

INF 120 Algorithmique

**Jean-François Remm
Jean-François Berdjugin
Jean-François Dufayard**

INF 120 Algorithmique

par Jean-François Remm, Jean-François Berdjugin, et Jean-François Dufayard

Date de publication 07/09/11

Table des matières

1. Introduction	1
2. Travaux Pratiques papier	2
1. Première séance : variables, constantes et affectation	2
1.1. Définitions	2
1.2. Exemple	3
1.3. Exercices	4
1.3.1. Affectation 1	4
1.3.2. Affectation 2	4
1.3.3. Affectation 3	4
1.3.4. Affectation 4	4
1.3.5. Affectation 5	4
1.3.6. Affectation 6	5
1.4. Devoir maison	5
1.4.1. Première exécution	5
1.4.2. Deuxième exécution	5
1.4.3. Échange	5
2. Deuxième séance : entrées/sorties, conditionnelle	6
2.1. Définitions	6
2.1.1. Entrées/sorties	6
2.1.2. Conditionnelle ou choix	6
2.2. Exemples	7
2.3. Exercices	7
2.3.1. Conditionnelle 1	8
2.3.2. Conditionnelle 2	8
2.3.3. Conditionnelle 3	8
2.3.4. Conditionnelle 4 (Calcul d'une facture d'électricité)	8
2.3.5. Monnayeur à caisse illimité	8
2.4. Devoirs maison	8
2.4.1. Maximum de trois entiers	8
2.4.2. Estimation du prix de revient d'un véhicule	8
2.4.3. Monnayeur à caisse limité	9
3. Troisième séance : boucles	9
3.1. Définitions	9
3.1.1. Boucle : Tant que	9
3.1.2. Boucle : Faire / Tant que	9
3.1.3. Boucle : Pour faire	10
3.2. Exemples	10
3.3. Exercices	11
3.3.1. Boucle 1	11
3.3.2. Boucle 2	11
3.3.3. Boucle 3	11
3.3.4. Boucle 4	11
3.3.5. Boucle 5 (subsidaire)	11
3.4. Devoir maison	12
3.4.1. Somme de n premiers entiers	12
3.4.2. Codage	12
3.4.3. à trouver	12
4. Quatrième séance : Sous programmes	12
4.1. Définitions	12
4.2. Exemples	13
4.2.1. fonction carre et son appel	14
4.3. Exercices	14
4.3.1. Maximum	14
4.4. Devoirs maison	15
4.4.1. Somme trois	15

4.4.2. Somme six	15
3. Travaux Pratiques machine	16
1. Début	16
1.1. Configuration de <i>Windows</i>	16
1.2. Démonstration	17
1.3. Création des répertoires et premier programme	17
2. Premiers éléments de syntaxe	18
2.1. Commentaires	18
2.2. Déclaration	18
2.3. Affectation	19
3. Méthodes	19
3.1. Écrire à l'écran	19
3.2. Lire au clavier	19
3.3. Un petit programme pour lire et écrire au clavier	19
3.4. Méthode statiques	20
4. Introduction à eclipse	20
5. Conditionnelles	21
5.1. Element de syntaxe	21
5.2. Exercices	21
5.2.1. Remise (Conditionnelle simple)	21
5.2.2. Remise (Conditionnelles multiples)	21
5.2.3. Maximum de trois entiers	22
5.2.4. Estimation du prix de revient d'un véhicule	22
5.2.5. Convertisseurs Euros/Francs	22
5.2.6. Monnayeur	22
5.2.7. Comparaison de durées	23
5.2.8. Problèmes du test d'égalité de flottants	23
5.2.9. Tri	23
6. Boucles	23
6.1. Syntaxe	23
6.2. Première Boucle	23
6.3. Lecture de caractère	24
6.3.1. Lecture avec affichage du 'y' final	24
6.3.2. Lecture sans affichage du 'y' final	24
6.4. Somme des n premiers entier	24
6.5. Remboursement d'emprunt	24
6.5.1. Calcul du nombre d'année	24
6.5.2. Calcul du coût	25
6.6. Calcul de maximum	25
6.7. Devine	25
6.7.1. Sans limite	25
6.7.2. Avec un nombre de coup limité	25
6.8. Décomposition en facteurs premiers	25
6.9. Monnayeur	26
6.9.1. Avec caisse illimitée	26
6.9.2. Avec caisse limitée	26
6.10. Boucle "for"	27
6.10.1. Affichage des n premiers entier	28
6.10.2. Somme des n premiers entier	28
6.10.3. Placement	28
Glossaire	29

Liste des illustrations

1.1. Principe	1
2.1. Appel d'un sous-programme	12
3.1. Duke	16

Liste des tableaux

2.1. Règles pour le calcul du prix de revient d'un véhicule	8
3.1. Prix de revient du véhicule	22

Liste des exemples

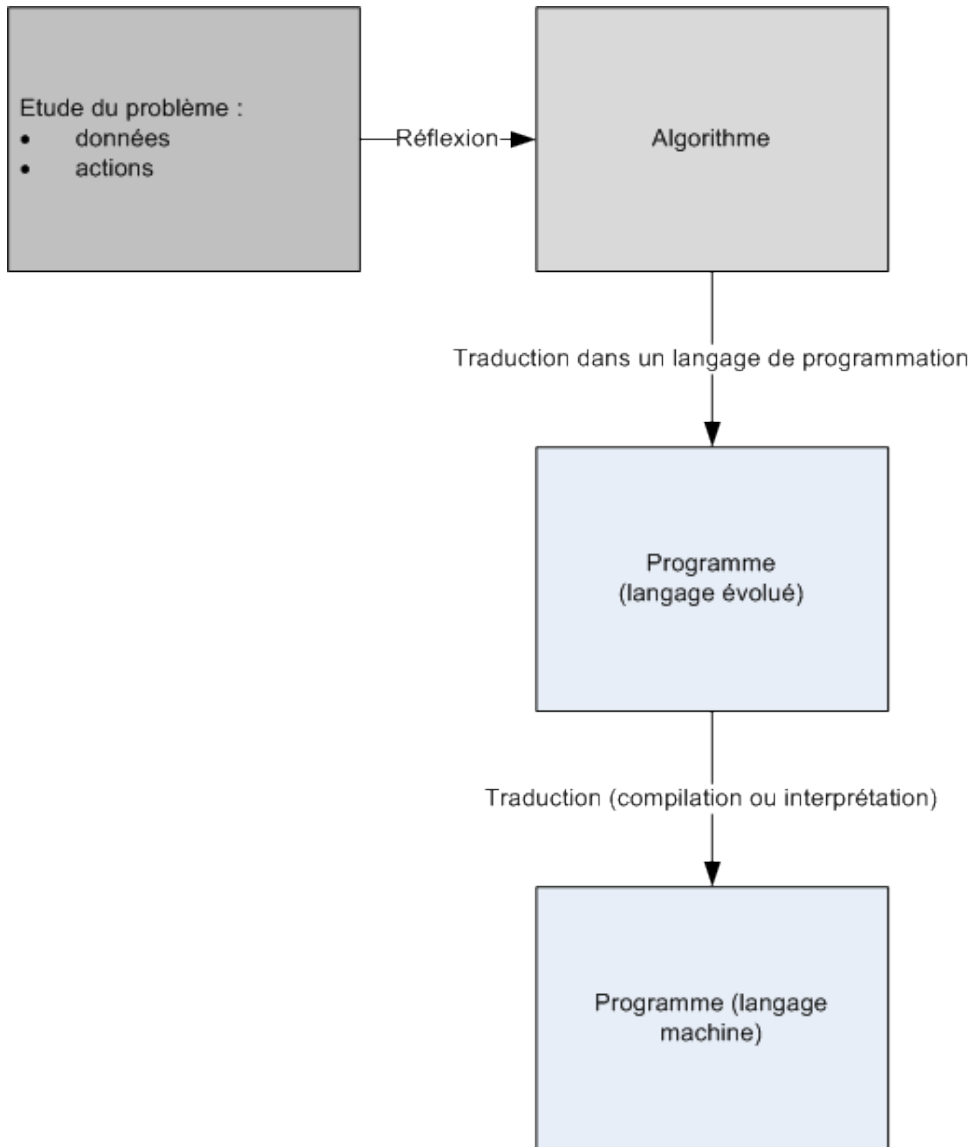
2.1. Omelette	2
2.2. exemple d'affectation	3
2.3. Majeur ? (première version)	7
2.4. Majeur ? (deuxième version)	7
2.5. Majeur ? (troisième version)	7
2.6. Boucle Tant Que	10
2.7. Boucle Faire/Tant que	10
2.8. Boucle "Pour"	11
2.9. Fonction carre : sa définition et son appel	14
3.1. Exemple de déclarations	18
3.2. Exemple séquence	25

Chapitre 1. Introduction

Nous avons à notre disposition pour découvrir l'algorithmique et le langage Java :

- 5 CM (Cours Magistraux),
- 4 TP (Travaux Pratiques) sur table portant sur l'algorithmique,
- 4 TP sur machine portant sur l'apprentissage du langage Java.

