51IF1IF1
Introduction à l'informatique et à la programmation

Support du cours-td

2013-2014

Table des matières

Présentation de l'enseignement **2**

1 Présentation du langage Java **3**
1.1 Qu'est-ce-qu'un programme ? À quoi ça sert ? 3
1.2 Exemples de programmes Java 3
1.3 Erreurs 5
1.4 Structure d'un programme Java simple 6
1.5 Identificateurs 6
1.6 Comment fabrique-t-on un programme ? 7

2 Variables **9**
2.1 Types 9
2.2 Variables et affectations 11

3 Méthodes et fonctions **14**
3.1 Structure générale d'une fonction 14
3.2 Paramètres des fonctions 15
3.3 Valeur renvoyée 16
3.4 Appel/invocation de fonction 17
3.5 Fonctions récursives 20

4 Structures de contrôle **22**
4.1 Structures conditionnelles 22
4.2 Boucles 29

5 Tableaux **38**
5.1 Tableaux à une dimension 38
5.2 Création d'un tableau à l'intérieur d'une fonction 40
5.3 Modification d'un tableau par une fonction 42
5.4 Tableaux à plusieurs dimensions 45

6 Types et Expressions **49**
6.1 Types primitifs et types références 49
6.2 Expressions 50

Présentation de l'enseignement

Objectifs et organisation

Cet enseignement a plusieurs objectifs : comprendre un certain nombre des concepts généraux des machines et de la programmation, réaliser le codage effectif d'algorithmes simples, compiler et exécuter des programmes dans un environnement de type Unix. Voici son organisation :

Cours le jeudi de 14h30 à 16h30 en amphi 1A

	septembre			octobre			novembre				décembre
MATH+MASS	12	19		17		31		14		28	
INFO+MI+CPEI+LI	*09*		26		24		07		21		05

Cours-TD un créneau de deux heures hebdomadaire

TP un créneau de trois heures hebdomadaire

En complément, un système de soutien est mis en place :

Tutorat le midi du lundi au vendredi au SCRIPT (fin septembre début octobre)

Commission de suivi sur rendez-vous au département sciences exactes

Contrôle de connaissances

L'évaluation se fait au cours de tests (sur papier et/ou machine) et d'examens (sur papier) :

- **Td** résultat des tests effectués en cours-TD
- **Tp** résultat des tests effectués en TP
- **E0** partiel (samedi 26 octobre de 13h à 15h, à confirmer)
- **E1** examen session 1 en décembre
- **E2** examen session 2 en juin

Les notes finales sont calculées selon le principe suivant :

Contrôle continu	**Cc** = (**Td** + **Tp**) / 2
Note d'écrit	**Ne** = max (**E1**, (**E0** + **E1**) / 2)

Note session 1	(3**Ne** + **Cc**) / 4
Note session 2	max (**E2**, (3**E2** + **Cc**) / 4)

La présence à au moins un test en cours-TD et un test en TP est nécessaire au calcul de **Cc**.
Attention : pas de note ⇒ pas de moyenne ⇒ pas de compensation.

Bibliographie

[1] Alfred Aho and Jeffrey Ullman, *Concepts fondamentaux de l'informatique*, Dunod, Paris, 1993, (cote bibliothèque : 004 AHO).

[2] Mary Campione, Kathy Walrath, and Alison Hulm, *The java tutorial (fourth edition)*, Addison-Wesley, 2000, `http://java.sun.com/docs/books/tutorial/`.

[3] Bruce Eckel, *Thinking in java*, `http://www.mindview.net/Books/TIJ/` ou `http://penserenjava.free.fr/`.

[4] Yakov Fain, *Programmation java pour les enfants, les parents et les grands-parents*, 2005, `http://java.developpez.com/livres/javaEnfants/`.

Chapitre 1

Présentation du langage Java

1.1 Qu'est-ce-qu'un programme ? À quoi ça sert ?

Extraits de wikipedia :

> En informatique, un programme est une suite d'opérations pré-déterminées destinées à être exécutées de manière automatique par un appareil informatique en vue d'effectuer des travaux, des calculs arithmétiques ou logiques, ou simuler un déroulement.
>
> Le *code source* [1] est la forme sous laquelle un programme est créé par un programmeur : un ensemble de textes décrivant formellement, étape par étape, les opérations à effectuer ; rédigés conformément aux règles d'un langage de programmation.
>
> [http://fr.wikipedia.org/wiki/Programme_(informatique)]

> La particularité principale de Java est que les logiciels écrits dans ce langage sont très facilement portables sur plusieurs systèmes d'exploitation [...] avec peu ou pas de modifications. [...] L'indépendance vis-à-vis de la plate-forme, signifie que les programmes écrits en Java fonctionnent de manière parfaitement similaire sur différentes architectures matérielles. On peut effectuer le développement sur une architecture donnée et faire tourner l'application sur toutes les autres.
>
> Ce résultat est obtenu par les compilateurs Java qui compilent le code source "à moitié" afin d'obtenir un *bytecode* (plus précisément le bytecode Java, un langage machine spécifique à la plate-forme Java). Le code est ensuite interprété sur une machine virtuelle Java (JVM en anglais), un programme écrit spécifiquement pour la machine cible qui interprète et exécute le bytecode Java. [http://fr.wikipedia.org/wiki/Java_(langage)]

Parler de l'exécution d'un programme Java est donc un abus de langage : c'est la machine virtuelle Java qui s'exécute ; on lui fournit un argument qui désigne le bytecode correspondant au programme.

1.2 Exemples de programmes Java

```
1 /** affiche Hello World!
   */
3 class Hello {
    public static void main(String[ ] args) {
5     System.out.println("Hello␣World!");
    } /* fin de la definition de main */
7 } // fin de la definition de Hello
```

Hello.java

1. On pourra parler de code ou de source pour désigner le code source. (NdA)

La compilation d'un code `Java` s'effectue avec la commande `javac` (*java compiler*) suivie du nom du fichier contenant le source qui porte le suffixe `.java`. Cette commande transforme le code qui est un texte lisible par un être humain en *bytecode* lisible par la machine virtuelle `Java`. Au passage le compilateur vérifie que la syntaxe du langage est respectée, par exemple qu'une opération effectuée sur une variable est compatible avec le type (déclaré) de celle-ci, par exemple on ne peut pas affecter la valeur `3.5` à un entier. Le compilateur ne connaît pas les valeurs des variables.
Un programme `Java` s'exécute grâce à la commande `java` suivie du nom de la classe à exécuter (celle qui contient la fonction `main` avec les bonnes spécifications). Durant l'exécution, la machine virtuelle suit dans l'ordre les instructions du `main`.
Par exemple, le programme suivant demande à l'utilisateur de donner un nombre et affiche sa partie entière.

```java
import java.util.Scanner;

class PartieEntiere{
    public static void main(String[] args){
        double x;
        Scanner sc = new Scanner(System.in);
        System.out.print("Donnez␣un␣nombre␣quelconque␣:␣");
        if(sc.hasNextDouble()){
            x = sc.nextDouble();
            System.out.println("[" + x + "]␣=␣" + partieEntiere(x));
        }
    }

    /** renvoie la partie entiere d'un reel
     * @param x
     * @return [x]
     */
    static int partieEntiere(double x){
        int e = 0;
        if(x >= 0){
            while(e <= x) e++;
            return e - 1;
        } else {
            while(e > x) e--;
            return e;
        }
    }
}
```

PartieEntiere.java

► **Exercice 1** (Examen 2006/2007, exercice 2, question 2.1) :
Soit le code Java suivant :

```java
class Exam2 {
  public static void main(String[] args){
    int v1 = 3, v2 = 6;
    if (v1+2 < v2) System.out.println("ric");
    if (v1+4 < v2) System.out.println("hoc");
    if (v1+4 > v2) System.out.println("het");
  }
}
```

Quel doit être le nom du fichier contenant ce code source, quelle commande permet de le compiler, quel est le nom du fichier *bytecode* obtenu et comment réalise-t-on l'exécution ?

1.3 Erreurs

1.3.1 Erreurs à la compilation

Si le fichier comporte des erreurs de syntaxe (c'est-à-dire s'il ne respecte pas les spécifications de `Java`), une erreur se produit à la compilation et aucun fichier `.class` n'est créé.

Exemples d'erreurs pouvant se produire :
- le `;` manque à la fin d'une instruction,
- un identificateur ne peut être interprété,
- il y a incohérence dans les paires d'accolades.

Attention : une compilation qui se passe bien ne signifie pas que votre programme est correct, ni même qu'il s'exécutera sans erreur.

1.3.2 Erreurs à l'exécution

Des erreurs peuvent se produire à l'exécution.
Par exemple dans le programme ci-dessous, on effectue une division par 0. Le compilateur ne peut pas la détecter car il ne connaît pas la valeur d'une variable mais juste son type.

```
class DivisionParZero{
    public static void main(String[] args){
        int c, a=3, b=0;
        System.out.println("debut");
        c = a/b; // erreur a l'execution
        System.out.println("fin");
    }
}
```

DivisionParZero.java

L'exécution s'arrête au moment de l'erreur.
Le message d'erreur est important car il permet de tracer l'erreur depuis son origine. On modifie la classe précédente :

```
class DivisionParZero2{
    static void diviserParZero(){
        int c, a=3, b=0;
        System.out.println("debut");
        c = a/b; // erreur a l'execution
        System.out.println("fin");
    }

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

DivisionParZero2.java

Son exécution donne le traçage d'erreur suivant :

```
$ java DivisionParZero2
debut
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at DivisionParZero2.diviserParZero(DivisionParZero2.java:5)
        at DivisionParZero2.main(DivisionParZero2.java:10)
```

trace de l'exécution de DivisionParZero2

▶ **Exercice 2** :
Écrire ce que serait le message d'erreur à l'exécution de `DivisionParZero`.

Autre exemple : le source ne contient pas de `main` avec les bonnes spécifications : la machine virtuelle ne sait pas ce qu'elle doit faire.

1.4 Structure d'un programme Java simple

La structure de base d'un programme est la suivante :

```
class ClasseLaPlusSimple {
    public static void main(String[] args){
        // declarations de variables
        // instructions
    }
}
```

ClasseLaPlusSimple.java

Le nom du fichier doit être le nom de la classe suivi du suffixe `.java`. Si la compilation n'amène pas à une erreur, la compilateur produit un fichier bytecode dont le nom est celui de la classe suivi du suffixe `.class`. Pour éviter d'avoir un `main` trop long et illisible, on fabrique des "boîtes noires" en écrivant des méthodes ou fonctions (la différence sera expliquée au second semestre). Celles-ci peuvent être appelées plusieurs fois et à n'importe quel moment dans le programme.

