Haute École de Bruxelles
École Supérieure d'Informatique

Bachelor en Informatique
Laboratoires Java - 1ère année

TD Boucles

Résumé

Voyons ici comment incorporer des boucles, les structures répétitives, dans nos codes et comment les utiliser à bon escient.

version 2014-2015 (21 janvier 2015)

1 Les boucles

Si on veut faire effectuer un travail répétitif, il faut indiquer deux choses :

1. Le travail à répéter
2. La manière de continuer la répétition ou de l'arrêter.

1.1 tant que

Le «`tant que` »est une structure qui demande à l'exécutant de répéter une tâche (une ou plusieurs instructions) tant qu'une condition donnée est vraie.

En pseudo-code :

```
tant que condition faire
    séquence d'instructions à exécuter
fin tant que
```

La **condition** est une expression délivrant un résultat **booléen** (vrai ou faux).

Il faut qu'il y ait dans la séquence d'instructions comprise entre `tant que` et `fin tant que` au moins **une instruction qui modifie** une des composantes de **la condition** de telle manière qu'elle puisse **devenir fausse** à un moment donné. Dans le cas contraire, la condition reste **indéfiniment vrai**e et la boucle va tourner sans fin, c'est ce qu'on appelle une **boucle infinie**.

Si la condition est fausse dès le début, la tâche n'est jamais exécutée.

Par exemple :

Afficher les nombres plus petits que 10

On affiche uniquement les nombres inférieurs (pas strictement) à 10 .

```
// Affiche les nombres de 1 à 10.
module compterJusque10 () // version avec tant que
    nb : entier
    nb ← 1 // c'est le premier nombre à afficher
    tant que nb ≤ 10 faire // c'est le premier nombre à afficher
      afficher nb  // on affiche la valeur de la variable nb
      nb ← nb + 1 // on passe au nombre suivant
```

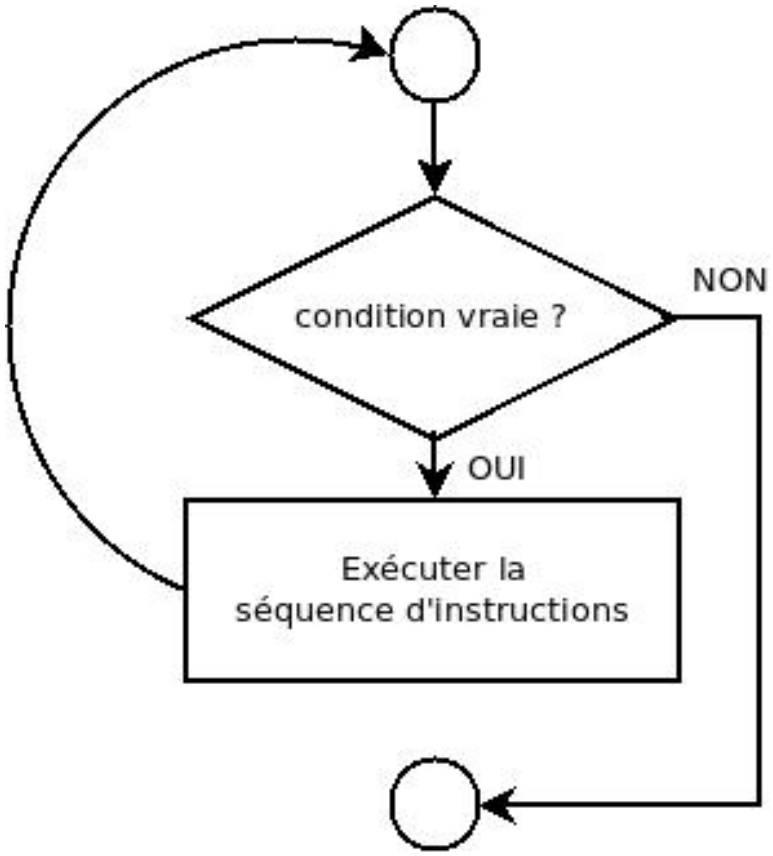

FIGURE 1 – boucleTq.jpg

```
    fin tant que
fin module
```

Somme de nombres

Après chaque nombre, on demande à l'utilisateur s'il y a encore un nombre à additionner.

```
// Lit des valeurs entières et retourne la somme des valeurs lues.
module sommeNombres() → entier
    valeur : entier // un des termes de l'addition
    somme : entier // la somme
    somme ← 0
    lire valeur
    tant que valeur ≥ 0 faire
```

```
      somme ← somme + valeur
      lire valeur // remarquer l'endroit où on lit une valeur.
    fin tant que
retourner somme
fin module
```

1.2 faire - jusqu'à ce que

Cette structure est très proche du «`tant que` »à deux différences près :

1. Le **test** est fait **à la fin** et pas au début. La tâche est donc toujours **exécutée au moins une fois**.
2. On donne la **condition pour arrêter** et pas pour continuer.

