

# Algorithmique en JavaScript

## Initiation à la programmation

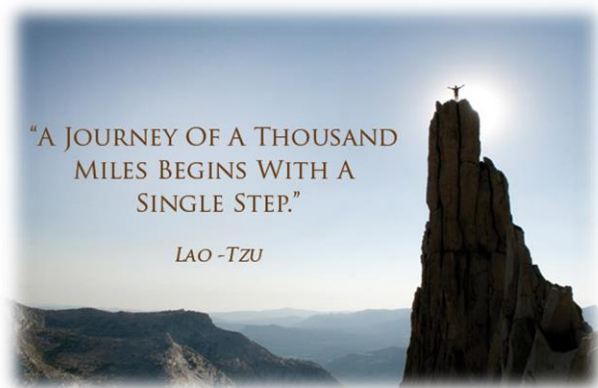
# Table des matières

- Introduction
- Les variables
- Les opérateurs conditionnels
- Les structures conditionnelles
- Les structures itératives
- Les tableaux
- Les sous-programmes
- Suppléments

# **INTRODUCTION À L'ALGORITHMIQUE**

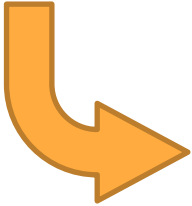
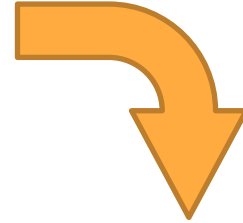
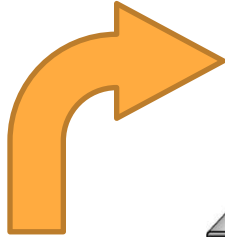
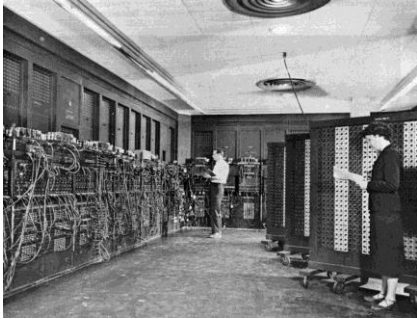
# Courage, rigueur et persévérance

- L'apprentissage des techniques et méthodes de programmation est un long processus constitué d'une multitude d'efforts à accomplir et d'embûches à surmonter !



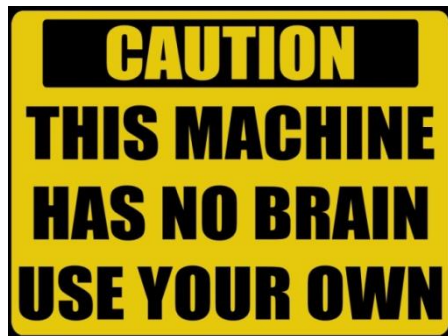
- De même, développer un programme informatique est un long chemin composé d'un grand nombre de petites étapes !

# Ordinateur, machine programmable



# Ordinateur, machine programmable

- Un ordinateur n'a pas de cerveau ! Il est donc nécessaire de lui dire tout ce qu'il doit faire.



- L'algorithmique est là pour nous aider.

# Composants importants d'un ordinateur

- Processeur : permet d'exécuter des opérations basiques comme des calculs arithmétiques. Il est composé de :
  - Une unité arithmétique et logique
  - Une unité de contrôle
  - Une horloge
  - Des registres
- Mémoire : permet à l'ordinateur d'enregistrer des informations.
- Bien qu'il y ait d'autres composants dans un ordinateur, nous nous intéresserons au processeur et à la mémoire

# La mémoire

- Zone où l'ordinateur enregistre les données dont il a besoin pour fonctionner.
- Plusieurs types de mémoire dans un ordinateur :
  - RAM (ou mémoire vive) : utilisée dynamiquement par l'ordinateur, aucune information n'est conservée une fois le système éteint. Cette mémoire est dite volatile.
  - ROM : mémoire stockant des informations de manière permanente. Cette mémoire est dite non volatile. Un exemple de cette mémoire est le disque dur.



# Algorithme ?

- Suite d'instructions précises permettant de résoudre un problème de manière automatique.
- Nous utilisons des « éléments algorithmiques » pour les construire.
- Composé de trois parties :
  - Ce que l'on donne à l'algorithme.
  - Ce qu'il fait (instructions précises).
  - Ce que l'on attend comme résultat.
- Contrairement à ce que l'on pourrait penser de prime abord, l'informatique n'est pas la première science concernée par l'algorithmique, cet honneur revient aux mathématiques.

# Recette de cuisine

- Un algorithme peut être comparé à une recette de cuisine. Elle décrit la marche à suivre, mais c'est la personne exécutant les instructions qui réalise le plat.
- Les 3 parties sont présentes :
  - Ce que l'on donne à l'algorithme : les ingrédients (pomme, poire, farine...)
  - Ce que l'algorithme fait : les instructions (découper les pommes en quart...)
  - Ce que l'on attend comme résultat : la recette réussie (crumble aux pommes et aux poires)
- Il n'y a donc rien de mystique ni de quantique dans l'algorithmique !

# Utilité d'un algorithme

- Nous retrouvons des algorithmes partout dans notre vie de tous les jours.
  - Lorsqu'un GPS calcule le chemin le plus rapide vers une destination, il utilise un algorithme
  - Une recherche sur un moteur de recherche fait appel à un algorithme
  - Lorsqu'une intelligence artificielle doit prendre une décision, cette prise de décision est basée sur un algorithme

# Instant réflexion

- À chaque utilisation de programme ou même parfois lors de certains raisonnements, vous utilisez des algorithmes potentiellement célèbres.
- Pouvez-vous en citer ? (Si ce n'est leur nom, au moins leur utilité)

# Algorithme et programme : Nuances

## ●Algorithme

- Suite d'instructions totalement indépendantes de la technologie.
- Solution abstraite à un problème donné.
- Peut être réalisé à l'aide d'un simple éditeur de texte ou même sur papier.
- Un même algorithme peut être utilisé par plusieurs programmes (qui peuvent être écrits dans différents langages)

## ●Programme

- Suite d'instructions, compréhensibles par l'ordinateur, permettant d'exécuter une tâche.
- Peut être écrit dans différents langages (dits de programmation) conçus pour être compris par l'ordinateur : C, C#, Java, Python, Ruby,...
- Un programme met en œuvre un (ou plusieurs) algorithme(s).

# Algorithme et programme : Dijkstra

- L'algorithme de Dijkstra (bien que méconnu par les non-informaticiens) est présent dans divers domaines... Et pour cause, il permet de trouver le plus court chemin entre un point A et un point B !
- Exemple ?

# Algorithme et programme : Dijkstra

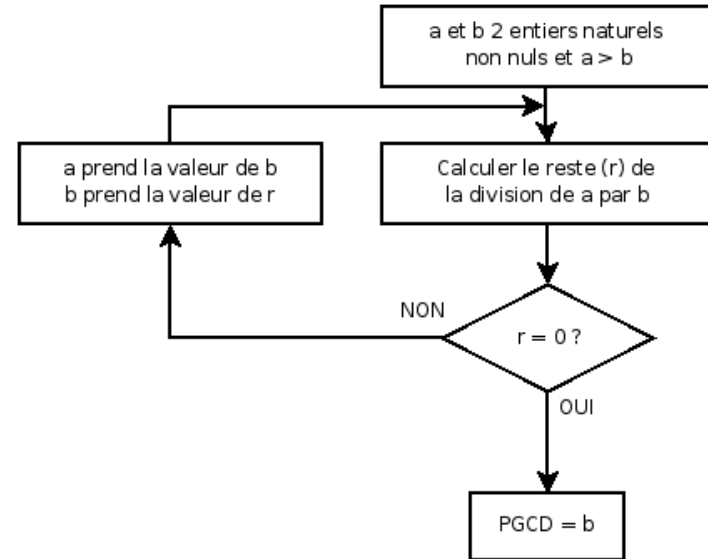
- Cet algorithme « n'est qu'une » suite d'instruction que même un humain peut suivre pour trouver le plus court chemin.
- Ceci dit, nous imaginons mal un conducteur faire lui-même cette suite d'instructions pour trouver le chemin qui lui convient (long et possibilité d'erreurs).
- Des programmeurs ont transformé cet algorithme en programmes.
- Dans les exemples suivants, nous utiliserons des algorithmes plus « simples », contenant moins d'instructions

# Représentation d'un algorithme

- Représentation des instructions de manière structurée.
- Initialement sous forme d'organigramme.
- Exemple :

Algorithme d'Euclide permettant de calculer le PGCD de deux entiers.

- Cette représentation fut rapidement abandonnée au profit du pseudo-code.





# Pseudo-code

- Représentation des étapes sous forme écrite, de manière structurée.
- La taille de l'algorithme n'influence pas la lisibilité et la compréhension du pseudo-code.
- Structure les instructions menant à la résolution du problème pour lequel l'algorithme a été élaboré.
- Exemple : algorithme d'Euclide en pseudo-code.

```
Algorithme EuclidePGCD  
  
DEBUT  
    Variable a, b, r : Entier  
    Lire(a)  
    Lire(b)  
    TANTQUE (r <- a MOD b) != 0  
    FAIRE  
        a <- b  
        b <- r  
    FINTQ  
    Ecrire("PGCD = "+ b)  
FIN
```

# Pseudo-code : Structure de base

**Algorithme** EuclidePGCD

**DEBUT**

**Variable** a, b, r : **Entier**

**Lire**(a)

**Lire**(b)

**TANTQUE** (r <- a MOD b) != 0

**FAIRE**

a <- b

b <- r

**FINTQ**

**Ecrire**("PGCD = " + b)

**FIN**

- Un algorithme est toujours divisé en trois parties distinctes :
- Le nom de l'algorithme.
- La partie déclarative. Nous y déclarons les variables et constantes (nous verrons bientôt ce qu'elles représentent).
- Les instructions de l'algorithme délimitées par les mots-clés « Debut » et « Fin ».

# Langage de programmation

- Permet de faire exécuter des instructions concrètes à un ordinateur.
- Réalise de manière concrète les instructions définies dans un algorithme.
- Il existe de nombreux langages différents. Ils possèdent des éléments communs, mais également des spécificités qui leur sont propres.

