonAge;
  }
}
```

```
enum Etat {nonDecide,absent,present}

class TestPopulation {
  public static void main(String[] arg) {
    Personne[] population = new Personne[5];
    for (int i=0; i<population.length; i++) {
      System.out.println("entrer une personne :");
      System.out.print("nom : "); String nom=Lecture.chaine();
      System.out.print("adresse : ");
      String adresse=Lecture.chaine();
      System.out.print("age : "); int age=Lecture.unEntier();
      population[i] = new Personne(nom, adresse, age);
    }
    System.out.print("nom de la personne cherchée : ");
    String nomCherche=Lecture.chaine();

    int etatRecherche=Etat.nonDecide;
    int i=0;
    while (etatRecherche==Etat.nonDecide) {
      if (i==population.length) {etatRecherche=Etat.absent;}
      else if (population[i].nom.equals(nomCherche)) {
        etatRecherche=Etat.present;
      }
      else {i++;}
    }
    if (etatRecherche==Etat.absent){
      System.out.println("aucune personne de ce nom");
    }
    else /*etatRecherche==Etat.present*/ {
      System.out.println("nom : " + population[i].nom
                         + " adresse : " + population[i].adresse
                         + " âge : " + population[i].age + " ans");
    }
  }
}
```

Exemple 2

On a 100 nombres entiers dans un fichier. On désire rechercher combien de nombres sont inférieurs ou égaux à la moyenne, et combien sont supérieurs. Il nous faut dans un premier temps calculer la moyenne des nombres, ce qui impose un parcours de toute la suite des nombres. Puis on doit la parcourir à nouveau pour compter ceux qui sont inférieurs ou égaux à la moyenne. La lecture d'un fichier est relativement longue. Pour éviter de lire deux fois le fichier on peut utiliser un tableau qui mémorise les nombres. Le comptage utilise alors le tableau.

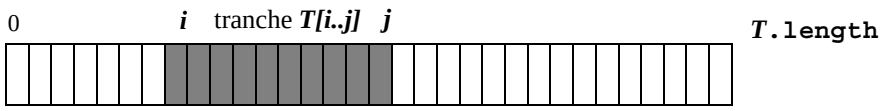
```
class ComptageDesInferieursALaMoyenne {

  public static void main(String[] arg) {

    LectureFichierTexte fichierDeNotes =
                 new LectureFichierTexte("notesDeMath.txt");
    final int nombreDeNotes=100;
    int[] note = new int[nombreDeNotes];

    // initialisation du tableau note
    // avec les valeurs lues dans le fichier de notes
    // et calcul de la moyenne des notes
    double sommeDesNotes=0;
    for (int i=0; i<nombreDeNotes; i++) {
      note[i] = fichierDeNotes.lireUnEntier();
      sommeDesNotes = sommeDesNotes+note[i];
    }
    double moyenneDesNotes = sommeDesNotes/nombreDeNotes;

    // comptage de nombre de notes inf. ou égales à la moyenne
    int nbNotesInferieuresALaMoyenne=0;
    for (int i=0; i<nombreDeNotes; i++) {
      if (note[i]<=moyenneDesNotes) {
        nbNotesInferieuresALaMoyenne++;
      }
    }

    System.out.println("moyenne des notes : " + moyenneDesNotes);
    System.out.println(nbNotesInferieuresALaMoyenne
        + " notes sont inférieures ou égales à la moyenne");
    System.out.println(
        (nombreDeNotes - nbNotesInferieuresALaMoyenne)
        + " notes sont supérieures à la moyenne");
    }
}
```

9.7 Raisonnement sur les tableaux - notion de tranche

objectif : *savoir raisonner en utilisant la notion de tranche, zone d'éléments consécutifs d'un tableau.*

De nombreux traitements sur des tableaux se résolvent en utilisant la notion de tranche. Une tranche est une portion de tableau délimitée par deux indices ***i*** et ***j***. La tranche allant de l'indice **`i`** à l'indice

`j` pourra être notée `T[i..j]` si la position j est comprise et `T[i..j[` si la position `j` est exclue.



Par convention, `T[i..j]` est une tranche vide si `j<i`.

Le principe de résolution suit généralement le schéma suivant :

- La propriété désirée pour le résultat est ***trivialement vraie pour une tranche vide***, ce qu'on obtient, par exemple en posant ***i***=***j***=0, pour la tranche `T[i..j[`.
- Cette propriété est maintenue vraie par une itération dont le corps fait ***strictement croître la taille de la tranche***.
- L'itération se termine lorsque la tranche est devenue tout le tableau.

La propriété concernant le tableau est alors assurée.

Exemple 1 : indice d'un élément maximal dans un tableau

La fonction suivante rend en résultat un indice contenant la valeur maximale d'un tableau. L'algorithme consiste à conserver dans une variable `max` le maximum de la tranche `T[0..i[` et dans une variable `indiceDuMax` l'indice de ce maximum. Ceci conduit à une itération qui initialise `max` à `T[0]`, `indiceDuMax` à 0 et fait varier `i` de 1 à `T.length` en maintenant vrai l'invariant :

`T[indiceDuMax]` est maximal sur la tranche `T[0..i[`

En sortie d'itération, `i` vaut `T.length` et donc `T[indiceDuMax]` est bien maximal sur `T`.

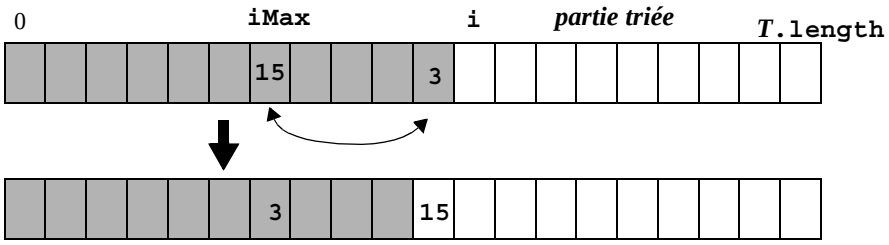
```
static int indiceDeMax(double[] T){
// prérequis : T.length>0
// résultat : indice d'un élément maximum de T
  double max=T[0]; int indiceDuMax=0;
  int i=1;
  while(i<T.length){
  // invariant : T[indiceDuMax] est maximal sur la tranche T[0..i[
    if(T[i]>max){max=T[i]; indiceDuMax=i;}
    i++;
  }
  return indiceDuMax;
}
```

Exemple 2 : tri d'un tableau par placements successifs des éléments maximaux

Soit à réaliser une procédure qui trie un tableau par ordre croissant. Le tri est une opération très utile car, une fois trié, la recherche d'information dans un tableau est considérablement plus rapide que dans un tableau "en vrac". Le tri d'un tableau `T1` est un tableau `T2` qui a les mêmes éléments que `T1` (en valeur et en quantité pour chaque valeur) mais tel que si ***i***>***j*** alors `T2[i]>=T2[j]`. Nous nous intéressons ici à une ***procédure*** de tri, qui a pour ***effet*** de trier un tableau passé en paramètre. Le tri se fera donc sur le tableau lui-même (par opposition à une fonction qui rendrait en résultat un nouveau tableau trié).Le principe que nous allons mettre en œuvre n'est pas très efficace, mais il a

l'avantage d'être simple à comprendre :

Supposons le tableau partiellement trié dans sa partie supérieure. Plus précisément, la tranche **`T[i..T.length[`** est triée et tous ses éléments sont supérieurs à ceux de la tranche **`T[0..i[`**.



Si nous cherchons un indice **`iMax`** du maximum de la tranche **`T[0..i[`**, en échangeant les valeurs de **`T[iMax]`** et de **`T[i-1]`**, la partie triée grossit d'une unité. Il suffit donc d'appliquer ce procédé pour **`i`** variant de **`T.length`** à **`1`**.

La recherche de **`iMax`** peut se faire au moyen d'une fonction similaire à la fonction de l'exemple précédent. La seule différence est que cette fonction cherche un indice du maximum sur une tranche **`T[0..k[`** et non sur le tableau complet. Il faut donc rajouter un paramètre **`k`** qui indique la borne supérieure de la tranche :

```
static int indiceDeMax(double[] T, int k){
// prérequis : k>0 et k<=T.length
// résultat : indice d'un élément maximum de la tranche T[0..k[
  double max=T[0]; int indiceDuMax=0;
  int i=1;
  while(i<k){
  // invariant : T[indiceDuMax] est maximal sur la tranche T[0..i[
    if(T[i]>max){max=T[i]; indiceDuMax=i;}
    i++;
  }
  return indiceDuMax;
}
```

Le tri est alors réalisé par une itération qui maintient l'invariant :

`T` a les mêmes éléments que **T*initial*** (valeur de **`T`** lors de l'appel), **`T[i..T.length[`** est trié et les éléments de **`T[i..T.length[`**sont supérieurs ou égaux à ceux de **`T[0..i[`**.

```
static void trier(double[] T){
// effet : tri T par ordre croissant
  int i=T.length;
  while(i>0){
  // invariant : T a les mêmes éléments que Tinitial
  // T[i..T.length[ est trié et les éléments de T[i..T.length[
  // sont supérieurs ou égaux à ceux de T[0..i[
    int iMax=indiceDeMax(T,i);
    double vMax=T[iMax]; T[iMax]=T[i-1]; T[i-1]=vMax;
    i--;
  }
}
```

En sortie d'itération, **`i`** vaut 0, et le tableau est donc intégralement trié.

Exemple 3 : recherche dichotomique dans un tableau trié

On se propose de trouver l'indice d'une valeur dans un tableau trié. Le tableau est supposé trié par ordre croissant. Si la valeur **v** cherchée se trouve dans le tableau, la fonction **recherche** doit rendre en résultat un indice **i** tel que **T[i]==v**, sinon elle doit rendre -1 (c'est sans ambiguïté puisque les indices commencent à 0).

Dans cet exemple nous cherchons une valeur réelle dans un tableau de nombres réels. Plus généralement ce principe s'applique à la recherche dans un tableau dont le type des éléments est doté d'une relation d'ordre de manière à donner un sens à la notion d'être trié selon cet ordre. Ce pourrait être par exemple un tableau de personnes triées selon l'ordre alphabétique de leurs noms.

Puisque le tableau est trié, on peut pratiquer un algorithme de recherche ***dichotomique***, qui est considérablement plus rapide qu'une recherche par examen des éléments successifs.

Le principe consiste à considérer une tanche **T[i..j]** dans lequel la valeur **v** est susceptible de se trouver (on est sûr qu'elle ne peut être ailleurs).

- Initialement **i**=0 et **j=T.length-1**, de sorte qu'on commence avec le tableau complet.
- La recherche est réalisée par une itération qui compare **v** à la valeur présente au milieu **m** de la tranche. Si **T[m]>v,** la tranche suivante est **T[i..m-1]** et si **T[m]<v**, la tranche suivante est **T[m+1..j]**.
- L'itération termine soit lorsque **T[m]==v**, auquel cas m est l'indice cherché, soit lorsque **i>j**, auquel cas **v** n'est pas dans le tableau puisque la tranche où il pourrait de trouver est devenue vide.

L'invariant de cette itération est :

v n'est pas présent en dehors de la tranche **T[i..j]**

```
static int recherche(double[] T, double v){
// prérequis : T est trié par ordre croissant
// résultat : un indice i tel que T[i]=v si v est présent dans T,
// -1 sinon
  int i=0; int j=T.length-1;
  int milieu=(i+j)/2;
  while(i<=j && T[milieu]!=v){
  // invariant : v n'est pas présent en dehors de la tranche T[i..j]
    if(T[milieu]>v){j=milieu-1;}
    else /* T[milieu]<v */ {i=milieu+1;}
    milieu=(i+j)/2;
  }
  if(i<=j){return milieu;}
  else {return -1;}
}
```

En sortie d'itération, si **i<=j**, **milieu** est l'indice cherché (puisque l'itération s'est terminée avec **T[milieu]==v**), sinon **v** n'est pas présent dans **T**, puisque l'intervalle où il est susceptible de se trouver est vide.

La terminaison de cette itération est garantie par la ***décroissance stricte de la taille de la tranche***

`T[i..j]`. Il faut cependant être prudent. Cette décroissance est stricte car on considère soit `[i..milieu-1]` soit `[milieu+1..j]` dont les tailles sont strictement inférieures à la taille de `[i..j]`. Ce n'est pas toujours le cas pour `[i..milieu]` ou `[milieu..j]` qui ont même taille que `[i..j]` si `i=j`. Dans ce cas, `milieu=i` et l'itération risque de ne pas terminer (c'est ce qui se passe effectivement si on programme ainsi).

Plus formellement, on peut prendre `i-j` comme fonction de terminaison. L'intervalle est divisé par 2 à chaque itération, ce qui garantit un temps de recherche inférieur à ***k.log(`T.length`)***. Par exemple, pour un tableau d'un million d'éléments, la recherche nécessite au plus 20 pas d'itération, car $2^{20} \approx 1000000$.

9.8 Tableaux à plusieurs dimensions

objectif : *connaître l'existence des tableaux à plusieurs dimensions et leur généralisation Java sous la forme de "tableaux de tableaux".*

Un tableau à plusieurs dimensions est un tableau dont les éléments sont désignés par plusieurs indices. Le domaine de chaque indice est un intervalle d'entiers. Par exemple un échiquier peut être représenté par un tableau 8×8, 8 lignes et 8 colonnes, dans lequel chaque élément indique la pièce qui s'y trouve (pion blanc, pion noir, tour blanche, tour noire...) ou bien "libre" s'il ne s'y trouve aucune pièce.

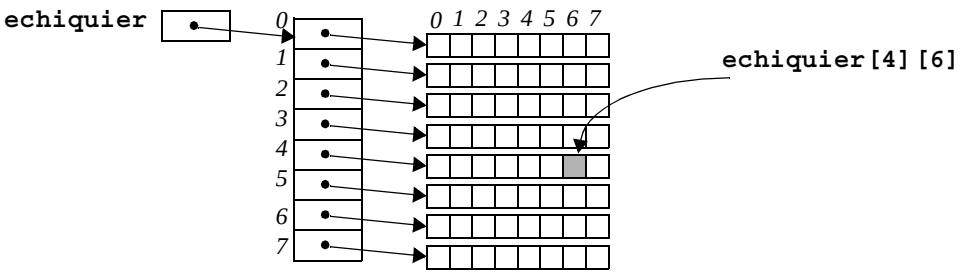
tableau de PièceDEchec[1...8] [1...8] échiquier

Avec un tel tableau, on accède à l'élément de la ligne ***i***, colonne ***j*** par ***échiquier[i,j]***.

En Java, un tableau à plusieurs dimensions est réalisé par un ***tableau de tableaux***, c'est-à-dire un tableau dont les éléments sont des tableaux. Un tel tableau peut être déclaré et créé ainsi :

```
PieceDEchec[][] echiquier = new PieceDEchec[8][8];
```

En Java, un tableau étant toujours désigné par référence, un tableau de tableaux est donc un tableau de références à des tableaux. Ainsi, l'expression `new PieceDEchec[8][8]` crée 8 tableaux de taille 8, les "lignes" et un tableau de 8 références à ces tableaux :



Accès aux éléments :

Ce qui a été dit pour les tableaux à une dimension se généralise facilement. L'accès à un élément se fait en indiquant la valeur d'indice pour chaque dimension. Par exemple :

```
echiquier[4][6]
```

désigne la pièce qui se trouve à la ligne 4 et colonne 6 de l'échiquier (se méfier du fait que les indices commencent à 0).

En supposant que les pièces d'échec sont représentées par une structure **`PieceDEchec`** :

```
enum Nature {pion,tour,cavalier,fou,reine,roi}
enum Couleur {blanc,noir}

class PieceDEchec {
  public Nature nature;
  public Couleur couleur;
  public PieceDEchec(Nature saNature, Couleur saCouleur){
    nature=saNature; couleur=saCouleur;
  }
}
```

Le placement d'un cavalier blanc en position (4,6) s'écrit :

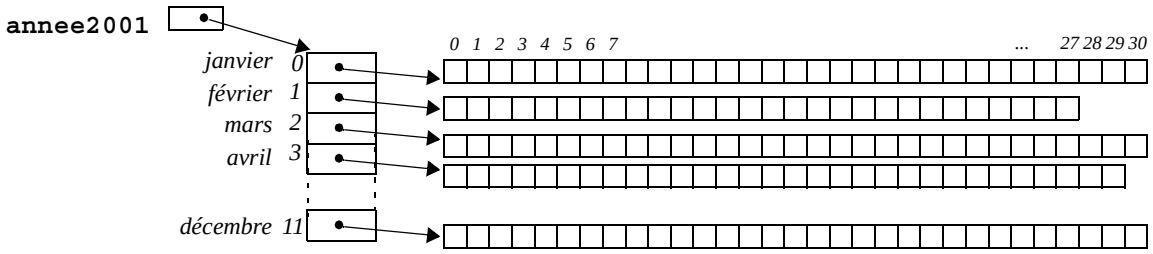
```
echiquier[4][6] = new PieceDEchec(Nature.cavalier,Couleur.blanc);
```

Tableaux "irréguliers" :

Le tableau de tableaux offert par Java est une notion plus générale que le tableau à plusieurs dimensions :

- Si on ne donne qu'un seul indice, on obtient un élément qui est lui même un tableau de dimension immédiatement inférieure : **`echiquier[4]`** désigne la ligne 4 de l'échiquier, et c'est la référence à un tableau de dimension 1.
- Rien n'interdit que les éléments du premier tableau soient de tailles différentes. L'exemple suivant illustre un tableau **`annee2001`** qui représente les 12 mois de l'année 2001. Chaque élément de ce tableau est un tableau indicé par le numéro de jour du mois, de taille 31, 28 ou 30 selon le mois :

```
int[][] annee2001= new int[][12]
annee2001[0]=new int[31];
annee2001[1]=new int[28];
...
annee2001[11]=new int[31];
```



Remarques :

- Le nombre de “lignes” d’un tableau **M** à deux dimensions s’obtient par **M.length**. C’est normal puisque **M**, en tant que tableau (de tableaux) est le tableau des “lignes”.
- Dans la mesure où la “ligne” **i** de **M** existe, son nombre d’élément est **M[i].length**.
- Si **M** est un tableau rectangulaire “régulier” et de taille non nulle, son nombre de “colonnes” peut s’obtenir par **M[0].length**.

Exemple d’utilisation de tableaux réguliers à 2 dimensions : produit de matrices

Une des utilisations importantes des tableaux à deux dimensions est tout ce qui se rapporte au calcul matriciel : transformations linéaires, résolutions de systèmes d’équations linéaires...

L’exemple suivant illustre le produit de matrices de nombres réels. La fonction **produitMatrice** a pour paramètres une matrice $n \times p$ **A** (n lignes, p colonnes) et une matrice $p \times q$ **B**. Elle rend en résultat la matrice produit **C**=**A**×**B**, de n lignes et q colonnes. Le calcul est réalisé selon la formule classique $C_{ij} = \Sigma A_{ik} \times B_{kj}$, ce calcul étant réalisé au moyen de trois boucles imbriquées. Le programme principal calcule et affiche le produit de deux matrices 3×3 données sous forme de tableaux initialisés.

```
class TestMatrice {

  static double[][] produitMatrice(double[][] A, double[][] B) {
  // prérequis : A est une matrice n x p et B une matrice p x q
  // résultat : le produit matriciel A X B
    final int n=A.length;
    final int p = B.length;
    final int q = B[0].length;
    double[][] C = new double[n][q]; // matrice résultat
    for(int i=0;i<n;i++){
      for (int j=0;j<q;j++){
        C[i][j]=0;
        for (int k=0;k<p;k++) {
          C[i][j]= C[i][j]+A[i][k]*B[k][j];
        }
      }
    }
    return C;
  }

  public static void main(String[] arg) {
    double[][] A = {{5,3,8},{6,9,0},{12,67,3}};
    double[][] B = {{2,0,0},{0,2,0},{0,0,2}};
    double[][] C = produitMatrice(A,B);
    System.out.println("produit AxB :");
    for (int i=0;i<3;i++) {
      for (int j=0;j<3;j++) {System.out.print(C[i][j]+" ");}
      System.out.println();
    }
  }
}
```

QCM 9.1

❑ 1 - Les tableaux servent à faire correspondre des nombres entiers (les indices) et des données de types quelconques mais identiques (les éléments du tableau).

❑ 2 - Un tableau remplace avantageusement l'usage de variables de même type, car il est plus concis d'utiliser des noms de la forme `T[0]`, `T[1]`, `T[2]`, `T[3]` ... que des identificateurs divers.

❑ 3 - En Java, les tableaux sont des objets toujours désignés par référence.

❑ 4 - En Java, on peut créer un tableau de 20 éléments (de type `int` par exemple) par une simple déclaration de la forme : `int[20] T;`

❑ 5 - En Java, un tableau de 20 éléments (de type `int` par exemple) est nécessairement créé par une instruction : `new int[20];`

❑ 6 - En Java, une déclaration de la forme `int[] T;` définit `T` comme étant une référence à un tableau d'entiers et cette référence est initialisée à `null`, de sorte que `T` ne désigne rien.

❑ 7 - En Java, `T2` étant déclaré `int[] T2;` et `T1` désignant un tableau d'entiers de taille 30, l'instruction d'affectation `T2=T1;` provoque la création d'un tableau de taille 30 dont la référence est captée par `T2` et la copie des éléments de `T1` dans le tableau désigné par `T2`.

❑ 8 - En Java, `T2` étant déclaré `int[] T2;` et `T1` désignant un tableau d'entiers, l'instruction d'affectation `T2=T1;` provoque le captage par `T2` de la référence au tableau désigné par `T1`.

QCM 9.2

Soit la séquence d'instructions :

```
String[] T1 = new String[30];
String[] T2 = T1;
T2[4]="bonjour";
```

❑ 1 - Après exécution de cette séquence, `T1[4]` vaut `null`.

❑ 2 - Après exécution de la séquence précédente, `T1[4]` vaut la chaîne vide `""`.

❑ 3 - Après exécution de la séquence précédente, `T1[4]` vaut `"bonjour"`.

❑ 4 - Après exécution de la séquence précédente, `T1[4]` n'a pas de valeur définie.

QCM 9.3

On considère les fonctions suivantes :

```
static void renverseLOrdre(int[] T){
  for(int i=0; i<T.length/2; i++){
    int v=T[i]; T[i]=T[T.length-i-1]; T[T.length-i-1]=v;
  }
}

static int [] ordreInverse(int[] T){
  int[] resul= new int[T.length];
  for(int i=0; i<T.length; i++){
    resul[i]=T[T.length-i-1];
  }
  return resul;
}
```

❑ 1 - La fonction **`renverseLOrdre`** modifie le tableau désigné par le paramètre **`T`**.

❑ 2 - La fonction **`renverseLOrdre`** rend en résultat un nouveau tableau contenant les mêmes éléments que **`T`** dans l'ordre inverse renverse de celui où ils figurent dans **`T`**.

❑ 3 - La fonction **`renverseLOrdre`** renverse l'ordre des éléments de **`T`**, c'est-à-dire : échange **`T[`*`i`*`]`** avec **`T[T.length-`*`i`*`-1]`** pour tout ***i*** de 0 à **`T.length-1`**.

❑ 4 - La fonction **`ordreInverse`** modifie le tableau désigné par le paramètre **`T`**.

❑ 5 - La fonction **`ordreInverse`** rend en résultat un nouveau tableau contenant les mêmes éléments que **`T`** dans l'ordre inverse renverse de celui où ils figurent dans **`T`**.

❑ 6 - La fonction **`ordreInverse`** renverse l'ordre des éléments de **`T`**, c'est-à-dire : échange **`T[`*`i`*`]`** avec **`T[T.length-`*`i`*`-1]`** pour tout ***i*** de 0 à **`T.length-1`**.

QCM 9.4

❑ 1 - En Java, on peut créer un tableau de taille nulle, par exemple un tableau d'entiers de zéro éléments, par **`new int[0]`**.

❑ 2 - Un tableau de taille nulle est désigné par **`null`**.

Considérons un tableau de tableau ainsi déclaré : **`double[][] T;`**.

❑ 3 - La taille des toutes les lignes de **`T`** s'obtient par **`T.length`**.

❑ 4 - La taille de toutes les lignes de **`T`** s'obtient par **`T[0].length`**.

❑ 5 - Si **`T`** est un tableau régulier rectangulaire et si **`T.length!=0,`** la taille de toutes les lignes de **`T`** s'obtient par **`T[0].length`**.

❑ 6 - Si **`i>=0`** et **`i<T.length,`** la taille de la ligne **`i`** de **`T`** s'obtient par **`T[i].length`**.

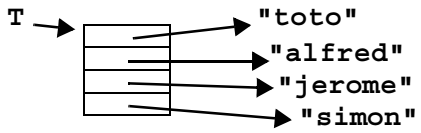
Exercice 9.1 Diverses fonctions sur tableaux de chaînes de caractères

objectif : *ces exercices ont pour but de se familiariser avec l'usage des tableaux. La fonction enClair demandé dans l'exercice 1 permet de visualiser un tableau. Elle sera très utile par la suite pour des tests.*

Rédiger et tester les fonctions suivantes de manipulation de tableaux de chaînes de caractères :

1 - Fonction **`enClair`** telle que **`enClair(T)`** rend en résultat la chaîne de caractères "en clair" constituée des éléments de **`T`** par ordre d'indices séparés par des blancs. Exemple :

avec :



résultat : **`"{toto,alfred,jerome,simon}"`**

2 - Fonction **`estPresent`** telle que **`estPresent(T,s)`** indique si la chaîne de caractère **`s`** est présente dans **`T`**. Avec l'exemple précédent :

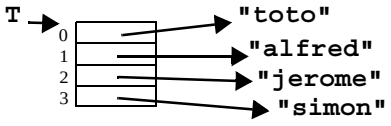
`estPresent(T,"jerome")` rend **`true`**

`estPresent(T,"jules")` rend **`false`**

3 - Fonction **`indiceDuplusPetitAlphabetique`** telle que **`indiceDuPlusPetitAlphabetique(T,i)`** soit l'indice de la plus petite chaîne, selon l'ordre alphabétique, contenu dans le tableau **`T`** à partir de l'indice **`i`**. Pour tester l'ordre alphabétique on utilisera la méthode ***`s1`*`.compareTo(`*`s2`*`)`** qui rend un entier négatif si la chaîne ***s1*** est inférieure à ***s2***, 0 si les chaînes sont égales et un entier positif si ***s1*** est supérieur à ***s2***.

Exemple :

avec :



résultat : 1 (indice dans **`T`** de **`"alfred"`**)

4 - Procédure **`trier`** telle que **`trier(T)`** ***transforme*** le tableau **`T`** en un tableau contenant tous les éléments que possédait **`T`** classés par ordre croissant.

Remarque : on peut créer un tableau initialisé au moyen d'une instruction de la forme :

```
String[] T={"toto","alfred","jerome","simon"};
```

Ceci est très pratique pour faire des tests reproductibles.

Exercice 9.2 Tri d'un tableau par détermination des rangs des éléments

objectif : *les exercices 9.2 à 9.6 concernent le tri de tableaux. L'objectif de ces exercice est double :*
- connaître quelques algorithmes classiques de tri,
- apprendre à raisonner sur des tranches de tableau.

1 - Rédiger la fonction

```
static int nombreDe(double v, double[] T)
```

qui rend en résultat le nombre d'occurrences de **v** dans **T**.

2 - Rédiger la fonction

```
static int rangDe(double v, double[] T)
```

qui rend en résultat le rang par ordre croissant de **v** dans **T**. Le rang de **v** dans **T** est le nombre d'éléments de **T** ***strictement inférieurs*** à **v**. Ainsi si aucun élément n'est inférieur à **v**, le résultat est 0, et si tous les éléments sont inférieurs à **v**, le résultat est **T.length**.

3 - Utiliser les fonctions précédentes pour programmer la fonction de tri :

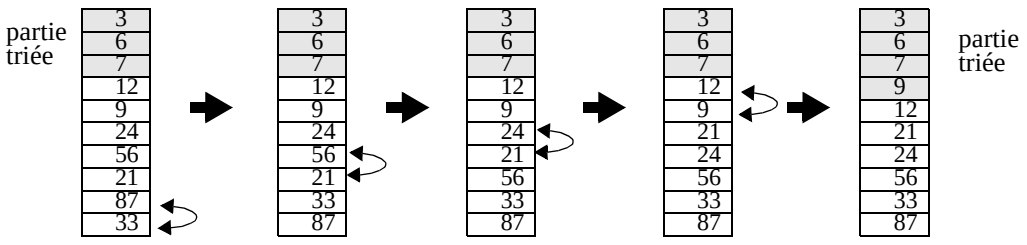
```
static double[] tri(double[] T){
// résultat : un nouveau tableau contenant les éléments de T
// triés par ordre croissant
```

Remarque : le principe de ce tri consiste, pour chaque élément de **T** à placer dans le résultat à partir du rang de cet élément autant de fois cet élément qu'il a d'occurrences dans **T**. Ce principe simple conduit à replacer plusieurs fois, à sa place, la même valeur dans le tableau résultat, mais ce n'est pas grave (l'éviter serait assez compliqué).

Exercice 9.3 Tri par permutations et tri à bulle (Bubble sort)

1 - Tri par permutations

Le tri par permutations consiste à parcourir le tableau en échangeant les éléments s'ils ne sont pas dans l'ordre souhaité. A chaque parcours, un élément de plus se trouve à sa place



Ainsi, comme le suggère l'exemple ci-dessus, à chaque parcours la partie triée du tableau s'accroît d'au moins un emplacement.

Pour un tableau de taille *n*, suffit de répéter *n* fois ce processus pour trier complètement le tableau.

Puisque à chaque parcours la partie triée s'accroît d'au moins une unité, on évite des comparaisons inutiles en effectuant le parcours suivant sur une tranche diminuée d'une unité, c'est à dire uniquement sur la partie possiblement non triée.

Cet algorithme est souvent présenté sous forme d'une métaphore, la "montée de bulles". Considérons le cas d'un tri par ordre croissant, comme le montre l'exemple précédent. Un éléments **`T[i]`** tel que **`T[i]<T[i-1]`** est appelé "bulle", par analogie avec une bulle de gaz dans un liquide. Une telle bulle est destinée à "monter" vers la surface (les indices faibles ici), par échange avec l'élément précédent.

Il est bon de concrétiser ce concept par une procédure **`monteeDeBulles`** qui réalise une remontée de bulles sur une tranche **`T[k..T.length[`**. Voici sa spécification :

```
static void monteeDeBulles(double[] T, int k)
// prérequis : 0<=k<T.length
// effet : réalise une montée de bulles
// sur la tranche T[k..T.length[
```

1.1 - Rédiger la procédure **`monteeDeBulles`**.

La procédure **`monteeDeBulles`** est destinée à être utilisée dans la procédure de tri, au moyen d'un itération qui procède à un succession de remontées de bulles sur des tranches **`T[k..T.length[`** de tailles décroissantes de sorte à trier totalement le tableau **`T`**.

1.2 - Rédiger cette procédure **`trier`**.

2 - Tri à bulle

Le tri à bulle est une amélioration du tri par permutations. Il consiste à terminer la procédure de tri dès que la tentative de montée de bulles ne fait monter aucune bulle (ne procède à aucun échange). En effet, cela signifie que la tranche sue laquelle on a tenté une montée de bulle est correctement ordonnée, et puisque tous ses éléments sont supérieurs à la partie déjà triée, le tableau est totalement trié.

Cela nécessite de modifier la procédure de montée de bulles de manière à ce qu'elle indique si une bulle a effectivement été montée. Cette nouvelle procédure, **`tenteMonteeDeBulles`** est ainsi spécifiée :

```
static boolean tenteMonteeDeBulles(double[] T, int k)
// prérequis : 0<=k<T.length
// effet : réalise une montée de bulles
// sur la tranche T[k..T.length[
// résultat : indique si une bulle a effectivement été montée
```

2.1 - Rédiger la procédure **`tenteMonteeDeBulles`**.

2.2 - Rédiger cette procédure **`trier`** qui réalise un tri à bulle.

Exercice 9.4 Tri par comptage (Distribution counting)

Le “tri par comptage” (Distribution counting) ne peut s’appliquer que si le domaine des valeurs est énumérable (un intervalle d’entiers par exemple) et pas trop grand. On procède en deux temps :

1 - on compte le nombre d’occurrences de chaque valeur présente dans le tableau à trier, dans un tableau dont la taille est la dimension du domaine possible pour les valeurs,

2 - ensuite on construit le tableau trié en parcourant le tableau des occurrences.

exemple, soit le tableau **`T`** :

5	4	2	5	7	4

Le tableau des occurrences est :

0	1	2	3	4	5	6	7
0	0	1	0	2	2	0	1

c’est-à-dire : T contient un 2, deux 4, deux 5 et un 7.

Le tableau T trié est alors :

2	4	4	5	5	7

obtenu par placements successifs de un 2, deux 4, deux 5 et un 7.

Rédiger la procédure **`triParComptage`** qui réalise cet algorithme.

Remarque : la complexité dépend du domaine de valeurs. C’est O(*n*)+O(*m*) *m* étant la taille de l’intervalle des valeurs. C’est intéressant si *m* est petit par rapport à *n*.

Exercice 9.5 Recherche dichotomique, version récursive

Nous rappelons le principe d’une recherche dichotomique dans un tableau trié :

Pour chercher une valeur ***v*** dans une portion du tableau comprise entre les indices ***i*** et ***j***, on prend l’indice du “milieu”, (***i***+***j***)/2. Si l’élément cherché s’y trouve, c’est terminé. Si l’élément cherché est plus petit que l’élément du milieu on cherche dans la moitié inférieure. S’il est plus grand on cherche dans la moitié supérieure. L’élément ne se trouve pas dans le tableau si l’intervalle de recherche devient vide.

Nous avons vu une réalisation itérative de cette fonction. Rédiger une réalisation ***récursive*** de cette fonction :

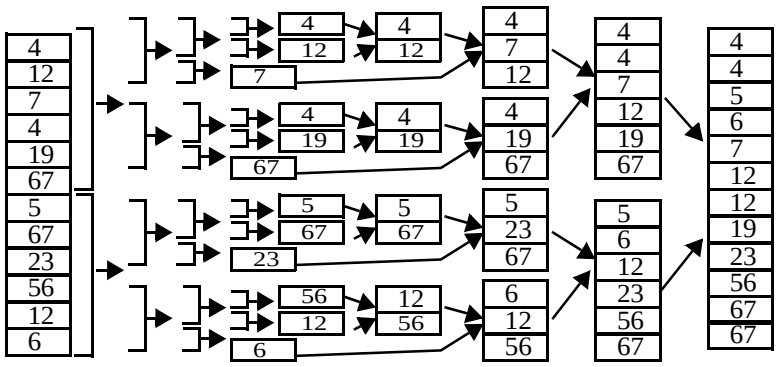
```
static int indiceDe(int v, int [] T)
// prérequis : T est trié par ordre croissant
// résultat : indice dans T où se trouve
// la valeur v si elle s'y trouve,
// -1 si v est absente.
```

Guide : il faut rédiger une fonction auxiliaire (qui est récursive) qui cherche **`v`** dans une tranche **`T[i..j]`**.

Exercice 9.6 Tri par fusion

Un algorithme rapide pour trier un tableau consiste à :

- couper le tableau en deux parties égales (à un près),
- trier les deux parties,
- fusionner les deux parties triées T1 et T2 en respectant l'ordre des éléments, c'est-à-dire en prenant à chaque fois le plus petit élément parmi les éléments non encore pris de T1 ou de T2.



Remarque : la complexité est O(*n.log(n)*). C'est le mieux que l'on puisse faire dans le cas général, pour un tableau n'ayant aucune propriété particulière.

On veut rédiger selon ce principe une fonction de tri d'un tableau d'entiers :

```
static int[] tri(int[] T)
// résultat : un nouveau tableau contenant
// les éléments de T triés par ordre croissant
```

L'algorithme suggéré est naturellement ***récursif***. Son petit défaut est de nécessiter des tableaux intermédiaires. Pour éviter de créer de nouveaux tableaux lors de l'éclatement d'un tableau en deux parties, on peut passer en paramètre une tranche du tableau originel, c'est-à-dire le tableau **`T`** et deux indices **`i`** et **`j`** qui délimitent les éléments à trier. Pour cela il faut rédiger une fonction auxiliaire :

```
static int[] triTranche(int[] T, int i, int j)
// prérequis : ???
// résultat : un nouveau tableau contenant
// les éléments de la tranche T[i..j[ triés par ordre croissant
```

- exprimer **`tri`** en utilisant **`triTranche`** (très simple),
- indiquer le ***prérequis*** raisonnable de **`triTranche`** (relation entre **`i`**, **`j`** et **`T.length`**),
- rédiger **`triTranche`** (fonction récursive).

Exercice 9.7 Manipulation de matrices

Rédiger les fonctions suivantes de manipulation de matrices :

1 - Visualisation en clair : rédiger une fonction **`enClair`** qui rend en résultat une chaîne de caractères qui visualise en clair un tableau d'entiers **`M`** à deux dimensions. Cette chaîne comportera un passage à la ligne avant chaque ligne et un espace entre les éléments d'une ligne.

Exemple : pour

```
int[][] M = {{6,1,8,5}, {7,5}, {2,9,6}}
```

l'impression du résultat de **`enClair(M)`** donne :

```
6 1 8 5
7 5
2 9 3
```

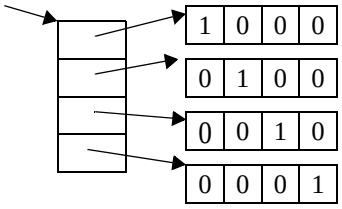
Spécification :

```
static String enClair(int[][] M){
// résultat : chaîne visualisant M en clair,
// passage à la ligne pour chaque ligne,
// un espace entre éléments d'une ligne
```

2 - Matrice unité : la fonction **`matriceUnite`** doit rendre en résultat un tableau a deux dimension carré de taille **`n`** qui représente la matrice unité : des 1 sur la diagonale principale et des 0 ailleurs. Les éléments seront des nombres entiers de type **`int`**.

Exemple : **`matriceUnite(4)`**

rend en résultat le tableau à 2 dimension :



qui représente la matrice :

1	0	0	0
0	1	0	0
0	0	1	0
0	0	0	1

Spécification :

```
static int[][] matriceUnite(int n)
// résultat : représentation de la matrice unité de taille n
```

3 - Transposée : la fonction **`Transposee`** doit rendre en résultat un tableau représentant la matrice transposée de celle représentée par un tableau carré **`M`**.

Rappel : la transposée d'une matrice M est la matrice MT telle que MT[i,j] = M[j,i] (interversion des lignes et des colonnes).

Exemple :

avec **M** représentant la matrice :

12	6	4
9	1	0
3	15	8

Transposee(M) représente la matrice :

12	9	3
6	1	15
4	0	8

Spécification :

```
static int[][] transposee(int[][] M)
// prérequis : M est un tableau carré
// résultat : représentation de la matrice transposée
// de celle représentée par M
```

4 - Test de diagonalité : la fonction **estDiagonale** doit indiquer si la matrice représentée par un tableau carré **M** est diagonale. Rappel : une matrice M est diagonale si seule les éléments M[i,i] sont différents de 0.

Exemples :

la matrice suivante est diagonale :

12	0	0
0	1	0
0	0	8

celle-ci n'est pas diagonale :

12	0	0
0	1	15
4	0	8

Spécification :

```
static boolean estDiagonale(int[][] M)
// prérequis : M est un tableau carré
// résultat : indique si la matrice représentée par M est diagonale
```

5 - Carré magique

Une matrice carrée d'entiers est "magique" si les sommes des valeurs de chaque ligne, de chaque colonne et des deux diagonales sont égales.

Exemple de carré magique :

6	1	8
7	5	3
2	9	4

La fonction **estMagique** doit indiquer si la matrice représentée par un tableau **M** est magique.

Spécification :

