Contenu

1.Introduction à la programmation. Algorithmique

Pour que l'ordinateur réalise la fonction qu'on souhaite on doit créer un **programme**. Un programme est une suite d'instructions dans un langage de programmation (PHP, ASP, Java, javascript, python) qui conduit à un résultat donné (si tout va bien!)

Avant de coder un programme on a besoin de définir **la suite d'opérations que ce programme va réaliser dans un langage compréhensible par l'être humain**. Cette suite d'opérations s'appelle **algorithme**.

Ex : on veut réaliser un programme qui lit deux valeurs du clavier et affiche la somme de ces deux valeurs. On ne sait pas encore quel langage on va utiliser mais on peut déjà faire l'algorithme.

Algorithme :

- Demander à l'utilisateur de saisir une valeur
- Lire la valeur saisie par l'utilisateur
- Demander à l'utilisateur de saisir une deuxième valeur
- Lire la valeur saisie par l'utilisateur
- Calculer la somme de ces deux valeurs
- Afficher le résultat

Nous pouvons écrire des algorithmes pour de procédures qui n'ont rien à voir avec l'informatique : cuisiner un bon repas, ouvrir une porte fermée à clé, monter un meuble...

Observez que nous venons d'écrire un algorithme dans le « langage humain ». Logiquement la machine ne comprend pas ce langage. Pour que la machine comprenne, on doit créer le programme à partir de l'algorithme dans un langage de programmation dont vous avez surement entendu parler: PHP, Java, C++, JavaScript...

Dans le cadre de ce cours **on utilisera le langage PHP**, qui est très similaire à celle de la plupart de langages modernes. Tous les langages de programmation sont basés sur quatre éléments :

1. les instructions de **lecture / écriture (READ/WRITE)**
2. les **variables** (variable= espace de stockage)
3. les **tests ou conditions (IF)**
4. les **boucles (WHILE, FOR)**

2.Écriture et lecture

2.1. Écriture

Les instructions d'**écriture** permettent au programme de « envoyer de données à un dispositif de sortie». Dans notre contexte ça veut juste dire « afficher de données sur l'écran »

(D'un point de vue plus large on peut aussi « écrire » de données sur un disque dur, une carte réseau, une imprimante… pour le moment on se concentre sur l'écran !)

Pour afficher quoi qui ce soit sur l'écran on utilise **echo** ou **print**

Exemple :

```
print ("Bienvenu");

echo (2*5+6);
```

2.2. Lecture

Les instructions de **lecture** permettent à l'utilisateur de « recevoir de données d'un dispositif d'entrée». Dans notre contexte, c'est juste « obtenir les données qu'on saisit au clavier ». Si nous sommes en train d'utiliser PHP pour faire une application WEB, les données seront saisies par l'utilisateur en utilisant de formulaires dans la page.

(D'un point de vue plus large on peut aussi lire de données d'un scanner, un disque dur, carte reseau…)

Il n'y a pas une instruction équivalente à **print** pour la lecture dans le langage PHP, mais **sin on travaille sur la console** on peut créer nous-mêmes le code qui réalisera cette fonction (lire du clavier). Ce code se trouve déjà dans le fichier **codeLectureConsole.php**.

Pour l'utiliser, vous devez faire appel à la fonction **read()**

Exemple :

```
$nomUtilisateur = read() ;
```

Quand le programme rencontre une instruction **read**, l'exécution s'interrompt et l'ordinateur attend la frappe d'une valeur au clavier. Une fois l'utilisateur a saisi la valeur et appuyé sur **Enter** l'exécution continue et la valeur saisie sera stockée dans la variable nomUtilisateur. Et qu'est-ce que c'est une variable ? On va le voir tout de suite.

