Wstęp do klas

Witaj w świecie PHP Zorientowanym obiektowo. Programowanie obiektowe jest najnowszym trendem w informatyce. Święci sukcesy z tego względu że pisane tą techniką programy przypominają widzenie świata przez ludzki mózg. Za pomocą klas, możemy stworzyć klasę radio, która zawiera w sobie obiekty klasy głośnik. Brzmi zachęcająco, prawda?

Czym że takim jest więc klasa ?

Klasa jest to konstrukcja grupująca elementy: funkcje i zmienne, jest to wzorzec do tworzenia obiektów.

Funkcje nazywane są metodami, natomiast zmienne właściwościami.

Oto przykład konstrukcji klasy

```
class nazwaKlasy{
    public $jakasZmienna;
    public function jakasMetoda(){
        echo "Wywolalismy metode naszej klasy";
    }
}
```

Jest to definicja klasy o nazwie nazwaKlasy, zawiera ona jedną właściwość - $jakasZmienna jedną metodę jakasFunkcja. Słowo pubic jest modyfikatorem dostępu.

Zbiór wartości zmiennych zwany jest stanem klasy.

Modyfikatory dostępu

Istnieją trzy rodzaje modyfikatorów dostępu :

- private - właściwość lub metoda tego typu może zostać użyta jedynie przez klasę której jest częścią.
- protected - wlaściwość lub metoda może być użyta wewnątrz klasy której jest częścią lub w klasach potomnych (o tym wkrótce)
- public - dana właściwość lub metoda jest ogólnodostępna

Cechy programowania obiektowego

Programy pisane obiektowo charakteryzują się trzema podstawowymi cechami

- Abstrakcja
- Hermetyzacja
- Dziedziczenie

Abstrakcja

Abstrakcja polega na przekazaniu do publicznej wiadomości jedynie potrzebnych wiadomości na temat klasy, bez zgłebiania się w szczegóły ich realizowania(implementacji). Przykładowo, mając klasę Kwadrat, użytkownik ma do dyspozycji metodę wraz z informacjami jakich argumentów metoda oczekuje, i jaki będzie wynik działania metody. Nie potrzebuje wiedzieć, w jakis sposób jest to dokładnie realizowane, nie musi wiedzieć jakie właściwości i metody są używane by osiągnąc zamierzony cel.

Hermetyzacja

Hermetyzacja zwana także enkapsulacją służy do ukrywania implementacji. Zapewnia utrzymanie właściwego stanu klasy. Oznacza to że dane właściwości inicjowane, modyfikowane i odczywywane są w określony sposób, zgodny z założeniem klasy. Mówiąc obrazowo, hermetyzacja dba by przykładowo właściwość wiek mogła przyjąć liczby z zakresu od 0 do 120.

Hermetyzacja realizowana jest poprzez uczynienie właściwości prywatnymi. Dostęp do nich realizowany jest za pomocą publicznych metod które gwarantują właściwą obsługę.

Dziedziczenie

Dziedziczenie pozwala rozszerzać klasy o nowe cechy, bez powielania kodu.Tworzymy logicznie powiązane struktury. Dzięki dziedziczeniu mając klasę Jednoślad możemy stworzyć klasę Rower dziedziczącą po niej, znaczy to że nowa klasa już na wstępie otrzymuje cechy klasy dziedziczonej. My dodajemy jedynie nowe cechy.

PHP obsługuje dziedziczenie proste, w przeciwieństwie do np C++ nie możemy dziedziczyć po kilku klasach.

Klasy i obiekty

Klasa jest teoretycznym projektem (wzorcem), fizyczny egzemplarz klasy nazywamy obiektem.

Obiekty danej klasy tworzymy operatorem **new**, zwraca on referencję do utworzonego obiektu.

Przykładowy kod:

```
<?
class Slon
{
    public $imie;
    public $dlugoscTraby;
    public $nieTegoChce = ':(';

    public function machnijTraba() {
        echo 'macham traba';
    }
};
$dumbo = new Slon;          //1
```

```
$dumbo->imie = 'dumbo';      //2
$dumbo->dlugoscTraby = 50;
$dumbo->machnijTraba();      //3
echo $dumbo->dlugoscTraby;   //4
?>
```

- //1 // Tworzymy obiekt klasy Slon, referencję czyli jego adres przechowujemy w zmiennej $dumbo.
- //2 // Ustawiamy właściwość $imie. Do właściwości lub metody odwołujemy się operatorem **->**. Nazwy właściwości podajemy bez znaku dolara **$**
- //3 // Wywołujemy metodę machnijTraba()
- //4 // Wypisujemy na ekran wartosc właściwości $dlugoscTraby

Należy pamiętać że nazwy właściwości do której się odwołujemy podajemy bez znaku dolara **$**

Wywołanie kodu :

```
<?php
$imie = 'nieTegoChce';
$dumbo = new Slon;
$dumbo->$imie = 'dumbo';
?>
```

Nie uzyskamy dostępu do właściwości $imie,obiektu pokazywanego przez $dumbo. PHP przeszuka bieżący zakres w poszukiwaniu zmiennej $imie i pobierze jej wartość, a następtnie użyje jej jako nazwę właściwości do pobrania (zmienna zmiennej).Tak więc w podanym przykładzie uzyskamy dotęp do właściwości - $nieTegoChce.

Dynamiczne deklarowanie właściwości

PHP zezwala na dynamiczne deklarowanie właściwości obiektów

Przykład :

```
<?php $dumbo = new Slon;
$dumbo->kolor = 'pomaranczowy';
?>
```

Klasa Slon nie posiada właściwości kolor, tak więc PHP dynamicznie utworzy tą właściwość i doda ją do obiektu wskazywanego przez zmienną $dumbo. PHP utworzy ją jedynie dla tego obiektu, tak więc wszystkie inne obiekty klasy Slon, nie będą posiadać tej właściowści.

$this

Wewnątrz metod obiektu posiadasz pełny dostęp do jego właściwośći, jednak dostęp wymaga wywołania specjalnej zmiennej, która inforumje metodę że działamy na elemencie klasy. Zmienna ta nosi nazwę **$this** i zawsze wskazuje na bieżąco wykorzystywany obiekt.

Zmienną $this musimy także użyć gdy chcemy wywołać metodę wewnątrz klasy.

```
<?
class Slon
{
    private $imie;
    public function ustawImie($dane){
        //$imie = $dane;     //3
        $this->imie = $dane; //4
    }
    public function podajImie(){
        return $this->imie;
    }
    public function przedstawSie(){
        return 'Mam na imie'.$this->podajImie();
    }
};
$dumbo = new Slon;
//$dumbo->imie = 'dumbo';    //1
$dumbo->ustawImie('dumbo');  //2
//echo $dumbo->imie;         //5
echo $dumbo->przedstawSie(); //6
?>
```

- //1 //-Nie możemy ustawić w taki sposób właściwości $imie gdyz jest ona prywatna.
- //2 // Uswawiamy imie publiczna metoda ustawImie() (heretyzajca), która jako metoda obiektu ma dostep do metod i właściwości prywatnych
- //3 // Stworzylibyśmy jedynie zmienną lokalną o nazwie $imie, która po wykonaniu metody (koniec zakresu) została by usunięta.
- //4 //Zmienna $this mówi nam że pracujemy na bieżącym obiekcie, (w tym przypadku na obiekcie wskazywanym przez $dumbo)
- //5 //Tego też nie możemy zrobić, $imie ma zakres private wiec nie mozemy go wypisać.
- 6-Wywolujemy metode przedstawSie, podaje ona $imie za pomoca metody podajImie

Podsumowanie

Obiekt jest reprezentantem klasy, tworzony za pomocą new.