Figure 1.1. Principe



Après chaque TP papier, vous aurez des exercices à faire à la maison, ces exercices sont à rendre, en début de séance prochain et seront corrigés par l'un d'entre vous sans support. Les exercices de la dernière séance sont à déposer avant la semaine suivante dans le casier de votre dernier enseignant.

Chapitre 2. Travaux Pratiques papier

Nous allons commencer par travailler sur papier, la programmation sur machine entraîne une complexité supplémentaire (syntaxe, grammaire) sans intérêt pour débiter l'algorithmique.

1. Première séance : variables, constantes et affectation

Un algorithme est un moyen pour un humain de présenter la résolution d'un problème par une *suite d'opérations*. Vous pouvez utiliser l'image d'une recette de cuisine :

Exemple 2.1. Omelette

Le problème est celui de faire une omelette, pour faire une omelette, il nous faut des oeufs. Les oeufs doivent être battus puis cuis.

Nous avons donc bien une *séquence d'action* qui nous permet de *résoudre* notre *problème* qui est celui de la réalisation d'une omelette.

Le français nous permet d'exprimer des idées, mais pas nécessairement de façon non ambiguë et le pouvoir d'expression de la langue est de loin bien trop supérieur à ce que peut comprendre une machine. Nous allons donc utiliser un formalisme le plus voisin possible des langages de programmation. Il existe deux familles de langages algorithmique :

- les *organigrammes*, qui souffrent de lourdeur et de manque de structure et
- les *pseudo code* ou pseudo langages que nous utiliserons.

Ainsi, pour nous un algorithme aura toujours le patron suivant :

```
Algo nomAlgo
  déclarations
Début
  instructions
Fin
```

C'est donc un ensemble défini par un *nom*, une partie *déclaration* et une suite *d'instructions*. Par la suite nous allons décrire plus précisément ces deux parties. A chaque fois, nous donnerons aussi la traduction *java* de notre pseudo code :

```
class nomAlgo
{
  public static void main(String[ ] args)
  {
    déclarations
    instructions
  }
}
```

Nous allons commencer par qualifier ce que nous étendons par *déclaration* puis nous verrons notre première *instruction d'affectation*.

1.1. Définitions

Des définitions plus précises se trouvent en annexes. Pour faire simple, une variable est une données dont la valeur peut évoluer lors de "l'exécution" de notre algorithme. L'ensemble des valeurs d'une *variable* est imposée par son *type*. Le type conditionne aussi les *opérations* possibles. Nous utiliserons les *types* suivants :

entier

les entiers relatifs : ..., -2, -1, 0, 1, 2, ...;

réel

les réels, *1.24* est par exemple un réel;

booléen

les booléen : *vrai, faux*;

caractère

un simple caractère, nous le noterons entre simple quote, '*a*' représente le caractère a;

chaîne de caractère

une chaîne de caractère est un ensemble ordonné de caractères, nous la noterons entre double quote, "*ma première chaîne de caractères*".

Les *constantes* sont des *variables* particulières dont la valeur ne peut changer, la *valeur est constante*.

Pour affecter une valeur à une variable nous utiliserons l'*affectation* (*<-*), *a <- 10*, signifie que la variable a reçoit la valeur 10.

Pour faciliter la lecture d'un algorithme ou d'un programme, on peut ajouter des *commentaires*, nous utiliserons *//* pour introduire un commentaire.

Nous avons donc les variables que nous pouvons déclarer puis nous pouvons les utiliser dans l'instruction d'affectation. Il nous est maintenant possible d'écrire notre premier algorithme :

```
Algo Decl
var a,b : entier //a et b sont deux variable de type entier
const c <- 0.5 : réel //c est une constante de type réel
                //dont la valeur est 0,5
Début
  a <- 10 //a reçoit 10 on dit encore 10 est affecté à a
  b <- a + 1 //b reçoit le résultat de l'évaluation de a+1,
            //soit la valeur de a augmentée de 1, soit encore 11
Fin
//A la fin de notre algorithme a vaut 10, b vaut 11 et c vaut 0,5
```

Comme toujours notre pseudo code peut être traduit en java, l'*affectation* (identifiant *<-* expression) devient égal (identifiant = expression;), le type *entier* devient *int*, le type *réel* devient *double*, le type *booléen* devient *boolean*, le type *caractère* devient *char*, le type *chaîne* devient *String*. Avec les règles précédentes, notre pseudo-code serait traduit :

```
public class Decl
{
  public static void main(String[ ] args)
  {
    int a, b;
    final double c = 0.5 ;

    a=10;
  }
}
```

1.2. Exemple

Exemple 2.2. exemple d'affectation

```
Algo Affect1
var a : entier
var b : entier
Début
  a <- 5
  b <- a+1
  a <- 2
Fin
```

Le résultat de l'exécution de l'algorithme est a vaut 2 et b vaut 6. Il faut bien remarquer qu'une variable n'a pas de mémoire, à un instant donné, elle ne contient qu'une valeur.

1.3. Exercices

Nous allons commencer par donner le résultat de l'exécution d'algorithme avant d'écrire les nôtres.

1.3.1. Affectation 1

Donner le résultat de l'exécution de l'algorithme suivant :

```
Algo Affect1b
  var a : entier
  var b : entier
Début
  b <- 5
  a <- b+1
  b <- 2
Fin
```

1.3.2. Affectation 2

Donner, si cela est possible, le résultat de l'exécution de l'algorithme suivant, sinon indiquer le problème : *Le résultat de l'exécution est la valeur de chacune des variables et les sorties produites (nous verrons les entrées/sorties plus tard).*

```
Algo Affect2
  var a : entier
  var b : entier
Début
  a <- 2
  a <- a+1
Fin
```

1.3.3. Affectation 3

Donner, si cela est possible, le résultat de l'exécution de l'algorithme suivant, sinon indiquer le problème :

```
Algo Affect3
  var a : entier
  var b : entier
Début
  b <- a+1
  a <- 2
Fin
```

1.3.4. Affectation 4

Donner, si cela est possible, le résultat de l'exécution de l'algorithme suivant, sinon indiquer le problème :

```
Algo Affect4
  var a : entier
  var b : entier
Début
  a+5 <- 3
Fin
```

1.3.5. Affectation 5

Donner le résultat de l'exécution de l'algorithme suivant :

```
Algo Affect5
  var a : entier
  var b : entier
Début
  a <- 5
  b <- a+4
  a <- a+1
  b <- a-4
Fin
```

1.3.6. Affectation 6

Donner le résultat de l'exécution de l'algorithme suivant :

```
Algo Affect6
var a : entier
var b : entier
var c : entier
Début
a <- 3
b <- 10
c <- a+b
b <- a+b
a <- c
Fin
```

1.4. Devoir maison

Donner le résultat de l'exécution de l'algorithme suivant :

1.4.1. Première exécution

Donner le résultat de l'exécution de l'algorithme suivant :

```
Algo Affect7
var a : entier
var b : entier
Début
a <- 5
b <- 7
a <- b
b <- a
Fin
```

1.4.2. Deuxième exécution

Donner le résultat de l'exécution de l'algorithme suivant :

```
Algo Affect8
var a : entier
var b : entier
Début
a <- 5
b <- 7
b <- a
a <- b
Fin
```

1.4.3. Échange

Compléter l'algorithme suivant pour qu'il réalise l'échange entre le contenu de la variable a et le contenu de la variable b ¹ :

```
Algo EchangeAB
var a : entier
var b : entier
var c : entier
Début
a <- 5
b <- 7
```

¹La troisième variable est bien nécessaire, pour transférer le contenu de deux aquariums, il en faut un troisième.

Fin

2. Deuxième séance : entrées/sorties, conditionnelle

Nous avons des *variables*, *l'affectation* et la première des structure de contrôle : la *séquence*. Nous allons introduire un moyen d'interagir avec l'utilisateur en utilisant les *entrées/sorties*, pour nous, elles se limiteront au clavier et à l'écran. Nous allons aussi introduire un moyen d'avoir des parcours d'algorithme différents en fonction de valeurs booléennes : la structure de contrôle *alternative*, aussi nommée *choix* ou *conditionnelle*.