```
// importations

class ClassePlusEvoluee {
    static void uneFonction(){
        // corps de la fonction
    }

    public static void main(String[] args){
        // declarations de variables
        // instructions (appel a uneFonction, etc.)
    }
}
```

ClassePlusEvoluee.java

1.5 Identificateurs

Pour désigner les éléments d'un programme (classes, variables, objets, fonctions, méthodes, constantes), on leur donne un nom, appelé *identificateur*. Cet identificateur commence par une lettre (le caractère souligné _ est considéré comme une lettre) qui est suivie de lettres ou de chiffres ; il a une longueur quelconque.
Les minuscules et majuscules sont des lettres disctinctes.

Exemple 1. `MaClasse`, `AutreClasse`, `uneMethode`, `uneAutreMethode`, `x`, `X`, `nbCotes`

Il existe des conventions de nommage en Java. Ne pas les respecter n'empêche pas la compilation, ni l'exécution, mais rend la relecture et le travail à plusieurs plus difficiles. Parmi ces conventions :
– les noms de classes commencent par des majuscules,
– les noms de méthodes/fonctions et de variables commencent par des minuscules,
– quand on "colle" plusieurs mots, chaque initiale (sauf éventuellement pour le premier mot) est une majuscule : `ClasseQuiRepresenteQuelqueChose`, `methodeQuiCalculeQuelqueChose`.

De même, utiliser un nom pas trop long mais qui permette d'identifier le rôle d'un élément facilite le travail.

1.6 Comment fabrique-t-on un programme ?

On commence par écrire un algorithme en français :
– pas de contrainte de syntaxe,
– niveau d'abstraction adapté au problème et aux connaissances.

Pour des programmes écrits dans cette syntaxe abstraite, on parle de langage algorithmique abstrait.
Par exemple, si l'on souhaite écrire un programme qui demande à l'utilisateur de fournir un entier n et qui ensuite calcule la factorielle de n et affiche le résultat, le programme en langage algorithmique abstrait pourra avoir la forme suivante :

```
Lire un entier et le mettre dans n
Si n>=0 alors
  res <- 1
  Si n>0 alors
    Pour i allant de 1 a n
      res <- res*i
    FinPour
  FinSi
FinSi
Afficher res
```

Le programme correspondant en Java s'écrira alors :

```
import java.util.Scanner;

class Factorielle{
    public static void main(String[] args){
        int n,res,i;
        res=0;
        Scanner sc=new Scanner(System.in);
        System.out.print("Donner un nombre entier : ");
        n=sc.nextInt();
        if(n>=0){
            res=1;
            if(n>0){
                for(i=1;i<=n;i++)
                    res=res*i;
            }
        }
        System.out.println("Le resultat est : "+res);
    }
}
```

Factorielle.java

► **Exercice 3** (Examen 2004/2005, exercice 2, questions 2.1.1 et 2.1.2) :

Un diviseur strict d'un entier $n > 0$ est un entier strictement positif, différent de n et qui le divise (cela signifie que le reste de la division entière est nul).

1. Donner une borne supérieure raisonnable du plus grand diviseur strict d'un entier $n > 0$.
2. Écrire en langage algorithmique abstrait un algorithmique qui demande un entier et qui affiche tous ses diviseurs stricts. (Pour se faire, on suppose que l'on dispose de l'opérateur booléen $|$ (divise) tel que, étant donnés deux entiers a et b, $a|b$ est vrai si a divise b et est faux sinon).

► **Exercice 4** (Partiel 2004/2005, exercice 2) :

1. On dispose de trois pièces de monnaie apparemment identiques nommées `a`, `b` et `c` dont on sait qu'au plus une est fausse (elle n'a pas le même poids que les deux autres). On dispose d'une balance permettant de comparer le poids des objets posés sur chacun des deux plateaux. Décrire un algorithme permettant de tester si une des trois pièces est fausse et si oui laquelle (pour alléger l'écriture, on pourra supposer que la valeur d'une variable `x` correspondant à une pièce est égale au poids de cette pièce).
2. Décrire un algorithme pour 4 pièces `a`, `b`, `c` et `d` (au plus une étant fausse) en procédant uniquement par comparaison du poids des pièces (on posera toujours deux pièces sur chaque plateau).

► **Exercice 5** (Partiel 2005/2006, exercice 2, question 2.3) :

On dispose de m pièces de 50 centimes, de n pièces de 20 centimes et de p pièces de 10 centimes (on supposera que p est non nul).
On s'intéresse au paiement de sommes sans rendu de monnaie avec un tel assortiment de pièces.
Exprimer, sous forme algorithmique simple, une stratégie de paiement d'une somme s avec un nombre minimum de pièces pour un assortiment donné de pièces de $50, 20, 10$ centimes si le paiement est possible et qui détecte l'impossibilité de réaliser le paiement.

Indication On prendra par ordre décroissant des valeurs des pièces, le maximum de pièces possibles. On admettra que cette stratégie donne une solution si le paiement est possible (cela en raison de l'hypothèse qu'on dispose d'au moins une pièce de 10 centimes).

Chapitre 2

Variables

2.1 Types

Un *type* correspond à la définition de :
- un ensemble de valeurs,
- un ensemble d'opérations applicables aux éléments de cet ensemble de valeurs et la spécification de comment utiliser ces opérations.

En Java, on distingue les *types primitifs* (par exemple entiers, réels et booléens) et les *types références* (tableaux, *classes*).
Nous définirons les types au fur et à mesure de leur utilisation.

2.1.1 Des types primitifs

Les tests suivants existent pour tous les types primitifs (il s'agit de comparer des éléments de même type) :

tests de comparaison	
`==`	égalité
`!=`	différence

le type booléen `boolean`

valeurs booléennes
`true`
`false`

opérations sur les booléens	
`&&` ou `&`	et
`||` ou `|`	ou
`!`	non
`^`	ou exclusif

un type entier (il en existe d'autres, voir § 6.1) : `int` (codé sur 4 octets)

opérations sur les entiers	
`-`	moins unaire
`+`	addition
`-`	soustraction
`*`	multiplication
`/`	quotient entier
`%`	reste

tests de comparaison	
> (ou >=)	supérieur (ou égal)
< (ou <=)	inférieur (ou égal)

Exemple 2. `5`, `-17`

La multiplication, le quotient entier et le reste sont prioritaires sur l'addition et la soustraction. Le moins unaire est l'opération la plus prioritaire. Quand deux opérations de même priorité apparaissent, c'est celle de gauche qui est effectuée en premier. On peut mettre des parenthèses pour forcer une priorité.

► **Exercice 6** :

Donner la valeur des trois expressions suivantes : `3+5/3` `4*1/4` `2/3*3-2`

► **Exercice 7** (Partiel 2005/2006, exercice 2, questions 2.1 et 2.2) :

On reprend la situation de l'exercice 5 : m pièces de 50, n pièces de 20 et $p \neq 0$ pièces de 10 centimes.

1. Exprimer, sous forme d'une expression arithmétique `Java`, la somme maximale que l'on peut espérer payer avec cet assortiement de pièces.
2. Ne disposant pas de pièces de 1, 2 et 5 centimes, un certain nombre de sommes ne sont pas payables a priori (celles non multiples de 10), en plus de celles supérieures à la somme maximale précédente.

   Exprimer au moyen d'une expression logique, combinant ces conditions, une condition *nécessaire* pour qu'une somme donnée soit *susceptible* d'être payée.

un type réel (il en existe un autre, voir § 6.1) : `double` (codé sur 8 octets)

opérations sur les réels	
`-`	moins unaire
`+`	addition
`-`	soustraction
`*`	multiplication
`/`	division

tests de comparaison	
> (ou >=)	supérieur (ou égal)
< (ou <=)	inférieur (ou égal)

Exemple 3. `3.57`, `-.002`, `12.6e-3`

La multiplication et la division sont prioritaires sur l'addition et la soustraction. Quand deux opérations de même priorité apparaissent, c'est celle de gauche qui est effectuée en premier.

Ordre des types primitifs

Il existe des compatibilités entre les types primitifs, en particulier certaines conversions implicites sont possibles. Ces compatibilités sont définies comme suit :

`int` $\rightarrow$ `double`

Ainsi, une valeur de type `int` peut être vue comme une valeur de type `double`, mais pas le contraire.
Quand on effectue une opération entre deux valeurs qui n'ont pas le même type (mais sont compatibles), le résultat obtenu est du type le plus grand (relativement à l'ordre ci-dessus).

Le type booléen n'est compatible avec aucun autre type.

2.1.2 Les types références

Il s'agit de types contenant souvent plusieurs éléments. Ils apparaissent sous forme de tableaux (plusieurs éléments de même type, voir chapitre 5) ou de classes (plusieurs éléments de types a priori différents). Les classes peuvent être définies dans des paquetages extérieurs ou par le programmeur lui-même.

le type `String` (*chaîne de caractères*) Il sert à coder une suite finie de caractères[1]. Les constantes littérales de ce type sont écrites entre guillemets.

Exemple 4. `"Ceci␣est␣une␣chaine␣de␣caracteres."`
`"Ceci␣est␣egalement\nune␣chaine␣de␣caracteres."`

opération sur les `String`	
`+`	concaténation

► **Exercice 8** (Partiel 2012/2013, exercice 4) :

Préciser quelle erreur empêche la compilation de la classe ci-dessous. Expliquer ce que produirait son exécution après mise en commentaire de la ligne erronée, sauvegarde et compilation.