```
static boolean estMagique(int[][] M)
// prérequis : M est un tableau carré
// résultat : indique si la matrice représentée par M est magique
```

Exercice 9.8 Calcul des nombres premiers par le crible d'Eratosthène

L'algorithme suivant permet de déterminer les nombres premiers inférieurs à ***n***. Soit un tableau de booléens **`premier`**, de taille ***n***, initialisé à ***vrai***.

- On commence avec l'indice *k*=2, après avoir mis premier[0] et premier[1] à faux.
- Si **`premier[`***k*`]`** =***vrai***, on met à ***faux*** tous les éléments d'indice multiple de *k* jusqu'à l'indice k^2.
- On incrémente *k* et on recommence tant que *k* est inférieur à la racine carré de ***n***.

Les nombres premiers sont alors les nombres ***i*** tels que **`premier[`*i*`]`** =***vrai***.

Rédiger la fonction suivante selon ce principe :

```
static boolean[] premier(int n)
// prérequis : n>=0
// résultat : tel que pour tout i <n,
// premier(n)[i] indique si i est premier
```

Aide 9.1 Diverses fonctions sur tableaux de chaînes de caractères

Fonction **`enClair`** :

Il faut se méfier de placer les "," uniquement entre deux éléments de **`T`**. Pour cela on peut concaténer au résultat "***,élément***" pour chaque élément ***sauf pour le premier***.

Encore faut-il que **`T`** possède au moins un élément. Nous acceptons que **`T`** soit vide (**`T.length`**=0) et le résultat est "**`{}`**" dans ce cas. Il convient de traiter ce cas séparément au début du corps de la fonction.

Fonction **`estPresent`** :

`T` n'étant pas a priori trié, nous sommes obligés d'effectuer un parcours des éléments successifs. L'itération s'arrête lorsqu'on a parcouru tout le tableau (**`i==T.length`**) ou quand on rencontre un élément égal à **`s`**.

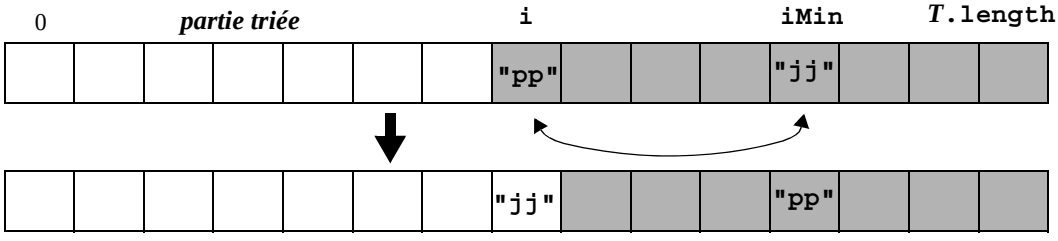
Fonction **`indiceDuPlusPetitAlphabetique`** :

La réalisation de cette fonction est similaire à celle de la fonction **`indiceDeMax`** vue dans ce chapitre du cours. Les différences sont :

- L'ordre est l'ordre alphabétique, qui se teste par **`T[i].compareTo(min)<0`** pour savoir si **`T[i]`** est "plus petit" (avant par ordre alphabétique) que **`min`**.
- On cherche le minimum sur la tranche **`T[k..T.length[`** au lieu de la tranche **`T[0..k[`**.

Procédure **`trier`** :

Cette procédure est similaire à la procédure de tri d'un tableau de nombres vue dans ce chapitre du cours. La seule différence est que l'on maintient triée la tranche **`T[0..i[`** au lieu de la tranche **`T[i..T.length[`** car on procède par déplacement des éléments minimaux successifs (au lieu des éléments maximaux).



Aide 9.2 - Tri d'un tableau par détermination des rangs des éléments

1 - Fonction **`nombreDe`**

Il suffit de parcourir le tableau **`T`** en incrémentant un compteur **`resul`** chaque fois que l'élément **`T[i]`** est égal à **`v`**.

2 - Fonction **`rangDe`**

Il suffit de parcourir le tableau **`T`** en incrémentant un compteur **`resul`** chaque fois que l'élément **`T[i]`** est strictement inférieur à **`v`**.

3 - Fonction **`tri`**

Pour chaque élément **`T[i]`**, on calcule son rang **`j`** et son nombre d'occurrences **`n`**, puis on place dans le tableau résultat, à partir de **`j`**, **`n`** occurrences successives de **`T[i]`**.

Aide 9.3 Tri par permutations et tri à bulle (Bubble sort)

1 - Tri par permutations

1.1 - Procédure **`monteeDeBulles`**.

Il faut parcourir le tableau depuis son dernier élément (**`i=T.length`**) jusqu'à l'indice **`k`** exclu (**`i==k-1`**, soit encore **`i<k`**) en réalisant un échange entre **`T[i]`** et **`T[i-1]`** si **`T[i]<T[i-1]`**.

1.2 - Procédure **`trier`**.

Il suffit de solliciter pour **`i`** variant de 0 à **`T.length-1`**. A chaque itération, un élément de plus est à sa place dans **`T`**. En sortie d'itération **`T`** est totalement trié.

2 - Tri à bulle

2.1 - Procédure **`tenteMonteeDeBulles`**.

Le principe est le même que celui de la procédure **`monteeDeBulles`**, mais avec en plus l'élaboration du résultat booléen au moyen d'une variable **`resul`** qui indique à tout moment s'il y a eu un échange sur la tranche d'indices compris entre **`i`** et **`T.length`**.

2.2 - Procédure **`trier`**.

Elle utilise une variable booléenne **`pasTermine`** destinée à capter le résultat de **`tenteMonteeDeBulles`**. L'itération s'arrête lorsque **`pasTermine`** devient faux, ce qui signifie qu'aucune bulle n'a été montée par **`tenteMonteeDeBulles`**.

Aide 9.4 Tri par comptage

La procédure de tri par comptage commence par chercher le plus grand élément de **T** de manière à déterminer la taille du tableau de comptage **nombreDOccurences**. Ce tableau est créé avec cette taille et initialisé avec des 0 partout.

Ensuite, le tableau **T** est parcouru en comptant le nombre d'occurrences de chaque valeur rencontrée, c'est-à-dire en incrémentant **nombreDOccurences[**v**]** pour chaque valeur ***v*** rencontrée.

Enfin le tableau **nombreDOccurences** est parcouru pour placer des séquences successives de **nombreDOccurences[*i*]** valeurs égales à ***i*** dans le tableau **T**.

Aide 9.5 Recherche dichotomique, version récursive

La fonction **indiceDe** utilise une f***onction auxiliaire récursive*** **rechercheDansTranche** dont voici les spécifications :

```
static int rechercheDansTranche(int i,int j,double[] T, double v){
// prérequis : T est trié par ordre croissant
// résultat : un indice k tel que T[k]=v si v est présent
// dans la trancheT[i..j], -1 sinon
```

Aide 9.6 Tri par fusion

Le tri consiste à utiliser **triTranche** sur tout le tableau, soit : **triTranche(T,0,T.length-1)**.

Fonction **triTranche** :

Le résultat est un nouveau tableau de même taille que la tranche. Il convient de créer ce tableau en début du corps de la fonction :

```
int[] resul = new int[j-i+1];
```

L'arrêt de la récursivité a lieu pour une tranche réduite à un seul élément. Le résultat est alors le tableau constitué de ce seul élément : **resul[0]=T[i]**

Pour un tableau de plus d'un élément, le découpage en deux parties de la tranche **T[i,j]** donne les deux tranches **T[i,(i+j)/2]** et **T[(i+j)2+1,j]**.

Le tri de ces deux tranches (appels récursifs) donne les tableaux triés **T1** et **T2**.

Ces tableaux sont alors fusionnés dans **resul**.

La fusion consiste à "piocher" dans **T1** ou dans **T2** selon en prélevant chaque fois le plus petit élément parmi **T1[k1]** et **T2[k2]**.

La fusion se termine par un "éclusage" d'un des tableaux. En effet, le prélèvement dans **T1** ou **T2** doit s'arrêter dès qu'un des tableaux est épuisé. Il faut alors compléter **resul** par les éléments restant dans l'autre tableau.

Aide 9.7 Manipulation de matrices

1 - Fonction `enClair`

Peu de difficulté. Placer un passage à la ligne, **`"\n"`**, devant chaque visualisation de ligne.

2 - Fonction `matriceUnite`

Le résultat est un tableau carré de taille **`n`**, créé par

```
int[][] resul = new int[n][n];
```

Deux boucles imbriquées parcourent le tableau **`resul`** pour y placer 1 aux indices correspondant à la diagonale principale (**`i==j`**) et 0 ailleurs (**`i!=j`**).

3 - Fonction `Transposee`

Le résultat est un tableau carré de même taille que **`M`**, créé par

```
int n=M.length; int[][] resul = new int[n][n];
```

Deux boucles imbriquées parcourent les tableaux **`M`** et **`resul`** pour appliquer la définition de ce qu'est la transposée :

```
resul[i][j] = M[j][i]
```

4 - Fonction `estDiagonale`

Il suffit de vérifier que tout élément hors de la diagonale principale est nul.

5 - Fonction `estMagique`

Il est bon de se donner les fonctions intermédiaires suivantes, qui permettront de programmer de façon aisée et compréhensible la fonction **`estMagique`**. Ces fonction sont :

- **`sommeLigne`** : la somme des éléments d'une ligne **`i`** de **`M`**,
- **`sommeColonne`** : la somme des éléments d'une colonne **`j`** de **`M`**,
- **`sommePremiereDiagonale`** : la somme des éléments de la première diagonale de **`M`**,
- **`sommeDeuxiemeDiagonale`** : la somme des éléments de la deuxième diagonale de **`M`**.

Disposant de ces fonctions, il suffit de calculer **`sommeLigne0`**, la somme des éléments de la ligne 0, puis de vérifier que c'est aussi la somme des éléments de chaque ligne, des éléments de chaque colonne et des éléments des deux diagonales.

Aide 9.8 Calcul des nombres premiers par le crible d'Eratosthène

Le tableau booléen résultat, **`resul`**, est de taille **`n`**. Il faut l'initialiser à **`true`**, sauf **`resul[0]`** et **`résul[1]`** qui doivent être positionnés à **`false`**.

Ensuite, pour chaque **`k`** depuis 2 jusqu'à **`n-1`**, si **`resul[k]`** vaut **`true`** il faut positionner à **`false`** les éléments d'indices multiples de **`k`**. Pour cela il convient d'utiliser une ***boucle qui progresse par pas de*** **`k`**, de la forme :

```
for(int i=2*k; i<n; i=i+k){...}
```

Voici un exemple de procédure de test de cette fonction.

Ce qui nous intéresse en fait ce n'est pas d'afficher le tableau de booléens qui indique si un nombre est premier, mais d'afficher la suite des nombres premiers inférieurs à n. Pour cela nous avons rédigé une fonction **`static String nombresPremiers(int n)`** qui rend en résultat la chaîne de caractères qui visualise en clair cette suite.

```
static String nombresPremiers(int n){
// résultat : la suite en clair des nombres premiers < n
  boolean[] estPremier = premier(n);
  String resul="";
  for(int i=0; i<n; i++){
    if(estPremier[i]){resul=resul+i+" ";}
  }
  return resul;
}

public static void main(String[] z){
  System.out.println("nombres premiers < 100 :");
  System.out.println(nombresPremiers(100));
}
```

La procédure principale de test utilise cette fonction pour imprimer la suite des nombres premiers inférieurs à 100. Cela doit donner :

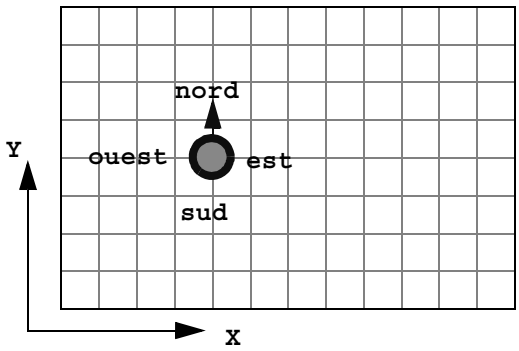
```
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89
97
```

CHAPITRE 10 Objets de type classe - abstraction

La notion de structure permet de définir un type de donnée qui possède plusieurs caractéristiques représentées par les valeurs des champs de la structure. On va maintenant étendre cette notion en permettant de regrouper non seulement des valeurs mais également des opérations dans la déclaration de type. Un tel regroupement de données (***attributs***) et d'opérateurs (***méthodes*)** s'appelle un ***objet***. Un ***type d'objet*** est défini au moyen d'une classe.

10.1 Approche "variable de type structure" - approche "objet"

Lorsque l'on conçoit un logiciel, on est amené à modéliser certains objets du domaine de l'application. Considérons un exemple très élémentaire : un robot simpliste. Un tel robot occupe une certaine position (*X*,*Y*), il a une orientation parmi {*nord*, *est*, *sud*, *ouest*}, il est initialisé avec une position et une orientation données, il peut tourner à droite, il peut avancer d'un pas... On le décrira par une association de variables qui représentent son état et des procédures qui modélisent son comportement.

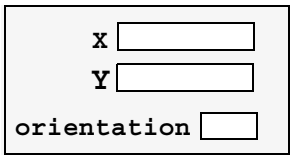


10.1.1 Approche "variable structurée"

Une façon de faire que nous avons déjà pratiquée est la suivante :

- d'une part on utilise un type structure, composé de champs de données, pour représenter les caractéristiques de l'objet à modéliser, un "robot" dans notre exemple :

données

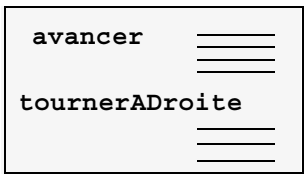



```
class Robot {
  public int X; public int Y;
  public Orientation o;
  public Robot(int x, int y, Orientation versOu) {
    X=x; Y=y; o=versOu;
  }
}
```

avec, pour l’orientation : **`enum Orientation {nord,est,sud,ouest}`**

- d’autre part, on rédige les procédures qui réalisent le comportement attendu :

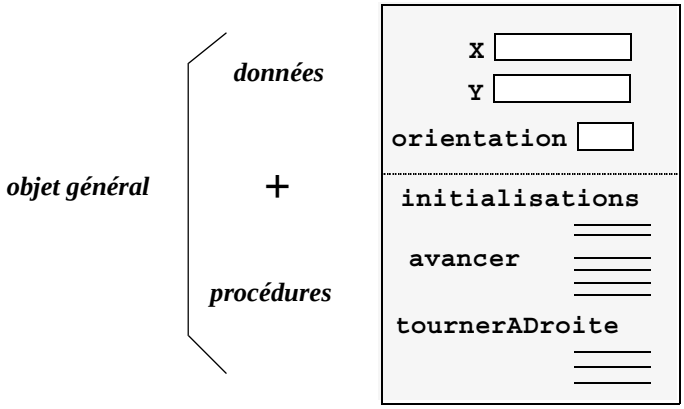
procédures




```
class utilisationDeRobot {

   static void avancer(Robot ceRobot) {
     switch (ceRobot.o) {
     case nord : ceRobot.Y = ceRobot.Y+1; break;
     case est  : ceRobot.X = ceRobot.X+1; break;
     case sud  : ceRobot.Y = ceRobot.Y-1; break;
     case ouest: ceRobot.X = ceRobot.X-1; break;
     }
   }

   static void tournerADroite(Robot ceRobot){
     switch (ceRobot.o) {
     case nord : ceRobot.o=Orientation.est; break;
     case est  : ceRobot.o=Orientation.sud; break;
     case sud  : ceRobot.o=Orientation.ouest; break;
     case ouest: ceRobot.o=Orientation.nord ; break;
     }
   }

   public static void main(String[] arg) {
   Pour créer un robot baptisé vigor initialisé en (1,3) orientation nord,
   puis le faire avancer d’une position, il faut écrire :
     Robot vigor = new Robot(1,3,Orientation.nord);
     avancer(vigor);
     ...
   }
}
```

10.1.2 Approche objet

Une façon de faire plus modulaire consiste à définir les opérations dans le texte de la classe qui définit le type de l'objet.



Cela s'écrit ainsi :

```
class Robot {
   variables qui représentent l'état d'un Robot
   public int X; public int Y;
   public Orientation o;

   initialisations à la création d'un Robot (constructeurs)
   public Robot(int x, int y, Orientation versOu) {constructeur
            X=x; Y=y; o=versOu;
   }
   public Robot() {constructeur sans paramètre, initialise à l'origine, orientation nord
            X=0; Y=0; o=Orientation.nord;
   }

   opérations (méthodes d'objet)

   public void avancer() {
      switch (o) {
      case nord : Y=Y+1; break;
      case st  : X=X+1; break;
      case sud  : Y=Y-1; break;
      case ouest: X=X-1; break;
      }
   }
   public void tournerADroite(){
      switch (o) {
      case nord : o=Orientation.est  ; break;
      case est  : o=Orientation.sud  ; break;
      case sud  : o=Orientation.ouest; break;
      case ouest: o=Orientation.nord ; break;
      }
   }
}
```

Avec cette nouvelle forme, on notera les différences suivantes :

1 - L’objet sur lequel porte une opération, le “*paramètre principal*” ou *paramètre implicite* de l’opération, est placé devant, séparé par un point. Ainsi, pour créer un robot baptisé **`vigor`** initialisé en (1,3) orientation *nord*, puis le faire avancer d’une position, on écrit :

```
Robot vigor = new Robot(1,3,Orientation.nord);
vigor.avancer();
```

2 - En plus du regroupement des opérations dans le texte de la classe, on notera une différence dans la forme : le paramètre explicite “**`ceRobot`**” a disparu de la définition des opérations. Les opérations portent sur le paramètre implicite, objet du type de la classe, qu’on appelle ***instance courante***. Une telle opération qui porte sur l’instance courante s’appelle ***méthode d’objet***.

3 - On remarquera également l’absence du vocable **`static`** : les fonctions qui sont des méthodes d’objet n’ont pas cet attribut. Seules les fonctions qui reçoivent tous leurs paramètres de façon traditionnelle “par les parenthèses” ont l’attribut **`static`**. On a maintenant l’explication de la signification du vocable **`static`** :

- Les composants “***statiques***” d’une classe, variables, constantes ou fonctions affublés de l’attribut **`static`**, sont des composants qui ***existent en un seul exemplaire*** et qui ne sont associés à aucun objet particulier. C’était le cas des fonctions utilisées jusqu’à présent ainsi que des constantes et variables globales utilisées dans certaines classes.
- Les composants “***non statiques***”, c’est-à-dire non affublés de l’attribut **`static`**, sont des composants qui ***existent dans chaque objet*** instance de la classe. C’est le cas des composants de données qui décrivent l’objet (**`X`**, **`Y`**, **`o`**) et des méthodes d’objet (**`avancer()`**, **`tournerADroite()`**).

4 - La forme de l’appel est différente : l’objet sur lequel porte l’opération n’est pas indiqué entre parenthèses comme pour un paramètre explicite, mais devant le nom de l’opération : “**`vigor.avancer()`**”. Pendant l’exécution de cet appel, l’instance courante est l’objet désigné par **`vigor`**.

5 - La rédaction des opérations est également un peu différente : les composants de l’instance courante sont désignées simplement par leurs identificateurs, **`X`**, **`Y`**, **`orientation`**, sans avoir à indiquer explicitement l’objet.

10.1.3 Citation explicite de l’instance courante : `this`

Dans certains cas, on peut avoir besoin de citer explicitement l’instance courante dans le texte d’une méthode d’objet, par exemple pour passer cette instance en paramètre d’une procédure. On dispose pour cela du vocable **`this`** (littéralement “celui-ci”).

Pendant l’exécution d’une méthode d’objet, **`this`** désigne l’instance courante.

Par exemple, pendant l’exécution de l’appel “**`vigor.avancer()`**”, **`this`** désigne l’objet désigné par **`vigor`**.

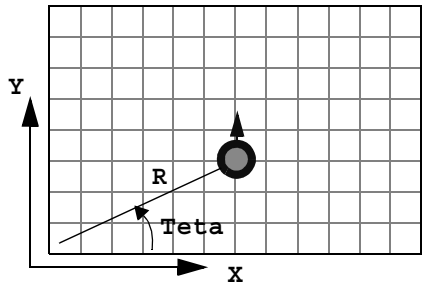
On peut, mais c’est inutile et lourd, préfixer par **`this`** les noms des composants de l’instance courante. Par exemple dans les méthodes de la classe **`Robot`**, **`this.X`**, **`this.Y`** et **`this.o`** signifient la même chose que **`X`**, **`Y`** et **`o`**.

10.1.4 Encapsulation - notion d'abstraction

L'un des principaux soucis du programmeur est l'exactitude de ses programmes : il faut pouvoir assurer que ***seules les choses vraiment voulues puissent se produire***. Pour cela un des moyens consiste à interdire l'accès incontrôlé aux données d'un objet. Voici quelques exemples concernant la classe **`Robot`** :

- On désire que la position d'un **`Robot`** ne puisse être modifiée que par la procédure **`avancer()`**, parce que, par exemple, le mouvement du robot que l'on modélise ne peut se faire que d'une unité dans sa direction courante.
- On désire que l'orientation d'un **`Robot`** ne puisse être modifiée que par la procédure **`tournerADroite()`**, parce que le robot ne peut faire que des rotations d'un quart de tour à droite.

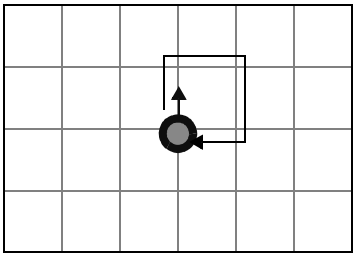
En règle générale, on désire "cacher" les données concrètes qui représentent l'état d'un objet car ces données concrètes, les entiers **`X`**, **`Y`** et **`orientation`** dans l'exemple du robot, résultent d'un choix plus ou moins arbitraire pour représenter l'objet et ne font pas partie des spécifications logiques de l'objet. Par exemple, pour fixer les idées, on pourrait très bien représenter concrètement la position du robot non pas par son abscisse **`X`** et son ordonnée **`Y`**, mais par sa distance à l'origine **`R`** et l'angle **`Teta`** de son rayon vecteur avec l'axe *Ox*, ou toute autre représentation.



Tout ce qui représente correctement l'état du robot peut convenir et c'est au programmeur de faire son choix pour diverses raisons de simplicité de calcul ou de facilité de programmation. Dans tous les cas, ***vu de l'extérieur***, le robot doit se comporter pareillement. Les propriétés du robot doivent être les mêmes quelle que soit la façon dont il est concrètement représenté. Ces propriétés intrinsèques du robot constituent ce qu'on appelle l'***abstraction*** du type **`Robot`**. On ne va pas développer ici une "théorie des robots qui avancent d'un pas et tournent à droite" car ce genre de théorie est inintéressante et par trop spécialisée. Mais pour bien saisir ce qu'est une telle abstraction voici une propriété que possède tout objet de type **`Robot`** :

"si on exécute la séquence d'opérations :
`avancer(), tournerADroite(), avancer(), tournerADroite(),`
`avancer(), tournerADroite(), avancer(), tournerADroite()`
le robot se retrouve dans la même position et la même orientation qu'avant la séquence."

Une telle propriété est ***intrinsèque*** : elle est totalement ***indépendante de la représentation concrète*** choisie.



Puisque les données concrètes ne font pas partie des propriétés abstraites de l'objet, il est malsain de permettre l'accès à ces données. Ces données n'existent que "par hasard", à cause du choix arbitraire de représentation.

La technique usuelle, appelée ***encapsulation***, consiste à interdire l’accès direct aux données des objets en dehors des programmes de la classe. En Java cela se note par le vocable **`private`**. L’identificateur d’un composant de la classe, que ce soit une donnée ou une fonction, affublé du qualificatif **`private`** peut être utilisé ***uniquement depuis les textes de programme de la classe***.

Notion d’accesseur

Dans l’exemple de la classe **`Robot`**, les données concrètes **`X`**, **`Y`** et **`orientation`** doivent être cachées. Cependant il existe bien ici des correspondants directs ***X***, ***Y*** et ***orientation*** qui font partie de l’abstraction “robot” : un robot possède vraiment une abscisse ***X***, une ordonnée ***Y*** et une orientation ***orientation***. Dans ce cas on rédige des méthodes dont le seul rôle est d’offrir un accès à ces informations. Ces méthodes sont généralement triviales : elles rendent en résultat la valeur d’un composant de l’objet, ou affectent un composant de l’objet, parfois après avoir opéré quelques vérifications de consistance. Ces méthodes s’appellent des ***accesseurs***. La convention la plus répandue est d’appeler **`get`*`XXX`** la méthode qui permet de lire la propriété ***XXX*** de l’objet et **`set`*`XXX`** celle qui permet de l’affecter. Avec ces conventions, une rédaction de la classe **`Robot`** devient :

```
class Robot {
   données concrètes cachées
   private int X; private int Y; private Orientation o;

   constructeurs
   public Robot(int x, int y, Orientation versOu) {...}
   public Robot() {...}

   opérations
   public void avancer() {...}
   public void tournerADroite(){...}
   accesseurs
   public int getX() {return X;}
   public int getY() {return Y;}
   public int getOrientation() {return o;}
}
```

On n’a fourni ici que des accesseurs en lecture, car on veut que les mouvements soient limités à ceux définis par **`avancer`** et **`tournerADroite`**.

Pour obtenir l’abscisse du robot **`vigor`**, il suffit d’appeler la méthode : **`vigor.getX()`**. L’écriture **`vigor.X`** est ***interdite*** (en dehors du texte de la classe **`Robot`**) : c’est une erreur de syntaxe.

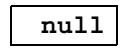
Rappels sur la nature des objets

Nous avons déjà expliqué la nature des objets à propos des cas particuliers que sont les chaînes de caractères (type **`String`**) et les structures, objets sans méthodes dont les composants de données ont le qualificatif **`public`**.

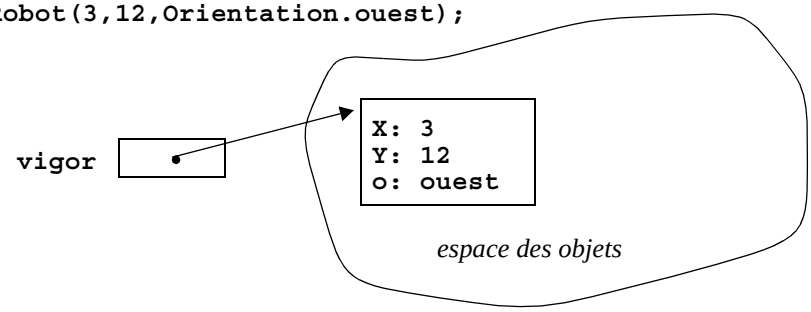
Les objets définis par un type classe ont les mêmes prérogatives :

- Il sont créés au moyen de l’expression : **`new`** ***nomDeLaClasse*** **(*paramètres du constructeur*)**
- Ils sont toujours ***désignés par référence***.
- La valeur des variables, paramètres et résultats de type classe est toujours une référence à un objet de ce type. La déclaration d’une variable de type classe initialise sa valeur à **`null`**, référence qui ne désigne rien.

```
Robot vigor;
```

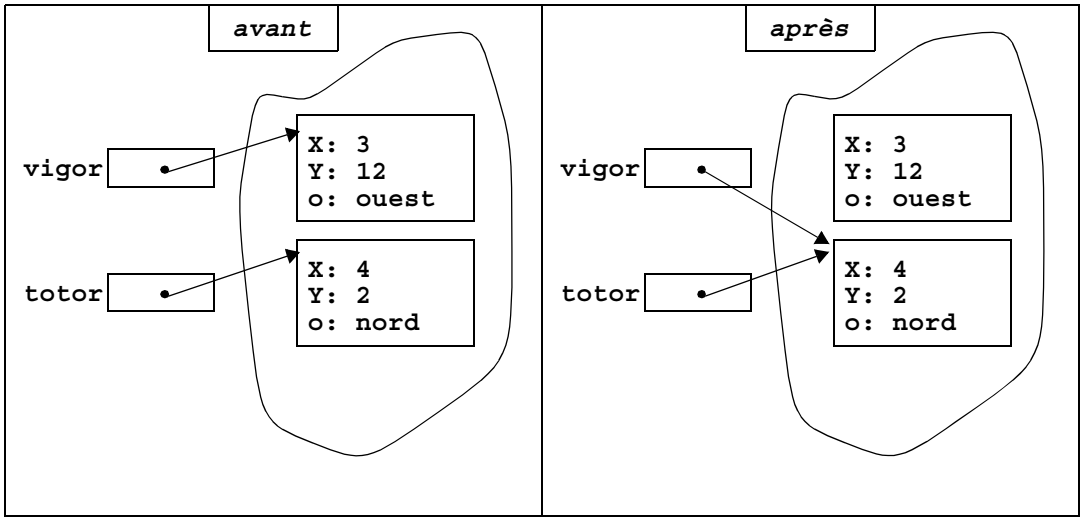


```
vigor = new Robot(3,12,Orientation.ouest);
```

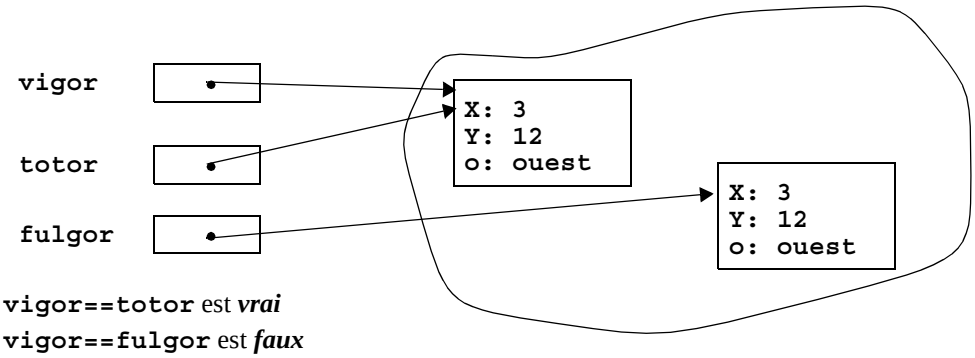


- L'affectation d'une variable de type classe est l'affectation d'une référence à la variable. L'objet anciennement référencé n'est pas affecté.

```
vigor=totor;
```



- L'opérateur de comparaison "`==`" entre expressions de type classe ***compare les références*** et non pas les valeurs des objets référencés. Le résultat est vrai si et seulement si les références ***désignent le même objet***.



`vigor==totor` est ***vrai***
`vigor==fulgor` est ***faux***

10.2 Exemple récapitulatif

La chasse au trésor :

On suppose qu'un trésor est placé à un certain endroit (x_0,y_0) du plan. Un robot modélisé par la classe **`Robot`** précédemment présentée se trouve initialement en *(0,0)*. Le but du jeu est de piloter le robot de façon à l'amener au même endroit que le trésor. Le déroulement de ce jeu peut être précisé ainsi :

choisir au hasard la position (x_0,y_0) du trésor (sans la communiquer au joueur),
fixer la position initiale du robot en *(0,0)*, orientation *nord*,
répéter tant que le robot n'est pas sur le trésor {
indiquer au joueur la distance entre le robot et le trésor,
obtenir un ordre "tourner à droite" ou "avancer" de la part du joueur et l'exécuter,
}
afficher "gagné" et la position du trésor

Pour tirer des nombres au hasard on dispose dans la bibliothèque standard de Java de la classe **`Random`**. Un objet de cette classe est un générateur aléatoire de nombres. Il possède une méthode **`nextInt(k)`** qui rend en résultat un nombre entier tiré au hasard compris entre 0 et *k* (exclu).

```
class ChasseAuTresor {
   // position du trésor (variable globale)
   static int x0; static int y0;

   // le robot (variable globale)
   static Robot leRobot;
   static int carreDistanceAuTresor() {
   // résultat : carré de la distance du robot au trésor
      int dx=leRobot.getX()-x0; int dy=leRobot.getY()-y0;
      return dx*dx + dy*dy;
   }

   public static void main(String[] arg) {
      // choisit au hasard la position (x0,y0) du trésor
      Random generateurAleatoire= new Random();
      x0=generateurAleatoire.nextInt(10);
      y0=generateurAleatoire.nextInt(10);
      // crée le robot en (0,0), orientation nord
      leRobot = new Robot();
      // répète tant que le robot n'est pas sur le trésor
      while(carreDistanceAuTresor()!=0) {
         // indique au joueur la distance entre le robot et le trésor
         System.out.print("distance="+carreDistanceAuTresor());
         // obtient un ordre 'a' (avancer) ou 't' (tourner à droite)
         System.out.print(" ordre :");
         char ordre=Lecture.unCar();
         // exécute l'ordre
         if (ordre=='a') {leRobot.avancer();}
         else /*ordre=='t'*/ {leRobot.tournerADroite();}
      }
      // fin de partie
      System.out.println("gagné, le trésor est en <"+x0+","+y0+">");
   }
}
```

10.3 Retour sur les composants statiques et non statiques

Toutes les choses en Java sont rédigées à l'intérieur d'une classe : constantes, variables, fonctions sont toujours définies dans une classe. ***La classe est l'unité de programmation***.

Le premier rôle de la classe est un rôle de ***modularité*** : regrouper les choses concernant un même thème. Ce premier aspect est simple mais cependant important car, dans une programmation complexe, c'est en rangeant convenablement les choses qu'on parvient à s'y retrouver. Les choses qui existent en permanence et sont associées de façon unique à leur nom sont dites ***statiques***. Elles sont déclarées au niveau 0 de la classe avec le vocable **`static`**.

Le deuxième rôle éventuel d'une classe est d'être un ***modèle d'objets***. Un modèle d'objets est en quelque sorte un "moule" qui sert à frabriquer des exemplaires similaires, appelées ***instances*** de la classe, grâce à l'opération **`new`**. Dans ce cas la classe possède des composants qui doivent exister en association avec chaque instance. Ces composants n'existent pas initialement. Un exemplaire est créé pour chaque instance créée. Ces composants sont ***non-statiques*** (on pourrait également dire "dynamiques")

Ce qui surprend avec un langage à objets, tel que Java, est que les choses statiques, qui sont les choses les plus simples, nécessitent le vocable **`static`** alors que les choses non-statiques, qui sont un peu plus compliquées, ne nécessitent aucun vocable particulier. Ceci est dû au fait qu'un langage à objets veut favoriser la programmation en terme d'objets, donc de classes modèles d'objets. Dans ce cas, les composants non-statiques sont plus fréquents que les composants statiques, et une loi "naturelle" veut qu'un langage soit bref pour dire les choses les plus fréquentes.

Exemple :

L'exemple simple suivant est une classe qui est un modèle d'objets pour représenter des personnes. Chaque personne est caractérisée par un nom et une taille. Les composants **`nom`** et **`taille`** sont propres à chaque individu : ce sont des ***variables non-statiques***.

Il existe une notion de "taille normale adulte" concrétisée par une constante. La constante **`tailleNormaleAdulte`** existe en un seul exemplaire et depuis le début de l'exécution du programme : c'est une ***constante statique***.

Une variable **`plusGrandeTaille`** mémorise en permanence la plus grande des tailles des personnes existantes. Cette variable existe bien sûr en un seul exemplaire, et depuis le début de l'exécution du programme, initialisée à 0 (plus exactement à l'élément neutre de la fonction maximum sur les tailles) : c'est une ***variable statique***.

