ООП. Модуль 5. Исключения и другие встроенные классы

Переходим к встроенным классам php.

Исключения

Первые встроенные классы, с которыми мы познакомимся - Исключения. Это особые классы. Вообще это целый механизм в языке программирования, предназначенный для реагирования на возникновения ошибок в программе. А также для обработки исключительных ситуаций, когда в случае невыполнения каких-либо условий нет смысла продолжать выполнение алгоритма. Все исключения наследуются от базового класса \Exception, который в свою очередь реализует интерфейс \Throwable

Вот так выглядит интерфейс Throwable, все методы которого реализует класс \Exception

У него есть методы для получения текста ошибки, кода ошибки, а также другие вспомогательные методы, которые вернут строку и файл где исключение было создано и трейс ошибки.

```php
interface Throwable
{
    public function getMessage();

    public function getCode();

    public function getFile();

    public function getLine();

    public function getTrace();

    public function getTraceAsString();
```

```
    public function getPrevious();

    public function __toString();
}
```

Выбрасывание исключения

Для того чтобы создать (чаще всего говорят: бросить) исключение необходимо использовать ключевое слово throw

Например выбросим новое исключение.

В конструктор Класса исключение принимает три необязательных параметра: Текст ошибки, код ошибки и Предыдущее исключение.

Выполним скрипт.

Как видно из результата скрипт упал с Фатальной ошибкой, в тексте этой ошибки мы видим сообщение из созданного нами объекта исключения. При этом строка THE END - не выполнилась. Т.е. скрипт был прерван выше вывода этой строки.

```
throw new Exception("Возникла какая-то ошибка", 300);

echo 'THE END';

//
PHP Fatal error:  Uncaught Exception:
Fatal error: Uncaught Exception: Возникла какая-то
ошибка in ...\exceptions.php:7
Stack trace:
#0 {main}
  thrown in ...\exceptions.php on line 7
```

Перехват исключений

Теперь мы научились с помощью исключений как бы “Ломать” нашу программу. Но такое поведение нам вряд ли пригодиться. Из-за того что мы не перехватили исключение - программа полностью упала. В обычной ситуации мы бы скорее всего должны были бы сделать какие-то отдельные действия из-за ошибки и продолжить выполнение программы. Это можно реализовать с помощью специальной конструкции перехвата исключений

Для перехвата исключений используется конструкция try ... catch … finally

Внутри блока try размещается код, который может выбросить исключения

Внутри блока catch - исключение перехватывается, т.е. этот блок выполнится только если было выброшено исключение внутри блока try

И блок finally - выполняется всегда.

Посмотрим на результат

```php
try {

    // Имитируем вызов ошибки внутри скрипта
    throw new Exception("Возникла какая-то ошибка", 300);

    echo "Эта строка не выведется" . PHP_EOL;

} catch (Exception $e) {

    echo "Возникла ошибка: " . $e->getMessage() . PHP_EOL;

} finally {
    echo "Эта строка выведется всегда" . PHP_EOL;
}
echo 'THE END' . PHP_EOL;
//
Возникла ошибка: Возникла какая-то ошибка
Эта строка выведется всегда
THE END
```

Теперь закомментируем выбрасывание исключения и посмотрим на результат программы	Эта строка не выведется Эта строка выведется всегда THE END

Блок catch может встречаться несколько раз, при этом внутри блока catch можно выбросить другое исключение.

Воспользуемся исключением RuntimeException - И добавим блок catch для его перехвата, перед блоком Exception RuntimeException и LogicException - это встроенные классы, наследники от класса Exception. Запись LogicException | Exception - означает что блок перехватывает любое из указанных исключений. В данном случае смысла в такой записи нет, т.к. Exception - является родителем для LogicException, а значит только его достаточно в описании, но пример	try { // Имитируем вызов ошибки внутри скрипта throw new RuntimeException("Ошибка во время выполнения программы"); echo "Эта строка не выведется" . PHP_EOL; } catch (RuntimeException $e) { echo "Возникла ошибка Runtime: " . $e->getMessage() . PHP_EOL; throw new Exception("Возникла какая-то ошибка", 300); } catch (LogicException | Exception $e) {

сделан для того, чтобы показать возможность указания нескольких типов исключений в одном блоке.