```
class Soixante17{
    public static void main(String[] args){
        int sept=7, dix=10, soixante=60;
        System.out.println("␣soixante-dix-sept␣␣+␣␣soixante-sept␣");
        System.out.println("␣soixante-dix-sept␣"+  soixante-sept  );
        System.out.println(  soixante-dix-sept  +  soixante-sept  );
        System.out.println(  soixante-dix-sept  +"␣soixante-sept␣");
    }
}
```

2.2 Variables et affectations

Une *variable* correspond à un emplacement en mémoire pouvant contenir une valeur d'un type donné. C'est le type de la variable qui permet l'interprétation correcte du contenu correspondant en mémoire.
Une variable est désignée par un identificateur valide (voir § 1.5). On peut lui affecter une *valeur* du type correspondant ou d'un type compatible.

2.2.1 Déclaration de variables

La première opération à effectuer avec une variable, avant toute utilisation de celle-ci, est sa déclaration : elle permet de lui attribuer un nom et un type.
Une instruction de déclaration prend la forme suivante :
`type nomDeLaVariable;`

Exemple 5. `int monEntier;`
`double x, y;`
`String str;`

Variables de types primitifs Leur déclaration alloue la place en mémoire pour y stocker une valeur du type correspondant.

Variables de types références Leur déclaration alloue la place pour stocker l'adresse en mémoire d'un objet de type correspondant. L'espace nécessaire à stocker l'objet pourra être alloué avec l'opérateur `new` (voir chapitre 5).

2.2.2 Affectation

`nomDeLaVariable = valeurAffectee;`
C'est l'opération qui consiste à attribuer à une variable d'un certain type une valeur compatible avec ce type.
L'affectation d'une valeur à une variable remplace la valeur préalablement associée à cette variable.
Cet opérateur est noté = en `Java`. À la gauche du signe = on trouve toujours le nom d'une variable. À la droite du signe = on trouve une expression (voir chapitre 6 ; en particulier une valeur est une expression) dont le type est compatible avec celui de la variable. La variable prend alors comme nouvelle valeur l'évaluation de cette expression.
On peut déclarer et affecter une variable "en même temps" :

1. Il existe un type primitif caractère `char`, voir § 6.1.

```
int n = 3;
double x = 4.5, y, z = 3;
String s = "chaine";
```

► **Exercice 9** (Partiel 2010/2011, exercice 1) :

Expliquer ce que produit l'exécution du morceau de code Java suivant :

```
int x, y = 7, z = 8;
double s, t;
x = y + 2; System.out.println(x);
y = z - 3; System.out.println(x);
y = y % z; System.out.println(y);
x = x + 4; System.out.println(x);
x = z / y; System.out.println(x);
s = z / y; System.out.println(s);
s = y; t = z / s; System.out.println(t);
```

2.2.3 Constantes

Une *constante* est une variable portant le qualificatif `final`. Elle ne peut être affectée qu'une fois au cours du programme.

```
class DefinitionConstante {
    public static void main(String[] args){
        final double PI = 3.14;
        System.out.println("definition de PI : "+PI);
    }
}
```

DefinitionConstante.java

2.2.4 Portée d'une variable

À partir du moment où une variable existe, elle est visible sur une partie du programme, il est très important de comprendre sur laquelle.

Un *bloc* est un ensemble d'instructions délimité par des accolades (`{ }`). Un bloc peut en contenir un autre. Une variable n'est pas visible avant sa déclaration, ni hors du bloc où elle est déclarée.

```
    static void porteeDesVariables1(){
        int i;
        // instructions avec i autorisees
        // instructions avec j, k et l non autorisees
        int j;
        // instructions avec i et j autorisees
        // instructions avec k et l non autorisees
        {
            // instructions avec i et j autorisees
            // instructions avec k et l non autorisees
```

```
            int k;
            // instructions avec i, j et k autorisees
            // instructions avec l non autorisees
        }
        // instructions avec i et j autorisees
        // instructions avec k et l non autorisees
        int l;
        // instructions avec i, j et l autorisees
        // instructions avec k non autorisees
    }
```

PorteeDesVariables1.java

Le corps d'une fonction (voir § 3.1) est un bloc, les noms de variables sont donc locaux à une fonction.

```
class PorteeDesVariables2 {
    static void deux(){
        int i=2; // variable i visible uniquement dans deux
    }
    static void affichage(){
        System.out.println(i); // erreur: variable non declaree
    }
}
```

PorteeDesVariables2.java

Un nom de variable ne peut pas être réutilisé dans un sous-bloc.

► **Exercice 10** (Partiel 2012/2013, exercice 1, programme 1) :
Expliquer ce que produit l'exécution du programme suivant :

```
class Omfang{
    public static void main(String[] args){
        int a = 3, b = 6+a;
        System.out.println(a+" "+b);
        if(a/b<b/a){
            int c;
            c = b+a; b = b+c; a = b-c;
            System.out.println(a+" "+b+" "+c);
        }else{
            int c;
            c = b-a; b = b-c; a = b+c;
            System.out.println(a+" "+b+" "+c);
        }
        int c;
        c = b+a; b = b+c; a = b-c;
        System.out.println(a+" "+b+" "+c);
    }
}
```

Chapitre 3

Méthodes et fonctions

Une *fonction* est un morceau de code qui permet d'effectuer un ensemble d'opérations par simple appel ailleurs dans le programme ou dans un autre programme. Les fonctions permettent de rendre plus claire la lecture d'un programme et d'écrire une seule fois une suite d'instructions qui pourra être utilisée plusieurs fois.
En Java, les fonctions s'appellent des *méthodes*. On peut toutefois utiliser le terme de fonctions quand elles ont le qualificatif `static`, ce qui sera le cas dans ce cours [1].

Une fois écrite, une fonction peut être vue comme une boîte noire à laquelle on fournit un ensemble de valeurs dont le nombre, l'ordre et les types sont prédéterminés et qui (pour l'instant) :

renvoie une valeur qui pourra être utilisée dans la suite du programme (au *retour* de la fonction),

ou *affiche* un message à l'intention de l'utilisateur du programme,

ou fait les deux (ce qui doit rarement être le cas).

Nous verrons à partir du chapitre 5 qu'une fonction permet également de modifier l'état de la mémoire.

Exemple 6 (Utilisation d'une fonction)**.**

```
class UtilisationSinus{
    public static void main(String[] args){
        double sinPIsur3 = Math.sin(Math.PI/3);
        System.out.println("sin(pi/3)␣=␣" + sinPIsur3);
        System.out.println("sin(pi/4)␣=␣" + Math.sin(Math.PI/4));
    }
}
```

UtilisationSinus.java

3.1 Structure générale d'une fonction

Une fonction se compose de :
- une *en-tête* qui précise le type de sa valeur de retour, son nom et les types et les noms de chacun de ses paramètres (ou arguments).
  On appelle *signature* l'en-tête sans le type de retour, ni les noms des paramètres [2].
- un *corps* de fonction qui est un bloc contenant des instructions,
- des qualificatifs divers (`static` déjà évoqué ou d'autres).

1. Ce terme sera expliqué par la suite dans des cours plus avancés.
2. L'utilité de cette signature apparaît quand on surcharge une fonction.

Exemple 7. La fonction qui prend en paramètres trois entiers `a`, `b` et `c` et renvoie la valeur de `a + b + c` aura pour en-tête :

```
static int somme(int a, int b, int c)
```

Dans cet en-tête :
- `somme` est le nom de la fonction ;
- `int` est le type de la valeur renvoyée par cette fonction. Un appel (ou invocation) de `somme` peut donc figurer dans une expression et sera interprété comme une valeur entière. On peut par exemple écrire dans un programme :
  `int a= 2*somme(4,12,7);`
  Et la valeur `2*(4+12+7)` sera affectée à la variable `a`.
- `a`, `b` et `c` sont les trois paramètres de la fonction ; chacun est de type `int`.

Le corps de la fonction `somme` pourra ensuite avoir la forme suivante :

```
static int somme(int a, int b, int c){
    return a+b+c;
}
```

► **Exercice 11** :

Dire pourquoi aucun des en-têtes suivants n'est correct.

```
static int fonc1(int a ; double b)
static fonc2(double c)
static int, double fonc3(String w)
static void fonc4(String u,v,w)
static boolean fonc5(double)
static String fonc6(String t, double d, int t)
static double 7fonc(double x)
```

► **Exercice 12** :

Donner l'en-tête d'une fonction qui prend en paramètre une chaîne de caractères et renvoie sa longueur.

3.2 Paramètres des fonctions

À l'intérieur du corps de la fonction, les paramètres sont considérés comme des variables
- qui ont le type de la déclaration des paramètres figurant dans l'en-tête de la fonction,
- qui seront initialisées au moment de l'appel de la fonction par les valeurs des expressions correspondantes.

On dit que les paramètres sont *passés par valeur*.
Toutes les autres variables utilisées dans le corps de la fonction doivent y être déclarées (pour l'instant).

Exemple 8. La fonction `func` décrite ci-dessous prend comme paramètres deux entiers et renvoie un réel. Dans le corps de cette fonction, on peut utiliser les variables `a` et `b` comme des variables classiques.

```
static double func(int a, int b){
    // corps de la methode
}
```

Il est possible qu'une fonction ne prenne pas de paramètre en entrée, alors il suffit de n'indiquer aucun paramètre entre les parenthèses suivant le nom de la fonction (mais il faut garder les parenthèses).

Exemple 9. La fonction `helloWord` décrite ci-dessous ne fait qu'afficher le message `Hello World!` dans la console. Elle ne prend aucun paramètre. Par ailleurs, elle ne renvoie aucune valeur (son type de retour est `void`).

```
static void helloWorld(){
    System.out.println("Hello␣World!");
}
```

► **Exercice 13** :

Écrire une fonction `helloWord` qui affiche le message constitué du mot `Hello` suivi de la chaîne de caractères passée en argument.

Il est essentiel de bien comprendre la différence entre une variable et un paramètre :
- Un paramètre sert à transmettre une donnée à une fonction, donnée nécessaire aux calculs effectués par la fonction et que celle-ci ne peut deviner. Par exemple une fonction qui calcule le volume d'un parallèlépipède rectangle a besoin de connaître sa hauteur, sa largeur et sa profondeur pour donner son résultat : elles apparaîtront donc en tant que paramètre.
- Une variable est un emplacement mémoire qu'une fonction utiliser pour faire ses calculs, par exemple si le calcul est complexe ou pour éviter de recalculer plusieurs fois la même chose, sa valeur est calculée et l'utilisateur de la fonction n'a pas besoin de connaître son existence. En reprenant l'exemple précédent, on peut imaginer que la surface du rectangle de base soit calculée et stockée dans une variable, avant d'être multipliée par la hauteur pour obtenir le résultat final.

3.3 Valeur renvoyée

Notons qu'une fonction ne peut renvoyer qu'une seule valeur. Comme le montre l'exemple 7, la valeur renvoyée est indiquée par le mot-clé `return`. Une fois que la fonction a renvoyé une valeur, son exécution s'arrête et l'exécution du programme reprend au niveau de la fonction appelante.
Il se peut qu'une fonction ne renvoie aucune valeur, si par exemple son rôle est d'afficher un message. Dans ce cas, le type de retour de la fonction est `void`.

Exemple 10. La fonction `afficherCarreEtCube` prend en paramètre un entier et affiche la valeur de son carré et de son cube. Elle ne renvoie pas de valeur.

