Yann.Secq@univ-lille.fr
Fabien.Delecroix@univ-lille.fr

TD n°6

Objectifs : Pratique des structures de contrôle imbriquées. Réaliser des fonctions et des fonctions de test sur celles-ci. Comprendre la notion de portée de variable : locale et globale.

1 Structures de contrôles imbriquées

Exercice 1 : Traces d'exécution

Voici trois algorithmes exploitant des imbrications de structures de contrôle.

```
1 print("A");
2 int i = 0;
3
4 while (i < 5) {
5   if ((i % 2) == 0) {
6     print("C");
7   }
8   i = i+1;
9 }
```

```
1  boolean fini = false;
2  String s = "abc.defg";
3  int i = length(s);
4
5  while ((i > 0) &&
6            ! fini) {
7    i = i-1;
8    print(charAt(s, i));
9    if (charAt(s, i) == '.') {
10     fini = true;
11   }
12 }
```

```
1 for (int i=1; i<5; i=i+1) {
2   for (int j=5; j <= 7; j=j+1) {
3     print(i*10 + j);
4   }
5 }
```

1. Dessinez les organigrammes de ces trois algorithmes.
2. En utilisant les organigrammes, faites la trace de ces algorithmes pour déterminer les affichages produits.

2 Fonctions et tests

Maintenant que nous pouvons définir nos propres fonctions pour enrichir le langage disponible pour résoudre des problèmes, il serait intéressant de pouvoir automatiser la vérification des programmes que nous écrivons. En effet, pour l'instant, nous testons nos programmes manuellement, en exécutant notre programme plusieurs fois et en saisissant différentes valeurs : c'est chronophage et peu rigoureux.

Afin d'améliorer la confiance que l'on peut avoir dans le code que l'on écrit, l'introduction de tests automatisés est un élément du génie logiciel qui s'est développé depuis plusieurs années dans l'industrie du logiciel. Le principe consiste à appeler automatiquement la fonction que l'on souhaite vérifier avec différentes données, dont on connaît les résultats devant être produit par la fonction. Pour vérifier que la fonction produit le bon résultat par rapport aux paramètres effectifs transmis, on utilise **la notion d'assertion**. Une assertion vérifie qu'une propriété est vérifiée ... ou pas ! Si la propriété est vérifiée, rien ne se passe, si ce n'est pas le cas, l'échec est mémorisé et affiché en fin d'exécution.

Pour l'instant, nous n'utiliserons qu'une seule assertion :

— `void assertEquals(<type> valeurAttentue, <type> appelDeLaFonctionTestée)` : vérifie que la valeur retournée par la fonction testée est bien égale à la `valeurAttendue`.

Les assertions doivent être définies dans des fonctions dont le nom débute par `test`, c'est ainsi qu'`iJava` détecte que vous souhaitez tester des fonctions. Cependant, pour que les tests soient lancés et pas le programme principal, il est nécessaire de renommer `void algorithm()`, typiquement en préfixant juste le nom de la fonction avec un tiret bas ainsi `void _algorithm()`. En effet, par défaut `iJava` exécute la fonction `void algorithm()` si elle est présente dans un programme. Par contre, si la fonction n'est pas présente, `iJava` recherche les fonctions dont le nom débute par le préfixe `text` et les exécutent, puis synthétise les assertions valides et invalides.

Exercice 2 : Une première fonction de test

Le but de cet exercice est d'écrire une fonction qui calcule le montant d'une commande d'ordinateurs. En supposant que l'on dispose en entrée du nombre d'ordinateurs et du prix d'un ordinateur, on calcule la valeur totale de la commande.

Le tableau ci-dessous donne 3 exemples d'exécution de ce programme avec trois entrées différentes:

	Entrée(s)	Sortie(s)
Première exécution	30, 749.95	22498.5
Deuxième exécution	10, 1225.5	12255.0
Troisième exécution	20, 1000.0	20000.0

1. Combien y-a-t-il d'entrées dans cet algorithme ? Quel est leur type ?
2. Combien y-a-t-il d'informations et quel est leur type ?
3. Dans le programme ci-dessous, on a fourni une fonction de test qui va s'assurer que la fonction `montant` produira les résultats attendus sur le jeu de tests fourni.