Порядок следования блоков catch имеет значение - поэтому сначала указываются более узкие исключения, а затем более широкие. Если переместить блок с перехватом базового класса Exception, то при вбрасывании скриптом любого исключения отработает только этот блок.

Вернемся к нашему примеру. Исключение будет перехвачено в первом блоке И в нем мы выбрасываем еще одно исключение.

Обратите внимание на результат работы.

```
    echo "Возникла ошибка Exception: " . $e->getMessage() .
PHP_EOL;

} finally {

    echo "Эта строка выведется всегда" . PHP_EOL;

}

echo 'THE END' . PHP_EOL;

//
Возникла ошибка Runtime: Ошибка во время выполнения
программы
Эта строка выведется всегда
PHP Fatal error:  Uncaught Exception:
Fatal error: Uncaught Exception: Возникла какая-то ошибка
in ...\exceptions.php:19
Stack trace:
#0 {main}
  thrown in ...\exceptions.php on line 19
```

Скрипт упал с фатальной ошибкой, т.к. не было перехвачено исключение. Выброшенное исключение внутри блоков catch не перехватывается этой же конструкцией try catch, а “пробрасывается выше”. При этом, несмотря на то что скрипт упал с ошибкой, блок finally все равно отработал.

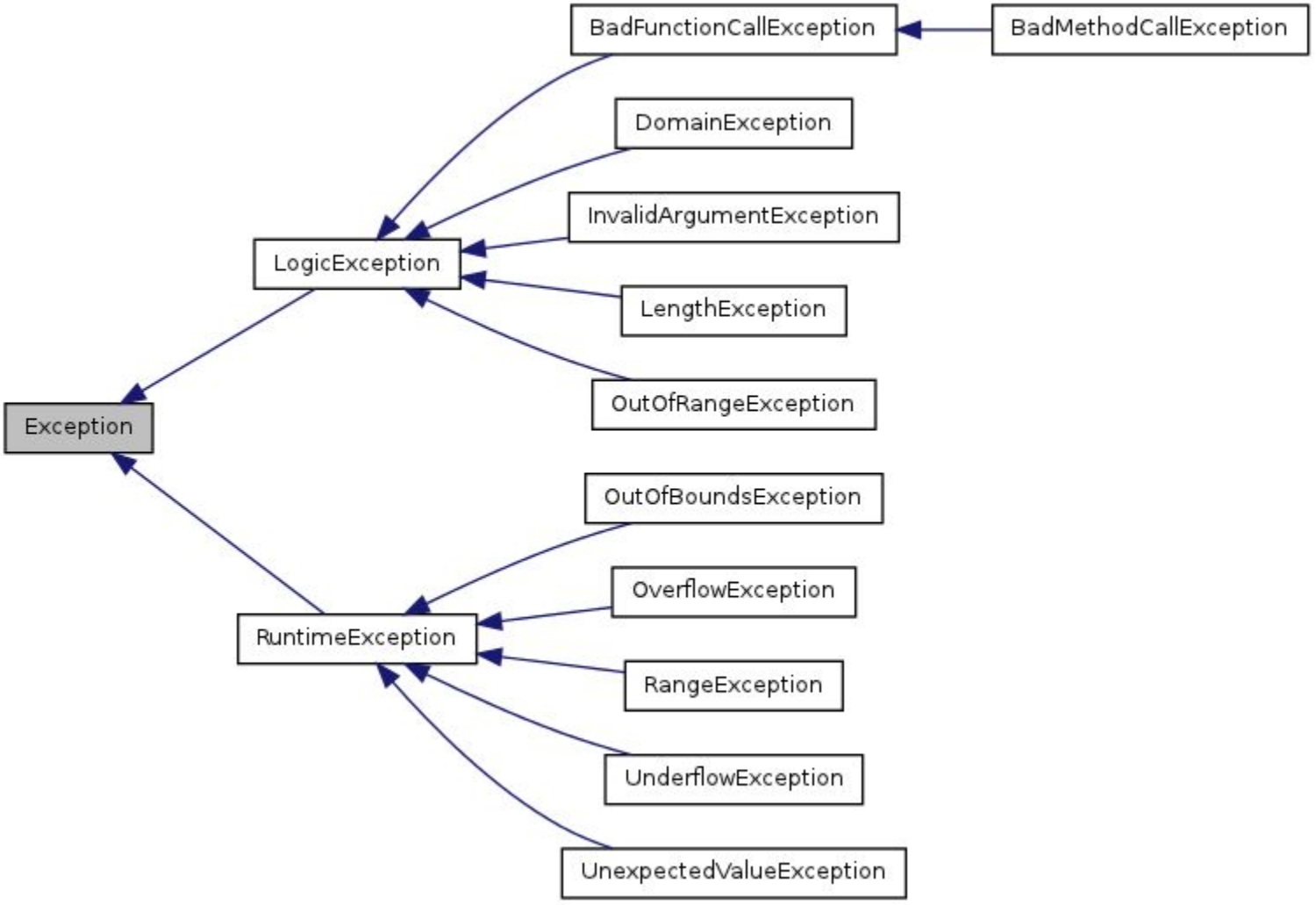
Иерархия исключений

Иерархия встроенных исключений представлена на рисунке

Базово встроенные исключения делятся на два типа:

- LogicException - это повторяемые (постоянные) ошибки внутри программы и появляются из-за ошибок в коде написания программы, соответственно должны быть исправлены - исправлением программного кода.
- RuntimeException - это исключения возникающие только в процессе выполнения программы.

Не будем вдаваться в подробности, остальных исключений, в целом по их названию понятно для чего они созданы, вы всегда можете посмотреть их описание в официальной документации.


BadFunctionCallException
BadMethodCallException
DomainException
InvalidArgumentException
LogicException
LengthException
OutOfRangeException
Exception
OutOfBoundsException
OverflowException
RuntimeException
RangeException
UnderflowException
UnexpectedValueException

Наследование исключений

Вы можете создавать свои виды исключений наследуя их от другого класса Исключений, вплоть до базового класса Exception

При этом при наследовании от Exception можно пользоваться только ограниченным набором свойств этого класса, и только два метода можно переопределить __construct и __toString, потому что остальные методы объявлены с ключевым словом final