2.1. Définitions

2.1.1. Entrées/sorties

Les *entrées/sorties* vont permettre à notre algorithme d'échanger des informations avec un utilisateur. Le point de vue choisi est celui de l'algorithme (l'ordinateur), la lecture est une opération bloquante qui attend une frappe au clavier pour en affecter le résultat à une variable, tant que la valeur n'a pas été saisie, le passage à la l'instruction suivante, ne peut avoir lieu ; l'écriture permet d'afficher la valeur d'une variable à l'écran. En pseudo-langage nous aurons :

```
x <- lire() //la valeur de x est saisie au clavier
ecrire(x) //la valeur de x est affichée à l'écran
```

La traduction java est un peu plus complexe, nous aurons le temps de la détailler au second semestre :

```
Scanner sc = new Scanner(System.in);
x=sc.next(); //la valeur de x est saisie au clavier

System.out.println(x) //la valeur de x est affichée à l'écran
```

2.1.2. Conditionnelle ou choix

La conditionnelle est le premier moyen dont nous disposons pour avoir deux exécutions différentes, en fonction d'une valeur booléenne. Vous pouvez utiliser l'analogie de l'aiguillage, si l'aiguillage est baissé, le train prend une voie, sinon, l'aiguillage n'est pas baissé et le train prend une autre voie, pour nous l'aiguillage est la condition booléenne. En pseudo-code, tout comme en java, nous aurons deux écritures différentes, la clause *sinon* étant *optionnelle* :

```
Si (condition) Alors
  Instructions //si la condition est vérifiée, je passe par la
Fin Si
//Dans tous les cas, je suis la
```

En java :

```
if (condition)
{
instructions //si la condition est vérifiée, je passe par la
}
//Dans tous les cas, je suis la
```

Ce qui nous donne avec la clause *sinon*, en pseudo code :

```
Si (condition) Alors
  instructions1 //si la condition est vérifiée, je passe par la
Sinon
  instructions2 //si la condition n'est pas vérifiée, je passe par la
Fin Si
//Dans tous les cas, je suis la
```

En java :

```
if (condition)
{
```

```
instructions1 //si la condition est vérifiée, je passe par la
}
else
{
instructions2 //si la condition n'est pas vérifiée, je passe par la
}
//Dans tous les cas, je suis la
```

2.2. Exemples

Un petit algorithme qui demande l'âge de l'utilisateur et qui affiche majeur ou non majeur suivant le résultat :

Exemple 2.3. Majeur ? (première version)

```
Algo MajeurV1
var age:entier
Début
Ecrire("Veuillez saisir votre age")
age <- lire()
Si (age >= 18) Alors
    ecrire("Majeur")
Sinon
    ecrire("Mineur")
Fin Si
Fin
```

Nous pouvons améliorer notre première version en ne réalisant qu'un affichage pour le résultat :

Exemple 2.4. Majeur ? (deuxième version)

```
Algo MajeurV2
var age:entier
var res: chaine
Début
Ecrire("Veuillez saisir votre age")
age <- lire()
Si (age >= 18) Alors
    res <- "Majeur"
Sinon
    res <- "Mineur"
Fin Si
Ecrire(res)
Fin
```

Nous pouvons aussi adopter un autre point de vue en supposant que la personne est mineur et modifier cet état, si elle a plus de 18 ans.

Exemple 2.5. Majeur ? (troisième version)

```
Algo MajeurV3
var age:entier
var res: chaine
Début
res <- "Mineur"
Ecrire("Veuillez saisir votre age")
age <- lire()
Si (age >= 18) Alors
    res <- "Majeur"
Fin Si
Ecrire(res)
Fin
```

Un cas global est choisi puis modifié en fonction d'une condition spécifique.

2.3. Exercices

2.3.1. Conditionnelle 1

Écrire un algorithme qui lit deux valeurs entières et affiche le maximum des deux.

2.3.2. Conditionnelle 2

A la caisse d'un supermarché (bien connu à l'Isle d'abeau), nous bénéficions d'une remise de 1% sur le montant de nos achat lorsque celui-ci dépasse 300 euros. Écrire un algorithme qui après lecture du montant initialement du, affiche le montant à payer. *Il est maladroit d'avoir plus d'une instruction d'écriture du résultat. Il est maladroit d'avoir une clause sinon. Il est inutile d'avoir recourt à une autre variable.*

2.3.3. Conditionnelle 3

Même exercice avec :

- 1% de remise pour un achat compris entre 300 et 750 euros
- 2% au delà de 750 euros

2.3.4. Conditionnelle 4 (Calcul d'une facture d'électricité)

Trouver le prix à payer sachant qu'une facture inclut une somme de 4 euros de frais fixes et que s'ajoute un prix en fonction de la consommation :

- 0,1 euro/kWH pour les 100 premiers kilowatts heures
- 0,07 euro/kWH pour les 150 suivants
- 0,04 euro/kWH au delà

2.3.5. Monnayeur à caisse illimité

Un distributeur qui rend de la monnaie doit rendre en priorité les pièces les plus grosses. En supposant que la machine rends des jetons de 5, 2 et 1 unités et qu'elle doit vous rendre *nb* unités, écrire un algorithme qui simule le rendu. On suppose que la caisse de départ de la machine est illimitée. I.e. Il y a toujours assez de jetons en caisse pour le rendu.

Astuce

Le reste (%) et la division (/) de deux entiers peuvent vous aider. $14/5$ donne 2 et $14\%5$ donne 4. *Pour cet exercice vous n'avez pas besoin de conditionnel*

2.4. Devoirs maison

2.4.1. Maximum de trois entiers

Lire trois valeurs entières *a*, *b* et *c*. Afficher le maximum des trois

2.4.2. Estimation du prix de revient d'un véhicule

Il existe un barème pour l'évaluation du prix de revient kilométrique des véhicules. Écrire un algorithme effectuant le calcul de ce prix en fonction de *nb*, nombre de kilomètres parcourus .

Tableau 2.1. Règles pour le calcul du prix de revient d'un véhicule

puissance fiscale	5CV	6CV
jusqu'à 5000 km	$n1 * 0,43 (=p1)$	$n * 0,47$

puissance fiscale	5CV	6CV
de 5001 à 20000 km	$(n_2 * 0,23) + p_1 (=p_2)$	$(n * 0,27) + 1000$
au delà de 20000	$(n_3 * 0,28) + p_2$	$n * 0,32$

où n est le nombre total de kilomètres parcourus, n_1 le nombre de kilomètres parcourus entre 0 et 5000, n_2 le nombre de kilomètres parcourus entre 5001 et 20000 et n_3 le nombre de kilomètre parcourus au delà de 20000. Exemple : si j'ai parcouru 8500 km, $n=8500$, $n_1=5000$, $n_2=3500$ et $n_3=0$.

2.4.3. Monnayeur à caisse limité

Cette fois ci, le monnayeur a une caisse limité, il a un stock de pièce de 5, de 2 et de 1. Il vous faut remarquer que la monnaie peut ne pas être rendue.

3. Troisième séance : boucles

Les boucles sont des structures de contrôle qui permettent à une suite d'instructions d'être exécuté plusieurs fois (itération).

3.1. Définitions

3.1.1. Boucle : Tant que

On répète une série d'instructions tant qu'une condition est *vrai*. En pseudo-code :

```
Tant que (condition) Faire
  instructions //tant que la condition est vrai,
                //on répète instructions
Fin Tant que
```

En java :

```
while (condition)
{
  instructions //tant que la condition est vrai,
                //on répète instructions
}
```

Il est a remarquer que l'on ne rentre pas forcément dans une boucle "*Tant que*", ce qui n'est pas le cas de la boucle suivante "*Faire/Tant que*".

3.1.2. Boucle : Faire / Tant que

En pseudo-code :

```
Faire
  instructions //je passe au moins une fois par là
Tantque (condition)
```

En java :

```
do
{
  instructions //je passe au moins une fois par là
}
while (condition);
```

La différence essentielle entre ces deux structures itératives est le moment où est évalué le test. L'usage de "*Faire/Tant que*" fait passer au moins une fois dans la boucle ; la condition de terminaison n'est évaluée qu'à l'issue du passage. À l'opposé, avec "*Tant que*" la condition de terminaison est évaluée avant toute entrée dans la boucle. Il peut donc n'y avoir aucune exécution de la boucle.

Pour les deux types de boucle précédent, il vous faudra vérifier, si une fois entré dans la boucle, vous avez une chance de vous en sortir. Un troisième type de boucle, la boucle "*Pour*" qui n'est qu'une réécriture du "*Tant que*" permet de ne pas avoir à se poser ce problème.

3.1.3. Boucle : Pour faire

La boucle "pour" permet de faire varier un indice entre deux valeur.

En pseudo-code :

```
Pour var de debut à fin pas de n
Faire
  instructions //var varie de debut à fin par pas de n
Fin Pour
```

En java, suivant le fait que le début est inférieur à la fin, nous avons deux traduction différentes :

```
//si le pas est négatif
for(int var=debut ;var<=fin ;var=var+n)
{
instructions
}

//si le pas est positif
for(int var=debut ;var<=fin ;var=var+n)
{
instructions
}
```

3.2. Exemples

Voici trois exemples, un pour chacune des boucles.