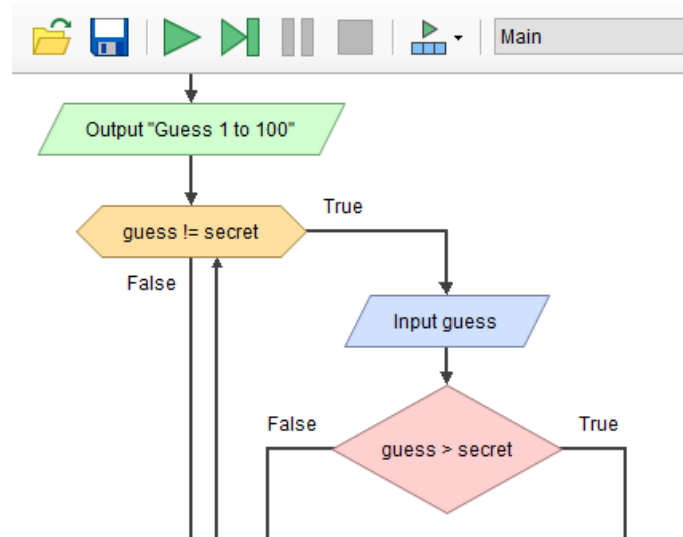
# Choix d'implémentation pour le cours : flowgorithm

- Afin de pouvoir mettre en application les notions d'algorithmique, nous utiliserons Flowgorithm

- Flowgorithm est un outil de création et modification graphiques de programmes informatiques sous forme d'ordinogramme. Il permet ensuite aux programmeurs d'exécuter ces programmes tout en pouvant en suivre graphiquement le déroulement .

- Il est disponible ici :

<http://www.flowgorithm.org/download/index.htm>



# Remarques importantes

- Il s'agit ici d'un cours d'algorithmique et non d'un cours de Java. Ce langage est utilisé à titre d'exemple afin d'illustrer de manière concrète les concepts d'algorithmique que nous allons aborder.
- Les notions d'orienté-objet ne seront pas abordées dans ce cours qui est uniquement dédié aux concepts d'algorithmique.



# Petite histoire de JavaScript

- JavaScript :

- est né en 1996
- a été développé par Netscape Communications Corporation et Mozilla Foundation
- a été créé par Brendan Eich en 1995
- a été standardisé sous le nom d'ECMAScript en 1997
- est un langage orienté objet interprété par un navigateur
- Est à la version 11

# Que peut-on faire avec lui ?

- Des scripts qui permettent de :
- Des jeux, des clients de messagerie...
- Dynamiser le front
- Accéder à des données via AJAX
- ...

# Télécharger et installer

- <https://code.visualstudio.com/>
- Cliquer sur Download
- Installer
- Lancer Visual Studio Code
- => Extension : Bracket Pair Colorizer 1.0.61 (pour mettre en couleur les parenthèses et accolades)
- => Extension : Live Server 5.6.1



# Changer le thème couleur

```
public class Demo {  
    private static final String CONSTANT;  
    private Object o;  
    /**  
     * Creates a new demo.  
     * @param o The object to demonstrate.  
     */  
    public Demo(Object o) {  
        this.o = o;  
        String s = CONSTANT + "Other";  
        int i = 123;  
    }  
    public static void main(String[] args) {  
        Demo demo = new Demo();  
    }  
}
```

- Visual Code > Preferences > Color thème > Monokai

# **ALGORITHMIQUE : LES BASES**

# LES VARIABLES

# Un monde de variables

- Un ordinateur doit retenir des informations temporaires pour exécuter des tâches.
- Une variable peut être comparée à un tiroir avec une étiquette.
  - L'étiquette permet de lui donner un nom
  - L'intérieur du tiroir permet d'y ranger l'information
- Ces variables se trouvent dans la RAM. Cette dernière est comme une immense armoire à tiroirs. En terme informatique :
  - L'étiquette du tiroir est une adresse
  - Le contenu du tiroir est une valeur

Adresse	Valeur
0	145
1	3.8028322
2	0.827551
3	3901930
...	...
3 448 765 900 126 (et des poussières)	940.5118

# Variables

- Une variable est donc un élément de programmation auquel on affecte une valeur particulière pouvant être modifiée au cours de l'algorithme
- Une variable est définie par :
  - Un nom : un identifiant unique permettant de la désigner
  - Un type : définit quel type de valeur la variable peut contenir
  - Un emplacement en mémoire RAM, représenté par son adresse
  - Une valeur : correspondant au type

# Typage des variables

- Comme expliqué précédemment, une variable est entre autres définie par le type de valeur qu'elle peut contenir.
- Il existe de nombreux types :
  - Entier : 1, 2, 5, 6, 42...
  - Réel : 0.14, 157.1547, 3.1415... ← /!\ Notez le . et non la ,
  - Caractère : 'a', 'B', '^'...
  - Texte : "Bonjour", "Je suis de type texte"...
  - Logique ou booléen : *vrai* ou *faux* (nous en reparlerons dans le chapitre sur les structures conditionnelles.)

# Déclarer une variable

- Déclarer une variable consiste à réserver un emplacement en mémoire afin de pouvoir l'utiliser par la suite.
- Pour déclarer une variable, il nous faut spécifier 2 choses : son nom et son type. Il est important de choisir un nom cohérent qui permet de facilement comprendre l'utilité de la variable.

- En pseudo-code :

*Variable* *identificateur* : *Type*

- Exemples :

*Variable* age : Entier  
*Variable* message : Chaîne

# Les constantes

- Dans nos algorithmes, nous avons parfois besoin de valeurs qui ne changent pas au cours de l'exécution.
- Exemple : la valeur du nombre  $\pi$  (pi) ou encore le pourcentage de TVA.
- Ces valeurs sont appelées des constantes, car elles ne changent pas durant l'exécution de l'algorithme.
- Voici comment déclarer une constante en pseudo-code :

```
Constante PI = 3.141592,  
          TAUX TVA = 0.21
```



# Bonnes pratiques

- Le choix des noms de variable doit rester cohérent. À la relecture de l'algorithme, il doit être possible de comprendre le sens d'une variable en fonction de son nom. L'alphabet n'est pas une bonne idée.
- Éviter les accents pour des raisons informatiques. Les langages sont écrits en anglais, et cette langue ne comprend pas les accents
- Le nom d'une variable ne peut commencer par un chiffre mais il peut en contenir
- Une bonne pratique est d'utiliser le « lowerCamelCase » ou le « UpperCamelCase ». Le choix dépend des conventions que certains langages utilisent. En JavaScript, c'est le « lowerCamelCase » qui est principalement utilisé. Et le « UpperCamelCase » pour le nom des classes.
- Pour les constantes, c'est « UPPERCASE ».

# Mots réservés en JavaScript

- Le nom d'une variable ne peut être un des mots suivants car ils ont une signification en JavaScript.

let	var	throw
case	if	while
do	function	switch

# Affectation

- Une fois une variable déclarée, il faut pouvoir lui donner une valeur. Cette opération s'appelle une affectation.
- La première affectation est appelée **initialisation**. Il s'agit d'une opération importante : on ne peut pas utiliser une variable ne possédant pas de valeur.
- En pseudo-code :
- Exemple :

```
identificateur <- valeur
```

```
message <- "Bonjour"
```

# Affectation

- L'initialisation est **primordiale** :

```
DEBUT  
  Variable a, b : Entier  
  b <- a + 2  
FIN
```

Quelle est la valeur de A ?

- Il est important de respecter le typage des variables :

```
DEBUT  
  Variable a : Entier  
  a <- "Bonsoir"  
FIN
```

**Ceci ne fonctionnera pas !**  
La variable A est de type Entier

# JavaScript : Déclaration et affectation d'une variable

JS

- En JavaScript, une variable se déclare de la manière suivante :

```
let nomVariable;
```

– Il est interdit d'utiliser des caractères accentués dans les noms de variables.

- L'affectation ne se fait pas avec une flèche, mais avec un « = ».
- Voici les différents types de variables en Java :

```
nomVariable = "valeur";
```

- Il est possible de déclarer et d'initialiser un variable en même temps

```
let nomVariable = "valeur";
```

Type Pseudo	Type Java
Entier	number
Réel	number
Caractère	string
Texte	string
Booléen	boolean

/!\  
Majuscule

# Exemple : Java

- Voici un exemple de déclaration et d'initialisation de variables :

```
let caractere = 'a';  
  
let message1 = "Bonsoir";  
let message2 = "Je m'appelle";  
let message3 = "Son \"nom\" est :";
```

```
let age = 10;  
let temperature = 15.6;  
let booleen = true;
```

- Pour afficher le contenu d'une variable :

```
// avec un retour à la ligne  
console.log(message1);
```

Astuce ☺  
log + tab

# Exercice

- exo00 - Stockez dans une variable nommé message la chaine de caractères "Bienvenue" et affichez son contenu à l'écran

# Inverser le contenu de deux variables

- exo01-Trouvez une méthode permettant d'inverser le contenu de deux variables (du même type évidemment). Si  $nb1 = 5$  et  $nb2 = 7$ , après ce traitement,  $nb1 = 7$  et  $nb2 = 5$ .



# Opérations entrées/sorties

- Il est très régulièrement nécessaire d'interagir avec un algorithme. Comme vu dans l'introduction, nous pouvons communiquer des informations à un algorithme et il peut nous en communiquer également.
- 2 opérations sont mises à notre disposition :
  - Lire, opération d'entrée
  - Ecrire, opération de sortie
- Attention ! Ces deux opérations doivent être comprises **du point de vue de l'ordinateur**. L'opération Ecrire va donc écrire une information (que nous pourrons lire sur l'écran par exemple) et l'opération Lire va enregistrer une valeur (écrite sur le clavier par exemple).

# Ecrire

- Opération permettant de communiquer une information au « monde extérieur ». Ce qui peut être :

- L'écran de l'ordinateur
- Une autre machine

- Déroulement de cette opération :

1. Lecture de la valeur de la variable spécifiée
2. Communication de cette valeur au monde extérieur

```
DEBUT
  Variable message : Chaine
  message <- "Bonne journée"
  Ecrire(message)
FIN
```

# Lire

- Opération permettant de récupérer de l'information du « monde extérieur ». Ce qui peut être :

- Saisie au clavier
- Info d'une machine (sonde de température ou autre ordinateur par exemple)

- Déroulement de cette opération :

1. Lecture de la valeur depuis la source de l'information
2. Affectation de cette valeur à une variable donnée

```
DEBUT
  Variable message : Chaine
  Lire(message)
FIN
```

# Lire et Ecrire en Java

- L'opération d'écriture est la suivante :

```
console.log("Bienvenue");
```

```
console.log(nomVariable);
```

```
console.log("Bienvenue " + nomVariable);
```

- L'opération de lecture est la suivante :

```
let nom = prompt("Entrez votre nom");
```

# Remarques : Lire en JavaScript

- Il existe une grande différence entre l'algorithmique et le JavaScript en ce qui concerne l'opération Lire.
- Cette dernière récupère la saisie au clavier sous la forme d'une chaîne de caractère (le type string). C'est d'ailleurs normal, l'utilisateur entre des touches clavier qui sont des caractères. La valeur « 123 » est la concaténation de trois caractères, le chiffre 1, le chiffre 2 et le chiffre 3. Pour récupérer la valeur numérique 123, il faut convertir les chiffres en nombre.
- Voici quelques exemples :

```
let nomVariable = prompt("Entrez ");  
let nbEntier = parseInt(nomVariable);  
let nbReel = parseFloat(nomVariable);  
let booleen = (nomVariable == "true") ? true : false;
```

# Exercice

- exo02-Récupérer le nom et le prénom du l'utilisateur puis affichez le message "Bienvenue prenom nom"
- Ex : Bienvenue Jules César

# Opérateurs

- Jusqu'à maintenant, nous avons vu plusieurs éléments importants :
  - La notion de variable
  - L'affectation
  - Les opérations d'entrées et de sorties
- Nous aimerions pouvoir effectuer des opérations dans nos algorithmes. C'est maintenant que les opérateurs entrent en jeu.

