Université Paris 7 - Denis Diderot
Licence Sciences et applications
IF1 : Initiation à l'informatique et la programmation
2010-2011
2ème partie

Les itérations

On a vu que l'exécution séquentielle des instructions d'un programme pouvait être rompue

1. par des instructions conditionnelles
2. par des appels de fonctions (méthodes)

Une des caractéristiques d'un ordinateur est par ailleurs sa capacité à effectuer des tâches simples de façon répétitive.

De nombreuses opérations algorithmiques s'expriment sous la forme de répétitions (ou d'itérations) de séquences d'instructions : des exemples simples en sont par exemple le calcul des n premiers nombres d'une suite récurrente de nombres u_i ou la recherche du premier nombre appartenant à une telle suite et possédant une propriété donnée (par exemple être supérieur à une valeur fixée).

Le schéma de boucle «Répéter»

Exemple introductif : afficher une ligne constituée de n caractères « * » consécutifs.

L'écriture d'un programme consistant à écrire explicitement n appels consécutifs à la fonction `Deug.print` n'est pas raisonnable lorsque n est grand.

Elle n'est pas possible lorsque n est variable (par exemple, si n est une valeur entrée par l'utilisateur).

On propose d'introduire dans notre langage de description d'algorithmes une construction du type

répéter n *fois «quelque chose»*

Schéma algorithmique :

```
répéter n fois
    afficher le caractère *
```

Exercices :

1. Écrire un algorithme affichant une ligne composée d'un caractère +, suivi de n caractères −, suivis par un caractère +.
2. Écrire un algorithme qui lit dix entiers puis affiche leur somme et leur moyenne.
3. Écrire un algorithme qui, après avoir lu l'entier n, calcule le terme u_n de la suite u_i telle que $u_0 = 1$ et pour tout $n \geq 1$, $u_n = 4 \times u_{n-1} + 3$.

Le schéma de boucle «pour»

Lors de l'exécution d'une boucle, il est souvent utile de connaître le numéro de l'itération courante (ou, de façon presque équivalente, le nombre d'itérations déjà faites).

Exemple : afficher un triangle de la forme

```
*
**
***
...
**********
```

Algorithme 1 (*première version*) :

```
pour i variant de 1 à 10
    afficher i étoiles
    aller à la ligne
```

Algorithme 2 (*version plus détaillée*) :

```
pour i variant de 1 à 10
    répéter i fois
        afficher une étoile
    aller à la ligne
```

Remarque : on a ici utilisé une notation informelle où l'indentation est significative. En Java, on ne pourra pas procéder de la même manière.

Traduction en boucle «for» de Java

En Java, la boucle *pour* décrite précédemment est exprimée sous la forme :

```
for (var = b1; var <= b2; var = var + 1)
    instruction ou bloc d'instructions
```

Par exemple :

```
int i;
for (i = 1; i <= n; i = i + 1)
    Deug.println("Itération numéro " + i);
```

et avec des boucles imbriquées :

```
int i, j;
for (i = 1; i <= n; i = i + 1) {
    for (j = 1; j <= n; j = j + 1) {
        Deug.print('*');
    }
    Deug.println();
}
```


Fonctionnement du «for» simple de Java

```
for (var = b1; var <= b2; var = var + 1)
    instruction ou bloc d'instructions
```

La partie de contrôle de la boucle `for` consiste en trois expressions séparées par des points-virgule «;» :

1. la première est évaluée avant d'entrer dans la boucle et vise à initialiser la variable de contrôle ;
2. la deuxième est de type booléen et est évaluée au début de chaque itération pour déterminer si la boucle doit être poursuivie ;
3. la dernière est évaluée à la fin de chaque itération et vise à mettre à jour la variable de contrôle.

Exercices :

1. Écrire un programme qui lit un entier n et affiche les n premiers nombres pairs.
2. Écrire un programme qui lit un entier n et affiche les n premières puissances de 2.
3. Traduire en Java l'algorithme 3 de la série d'exercices précédente.
4. Écrire un programme qui lit un entier n, dont on vérifiera qu'il satisfait la condition $n \geq 2$ et si c'est le cas affichera les $n + 1$ premiers nombres de la suite telle que $u_0 = 1$, $u_1 = 4$ et $u_n = 2 \times u_{n-1} + u_{n-2}$.