3.Les variables

Nous avons parlé de données (« prix », « notes »…) dans un programme informatique, on devra stocker provisoirement des valeurs pour pouvoir les manipuler. Ces valeurs proviennent de l'utilisateur (clavier), du disque dur, d'une base de données qui se trouve dans un autre ordinateur…

Les valeurs utilisées dans les algorithmes peuvent être de différents types :

1. **entières** (ex : un code postal),
2. **décimales** (ex : le prix d'un vol)
3. **dates** (ex : date de naissance d'un réalisateur)
4. **texte** (ex : la description d'un appartement)
5. **booléens** ou vrai/faux (ex : un compte bancaire bloquée ou pas bloquée)

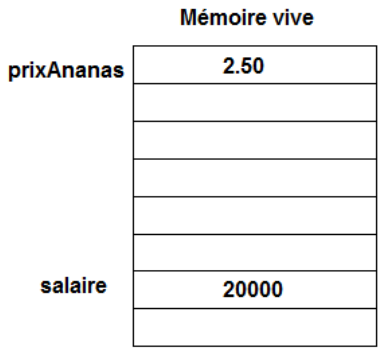
(PHP utilise ces types : String, Integer, Float, Boolean, Array, Object, NULL, Resource)

Pour stocker ces valeurs on utilise de **variables**.

Une **variable peut être vue comme une boite dans la mémoire qui contient une valeur et qui est identifiée par un nom**. Sa valeur et son nom sont stockées dans la mémoire vive. Sa valeur peut être obtenue et modifiée (pas son nom !).

Ex: quelques variables dans la mémoire vive

```
$prixAnanas = 2.50;

$salaire= 20000 ;
```



On peut créer de variables et y stocker de valeurs. Une variable a toujours un nom, un type et une valeur.

Nous allons représenter une affectation par le nom de la variable suivi d'un "=" dans tous les langages de programmation les plus connus. On écrit la nouvelle valeur de la variable juste après le symbole d'égal et sa valeur changera dans la mémoire vive !

On peut affecter une variable (changer sa valeur) autant de fois qu'on veut. Par exemple:

Le prix des ananas est maintenant 3 euros...

```
$prixAnanas = 3.00;
```

les ananas coutent maintenant 1 euro de moins...

```
$prixAnanas = $prixAnanas – 1;
```

et le prix est doublé!

```
$prixAnanas = $prixAnanas * 2;
```

Considérons maintenant la variable **monPrenom**

```
$monPrenom = "Ismael";
```

Si je change mon prénom je dois juste faire:

```
$monPrenom = "Achab";
```

Pour finir, je peux affecter une variable avec la valeur d'une autre variable. Considérons les variables **prixOrangeTable** et **prixOrangeJus**.

```
$prixOrangeTable = 2.00;
$prixOrangeJus = 1.80;
```

Si on fait :

```
$prixOrangeTable = $prixOrangeJus;
```

le prix des oranges de table sera maintenant de 1.80 euros.

Si on **continue** l'algorithme en faisant:

```
$prixOrangeJus = $prixOrangeTable;
```

la variable $prixOrangeJus contiendra 1.80, vu qu'on a changé $prixOrangeTable à 1.80 juste avant. **L'ordre des instructions déterminera toujours le résultat de l'algorithme, qui est lancé séquentiellement.**

En **PHP**, toutes les variables sont précédées du symbole '$' et chaque instruction fini par ";"

$prixFinal = $prixBase - $montantReduction;

4.Expressions et operateurs

Prenons quelques instructions qu'on a utilisées dans les algorithmes précédents:

```
$prixOrangeTable = 2.00;
$prixOrangeJus = 1.80;
$prixAnanas = $prixAnanas * 2;
```

A gauche du signe **=** il y a **toujours** uniquement un nom de variable, mais à droit il y a ce qu'on appelle une **expression.**

Une expression est un ensemble de valeurs relié par des opérateurs et équivalent à une seule valeur. Voici quelques exemples:

Expression:	**Valeur de l'expression:**
`5+3`	`8`
`10-5*2`	`0`
`8/2+3`	`7`

Dans une expression on peut aussi utiliser des variables. Considérons que prixAnanas contient la valeur numérique 3.50 et prixOrangeJus contient 1.80

Expression:	**Valeur de l'expression:**
`$prixAnanas*2`	`7`
`6+$prixAnanas`	`9.5`
`$prixOrangeJus+2`	`3.80`
`$prixOrangeJus+$prixAnanas`	`5.30`

A continuation on va étudier les expressions arithmétiques et les opérateurs arithmétiques.