En pseudo-code :

```
faire
    séquence d'instructions à exécuter
jusqu'à ce que condition
```

La **condition** est une expression délivrant un résultat **booléen** (vrai ou faux).

Il faut que la séquence d'instructions comprise entre `faire` et `jusqu'à ce que` contienne au moins **une instruction qui modifie la condition** de telle manière qu'elle puisse **devenir vraie** à un moment donné pour arrêter l'itération.

La tâche est toujours **exécutée au moins une fois**.

Par exemple :

Somme de nombres

Après chaque nombre, on demande à l'utilisateur s'il y a encore un nombre à additionner.

```
// Lit des valeurs entières et retourne la somme des valeurs lues.
module sommeNombres() → entier
    encore : booléen // est-ce qu'il reste encore une valeur à additionner ?
    valeur : entier // un des termes de l'addition
    somme : entier // la somme
```

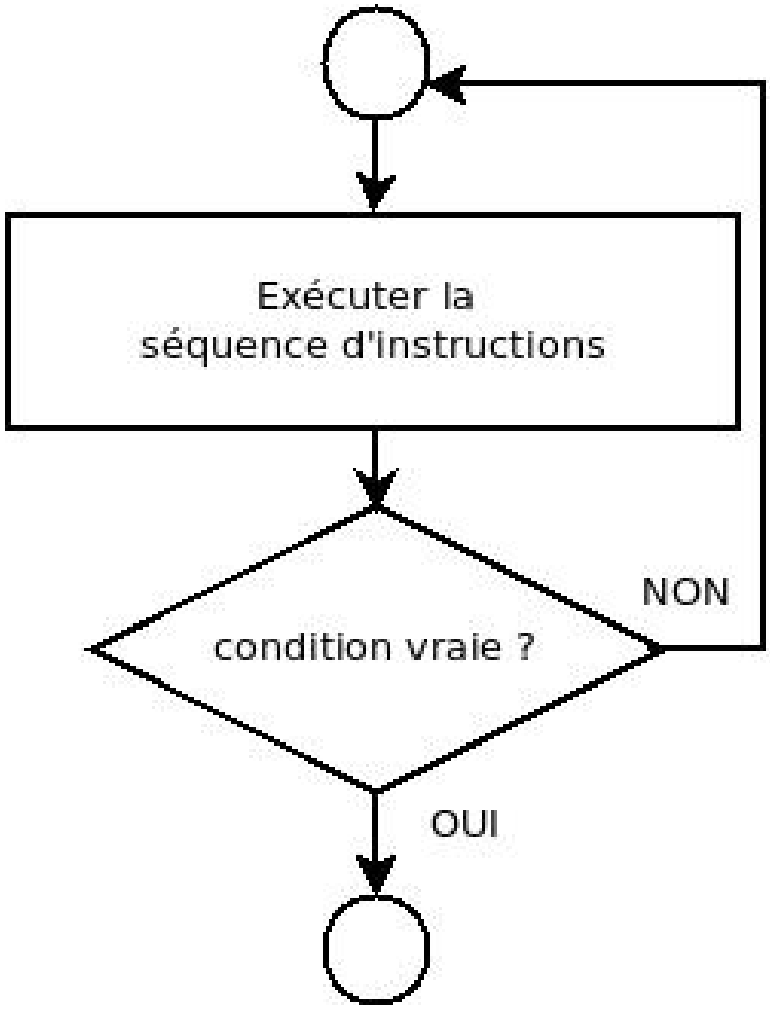

FIGURE 2 – boucleFaire.jpg

```
    somme ← 0
    faire
      lire valeur
      somme ← somme + valeur
      lire encore
    jusqu'à ce que NON encore
    retourner somme
fin module
```

Avec cette solution, on additionne au moins une valeur.

1.3 pour

On va indiquer combien de fois la tâche doit être répétée. Cela se fait au travers d'une **variable de contrôle** dont la valeur va évoluer **à partir d'une valeur de départ jusqu'à une valeur finale**.