# Opérateurs

- Il existe plusieurs types d'opérateurs :
  - Opérateurs arithmétiques
  - Opérateurs de comparaisons
  - Opérateurs logiques
- Les deux derniers seront abordés un peu plus tard dans le cours.



# Opérateurs arithmétiques

- Porte sur des valeurs de type numérique (entiers ou réels).

+	Addition
-	Soustraction
*	Multiplication
/	Division
DIV	Division entière (partie entière uniquement)
MOD	Modulo (reste d'une division d'entiers)

# Opérateurs arithmétiques : Exemples

DEBUT

Variable a, b, c, d : Entier,  
e, f : Reel

a <- 10

b <- 5 DIV 2

c <- b + a

d <- 5 MOD 2

e <- 5 / 2

f <- (e - 0.5) \* a

FIN

• Dans cet exemple, nous utilisons des opérateurs entre des valeurs, mais également des variables !

• Les valeurs de chaque variable sont donc les suivantes :

Variables	Valeurs
a	10
b	2
c	12
d	1
e	2.5
f	20.0

# Opérateurs arithmétiques : Java

- Voici la table des opérateurs arithmétiques en Java.

+	Addition
-	Soustraction
*	Multiplication
/	Division
N'existe pas	Division entière => parseInt()
%	Modulo (reste d'une division d'entiers)

# Opérateurs arithmétiques : Exercices

- exo03a - Voici l'exemple précédent en JavaScript

```
let a, b, c, d, e, f;
```

```
a = 10;
```

```
b = parseInt(5 / 2);
```

```
c = b + a;
```

```
d = 5 % 2;
```

```
e = 5 / 2.0;
```

```
f = (e - 0.5) * a;
```

Variables	Valeurs
a	
b	
c	
d	
e	
f	

# Opérateurs arithmétiques : Exercices

- exo03a-Déterminez la valeur des variables de ce pseudo-code.

```
DEBUT
  Variable a, b, c, d, e :
Entier
  a <- 8 MOD 3
  b <- 4 + a
  c <- b * a
  d <- (c - a) * b
  e <- ((a + 7) * (d DIV a)) * 0
FIN
```

- exo03b-Vérifiez vos réponses en utilisant JavaScript.

- exo04-Bonus : Imaginez une méthode permettant d'inverser le contenu d'une variable entière SANS utiliser une variable temporaire.

# Un dernier pour la route

- exo05-Réalisez un algorithme convertisseur de secondes. Ce dernier reçoit un nombre de secondes et détermine le nombre de jours, heures, minutes et secondes auquel il correspond.

- Exemple :

4561 secondes correspondent à 0 jour 1 heure 16 minutes et 1 seconde.

- Réfléchissez à la méthode que nous devons utiliser.

- Une fois l'algorithme réalisé, testez-le en JavaScript.

# **LES OPÉRATEURS CONDITIONNELS**

# Les booléens

- Type de variable n'ayant que deux valeurs possibles : *vrai* ou *faux*.
- Ce type permet de connaître un état.
- Exemples :
  - Considérons un interrupteur allumé ou éteint :
  - Nous utilisons la variable `interrupteurEteint` :
    - Vrai : l'interrupteur est éteint
    - Faux : l'interrupteur est allumé
  - Un panier est vide ou rempli :
  - Nous utilisons la variable `panierRempli` :
    - Vrai : le panier est rempli
    - Faux : le panier est vide



# Comparer des variables

- Dans un algorithme, il est souvent intéressant de pouvoir comparer des variables.
- Par exemple, pour savoir si un étudiant a réussi un examen, il faut savoir si ses points sont supérieurs ou égaux à 10 (sur un total de 20).
- Les opérateurs de comparaisons sont là pour nous aider.
- Le résultat d'une comparaison sera un booléen.

# Opérateurs de comparaisons

- Comparaison de deux valeurs et obtenir une valeur logique (*vrai* ou *faux*) comme résultat.

==	Égalité stricte
!=	Différence
>	Strictement supérieur
<	Strictement inférieur
>=	Supérieur ou égal
<=	Inférieur ou égal

# Comparaisons : Exemples

- Voici un exemple dans lequel nous comparons les points de Jean et Paul pour savoir s'ils ont réussi.

DEBUT

```
Variable pointsJean, pointsPaul : Entier,  
      reussiteJean, reussitePaul : Booleen
```

```
pointsJean <- 15
```

```
pointsPaul <- 9
```

```
reussiteJean <- (pointsJean >= 10)
```

```
reussitePaul <- (pointsPaul >= 10)
```

FIN

Variables	Valeurs
reussiteJean	Vrai
reussitePaul	Faux

# Comparaison : JavaScript

•Voici l'exemple précédent en Java ainsi que les opérateurs de comparaison en JavaScript:

```
let pointsJean = 15;
let pointsPaul = 9;

let reussiteJean = (pointsJean >= 10);
let reussitePaul = (pointsPaul >= 10);
```

Variables	Valeurs
reussiteJean	true
reussitePaul	false

Pseudo-code	Java
==	==
!=	!=
>	>
<	<
>=	>=
<=	<=

# Combiner des comparaisons

- Nous désirons démarrer la chaudière lorsque la température est comprise entre 5 et 15 degrés. En considérant la variable température, deux comparaisons

apparaissent :

- (température  $\leq 15$ )

- (température  $\geq 5$ )

Pour allumer la chaudière :

(température  $\geq 5$ ) et (température  $\leq 15$ ) doivent être vrais.

Il serait donc intéressant de pouvoir combiner ses deux comparaisons.

- Les opérateurs logiques vont nous permettre de le faire

# Opérateurs logiques

- Permettent de combiner des valeurs logiques. Ce qui va nous permettre de combiner des comparaisons. Le résultat est toujours une valeur logique.

!	NON	Négation logique
&&	ET	Le 'et' logique
	OU	Le 'ou' inclusif logique

# Opérateurs logiques : JavaScript

- Voici la table des opérateurs logique en JavaScript

!	Négation logique
&&	Le 'et' logique
	Le 'ou' inclusif logique

```
let majeur = false;

if (majeur == false)
{
    console.log("Pas majeur");
}
else
{
    console.log("Majeur");
}
```

```
let majeur = false;

if (!majeur)
{
    console.log("Pas majeur");
}
else
{
    console.log("Majeur");
}
```

# Tables de vérité : ET – OU – NON

- Une table de vérité permet de modéliser le comportement d'un opérateur logique en fonction des valeurs des deux opérandes.
- Voici les tables de vérité des trois opérateurs :

ET

$A \wedge B$	Vrai	Faux
Vrai	V	F
Faux	F	F

OU

$A \vee B$	Vrai	Faux
Vrai	V	V
Faux	V	F

NON

	$\neg A$
Vrai	F
Faux	V



# Exemples

- Si  $A = \text{vrai}$  et  $B = \text{faux}$  :

- $A \text{ ET } B$  est *faux*

- $\text{NON}(B)$  est *vrai*

- $\text{NON}(A \text{ ET } B)$  est *vrai*

- L'exemple de la chaudière sera donc :

- $(\text{température} \geq 5) \text{ ET } (\text{température} \leq 15)$

- Nous pouvons donc utiliser les opérateurs logiques avec des booléens et des comparaisons.

# Lois de De Morgan

- De Morgan est un mathématicien britannique ayant formulé des propriétés d'algèbre booléenne. Ces propriétés peuvent nous être utiles.

$$\neg(A \vee B) \leftrightarrow (\neg A) \wedge (\neg B)$$

$$\neg(A \wedge B) \leftrightarrow (\neg A) \vee (\neg B)$$

$$\text{NON}(A \text{ OU } B) \leftrightarrow (\text{NON } A) \text{ ET } (\text{NON } B)$$

$$\text{NON}(A \text{ ET } B) \leftrightarrow (\text{NON } A) \text{ OU } (\text{NON } B)$$

# Lois de De Morgan : Pseudo-code

$$\text{NON}(A \text{ ET } B) = \text{NON}(A) \text{ OU } \text{NON}(B)$$

A	B	A ET B	NON(A ET B)	NON(A)	NON(B)	NON(A) OU NON(B)
0	0	0	1	1	1	1
0	1	0	1	1	0	1
1	0	0	1	0	1	1
1	1	1	0	0	0	0

$$\text{NON}(A \text{ OU } B) = \text{NON}(A) \text{ ET } \text{NON}(B)$$

A	B	A OU B	NON(A OU B)	NON(A)	NON(B)	NON(A) ET NON(B)
0	0	0	1	1	1	1
0	1	1	0	1	0	0
1	0	1	0	0	1	0
1	1	1	0	0	0	0

# XOR

- Il existe un quatrième opérateur logique :  
OU exclusif ( $\oplus$ )
- XOR vient de l'anglais : eXclusive OR
- Cet opérateur logique reste très peu utilisé en programmation.
- Essayez d'en déterminer la table de vérité, voici deux indices :
  - « l'un ou l'autre ».
  - $A \text{ XOR } B \leftrightarrow ((\text{NON}(A)) \text{ OU } B) \text{ ET } (A \text{ OU } \text{NON}(B))$

# Table de vérité du XOR

- Soit l'un, soit l'autre, mais pas les deux

XOR

$A \oplus B$	Vrai	Faux
Vrai	F	V
Faux	V	F

# Exercices

- Considérons  $A = 3$ ,  $B = 9$ ,  $C = \text{Faux}$ ,  $D = \text{NON}(C)$ ,  $E = 9$ .
- Donnez le résultat pour chacune de ces instructions :

## Notation pseudo-code :

1.  $(A > 8)$
2.  $(B == 9)$
3.  $(\text{NON}(A != 3))$
4.  $(\text{NON}(C))$
5.  $((A < B) \text{ OU } C)$
6.  $\text{NON}((A + B) != 12)$
7.  $((B == 5) \text{ OU } ((E > 10) \text{ ET } (A < 8)))$
8.  $(((((B == 5) \text{ OU } ((E > 10) \text{ ET } (A < 8)))) \text{ OU } (A < B) \text{ OU } C) \text{ ET } C)$

## Notation Java :

$A != 3$

$!(D) \parallel C$

$((A + B) == 12) \&\& D$

# **LES STRUCTURES CONDITIONNELLES**

# Les structures conditionnelles

- Souvent, un algorithme se retrouve face à plusieurs situations qui ne peuvent pas être traitées avec les mêmes instructions. Nous ne savons pas quel sera le cas de figure à l'exécution.
- Nous devons alors prévoir tous les cas possibles. Nous allons donc utiliser les structures conditionnelles.



