

```

algorithm friendsAtHome()
    boolean friendsAreThere = true
    while friendsAreThere
        if noMoreCoffee then
            makeCoffee()
        friendsAreThere = lookArround()

```

```

algorithm makeCoffee()
    // Check ingredients
    if !hasCoffeePowder OR !hasFilter then
        print Missing ingredients
    else
        putWaterInPercolator
        putFilterInPercolator
        putCoffeeInFilter
        pushOnButton

```

version 0.1
septembre 2018

```

algorithm lookArround() → boolean
    // Not yet implemented TODO

```

Pseudocode

Équipe DEV1



Un traitement, un mot, un algorithme	2
Les types	3
Les structures alternatives, if	4
Les structures répétitives	6
Les paramètres et la valeur de retour	7
Les interactions avec l'utilisateur	8
Les commentaires	8
Exemple	9

Le **pseudocode** est une manière de décrire un algorithme en langage presque naturel. C'est un ensemble de phrases représentant l'enchaînement des opérations nécessaires à la résolution du problème.

Par exemple, un jardinier pourrait dire:

Tant que l'on est pas arrivé à la fin de la route, faire un trou et repiquer un poireau.

Nous allons formaliser un peu tout ça.

On pourrait croire à priori, qu'il existe autant de pseudocodes que de personnes décrivant un algorithme. C'est un peu vrai... et faux. Dès lors que l'on veut décrire un algorithme, c'est pour le partager avec d'autres... Ceci implique que certaines règles soient définies.

Nous présentons ici ce que nous utilisons dans les notes et pensons être le sous-ensemble minimal de règles à respecter pour ne pas être (trop) ambigu et pour ne pas devoir *apprendre* le pseudocode. Ceci dit, si le lecteur d'un algorithme trouve que c'est ambigu, c'est ambigu.

Un traitement, un mot, un algorithme

Pour faire appel à un traitement, une opération, un algorithme, nous utilisons un mot en *mixedCase*.

Un mot en *mixedCase* est un mot composé de plusieurs mots ! Collés. Chaque mot commençant par une majuscule. Excepté le premier. Par exemple: `faireUnTrou`, `remplirLeFiltre`, `putLeekInHole`...

```
faireUnTrou()  
remplirLeFiltre()  
putLeekInHole()
```

pseudocode

Remarque:

- si nous écrivons `faireUnTrou()`, nous comprenons *faire un trou*. L'usage des parenthèses n'est pas toujours utile.

Certaines actions sont des actions élémentaires qui ne demandent aucune explication, d'autres sont plus complexes et doivent être expliquées. Elles le sont dans un algorithme. Un algorithme est une suite d'opérations... qui sont des actions élémentaires ou des opérations plus complexes qui doivent être expliquées... et ainsi de suite.

Définir un algorithme, c'est:

- lui donner un nom représentatif de ce qu'il fait;
- commencer par le mot **algorithm** (ou **algorithme** ou encore **algo** voire rien);
- **indenter** les opérations de manière à **marquer clairement le bloc** d'opérations (avec une ligne verticale blanche, ou au crayon, ou avec des accolades, ou sans, ou... du moment que l'ensemble est cohérent).

```
algorithm plantOneLeek()  
    makeHole()  
    putLeekInHole()
```

pseudocode

Les types

Nous voulons distinguer les nombres entiers des nombres décimaux. Nous voulons distinguer (parfois) les caractères des chaînes de caractères. Nous voulons pouvoir représenter des tableaux et des types plus complexes, définis par le ou la développeur·euse.

Nous utiliserons:

- **integer** pour les nombres entiers et comprenons *int*, *long*, *entier*...
- **real** pour les nombres décimaux et comprenons *double*, *float*, *réel*, *pseudoréel*...
- **char** pour les caractères et comprenons *character*...;
- **string** pour les chaînes et comprenons *chaîne* et ne nous inquiétons pas de la casse;
- **[]** pour les tableaux. Ainsi, nous noterons un tableau d'entiers: *integer[]*;

Pour déclarer une variable:

```
real beautifulReal
```

pseudocode

Pour l'initialiser, lui donner une valeur, nous utilisons naturellement le symbole **=**:

```
beautifulInteger = 7
```

pseudocode

Remarque

- nous utilisons `=` pour l'assignation et `==` pour tester l'égalité. Nous comprenons l'usage de `←` pour l'assignation... mais nous déconseillons d'utiliser `=` pour tester l'égalité.

Pour les types plus complexes nous utiliserons le mot **structure** (et nous comprenons le mot classe):

```
structure StructureName
  type1 fieldName1
  type2 fieldName2
  ...
  typeN fieldNameN
```

pseudocode

Les structures alternatives, if

Pour représenter le **si** (**if**) nous utiliserons cette notation:

```
if condition then
  | statement
```

pseudocode

où:

- **condition** est une expression booléenne... une expression vraie ou fausse;
- **statement** est une instruction (une opération) ou plusieurs.

