Créer ses propres fonctions

Table des matières

I. Contexte **3**

II. Écrire des fonctions en PHP **3**

A. Déclarer, nommer et exécuter ses fonctions 3

B. Signature d'une fonction : arguments, retour et indications de type 6

C. Passage des arguments par valeur 8

D. Exercice : Quiz 12

III. Utiliser les fonctions pour améliorer son code **14**

A. Utiliser les fonctions pour améliorer son code 14

B. Exercice : Quiz 17

IV. Essentiel **18**

V. Auto-évaluation **18**

A. Exercice 18

B. Test 19

Solutions des exercices **20**

I. Contexte

Durée : 1 h

Environnement de travail : éditeur de texte, terminal ou IDE Visual Studio Code

Prérequis : les bases en PHP

Contexte

Tout programme informatique, aussi complexe soit-il, est composé de deux parties : un ensemble de **fonctions** ou procédures, et des **données**. Les données sont des choses que nous souhaitons manipuler, les fonctions sont les descriptions des règles pour manipuler ces données. Comparez votre programme à un haut fourneau : il y a la matière première (minerai de fer, charbon, oxygène, etc.), ce sont les données, et tout le matériel pour les transformer en acier (le fourneau, les conduites, les rigoles pour faire s'écouler le métal fondu, etc.), ce sont les fonctions. Ces deux parties forment le **code source** qui, une fois compilé ou interprété, est transformé en programme dont l'exécution par la machine résout un problème donné.

Le code source, en plus de refléter la complexité inhérente du problème qu'il est censé résoudre, hérite toujours d'une complexité résiduelle et indésirable. Cette complexité vient du fait qu'il faille traduire une idée, un processus de pensée, en un ensemble limité d'instructions fournies par le langage de programmation. Il faut par exemple manipuler de la mémoire, vérifier qu'une variable est initialisée, gérer des états, etc. Pour réduire cette complexité résiduelle, nous devons rendre notre code le plus simple possible, c'est-à-dire l'écrire de manière à ce qu'il puisse être découpé en petites parties compréhensibles.

Les fonctions sont justement les plus petites parties, ou modules, qu'il est possible de créer et manipuler en programmation. Savoir utiliser, comprendre et écrire des fonctions est un savoir-faire indispensable à tout programmeur ou programmeuse. Dans ce cours, nous allons apprendre à créer et utiliser des fonctions en PHP en gardant en tête l'idée que les fonctions doivent améliorer la qualité de votre code (et de vos programmes), afin qu'il soit plus facile à lire.

II. Écrire des fonctions en PHP

A. Déclarer, nommer et exécuter ses fonctions

Méthode **Déclarer une fonction**

Une fonction (ou une procédure) est un ensemble d'instructions réalisant une tâche particulière à laquelle nous donnons un nom. Elle doit d'abord être *déclarée* avant d'être utilisée (exécutée). En PHP, une fonction se déclare à l'aide du mot clé **`function`** suivi du nom de la fonction et de ses arguments entre parenthèses. Le corps de la fonction est lui placé entre accolades (`{}`), c'est son **implémentation**. Une fonction retourne toujours une valeur, même si elle est `null` (`null` en PHP indique l'absence de valeur).

```php
<?php

//Déclaration de la fonction sayHi
function sayHi(string $firstName, string $lastName): string
{
  $currentTime = date('H:i');
  return "Bonjour " . $firstName . " " . $lastName . " ! Il est " . $currentTime ;
}

//Appel de la fonction sayHi (le corps de la fonction sayHi est éxecuté)

$message = sayHi('John', 'Doe');
echo $message;
```

Analysons un peu cette déclaration : `sayHi` prend en argument deux variables, `$firstName` et `$lastName` de type `string`, et retourne un résultat de type `string`, indiqué par le label `string` placé après la parenthèse fermante.

La fonction `sayHi` appelle la fonction date fournie par PHP pour récupérer l'heure courante. La variable `$currentTime` déclarée dans le corps de la fonction est locale à la fonction, elle n'existera plus lorsque la fonction aura retournée. La fonction fabrique une chaîne de caractères en concaténant (.) le prénom, le nom et l'heure. Nous appelons `sayHi` avec l'instruction `sayHi()`, le nom de la fonction suivi de parenthèses.

La fonction retourne le résultat avec l'instruction `return`, que l'on stocke dans la variable `$message`. Nous imprimons ensuite sur la sortie standard avec le construct `echo`. Les constructs, comme `echo`, sont des instructions spéciales en PHP. Ils suivent d'autres règles de syntaxe, et ne sont pas considérés comme des fonctions.

Le bénéfice immédiat d'une fonction est de pouvoir exécuter un ensemble d'instructions sans dupliquer de code. Le deuxième bénéfice est que l'intention de mon code est à présent plus claire. Une fonction est une abstraction autour d'un ensemble d'instructions. C'est un aspect essentiel de la programmation.

En PHP, les fonctions sont définies par défaut dans l'espace de nom global, c'est-à-dire qu'elles sont accessibles depuis n'importe quel endroit dans le script courant, tout comme vos variables.

Dans votre code source, les fonctions n'ont pas besoin d'être déclarées avant d'être utilisées. Par exemple, le code suivant est équivalent au précédent et fonctionnera normalement :

```php
<?php
//Je peux appeler la fonction 'avant' sa déclaration
$message = sayHi('John', 'Doe');
echo $message;

//Déclaration de la fonction sayHi
function sayHi(string $firstName, string $lastName): string
{
  $currentTime = date('H:i');
  return "Bonjour " . $firstName . " " . $lastName . " ! Il est " . $currentTime ;
}
```

Ceci est possible, car PHP est un langage interprété : le contenu du code source est parsé (analysé par le moteur PHP) à la volée avant d'être exécuté. Durant cette phase de parsing, toutes les fonctions déclarées dans l'espace de nom global sont enregistrées et rendues accessibles pour la phase d'exécution. Cette particularité de PHP est importante à connaître, même de manière simplifiée.

Fonctions conditionnelles et nature interprétée du langage PHP

Le fonctionnement de PHP permet d'expliquer le comportement des fonctions conditionnelles, les fonctions déclarées sous certaines conditions. Si une fonction est déclarée de manière conditionnelle, c'est-à-dire uniquement lorsqu'une expression est évaluée à vrai, alors vous ne pouvez pas l'appeler avant sa déclaration. Prenons l'exemple ci-dessous :

```php
<?php
// isFunctionDefined a 1 chance sur 2 d'être vrai, car
//rand(0,1) génère de manière aléatoire les valeurs 0 et 1
$isFunctionDefined = rand(0,1);

if ($isFunctionDefined) {
  //Déclaration conditionnelle de la fonction sayHi
  function sayHi()
  {
    echo "sayHi: Bonjour !";
  }
}
```

```
13
14 //Appel de la fonction sayHi génère une Fatal error en moyenne 1 fois sur 2, //lorsque la
   fonction n'est pas déclarée
15
16 sayHi();
```

Ce code s'exécute parfois de manière normale, et parfois il génère une erreur, car la fonction `sayHi` n'est déclarée que lorsque `$isFunctionDefined` est évaluée à vrai, lorsque le générateur aléatoire `rand` lui affecte la valeur 1 (1 chance sur 2 en moyenne).