```
static  void afficherCarreEtCube(int n){
    int carre=n*n, cube=carre*n;
    System.out.println("Le␣carre␣vaut␣:␣"+carre);
    System.out.println("Le␣cube␣vaut␣:␣"+cube);
}
```

► **Exercice 14** :

Donner deux exemples de fonctions qui renvoient *quelque chose* et deux exemples de fonctions qui ne renvoient rien. Écrire des instructions utilisant des appels à ces fonctions.

► **Exercice 15** :

La fonction `Math.random()` renvoie un réel de type `double` pseudo-aléatoire de l'intervalle $[0;1[$. Écrire une fonction `booleenAleatoire` qui renvoie une valeur booléenne pseudo-aléatoire.

► **Exercice 16** (Examen 2003/2004, exercice 3, question 3.1) :

Écrire une fonction `distance` qui prend quatre entiers `x0`, `y0`, `x1`, `y1` et qui renvoie la distance, au sens géométrique usuel, entre les points du plan de coordonnées `(x0,y0)` et `(x1,y1)`.
On rappelle que la distance entre les points (x,y) et (x',y') est donnée par la formule

$$\sqrt{(x-x')^2+(y-y')^2}.$$

La fonction qui permet de calculer une racine carrée est `Math.sqrt(double a)`.

3.4 Appel/invocation de fonction

Une fois définie, il est possible d'utiliser la fonction dans le code, on parle alors d'*appel* (ou d'*invocation*) de fonctions.

Exemple 11. Dans cet exemple, la fonction `somme` se trouve dans la même classe (`Somme`) que le `main`. L'appel se fait à la ligne 15. Le programme demande deux entiers à l'utilisateur qu'il met respectivement dans les variables `p` et `q` et il appelle ensuite la fonction `somme` pour calculer la somme de ces deux entiers et l'afficher.

```
import java.util.Scanner;
class Somme{
    static int somme(int a, int b){
        return a+b;
    }
    public static void main(String[] args){
        int p,q;
        Scanner sc=new Scanner(System.in);
        System.out.print("Donner un nombre entier : ");
        p=sc.nextInt();
        System.out.print("Donner un autre nombre entier : ");
        q=sc.nextInt();
        System.out.println("La somme vaut : "+somme(p,q));
    }
}
```

Somme.java

► **Exercice 17** :

– Donner l'en-tête d'une fonction qui prend en paramètre un entier et renvoie la valeur de sa factorielle.
– Ensuite donner l'en-tête d'une fonction qui prend un entier et affiche sa factorielle.
– Écrire deux programmes Java chacun utilisant une fonction définie auparavant et tous deux demandant un entier à l'utilisateur et affichant la valeur de sa factorielle.

Il est également possible d'appeler une fonction au sein d'une autre fonction que la fonction `main`.

Exemple 12. Dans cet exemple, la fonction `somme` renvoie la somme des deux entiers donnés en paramètres et la fonction `sommeTrois` renvoie la somme des trois entiers donnés en paramètres. Cette dernière fonction utilise la fonction `somme`.

```java
import java.util.Scanner;
class SommeTrois{
    static int somme(int a, int b){
        return a+b;
    }
    static int sommeTrois(int a, int b, int c){
        return somme(somme(a,b),c);
    }
    public static void main(String[] args){
        int p=0,q=0,r=0;
        Scanner sc=new Scanner(System.in);
        System.out.print("Donner␣un␣nombre␣entier␣:␣");
        p=sc.nextInt();
        System.out.print("Donner␣un␣deuxieme␣entier␣:␣");
        q=sc.nextInt();
        System.out.print("Donner␣un␣troisieme␣entier␣:␣");
        r=sc.nextInt();
        System.out.println("La␣somme␣vaut␣:␣"+sommeTrois(p,q,r));
    }
}
```

SommeTrois.java

Il se peut que la fonction que l'on souhaite utiliser soit définie dans une autre classe ; alors, lors de l'appel à la fonction, il faut préciser le nom de la classe devant le nom de la fonction invoquée.

Exemple 13. Dans cet exemple, la fonction `produit` est une fonction de la classe `Multiplication` et le `main` de la classe `ProgMultiplication` invoque cette fonction par `Multiplication.produit`.

```java
class Multiplication{
    static int produit(int a, int b){
        return a*b;
    }
}
```

Multiplication.java

```java
import java.util.Scanner;
class ProgMultiplication{
    public static void main(String[] args){
        int p=0,q=0;
        Scanner sc=new Scanner(System.in);
        System.out.print("Donner␣un␣nombre␣entier␣:␣");
        p=sc.nextInt();
        System.out.print("Donner␣un␣deuxieme␣entier␣:␣");
        q=sc.nextInt();
        System.out.println("Le␣produit␣vaut␣:␣" +
                           Multiplication.produit(p,q));
    }
}
```

ProgMultiplication.java

Finalement, rappelons que le passage des paramètres se fait par valeur : cela signifie que dans la fonction appelante la valeur des variables passées en argument de la fonction appelée ne change pas, on ne fait que copier leur valeur au moment de l'appel.

Exemple 14. Dans le programme `Carre`, au moment de l'appel à la fonction `carre`, le programme donne la valeur contenue dans la variable `p` en paramètre et cette valeur est copiée dans la variable `a`. Ainsi la valeur de `p` n'est pas changée par l'appel à la fonction `carre`, ce que l'on peut vérifier lors de l'affichage (ligne 15).

```
import java.util.Scanner;
class Carre{
    static int carre(int a){
        a=a*a;
        return a;
    }

    public static void main(String[] args){
        int p=0,c;
        Scanner sc=new Scanner(System.in);
        System.out.print("Donner un nombre entier : ");
        p=sc.nextInt();
        c=carre(p);
        System.out.println("Le carre de "+p+" vaut : "+c);
    }
}
```

Carre.java

► **Exercice 18** (Examen 2006/2007, exercice 1, question 1.1) :

Exécuter à la main le programme suivant :

```
class Param1 {
  static int fonc1(int a, int b, int c){
    a = a+1; b = b+2; c = a+b; return 2*c;
  }

  public static void main(String[] args){
    int a = 1, b = 2, c = 3, d;
    d = fonc1(a, b, c);
    System.out.println(a); System.out.println(b);
    System.out.println(c); System.out.println(d);
  }
}
```

► **Exercice 19** (Examen 2010/2011, exercice 1) :

Expliquer ce que produit l'exécution du programme suivant :

```
class Kezako {
    static int f(int i){
        System.out.println("f prends " + i);
        return i + g(i*2);
    }

    static int g(int j){
        System.out.println("g prends " + j);
```

```
        return j-1;
10  }

12  public static void main(String[] args){
        int a = f(12);
14      System.out.println("resultat " + a);
        a = f(g(f(3)));
16      System.out.println("resultat " + a);
    }
18 }
```

Kezako.java

► **Exercice 20** :

On veut afficher les paroles de la comptine *Savez-vous planter les choux ?*

1. Écrire les fonctions `refrain` et `couplet` correspondantes.
2. Écrire alors la fonction `comptine`.
3. Proposer d'autres possibilités de fonctions permettant de raccourcir le code.

3.5 Fonctions récursives

Une fonction peut s'appeler elle-même. On parle alors de fonction *récursive*. Il faut bien entendu être sûr que cet appel récursif n'aboutit pas à une boucle d'appels infinie.
La récursivité apporte une réponse simple aux problèmes paramétrés dont la résolution avec un paramètre "plus petit" aboutit facilement à la résolution du problème avec le paramètre courant.
Voici deux exemples de fonctions récursives :

```
class FactorielleRecursive {
2   /**
     * renvoie la factorielle d'un entier
4    * @param n entier (suppose positif ou nul)
     * @return n! (si n>=0)
6    */
    static int factorielle(int n){
8       if(n <= 0)
            return 1;
10      else
            return n*factorielle(n-1);
12  }
}
```

FactorielleRecursive.java

```
1 class PartieEntiereRecursive {
    /** renvoie la partie entiere d'un reel
3    * @param x
     * @return [x]
5    */
    static int partieEntiere(double x){
7       if(x>=0 && x<1)
```

```
                return 0;
 9          else
                if(x > 0)
11                  return 1+partieEntiere(x-1);
                else
13                  return -1+partieEntiere(x+1);
        }
15  }
```

PartieEntiereRecursive.java

Pour éviter les boucles d'appels infinies, il faut tester la valeur des paramètres de la fonction. On reviendra donc en détail sur cette notion à la fin du chapitre 4.

► **Exercice 21** :

Écrire une fonction `doubleFactorielle` récursive qui prend en argument un entier `n` et renvoie sa *double factorielle* définie comme $n \times (n-2) \times \cdots \times 4 \times 2$ pour `n` pair et $n \times (n-2) \times \cdots \times 5 \times 3$ pour `n` impair.

► **Exercice 22** :

Écrire une fonction `multiFactorielle` récursive.

► **Exercice 23** (Partiel 2011/2012, exercice 5) :

Soit `n` un entier compris entre 2 et 9. Un entier positif `p` est appelé `n`-*parasite* si le produit $p \times n$ est égal à la rotation de `p`, c'est-à-dire l'entier dont l'écriture est obtenue à partir de celle de `p` en déplaçant le chiffre des unités au début. Par exemple, `142857` est 5-parasite car `142857×5` vaut `714285`.

1. Écrire une fonction récursive `puissance` qui prend en arguments deux entiers positifs `a` et `b` et renvoie l'entier `a` élevé à la puissance `b`.
2. On définit la fonction récursive `ordre` comme suit :