# SI... SINON

- Reprenons l'exemple de la chaudière :

Nous désirons l'allumer si la température est comprise entre 5 et 15 degrés. Pour cela, nous avons besoin de dire :

**SI** la température est comprise entre 5 et 15 degrés **ALORS**  
allume la chaudière **SINON** éteindre la chaudière.

- Cette phrase illustre une structure conditionnelle.

# SI... SINON

- Il s'agit d'une structure conditionnelle qui va nous permettre d'effectuer des instructions différentes si une condition est vérifiée ou non.

- En pseudo-code :

```
SI [condition] ALORS  
    [instructions1]  
SINON  
    [instructions2]  
FINSI
```

# Exemples : Pseudo-code

DEBUT

Variable age : Entier

Lire(age)

SI age  $\geq$  18 ALORS

Ecrire("Bienvenue")

SINON

Ecrire("Accès interdit")

FINSI

FIN

# Exemples : JavaScript

```
let age;
age = parseInt(prompt("Entrez votre âge"));
if (age >= 18)
{
    console.log("Bienvenue");
}
else
{
    console.log("Accès interdit");
}
```

```
if (conditions)
{
    //instructions;
}
else
{
    //instructions;
}
```

NB : L'instruction  
else  
est optionnelle

# Instant Réflexion

Peut-on placer un SI..SINON dans un autre ?

# Beaucoup de conditions

- Lorsque de nombreux traitements sont possibles en fonction de la valeur d'une variable, nous pouvons imbriquer des SI..SINON, mais cela devient rapidement illisible et laborieux à écrire.
- Voyons ça avec un petit algorithme déterminant le jour de la semaine.

# SI..SINON imbriqués

```
DEBUT
  Variable jour : Entier

  Lire(jour)
  SI jour == 1 ALORS Ecrire("Lundi")
  SINON
    SI jour == 2 ALORS Ecrire("Mardi")
    SINON
      SI jour == 3 ALORS Ecrire("Mercredi")
      SINON
        SI jour == 4 ALORS Ecrire("Jeudi")
        SINON
          SI jour == 5 ALORS Ecrire("Vendredi")
          SINON
            SI jour == 6 ALORS Ecrire("Samedi")
            SINON
              SI jour == 7 ALORS Ecrire("Dimanche")
              SINON
                Ecrire("Dimanche")
              FINSI
            FINSI
          FINSI
        FINSI
      FINSI
    FINSI
  FINSI
FIN
```

Pas très intuitif



# L'indentation

- Nous commençons à voir apparaître, dans le pseudo-code, des blocs d'instructions. Afin de garder le code lisible, nous utilisons des espaces (ou des tabulations) pour structurer le code en fonction de sa logique.
- Ce concept se nomme l'**indentation**. Il est primordial de le mettre en application sous peine de vexer le formateur 😞 !
- Le slide suivant vous démontre son importance.



Algorithmique en JavaScript 81

**ILLISIBLE !!!**

```

DEBUT
  Variable jour : Entier

  Lire(jour)
  SI jour == 1 ALORS Ecrire("Lundi")
  SINON
    SI jour == 2 ALORS Ecrire("Mardi")
    SINON
      SI jour == 3 ALORS Ecrire("Mercredi")
      SINON
        SI jour == 4 ALORS Ecrire("Jeudi")
        SINON
          SI jour == 5 ALORS Ecrire("Vendredi")
          SINON
            SI jour == 6 ALORS Ecrire("Samedi")
            SINON
              SI jour == 7 ALORS Ecrire("Dimanche")
              SINON
                Ecrire("Erreur")
            FINSI
          FINSI
        FINSI
      FINSI
    FINSI
  FINSI
FINSI
FIN

```

# SI... SINONSI... SINON

- Lorsque nous avons plus de deux conditions à vérifier, nous pouvons utiliser le **SINON SI** dans la structure conditionnelle que nous avons vue précédemment.

- En pseudo-code :

```
SI condition ALORS
    instructions1

SINONSI condition2 ALORS
    instructions2

SINON
    instructions3

FINSI
```

# Exemples : Pseudo-code

DEBUT

Variable jour : Entier

Lire(jour)

SI jour == 1 ALORS Ecrire("Lundi")

SINONSI jour == 2 ALORS Ecrire("Mardi")

SINONSI jour == 3 ALORS Ecrire("Mercredi")

SINONSI jour == 4 ALORS Ecrire("Jeudi")

SINONSI jour == 5 ALORS Ecrire("Vendredi")

SINONSI jour == 6 ALORS Ecrire("Samedi")

SINONSI jour == 7 ALORS Ecrire("Dimanche")

SINON : Ecrire("Erreur")

FINSI

FIN



C'est déjà mieux

# Exemples : JavaScript

```
if (condition)
{
    instructions;
}
else if (condition)
{
    instructions;
}
else
{
    instructions;
}
```

**NB :** SI (il n'y a qu'une seule ligne d'instructions) ALORS les {} ne sont pas obligatoires. On les met pour ne pas devoir les mettre si on doit ajouter une instruction.

```
let jour;
jour = parseInt(prompt("Entrez le num"));
if (jour == 1)
    console.log("Lundi");
else if (jour == 2)
    console.log("Mardi");
else if (jour == 3)
    console.log("Mercredi");
else if (jour == 4)
    console.log("Jeudi");
else if (jour == 5)
    console.log("Vendredi");
else if (jour == 6)
    console.log("Samedi");
else if (jour == 7)
    console.log("Dimanche");
else
    console.log("Erreur");
```

# SELON QUE

- Bien que le SI.. SINON SI..SINON soit plus compact, cela reste un peu lourd. Une autre instruction va nous permettre de soulager l'écriture de ce genre de tests : le SELONQUE.

- Permet de condenser une armée de SI..SINON SI..SINON de manière plus élégante.

- Pseudo-code :

```
SELONQUE expression VAUT  
    valeur1 : instructions1  
    valeur2 : instructions2  
    [SINON : instructions3]  
FINSQ
```

# Exemple : Pseudo-code

- Voici l'algorithme du jour de la semaine en utilisant un SELON QUE.

```
DEBUT
  Variable jour : Entier

  Lire(jour)
  SELONQUE jour VAUT
    1 : Ecrire("Lundi")
    2 : Ecrire("Mardi")
    3 : Ecrire("Mercredi")
    4 : Ecrire("Jeudi")
    5 : Ecrire("Vendredi")
    6 : Ecrire("Samedi")
    7 : Ecrire("Dimanche")

    SINON : Ecrire("Erreur")
  FINSQ
FIN
```

# SWITCH : JavaScript

```
switch (nomVariable)
{
    case valeur1 :
        instructions;
        break;
    case valeur2 :
        instructions;
        break;
    case valeur3 :
    case valeur4:
        instructions;
        break;
    default :
        instructions;
        break;
}
```

```
let jour;

jour = parseInt(prompt("Entrez le numéro"));

switch (jour)
{
    case 1:
        console.log("Lundi");
        break;
    case 2:
        console.log("Mardi");
        break;
    case 3:
        console.log("Mercredi");
        break;
    case 4:
        console.log("Jeudi");
        break;
    case 5:
        console.log("Vendredi");
        break;
    case 6:
        console.log("Samedi");
        break;
    case 7:
        console.log("Dimanche");
        break;
    default:
        console.log("Erreur");
        break;
}
```

# Exercices

## •exo07-Année bissextile (Pseudo-Code + JavaScript)

Réalisez un petit algorithme qui sur base d'une année donnée va déterminer s'il s'agit d'une année bissextile. Une année est bissextile si elle est divisible par 4, mais non divisible par 100. Ou si elle est divisible par 400.

Bissextile : 2000, 1996

!Bissextile : 1900, 1997

## •exo8-Lanceur de balles de tennis (Pseudo-Code + JavaScript)

(Indice JavaScript : `booléen = texteEntree == "true" ? true : false`)

JS

Réalisez l'algorithme d'un lanceur de balles de tennis. Ce lanceur possède deux états :

- pret : permet de savoir si le tennisman est prêt. Il ne faut pas lancer de balles dans le cas contraire
- panierVide : permet de savoir s'il y a encore des balles disponibles

Le lanceur de balle possède l'opération « lancerBalle » qui, vous l'aurez compris, permet de lancer une balle.

## •exo09-Distributeur de boissons (Pseudo-Code + JavaScript) (switch())

Réalisez l'algorithme d'un distributeur de boissons. Ce dernier propose plusieurs boissons et l'utilisateur choisit celle qu'il désire en entrant le numéro correspondant. N'oubliez pas de vérifier s'il y a encore des boissons en stock.



# Exercices

## • **exo10-Calculatrice (Pseudo-Code + JavaScript) (switch())**

Réaliser l'algorithme d'une calculatrice basique. L'utilisateur est invité à saisir un nombre, un opérateur, et un deuxième nombre. La calculatrice affiche ensuite le résultat. (Gérer la division par 0)

### ● **exo11-Note (Pseudo-Code + JavaScript)**

Ecrire un algorithme qui met l'appréciation par rapport à des notes. Ces notes sont comprises entre 0 et 20.

- 0-10 : I, 11-12 : S, 13-15 : B, 16-18 : TB, 19-20 : Excellent

/!\ Gérer les erreurs : ex : -2; 25

# Exercices

- exo12-Réalisez un algorithme utilisant le convertisseur de secondes, il reçoit deux durées (jours, heures, minutes et secondes) et calcule la différence entre ces dernières.

# **LES STRUCTURES ITÉRATIVES**

# Introduction aux structures itératives

- Nous pouvons maintenant faire pas mal de choses avec ce que nous avons appris.
- Mais jusqu'à présent, nos algorithmes ne peuvent exécuter leurs instructions qu'une seule fois. Ce n'est pas très pratique !
- Reprenons le lanceur de balle de tennis, le distributeur de boissons et la calculatrice :
  - Le lanceur de balle devrait pouvoir lancer des balles jusqu'à ce que son stock soit vide
  - Le distributeur de boissons pourrait proposer une autre boisson au client tant que ce dernier le désire
  - Pareil pour la calculatrice

# Les boucles

- Afin répéter des instructions dans nos algorithmes, nous utilisons les boucles.
- Une boucle est une structure itérative permettant de réitérer un comportement sur base d'une condition précise.
- Exemple :
  - TANT QU'**il y a des balles, lancer une balle.
  - Proposer une boisson **JUSQU'À** ce que le client soit comblé.
  - FAIRE** un calcul **TANT QUE** l'utilisateur veut faire des calculs.

# Les boucles

- Pour utiliser une boucle, il est nécessaire de pouvoir identifier clairement :
  - Ce qu'il faut répéter
  - Pourquoi le répéter
  - Maîtriser cette répétition (principalement, comment l'arrêter)
- Nous répétons des instructions afin d'atteindre un état défini. Il est donc primordial que les instructions répétées permettent d'atteindre l'état voulu sous peine de se retrouver dans une boucle infinie !