Do właściwości, metody odwołujemy się za pomocą -> , w przypadku właściwości nie używamy znaku $

Zmienna $this służy do odwoływania się do metody, obiektu wewnątrz klasy

Praktyka

Poniżej przedstawię praktyczne zastosowanie podstawowej klasy. Zaimplementujemy klasę pozwalającą na dostęp do bazy danych MySQL. Najpierw zaczniemy od mini projektu. Najpierw musimy zadecydować z jakich metod ma się składać nasza klasa. Na początek wystarczy nam metoda do łączenia się z bazą, rozłączania i pobierania podanej liczby elementów z danej tabeli. W generatorze dodajemy nową klasę, nazwijmy ją db_mysql. Następnie dodajmy dwa pola i ustawiamy je na private. Jedno connect będzie przechowywać utworzone połączenie, domyślnie ustawiamy je na NULL, drugie limit posłuży nam do ograniczenia ilości pobieranych elementów - ustawmy je domyślnie na 10. Obie ustawiliśmy jako private, ponieważ nie chcemy aby były zmieniane poza klasą Wszystkie metody jakie

tworzymy muszą być publiczne, ponieważ będziemy korzystać z nich poza klasą. Metody nazwijmy kolejno: connect, disconnect, selectAll. Projekt mamy gotowy, w generatorze Zapisz/Otwórz i kolejnie Generuj PHP. Mamy gotowy szkielet naszej klasy, kopiujemy i poddajemy dalszej obróbce.

Tak powinien wyglądać projekt w graficznym generatorze. KLIKNIJ

```
<?php
class db_mysql {
        private connect = NULL;
        private limit = 10;
        public connect() {
            //Treść funkcji
        }
        public disconnect() {
            //Treść funkcji
        }
        public selectAll() {
            //Treść funkcji
        }

};
?>
```

W metodzie connect musimy przyjąć 4 parametry, serwer bazy danych, nazwę bazy, login i hasło. Za pomocą mysl_connect łączymy się z serwerem bazy i kolejnie za pomocą mysql_select_db ustawiamy aktywną bazę danych.
W metodzie disconnect jeśli istnieje połaczenie z bazą to wykonujemy rozłączenie za pomocą mysql_close.
W najciekawszej metody selectAll, przyjmijmy dwa parametry, $table - z nazwą tabeli z której chcemy pobrać dane i opcjonalnie $limit - ilość elementów jaką chcemy pobrać. W ciele metody zaczynamy od sprawdzenia czy istenieje połączenie z bazą danych, jeśli nie wychodzimy z niej. Kolejnie sprawdamy czy podana wartość $limit jest poprawna, jeśli jej nie podaliśmy lub przekraszamy dozwolony zakres to ustawiamy ją na wartość maksymalną ustawioną na początku klasy. Następnie wykonujemy proste zapytanie do bazy danych (mysql_query). Jego rezultat przekazujemy do mysql_fetch_assoc i w petli while dokonujemy operacji przepisania do tablicy, którą następnie zwracamy za pomocą return. Klasę mamy gotową. Tworzymy obiekt klasy, wykonujemy metodę connect, następnie selectAll i jej rezultat wyświetlamy za pomocą print_r. Na koniec jeszcze disconnect. A wygląda to mniej więcej tak: (oczywiście taką klase można napisać na miliony różnych sposobów, wszystko zależy od potrzeb jakie ma spełniać i sposobu w jaki ją zaimplementujemy)

```
<?php
class db_mysql {
    private $connect   = NULL;
    private $limit      = 10;

    public function connect($host, $login, $pass)
    {
        $this->connect = mysql_connect(
                                $host,
```

```
                                $login,
                                $pass);
        if ($this->connect)
           mysql_select_db($database, $this->connect);
    }

    public function disconnect()
    {
        if(is_resource($this->connect)) {
            mysql_close($this->connect);
        }
    }

    public function selectAll($table, $limit = null, )
    {
        if (!$this->connect) return false;

        if ($limit === null || $limit > $this->limit)
        {
            $limit = $this->limit;
        }
        $result = (
        mysql_query(
         "SELECT * FROM ".$table." LIMIT ".$limit
        ));
        while ($row = mysql_fetch_assoc($result)) {
            $a[] = $row;
                }
        return $a;
    }
};

$mysql = new db_mysql;
$mysql->connect('localhost', 'test', 'root', '1234');
print_r($mysql->selectAll('answer', 30));
$mysql->disconnect();
?>
```

Zadania

Napisz klasę odcinek w przestrzeni dwuwymiarowej (x,y). Klasa ta ma zawierać:

- Punkt początkowy i punkt końcowy o zakresie prywatnym.
- Dostęp do punktów zrealizowany ma być za pomocą publicznych metod.
- Metodę zwracającą długość odcinka.

Podpowiedź, do obliczania pierwsiatka służy funkcja **sqrt()**, przyjmuje ona jako argument liczbę a zwraca pierwiastek tej liczy.

Schowaj rozwiązanie

```
<?php
  class Odcinek{
    /* $pocz jest to tablica asocjacyjna przechowująca
    punkt początkowy odcinka, element o kluczu x
    przechowuje składową x wektora element o kluczu y,
    przechowuje składową y odcinka */
```

```
    private $pocz = array('x'=> 0, 'y' => 0);

    // $kon, jest to analogiczna tablica jak $pocz
    // przechowuje koncowy punkt odcinka
    private $kon = array('x'=> 0, 'y' => 0);

    /* metoda ustawWspolrzedne, pobiera 4 argumenty
     pierwsze dwa ustawiają punkt początkowy odcina
     druga para ustawia punkt końcowy odcinka */
    public function ustawWspolrzedne(
        $pocz_x, $pocz_y, $kon_x, $kon_y)
    {
        $this->pocz['x'] = $pocz_x;
        $this->pocz['y'] = $pocz_y;
        $this->kon['x'] = $kon_x;
        $this->kon['y'] = $kon_y;
    }

    /* metoda wypisz wspolrzedne wypisuje wspolrzedne
    odcinka wypis następuje w formacie:[x1,y1]->[x2,y2]
    gdzie x1, y1 są to współrzędne punktu początkowego
    x2, y2 - współrzędne punktu końcwego */

    public function wypiszWspolrzedne(){
        echo "wektor ma wsporzedne :
            [{$this->pocz['x']},{$this->pocz['y']}]
          ->[{$this->kon['x']},{$this->kon['y']}]";
    }

    /* metoda zwrocDlugosc zwraca długość odcinka
     realizowane jest to za pomocą wzoru znanego
     z matematyki
     dlugosc = sqrt( (x2 - x1)² + (y2 - y1)² ) */
    public function zwrocDlugosc(){
        return sqrt(
                   ($this->kon['x'] - $this->pocz['x'])
                 * ($this->kon['x'] - $this->pocz['x'])
                                    +
                   ($this->kon['y'] - $this->pocz['y'])
                 * ($this->kon['y'] - $this->pocz['y'])
                   );
    }
  }
?>;
```

Agregaty

Klasy mogą oprócz właściowści i metód zawierać obiekty innych klas

Zawieranie takie nosi nazwę agregacji.