Exemple:

```
if thereAreLeeks then
  | plantOneLeek
```

pseudocode

Remarques:

- dans les notes, nous utiliserons l'anglais mais le français est bien aussi;
- il est important de marquer le bloc d'instructions. Nous utilisons une barre verticale mais un **endif** ou des accolades pourrait faire l'affaire;

- le mot *then* peut être omis ou remplacé par des parenthèses lorsque la condition ne s'étend pas sur plusieurs lignes par exemple;
- nous utilisons **if-then** mais nous comprenons **si-alors**¹;
- ...

Nous écrirons les autres structures conditionnelles comme suit:

```
if condition then
| statement
else
| statement
```

pseudocode

```
if condition then
| statement
else if condition then
| statement
else
| statement
```

pseudocode

À nouveau nous comprenons si l'on délimite les blocs par des accolades et les expressions conditionnelles par des parenthèses².

La structure **switch**

Le *selon que* (**switch**), s'écrit:

```
switch dayNumber
| case 1: dayName = "lundi"
| case 2: dayName = "mardi"
| case 3: dayName = "mercredi"
| case 4: dayName = "jeudi"
| case 5: dayName = "vendredi"
| case 6: dayName = "samedi"
| case 7: dayName = "dimanche"
```

pseudocode

¹Nous comprenons aussi **if-alors** ou **si-then**. . . mais bon, *faut pas pousser* !

²Et pas l'inverse parce qu'aucun langage ne fait ça.

Remarques:

- nous utilisons *switch* mais nous comprenons *selon que*;
- nous ajoutons un **case** pour chaque cas mais nous comprenons *cas* ou l'utilisation d'un tiret ou autre;
- en langage Java le **switch** est associé au **break**. Nous n'en utilisons pas mais nous comprenons s'il y en a.

... du moment que l'ensemble est cohérent.

Les structures répétitives

Nous formalisons les structures répétitives les plus courantes: *tant que* (*while*), *faire tant que* (*do while*) et *pour* (*for*).

```
while condition
|
| statement
```

pseudocode

```
do
| statement
while condition
```

pseudocode

Le **for** est utilisé de deux manières différentes; pour parcourir de n à m avec éventuellement un pas ou en définissant une situation de départ, un condition de fin et un incrément.

Pour un simple parcours, nous écrivons:

```
for i from n to m
| statement

for i from m to n by -1
| statement
```

pseudocode

Pour un *for* plus général, nous écrirons:

```
for initialisation ; condition ; update  
| statement
```

pseudocode

Remarques:

- nous utilisons des mots anglais mais comprenons les équivalents français;
- il est toujours possible d'ajouter des parenthèses pour accroître la lisibilité.

Les paramètres et la valeur de retour

Un paramètre est une valeur passée à un algorithme. Il vient entre parenthèses après le nom de l'algorithme. Il peut être *en entrée* et ne sera pas modifié par l'algorithme ou *en entrée-sortie* auquel cas, il pourra être modifié par l'algorithme. Il peut y en avoir plusieurs, de sortes différentes, séparés par une virgule.

- en entrée, il peut être affublé d'une flèche:

```
algorithm algorithmName(paramName ↓ : type)  
| statement
```

pseudocode

- en entrée/sortie, il sera affublé d'une double flèche:

```
algorithm algorithmName(paramName ↓↑ : type)  
| statement
```

pseudocode

La valeur de retour est la valeur que retourne l'algorithme. Elle n'est pas obligatoire. Nous la signalons pas une flèche « \rightarrow » et l'algorithme devra se terminer en « retournant » une valeur en utilisant **return**.

```
algorithm algorithmName() → type  
    statement  
    return expression
```

```
algorithm algorithmName(paramName1 ↓ : type paramName2 ↓↑ : type ) → type  
    statement  
    return expression
```

pseudocode

Les interactions avec l'utilisateur

Pour faire une lecture « au clavier », nous utiliserons simplement **read** et pour une écriture à l'écran, **print**.

```
read a  
read "Entre une valeur: ", a  
print b  
print "L'aire du rectangle vaut: " area
```

pseudocode

Les commentaires

Les commentaires commencent par `//`.

```
// Firts comment  
statement
```

```
// An other (righ) comment
```

pseudocode

Nous comprenons également `#`, `/* ... */`...

Exemple

```
algorithm friendsAtHome()
| boolean friendsAreThere = true
| while friendsAreThere
| | if noMoreCoffee then
| | | makeCoffee()
| | friendsAreThere = lookArround()

algorithm makeCoffee()
| // Check ingredients
| if !hasCoffeePowder OR !hasFilter then
| | print Missing ingredients
| else
| | putWaterInPercolator
| | putFilterInPercolator
| | putCoffeeInFilter
| | pushOnButton

algorithm lookArround() → boolean
| // Not yet implemented TODO
```

pseudocode