# Les types de boucles

- Nous avons à notre disposition 3 types de boucles :
  - Boucles à tests antérieurs (TANT QUE... FAIRE)
  - Boucles à tests postérieurs, décomposées en deux (FAIRE... TANT QUE et FAIRE... JUSQU'À)



# TANTQUE.. FAIRE

- Dans cette boucle, nous répétons nos instructions tant que la condition définie est vraie. De plus, si cette condition n'est pas vérifiée au départ, nous n'exécutons pas du tout les instructions.

- En pseudo-code :

```
TANTQUE condition(s) FAIRE  
    instructions  
FINTQ
```



# JavaScript : Boucle Tant Que

- Voici à quoi ressemble une boucle TANT QUE en JavaScript.

```
while (conditions)
{
    instructions;
}
```

# Exemples : Pseudo-code

- Nous pouvons utiliser une boucle pour valider la saisie d'un utilisateur par exemple :

```
DEBUT
  Variable nombre : Entier

  Ecrire("Entrez un nombre entre 1 et 10")
  Lire(nombre)
  TANTQUE (nombre < 1) OU (nombre > 10) FAIRE
    Ecrire("Valeur incorrecte ! Réessayer")
    Lire(nombre)
  FINTQ
FIN
```

# Exemples : JavaScript

```
nombre = parseInt(prompt("Entrez un nombre entre 1 et 10"));
while ((nombre < 1) || (nombre > 10))
{
    console.log("Nombre incorrect ! Réessayez.");
    nombre = parseInt(prompt("Entrez un nombre entre 1 et 10"));
}
```

# Exercices

- exo13-À l'aide d'une boucle, affichez la table de multiplication par 2. Ensuite, codez votre algorithme en JavaScript.
- exo14-Reprenez l'algorithme du lanceur de balles de tennis et faites en sorte qu'il lance une balle tant que le stock n'est pas vide. Il y a donc 2 variables stockBalles et pret
- exo15-À l'aide de deux boucles, affichez les tables de multiplication de 1 à 9. Ensuite, codez votre algorithme en JavaScript.

# Plus ou Moins

- exo16-Un algorithme reçoit deux nombres de l'utilisateur (opération Lire) : justePrix et proposition.
- Il répond : « C'est plus » lorsque proposition est **plus petit** que justePrix.
- Et inversement, il répond : « C'est moins » lorsque proposition est **plus grand** que justePrix.
- Si justePrix est égal à proposition, il répond : « C'est gagné ».

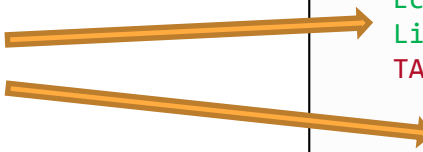
# Plus ou Moins

Améliorons ce Plus ou Moins !

# Remarques

- Dans certains cas, il est nécessaire d'effectuer les instructions au moins une fois avant d'évaluer s'il est nécessaire de les répéter.
- Reprenons l'exemple de la validation de la saisie d'un utilisateur :

L'instruction Lire()  
est répétée deux fois.



```
DEBUT
  Variable nombre : Entier

  Ecrire("Entrez un nombre entre 1 et 10")
  Lire(nombre)
  TANTQUE (nombre < 1) OU (nombre > 10) FAIRE
    Ecrire("Valeur incorrecte ! Réessayer")
    Lire(nombre)
  FINTQ
FIN
```

- La solution serait d'évaluer la condition après une première répétition.

# FAIRE .. TANT QUE

- La boucle FAIRE.. TANT QUE nous permet d'effectuer les instructions de la boucle une première fois avant d'évaluer s'il est nécessaire de les réitérer.
- Même si la condition de répétition est fausse dès le départ, les instructions seront exécutées une fois.
- En pseudo-code :

```
FAIRE  
    instructions à répéter  
TANTQUE conditions FINTQ
```



# Exemples : Pseudo-code

DEBUT

Variable nombre : Entier

FAIRE

Ecrire("Entrez un nombre entre 1 et 10")

Lire(nombre)

TANTQUE (nombre < 1) OU (nombre > 10) FINTQ

FIN

# Exemples : JavaScript

```
do  
{  
    // instructions  
} while (conditions) ;
```

! N'oubliez pas le ;

# Exemples : JavaScript

```
let nombre;  
do  
{  
    nombre = parseInt(prompt("Entrez un nombre entre 1 et  
10"));  
} while (nombre < 1) || (nombre > 10) ;
```

# FAIRE .. JUSQU'À

- La troisième boucle est similaire à FAIRE .. TANT QUE, seule l'évaluation de la condition diffère.
- Comme vous l'aurez compris, cette boucle va répéter les instructions **JUSQU'À** ce que la condition soit vraie.
- En pseudo-code :

```
FAIRE  
    instruction  
JUSQU'À condition(s)
```

# Quel intérêt entre les deux ?

## •FAIRE .. TANT QUE

- Répète les instructions tant que la condition est **VRAIE**.
- Répète les instructions jusqu'à ce que la condition soit **FAUSSE**.
- Forme la plus répandue dans les langages de programmation courants : C, C#, Java...

## •FAIRE .. JUSQU'À

- Répète les instructions tant que la condition est **FAUSSE**.
- Répète les instructions jusqu'à ce que la condition soit **VRAIE**.
- Présente dans certains langages comme le Pascal, Visual Basic...
- Cette boucle n'existe pas en C#, Java

# Attention aux conditions !

- La subtilité entre les deux boucles se trouve principalement dans la formulation de la condition. Les deux exemples ci-dessous sont équivalents !

```
DEBUT
  Variable nombre : Entier

  Ecrire("Entrez un nombre entre 1 et 10")
  FAIRE
    Lire(nombre)
  TANTQUE (nombre < 1) OU (nombre > 10) FINTQ
FIN
```

```
DEBUT
  Variable nombre : Entier

  Ecrire("Entrez un nombre entre 1 et 10")
  FAIRE
    Lire(nombre)
  JUSQU'À (nombre >= 1) ET (nombre <= 10) FINJ
FIN
```

Remarquez la différence entre les deux conditions.

# Exercices

- exo17-À l'aide d'une boucle FAIRE ... TANTQUE , améliorez l'algorithme du distributeur de boissons pour qu'il demande au client s'il désire une autre boisson (Tant qu'il en a envie).
- exo18-À l'aide d'une boucle FAIRE ... TANTQUE, améliorez l'algorithme de la calculatrice afin qu'elle demande à l'utilisateur s'il veut faire un autre calcul (tant qu'il le désire).
- exo19-À l'aide de la boucle TANTQUE ... FAIRE, réalisez un algorithme calculant le résultat de  $N^{10}$ . N étant un nombre saisi par l'utilisateur.
- exo20-Reprenez l'exercice précédent et modifiez-le pour que l'utilisateur entre également l'exposant qu'il désire calculer.

# Exercices

- exo21-Améliorez le "C'est plus, c'est moins, c'est gagné" pour qu'il tourne en boucle tant que le juste\_prix n'a pas été trouvé. L'ordinateur choisit un nombre aléatoirement entre 1 et 100. L'utilisateur est invité à entrer un nombre et l'algorithme nous répond "C'est plus" ou "C'est moins". Lorsqu'on a trouvé le bon nombre, l'algorithme affiche le nombre de tentatives effectuées pour trouver le résultat



# Exercices

Exercices supplémentaires:

● 1) Réalisez un système de connexion à l'aide d'un mot de passe. L'algorithme demande à l'utilisateur de saisir son mot de passe. Si ce dernier valide de bon mot de passe, on le salue. Par contre, si il fait une erreur trois fois de suite, un message lui signalera que son compte est bloqué et il ne pourra pas réessayer une quatrième fois

● 2) Réalisez un algorithme qui demande un nombre à l'utilisateur et affiche autant de ligne que le nombre spécifié par l'utilisateur. Exemple : l'utilisateur a rentré le nombre 5 et l'algorithme affiche :

```
*  
  
**  
  
***  
  
****  
  
*****
```

● 3) Ecrivez un algorithme qui demande à l'utilisateur de taper 10 entiers et qui affiche le plus petit de ces entiers.

● 4) Algorithme demandant 3 nombres : nbRep, nbTiret, nbEspace. Ce dernier affiche à l'écran autant de tiret que la valeur de nbTiret, suivi d'autant d'espace que la valeur de nbEspace. Le tout autant de fois que la valeur de nbRep. Exemple : si nbRep = 2, nbTiret = 1 et nbEspace = 3 le résultat est le suivant :|- - |

# Projet : justePrix

- Améliorez encore le justePrix : l'utilisateur a droit à 10 essais après ces 10 essais, il a perdu et l'ordinateur affiche le justePrix
- Ajouter un niveau :
  - facile : entre 1 et 10
  - moyen : entre 1 et 100
  - difficile : entre 1 et 1000
- Tant que la personne veut rejouer, redemandez le niveau et générez un nombre.
- Vérifiez que tout caractère entré est correct, c'est-à-dire pour que le programme ne plante jamais.

# Indices

- Pour générer un nombre entre 0 et 10 non compris, il faut :

```
let max = 10;  
  
let random = Math.floor(Math.random() * max);
```

- Pour effacer l'écran, il faut 2 choses :

```
console.clear();
```

# Indices

- Pour vérifier qu'une chaîne de caractères est un nombre :

```
let chaine = "4";  
console.log(Number.isInteger(chaine));
```

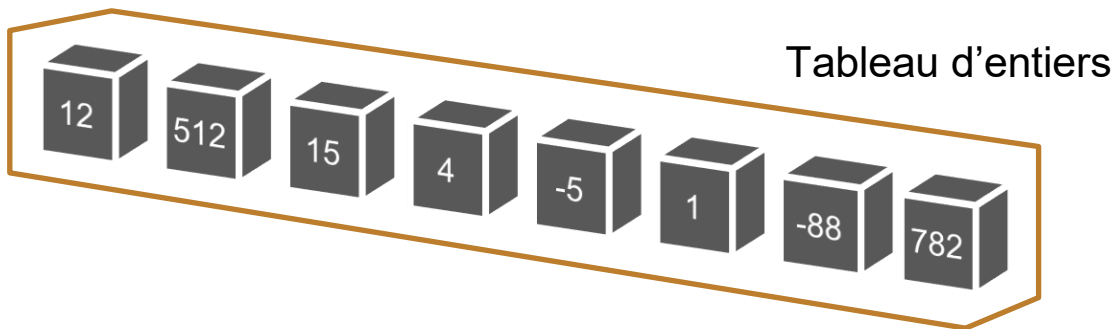
# LES TABLEAUX

# Utilité

- Dans beaucoup de cas, nous pouvons avoir besoin de nombreuses variables d'un même type :
  - Garder un ensemble de relevés de température pour fournir des statistiques.
  - Conserver les notes d'un élève pour calculer sa moyenne
  - ...
- Nous serions tentés de déclarer autant de variables que le nombre de valeurs dont nous avons besoin. C'est embêtant et laborieux ! Et les informaticiens sont paresseux, ils n'aiment pas faire deux fois la même chose 😊 !
- Les tableaux nous permettent d'obtenir un nombre donné d'éléments d'un même type.