Exemple 2.6. Boucle Tant Que

```
Algo Boucle1
  var a : entier
Début
  a <- 5
  Tant que (a > 0) Faire
    Ecrire(a)
    a <- a - 2
  Fin Tant que
Fin
```

L'exécution cet algorithme nous donne :

- a vaut -1
- Les affichages sont : 5, 3, 1

Exemple 2.7. Boucle Faire/Tant que

```
Algo Boucle2
  var a,b : entier
Début
  faire
    a <- Lire()
    b <- a*a
    Ecrire(b)
  Tanque (a ≠ 0)
Fin
```

Si nous saisissons les valeurs : 1, -1, 2, 0 ; nous avons l'exécution suivante :

- a vaut 0, b vaut 0
- Les affichages sont : 1, 1, 4, 0

Exemple 2.8. Boucle "Pour"

```

Algo Boucle3
  var i : entier
Début
  Pour i de 0 à 10 pas de 2
    ecrire(i*i)
  Fin Pour
Fin

```

Le résultat de l'exécution est :

- i vaut 12
- Les affichages sont : 0, 4, 16, 36, 64 , 100

3.3. Exercices**3.3.1. Boucle 1**

Lire un caractère et l'afficher jusqu'à ce que l'on saisisse 'y'. Réaliser deux versions cet algorithme :

- une version avec affichage du 'y' final
- une version sans affichage du 'y' final

3.3.2. Boucle 2

Écrire un algorithme qui affiche les n premiers entiers (de 1 à n).

3.3.3. Boucle 3

Les nénuphars et les grenouilles, chaque année pendant l'hivers les nénuphars se multiplient, une fois que les nénuphars ont grandi les grenouilles, qui croissent mais qui ne croissent pas, mangent les nénuphars à raison de un nénuphar par grenouille et par an.

Nicolas a un étang avec n nénuphars qui se multiplient de p pourcents par an, il introduit g grenouilles. Calculer le nombre d'années pour être débarassé des nénuphars et si ce n'est pas possible le signaler.

Par exemple si l'étang contient 100 nénuphars (n) qui se se multiplient de 10 pourcents (p) et que 10 grenouilles (g) sont introduites ; l'étang ne pourra jamais être débarassé des nénuphars. Après la multiplication nous avons $n + (n*p)/100 = 110$ nénuphars, les grenouilles en mangent 10, il en reste donc 100 tout comme au début, c'est un système sans fin.

Par contre si nous avons 100 nénuphars qui se multiplient de 20 pourcent et 50 grenouilles, nous avons après la croissance 120 nénuphars, les grenouilles en mangent 50, il en reste donc 70 pour commencer un nouveau cycle. Lors du second cycle nous avons après multiplication 84 nénuphars, les grenouilles en mangent 50, il en reste 34. Lors du troisième cycle nous avons toujours, après multiplication 40,8 nénuphars les grenouilles les mangent tous.

3.3.4. Boucle 4

Décomposer un nombre en nombre premiers. Essayer les divisions du nombre par les tous les entiers (à partir de 2) et faire afficher simplement les différents diviseurs.

Note

On effectue les divisions du nombre par les différents entiers, qu'ils soient premiers ou non, de toute façon, un nombre qui n'est pas premier ne pourrait diviser car tous ses diviseurs (plus petit que lui) auraient précédemment divisé le nombre.

3.3.5. Boucle 5 (subsidaire)

Même exercice mais avec affichage des puissances.

3.4. Devoir maison

3.4.1. Somme de n premiers entiers

Écrire un algorithme qui affiche la somme des n premiers entiers. *La somme à l'étape courante est égale à la somme à l'étape précédente plus l'entier courant. Par exemple la somme de 3 premiers entiers est égale à la somme des 2 premiers plus 3.*

3.4.2. Codage

Coder un nombre nb écrit en base 10 en base "base" (base < 10). Rappel : il faut faire les divisions successives de nb par base jusqu'à obtenir un quotient nul.

Note

On formate le résultat par une concaténation de chaînes. Si s est une chaîne qui vaut "bb" alors "aa"+ s vaut "aabb".

3.4.3. à trouver

Écrire un algorithme qui fasse deviner un nombre entier a *Trouver* en donnant des indications (trop grand, trop petit) avec nb *Essai* autorisé. Il faut obtenir un affichage final Gagné ! ou Perdu !.

4. Quatrième séance : Sous programmes

Un algorithme, si on lui conservait sa structure monolithique, pourrait comporter plusieurs centaines de lignes et deviendrait rapidement illisible. Un bon algorithme (et le futur programme issu de la traduction de l'algorithme) se doit d'être compris rapidement. Il faut donc modulariser l'écriture d'algorithmes de manière à rendre un algorithme :

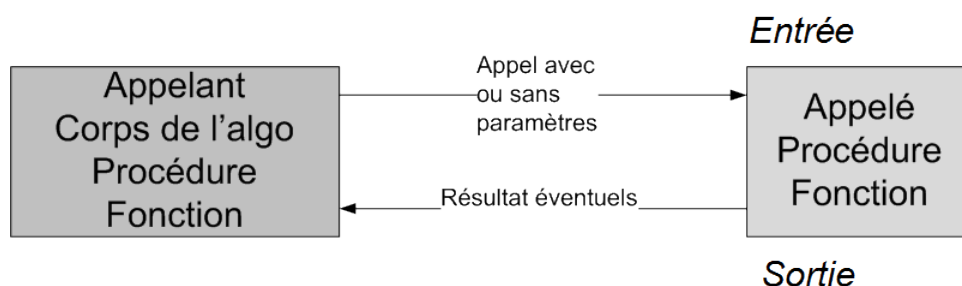
- court,
- clair,
- compréhensible,
- cohérent.

Ces qualités permettront de rendre un algorithme réutilisable, maintenable, ...

4.1. Définitions

Un moyen de factoriser (réutiliser) du code est de prendre un ensemble d'instructions réalisant une certaine tâche et de donner à cet ensemble d'instruction un nom. Nous appellerons cet ensemble d'instruction une fonction. Une fonction prend ou prend pas de paramètres et renvoie ou non un résultat. Si la fonction ne renvoie pas de résultat, son type de retour est "vide", ces fonctions sont parfois appelées procédures .

Figure 2.1. Appel d'un sous-programme



Les paramètres peuvent être transmis de deux manières (source wikipédia) :

copie

le code appelé dispose d'une *copie* de la valeur. Il peut la modifier, l'information initiale dans le code appelant n'est pas affectée par ces modifications.

référence

le code appelé dispose d'une information lui permettant d'accéder en mémoire à la valeur que le code appelant cherche à lui transmettre. Il peut alors modifier cette valeur là où elle se trouve, le code appelant aura accès aux modifications faites sur la valeur.

Nous utiliserons exclusivement le passage par copie (tout comme en java)².

Pseudo Langage

```
Fonction nomFonc(param1 : type1 [, param2 : type2] ) : TypeRetour
...
Instructions
...
Renvoie (resultat) // si nécessaire (TypeRetour != vide)
Fin Fonction

Algo Appel
  var r : TypeRetour      // typeRetour ne peut ici etre void
Début
  r <- nomFonc(val1,val2) // si la fonction ne retourne rien, l'affectation n'existe pas
Fin
```

Java

```
class Appelee
{
  static typeRetour nomFonc(type1 param1[type2 param2])
  {
    ...
    instructions
    ...
    return (resultat) ;
  }
}

class Appelante
{
  public static void main(String[ ] args)
  {
    TypeRetour r;
    r = Appelee.nomFonc (val1, val2);
  }
}
```

4.2. Exemples

²En java, aussi bien les types primitifs que les types références (objets et tableaux) sont passés par valeur. Cependant extérieurement, comme c'est la références qui est passée, cela s'apparente presque à un passage par adresse. C'est à dire que l'état d'un objet peut être changé par l'application de méthodes s'appliquant à celui-ci, mais la référence à l'objet, contenue dans la variable, reste la même. Nous verrons cela au semestre prochain.

4.2.1. fonction carre et son appel

Exemple 2.9. Fonction carre : sa définition et son appel

```
Fonction carre(nombre : entier) : entier
  var res : entier

  res <- nombre * nombre
  Renvoie(res)
Fin Fonction

Algo ExAppelCarre
  var r : réel
Début
  r <- Lire()
  Ecrire(carre(r)) //r au carré
  Ecrire(carre(r)*r)//r au cube
  Ecrire(carre(carre(r))) //r exposant 4
Fin
```

Si nous lisons 2 que se passe-t-il ? Lorsque nous utilisons *carre(r)* :

1. Le programme appelant (ExAppelCarre) passe par *copie* la *valeur de r* (2) au programme appelé (carre) puis attend le résultat pour l'afficher.
2. Le programme appelé *reçoit la valeur de r* (2) et l'affecte à *nombre*, puis commence son exécution.
3. Le programme appelé *renvoie* la valeur de res qui vaut ici 4 au programme appelant.
4. Le programme appelant *reçoit* la valeur de res donc 4 et l'affiche.