Boucles : compléments

Quelques abréviations utiles :

- «utilisée seule», l'expression `i++` est équivalente à `i = i + 1`
- «utilisée seule», l'expression `i--` est équivalente à `i = i - 1`

Il est par ailleurs possible de déclarer la variable de contrôle «à la volée» dans l'expression de contrôle de la boucle.

Dans ce cas la portée de la variable est limitée à la boucle (elle n'a de sens et n'est utilisable que dans la boucle).

```
int n = 10;
for (int i = 1; i < n; i++) {
    Deug.println("J'ai collé " + i + " timbre(s).");
    Deug.println("Il m'en reste " + (n - i) + ".");
}
// À partir d'ici la variable i est inconnue
```

Quelle est la toute dernière ligne affichée dans l'exemple précédent ?

Des exemples non numériques

La classe Deug permet la réalisation de graphismes.

Pour cela, elle contient les méthodes suivantes :
- `void startDrawings(int largeur,int hauteur)`, qui ouvre la zone graphique et qu'il faut appeler avant de commencer à dessiner ;
- `void drawPoint(int x, int y)`, qui affiche un point aux coordonnées $(\mathtt{x}, \mathtt{y})$;
- `void clearArea()`, qui efface le dessin.
- `void stopDrawings()`, qui ferme la zone graphique.

Attention : le coin supérieur gauche a pour coordonnées $(0, 0)$, et l'axe des ordonnées est dirigé vers le bas. C'est l'inverse de la convention utilisée en mathématiques.

Exercices : Écrire des programmes qui dessinent

1. un segment reliant les points de coordonnées $(100, 100)$ et $(200, 100)$;
2. un carré de taille et position données par l'utilisateur ;
3. un segment reliant les points de coordonnées $(100, 100)$ et $(200, 200)$;
4. un cercle de centre et rayon donnés par l'utilisateur.

Les tableaux

Il est parfois nécessaire de conserver en mémoire plusieurs données de même type, par exemple pour conserver une valeur calculée à chaque itération d'une boucle.

Cela nécessite de pouvoir écrire un programme dans lequel le nombre de variables disponibles est lui-même un paramètre — une suite finie (mais de longueur variable) de valeurs du même type.

Une telle construction s'appelle un *tableau*.

En Java, le type «tableau d'éléments de type T» se note `T[]`.

Les éléments d'un tableau `t` de taille n sont numérotés de 0 à $n-1$, et sont désignés par
`t[0]`, `t[1]`,. . . `t[n - 1]`.

Remarque : on a en a déjà rencontré un exemple dans la définition de la fonction `main` d'un programme dont le paramètre est un tableau sur le type `String` (il contient la liste des paramètres donnés au lancement du programme).

Le fragment de programme suivant déclare une variable `t` de type «tableau d'entiers», puis il alloue un tableau de 5 entiers (opérateur `new`) et l'affecte à la variable `t` :

```
int[] t;   // l'écriture int t[] est admise
t = new int[5];
```

Comme nous le verrons plus loin, l'affectation réalisée ne recopie pas les données : elle permet de mémoriser dans la variable `t` l'emplacement des éléments du tableau en mémoire et donc d'y accéder.

Les différents entiers du tableau sont ici 5 entiers, désignés par `t[0]`, `t[1]`,. . . `t[4]`.

On peut obtenir la longueur du tableau correspondant à la variable `t` à l'aide de l'expression `t.length`.

Les éléments d'un tableau `t` sont donc
`t[0]`, `t[1]`,. . . `t[t.length - 1]`.

Les tableaux (suite)

Exemple : lire un entier n, puis lire n entiers et les afficher dans l'ordre inverse de celui dans lequel ils ont été lus.

Séquence de code Java correspondante :

```
int n;
n = Deug.readInt();
int[] t = new int[n];
for (int i = 0; i < t.length; i++) {
    Deug.print("Donnez-moi l'entier " + i + " > ");
    t[i] = Deug.readInt();
}
Deug.println("Voici la liste inversée: ");
for (int i = 1; i <= t.length; i++) {
    Deug.print(t[t.length - i]);
    Deug.print(' ');
}
Deug.println();
```

Exercice : écrire un programme qui affiche les paramètres avec lesquels il a été lancé à raison de un paramètre par ligne.