En pseudo-code :

```
pour variable de début à fin [par pas] faire
    séquence d'instructions à exécuter
fin pour
```

est équivalent à

```
variable ← début
tant que variable ≤ fin faire
    séquence d'instructions à exécuter
    variable ← variable + pas // ou variable ← variable + 1 si le pas est omis.
fin tant que
```

Dans ce type de structure, `début`, `fin` et `pas` peuvent être des constantes, des variables ou des expressions (le plus souvent à valeurs entières mais on admettra parfois des réels).

Le `pas` est facultatif, et généralement omis (dans ce cas, sa valeur par défaut est 1).

Ce `pas` est parfois négatif, dans le cas d'un compte à rebours, par exemple `pour n de 10 à 1 par -1`.

1. Quand le `pas` est positif, la boucle s'arrête lorsque la `variable` dépasse la valeur de `fin`.
2. Par contre, avec un `pas` négatif, la boucle s'arrête lorsque la `variable` prend une valeur plus petite que la valeur de `fin`.

On considérera qu'au cas (à éviter) où

1. `début` est strictement supérieur à `fin` et le `pas` est positif, la séquence d'instructions n'est jamais exécutée (mais ce n'est pas le cas dans tous les langages de programmation !).
2. Idem si `début` est strictement inférieur à `fin` mais avec un `pas` négatif.

Attention de **ne pas modifier** dans la séquence d'instructions une des variables de contrôle `début`, `fin` ou `pas` !

Il est aussi fortement **déconseillé de modifier «manuellement** »la `variable` au sein de la boucle `pour`. Il ne faut pas l'initialiser en début de boucle, et ne pas s'occuper de sa modification, l'instruction `variable` ← `variable + pas` étant automatique et implicite à chaque étape de la boucle.

Il est aussi déconseillé d'utiliser `variable` à la sortie de la structure pour sans lui affecter une nouvelle valeur.

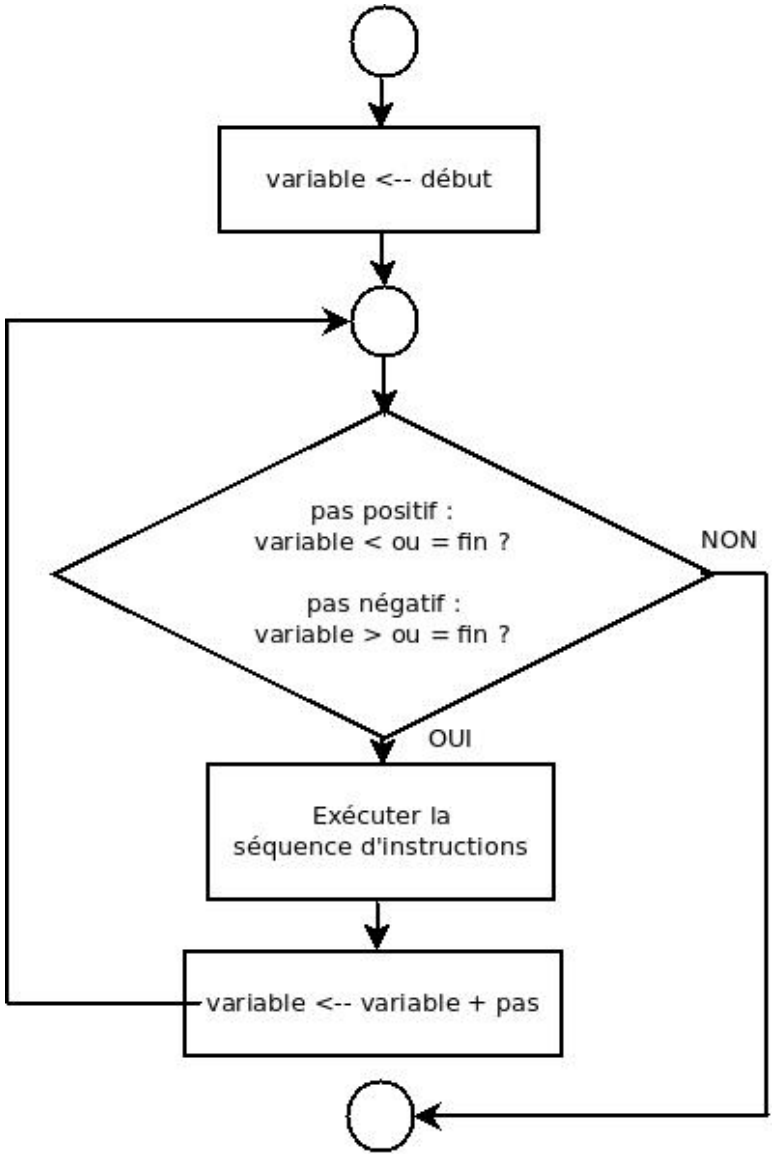

FIGURE 3 – bouclePour.jpg

Par exemple :

```
// Affiche les nombres de 1 à 10.
module compterJusque10 () // version avec pour
    nb : entier
    pour nb de 1 à 10 faire // par défaut le pas est de 1
      afficher nb
    fin pour
fin module
```

