Chapitre 4
Écrire des fonctions et des classes PHP

1. Fonctions

1.1 Introduction

À l'instar des différents langages de développement, PHP offre la possibilité de définir ses propres fonctions (appelées fonctions "utilisateur") avec tous les avantages associés (modularité, capitalisation...). Une fonction est un ensemble d'instructions identifiées par un nom, dont l'exécution retourne une valeur et dont l'appel peut être utilisé comme opérande dans une expression. Une procédure est un ensemble d'instructions identifiées par un nom qui peut être appelé comme une instruction.

1.2 Déclaration et appel

Le mot-clé `function` permet d'introduire la définition d'une fonction.

Syntaxe

```
function nom_fonction([paramètre]) [: type]{
  instructions;
}
```

`nom_fonction` Nom de la fonction (doit respecter les règles de nommage présentées dans le chapitre Introduction à PHP - Structure de base d'une page PHP). Ce nom n'est pas sensible à la casse (pour PHP, les fonctions `unefonction` et `UneFonction` sont les mêmes).

`paramètre` Paramètres éventuels de la fonction exprimés sous forme d'une liste de variables (cf. section Fonctions - Paramètres) : `$paramètre1, $paramètre2, ...`

`type` Déclaration du type de données retourné par la fonction. Valeurs possibles : `int`, `float`, `string`, `bool`, `array`, `callable`, `iterable`, `object`, `mixed`, `void`, un nom de classe ou d'interface (cf. dans ce chapitre la section Classes), ou une union de types. Le nom du type peut être précédé d'un point d'interrogation (`?`) qui indique que la fonction peut retourner une valeur `NULL`. Voir le chapitre Les bases du langage PHP pour la définition des types de données (section Les bases du langage PHP - Types de données).

`instructions` Ensemble des instructions qui composent la fonction.

Le nom de la fonction ne doit pas être un mot réservé PHP (nom de fonction native, d'instruction) ni être égal au nom d'une autre fonction préalablement définie.

Une fonction utilisateur peut être appelée comme une fonction native de PHP : dans une affectation, dans une comparaison, etc.

Si la fonction retourne une valeur, il est possible d'utiliser l'instruction `return` pour définir la valeur de retour de la fonction.

Syntaxe

```
return expression;
```

`expression` Expression dont le résultat constitue la valeur de retour de la fonction (`NULL` par défaut).

Le résultat d'une fonction peut être de n'importe quel type (chaîne, nombre, tableau, etc.).

L'instruction `return` stoppe l'exécution de la fonction et retourne le résultat de `expression` à l'appelant. Si plusieurs instructions `return` sont présentes dans la fonction, c'est la première rencontrée dans le déroulement des instructions qui définit la valeur de retour et provoque l'interruption de la fonction. Si la fonction ne comporte aucune instruction `return` (ou si aucune instruction `return` n'est exécutée), la valeur de retour de la fonction est `NULL`.

Exemple

```
<?php
// Fonction sans paramètre qui affiche "Bonjour !"
// Pas de valeur de retour.
function afficher_bonjour() {
  echo 'Bonjour !<br />';
}
// Fonction avec deux paramètres qui retourne le produit
// des deux paramètres.
function produit($valeur1,$valeur2) {
  return $valeur1 * $valeur2;
}
// Appel de la fonction afficher_bonjour.
afficher_bonjour();
// Utilisations de la fonction produit :
// - dans une affectation
$résultat = produit(2,4);
echo "2 x 4 = $résultat<br />";
// - dans une comparaison
if (produit(10,12) > 100) {
  echo '10 x 12 est supérieur à 100.<br />';
}
?>
```

Résultat

```
Bonjour !
2 x 4 = 8
10 x 12 est supérieur à 100.
```

Remarque

Dans le langage PHP, il n'existe pas à proprement parler de procédure. Pour définir quelque chose d'équivalent à une procédure, il suffit de définir une fonction qui ne retourne pas de valeur et d'appeler la fonction comme s'il s'agissait d'une instruction (comme la fonction `afficher_bonjour` par exemple). Une fonction qui ne retourne rien peut explicitement être déclarée avec le type de retour `void`.