Pas arbitraires dans une boucle

La dernière boucle du programme précédent peut s'écrire de la manière suivante :

```
for (int i = t.length - 1; i >= 0; i--) {
    Deug.print(t[i]);
    Deug.print(' ');
}
```

Dans cet exemple, on fait évoluer la variable de contrôle en descendant (c'est-à-dire en faisant décroître sa valeur).

Dans une boucle for, on peut, à chaque itération de la boucle, ajouter à la variable de contrôle une valeur quelconque, même négative.

```
pour tous les i de n1 à n2 par pas de c faire
    quelquechose(i)
```

Avec un pas positif, on écrira :

```
for (i = n1; i <= n2; i += c)
    quelquechose(i);
```

et avec un pas négatif, on écrira :

```
for (i = n1; i >= n2; i -= c)  // c est positif
    quelquechose(i);
```


Le schéma de boucle «tant que»

Le schéma de boucle *tant que* permet de répéter l'exécution d'une séquence d'instructions tant qu'une condition donnée est satisfaite.

Exemple : pour un entier K donné, afficher le plus petit n tel que $n^2 \geq K$.

On peut résoudre ce problème de la manière suivante :

```
n = 0;
tant que n * n < K
    n = n + 1
afficher n
```

Contrairement à ce qui se passait avec le schéma de boucle *pour* utilisé précédemment, le nombre d'itérations à réaliser n'est pas a priori connu, pas nécessairement facile à calculer, voire non calculable.

La boucle «while» de Java

En Java, la boucle *tant que* a la forme suivante :

```
while (condition)
    instruction ou bloc d'instructions
```

La condition est une expression booléenne qui est évaluée au début de chaque itération et sert à déterminer si l'itération doit être poursuivie ou non.

Exemple :

```
int n = 0;
while(n * n < K)
    n++;
Deug.println(n);
```


Exercices

1. Écrire une fonction qui calcule le plus petit n tel que la somme des n premiers nombres impairs est plus grande qu'une constante K donnée.
2. Écrire une fonction qui calcule le plus petit n tel que le terme a_n de la suite de Syracuse(p) (avec $p > 1$ donné) est égal à 1.

   La suite de Syracuse(p) est définie par :

$$\begin{aligned} a_0 &= p; \\ a_{n+1} &= \frac{a_n}{2} && \text{si } a_n \text{ est pair;} \\ a_{n+1} &= 3a_n + 1 && \text{si } a_n \text{ est impair.} \end{aligned}$$

La boucle «do ... while» de Java

La boucle `while` évalue la condition de contrôle de la boucle au début de chaque itération : le corps n'est donc jamais exécuté si cette expression est fausse avant l'entrée de la boucle.

La boucle `do...while` est semblable à la boucle `while`, mais elle évalue son expression de contrôle à la fin de chaque itération. Le corps est donc toujours exécuté au moins une fois.

La boucle `do...while` est utile lorsque la condition de contrôle dépend de données calculées dans le corps de la boucle, et n'a donc pas de sens avant d'avoir effectué la première itération.

Exemple : un petit menu

```
do {
    Deug.println("1) commander une pizza");
    Deug.println("2) commander des chaussures");
    Deug.println("T) Terminer la commande");
    c = Deug.readChar();
    if (c == '1') ...
    else if (c == '2') ...
} while (c != 'T');
```


Exercices

1. Écrire un programme qui demande à l'utilisateur d'entrer une liste d'entiers terminée par l'entier 0 et affiche leur somme. Quel résultat fournit votre programme si on remplace l'addition par une multiplication ?
2. Écrire un programme qui commence à lire un premier nombre n, puis lit des nombres jusqu'à ce que la somme des nombres lus soit au moins égale à n.

La rupture

Il est parfois nécessaire d'interrompre l'exécution d'une boucle avant la fin «normale» de l'itération (s'échapper de la boucle).

Problème : étant donné un tableau `t` et une valeur `v`, trouver le plus petit entier `i` tel que `t[i]` soit égal à `v`.