Si l'on se rappelle que PHP est un langage interprété, ce résultat fait sens : la déclaration de la fonction `sayHi` a lieu dans un bloc qui ne sera pas nécessairement exécuté. Cela dépendra de la valeur prise par `$defineFunction` à l'exécution. PHP ne peut donc pas savoir à l'avance si la fonction `sayHi` sera déclarée. Aussi, celle-ci n'est pas enregistrée lors de la phase de parsing.

Conseil **Déclarer des fonctions dans un espace de noms**

En pratique, il est très rare de déclarer des fonctions de manière conditionnelle. Ne le faites pas, sauf si vous avez de très bonnes raisons de le faire. Déclarez toujours vos fonctions dans l'espace global ou un espace de noms. Ce cas est mentionné, car il illustre bien la nature interprétée du langage PHP.

Fondamental **(Bien) nommer ses fonctions**

En PHP, les noms de fonction ne sont pas sensibles à la casse. Aussi les noms `SAYHI` et `sayHi` sont deux noms de fonctions identiques pour PHP. Les noms de fonction suivent les mêmes conventions que tous les autres labels en PHP (nom de variable, constante, etc.) : ils doivent commencer avec un caractère ou un underscore (_), suivi de n'importe quel caractère alphanumérique ou underscore.

Par exemple :

- **_foo123** est un nom de fonction valide
- __ est un nom (étrange) de fonction valide
- **fooBarBaz** est un nom de fonction valide
- **1foo** n'est pas un nom de fonction valide
- **foo-bar** n'est pas un nom de fonction valide

Complément **Les constructs PHP**

Pour information, voici la liste complète des constructs PHP (les noms sont réservés par le langage) : `assert`, `echo`, `print`, `exit`, `die`, `return`, `include`, `include_once`, `require`, `require_once`, `eval`, `empty`, `isset`, `unset` et `list`.

Rappel **Nommage et formatage des fonctions, les règles à suivre**

PHP impose quelques contraintes strictes sur la formation de nom de fonction et les caractères utilisables. Cela dit, vous jouissez encore d'une trop grande liberté pour nommer vos fonctions et formater votre code.

Afin d'améliorer la lisibilité de votre code, et votre capacité à lire celui des autres, le PHP-FIG[1], un groupe de travail de la communauté PHP, a établi des standards que l'on nomme PSR[2] (*PHP Standard Recommandation*) qu'il est bon de lire et de respecter. Les PSR-1[3] et PSR-12[4] définissent des conventions de nommage et de formatage. Voici quelques conventions à suivre :

- Les noms de fonctions doivent être écrits en **camelCase**. Par exemple, fooBar est un nom valide, FooBar ou foo_bar ne le sont pas.
- Les noms de fonction ne doivent pas être suivis d'un espace, les accolades ouvrantes et fermantes délimitant le corps de la fonction doivent être sur leurs propres lignes. Il ne doit pas y avoir non plus d'espace avant la parenthèse ouvrante.
- L'instruction de retour doit être placée sur sa propre ligne.

Par exemple, les déclarations suivantes ne respectent pas les PSR :

```
1 function foo(): string{
2   return "Bonjour !";
3 }
4
5 function bar(): string { return "Bonjour !";}
```

La déclaration suivante en revanche respecte les PSR.

```
1 function fooBar(): string
2 {
3   return "Bonjour !";
4 }
```

Vous pouvez appliquer ces règles de manière automatique lors du formatage de votre code source dans votre éditeur de texte favori.

Si vous écrivez ou distribuez du code PHP, il est important de partager des conventions communes avec les autres membres de la communauté. Il est également recommandé d'utiliser un verbe à l'infinitif pour nommer une fonction. Une fonction agit : elle manipule des données. Un verbe est donc un bon nom de fonction. Par exemple search, remove ou createNewGame sont de bons candidats pour des noms de fonction.

Il est aussi important de bien nommer les arguments pour que l'utilisateur de votre fonction comprenne facilement à quoi ils correspondent. N'utilisez jamais de noms à un caractère comme $a ou $i pour des arguments.

B. Signature d'une fonction : arguments, retour et indications de type

Une fonction accepte en entrée des arguments et retourne toujours une valeur avec le mot clé `return`.

1 https://www.php-fig.org/personnel/
2 https://www.php-fig.org/psr/
3 https://www.php-fig.org/psr/psr-1/
4 https://www.php-fig.org/psr/psr-12/

Méthode **Signature d'une fonction**

Les arguments sont passés à la fonction dans une liste d'arguments entre parenthèses, où chaque argument est séparé par une virgule. Les arguments sont ensuite évalués de la gauche vers la droite. Reprenons notre fonction `sayHi` :

```php
<?php
//Déclaration de la fonction sayHi
function sayHi(string $firstName = 'John', string $lastName = 'Doe'): string
{
  $currentTime = date('H:i');
  return "Bonjour " . $firstName . " " . $lastName . " ! Il est " . $currentTime ;
}
```

La fonction `sayHi` prend deux arguments : `$firstName` et `$lastName`. Ces deux variables sont mises à disposition de la fonction au moment de son exécution. Nous avons indiqué qu'elles étaient de type string et qu'elles avaient une valeur par défaut, « *John* » et « *Doe* » respectivement.

Enfin, nous avons indiqué le type de retour attendu avec le label `string`, qui signifie « *retourne une valeur de type string* ». L'ensemble nom de fonction, arguments et leurs types et type de retour constitue ce que l'on appelle la **signature** de la fonction, ou prototype de la fonction.

La signature d'une fonction est toujours donnée dans la documentation officielle. C'est sa carte d'identité. C'est un concept important, car la signature de la fonction est fondamentalement tout ce que vous avez à savoir sur la fonction pour pouvoir l'utiliser.

Retour d'une fonction

Une fonction retourne toujours une valeur. Si elle ne retourne rien, elle retourne `null`. Le retour de la fonction est indiqué avec le mot clé `return`. L'instruction de retour n'est pas obligatoire. Si elle est omise, la fonction retourne `null` à la fin de l'exécution du corps de la fonction.

L'instruction `return` met fin à l'exécution du corps de la fonction, toute instruction placée derrière ne sera jamais exécutée. Pour récupérer le résultat d'une fonction, il suffit de déclarer une variable et de l'assigner à l'appel de la fonction.

```php
<?php
//Le résultat de foo est stocké dans la variable $result
$result = foo(1, 2, 3);
```

Le type de retour est indiqué après la liste des arguments avec deux-points suivis du type (`: <type>`). Par exemple `foo(): string` indique à l'utilisateur que la fonction retourne une valeur de type `string`, `bar(): void` indique à l'utilisateur que la fonction ne va rien retourner d'intéressant (retourne `null`).