Comme nous l'avons déjà évoqué, le contenu d'un tableau peut être transformé en liste de paramètres dans un appel de fonction grâce à l'opérateur `...` (points de suspension).

Exemple

```
<?php
// Fonction avec trois paramètres qui retourne la somme
// des trois paramètres.
function somme($valeur1,$valeur2,$valeur3) {
  return $valeur1 + $valeur2 + $valeur3;
}
// Transformation du contenu d'un tableau en
// liste de paramètres.
$valeurs = [1,2,3];
echo '1 + 2 + 3 = ',somme(...$valeurs),'<br />';
// La même chose pour une partie seulement des paramètres
// avec un tableau défini directement dans l'appel.
echo '1 + 2 + 4 = ',somme(1,...[2,4]),'<br />';
?>
```

Résultat

```
1 + 2 + 3 = 6
1 + 2 + 4 = 7
```

Lorsqu'une fonction retourne un tableau, il est possible d'accéder directement à un élément du tableau lors de l'appel à la fonction avec une syntaxe du type `fonction(...)[clé]`.

Exemple

```
<?php
// Définition d'une fonction qui retourne un tableau.
function qui() {
  return ['Olivier','Heurtel'];
}
// Appel de la fonction et récupération directe du prénom stocké
// à l'indice 0 du tableau retourné.
$prénom = qui()[0];
echo "qui()[0] = $prénom<br />";
?>
```

Résultat

```
qui()[0] = Olivier
```

Cette technique fonctionne aussi lorsque la fonction retourne un tableau multidimensionnel avec une syntaxe du type `fonction(...)[clé1][clé2]`.

Il est possible d'utiliser une fonction avant de la définir.

Exemple

```
<?php
// Utilisation de la fonction produit.
echo produit(5,5);
// Définition de la fonction produit.
function produit($valeur1,$valeur2) {
  return $valeur1 * $valeur2;
}
?>
```

Résultat

```
25
```

Il n'y a donc aucun problème pour définir des fonctions qui s'appellent entre elles.

Remarque

Une fonction est utilisable uniquement dans le script où elle est définie. Pour l'employer dans plusieurs scripts, il faut, soit recopier sa définition dans les différents scripts (vous perdez l'intérêt de définir une fonction), soit la définir dans un fichier inclus partout où la fonction est nécessaire.

Exemple

– Fichier `fonctions.inc` contenant des définitions de fonctions :

```
<?php
// Définition de la fonction produit.
function produit($valeur1,$valeur2) {
  return $valeur1 * $valeur2;
}
?>
```

– Script utilisant les fonctions définies dans `fonctions.inc` :

```
<?php
// Inclusion du fichier contenant la définition des fonctions.
include('fonctions.inc');
// Utilisation de la fonction produit.
echo produit(5,5);
?>
```

Déclaration du type de retour

Il est possible de définir le type de données retourné par une fonction.

Lorsque c'est le cas, dans le mode de fonctionnement par défaut (à opposer au mode strict présenté ci-dessous), PHP effectue si besoin une conversion automatique de la valeur retournée dans le type de données déclaré.

Exemple

```
<?php
// Déclaration de deux fonctions qui retournent le produit
// des deux paramètres, la deuxième spécifiant un type
// de données "entier" pour la valeur de retour.
function produit1($valeur1,$valeur2) {
  return $valeur1 * $valeur2;
}
function produit2($valeur1,$valeur2) : int {
  return $valeur1 * $valeur2;
}
// Appel des deux fonctions avec les mêmes paramètres
echo 'produit1(20,1/7) => ',var_dump(produit1(20,1/7)),'<br />';
echo 'produit2(20,1/7) => <b>',var_dump(produit2(20,1/7)),'</b><br />';
?>
```

Résultat

```
produit1(20,1/7) => float(2.8571428571428568)
produit2(20,1/7) => int(2)
```

Sur cet exemple, nous voyons bien que la valeur retournée par la deuxième fonction a été convertie en entier par PHP (avec les règles de conversion évoquées dans le chapitre Introduction à PHP - Les bases du langage PHP - Types de données).