4.3. Exercices

4.3.1. Maximum

Soit les fonctions suivantes

```
Fonction max2(param1 : entier, param2 : entier) : entier
Var max : entier

  max <- param1
  Si max < param2 alors
    max <- param2
  fin si
  Renvoie (max)
Fin Fonction

Fonction max3(param1 : entier, param2 : entier, param3 : entier) : entier
Var max : entier

  max <- Max2(param2, param3)
  si max < param1 alors
    max <- param1
  fin si
  Renvoie (max)
Fin Fonction

Fonction blaBla(p1 : entier, p2 : entier, p3 : entier) :vide
  Ecrire("Le maximum de ")
  Ecrire(p1)
  Ecrire(p2)
  Ecrire(p3)
  Ecrire(« est »)
  Ecrire(max3(p1,p2,p3))
Fin Fonction
```

Que donne l'algorithme suivant :

```
Algo finDeTD
  Var i,j,k,l : entier
Début
  i <- 10
  j <- 20
  k <- 30
  l <- max2(10,20)
  Ecrire(l)
  Ecrire(Max2(Max2(i,j),k))
  Ecrire(Max2(l,k))
  Ecrire(Max3(i,j,k))
  blaBla(i,j,k)
Fin
```

4.4. Devoirs maison

Vous allez devoir écrire des fonctions qui réalisent la somme des valeurs des variables passées en arguments.

4.4.1. Somme trois

Écrire une fonction qui renvoie la somme de trois arguments de type entier. Donner un exemple d'appel.

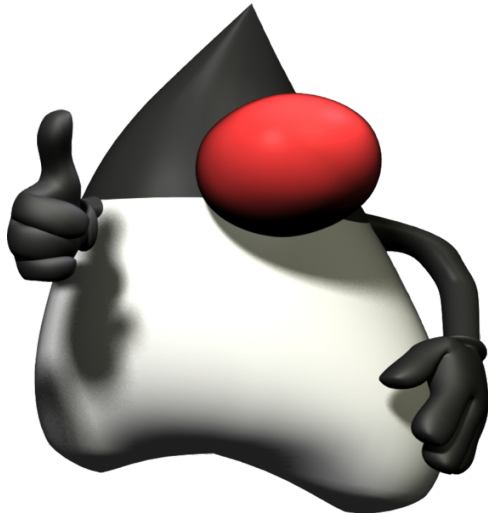
4.4.2. Somme six

Écrire une fonction qui renvoie la somme de six arguments de type entier. Donner un exemple d'appel. *Vous pouvez utiliser la fonction précédente.*

Chapitre 3. Travaux Pratiques machine

Dans cette série de quatre TP (Travaux Pratiques), vous aller, principalement, coder, en *java* les algorithmes vus en TDs. L'ensemble des TP doit être fait. Nous commencerons en ligne de commande pour fixer les notions de compilation et d'édition de lien, puis nous utiliserons l'IDE (Integrated Development Environment) eclipse.

Figure 3.1. Duke



1. Début

Dans cette première partie nous allons écrire notre premier programme *java* qui affiche

```
Hello World
```

à l'écran.

1.1. Configuration de *Windows*

Pour travailler nous utiliserons le JDK Java Development Kit, l'environnement dans lequel le code *Java* est compilé pour être transformé en *bytecode* afin que la JVM (Machine Virtuelle de Java) puisse l'interpréter. Le JDK est aussi appelé Java 2 Software Development Kit.

Le JDK contient un ensemble d'outils parmi lesquels nous trouvons :

`javac`

le compilateur java,

`jar`

l'archiveur, qui met sous forme d'un paquetage les bibliothèques de classes relatives au projet fusionné en un fichier *jar*,

`javadoc`

le générateur de documentation, qui génère automatiquement de la documentation à partir des *annotations* du code source,

`jdb`

le débogueur,

jre

un ensemble d'outils permettant l'exécution de programmes Java sur toutes les plates-formes supportées et qui contient la machine virtuelle java.

Vous devez avoir d'installé sur votre machine, le jdk, il doit se trouver dans `C:\Program Files\Java`. Mais pour l'utiliser, il vous faut définir un ensemble de variables d'environnement :

PATH

qui contient la liste des dossiers dans lesquels Windows ira chercher les commandes par défaut.

CLASSPATH

qui permet lors de la compilation ou de l'exécution d'un programme à l'aide de Java, de spécifier l'emplacement des bibliothèques utilisées.

Pour ces TP, nous allons commencer par définir la variable `path` pour quelle contienne la valeur du répertoire contenant les exécutable `javac` et `java`. Il vous faut suivre la procédure suivante pour la définir :

1. clic droit sur « poste de travail »,
2. afficher les propriétés,
3. choisir l'onglet « avancé »,
4. cliquer sur « variables d'environnement »,
5. éditer ou créer la variable `path` en lui affectant le répertoire contenant `javac` et `java`, le séparateur est le point-virgule.

Tester en tapant dans une invite de commande **javac**.

1.2. Démonstration

Le langage java peut-être utilisé dans le domaine du Web pour exécuter du code sur le client :

applet

du code inclus dans une page Web

Java Web Start (JWS)

un moyen de déploiement d'applications via le web.

Le langage java peut aussi être utilisé pour s'exécuter sur les serveurs (Web dynamique) :

EJB

Enterprise Java Bean composant Java, réutilisable, destiné à être déployé sur des serveurs d'applications.

JSP

Java Server Page, technologie Java de création de pages dynamique en XHTML

Servlet

Le coeur, le programme java qui s'exécute sur le serveur.

Le langage Java est aussi présent sur les systèmes embarqués, java ME (Micro Edition) et sur les systèmes mobiles. Par contre java n'est pas javascript (JS) ou javaFX.

1.3. Création des répertoires et premier programme

Nous allons structurer notre travail en répertoire, reproduisez en utilisant l'archive fournie, le modèle suivant sur votre partage (z:) :

```
| -code
| -bin
| -boucles
```



```
| -conditionnelles
| -exemples
| -perso
| -tests
```

Placez-vous, à l'aide du ligne de commande (**cmd**), dans le répertoire `code/tests`, nous allons créer notre premier programme. Il est de tradition de commencer l'apprentissage d'un langage par la production d'un programme écrivant un texte sur la sortie standard. Écrire le fichier `Hello.java` suivant :

```
//Mon premier programme Java
//JFB

public class Hello
{
    public static void main (String[] args)
    { System.out.println("Hello World");
    }
}
```

Le compiler :

```
javac Hello.java
```

Charger le fichier `Hello.class` avec le JDK

```
java Hello
```

Attention : notez bien que le fichier obtenu s'appelle `Hello.class`, mais qu'on lance bien la commande `java Hello`.

Pour bien comprendre la distinction entre le code source et le bytecode, nous allons placer le bytecode dans un autre répertoire en utilisant la ligne de commande suivante :

```
javac -d ../bin Hello.java
```

Ouvrir une autre invite de commande (vous en avez deux maintenant) et rendez-vous dans `code/bin` puis taper

```
java Hello
```

. Dans la suite nous continuerons avec cette approche de séparation entre le code et le *bytecode*.

2. Premiers éléments de syntaxe

Dans cette partie, nous allons apprendre à déclarer des variables, et à faire des affectations.

2.1. Commentaires

Les commentaires peuvent être en ligne `//` ou en bloc `/* */`. Tout ce qui suit le signe `//` ou qui est dans le bloc `/* */` est considéré comme du commentaire c'est à dire que le compilateur n'en tient pas compte. Dans ces TP vous mettez systématiquement un commentaire portant votre nom et un résumé de votre programme.

2.2. Déclaration

En Java *TOUTES* les variables doivent être déclarées. Une déclaration précise un type et comporte une ou plusieurs variables de ce type, par exemple :

Exemple 3.1. Exemple de déclarations

```
int mini, maxi ; // mini et maxi sont deux entiers.
char c ; // c est un caractère
final double tva = 0.196 ; // tva est un réel dont la valeur (non modifiable) est 0.196
```

On peut répartir les variables entre les déclarations comme on le désire ; les déclarations précédentes auraient pu s'écrire :

```
int mini ; // un entier pour le mini
int maxi ; // un entier pour le maxi
```

Cette forme développée prend plus de place, mais elle permet d'ajouter un commentaire à chaque déclaration (ce qui facilite les modifications futures). On peut également initialiser les variables au moment où on les déclare.

```
int mini = 0 ;
int maxi = 10 ;
char c = 'c' ;
```

2.3. Affectation

Toute variable déclarée, doit être initialisée. Si l'initialisation n'est pas incluse dans la déclaration (comme cela a été fait ci-dessus) , il faut pas exemple ajouter cette initialisation :

```
mini = 0 ;
maxi = mini + 10 ;
c = 'c' ;
```

Lors des affectations, il vous faut respecter le *type* ou faire appel au *transtypage*.