```
class Personne {

  public static final int tailleNormaleAdulte = 170;
  private static int plusGrandeTaille = 0;

  private String nom;
  private int taille;

  public Personne(String ceNom, int cetteTaille) {
    nom = ceNom; taille = cetteTaille;
    if (cetteTaille > plusGrandeTaille) {
      plusGrandeTaille=cetteTaille;
    }
  }

  public String getNom() { return nom; }

  public int getTaille() { return taille; }

  public void setTaille(int nouvelleTaille) {
    taille = nouvelleTaille;
    if (nouvelleTaille > plusGrandeTaille) {
              plusGrandeTaille=nouvelleTaille;
    }
  }

  public static getPlusGrandeTaille { return plusGrandeTaille;}
}
```

Pour éviter des affectations incohérentes de la variable **`plusGrandeTaille`**, celle-ci a été déclarée ***privée*** (**`private`**). Ainsi elle ne peut être citée que depuis le texte de la classe **`Personne`**, à l'occasion du constructeur et de la méthode **`setTaille`** de changement de taille.

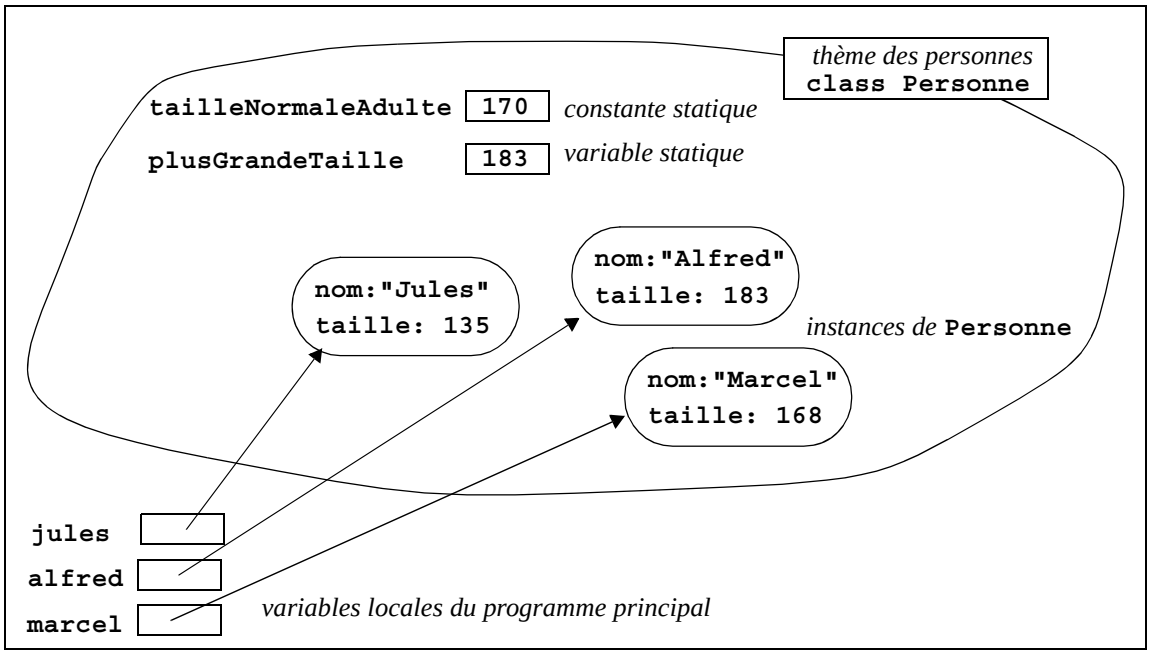
Pour pouvoir accéder à la valeur de cette variable depuis l'extérieur de la classe, on a donc programmé un ***accesseur*** **`getPlusGrandeTaille`**.

Cet accesseur ne s'applique pas à une instance particulière de **`Personne`**, c'est donc une ***fonction statique***. Depuis l'extérieur de la classe **`Personne`** on l'appelle sous la forme

`Personne.getPlusGrandeTaille()`.

Voici un exemple de programme qui utilise la classe **`Personne`** :

```
class TestPersonne {
  public static void main(String[] x) {

     // création de trois personnes
     Personne jules = new Personne("Jules", 135);
     Personne alfred = new Personne("Alfred", 183);
     Personne marcel = new Personne("Marcel", 168);

     int k = Personne.getPlusGrandeTaille()
     System.out.print("la plus grande taille est " + k);

     jules.setTaille(183);
     k = Personne.getPlusGrandeTaille()
     System.out.print("la plus grande taille est " + k);
     if (k>Personne.tailleNormaleAdulte) {
       System.out.print("c'est plus que la normale");
     }
  }
}
```



10.4 Commentaires de spécification pour une classe modèle d'objets

Les fonctions et les classes permettent de réaliser des ***abstractions***. Les services offerts par une fonction ou une classe doivent être clairement (et si possible formellement) définis. Dans les programmes, ceci se fait au moyen de ***commentaires de spécification***. Ces commentaires se placent avec l'en-tête de chaque fonction, comme nous l'avons fait systématiquement ici. Ces commentaires comportent trois rubriques (pas nécessairement toutes présentes) : ***prérequis***, ***effet*** et ***résultat*** .

Rubrique *prérequis* **`// prérequis :` *condition***

Cette rubrique indique les conditions que doivent satisfaire les paramètres de la fonction et l'état de l'objet (l'état de **`this`**) lorsqu'il s'agit d'une méthode d'objet. La condition est donc une formule logique (un prédicat). Cette condition n'a pas à être testée par le programme de la fonction : il est de la responsabilité de l'utilisateur de respecter cette condition. En d'autres termes, tant pis pour l'utilisateur s'il ne respecte pas le prérequis, la fonction peut faire n'importe-quoi dans ce cas sans que le programmeur de la fonction en soit responsable.

Dans le cas d'une méthode d'objet, l'état de l'objet (**`this`**) ne doit faire en aucune façon usage des composants de l'objet, qui ne sont que des moyens concrets de mise en œuvre. Il ne faut utiliser que les notions abstraites de l'objet.

Par exemple, pour une liste de personnes représentée par un tableau et une variable comptabilisant le nombre d'éléments de la liste :

```
class ListeDePersonnes {
  private Personne[] lesElements; private int nbElements;
  public ListeDePersonnes() { // liste vide
    lesElements= new Personnne[1000]; nbElements=0;
  }
  ...
  public Personne laPremiere(){
  // prérequis : this n'est pas vide
  // résultat : la première personne de this
  }
  ...
}
```

Le prérequis "**`this n'est pas vide`**", est acceptable car "être vide" fait partie de la notion abstraite de liste. La condition **`nbElements>0`** serait en revanche innaceptable car elle fait référence à la mise en œuvre.

Rubrique *effet* **`// effet :` *changements d'état ou interractions avec l'extérieur***

Ici encore, pour une méthode d'objet qui change l'état de **`this`**, ou une fonction qui change l'état d'un objet de la classe passé en paramètre, le changement d'état doit être indiqué en termes de l'abstraction représentée par la classe, et surtout pas en terme de changement d'état des composants. Par exemple, pour une classe **`ListeDePersonne`**, la méthode qui permet d'ajouter une personne à la liste serait ainsi spécifiée :

```
  public void ajouter(Personne p) { // effet : ajoute p à this
    lesElements[nbElements]=p; nbElements++;
  }
```

Et surtout pas : "range **`p`** dans **`lesElements[nbElements]`** et incrémente **`nbElements`**" car les composants **`lesElements`** et **`nbElements`** ne font pas partie des choses sensibles à l'utilisateur. Ce ne sont que des moyens de mise en œuvre, ils pourraient ne pas exister avec un autre mise en œuvre.

Rubrique *résultat* **`// résultat :` *expression indiquant le résultat rendu***

l'expression qui indique ce que vaut le résultat ne doit pas faire référence aux moyens utilisés pour le calculer. Il ne doit faire référence qu'aux paramètres, à l'état abstrait de **`this`** (pour une méthode d'objet) ou d'un paramètre de la classe, et à toutes les fonctions connues par ailleurs, mathématiques ou spécifiques au domaine traité.

Exercice 10.1 Représentation des complexes au moyen d'une (vraie) classe

Dans un exercice précédent nous avions représenté les nombres complexes au moyen d'une structure. Ce n'est pas la bonne façon de faire. Il faut ***totalement abstraire les données concrètes*** qui représentent un nombre complexe (x et y dans cet exemple), rédiger les fonctions de manipulation des complexes ***à l'intérieur de cette classe*** et ne rendre visible aux utilisateurs que ces fonctions.

Rédiger la classe **`Complexe`** dans cet esprit.

Pour satisfaire aux goûts divers et variés des utilisateurs potentiels, on pourra réaliser deux versions de chaque fonction de calcul :

une version ***fonction statique***, par exemple :

```
public static Complexe add(Complexe z1, Complexe z2)
// résultat : la somme de z1 et de z2
```

une version ***méthode d'obje***t :

```
public Complexe add(Complexe z)
// résultat : la somme de this et de z
```

Exercice 10.2 Représentation de personnes

Rédiger une classe **`Personne`** qui représente une personne dotée des attributs suivants : ***nom***, ***sexe***, ***adresse***, ***année de naissance***.

Le nom et l'adresse sont des chaînes de caractères (quelconque, on ne cherche pas à vérifier la vraisemblance d'être un nom ou une adresse). Le sexe a deux valeurs possibles, **`masculin`** ou **`feminin`**. L'année de naissance est donnée sous la forme d'un entier.

Les composants de données seront privés. On donne accès en consultation à ces attributs au moyen d'accesseurs.

La classe doit offrir :

un ***constructeur trivial***, qui reçoit en paramètres la valeurs des attributs (nom, sexe, adresse et année de naissance),

Une méthode **`toString`** qui rend en résultat une chaîne de caractères qui visualise "en clair" les attributs de la personne.

Une fonction statique **`laPlusJeune`** telle que, ***p1*** et ***p2*** étant des personnes, **`laPlusJeune(`*`p1`*`,`*`p2`*`)`** est la personne la plus jeune parmi ***p1*** et ***p2*** (***p1*** en cas d'égalité des âges).

Tester cette classe dans un programme principal (**`TestPersonne`**) qui crée une personne ***toto***,et une personne ***x*** et imprime la plus jeune de ces deux personnes.

Exercice 10.3 Abstraction : collection de personnes

On dispose d'une classe **Personne** ainsi spécifiée :

```
class Personne {// représentation d'individus

  public Personne(String n, int s, String adr, int an)
  // construit une personne de nom n, de sexe s, d'adresse adr
  // et d'année de naissance an

  public static Personne lire()
  // effet : lit au clavier les caractéristiques d'une personne
  // résultat : une nouvelle personne avec ces caractéristiques

  public String toString()
  // résultat : la chaîne "en clair" caractérisant la personne

  public int getSexe()
  // résultat : sexe de this

  public String getNom()
  // résultat : nom de this

  public String getAdresse()
  // résultat : adresse de this

  public int getAnneeNaissance()
  // résultat : année de naissance de this

  public static Personne laPlusJeune(Personne p1, Personne p2)
  // résultat : la personne la plus jeune de p1 et p2,
  // p1 en cas d'égalité
}
```

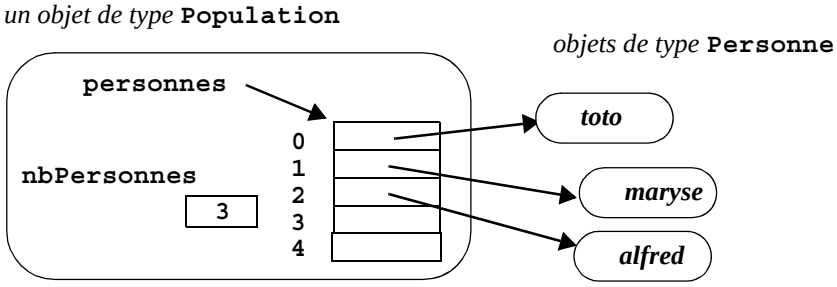
Une ***collection de personnes*** est un objet qui permet d'enregistrer un nombre quelconque d'éléments de type **Personne**. Une telle collection sera réalisée par la classe **Population** dont voici la spécification :

```
class Population{

  public Population()
  // constructeur : population vide

  public void ajouter(Personne p)
  // effet : ajoute p à this (ne fait rien si p s'y trouve déjà)

  public int cardinal()
  //résultat : le nombre de personnes dans this

  public Personne obtenir(int i)
  // prérequis : 0<=i<nbPersonnes()
  // résultat : la ième personne de la collection. Les personnes
  // ne sont pas classées, mais on garantit que obtenir(i) est
  // toujours la même personne et que si i!=j,
  // obtenir(i)!=obtenir(j)
}
```

Mise en œuvre

On décide de représenter une population au moyen d'un tableau **`personnes`** d'éléments de type **`Personne`** et d'une variable entière **`nbPersonnes`**.

Correspondance abstrait-concret : La variable **`nbPersonnes`** indique combien de personnes sont dans la collection. Les personnes présentes dans la collection sont rangées dans le tableau **`personnes`** depuis l'indice **`0`** jusqu'à l'indice **`nbPersonnes-1`**. Le tableau **`personnes`** doit toujours avoir la taille suffisante pour contenir les personnes de la collection. Pour cela, il est créé initialement avec une taille arbitraire **`tailleInitiale=5`**, puis chaque fois que la taille du tabeau devient insuffisante, on le "remplace" par un tableau de taille double.



Rédiger cette mise en œuvre.

Exercice 10.4 Abstraction : représentation des nombres rationnels

On se propose de rédiger une classe **`Rationnel`** qui représente les nombres rationnels (c'est-à-dire les fractions ***p/q***) dotée des opérations classiques : somme, produit, quotient, test d'égalité. Voici la ***spécification*** de la classe **`Rationnel`** :

```
class Rationnel {

  public Rationnel(int a, int b)
  // prérequis : b!=0 (a et b de signe quelconque)
  // constructeur : crée un rationnel égal à a/b

  public Rationnel(int a)
  // constructeur : crée un rationnel égal à a

  public String toString()
  // résultat : chaîne de caractère figurant this,
  //            par exemple "-235/12" ou "235"

  public static boolean egal(Rationnel x, Rationnel y)
  // résultat : indique si x et y sont égaux

  public static Rationnel somme(Rationnel x, Rationnel y)
  // résultat : somme de x et y

  public static Rationnel produit(Rationnel x, Rationnel y)
  // résultat : produit de x par y

  public static Rationnel quotient(Rationnel x,Rationnel y)
  // prérequis : y différent de zéro
  // résultat : quotient de x par y
}
```

Mise en œuvre :

On décide de représenter un rationnel au moyen de deux entiers, le numérateur **`p`** et le dénominateur **`q`**. Pour simplifier, les entiers seront réalisés au moyen du type **`int`**.

Afin de faciliter le test d'égalité et pour profiter au maximum de l'intervalle de représentation des entiers, on impose les contraintes suivantes sur **`p`** et **`q`** :

Le dénominateur **`q`** est >0. Le signe du nombre rationnel est donc celui de l'entier **`p`** (qui est quelconque).

Le rationnel 0 est représenté par **`p`**=0 et **`q`**=1.

Pour un rationnel différent de 0, **`p`** et **`q`** sont tels que la fraction **`p/q`** est ***irréductible*** (**`p`** et **`q`** sont premiers entre eux). Par exemple, 27/18 devra être simplifié en 3/2.

Rédiger cette mise en œuvre.

Exercice 10.5 Abstraction : tables de correspondance

On se propose de réaliser une structure de données ***table de correspondance*** permettant d'associer des ***valeurs*** à des ***clés***. Les clés seront des chaînes de caractères et les valeurs des nombres entiers. Ceci permet par exemple de représenter le résultat d'une course en prenant pour clés les noms des coureurs et pour valeurs les temps réalisés :

`<("`dupont**`",28), ("alfred",45), ("durand",12)>`**

La classe **`TableDeCorrespondance`** doit offrir les fonctionnalités suivantes :

Constructeur permettant de créer une table vide.

Création d'une association **`(`*clé*`,`*valeur*`)`**. Si la clé se trouve déjà dans la table, cela remplace l'ancienne valeur associée par la nouvelle.

Retrait d'une association indiquée par sa clé. Ne fait rien si la clé est absente.

Recherche de la valeur associée à une clé. Si la clé est absente, le résultat est une valeur conventionnelle **`valeurAbsente`**, par exemple 1000000000.

Commencer par rédiger les ***spécifications*** de cette classe : données, constructeur et méthodes publiques avec commentaires ***précis*** indiquant leur ***fonctionalités***.

Mise en œuvre

On décide de représenter une table de correspondance au moyen d'un tableau **`T`** de paires **`(`*clé*`,`*valeur*`)`** et d'une variable entière **`nbElements`** qui indique le nombre d'associations présentes dans la table. Les paires seront réalisées au moyen d'une classe interne **`Paire`**. La taille du tableau, constante statique, fixera la capacité maximum de la table. Un dépassement de capacité provoquera l'impression d'un message d'erreur.

On rédigera une fonction auxiliaire :

`int indiceDeCle(String c)`

qui rend l'indice de l'association de clé **`c`** si elle existe, ou **`nbElements`** si la clé **`c`** est absente.

Test

Rédiger un programme principal de test qui

construit, par ajouts successifs, la table :

`<("dupont",28), ("alfred",45), ("durand",12)>`

recherche les valeurs associées à **`"durand"`** et à **`"toto"`**,

modifie la valeur associée à **`"durand"`**, supprime **`"alfred"`**,

ajoute l'association **`("toto",66)`**,

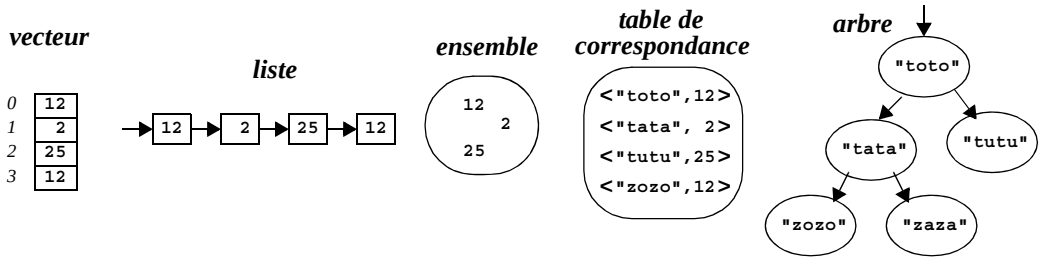
recherche les valeurs associées à **`"durand"`**, **`"alfred"`** et **`"toto"`**.

CHAPITRE 11 Structures de données : listes

11.1 Notion de structure de données

Parmi les types de données on distingue ceux qu'on appelle des ***structures de données***. Comme le terme l'indique, une structure de données permet de structurer des données : elle offre une gamme d'opérations pour "emmagasiner" des données et une gamme d'opérations pour "retrouver" les données emmagasinées. À titre d'exemples voici quelques types usuels qui méritent l'appellation "structure de données", accompagnés d'une description approximative des opérations qu'ils offrent :

- ***tableau*** (ou, plus généralement ***vecteur***) : suite de données accessibles directement au moyen d'un indice entier.
- ***liste*** : suite de données accessibles l'une après l'autre selon leur rang dans la suite.
- ***ensemble*** : collection de données "en vrac" telle que toutes les données ont une valeur différente, dotée de l'opération "test d'appartenance" et d'un moyen de parcourir la collection.
- ***table de correspondance*** : collection de paires de la forme <clé,valeur> qui permet de trouver la valeur associée à une clé.
- ***arbre*** : collection de données appelées nœuds, composés d'une valeur et d'une collection de liens vers des nœuds appelés fils.



Chacune de ces structures de données donne lieu à :

- Une ***définition abstraite***, encore appelée ***spécification***, qui décrit les opérations qu'elle offre, sans aucune allusion à la moindre réalisation concrète des mécanismes mis en œuvre pour réaliser ces opérations.
- Une ou plusieurs ***réalisations concrètes***, encore appelées ***mises en œuvre***[1], au moyen de données de types existants et de procédures.

Pour la mise en œuvre, on utilise un type classe qui permet de définir des objets abstraits. Nous avons déjà vu que l'encapsulation permet d'isoler la mise en œuvre et l'utilisation.

1. On dit ***implementation*** en anglais, et on dit ***implémentation*** (avec accent sur le "e") en franglais.

11.2 Spécification du type liste

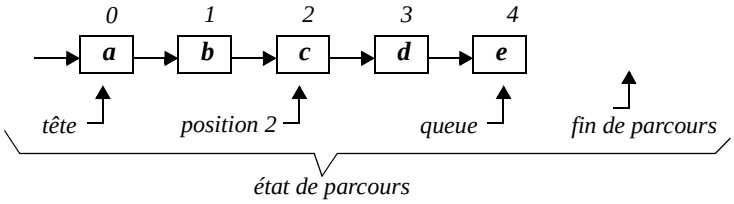
On s'intéresse ici à des ***listes modifiables*** : une liste est un objet dont l'état est une ***suite finie***, éventuellement vide, de données de même type : e_0, e_1... e_{n-1}

Le nombre d'éléments, *n*, s'appelle la ***longueur*** de la liste. La longueur de la ***liste vide*** est nulle.

La position 0 s'appelle la ***tête***, la position *n*-1 s'appelle la ***queue***.

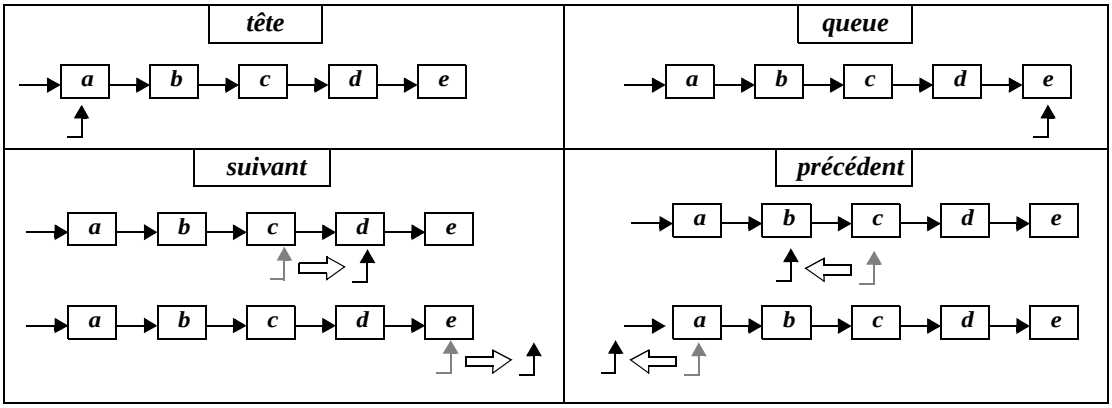
Pour se positionner sur les éléments et accéder à ces éléments on utilise un objet spécifique que nous appellerons un ***parcours de liste*** ou plus simplement ***parcours***. L'état d'un parcours est :

- soit une position dans la liste (0... *n*-1)
- soit un état particulier, ***fin de parcours***, qui ne correspond à aucune position ("avant le début" ou "après la fin").



Au sujet d'un parcours on peut :

- le ***positionner en tête*** de la liste (position *0* si elle existe, ***fin de parcours*** si la liste est vide)
- le ***positionner en queue*** de la liste (position *n-1* si elle existe, ***fin de parcours*** si la liste est vide)
- passer à la ***position suivante*** de la position courante. Ceci positionne en ***fin de parcours*** si le parcours est en ***queue***.
- passer à la ***position précédente*** de la position courante. Ceci positionne en ***fin de parcours*** si le parcours est en ***tête***.



Étant donné un état de parcours, on peut :

- ***ajouter*** un élément par insertion à la suite de la position courante du parcours. L'état de parcours se positionne alors sur le nouvel élément.
- ***supprimer*** l'élément courant du parcours. L'état de parcours se positionne alors sur l'élément qui suivait l'élément supprimé, ou en ***fin de parcours*** si le parcours était en ***queue***.
- ***modifier la valeur*** de l'élément courant du parcours,
- ***consulter la valeur*** de l'élément courant du parcours.

Plus précisément, voici les méthodes offertes par les classes **`ListeDEntiers`** (listes) et **`Parcours`** (parcours de listes).

Nous présentons ici des listes d'entiers. Mais bien évidemment on peut imaginer les mêmes fonctionnalités pour des listes de n'importe quel type d'éléments.

```
public class ListeDEntiers {

  public class Parcours { //-----------------------------------

    public void tete()
    // effet : positionne this en tête, estEnFin si liste vide

    public void queue()
    // effet : positionne this en queue, estEnFin si liste vide

    public void suivant()
    // prérequis : this non estEnFin
    // effet : positionne this sur l'élément suivant

    public void precedent()
    // prérequis : this non estEnFin
    // effet : positionne this sur l'élément précédent

    public boolean estEnFin()
    // résultat : indique si this est en fin de parcours

    public int elementCourant()
    // prérequis : this non estEnFin
    // résultat : l'élément courant de this

    public void modifElement(int nouvelElement)
    // prérequis : this non estEnFin
    // effet : remplace l'élément courant de this par nouvelElement

    public void ajouteElement(int nouvelElement)
    // effet : insère nouvelElement après l'élément courant de this,
    // en tête si estEnFin, positionne this sur l'élément ajouté

    public void retireElement()
    // prérequis : this non estEnFin
    // effet : retire l'élément courant de this, le suivant de
    // l'élément retiré, s'il existe, devient l'élement courant,
    // sinon this passe en fin de parcours
  } // ----------------------------------------------------------
```

```
    public ListeDEntiers() // constructeur : liste vide

    public boolean estVide() // indique si this est vide

    public void ajouteEnTete(int nouvelElement)
    // effet : ajoute nouvelElement en tête

    public void ajouteEnQueue(int nouvelElement)
    // effet : ajoute nouvelElement en queue

    public int retireEnQueue()
    // prérequis : this n'est pas vide
    // effet : retire de this son dernier élément
    // résultat : l'élément retiré

    public int retireEnTete()
    // prérequis : this n'est pas vide
    // effet : retire de this son premier élément
    // résultat : l'élément retiré

    public String toString()
    // résultat : représentation en clair de this,
    // de la forme : < 12 | 23 | 45 | 12 | 6>, <> pour la liste vide

   public static ListeDEntiers aPartirDe(int[] T){
    // résultat : une nouvelle liste contenant les éléments de T

    public Parcours nouveauParcours()
    // résultat : un nouveau parcours initialisé en tête de this

    public Parcours nouveauParcours(Parcours p)
    // résultat : un nouveau parcours initialisé dans l'état de p
}
```

Pour des raisons de modularité, la classe **`Parcours`** est une ***classe interne*** à la classe **`ListeDEntiers`**. Nous verrons plus loin la notion de classe interne : c'est simplement une classe définie à l'intérieur d'une autre classe.

11.3 Exemples d'utilisation d'une liste

```
class utilisationDeListe {
  public static void main(String[] arg) {
     Déclaration et création d'une liste initialisée à vide :
     ListeDEntiers uneListe = new ListeDEntiers();
     Introduction de valeurs par la méthode ajouteEnQueue. Les instructions suivantes
     construisent une liste contenant les carrés des nombres entiers de 1 à 9 :
     for (int i=1; i<=9; i++) {uneListe.ajouteEnQueue(i*i);}
     Recherche dans la liste du plus grand multiple d'un nombre donné.
     Pour cela il suffit de parcourir la liste à partir de la queue.
     System.out.print("recherche du plus grand multiple de j= ");
     int j = Lecture.unEntier();
     ListeDEntiers.Parcours p = uneListe.nouveauParcours();
     p.queue();
     while (!p.estEnFin()&&p.elementCourant()%j!=0) {
       p.precedent();
     }
     Cet exemple illustre l'utilisation d'un parcours de liste :
     la classe Parcours étant interne à la classe ListeDEntiers, depuis l'extérieur de la
     classe il faut la nommer par :
     ListeDEntiers.Parcours
     Après la boucle, si le parcours p est en fin, cela signifie qu'il n'y a pas de multiple de j.
     Sinon l'élément courant du parcours p est la valeur cherchée :
     if (p.estEnFin()) {
       System.out.println(j+" n'a pas de multiple dans la liste");
     }
     else {
       System.out.print("plus grand multiple de "+j+ " :");
       System.out.println(p.elementCourant());
      }
     Modification d'éléments :
     l'exemple suivant ajoute 1 aux multiples de 3 dans la liste.
     p.tete();
     while (!p.estEnFin()) {
       int v=p.elementCourant();
       if (v%3==0) {p.modifElement(v+1);}
       p.suivant();
     }
     Impression en clair de la liste :
     System.out.println(uneListe.toString());
     Suppression d'éléments :
     suppression de tous les multiples de 4 de la liste :
     p.tete();
     while (!p.estEnFin()) {
       if (p.elementCourant()%4==0) {p.retireElement();}
       else {p.suivant();}
     }
  }
}
```

Autres exemples : fonctions manipulant des listes

Comme tout objet, une liste peut être passée en paramètre ou rendue en résultat de fonction. Nous donnons dans ce qui suit, quelques exemples de fonctions qui manipulent des listes.

```
class DiversesProceduresManipulantDesListes {

static ListeDEntiers lectureListe() {
// effet : lit au clavier des entiers terminés par -1
// résulat : une nouvelle liste contenant les entiers lus
     ListeDEntiers resultat = new ListeDEntiers();
     int k=Lecture.unEntier();
     while (k!=-1) {
        resultat.ajouteEnQueue(k);
        k=Lecture.unEntier();
     }
     return resultat;
  }

  static boolean estDansListe(int k, ListeDEntiers L) {
  // résultat : indique si k est un élément de L
     boolean kTrouveDansL=false;
     ListeDEntiers.Parcours p= L.nouveauParcours();
     while (!p.estEnFin() && !kTrouveDansL){
        if(p.elementCourant()==k) {kTrouveDansL=true;}
        else {p.suivant();}
     }
     return kTrouveDansL;
  }

  Procédure qui remet à vide une liste.
  static void viderListe(ListeDEntiers L) {
  // effet : rend L vide
     while (!L.estVide()) {L.retireEnTete();}
  }


  public static void main(String[] arg) {
  Test des procédures précédentes :
     System.out.println(
          "entrer une séquence d'entiers terminée par -1");
     ListeDEntiers L = lectureListe();
     if (estDansListe(12,L)) {
          System.out.println("12 est dans la liste");
     }
     else {System.out.println("12 n'est pas dans la liste");}
     viderListe(L);
     System.out.println("apres vidage, L = " + L.toString());
  }

}
```

11.4 Mise en œuvre des listes

Les exemples précédents montrent que l'on peut utiliser les listes sans avoir la moindre idée de la façon dont elles sont réalisées. Nous insistons sur le fait qu'il n'est pas utile de savoir ***comment*** quelque chose est réalisé pour pouvoir l'utiliser : c'est la base même de toute abstraction. Il est même parfois préférable de ne pas savoir comment fonctionne quelque chose pour l'utiliser : ainsi on n'est pas tenté d'utiliser des propriétés qui n'appartiennent pas à la structure de donnée abstraite mais seulement à une mise en œuvre particulière.

Dans ce paragraphe nous nous intéressons à la mise en œuvre au moyen de classes. Ce faisant nous "changeons de casquette" : dans les paragraphes précédents nous avions la casquette "utilisateur" des listes, nous coiffons maintenant la casquette "réalisateur" ou "implémenteur" des listes.

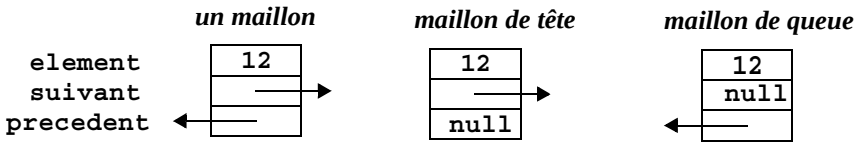
11.4.1 Représentation des éléments d'une liste

Les listes peuvent être réalisées de plusieurs façons. Les deux façons traditionnelles sont :

- à base de ***structures chaînées*** (maillons chaînés),
- à base de ***tableau***.

Nous étudions dans ce qui suit la mise en œuvre à base de maillons chaînés.Pour chaque élément de la liste il est créé une structure à trois champs, appelée un ***maillon***, composée de :

- **`element`** : la valeur de l'élément pour lequel ce maillon existe,
- **`suivant`** : la référence au maillon associé à l'élément suivant dans la liste,
- **`precedent`** : la référence au maillon associé à l'élément précédent dans la liste.



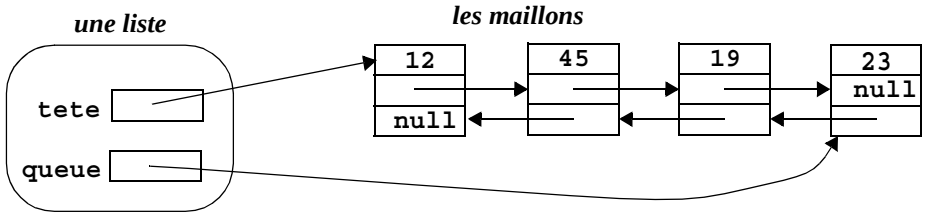
Dans le cas d'un maillon associé à l'élément de tête, **`precedent`** vaut **`null`** (il n'y a pas de précédent) et dans le cas d'un maillon associé à l'élément de queue, **`suivant`** vaut **`null`** (il n'y a pas de suivant). La définition de la structure **`Maillon`** est :

```
class Maillon {
     int element; Maillon suivant; Maillon precedent;
     Maillon(Maillon s, Maillon p, int e) {
        suivant=s; precedent=p; element=e;
     }
}
```

La liste est alors représentée par deux références :

- **`tete`** : la référence au maillon associé à l'élément de tête,
- **`queue`** : la référence au maillon associé à l'élément de queue.

La figure suivante illustre la représentation de la liste <12, 45, 19, 23> :



La référence à l'élément de queue est "redondant" : on pourrait s'en passer et le retrouver en parcourant les maillons à partir de la tête. Mais on la conserve pour une meilleure efficacité car elle permet d'avoir un accès immédiat à la queue de la liste.

Il faut bien sûr se méfier du cas particulier de la ***liste vide***. La liste vide est simplement représentée par **`tete=null`** et **`queue=null`** (il n'y a aucun élément, donc aucun maillon).

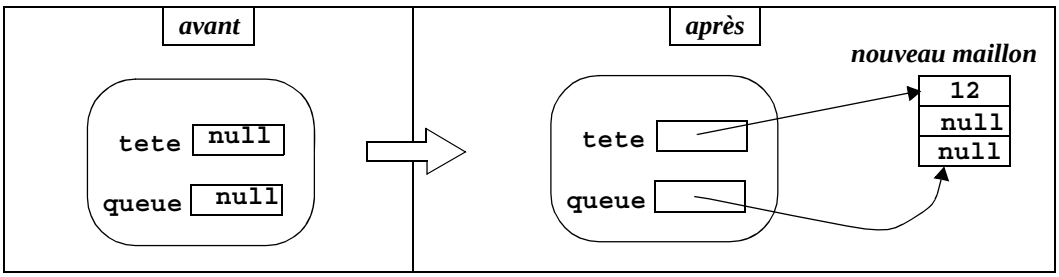
La définition des données de la classe **`ListeDEntiers`** est donc :

```
public class ListeDEntiers {

  private Maillon tete;
  private Maillon queue;

  public ListeDEntiers() { // constructeur liste vide
     tete=null; queue=null;
  }
   ...
```

La liste vide est caractérisée par **`tete==null`** et **`queue==null`**. Il suffit de tester l'une des deux égalités à **`null`**, car l'une implique l'autre (si la programmation de la liste est correcte). Le test **`estVide`** se programme donc simplement ainsi :

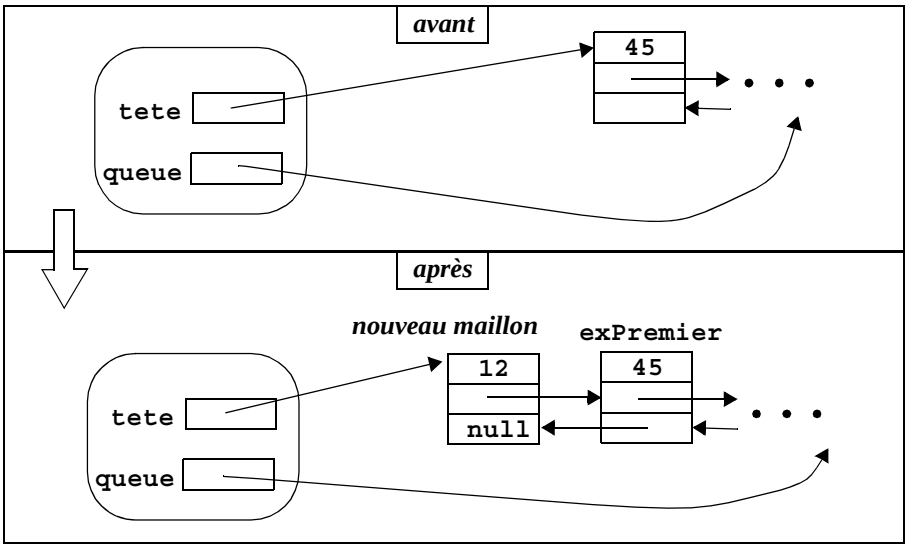
```
public boolean estVide() {return tete==null;}
```

Un peu plus compliqués sont les ajouts et retraits d'un élément. Considérons l'ajout d'un élément en tête de liste. Il faut distinguer le cas où la liste est vide :



`tete` et **`queue`** doivent recevoir la référence au nouveau maillon créé pour l'élément rajouté.

et le cas où la liste contient déjà des éléments :



La variable **tete** et le champ **precedent** de **exPremier** (le maillon de tête avant l'ajout) doivent recevoir la référence au nouveau maillon. La variable **queue** reste inchangée dans ce cas.

Cette analyse conduit à la rédaction suivante pour la méthode **ajouteEnTete** :

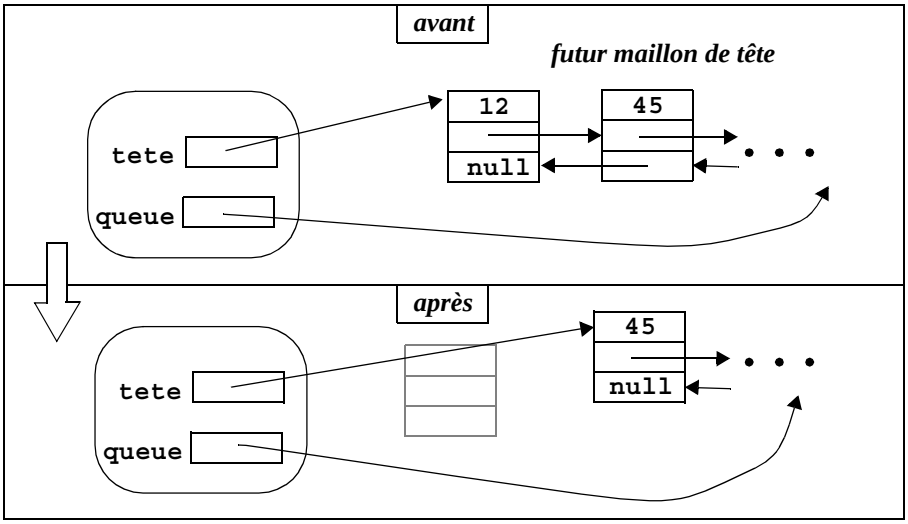
```
public void ajouteEnTete(int nouvelElement) {
   if (tete==null) { // cas liste vide
      tete = new Maillon(null,null,nouvelElement);
      queue = tete;
   }
   else { // chaîne en tete
      Maillon exPremier = tete;
      tete = new Maillon(exPremier,null,nouvelElement);
      exPremier.precedent=tete;
   }
}
```

La méthode **ajouteEnQueue** s'obtient par une analyse similaire. Il suffit d'échanger les vocables **tete** et **queue** ainsi que **precedent** et **suivant** :

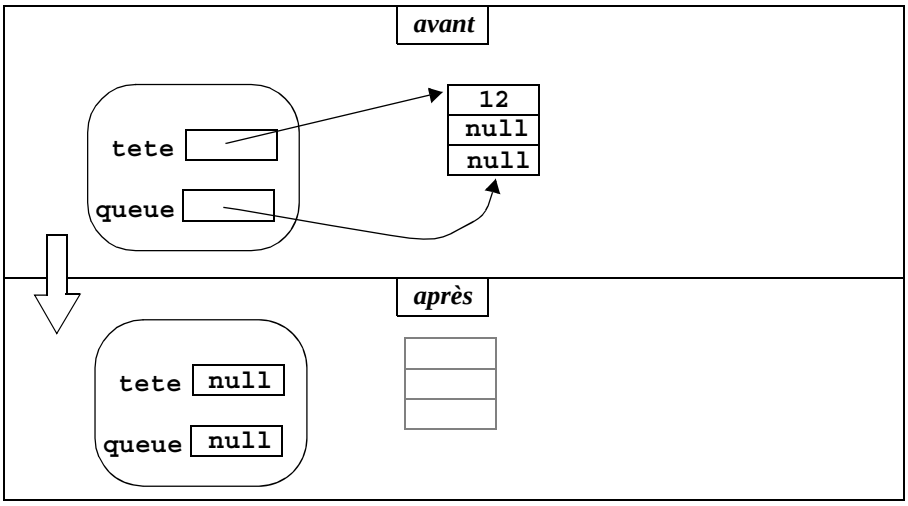
```
public void ajouteEnQueue(int nouvelElement) {
   if (queue==null) { // cas liste vide
      queue = new Maillon(null,null,nouvelElement);
      tete = queue;
   }
   else { // chaine en queue
      Maillon exDernier = queue;
      queue = new Maillon(null,exDernier,nouvelElement);
      exDernier.suivant=queue;
   }
}
```

Pour le retrait de l'élément de tête, **tete** reçoit la référence au maillon suivant de **tete**. Il faut également gérer correctement le chaînage des précédents. Dans le cas où la liste ne devient pas

vide, le champ précédent du nouveau maillon de tête devient **`null`** :



et si la liste devient vide c'est **`queue`** qui devient **`null`** :



Cette analyse conduit à la programmation suivante pour la méthode **`retireEnTete`** :

```
public int retireEnTete() {
    int resul = tete.element;
    tete = tete.suivant;
    if (tete!=null) {tete.precedent=null;} else {queue = null;}
    return resul;
  }
```

La programmation de **`retireEnQueue`** est similaire :

```
public void retireEnQueue() {
    int resul = queue.element;
    queue = queue.precedent;
    if (queue!=null) {queue.suivant=null;} else {tete = null;}
    return resul;
  }
```

Remarque sur les erreurs

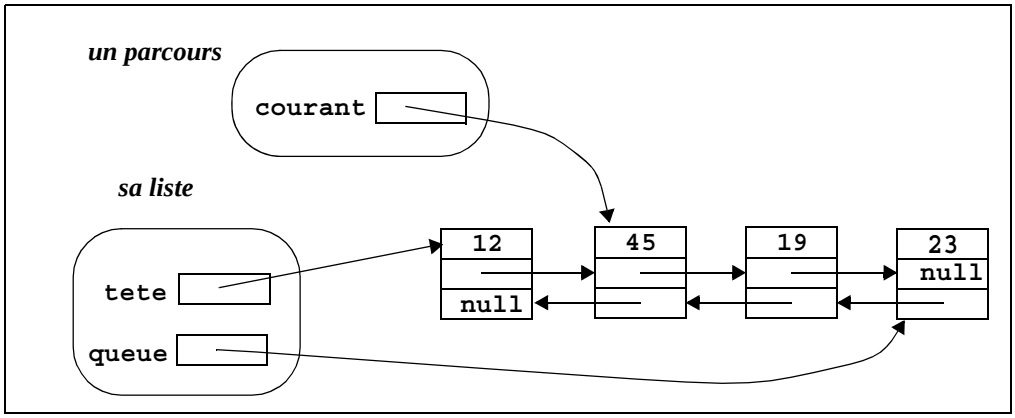
La tentative de retrait d'un élément de la liste vide est considérée comme une erreur. Nous n'avons pas cherché ici à détecter ces erreurs. Il y a plusieurs raisons à cela :

- Une raison est de ne pas alourdir le texte des méthodes susceptibles de provoquer une erreur.
- Une autre est de considérer que ***c'est à l'utilisateur*** de la classe de ne pas provoquer d'erreur, et on lui a clairement indiqué dans les spécifications quelles sont les opérations interdites.
- Une dernière raison est que, étant donnée la mise en œuvre, ces erreurs seront détectées car elles conduisent à tenter d'accéder à un maillon désigné par la référence nulle, ce qui provoque une erreur "**`null pointer exception`**" signalée par le langage à l'exécution.

11.4.2 Parcours de listes

11.4.2.1 Représentation d'un parcours

Un parcours de liste se fait au moyen d'un objet de la classe **`Parcours`**. Avec la représentation de la liste par maillons chaînés, un état de parcours est simplement la référence au maillon associé à l'élément courant du parcours. Nous appelons **`courant`** cette référence. La figure suivante illustre un parcours dont l'état désigne le deuxième élément de sa liste :