```
static int ordre(int n, int m){
    if (puissance(10,m)%(10*n-1)==1) return m;
    return ordre(n,m+1);
}
```

   Expliquer ce que produit l'évaluation de `ordre(4,2)`.
3. Écrire une fonction `unParasite` qui prend en argument en entier `n` compris entre 2 et 9 et renvoie l'entier $\frac{10^m - 1}{10n - 1}n$ où `m` est l'entier renvoyé par la fonction `ordre` appelée avec les paramètres `n` et 2. On sait que cet entier est alors `n`-parasite.
4. Écrire une fonction récursive `nbChiffres` qui prend en argument un entier positif `a` et renvoie le nombre de chiffre(s) dans l'écriture de `a`.
5. Écrire une fonction `rotation` qui prend en argument un entier positif `p` et renvoie l'entier dont l'écriture est obtenue à partir de celle de `p` en déplaçant le chiffre des unités au début.
6. Écrire une fonction `estParasite` qui prend en argument un entier `n` compris entre 2 et 9 et un entier positif `p` et teste si `p` est `n`-parasite.

Chapitre 4

Structures de contrôle

Jusqu'à présent le déroulement d'un programme se faisait de façon linéaire, comme illustré en Figure 4.1.

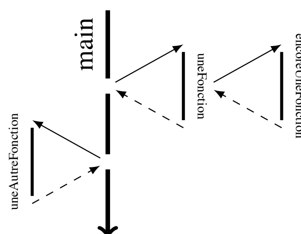


FIGURE 4.1 – Déroulement linéaire d'un programme

Plus généralement, le déroulement de toute fonction se faisait jusqu'à présent de façon linéaire, avec d'éventuels appels à d'autres fonctions. Or parfois on veut qu'un programme ait un comportement si certaines conditions sont réunies et un autre sinon ; ou qu'une suite d'actions se répète un certain nombre de fois. Les structures de contrôle servent à cela.

4.1 Structures conditionnelles

L'objectif des instructions de ce type est de permettre de sélectionner une séquence d'instructions sur la base de la satisfaction de certaines conditions ; c'est-à-dire adapter le comportement du programme aux données.

4.1.1 Si ... alors ...

On souhaite qu'une liste d'instructions ne soit exécutée que si une certaine condition est vérifiée. Cette condition est donnée sous forme d'une expression booléenne [1], c'est-à-dire une expression évaluée comme vrai ou faux. Par exemple, on peut tester si un entier est strictement positif ou si deux entiers sont égaux.
Si l'évaluation de l'expression est *vrai* (`true`) alors on exécute un bloc d'instructions, puis on passe à la suite. Sinon (l'évaluation de l'expression est `false`), on passe directement à la suite.
La syntaxe de cette structure conditionnelle est la suivante :

```
if (expression booléenne E)
  bloc d'instructions B /* execute si l'expression testee est vraie */
```

syntaxe du `if`

Le déroulement d'une conditionnelle `if` suit le schéma de la Figure 4.2.

1. La notion d'*expression* sera abordée plus en détail dans le chapitre 6, pour l'instant on se contente d'une notion intuitive.

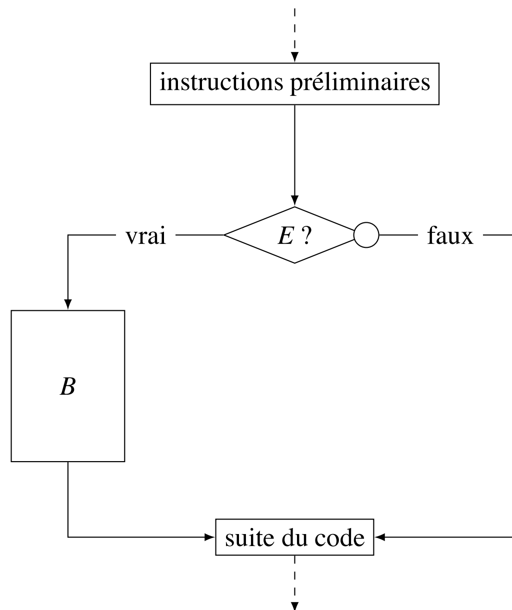


FIGURE 4.2 – `if` dans un programme

Exemple 15. La fonction suivante calcule la valeur absolue d'un réel `x`.

```
    static double valeurAbsolue(double x){
        if(x < 0) x=-x;
        return x;
    }
```

ValeurAbsolue.java

Exemple 16. Le programme suivant affiche le premier des arguments du `main`, s'il y en a.

```
class TraitementArguments{
    public static void main(String[] args){
        String arg0 = "il n'y en a pas";
        if(args.length>0) arg0 = args[0];
        System.out.println("premier argument : " + arg0);
    }
}
```

TraitementArguments.java

4.1.2 Si ... alors ... sinon ...

On teste à nouveau une expression booléenne. Si elle est évaluée à *vrai* alors on exécute un bloc d'instructions, puis on passe à la suite. Sinon, on exécute un autre bloc d'instructions, puis on passe à la suite.
La syntaxe de cette structure conditionnelle est la suivante :

```
if (expression booléenne E)
  bloc d'instructions B1 /* execute si l'expression testee est vraie */
else
  bloc d'instructions B2 /* execute si l'expression testee est fausse */
```

syntaxe du `if ... else`

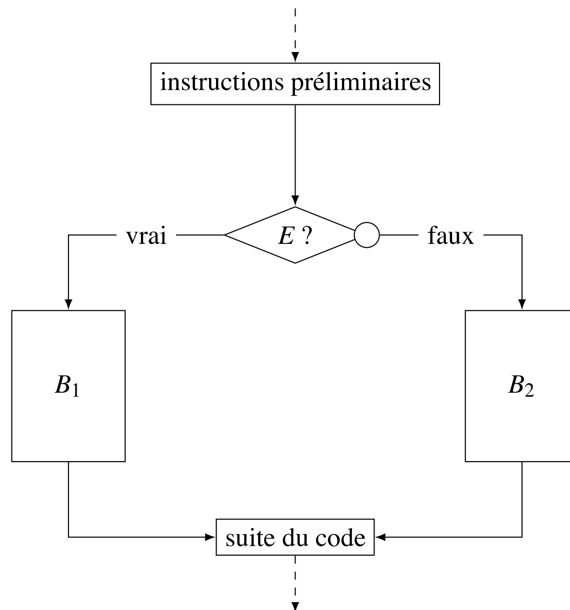


FIGURE 4.3 – `if ... else` dans un programme

Le déroulement d'une conditionnelle `if .. else` suit le schéma de la Figure 4.3.

Exemple 17. La fonction suivante calcule la valeur absolue d'un réel `x`.

```
    static double valeurAbsolue(double x){
2       if(x >= 0)
            return x;
4       else
            return -x;
6   }
```

ValeurAbsolue2.java

Exemple 18. La fonction suivante affiche un message qui dit si le paramètre (de type `char`) est une voyelle ou pas.

```
    static void estVoyelle(char c){
2       if(c=='a' || c=='e' || c=='i' || c=='o'
           || c=='u' || c=='y' || c=='A' || c=='E'
4          || c=='I' || c=='O' || c=='U' || c=='Y')
            System.out.println(c + " est une voyelle.");
6       else
            System.out.println(c + " n'est pas une voyelle");
8   }
```

Voyelles.java

4.1.3 Imbrication de conditionnelles

Les blocs d'instructions d'un `if` et d'un `else` peuvent contenir des instructions quelconques, en particulier des instructions conditionnelles.

Considérons par exemple la séquence suivante d'imbrications :

```
if (condition C₁)
     bloc B₁
else if (condition C₂)
     bloc B₂
...
else if (condition Cₙ)
     bloc Bₙ
else
     bloc Bₙ₊₁
```

Dans ce cas, on exécute la séquence d'instructions B_i pour la plus petite valeur de i pour laquelle la condition C_i est vraie (donc pour tout $j < i$, la condition C_j est fausse). On exécute B_{n+1} si aucune des conditions n'est vérifiée. Le déroulement de cette imbrication d'instructions `if ... else` suit le schéma de la Figure 4.4.

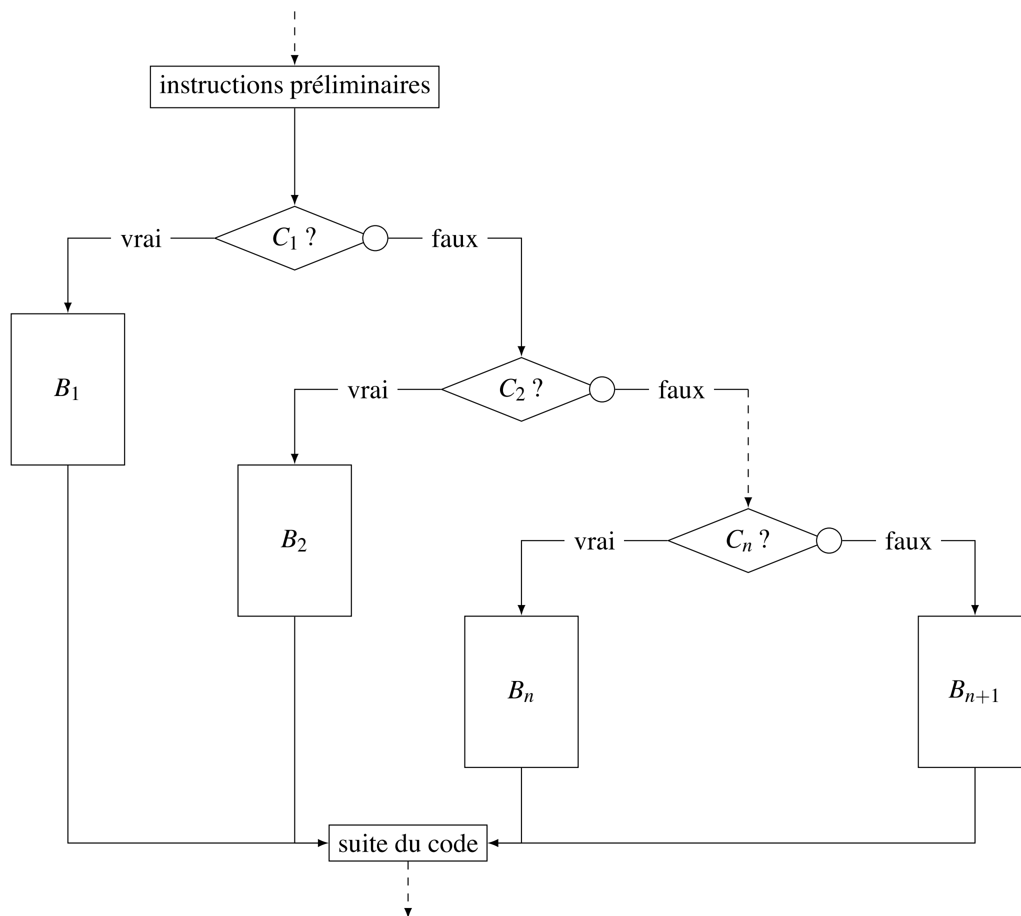


FIGURE 4.4 – `ifs` imbriqués dans un programme

Exemple 19. La fonction suivante affiche un message qui dit si une droite, donnée par deux points (`x1`,`y1`) et (`x2`,`y2`), est verticale, horizontale ou quelconque.