Класс Exception описывается примерно так (просто наглядно показываем, что свойства private а методы final - особых подробностей рассказывать больше не

```
class Exception implements Throwable
{
   protected $message = 'Unknown exception';   // сообщение об исключении
   protected $code = 0;                        // пользовательский код исключения
   protected $file;                            // файл, в котором было выброшено
исключение
   protected $line;                            // строка, в которой было
выброшено исключение
   private   $string;                          // свойство для __toString
   private   $trace;                           // трассировка вызовов методов и
функций
   private   $previous;                        // предыдущее исключение, если
исключение вложенное

   public function __construct($message = null, $code = 0, Exception $previous =
null);

   // Переопределяемый
   public function __toString();               // отформатированная строка для
```

нужно про этот класс)	отображения final private function __clone(); // запрещает клонирования исключения final public function getMessage(); // сообщение исключения final public function getCode(); // код исключения final public function getFile(); // файл, где выброшено исключение final public function getLine(); // строка, на которой выброшено исключение final public function getTrace(); // массив backtrace() final public function getPrevious(); // предыдущее исключение final public function getTraceAsString(); // отформатированная строка трассировки }

Вложенные исключения

Блоки с перехватом исключений можно вкладывать один в другой, при этом уровень вложенности практически не ограничен. В этом случае при выбрасывании очередного исключения - это исключение будет проверяться в каждом следующей обрамляющей его перехватывающей конструкции. Если конструкция не перехватывает выброшенный тип исключения - то управление передаются конструкции выше и так далее до появления фатальной ошибки.

Рассмотрим пример, для этого реализуем следующую задачу:

Разработать функцию, которая поделит $a на $b, при этом делимое ($a) должно быть строго больше делителя, оба числа должны быть положительными. Программа должна вывести результат или ошибку.