Afficher les nombres plus petits que n

On affiche uniquement les nombres inférieurs (pas strictement) à n.

```
// Reçoit un nombre et affiche les nombres de 1 à ce nombre.
module afficherN(n↓ : entier)
    nb : entier
    pour nb de 1 à n faire
      afficher nb
    fin pour
fin module
```

Afficher les nombres pairs plus petits que 10

On affiche uniquement les nombres pairs jusqu'à 10.

```
// Reçoit un nombre et affiche les nombres pairs jusqu'à ce nombre.
// n : limite des nombres à afficher.
Exemple : si n vaut 10, les nombres pairs de 1 à 10 sont : 2, 4, 6, 8, 10.
module afficherPair (n↓ : entier)
    nb : entier
    pour nb de 2 à n par 2 faire
      afficher nb
    fin pour
fin module
```

Afficher les nombres pairs plus petits que n

On affiche uniquement les nombres pairs jusqu'à la limite n.

```
// Reçoit un nombre et affiche les nombres pairs jusqu'à ce nombre.
// n : limite des nombres à afficher.
// Exemple : si n vaut 10, les nombres pairs de 1 à 10 sont : 2, 4, 6, 8, 10.
module afficherPair (n↓ : entier)
    i: entier
    pour i de 1 à n DIV 2 faire
      afficher 2 * i
    fin pour
fin module
```

Afficher n nombres pairs

On affiche les n premiers nombres pairs.

```
// Reçoit un nombre et affiche ce nombre de nombres pairs.
// n: le nombre de nombres à afficher.
// Exemple : si n vaut 10, les 10 premiers nombres pairs sont : 2, 4, 6, 8, 10, 12,
module afficherPair ()
    i : entier
    pour i de 1 à n faire
      afficher 2 * i
    fin pour
fin module
```

Somme de nombres

L'utilisateur indique le nombre de termes au départ.

```
// Lit des valeurs entières et retourne la somme des valeurs lues.
module sommeNombres() → entier
    nbValeurs : entier // nb de valeurs à additionner
    valeur : entier // un des termes de l'addition
    somme : entier // la somme
    i : entier // itérateur
    somme ← 0 // la somme se construit petit à petit. Elle vaut 0 au départ
    lire nbValeurs
    pour i de 1 à nbValeurs faire
      lire valeur
      somme ← somme + valeur
    fin pour
    retourner somme
```

```
fin module
```

1.4 Quel type de boucle choisir ?

En pratique, il est possible d'utiliser systématiquement la boucle tant que qui peut s'adapter à toutes les situations. Cependant,

- il est plus clair d'utiliser la boucle `pour` dans les cas où le nombre d'itérations est fixé et connu à l'avance (par là, on veut dire que le nombre d'itérations est déterminé au moment où on arrive à la boucle).
- La boucle `faire` convient quant à elle dans les cas où le contenu de la boucle doit être parcouru au moins une fois,
- alors que dans `tant que`, le nombre de parcours peut être nul si la condition initiale est fausse.