# Les tableaux

- Un tableau est une structure de données composée d'un ensemble de variables, du même type, accessibles par leur indice.



- La déclaration d'un tableau est similaire à celle des variables avec deux éléments supplémentaires :
  - Le type de variables que va contenir le tableau
  - La taille du tableau : représente le nombre d'éléments que peut contenir le tableau

# Déclarer un tableau

- Pour déclarer un tableau, rien de plus simple :

```
Variable nomTableau : Tableau[nombreElement] de typeElement
```

- Exemple :

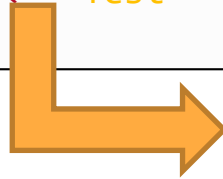
```
Variable tableauCotes : Tableau[10] de Reel
```

- Lorsque nous déclarons un tableau, l'ordinateur réserve le nombre de cases nécessaires dans la mémoire vive.
- Contrairement aux variables que nous avons vues précédemment, il n'y a pas de nom pour chaque cellule du tableau. Mais comment accéder aux éléments du tableau ?



# Affectation dans un tableau

```
DEBUT  
  Variable tab : Tableau[10] de Chaine  
  
  tab <- "Test"  
FIN
```



Est-ce possible ? Si oui, pourquoi et comment ?

# L'opérateur d'accès [ ]

- Nous sommes capables de déclarer un tableau de N cases d'un certain type. Mais nous devons pouvoir accéder à chaque case du tableau !
- Les cellules d'un tableau sont toutes numérotées à partir de 0 jusqu'à la taille du tableau -1.
- Le numéro d'une cellule peut être appelé l'**indice** de la cellule.
- Nous utilisons dès lors l'opérateur d'accès **[ ]** qui va nous permettre de l'indice de la cellule à laquelle nous désirons accéder.

# L'opérateur d'accès [ ] : Exemples

Constatez que la numérotation commence bien à 0 et termine à 2 (taille du tableau -1).

DEBUT

Variable tabSMS : Tableau[3] de Chaine

tabSMS[0] <- "Première case du tableau"

tabSMS[1] <- "Deuxième case du tableau"

tabSMS[2] <- "Dernière case du tableau"

FIN

DEBUT

Variable tab : Tableau[2] de Entier

tab[0] <- 10

tab[1] <- "valeur"

tab[2] <- 1

FIN

**INTERDIT !**

Car il s'agit d'un tableau d'entiers

Ne fonctionnera pas, car nous sommes en dehors du tableau.

# Les tableaux en JavaScript

- Un tableau en JavaScript se déclare de cette manière.

```
// déclaration
let nomTableau = new Array();
// initialisation/affectation
nomTableau[0] = "element 1";
nomTableau[1] = "element 2";
// ou directement : décl. + initia.
let nomTableau = ["elt1", "elt2"];
```

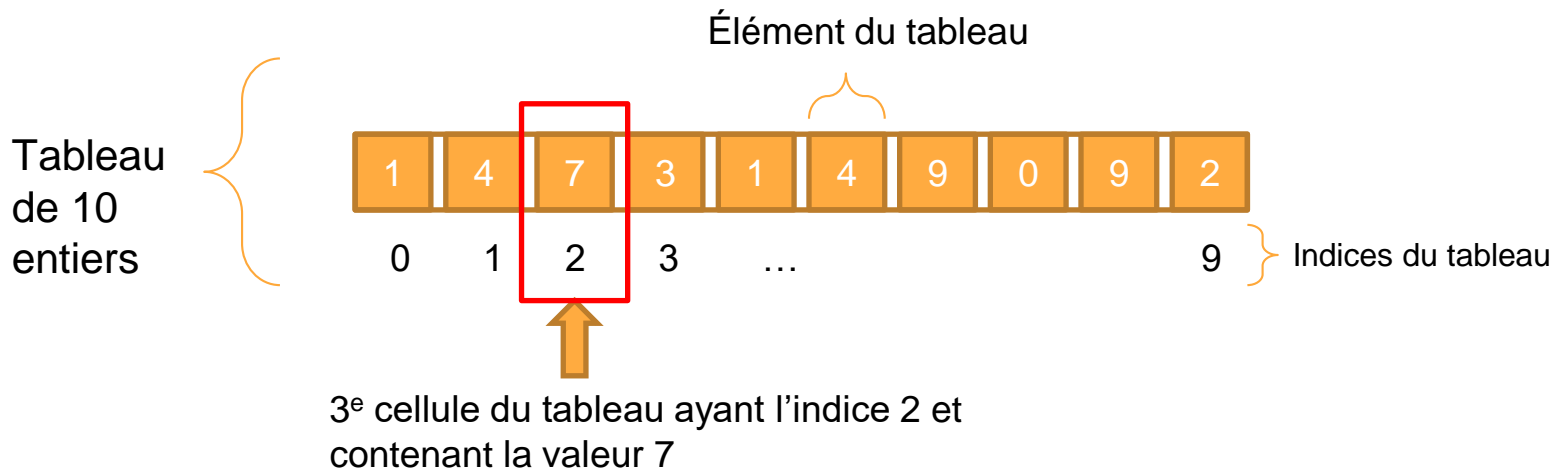
```
// déclaration
let tableau = new Array();
// initial./affectation
tableau[0] = 1;
tableau[1] = 2;
tableau[2] = 3;
// ou
let tableau = [ 1, 2, 3 ];
```

- Pour afficher un élément du tableau

```
System.out.println(tableau[0]);
```

# Représentation d'un tableau en mémoire

- Voici un petit résumé de ce que nous avons vu sur les tableaux :



## Exemple :

- Voici un exemple d'utilisation d'un tableau
- Nous demandons à l'utilisateur de rentrer 3 cotes sur 20, que nous enregistrons dans un tableau pour ensuite calculer la moyenne des trois cotes.
- Nous verrons par la suite comment améliorer cet exemple.

```
DEBUT
  Variable  tabCotes : Tableau[3] de Reel,
           moyenne : Reel

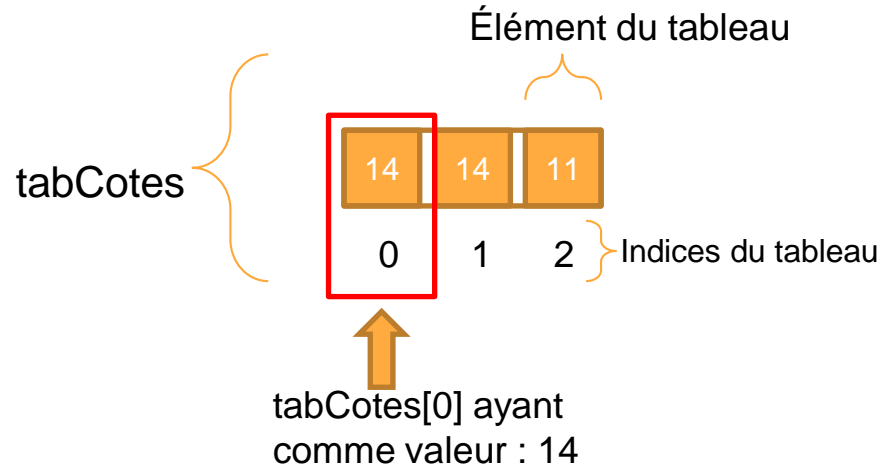
  Ecrire("Veuillez entrer la première cote")
  Lire(tabCotes[0])
  Ecrire("Veuillez entrer la deuxième cote")
  Lire(tabCotes[1])
  Ecrire("Veuillez entrer la troisième cote")
  Lire(tabCotes[2])

  moyenne <- tabCotes[0]
  moyenne <- moyenne + tabCotes[1]
  moyenne <- moyenne + tabCotes[2]

  Ecrire("La moyenne est de "+ moyenne/3+" sur 20")
FIN
```

# Exemple : Représentation du tableau en mémoire

- En considérant les cotes suivantes : 14/20, 14/20 et 11/20.  
Le tableau en mémoire serait représenté comme suit :



# Trucs et astuces

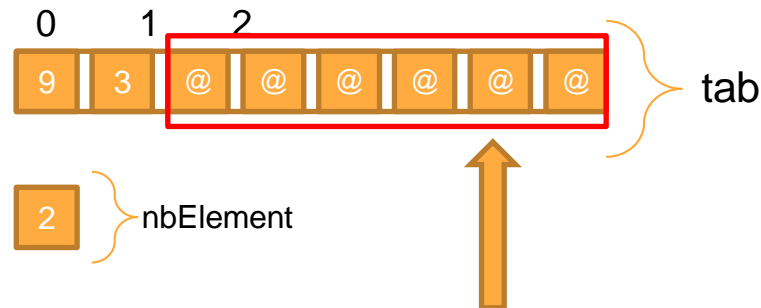
- Il est souvent utile de stocker la taille d'un tableau dans une constante. Cela rend l'algorithme plus facilement modifiable. Nous illustrerons cela un peu plus loin dans le chapitre.
- Nous avons donc la taille du tableau en constante, mais comment savoir quelles sont la ou les cellules possédant déjà une valeur ou non ? Une solution simple consiste à utiliser une variable supplémentaire pour retenir le nombre d'éléments utiles dans le tableau.
- De plus, l'utilisation de cette variable va nous permettre d'enregistrer les nouvelles valeurs au bon endroit sans nous tromper.



# Trucs et astuces : Exemple

```
Constante : TAILLE = 10
Variable : tab : Tableau[TAILLE] de Entier
           nbElement : Entier

DEBUT
  nbElement ← 0
  tab[nbElement] ← 9
  nbElement ← nbElement + 1
  tab[nbElement] ← 79
  nbElement ← nbElement + 1
FIN
```



Nous ne connaissons pas le contenu de ces cellules (résidus de mémoire). La variable nbElement nous permet donc de connaître le nombre de cellule ayant une valeur utile.

# Les tableaux en JavaScript

- Pour ajouter un élément au tableau.

```
// déclaration
let nomTableau = new Array();

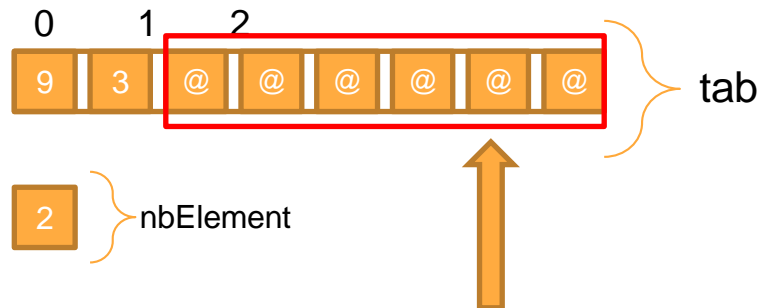
// récupération
let nomVariable = prompt("Saisissez une valeur à ajouter");

// ajouter le contenu d'une variable dans un tableau
nomTableau[0] = nomVariable;
// ou directement
nomTableau[1] = prompt("Saisissez une valeur à ajouter");
console.log(nomTableau[0]);
console.log(nomTableau[1]);
```

# Trucs et astuces : Exemple

```
Constante TAILLE = 10
Variable tab : Tableau[TAILLE] de Entier
           nbElement : Entier

DEBUT
    tab[0] <- 9
    tab[1] <- 79
    Ecrire(tab[0])
    Ecrire(tab[1])
FIN
```



Pour accéder un élément précis, il suffit d'ajouter l'indice entre [].