Arguments d'une fonction

Un argument à qui nous donnons une valeur par défaut est un argument dit facultatif. En effet, étant donné qu'il possède une valeur par défaut, vous n'êtes pas tenu d'en fournir une lorsque vous appelez la fonction.

```php
//Appel de la fonction sayHi avec une valeur passée à l'argument $firstName
sayHi('Eve')

//Appel de la fonction sayHi avec la valeur de l'argument $firstName par défaut ('John')
sayHi();
```

La valeur par défaut d'un argument doit être une constante (sa valeur ne doit pas être définie au moment de l'exécution).

Complément **Nombre fini ou indéterminé d'arguments**

Le nombre d'arguments pris par une fonction est ce qu'on appelle son **arité**. Par exemple, la fonction `foo($a, $b, $c ='bar')` a une arité de 3. Une fonction peut avoir un nombre indéfini de paramètres : nous parlons alors de fonction variadique. En PHP, pour déclarer une fonction variadique vous pouvez utiliser le symbole « `...` » devant le nom de l'argument. PHP rangera toutes les valeurs passées à la fonction dans un tableau accessible via le nom de l'argument.

Par exemple :

```php
<?php
function foo(int ...$args): void
{
  //print_r permet d'afficher le contenu d'un tableau sur la sortie standard
  print_r($args);
}
foo(1,2,3,4,5);
```

Vous pouvez aussi vous servir de cet opérateur variadique pour « *déballer* » des données fournies dans un tableau.

Par exemple, imaginons que nous ayons une fonction `multiply` qui calcule le produit de deux nombres fournis en arguments. Nous pouvons donc fournir deux valeurs, séparées par une virgule. Nous pouvons aussi fournir un tableau en argument et « *déballer* » les valeurs une par une dans chaque argument défini par la signature en plaçant l'opérateur « `...` » devant le tableau.

```php
<?php
function multiply(int $x, int $y): int
{
  return $x  * $y;
}

//Affiche 2
echo multiply(1, 2);

//Affiche 2 aussi: le tableau est "déballé" en deux valeurs, une pour chaque chaque argument $x et $y
echo multiply(...[1, 2]);

//Et dans ce cas ? Essayer, cela affiche toujours 2 ! Les valeurs suivantes, faute de paramètres correspondants, sont simplement ignorées !
echo multiply(...[1, 2, 3]);
```

C. Passage des arguments par valeur

Attention **PHP utilise par défaut le passage par valeur**

Il existe plusieurs manières de fournir ou passer des arguments à une fonction. Le passage par valeur (ou par copie) et le passage par référence sont les plus connus. PHP utilise par défaut le passage par valeur.

Passage par valeur

Le passage par valeur consiste à fournir à la fonction une copie de la valeur contenue dans votre variable. Ainsi, si vous changez la valeur dans la fonction, cela n'affectera pas le contenu de la variable initiale.

```php
<?php
function sayHi(string $firstName = 'John')
{
  //On modifie $firstName pour la mettre en majuscules avec strtoupper
  $firstName = strtoupper($firstName);
  echo 'Bonjour ' . $firstName . ' ! ' . PHP_EOL;
```

```php
}

$someone = 'Eve';
//Passage par valeur
sayHi($someone);
//La valeur de $someone n'as pas été modifiée
echo $someone;
```

Passage par référence

En PHP, une référence est un moyen d'accéder à une même variable avec un autre nom. Passer une valeur par référence signifie passer non pas une copie de la valeur contenue dans la variable, mais la variable elle-même. Vous pouvez donc modifier sa valeur directement, sur place.

Le passage par référence s'effectue en faisant précéder l'argument par le symbole de l'esperluette (`&`).

```php
//$array est passé par référence à la fonction
function emptyArray(array &$array)
{
  $array = array();
  On retourne une copie de $array
  return $array;
}
$foo = array(1,2,3);

//$foo est passé par référence
$empty = emptyArray($foo);

print_r($foo); //$foo a été modifié, il est vide
print_r($empty); //$empty est une copie de $foo, un tableau vide aussi
```

Dans cet exemple, nous passons le tableau `$foo` par référence. Dans la fonction `emptyArray`, `$array` n'est pas une copie de `$foo`, mais un autre nom pour `$foo`. Vider `$array` est donc équivalent à vider `$foo`. Ensuite, la fonction retourne une copie (par défaut PHP retourne *par valeur*) du tableau vide, qui est stockée dans `$empty`.

Conseil **Passage par valeur ou passage par référence : lequel choisir ?**

Préférez toujours le passage par valeur, et n'utilisez le passage par référence que si cela se justifie. Pourquoi ? Les références ajoutent de la complexité à votre code et de potentiels bugs. Le passage par valeur est plus simple à utiliser, à comprendre et donc à maintenir.

Les arguments nommés

PHP 8.0.0 introduit les **arguments nommés**. Cette fonctionnalité permet de passer des arguments à une fonction en s'appuyant sur le nom des paramètres et non plus sur leurs positions. Prenons par exemple la fonction [1]`array_fill`[2], dont la signature, fournie sur la documentation officielle, est la suivante :

```php
array_fill(int $start_index, int $count, mixed $value): array
```

Essayez de deviner ce que fait cette fonction juste à partir de son nom et de sa signature. Comme vous l'avez peut-être deviné, cette fonction initialise un tableau d'une certaine taille `$count`, dont chaque élément sera initialisé avec la valeur `$value`. Le premier argument `$start_index` définit à quel index on souhaite démarrer le tableau. Avant l'apparition des arguments nommés, il fallait se souvenir de l'ordre des arguments pour appeler cette fonction.

1 https://www.php.net/manual/fr/function.array-fill.php
2 https://www.php.net/manual/fr/function.array-fill.php

```php
//$foo contient un tableau commençant à l'index 0, contenant 100 éléments initialisés à la valeur 'Bonjour !'
$foo = array_fill(0, 100, 'Bonjour !');
```

Avec les arguments nommés, il est possible de changer l'ordre des arguments, et également d'améliorer la lisibilité de l'appel (la personne qui relit le code n'a pas besoin d'aller voir la documentation pour savoir à quels arguments correspond chaque valeur).

```php
$foo = array_fill(value: 'Bonjour !', count: 100, start_index: 0)
```

Plus une fonction accepte d'arguments, plus il est difficile de se souvenir de l'ordre dans lequel fournir leurs valeurs. Cela crée une complexité accidentelle, et nous demande des efforts mentaux dont on se passerait volontiers.

Remarque **Typage des arguments : le type hinting de PHP**

PHP est un langage typé dynamiquement, contrairement à Java ou au langage C qui sont des langages statiques.

Cela signifie qu'il n'y a pas de vérification des types de vos données à la compilation, la vérification se fait au moment de l'exécution (lorsqu'il est déjà trop tard s'il y a une erreur !). Si votre fonction attend un tableau en argument et reçoit un entier à la place, toutes sortes de comportements indéfinis peuvent se produire, ce qui amène généralement à des bugs ou à un crash de votre programme.