Si PHP n'est pas en mesure d'effectuer la conversion (types de données non convertibles entre eux), une exception `TypeError` est levée ; cette exception interrompt le script si elle n'est pas gérée (cf. dans ce chapitre la section Classes - Exceptions).

Exemple

```
<?php
// Déclaration et appel d'une fonction qui doit retourner un
// tableau mais qui retourne une chaîne de caractères.
function qui() : array {
  return 'Olivier Heurtel';
}
echo 'qui()[0] = ',qui()[0];
?>
```

Résultat

```
qui()[0] =
Fatal error: Uncaught TypeError: Return value of qui() must be of the
type array, string returned in /app/scripts/index.php:5 Stack trace: #0
/ app/scripts/index.php(7): qui() #1 {main} thrown in/app/scripts/
index.php on
line 5
```

Une fonction déclarée avec un type de retour autre que `void` doit retourner une valeur non `NULL`. Si ce n'est pas le cas, une erreur est retournée, différente selon les cas :

Absence d'instruction `return`

```
Fatal error: Uncaught TypeError: Return value of MaFonction() must be of
the type int, none returned in ...
```

Instruction `return` vide

```
Fatal error: A function with return type must return a value in ...
```

Instruction `return NULL`

```
Fatal error: Uncaught TypeError: Return value of MaFonction() must be of
type int, null returned in ...
```

Pour autoriser une fonction à retourner une valeur `NULL`, il faut faire précéder le nom du type (autre que `void`) d'un point d'interrogation (`?`).

```
<?php
// Déclaration et appel d'une fonction qui spécifie un
// type de donnée de retour qui peut être NULL
function cube($valeur) : ?int {
  if (is_null($valeur)) {
    return NULL;
  } else {
    return $valeur ** 3 ;
  }
}
echo 'cube(2) => <b>',var_dump(cube(2)),'</b><br />';
echo 'cube(NULL) => <b>',var_dump(cube(NULL)),'</b><br />';
?>
```

Résultat

```
cube(2) => int(8)
cube(NULL) => NULL
```

Même avec cette option, la fonction doit avoir une instruction `return` non vide. Si ce n'est pas le cas, une erreur est retournée, différente selon les cas :

Absence d'instruction `return`

```
Fatal error: Uncaught TypeError: Return value of MaFonction() must be of
type ?int, none returned in ...
```

Instruction `return` vide

```
Fatal error: A function with return type must return a value (did you mean
"return null;" instead of "return;"?) in ...
```

À l'inverse, il est possible de déclarer qu'une fonction ne retourne rien, en utilisant `void` comme nom de type. Dans ce cas, la fonction doit omettre l'instruction `return` ou mettre une instruction `return` vide, sans valeur (même pas `NULL`).

Chapitre 3
Tests et logique booléenne

1. Les tests et conditions

1.1 Principe

Dans le précédent chapitre, vous avez pu vous familiariser avec les expressions mettant en place des opérateurs, qu'ils soient de calcul, de comparaison (l'égalité) ou booléens. Ces opérateurs et expressions trouvent tout leur sens une fois utilisés dans des conditions (qu'on appelle aussi des branchements conditionnels). Une expression évaluée est ou vraie (le résultat est différent de zéro) ou fausse. Suivant ce résultat, l'algorithme va effectuer une action, ou une autre. C'est le principe de la condition.

Grâce aux opérateurs booléens, l'expression peut être composée : plusieurs expressions sont liées entre elles à l'aide d'un opérateur booléen, éventuellement regroupées avec des parenthèses pour en modifier la priorité.

```
(a=1 OU (b*3=6)) ET c>10
```

est une expression tout à fait valable. Celle-ci sera vraie si chacun de ses composants respecte les conditions imposées. Cette expression est vraie si a vaut 1 et c est supérieur à 10 ou si b vaut 2 (2*3=6) et c est supérieur à 10.

Reprenez l'algorithme du précédent chapitre qui calcule les deux résultats possibles d'une équation du second degré. L'énoncé simplifié disait que pour des raisons pratiques seul le cas où l'équation a deux solutions fonctionne. Autrement dit, l'algorithme n'est pas faux dans ce cas de figure, mais il est incomplet. Il manque des conditions pour tester la valeur du déterminant : celui-ci est-il positif, négatif ou nul ? Et dans ces cas, que faire et comment le faire ?