Создадим свое пользовательское исключение BadValueException, наследуем от InvalidArgumentException - это исключение похоже на наш случай больше всего.	`class BadValueException extends \InvalidArgumentException {}`
Добавим два своих исключения, на каждый вид ошибки	`class ToBigValueException extends BadValueException {};` `class NegativeValueException extends BadValueException {};`
Реализуем необходимую функцию, которая будет выполнять логику нашей программы. В каждом соответствующем случае функция будет выбрасывать подходящее исключение. Не забываем что делить на ноль нельзя - в этом случае мы выбросим другое	`function arithmeticOperation($a, $b) {` `  if ($a < 0 || $b < 0) {` `  throw new NegativeValueException("a < 0 \|\| b < 0");` `  }` `  if ($a <= $b) {` `  throw new ToBigValueException("a <= b");` `  }` `  if ($b == 0) {` `  throw new \InvalidArgumentException('b == 0');` `  }`

исключение, т.к. оно не описано в нашей задаче	
Реализуем остальную часть программы и протестируем результат. Обратите внимание, что мы используем вложенные блоки try catch, внутренний блок ловит только наши исключения - и в явном виде выводит ошибку. А внешний блок ловит все остальные исключения и логирует их. При этом программа выполнится для всех элементов $values	

```
    return $a / $b;
}
```

```
$values = [
    ['a' =>  0, 'b' => 2],
    ['a' => -1, 'b' => -3],
    ['a' => 10, 'b' => 0],
    ['a' => 3,  'b' => 1],
];


foreach ($values as $value) {
    try {

        try {

            echo "a = " . $value['a'] . " b = " .
$value['b'] . "   ";
            $c = arithmeticOperation($value['a'],
$value['b']);
            echo "Результат: " . $c . PHP_EOL;

        } catch (BadValueException $e) {
            echo "Проблема с числами: " . $e->getMessage() .
PHP_EOL;
        }

    } catch (Exception $e) {
        echo "Логируем ошибку: " . $e->getMessage() .
```

	PHP_EOL; } }
Посмотрим на результат выполнения программы. Все как и требовалось	// a = 0 b = 2 Проблема с числами: a <= b a = -1 b = -3 Проблема с числами: a < 0 || b < 0 a = 10 b = 0 Логируем ошибку: b == 0 a = 3 b = 1 Результат: 3

Исключения много где могут быть применены, не только для выбрасывания ошибок но и для упрощения структуры программы.

Очень подходящий пример для применения исключений - сложная валидация. Очень часто различные условия и проверки вкладывают одно в другое, if, которые лежит где-то внутри другого if, который внутри другого и т.д - большой уровень вложенности, и не совсем понятно как быть с сообщениями об ошибке, которые заполняются внутри одно сообщение там, другое тут. При применении исключение все становится предельно просто

Просто выбрасываем исключение при возникновении ошибки внутри валидации, и в $errorMessage - у вас всегда будет текст этой ошибки, если что-то пошло не так.	try { // сложная валидация, при ошибке валидации просто выбрасываем исключение // после нее дальнейшая логика работы } catch (ValidationException $e) { $errorMessage = $e->getMessage(); }

Перехват непойманных исключений

Можно изменить поведение скрипта, в случае если выброшенное исключение не было поймано ни одним блоком try..catch. Для этого существует специальная функция set_exception_handler(callable $exception_handler)

У функции один параметр - callable - объект, которому в качестве единственного параметра приходит исключение.

А внутри функции уже ваша логика, что с таким исключением делать. Если установлен такой обработчик, то фатальная ошибка не появиться, если вы ее не сформируете внутри этой функции.

```php
set_exception_handler(function(Throwable $e) {
    //
});
```

Такой перехват используется во всех популярных фреймфорках, чтобы красиво отобразить исключение программисту при разработке, или чтобы не показывать ошибки в продуктивной среде.

Замыкания

Говоря о замыканиях, чаще всего подразумеваются анонимные функции. Мы уже немного сталкивались с ними ранее, пришло время разобраться с ними поподробнее. Некоторые встроенные функции в php, в качестве параметра принимают принимают псевдотип callable.