Oto przykład

```
<?php
class Traba{
    public $dlugoscTraby;
```

```
}
class Slon
{
    public $traba;
    public function zmienDlugos( $nowe ){
          $this->traba->dlugoscTraby = $nowe;
    }
}
$dumbo = new Slon;
$dumbo->traba = new Traba;         //1
$dumbo->traba->dlugoscTraby = 2;  //2
echo $dumbo->traba->dlugoscTraby; //3
$dumbo->zmienDlugosc(3);           //4
?>
```

- //1// Tworzymy obiekt klasy Traba wewnatrz obiektu klasy Slon
- //2// Zmieniamy wartosc właściwości $dlugoscTraby, możliwe jest to gdyz wlasciwość $traba jak i $dlugoscTraby maja modyfikator dostepu public
- //3// Wypisujemy wartosc właściwości $dlugoscTraby
- //4// Wlasciwość $dlugoscTraby możemy modyfikowac także za pomoca metod.

Tak zmienił by się nasz kod gdyby $traba jak i $dlogoscTraby były private

```
<?php
class Traba{
    private $dlugoscTraby;
    public function zmienDlugosc( $dana ){
        $this->dlugoscTraby = $dana;
    }
    public function podajDlugosc(){
    return $this->dlugoscTraby;
    }
}
class Slon
{
    private $traba;
    public function stworzTrabe(){
        $this->traba = new Traba;
    }
    public function zmienDlugosc( $nowe ){
          $this->traba->zmienDlugosc( $nowe );
    }
    function podajDlugosc(){
    return $this->traba->podajDlugosc();
    }
}
$dumbo = new Slon;
//$dumbo->traba = new Traba;
$dumbo->stworzTrabe();
$dumbo->zmienDlugosc(2);
//echo $dumbo->traba->dlugoscTraby;
echo $dumbo->podajDlugosc();

?>
```

- //1// Nie możemy stworzyc obiektu Traba jak poprzednio gdyz wlasciwosc $traba jest private, angazujemy wiec metode stworzTrabe() która posiada do niej dostep

- //2// Chcac zmienic dlugoscTraba, wywołujemy metodę zmienDlugosc obiektu $dumbo, ktora wywoluje metode zmienDlugosc obiektu $traba. Użytkownik wcale nie musi o tym wiedzieć

UWAGA : poprawność programistyczna stanowi by obiekty agregowane miały zakres private, i tworzene/modyfikowane były za pomocą specjalnie dostosowanych do tego metod. W przeciwnym wypadku, mając swobodny dostęp mamy możliwość użycia ich niezgodnie z przeznaczeniem. Jest to jeden z wymogów hermetyzacji.

Zadania

Napisz klasy Przycisk i Panel

Cechy klasy Przycisk:

- nazwa, wysokość, szerokość (właściwości powinny być prywatne)

Cechy klasy Panel

- klasa Panel zawiera dynamiczną ilość przycisków co znaczy że możemy dodawać i usuwać przyciski z panelu,
- klasa potrafi podać ilość przycisków które zawiera
- właściwości powinny być private

Schowaj rozwiązanie

```
 <?php
 /* Tworzymy klase Przycisk :
  tablica asocjacyja $rozmiar przechowuje
  szerokosc i wysokosc, a metoda stworz
  ustawia szerokosc, wysokosc i nazwe przycisku */
class Przycisk{
    private $nazwa;
    private $rozmiar = array(
                        'szerokość' => 0,
                        'wysokość' => 0);

    public function stworz($nazwa,$szerokosc,$wysokosc)
    {
    $this->nazwa = $nazwa;
    $this->rozmiar['szerokość'] = $szerokosc;
    $this->rozmiar['wysokość']  = $wysokosc;
    }
}

/* Tworzymy klase Panel:
 tablica $przyciski przechowuje obiekty
 klasy Przycisk, wlasciwosc $ilosc mowi nam
 ile jest agregowanych przyciskow,
 metoda dodaj dodaje nowy Przycisk do Panelu,
 metoda usun usuwa dany Przycisk z Panelu,
 metoda zwrocIlosc zwraca ilosc
 agregowanych przyciskow*/
```

```
class Panel{
    private $przyciski = array();
    private $ilosc = 0;
    public function dodaj($nazwa,$szerokosc,$wysokosc)
    {
    /* jeśli podana szerokość i wysokość nie są typu
    całkowitego to nie tworzymy nowego przycisku */
    if(!(is_int($szerokosc) && is_int($wysokosc))){
      echo 'podales zle dane';
      return;
    }
    else{
      //tworzymy nowy przycisk, pamietajac ze tablica
      //indexowana jest od 0
      $this->przyciski[$this->ilosc] = new Przycisk;
      /* wywołujemy f. stworz dla nowo utworzonego
      przycisku jednocześnie zwiększamy licznik */
      $this->przyciski[$this->ilosc++]->stworz(
                    $nazwa,$szerokosc,$wysokosc);
      }
    }
    public function usun($ktory){
    //Najpierw sprawdzamy czy jest taki przycisk
    if($ktory > $this->ilosc){
        echo 'nie ma takiego przycisku';
        return;
    }
    else {
        /* jesli jest to usuwamy dany przycisk,
        pamietając że tablica indexowana jest od 0,
        musimy zmniejszyć $ktory o 1 */
        unset($this->przyciski[--$ktory]);
        // zmniejszamy licznik
        --$this->ilosc;
        }
    }

    function zwrocIlosc(){
        return $this->ilosc;
    }

}
?>
```

Konstruktor i Destruktor

Konstruktor

Konstruktor jest to specjalna metoda która wywoływana jest podczas tworzenia obiektu.

Metoda konstruktora musi nazywać się **__construct**. Pobiera dowolną ilość argumentów jednak nie może nic zwracać.

Klasa nie musi zawierać konstruktora (te z poprzednich przykładów nie posiadały). Jednak gdy klasa posiada konstruktor, podczas tworzenia obiektów należy podać argumenty, które zostaną przesłane do konstruktora.

```
<?php
class Traba{
    public function __construct( $dlugosc ){
        $this->dlugoscTraby = $dlugosc;
    }
    private $dlugoscTraby;
}
class Slon{
    private $imie;
    private $traba;
    public function _construct( $imie,$dlugosc ){ //2
        $this->imie = $imie;
        $this->traba = new Traba($dlugosc);  //3
    }

    }

}
$dumbo = new Slon('dumbo',2);  //1

?>
```

//1 Tworzymy obiekt klasy Slon, klasa ta posiada konstruktor który wywolywany jest automatycznie, argumenty podajemy w nawiasach.

//2 Konstruktor przypisuje wartosc do właściwości $imie, następnie //3 tworzy obiekt klasy Traba, wywolujac jej konstruktor.