Imaginez un second algorithme permettant de se rendre d'un point A à un point B. Vous n'allez pas le faire ici réellement, car c'est quelque chose de très complexe sur un réseau routier important. De nombreux sites Internet vous proposent d'établir un trajet avec des indications. C'est le résultat qui est intéressant. Les indications sont simples : allez tout droit, tournez à droite au prochain carrefour, faites trois kilomètres et au rond-point prenez la troisième sortie direction B. Dans la plupart des cas, si vous suivez ce trajet vous arrivez à bon port. Mais quid des impondérables ? Par où allez-vous passer si la route à droite au prochain carrefour est devenue un sens interdit (cela arrive parfois, y compris avec un GPS, prudence) ou que des travaux empêchent de prendre la troisième sortie du rond-point ?

Reprenez le trajet : allez tout droit. Si au prochain carrefour la route à droite est en sens interdit : continuez tout droit puis prenez à droite au carrefour suivant puis à gauche sur deux kilomètres jusqu'au rond-point. Sinon : tournez à droite et faites trois kilomètres jusqu'au rond-point. Au rond-point, si la sortie vers B est libre, prenez cette sortie. Sinon, prenez vers C puis trois cents mètres plus loin tournez à droite vers B.

Ce petit parcours ne met pas uniquement en lumière la complexité d'un trajet en cas de détour, mais aussi les nombreuses conditions qui permettent d'établir un trajet en cas de problème. Si vous en possédez, certains logiciels de navigation par GPS disposent de possibilités d'itinéraire Bis, de trajectoire d'évitement sur telle section, ou encore pour éviter les sections à péage. Pouvez-vous maintenant imaginer le nombre d'expressions à évaluer dans tous ces cas de figure, en plus de la vitesse autorisée sur chaque route pour optimiser l'heure d'arrivée ?

1.2 Que tester ?

Les opérateurs s'appliquent sur quasiment tous les types de données, y compris les chaînes de caractères, tout au moins en pseudo-code algorithmique. Vous pouvez donc quasiment tout tester. Par tester, comprenez ici évaluer une expression qui est une condition. Une condition est donc le fait d'effectuer des tests pour, en fonction du résultat de ceux-ci, effectuer certaines actions ou d'autres.

Une condition est donc une affirmation : l'algorithme et le programme ensuite détermineront si celle-ci est vraie, ou fausse.

Une condition retournant VRAI ou FAUX a comme résultat un **booléen**.

En règle générale, une condition est une comparaison même si en programmation une condition peut être décrite par une simple variable (ou même une affectation) par exemple. Pour rappel, une comparaison est une expression composée de trois éléments :

– une première valeur : variable ou scalaire
– un opérateur de comparaison
– une seconde valeur : variable ou scalaire.

Les opérateurs de comparaison sont :

– L'égalité : **=**
– La différence : **!=** ou **<>**
– Inférieur : **<**
– Inférieur ou égal : **<=**
– Supérieur : **>**
– Supérieur ou égal : **>=**

Le pseudo-code algorithmique n'interdit pas de comparer des chaînes de caractères. Évidemment, vous prendrez soin de ne pas mélanger les torchons et les serviettes, en ne comparant que les variables de types compatibles. Dans une condition une expression, quel que soit le résultat de celle-ci, sera toujours évaluée comme étant soit vraie, soit fausse.

Remarque

L'opérateur d'affectation peut aussi être utilisé dans une condition. Dans ce cas, si vous affectez 0 à une variable, l'expression sera fausse et si vous affectez n'importe quelle autre valeur, elle sera vraie.