```
    /**
2    * affiche la position (horizontale, verticale ou quelconque)
```

```java
     * d'une droite
4    * @param x1,y1,x2,y2 coordonnees de deux points de la droite
     */
6 static void afficherPositionDroite(int x1, int y1,
                                    int x2, int y2){
8    if (x1==x2 && y1==y2)
         System.out.println("droite mal definie");
10   else if (x1==x2)
         System.out.println("droite verticale");
12   else if (y1==y2)
         System.out.println("droite horizontale");
14   else System.out.println("droite quelconque");
  }
```

AfficherPositionDroite.java

De manière générale, un `else` se rapporte toujours au dernier `if` du même bloc qui n'a pas encore de `else` associé.

▶ **Exercice 24** :

Écrire une fonction `nbSolutionsReelles` qui prend en arguments trois réels `a`, `b`, `c` et renvoie le nombre de solutions réelles de l'équation $\mathtt{a}x^2 + \mathtt{b}x + \mathtt{c} = 0$.

▶ **Exercice 25** (Partiel 2010/2011, exercice 2) :

Un flacon de Betasmurt© pédiatrique contient 1200 gouttes buvables. La posologie doit être adaptée à l'affection, à l'âge et au poids de l'enfant. Selon l'affection et son stade, on opte
– soit pour un *traitement d'attaque* d'une durée comprise entre 3 et 7 jours,
– soit pour un *traitement d'entretien* d'une durée supérieure à 12 jours.
Jusqu'à l'âge de 24 mois, la prise quotidienne est de 10 gouttes/kg pour le traitement d'attaque et de 2 gouttes/kg pour le traitement d'entretien. À partir de 25 mois, la prise quotidienne est de 13 gouttes/kg pour le traitement d'attaque et de 5 gouttes/kg pour le traitement d'entretien.

1. Écrire une fonction `plafond` qui renvoie le plus petit entier supérieur ou égal à l'argument réel `x`.
2. Écrire une fonction `betasmurt` qui prend en argument la durée du traitement, l'âge et le poids de l'enfant et renvoie le nombre de flacons nécessaires au traitement (ou `-1` si un argument est incorrect : durée inférieure à 2 jours ou comprise entre 8 et 11 jours, âge ou poids négatif).

▶ **Exercice 26** (Partiel 2009/2010, exercice 3, question 1) :

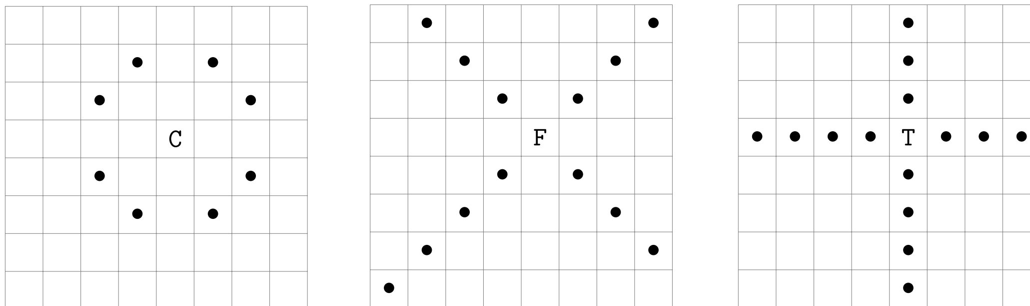
On s'intéresse aux *ordinaux anglais abrégés*, où le nombre (le cardinal) est écrit en chiffres :

1st, 2nd, 3rd, 4th,..., 9th, 10th, 11th, 12th,..., 19th, 20th, 21st, 22nd, 23rd,...

Pour déterminer le suffixe, on regarde le dernier chiffre du nombre : si c'est 1, on ajoute le suffixe "st" ; si c'est 2, le suffixe est "nd" ; si c'est 3, le suffixe est "rd" ; sinon le suffixe est "th". Il y a cependant une exception : si l'avant-dernier chiffre du nombre est 1, le suffixe est toujours "th".
Écrire une fonction `afficherOrdinal` qui prend un entier de type `int` en argument et affiche l'ordinal anglais abrégé correspondant.

► **Exercice 27** (Examen 2004/2005, exercice 4) :

Un échiquier est un carré constitué de 64 cases. Dans cet exercice, l'échiquier sera sous-jacent et ne sera pas matérialisé par un tableau. On s'intéresse ici à trois pièces particulières de ce jeu : les cavaliers représentés symboliquement par le caractère `'C'`, les fous représentés symboliquement par le caractère `'F'` et les tours représentées symboliquement par le caractère `'T'`.



Écrire une fonction qui, étant donnés deux positions sur l'échiquier et un caractère désignant une des pièces précédentes, teste si la deuxième position est différente de la première et est accessible, en un seul mouvement, par la pièce spécifiée à partir de la première position.

► **Exercice 28** (Partiel 2011/2012, exercice 4) :

On considère un jeu de dé pour deux joueurs (identifiés par `1` et `2`) dont voici le principe :

– une partie est composée de 3 tours,
– à chaque tour, chacun des deux joueurs peut choisir de jeter ou non un dé à 6 faces,
– en fin de partie, le score d'un joueur est la somme des valeurs obtenues lors de ses jets de dé,
– un joueur gagne si son score est inférieur ou égal à 14 et strictement supérieur au score de l'autre joueur,
– si les scores sont égaux ou sont tous les deux strictement supérieurs à 14, aucun des joueurs ne gagne.

1. Écrire une fonction `jetDeDe` qui renvoie un entier tiré au hasard entre `1` et `6`. On pourra utiliser la fonction `random` de la classe `Math` qui renvoie un réel aléatoire supérieur ou égal à `0` et strictement inférieur à `1` et
   – soit utiliser la fonction `partieEntiere` de la classe `PartieEntiere` au chapitre 1,
   – soit découper l'intervalle `[0,1[` en six intervalles de même longueur.
2. Écrire une fonction `tourJoueur` qui prend un numéro `k` de joueur en argument, demande à ce joueur s'il souhaite jeter le dé, auquel cas renvoie la valeur d'un jet de dé et renvoie `0` sinon.
3. Écrire une fonction `annoncerValeurs` qui prend en arguments deux valeurs `v1` (valeur du jet du joueur `1`) et `v2` (valeur du jet du joueur `2`) et affiche un message annonçant la valeur associée à chaque joueur.
4. Écrire une fonction `gagnant` qui prend en arguments deux scores `s1` (score du joueur `1`) et `s2` (score du joueur `2`) et renvoie le numéro du joueur gagnant s'il existe ou `0` sinon.
5. Écrire une fonction `partie` mettant en œuvre le jeu et utilisant les quatre fonctions précédentes. Cette fonction demande à chaque tour à chacun des joueurs (identifiés par `1` et `2`) s'il souhaite lancer le dé (réponse `1`) ou pas (réponse `0`) ; une fois que chacun des joueurs a choisi, la fonction affiche la valeur du jet de dé pour chacun des joueurs pour ce tour et met à jour les scores. À la fin du troisième tour, la fonction affiche un message annonçant le gagnant s'il existe, ou un message indiquant qu'aucun des joueurs n'a gagné sinon.

► **Exercice 29** (Examen 2011/2012, exercice 2, question 2) :

Écrire une fonction `string` qui prend en arguments une chaîne de caractères `w` et un entier `k` et renvoie une nouvelle chaîne de caractères de longueur `k` correspondant à la troncature ou à la complétion de `w` comme dans les exemples suivants :

`string("abc",-5)` renvoie la chaîne `"   abc"`	`string("abc",0)` renvoie la chaîne `""`
`string("abc",-4)` renvoie la chaîne `"  abc"`	`string("abc",1)` renvoie la chaîne `"a"`
`string("abc",-3)` renvoie la chaîne `"abc"`	`string("abc",2)` renvoie la chaîne `"ab"`
`string("abc",-2)` renvoie la chaîne `"bc"`	`string("abc",3)` renvoie la chaîne `"abc"`
`string("abc",-1)` renvoie la chaîne `"c"`	`string("abc",4)` renvoie la chaîne `"abc "`

► **Exercice 30** :

Écrire une fonction `conjugaison` qui prend en arguments un pronom parmi `"je"`, `"j’"`, `"tu"`, `"il"`, `"elle"`, `"on"`, `"nous"`, `"vous"`, `"ils"`, `"elles"` et l’infinitif d’un verbe finissant en `"er"` et renvoie l’imparfait du subjonctif correspondant (ou la chaîne "erreur" dès qu’un argument est invalide). Les accents et cédilles seront ignorés.

`conjugaison("j’","enoncer");`	renvoie	`"que j’enoncasse"`
`conjugaison("tu","conjuguer");`	renvoie	`"que tu conjuguasses"`
`conjugaison("elle","conditionner");`	renvoie	`"qu’elle conditionnat"`
`conjugaison("nous","switcher");`	renvoie	`"que nous switchassions"`
`conjugaison("vous","programmer");`	renvoie	`"que vous programmassiez"`
`conjugaison("ils","corriger");`	renvoie	`"qu’ils corrigeassent"`

4.1.4 Aiguillage

Le mécanisme d’aiguillage est representé par l’instruction `switch` qui permet de tester la valeur d’une expression entière (c’est-à-dire une expression dont le type est entier) : on peut spécifier le comportement du programme pour un nombre fini de valeurs (données de façons explicites), puis éventuellement pour l’ensemble des autres valeurs.
La syntaxe de cette structure est :