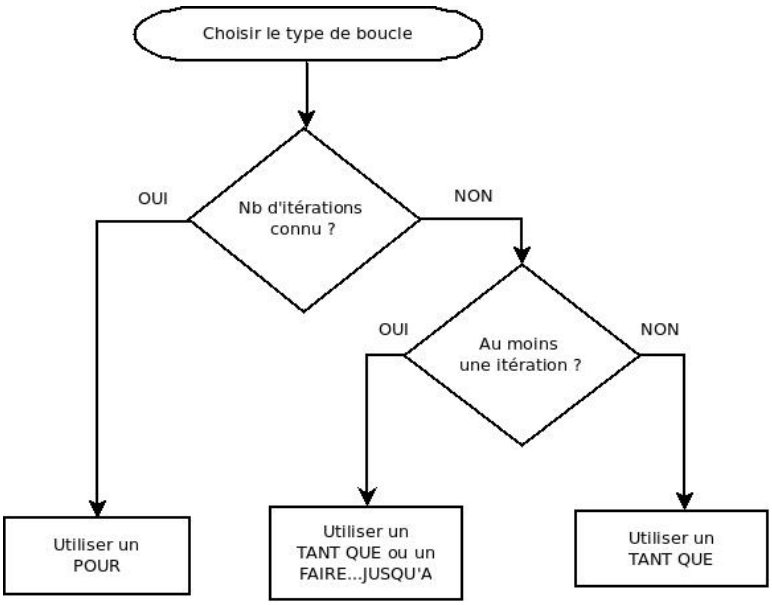

FIGURE 4 – boucleChoixType.jpg

1.5 suite de nombres

Un exemple simple pourrait être celui-ci : «*Écrire l'algorithme qui affiche les n premiers termes de la suite : 2, 4, 6. . .* »

Puisqu'on doit écrire plusieurs nombres et qu'on sait exactement combien, on se tournera tout naturellement vers une boucle `pour`.

Le cas le plus simple est lorsque le nombre à afficher à l'étape `i` peut être calculé en fonction de `i` seulement. L'algorithme est alors

```
pour i de 1 à n faire
    afficher f (i)
fin pour
```

Par exemple, pour afficher la suite des n premiers nombres pairs

```
module nombrePair (n↓ : entier)
    i : entier
    pour i de 1 à n faire
      afficher 2 * i
    fin pour
fin module
```

Parfois, il est difficile (voire impossible) de trouver f (i). On suivra alors une autre approche qui revient à calculer un nombre à afficher à partir du nombre précédemment affiché (ou, plus exactement, de calculer le suivant à partir du nombre qu'on vient d'afficher). La structure générale est alors

```
nb ← {1re valeur à afficher}
pour i de 1 à n faire
    afficher nb
    nb ← {calculer ici le nb suivant}
fin pour
```

Dans l'exemple de la suite paire, le 1er nombre à afficher est 2 et le nombre suivant se calcule en ajoutant 2 au nombre courant.

```
module suite1 (n↓ : entier)
    nb, i : entiers
    nb ← 2
    pour i de 1 à n faire
      afficher nb
      nb ← nb + 2
    fin pour
fin module
```

1.6 3 pas en avant, 2 pas en arrière

Dans certains cas, il n'est pas possible de déduire directement le nombre suivant en connaissant juste le nombre précédent. Prenons un exemple un peu plus compliqué pour l'illustrer. «*Écrire l'algorithme qui affiche les n premiers termes de la suite : 1, 2, 3, 4, 3, 2, 3, 4, 5, 4, 3. . .* »

Si on vient d'écrire, disons un 3, impossible sans information supplémentaire, de connaitre le nombre suivant. Il faudrait savoir si on est en phase d'avancement ou de recul et combien de pas il reste à faire dans cette direction.

Ajoutons des variables pour retenir l'`état` où on est.

```
module suite3Avant2Arrière(n↓ : entier)
    nb, nbPasRestants, direction, i : entiers
    nb ← 1
    nbPasRestants ← 3 // 3 pas
    direction ← 1 // en avant
    pour i de 1 à n faire
      afficher nb
      nb ← nb + direction // faire un pas dans la bonne direction
      nbPasRestants ← nbPasRestants - 1
      si nbPasRestants = 0 alors // il est temps de changer de direction
        direction ← -direction
        si direction = 1 alors
          nbPasRestants ← 3
        sinon
          nbPasRestants ← 2
        fin si
      fin si
    fin pour
fin module
```

On obtient un algorithme plus long mais qui respecte toujours le schéma vu.

Un conseil : essayez de respecter ce schéma et vous obtiendrez plus facilement un algorithme correct et lisible, également dans les cas particuliers.

2 selon que

Avec ces structures, plusieurs branches d'exécution sont disponibles. L'ordinateur choisit la branche à exécuter en fonction de la valeur d'une variable (ou parfois d'une expression) ou de la condition qui est vraie.

2.1 selon que (version avec listes de valeurs)

En pseudo-code :

```
selon que variable vaut
    liste_1 de valeurs séparées par des virgules :
      // instructions lorsque la valeur de la variable est dans liste_1
    liste_2 de valeurs séparées par des virgules :
      // instructions lorsque la valeur de la variable est dans liste_2
    ...
    liste_n de valeurs séparées par des virgules :
      // instructions lorsque la valeur de la variable est dans liste_n
    autres :
      // instructions lorsque la valeur de la variable
      // ne se trouve dans aucune des listes précédentes
fin selon que
```

Notez que le cas `autres` est facultatif.