Domyślny konstruktor jest to konstruktor nie pobierający argumentów lub którego wszystkie argumenty są domiemane, podczas tworzenia obiektu możemy pominąć nawiasy.

```
<?php
class Domyslny{
    private $cecha1;
    private $cecha2;
    public function __construct($cecha1=0, $cecha2=10){
    $this->cecha1 = $cecha1;
    $this->cecha2 = $cecha2;
    }
}
$obiekt1 = new Domyslny; // mozemy pominac nawiasy
$obiekt2 = new Domyslny(2,2); // badz wprowadzic inne
                              // wartosci niz domyslne
 ?>
```

Przeciążanie konstruktorów

W PHP nie ma możliwości przeciążania konstruktorów -jednak można osiągnąć podobny rezultat.

Przykładowo tworzymy obiekt klasy odcinek podając współrzędne punktu końcowego. Bądź podając długość odcinka, w takim przypadku odcinek leżeć będzie na osi OX.

```
<?php
```

```
class odcinek{
private $koniec;
function __construct(){
if(func_num_args() == 2){
    $koniec = array(
                'x' => func_get_agr(0),
                'y' => func_get_agr(1),
                );
    }
elseif(func_num_args() == 1) {
    $koniec = array(
                'x' => func_get_agr(0),
                'y' => 0,
                );
    }
  }
}
?>
```

Destruktor

Destruktor jest to metoda wywoływana automatczynie gdy obiekt danej klasy jest usuwany.

Metoda destruktora nosi nazwę _destruct(), nie może ona pobierać argumentów ani zwracać.

Klasa nie musi posiadać destruktora, jest on wymagany gdy obiekt kończąc działanie zwalnia zarezerwowane zasoby, bądź informuje o informuje o tym fakcie.

Obiekty są kasowne przy wyjściu z zakresu (w tm przypadku na końcu skryptu). Obiekt można kasować samodzielnie przy użyciu funkcji unset(). Pobiera ona jako argument referencję do obiektu który chcemy skasować.

Gdy obiekt ginie, giną wraz z nim obiekty agregowane. Najpierw wywoływany jest destruktor obiektu a następnie destruktory obiektów agregowanych.

```
<?php
class DomekZkart{
    function __destruct(){
        echo 'runal domek  z kart';
    }
}
$domek = new DomekZkart;
unset($domek);
?>
```

Po wywołaniu powyższego skryptu ujrzymy na ekranie napis - 'runal domek z kart'

Praktyka

W lekcji "Klasy i obiekty" w części "Praktyka" stworzyliśmy klasę odpowiedzialną za połączenie z bazą danych. Niestety za każdym razem gdy ją tworzymy musimy wykonywać metodę connect, a na zakończenie pracy musimy pamiętać o disconnect. Teraz gdy

poznaliśmy konstruktor i destruktor możemy sobie to ułatwić. Najprościej będzie poprostu zmienić nazwy metod: z connect na __construct i z disconnect na __destruct.

```
<?php
class db_mysql {
    private $connect   = NULL;
    private $limit      = 10;

    public function __construct($host, $login, $pass)
    {
        $this->connect = mysql_connect(
                                $host,
                                $login,
                                $pass);
        if ($this->connect)
           mysql_select_db($database, $this->connect);
    }

    public function __destruct()
    {
        if(is_resource($this->connect)) {
            mysql_close($this->connect);
        }
    }

    public function selectAll($table, $limit = null, )
    {
        if (!$this->connect) return false;

        if ($limit === null || $limit > $this->limit)
        {
            $limit = $this->limit;
        }
        $result = (
        mysql_query(
         "SELECT * FROM ".$table." LIMIT ".$limit
        ));
        while ($row = mysql_fetch_assoc($result)) {
            $a[] = $row;
               }
        return $a;
    }
};

$mysql = new db_mysql('localhost', 'test', 'root', '1234');
print_r($mysql->selectAll('answer', 30));
?>
```

Proszę zwrócić uwagę jak teraz wygląda wykorzystanie naszej klasy. Zamiast 3 linijek mamy wszystko w 1. Na tym przykładzie widzimy z tego małe korzyści, ale przy większych projektach takie rozwiązanie jest bardzo wygodne. Pewna część czynności jest wykonywana automatycznie, nie musimy o nich pamiętać. Tworząc obiekt automatycznie łączymy się z bazą, a gdy go usuwamy to następuje rozłączenie z bazą. Zmniejsza to ryzyko wystąpienia błędu.

Zadania

Napisz klasy Gracz i Punktacja.

Cechy klasy punktacja:

- klasa przechowuje ilość punktów
- klasa posiada konstruktor, który domyślnie ustawia punkty na zero, dodatkowo potrafi aktualizować (dodawać, odejmować) punkty i podawać ich ilość
- Destruktor podający końcową punktacje

Cechy klasy Gracz:

- każdy gracz posiada punktacje, imie i wzrosct
- konstruktor
- destruktor podający imie i wzrost gracza
- Metodę aktualizującą punktacje

Schowaj rozwiązanie

```
<?php
class Punktacja{
// $punkty przechowuje punktacje
    private $punkty;
// konstruktor domyślnie pobierający wartość 0,
// ustawia początkową punktacje
    public function __construct($ilosc=0){
        $this->punkty = $ilosc;
    }
// desruktor, podaje ilość punktów
    public function __destruct(){
      echo 'koncowa ilosc punktow to: '.$this->punkty;
    }
// metoda aktualizująca ilość punktów
    public function aktualizuj($ilosc){
        $this->punkty += $ilosc;
    }
}
class Gracz
{
    private $imie;
    private $wzrost;

    // właściwość $punkty przechowywać będzie obiekt
    // klasy Punktacja
    private $punkty;

    // konstruktor, ustawia imie i wzrost
    // tworzy obiekt Punktacja poprzez konstruktor
    public function __construct($imie, $wzrost){
        $this->imie = $imie;
        $this->wzrost = $wzrost;
        $this->punkty = new Punktacja;
    }

    // destruktor podający imie i wzrost
   public function __destruct(){
       echo 'To byl gracz '.$this->imie.
```

```
            ' o wzroscie '.$this->wzrost;
   }

 /* metoda aktualizuj wywołująca analogiczną
    metodę dla obiektu klasy Punktcja */
   public function aktualizuj($ilosc) {
       $this->punkty->aktualizuj($ilosc);
   }
}

?>
```

Referencje - kopiowanie obiektów

Operator **new** tworzy obiekt danej klasy.Jako efekt zwraca referencję, czyli adres w pamięci nowo utworzonego obiektu.