PHP offre un mécanisme pour vous aider à gérer les types de manière dynamique : le type hinting, littéralement donner un indice sur le type attendu. Vous pouvez préciser le type attendu lorsque vous déclarez vos arguments de fonction, ainsi que le type de retour. Par exemple, la signature de la fonction suivante :

```php
strlen(string $string): int
```

Indique (*hint*, donne un indice) une valeur de type `string` pour l'argument `$string`, et qu'elle retourne une valeur de type `int`. Si vous passez un tableau en argument à la fonction `strlen`, votre script sera quand même exécuté, mais vous récupérerez une `Fatal Error: TypeError`, et votre programme va s'arrêter. Cette erreur est levée grâce au type hinting avant que la fonction `strlen` ne soit exécutée. Sans le type hinting, votre programme aurait essayé d'exécuter la fonction `strlen` jusqu'à rencontrer un problème peut-être plus grave encore.

Le type hinting est donc un mécanisme à la fois de documentation de votre code (la signature est enrichie) et de protection à l'exécution. Utilisez toujours le type hinting.

Enfin, la déclaration des types scalaires (`string, int, float, boolean`) est par défaut en mode coercitif. Cela signifie que PHP va silencieusement tenter de transtyper (*cast* en anglais) une valeur s'il peut, c'est-à-dire migrer la donnée d'un type de données vers le type demandé. Prenons un exemple.

```php
<?php

function sayHi(string $firstName): void
{
  //var_dump affiche le type et la valeur de la variable sur la sortie standard
  var_dump($firstName);
}

sayHi(42); // Affiche string(2) "42"
```

La fonction `sayHi` accepte en argument une donnée de type `string`. Lors de l'appel, nous lui envoyons un entier à la place. Comme par défaut nous sommes en mode coercitif, PHP va silencieusement transformer 42 en une chaîne de caractères « *42* ». Ce transtypage silencieux peut avoir des effets inattendus et provoquer des bugs difficiles à identifier. C'est pour cela qu'il est préférable de toujours travailler dans le mode strict. Dans ce mode, PHP ne va plus faire de transtypage.

Pour passer en mode strict, il suffit d'ajouter l'instruction `declare(strict_types=1);` au début du fichier. Si l'on reprend l'exemple précédent en mode strict, l'appel à `sayHi(42)` lèvera une erreur `TypeError`.

```php
<?php
declare(strict_types=1);

function sayHi(string $firstName)
{
  var_dump($firstName);
}

sayHi(42); // PHP Fatal error: Uncaught TypeError: sayHi(): Argument #1 ($firstName) must be of type string, int given
```

Conseil **Documenter sa fonction avec un DocBlock**

Il est recommandé d'ajouter un commentaire au-dessus de la déclaration de la fonction sous forme de commentaire DocBlock[1]. Le commentaire doit être concis et indiquer sans rentrer dans les détails à quoi sert la fonction (si nous avons besoin d'en savoir plus, il suffit de lire le code source directement !).

Un DocBlock va permettre à votre IDE favori, ainsi qu'à d'autres outils comme PhpDocumentor[2], de fabriquer automatiquement de la documentation pour votre code source. Il s'écrit sous la forme d'un commentaire PHP. Il commence par **/**** (ouverture) et se termine par ***/** (fermeture). Chaque ligne, entre l'ouverture et la fermeture, doit commencer par un astérisque (*).

```php
<?php
/**
* Ceci est un DockBlock
*/
function foo()
{
}
```

À l'intérieur de commentaire spécial, vous pouvez documenter votre fonction, ses arguments et son type de retour. Vous pouvez utiliser des labels spéciaux appelés *tags,* commençant par un caractère arobase (**@**). Vous pouvez documenter par exemple :

- Un argument avec la chaîne `@param <type de l'argument> <nomArgument> <Un commentaire>`,
- Le type de retour avec `@return <type>`,
- Un lien vers une ressource sur le web avec `@link <URL>`.

```php
<?php

/**
*Retourne une chaîne de caractères qui salue une personne et lui indique l'heure *courante
*
*<Une description éventuellement plus détaillée de ce que la fonction fait, sur *plusieurs lignes>
*
*@param string $firstName Le prénom de la personne à saluer
* @param string $lastName Le nom de famille de la personne à saluer
* @link https://www.php.net/manual/fr/function.date.php
*
* @return string
*/
function sayHi(string $firstName, string $lastName): string
{
  $currentTime = date('H:i');
```

1 https://docs.phpdoc.org/guide/getting-started/what-is-a-docblock.html
2 https://docs.phpdoc.org/

```php
    return "Bonjour " . $firstName . " " . $lastName . " ! Il est " . $currentTime ;
}
```

D. Exercice : Quiz

[solution n°1 p.21]

Question 1

Parmi les noms de fonctions suivantes, lequel est valide en PHP ?

- `sendRequestToClient`
- `echo`
- `findBy@id`

Question 2

Qu'affiche l'exécution du code suivant ?

```php
<?php

function foo()
{
  function bar()
  {
    function baz()
    {
      echo 'baz';
    }

    echo 'bar';
  }

  echo 'foo' ;
}

foo();
baz();
```

- `fooPHP Fatal error: Uncaught Error: Call to undefined function baz()`
- `foobarbaz`
- `foobaz`

Question 3

Qu'affiche l'exécution du code suivant ?

```php
<?php

function foo(int $a, int $b, int $c):int
{
  return $a + $b + $c;
}

echo foo(1, 2, '3') . PHP_EOL;
echo foo(...[1, 2, 3, 4, 5]). PHP_EOL;
echo foo(1, 2). PHP_EOL;
```

Note : `PHP_EOL` est une constante PHP pour le caractère retour à la ligne (*PHP End Of Line*), afin d'afficher le résultat de chaque appel à `echo` sur une ligne différente.

- 6
  6
  Fatal Error (crash du programme)
- 6
  Fatal Error (crash du programme)
- 6
  15
  3

Question 4

Qu'affiche l'exécution du code suivant ?

```php
<?php

function bar(array $array)
{
  $foo[0] = 'Bonjour !';
}

$foo = array(
  1,
  2,
  3
);

bar($foo);

//On affiche le contenu de fou
print_r($foo);
```

- `'Bonjour !', 2, 3`
- 1, 2, 3
- Rien (tableau vide)

Question 5

Qu'affiche l'exécution du code suivant ?

```php
<?php
declare(strict_types=1);

function sayHi(string $firstName): void
{
  echo 'Bonjour, ' . $firstName . ' !' . PHP_EOL;
}

sayHi('John');
sayHi(12345);
```

- `'Bonjour, John !'`
  `'Bonjour, 12345 !'`

- `PHP Fatal error: Uncaught TypeError`
- `'Bonjour, John !'`
  `PHP Fatal error: Uncaught TypeError`

III. Utiliser les fonctions pour améliorer son code

A. Utiliser les fonctions pour améliorer son code

Avantages liés au bon usage des fonctions

Créer des fonctions à bon escient vous permet de :

- Réutiliser et partager du code.
- Masquer la complexité.
- Fabriquer des abstractions. C'est peut-être la propriété la plus importante. Une fonction est simplement un nom donné à un morceau de code exécutable. En nommant ce code, vous exprimez une **intention** de design.
- Améliorer la lisibilité de votre code. « *Any fool can write code that a computer can understand. Good programmers write code that humans can understand* » - Martin Fowler ou traduit en français : « *N'importe quel idiot peut écrire du code qu'une machine peut comprendre. Un bon programmeur ou programmeuse écrit du code que d'autres humains peuvent comprendre* ».

