CODE GUIDE

ДЛЯ ЧЕГО, СОБСТВЕННО:

Код, к которому мы будем стремиться даст следующее: - Увеличение скорости разбора кода; - Легкость отлаживания кода; - Легкость внесения изменений; - Возможность переиспользования кода в других проектах;

СТИЛЬ КОДА:

- Открывающиеся **теги**: `<?php ?> <?= ?>`;
- Для оформления **отступов** должны использоваться четыре пробела;
- Контролируйте **ширину строки**: рекомендуемая ширина составляет - до 80 символов. Максимальная до 120.
- Если файл содержит только код PHP, закрывающий тег `?>` в конце файла не ставим.
- Все файлы PHP обязаны заканчиваться одной пустой строкой.**Основные положения**:

```
 # Константы:
   // константы класса объявляются в верхнем регистре с разделителями подчеркивания;
const VERSION = '1.0';

# Массивы:
   // объявление массива делается через квадратные скопки
$files = ['users.csv'];

# Управляющие конструкции:
   // после ключевых слов управляющих структур необходимо ставить один пробел;
   // открывающие фигурные скобки управляющих структур необходимо оставлять на той же стр
if ($condition1) {
   // тело if
} elseif ($condition2) {
   // тело elseif
} else {
   // тело else
}

# switch:
   // имеют следующее размещение круглых и фигурных скобок, а так же пробелов:
switch ($expr) {
   case 0:
     echo 'Первый case, заканчивается на break';
     break;
   case 1:
     echo 'Второй case, с умышленным проваливанием';
     // no break
   case 2:
```

```
  case 3:
  case 4:
    echo 'Третий case, завершается словом return вместо break';
    return;
  default:
    echo 'По умолчанию';
    break;
}

# Циклы:
  // имеют следующее размещение круглых скобок, пробелов и фигурных скобок:
while ($arg3) {
  // тело конструкции
}

do {
  // тело конструкции;
} while ($expr);

for ($i = 0; $i < 10; $i++) {
  // тело конструкции;
}

foreach ($iterable as $key => $value) {
  // тело конструкции;
}

# Функции:
  // аргументы метода со значениями по умолчанию ОБЯЗАНЫ идти в конце списка аргументов.
  // имеют следующее размещение круглых и фигурных скобок, а так же пробелов:
function getSumArgs($arg1, &$arg2, $arg3 = [])
{
  // тело функции
}

# try, catch
  // имеют следующее размещение круглых и фигурных скобок, а так же пробелов:
try {
   // тело try
} catch (FirstExceptionType $e) {
  // тело catch
} catch (OtherExceptionType $e) {
  // catch body
}
```

По всем спорным моментам обращаемся к документации: PSR 1 PSR 2 PSR 3 PSR 4 PSR 6 PSR 7 PSR 11 PSR 12 PSR 13 PSR 14

ПЕРЕМЕННЫЕ

НЕЙМИНГ

- **Нотация**: Верблюжья нотация (CamelCase):

```
$productCount
```

- **Имя переменной** должно быть СУЩЕСТВИТЕЛЬНЫМ:

```
$productRating
$brandLinks
$properties
```

Переменная, в которой содержится количество чего-либо (сущность во множественном числе + count):

```
$usersCount
$productsCount
```

PHPDoc

> При использовании переменных, объявленных в подлючаемых файлах их нужно описать в PHPDoc.

Пример:

```
require "./obSectionVariables.php";
require "./arSectionsVariables.php";
require "./obElementVariables.php";

/**
* @var $obSection # объявлена в obSectionVariables.php
* @var $arSections # объявлена в arSectionsVariables.php
* @var $obElement # объявлена в obElementVariables.php
*/
```

ФУНКЦИИ (И МЕТОДЫ)

НЕЙМИНГ:

- **Имя функции** должно быть ГЛАГОЛОМ

```
getName($user)
setName($user, "Alexandr")
saveProperties($properties)
```

или предикатом:

```
isEmpty($users)
isValid($field)
```

Называть функцию необходимо так, чтобы **не заходя в код функции можно понять, что функция делает**.