3. Méthodes

L'élément de structuration en java, comme dans les autres langages orientés *objet* est la *class*. Les classes sont composée d'*attributs* et de *méthodes*. Java est fourni avec une API (Application Programming Interface) qui contient un ensemble de classes. Vous trouverez la documentation de l'API à l'url suivante: <http://docs.ens.src>.

3.1. Écrire à l'écran

Pour écrire à l'écran nous allons utiliser des méthode de l'API.

En suivant la documentation dans le package `java.lang` vous trouvez une classe `System`.

La classe `System` possède un attribut *static* nommé `out` de type `PrintStream`, un attribut *static* est accessible via le nom de la *class* suivie d'un point puis du nom de l'attribut. Ainsi nous pouvons y accéder via

```
System.out
```

La classe `PrintStream` possède un ensemble de méthode *static* (de *class*) dont des `println()`,

```
int a = 4;
System.out.println("Il est "+a+" heure");
```

nous affichera à l'écran, il est 4 heures. Tout comme pour l'attribut, l'accès à une méthode se fait avec le point.

3.2. Lire au clavier

Pour lire au clavier, nous utiliserons la *class* `scanner` présente dans le *package* `java.util`.

Cette classe ne possède pas d'attributs ou de méthode *static*, il nous faut donc en créer une instance :

```
Scanner clavier; //créer un objet de type scanner
clavier = new Scanner(System.in); //instancie clavier et le lie à l'entrée standard (le clavier)
int a;
a=clavier.nextInt(); //utilisation de la méthode nextInt() pour lire sur l'entrée standard.
```

3.3. Un petit programme pour lire et écrire au clavier

Dans le dossier `code/tests`, créer une *class* `LireEcrire` qui lit un entier puis un double et enfin affiche la somme des deux à l'écran. L'ossature de cette class ressemblera à :

```
import java.util.Scanner;

public class LireEcrire
{
    public static void main(String[] args)
    {
```

```
}
}
```

L'*import* permet au compilateur de savoir où se trouve la *class* *Scanner*.

N'oubliez pas de compiler dans le répertoire courant en déposant vos bytecode dans *code/bin*, puis de tester dans *code/bin*.

3.4. Méthode statiques

Pour continuer à prendre des habitudes de programmation, nous allons séparer notre code en deux parties : les méthodes et leur test.

Vous aller créer dans */code/conditionnelles* une class *Tp1* qui contiendra la méthode *static int prixPizza(int a)* dont le code est le suivant :

```
public class Tp1
{
//calcul le prix de nb pizza
//sachant que la pizza vaut 10€ et que la dixième est gratuite
//
static int prixPizza(int nb)
{
final int prixU=10;
return nb*prixU-((nb/10)*prixU);
}
}
```

Pour ne pas avoir à naviguer entre les répertoire, vous allez vous placer dans *code* compiler *conditionnelles/Tp1.java* et placer le résultat dans *code/bin* :

```
javac -d ../bin Tp1.java
```

Le programme que nous avons ne peut-être lancé, il ne contient qu'une méthode qui n'est pas un *main*. Nous allons créer une *class TestTp1* dans *code/test* dont le code est le suivant :

```
public class TestTp1
{
public static void main(String[] args)
{
System.out.println("10 pizzas => " + Tp1.prixPizza(10));
System.out.println("20 pizzas => " + Tp1.prixPizza(20));
}
}
```

La compilation est obtenue en utilisant la ligne de commande suivante depuis le répertoire *code* :

```
javac -classpath ".;bin" -d bin tests\TestTp1.java
```

Nous voyons apparaître le *classpath* qui indique au compilateur où trouver *Tp1.class*. Tester avec :

```
java TestTp1
```

Nous pouvons pour ne pas avoir à spécifier à chaque fois le *classpath* définir le variable d'environnement *CLASSPATH* de Windows avec la valeur *.;bin*. Puis tester dans un nouveau terminal avec

```
javac -d bin tests\TestTp1.java
```

, le nouveau terminal est nécessaire car les variables d'environnement ne sont prise en compte qu'a l'ouverture.

Modifier la *class TestTp1* pour qu'elle demande une lecture au clavier du nombre des pizzas.

4. Introduction à eclipse

Maintenant que nous avons compris la notion de compilation et d'édition de liens, nous allons utiliser l'IDE eclipse :

1. Créer un espace de travail, le workspace sur votre partage netBios (z:), nommé *workspace_inf120*.

2. Lancer eclipse (le classique et non le web) et choisir le workspace précédent.
3. Créer un projet java nommé *tp*.
4. Dans ce projet en utilisant le "drag and drop" depuis l'explorateur et le répertoire `code` vers eclipse et le répertoire `src`, copier les répertoires : `boucles`, `conditionnelles`, `perso`, `exemples` et `tests`.

vous devez avoir un certain nombre d'erreurs, à vous de les corriger en sachant que pour maintenir l'édition de lien, les classes doivent avoir en entête leur déclaration de paquetage par exemple `package tests;` pour la classe `Hello.java` qui se trouve dans le paquetage `tests`. De plus les classes doivent avec `import` indiquer les classes dont elles ont besoins *En cliquant sur la croix du problème eclipse peut vous proposer des solutions..*

5. Conditionnelles

5.1. Element de syntaxe

La comparaison se fait de la manière suivante :

```
if (testBooléen) // testBooléen est un booléen
                //ou un test dont le résultat est une valeur booléenne
traitement à effectuer si le test est vérifié
[else traitement à effectuer si le test n'est pas vérifié]
```

Ce qui est entre [] est optionnel. On l'utilise si on en a besoin. Exemple :

```
if (maxi > 5) // ici testBooléen est un test
            //dont le résultat est true si maxi est supérieur à 5, false sinon.
System.out.println("maxi est plus grand que 5");
else
System.out.println("maxi est plus petit ou égal que 5");
```

Je vous conseil d'utiliser systématiquement les accolades pour définir un bloc: if (cond) {...} else {...}

5.2. Exercices

5.2.1. Remise (Conditionnelle simple)

Écrire une méthode static *remise1* de Tp1 qui reproduise l'algorithme suivant :

```
Methode remise(montant: reel):reel
const tauxRemise <- 0.01 : réel
Si (montant>300) Alors
  montant <- montant * (1-tauxRemise)
Fin Si
Renvoie(montant)
Fin
```

Modifier la *class TestTp1* pour tester la méthode static *remise*.

5.2.2. Remise (Conditionnelles multiples)

Écrire une méthode static *remise2* de Tp1 qui reproduise l'algorithme suivant :

```
Methode remise2(montant: reel):reel
  var tauxRemise : réel
  var res : réel
  Si (montant>750) Alors
    tauxRemise <- 0.02
  Sinon
    Si (montant>300) Alors
      tauxRemise <- 0.01
    Sinon
      tauxRemise <- 0
  Fin Si
```

```

Fin Si
res <- montant * (1-tauxRemise)
renvoie(res)
Fin methode

```

5.2.3. Maximum de trois entiers

Implanter la méthode static suivante (donnée en TD).

```

Methode max(a: entier, b:entier, c:entier):entier
var max : entier
max <- a
Si (b>=max)
max <- b
Fin Si
Si (c>=max)
max <- c
Fin Si
renvoie(max)
Fin methode

```

5.2.4. Estimation du prix de revient d'un véhicule

Il existe un barème pour l'évaluation du prix de revient kilométrique des véhicules. Écrire une méthode *prixRevient* de la *class Tp2* effectuant le calcul de ce prix en fonction de *nbKm*, nombre de kilomètres parcourus . Règles :

Tableau 3.1. Prix de revient du véhicule

nb de km / puissance fiscale	5CV	6CV
jusqu'à 5000	nbKm * 0,43	nbKm * 0,47
de 5001 à 20000	nbKm * 0,23+1000	nbKm * 0,27 + 1000
au delà de 20000	nbKm*0,28	nbKm * 0,32

Ecrire une *class TestTp2* dans *code/tests* et tester.

5.2.5. Convertisseurs Euros/Francs

Vous pouvez tester le corrigé (après avoir téléchargé le cher : Convertisseur.jar)

```
java -jar Convertisseur.jar
```

1. Ecrire une methode qui résolve le problème de la conversion de francs <-> euros
2. Traduire cette méthode (sur papier)
3. Coder puis compiler
4. Tester

5.2.6. Monnayeur

Nous allons coder deux monnayeurs qui rendent des pièces de 5, 2 et 1 €.