Fondamental **Une fonction est une donnée comme une autre ! Introduction aux fonctions variables**

Dans l'introduction, nous avons dit que tout programme informatique, aussi complexe soit-il, était composé de deux parties : des fonctions et des données. Il se trouve que rien ne distingue les deux de manière fondamentale. Vous pouvez ainsi stocker une fonction dans une variable. Si par exemple, vous devez élever au carré des valeurs plusieurs fois dans votre programme, vous voulez peut-être déclarer cette fonction afin de la réutiliser.

```php
<?php
//Une fonction qui prend une fonction en argument
function sum(array $array, callable $operation){

  $sum = 0;
  foreach ($array as $item) {
    //Application de la fonction $operation à chaque item
    $sum += $operation($item);
  }

  return $sum;
}
//On stocke notre fonction dans une variable
$square = function($item)
{
  return $item  * $item;
};

//On passe la variable qui contient notre fonction à sum. Affiche 14
echo sum(array(1,2,3), $square);

//On peut se resservir de $square pour faire un autre calcul
echo sum(array(4,5,6), $square);
```

Remarque **Toutes les fonctions ne sont pas des valeurs**

Pour qu'une fonction puisse être considérée comme une valeur, il faut qu'elle soit pure : sa valeur de retour doit être la même pour les mêmes arguments (pas d'état interne), et son évaluation ne doit pas causer d'effets de bords (mise à jour d'état global, manipulation de flux d'entrée sortie, par exemple écrire sur la sortie standard, etc.).

Résoudre des problèmes avec les fonctions récursives

Nous finirons ce module en discutant d'un type de fonction très puissant : les fonctions récursives. Les fonctions récursives sont particulièrement adaptées pour résoudre des problèmes impliquant des structures récursives (par exemple un graphe, un réseau, un système de fichiers, etc.).

Elles permettent de trouver une solution généralement beaucoup plus simple, lisible et élégante que si vous deviez l'écrire de manière itérative ou procédurale, c'est-à-dire avec des variables d'état et des règles pour les mettre à jour. Illustrons la différence entre les deux approches sur un exemple classique : le calcul de la factorielle d'un nombre.

Nous rappelons que le produit factoriel est une fonction qui, à chaque nombre entier positif, associe le produit dont les facteurs sont les nombres entiers entre un et ce nombre.

Le produit factoriel d'un entier positif N se note `N!`, et se calcule comme `N x (N-1) x (N-2) x ...x 1`. Par exemple, `5!` est égal à `5 x 4 x 3 x 2 x 1`, soit 120. Par définition, `0!= 1`.

Écrivons d'abord cette fonction de manière itérative, sûrement la manière qui vous est la plus familière. Nous devons décrire le processus. Voici à quoi ressemblerait la fonction écrite de manière itérative :

```php
<?php
/**
* Calcul du produit factoriel d'un nombre de manière itérative
* @param int $n Un entier positif
* @return int Le produit factoriel
*/
function factorielIterative(int $n): int
{
  //Par définition, 0! = 1
  if ($n === 0) {
    return 1;
  }

  $result = $n;

  for ($i = $n - 1; $i !== 0; $i = $i - 1) {
    $result *= $i;
  }

  return $result;
}

//Affiche 1
echo factorielIterative(0);

//Affiche 120
echo factorielIterative(5);
```

Cette implémentation du produit factoriel est correcte. Une autre manière d'approcher le problème est d'écrire une fonction récursive. Une fonction récursive est une fonction qui contient au moins un appel à elle-même. Écrivons la fonction `factorielRecursive(int $n): int` :

```php
<?php
/**
* Calcul du produit factoriel d'un nombre de manière récursive
* @param int $n Un entier positif
* @return int Le produit factoriel
*/
function factorielRecursive(int $n): int
{
  //Par définition, 0! = 1
  if ($n === 0) {
    return 1;
  }
  return $n * factorielRecursive($n - 1);
}
//Tests
echo factorielRecursive(0);
echo factorielRecursive(1);
echo factorielRecursive(5);
```

Voici donc l'implémentation du même calcul, mais de manière récursive. Apprécier l'élégance, la concision et la netteté de cette fonction, comparée à son homologue itérative.

Que se passe-t-il derrière l'appel `factorielRecursive(5)` ? La fonction est exécutée avec `$n=5`, la fonction retourne donc la valeur `5 * factorielRecursive(4)`. La fonction doit évaluer l'expression `factorielRecursive(4)` avant de *retourner*, et donc, exécuter la fonction avec `$n=4`. De la même manière, `factorielRecursive(4)` retourne `4 * factorielRecursive(3)`. De la même manière, la fonction doit évaluer l'expression `factorielRecursive(3)` et donc exécute la fonction avec `$n=3`. Et ainsi de suite. Quand `$n` est égal à 1, `factorielRecursive(1)` est évalué à 1. Notre appel récursif rencontre son point d'arrêt avec l'instruction `if ($n === 0)`. Ce qui permet d'évaluer `factorielRecursive(2)` qui est égal à `2 * factorielRecursive(1)` soit 2 * 1, soit 2. Puis `factorielRecursive(3) = 3 * factorielRecursive(2)` peut être évalué à 6, et ainsi de suite jusqu'à remonter à l'évaluation de `factorielRecursive(5) = 5 * factorielRecursive(4)`.

Nous remarquons que la solution récursive ne contient aucune variable d'état : il n'y a aucune variable locale mise à jour. Les variables d'états et la boucle de la solution itérative ont été remplacées par des appels récursifs.

Attention **Condition d'arrêt**

Toutes les fonctions récursives ont une condition d'arrêt. Elle a un rôle équivalent à celui joué par les limites de notre boucle `for` dans la solution itérative : elle permet de mettre un terme aux appels récursifs et de retourner un résultat utile.

Méthode **Construire une fonction récursive**

Pour fabriquer une fonction récursive, il faut commencer par trouver la condition d'arrêt. Ici, c'est quand `$n` est égal à 0.

Ensuite, il faut faire attention à ce que cette condition soit rencontrée à un moment donné sous peine de tomber dans une boucle d'appels infinie ! Pour cela, chaque appel enfant doit travailler sur un plus petit problème que l'appel du parent.

Par exemple ici, `factorielRecursive(5)` travaille sur `5!`. Elle le réalise en évaluant `5 * factorielRecursive(4)`. L'appel enfant travaille donc sur un problème réduit (4) par rapport à son parent (5). Et ainsi de suite, jusqu'à arriver à la condition d'arrêt.