```
public class Parcours {

   private Maillon courant;

   Parcours() {courant=tete;} // Parcours initialisé au début
   Parcours(Parcours p) { // Parcours initialisé avec l'état de p
      courant=p.courant;
   }
   ...
}
```

11.4.2.2 Test de fin d'un parcours

L'état **`estEnFin`**, état de parcours qui ne désigne aucun élément, est naturellement représenté par **`courant=null`**. D'où la programmation du test **`estEnFin`** :

```
public boolean estEnFin() {return courant==null;}
```

11.4.2.3 Méthodes de déplacement d'un parcours

Les positionnements en tête et en queue de liste se font simplement en affectant **`courant`** respectivement à la valeur de **`tete`** et de **`queue`** de la liste :

```
public void tete() {courant=tete;}

public void queue() {courant=queue;}
```

Les positionnements sur l'élément suivant et précédent donnent lieu à une erreur si le parcours est en fin. Sinon, il suffit d'affecter **`courant`** respectivement par la référence au maillon suivant ou précédent, c'est-à-dire par les valeurs contenues dans les champs **`suivant`** ou **`precedent`** du maillon courant :

```
public void suivant() {courant=courant.suivant;}


public void precedent() {courant=courant.precedent;}
```

11.4.2.4 Méthodes d'accès à l'élément courant

Les accès à l'élément courant, en lecture comme en modification, donnent lieu à une erreur si le parcours est en fin. Sinon, il suffit d'accéder au champ **`element`** du maillon courant :

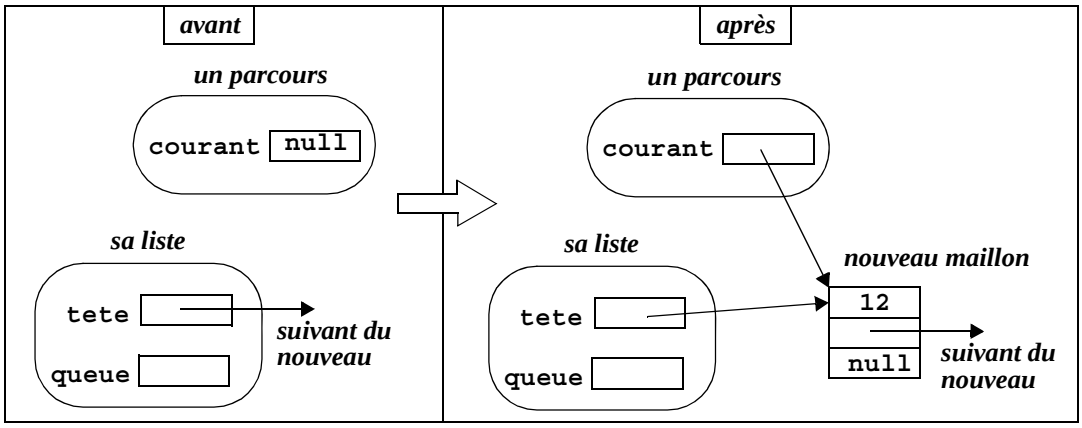
```
public int elementCourant() {return courant.element;}


public void modifElement(int nouvelElement) {
     courant.element=nouvelElement;
}
```

11.4.2.5 Méthodes d'ajout et de retrait d'éléments

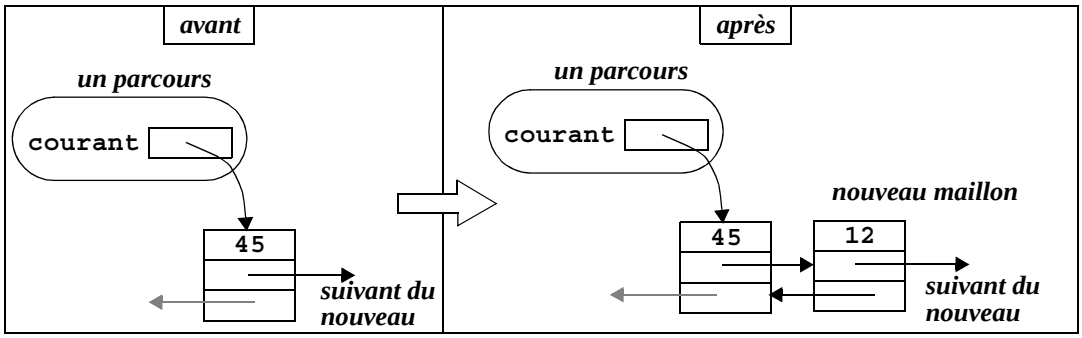
Ajout d'un élément

L'ajout d'un élément est assez compliqué à cause des cas particuliers.

Si le parcours est en fin, l'élément doit être ajouté en tête. Dans ce cas le suivant du nouveau maillon est le contenu de **`tete`**, son précédent est **`null`** et **`tete`** doit recevoir la référence au nouveau maillon :

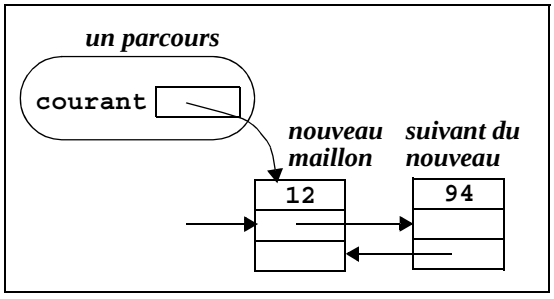


Si le parcours n'est pas en fin, l'élément doit être inséré à la suite de l'élément courant. Dans ce cas le suivant du nouveau maillon est le suivant du courant, son précédent est le maillon courant et le champ **`suivant`** du maillon courant doit recevoir la référence au nouveau maillon :

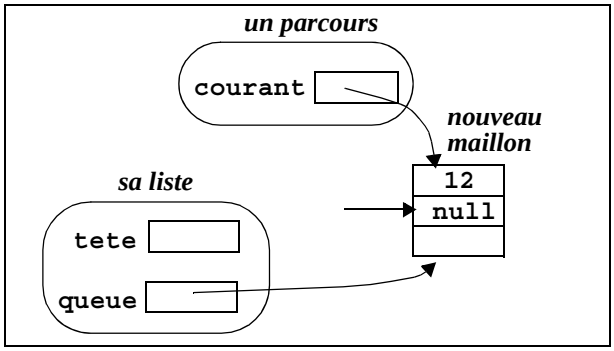


D'autre part, dans tous les cas, le nouvel élément doit devenir l'élément courant du parcours. Cela est réalisé en affectant **`courant`** avec la référence au nouveau maillon.

Il reste à gérer correctement le chaînage permettant le passage au précédent. Dans le cas où le suivant du nouveau maillon n'est pas **`null`**, le champ **`precedent`** du suivant du nouveau maillon reçoit la référence au nouveau maillon.



Dans le cas où le suivant du nouveau maillon est **null**, cela signifie que le nouvel élément est l'élément de queue. C'est donc la variable **queue** qui reçoit la référence au nouveau maillon.



Cette analyse conduit à la programmation suivante :

```
public void ajouteElement(int nouvelElement) {
   Maillon suivantDuNouveau; Maillon nouveau;
   if (courant==null) { // chaine en tête
      suivantDuNouveau=tete;
      nouveau = new Maillon(tete,null,nouvelElement);
      tete=nouveau;
   }
   else { // chaine sur courant
      suivantDuNouveau=courant.suivant;
      nouveau = new Maillon(suivantDuNouveau,courant,nouvelElement);
      courant.suivant=nouveau;
   }
   courant=nouveau;
   if (suivantDuNouveau!=null) {
      suivantDuNouveau.precedent=courant;
   }
   else { // nouvel élément de queue
      queue=courant;
   }
}
```

Retrait d'un élément

Pour le retrait d'un élément, il suffit de gérer convenablement les chaînages *suivant* et *précédent*.

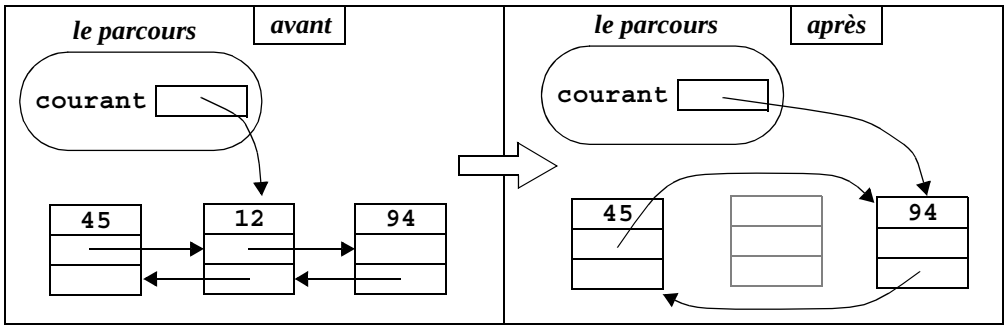
Pour le chaînage *suivant*, si le précédent du courant existe, il faut affecter son champ **suivant** avec la valeur du champ **suivant** du courant. S'il n'existe pas, c'est-à-dire s'il vaut **null**, cela correspond au cas où on retire l'élément de tête. Le suivant du courant devient la nouvelle tête et c'est donc la variable **tete** de la liste qui doit recevoir la valeur du champ **suivant** du courant.

Le principe est similaire pour le chaînage *précédent*. Si le suivant du courant existe, il faut affecter son champ **precedent** avec la valeur du champ **precedent** du courant. S'il n'existe pas, c'est-

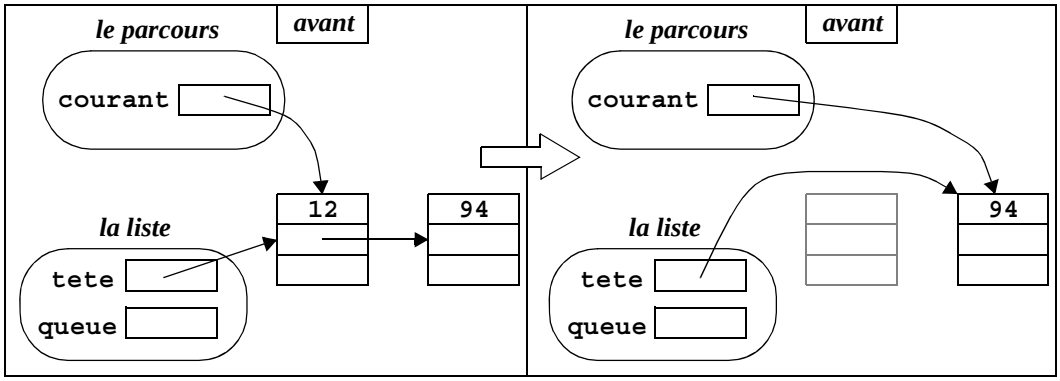
à-dire s'il vaut **`null`**, cela correspond au cas où on retire l'élément de queue. Le précédent du courant devient la nouvelle queue et c'est donc la variable **`queue`** de la liste qui doit recevoir la valeur du champ **`precedent`** du courant (ouf !).

Le parcours doit se positionner sur l'élément suivant l'élément supprimé (ou en fin de parcours si on vient de supprimer la queue). Dans tous les cas, cela se fait simplement en affectant la variable **`courant`** du parcours avec la référence au suivant de l'élément supprimé.

retrait dans le milieu



retrait en tête

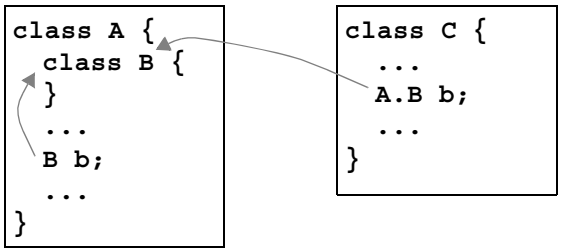


De cette analyse résulte la programmation suivante. Comme c'est souvent le cas, le programme est plus simple que les explications qui le justifient :

```
public void retireElement() {
     Maillon suivant=courant.suivant;
     Maillon precedent=courant.precedent;
     if (precedent!=null) {precedent.suivant=suivant;}
     else {tete=suivant;}
     if (suivant!=null) {suivant.precedent=precedent;}
     else {queue=precedent;}
     courant=suivant;
   }
```

11.5 Précisions sur les classes internes

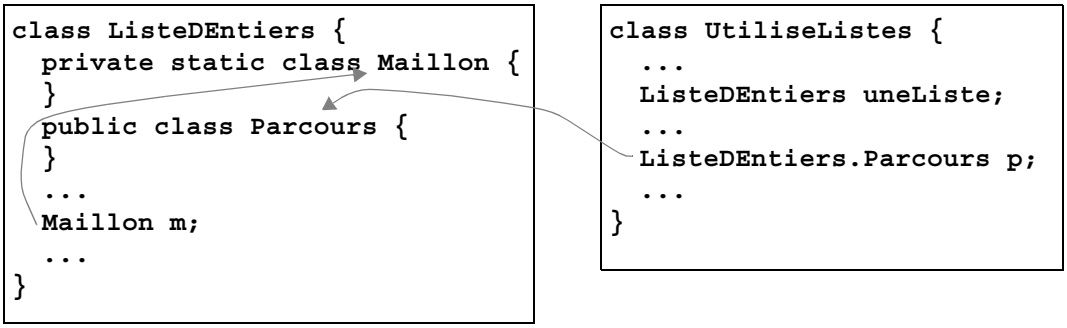
Java permet de définir une classe à l'intérieur d'une autre classe. Une telle classe est dite ***interne*** (***inner class*** en anglais). L'intérêt des classes internes est de mieux structurer les programmes. Notamment, le nom d'une classe interne est local à la classe englobante. Soit une classe **B** interne à une classe **A**. Depuis le texte de la classe **A**, la classe **B** est désignée directement par "**B**". Depuis le texte d'une autre classe **C**, la classe **B** interne à **A** est désignée par "**A.B**".



Les vocables **private** et **public** s'appliquent aux classes internes. Une classe interne **public** est utilisable depuis toute classe, alors qu'une classe interne **private** n'est utilisable que depuis le texte de la classe englobante. Cela permet de cacher les classes internes qui sont des outils destinés à n'être utilisés que par la classe englobante, pour une mise en œuvre particulière de l'abstraction représentée par cette classe.

Exemple : la classe **Maillon**, interne à la classe **ListeDEntiers**, n'est qu'un outil de mise en œuvre des listes. Elle n'a pas à être proposée à l'extérieur de la classe **ListeDEntiers**, c'est une notion qui doit rester totalement étrangère aux utilisateurs de la classe **ListeDEntiers**. La classe **Maillon** est donc **private**.

Autre exemple : en revanche, la classe **Parcours**, également interne à la classe **ListeDEntiers,** représente une notion offerte aux utilisateurs de la classe **ListeDEntiers**. Cela fait partie de l'abstraction "liste" de pouvoir être parcourue au moyen d'un **Parcours**. Les utilisateurs des listes vont explicitement utiliser des objets de type **Parcours**. La classe **Parcours** est donc **public**.



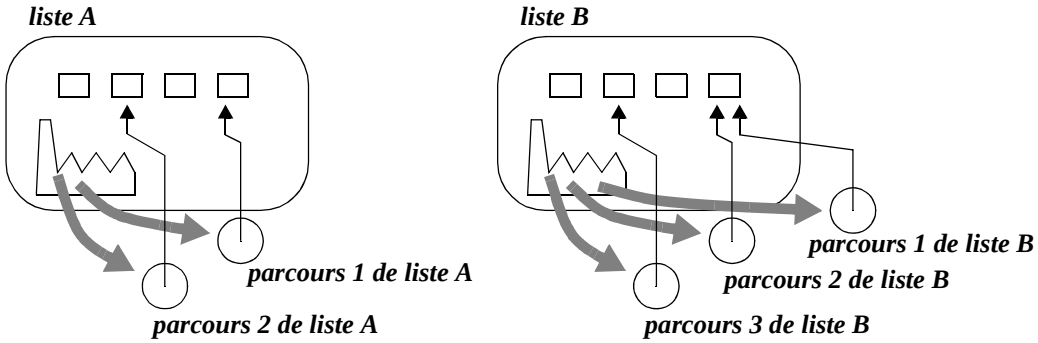
En Java, il existe deux sortes de classes internes :

Les classes internes ***statiques***, affublées du vocable **static** : une telle classe n'est attachée à aucun objet instance courante de la classe englobante. C'est le cas ici de la classe **Maillon**. La classe **Maillon** n'est liée à aucun objet de type **ListeDEntiers**.

Les classes internes ***dynamiques***, affublées d'aucun vocable supplémentaire : une telle classe est attachée à un objet instance courante de la classe englobante. C'est le cas ici de la classe **Parcours**. Un **Parcours** est toujours créé à propos d'un objet **ListeDEntiers** particulier, "***sa liste***". Depuis le texte d'une telle classe, on peut citer directement les données et méthodes membres

de l'objet de la classe englobante à propos duquel l'objet de la classe interne a été créé (on l'appelle ***instance courante de la classe englobante***). Par exemple, depuis la classe **`Parcours`**, on utilise **`tete`** et **`queue`** : ces identificateurs désignent directement les variables **`tete`** et **`queue`** de l'objet de type **`ListeDEntiers`** auquel cet objet de type **`Parcours`** est attaché.

Une image que l'on peut donner de cette notion de classe interne dynamique est celle d'une usine à fabriquer des objets rattachée à chaque objet de la classe englobante. Ainsi, chaque objet de type **`ListeDEntiers`** possède une usine à fabriquer des parcours sur cette liste.



Pour des raisons de structuration, il est préférable que les objets de la classe interne soient créés par des méthodes de la classe englobante. C'est ce que nous avons fait pour les parcours de listes : la classe **`ListeDEntiers`** possède les méthodes **`nouveauParcours`** qui rendent en résultat un nouvel objet de type **`Parcours`** opérant sur la liste à laquelle on applique la méthode. On peut appeler ces méthodes des "***fabricateurs***". Elle encapsulent des appels aux vrais constructeurs, pour des raisons de modularité et de contrôle lors de la création (ici il n'y a pas de contrôle, mais il pourrait y en avoir) :

```
public class ListeDEntiers {

  ...

  public Parcours nouveauParcours() {
  // un nouveau parcours initialisé au début de la liste
     return new Parcours();
  }

  public Parcours nouveauParcours(Parcours p) {
  // nouveau parcours initialisé à l'état de parcours de p
      return new Parcours(p);
  }

  ...
}
```

La première méthode, la plus utilisée, crée un parcours positionné sur le début de la liste (ou dans l'état **`estEnFin`** si la liste est vide).

La deuxième méthode crée un parcours positionné dans le même état qu'un autre parcours **`p`** passé en paramètre. Cela peut être utile pour effectuer certaine recherches complexes dans une liste.

Exercice 11.1 Manipulations de listes

Faire tourner sur un schéma chacune des procédures suivantes et indiquer ce qu'elles font.

```
import list.*;

class ManipulationDeListes {

static ListeDEntiers carres(int j) {
  ListeDEntiers c = new ListeDEntiers();
  for(int i=1;i<=j;i++){c.ajouteEnQueue(i*i);}
  return c;
}

static int p1(int j) {
  ListeDEntiers c = carres(9);
  ListeDEntiers.Parcours p= c.nouveauParcours();
  p.queue();
  while(!p.estEnFin()){
    int i=p.elementCourant();
    if (i%j==0){ return i;}
    else { p.precedent();}
  }
  return 0;
}


static ListeDEntiers p2(){
  ListeDEntiers l = carres(9);
  ListeDEntiers.Parcours p= l.nouveauParcours();
  while(!p.estEnFin()){
    int i=p.elementCourant();
    if (i%3==0) { p.modifElement(i+1);}
    p.suivant();
  }
  return l;
}


static ListeDEntiers p3(){
  ListeDEntiers resul = carres(9);
  ListeDEntiers.Parcours p= resul.nouveauParcours();
  p.queue();
  while(!p.estEnFin()){
    int i=p.elementCourant();
    if (i%4==0) { p.retireElement();}
    p.precedent();
  }
  return resul;
}
}
```

Exercice 11.2 Fusion de deux listes triées

Soient L_1 et L_2 deux listes d'entiers, triées par ordre croissant de la tête à la queue.

On veut construire la liste L qui contient tous les éléments de L_1 et L_2 triée par ordre croissant.

exemple : L_1 = 0 1 3 5 12 L_2 = 1 2 5 13 15 16

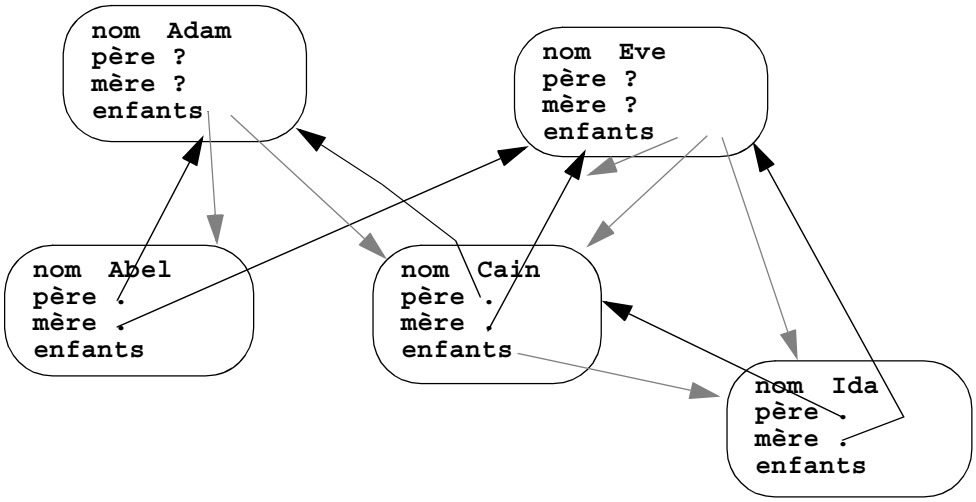
L = 0 1 1 2 3 5 5 12 13 15 16

Rédiger une fonction qui rend en résultat la liste L ainsi construite.

Exercice 11.3 Usage des références : arbre généalogique

Afin de construire des arbres généalogiques, on se propose de rédiger la classe **`Personne`** qui satisfait aux spécifications suivantes :

Une personne possède un ***nom***, un ***père***, une ***mère*** et ***des enfants***. Lors de la création d'une personne on indique son nom, son père et sa mère. La liste de ses enfants est initialisée à vide et cette personne est rajoutée dans la liste des enfants de son père et de sa mère.



On dispose de la classe **`ListeDePersonnes`**, dotée des mêmes fonctionnalités que **`ListeDEntiers`** (sauf que les élements sont des références à des objets de type **`Personne`**)

Rédiger la classe **`Personne`** et un programme principal de test qui crée la population illustrée sur la figure, puis imprime la liste des enfants de la grand-mère paternelle de **`Ida`**.

Exercice 11.4 Compression - décompression

Certaines applications utilisent des suites de données dans lesquelles de nombreuses valeurs successives sont identiques. C'est par exemple le cas pour représenter des images formées de grandes plages de même couleur. Une telle suite de valeurs entières est représentée sous forme ***compressée*** de la façon suivante : chaque sous-suite de k éléments consécutifs égaux de valeur e est représentée dans la liste compressée par k suivi de **e**. Exemple :

La compression de la suite L=<6, 6, 6, 6, 6, 2, 3, 3, 3, 3, 1, 1> est la suite <5, 6, 1, 2, 4, 3, 2, 1> car L contient cinq 6 suivi de un 2 suivi de quatre 3 suivi de deux 1.

Rédiger la fonction **`compression`** qui rend en résultat la compression d'une liste d'entiers passée en paramètre. Le résultat devra être une nouvelle liste créée par la fonction.

La ***décompression***, opération inverse de la compression, consiste à reconstituer la liste des valeurs représentées par une liste compressée. Exemple :

La décompression de la liste ***L***=<5, 6, 1, 2, 4, 3, 2, 1> est la liste <6, 6, 6, 6, 6, 2, 3, 3, 3, 3, 1, 1>.

Rédiger la fonction **`deCompression`** qui rend en résultat la décompression d'une liste d'entiers passée en paramètre.

Exercice 11.5 Ordre alphabétique d'une population

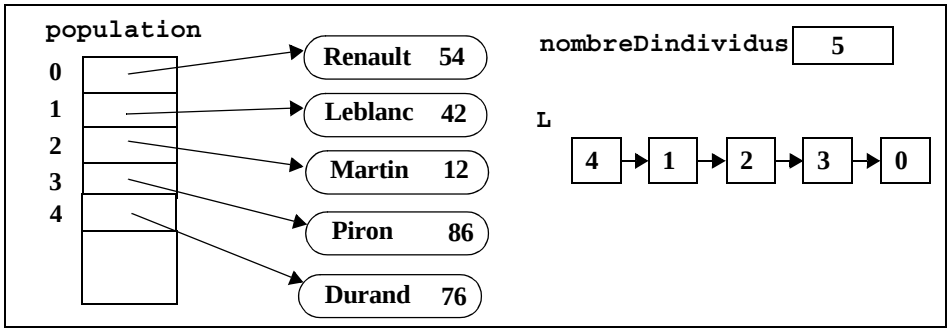
On suppose faites les déclarations et créations suivantes :

```
class Individu {
  public String nom; public int age;
  public Individu(String n, int a) {
    nom=n; age=a
  }
}

class TestPopulation {
  static Individu[]
    population = new Individu[1000];
  static int nombreDIndividus = 0;
  static ListeDEntiers L = new ListeDEntiers();
   ...
}
```

`nombreDIndividus` doit indiquer le nombre d'individus introduits dans le tableau population. Chaque nouvel individu est introduit dans le tableau **`population`**, à la suite des précédents.

La liste **`L`** sert à indiquer ***l'ordre alphabétique*** des individus entrés dans le tableau **`population`**. Chaque élément de la liste contient l'indice d'un individu dans le tableau **`population`** et cette liste est en permanence triée selon l'ordre alphabétique des noms des individus.

Exemple :



Rédiger la procédure **`afficheParOrdreAlphabetique`** qui affiche les noms des individus contenus dans le tableau population par ordre alphabétique.

Rédiger la procédure **`ajouteIndividu`** qui introduit un nouvel individu dans le tableau **`population`** et met à jour **`nombreDIndividus`** et la liste **`L`** de telle sorte qu'elle indique toujours l'ordre alphabétique des éléments du tableau **`population`**.

CHAPITRE 12 Utilisation des fichiers

Un ***fichier*** est une collection de données stockée de façon persistante sur un support périphérique (disque, disquette, bande magnétique, CDROM, etc...). Sur un même support on peut mettre plusieurs fichiers, d'où la nécessité de nommer un fichier et parfois son support également.

L'utilisation des fichiers répond à trois besoins principaux :

- conservation persistante d'information, alors que la mémoire est en général volatile,
- stockage de grandes quantités d'informations qui ne tiendraient pas en mémoire centrale,
- transport d'information d'un ordinateur à un autre.

Les exemples d'utilisation de fichiers sont nombreuses :

- Les relevés pluviométriques ou de températures d'une région sont stockés sur fichiers, ce qui permet de faire quand on le désire certaines statistiques.
- Une banque doit lors de chaque transaction, mettre à jour un compte client. Ceci impose que la banque possède de façon permanente les renseignements concernant chacun de ses clients : nom, prénom, adresse, numéro de compte, état du compte...
- Plus proche des applications "grand public", on trouve tout ce qui concerne le stockage de documents de traitement de textes ou de photos.

La structure des fichiers est très spécifique à chaque langage de programmation. Nous présentons ici uniquement les ***fichiers séquentiels de texte***.

12.1 Fichiers séquentiels de texte

Un ***fichier séquentiel*** contient une suite d'éléments**.** L'accès est séquentiel, ce qui signifie que l'accès au $n^{\text{ième}}$ élément ne peut se faire qu'après avoir accédé aux n-1 éléments précédents. Ceci s'oppose aux fichiers à accès direct où l'on peut accéder directement au $n^{\text{ième}}$ élément.

Un fichier séquentiel de ***texte*** contient simplement une suite de ***caractères*.** Il présente l'avantage d'être généralement à peu près compatible avec tous les langages de programmation, tout simplement parce que la représentation des caractères est normalisée (codages ASCII sur 7 et 8 bits et plus récemment codage UNICODE sur 16 bits). Les autres fichiers séquentiels s'appellent fichiers séquentiels "***binaires***"[1] : ils contiennent des données de n'importe quel type, mais ces données étant codées selon une convention propre à chaque langage de programmation, ils ne sont utilisables que par des programmes rédigés dans le langage qui les a produits.

1. Le terme "binaire" est mal choisi, car toute information, en mémoire ou sur fichier, est représentée par des successions de bits. Un terme plus juste serait "codage brut" ou "spécifique", qui ne passe pas par l'intermédiaire de caractères pour représenter l'information.

Bien que composé physiquement d'une suite de caractères, un fichier texte peut contenir tous types d'information. Il suffit d'avoir une convention de représentation des divers types de données sous forme de chaînes de caractères et de disposer des fonctions de conversion correspondantes. Pour les types de base, on utilise les conventions naturelles déjà pratiquées pour les lectures clavier et les écritures écran : notation décimale pour les entiers, notation avec point et exposant pour les réels, "true" et "false" pour les booléens...

Un avantage des fichiers textes, en plus de la compatibilité avec tous les langages de programmation, est la lisibilité "humaine" des informations. Les informations sont écrites "en clair", sous forme de chaînes de caractères que l'on peut lire ou construire avec tout éditeur de texte disponible sur tout ordinateur. Ainsi, moyennant la connaissance de certaines conventions propres à la gamme d'application manipulant ces fichiers, ils sont compréhensibles par un lecteur humain.

Par exemple, un fichier contenant les noms et les âges d'une population se présentera ainsi :

```
toto 12 jules 14 alfred 18 marcel 9
```

et pour le lire il suffira d'exécuter, comme pour une entrée depuis le clavier, les procédures spécifiques de lecture suivantes :

`...lireChaine()...` lit le nom **`"toto"`** sous forme d'une donnée de type **`String`**

puis **`...lireUnEntier()...`** lit l'âge **`12`** sous forme d'une donnée de type **`int`**

puis **`...lireChaine()...`** lit le nom **`"jules"`** sous forme d'une donnée de type **`String`**

puis **`... lireUnEntier() ...`** lit l'âge **`14`** sous forme d'une donnée de type **`int`**

etc...

12.2 Utilisation de fichiers textes en Java

Nous présentons ici un moyen relativement simple d'utiliser les fichiers séquentiels de texte en Java. On propose deux classes :

- la classe **`LectureFichierTexte`** pour lire des fichiers textes,
- et la classe **`EcritureFichierTexte`** pour écrire des fichiers textes.

Ces deux classes, comme la classe **`Lecture`** que nous avons utilisée pour les entrées clavier, ne font pas partie de la bibliothèque standard de Java. La bibliothèque standard, **`java.io`**, est riche en possibilités mais est sans doute un peu trop technique pour les besoins courants.

12.2.1 Lecture de fichiers textes

Pour lire un fichier texte, on utilise un objet de type **`LectureFichierTexte`**. Pour créer un tel objet, il suffit d'exécuter :

```
LectureFichierTexte population
                  = new LectureFichierTexte("lesEnfants.txt");
```

Ceci crée un accès en lecture au fichier appelé **`lesEnfants.txt`** supposé exister dans la machine (dans le répertoire courant dans ce cas, mais on peut indiquer un chemin relatif ou absolu). Cette instruction provoque une erreur si ce fichier n'existe pas. Dans cet exemple l'objet qui sert d'accès a été baptisé **`population`**.

Ensuite, on peut lire le fichier au moyen des méthodes offertes par cette classe. Ces méthodes sont similaires à celles utilisées pour les lectures clavier :

```
public class LectureFichierTexte {

   public LectureFichierTexte(String nom)
   accès en lecture au fichier indiqué par nom, erreur si le fichier n'existe pas

   public void fermer()
   ferme le fichier

   public char lireUnCar()
   lecture d'un caractère, erreur si lecture impossible

   public String lireChaine(String delimiteurs)
   lecture d'une chaîne, les délimiteurs étant les caractères de delimiteurs,
   rend la chaîne vide si fin de fichier, erreur si lecture impossible

   public String lireChaine()
   lecture d'une chaîne comprise entre délimiteurs,
   les délimiteurs étant ' ' (espace) '\r' (fin ligne Microsoft) ou '\n' (fin ligne Unix)
   rend la chaîne vide si fin de fichier, erreur si lecture impossible

   public int lireUnEntier()
   lecture d'un entier représenté en décimal, erreur si lecture impossible ou format incorrect

   public double lireUnReel()
   lecture d'un nombre réel, erreur si lecture impossible ou format incorrect

   public boolean finDeFichier()
   indique si fin de fichier
}
```

La méthode **`finDeFichier`** permet de savoir si on est en fin de fichier, c'est-à-dire si tous les caractères ont été lus. Le résultat de cette méthode sert souvent d'argument pour terminer une boucle qui lit le fichier.

La méthode **`fermer`** clôt l'accès au fichier. Une fois fermé, c'est une erreur que de chercher à accéder au fichier par cet accès. Une programmation "saine" ferme les fichiers avant de terminer l'exécution du programme, bien que ça marche souvent si on ne le fait pas pour un accès en lecture.

La lecture du fichier donné en exemple pourrait être réalisé par la boucle suivante :

```
LectureFichierTexte population
              = new LectureFichierTexte("lesEnfants.txt");
...
while (!population.finDeFichier()) {
   String nom = population.lireChaine();
   int age = population.lireUnEntier();
   ... traitement des données lues nom et age ...
}
population.fermer();
...
```

12.2.2 Écriture de fichiers textes

Pour écrire dans un fichier texte, on utilise un objet de type **`EcritureFichierTexte`**. Pour créer un tel objet, il suffit d'exécuter :

```
EcritureFichierTexte population
                = new EcritureFichierTexte("lesEnfants.txt");
```

Ceci crée un accès en écriture au fichier appelé **`lesEnfants.txt`**. Le fichier n'existe pas nécessairement. S'il existe, il sera effacé et réécrit par le programme. S'il n'existe pas, il sera créé par cette instruction.

Ensuite, on peut écrire dans le fichier au moyen des méthodes offertes par cette classe. Comme pour les écritures sur écran, il existe une méthode d'écriture pour tous les types de base :

```
public class EcritureFichierTexte {

   public EcritureFichierTexte(String nom)

   public void ecrire(char c)
   public void ecrire(String s)
   public void ecrire(int k)
   public void ecrire(boolean b)
   public void ecrire(double x)

   public void fermer()
}
```

La méthode **`fermer`** clôt l'accès au fichier. L'opération **`fermer`** est ici essentielle : si on ne l'exécute pas, les informations ne sont pas conservées dans le fichier.

Voici, à titre d'exemple, un programme qui crée un fichier appelé **`lesEntiers.txt`** contenant les nombres entiers de 0 à 39 avec, pour chacun d'eux, l'indication s'il est pair ou non. Après exécution, le fichier contiendra :

```
0 est pair : true
1 est pair : false
2 est pair : true
...
38 est pair : true
39 est pair : false
```

```
class TestEcriture {
  public static void main(String[] arg) {
     EcritureFichierTexte sortie =
               new EcritureFichierTexte("lesEntiers.txt");
     for (int i=0; i<40; i++) {
        sortie.ecrire(i);
        sortie.ecrire(" est pair : ");
        sortie.ecrire((i%2)==0);
        sortie.ecrire('\n'); passage à la ligne par écriture explicite de '\n'
     }
    sortie.fermer();
  }
}
```

Exercice 12.1 Création d'un fichier de nombres au carré

Rédiger un programme qui lit un fichier ***entree*** contenant des entiers et écrit dans un fichier ***sortie*** les carrés des valeurs lues.

Le nom (ou le chemin) du fichier ***entree*** sera donné au clavier et le nom du fichier ***sortie*** sera fabriqué en ajoutant au nom du fichier d'entrée le suffixe "**.carre**".

Exercice 12.2 Décodage des accents

Il est courant de recevoir des messages où les caractères accentués ont été saisis de la façon suivante :

```
Bonjour He'le`ne,
vas-tu e^tre disponible pour aller to^t sur l'i^le de Bre'hat,
ou` il fait beau a` Noe¨l ?
```

On souhaite réaliser un programme pour transcrire ces messages sous la forme plus lisible :

```
Bonjour Hélène,
vas-tu être disponible pour aller tôt sur l'île de Bréhat,
où il fait beau à Noël ?
```

Les accentuations possibles sont **a** avec (**`**, **^**), **e** avec (**`**, **'**, **^**, **¨**), **i** avec (**^**, **¨**), **o** avec (**^**, **¨**) et **u** avec (**`**,**^**, **¨**).

Remarque : on ne cherche à traiter ni les majuscules accentuées ni le caractère **ç**.

Le texte original sera contenu dans un fichier d'entrée et le résultat sera

écrit dans un fichier de sortie.

CHAPITRE 13 Structures de données : ensembles

13.1 Spécification du type ensemble

Un ***ensemble*** est une collection de données dans laquelle toutes les données ont une valeur différente. Ajouter un élément qui s'y trouve déjà ne change pas l'ensemble. Cette structure de données est donc très proche de la notion mathématique d'ensemble.

La notion d'ensemble exige que le type des éléments soit doté de la notion d'***égalité***. Ce n'était pas le cas pour des structures telles que le vecteur ou la liste, mais cela est obligatoire ici pour donner un sens aux opérations telles que le test d'appartenance, l'union, l'intersection...

La spécification d'un type ensemble exige généralement que le type des éléments possède une ***relation d'ordre***, "supérieur ou égal", afin de pouvoir comparer et ordonner les éléments. Moins fondamentale que le besoin d'égalité, cette exigence permet :

- de parcourir un ensemble par ordre croissant de ses éléments,
- d'offrir un test d'appartenance et des opérations ensemblistes très ***performantes*** : ceci concerne la mise en œuvre, mais sans relation d'ordre sur les éléments, la recherche d'un élément est nécessairement ***proportionnelle à la taille*** de l'ensemble (en moyenne), alors qu'il est ***proportionnel au logarithme*** de la taille si on opère sur une collection triée (voir les exercices concernant la recherche par dichotomie). Les opérations ensemblistes d'union, d'intersection et de différence prennent un temps proportionnel au produit des tailles des ensembles opérandes dans le cas d'ensembles non triés, alors qu'il ne prennent qu'un temps proportionnel à la taille des opérandes si les ensembles sont triés. Ces rapports de performance sont considérables dès que la taille des ensembles devient importante.

Nous présentons ici des ensembles d'entiers, réalisés par la classe **`EnsembleDEntiers`**. Bien évidemment on peut imaginer les mêmes fonctionnalités pour des ensembles de n'importe quel type d'éléments, pourvu que ce type soit doté des opérations de comparaison d'égalité et de supériorité.

La classe **`EnsembleDEntiers`** réalise des ***ensembles modifiables*** : ce sont des objets dont l'état est un ***ensemble fini***, éventuellement vide, ***de valeurs entières***. On peut le faire évoluer en taille en ***ajoutant*** ou en ***retirant*** un élément.

Comme pour les listes, cette classe offre une classe interne **`EnsembleDEntiers.Parcours`** qui permet de créer des objets pour parcourir un ensemble par ordre croissant de ses éléments.

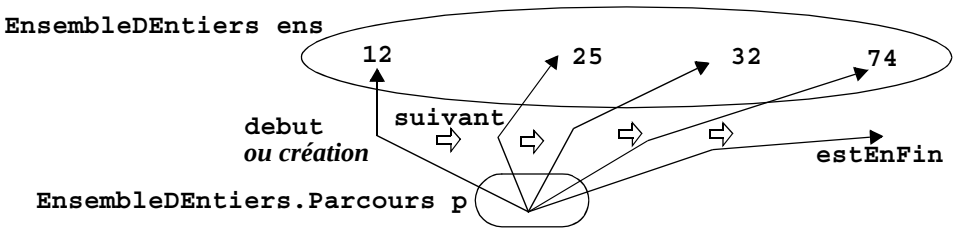
La spécification de la classe **EnsembleDEntiers** est :

```
public class EnsembleDEntiers {

  public EnsembleDEntiers() // constructeur : ensemble vide

  public boolean estVide() // résultat : indique si this est vide

  public int cardinal() // résultat : nombre d'éléments de this

  public boolean contient(int e)
  // résultat : indique si e appartient à this

  public void ajouteElement(int e)
  // effet : ajoute e à this (aucun effet si e est déjà dans this)

  public void retireElement(int e)
  // effet : retire e de this (aucun effet si e n'est pas dans this)

  public static EnsembleDEntiers
           union(EnsembleDEntiers e1, EnsembleDEntiers e2)
  // résultat : un nouvel ensemble égal à l'union de e1 et e2

  public static EnsembleDEntiers
           intersection(EnsembleDEntiers e1, EnsembleDEntiers e2)
  // résultat : un nouvel ensemble égal à l'intersection de e1 et e2

  public static EnsembleDEntiers
           difference(EnsembleDEntiers e1, EnsembleDEntiers e2)
  // résultat : un nouvel ensemble égal à la différence de e1 et e2

  public String toString()
  // résultat : une représentation en clair de this, sous la forme
  //             { 12 | 23 | 45 | 72}, {} pour l'ensemble vide

  public Parcours nouveauParcours()
  // résultat : un nouveau parcours initialisé sur le plus petit
  // élément de this ou en fin si this est vide

  public Parcours nouveauParcours(EnsembleDEntiers.Parcours p)
  // résultat : un nouveau parcours initialisé à l'état de p

  //=================================================================
  public class Parcours {
    public void tete()
    // effet : positionne this sur le plus petit élément
    public void suivant()
    // effet : positionne this sur l'elément suivant
    public boolean estEnFin() // résultat : indique si this est terminé
    public int elementCourant() // prérequis : this n'est pas en fin
    // résultat : élément courant de this
    public void retireElement() // prérequis : this n'est pas en fin
    // effet : retire l'élément courant de this
  } //=================================================================
}
```

13.2 Exemples d'utilisation d'ensembles

Pour parcourir un ensemble par ordre croissant de ses éléments, on procède de la façon suivante :

```
EnsembleDEntiers ens; ...
EnsembleDEntiers.Parcours p = ens.nouveauParcours();
while(!p.estEnFin()) {
   ... exploiter p.elementCourant()... p.suivant();
}
```



On peut prendre comme exemple d'application la rédaction d'une fonction **inclus** qui indique si un ensemble **e1** est inclus dans un ensemble **e2**. L'algorithme de cette fonction consiste à parcourir l'ensemble **e1** et à vérifier que chacun de ses éléments appartient à **e2** :

```
static boolean inclus(EnsembleDEntiers e1, EnsembleDEntiers e2) {
  EnsembleDEntiers.Parcours p = e1.nouveauParcours();
  boolean estInclus=true;
  while(!p.estEnFin() && estInclus) {
    if(!e2.contient(p.elementCourant())){estInclus=false;}
    p.suivant();
  }
  return estInclus;
}
```

13.3 Mise en œuvre des ensembles

Il y a de nombreuses façons de représenter un ensemble.

On peut utiliser simplement une liste telle que celle spécifiée dans un chapitre précédent, mais dans ce cas le test d'appartenance nécessite un temps proportionnel (en moyenne) à la taille de l'ensemble car la recherche dans la liste ne peut se faire que par un parcours séquentiel.

Nous allons décrire ici une réalisation au moyen de tableaux dont la taille s'adapte à la taille de l'ensemble. Le tableau contient les éléments de l'ensemble triés par ordre croissant. Ainsi la recherche d'un élément peut être réalisée en un temps proportionnel au logarithme de la taille de l'ensemble, grâce à une recherche dichotomique.

Une variable entière **nbElements** contient à tout moment la taille effective de l'ensemble. L'ensemble vide est représenté par **nbElements=0** et aucun tableau (on aurait pu utiliser un tableau de taille nulle, mais aucun tableau occupe encore moins de place) :

```
public class EnsembleDEntiers {
  private int nbElements;
  private int[] elements; tableau de taille ajustée aux besoins
  public EnsembleDEntiers() { ensemble vide
      elements=null; nbElements=0;
  } ...
```

La fonction **estVide** rend simplement vrai si et seulement si **nbElements==0**.