5.2.6.1. Caisse illimitée

Ecrire et tester une méthode static *caissellimite* de *Tp2* qui ne renvoie rien et qui affiche le nombre de pièce de 5, de 2 et de 1 à rendre.

5.2.6.2. Caisse limitée

Ecrire et tester une méthode static *caisseLimite* de *Tp2* qui renvoie true si la monnaie est disponible et qui affiche le nombre de pièce de 5, de 2 et de 1 à rendre.

5.2.7. Comparaison de durées

On considère des durées notées avec des valeurs entières en heures, minutes et secondes (h,m,s). Exemple : $d=(3,5,1)$: 3 heures, 5 minutes et 1 seconde. Après lecture de deux durées d_1 et d_2 dans la *class TestTp2* :

- Écrire puis implanter une première méthode static (*duree1* de *Tp2*) qui détermine la durée la plus courte, en convertissant les deux durées en secondes.
- Écrire puis implanter une deuxième static méthode (*duree2* de *Tp2*) qui réalise la même action, mais sans convertir en seconde

Les méthodes doivent retourner -1 si la première date est plus courte, 0 si égalité et 1 sinon.

5.2.8. Problèmes du test d'égalité de flottants

Avant de passer au tp suivant consacré à l'étude des boucles, voici quelques exemples qui devraient vous prouver l'intérêt des entiers dans le comptage du nombre d'itérations. Il vous est demandé de compiler et exécuter et de comprendre les programmes suivants :

- Flottant1
- Flottant2
- Flottant3

Ces trois exemples sont tirés d'un point technique disponible sur le site <http://java.sun.com>. Des explications complémentaires sont disponible dans la section 4.2.3 des spécifications du langage java et dans les documents IEEE 754 suivants.

5.2.9. Tri

Écrire un méthode static *tri* de *Tp2* qui lit trois valeurs entières (*a*, *b* et *c*) et qui affiche ces trois valeurs dans un ordre croissant.

6. Boucles

6.1. Syntaxe

En java les structures de boucle sont le *while* et le *for*. Nous verrons le *for* plus tard. Le *while* s'utilise de deux manières différentes :

```
while (condition) {
    instructions
}

ou

do{
    instructions
}
while (condition);
```

La boucle est exécutée tant que la condition est vérifiée.

6.2. Première Boucle

Vous aller créer une *class TestTp4* dans *code/tests* dans laquelle vous allez traduire l'algorithme suivant :

```
Algo Boucle
var a : entier
Début
a <- 5
```

```
Tant que (a > 0)
  Ecrire(a)
  a <- a - 2
Fin Tant que
Fin
```

Modifier le code, pour pouvoir lire la valeur de a. Faire différents tests avec a = 1, -1,... Modifier la condition ; par exemple a = 0. Tester avec a = 2, 3,... Commentaires ?

Astuce

CTRL+C vous permet de sortir d'une boucle sans fin.

6.3. Lecture de caractère

Pour cette nouvelle partie, nous allons créer dans `code/boucles` une `class` `Tp3` qui contiendra nos méthodes `static` et continuer avec la `class` `TestTp4` dans `code/tests` qui nous permettra de tester nos méthodes.

Commençons, par lire un caractère et l'afficher jusqu'à ce que l'on saisisse le caractère 'y'. Faire deux versions, l'une où ce dernier caractère est affiché à l'écran, l'autre où il ne l'est pas.

La `class` `Scanner` ne possède pas de méthode `nextChar()`, nous allons contourner le problème en lisant une chaîne de caractères et en récupérant le premier caractère avec le code suivant :

```
Scanner cl = new Scanner(System.in);
char c = cl.next().charAt(0);
```

6.3.1. Lecture avec affichage du 'y' final

La méthode `static` ne prend pas de paramètres et ne retourne rien.

```
public static void LireChar1()
```

6.3.2. Lecture sans affichage du 'y' final

La méthode `static` ne prend pas de paramètres et ne retourne rien.

```
public static void LireChar2()
```

6.4. Somme des n premiers entiers

Ecrire une méthode `static` qui retourne la somme des n premiers entiers.

```
public static int somme(int n)
```

6.5. Remboursement d'emprunt

Un emprunt ne peut être remboursé que si le remboursement annuel est supérieur au coût annuel de l'emprunt : $\text{emprunt} * \text{taux}$.

Chaque année, la valeur de l'emprunt est augmentée de son coût annuel et diminuée du remboursement.

Un emprunt est terminé lorsqu'il n'y a plus rien à rembourser.

6.5.1. Calcul du nombre d'année

Calculer le nombre d'années nécessaires au remboursement d'un emprunt à taux d'intérêt fixe et dont le remboursement annuel est fixe également. (Attention : le remboursement de la première année doit être strictement supérieur à l'intérêt payé la première année).

La méthode `static` devra retourner -1 si l'emprunt ne peut-être remboursé et le nombre d'années nécessaires sinon.

```
public static int remboursement1(double emprunt, double taux, double remboursement)
```

6.5.2. Calcul du coût

Même exercice, mais avec calcul du taux d'intérêt effectif à savoir la parts des interêts dans la somme totale payée (somme des intérêts divisée par l'emprunt initial).

```
public static double remboursement2(double emprunt, double taux, double remboursement)
```

6.6. Calcul de maximum

On lit des entiers jusqu'à lire la valeur -1. Déterminer la valeur maximale des valeurs lues (sans tenir compte du -1).

Exemple 3.2. Exemple séquence

```
4
17
13
-6
-1
Valeur max : 17
```

La fonction ne produira pas d'affichage et retournera le maximum :

```
public static int max()
```

6.7. Devine

Écrire deux méthodes qui fassent deviner en un nombre d'essais limités ou illimités un entier *aTrouver*. Pour chaque valeur entrée au clavier par l'utilisateur, le programme renvoie des indications : trop grand, trop petit, gagné et perdu (dans le cas d'un nombre d'essais limités). Pour tirer un nombre entier au hasard vous pourrez utiliser :

```
int aTrouver = (int) (Math.random()*plage);
```

```
Math.random()
```

permet d'obtenir un nombre réel, compris entre 0 et 1, tiré au hasard.

```
Math.random()*plage
```

permet d'obtenir un nombre réel, compris entre 0 et *plage*, tiré au hasard. Enfin

```
(int) (Math.random()*plage)
```

permet le transtypage vers un entier.

6.7.1. Sans limite

Ecrire une première version avec un nombre d'essais illimité.

```
public static void devine(int plage)
```

6.7.2. Avec un nombre de coup limité

Ecrire une version avec avec *nbEssai* autorisés.

```
public static void devine(int plage, int nbEssai)
```

6.8. Décomposition en facteurs premiers

Pour cette nouvelle et dernière partie, nous allons créer dans `code/boucles` une *class* `Tp4` qui contiendra nos méthodes *static* et une *class* `TestTp4` dans `code/tests` qui nous permettra de tester nos méthodes.

Décomposer un nombre en nombre premiers. Essayer les divisions du nombre par les tous les entiers (à partir de 2) et faire afficher simplement les différent diviseur.

N.B. Pour simplifier, on effectue les divisions du nombre par tous les entiers, qu'ils soient premiers ou non, de toute façon, un nombre qui n'est pas premier ne pourrait diviser car tous ses diviseurs (plus petit que lui) auraient précédemment divisé le nombre.

Exemple : pour 20, on exécute les divisions par 2 (oui), puis par 2 (oui), puis par 2 (non), puis par 3 (non), puis par 4 (non), puis par 5 (oui), ... On voit bien qu'on ne peut plus diviser par un nombre non pair (4) parce que le nombre de départ a déjà été divisé par ses diviseur (2 et 2).

La méthode prendra en paramètre le `nombre` à décomposer et ne retournera rien, les affichages se feront dans la méthode.

```
public static void decomposition(int nombre)
```

6.9. Monnayeur

Nous allons faire deux versions du monnayeur, la première avec une caisse illimitée et l'autre avec un caisse limitée. Les monnayeur ne connaissent que les pièces de 5, de 2 et de 1. Ils cherchent à rendre en premier les pièces de 5 puis celles de 2 et enfin celles de 1. Les implémentations seront basées sur des boucles.