   Complétez le programme ci-dessous afin de calculer le montant d'une commande en fonction du nombre d'ordinateurs et du prix unitaire.

```
class MontantCommande extends Program {
  void testMontantTotal() {
    assertEquals(22498.5, montant(30, 749.95));
    assertEquals(12255.0, montant(10, 1225.5));
    assertEquals(20000.0, montant(20, 1000.0));
  }
  ... montant(... nombreOrdis, ... prixUnitaire) {
    // Déclaration des variables
    ...
    // Calcul de la valeur de la commande
    ...
    // Résultat de la fonction
    return ...;
  }
}
```

Exercice 3 : Calcul de l'âge d'une personne

Écrire une fonction qui permet de calculer l'âge d'une personne, à partir de son année de naissance et de l'année en cours. La fonction à écrire doit passer le test ci-dessous.

```
class CalculAge extends Program {
   void testAge () {
      assertEquals(19, age(1999, 2018));
      assertEquals(40, age(1978, 2018));
   }
   .....
}
```

3 Portée des variables

Au début du module, nous définissions toujours les variables au tout début de la fonction `void algorithm()`. Avec notre exploration des répétitions et plus précisément de la structure de contrôle des boucles à compteur (`for`), nous avons avons découvert qu'une variable existait dans le bloc dans lequel elle est définie. Ainsi, le compteur de boucle n'ayant de raison d'être que lorsque la boucle s'exécute, il est déclaré dans la structure de contrôle et n'existe ni avant, ni après la boucle. C'est ce que l'on appelle une **variable locale** à la boucle.

Avec l'usage de plusieurs fonctions, nous avons découvert que les variables déclarées dans les fonctions sont aussi des variables locales à ces dernières. Si l'on généralise, **la portée d'une variable, c'est-à-dire l'endroit où il est possible de l'utiliser, correspond au bloc dans lequel elle est déclarée**. Ainsi, si vous déclarez une variable dans la première branche d'une alternative, elle n'existera que dans cette branche, alors que si vous la déclarez au début d'une fonction, elle existera pour l'ensemble de la fonction.

Finalement, nous avons vu qu'il était possible de déclarer une variable (ou constante) en dehors de toute fonction ! Cette dernière étant déclarée dans le bloc définissant le programme, c'est-à-dire l'ensemble des fonctions, ces variables sont accessibles pour l'ensemble des fonctions. C'est que l'on appelle **une variable globale**. Même si cela peut sembler pratique, **IL FAUT ABSOLUMENT LIMITER L'USAGE DES VARIABLES GLOBALES DANS VOS PROGRAMMES** car cela les rend bien plus difficiles à déboguer.

Voici un exemple illustrant des variables ayant différentes portées :

```
1  class PorteeDesVariables extends Program {
2    final String HUM  = "Humanoide"; // Constante globale
3    final String PROG = "Programme"; // Constante globale
4    int numeroteuse = 0; // Variable globale
5
6    void accueillir(boolean hum) {
7      String nom;
8      numeroteuse = numeroteuse+1;
9      if (hum){
10        nom = HUM;
11     }else{
12        nom = PROG;
13     }
14     println("Bonjour␣" + nom + "n" + numeroteuse);
15   }
16
17   void algorithm(){
18        do{
19            accueillir(random() > 0.5);
20        } while (numeroteuse < 10);
21        println("J'ai␣accueilli␣" + numeroteuse
22        + "␣joueurs␣(" + HUM + "s␣et␣" + PROG + "s␣confondus)");
23
24        do{
25            accueillir(false);
26        } while (numeroteuse < 100);
27        println("J'ai␣accueilli␣" + numeroteuse
28        + "␣joueurs␣(" + HUM + "s␣et␣" + PROG + "s␣confondus)");
29   }
30 }
```

On identifie ici 3 variables globales, ou plus précisément, deux constantes globales (`HUM` et `PROG`) et une variable globale `numeroteuse`. Ces variables étant déclarée au sein du bloc définissant le programme, elles sont accessibles depuis toutes les fonctions (et sont d'ailleurs utilisées à la fois dans `accueillir` et `algorithm`). Par contre, la variable `nom` est locale à la fonction `accueillir`.

Exercice 4 : Utilisation d'une variable globale [O]

n souahite dans cet exercice compter le nombre de fois qu'une certaine opération se réalise au sein de différentes fonctions. Pour cela, nous allons avoir besoin d'une variable globale afin de mettre à jour ce compteur.

— Écrire une fonction `nbOccurences` qui calcule le nombre d'occurences d'un caractère dans une chaîne de caractères.

— Écrire une fonction `testNbOccurences` qui teste la fonction précédente.

— Écrire une fonction `estPresent` qui vérifie si un caractère est présent dans une chaîne de caractères et retourne son indice si c'est le cas et -1 sinon.
— Écrire une fonction `testEstPresent` qui teste la fonction précédente.
— Écrire la fonction `void algorithm()` qui appelle ces deux fonctions et détermine le nombre d'opérations d'égalité réalisées lors de l'appel de chacune de ces fonctions avec ces données : `"L'algorithmique, c'est fantastique !", 'f'`.