```
public boolean estVide() {return nbElements==0;}
```

La fonction **cardinal** rend simplement **nbElements**.

```
public int cardinal() {return nbElements;}
```

Le tableau **elements** contient à tout moment les éléments de l'ensemble, triés par ordre croissant, dans les positions d'indice **0** à **nbElements-1**.

La fonction suivante recherche un élément par dichotomie. Elle rend en résultat l'indice de l'élément cherché s'il est présent et -1 s'il est absent. C'est une fonction privée (non offerte aux utilisateurs, car elle est spécifique à la représentation par tableau et n'a aucun sens au niveau de l'ensemble).

```
  private int indiceDe(int e) {
  // résultat : indice de l'élément égal à e, 0..nbElements-1,
  // -1 si e est absent
      int i = 0; int j = nbElements-1;
      while(j>=i){
         int m=(i+j)/2;
         if (elements[m]==e) {return m;}
         else if (elements[m]<e) {i=m+1;}
         else {j=m-1;}
      }
      return -1;
  }
```

Méthode `contient`

Cette fonction est utilisée pour réaliser la fonction **contient**, offerte aux utilisateurs, qui sert à tester si l'ensemble contient un élément :

```
public boolean contient(int e) {
// indique si e appartient à this
  return indiceDe(e)!=-1;
}
```

Méthode `ajouteElement`

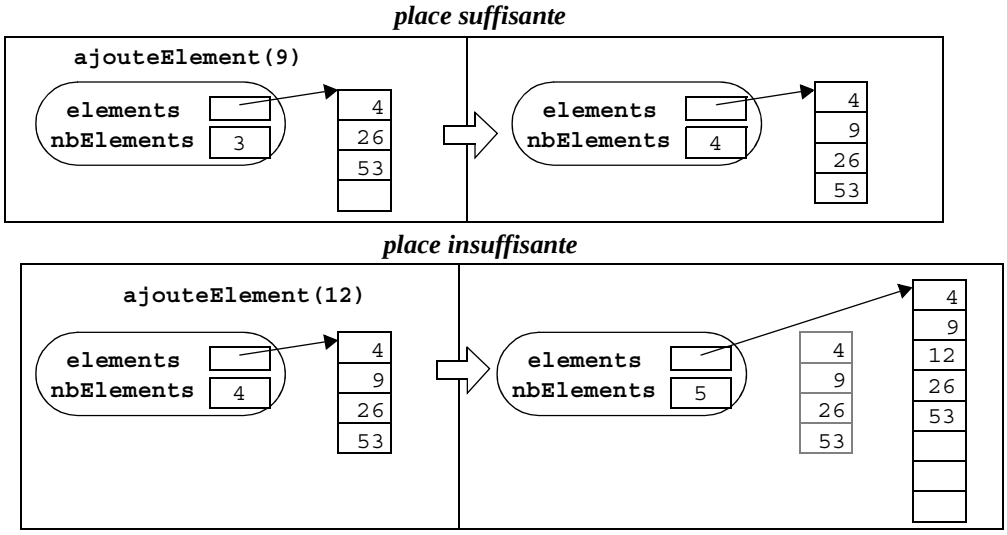
L'ajout d'un élément à l'ensemble est assez complexe. Il faut d'abord tester si l'élément appartient à l'ensemble. Dans ce cas il n'y a rien à faire.

Ensuite il faut vérifier s'il y a assez de place dans le tableau pour rajouter un élément. Sinon, il faut créer un nouveau tableau 2 fois plus grand (ou de taille 1 si l'ensemble était vide). Afin de pouvoir modifier les aspects quantitatifs, on a utilisé une constante **facteurDeCroissance** pour le facteur de croissance (égal à 2 ici) et une constante **taillePourSingleton** pour la taille du tableau qui représente un ensemble à un seul élément (égale à 1 ici).

```
  private static final int facteurDeCroissance=2;
  private static final int taillePourSingleton=1;
```

Dans les deux cas il faut insérer le nouvel élément à sa place dans le tableau, de façon à le conserver trié par ordre croissant.

Remarque : la représentation au moyen d'un tableau n'est pas efficace en ce qui concerne l'ajout et le retrait d'éléments : l'insertion pour l'ajout et le tassement pour le retrait sont des opérations qui nécessitent en moyenne un temps proportionnel à la taille de l'ensemble. Une représentation plus efficace utilise un arbre binaire ordonné et équilibré, appelé "arbre AVL", dont la mise en œuvre sort du cadre de ce cours.




```
public void ajouteElement(int e) {
// ajoute e à this (aucun effet si e est déjà dans this)
  if (!contient(e)) {
    if (elements!=null && nbElements<elements.length){
      // place suffisante
      int i=nbElements;
      while (i>0 && elements[i-1]>e) {
      elements[i]=elements[i-1]; i--;
      }
      elements[i]=e;
    }
    else { // place insuffisante, création d'un nouveau tableau
      int[] nouveau;
      if (nbElements==0) {
        nouveau = new int[taillePourSingleton];
      }
      else {
        nouveau=new int[facteurDeCroissance*elements.length];
      }
      int i=nbElements;
      while (i>0 && elements[i-1]>e) {
        nouveau[i]=elements[i-1]; i--;
      }
      nouveau[i] = e; i--;
      while (i>=0) { nouveau[i]=elements[i]; i--;}
      elements=nouveau;
    }
    nbElements++;
  }
}
```

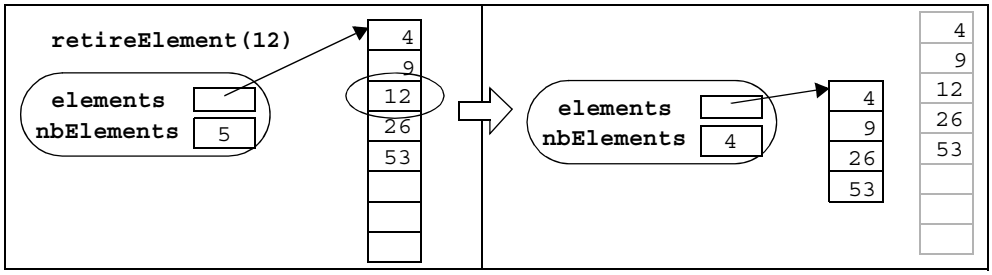
Méthode `retireElement`

Le retrait d'un élément présente des difficultés similaires. Il faut commencer par tester si l'élément appartient à l'ensemble, sinon il n'y a rien à faire. Le test d'appartenance ayant donné l'indice de l'élément à retirer, on utilise cet indice en appelant une procédure privée **`retireIemeElement`** qui retire le $i^{ème}$ élément.

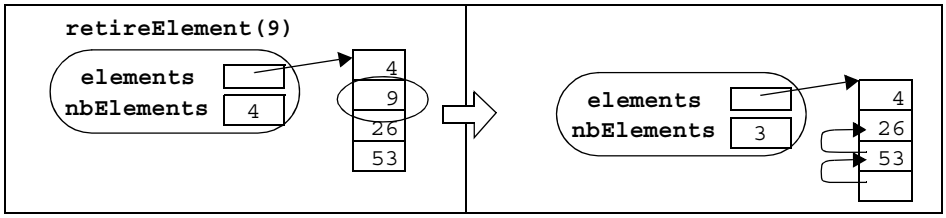
```
public void retireElement(int e) {
// retire e de this (aucun effet si e n'est pas dans this)
     int k = indiceDe(e);
     if (k!=-1) {retireIemeElement(k);}
}
```

Lorsque le nombre d'éléments devient plus petit que la moitié de la taille du tableau, un nouveau tableau plus petit est créé de façon à ne pas encombrer inutilement la mémoire. L'ancien tableau, sauf l'élément retiré, est recopié dans le nouveau. Dans le cas où un tableau plus petit n'est pas recréé, le tableau est simplement "tassé" d'une position à partir de l'indice de l'élément supprimé. Il y a un cas particulier lorsque l'ensemble devient vide, car on a décidé de représenter l'ensemble vide par aucun tableau. Dans ce cas, la référence au tableau est mise à **`null`**.

cas de création d'un tableau plus petit



cas de simple tassement




```
private void retireIemeElement(int k) {
// retire le k ième élément
  nbElements--;
  if (nbElements==0) {elements = null;}
  else if (nbElements < elements.length/facteurDeCroissance) {
    // création d'un tableau plus petit
    int[] nouveau =
    new int[elements.length/facteurDeCroissance];
    for (int i=0;i<k; i++) {nouveau[i]=elements[i];}
    for (int i=k;i<nbElements; i++) {nouveau[i]=elements[i+1];}
    elements=nouveau;
  }
  else {
    for (int i=k;i<nbElements; i++) {elements[i]=elements[i+1];}
  }
}
```

Opérations ensemblistes

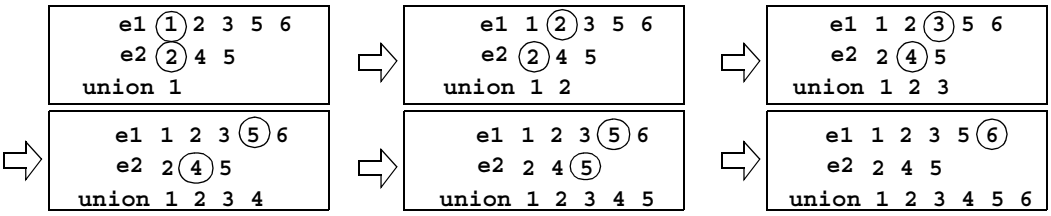
La réalisation des opérations ensemblistes, union, intersection et différence, profite du fait que les tableaux sont triés par ordre croissant. Puisque l'on parcourt les éléments des ensembles opérandes par ordre croissant, on génère les éléments du résultat également par ordre croissant. Chaque élément du résultat est donc ajouté "au bout" du tableau, à la suite de ceux déjà présents. On utilise pour cela une procédure outil **`ajouteAuBout`** qui ajoute un élément "au bout" du tableau représentant le résultat. Cette procédure double la taille du tableau chaque fois que c'est nécessaire.

```
private void ajouteAuBout(int e) {
   if (elements!=null && nbElements<elements.length){
   // place suffisante
      elements[nbElements]=e;
   }
   else if (nbElements==0) {
      elements= new int[taillePourSingleton]; elements[0]=e;
   }
   else {
   // place insuffisante, création d'un nouveau tableau
      int[] nouveau = new int[facteurDeCroissance*elements.length];
      for (int i=0;i<nbElements;i++) {nouveau[i]=elements[i];}
      nouveau[nbElements] = e;
      elements=nouveau;
   }
   nbElements++;
}
```

Nous avons choisi de réaliser les opérations ensemblistes sous forme de ***fonctions pures***, et non pas par modification d'un des opérandes. Le résultat est donc un nouvel ensemble créé par l'opération.

Fonction `union`

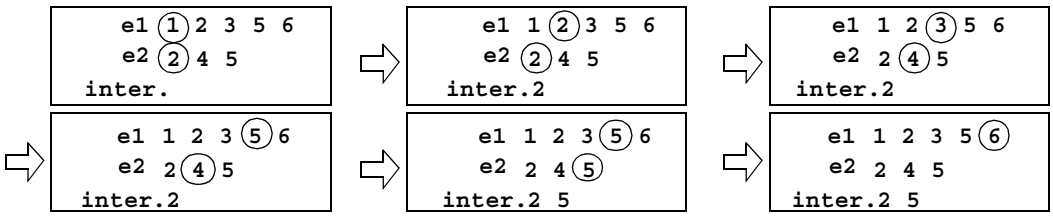
L'union est réalisée comme le montre l'exemple suivant :



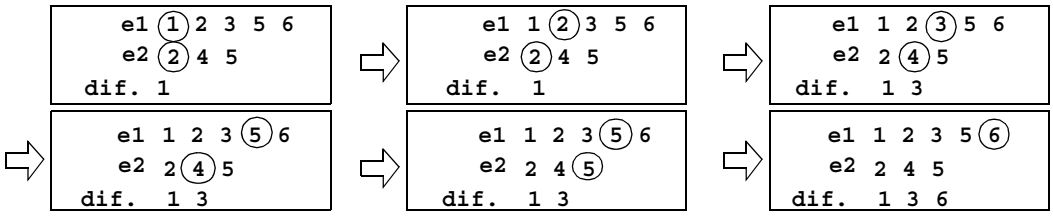

```
public static EnsembleDEntiers
            union(EnsembleDEntiers e1, EnsembleDEntiers e2) {
   EnsembleDEntiers resul=new EnsembleDEntiers(); int i1=0; int i2=0;
   while (i1<e1.nbElements && i2<e2.nbElements) {
      int v1 = e1.elements[i1]; int v2 = e2.elements[i2];
      if (v1<v2) {resul.ajouteAuBout(v1); i1++;}
      else if (v1>v2) {resul.ajouteAuBout(v2); i2++;}
      else {resul.ajouteAuBout(v1); i1++; i2++;}
   }
   while(i1<e1.nbElements){resul.ajouteAuBout(e1.elements[i1]); i1++;}
   while(i2<e2.nbElements){resul.ajouteAuBout(e2.elements[i2]); i2++;}
   return resul;
}
```

Fonctions `intersection` et `difference`

L'intersection et la différence sont réalisées de façon similaire :




```
public static EnsembleDEntiers
              intersection(EnsembleDEntiers e1, EnsembleDEntiers e2){
   EnsembleDEntiers resul=new EnsembleDEntiers();
   int i1=0; int i2=0;
   while (i1<e1.nbElements && i2<e2.nbElements) {
      int v1 = e1.elements[i1]; int v2 = e2.elements[i2];
      if (v1<v2) {i1++;}
      else if (v1>v2) {i2++;}
      else {resul.ajouteAuBout(v1); i1++; i2++;}
   }
   return resul;
}
```




```
public static EnsembleDEntiers
              difference(EnsembleDEntiers e1, EnsembleDEntiers e2) {
   EnsembleDEntiers resul=new EnsembleDEntiers();
   int i1=0; int i2=0;
   while (i1<e1.nbElements && i2<e2.nbElements) {
      int v1 = e1.elements[i1]; int v2 = e2.elements[i2];
      if (v1<v2) {resul.ajouteAuBout(v1); i1++;}
      else if (v1>v2) {i2++;}
      else {i1++; i2++;}
   }
   while (i1<e1.nbElements) {
      resul.ajouteAuBout(e1.elements[i1]); i1++;
   }
   return resul;
}
```

Méthode `toString` :

```
public String toString() {//représentation en clair
   if (estVide()) {return "{}";}
   StringBuffer resul = new StringBuffer("{| "+ elements[0]);
   for (int i=1; i<nbElements; i++) {
      resul.append(" , " + elements[i]);
   }
   resul.append(" |}"); return resul.toString();
}
```

Méthodes de création de parcours :

```
public Parcours nouveauParcours() {
// résultat : nouveau parcours initialisé en début d'ensemble
     return new Parcours();
}

public Parcours nouveauParcours(Parcours p) {
// résultat : nouveau parcours initialisé à l'état de p
     return new Parcours(p);
}
```

Mise en œuvre des parcours

Un **`Parcours`** est simplement représenté par un nombre entier, indice dans le tableau des éléments de l'élément couramment désigné.

```
public class Parcours {

  private int courant;

  Parcours() {courant=0;}
  Parcours(Parcours p) {courant=p.courant;}

  public void tete() {courant=0;}
  public void suivant() {if (courant<nbElements) {courant++;}}
  public boolean estEnFin() {return courant==nbElements;}
  public void retireElement() {retireIemeElement(courant);}
  public int elementCourant() {return elements[courant];}
}
```

Exercice 13.1 Manipulation d'ensembles

Rédiger les fonctions suivantes :

La fonction **egalite** qui détermine si deux ensembles sont égaux (contiennent les mêmes éléments).

```
static boolean egalite (EnsembleDEntiers E, EnsembleDEntiers F)
//résultat : indique si E et F sont égaux
```

La fonction **comprisEntre**, qui étant donnés un ensemble ***E*** et deux entiers ***a*** et ***b***, rend un ensemble contenant les élements de ***E*** qui sont compris entre ***a*** et ***b***.

```
static EnsembleDEntiers comprisEntre(EnsembleDEntiers E,int a,int b)
// résultat : ensemble d'entiers formé des
// éléments de E qui sont compris entre a et b
```

Exercice 13.2 Représentation des Polynômes

On se propose de représenter les polynômes à coefficients entiers. Afin de représenter de façon économique les polynômes de degré important mais dont la plupart des coefficients sont nuls, on décide de réaliser une mise en œuvre de la classe **Polynome** au moyen d'une ***liste de paires d'entiers*** $<c_i,d_i>$ où c_i est le $i^{ème}$ coefficient non nul par ordre croissant des degrés et d_i est le degré correspondant.

Exemple : le polynôme $6\boldsymbol{x}^{45} + 33\boldsymbol{x}^{12} + 5\boldsymbol{x} + 1$

sera représenté par la liste de paires d'entiers : <1,0> <5,1> <33,12> <6,45>

Le polynôme nul sera naturellement représenté par une liste vide.

La classe **Polynome** offre les fonctionnalités suivantes :

static Polynome zero

le polynôme nul

static Polynome aPartirDe(int[] tab)

polynôme indiqué par un tableau contenant alternativement les coefficients non nuls et les degrés correspondants, par ordre décroissant des degrés, **tab** = $\{c_n,d_n,...,c_1,d_1,c_0,d_0\}$. Ce constructeur sera essentiellement destiné à programmer des test. Ainsi avec

```
int[] tab = {6,45,33,12,5,1,1,0};
```

Polynome.aPartirDe(tab)

rendra le polynôme $6\boldsymbol{x}^{45} + 33\boldsymbol{x}^{12} + 5\boldsymbol{x} + 1$.

String toString()

forme textuelle "en clair" du polynôme :

a_n*X^n +... + a_2*X^2 + a_1*X + a_0 (0 si **this** est le polynome nul)

`double valeur (double x)`

valeur du polynôme **`this`** au point **`x`**

`int degre()`

degré du polynôme. Le degré du polynôme nul (-infini en math) sera représenté par -1

`static boolean egalite(Polynome x, Polynome y)`

indique si **`x`** et **`y`** sont égaux

`static Polynome somme(Polynome x, Polynome y)`

somme de **`x`** et de **`y`**

`static Polynome produit(Polynome x, Polynome y)`

produit de **`x`** par **`y`**

`static Polynome derive(Polynome x)`

dérivé de **`x`**

On dispose des classes : **`PaireDEntiers`** et **`ListeDePairesDEntiers.`**

```
public class PaireDEntiers {
// Paires constantes d'entiers <premier,second>
  public final int premier; public final int second;
  public PaireDEntiers(int p, int s) {premier=p; second=s;}
  public boolean equals(PaireDEntiers p2) {
  // resultat : indique si this est egal à p
    return premier==p2.premier && second==p2.second;
 }
}
```

La classe **`PaireDEntiers`** offre des ***paires constantes***, c'est-à-dire une structure avec des champs non modifiable. Une telle structure représente donc une ***valeur*** du domaine (entier × entier), inaltérable donc. Cela est d'une part naturel et d'autre part permet sans aucun soucis de "partager" des mêmes objets **`PaireDEntiers`** dans la représentation de plusieurs polynômes. Par exemple pour la fonction **`somme`**, on n'a besoin de créer une nouvelle **`PaireDEntiers`** que lorsque l'on ajoute des monômes non nuls de même degré. On essayera de profiter de cette opportunité pour économiser la mémoire.

La classe **`ListeDePairesDEntiers`** a les mêmes fonctionnalités que **`ListeDEntiers`**. La seule différence est que les éléments sont des **`PaireDEntiers`**.

Remarque :

on a volontairement pas prévu de constructeur public. Ceci signifie que le constructeur **`Polynome()`** est ***privé***. Il est uniquement à usage interne, pour construire les résultats des opérations de calcul. C'est logique car la classe **`Polynome`** représente un ***domaine de valeurs***. Tout polynôme s'obtient au moyen soit de la constante **`Polynome.zero`**, soit du convertisseur à partir d'un tableau **`Polynome.apartirDe(T)`**, soit par les opérateurs **`somme(x,y)`**, **`produit(x,y)`** et **`derive(x)`**. Le fait de n'offrir aucun constructeur public a pour conséquence que l'utilisateur de la classe **`Polynome`** n'aura jamais possibilité de faire **`new Polynome()`**, et c'est parfaitement cohérent avec la notion de ***valeur***. En d'autres termes, un objet de type **`Polynome`** se comporte

comme une valeur, par exemple comme l'entier bien connu qui s'appelle 12, et jamais personne n'a eu à faire "**`new 12`**".

La fonction **`Polynome.apartirDe(T)`** est très pratique pour programmer les tests.

Tests :

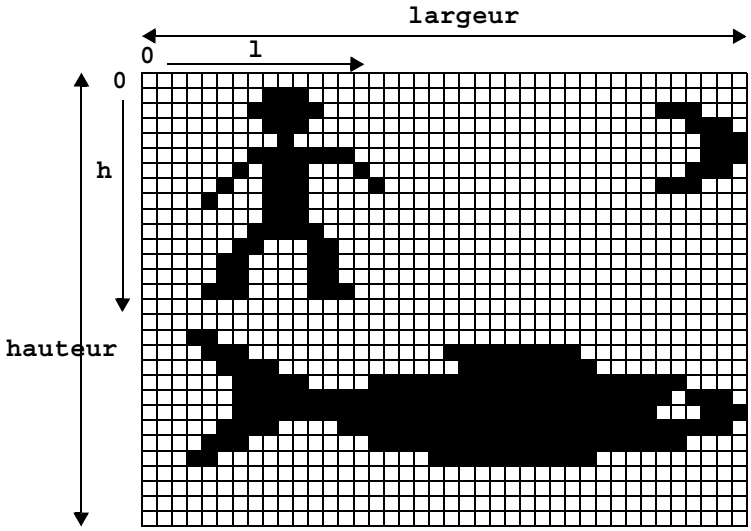
Il vaut mieux faire des tests entièrement contenus dans un programme de test plutôt que des tests par lecture des données, pour des raisons évidente de reproductibilité et de gain de temps. Les tests seront programmés dans la procédure principale d'une classe **`TestPolynome`**. On s'efforcera d'inventer les tests les plus judicieux possibles. Un bon test n'est pas un test qui s'arrange "pour que ça marche" mais qui, au contraire, fait tout "pour que ça ne marche pas". Il faut certes tester les cas habituels mais aussi tester les cas particuliers.

Exercice 13.3 Représentation d'images noir et blanc

On se propose de représenter des images constituées de points noirs ou blancs. Une telle image se présente comme un damier de points de taille **`hauteur`**×**`largeur`**.

Chaque point est repéré par sa position verticale **`h`** et horizontale **`l`**.

Le coin supérieur gauche a pour coordonnées (0,0).



On décide de représenter de telles images par une classe **`ImageBinaire`** dont les spécifications sont données ci-après.

```
class ImageBinaire { // représentations d'images en noir et blanc

public static ImageBinaire blanche(int hauteur, int largeur)
// prérequis : hauteur>0, largeur>0
// résultat : nouvelle image blanche de taille largeur x hauteur

public static
    ImageBinaire aPartirDe(int hauteur, int largeur, String s)
// prérequis : hauteur>0, largeur>0, s.leghth>=hauteur*largeur
// résultat : image de taille largeur x hauteur figurée par s,
// '*' pour noir, '.' pour blanc, présentée ligne par ligne

public int hauteur() // résultat : hauteur de this

public int largeur() // résultat : largeur de this

public void noircir(int h, int l)
// prérequis : 0=<h<hauteur, 0=<l<largeur
// effet : met à noir le point (h,l)

public void blanchir(int h, int l)
// prérequis : 0=<h<hauteur, 0=<l<largeur
// effet : met à blanc le point (h,l)

public void boolean estNoir(int h, int l)
// prérequis : 0=<h<hauteur-1, 0=<l<largeur-1
// résultat : indique si le point (h,l) est noir

public String toString()
// résultat : visualisation textuelle de this
// les points noirs sont des '*' et les points blancs des '.'

public ImageBinaire extraitPartieConnexe()
// prérequis : this n'est pas blanche
// résultat : une partie connexe (arbitraire) de this (par exemple
// la partie connexe au premier point noir en haut à gauche)
// effet : met à blanc dans this les points de cette partie connexe
}
```

Pour la mise en œuvre, on décide de représenter une image au moyen d'une ***matrice*** **`M`** de ***booléens*** de taille **`hauteur`**×**`largeur`**.

`M[`*i***`] [`***j***`]`** = **`true`** si le point (*i*,*j*) est noir,

`M[`*i***`] [`***j***`]`** = **`false`** si le point (*i*,*j*) est blanc.

On commencera par rédiger et tester :

la déclaration des données de la classe **`ImageBinaire`**,

le constructeur (privé) qui crée la matrice **`M`**, non initialisée.

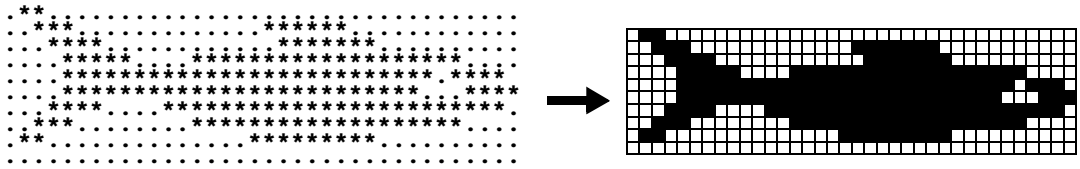
les méthodes **`hauteur`** et **`largeur`**,

la fonction statique **`blanche`**,

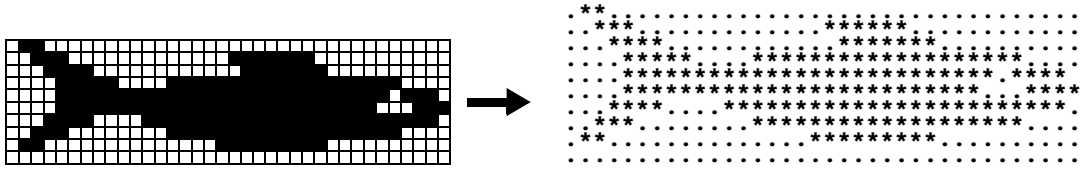
les méthodes **`noircir`** et **`blanchir`**,

la méthode **`estNoir`**,

la fonction **`apartirDe`** qui rend en résultat l'image binaire figurée par la chaîne de caractères **`s`**. La chaîne **`s`** présente l'image ligne par ligne et utilise des caractères **`'*'`** pour les points noirs, des caractères **`'.'`** pour les points blancs. Les caractères autres que **`'*'`** ou **`'.'`** sont autorisés mais devront être ignorés.
Par exemple, avec **`hauteur`** = 10 et **`largeur`** = 36, la chaîne de caractères de gauche donne pour résultat l'image de droite :



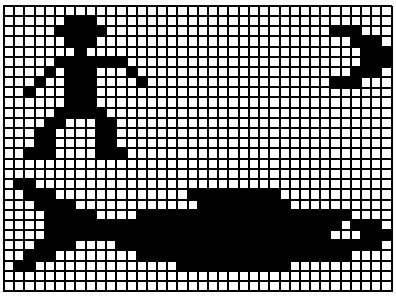
la méthode **`toString`** qui rend en résultat la chaîne de caractères qui visualise l'image, en utilisant des caractères **`'*'`** pour les points noirs, des caractères **`'.'`** pour les points blancs et des passages à la ligne **`'\n'`** pour terminer les lignes de l'image. Par exemple, pour l'image de gauche, le résultat est la chaîne de caractères de droite :



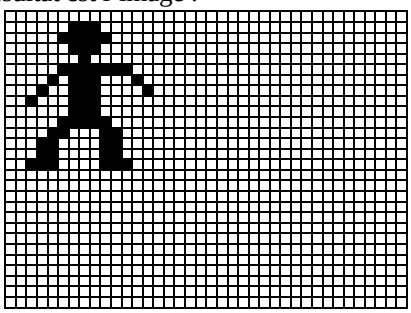
Extraction de parties connexes

Deux points sont ***adjacents*** s'il se touchent, c'est-à-dire si chacune de leurs coordonnées ne diffèrent que d'au plus une unité. On appelle ***partie connexe*** d'un point ***p*** dans une image binaire une image constituée de tous les points noirs que l'on peut atteindre depuis ***p*** en cheminant par des points noirs adjacents. Par exemple, l'image ci-après comporte 3 parties connexes (bonhomme, petite lune et poisson). On se propose de rédiger la méthode **`extraitPartieConnexe`** qui extrait une partie connexe d'une image.

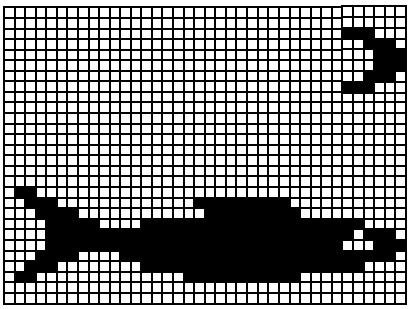
Par exemple, avec l'image **`im`** suivante :



le résultat est l'image :



et **`im`** est changée en :



Principe :

On utilise une liste de paires d'entiers **`aVisiter`** qui contient à tout moment de l'algorithme un ensemble de points qui restent "à visiter". Ces points à visiter sont les points adjacents aux points noirs déjà répertoriés de la partie connexe en cours de construction. On initialise la liste **`aVisiter`** avec un point noir de l'image, par exemple le premier point noir rencontré en parcourant l'image ligne par ligne, de gauche à droite et on met à blanc ce point de l'image. Ensuite, tant que la liste **`aVisiter`** n'est pas devenue vide :

on retire un point **`pt`** de la liste **`aVisiter`**,

on noircit ce point **`pt`** dans l'image résultat,

on ajoute à **`aVisiter`** les points noirs de l'image adjacents à **`pt`** et on les mets à blanc dans l'image.

Lorsque la liste **`aVisiter`** est devenue vide, l'image résultat est totalement construite : elle contient tous les points noirs de l'image origine connexes au premier point noir choisi.

Exercice 13.4 logique d'intervalles

1 - Opérations sur des intervalles d'entiers

On dispose de la classe **`PaireDEntiers`** :

```
class PaireDEntiers {
  public int premier;
  public int second;
  public PaireDEntiers(int p, int s) {
    premier=p; second=s;
  }
}
```

Dans ce qui suit on représente un intervalle d'entiers **[*inf*,*sup*]** par une paire d'entiers avec **`premier`**=***inf*** et **`second`**=***sup***. Une paire d'entiers **`p`** représente un intervalle si **`p.premier<=p.second`**.

On dit que deux intervalles d'entiers sont ***d'intersection vide*** si leur intersection est un ensemble vide. Exemples :

[2,6] et [7,9] sont d'intersection vide,

[2,6] et [8,12] sont d'intersection vide.

[2,6] et [3,9] ne sont pas d'intersection vide,

[2,6] et [3,5] ne sont pas d'intersection vide,

Étant donnés deux intervalles d'intersection non vide ***x*** et ***y***, ***intersection(x,y)*** est l'intervalle intersection de ***x*** et ***y***. Exemples :

intersection([2,6], [3,9]) = [3,6] ***intersection***([2,6], [3,5]) = [3,5]

On dispose des fonctions **`min`** et **`max`** qui rendent en résultat respectivement le minimum et le maximum de deux entiers passés en paramètres. On utilisera ces fonctions pour exprimer le plus simplement possible les réponses aux questions suivantes.

Rédiger la fonction **`intersectionVide`** spécifiée par :

```
static boolean intersectionVide(PaireDEntiers x, PaireDEntiers y)
// prérequis : x et y représentent des intervalles d'entiers
// résultat : indique si x et y sont d'intersection vide
```

Rédiger la fonction **`IntervalleIntersection`** spécifiée par :

```
static PaireDEntiers IntervalleIntersection
                              (PaireDEntiers x, PaireDEntiers y)
// prérequis : x et y représentent des intervalles
// d'intersection non vide
// résultat : l'intersection de x et y
```

On dit que deux intervalles d'entiers sont ***fusionnables*** s'ils ont une partie commune ou bien s'ils se touchent. Exemples :

[2,6] et [3,9] sont fusionnables,

[2,6] et [3,5] sont fusionnables,

[2,6] et [7,9] sont fusionnables.

[2,6] et [8,12] ne sont pas fusionnables.

Étant donnés deux intervalles fusionnables ***x*** et ***y***, on note ***fusion(x,y)*** l'intervalle union des ensembles représentés par ***x*** et ***y***. Exemples :

fusion([2,6], [3,9]) = [2,9]

fusion([2,6], [3,5]) = [2,6]

fusion([2,6], [7,9]) = [2,9]

Rédiger la fonction **`fusionnable`** spécifiée par :

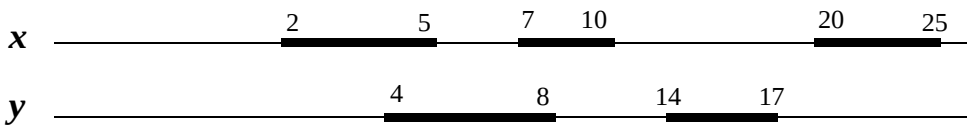
```
static boolean fusionnable(PaireDEntiers x, PaireDEntiers y)
// prérequis : x et y représentent
// des intervalles d'entiers
// résultat :
// indique si x et y sont fusionnables
```

Rédiger la fonction **`fusion`** spécifiée par :

```
static PaireDEntiers fusion(PaireDEntiers x, PaireDEntiers y)
// prérequis : x et y sont fusionnables
// résultat : la fusion de x et y
```

2 - Représentation des ensembles d'entiers par des collections d'intervalles

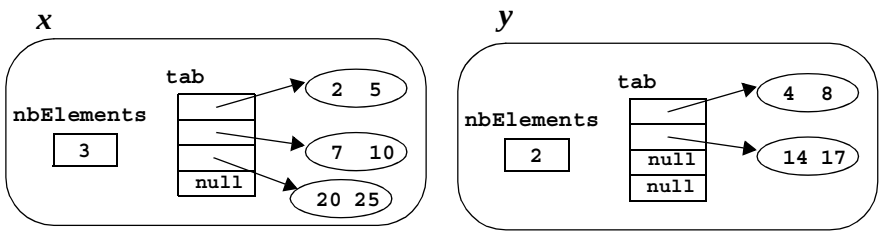
On se propose de représenter un ensemble d'entiers au moyen d'une suite d'intervalles ***non fusionnables***, triée par ordre croissant de leurs bornes. Par exemple, les ensembles ***x*** et ***y*** illustrés par les traits épais des dessins suivants :



seront représentés par les suites d'intervalles ***x*** : [2,5] [7,10] [20,25] et ***y*** : [4,8] [14,17]

On décide de représenter un ensemble par un tableau de paires d'entiers **`tab`** pour mémoriser ses intervalles, par ordre croissant de leurs bornes, et une variable entière **`nbElements`** qui indique le

nombre d'intervalles. Exemples :



Comme le montre ces exemples, le tableau **tab** n'est pas nécessairement entièrement utilisé.

On désire programmer une classe **EnsembleDEntiers** selon ce principe, dotée des opérations usuelles d'***union*** et d'***intersection*** et de ***test d'appartenance***. Les spécifications de cette classe sont données ci-dessous. Nous insistons sur le fait que les intervalles qui constituent un ensemble sont ***non fusionnables*** deux à deux et sont triés par ordre croissant dans **tab**.