6.9.1. Avec caisse illimitée

En vous aidant de l'algorithme suivant :

```
Algo Monnayeur
  var somme : entier
  var nb5,nb2,nb1 : entier
Début
  Lire(somme)
  nb5 <- 0 //initialisation
  nb2 <- 0 nb1 <- 0
  Tant que(somme >= 5)
    nb5 <- nb5 + 1
    somme <- somme - 5
  Fin Tant que
  Tant que(somme >= 2)
    nb2 <- nb2 + 1
    somme <- somme - 2
  Fin Tant que
  Tant que(somme >= 1)
    nb1 <- nb1 + 1 // on aurait pu faire une conditionnelle
    somme <- somme - 1
  Fin Tant que
  Ecrire("Il faut rendre : ")
  Ecrire(nb5 + "jetons de 5")
  Ecrire(nb2 + "jetons de 2")
  Ecrire(nb1 + "jetons de 1")
Fin
```

Ecrire une fonction qui prend en paramètres la somme à rendre. C'est fonction affichera, le nombre de pièces de 5, de 2 et de 1 rendues.

```
public static void monnayeur(int somme)
```

6.9.2. Avec caisse limitée

En vous aidant de l'algorithme suivant :

```
Algo Monnayeur
  var somme : entier
  var nb5,nb2,nb1 : entier
  var nb5Dispo, nb2Dispo, nb1Dispo : entier
Début
  Lire(somme)
  Lire(nb5Dispo)
  Lire(nb2Dispo)
  Lire(nb1Dispo)
```

```

nb5 <- 0 //initialisation
nb2 <- 0
nb1 <- 0
Tant que(somme >= 5 et nb5 < nb5Dispo)
  nb5 <- nb5 + 1
  somme <- somme - 5
Fin Tant que
Tant que(somme >= 2 et nb2 < nb2Dispo)
  nb2 <- nb2 + 1
  somme <- somme - 2
Fin Tant que
Tant que(somme >= 1 et nb1 < nb1Dispo )
  nb1 <- nb1 + 1
  somme <- somme - 1
Fin Tant que
Si (somme = 0) Alors
  Ecrire("Il faut rendre : ")
  Ecrire(nb5 + "jetons de 5")
  Ecrire(nb2 + "jetons de 2")
  Ecrire(nb1 + "jetons de 1")
  nb5Dispo <- nb5Dispo - nb5
  nb2Dispo <- nb2Dispo - nb2
  nb1Dispo <- nb1Dispo - nb1
Sinon
  Ecrire("Impossible")
Fin Si
Fin

```

Ecrire une fonction qui prend en paramètres la somme à rendre, le nombre de pièces de 5, de 2 et de 1 disponibles qui retourne vraie si c'est possible et faux si c'est impossible. C'est fonction affichera, lorsque c'est possible, le nombre de pièces de 5, de 2 et de 1 rendues.

```
public static boolean monnayeur(int somme, int nb5Dispo, int nb2Dispo, int nb1Dispo)
```

6.10. Boucle "for"

La boucle *for* et une écriture condensée du *while*.

```

for (init;cond;post-traitement)
{
  traitement
}

équivalent à

init;
while (cond)
{
  traitement
  post-traitement
}

```

L'utilisation la plus courante est la suivante :

```

for (int i = 0; i < limite; i++)
{
  // instructions à exécuter
}

```

Dans cet exemple, *i* prendra les valeurs 0, 1, ..., limite-1

Astuce

Il n'est pas nécessaire d'incrémenter la variable de boucle (*i*) dans la boucle

Si l'initialisation ou le post-traitement contiennent plusieurs instructions, il faut les séparer par des virgules.

6.10.1. Affichage des n premiers entier

Ecrire un algorithme à l'aide d'une boucle *for* qui affiche les n premiers entiers (de 1 à n ou de n à 1), le traduire, puis le tester.

```
public static void nPremiersEntier(int n)
```

6.10.2. Somme des n premiers entier

Ecrire un algorithme à l'aide d'une boucle *for* qui affiche la somme des n premiers entiers (de 1 à n ou de n à 1), le traduire, puis le tester.

```
public static int sommeNPremiersEntier(int n)
```

6.10.3. Placement

Ecrire l'algorithme puis le programme répondant à la question suivante : Si l'on place `somme` au 1 janvier de l'année `anDepot` à `taux%` (en accumulant les intérêts), quelle va être la somme présente sur le compte le 1 janvier de de l'année `anRetrait` ? Les variables `somme`, `anDepot`, `anRetrait` et `taux` sont à passer en paramètres.

```
public static double placement(double somme, int anDepot, int anRetrait, double taux)
```

Glossaire

Algorithmique

Algorithme	Suite nie séquentielle de règles que l'on applique à un nombre ni de données, permettant de résoudre des classes de problèmes semblables (petit Robert). Calcul, enchaînement des actions nécessaires à l'accomplissement d'une tâche. Transformation d'un problème en une suite ordonnée d'opérations comportant un nombre ni d'étapes permettant de résoudre le problème (Vaccari).
Programme	Suite d'instructions données à l'ordinateur et codées dans un langage de programmation compréhensible (après traduction) par l'ordinateur.
Instruction	On distingue deux types d'instructions : <ul style="list-style-type: none">• des instructions de traitement de l'information• des instructions destinées à commander le déroulement du programme

Variables et types

Donnée	Toute donnée est soit une variable, soit une constante.
Variable, constante	Une variable est une donnée dont la valeur va évoluer lors de l'exécution du programme. À l'opposé, une constante va conserver la même valeur tout au long du traitement. Chaque donnée (variable ou constante) est définie par : <ul style="list-style-type: none">• un identificateur : c'est le nom que l'on lui donne• un type• une valeur
Type	La valeur d'une variable est codé en binaire. À une nature (numérique, caractère,...) de l'information mémorisée correspond généralement une manière de coder cette information. Le type d'une variable permet d'associer entre eux, la nature des informations, le codage mais aussi les limites et opérations associés.
Opérateur	Un opérateur est un symbole indiquant une opération à effectuer. Les opérateurs sont bien souvent binaires reliant deux opérands , mais peuvent aussi être unaires ou ternaires. Un opérateur est bien souvent associé à un type de données. On peut classer la plupart des opérateurs dans trois grandes familles : <ul style="list-style-type: none">• opérateurs arithmétiques : donnent un résultat numérique à partir d'opérands numériques (addition, soustraction,...)• opérateurs relationnels : donnent un résultat logique à partir d'opérands numériques (plus grand que, égal, ...)• opérateurs logiques : donnent un résultat logique à partir d'opérands logiques (et logiques, ...)

Instruction

Commentaire	phrase ou mot d'un code source ou d'un algorithme sans sens (ignoré) pour le l'algorithme ou le programme, son objectif est d'éclairer le lecteur.
-------------	--

Déclaration	<p>L'utilisation d'une variable ou d'une constante sera TOUJOURS précédé d'une déclaration. Cette déclaration permet :</p> <ul style="list-style-type: none"> • d'associer formellement un identificateur à un type • d'éviter l'utilisation multiple d'un nom pour deux données différentes (double déclaration).
Affectation	<p>Une affectation $x \leftarrow \text{expression}$ est une instruction qui permet de spécifier qu'au moment de son exécution, la variable x recevra comme nouvelle valeur la valeur de l'expression expression spécifiée en partie droite de l'instruction.</p>
Expression	<p>Une expression permet de désigner le calcul d'une nouvelle valeur à partir d'autres valeurs et d'opérations. Les valeurs utilisés dans l'expression peuvent être des constantes, des valeurs de variables, des données littérales, ...</p>
Entrées/sorties	<p>Les dispositifs d'entrée/sortie permettent à l'ordinateur de communiquer avec l'extérieur. Ces dispositifs sont très importants, du clavier à l'écran. Nous aurons deux instruction d'entrées/sorties :</p> <p>lire(x) pour attendre la saisie d'une valeur saisie au clavier affectée à x.</p> <p>Ecrire(x) pour afficher la valeur de x sur l'écran.</p>

Structure de contrôle

Structure de contrôle	<p>Une structure de contrôle est une instruction destinée à commander le déroulement du programme.</p>
Enchaînement/Séquence	<p>Structure de base, qui permet d'enchaîner deux instructions.</p>
Conditionnelle	<p>Appelée aussi choix ou alternative, cette structure permet d'exécuter une série d'instructions plutôt qu'une autre en fonction du résultat d'un test (expression booléenne).</p>
Boucle	<p>Une boucle est une structure de contrôle destinée à exécuter une portion de code plusieurs fois de suite.</p>

Fonction procédure

Sous-algorithme/sous-programme	<p>Un sous programme est un ensemble d'instructions, calculant un certain nombre de résultat en fonction d'un certain nombre de données. On appelle argument ou para- mètre les données et résultats du sous programme.</p>
Définition d'un sous-programme	<p>La définition d'un sous-programme comporte :</p> <ul style="list-style-type: none"> • une signature ; l'indication de son nom, du nombre et types des paramètres à fournir et éventuellement type du résultat, • un corps ; la description des actions à enchaîner.
Paramètre formel	<p>Il s'agit de la variable utilisée dans le corps du sous-programme.</p>