Dans ce type de structure, comme pour la structure `si-alors-sinon`, une seule des séquences d'instructions sera exécutée. On veillera à ne pas faire apparaitre une même valeur dans plusieurs listes. Cette structure est une simplification d'écriture de plusieurs alternatives imbriquées.

Elle est équivalente à :

```
si variable = une des valeurs de la liste_1 alors
    // instructions lorsque la valeur est dans liste_1
sinon
    si variable = une des valeurs de la liste_2 alors
      // instructions lorsque la valeur est dans liste_2
    sinon
      ...
      si variable = une des valeurs de la liste_n alors
        // instructions lorsque la valeur est dans liste_n
      sinon
        // instructions lorsque la valeur de la variable
        // ne se trouve dans aucune des listes précédentes
      fin si
    fin si
fin si
```

Écrivons un algorithme qui lit un jour de la semaine sous forme d'un nombre entier (1 pour lundi, . . ., 7 pour dimanche) et qui affiche en clair ce jour de la semaine.

```
// Lit un nombre entre 1 et 7 et affiche en clair le jour de la semaine correspondan
module jourSemaine()
    jour : entier
    lire jour
    selon que jour vaut
      1 : afficher "lundi"
      2 : afficher "mardi"
      3 : afficher "mercredi"
      4 : afficher "jeudi"
      5 : afficher "vendredi"
      6 : afficher "samedi"
      7 : afficher "dimanche"
    fin selon que
fin module
```

En Java :

```
switch (variable){
    case val1 :
      // instructions lorsque la valeur de la variable est val1
      break;
    case val2 :
    case val3 :
    case val4 :
      // instructions lorsque la valeur de la variable est val2 ou val3 ou val4
      break;
    ...
    case valN  :
      // instructions lorsque la valeur de la variable est valN
      break;
    default :
      // instructions lorsque la valeur de la variable
      // ne se trouve dans aucune des listes précédentes
}
```

Notez que le cas `default` est facultatif.

Notez le `break` à la fin de chaque (groupe de) `case`.

La variable peut être de type `byte`, `short`, `char`, `intString` et les types énumérés que nous verrons plus tard.

Elle est équivalente à :

```
if (variable == val1){
      // instructions lorsque la valeur de la variable est val1
} else if (variable == val2 || variable == val3 || variable == val4){
      // instructions lorsque la valeur de la variable est val2 ou val3 ou val4
} else if (variable == valN){
      // instructions lorsque la valeur de la variable est valN
} else {
      // instructions lorsque la valeur de la variable
      // ne se trouve dans aucune des listes precedentes
}
```

Par exemple :

```
import java.util.Scanner;
public class Test{
  public static void main(String[] args){
      Scanner clavier = new Scanner(System.in);
      String produit = clavier.next();
      switch(produit) {
        case "Coca" :
        case "Sprite" :
        case "Fanta" :
        prixDistributeur = 60;
        break;
      case "IceTea" :
        prixDistributeur = 70;
        break;
      default :
        prixDistributeur = 0;
        break;
      }
      System.out.println(prixDistributeur);
    }
}
```

2.2 selon que (version avec conditions)

En pseudo-code :

```
selon que
    condition_1 :
      // instructions lorsque la condition_1 est vraie
    condition_2 :
      // instructions lorsque la condition_2 est vraie
    ...
    condition_n :
      // instructions lorsque la condition_n est vraie
```

```
    autres :
      // instructions à exécuter quand aucune
      // des conditions précédentes n'est vérifiée
fin selon que
```

Comme précédemment, une et une seule des séquences d'instructions est exécutée. On veillera à ce que les conditions ne se «recouvrent »pas, c'est-à-dire que deux d'entre elles ne soient jamais vraies simultanément.