Wykonanie kodu :

```
<?php
class Klasa{
    public $nazwa;
}
$pierwszyObiekt = new Klasa; //1
$drugiObiekt->nazwa = 'pierwsza';
$drugiObiekt = $pierwszyObiekt; //2
$drugiObiekt->nazwa = 'druga'; //3
echo '$nazwa w $pierwszyObiekt
        = '.$pierwszyObiekt->nazwa;
unset($druaKlasa);
?>
```

Na ekranie ujrzymy :

$nazwa w $pierwszyObiekt = druga;

- //1// Wywolanie powyzszego kodu spowoduje przypisanie referencji nowo utworzonego obiektu do zmiennej $pierwszyObiekt.
- //2// Następnie zmiennej $drugiObiekt przypisujemy wartość zmiennej $pierwszyObiekt. Jako że $pierwszyObiekt przechowuje referencje(adres) do obiektu, do zmiennej $drugiObiekt zostaje przypisany ten sam adres.
- //3// Jakakolwiek zmiana poprzez zmienną $pierwszyObiekt pociąga zmianę także w $drugiObiekt. Dzieje się tak ponieważ zmienne przechowują referencję do tego samego obiektu.
- //4// Tak więc funkcja unset(), wywolana z zmienna $drugiObiekt,usuwa wskazywany obiekt. Odtąd $pierwszyObiekt i $drugiObiekt wskazują na NULL, nieprzydzielone miejsce w pamięci.

Funkcje a obiekty

Gdy przesyłamy jako argument do funkcji zmienną zawierającą referencję do obiektu klasy jest ona automatycznie przesyłana przez referencję.

Tak więc w ciele funkcji działamy na oryginalnym obiekcie a nie na jego kopii.

```
<?php class Klasa{
    public $wartosc;
}
//////////////////////////
function zmieniam($klasa){
$klasa->wartosc = 20;
}
//////////////////////////
$klasa = new Klasa;
zmieniam($klasa);
echo $klasa->wartosc;
?>
```

Na ekranie ujrzymy :

20

Kopiowanie obiektów

Do kopiowania obiektów służy polecenie clone. Używamy go gdy chcemy przypisać do zmiennej kopie obiektu, zamiast jego referencji, lub gdy do funkcji przesyłamy kopie obiektu, nie chcąc by ta pracowała na oryginale i miała możliwość zmian.

Polecenie copy tworzy lustrzaną kopię obiektu a następnie, wykonuję funkcję __clone() jeśli takowa jest zdeklarowana w klasie. Następnie przesyła referencję do nowo utworzonego obiektu do zmiennej bądź do funkcji.

```
<?php
class Klasa1{
    public $wartosc;
    function __construct($wartosc){
         $this->wartosc = $wartosc;
    }
    function __destruct(){
        echo 'gine, moja wartosc podczas
            konania to '.$this->wartosc;
    }

}
class Klasa2{
    public $wartosc;
    function __construct($wartosc){
         $this->wartosc = $wartosc;
    }
    function __clone(){
        $this->wartosc += 20;
    }
```

```
}
function zmien($klasa){
 $klasa->wartosc *= -1;
}
$zmienna=new Klasa1(10); //1
zmien(clone $zmienna); //2
echo 'wartosc oryginalu po wywolaniu funkcji
     '.$zmienna->wartosc.'<br />'; //3
$zmienna2 = new Klasa2(0);//4
$kopia = clone $zmienna2; //5
echo '$wartosc w kopii = '.$kopia->wartosc; //6
?>
```

Na ekranie zobaczymy :

gine, moja wartosc podczas konania to -10
wartosc oryginalu po wywolaniu funkcji 10
$wartosc w kopii = 20
gine, moja wartosc podczas konania to 10

- //1// Tworzymy obiekt klasy Klasa1, konstruktor ustawia $wartosc na 10
- //2// Następnie przesyłay kopie obiektu, do funkcji zmien, funkcja ta zmienia $wartosc na liczbe przeciwna. Po wykonaniu ciala funkcji, kopia wychodzi z bieżącego zakresu więc jest niszczona. Uruchamiany jest destruktor ktory wypisuje $wartosc = - 10
- //3// Widzimy że pomimo wykonania funkcji oryginal nie zostal zmieniony, dzialalismy na kopii
- //4// Tworzymy obiekt klasy Klasa2, konstruktor inicjuje $wartosc liczba 0
- //5// Następnie przypisujemy zmiennej $kopia kopie obiektu, ponieważ klasa posiada metodę, __clone() jest ona uruchamiana.
- //6// Widzimy efekt działania metody clone, nie nasąpiła lustrzana kopia lecz $wartosc z oryginalu zostala zmodyfikowana o liczbe 20
- Po zakończeniu skryptu, ginie obiekt pokazywany przez $zmienna, wykonywany jest jej destruktor.

Agregaty raz jeszcze

Gdy wywołujemy clone dla klasy zawierającej obiekty innych klas (agregaty), należy pamiętać by w metodzie __clone() wywołać polecenie clone dla obiektów zawieranych, w innym wypadku kopia będzie posiadać referencję do obiektów zawieranych w oryginale.

```
<?
class Bateria{
    //cialo klasy
}
class Pilot{
    public $bateria = new Bateria;
    function __clone(){
        $this->bateria = clone $this->bateria;
    }
}
?>
```

Jak to działa ? Najpierw wykonywana jest dokładna kopia obiektu, właściwość po właściwośći, tak więc baterii, jako że przechowuje referencję do obiektu przypisany zostanie taki sam adres jak w obiekcie kopiowanym. Następnie rusza do walki metoda __clone(), która bateri przypisuje jej kopie. Gdybyśmy nie napisali metody __clone, wyczerpanie baterii w kopiowanym obiekcie niosło by za sobą wyczerapnie baterii w kopii i odwrotnie. Pomyśl co by się stało gdyby któryś z obiektów został usunięty...

Operator ===

PHP rozróżnia obiekty za pomocą ich jednoznacznych ID, które są nadawane automatycznie podczas ich tworzenia.

Za pomocą operatora === możemy spradzić czy dane zmienne przechowują referencję do tych samych obiektów. Operator ten porównuje ID obiektów.

```
<?php
$zmienna = new Klasa;
$zmienna2 = $zmienna;
echo (int)$zmienna2 === $zmienna.'<br />';
$zmienna2=clone $zmienna;
echo (int)$zmienna2 === $zmienna.'<br />';

?>
```

Na ekranie ujrzymy :

1
0

Zadania

Napisz klasę Ubranie zawierającą zawierającą poniższe cechy:

- każde ubranie posiada swoją nazwę, jak i jakość, czym numer jakości wyższy tym jakość słabsza
- każde ubranie można podrobić za pomocą specjalnej metody, efektem tego powstaje kopia posiadająca jakość słabszą od wzorca. dodatkowo każde ubranie posiada informację ile jego kopii zostało wykonanych
- konstuktor, za jego pomocą tworzymy oryginalne Ubrania
- prywatną metodę clone
- ubranie posiada metodę wypisujące informacje o nim, podaje czy jest to podróbka, nazwę, jakość i ilość kopii

Podpowiedź - podrobione ubrania możesz tworzyć za pomocą metody pobierającej zmienną i przypisującą jej kopie obiektu dla którego została wywołana. Jako metoda klasy ma dostęp do prywatnej metody clone.