4.1. Expressions Arithmétiques et Operateurs Arithmétiques

Une expression arithmétique contient de **valeurs numériques connectées par des opérateurs arithmétiques**. Les quatre **opérateurs arithmétiques** classiques sont:

+ : addition
- : soustraction
*** : multiplication**
/ : division

Exemples :

5 + 6 * 7

23 / 4

$prix * 3

$age - 20

$prix * $tva - 30

On a le droit d'utiliser les parenthèses avec les mêmes règles qu'en mathématiques. La multiplication et la division ont « naturellement » priorité sur l'addition et la soustraction. Les parenthèses ne sont ainsi utiles que pour modifier cette priorité naturelle.

Cela signifie qu'en informatique, **12 * 3 + 5** et **(12 * 3) + 5** valent strictement la même chose, à savoir 41. Pourquoi dès lors se fatiguer à mettre des parenthèses inutiles ?

En revanche, **12 * (3 + 5)** vaut 12 * 8 soit 96. Rien de difficile là-dedans, que du normal.

4.2. Opérateur de concaténation

Cet opérateur permet de concaténer, autrement dit d'agglomérer, **deux chaînes de caractères**. En PHP l'opérateur est le symbole "."

```
$nom1 = "Joe";
$nom2 = "Mary";
$nom3 = $nom2 . $nom1;
```

La valeur de $nom3 sera "MaryJoe"

5.Les tests (IFs)

Nous avons découvert déjà deux éléments de la programmation : les **variables** et **la lecture/écriture**. Les **tests** sont la troisième, et il nous restera uniquement les boucles !

Considérons l'algorithme qui gère l'accès à un site en tenant compte l'âge de l'utilisateur. On pourrait résumer l'algorithme en:

```
Demander à l'utilisateur de taper sa date de naissance

Lire l'âge de l'utilisateur

Vérifier que l'utilisateur a au moins 18 ans

Afficher un message de bienvenue au site
```

Mais qu'est-ce qui arrive si l'utilisateur est trop jeune? On devrait agir différemment et montrer un message d'erreur à l'utilisateur au lieu d'entrer dans le système et afficher le bureau !

Dans la programmation on a la possibilité d'**établir deux façons d'agir selon que la situation se présente d'une manière ou d'une autre**, grâce aux **tests**.

Voici la structure d'un test en PHP:

```php
if (condition) {
    // suite d'instructions 1
}
else  {
    // suite d'instructions 2 (alternative)
}
```

On utilise des **accolades** pour délimiter chaque section et la **condition** de l'IF se trouve **entre parenthèses**.

Si on reformule notre algorithme pour agir différemment selon la valeur de notre âge on obtient :

```
   Demander à l'utilisateur de taper sa date de naissance

   Lire l'âge de l'utilisateur

if (l'utilisateur a plus de 18 ans){

   Afficher un message de bienvenue au site

}

else {

   Afficher un message d'erreur « Reviens quand t'auras 18! »

}
```

Le fonctionnement d'un test est le suivant:

1. On évalue la **condition** du **If**

2. Si la valeur obtenue est **VRAIE (TRUE)**, on lance la **suite d'instructions 1** et on continue l'exécution de l'algorithme après la dernière accolade du if (la suite d'instructions 1 ne s'exécute pas)

3. Si la valeur obtenue est **FAUSSE (FALSE)**, on lance la **suite d'instructions 2** et on continue l'exécution de l'algorithme après la dernière accolade de l'if (la suite d'instructions 1 ne s'exécute pas)

Faites cet exemple en PHP en suivant votre intuition !

5.1. Expressions Logiques: Operateurs Logiques de base (<,>,>=,<=,==,!=)

Une **condition** est une expression qui peut être **VRAIE** ou **FAUSSE. Les conditions sont des « expressions logiques »** (pas arithmétiques!). Voici quelques exemples de base:

```
$age>18
$prix<=300
$nomUtilisateur == "Marie"
$motPasse != "3JK2-35"
```

Observez que les expressions logiques contiennent **des valeurs connectées par des opérateurs logiques**. Les **opérateurs logiques de base** sont :

- strictement plus petit que (**<**)
- strictement plus grand que (**>**)
- plus petit ou égal à (**<=**)
- plus grand ou égal à (**>=**)
- égal à (**==**). **Notez la difference avec le = simple, qui indique l'affectation!**
- différent de (**!=**)