```
class EnsembleDEntiers {

  public static EnsembleDEntiers aPartirDe(int[] T)
  // prérequis : le tableau T contient alternativement
  // les bornes inf et sup d'intervalles non fusionnables,
  // par ordre croissant des bornes
  // résultat : l'ensemble constitué de ces intervalles

  public boolean contient(int k)
  // résultat : indique si k appartient à this

  public static EnsembleDEntiers intersection
                  (EnsembleDEntiers x, EnsembleDEntiers y)
  // résultat: l'intersection de x et de y

  public static EnsembleDEntiers union
                (EnsembleDEntiers x,EnsembleDEntiers y)
  // résultat: l'union de x et de y
}
```

Commencer par rédiger et tester :

- Les déclarations de données de la classe **EnsembleDEntiers**,
- Son constructeur : **private EnsembleDEntiers(int tailleMax)**

  qui initialise l'état des données de façon à représenter une collection vide d'intervalles capable de contenir **tailleMax** intervalles,
- La méthode **aPartirDe** qui rend en résultat un ensemble formé par les intervalles indiqués dans un tableau d'entiers.

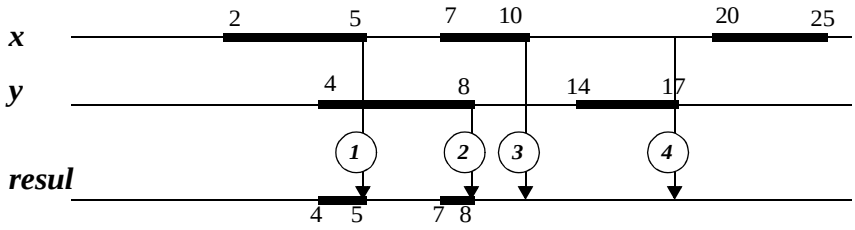
  Exemple : avec **T={2,5,7,10,20,25}**,

  le résultat de **EnsembleDEntiers.aPartirDe(T)** est l'ensemble constitué des intervalles : [2,5] [7,10] [20,25]
- La méthode **contient** qui indique l'appartenance d'un entier **k** à l'ensemble.

Opération d'intersection

Rédiger et tester la fonction **`intersection`** qui rend en résultat l'ensemble intersection de **`x`** et de **`y`**. On utilisera un algorithme analogue à celui vu en cours, qui consiste à parcourir les collections d'intervalles de **`x`** et **`y`** par ordre croissant, en progressant dans l'ensemble qui possède l'intervalle de borne supérieure minimale (ou dans les deux si les bornes supérieures sont égales).

Exemple :



En (***1***) on génère [4,5] dans le résultat et on progresse dans ***x*** sur l'intervalle [7,10]. En (***2***) on génère [7,8] dans le résultat et on progresse dans ***y***. En (***3***) on ne génère rien, car [7,10] et [14,17] sont d'intersection vide, et on progresse dans ***x***. En (***4***) on ne génère rien, car [14,17] et [20,25] sont d'intersection vide, et on progresse dans ***y***. Après (***4***), c'est terminé car un des ensembles, ***y*** ici, est entièrement visité.

Pour la taille du résultat, on utilisera la propriété suivante :

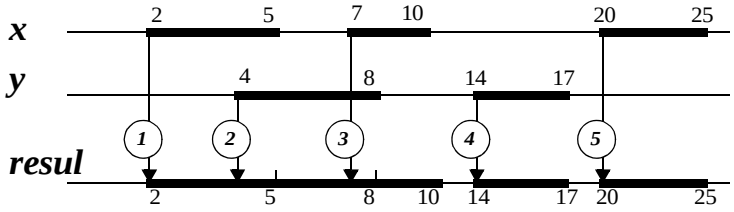
si ***x*** et ***y*** ne sont pas tous les deux vides,

nombreDIntervalles(intersection(x,y)) ≤ nombreDIntervalles(x) + nombreDIntervalles(y) - 1

Programmation de l'union

La fonction **`union`**, qui rend en résultat l'ensemble union de **`x`** et de **`y`**, est un peu plus difficile à programmer. Le principe général est est le suivant : on progresse dans les ensembles ***x*** et ***y*** par ordre croissant des bornes inférieures des intervalles ; chaque intervalle de ***x*** ou de ***y*** est soit fusionné au dernier intervalle du résultat s'il est fusionnable avec celui-ci, soit ajouté s'il n'est pas fusionnable.

Exemple :



En (***1***) on génère initialement [2,5] dans le résultat et on progresse dans ***x***. En (***2***) on fusionne [4,8] à [2,5] ce qui donne [2,8] et on progresse dans ***y***. En (***3***) on fusionne [7,10] à [2,8] ce qui donne [2,10] et on progresse dans ***x***. En (***4***) on génère [14,17] dans le résultat, car [14,17] et [2,10] ne sont pas fusionnables, et on progresse dans ***y***. En (***5***), on génère [20,25] dans le résultat, car [14,17] et [20,25] ne sont pas fusionnables et c'est terminé.

En analysant cet exemple, on constate qu'il est souhaitable de se doter d'une méthode :

```
private void cumuleAuBout(PaireDEntiers nouvelIntervalle)
```

qui, si **`nouvelIntervalle`** est fusionnable avec le dernier intervalle de **`this`** remplace ce dernier intervalle par sa fusion avec **`nouvelIntervalle`**, et sinon ajoute **`nouvelIntervalle`** à la suite dans le tableau.

Rédiger la méthode **`cumuleAuBout`** et l'utiliser pou rédiger la fonction **`union`**.

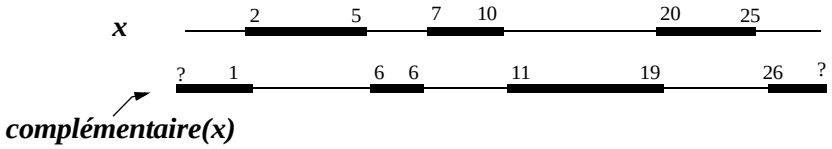
Pour la taille du résultat, on utilisera la propriété suivante :

nombreDIntervalles(union(x,y)) ≤ nombreDIntervalles(x) + nombreDIntervalles(y)

Complémentaire

Envisager la programmation d'une fonction **`complementaire`** qui rend en résultat le complémentaire d'un ensemble.

Il faut pour cela préciser le problème et trouver l'algorithme qui permet d'élaborer la représentation du complémentaire :



Exercice 13.5 Entiers non borné

On désire manipuler les entiers naturels (positifs ou nuls) sans limitation de grandeur (si ce n'est par la capacité mémoire de l'ordinateur). Ces entiers ne sont bien sûr pas représentables directement dans des mots du calculateur qui sont de taille fixe (et petite, 32 ou 64 bits). Nous devons donc les représenter par une classe **`GrandEntier`** dont voici les spécifications :

```
public class GrandEntier {
public static GrandEntier deValeur(int k)
// prérequis : k>=0
// résultat : le GrandEntier de valeur k
public String toString()
// résultat : l'écriture décimale de this

public static int comparaison(GrandEntier x, GrandEntier y)
// résultat : -1 si x<y, 0 si x=y, +1 si x>y
public static GrandEntier somme(GrandEntier x, GrandEntier y)
// résultat : x + y
  public static GrandEntier produit(GrandEntier x,GrandEntier y)
// résultat : x × y
}
```

Mise en œuvre

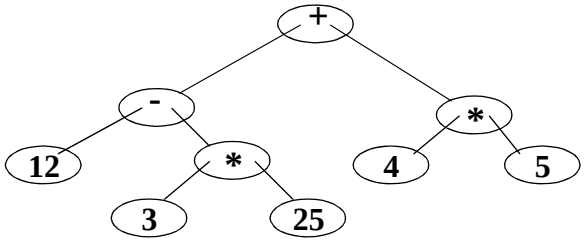
On décide de représenter un grand entier par la liste de ses chiffres (décimaux par exemple, bien qu'une base plus grande soit plus économique).

Exemple : l'entier 6 227 020 800 sera représenté par la liste <6,2,2,7,0,2,0,8,0,0>, chaque chiffre étant un "petit entier" (compris entre 0 et 9 si la base est décimale). On "normalisera" la représentation" en s'interdisant les chiffres 0 en poids fort.

CHAPITRE 14 Structures de données : arbres

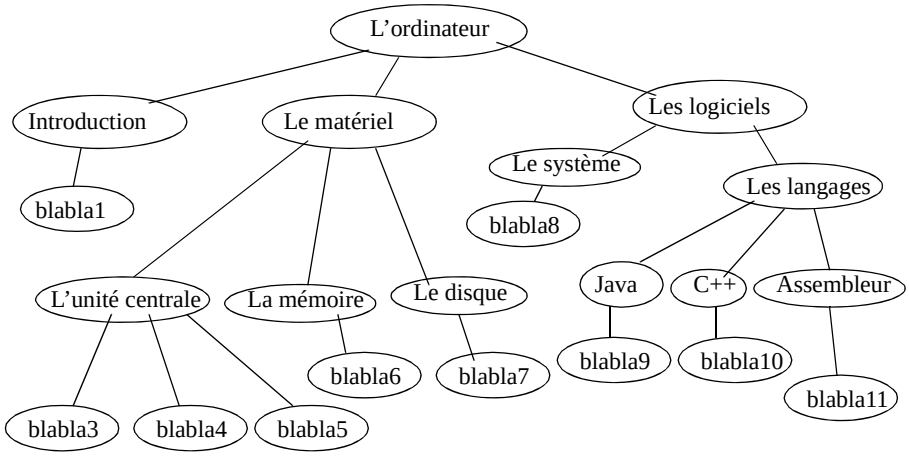
Un arbre est une structure de données qui permet de conserver de l'information de manière hiérarchique. Les arbres sont utilisés pour représenter des choses aussi différentes que :

- Une expression arithmétique : (12-3*25)*8 + 4*5 est vue comme un nœud "+" avec comme fils les deux expressions, l'une 12-3*25 et 4*5. A son tour, l'expression 12+3*25 est vue comme un nœud "-" avec comme fils les deux expressions, 12 et 3*25. L'expression simple 12 est vue comme une feuille de l'arbre (elle est atomique). L'expression 3*25 est vue comme un nœud "*" avec comme fils les deux expressions, 3 et 25, toutes deux vues comme des feuilles...



- Un texte documentaire, tel un cours ou un mode d'emploi : le document est découpé en chapitres, eux-mêmes découpes en sous-chapitres, découpées en sous-sous chapitres..., la hiérarchie se terminant par des paragraphes. Les divers niveaux de chapitrage sont vus comme des nœuds, chacun portant comme information son titre et comme fils les niveaux de chapitrage inférieurs ou les paragraphes qu'il contient. Les paragraphes sont vues comme les feuilles d'un tel arbre.

L'ordinateur
  1 - Introduction
    blabla1 blabla2
  2 - Le matériel
    2.1 - L'unité centrale
      blabla3 blabla4 blabla5
    2.2 - La mémoire
      blabla6
    2.3 - Le disque
      blabla7
  3 - Les logiciels
    3.1 - Le système
      blabla8
    3.2 - Les langages
      3.2.1 - Java
        blabla 9
      3.2.2 - C++
        blabla10
      3.3.3 - Assembleur
        blabla11



14.1 Spécification de la classe `ArbreBinaire`

Dans un premier temps nous allons nous intéresser à un cas simple d'arbres, les ***arbres binaires***. Ce sont des arbres dont les nœuds ont exactement 2 fils, appelés ***gauche*** et ***droite***. Nous considérons ici des arbres "non modifiables". Un tel arbre est simplement une valeur structurée que l'on peut :

- fabriquer à l'aide de l'opérateur **`feuille(`*`i`*`)`** pour obtenir une feuille porteuse de l'information ***i***,
- fabriquer à l'aide de l'opérateur **`noeud(`*`i`*`,`*`g`*`,`*`d`*`)`** pour obtenir un nœud porteur de l'information ***i*** et de fils ***g*** et ***d***,
- ou que l'on peut obtenir en tant que sous-arbres gauche ou droite d'un nœud par les opérations **`gauche`** et **`droite`**.

Voici la spécification proposée pour la classe **`ArbreBinaire`** :

```
class ArbreBinaire {

  public static ArbreBinaire
          noeud(String info,ArbreBinaire g,ArbreBinaire d)
  // résultat : le noeud <info,g,d>

  public static ArbreBinaire feuille(String info)
  // résultat : la feuille <info>

  public static ArbreBinaire aPartirDe(String s)
  // résultat : l'arbre binaire figuré par s selon une notation
  // parenthésée préfixée

  public boolean estNoeud()
  // résultat : indique si this est un noeud

  public boolean estFeuille()
  // résultat : indique si this est une feuille

  public String info()
  // résultat : l'info associée à this

  public ArbreBinaire gauche()
  // prérequis : this est un noeud
  // résultat : le fils gauche de this

  public ArbreBinaire droite()
  // prérequis : this est un noeud
  // résultat : le fils droite de this

  public String toString()
  // résultat : this en clair sous forme parenthésée préfixée

  public String enClairInfixe()
  // résultat : this en clair sous forme parenthésée infixe
}
```

14.2 Exemples simples d'utilisation de la classe `ArbreBinaire`

Avant d'étudier une mise en œuvre possible, voici quelques exemples simples qui illustrent les utilisations possibles de la classe **`ArbreBinaire`**. On considère la représentation d'expressions arithmétiques. Dans cet exemple :

- les informations associées aux nœud sont des symboles d'opération **`"+"`** et **`"*"`**,
- les informations associées aux feuilles sont des valeurs numériques exprimées en décimal, par exemple **`"3622"`**.

Soit l'expression ***expr1***= "12*56 +24",

elle s'écrit de façon infixée : +(*(12,56),24).

On peut obtenir l'arbre correspondant par :

```
ArbreBinaire expr1 = ArbreBinaire.apartirDe("+(*(12,56),24)");
```

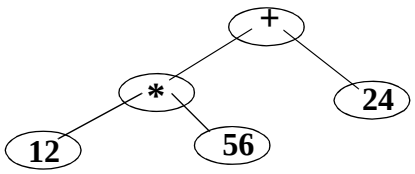
On pourrait également obtenir cet arbre par la séquence :

```
ArbreBinaire d12 = ArbreBinaire.feuille(12);
ArbreBinaire d56 = ArbreBinaire.feuille(56);
ArbreBinaire mulD12D56 = ArbreBinaire.noeud("*",d12,d56);
ArbreBinaire d24 = ArbreBinaire.feuille(24);
ArbreBinaire expr1=ArbreBinaire.noeud("+",mulD12D56,d24);
```

ou encore, sans identifications intermédiaires :

```
ArbreBinaire expr1=
    ArbreBinaire.noeud(
        "+",
        ArbreBinaire.noeud(
            "*",
            ArbreBinaire.feuille(12),
            ArbreBinaire.feuille(56)
        ),
        ArbreBinaire.feuille(24)
    );
```

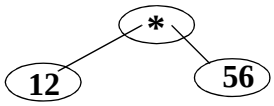
On obtient ainsi l'arbre que l'on peut figurer par le dessin suivant :



On peut capter le sous-arbre de gauche de **`expr1`** par :

```
ArbreBinaire expr2 = expr1.gauche();
```

`expr2` est l'arbre suivant (baptisé **`mulD12D56`** précédemment) :



14.3 Mise en œuvre de la classe `ArbreBinaire`

Nous allons représenter un arbre binaire au moyen des attributs de données suivant :

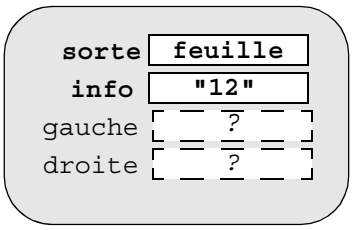
- **`sorte`** : l'indication de sa sorte, nœud ou feuille. Le type de cet attribut sera d'un type énuméré **`Sorte`**,
- **`info`** : l'information associée, de type **`String`**,

et, utiles uniquement ***dans le cas d'un nœud*** :

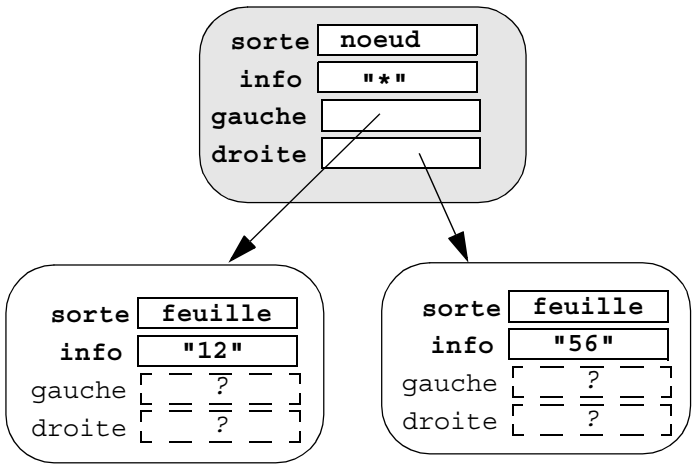
- **`gauche`** : la référence au fils gauche, de type **`ArbreBinaire`**,
- **`droite`** : la référence au fils droite, de type **`ArbreBinaire`**.

Exemples :

représentation de la feuille 12



représentation du nœud *(12,56)



Ceci conduit aux déclarations d'attributs de données et aux constructeurs suivants :

```
class ArbreBinaire {
```

```
  private static enum Sorte {noeud, feuille};
  private Sorte sorte;
  private String info;
  // cas d'un noeud
  private ArbreBinaire gauche;
  private ArbreBinaire droite;
```

```
  private ArbreBinaire(String info){
  // constructeur : feuille <info>
    sorte = Sorte.feuille;
    this.info=info;
  }
```

```
  private ArbreBinaire(String info,
                       ArbreBinaire g,ArbreBinaire d){
  // constructeur : noeud <info,gauche,droite>
    sorte = Sorte.noeud;
    this.info=info; gauche=g; droite=d;
  }
```

Les constructeurs sont privés : l'utilisateur ne pourra faire **`new ArbreBinaire(...)`**. C'est une bonne discipline lorsque, comme c'est la cas ici, on représente un "domaine de valeurs" et non pas de "vrais objets". On ne crée pas à proprement parler un arbre, mais on le "dénote" au moyen d'opérations telles que **`noeud`**, **`feuille`** ou **`aPartirDe`**. C'est comme pour les entiers, on ne crée pas 12, on le dénote par "12" ou par "8+4" ou par "2*6"...

Les opération les plus simples qui fabriquent des arbres sont **`feuille`** et **`noeud`** :

```
public static ArbreBinaire feuille(String info) {
// résultat : la feuille <info>
  return new ArbreBinaire(info);
}
```

```
public static ArbreBinaire
        noeud(String info,ArbreBinaire g,ArbreBinaire d){
// résultat : le noeud <info,g,d>
  return new ArbreBinaire(info,g,d);
}
```

L'opération suivante **`aPartirDe`** permet d'obtenir l'arbre dénoté par une chaîne de caractères sous forme parenthésée préfixée :

- la **feuille <*info*>** est simplement dénotée par la chaîne de caractère ***info***,
- le **nœud <*info*,g,d>** est dénotée par la chaîne de caractère ***info*`(`*gg*`,`*dd*`)`** où ***gg*** et ***dd*** sont les chaînes de caractères qui dénotent respectivement **g** et **d**.

Cette fonction n'est pas simple à réaliser. En toute rigueur, sa réalisation exige une "analyse syntaxique" de la chaîne de caractères qui dénote l'arbre, mais cela nous emmènerait trop loin. Ici, on peut profiter de ce que le prérequis nous garantit que la chaîne est "correcte" (sans erreur de syntaxe) : on peut alors s'en tirer au moyen de ***deux fonctions (privées) auxiliaires*** :

- **`extraitInfo(String s)`** qui rend en résultat la sous-chaîne du début de **`s`** qui constitue la donnée ***info*** de l'arbre : cette sous-chaîne commence au début de **`s`** et s'arrête soit à la rencontre du premier caractère séparateur **`'('`** ou **`','`** ou bien à la fin de **`s`**.

  Exemples :

  `extraitInfo("add(mul(12,56),24)")` vaut **`"add"`**

  `extraitInfo("12")` vaut **`"12"`**

- **`extraitMembre(String s, int i)`** qui rend en résultat la sous-chaîne de **`s`** qui commence en position **`i`** et représente un sous-arbre ***gauche*** ou ***droite*** d'un noeud : cette sous-chaîne se termine sur le premier caractère séparateur **`','`** ou **`')'`** rencontré qui ne soit pas inhibé par un prenthésage "**`(`**...**`)`**".

  Exemples :

  `extraitMembre("add(mul(12,56),24)",4)` vaut **`"mul(12,56)"`**

  `extraitMembre("add(mul(12,56),24)",15)` vaut **`"24"`**

```
private static String extraitInfo(String s){
// résultat : sous-chaîne de s commençant en 0 et se terminant
// au premier séparateur '(' ou ',' (exclu) rencontré
// ou à la fin de s
  int i=0;
  while(i<s.length() && s.charAt(i)!='(' && s.charAt(i)!=','){
    i++;
  }
  if(i==s.length()){return s;}
  else {return s.substring(0,i);}
}
```

```
private static String extraitMembre(String s,int i){
// résultat : sous-chaine de s, à partir de i, se terminant au
// premier séparateur ')' ou ',' non inhibé par un parenthésage
// '('...')'
  int j=i;
  int niveau=0;
  while(niveau!=0 || (s.charAt(j)!=')' && s.charAt(j)!=',')){
    if(s.charAt(j)=='('){niveau++;}
    else if(s.charAt(j)==')'){niveau--;}
    j++;
  }
  return s.substring(i,j);
}
```

Avec ces deux fonctions, nous sommes en mesure de programmer **`aPartirDe`**. Sa réalisation est naturellement ***récursive*** :

```
public static ArbreBinaire aPartirDe(String s){
  // résultat : l'arbre binaire figuré par s selon une notation
  // parenthésée préfixée
    String info=extraitInfo(s);
    if(info.length()==s.length()){// feuille
      return feuille(info);
    }
    else{// noeud
      String sg=extraitMembre(s,info.length()+1);
      String sd=
       extraitMembre(s,info.length()+1+sg.length()+1);
      return noeud(info,aPartirDe(sg),aPartirDe(sd));
    }
  }
```

On extrait la partie ***info*** en début de **`s`**. Si cette partie ***info*** est **`s`** dans son intégralité, **`s`** représente la feuille **`feuille(`*`info`*`)`**. Sinon **`s`** représente un nœud : on extrait les chaînes **`sg`** et **`sd`** qui représentent les fils ***gauche*** et ***droite*** et **`s`** représente le nœud **`noeud(`*`gauche,droite`*`)`**.

Les fonctions **estNoeud**, **estFeuille**, **gauche** et **droite** sont particulièrement simples à réaliser car leur résultat s'obtient directement par un attribut de l'objet :

```
public boolean estNoeud(){
  // résultat : indique si this est un noeud
    return sorte==Sorte.noeud;
  }
```

```
  public boolean estFeuille(){
  // résultat : indique si this est une feuille
    return sorte==Sorte.feuille;
  }
```

```
  public String info(){
  // résultat : l'info associée à this
    return info;
  }
```

```
  public ArbreBinaire gauche(){
  // prérequis : this est un noeud
  // résultat : le fils gauche de this
    return gauche;
  }
```

```
  public ArbreBinaire droite(){
  // prérequis : this est un noeud
  // résultat : le fils droite de this
    return droite;
  }
```

Les fonctions de visualisation en clair, **toString** et **enClairInfixe**, sont simple à réaliser de façon ***récursive***. Pour une feuille, le résultat est l'attribut **info**, et pour un nœud le résultat s'obtient soit par concaténation de **info**, de parenthèses, de virgule et de visualisation en clair des fils **gauche** et **droite** :

```
 public String toString() {
  // résultat : this en clair sous forme parenthésée préfixée
    if (sorte==Sorte.noeud){
      return info+"("+gauche+","+droite+")";
    }
    else /* sorte==feuille */ {return info;}
  }
```

```
  public String enClairInfixe() {
  // résultat : this en clair sous forme parenthésée infixe
    if (sorte==Sorte.noeud){
      return "("+gauche.enClairInfixe()
                +info+droite.enClairInfixe()+")";
    }
    else /*sorte==feuille*/ {return info;}
  }
}
```

14.4 Autre exemple d'utilisation de `ArbreBinaire`

Les exemples suivants utilisent, comme précédemment, des arbres binaires pour représenter des expressions arithmétiques. Les opérations sont (par exemple) "addition" et "multiplication". Ces opérations sont notées par les symboles **`"+"`** et **`"*"`**.

On peut vouloir traiter des expressions arithmétiques dans lesquelles les opérations sont notées par les chaînes de caractères **`"add"`** et **`"mul"`**. Un besoin se fait sentir de vouloir remplacer ces chaînes par les symboles **`"+"`** et **`"*"`** pour en quelque sorte les "normaliser". C'est ce que réalise la fonction suivante :

```
static ArbreBinaire
        remplaceInfoNoeud(String s1,String s2,ArbreBinaire a){
// résultat : arbre obtenu en remplaçant dans a
// les infos des noeuds égaux à s1 par s2
  if(a.estNoeud()){
    String i=a.info(); if(i.equals(s1)){i=s2;}
    return ArbreBinaire.noeud(i,
                      remplaceInfoNoeud(s1,s2,a.gauche()),
                      remplaceInfoNoeud(s1,s2,a.droite())
          );
    }
    else /* a.estFeuille() */ {return a;}
  }
```

Ainsi, l'expression **`expr1`** :

```
ArbreBinaire expr1 =
    ArbreBinaire.aPartirDe("add(12,mul(25,add(mul(34,56),78))))");
```

Pourra être convertie en une expression **`expr2`** dont la forme en clair serait

```
"+(12,*(25,+(mul(34,56),78))))"
```

grâce à :

```
ArbreBinaire expr2 =
  remplaceInfoNoeud("add","+",remplaceInfoNoeud("mul","*",expr1));
```

Comme le montre cet exemple, les parcours d'arbres sont naturellement récursifs : pour remplacer ***s1*** par ***s2*** dans un arbre qui est un nœud, on utilise le remplacement de ***s1*** par ***s2*** dans les fils ***gauche*** et ***droite*** et on recrée le nœud en remplaçant éventuellement l'information associée.

La fonction suivante “évalue” une expression arithmétique constituées d’opérations notées **"+"** et **"*"**, c’est-à-dire rend le résultat numérique du calcul suggéré par l’expression :

```
static int eval(ArbreBinaire expr) {
// prérequis : expr représente un expression entière
// avec "+" ou "*" pour info associées aux noeuds et des
// représentation décimales de nombres associées aux feuilles
// résultat : l'évaluation de expr
  if(expr.estFeuille()) {return Integer.valueOf(expr.info());}
  else /* expr.estNoeud() */ {
    if (expr.info().equals("+")){
      return eval(expr.gauche())+eval(expr.droite());
    }
    else /* expr.info().equals("*") */ {
      return eval(expr.gauche())*eval(expr.droite());
    }
  }
}
```

On peut ainsi construire une expression et l’évaluer :

```
ArbreBinaire expr=ArbreBinaire.aPartirDe("+(*(12,56),24)");
System.out.println("expr = "+expr);
System.out.println(
       "expr.enClairInfixe() = "+expr.enClairInfixe());
System.out.println("eval(expr) = "+eval(expr));
```

Ce morceau de programme affiche :

```
expr = +(*(12,56),24)
expr.enClairInfixe() = ((12*56)+24)
eval(expr) = 696
```

14.5 Spécification de la classe `Arbre`

Nous allons nous intéresser maintenant aux arbres dont chaque nœud peut avoir un nombre quelconque de fils. Ces arbres seront offerts par la classe **`Arbre`**. La spécification de la classe **`Arbre`** ressemble beaucoup à celle de **`ArbreBinaire`**. Les seules différences proviennent de la multiplicité quelconque des fils d’un nœud. Ces différences sont :

- la ***création d’un nœud***, réalisée par la fonction :

  `Arbre noeud(String info, Arbre[] fils)`

  qui a pour paramètre un tableau d’arbres pour indiquer les fils du nœud,
- l’accès ***aux fils d’un nœud*** qui ne peut plus se faire au moyen de **`gauche`** et **`droite`** puis qu’il y a un nombre quelconque de fils; cet accès se fait au moyen de la méthode :

  `public Parcours<Arbre> parcoursLesFils()`

qui rend en résultat un parcours initialisé sur le premier fils du nœud; les fils pourront alors être obtenus par les méthodes usuelles d'un parcours : **elementCourant**, **suivant** et le test **estEnFin**.

```
class Arbre {

  public static Arbre noeud(String info, Arbre[] fils)
  // résultat : l'arbre noeud<info,fils[0]...>

  public static Arbre feuille(String info)
  // résultat : l'arbre feuille<info>

  public boolean estNoeud()
  // résultat : indique si this est un noeud

  public boolean estFeuille()
  // résultat : indique si this est une feuille

  public String info()
  // résultat : l'info associée à this

  public Parcours<Arbre> parcoursLesFils()
  // prérequis : this est un Noeud
  // résultat : parcours initialisé sur le premier fils de this

  public static Arbre aPartirDe(String s)
  // résultat : l'arbre binaire figuré par s selon une notation
  // parenthésée préfixée

  public String toString()
  // résultat : this en clair sous forme parenthésée préfixée
}
```

14.6 Exemple d'utilisation de la classe **Arbre**

Avant d'étudier une mise en œuvre possible, voici un exemple simple qui illustre une utilisation de la classe **Arbre**.

On considère la représentation de documents. Dans cet exemple :

- les informations associées aux nœud sont des titres de chapitres de divers niveaux,
- les informations associées aux feuilles sont les textes des paragraphes du document.

Voici un exemple d'un tel document :

Le monde animal
- 1 - Les mollusques
  - Ils sont mous
  - 1.1 - les bivalves
    - ils ont deux valves
  - 1.2 - Les gastéropodes
    - Ils ont une coquille en colimaçon
    - Ils ont deux cornes
- 2 - Les vertébrés
  - 2.1 - Les poissons
    - Ils nagent
  - 2.2 - les reptiles
    - Ils marchent ou rampent
    - Ils ont le sang froid
  - 2.3 - Les mammifères
    - Ils ont le sang chaud
    - 2.3.1 - Les cétacés
      - Ils nagent
    - 2.3.2 - Les bovidés
      - Ils marchent
      - Ils vivent dans les pâturages
- 3 - Conclusion
  - voila c'est fini

Pour obtenir l'arbre correspondant à ce document, il suffit d'utiliser la fonction **aPartirDe** :

```
Arbre document = Arbre.aPartirDe(
    "Le monde animal("
    + "1 - Les mollusques("
      + "Ils sont mous,"
      + "1.1 - Les bivalves(ils ont deux valves),"
      + "1.2 - Les gastéropodes("
          + "Ils ont une coquille en colimaçon,Ils ont deux cornes)"
    + "),"
    + "2 - Les vertébrés("
      + "2.1 - Les poissons(Ils nagent),"
      + "2.2 - Les reptiles("
          + "Ils marchent ou rampent,Ils ont le sang froid)"
      + "2.3 - Les mammifères(Ils ont le sang chaud),"
        + "2.3.1 - Les cétacés(Ils nagent),"
        + "2.3.2 - Les bovidés("
          + "Ils marchent,Ils vivent dans les pâturages)"
    + "),"
    + "3 - Conculsion(voila c'est fini)"
    +")"
    );
```

Si on visualise cet arbre par la méthode **`toString()`** :

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

on obtient :

```
Le monde animal(1 - Les mollusques(Ils sont mous,1.1 - Les bivalves(Ils ont deux valves),1.2 - Les gastéropodes(Ils ont une coquille en colimaçon,Ils ont deux cornes)),2 - Les vertébrés(2.1 - Les poissons(ils nagent),2.2 - Les reptiles(Ils marchent ou rampent,Ils ont le sang froid),2.3 - Les mammifères(ils ont le sang chaud,2.3.1 - Les cétacés(Ils nagent),2.3.2 - Les bovidés(Ils marchent,Ils vivent dans les pâturages))),3 - Conculsion(voila c'est fini))
```

Ce n'est pas très joli... mais l'information y est.

Un traitement que l'on peut vouloir faire sur un tel document est de ***l'indenter***. Ceci consiste à rajouter un certain nombre d'espaces supplémentaires à gauche à chaque passage à un niveau de chapitrage inférieur. Cette fonction d'indentation pourra être réalisée par la fonction suivante :

```
static String indente(Arbre a) {
// résultat : a en clair indenté
  return indente(a,0);
}

static String indente(Arbre a,int i) {
// résultat : a en clair indenté de i espaces
  if (a.estNoeud()){
    String resul=blancs(i)+a.info()+"\n";
    Parcours<Arbre> p = a.parcoursLesFils();
    while(!p.estEnFin()){
      resul=resul+indente(p.elementCourant(),i+4);
      p.suivant();
    }
    return resul;
  }
  else /* a.estFeuille() */{
    return blancs(i)+a.info()+"\n";
  }
}

static String blancs(int i){
// résultat : une chaîne de i espaces
  String resul="";
  for(int k=0; k<i; k++){resul=resul+" ";}
  return resul;
}
```

C'est une fonction naturellement récursive. Plus exactement la fonction proprement récursive est

```
static String indente(Arbre a,int i)
```

qui a un paramètre de plus, **`i`**, qui est la quantité initiale désirée d'espaces à gauche.

La fonction **`static String indente(Arbre a)`** appelle cette dernière avec **`i`**=0.

Si on imprime le résultat de l'indentation :

```
System.out.println(indente(document));
```

On obtient le texte indenté :

```
Le monde animal
    1 - Les mollusques
        Ils sont mous
        1.1 - Les bivalves
            ils ont deux valves
        1.2 - Les gastéropodes
            Ils ont une coquille en colimaçon
            Ils ont deux cornes
    2 - Les vertébrés
        2.1 - Les poissons
            Ils nagent
        2.2 - Les reptiles
            Ils marchent ou rampent
            Ils ont le sang froid
        2.3 - Les mammifères
            2.3.1 - Les cétacés
                Ils nagent
            2.3.2 - Les bovidés
                Ils marchent
                Ils vivent dans les pâturages
    3 - Conclusion
        voila c'est fini
```

14.7 Mise en œuvre de la classe `Arbre`

Nous allons représenter un arbre au moyen des attributs de données suivant :

- **`sorte`** : l'indication de sa sorte, nœud ou feuille.
- **`info`** : l'information associée, de type **`String`**,

et, utiles uniquement ***dans le cas d'un nœud*** :

- **`lesFils`** : une liste d'éléments de type **`Arbre`**.

Voici les déclarations des attributs de donnée et les constructeurs :

```
class Arbre {
  public static enum Sorte {noeud,feuille}; private Sorte sorte;
  // cas d'un noeud
  Liste<Arbre> lesFils;
  // cas d'un noeud ou d'une feuille
  private String info;
```

```
  private Arbre(String info){
  // constructeur : arbre feuille<info>
    sorte = Sorte.feuille; this.info=info;
  }
```

```
  private Arbre(String info, Liste<Arbre> fils){
  // constructeur : arbre noeud<info,liste des fils>
    sorte = Sorte.noeud; this.info=info; lesFils=fils;
  }
```

Les fonctions (publiques) de fabrication des nœuds et des feuilles :

```
public static Arbre noeud(String info, Arbre[] fils) {
 // résultat : l'arbre noeud<info,fils[0]...>
   Liste<Arbre> lesFils=new Liste<Arbre>();
   for(int i=0;i<fils.length;i++){
     lesFils.ajouteEnQueue(fils[i]);
   }
   return new Arbre(info,lesFils);
 }
```

```
 public static Arbre feuille(String info) {
 // résultat : l'arbre feuille<info>
   return new Arbre(info);
 }
```

Les fonctions **estNoeud**, **estFeuille** et **info** sont strictement identiques à celles de **ArbreBinaire** :

```
public boolean estNoeud(){
 // résultat : indique si this est un noeud
   return sorte==Sorte.noeud;
 }
```

```
 public boolean estFeuille(){
 // résultat : indique si this est une feuille
   return sorte==Sorte.feuille;
 }
```

```
 public String info(){
 // résultat : l'info associée à this
   return info;
 }
```

La fonction **parcoursLesFils** rend simplement un parcours sur **lesFils** :

```
public Parcours<Arbre> parcoursLesFils(){
 // prérequis : this est un Noeud
 // résultat : parcours initialisé sur le premier fils de this
   return lesFils.nouveauParcours();
 }
```

La réalisation de **aPartirDe** est un peu plus complexe. La méthode utilisée est similaire à celle de **ArbreBinaire**, mais au lieu d'extraire les chaînes qui représentent les fils **gauche** et **droite** d'un nœud, il faut extraire une quantité non déterminée à l'avance de chaînes qui représentent les fils du nœud.