C'est équivalent à :

```
si condition_1 alors
    // instructions lorsque la condition_1 est vraie
sinon
    si condition_2 alors
      // instructions lorsque la condition_2 est vraie
    sinon
      ...
      si condition_n alors
        // instructions lorsque la condition_n est vraie
      sinon
        // instructions à exécuter quand aucune
        // des conditions précédentes n'est vérifiée
      fin si
    fin si
fin si
```

Par exemple :

```
// Lit un nombre et affiche si ce nombre est strictement positif , strictement négat
module signeNombre()
    nb : entier
    lire nb
    selon que
      nb < 0 :
        afficher "le nombre", nb, " est négatif"
      nb > 0 :
        afficher "le nombre", nb, " est positif"
      autres :
        afficher "le nombre", nb, " est nul"
    fin selon que
```

`fin module`

En Java :

Il n'existe pas de `switch` avec condition, il faut l'écrire comme une succession de `if`.

```
if (condition_1){
      // instructions lorsque la condition_1 est vraie
} else if (condition_2){
      // instructions lorsque la condition_2 est vraie
} ...
} else if (condition_n){
      // instructions lorsque la condition_n est vraie
} else {
      // instructions a executer quand aucune
      // des conditions precedentes n est verifiee
}
```

Par exemple :

```
import java. util .Scanner;
public class Test {
    public static void main(String [] args) {
      Scanner clavier = new Scanner(System.in);
      int nb = clavier.nextInt();
      if (nb>0) {
        System.out. println ("␣positif␣");
      } else if (nb==0) {
        System.out. println ("nul");
      } else {
        System.out. println ("␣negatif␣");
      }
    }
}
```

3 Exercices

Maintenant, mettons tout ça en pratique.

3.1 Compréhension d'algorithme

Pour ces exercices, nous vous demandons de comprendre des algorithmes donnés.

Compréhension

Que vont-ils afficher ?

```
— module boucle1 ()
      x : entier
      x ← 0
      tant que x < 12 faire
        x ← x+2
      fin tant que
      afficher x
  fin module
```

```
— module exerciceB()
      a,b,c : entier
      lire b,a
      si a > b alors
        c ← a DIV b
      sinon
        c ← b MOD a
      fin si
      afficher c
  fin module
```

Si les nombres lus sont respectivement 2 et 3 ?

__ Si les nombres lus sont respectivement 4 et 1 ?

```
— module exerciceC ()
      x1, x2 : entier
      ok : booléen
      lire x1, x2
      ok ← x1 > x2
      si ok alors
        ok ← ok ET x1 = 4
      sinon
        ok ← ok OU x2 = 3
      fin si
      si ok alors
        x1 ← x1 * 1000
      fin si
      afficher x1 + x2
  fin module
```

Si les nombres lus sont respectivement 2 et 3 ?

_____ Si les nombres lus sont respectivement 4 et 1 ?

3.2 Compréhension de codes Java

Pour ces exercices, nous vous demandons de comprendre des codes Java donnés.

Compréhension

Que vont-ils afficher si à chaque fois les deux nombres lus au départ sont successivement 2, 3 et 4 ?

```
import java.util.Scanner;
public class Exercice1 {
    public static void main(String [] args) {
        Scanner clavier = new Scanner(System.in);
        int nb1 = clavier.nextInt();
        int nb2 = clavier.nextInt();
        int nb3 = clavier.nextInt();
        if (nb1 < nb2){
          System.out.print(nb1);
        } else {
          System.out.print(nb2);
        }
    }
}
```

```
import java.util.Scanner;
public class Exercice2 {
    public static void main(String [] args) {
        Scanner clavier = new Scanner(System.in);
        int nb1 = clavier.nextInt();
        int nb2 = clavier.nextInt();
        int nb3 = clavier.nextInt();
        if (nb1 > nb2 && nb1 > nb3){
          System.out.print(nb1);
        } else {
            if (nb2 > nb3){
              System.out.print(nb2);
            } else {
              System.out.print(nb3);
            }
        }
    }
}
```

```
import java.util.Scanner;
public class Exercice3 {
    public static void main(String [] args) {
        Scanner clavier = new Scanner(System.in);
        int nb1 = clavier.nextInt();
        int nb2 = clavier.nextInt();
        int nb3 = clavier.nextInt();
        switch (nb1){
```

```
            case 1 : System.out.print("premier"); break;
            case 2 : System.out.print("deuxieme"); break;
            case 3 : System.out.print("troisieme"); break;
            default : System.out.print("pas dans le trio");
        }
    }
}
```

```
import java.util.Scanner;
public class Exercice3 {
    public static void main(String [] args) {
        Scanner clavier = new Scanner(System.in);
        int nb1 = clavier.nextInt();
        int nb2 = clavier.nextInt();
        int nb3 = clavier.nextInt();
        switch (nb1){
            case 1 : System.out.print("premier");
            case 2 : System.out.print("deuxieme");
            case 3 : System.out.print("troisieme");
            default : System.out.print("pas dans le trio");
        }
    }
}
```

3.3 À vous de jouer...