5.2. Operateurs Logiques AND, OR, NOT et XOR

Dans certaines situations les conditions sont plus complexes que celles qu'on a vues jusqu'à présent. Considérez par exemple « l'âge est entre 12 et 18 ». On pourrait dire aussi : « l'âge est supérieure à 12 **AND** l'âge est inférieure à 18 ». On est en train de combiner deux conditions.

```
($age>12 AND $age<18)
```

On vient d'utiliser un autre **opérateur logique**, le mot clé **AND.** Cette expression logique vaut TRUE si l'âge vaut entre 13 et 17 et false dans le reste des cas.

Operateur AND : l'expression Condition1 AND Condition2 sera vraie uniquement si les deux conditions sont vraies

```
Ex : $age>12 AND $age<18
```

Sera vrai si l'âge est supérieur à 12 et inférieure à 18

Operateur OR : l'expression Condition1 OR Condition2 sera vraie si au moins une de conditions qui interviennent dans le OR est vraie.

```
Ex : $nombreEnfants>2 OR $revenus<10000
```

Operateur NOT: l'expression NOT (Condition1) sera vraie si la condition est fausse.

```
Ex :

!($etatCompte == "bloquée")
!($etatCivil == "marié")
```

On peut se demander pourquoi utiliser NOT si on pourrait faire juste:

```
$etatCompte != "bloquée"
$etatCivil != "marié"
```

C'est dans la programmation pratique où cet opérateur logique va s'avérer très utile, et il sera souvent exprimé avec le symbole "**!**" au lieu de "NOT".

Exercice: évaluez les expressions suivantes

```
3>5

4!=5

"bonjour" == "salut"

3<4 AND 9<8

3<4 OR 9<8

9<2 AND 4>3 AND 1<8

9>2 AND 8>5 AND 8>=1

8>=8

9<2 OR 4<5 AND 1<8

(9<2 OR 4<5) AND 1<8

4<9

!(4<9)

!(9<2 OR 4<5)

!(9<2) OR 4<5

!(9<2) AND !(4>10)

5==4 OR (5 < 4+3)

true AND false

true AND false OR (4<8)
```

5.3. Les IF imbriqués

Considérez un algorithme qui nous indique l'état de l'eau selon sa température (solide, liquide, gazeuse).

Une première solution serait :

```
echo "Entrez la température de l'eau :";

$temp= read();

if ($temp <= 0) {
  echo "C'est de la glace";
}

if ($temp > 0 AND $temp < 100 ) {
  echo "C'est du liquide";
}

if ($temp > 100 ) {
  echo "C'est de la vapeur";
}

echo "Au revoir";
```

Construire un test de cette manière a un désavantage : la machine est obligée de réaliser tous les tests ($temp<=0 ? $temp>0 et <100 ? $temp >100 ?), même s'ils sont mutuellement exclusifs...

Nous pouvons structurer le test d'une autre façon pour obtenir le même résultat, grâce **à l'imbrication de tests**:

```
echo "Entrez la temp. de l'eau :";

$temp = read();

if ($temp <= 0){
    echo "C'est de la glace";
}
elseif ($temp < 100){
    echo "C'est du liquide";
}
else{
    echo "C'est de la vapeur";
}

echo "Au revoir";
```

On explique ce code à continuation. Observez qu'il y a un nouveau mot « **elseIf** »

Explication :

On exécute le premier test (Temp<=0) et

1. Si c'est VRAI, la machine lance la suite d'instructions jusqu'au **elseIf** et après elle saut à l'instruction après la dernière accolade du if (echo « au revoir »). Le reste de tests ne se font pas, on n'a plus besoin.

2. Si c'est FAUX, la machine fait le test du elseIf (Temp<100?).
   - Si le test est Vrai, on lance la suite d'instructions correspondante et après on saut à l'instruction après la dernière accolade du if
   - Si le test est Faux on sait déjà que c'est de la vapeur : la température n'est pas <0 ni <100… alors elle est >=100 !