Например функции: call_user_func() ее аналог call_user_func_array() функции usort() и другие	`function call_user_func(callable $function, ...$parameter);` `function call_user_func_array(callable $function, array $params);` `function usort(array $array, callable $function);`

Что может быть передано в качестве значения для псевдотипа callable:

- Полное имя функции в виде строки, например ‘strlen’ или ‘trim’
- Анонимная функция, пример будет чуть позже
- В виде массива имя класса и имя его публичного статический метод, или реальный объект и имя его публичного метода.

Давайте посмотрим как выглядит каждый из способов передачи callback-функции.

Для начала создадим вспомогательную функцию для тестирования

Создадим функцию test(), она принимает два параметра имя - строка, и callable - функция.

Вызов callback-функции для примеры мы производим двумя различными способами - обращаясь к функции по имени, и с помощью call_user_func_array()

```php
function test($name, callable $callback)
{
    echo 'invoke   : ' . $callback($name) . PHP_EOL;
    echo 'user_func: ' . call_user_func_array($callback, [$name]) . PHP_EOL;
}
```

Callback в виде строки

Мы можем передать в виде строки полное имя функции - и она будет вызвана при выполнении callback-функции.

Создадим простую функцию simpleHello и передадим ее. (мой файл

```php
function simpleHello($name = '')
{
    return 'Simple Hello To: ' . $name;
}
test('User 2', "\level2\module5\callbacks\simpleHello");
```

создан в пространстве имен …, поэтому полное имя выглядит так)

Результат выглядит так как мы и ожидали, в обоих случаях он одинаков.

При выводе строк была вызвана наша функция, и ей был передан параметр $name - который мы указали еще при вызове нашей тестовой функции.

```
//
invoke   : Simple Hello To: User 2
user_func: Simple Hello To: User 2
```

Анонимная callback-функция

Мы можем не создавать внутри нашего кода дополнительные функции и выделять особое имя для них, чаще гораздо удобнее использовать анонимную функцию - функцию без имени.

Для вызова функции test нам не пришлось создавать отдельно никаких функций, а мы создали функцию прямо на ходу. У этой функции нет имени, поэтому это анонимная функция.

Результат работы аналогичен.

```
test('User 1', function($name) {
   return 'Anonim Hello To: ' . $name;
});

//
invoke   : Anonim Hello To: User 1
user_func: Anonim Hello To: User 1
```

Использование анонимных функций также дает дополнительную выгоду: в нее можно пробрасывать переменные из другой области видимости, с помощью оператора use. В одной из наших домашних работ мы уже сделали это, создав универсальную функцию сортировки массива по его ключу.

```php
function array_sort($arr, $key = 'sort', $sort = 'asc') {
   usort($arr, function($a, $b) use ($key, $sort) {
       return $sort == 'desc' ? $b[$key] <=> $a[$key] : $a[$key] <=> $b[$key];
   });
   return $arr;
}
```

В этом примере в анонимную callback-функцию пробрасываются значения переменных $key и $sort, которые используются для управления порядком сортировки - по какому ключу и в каком направлении.

При вызове функции по имени - такое проделать не получится.

Callback-метод объекта или класса

Также в качестве callback-функции может быть передан массив первым элементом которого является имя класса или реальный объект, а вторым - название функции, которую нужно вызвать.

Создадим класс и передадим один из его методов в качестве callback-функции

```php
class SayHello
{
    public static function helloStatic($name = '')
    {
        return 'Say Hello To: ' . $name;
    }

    public function hello($name = '')
    {
        return 'Say Hello To: ' . $name;
    }
}
```

В случае когда мы указываем имя класса - метод должен быть статическим.

Когда для callback параметра передаем объект - то метод может быть и не статическим.