# Les tableaux en JavaScript

- Pour récupérer un élément bien précis.

```
maVariable = nomTableau[indice];  
  
maVariable = nomTableau[0];
```

- Pour afficher un élément bien précis.

```
console.log(nomTableau[indice]);  
  
console.log(nomTableau[0]);
```

# Les tableaux en JavaScript

- Pour parcourir un tableau.

```
// déclarer
let nomTableau = new Array();
nomTableau[0] = "element 1";
nomTableau[1] = "element 2";
nomTableau[2] = "element 3";

// initialiser le compteur de la boucle
let i = 0;

// parcourir le tableau :
// .length renvoie le nombre d'éléments dans le tableau
while (i < nomTableau.length)
{
    System.out.println(nomTableau[i]);
    i++; // incrémentation
}
```

# Exercices

- exo22-Écrire un algorithme qui saisit 6 entiers et les stocke dans un tableau, puis affiche le contenu de ce tableau une fois qu'il est rempli.
- exo23-BONUS : initialiser un tableau de 10 entiers avec les valeurs 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024 à l'aide d'une boucle. Ensuite, à l'aide d'une boucle afficher la valeur de chaque cellule du tableau avec l'opération Ecrire().

# Correction de l'exercice BONUS

- Pour pouvoir parcourir un tableau, nous avons besoin de plusieurs éléments :
  - La taille du tableau
  - Le nombre d'éléments dans le tableau
  - Une variable servant d'indice
  - Une boucle

# Exercice BONUS

Déclaration des constantes et des variables.

Initialisation du compteur d'élément, de la variable indice ainsi que du premier élément du tableau.

Boucle d'initialisation du tableau. Nous attribuons la valeur d'une cellule en multipliant la valeur de la cellule précédente par deux. Ensuite nous mettons à jour le compteur d'élément (nbElem).

Boucle d'affichage des éléments du tableau. Nous utilisons la variable indice pour parcourir le tableau à chaque passage dans la boucle.

DEBUT

Constante TAILLE = 10

Variable tab : Tableau[TAILLE] d'Entier

Variable nbElem, indice : Entier

nbElem <- 0

indice <- 0

tab[nbElem] <- 2

nbElem <- nbElem + 1

TANTQUE nbElem < TAILLE FAIRE

tab[nbElem] <- tab[nbElem-1] \* 2

nbElem <- nbElem + 1

FINTQ

TANTQUE indice < nbElem FAIRE

Ecrire(tab[indice])

indice <- indice + 1

FINTQ

FIN



# Une nouvelle boucle sauvage apparaît !

- Une boucle spéciale va nous permettre d'itérer plus facilement un tableau.
- Nous aurons souvent besoin de faire ceci :  
**Pour** chaque élément du tableau, effectue les instructions données.

# Boucle POUR

- En pseudo-code :

```
POUR indice DE valeurDebut À valeurFin [PAR p] FAIRE  
    instructions  
FINP
```

- *indice* : correspond à la variable utilisée pour l'indice du tableau (comme dans l'exercice bonus)
- *valeurDebut* : représente la valeur à laquelle la variable *indice* va être initialisée
- *valeurFin* : représente la valeur à laquelle la boucle va s'arrêter
- *p* : indique un pas d'incrément, c'est-à-dire de combien *indice* va être augmenté (ou diminuer) à chaque itération. Cette clause est facultative, elle vaut 1 par défaut

# JavaScript : Boucle POUR (version 1)

- La boucle Pour en JavaScript :

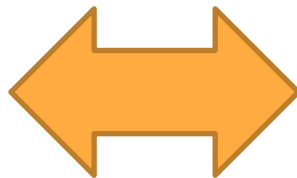
```
// initial.      condition      incrémentation  
for(let i = 0; i < nomTableau.length; i++)  
{  
    console.log(nomTableau[i]);  
}
```

- La boucle for parcourt tout le tableau, il est possible d'accéder à chaque élément du tableau grâce `nomTableau[i]`

# Équivalence POUR et TANT QUE

Les deux exemples ci-dessous sont strictement **équivalents** !

```
POUR indice de 0 à 9 par 2 FAIRE  
    Ecrire(tableau[indice])  
FINPR
```



```
indice ← 0  
TANT QUE non(indice > 9) FAIRE  
    Ecrire(tableau[indice])  
    indice ← indice + 2  
FINTQ
```

# Exemple : Pseudo-code

- Reprenons l'exemple de l'exercice bonus en utilisant une boucle POUR :

```
DEBUT
  Constante MAXCOTES = 10
  Variable tabCotes : Tableau[MAXCOTES] de
Entier
  Variable nbCotes, indice : Entier

  nbCotes <- 0
  indice <- 0
  tabCotes[nbCotes] <- 2
  nbCotes <- nbCotes + 1

  POUR indice DE nbCotes À maxCotes - 1 FAIRE
    tabCotes[indice] <- tabCotes[indice-1] * 2
    nbCotes <- nbCotes + 1
    Ecrire(tabCotes[indice])
  FINPR
FIN
```

Nous en profitons pour afficher la valeur calculée directement afin ne pas devoir utiliser une deuxième boucle.

Voilà le résultat,  
Plus compréhensible non ? ☺

# Exercices

- exo24-Écrire un algorithme demandant à l'utilisateur le nombre de joueurs (max 10 joueurs). Ensuite, l'algorithme doit demander à l'utilisateur le score de chaque joueur. Une fois ceci fini, il faut afficher la moyenne des scores. Faites de même en Java
- exo25-Inverser un tableau : soit un tableau T. Saisir ce tableau. Changer de place les éléments de ce tableau de façon à ce que le nouveau tableau soit une sorte de miroir de l'ancien et afficher le nouveau tableau.
- exo26-À l'aide des boucles, réalisez un algorithme permettant de trier un tableau d'entiers dans l'ordre croissant. Mettez-le ensuite en pratique avec JavaScript.

# Java : Quelle touche enfoncée ?

- Pour récupérer le premier caractère d'une String :

```
let mot = prompt("Entrez un mot");  
let lettre = mot.charAt(0);
```

# Exercices

- exo27-Refaites l'algorithme qui demande à l'utilisateur de taper 10 entiers et qui affiche le plus petit de ces entiers mais cette fois-ci à l'aide d'un tableau et sans retenir le minimum lors de la saisie.
- exo28-Réalisez un algorithme permettant de rechercher une valeur dans un tableau. Si la valeur se trouve bien dans le tableau, nous affichons sa position.
- exo29-Réalisez un algorithme nous permettant de déplacer un pion dans un tableau de 10 éléments. Au début, le pion se trouve dans la première case du tableau. Nous pouvons ensuite le déplacer par la gauche (g), par la droite (d) ou de stopper l'algorithme (q).



# Exercices

- exo30-En considérant un tableau d'entiers trié dans l'ordre croissant, réalisez un algorithme étant capable d'insérer une nouvelle valeur dans le tableau de façon à ce que le tableau reste trié. Le but n'est évidemment pas d'insérer la valeur à la fin et de trier après mais bien de l'insérer au bon endroit directement.
- exo31-Réalisez un algorithme dans lequel nous devons rechercher une valeur (entrée par l'utilisateur) dans un tableau d'entiers.
- exo32-En considérant deux tableaux d'entiers (non triés), réalisez un algorithme qui place tous les éléments des deux tableaux dans un troisième. Ce dernier doit être trié une fois l'algorithme terminé. Notez que le tri doit être fait en même temps que la fusion des deux tableaux et pas après.

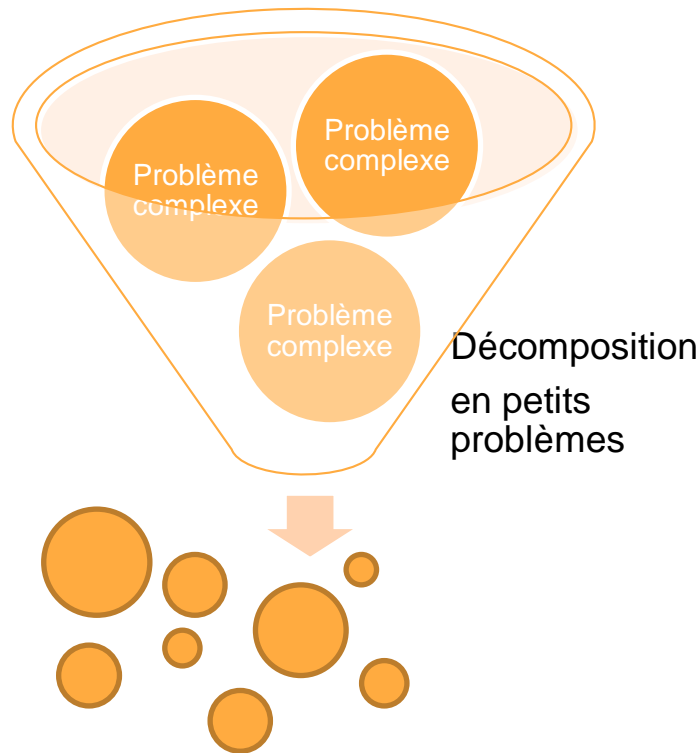
# **LES SOUS-PROGRAMMES**

# Diviser pour mieux régner

- Jusqu'à maintenant, nous avons écrit des petits algorithmes ne dépassant pas la trentaine de lignes. Mais comment s'y retrouver lorsqu'il y en a plusieurs milliers ?
- « À tout problème, il existe une solution ». La plus utilisée en algorithmique est la suivante : *Décomposition d'un problème complexe en plusieurs problèmes de complexité moindre.*

# Diviser pour mieux régner : Illustration

- Exemple : un robot doit pouvoir conduire un véhicule.
- Conduire peut être décomposé en :
  1. Démarrer le moteur
  2. Rouler jusqu'à la destination
  3. Couper le moteur
- Ces trois étapes sont encore décomposables en :
  1. Mettre la clé dans le contact
  2. Se mettre au point mort
  3. Embrayer
  4. Tourner la clé de contact
  5. Passer la première vitesse
  6. ...



# Diviser pour mieux régner

- Ce principe est fortement illustré par l'adage informatique KISS :

« Keep It Simple, Stupid »

*« Garde ça simple, idiot »*

- En d'autres termes : Ne nous compliquons pas la tâche.