Ce qu'il faut retenir c'est que :

- Une fonction récursive est une fonction qui s'appelle elle-même.
- Une solution récursive fonctionne sur la résolution d'un problème en résolvant successivement des problèmes plus petits jusqu'à arriver à une solution triviale, connue ou définie (comme `0!` dans notre exemple).
- En général, un problème peut être résolu de manière itérative ou récursive. Chaque solution vient avec ses avantages et ses inconvénients.

B. Exercice : Quiz

[solution n°2 p.23]

Question 1

Complétez la proposition suivante : « *Lorsqu'on dit qu'une fonction ne doit faire qu'une seule chose, cela signifie qu'une fonction* » :

- ○ Ne doit contenir qu'une seule instruction
- ○ Doit réaliser une tâche précise (ce qui peut demander plusieurs instructions)
- ○ Ne doit pas appeler une autre fonction

Question 2

Complétez la proposition suivante : « *La signature, ou interface, d'une fonction est définie par son* » :

- ○ Nombre d'arguments uniquement
- ○ Implémentation
- ○ Nom, ses arguments et leurs types, son type de retour

Question 3

Complétez la proposition suivante **:** « *Une fonction devrait* » :

- ○ Avoir une interface (une signature) la plus simple possible
- ○ Réaliser le plus de fonctionnalités possible
- ○ Avoir le plus d'arguments possible

Question 4

Complétez la proposition suivante : « *Lorsqu'on dit qu'en PHP, une fonction est un citoyen de première classe, cela signifie que les fonctions* » :

- ○ Peuvent être passées en argument à d'autres fonctions
- ○ Ne peuvent pas être stockées dans des variables
- ○ Sont des variables comme les autres

Question 5

Complétez la proposition suivante : « *Une fonction récursive* » :

- ○ Est une fonction qui s'appelle elle-même
- ○ Ne peut résoudre que des problèmes liés à des structures de données récursives comme les arbres
- ○ Résout un problème complexe en utilisant des boucles et des variables d'état

IV. Essentiel

Nous avons vu qu'un programme était composé de fonctions et de données. Les fonctions sont les entités qui manipulent les données. Une fonction permet de créer une abstraction autour d'une suite complexe d'instructions. C'est un composant essentiel de tout code source auquel il faut apporter le plus grand soin.

Toute fonction prend en entrée des arguments que l'on peut type-hinter (indiquer le type attendu) et retourne toujours un résultat, avec le mot clef `return`. Pour déclarer une fonction en PHP, nous utilisons le mot clef `function`. Les fonctions devraient être nommées en respectant les contraintes du langage, les conventions du PSR-12 (en **camelCase**). Leur nom devrait également contenir un verbe pour indiquer l'action qu'elle réalise, par exemple la fonction `sendEmail`.

Nous avons vu que le nom, les arguments (et leurs types) et le type de retour forment la **signature,** ou interface, de la fonction. Cette interface devrait être documentée par un **DocBlock**, afin de constituer une documentation essentielle de la fonction dans le but de transmettre son intention et de l'utiliser facilement.

Nous avons également vu qu'en PHP, les fonctions sont des citoyens de première classe : elles peuvent être passées en argument à d'autres fonctions comme n'importe quelle valeur, être stockées dans des variables ou être déclarées de manière anonyme, à la volée.

Enfin, nous avons découvert les fonctions récursives : une fonction qui s'appelle elle-même, jusqu'à arriver à une condition d'arrêt. Ces fonctions ont la particularité de résoudre un problème en résolvant successivement des problèmes plus petits. La résolution de problèmes par récursion est une alternative à la méthode itérative, et dans certains cas s'avère beaucoup plus efficace.

V. Auto-évaluation

A. Exercice

Vous êtes développeur et devez écrire un programme PHP pour l'entreprise Strategy, en vue d'améliorer un processus de publication vers le web. L'entreprise Strategy est dans le domaine de la veille stratégique. Elle recense et traite quotidiennement des centaines de publications dans différents domaines : académique, juridique, commercial, etc. Depuis peu, elle récupère des résumés (*abstracts*) de publications au format Markdown[1]. Elle aimerait pouvoir les transformer automatiquement au format HTML pour les publier sur un site web. Voici un exemple représentatif de résumé d'article que votre programme doit pouvoir traiter :

Git is a fast, **scalable**, distributed revision control system with an unusually rich command set that provides both high-level operations and full access to internals. Git is an **Open Source project** covered by the **GNU General Public* License* version 2 (some parts of it are under different licenses, compatible with the GPLv2). It was originally written by **Linus Torvalds** with help of a group of *hackers* around the net.

Pour simplifier le problème, nous supposons que :

- Les astérisques sont uniquement présents pour former des balises et ne sont pas utilisés dans le texte.
- Tous les abstracts sont bien formés : chaque balise ouvrante possède une balise fermante. Par exemple, nous n'avons pas à traiter un cas comme « *Git is a fast, *scalable, distributed system* » où la balise fermante a été oubliée.

1 https://fr.wikipedia.org/wiki/Markdown

Question 1

[solution n°3 p.24]

Nous souhaitons transformer les balises Markdown indiquant une emphase (*) en balise HTML (). Par exemple, transformer *scalable* en scalable. Écrire une fonction qui prend en entrée le texte au format Markdown et retourne un tableau contenant les tokens. Par exemple, si l'entrée est la chaîne suivante « *Git is an *Open Source project* covered by the *GNU General Public License* version 2* », votre fonction doit renvoyer un tableau contenant « *Open Source project* » et « *GNU General Public License* ». Le nom de votre fonction et de ses arguments doivent être significatifs. Elle doit être documentée par un Docblock. L'usage d'expressions régulières (regex) n'est pas autorisé.

Conseils :

- Stockez le résumé test dans une chaîne de caractères.
- Vous pouvez écrire d'autres fonctions intermédiaires pour écrire les fonctions demandées.
- Vous pouvez vous aider des fonctions PHP strpos(string $haystack, string $needle, int $offset = 0): int | false[1] et pour écrire votre fonction.

Question 2

[solution n°4 p.25]

Écrire une fonction récursive qui compte le nombre de tokens entre balises Markdown d'« *emphase* » (*) dans une chaîne de caractères. Par exemple, la chaîne « *Git is an *Open Source project* covered by the *GNU General Public License* version 2* » en contient 2.

B. Test

Exercice 1 : Quiz

[solution n°5 p.26]

Question 1

Quel symbole permet d'indiquer que la valeur d'un paramètre sera passée par référence lors de la déclaration d'une fonction ?

- ?
- ~
- &

Question 2

Qu'affiche l'exécution du code suivant ?

```php
<?php
declare(strict_types=1);

function add(int $a, int $b)
{
  return $a + $b;
}

echo add('3','4');
```

- 7
- '7'
- Rien, une erreur fatale va être générée

Question 3

1 https://www.php.net/manual/fr/function.strpos.php

Une fonction PHP peut accepter un nombre indéfini d'arguments.