Schowaj rozwiązanie

```
<?php
```

```
class Ubranie{
    private $ilosc_kopi;
    private $jakosc;
    private $nazwa;
     public function __construct($nazwa){
        // konstruktor nadaje nazwe, ustala jakosc
        // na pierwsza i zeruje ilosc kopii
        $this->nazwa = $nazwa;
        $this->jakosc = 1;
        $this->ilosc_kopi = 0;
    }
    private function __clone(){
        //metoda clone, zwiększa jakość o 1
        // zeruje ilosc_kopi
        ++$this->jakosc;
       $this->ilosc_kopi = 0;

    }

    public function podrobka(&$zmienna){
        /* metoda podrobka, musi pobierac zmienna,
        poprzez referencje w przeciwnym wypadku
        przypisalibysmy kopie kopii zmiennej
        ktora zginelaby po wykonaniu metody */

        $obiekt = clone $this;
        //po stworzeniu kopi,
        //zwiekszamy ilosc_kopi obiektu kopiowanego
        ++$this->ilosc_kopi;
    }
    public function przedstaw(){
        // metoda przedstaw sie,
        // w razie podrobki wypisuje Podrobka
        if( $this->jakosc > 1 ){
            echo 'Podrobka ';
        }
        // nastepnie podawana jest nazwa,
        // jakosc i ilosc kopii
        echo $this->nazwa.' '.$this->jakosc.' jakosci '.
        ' skopiowany '.$this->ilosc_kopi.' razy';

    }

}
?>
```

Static i stałe

Właściwości statyczne są to właściwości nadawane dla całej klasy, nie dla poszczególnych obiektów.

Właściwość statyczną tworzymy przy pomocy modyfikatra **static**

Jako że właściwość statyczna dana jest dla klasy nie dla obiektu, chcą ją wywołać w metodzie klasy nie możemy posłużyć się **$this**, używamy modyfikatora **self::**

Chcąc wywołać właściwość poza klasą używamy modyfikatora **nazwaKlasy::**

```
<?php
class Licznik{
    public static  $ilosc = 0;
    public function __construct(){
        self::$ilosc++;
    }

}
echo '$ilosc ='.Licznik::$ilosc.'<br />';
$pierwsza = new Licznkik;
$druga = new Licznik;
$trzecia = new Licznik;
echo '$ilosc = '.Licznik::$ilosc;
?>
```

Metody statyczne

Metody statyczne są to metody nadawane dla całej klasy, nie dla poszczególnych obiektów.

Metodę taką tworzymy przy pomocy modyfikatra **static**

Metodę taką możemy wywołać choć nie istnieje żaden obiekt klasy

W ciele klasy wywołujemy ją używając modyfikatora **self::**

Chcąc wywołać metodę statyczbą poza klasą używamy modyfikatora **nazwaKlasy::**

```
<?php
class Licznik{
    private static  $ilosc = 0;
    public static function ile(){
        echo 'Jest stworznych'.self::$ilosc.'
        obiektów klasy licznik';
    }
    public function __construct(){
        self::$ilosc++;
    }

}
Licznik::ile();
$pierwsza = new Licznkik;
$druga = new Licznkik;
$trzecia = new Licznik;
Licznik::ile();
?>
```

Stałe

Stała jest to właściwość klasy tylko do odczytu. Oznacza to że jej wartość deklarujemy już w definicji klasy, wartość nie może być modyfikowana.

Deklarujemy je używając modyfikatora const. Stałe zawszę są one dostępne public

Chcą wywołać stałą w metodzie klasy używamy modyfikatora **self::**

Chcąc wywołać stałą poza klasą używamy modyfikatora **nazwaKlasy::**

```
<?php
class Okrag{
    const pi = 3.14;
    private $promien;
    public function __construct($promien){
        $this->promien = $promien;
    }
    public function obwod(){
        return 2*$this->promien*self::pi;
    }

}
// Okrag::pi = 40; // nie mozna modyfikowac
$dane=new Okrag(2);
echo $dane->obwod();
echo Okrag::pi;

?>
```

Zadania

Napisz klasy Silnik i Samolot

Klasa Silnik zawiera:

- stałą moc dla wszystkich silników

Cechy klasy Samolot

- klasa ta potrafi podać ile samolotów zostało utworzonyc
- każdy samolot posiada określoną ilość silników od 0 do 4, za pomocą metody informuje ile ich posiada
- Prywatny konstruktor który jako jeden z arumentów przyjmuje ilość funduszy, na podstawie tej liczby tworzy samolot o maxymalnej możliwej ilości silników. Przy czym - Ceny silników:
    - 1000 - 1 silnik
    - 2000 - 2 silniki
    - 3000 - 3 silniki
    - 4000 - 4 silniki

    Koszt zakupu silników odejmowany jest od funduszy.

Podpowiedź : w przypadku prywatnego konstruktora, obiekty danej klasy tworzyć można za pomocą metody statycznej.

Schowaj rozwiązanie

```
<?php
```

```
class Silnik{
    const moc = 500;
    }
class Samolot{
// statyczna właściwość przechowująca ilosc powstalych
// samolotow na początku wynosi 0
    private static $ilosc = 0;

// właściwość przechowująca ilość silników
//danego samolotu
    private $ilSilnikow;

// właściwość przechowująca silniki,
// obiekty klasy Silnik
    private $silniki = array();

// statyczna metoda podająca ilość samolotów
    public static function podajIlosc(){
        return self::$ilosc;
    }

// konstruktor pobiera ilość silników do utworzenia,
// oraz zwiększający ilość Samolotów
        private function __construct($ile){
        $this->ilSilnikow = $ile;
        for($i = 0; $i < $ile; ++$i ){
            // dodawanie silnikow
            $this->silniki[$i] = new Silnik;
        }
        // zwiekszanie ilosci Samolotow
        ++self::$ilosc;

// destruktor zmniejszajacy ilosc samolotow
    public function __destruct(){
       --self::$ilosc;
    }
// metoda podająca ilość silników danego samolotu
    public function ileSilnikow(){
        return $this->ilSilnikow;
    }

/* statyczna metoda służąca do tworzenia obiektów
  klasy Samolot pobiera zmienna poprzez referencję do
  której przypisany zostanie nowo powstały obiekt
  pobiera także ilość funduszy,
  również poprzez referencje */
    public static function stworz(&$zmienna,&$fundusz)
    {
    /* na podstawie ilości funduszy wywoływany jest
     konstruktor  z opowiednią ilością silników do
     utworzenia.
     Dodatkowo koszt zamontowania silnika odejmowany */

        if($fundusz >= 1000){
            if($fundusz < 2000){
                $fundusz -= 1000;
                $zmienna = new Samolot(1);
            }
            elseif($fundusz < 3000){
                $fundusz -= 2000;
                $zmienna = new Samolot(2);
```

```
            }
            elseif($fundusz < 4000){
                $fundusz -= 3000;
                $zmienna = new Samolot(3);
            }
            else{
                $fundusz -= 4000;
                $zmienna = new Samolot(4);
            }
        }
        else{
            echo 'Nie masz wystarczajacych funduszy ';
        }
    }



}

?>
```

Dziedziczenie

Dziedziczenie służy do tworzenia nowych klas na podstawie innych, rozszerzając ich możliwości. Wykorzystywać będziemy następującej nomenklatury:

- Klasa pochodna - klasa która dziedziczy.
- Klasa rodzica - klasa będąca dziedziczona

Wyobraź sobię klasę jako worek, worek zawierający metody i właściwości, dziedzicząc klasę, przesypujemy właściwości i metody z klasy dziedziczonej.