```
private static String extraitInfo(String s){
// résultat : sous-chaîne de s commençant en 0 et se terminant
// au premier caractère séparateur '(' ou ',' (exclu) rencontré
// ou à la fin de s
  int i=0;
  while(i<s.length() && s.charAt(i)!='(' && s.charAt(i)!=','){
    i++;
  }
  if(i==s.length()){return s;}
  else {return s.substring(0,i);}
}
```

```
private static String extraitMembre(String s,int i){
// résultat : sous-chaine de s, à partir de i, se terminant au
// premier séparateur ')' ou ',' non inhibé par un parenthésage
// ouverture '(' fermeture ')'
  int j=i;
  int niveau=0;
  while(niveau!=0 || (s.charAt(j)!=')' && s.charAt(j)!=',')){
    if(s.charAt(j)=='('){niveau++;}
    else if(s.charAt(j)==')'){niveau--;}
    j++;
  }
  return s.substring(i,j);
}
```

```
public static Arbre aPartirDe(String s){
// résultat : l'arbre binaire figuré par s selon une notation
// parenthésée préfixée
  String info=extraitInfo(s);
  if(info.length()==s.length()){// feuille
    return feuille(info);
  }
  else{// noeud
    int j=info.length()+1;
    Liste<Arbre> lesFils = new Liste<Arbre>();
    if(s.charAt(j)!=')'){// on autorise un nœœœoeud avec 0 fils
      String sFils=extraitMembre(s,j);
      lesFils.ajouteEnQueue(aPartirDe(sFils));
      j=j+sFils.length();
      while(s.charAt(j)==','){
        j=j+1;
        sFils=extraitMembre(s,j);
        lesFils.ajouteEnQueue(aPartirDe(sFils));
        j=j+sFils.length();
      }
    }
    return new Arbre(info,lesFils);
  }
```

Pour terminer, la fonction **toString**, très semblable à celle de **ArbreBinaire** :

```
 public String toString() {
  // résultat : this en clair sous forme parenthésée préfixée
    if (sorte==Sorte.noeud){
      String resul =  info+"(";
      Parcours<Arbre> p = lesFils.nouveauParcours();
      if(!p.estEnFin()){/* liste de fils non vide */
        resul = resul+p.elementCourant();
        p.suivant();
        while(!p.estEnFin()){
          resul=resul+","+p.elementCourant();
          p.suivant();
        }
      }
      return resul+")";
    }
    else /* sorte==feuille */ {
      return info;
    }
  }
}
```

CHAPITRE 15 Représentation des informations dans les ordinateurs

15.1 Nombres et représentations de nombres

15.1.1 Notion de représentation

Les nombres entiers sont une ***abstraction***. On peut définir les nombres entiers de façon informelle : un nombre entier est la mesure de la quantité de choses individuellement discernables, par exemple une quantité de pommes ou de billes. On peut les définir plus formellement : ce sont les éléments d'un ensemble, habituellement appelé *N*, qui satisfait aux axiomes de l'arithmétique.

Au sujet des nombres, nous sommes amenés à ***faire des calculs***. Un calcul, qu'il soit fait à la main ou par une machine, est nécessairement un processus physique : il doit travailler sur des ***marques concrètes*** qui représentent les nombres, selon certaines conventions.

Nous connaissons plusieurs systèmes de marques concrètes pour représenter les nombres, par exemple le nombre ***12*** peut être représenté par :

un lot de 12 billes

des chiffres romains

XII

une suite de deux chiffres décimaux

12

une suite de quatre chiffres binaires

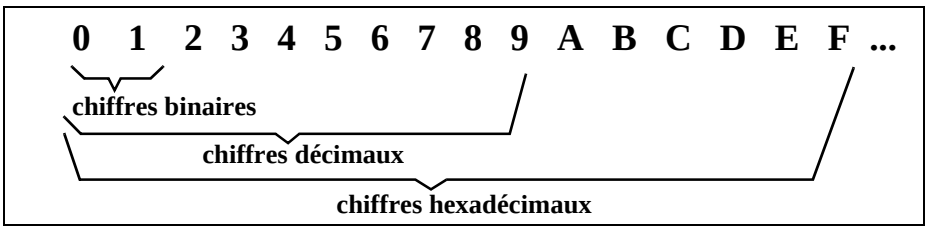
1100

Il ne faut pas confondre nombre, entité abstraite, et représentation de nombre, marque concrète qui permet de faire des calculs au moyen de machines (éventuellement humaines) exécutant des algorithmes. Si on accepte cette distinction, ***il n'y a pas de nombres dans les machines***, et il n'y en aura jamais, il n'y a que des représentations de nombres. Ce n'est pas un problème technologique, c'est par nature qu'un nombre (ou toute autre entité abstraite) n'a pas sa place dans une machine.

15.1.2 Numérations positionnelles

Les représentations des nombres les plus utilisées sont les ***représentations positionnelles***. Dans une telle représentation, on utilise une collection finie (et généralement petite) de symboles appelés ***chiffres***.

On utilise généralement les symboles suivants :



Le nombre de chiffres utilisés s'appelle la ***base***.

Les bases les plus utilisées sont la base 2 (deux chiffres 0 et 1 appelés chiffres binaires), la base 10 (dix chiffres de 0 à 9 appelés chiffres décimaux) et la base 16 (seize chiffres de 0 à 9 puis de A à F appelés chiffres hexadécimaux).

La base 60 est également utilisée, de façon limitée mais quotidienne, pour compter le temps en heures/minutes/secondes.

Dans une représentation positionnelle en base *b*, à chaque chiffre on associe une valeur de 0 à *b*-1 :

chiffre	***c***	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**A**	**B**	**C**	**D**	**E**	**F**
valeur entière associée	***val(c)***	***0***	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***	***9***	***10***	***11***	***12***	***13***	***14***	***15***

Un nombre est représenté par une succession de chiffres $c_{n-1}c_{n-2}...c_1c_0$. Le nombre représenté est :

$$interprétation_b(c_{n-1}c_{n-2}...c_1c_0) = val(c_{n-1})\times b^{n-1} + val(c_{n-2})\times b^{n-2} + ... \; val(c_1)\times b + val(c_0)$$

On a coutume d'appeler ***interprétation*** la fonction qui va des marques concrètes de la représentation (succession de chiffres ici) vers les valeurs abstraites représentées (un nombre entier ici). Pour distinguer les diverses fonctions d'interprétation associées aux diverses bases, nous avons placé la base en indice du nom de la fonction.

Exemples :

interprétation en base 10 de 3058 :

$$interprétation_{10}(3058) = 3\times 10^3 + 0\times 10^2 + 5\times 10 + 8 = 3058$$ (on s'y attendait)

interprétation en base 2 de 1101 :

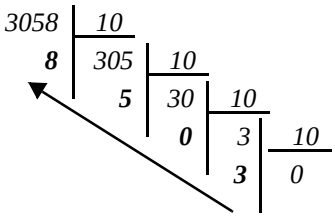
$$interprétation_2(1101) = 1\times 2^3 + 1\times 2^2 + 0\times 2 + 1 = 8+4+1 = 13$$

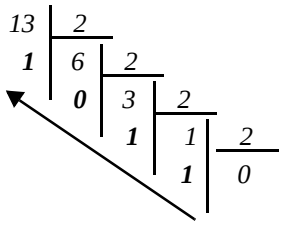
interprétation en base 16 de 1A4 :

$$interprétation_{16}(1A4) = 1\times 16^2 + 10\times 16 + 4 = 256+160+4 = 420$$

On a coutume d'appeler ***représentation*** la fonction inverse qui va des valeurs abstraites vers les marques concrètes de la représentation. La méthode pour trouver la représentation en base *b* d'un nombre consiste à diviser itérativement par la base. Les chiffres de la représentation sont les chiffres correspondants aux restes des divisions successives.

Exemples : représentations en base 10 et en base 2 :





***représentation*$_{10}$*(3058)* = 3058** (on s'y attendait) ***représentation*$_{2}$*(13)* = 1101**

Les représentations positionnelles sont pratiques pour effectuer les calculs, notamment addition et soustraction selon les algorithmes bien connus enseignés à l'école primaire.

Une invention essentielle de ce mode de représentation fut le chiffre 0. Il sert, comme on le sait, à indiquer qu'il n'y a aucun terme b^i en position ***i***. Son rôle est donc essentiellement "d'occuper une place où il n'y a rien". Avant l'invention du 0, il fallait user de périphrases pour citer certains nombres : *246* s'écrivait 246, alors que 206 devait s'énoncer par quelque chose comme "2 centaines et 6".

15.1.3 Chiffres binaires : bit

Les machines informatiques utilisent la base 2, car le plus petit élément d'information que l'on sait réaliser économiquement et avec fiabilité possède 2 valeurs, 0 et 1. Un chiffre binaire est souvent appelé ***bit***, abréviation pour *binary digit*, terme anglais pour "chiffre binaire".

À cause de son petit nombre de chiffres, l'écriture binaire est peu lisible pour l'homme : il est difficile de voir si deux suites de chiffres binaires sont égales, ou de voir lequel de deux nombres représentés est le plus grand.

Pour faciliter la lecture, on utilise souvent une base "cousine", la base 16. La correspondance entre base 2 et base 16 est triviale car $16=2^4$. Il suffit donc de regrouper les chiffres binaires par 4 et les remplacer par leur correspondant en base 16 :

Binaire	Hexadécimal
0 0 0 0	0
0 0 0 1	1
0 0 1 0	2
0 0 1 1	3
0 1 0 0	4
0 1 0 1	5
0 1 1 0	6
0 1 1 1	7
1 0 0 0	8
1 0 0 1	9
1 0 1 0	A
1 0 1 1	B
1 1 0 0	C
1 1 0 1	D
1 1 1 0	E
1 1 1 1	F

Exemple :

le nombre qui se représente en binaire par

0101 1011 0011 0001

se représente en hexadécimal par :

5 B 3 1

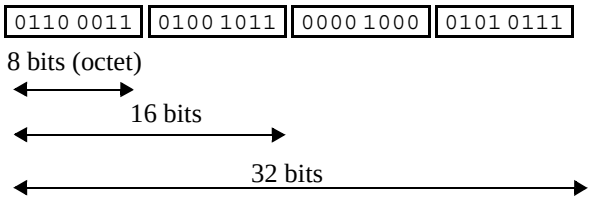
Remarque : ceci est une propriété générale. Le passage de la représentation en base b à la représentation en base b^k se fait en regroupant les chiffres par paquets de k chiffres et en remplaçant chaque paquet par le chiffre correspondant de la base b^k. Ici nous sommes passés de la base b=2 à la base $b^k=2^4=16$.

15.2 Mémoire

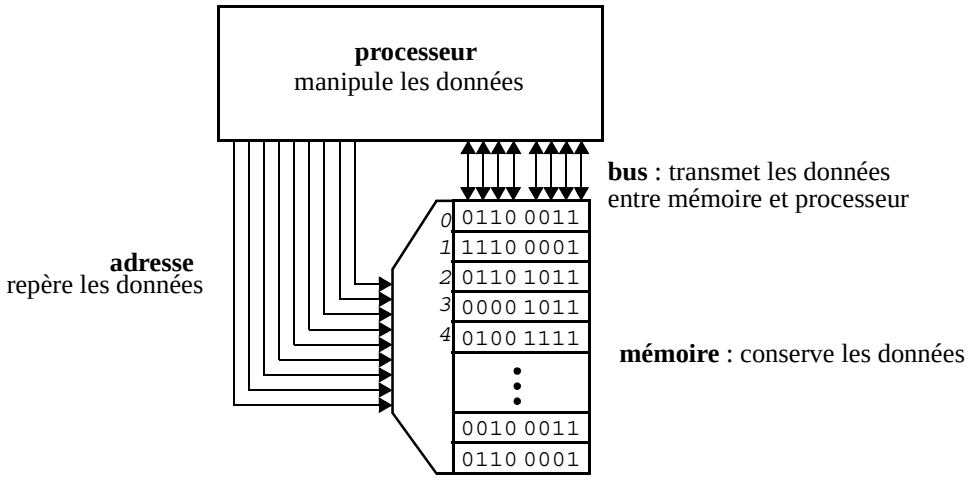
La mémoire est le composant d'un ordinateur dans lequel le processeur range et récupère les informations. Le plus petit élément de mémorisation est capable de conserver un bit. C'est un élément électronique à deux états stables dans lequel l'information est représentée par un potentiel électrique valant 0v ou +5v. Une mémoire de un bit permet donc la représentation d'une valeur binaire, selon une convention, par exemple 0 : 0v et 1 : +5v.

Les bits sont regroupés en paquets de taille fixe appelés ***mots***.

Les tailles les plus courantes sont 8 bits, appelé aussi ***octet*** (*byte* en anglais), 16 bits et 32 bits.



Les mots d'une mémoire sont repérés par un numéro appelé ***adresse***. Depuis plusieurs années, l'octet s'est imposé comme unité adressable. Pour lire ou écrire des données dans la mémoire, le processeur indique l'adresse du mot concerné et échange la donnée au moyen d'un ***bus*** de données.



15.3 Représentations usuelles des types simples

Nous présentons succinctement dans ce paragraphe les représentations usuelles des types simples dans les ordinateurs :

- caractères,
- nombres entiers positifs,
- nombres entiers relatifs,
- nombres réels.

Dans un ordinateur, tous les types de données sont nécessairement représentés par des ***suites finies de bits***. Une suite de n bits peut prendre 2^n valeurs et peut donc représenter un domaine d'au plus 2^n éléments. À part cette contrainte sur sa taille, le domaine représenté peut être quelconque : il suffit d'attribuer une signification particulière à chacune des suites de bits.

Exemples :

Avec *2* bits x_1x_0, on dispose de $2^2 = 4$ valeurs, avec lesquels on peut représenter les domaines suivants :

bits x_1x_0	ensemble de 4 couleurs	ensemble de 4 lettres	4 nombres	4 instructions
00	rouge	A	0	avancer
01	bleu	B	1	reculer
10	vert	C	2	tourner
11	jaune	D	3	arrêter

Comme le montrent ces exemples, une suite de bits (plus généralement une collection de marques concrètes) ***n'a pas de signification en soi***. C'est seulement ***à travers une interprétation*** particulière qu'elle prend un sens. Il est important de remarquer que les interprétations ne sont pas du domaine des machines. Ce sont toujours des êtres humains, ou des phénomènes physiques dans le cas d'application de contrôle d'appareils, qui en dernier ressort interprètent l'information.

Dans les langages de programmation, c'est grâce en partie à la notion de ***type*** que le programmeur indique l'interprétation qu'il attribue à ses données. Ceci permet au compilateur de choisir la représentation convenable en terme de bits, et sert de garde fou en n'autorisant sur ces représentations que les opérations qui ont un sens dans le cadre de ces interprétations : addition et soustraction pour les entiers, test pour les booléens, concaténation pour les chaînes de caractères...

Exemple :

Avec les représentations sur 2 bits évoquées précédemment, le programmeur ayant déclaré une variable **`coul`**, de type **`couleur`**, il pourra exécuter l'affectation **`coul=vert`**. Le compilateur choisira une mémoire de deux bits d'adresse ***coul*** (deux bits choisis dans un octet puisque la mémoire est adressable par octets).

L'affectation : **`coul=vert;`**

sera traduite par le compilateur en : ***ranger dans la mémoire d'adresse coul les bits*** **`10`**

Le programmeur pourra utiliser toute opération ayant un sens pour les couleurs, par exemple les fonctions **`estCouleurFroide`** ou **`estCouleurChaude`** à résultat booléen qui indiquent respectivement si une couleur est froide (vert, bleu) ou chaude (rouge, jaune). Il pourra écrire :

```
if (estCouleurFroide(coul)) {...}
```

cela sera accepté et traduit en

si la mémoire d'adresse **`coul`** ***contient*** **`01`** ***ou*** **`10 : ...`**

En revanche, le compilateur refusera l'instruction :

```
coul = coul+1;
```

car c'est un ***non-sens*** d'incrémenter une couleur, bien que la machine sache incrémenter **`10`** pour donner **`11`**.

15.3.1 Représentation des caractères : ASCII et UNICODE

Les caractères sont les marques qui servent à constituer des textes. Une représentation des caractères doit d'abord fixer l'ensemble des caractères que l'on représente. Une représentation très répandue est la représentation ASCII[1]. Elle représente un ensemble de 128 caractères sur 7 bits ($128=2^7$).

La représentation est indiquée sur la table suivante :

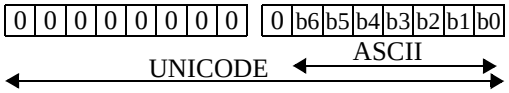
b3-0 \ b6-4	0 0 0	0 0 1	0 1 0	0 1 1	1 0 0	1 0 1	1 1 0	1 1 1
0 0 0 0	NUL	DLE	SP	*0*	*@*	*P*	*`*	*p*
0 0 0 1	SOH	DC1	*!*	*1*	*A*	*Q*	*a*	*q*
0 0 1 0	STX	DC2	*"*	*2*	*B*	*R*	*b*	*r*
0 0 1 1	ETX	DC3	*#*	*3*	*C*	*S*	*c*	*s*
0 1 0 0	EOT	DC4	*$*	*4*	*D*	*T*	*d*	*t*
0 1 0 1	ENQ	NAK	*%*	*5*	*E*	*U*	*e*	*u*
0 1 1 0	ACK	SYN	*&*	*6*	*F*	*V*	*f*	*v*
0 1 1 1	BEL	ETB	*'*	*7*	*G*	*W*	*g*	*w*
1 0 0 0	BS	CAN	*(*	*8*	*H*	*X*	*h*	*x*
1 0 0 1	SKIP	EM	*)*	*9*	*I*	*Y*	*i*	*y*
1 0 1 0	LF	SUB	*	*:*	*J*	*Z*	*j*	*z*
1 0 1 1	VT	ESC	+	*;*	*K*	*[*	*k*	*{*
1 1 0 0	FF	FS	,	<	*L*	\	*l*	¦
1 1 0 1	CR	GS	-	=	*M*	*]*	*m*	*}*
1 1 1 0	SO	HOME	.	>	*N*	^	*n*	~
1 1 1 1	SI	NL	*/*	*?*	*O*	–	*o*	DEL

Les 7 bits sont notés $b_6b_5b_4b_3b_2b_1b_0$. Par exemple, 1000001 représente **`'A'`**, 1000010 représente **`'B'`**, 110001 représente **`'a'`**... Certains caractères ne sont pas des marques imprimables mais des caractères de contrôle ou de mise en page : CR (*Carriage Return*) positionnement en début de ligne, LF (*Line Feed*) passage à la ligne suivante, BEL (*Bell*) sonnerie...

Ce jeu de caractères est cependant limité à 127 éléments. Il ne permet pas de représenter les caractères accentués ni les caractères spécifiques à certaines langues.

L'unité de stockage est l'octet, et non pas 7 bits. Lorsqu'une représentation ASCII de caractère est rangée dans un octet, le bit de poids fort de l'octet, le bit 7, inutilisé, est mis à 0. La représentation ASCII étendue sur 8 bits permet d'étendre le répertoire à 256 caractères.

Plus récemment est apparue une nouvelle représentation, sur 32 et 16 bits, appelée UNICODE. C'est un standard destiné à remplacer peu à peu l'usage de l'ASCII. Il est deux fois plus coûteux en place mais il permet de représenter 2^{16} caractères, soit plus de 65000, ce qui permet d'inclure non seulement les caractères accentués ou spécifiques à certaines langues européennes, mais également tous les idéogrammes des langues orientales. Pour faciliter le passage de la représentation ASCII à la représentation UNICODE, cette dernière est telle que les caractères du répertoire ASCII sont représentés par les 7 bits de la représentation ASCII en poids faible, complétés par des 0 en poids forts :



Java représente les caractères selon le standard UNICODE.

15.3.2 Représentation des entiers positifs : binaire

La représentation usuelle des nombres entiers positifs (N) est la représentation en base 2, ou binaire. Les nombres sont représentés sur un nombre fixé de bits, ce qui limite l'intervalle représentable.

1. ASCII est le sigle de *American Standard Code for Information Interchange*

Sur *n* bits, on représente en binaire l'intervalle d'entiers $[0, 2^n-1]$.

Les tailles usuelles sont :

- 8 bits : représentation des entiers de 0 à 255,
- 16 bits : représentation des entiers de 0 à 65 535
- 32 bits : représentation des entiers de 0 à 4 294 967 295
- 64 bits : représentation des entiers de 0 à $2^{64}-1$, soit environ 16×10^{18}

15.3.3 Représentation des entiers relatifs : complément à 2

Une méthode directe pour représenter les nombres entiers relatifs (Z) consisterait à utiliser un bit pour indiquer le signe. Par exemple, sur 8 bits, on peut utiliser les 7 bits $b_6...b_0$ pour représenter la valeur absolue du nombre et le bit 7 pour représenter le signe, selon la convention : $b_7=0$ si le nombre est positif, $b_7=1$ si le nombre est négatif. Ainsi :

- 0 000 0101 représente 5
- 1 000 0101 représente -5

Cette représentation, appelée *valeur absolue plus signe*, n'est pratiquement pas utilisée. Bien qu'elle soit facile à comprendre, elle présente quelques inconvénients. Notamment, la représentation de 0 n'est pas unique, ce qui pose quelques difficultés pour les tests d'égalité des nombres. En effet, 0 peut être représenté par 0000 0000 ou par 1000 0000.

On lui préfère la représentation suivante, appelée ***complément à 2***, qui offre de nombreux avantages et de ce fait a été adoptée très tôt sur tous les ordinateurs.

- pour un nombre positif, de 0 à $2^{n-1}-1$, la représentation en complément à 2 est égale à la représentation binaire,
- pour un nombre négatif, de -2^{n-1} à -1, la représentation en complément à 2 de k est la représentation binaire de $2^n-|k|$.

exemple sur 4 bits

nombre	complément à 2	interp. binaire
-8	1000	8
-7	1001	9
-6	1010	10
-5	1011	11
-4	1100	12
-3	1101	13
-2	1110	14
-1	1111	15
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7

En complément à 2, le bit de rang *n-1* (bit de poids fort) est significatif du signe :

- Si $x_{n-1}=0$, le nombre représenté est positif et il vaut, comme en binaire, $x_{n-2}.2^{n-2}+...+ x_1.2 + x_0$
- Si $x_{n-1}=1$, le nombre représenté est négatif et il vaut $-2^{n-1} +x_{n-2}.2^{n-2}+...+ x_1.2 + x_0$

Pour cette raison on a coutume d'appeler ***bit de signe*** le bit de poids fort d'une suite de bits.

Une formule équivalente permettant d'interpréter en complément à 2 une suite de bits est :

interprétation en complément $(x_{n-1}x_{n-2}....x_1.x_0) = \mathbf{-x_{n-1}2^{n-1}} + x_{n-2}.2^{n-2}+...+ x_1.2 + x_0$

cette formule considère simplement le chiffre de poids fort avec un poids négatif.

Les avantages de cette représentation sont l'unicité de la représentation de 0 et surtout que le même opérateur matériel d'addition fonctionne aussi bien pour la représentation binaire simple que pour la représentation en complément à 2. Ceci est quasiment miraculeux : un seul opérateur, c'est-à-dire un seul algorithme, donne le bon résultat ***pour les deux interprétations*** de ces suites de bits.

Exemple sur 4 bits :

opération machine	interprétation binaire	interprétation en complément à 2
1001	9	-7
+ 0011	+3	+3
1100	12	-4

Sans chercher à faire la démonstration de cette propriété remarquable (ce genre de démonstration est assez pénible à suivre, et sans grand intérêt), on peut signaler qu'il s'agit là de propriétés arithmétiques des classes de congruence modulo 2^n.

Dépassements de capacité

En fait, au lieu de dire que le même opérateur "marche aussi bien" pour les deux représentations, on devrait plutôt dire qu'il "ne marche pas plus mal" pour l'une que pour l'autre. En effet, dans chacune des représentations il se pose le problème de dépassement de capacité : la somme peut sortir de l'intervalle de représentation. Bien évidemment dans ce cas, le résultat ne peut pas être correct puisqu'il n'est pas représentable sur *n* bits.

C'est à propos des dépassements de capacité que l'opérateur d'addition diffère. Les conditions de dépassement ne sont pas les mêmes dans les deux représentations.

- Pour la représentation binaire, le dépassement de capacité a lieu lorsque la somme des nombres représentés par les opérandes est supérieure ou égale à 2^n. Cette condition est détectée sur la machine par le fait que l'algorithme d'addition binaire génère un ***report*** à 1 dans l'addition des chiffres de poids fort (*carry* en Anglais).

- Pour la représentation en complément à 2, le dépassement de capacité a lieu lorsque la somme des nombres représentés par les opérandes est soit strictement inférieure à -2^{n-1}, soit supérieure ou égale à 2^{n-1}. Cette condition s'appelle un ***débordement*** (*overflow* en anglais). Elle peut être détectée sur la machine de deux façons :

  Première façon : les opérandes sont de même signe et le résultat de l'opération a un bit de signe différent. De toute évidence le résultat est incorrect dans ce cas. Ce qui est plus difficile c'est de montrer que ce sont là les seuls cas de dépassement.

Deuxième façon : la condition de débordement correspond également au cas où les reports générés par les chiffres de rang *n-1* et de rang *n* sont différents.

Les calculs suivants sur 4 bits montrent :

- à gauche, un cas de résultat correct en interprétation binaire et incorrect en complément à 2,
- à droite, un cas de résultat incorrect en interprétation binaire et correct en complément à 2.

	binaire	complément à 2
0 1 1 0	6	+6
0 0 1 1	3	+3
1 0 0 1	9	-7
report = 0	correct	
débord^{nt}=1		incorrect

	binaire	complément à 2
1 1 1 0	14	-2
1 0 1 1	11	-5
1 0 0 1	9	-7
report = 1	incorrect	
débord^{nt}=0		correct

Voici quelques règles (sans démonstration) concernant les calculs en complément à 2 :

Passage à la représentation de l'opposé :

Connaissant la représentation du nombre *k*, on désire connaître celle de *-k*.

Il suffit de changer tous les 0 en 1 et les 1 en 0, puis effectuer l'algorithme d'addition de 1 au résultat. Par exemple, sur 4 bits, 0100 est la représentation de +4. La représentation de -4 s'obtient en inversant les bits puis en ajoutant 1 :

0100 → 1011
+ 1

1100, représentation de -4

Cette propriété peut se justifier ainsi : si on considère la somme des interprétations en complément à 2 d'une suite de n bits $\mathbf{x_{n-1}...x_1\ x_0}$ et de son complément bit à bit $\mathbf{/x_{n-1}.../x_1\ /x_0}$ on a :

interprétation($x_{n-1}...x_1\ x_0$) + interprétation($/x_{n-1}.../x_1\ /x_0$) = "111...11" = $-2^{n-1} + 2^{n-2} + ... + 2 + 1 = -1$

donc : interprétation($/x_{n-1}.../x_1\ /x_0$) +1 = -interprétation($x_{n-1}...x_1\ x_0$)

Une autre façon de procéder consiste à inverser les bits de poids supérieur au premier bit à 1 rencontré à partir des poids faibles : **0**100 → **1**100

Augmentation du nombre de bits :

Connaissant la représentation du nombre *k* sur *n* bits, on désire connaître sa représentation sur *n+p* bits.

Il suffit de procéder à une extension du bit de signe, c'est-à-dire concaténer en poids forts *p* bits égaux au bit de poids fort de la représentation sur *n* bits. Exemples :

+5 est représenté par **0**101 sur 4 bits, sa représentation sur 8 bits est **0000 0**101,

-5 est représenté par **1**011 sur 4 bits, sa représentation sur 8 bits est **1111 1**011,

Représentations de 0 et de -1 :

Quelque soit le nombre de bits, la représentation de *0* est de la forme 000...0 (que des 0) et celle de *-1* est 111...1 (que des 1).

En Java il y a quatre types d'entiers relatifs :

`byte` : sur 8 bits, représente l'intervalle -128...+127,

`short` : sur 16 bits, représente l'intervalle -32 668...+32 767,

`int` : sur 32 bits, représente l'intervalle -2 147 483 648...+2 147 483 647,

`long` : sur 64 bits, représente l'intervalle

-9 223 372 036 854 775 808 ...+9 223 372 036 854 775 807.

En Java, comme dans de nombreux autres langages de programmation, les opérations **`+`**, **`-`**, **`*`**, **`/`** sont celle de l'***arithmétique modulo 2^n***, *n* étant le nombre de bits de la représentation. Ceci signifie que les dépassements de capacité ne provoquent pas de détection d'erreur pendant l'exécution mais donnent le résultat modulo 2^n de l'opération, compris entre -2^{n-1} et $2^{n-1}-1$.

Par exemple, pour le type **`byte`**, représenté sur 8 bits :

l'opération **`127+1`** donne **`-128`**, et **`127+2`** donne **`-1`**.

15.3.4 Représentation des nombres réels

Quelle que soit la représentation adoptée, un nombre fini de bits ne pourra représenter qu'un nombre fini de valeurs. Pour les entiers cela se traduit par l'imposition de valeurs limites inférieures et supérieures de l'ensemble des valeurs représentées. Pour les réels cela se traduit en plus par une ***précision limitée*** de représentation des valeurs d'un intervalle.

Les représentations de nombres réels les plus utilisées sont les représentations en ***virgule flottante***,

ou plus simplement ***flottante*** (*floating point* ou *float* en anglais).

Principe général

Le principe général d'une représentation flottante consiste à utiliser deux représentations d'entiers :

- la mantisse ***m*** qui apporte la précision,
- l'exposant ***e*** qui indique l'ordre de grandeur du nombre.

Le nombre réel représenté par ces deux entiers est : $\boldsymbol{m} \times b^{\boldsymbol{e}}$

où *b* est une constante, généralement égale à la base choisie pour la représentation de ***m***.

En base 10, comme sur les calculettes, *b*=10. Sur ordinateur, en règle générale *b*=2 car la mantisse est représentée en binaire. Le fait que *b* soit la base de représentation de ***m*** facilite certains traitements, notamment pour représenter le même nombre en changeant d'exposant : il suffit alors de décaler convenablement les chiffres de la mantisse.

Exemple, en base 10 : $3567 \times 10^{-4} = 356700 \times 10^{-6}$

Intérêt des représentations flottantes

Les représentations flottantes permettent une ***précision relative*** à peu près constante sur toute la

gamme de représentation ainsi qu'une gamme étendue.

La précision relative d'un nombre ***x*** représenté avec une incertitude Δx est $\Delta x/x$. De toute évidence, si un résultat de calcul est donné avec une mantisse de ***n*** chiffres significatifs (sans 0 en poids forts), la précision relative du résultat de cette opération est $\Delta x/x < 1/b^n$.

Notion de représentation normalisée

Plusieurs représentations flottantes sont possibles pour un même nombre. Par exemple, en représentation décimale de la mantisse sur 5 chiffres :

$00498 \times 10^{-5} = 49800 \times 10^{-7}$

Ceci présente deux inconvénients : le manque d'unicité de la représentation et, surtout, la perte potentielle de précision si on autorise des 0 non significatifs en poids fort de la mantisse. Pour cela, on s'impose d'utiliser une représentation dite normalisée, dans laquelle le chiffre de poids fort de la mantisse est différent de 0.

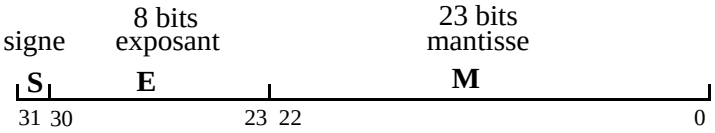
Par exemple, pour représenter 498592 avec seulement 5 chiffres pour la mantisse :

- avec la représentation normalisée $49859 \times 10^{+1}$ l'erreur relative est de l'ordre de 10^{-5}
- avec la représentation non normalisée $00049 \times 10^{+4}$ l'erreur relative est de l'ordre de 10^{-2}

Flottant standard IEEE

Les nombres réels sont d'une grande importance pour les applications de calcul scientifique. Afin d'assurer que les mêmes calculs donnent les mêmes résultats sur toutes les machines, un standard à été défini : il s'agit du ***standard IEEE 754*** qui définit deux représentations, la représentation flottante simple précision et la représentation flottante double précision. Ce standard définit non seulement le format de représentation mais également le comportement des opérations.

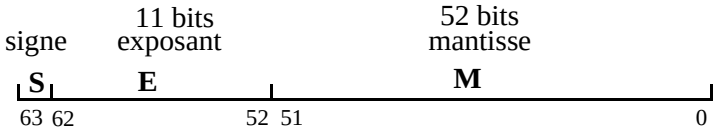
Flottant IEEE simple précision : la représentation occupe 32 bits, un bit pour le signe du nombre, 8 bits pour l'exposant et 23 bits pour la mantisse.



Les 23 bits de mantisse sont l'équivalent d'environ 7 chiffres décimaux, ce qui donne une précision relative d'environ 10^{-7}, ce qui est peu (la moindre calculette est généralement plus précise).

L'exposant, représenté sur 8 bits, peut prendre des valeurs comprises entre -126 et +127, ce qui offre une étendue de $2^{127} \approx 10^{38}$

Flottant IEEE double précision : la représentation occupe 64 bits, un bit pour le signe du nombre, 11 bits pour l'exposant et 52 bits pour la mantisse.



Les 52 bits de mantisse sont l'équivalent d'environ 16 chiffres décimaux, ce qui donne une précision relative d'environ 10^{-16}.

L'exposant, représenté sur 11 bits, peut prendre des valeurs comprises entre -1022 et +1023, ce qui offre une étendue de $2^{1023} \approx 10^{308}$

Interprétation flottante

Voici plus précisément comment fonctionne la représentation, pour le cas du flottant simple précision (c'est le même principe pour le double précision en changeant les nombres de bits).

Pour une représentation normalisée :

l'exposant **E** est interprété en un entier $e \in [-126,+127]$ selon la formule :

e* = *interprétation binaire(*E*)-127

Cette représentation particulière des entiers relatifs s'appelle représentation par excès à 127.

La mantisse **M** est interprétée en un nombre fractionnaire $m \in [1, 2[$ selon la formule :

$$m = 1 + \text{interprétation binaire(M)} / 2^{23}$$

c'est-à-dire en un nombre qui en écriture binaire fractionnaire s'écrirait : $\mathbf{1,M_{22}M_{21}...M_1M_0}$

Le 1, chiffre de poids fort de la mantisse, est implicite, il ne figure pas dans la mantisse. Il y a là une astuce qui ne marche qu'en binaire : puisque la mantisse est normalisée, son chiffre de poids fort est différent de 0, il vaut donc 1 et ce n'est pas la peine de le noter (économie d'un bit).

Enfin, l'interprétation flottante des 32 bits formant le triplet **<S,E,M> est :**

$$\textit{interprétation flottante } \langle \mathbf{S},\mathbf{E},\mathbf{M}\rangle = (-1)^{\mathbf{S}} \times \mathbf{m} \times 2^{\mathbf{e}}$$

Le standard IEEE prévoit également :

- La représentation de nombres non normalisés, pour représenter de tous petits nombres mais avec une précision relative dégradée.
- La représentation de 0 : le réel 0 est représenté par 0000.....00.
- La représentation de valeurs spéciales : les infinis signés, les cas d'erreurs (appelés NaNs abréviation du terme anglais "*Not a Number*", c'est-à-dire un "non-nombre").

Toutes ces représentations spéciales sont distinguées des représentations normalisées car elles ont un exposant **E** de la forme 00000000 ou 1111111, formes non utilisées pour les représentations normalisées.

Voici quelques exemples :

nombre	représentation flottant IEEE simple précision
$1 = +1.0\times2^{0} = +1.0 \times2^{127-127}$	`0 01111111 0000...0`
$0.5 = +1.0\times2^{-1} = +1.0 \times2^{126-127}$	`0 01111110 0000...0`
$4 = +1.0\times2^{2} = +1.0 \times2^{129-127}$	`0 10000001 0000...0`
$0.75 = +1.5\times2^{-1} = +1.5 \times2^{126-127}$	`0 01111110 1000...0`
$4.5 = +1.125\times2^{2} = +1.125 \times2^{129-127}$	`0 10000001 0010...0`
$2.75 = +1.375\times2 = +1.375 \times2^{128-127}$	`0 10000000 0110...0`

Java offre les deux sortes de représentations de réel :

- **`float`** : flottant IEEE simple précision,
- **`double`** : flottant IEEE double précision,

15.4 Représentation des types composites : structures, objets, tableaux

Les types que nous avons vus précédemment sont des types simples, encore appelés ***types scalaires***. En règle générale, les valeurs de types scalaires sont manipulables directement et dans leur totalité par les processeurs : les processeurs disposent d'instructions pour réaliser les opérations et des registres pour contenir les opérandes et résultats de calcul.

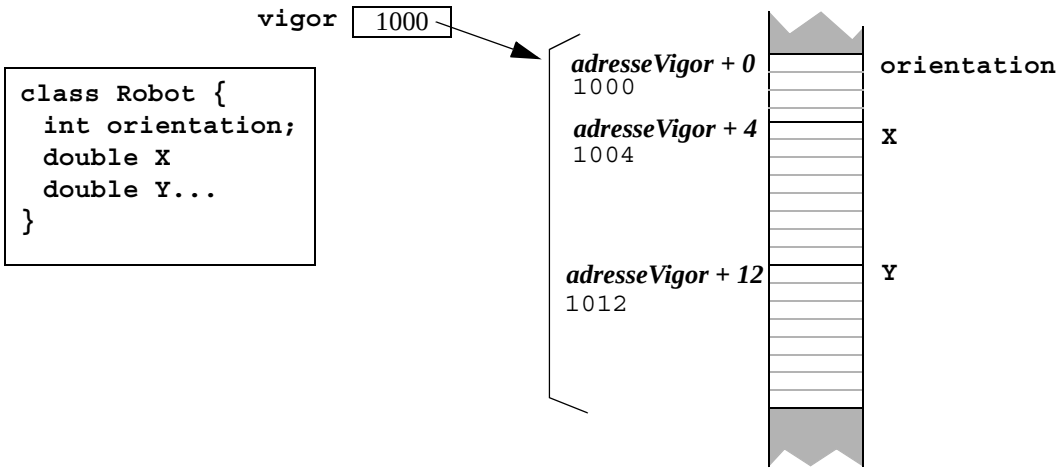
Pour les types composites, qui sont des assemblages de plusieurs données, les processeurs usuels ne savent pas les traiter directement dans leur totalité. La représentation des types composites utilise la mémoire de l'ordinateur, son accès par adresses et les possibilités de ***calcul d'adresses*** du processeur.

15.4.1 Représentation des structures et des objets de type classe

Les données d'un objet de type classe sont rangées en mémoire à des adresses consécutives. La figure suivante illustre la représentation en mémoire d'un objet de type **`Robot`**, appelé **`vigor`**, rangé en mémoire à partir de l'adresse ***adresseVigor***. Ses données sont constituées :

- d'un **`int orientation`**, 32 bits ou 4 octets, situé à l'adresse ***adresseVigor+0***,
- d'un **`double X`**, 64 bits ou 8 octets, situé à l'adresse ***adresseVigor+4***,
- d'un **`double Y`**, 64 bits ou 8 octets, situé à l'adresse ***adresseVigor+12.***

Si, par exemple, ***adresseVigor*** vaut 1000, on a le schéma suivant :

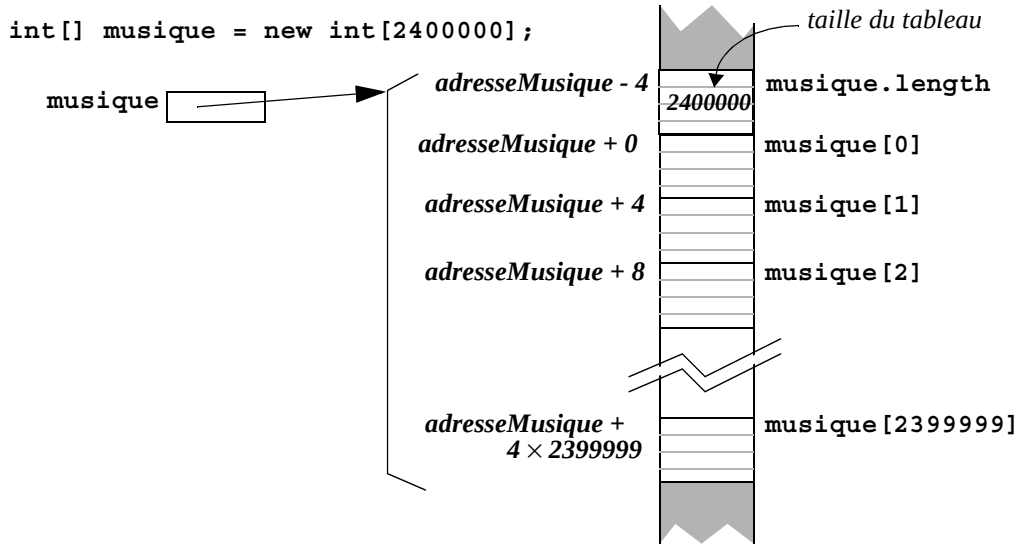


Comme le montre cet exemple, à chaque composant de l'objet correspond un ***déplacement***, en nombre d'octets, à partir de l'adresse de début de l'objet. L'adresse de l'objet est la ***référence*** à l'objet : c'est cette adresse qui constitue la valeur des variables, des paramètres ou des résultats de ce type. C'est également la valeur du paramètre implicite **`this`** lors d'un appel de méthode d'un objet de ce type.

Le compilateur traduit chaque composant non statique d'une classe en un déplacement : déplacement 0 pour **`orientation`**, déplacement 4 pour **`X`** et déplacement 12 pour **`Y`**. Pendant l'exécution d'une méthode d'un objet de ce type, pour atteindre un composant de l'objet le processeur devra ajouter le déplacement correspondant à l'adresse de l'objet.

15.4.2 Représentation des tableaux

La représentation des tableaux utilise avec profit le fait que la mémoire d'un ordinateur se comporte comme un tableau d'octets indicé par les adresses. Un tableau est représenté par une suite d'emplacements consécutifs à partir d'une adresse de début :



L'exemple illustre un tableau appelé **`musique`** de 2400000 éléments de type **`int`**. Chaque élément occupe 32 bits, soit 4 octets. Le début du tableau, qui correspond au premier élément **`musique[0]`**, est placé à l'adresse ***adresseMusique***. Chaque élément **`musique[`*`i`*`]`** est placée à l'adresse ***adresseMusique+4×i***, le facteur 4 étant la taille de représentation du type **`int`** en octets.

Dans un langage comme Java, la taille d'un tableau est indiquée au moment de sa création. La taille doit être notée en association avec le tableau, en un endroit que l'exécuteur du programme puisse facilement retrouver. La taille peut par exemple être placée en mémoire juste avant le tableau, à l'adresse ***début du tableau- 4*** si la taille est un entier sur 32 bits.

15.5 Opérations sur les réels - erreurs

15.5.1 Addition - soustraction

Ces opérations sont réalisées selon la méthode suivante :

- alignement des exposants sur le plus élevé,
- addition des mantisses,
- normalisation du résultat et arrondi.

Pour rendre les exemples compréhensibles, nous les donnons en décimal. Le principe est bien évidemment le même en binaire, par exemple avec la représentation IEEE. Nous prenons 4 chiffres décimaux pour la mantisse et 2 chiffres décimaux pour l'exposant.

$4.832\ 10^3 + 2.104\ 10^0$

$$\begin{array}{rll} & 4.832 & 10^3 \\ + & \underline{0.002104} & \underline{10^3} \\ & \mathbf{4.834}104 & 10^3 \end{array}$$

l'arrondi ne conserve que 4 chiffres pour la mantisse : 4.834

$4.832\ 10^1 - 4.827\ 10^1$

$$\begin{array}{rll} & 4.832 & 10^1 \\ - & \underline{4.827} & \underline{10^1} \\ & \mathbf{0.005} & 10^1 \end{array}$$

normalisé en **5.000** 10^{-2}

15.5.2 Multiplication - Division

Ces opérations sont réalisées selon la méthode suivante :

- addition (soustraction) des exposants,
- multiplication (division) des mantisses,
- troncature/normalisation.

Exemples

$$\begin{array}{rll} & 1.564\ 10^0 & \\ \times & \underline{1.589\ 10^0} & \\ & \mathbf{2.485}196\ 10^0 & \text{arrondi en } 2.485 \end{array} \qquad \begin{array}{rll} & 2.0\ 10^1 & \\ \div & \underline{3.0\ 10^0} & \\ & \mathbf{6.666}6666\ 10^0 & \text{arrondi en } 6.666 \end{array}$$

15.5.3 Erreurs dans les calculs sur les réels

Les erreurs lors des calculs sur des nombres réels constituent un des problèmes les plus importants et les plus difficiles du calcul scientifique. Chaque opération peut introduire des erreurs d'arrondis à cause du nombre de chiffres limités de la mantisse. La détermination du degré de confiance que l'on peut attribuer aux résultats est généralement bien plus compliqué que les algorithmes qui calculent ces résultats.

15.5.3.1 Erreurs lors des multiplications et des divisions

L'erreur introduite lors d'une multiplication ou d'une division est assez facile à maîtriser :

- L'erreur ***relative*** due à l'opération elle-même est majorée par la valeur représentée par l'unité du chiffre de poids faible de la mantisse ($1/b^k$ pour k chiffres en base b).
- En ce qui concerne la propagation des erreurs, c'est-à-dire l'erreur sur le résultat dû aux incertitudes sur les opérandes, ***les erreurs relatives sur les opérandes s'ajoutent*** :

  $x.y = (x + \alpha x)\ (y + \beta y) = x.y + x.y\ (\alpha + \beta + \alpha.\beta)$ soit, en négligeant $\alpha.\beta$: $= x.y + (1 + \alpha+\beta)$

15.5.4 Erreurs lors des additions et des soustractions

Les erreurs liées à ces deux opérations sont plus difficiles à formaliser et peuvent avoir des conséquences spectaculaires. Ces erreurs sont dues à deux phénomènes : ***l'absorption*** et ***l'élimination.***

Erreurs d'élimination

Ce phénomène se produit lorsqu'on soustrait des nombres très voisins. Les chiffres de poids forts s'éliminent et il faut alors normaliser le résultat en rajoutant des zéros en poids faible. Or, les opérandes sont en général des résultats arrondis d'opérations antérieures et l'utilisation des opérandes exacts feraient apparaître des chiffres à la place de ces zéros. La valeur calculée peut alors être très différente de la valeur exacte.

L'erreur relative est très souvent de l'ordre de 100%, ce qui revient à dire que le résultat est le fruit du hasard.

Exemple d'élimination :

$$X = 1.405\ 10^3$$
$$- \quad Y = \underline{1.404\ 10^3}$$
$$0.001\ 10^3$$

Ce qui donne après normalisation : $1.000\ 10^0$

Si X et Y résultent de calcul et sont les arrondis de :

X = 1.405456 et Y= 1,404012, le résultat exact est 1.444.

L'erreur relative est ici de 40%.

Erreurs d'absorption

Les erreurs d'absorption se produisent lors de l'addition ou de la soustraction de deux nombres de grandeurs très différentes. Ceci est dû à la perte de chiffres lors de l'***alignement*** des exposants. On peut même perdre tous les chiffres significatifs du plus petit des nombres de sorte que le résultat est simplement le plus grand des deux.

Exemple d'absorption :

$1.232 \times 10^4 + 2.104 \times 10^1$

$$1{,}232 \quad 10^4$$
$$+ \quad \underline{0.002404} \quad \underline{10^4}$$
$$1.234404 \quad 10^4$$

Ce qui après arrondi donne $1.234\ 10^4$

Autre exemple d'absorption (catastrophique) : soit à calculer $1.6 + 10^9 - 10^9$

Si on effectue le calcul selon l'ordre induit par le parenthèsage suivant : $1.6 + (10^9 - 10^9)$

on obtient : $1.6 + 0 = 1.6$, ce qui est normal.

Mais si on l'effectue selon l'ordre correspondant à : $(1.6 + 10^9) - 10^9$

on obtient : $10^9 - 10^9 = 0$, car par arrondi $1.6 + 10^9 = 10^9$

Cet exemple montre que l'addition avec les flottants est une opération ***non associative* !**

Exemple d'algorithme sensible à l'absorption : calcul de la somme de la série harmonique

$$S = \sum_{i=1}^{n} \frac{1}{i}$$

Sommation par ordre décroissant des termes :