```
switch (expression entière E){
  case valeur v1: liste d’instructions L1
    break;
  case valeur v2: liste d’instructions L2
    break;
  ...
  case valeur vn: liste d’instructions Ln
    break;
  default: liste d’instructions K
}
```

syntaxe du `switch`

Le déroulement d’un branchement `switch` suit le schéma de la Figure 4.5.

► **Exercice 31** :

Reprendre l’exercice 26 en utilisant un branchement `switch`.

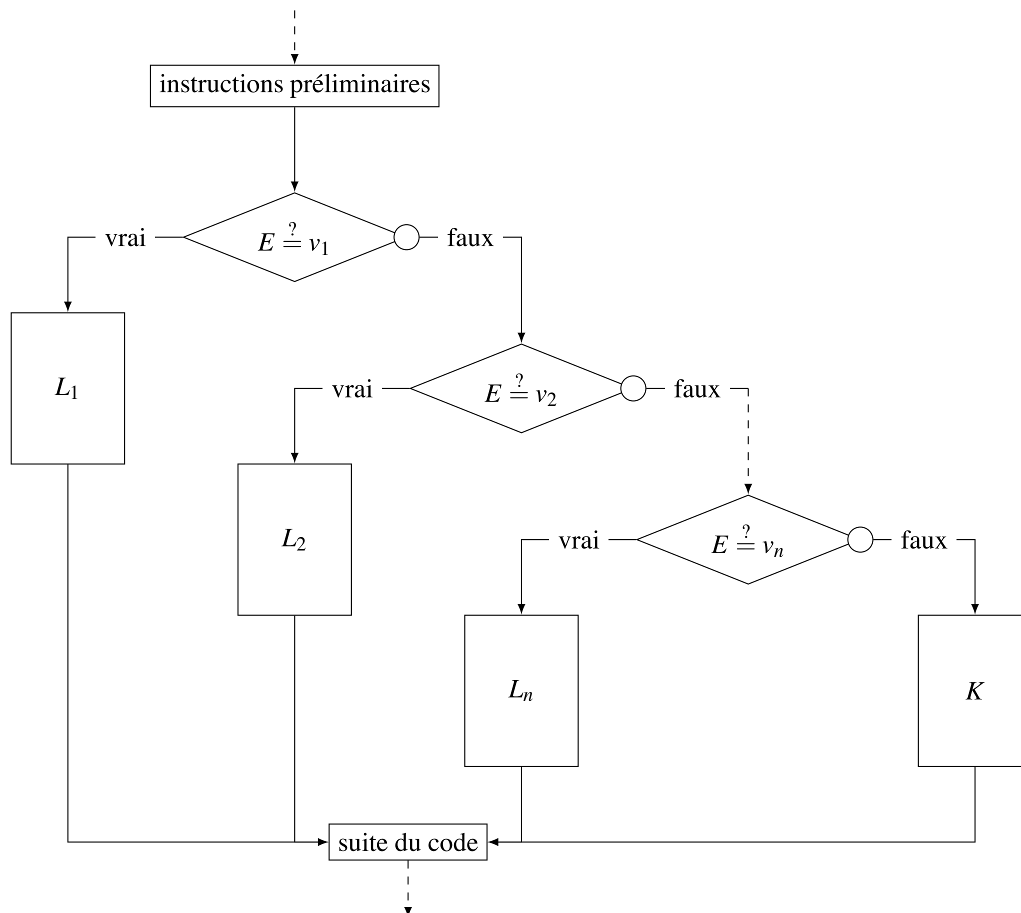


FIGURE 4.5 – `switch` dans un programme

4.2 Boucles

Une des caractéristiques d'un ordinateur est 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 le calcul des n premiers termes 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).

Les boucles permettent de répéter un morceau de code (paramétré). On spécifie soit le nombre de pas (boucle `for`), soit une condition d'arrêt (boucles `while` et `do-while`).

La façon la plus commune de faire une boucle, est de créer un compteur (une variable qui s'incrémente, c'est-à-dire qui augmente de 1 à chaque tour de boucle) et de faire arrêter la boucle lorsque le compteur dépasse une certaine valeur.

4.2.1 Pour

La syntaxe générale de l'instruction `for` est :

```
for (initialisation; condition pour continuer C; mise à jour)
  bloc d'instructions B
```

Syntaxe du `for`

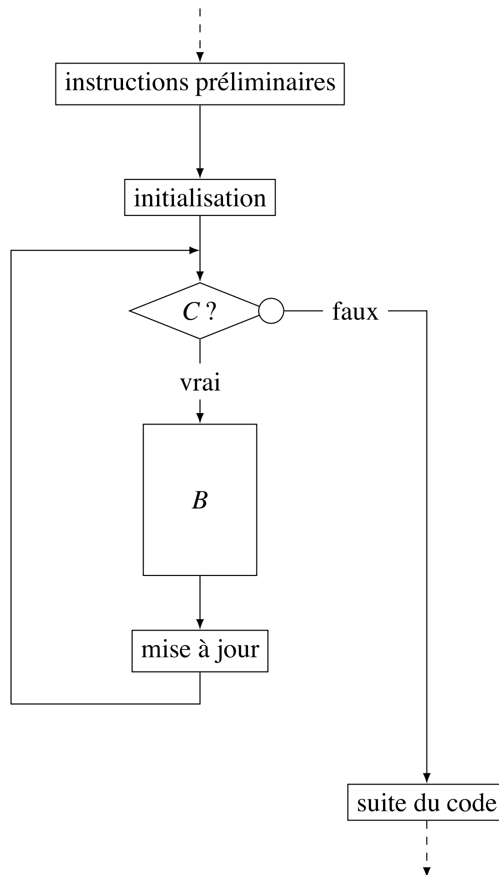


FIGURE 4.6 – `for` dans un programme

L'exécution de cette instruction est donnée dans la Figure 4.6 :

- l'expression *initialisation* est évaluée avant d'entrer dans la boucle et vise à initialiser la variable de contrôle,
- l'expression *condition pour continuer* est de type booléen et est évaluée au début de chaque itération pour déterminer si la boucle doit être poursuivie,
- l'expression *mise à jour* est évaluée à la fin de chaque itération et vise à mettre à jour la variable de contrôle.

Le bloc d'instructions du `for` peut ne jamais être exécuté si dès le départ la condition n'est pas vérifiée.

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`.

Exemple 20. Considérons le code suivant :

```
for (int i=1; i<6; i++){
  System.out.print(i+",");
}
```

Cette boucle affiche 5 fois la valeur de `i`, c'est-à-dire `1,2,3,4,5`.
Elle commence à `i=1`, vérifie que `i` est bien inférieur à 6, etc... jusqu'à atteindre la valeur `i=6`, pour laquelle la condition ne sera plus réalisée, la boucle s'interrompra et le programme continuera son cours.

`Java` autorise la déclaration de la variable de boucle dans l'instruction `for` elle-même. 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++){
  System.out.println("J'ai colle " + i + " timbre(s).");
  System.out.println("Il m'en reste " + (n - i) + ".");
}// A partir d'ici la variable i est inconnue
```

On peut écrire des boucles `for` dans lesquelles on fait évoluer la variable de contrôle en descendant (c'est-à-dire en faisant décroître sa valeur). Par exemple, la boucle suivante affiche 5 fois la valeur de `i`, c'est-à-dire `5,4,3,2,1`.

```
int n = 5;
for(int i=n; i > 0; i--){
  System.out.print(i+",");
}
```

Dans une boucle `for`, on peut, à chaque itération de la boucle, ajouter à la variable de contrôle une valeur quelconque, même négative.
Avec un pas positif, on écrira par exemple :

```
for(int i=n1; i <= n2; i += 3){ ... }
```

et avec un pas négatif :

```
for(int i=n1; i >= n2; i -= 3){ ... }
```

On peut aussi imaginer des mises à jour de la forme `i *= 3`, etc. Il est également possible d'utiliser une variable de contrôle de type `char`, `String`, etc.

▶ **Exercice 32** :

Écrire des fonctions qui prennent un argument entier `n` et affichent les `n` premiers nombres pairs, les `n` premières puissances de 2, les `n` premiers grognements `brhh`, `brrhh`, `brrrhh`, etc.

▶ **Exercice 33** (Partiel 2007/2008, exercice 4) :

Écrire un programme qui demande à l'utilisateur un entier `n`, puis lit une suite de `n` entiers, et affiche combien de fois l'utilisateur a entré deux nombres *successifs* égaux. Les nombres seront lus et traités successivement. Par exemple :
- pour 5 puis la suite 5, 3, 3, 5, 7 le programme doit répondre 1.
- pour 4 puis la suite 1, 5, 5, 5 le programme doit répondre 2.
- pour 7 puis la suite 8, 6, 6, 1, 6, 6, 6 le programme doit répondre 3.

▶ **Exercice 34** (Examen 2009/2010, exercice 3, question 1) :

Soient p un réel positif et r sa racine carrée. On a $r^2 = p$, d'où $r^2 + r = r + p$ puis $r = \frac{r+p}{r+1}$. Une approximation de r est alors obtenue par un calcul itératif : $r_{i+1} = \frac{r_i + p}{r_i + 1}$ avec r_0 que l'on choisit ici égal à $\frac{p}{2}$.
Écrire une fonction `termeN` qui prend en arguments un réel `p` et un entier `n` et qui renvoie le terme r_n.

▶ **Exercice 35** :

Écrire une fonction qui prend en argument un entier `n` et renvoie le terme d'indice `n` de la suite définie par $u_0 = 1, u_1 = 4$ et $u_n = 2 * u_{n-1} + u_{n-2}$ pour $n \geq 2$.

▶ **Exercice 36** (Partiel 2010/2011, exercice 3) :

On souhaite afficher les premiers termes du développement du binôme $(1+z)^t$ où `t` est un réel.

```
> javac Newton.java
> java Newton
Développement de (1 + z)^t à l'ordre n
Entrez l'exposant réel t : 3
Entrez l'ordre entier n : 4
1.0 + 3.0z^1 + 3.0z^2 + 1.0z^3 + 0.0z^4
> java Newton
Développement de (1 + z)^t à l'ordre n
Entrez l'exposant réel t : 0.5
Entrez l'ordre entier n : 5
1.0 + 0.5z^1 - 0.125z^2 + 0.0625z^3 - 0.0390625z^4 + 0.02734375z^5
>
```

1. Écrire une fonction `factorielle` version itérative qui retourne la factorielle de l'entier `k` en argument. Décrire alors les affectations réalisées lors de l'appel `factorielle(5)`.
2. Écrire une fonction `binomial` qui prend en arguments un réel `t` et un entier `k` et retourne la valeur du coefficient binomial $\binom{t}{k} = \frac{\overbrace{t(t-1)(t-2)\cdots(t-(k-1))}^{k \text{ facteurs}}}{k!}$.
3. En déduire une fonction `newton` qui prend en arguments un réel `t` et un entier `n` et retourne une chaîne de caractères représentant le développement de $(1+z)^t$ à l'ordre `n` :
$$\binom{t}{0} + \binom{t}{1}z + \binom{t}{2}z^2 + \cdots + \binom{t}{n}z^n.$$
4. Écrire une classe `Newton` permettant d'utiliser ces fonctions selon l'exemple ci-dessus.

4.2.2 Tant que ...

La syntaxe générale de l'instruction `while` est :

```
while (condition C)
  bloc d'instructions B
```

syntaxe du `while`

L'exécution de cette instruction est donnée dans la Figure 4.7 : l'expression *condition* est de type booléen et est évaluée au début de chaque itération et sert à déterminer si l'itération doit être poursuivie ou non. Le bloc d'instructions du `while` peut ne jamais être exécuté si dès le départ la condition n'est pas vérifiée. Contrairement à ce qui se passait avec le schéma standard de boucle `for`, le nombre d'itérations à réaliser n'est pas a priori connu, pas nécessairement facile à calculer, voire non calculable.

▶ **Exercice 37** :

Écrire une fonction qui prend un entier `k` en argument et renvoie le plus petit `n` pour lequel que la somme des carrés des `n` premiers nombres impairs est plus grande que `k`.