Remarquez aussi que si les deux premiers tests ne sont pas vrais nous ne sommes même pas obligés de réaliser un troisième test: si $temp n'est pas inférieure à 0 ni inférieure à 100…. $temp est forcément >=100 et l'eau est forcément de la vapeur!

On aurait pu mettre autant de elseIfs qu'on veut, il suffit de bien placer les accolades.

6.Les Boucles

Les boucles sont la quatrième et dernière structure de programmation. Elles permettent de **répéter une suite d'instructions**: on les appelle **structures répétitives** ou **structures itératives**.

Considérez le cas d'un logiciel de console (pas une page web avec un formulaire) qui pose une question à l'utilisateur. L'utilisateur doit répondre Oui ("O") ou Non ("N"):

```
echo "Voulez-vous un chocolat ? (O/N)";

$rep= read();


if ($rep == "O"){
  echo "Tenez votre chocolat";
}
elseif ($rep == "N") {
  echo "Ok, pas de chocolat!";
}
```

Si l'utilisateur se trompe (ce qui arrive tout le temps!) et saisie une autre valeur que "O" ou "N", le programme n'affichera rien. On peut améliorer le programme: si l'utilisateur se trompe, on va lui demander de saisir une valeur à nouveau:

```
echo "Voulez-vous un chocolat ? (O/N)";

$rep= read();


if ($rep!="O" AND $rep!="N"){
    echo "Saisie erronée. Recommencez.";
    $rep= read();
}

if ($rep == "O"){
  echo "Tenez votre chocolat";
}
elseif ($rep == "N") {
  echo "Ok, pas de chocolat!";
}
```

L'utilisateur a le droit à se tromper une fois en tapant autre que "O" ou "N", mais… qu'est-ce qui se passe s'il se trompe une deuxième fois? Si on veut permettre à l'utilisateur de se tromper plusieurs fois, on devrait utiliser plusieurs IF:

```
echo "Voulez-vous un chocolat ? (O/N)";

$rep= read();


if ($rep!="O" AND $rep!="N"){
    echo "Saisie erronée. Recommencez.";
    $rep= read();
}
if ($rep!="O" AND $rep!="N"){
    echo "Saisie erronée. Recommencez.";
    $rep= read();
}

if ($rep == "O"){
  echo "Tenez votre chocolat";
}
elseif ($rep == "N") {
  echo "Ok, pas de chocolat!";
}
```

.
.
.

Pas terrible! Pourquoi?

La **solution** à ce problème passe par l'utilisation d'une **boucle**. Une boucle nous permet de répéter une suite d'instructions: dans notre cas on vuex répéter la lecture de la réponse de l'utilisateur jusqu'à elle nous convienne. Il y existe plusieurs types de boucles, dont les plus importants sont **while** et **for**. Bien qu'on puisse tout faire avec la boucle **while**, les boucles **for** nous aident à simplifier le code et on les utilise partout.

6.1. Boucle While

```
while (condition) {
…
   Instructions
}
```

Boucle While: Le programme arrive sur la ligne **While** et il examine la condition (ou une valeur booléenne directement). Si la condition est vraie il exécute les instructions à l'intérieur des accolades. Le processus se répète pendant que la condition soit vraie.

Exemple:

Le programme teste la valeur de la variable $rep et répète le code dans la boucle pendant que la valeur est différente de "O" et "N".

En **PHP (attention aux accolades!)**

```php
echo "Voulez-vous un chocolat ? (O/N)";

$rep= read();

while  ($rep!="O" AND $rep!="N"){
    echo "Saisie erronée. Recommencez.";
    $rep= read();
}



if ($rep == "O"){
  echo "Tenez votre chocolat";
}
elseif ($rep == "N") {
  echo "Ok, pas de chocolat!";
}
```

6.2. Compter en utilisant une boucle

On peut utiliser une boucle pour compter ou pour réaliser une action un nombre défini de fois.

Exemple: afficher les valeurs de 0 à 15

```
$compteur = 0;

while ($compteur<= 15){
    echo $compteur;
    $compteur = $compteur + 1;
}
```

Pour compter en utilisant une boucle while on doit d'abord **créer une variable compteur et l'initialiser**.