```
double S=0;
for (int i=1; i<=n; i++) {S=S+1/i;}
```

On commence par les grands termes. Les plus petits termes sont absorbés par la somme partielle déjà calculée.

Sommation par ordre croissant des termes :

```
double S=0;
for (int i=n; i>=1; i--) {S=S+1/i;}
```

Cette fois les plus petits termes ne sont pas absorbés, le résultat est meilleur.

n	**10 000**	**100 000**	**10 000 000**	**10 000 000**
par ordre décroissant des termes	9.787613	14.35736	15.40368	15.40368
par ordre croissant des termes	9.787604	14.39265	16.68603	18.80792
valeur exacte	9.787606	14.39273	16.69531	18.99789

Exemple : Résolution d'une équation du second degré

Considérons l'équation $x^2 - 205\ x + 10500 = 0$

Les deux racines réelles sont 100 et 105.

Avec une représentation dont les mantisses ont 3 chiffres, les résultats seront faux :

$\Delta = (2.05\ 10^2)^2 - 4 \times 1.05\ 10^4$

$= 4.20\ 10^4 - 4.20\ 10^4$

$= 0 \qquad \Rightarrow x_1 = x_2 = 100$

En revanche, avec 5 chiffres de mantisse, les résultats sont corrects :

$\Delta = (2.05\ 10^2)^2 - 4 \times 1.05\ 10^4$

$= 4.2025\ 10^4 - 4.2000\ 10^4$

$= 0.0025\ 10^4$

$= 25 \qquad \Rightarrow x_1 = (205-5) / 2 = 100 \quad x_2 = (205+5) / 2 = 105$

ANNEXE 1 Installations

1.1 Installation de Java

On peut télécharger Java depuis le site de Sun Micro-System :

http://java.sun.com/javase/downloads/index.jsp

le lien ci-dessus contient les versions de 2007. Bien évidemment il convient de prendre la plus récente version du moment. Le site propose une installation pour la plupart des systèmes : Windows, Linux, Mac-OS, Sun-OS. Le fichier téléchargé est un programme d'installation qu'il suffit d'exécuter.

1.2 Test de l'installation de java

Pour tester si java est correctement installé, on peut essayer le programme simple suivant :

```
class Test1{
  public static void main(String [] arg){
      System.out.println("un test");
  }
}
```

Pour cela introduire ce programme dans un fichier texte **Test1.java**, au moyen d'un éditeur de texte brut (par exemple bloc-notes ou Xemacs).

Pour compiler le programme, il faut pouvoir appeler le compilateur. Dans ce premier test, nous le ferons par une commande tapée dans une fenêtre de commande. On peut solliciter le compilateur par son chemin absolu. Par exemple, pour une installation Windows :

>"c:\Program Files\jdk1.5.0_06\bin\javac" Test1.java

Pour disposer plus simplement des commandes javac et java, il est préférable de rajouter le chemin

c:\Program Files\jdk1.5.0_06\bin à la variable d'environnement **path**. On peut alors lancer la compilation par :

> javac Test1.java

Si le compilateur ne trouve pas d'erreur, il doit produire le fichier **Test1.class** exécutable par l'interpréteur java. Pour l'exécuter il faut taper la commande :

> java Test1

L'exécution doit produire l'affichage :

> un test

1.3 Ajout de paquetages

Pour faire certains exercices, nous utiliserons des classes regroupées dans des “paquetages” (package). Ces paquetages se trouvent dans le répertoire **`paquetagesMaison`**. Voici ce qu’offrent ces paquetages :

`es` : lecture de données depuis le clavier et depuis des fichier, écriture de données sur des fichiers,

`list` : listes de données de types quelconque,

`ens` : ensembles de valeurs de divers types,

`ihm` : placement de composants graphiques.

Chaque paquetage est un répertoire qui contient des classes destinées à être utilisées dans des applications. Pour utiliser un tel paquetage, il faut en début du texte du programme, placer une directive :

`import` ***nomDuPaquetage*****`.*;`**

Le programme suivant utilise la procédure **`Lecture.unEntier`** du paquetage **`es`** pour lire des données frappées au clavier.

```
import es.*;

class Test2 {
  public static void main(String [] arg){
      System.out.print("entrer deux nombres :");
      int nombre1=Lecture.unEntier();
      int nombre2=Lecture.unEntier();
      System.out.print("leur somme = ");
      System.out.println(nombre1+nombre2);
  }
}
```

Ce programme lit deux nombres entiers, frappés au clavier, en décimal, séparés par un espace ou par des passages à la ligne, puis il affiche leur somme.

Si on essaye de le compiler comme précédemment, le compilateur indique une erreur car il ne connaît pas la classe **`Lecture`**. Il faut en plus lui indiquer le répertoire où se trouve le paquetage **`es`**. On peut le lui indiquer :

- Soit grâce à la variable d’environnement **`CLASSPATH`** : la positionner en lui indiquant le chemin du répertoire **`paquetagesMaison`** ainsi que le répertoire courant, noté “**`.`**”. Sur un système Windows, la commande est

  `set CLASSPATH=.../paquetagesMaison;.`

  `.../paquetagesMaison` dénote ici le chemin absolu du répertoire **`paquetageMaison`** (cela dépend de l’endroit où le dossier a été placé).
- Soit en lançant la commande de compilation avec l’option **`-classpath`** :

  `javac -classpath .../paquetagesMaison;. Test2.java`

  Dans ce cas, la commande d’exécution doit également se faire avec cette option :

  `java -classpath .../paquetagesMaison;. Test2`

Le programme Test2 attend l’introduction de deux nombres entiers au clavier, terminé par un passage à la ligne, et affiche leur somme :

```
entrer deux nombres : 45 25
leur somme = 60
```

1.4 Installation d'Eclipse

Eclipse est un outil de développement sophistiqué pour de nombreux langages (langages de programmation, principalement pour Java, mais aussi C++, ou bien langages documentaires tels HTML, XML...). On peut télécharger Eclipse depuis le site :

`http://download.eclipse.org/eclipse/downloads`

Il convient de charger la plus récente version. En 2007, pour du développement Java, sur Windows, le fichier à télécharger est :

`eclipse-SDK-3.3.1.win32.zip`

Il existe des installation pour quasiment tous les autres systèmes : Linux, Unix-Solaris, MacOS.

Il s'agit simplement d'un dossier compressé appelé **`eclipse`** qu'il suffit de déployer, par exemple, sous windows, dans **`C:/Programm File`**. Sous Windows, on le lance en exécutant **`eclipse.exe`** qui est dans le dossier eclipse.

1.5 Ouverture d'un projet Eclipse

Avec Eclipse, les applications Java sont organisées en "projets". Pour de simples exercices, on peut les réaliser dans un même projet.

Commencer par créer le répertoire du projet, par exemple **`c:\ProjetCoucou`** (on peut éventuellement y placer des programmes sources java que l'on posséderait déjà. Ils seront pris en compte dans le projet).

Par les menus d'Eclipse :

- **`File - New - Project`**
- choisir **`Java Project`**, ***next***
- donner le nom du projet dans **`Project name`** : **`ProjetCoucou`**
- sélectionner **`Create project at external location`**
- indiquer le répertoire du projet (***browse***) : **`c:\ProjetCoucou`**
- ***next***
- Si on veut utiliser des paquetages autres que ceux de la bibliothèque standard, par exemple les paquetages "maison", onglet **`Librairies`** :

  ajouter par **`Add External JARs`** (***browse***) :

  `...\paquetagesMaison.jar`

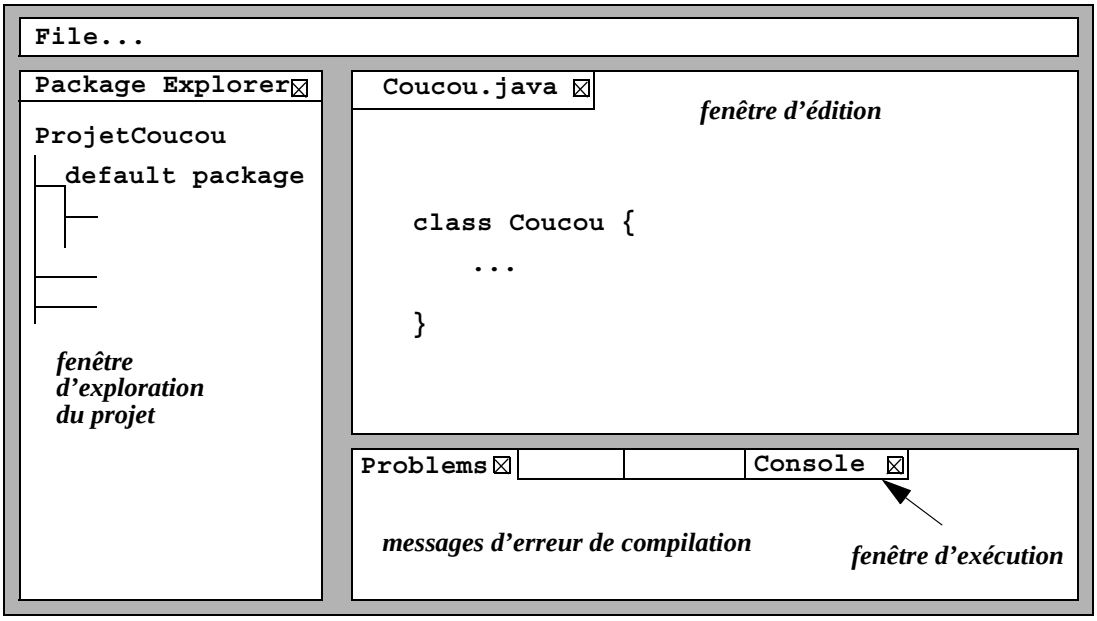
  ouvrir, ***finish***.

A la question "voulez vous passer tout de suite à la perspective Java ?" répondre ***yes***

La perspective Java utilise essentiellement 3 fenêtres :

- La fenêtre d'exploration du projet (à gauche). Les classes du projet sont visibles dans la rubrique **`default package`**.
- La fenêtre d'édition des programmes source Java (au milieu en haut).
- La fenêtre (au milieu en bas) qui sert aux messages d'erreur de compilation (onglet **`Problems`**) et en tant que console d'exécution (onglet **`Console`**).

Les fenêtres “inutiles” s’enlèvent en cliquant sur les croix qui leur correspondent. Les fenêtres se réaffichent par **`Window - Show View`**.



Suppression d’un projet :

sélectionner le projet dans la fenêtre d’exploration et : **`File delete`**

(***attention*** : laisser l’option par défaut ***“ne pas détruire le contenu”*** de façon à simplement retirer le projet d’Eclipse sans détruire les fichiers).

En cas de panique :

Pour repartir dans un état vierge, supprimer les fichiers **`.project`** et **`.classpath`** du répertoire du projet.

1.6 Création d’une classe

Menu : **`File - New - Class`**

`Name` : **`Coucou`**

Pour une classe “exécutable” (doté d’une procédure **`main`**) choisir

`public static void main(String[] arg)`

finish

Compilation :

la compilation de tout ce qui doit être compilé se fait à chaque sauvegarde (menu **`File Save`** ou icône disquette dans le bandeau).

1.7 Exécution

Sélectionner la fenêtre d'édition du programme à exécuter (la classe qui contient le **`main`**) puis :

`Run - Run as - Java application`

Pour communiquer avec le programme ou voir les messages d'erreur d'exécution, sélectionner l'onglet **`Console`** sur la fenêtre du bas.

Arrêt forcé de l'exécution :

(par exemple pour arrêter un programme qui boucle indéfiniment)

en haut de la fenêtre **`Console`**, cliquer sur le carré rouge.

Pour en savoir plus...

Lire le **"petit manuel à l'usage du développeur Java sous Eclipse"** de Yves Bekkers - Ifsic

ANNEXE 2 Paquetages pour les exercices

Nous donnons ici les programmes sources des paquetages “maison” utilisés pour les exercices et les travaux pratiques. Ce sont :

Paquetage **`es`** :

classe **`Lecture`** : entrées depuis le clavier.

classe **`LectureFichierTexte`** : entrées depuis un fichier texte.

classe **`EcritureFichierTexte`** : sorties sur un fichier texte.

Paquetage **`list`** :

classe **`Liste<T>`** : listes génériques

La classe **`ListeDEntiers`** en est une spécialisation :

```
class ListeDEntiers extends Liste<Integer>
```

Paquetage **`ens`** :

classe **`Ensemble<T>`** : ensembles génériques

La classe **`EnsembleDEntiers`** en est une spécialisation :

```
class EnsembleDEntiers extends Ensemble<Integer>
```

2.1 Entrées-sorties clavier et fichiers textes

```
package es;
import java.io.*;

public class Lecture { // lectures clavier
  public static char unCar() {
  // effet : lit un caractère frappé
  // résultat : le caractère frappé
    char c;
    try {c=(char) System.in.read();}
    catch(IOException e) {c=(char) 0;};
    return c;
  }
```

```
  public static String chaine(String delimiteurs) {
  // prérequis : delimiteurs.length()!=0
  // effet : lit une séquence de caractères constituée
  // de délimiteurs puis de caractères autres que des délimiteurs.
  // les délimiteurs sont les caractères de délimiteurs.
  // résultat : la chaine formée des caractères compris entre
  // les délimiteurs
    StringBuffer b = new  StringBuffer();
    char c=unCar();
    // ignore les délimiteurs de tête
    while (delimiteurs.indexOf(c)!=-1) {c=unCar();};
    // lit jusqu'au prochain délimiteur
    while (delimiteurs.indexOf(c)==-1) {b.append(c); c=unCar();};
    return b.toString();
  }

  public static String chaine() {
  // résultat : lecture d'une chaîne avec pour délimiteurs
  // l'espace et le passage à la ligne
    return  chaine(" \r\n");
  }

  public static int unEntier() {
  // effet : lit une chaîne de caractères avec pour délimiteurs
  // l'espace et le passage à la ligne
  // résultat : l'entier representé en decimal par cette chaîne
  // si la chaîne ne respecte pas la syntaxe d'un entier décimal,
  // affiche un message d'erreur et retourne 0.
    String s=Lecture.chaine(" \r\n");
    try { return Integer.parseInt(s);}
    catch(NumberFormatException e) {
      System.err.println("\nErreur lecture d'entier");
      System.err.println("valeur 0 retournée");
      return 0;
    }
  }

  public static double unReel() {
  // effet : lit une chaîne de caractères avec pour délimiteurs
  // l'espace et le passage à la ligne
  // résultat : le flottant representé par cette chaîne
  // si la chaîne ne respecte pas la syntaxe pour un nombre réel,
  // affiche un message d'erreur et retourne 0.
    String s=Lecture.chaine(" \r\n");
    try { return  (Double.valueOf(s)).doubleValue();}
    catch(NumberFormatException e) {
      System.err.println("\nErreur lecture de reél");
      System.err.println("valeur 0 retournée");
      return 0;
    }
  }
}
```

```
package es;
import java.io.*;

public class LectureFichierTexte {


  private BufferedReader leFichier;
  private char prochainCaractere;
  private boolean finDeFichier;
  private String nom;

  private void tenteDeLireProchainCaractere() {
    try {
      int x = leFichier.read(); prochainCaractere = (char) x;
      if (x==-1) {finDeFichier = true;}
    }
    catch(IOException e){
      System.err.println("erreur de lecture du fichier "+nom);
      Thread.dumpStack();
    }
  }

  public LectureFichierTexte(String nom) {
  // initialise un accès en lecture sur le fichier de nom nom
  // erreur si le fichier n'existe pas
    this.nom=nom;
    try {
      leFichier = new BufferedReader(new FileReader(nom));
      finDeFichier = false; tenteDeLireProchainCaractere();
    }
    catch(FileNotFoundException e){
      System.err.println("fichier "+nom+" inexistant");
      Thread.dumpStack();
    }
  }


  public void fermer() { // fermeture du fichier
    try {leFichier.close();}
    catch(IOException e) {
      System.err.println(
          "erreur lors de la fermeture du fichier "+nom);
      Thread.dumpStack();
   }
  }


  public char lireUnCar() { // lecture d'un caractère
    char courant = prochainCaractere;
    tenteDeLireProchainCaractere();
    return courant;
  }
```

```
  public String lireChaine(String delimiteurs) {
  // lecture d'une chaine comprise entre délimiteurs ou jusqu'à la
  // fin du fichier. Rend la chaine vide si fin de fichier.
    if (finDeFichier()) {return "";}
    char c=lireUnCar();
    // ignore les delimiteurs de tête
    while (!finDeFichier() && delimiteurs.indexOf(c)!=-1) {
      c=lireUnCar();
    }
    if (finDeFichier()) {return "";}
    // lit jusqu'au prochain délimiteur ou fin de fichier
    StringBuffer b = new StringBuffer(); b.append(c);
    c=lireUnCar();
    while (!finDeFichier() && delimiteurs.indexOf(c)==-1) {
       b.append(c); c=lireUnCar();
    }
    if(delimiteurs.indexOf(c)==-1){b.append(c);}
    // consomme les éventuels délimiteurs suivants
    while (!finDeFichier()
            && delimiteurs.indexOf(prochainCaractere)!=-1) {
      c=lireUnCar();
    }
    return b.toString();
  }

  public String lireChaine() {
  // lecture d'une chaine comprise entre delimiteurs standards
    return lireChaine(" \r\n");
  }

  public int lireUnEntier() { // lecture d'un entier
    try { return Integer.parseInt(lireChaine());
    }
    catch(NumberFormatException e) {
      System.err.println(
              "Erreur lecture d'entier sur fichier "+nom);
      System.err.println("valeur 0 retournée");
      return 0;
    }
  }

  public double lireUnReel() { // lecture d'un nombre réel
    try { return  (Double.valueOf(lireChaine())).floatValue();}
    catch(NumberFormatException e) {
      System.err.println(
         "Erreur lecture de reél sur fichier "+nom);
      System.err.println("valeur 0 retournée"); return 0;
    }
  }

  public boolean finDeFichier() { // indique si fin de fichier
    return finDeFichier;
  }
}
```

```
package es;
import java.io.*;

public class EcritureFichierTexte {

   private PrintWriter leFichier;
   private String nom;

  public EcritureFichierTexte(String nom) {
    this.nom=nom;
    try {leFichier = new PrintWriter(new FileOutputStream(nom));}
    catch(IOException e){
      System.out.println(
           "erreur lors de la création du fichier "+nom);
      Thread.dumpStack();
    }
  }

  public void ecrire(char c) {leFichier.print(c);}

  public void ecrire(String s) {leFichier.print(s);}

  public void ecrire(int k) {leFichier.print(k);}

  public void ecrire(boolean b) {leFichier.print(b);}

  public void ecrire(double x) {leFichier.print(x);}

  public void fermer() {leFichier.close();}
}
```

2.2 Listes

La classe **`Liste`** présentée ici est une classe ***générique***. Le type des éléments (désigné par **`T`**) peut être n'importe quelle classe.

```
package list;
import parcours.*;

public class Liste<T> {

  private static class Maillon<TE> {
    Maillon<TE> suivant; Maillon<TE> precedent; TE element;
    Maillon(Maillon<TE> s,Maillon<TE> p,TE e) {
            suivant=s;precedent=p;element=e;
    }
  }

  private Maillon<T> tete;
  private Maillon<T> queue;

  public Liste() { // liste vide
    tete=null; queue=null;
  }
```

```
  public Liste(T[] e){ // liste contenant les éléments de e
    tete=null; queue=null;
    for (int i=0;i<e.length;i++){ajouteEnQueue(e[i]);}
  }

  public boolean estVide() {
  // résulat : indique si this est vide
    return tete==null;
  }


public String toString() {
  // résultat : this en clair
    if (estVide()) {return "<>";}
    Parcours p= new Parcours();
    StringBuffer resul =
      new StringBuffer("< "+ p.elementCourant().toString());
    p.suivant();
    while (!p.estEnFin()) {
      resul.append(" | " + p.elementCourant().toString());
      p.suivant();
    }
    resul.append(" >"); return resul.toString();
  }


  public void ajouteEnQueue(T nouvelElement) {
  // effet : ajoute nouvelElement en queue de this
    if (queue==null) { // cas liste vide
      queue = new Maillon<T>(null,null,nouvelElement);
      tete = queue;
    }
    else { // chaine en queue
      Maillon<T> exDernier = queue;
      queue = new Maillon<T>(null,exDernier,nouvelElement);
      exDernier.suivant=queue;
    }
  }

  public void ajouteEnTete(T nouvelElement) {
  // effet : ajoute nouvelElement en tête de this
    if (tete==null) { // cas liste vide
      tete = new Maillon<T>(null,null,nouvelElement);
      queue = tete;
    }
    else { // chaine en tete
      Maillon<T> exPremier = tete;
      tete = new Maillon<T>(exPremier,null,nouvelElement);
      exPremier.precedent=tete;
    }
  }
```

```
public T retireEnQueue() {
// prérequis : this n'est pas vide
// effet : retire l'élément de queue
// résultat : l'élément retiré
    T resul = queue.element;
    queue = queue.precedent;
    if (queue!=null) {queue.suivant=null;}
    else {tete = null;}
    return resul;
}


public T retireEnTete() {
// prérequis : this n'est pas vide
// effet : retire l'élément de tête
// résultat : l'élément retiré
    T resul = tete.element;
    tete = tete.suivant;
    if (tete!=null) {tete.precedent=null;}
    else {queue = null;}
    return resul;
}



public static <TE>  Liste<TE> aPartirDe(TE[] e){
// résultat : une nouvelle liste contenant les éléments de e
  Liste<TE> resul=new Liste<TE>();
  for (int i=0;i<e.length;i++){
    resul.ajouteEnQueue(e[i]);
  }
  return resul;
}



public Liste<T>.Parcours nouveauParcours() {
// résultat : un nouveau parcours initialisé au début de this
  return new Parcours();
}

public Liste<T>.Parcours nouveauParcours (Liste<T>.Parcours p) {
// résultat : nouveau parcours initialisé à l'état de p
  return new Parcours(p);
}
```

```
//=================== parcoureur de liste ====================
public class Parcours implements ParcoursBidirectionnel<T>,
                                  ParcoursModificateur<T> {

  private Maillon<T> courant;

  public Parcours() {courant=tete;}
  // Parcours initialisé en tête

  private Liste<T> laListe() { return Liste.this; }

  public Parcours(Parcours p) {
  // Parcours initialisé avec l'état du Parcours p
    if (p.laListe()!=Liste.this) {
      System.out.println("erreur :\n"
        + "parcours initialisé avec celui d'une autre liste");
      Thread.dumpStack(); System.exit(0);
    }
    courant=p.courant;
  }

  public void tete() {
  // effet : positionne this en tete de liste
  // (parcours en fin si liste vide)
    courant=tete;
  }

  public void queue() {
  // effet : positionne this en queue de liste
  // (parcours en fin si liste vide)
    courant=queue;
  }

  public void suivant() {
  // effet : positionne this sur l'élément suivant,
  // ne fait rien si estEnFin()
    if (courant!=null) {courant=courant.suivant;}
  }

  public void precedent() {
  // effet : positionne this sur l'élément précédent,
  // ne fait rien si estEnFin()
    if (courant!=null) {courant=courant.precedent;}
  }

  public boolean estEnFin() {
  // résultat : indique si this est en fin
  // (au dela de la queue, en deça de la tete)
    return courant==null;
  }
```

```
    public T elementCourant() {
    // prérequis : this n'est pas en fin
    // résultat : l'élément courant de this
      return courant.element;
    }

    public void modifElement(T nouvelElement) {
    // prérequis : this n'est pas en fin
    // effet : remplace l'élément courant de this par nouvelElement
      if (courant!=null) {courant.element=nouvelElement;}
    }

    public void ajouteElement(T nouvelElement) {
    // effet : insère nouvelElement à la suite de l'élément courant
    // insère en tête si this est en fin de parcours
    // l'élément inséré devient l'élément courant
      Maillon<T> suivant; Maillon<T> nouveau;
      if (courant==null) { // chaine en tête
        suivant=tete;
        nouveau = new Maillon<T>(tete,null,nouvelElement);
        tete=nouveau; courant=nouveau;
      }
      else { // chaine sur courant
        suivant=courant.suivant;
        nouveau = new Maillon<T>(suivant,courant,nouvelElement);
        courant.suivant=nouveau; courant=nouveau;
      }
      if (suivant!=null) {suivant.precedent=courant;}
      else { // nouvel element de queue
        queue=courant;
      }
    }


    public void retireElement() {
    // prérequis : this n'est pas en fin
    // effet : retire l'élément courant de this
    // le suivant de l'élément retiré, s'il existe, devient élément
    // courant, sinon this passe en fin de parcours
      Maillon<T> suivant=courant.suivant;
      Maillon<T> precedent=courant.precedent;
      if (precedent!=null) {precedent.suivant=suivant;}
      else {tete=suivant;}
      if (suivant!=null) {suivant.precedent=precedent;}
      else {queue=precedent;}
      courant=suivant;
    }
  }
  //===============================================================
}
```

2.3 Ensembles

La classe **`Ensemble`** présentée ici est une classe ***générique***. Le type des éléments (désigné par **`T`**) peut être n'importe quelle classe doté d'une fonction de comparaison appelée **`compareTo`** qui induise une relation d'ordre.

```
package ens;
import parcours.*;

public class Ensemble<T extends Comparable<T>> {

  private Object[] elements; // tableau d'éléments de type T
                             // de taille ajustée aux besoins
  private int nbElements;
  private static final int facteurDeCroissance=2;
  private static final int taillePourSingleton=1;


  public Ensemble() { // ensemble vide
    elements=null; nbElements=0;
  }

  public static <T extends Comparable<T>>
                               Ensemble<T> aPartirDe(T[] elts){
  // résultat : un nouvel ensemble d'entiers composé
  // des éléments de elts
    Ensemble<T> resul = new Ensemble();
    for (int i=0; i<elts.length; i++){
      resul.ajouteElement(elts[i]);
    }
    return resul;
  }

  public int cardinal() {
  // résultat : le nombre d'éléments de this
    return nbElements;
  }

  public boolean estVide() {
  // résultat : indique si this est vide
    return nbElements==0;
  }

  public String toString() {
  // résultat : this en clair
    if (estVide()) {return "{}";}
    StringBuffer resul = new StringBuffer("{ "+ elements[0]);
    for (int i=1; i<nbElements; i++) {
      resul.append(" , " + elements[i]);
    }
    resul.append(" }"); return resul.toString();
  }
```

```
private int indiceDe(T e) {
// résultat : indice de l'élément égal à e, 0..nbElements-1,
// -1 si e est absent
  int i = 0; int j = nbElements-1;
  while(j>=i){
     int m=(i+j)/2;
     int comparaison = ((T) elements[m]).compareTo(e);
     if (comparaison==0) {return m;}
     else if (comparaison<0) {i=m+1;}
     else {j=m-1;}
  }
  return -1;
}


public boolean contient(T e) {
// résultat : indique si e appartient à this
  return indiceDe(e)!=-1;
}


public void ajouteElement(T e) {
// effet : ajoute e à this (aucun effet si e est déjà dans this)
// (après, e appartient à this)
  if (!contient(e)) {
    if (elements!=null && nbElements<elements.length){
    // place suffisante
      int i=nbElements;
      while (i>0 && ((T) elements[i-1]).compareTo(e)>0) {
        elements[i]=elements[i-1]; i--;
      }
      elements[i]=e;
    }
    else { // place insuffisante, création d'un nouveau tableau
      Object[] nouveau;
      if (nbElements==0) {
        nouveau = new Object[taillePourSingleton];
      }
      else {
        nouveau=new Object[facteurDeCroissance*elements.length];
      }
      int i=nbElements;
      while (i>0 && ((T) elements[i-1]).compareTo(e)>0) {
        nouveau[i]=elements[i-1]; i--;
      }
      nouveau[i] = e; i--;
      while (i>=0) { nouveau[i]=elements[i]; i--;}
      elements=nouveau;
    }
    nbElements++;
  }
}
```

```
public void retireElement(T e) {
// effet : retire e de this (aucun effet si e n'est pas dans this)
// (après, e n'appartient pas à this)
  int k = indiceDe(e);
  if (k!=-1) {retireIemeElement(k);}
}


private void retireIemeElement(int k) {
// prérequis : 0<=k<nbElements
// effet : retire le k ième élément
  nbElements--;
  if (nbElements==0) {elements = null;}
  else if (nbElements < elements.length/facteurDeCroissance) {
  // création d'un tableau plus petit
    Object[] nouveau =
        new Object[elements.length/facteurDeCroissance];
    for (int i=0;i<k; i++) {nouveau[i]=elements[i];}
    for (int i=k;i<nbElements; i++) {nouveau[i]=elements[i+1];}
    elements=nouveau;
  }
  else {
    for (int i=k;i<nbElements; i++) {elements[i]=elements[i+1];}
  }
}


private void ajouteAuBout(T e) {
// effet : ajoute l'élément e "au bout" du tableau des éléments
  if (elements!=null && nbElements<elements.length){
  // place suffisante
    elements[nbElements]=e;
  }
  // place insuffisante, création d'un nouveau tableau
  else if (nbElements==0) {
    elements= new Object[taillePourSingleton]; elements[0]=e;
  }
  else {
    Object[] nouveau =
          new Object[facteurDeCroissance*elements.length];
    for (int i=0; i<nbElements; i++) {nouveau[i]=elements[i];}
    nouveau[nbElements] = e;
    elements=nouveau;
  }
  nbElements++;
}
```

```
public static <T extends Comparable<T>>
     Ensemble<T> union(Ensemble<T> e1, Ensemble<T> e2) {
  Ensemble<T> resul = new Ensemble();
  int i1=0; int i2=0;
  while (i1<e1.nbElements && i2<e2.nbElements) {
    T v1 = (T) e1.elements[i1]; T v2 = (T) e2.elements[i2];
    int compar=v1.compareTo(v2);
    if (compar<0) {resul.ajouteAuBout(v1); i1++;}
    else if (compar>0) {resul.ajouteAuBout(v2); i2++;}
    else {resul.ajouteAuBout(v1); i1++; i2++;}
  }
  while (i1<e1.nbElements) {
    resul.ajouteAuBout((T) e1.elements[i1]); i1++;
  }
  while (i2<e2.nbElements) {
    resul.ajouteAuBout((T) e2.elements[i2]); i2++;
  }
  return resul;
}


public static <T extends Comparable<T>>
      Ensemble<T> intersection(Ensemble<T> e1, Ensemble<T> e2) {
  Ensemble<T> resul = new Ensemble();
  int i1=0; int i2=0;
  while (i1<e1.nbElements && i2<e2.nbElements) {
    T v1 = (T) e1.elements[i1]; T v2 = (T) e2.elements[i2];
    int compar=v1.compareTo(v2);
    if (compar<0) {i1++;}
    else if (compar>0) {i2++;}
    else {resul.ajouteAuBout(v1); i1++; i2++;}
  }
  return resul;
}


public static <T extends Comparable<T>>
       Ensemble<T> difference(Ensemble<T> e1, Ensemble<T> e2) {
  Ensemble<T> resul = new Ensemble();
  int i1=0; int i2=0;
  while (i1<e1.nbElements && i2<e2.nbElements) {
    T v1 = (T) e1.elements[i1]; T v2 = (T) e2.elements[i2];
    int compar=v1.compareTo(v2);
    if (compar<0) {resul.ajouteAuBout(v1); i1++;}
    else if (compar>0) {i2++;}
    else {i1++; i2++;}
  }
  while (i1<e1.nbElements) {
    resul.ajouteAuBout((T) e1.elements[i1]); i1++;
  }
  return resul;
}
```

```
  //=================== parcoureur d'ensemble ===================

  public class Parcours implements parcours.Parcours<T>{
    private int courant;
    Parcours() {
    // Parcours initialisé sur le plus petir élément
      courant=0;
    }
    Parcours(Parcours p) {
    // Parcours initialisé avec l'état du Parcours p
      courant=p.courant;
    }
    public void tete() {
    // effet: positionne this sur le plus petit élément de l'ensemble
    // (parcours en fin si ensemble vide)
      courant=0;
    }
    public void suivant() {
    // effet : positionne this sur l'élément suivant,
    // ne fait rien si estEnFin()
      if (courant<nbElements) {courant++;}
    }
    public boolean estEnFin() {
    // résultat : indique si this est en fin
    // (au dela de la queue, en deça de la tête)
      return courant==nbElements;
    }
    public void retireElement() {
    // prérequis : this n'est pas en fin
    // effet : retire l'élément courant de this
    // le suivant de l'élément retiré, s'il existe, devient
    // l'élement courant, sinon this passe en fin de parcours
      retireIemeElement(courant);
    }
    public T elementCourant() {
    // prérequis : this n'est pas en fin
    // résultat : l'élément courant de this
      if (estEnFin()) {
        System.out.println(
           "erreur : acces hors domaine de l'ensemble");
         Thread.dumpStack(); System.exit(0);
      }
      return (T) elements[courant];
    }
  }
  //=============================================================

}
```