СЕМАНТИКА

- Любая функция доложна отвечать за один функционал и только за него.
- Если фунция отвечает сразу за два или более действий, то её функционал нужно разделять.
- Если входящих аргументов > 3, то стоит задуматься, каким образом лучше от них избавится.
- Контролируйте количество строк:
  - для функций не более 20 строк;
  - для метода не более 50 строк;
  - для класса не более 300 строк;
  - для файла не более 1000 строк.
- Функции с побочными эффектами (обращение к БД, чтение/запись в файл, печать на экран, загрузка файла) всегда выносим в отдельную функцию.
- Не используем переменные из глобальной области видимости:

```
function foo()
{
  global $variable; # ЗЛО!
  // или
  $GLOBALS['variable']; # ЗЛО!
}
```

- Запросы в цикле ЗЛО!
- Делаем код максимально плоским (речь про вложенность в управляющих конструкциях и циклах);
- DRY (Don’t repeat yourself) – принцип призывает не повторяться при написании кода. > Все что Вы пишите в проекте, должно быть определено только один раз.
- KISS (keep it simple stupid) - принцип призывает делайте вещи проще. > Порой наиболее правильное решение – это наиболее простая реализация задачи, в которой нет ничего лишнего.
- YAGNI («You aren't gonna need it») - вам это не понадобится» > Все что не предусмотрено техническим заданием проекта, не должно быть в нем.

НАЧИНАЕМ ИСПОЛЬЗОВАТЬ: PHPDoc + Type Hinting

- Для каждой функции:
    - Для **старых версий** PHP:
    - В PHPDoc:
        - указываем кратное описание функции;
        - указываем тип данных для:
        - аргументов функции;
        - возвращаемого значения;
    - Для **PHP 7.2 и выше**:
    - В PHPDoc:
        - указываем кратное описание функции;
        - указываем тип данных для:
            - аргументов функции;
            - возвращаемого значения;
    - Type Hinting (контроль типа):
        - указываем тип данных для:
        - аргументов функции;
        - возвращаемого значения *;
    - При изменении функции обновляем PHPDoc и Type Hinting.

```
 # Образец для старых версий PHP:
/**
 * Возвращает названия брендов из фильтровой части URN:
 *
 * @param string $brandsFilterPath
 * @return array
 */
function getFilterNames($brandsFilterPath)
{
  if (!$brandsFilterPath) {
    return [];
  }

  return explode("-", $brandsFilterPath);
}


# Образец PHP 7.2 и выше:
declare(strict_types=1); // включаем режим строгой типизации

/**
 * Возвращает названия брендов из фильтровой части URN:
 *
 * @param string $brandsFilterPath
 * @return array
 */
function getFilterNames(string $brandsFilterPath): array
{
  if (!$brandsFilterPath) {
      return [];
  }

  return explode("-", $brandsFilterPath);
}

/*
  * - примечание
  :void - если функция ничего не возвращает;
  :array | callable | bool | float | int | string | object и т.д.
  :?string - если функция возвращает null или string (вместо string может быть любой тип
 */
```

ССЫЛКА НА ДОКУМЕНТАЦИЮ

НАЧИНАЕМ ИСПОЛЬЗОВАТЬ: ИНТЕРПОЛЯЦИЯ:

Подставить переменную типа (int, sting, float) в строку довольно просто:

```
$userName = "Johny";
$userGreeting = "Hello, {$userName}";
```

НАЧИНАЕМ ИСПОЛЬЗОВАТЬ: GUARD EXPRESSION

> Дословно - "защитное выражение", некоторая проверка на входе в функцию.