Cette variable doit être **incrémentée dans la boucle, ou on restera indéfiniment bloqués à l'intérieur!**. Si ça arrive, on est dans une **boucle infinie**. Cette erreur peut avoir de conséquences très importantes.

On peut aussi utiliser ce système pour exécuter une suite d'instructions un nombre défini de fois. Dans ce cas, le compteur sert uniquement à déterminer le nombre de répétitions.

Exemple: lire le nom de 5 personnes et leur dire bonjour à chacune. Dans ce cas, le compteur nous sert uniquement à répéter les instructions, mais on ne l'utilise nulle part ailleurs.

```
$compteur = 0;
while ($compteur<= 5){
    $nom= read();
    echo "Bonjour " . $nom;
    $compteur = $compteur + 1;
}
```

ATTENTION:

J'insiste sur le fait que la condition de la boucle doit devenir fausse à un certain moment. Si non, vous créerez une boucle infinie qui, sauf dans certains cas très précis, bloquera votre application et peut avoir de conséquences graves.

Exemple: quel est le problème de cette boucle? Et la solution?

```
$compteur = 0;

while ($compteur<= 5){
    $nom= read();
    echo "Bonjour " . $nom;
}
```

Exemple: quel est le problème de cette boucle? Et la solution?

```
$compteur = 0;

while ($compteur<= 50){
    $nom= read();
    echo "Bonjour " . $nom;
    $compteur + 1;
}
```

6.3. Quand est-ce qu'on utilise de boucles ?

- Répéter une action x fois: condition dépendante d'un compteur

  Ex: afficher dix fois le message "Bonjour"

- Répéter une action jusqu'au changement de la valeur d'une variable qui fasse partie dans la condition

  Ex: voulez-vous un chocolat? Tapez (oui/non)

- Combiner les deux exemples précédant

  Ex: voulez-vous un chocolat? Tapez (oui/non). L'utilisateur peut se tromper deux fois

- Répéter une action x fois et accumuler un résultat dans une ou plusieurs variables qu'on utilisera en dehors de la boucle

  Ex: additionner les chiffres entre 10 et 20 (incluses)

- De combinaisons de toutes les actions précédantes

Voici quelques exemples :

```
/* Exemple du café, sans compteur */
print ("Voulez vous un café?");
$reponse = read();

while ($reponse != "oui" && $reponse != "non"){
    print ("Svp. tapez oui ou non");
    $reponse = read();
}
// c'est oui ou non
if ($reponse == "oui"){
    print ("Tenez votre café");
}
else {
    print ("Pas de café!");
}

/* Exemple de compteur */
$nombreBombons = 20;

while ($nombreBombons > 0){
    print ("Il vous reste " + $nombreBombons);
    print ("Tenez un bombon");
    $nombreBombons = $nombreBombons - 1;
}
print ("Plus de bombons!");
```

```
/* Exemple d'essais: variable + compteur */
$chiffreDeviner = 3;
$nombreEssais = 10;

print ("Tapez un chiffre entre 1 et 5");
$reponse = read();
while ($nombreEssais > 0 && $chiffreDeviner != $reponse){
    print ("Essayez encore!");
    $reponse = read();
    $nombreEssais = $nombreEssais -1;
}
if ($reponse == $chiffreDeviner){
    print ("Gagné!");
}
else {
    print ("Perdu! Plus d'essais!");
}
```

6.4. Boucle For

La boucle For est une façon de simplifier le comptage à l'utilisateur. Observez bien cette boucle While, qu'on a déjà vu:

```
$compteur=0;

while ($compteur< 5){
    echo $compteur;
    $compteur = $compteur + 1;
}
```

Elle fait exactement la même chose que cette boucle **For:**

```
for ($compteur=0;$compteur<5;$compteur++){
    echo $compteur;
}
```

Note :

```
$compteur++ : post-incrémentation
++$compteur : pre-incrémentation
```

$a = 3 ;

print ($a++) ; // 3

print (++$a) ; // 4

Il y a juste deux détails concernant la syntaxe, mais la fonction est la même:

1. dans la deuxième boucle **la variable est initialisée dans la boucle même**
2. dans la deuxième boucle **on incrémente le compteur dans la boucle même**. il passera par tous les valeurs entre 0 et 4, alors la boucle s'exécutera 5 fois)

Boucle For

Quand le programme arrive par la première fois à la ligne **For**, il **assigne au compteur la valeur Initial, vérifie la condition (ex: $compteur< $valeurFinale)** et **lance une première fois la suite d'instructions.**

Une fois exécutée la suite d'instructions, il **incrémente le compteur** et **revérifie la condition.** Si c'est le cas, il lance à nouveau la suite d'instructions, incrémente le compteur et revient à la vérification. Si non, il sort de la boucle.