# Sous-programme

- Le concept de sous-programme est donc une illustration de la décomposition de problèmes complexes en nombreux problèmes de complexité moindre.
- Concrètement, un sous-programme enferme un ensemble d'instructions dans une « boîte noire » que nous pourrions réutiliser dans l'algorithme en faisant appel à cette boîte.
- Les sous-programmes permettent donc de généraliser des actions ou des fonctionnalités.
- Il existe deux types de sous-programmes :
  - Les procédures
  - Les fonctions

# Utilité

- Outre le fait de décomposer un algorithme en plusieurs parties, les sous-programmes sont très utiles pour de nombreuses choses :

- Ils permettent une réutilisabilité du code. En effet, lorsque nous effectuons régulièrement les mêmes instructions, il est plus pratique d'écrire ces instructions une seule fois ! (Souvenez-vous, les informaticiens sont paresseux)

- Les sous-programmes réduisent également l'impact d'une erreur : si nous utilisons plusieurs fois les mêmes instructions dans notre algorithme, une erreur dans ces dernières nous obligerait à corriger l'erreur à tous les endroits où nous avons utilisé ces instructions (au risque d'en oublier une quelque part)

# Les procédures et fonctions

- Une procédure est un sous-programme que nous pouvons appeler pour utiliser des actions : traitement de données.
- Une fonction est un sous-programme que nous pouvons utiliser pour sa fonctionnalité, un peu comme un opérateur (+, /...). Une fonction va donc nous communiquer un résultat.



# Exemple introductif

Algorithme  
abstrait  
divisant deux  
nombres.

```
Algorithme diviseur :  
DEBUT  
    saisirDonnees()  
    traiterDonnees()  
    afficherResultats()  
FIN
```

Concrétisation

```
Procedure saisirDonnees() :  
DEBUT  
    Lire(nombre1)  
    Lire(nombre2)  
FIN
```

```
Algorithme diviseur :  
DEBUT  
    Variable nombre1, nombre2, resultat : Reel  
    saisirDonnees()  
    traiterDonnees()  
    afficherResultats()  
FIN
```

```
Procedure traiterDonnees() :  
DEBUT  
    SI nombre2 != 0 ALORS  
        resultat <- nombre1/nombre2  
    FINSI  
FIN
```

```
Procedure afficherResultats() :  
DEBUT  
    Ecrire(resultat)  
FIN
```

# Les paramètres : Introduction

- Notre exemple précédent comporte un **gros** défaut :

Nous utilisons les variables déclarées dans l'algorithme au sein de nos procédures. Si nous devons changer le nom d'une variable, nous devons le changer dans toutes nos procédures. Nous nous retrouvons donc avec le même problème que nous désirions éviter grâce au sous-programme.

- Les variables déclarées dans l'algorithme principal sont appelées des **variables globales**.
- C'est maintenant que les paramètres arrivent sur la scène !

# Portée lexicale

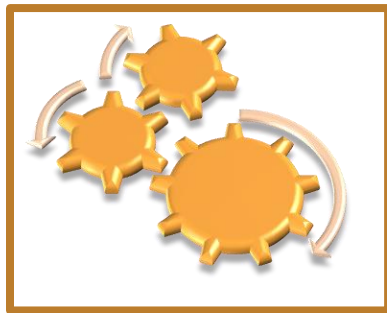
- Comme expliqué précédemment, les variables déclarées dans l'algorithme principal sont des variables globales. Mais qu'en est-il des paramètres et des variables déclarés dans nos procédures/fonctions ?
- Ces variables sont appelées **variables locales**, car elles existent en mémoire uniquement lorsque nous nous trouvons à l'intérieur de la procédure/fonction.

# Les paramètres

- Les paramètres nous permettent de transmettre des informations de notre algorithme principal à nos sous-programmes et inversement.
- Ils se présentent sous la forme de variables.
- Considérons un sous-programme qui affiche le contenu d'un tableau. Ce dernier reçoit un tableau et affiche les valeurs du tableau à l'aide de l'opération Ecrire. Il effectue une action, il s'agit donc d'une procédure. Pour fonctionner, cette procédure a besoin de deux éléments :
  - Le tableau
  - La taille du tableau
- Ces deux éléments représentent les **paramètres** de la procédure.

Tableau  
Taille du tableau

Procédure afficherTableau()



# Les deux types de paramètres

- Nous distinguons deux paramètres différents :

- Les paramètres de données : variables permettant de transmettre des données à notre sous-programme
- Les paramètres de résultats : variables permettant à un sous-programme de transmettre les données calculées à l'algorithme principal

- Les procédures sont caractérisées par le concept d'action, elles n'utilisent pas les paramètres de résultats contrairement aux fonctions.

- Il est important de préciser qu'une fonction ou une procédure n'a pas obligatoirement des paramètres de données.

# Les procédures : Concrètement

- Une procédure se déclare de cette façon en pseudo-code :

```
Procedure nomProcedure(parametre1 : type,...) :  
DEBUT  
    instructions  
FIN
```

- Voici un exemple de procédure :

```
Procedure direBonjour() :  
DEBUT  
    Ecrire("Bonjour jeune Padawan")  
FIN
```

# JavaScript : Procédures

Exemple :

```
function nomProcedure(parametre, ...)  
  
{  
  
    // instructions  
  
}
```

```
// définition des procédures à l'extérieur du main  
function direBonjour()  
{  
    console.log("Bonjour");  
}  
function direBonjourPrenom(prenom)  
{  
    console.log("Bonjour " + prenom);  
}  
  
// appel des procedures dans le main  
direBonjour();  
direBonjourPrenom("Jules");
```

# Les fonctions : Concrètement

- Une fonction se déclare à peu de chose près de la même façon qu'une procédure :

Nouvelle action permettant de revenir à l'algorithme principal en donnant une valeur correspondant au type de paramètre de résultat.

```
Fonction nomFonction(parametre1 : type,...) : type_retour
DEBUT
    instructions
renvoie(valeur)
FIN
```

Spécifie le type du paramètre de résultat

- Voici un exemple de fonction :

```
Fonction multiplierPar2(nb : Entier) : Entier
DEBUT
    renvoie(nb*2)
FIN
```

Et son utilisation :

```
DEBUT
    Variable chiffre : Entier
    chiffre <- 5
    chiffre <- multiplierPar2(chiffre)
FIN
```



# JavaScript : Fonctions = **return**

```
function nomFonction(parametre, ...)  
{  
    // instructions  
    return variableDuTypeDeRetour;  
}
```

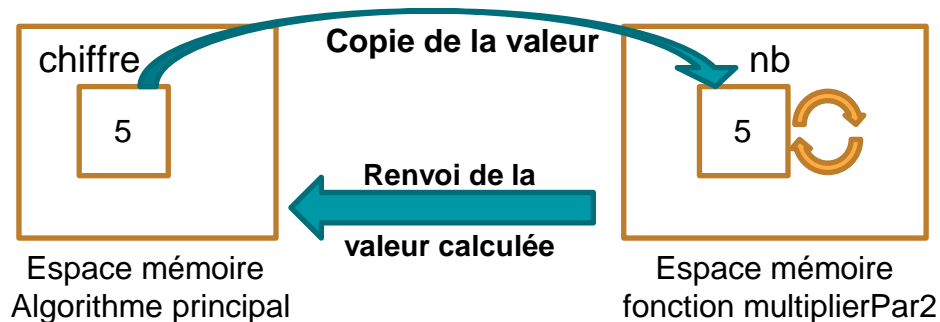
Exemple :

```
// définition de la fonction à l'extérieur du main  
function multiplierPar2(x)  
{  
    return x * 2;  
}  
//appel de la fonction à l'intérieur du main  
console.log(multiplierPar2(2));
```

# Passage de paramètres à un sous-programme

- Comme nous l'avons vu dans la déclaration des procédures/fonctions, nous donnons des paramètres dans la parenthèse qui suit le nom de la procédure/fonction. Mais qu'est-ce qui se cache derrière ça ?
- Reprenons l'exemple précédent :

Lorsque nous donnons la variable `chiffre` à la fonction, la valeur de cette dernière est copiée dans la variable locale `nb` de la fonction `multiplierPar2()`. Ce processus se nomme le **passage par valeur**. Voici une petite illustration :



# Les paramètres par référence

- Nous connaissons maintenant le fonctionnement du passage de paramètres. Mais comment faire si nous désirons modifier une variable globale à l'intérieur d'une procédure/fonction ?
- Le passage de paramètres par référence va nous le permettre.
- Nous savons que la valeur d'une variable est copiée dans la variable locale d'une procédure/fonction. Nous utilisons donc la référence de la variable ! La référence peut être comparée à l'adresse en mémoire de la variable globale.
- Si nous donnons cette référence à notre sous-programme, il sera capable d'accéder à la variable globale de l'algorithme et donc de pouvoir la modifier.

# Les paramètres par référence : Pseudo-code

- Dans la déclaration d'une procédure/fonction, il suffit d'ajouter le mot-clé **ref** devant les paramètres devant être passés par référence.

- Exemple :

```
Procédure ajouterUn(ref a : Entier) :  
DEBUT  
    a <- a + 1  
FIN
```

- Facile n'est-ce pas ?

# JavaScript : Passage par référence

Le passage par référence n'est possible qu'avec les tableaux. Tout tableau passé en paramètre dans un sous-programme sera mis à jour

Exemple :

```
// définition de la procédure à l'extérieur du main
function metajour(tab)
{
    tab[0] = 3;
}
// appel de la procédure à l'intérieur du main
let tableau = new Array();
tableau[0] = 1;
tableau[1] = 2;
//appel de la fonction
console.log("Avant : " + tableau[0]);
metajour(tableau);
console.log("Après : " + tableau[0]);
```

# Exercices

- exo33-Réalisez une fonction calculant le carré d'un nombre entier donné en paramètre.
- exo34-Réalisez une fonction de recherche dans un tableau. Cette fonction va recevoir un tableau, la taille du tableau, et la valeur recherchée en paramètres et renvoyer l'indice de l'élément dans le tableau. Si l'élément ne s'y trouve pas, la fonction renvoie -1.
- exo35-Réalisez une procédure dont l'objectif est de fusionner deux tableaux d'entiers.

# SUPPLÉMENTS

# Les tableaux à n dimensions

•Jusqu'à présent, nous avons utilisé des tableaux à une dimension. Mais il est possible d'avoir autant de dimension qu'on le souhaite.

•Un tableau à 2 dimensions représente une matrice d'éléments.

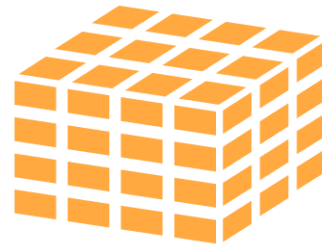
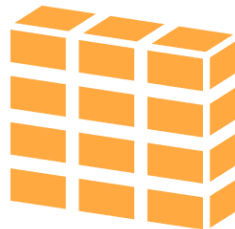
```
Variable matrice : Tableau[4,3] d'Entier
```

```
Variable matrice : Tableau[ligne, colonne] d'Entier
```

•Un tableau à 3 dimensions représente un cube d'éléments.

```
Variable matrice : Tableau[4][4][3] d'Entier
```

```
Variable matrice : Tableau[ligne][colonne][profondeur] d'Entier
```





# Merci pour votre attention.