При этом методы обязательно должны быть публичными - или получим фатальную ошибку из-за недостаточного уровня доступа.

```php
test('User 3', [SayHello::class, 'helloStatic']);

$sayHello = new SayHello();

test('User 4', [$sayHello, 'hello']);

//
invoke   : Say Hello To: User 3
user_func: Say Hello To: User 3
invoke   : Say Hello To: User 4
user_func: Say Hello To: User 4
```

Класс Closure

При передаче callable параметра в функцию нет возможности как-то управлять этой функцией, однако существует специальный объект Closure, который такую возможность дает.

Класс содержит в себе такие методы

```
class Closure {
   private __construct ( void ) //запрет на создание нового экземпляра
   // Дублирование замыкания с указанием конкретного связанного объекта и
области видимости
   public static Closure bind ( Closure $closure , object $newthis [, mixed
$newscope = "static" ] )
   // Дублирование замыкания с указанием связанного объекта и области видимости
   public Closure bindTo ( object $newthis [, mixed $newscope = "static" ] )
   // Связывает и запускает замыкание
   public mixed call ( object $newthis [, mixed $... ] )
   // конвертирует любой callable в замыкание
   public static Closure fromCallable ( callable $callable )
}
```

Основное назначение этого класса - изменить контекст вызываемой callback-функций, т.е область видимости. Проще разобраться зачем это нужно на примере:

Создадим класс с приватным свойством $value. Это свойство инициализируется в конструкторе и на него установлен геттер (приватный специально для демонстрации).

Дополнительно у класса есть метод formatValue - который принимает объект Closure.

Внутри этого метода у объекта - замыкания мы вызываем метод и передаем ему в качестве контекста - указатель на текущий класс, это будет значить, что функция внутри замыкания, как-будто будет частью класса, а значит внутри нее можно будет вызывать все методы и обращаться к любому свойству текущего класса.

```php
class ClosureUsageExample
{
    private $value = 0;

    public function __construct(int $value)
    {
        $this->value = $value;
    }

    private function getValue()
    {
        return $this->value;
    }

    public function formatValue(Closure $closure)
    {
        return $closure->call($this);
    }
}
```

Создадим анонимную функцию, **кстати ее можно положить в переменную**

```php
$formatter = function() {
    return sprintf("Price: %01.2f$", $this->getValue());
};
```

Создадим два объекта и посмотрим на результат, **кстати анонимные функции автоматически силами языка оборачиваются в объекты Closure**	`$example1 = new ClosureUsageExample(4);` `$example2 = new ClosureUsageExample(3);` `echo $example1->formatValue($formatter) . PHP_EOL;` `echo $example2->formatValue($formatter) . PHP_EOL;` `//` `Price: 4.00$` `Price: 3.00$`

Код выполнится без ошибок, даже несмотря на то, что обращение внутри функции замыкания, объявленной где-то в нашем коде, происходит к приватному методу классу, но благодаря изменению контекста выполнения функции замыкания - такое обращение будет корректным.

Вкратце о некоторых других встроенных классах

В языке есть множество других встроенных классов и интерфейсов, мы упомянем некоторые из них, без погружения в подробности, т.к. их использование не настолько широко в своем коде, как скажем использование замыканий и исключений. Но об этих интерфейсах стоит упомянуть.

Интерфейс Traversable

Интерфейс, определяющий, является ли класс обходимым (traversable) с использованием foreach. Т.е. если класс реализует этот интерфейс, то можно смело использовать foreach на нем.

Но это особый внутренний интерфейс и вы не можете реализовать его своих классах. Вместо него нужно использовать либо IteratorAggregate, либо Iterator

Интерфейс Iterator

Интерфейс для внешних итераторов или объектов, которые могут повторять себя изнутри.

Интерфейс IteratorAggregate

Интерфейс для создания внешнего итератора.

С помощью этих итераторов можно создавать объекты, содержащие в себе какие-либо коллекции данных, для которых работе с которыми подойдут циклы. Или если вам нужно циклически проходить по свойствам этого объекта, при этом не будет необходимости как-то пытаться самостоятельно собрать все свойства в массив.

Интерфейс ArrayAccess

Интерфейс обеспечивает доступ к объектам в виде массивов. Можно будет получать и устанавливать свойства, как будто элементы массива, при этом реализуя методы этого интерфейса - вы можете назначить нужную вам логику при добавлении и получении свойства. Чем-то схоже с магическими методами перегрузки свойств.