Autre exemple :

```
echo "Voulez-vous un chocolat ? (O/N)";

$rep= read();

while  ($rep!="O" AND $rep!="N"){
    echo "Saisie erronée. Recommencez.";
    $rep= read();
}


for ( $rep = read(); $rep!="O" AND $rep!="N"; $rep = read()){
   echo "Saisie erronée. Recommencez.";
}
```

6.5. Boucles imbriqués

De la même manière qu'on a des **if** successifs (un après l'autre) et imbriqués (un à l'intérieur de l'autre) on peut avoir de boucles successifs et de boucles imbriquées.

Faisons quelques exemples:

Boucles successifs:

```
for ($compteur1=1;$compteur1<15;$compteur1=$compteur1+1){
    echo "Il est passé par ici";
}
for ($compteur2=1;$compteur2<5;$compteur2=$compteur2+1){
    echo "Il repassera par là";
}
```

Ce premier cas est simple: il y aura quinze écritures consécutives de "il est passé par ici", puis cinq écritures consécutives de "il repassera par là".

Boucles imbriqués:

```
for ($compteur1=1;$compteur1<15;$compteur1=$compteur1+1){
    echo "Il est passé par ici" . $compteur1 ;
    for ($compteur2=1;$compteur2<5;$compteur2=$compteur2+1){
            echo "Il repassera par là" . $compteur2;
    }
}
```

Résultat :

```
Il est passé par ici 1
      Il repassera par là 1
      Il repassera par là 2
      Il repassera par là 3
      Il repassera par là 4
Il est passé par ici 2
      Il repassera par là 1
      Il repassera par là 2
      Il repassera par là 3
      Il repassera par là 4
.
.
Il est passé par ici 14
      Il repassera par là 1
      Il repassera par là 2
      Il repassera par là 3
```

```
      Il repassera par là 4

```

Ici, le programme écrira une fois "il est passé par ici" puis cinq fois de suite "il repassera par là", et ceci quinze fois en tout.

Pourquoi imbriquer les boucles?

Une **boucle, c'est un traitement systématique, un examen d'une série d'éléments un par un** (par exemple, « prenons tous les employés de l'entreprise un par un »).

On peut imaginer que **pour chaque élément** ainsi considéré (pour chaque employé), on doive procéder à **un examen systématique d'autre chose** (« prenons chacune des commandes que cet employé a traitées »). Voilà un exemple typique de boucles imbriquées : on devra programmer une boucle principale (celle qui prend les employés un par un) et à l'intérieur, une boucle secondaire (celle qui prend les commandes de cet employé une par une).

Les boucles imbriquées sont **indispensables pour, par exemple, parcourir un tableau de deux (ou plus) dimensions**: pour chaque ligne du tableau on doit parcourir tous les éléments (chaque colonne). Les résultats d'une requête à une base de données sont traités de cette façon.

7.Les tableaux (arrays)

Considérez que vous avez besoin, dans votre application web d'achat online, de manipuler simultanément le prix de 10 produits différents dans un panier pour calculer le prix total.

Notre premier reflex est de créer une variable pour chaque prix: prix1, prix2 prix3… prix10.

Imaginez maintenant qu'il y a 100 produits: err…. ce système est impossible à gérer! On devrait créer à la main 100 variables!

Les **tableaux (arrays)** nous permettent de rassembler les variables indépendantes dans une seule variable, référencée par un seul nom.

Dans notre cas, on va créer un tableau nommé **$listePrix** de 10 positions qui contiendra un prix dans chaque position.