Solution élémentaire :

```
int[] t;
int i, valeur = 5;
...   // allocation d'un tableau, initialisation de t
for (i = 0; i < t.length; i++) {
    if (t[i] == valeur) {
        Deug.println("Trouvé à l'indice " + i);
        System.exit(); // terminaison du programme
    }
}
Deug.println("Valeur non trouvée.");
```


La rupture (suite)

La solution précédente a le défaut de causer la fin de l'exécution du programme tout entier dès que la valeur a été trouvée dans le tableau. On peut contourner le problème en utilisant une fonction réalisant la boucle dont on sortira à l'aide de l'instruction `return` renvoyant l'indice recherché :

```
static int cherche(int[] t, int valeur) {
    for(int i = 0; i < t.length; i++) {
        if(t[i] == valeur)
            return i;
    }
    return -1;  // la valeur n'a pas été trouvée
}

public static int main(String[] args) {
    int[] t;
    int i, valeur = 5;
    ...    // allocation, initialisation de t
    i = cherche(t, valeur);
    if (i >= 0)
        Deug.println("Trouvé à l'indice " + i);
    else
        Deug.println("Valeur non trouvée.");
}
```

La rupture (suite)

Une troisième solution consiste à utiliser une nouvelle variable j pour stocker le résultat, et un «drapeau» trouve pour indiquer si la solution est valide :

```
int[] t;
int i, valeur = 5, j = 0;
boolean trouve = false;   // le drapeau
...
for (i = 0; i < t.length; i++) {
    if (!trouve)   // on n'a pas encore trouvé la valeur
        if (t[i] == valeur) {
            trouve = true;  // on a trouvé la valeur
            j = i;   // on mémorise l'indice
        }
}
if (trouve)
    Deug.println("Trouvé à l'indice " + j);
else
    Deug.println("Valeur non trouvée.");
```

Cette solution présente cependant l'inconvénient de parcourir tout le tableau même si la valeur a été trouvée dès le début.

La rupture : «break»

L'instruction break permet d'interrompre l'exécution d'une boucle sans pour autant interrompre l'exécution du programme ou celle de la fonction courante. Cette interruption peut être vue comme un «saut» à l'instruction qui suit la boucle :

```
int[] t;
int i, valeur = 5;
...
for (i = 0; i < t.length; i++) {
    if (t[i] == valeur) {
        break;
    }
}
if (i < t.length)
    Deug.println("Trouvé à l'indice " + i);
else
    Deug.println("Valeur non trouvée.");
```

L'instruction break n'interrompt que l'exécution de la boucle dans le corps de laquelle elle se trouve : elle n'a aucune action sur d'éventuelles boucles externes. Lorsqu'il faut interrompre plusieurs boucles imbriquées, il faut utiliser la technique du «drapeau» présentée précédemment.

Exercices

1. Écrire un programme permettant de déterminer si un nombre donné est premier ou non.
2. Écrire un programme qui lit un entier n et détermine s'il est présent dans la table de multiplication des nombres de 1 à 10.
3. Écrire un programme équivalent à celui du transparent précédent en utilisant une boucle `while`. Est-il possible d'utiliser la technique du drapeau sans pour autant utiliser l'instruction `break` ?

Le «court-circuit»

Il peut être utile de court-circuiter ou d'abréger l'exécution de l'itération en cours pour passer à la suivante, sans sortir définitivement de la boucle.

L'instruction continue permet de court-circuiter l'itération courante. Elle peut être vue comme un saut vers la fin de l'itération courante.

Exemple : compter les lignes et les caractères qui ne sont ni un caractère de fin de ligne, ni un espace, ni un caractère de tabulation.

```
int lignes = 0, car = 0;
 ....
while(true) {
    c = Deug.readChar();
    if (!Deug.isOk()) break;
    /* si caractère ni espace, ni tabulation */
    /* non traité et lire un nouveau caractère */
    if (c ==' ' || c == '\t') continue;
    /* si fin de ligne, augmenter compteur */
    if (c == '\n') {
        lignes++;
        continue;
    }
    car ++; // un caractère de plus
}
```

Équivalence entre itérations

Une boucle `for` peut être traduite en une boucle `while` :

```
for (e1; e2; e3)
    instruction ou bloc d'instructions
```

est équivalent à

```
e1;
while (e2) {
    instruction ou bloc d'instructions
    e3;
}
```

Cette équivalence n'est cependant correcte que dans le cas où on n'utilise pas l'instruction `continue`.
Pourquoi ?

On peut donc voir l'instruction `for` comme codifiant un cas d'usage habituel de l'instruction `while` : c'est analogue à l'intruction `switch`, qui codifie un usage de la conditionnelle `if`.