► **Exercice 38** :

La suite de Syracuse de premier terme p (entier strictement positif) est définie par $a_0 = p$ et

$$a_{n+1} = \begin{cases} a_n/2 & \text{si } a_n \text{ est pair} \\ 3 \cdot a_n + 1 & \text{si } a_n \text{ est impair} \end{cases} \qquad \text{pour } n \geq 0$$

Écrire une fonction `porteeSyracuse` qui prend un entier p en argument et renvoie le plus petit indice n vérifiant $a_n = 1$ (et renvoie -1 si p est négatif).

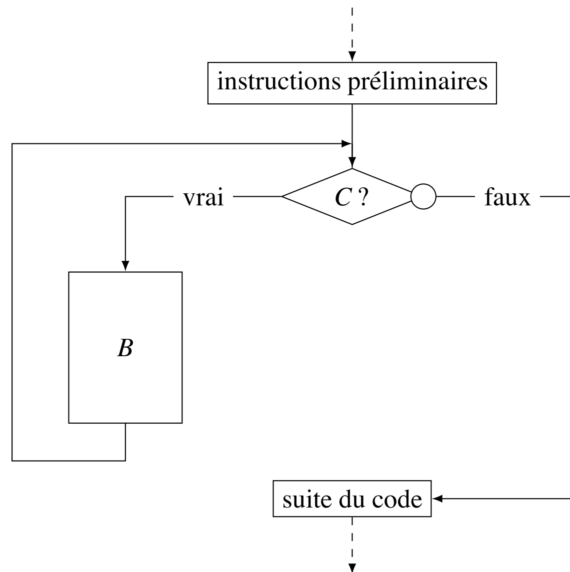


FIGURE 4.7 – `while` dans un programme

► **Exercice 39** :

Exécuter à la main la séquence suivante (dire en particulier ce qui est affiché à l'écran) :

```
int dest1 = 15, dest2 = 27;
int orig1 = 20, orig2 = 20;
boolean arrive1 = false;
boolean arrive2 = false;
while(!arrive1 && !arrive2) {
   orig1 = orig1 - 1;
   orig2 = orig2 + 2;
   arrive1 = (orig1 <= dest1);
   arrive2 = (orig2 >= dest2);
   System.out.println("Les voyageurs se trouvent en "
                      + orig1 + "et" + orig2);
}
System.out.println("Positions finales des voyageurs : "
                   + orig1 + "et" + orig2);
```

► **Exercice 40** :

Écrire un programme qui permet à son utilisateur d'entrer au clavier des valeurs entières non nulles de son choix et calcule à la volée (sans mémoriser tous les entiers lus) d'une part la somme `sPos` et le produit `pPos` des valeurs positives lues et d'autre part la somme `sNeg` et le produit `pNeg` des valeurs négatives lues. Lorsque le programme lira la valeur 0 (cette valeur n'étant pas considérée comme faisant partie des données à traiter), la lecture se terminera, les quatre valeurs calculées seront affichées et le programme s'arrêtera. Si aucun nombre positif n'est lu, `sPos` sera nul et `pPos` vaudra 1 et, si aucun nombre négatif lu, `sNeg` sera nul et `pNeg` vaudra 1.

Exemple : pour les entiers saisis `4,2,-8,3,-1,-2,5,0`, le programme affiche `sPos=14`, `pPos=120`, `sNeg=-11` et `pNeg=-16`.

► **Exercice 41** :

Écrire un programme qui lit une liste d'**au moins deux** entiers strictement positifs dont le nombre n'est pas connu a priori et affiche le deuxième plus grand entier de cette liste. Les nombres seront lus et traités successivement et la fin de la liste sera marquée par un entier négatif ou nul ne faisant pas partie de la liste. Dans le cas où l'utilisateur entre moins de deux entiers, un message d'erreur sera affiché.

Exemple :
- pour la suite de nombres `3,5,7,1,3,8,0`, on obtient `7` ;
- pour la suite de nombres `8,2,7,8,3,6,0`, on obtient `8` (le plus grand entier apparaît deux fois).

► **Exercice 42** (Examen 2011/2012, exercice 4) :

On dit qu'un nombre est *Harshad* ou *Niven* s'il est divisible par la somme de ses chiffres (dans sa représentation décimale).

1. Donner deux entiers à trois chiffres, l'un étant *Harshad*, l'autre non.
2. Écrire une fonction `sommeChiffres` qui renvoie la somme des chiffres de l'entier en argument.
3. Écrire une fonction `estHarshad` qui teste si l'entier en argument est Harshad.

4.2.3 Faire ... tant que

La syntaxe de l'instruction `do...while` est la suivante :

```
do
  bloc d'instructions B
while (condition C);
```

syntaxe du `do ... while`

Le bloc d'instructions du `do ... while` est toujours exécuté au moins une fois avant la première évaluation de la condition.

► **Exercice 43** (Examen 2009/2010, exercice 3, question 2) :

Il s'agit de la suite de l'exercice 34. Écrire une fonction `termeE` qui prend en arguments deux réels `p` et `e` et qui renvoie le terme r_i de plus petit indice vérifiant $-e < r_{i+1} - r_i < e$.

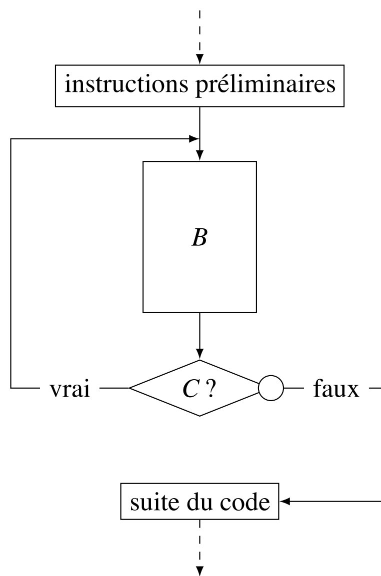


FIGURE 4.8 – `do ... while` dans un programme

4.2.4 Les ruptures

Il est parfois nécessaire d'interrompre l'exécution d'une boucle avant sa fin "normale" (s'échapper de la boucle). Pour cela il existe plusieurs solutions.
La première solution est de faire terminer le programme en utilisant la fonction `System.exit(0)`.

Exemple 21. On souhaite faire un programme qui lit un entier et affiche son plus petit diviseur supérieur à 2. Dans ce cas, dès qu'un diviseur est trouvé, il n'est pas nécessaire de poursuivre les tests de divisibilité pour les autres nombres.

```
import java.util.Scanner;

class Diviseur{
    public static void main(String[] args){
        int n;
        Scanner sc=new Scanner(System.in);
        System.out.println("Donner un nombre entier");
        n=sc.nextInt();
        for(int i=2;i<=n;i++){
            if(n%i==0) { //le reste de la division entiere vaut 0
                System.out.println("Le nombre "+i+" divise "+n);
                System.exit(0); //force la terminaison du programme
            }
        }
    }
}
```

Diviseur.java

Une autre solution, qui permet de sortir d'une fonction sans mettre fin au programme, est d'utiliser le `return` pour faire terminer l'itération.

Exemple 22. On souhaite faire une fonction qui renvoie le plus petit diviseur supérieur à 2 d'un entier donné en argument ou renvoie `-1` si un tel diviseur n'existe pas.

```
static int diviseur(int n){
    for(int i=2;i<=n;i++)
        if(n%i==0)     //le reste de la division entiere vaut 0
            return i; //renvoie le diviseur trouve
    return -1;
}
```

Diviseur2.java

Une troisième solution consiste à utiliser une variable booléenne que l'on utilise dans la condition de la boucle et qui permet de sortir de la boucle. L'avantage de cette solution est qu'elle permet de sortir d'une boucle sans interrompre l'exécution du programme ou sortir de la fonction.

Exemple 23. La fonction `diviseur` dans l'exemple 22 peut être réécrite de la manière suivante :

```
static int diviseur(int n){
    boolean trouve=false;
    int i=2;
    while(i<=n && !trouve){
        if(n%i==0)        //le reste de la division vaut 0
            trouve=true; //maj du drapeau
        i++;
    }
    if(trouve)
        return i-1;
    else
        return -1;
}
```

Diviseur3.java

Finalement, on peut aussi utiliser l'instruction `break` qui permet de sortir d'une boucle pour aller directement à la première instruction se trouvant en dehors de la boucle. 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**.

Exemple 24. La fonction `diviseur` dans l'exemple 23 peut être réécrite comme suit :

```
static int diviseur(int n){
    int i;
    for(i=2;i<=n;i++)
        if(n%i==0) //le reste de la division entiere vaut 0
            break; //on sort de la boucle
    if(i<=n)
        return i;
    else
        return -1;
}
```

Diviseur4.java

► **Exercice 44** :

Écrire une fonction `estPremier` qui teste si un nombre est premier ou non.

► **Exercice 45** :

Écrire une fonction qui prend comme argument un entier `n` et qui teste s'il est présent dans la table de multiplication d'un entier compris entre 1 et 10 (c'est-à-dire s'il existe un entier entre 1 et 10 tel que `n` appartient à la table de multiplication de cet entier).

4.2.5 Les "court-circuits"

Il peut être parfois utile de court-circuiter ou d'abréger l'exécution de l'itération en cours pour passer directement à l'itération 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 25. On suppose qu'on dispose des deux fonctions suivantes :

- `static boolean dateCorrecte(String maDate)` qui vérifie si une chaîne de caractères est bien au format `"jj/mm/aaaa"` et qu'elle représente vraiment une date,
- `static String jourDeLaDate(String maDate)` qui renvoie sous forme de chaîne de caractères le jour correspondant à la date fournie en argument.

La fonction suivante permet alors de jouer en boucle à donner une date et faire afficher le jour correspondant :

```
static void jeuDeLaDate(){
    Scanner sc = new Scanner(System.in);
    String date;

    while (true) {
        System.out.println("entrez une date au format jj/mm/aaaa"
                           + "(ou stop pour arreter)");
        if(sc.hasNextLine()) // lecture de la date ecrite au clavier
            date = sc.nextLine();
        if(date.equals("stop")) // si "stop", on s'arrete en sortant
            break;              // de la boucle
        if(!dateCorrecte(date)) // si format incorrect on recommence
            continue;           // la boucle
        System.out.println("c'etait un " + jourDeLaDate(date));
    }
}
```

► **Exercice 46** :

Quel est le problème avec la fonction suivante :

```
static void xMoinsSept(int n){
    int i=1;
    while(i<=n){
        if(i==7)
            continue;
        int a=1/(i-7);
        System.out.println("Valeur pour "+i+" : "+a);
        i++;
    }
}
```

Comment remédier à ce problème ?