Интерфейс Serializable

Интерфейс для индивидуальной сериализации.

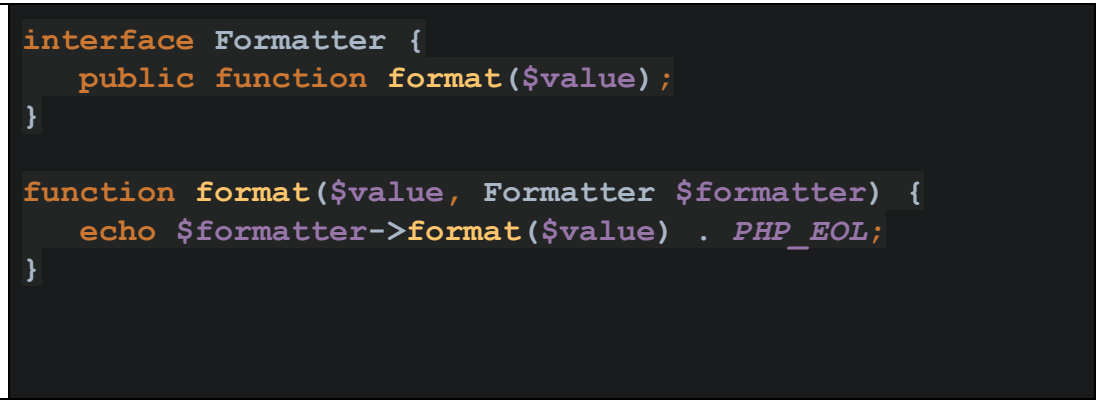
Классы, реализующие этот интерфейс, больше не поддерживают магические методы __sleep() и __wakeup(). Метод serialize вызывается всякий раз, когда необходима сериализация экземпляру класса, и соответствующий метод unserialize() вызывается как конструктор вместо вызова __construct() при десериализации.

Бонус - Анонимные классы

Помимо анонимных функции в php 7+ появились и анонимные классы. Иногда создавать отдельный класс внутри приложения не имеет особого смысла, но это необходимо, например, чтобы заработала какая-нибудь подключенная библиотека. Или в каком-то месте программы вам нужна уникальная реализация интерфейса, которая больше нигде не понадобиться. Например, в какой-нибудь части административного интерфейса, вы меняете элемент, и вам нужно поставить заглушку чтобы не уходили уведомления при изменении элемента.

В этом случае очень не хочется создавать отдельный класс, но без него не будет работать. Здесь и могут пригодиться анонимные классы.

Эти классы - это точно такие же классы, как и обычные другие, они могут быть наследованы от других классов, могут реализовывать интерфейсы, переопределять методы и все, что умеют другие классы, за исключением того - что вы не можете создавать экземпляры такого класса, потому что у него нет имени - и его описание сразуже превращается в экземпляр класса.

Воспроизведем проблемную ситуацию Есть интерфейс Formatter и вспомогательная функция, завязанная на этот интерфейс. И в какой-то части программы вам нужен совершенно уникальный форматтер, например с 10-ю знаками после запятой.	

```php
interface Formatter {
    public function format($value);
}

function format($value, Formatter $formatter) {
    echo $formatter->format($value) . PHP_EOL;
}
```

Воспользуемся анонимным классом. С помощью ключевого слова new - сразу же будет создан экземпляр класса, у которого нет имени. Кстати анонимный класс также как и анонимную функцию можно положить в переменную	`$tmpClass = new class implements Formatter {` `  public function format($value)` `  {` `  return sprintf("Price: %01.10f$", $value);` `  }` `};`
Протестируем программу. Готово	`format(10, $tmpClass);` `//` `Price: 10.0000000000$`

Можно было не создавать переменную, а сразу в качестве второго параметра создать экземпляр анонимного класса. В примере они отделены для наглядности.

Вот так без создания новых файлов с классами внутри приложения удалось реализовать программу с уникальным форматированием.