Do odziedziczonych elementów public i protected, mamy taki sam dostęp jak, w klasie rodzica. Tutaj właśnie objawiają się cechy dostępu protected, elementy takie są dostępne w klasie zawierającej jak i klasach pochodnych.

Elementy private są również dziedziczone, jednak do nich nie uzyskamy bezpośredniego dostępu, pośrednio odwołać możemy się jedynie za pomocą odziedziczonych metod.

PHP obsługuje dziedziczenie proste, oznacza to że dana klasa może dziedziczyć jedynie po jednej klasie. Jednak klasa rodzica także może być pochodną(dziedziczyć) od klasy na wyższym szczeblu. Nazywamy to dziedziczeniem wielopoziomowym.

Przypuśćmy że mamy klasę pradziadek po której dziedziczy klasa dziadek, po niej dziedziczy ojciec a po nim syn. Mówimy że ojciec jest wyżej w hierarhii dziedziczenia, niż syn.

Klasa potomna może dodawać nowe metody i właściwości, jak także przesłaniać te odziedziczone od rodzica. Przesłaniać to znaczy tworzyc metody o takiej samej nazwie jak te w klasie pochodnej.

Dziedziczenie nadajemy za pomocą słowa kluczowego **extends**

```php
<?php
class Kubek{
    protected $kolor;
    public function ustawKolor($kolor){
        $this->kolor = $kolor;
    }
    public function podajKolor(){
        return $this->kolor;
    }
    private function wyslizgnijSie(){
        echo 'OOPS !';
    }
    public function przedstawSie(){
        return 'Jestem kubek mój kolor '.$this->kolor;
    }
}
class KubekUcho extends Kubek{ //1
    protected $kolorUcha;
     public function ustawKolorUcha($kolor){
        $this->kolorUcha = $kolor;
    }
    public function podajKolorUcha(){
        return $this->kolorUcha;
    }
    public function przedstawSie(){
        //$this->wyslizgnijSie(); //7
        return 'Jestem kubek z uchem moj kolor
         '.$this->kolor.' a ucho '.$this->kolorUcha;
    }

}
class KubekUchoZaroodporny{
// nowe metody i właściwości
}
$zmienna = new KubekUcho; //2
//$zmienna->kolor //3
$zmienna->ustawKolor('czerwony'); //4

echo $zmienna->przedstawSie(); //6

?>
```

W powyższym kodzie rozszerzamy klase Kubek, tworząc klasę KubekUcho. Nowa klasa dziedziczy właściwości i metody z klasy pochodnej,dodaje swoje właściwości i metody przesłaniając przy tym metodę przedstawSie. Następnie rozszerzamy klasę KubekUchoZaroodporny, klasa ta dziedziczy metody i właściwości od klasy KubekUcho tak więc pośrednio dziedziczy też metody i właściwośći klasy Kubek.

- //1 to tutaj deklarujemy dziedziczenie, podajemy nazwę nowej klasy, a po słowie kluczowym extends nazwę klasy z której dziedziczymy.
- //2 tworzymy obiekt klasy KubekUcho
- //3, musieliśmy zakomentować tą komendę ponieważ do elementów z modyfikatorem protected dostęp mamy jedynie w metodach klasy zawierającej bądź w metodach klas pochodnych
- //4 dokonujemy ustawienia koloru za pomocą odziedziczonej metody

- //5 ustawiamy kolorUcha
- //6 wywołujemy metodę przedstaw się, klasa pochodna zasłania wersję metody odziedziczonej
- //7 ten fragment kodu musieliśmy zakomentować gdyż klasa nie ma bezpośredniego dostępu do elementów odziedziczonych prywatnych.

UWAGA - WYWOŁANIE KODU

```
<?php
class Rodzic{
    private $wartosc;
}
class Dziecko extends Rodzic{

}
$zmienna = new Dziecko;
$zmienna->wartosc = 40;
?>
```

Nie spowoduje błędu. Zostanie dynamicznie utworzona właściwość $wartosc wewnątrz obiektu wskazywanego przez $zmienna.

Konstruktory, destruktory a dziedziczenie

PHP wykonuje zawsze pierwszy znaleziony konstruktor to znaczy :jeśli w klasie nie ma konstruktora, wywołany zostanie konstruktor klasy rodzica, jeśli rodzic takiego nie posiada wywołany zostanie kostruktor klasy jego rodzica i tak aż do znalezienia konstruktora bądź końca hierarhii klas.

Jeżeli zostanie odnaleziony konstruktor, PHP nie będzie wywoływać konstruktorów położonych wyżej w hierarhii, oznacza to że gdy w klasie znajduje się konstruktor,nie zostanie wywołany konstruktor rodzica.

Nie zachodzi tu nic nowego, poprostu gdy w klasie zdefiniowany zostanie konstruktor, przesłania on wszystkie odziedziczone konstruktory.

Analogicznie ma się sprawa z destruktorami.

```
<?php
class Dziadek{
protected $nazwisko;
public function podajNazwisko(){
echo 'nazywam sie '.$this->nazwisko;
}
public function __construct($nazwisko){
   $this->nazwisko = $nazwisko;

}}
class Ojciec extends Dziadek{}
class Syn extends Ojciec{}
//$piotr = new Syn;
$piotr = new Syn('Mak');
```

```
$piotr->podajNazwisko();

?>
```

Klasa Syn, dziedziczy konstruktor z klasy Dziadek, tak więc nie możemy stworzyc obiektu nie podając argumentu.

Modyfikator dostępu

Przesłonięte metody możemy wywoływać za pomocą modyfikatora parent:: . Dzięki temu możemy np wywołać konstruktor, destruktor rodziców.

Przykład z kubkami z wykorzystaniem nowych wiadomości.

```
<?php
class Kubek{
    private $kolor;
    public function __construct($kolor){
         $this->kolor = $kolor;
    }
    public function  __destruct() {
        echo 'zniszczony kubek';
    }
    public function przedstawSie(){
        return 'Jestem kubek moj kolor '.$this->kolor;
    }
}
class KubekUcho extends Kubek{
    protected $kolorUcha;
    public function  __construct($kolor,$kolorUcha) {
        parent::__construct($kolor);//1
        $this->kolorUcha = $kolorUcha;
    }
    public function podajKolorUcha(){
        return $this->kolorUcha;
    }
    public function przedstawSie(){
        return parent::przedstawSie(). //2
               ' a ucho '.$this->kolorUcha;
    }

}

?>
```

- //1// wywołujemy konstuktor rodzica
- //2// wywołujemy przesłoniętą funkcję

Modyfikator final

Modyfikator ten oznacza że w klasie znajduje się końcowa wersja funkcji, oznacza to że w funkcjach pochodnych nie wolno przesłaniać danej funkcji.

Gdy użyjemy modyfikatora final w stosunku do klasy, oznacza to że dana klasa nie może być dziedziczona.

```
<?php
final class Ostateczna{}
class Rodzic{
    final public function ostateczna(){

    }
}
class Dziecko extends Rodzic{
    //public function ostateczna(){}
}
//class Wredna extends Ostateczna{}

?>
```

Praktyka

Poniżej przedstawię ciekawy przykład wykorzystania dziedziczenia. Zaprojektuję i przedstawię przykładową implementację klas do tworzenia formularzy HTML.