- Vrai
- Faux

Question 4

Les constructs `echo`, `include` et `print` sont des fonctions comme les autres en PHP.

- Vrai
- Faux

Question 5

Parmi les déclarations de fonction suivantes, laquelle est une déclaration valide en PHP ?

-
```
<?php
function(int $a = 0, int $b = 0){
                return $a + $b;
}
```
-
```
<?php
$add = fn(int $a = 0, int $b = 0): int => $a + $b;
```
-
```
<?php
function operation+(int $a = 0, int $b = 0): int
{
                return $a + $b;
}
```

Solutions des exercices

Exercice p. 12 **Solution n°1**

Question 1

Parmi les noms de fonctions suivantes, lequel est valide en PHP ?

- [x] `sendRequestToClient`
- [ ] `echo`
- [ ] `findBy@id`

Un nom de fonction valide en PHP doit commencer par un underscore ou un caractère alphabétique. Il peut ensuite contenir, au choix, un underscore ou n'importe quel caractère alphanumérique ASCII. Le symbole « @ » n'est pas autorisé. « *echo* » n'est pas un nom de fonction valide, car c'est un mot réservé par PHP pour le construct echo.

Question 2

Qu'affiche l'exécution du code suivant ?

```php
<?php

function foo()
{
  function bar()
  {
    function baz()
    {
      echo 'baz';
    }

    echo 'bar';
  }

  echo 'foo' ;
}

foo();
baz();
```

- [x] `fooPHP Fatal error: Uncaught Error: Call to undefined function baz()`
- [ ] `foobarbaz`
- [ ] `foobaz`

La réponse 1 est la bonne. Lorsque l'on appelle `foo`, la fonction `bar` est déclarée puis « *foo* » est écrit sur la sortie standard. Lorsqu'on appelle `baz` à la ligne suivante**,** une erreur fatale est générée, car la fonction `baz` n'est définie nulle part. Pour qu'elle soit déclarée, il aurait fallu appeler `bar` avant.

Question 3

Qu'affiche l'exécution du code suivant ?

```php
<?php

function foo(int $a, int $b, int $c):int
{
  return $a + $b + $c;
}
```

```php

echo foo(1, 2, '3') . PHP_EOL;
echo foo(...[1, 2, 3, 4, 5]). PHP_EOL;
echo foo(1, 2). PHP_EOL;
```

Note : `PHP_EOL` est une constante PHP pour le caractère retour à la ligne (*PHP End Of Line*), afin d'afficher le résultat de chaque appel à `echo` sur une ligne différente.

- [x] 6
  6
  Fatal Error (crash du programme)
- [ ] 6
  Fatal Error (crash du programme)
- [ ] 6
  15
  3

Les deux premiers appels sont corrects :

- `foo(1,2,'3')` : par défaut nous sommes en mode coercitif, donc PHP va transtyper silencieusement la chaîne de caractères '3' en entier 3 et la fonction va s'exécuter et donner le résultat correct.
- `foo(...[1, 2, 3, 4, 5])` : nous faisons appel ici à l'opérateur `...` pour déballer le tableau de valeurs. Les trois premières valeurs vont donc être assignées aux arguments `$a`, `$b` et `$c` respectivement. Les valeurs suivantes sont simplement ignorées.
- Le troisième appel `foo(1, 2)` génère une erreur `Fatale Uncaught ArgumentCountError: Too few arguments to function foo()`. En effet, tous les paramètres sont ici obligatoires, il n'y a aucune valeur par défaut fournie dans la signature. PHP ne peut donc pas exécuter le corps de la fonction et génère une erreur.

Question 4

Qu'affiche l'exécution du code suivant ?

```php
<?php

function bar(array $array)
{
  $foo[0] = 'Bonjour !';
}

$foo = array(
  1,
  2,
  3
);

bar($foo);

//On affiche le contenu de fou
print_r($foo);
```

- [ ] `'Bonjour !', 2, 3`
- [x] 1, 2, 3
- [ ] Rien (tableau vide)

La bonne réponse est la 2. Par défaut, PHP passe des arguments par valeur. La fonction `bar` reçoit dans son argument `$array`, une copie de `$foo`. Le tableau est modifié, mais comme la fonction ne retourne rien, la variable locale `$array` est détruite à la fin de l'exécution de `bar`. La fonction `bar` ne modifie donc pas `$foo` et nous affichons bien la valeur initiale, soit « *1,2,3* ».

Question 5

Qu'affiche l'exécution du code suivant ?

```php
<?php
declare(strict_types=1);

function sayHi(string $firstName): void
{
  echo 'Bonjour, ' . $firstName . ' !' . PHP_EOL;
}

sayHi('John');
sayHi(12345);
```

- `'Bonjour, John !'`
  `'Bonjour, 12345 !'`
- `PHP Fatal error: Uncaught TypeError`
- `'Bonjour, John !'`
  `PHP Fatal error: Uncaught TypeError`

La bonne réponse est la 3, car nous sommes en mode strict. Le premier appel `sayHi('John')` est correct. Le deuxième appel `sayHi(12345)` est incorrect : la valeur passée en argument est de type `int` alors que la fonction attend une valeur de type `string`. L'appel génère donc une erreur fatale de type `TypeError`.

Exercice p. 17 **Solution n°2**

Question 1

Complétez la proposition suivante : « *Lorsqu'on dit qu'une fonction ne doit faire qu'une seule chose, cela signifie qu'une fonction* » :

- Ne doit contenir qu'une seule instruction
- Doit réaliser une tâche précise (ce qui peut demander plusieurs instructions)
- Ne doit pas appeler une autre fonction

Une fonction ne doit pas nécessairement contenir qu'une seule instruction, elle doit réaliser une tâche (par exemple envoyer un e-mail, générer un fichier PDF, réaliser un calcul complexe, etc.) et se spécialiser à la réalisation de celle-ci. Pour cela, elle peut évidemment appeler d'autres fonctions.

Question 2

Complétez la proposition suivante : « *La signature, ou interface, d'une fonction est définie par son* » :

- Nombre d'arguments uniquement
- Implémentation
- Nom, ses arguments et leurs types, son type de retour

L'interface d'une fonction ou sa signature (aussi appelé prototype) est la surface exposée au reste du monde. Elle est censée être la seule chose connue par l'utilisateur du code et par le reste du programme. Elle définit ses entrées (arguments) et sa sortie (valeur retournée) et masque ses détails d'implémentation (son corps).

Question 3

Complétez la proposition suivante **:** « *Une fonction devrait* » :

- (•) Avoir une interface (une signature) la plus simple possible
- ( ) Réaliser le plus de fonctionnalités possible
- ( ) Avoir le plus d'arguments possible

Une fonction devrait avoir une interface simple. Plus son interface est simple, plus il est simple de la comprendre et de l'utiliser. Exposer une interface réduite minimise également les risques d'entrées non prévues par la personne qui a programmé la fonction, pouvant amener à des comportements non désirés (bugs). Une fonction devrait, quitte à être un peu longue, faire une chose, mais la faire bien (gérer tous les cas d'usage, avoir une gestion d'erreurs propres, etc.).