En langage courant, il vous arrive de dire "choisissez un nombre entre 1 et 10". En mathématique, vous écrivez cela comme ceci :

```
1 ≤ nombre ≤ 10
```

Si vous écrivez ceci dans votre algorithme, attendez-vous à des résultats surprenants le jour où vous allez le convertir en véritable programme. En effet les opérateurs de comparaison ont une priorité, ce que vous savez déjà, mais l'expression qu'ils composent est aussi souvent évaluée de gauche à droite. Si la variable nombre contient la valeur 15, voici ce qui ce passe :

– L'expression 1 = 15 est évaluée : elle est vraie.
– Et ensuite ? Tout va dépendre du langage, l'expression suivante vrai = 10 peut être vraie aussi.
– Le résultat est épouvantable : la condition est vérifiée et le code correspondant va être exécuté !

Vous devez donc proscrire cette forme d'expression. Voici celles qui conviennent dans ce cas :

```
nombre>=1 ET nombre<=10
```

Ou encore

```
1<=nombre ET nombre<=10
```

1.3 Tests SI

1.3.1 Forme simple

Il n'y a, en algorithmique, qu'une seule instruction de test, "**Si**", qui prend cependant deux formes : une simple et une complexe. Le test SI permet d'exécuter du code si la condition (la ou les expressions qui la composent) est vraie.

La forme simple est la suivante :

```
Si booléen Alors
  Bloc d'instructions
FinSi
```

Notez ici que le booléen est la condition. Comme indiqué précédemment, la condition peut aussi être représentée par une seule variable. Si elle contient 0, elle représente le booléen FAUX, sinon le booléen VRAI.

Que se passe-t-il si la condition est vraie ? Le bloc d'instructions situé après le "**Alors**" est exécuté. Sa taille (le nombre d'instructions) n'a aucune importance : de une ligne à n lignes, sans limite. Dans le cas contraire, le programme continue à l'instruction suivant le "**FinSi**". L'exemple suivant montre comment obtenir la valeur absolue d'un nombre avec cette méthode.

```
PROGRAMME ABS
VAR
  Nombre :entier
DEBUT
  nombre←-15
  Si nombre<0 Alors
    nombre←-nombre
  FinSi
  Afficher nombre
FIN
```

En PHP, c'est le "if" qui doit être utilisé avec l'expression booléenne entre parenthèses. La syntaxe est celle-ci :

```
if(boolean) { /*code */ }
```

Si le code PHP ne tient que sur une ligne, les accolades peuvent être supprimées, comme dans l'exemple de la valeur absolue. Cet exemple montre une seconde possibilité par les mécanismes offerts par les bibliothèques de fonction de PHP.

```
<html>
  <head><meta/>
    <title>ABS</title>
  </head>
  <body>
  <?php
    $nombre=-15;
```

```
    if($nombre<0) $nombre=-$nombre;
    echo $nombre;

    echo "<br />";
  // seconde possibilitité
    $nombre2=-32;
    $nombre2=abs($nombre2);
    echo $nombre2;
  ?>
  </body>
</html>
```

1.3.2 Forme complexe

La forme complexe n'a de complexe que le nom. Il y a des cas où il faut exécuter quelques instructions si la condition est fausse sans vouloir passer tout de suite à l'instruction située après le FinSi. Dans ce cas, utilisez la forme suivante :

```
Si booléen Alors
  Bloc d'instructions
Sinon
  Bloc d'instructions
FinSi
```

Si la condition est vraie, le bloc d'instructions situé après le Alors est exécuté. Ceci ne diffère pas du tout de la première forme. Cependant, si la condition est fausse, cette fois c'est le bloc d'instructions situé après le Sinon qui est exécuté. Ensuite, le programme reprend le cours normal de son exécution après le FinSi.

Notez que vous auriez pu très bien faire un équivalent de la forme complexe en utilisant deux formes simples : la première avec la condition vraie, la seconde avec la négation de cette condition. Mais ce n'est pas très joli, même si c'est correct. Retenez que :

– Si dans une forme complexe, l'un des deux blocs d'instructions est vide, alors transformez-la en forme simple : modifiez la condition en conséquence.
– Laisser un bloc d'instructions vide dans une forme complexe n'est pas conseillé : c'est une grosse maladresse de programmation qui peut être facilement évitée, c'est un programme sale. Cependant, certains langages l'autorisent, ce n'est pas une erreur ni même une faute lourde, mais ce n'est pas une raison…