Il est temps de se lancer et d'écrire vos premiers modules et programmes Java correspondant. Voici quelques conseils pour vous guider dans la résolution de tels problèmes :

- il convient d'abord de bien comprendre le problème posé ; assurez-vous qu'il est parfaitement spécifié ;
- déclarez ensuite les variables (et leur type) qui interviennent dans l'algorithme ; les noms des variables risquant de ne pas être suffisamment explicites ;
- **mettez en évidence les variables «données », les variables «résultats »et les variables de travail ;**
- n'hésitez pas à faire une ébauche de résolution en français avant d'élaborer l'algorithme définitif pseudo-codé.
- Écrivez la partie algorithmique **AVANT** de vous lancer dans la programmation en Java.

Écrivez les algorithmes et codez les programmes Java correspondant qui

1. étant donné deux nombres quelconques, recherche et affiche le plus grand des deux. Attention ! On ne veut pas savoir si c'est le premier ou le deuxième qui est le plus grand mais bien quelle est cette plus

grande valeur. Le problème est donc bien défini même si les deux nombres sont identiques.

2. étant donné trois nombres quelconques, recherche et affiche le plus grand des trois.
3. affiche un message indiquant si un entier est strictement négatif, nul ou strictement positif.
4. étant donné trois nombres, recherche et affiche si le premier des trois appartient à l'intervalle donné par le plus petit et le plus grand des deux autres (bornes exclues). Qu'est-ce qui change si on inclut les bornes ?
5. étant donné une équation du deuxième degré, déterminée par le coefficient de x^2, le coefficient de x et le terme indépendant, recherche et affiche la (ou les) racine(s) de l'équation (ou un message adéquat s'il n'existe pas de racine réelle).
6. à partir d'un moment exprimé par 2 entiers, heure et minute, affiche le moment qu'il sera une minute plus tard.
7. vérifie si une année est bissextile. Pour rappel, les années bissextiles sont les années multiples de 4. Font exception, les multiples de 100 (sauf les multiples de 400 qui sont bien bissextiles). Ainsi 2012 et 2400 sont bissextile mais pas 2010 ni 2100.

Stationnement alternatif

Dans une rue où se pratique le stationnement alternatif, du 1 au 15 du mois, on se gare du côté des maisons ayant un numéro impair, et le reste du mois, on se gare de l'autre côté. Écrivez un algorithme et le code java correspondant qui, sur base de la date du jour et du numéro de maison devant laquelle vous vous êtes arrêté, indique si vous êtes bien stationné ou non.

La fièvre monte

Chez l'humain la température corporelle normale moyenne est de 37 °C (entre 36,5 °C et 37,5 °C selon les individus). La fièvre est définie par une température rectale au repos supérieure ou égale à 38,0 °C. Une fièvre au-delà de 40 °C est considérée comme un risque de santé majeur et immédiat. Lorsque la fièvre est modérée (de 37,7 °C à 37,9 °C), on parle de fébricule.

[Wikipedia]

Écrivez un module fièvre qui lit une température au clavier et qui affiche si le patient a de la température (supérieure ou égale à 38,0°C) ET si cette cette fièvre est modérée (entre 38,0°C et 40,0°C) ou à risque (strictement supérieur à 40,0°C). Rien ne doit être affiché si le patient n'a pas de fièvre.

Écrivez le code java correspondant.

Taxes communales

Dans ma commune, les taxes communales des enlèvements des immondices s'élèvent à

- 80€ pour une personne isolée ;
- 135€ pour une famille de 2 ou 3 personnes ;
- 175€ pour une famille de 4 personnes ou plus.

Écrivez un module qui lit le nombre de personnes composant la famille et qui affiche le prix de la taxe à payer.

Écrivez le code java correspondant.

Au cinéma

À Bruxelles, lors de chaque projection cinématographique, une taxe de 0,5€ est prélevée sur le prix du billet de chaque spectateur.

- Écrivez un module qui lit le nombre de spectateurs et qui affiche le prix de la taxe à payer.
- Écrivez le code java correspondant.
- Si le film projeté est un documentaire, aucune taxe n'est prélevée. Écrivez un module qui lit le nombre de spectateurs et un booléen (à vrai si le film est un documentaire et faux sinon) et qui affiche le prix de la taxe à payer.
- Écrivez le code java correspondant.