Question 4

Complétez la proposition suivante : « *Lorsqu'on dit qu'en PHP, une fonction est un citoyen de première classe, cela signifie que les fonctions* » :

- (•) Peuvent être passées en argument à d'autres fonctions
- ( ) Ne peuvent pas être stockées dans des variables
- ( ) Sont des variables comme les autres

Une fonction stockée dans une variable n'est pas une variable comme les autres. Elle peut être exécutée en plaçant des parenthèses à la suite du nom, par exemple si la variable `$foo` désigne une fonction, l'instruction `$foo()` exécute la fonction stockée. En PHP, une fonction variable est implémentée par la classe Closure de PHP.

Question 5

Complétez la proposition suivante : « *Une fonction récursive* » :

- (•) Est une fonction qui s'appelle elle-même
- ( ) Ne peut résoudre que des problèmes liés à des structures de données récursives comme les arbres
- ( ) Résout un problème complexe en utilisant des boucles et des variables d'état

Par définition, une fonction récursive est une fonction qui doit s'appeler elle-même au moins une fois. Les fonctions récursives peuvent s'appliquer généralement à la résolution de tout type de problèmes, y compris les problèmes concernant des données non récursives (des listes, un calcul, etc.). Une fonction récursive n'a pas d'état interne et n'utilise pas de boucles, contrairement aux solutions itératives.

p. 19 Solution n°3

```php
/**
* Retourne tous les tokens (chaîne de caractères)
* se trouvant à l'intérieur d'un tag, tags exclus
* @param string $string La chaîne de caractères à analyser
* @param string $openingTag chaîne de caractères ouvrante du tag
* @param string $closingTag chaîne de caractères fermante du tag
* @return array La liste des tokens trouvés
*/
```

```php
function findTokens(string $string, string $openingTag, string $closingTag): array
{
  $openingTagPosition = strpos($string, $openingTag);

  //S'il n'y a pas de balise ouvrante, on retourne une liste vide
  if ($openingTagPosition === false)
    return array();

  $tokens = array();

  //Solution itérative (boucle + variables d'état $tokens et   $openingTagPosition mises à jour)
  while ($openingTagPosition !== false) {

    $closingTagPosition = strpos($string, $closingTag, $openingTagPosition + strlen($openingTag));

    $tokenLength = $closingTagPosition - ($openingTagPosition + strlen($openingTag));

    $token = substr($string, $openingTagPosition + strlen($openingTag), $tokenLength);

    $tokens[] = $token;

    $openingTagPosition = strpos($string, $openingTag, $closingTagPosition + strlen($openingTag));
  }

  return $tokens;
}
//Test
$abstract = 'Git is a fast, *scalable*, distributed revision control system with an unusually rich command set that provides both high-level operations and full access to internals. Git is an *Open Source project* covered by the *GNU General Public License* version 2 (some parts of it are under different licenses, compatible with the GPLv2). It was originally written by *Linus Torvalds* with help of a group of *hackers* around the net.';

print_r(findTokens($abstract, '*', '*'));
```

p. 19 **Solution n°4**

```php
/**
* Retourne le nombre de tokens compris dans les balises ouvrante et fermante
* @param string $openingTag chaîne de caractères ouvrante du tag
* @param string $closingTag chaîne de caractères fermante du ta
* @return int Le nombre de tokens trouvés
*/
function countTokensBetweenTags(string $string, string $openingTag, string $closingTag): int
{

// Condition d'arrêt: longueur de la chaîne à analyser est égale à 0
  if (strlen($string) === 0)
    return 0;

  $openingTagPosition = strpos($string, $openingTag);
  $closingTagPosition = strpos($string, $closingTag, $openingTagPosition + strlen($openingTag));

  //Condition d'arrêt: absence de balise ouvrante ou absence de balise fermante (input mal formée)
  if ($openingTagPosition === false || $closingTagPosition === false)
```

```
19    return 0;
20
21    $offset = $closingTagPosition + strlen($closingTag);
22
23    //On extrait la chaîne restante, située après la dernière balise fermante: problème plus
petit
24    $substr = substr($string, $offset);
25
26    //On appelle la fonction de manière récursive sur le problème plus petit
27    return 1 + countTokensBetweenTags($substr, $openingTag, $closingTag);
28 }
29 echo "Question 2: test" . PHP_EOL;
30 echo countTokensBetweenTags($abstract, '*', '*') . PHP_EOL;
```

Exercice p. 19 Solution n°5

Question 1

Quel symbole permet d'indiquer que la valeur d'un paramètre sera passée par référence lors de la déclaration d'une fonction ?

- ○ ?
- ○ ~
- ◉ &

Le symbole & placé devant un argument (par exemple `sayHi(&$firstName)`) indique que la fonction reçoit une référence vers la variable, et non une copie de la valeur contenue dans la variable. La fonction manipule directement la valeur contenue dans la variable passée en argument. Elle peut donc générer des effets de bord, c'est-à-dire modifier l'état global du programme.

Question 2

Qu'affiche l'exécution du code suivant ?

```
1 <?php
2 declare(strict_types=1);
3
4 function add(int $a, int $b)
5 {
6    return $a + $b;
7 }
8
9 echo add('3','4');
```

- ○ 7
- ○ '7'
- ◉ Rien, une erreur fatale va être générée

Nous avons activé le mode strict avec l'instruction `declare(strict_types=1);`. PHP ne va plus automatiquement essayer de changer le type d'une variable comme il le ferait par défaut.

Comme nous passons une chaîne de caractères et non un entier à la fonction `add`, une `Fatal Error` de type `TypeError` va être renvoyée, et la fonction `add` ne sera pas exécutée.

Question 3

Une fonction PHP peut accepter un nombre indéfini d'arguments.

- [x] Vrai
- [ ] Faux

Avec l'opérateur, il est possible de construire des fonctions variadiques qui prennent un nombre indéfini d'arguments, passés sous forme de tableau à la fonction. Par exemple `add(...$integers): int`.

Question 4

Les constructs `echo`, `include` et `print` sont des fonctions comme les autres en PHP.

- [ ] Vrai
- [x] Faux

Les constructs `echo`, `include` et `print` en PHP ne sont pas des fonctions au sens strict du terme. Ce sont plutôt des instructions spéciales ou des constructions de langage.

Question 5

Parmi les déclarations de fonction suivantes, laquelle est une déclaration valide en PHP ?

- [ ]
```
<?php
function(int $a = 0, int $b = 0){
                return $a + $b;
}
```
- [x]
```
<?php
$add = fn(int $a = 0, int $b = 0): int => $a + $b;
```
- [ ]
```
<?php
function operation+(int $a = 0, int $b = 0): int
{
                return $a + $b;
}
```

Pour la première, il manque le nom de la fonction, la troisième a un nom invalide (le caractère + n'est pas autorisé). La deuxième est correcte : c'est une fonction variable avec la méthode raccourcie de fonction fléchée.