Пример:

```
/**
 * Возвращает среднее арифметическое элементов массива
 *
 * @param array $arr
 * @return int|null
 */
function calculateAverage(array $arr): ?int
{
  if (empty($arr)) { // GUARD EXPRESSION
    return null;
  }

  return array_sum($arr) / sizeof($arr);
}
```

НАЧИНАЕМ ИСПОЛЬЗОВАТЬ: ДЕСТРУКТУРИЗАЦИЯ (PHP7)

```
 # Массивы:
$languages = ['Java', 'PHP', 'JS'];
[$java, $php, $js] = $languages;
echo $java; // => Java
echo $php; // => PHP
echo $js; // => JS

# Ассоциативные массивы:
$languagesComplexity = ['PHP' => '4/10', 'JS' => '6/10'];
['PHP' => $phpComplexity, 'JS' => $jsComplexity] = $languagesComplexity;

echo $phpComplexity; // => 4/10
echo $jsComplexity; // => 6/10

# Использование в циклах:
$workersData = [
  'developer' => ['name' => 'John', 'age' => 21],
  'managers' => ['name' => 'Vera', 'age' => 23]
];

foreach($workersData as ['name' => $name, 'age' => $age]) {
  echo "Worker name: {$name} | age: {$age} <br>";
}
// =>
// Worker name: John | age: 21
// Worker name: Vera | age: 23
```

НАЧИНАЕМ ИСПОЛЬЗОВАТЬ: Splat Operator (PHP7)

```
 #1: Пример 1:
/**
 * Распечатываем любое количество аргументов
 *
 * @param mixed ...$args
 */
function dbg(...$args)
{
  # В $args массив переданных функции аргументов
  echo '<pre>';

  foreach ($args as $arg) {
    $type = gettype($arg);

    echo "TYPE: {$type}<br>";
    print_r($arg);
  }

  echo '</pre>';
}

dbg(7, 'ABC', [1]); // =>
/*
TYPE: integer
7
TYPE: string
ABC
TYPE: array
Array
(
  [0] => 1
)
*/

# Пример 2:
function getSum($arg1, $arg2)
{
  return $arg1 + $arg2;
}

$integers = [1, 2];
echo getSum(...$integers); // => 3
```

НАЧИНАЕМ ИСПОЛЬЗОВАТЬ: ГЕНЕРАЦИЯ СТРОК В ЦИКЛЕ

Конкатенация и интерполяция порождают новую строчку вместо старой и подобная ситуация повторяется на каждой итерации. Причем строка становится все больше и больше. Копирование

строк приводит к серьезному расходу памяти и может влиять на производительность. Правильный путь:

```
 $coll = ['milk', 'butter', 'eggs', 'bread'];
$parts = [];

foreach ($coll as $item) {
    $parts[] = "<li>{$item}</li>";
}

$innerValue = implode("\n", $parts);
$result = "<ul>{$innerValue}</ul>";
```

НАЧИНАЕМ ИСПОЛЬЗОВАТЬ: ФУНКЦИИ ВЫСШЕГО ПОРЯДКА

> Функции высшего порядка — это функции, которые либо принимают, либо возвращают другие функции, либо делают все сразу. Такие функции, как правило, реализуют некий обобщенный алгоритм (например, сортировку), а ключевую часть логики делегируют вам через анонимную функцию. Главный плюс от применения таких функций — серьезное повышение коэффициента повторного использования кода.

В некоторых случаях вместо объявления пустого массива и наполнением его через циклы, наиболее коротким и оптимальным путем будет использование функций высшего порядка:

```
$languages = ['java', 'php', 'js'];

# array_map:
$languagesWithBigLetter = array_map(function($language) {
  return ucfirst($language);
}, $languages);
print_r($languagesWithBigLetter); // =>
/*
(
Array (
  [0] => Java
  [1] => Php
  [2] => Js
)
*/

# array_filter:
$languagesComplexity = ['PHP' => '4', 'JS' => '6', 'Java' => '8'];
$difficultLanguages = array_filter($languagesComplexity, function ($complexity) {
  return $complexity > 5;
});
print_r($difficultLanguages); // =>
/*
(
Array (
  [JS] => 6
  [Java] => 8
)
*/

# array_reduce:
$languagesComplexity = ['PHP' => '4', 'JS' => '6', 'Java' => '8'];
$sumComplexity = array_reduce($languagesComplexity, function($acc, $complexity) {
  return $acc + $complexity;
}, 0);
echo $sumComplexity; // => 18
```