Pour créer l'array on va utiliser la ligne suivante:

```
$listePrix = [];
```

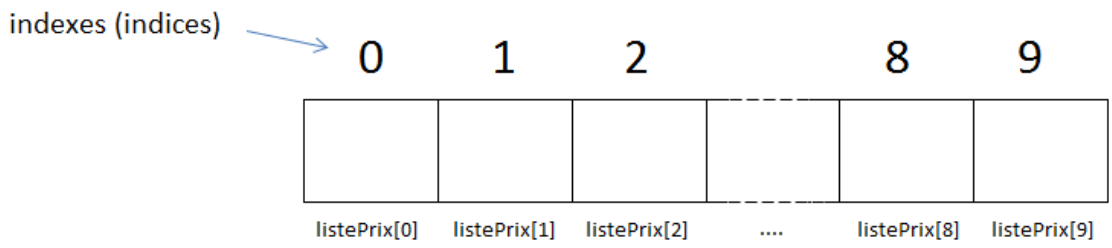
ou

```
$listePrix = [];
```

Note: dans plein de langages de programmation on peut spécifier la taille initiale d'un array dans la forme:

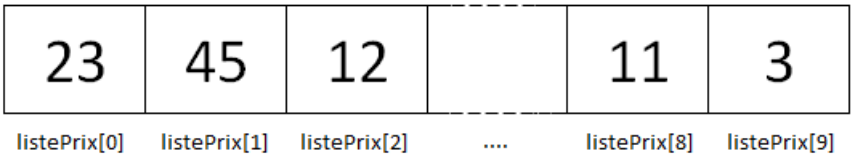
```
$listePrix = int[10];   // array de 10 entiers
```

Mais en php les arrays sont dynamiques et on n'indique pas leur taille lors de sa création



Pour accéder à chaque **valeur** de l'array (chaque prix dans notre cas), on utilise le nom de la variable suivie d'un index **(clé)** entre crochets. Voici quelques exemples:

```php
$listePrix[0] = 23; // modifier la 1ere valeur, accessible par la clé 0


$listePrix[1] = 45; // modifier la 2ème valeur (clé 1)
// avec 45

echo $listePrix[1];     // afficher la 2ème valeur


$listePrix[1]=read();
// stocker le contenu saisi
// par l'utilisateur dans la position 2


echo $listePrix[1];     // afficher la deuxième valeur

$listePrix[9] = $listePrix[0]-20; // reduire le prix de la valeur
                                  // dans la position 9
```



La notation de crochets nous sert à accéder au tableau. On peut réaliser deux opérations:

1) Accéder au contenu d'un élément du tableau

Ex1: afficher le troisième prix

```php
echo $listePrix [2]
```

2) Modifier le contenu d'un élément du tableau

Encore des exemples:

```php
$listePrix[5] = 50;
$listePrix[5] = $listePrix[3];
$listePrix[9] = 2* $listePrix[2];
$val= read();
$listePrix[2]=$val;
```

Important!

1. Un tableau **peut contenir des variables de toute sorte** (numériques, texte, booléens, même d'autres tableaux!). Normalement, tous les éléments d'un tableau sont du même type.
2. Les indices généres automatiquement par PHP dans les arrays qu'on a étudié sont toujours **positifs et commencent par 0, pas par 1**! L'index 1 est déjà la deuxième position du tableau
3. Les **clés (ou indices) sont souvent de nombres entiers** (exception: tableaux associatifs dans certain langages comme PHP)

7.1. Les tableaux et les boucles

On apprécie vraiment les avantages des tableaux quand on les utilise avec de boucles:

Ex: afficher les éléments du tableau $listePrix (déjà rempli)

```
for ($compteur=0;$compteur<10;$compteur=$compteur+1){
     echo $listePrix [$compteur];
}
```

Ex: lire 5 noms du clavier et les stocker dans un tableau

```
$listeNoms = [];
for ($compteur=0;$compteur<5;$compteur=$compteur+1){
    $val= read();
    $listeNoms[$compteur]=$val;
}
```

Pour simplifier la notation, normalement on utilise les noms **i**, **j**, **k** ou même **n** pour les variables compteur:

```
$listeNoms = [];
for ($i=0;$i<5;$i=$i+1){
    $val= read();
    $listeNoms[$i]=$val;
}
```