Główną klasą będzie *form_tag*. Każdy element formularza jak i sam formularz może posiadać swoje atrybuty. Najczęściej używane to style i class. Nasza klasa zawierać będzie:

Wartości:

- attr - tablica zawierająca atrybuty (klucz - nazwa atrybutu => wartość)
- style - tablica zawierająca style css (klucz - nazwa stylu => wartość)
- class - tablica zawierająca nazwy clas do jakich należy element

Metody:

- css - posłuży do dodawania styli
- removecss - usunie wszystkie style, lub podaną
- addClass - doda clasę
- removeClass - usnie wszystkie clasy lub podaną
- attr - doda atrybuty
- removeAttr - usnie wszystkie atrybuty lub podany
- renderAttr - wygeneruje i zwróci kod atrybutów

Kolejną klasą jest klasa form, będzie ona tworzyła formularz

Wartości:

- element - tablica elementów formularza

Metody:

- __toString - metoda magiczna zwróci to samo co render
- action - ustawia atrybut action

- method - ustawia atrybut method
- element - dodaje elementy lub zwraca podany element
- removeElement - usuwa elementy
- render - generuje i zwraca kod

Klasa form_element zawierać będzie części wspólne dla elementów formularza takie jak nazwa itp.

Klasy poszczególnych elementów (np. text, textarea) zawierać będą funkcje renderowania.

Foreach a obiekty

Za pomocą pętli foreach możemy przejść kolejno przez właściwości klasy.

Pętla foreach wywołana poza ciałem klasy przejdzie jedynie przez publiczne właściwości klasy.

Chcąc przejść przez wszystkie właściwości musimy umieścić pętle foreach wewnątrz metody klasy.

```
<?php
class Wypisz{
    public $imie;
    public $nazwisko;
    public $wiek;
    private $tajne;
    protected $tajne2;
    public function __construct(
          $imie,$nazwisko,$wiek,$tajne,$tajne2)
    {
        $this->imie = $imie;
        $this->nazwisko = $nazwisko;
        $this->wiek = $wiek;
        $this->tajne = $tajne;
        $this->tajne2 = $tajne2;
    }

    public function przejrzyj(){
        echo '<br>wypis dzieku metodzie';
        foreach($this as $klucz => $wartosc){
           echo $klucz.' = '.$wartosc;
        }
    }
}
$zmienna=new Wypisz('Pawel','Kowal',20,7806,'%#');
foreach($zmienna as $klucz => $wartosc){
           echo $klucz.' = '.$wartosc;

           }
$zmienna->przejrzyj();
?>
```

Foreach a dziedziczenie

Jak pamiętasz, elementy private podczas dziedzicznie nie są dostępne bezpośrednio. Tak więc wywołanie metody wypisz klasy Syn, wypisze jedynie właściwości $Pro i $Pub;

```
<?php
class Ojciec{
    protected $Pro = 'protected';
    private $Pri = 'private';
    public $Pub = 'public';
}
class Syn extends Ojciec{
    public function wypisz(){
        foreach($this as $wlasciwosc => $tresc){
            echo $wlasciwosc.' '.$tresc;
        }
    }
}
?>
```

Klasy abstrakcyjne i interfejsy

Klasa abstrakcyjna jest to klasa której obiekty nie mogą być tworzone,może być natomiast dziedziczona.

Klasa abstrakcyjna może posiadać konstruktor jak i destruktor, mogą być one jednak wywołane tylko przez klasy pochodne.

Klasę abstrakcyjną tworzymy przy pomocy modyfikatora **abstract**

Dodatkowo klasa abstrakcyjna może posiadać funkcje abstrakcyjne, funkcja taka posiada liste argumentów, jednak nie posiada ciała.

Funkcje abstrakcyjne muszą zostać nadpisane w klasach pochodnych.

Tworzone są one również modyfikatorem **abstract**

```
<?php
abstract class Zwierze{
    protected $imie;
    public function __construct( $imie ){
        $this->imie = $imie;
    }
    final public function  podajImie(){
        return $imie;
    }
    abstract public function wydajOdglos();

}
class Slon extends Zwierze{
    public function __construct( $imie ){
        parent::__construct($imie);
    }
    public function wydajOdglos(){
```

```
        echo 'Turr-uuu';
    }
}
?>
```

Stworzyliśmy klase abstrakcyjną Zwierze która posiada konstruktor, jedną funkcje finalną i funkcję abstrakcyjną. Klasa Slon ktora dziedziczy klase abstrakcyjna Zwierze nadpisuje funkcje abstrakcyjna(musi to zrobic), jak i wywoluje konstruktor klasy abstarcyjnej w ciele swojego konstruktora. Nalezey pamietac ze klasa Slon nie moze nadpisac funkcji podajImie.

Interfejsy

Interfejs jest jakby mocnie abstrakcyjną klasą, może mieścić jedynie metody, które z założenia są abstrakcyjne.

Metody w interfejsie są domyślnie abstrakcyjne tak więc, wszystkie metody zawarte w interfejsie muszą zostać nadpisane w klasach pochodnych.

Przy pomocy interfejsów masz możliwość wymuszenia na klasach, aby posiadały niezbędne do współpracy z innymi elementami metody, tak więc intersejsy zapewniają określone cechy klasy.

Interfejs tworzymy słowem kluczowym **interface**

Klasy nie dziedzicza interfejsów, klasy je implemetują, dokonujemy tego używając modyfikatora **implements** po którym podajemy nazwy interfejów do zaimplementowania.

Do danej klasy możemy implementować dowoln± ilość interfejsów.

Interfejsy mogą dziedziczyć inne interfejsy.

Interfejsy nie mogą dziedziczyć ani implementować klas.

```
<?php
interface miesozerne{
    function zjedzMieso($mieso);
}
interface roslinozerne{
    function zjedzWarzywo($warzywo);
}
class Slon implements miesozerne,roslinozerne{
    //rozne wlasciwosci i metody
    function zjedzMieso($mieso){
        echo 'Ja slon zjadam '.$mieso;
    }
    function zjedzWarzywo($warzywo){
        echo 'Ja slon zjadam '.$warzywo;
}
}
interface wszystko extends miesozerne, roslinozerne {

}
```

```
?>
```

Stworzylismy dwa interfejsy, miesozerne i roslinozerne. Interfejsy te posiadają odpowiednio metody zjedzMieso jak i zjedzWarzywo. Tak więc klasy implementujące je muszą te metody nadpisac. Widzimy że klasa Slon implementuje interfejs miesozerne jak i roslinozerne, taki sam efekt moglibysmy otrzymac implementując interfejs wszystkozerne ktory dziedziczy po roslinozerne i miesozerne.

Różnice między klasami abstrakcyjnymi a interfejsami

- W interfejsach wszystkie metody są abstrakcyjne, natomiast w klasie abstrakcyjnej można stworzyc metody posiadające ciało, jak i abstrakcyjne.
- W php można dzidziczyć jedynie po jednej klasie, natomiat interfejsów, można implementować wiele. Ponadto interfejsy mogą dziedziczyc wiele interfejów
- Klasa abstrakcyjna zazwyczaj jest mocno związana z klasami dziedziczącymi w sensie logicznym, czyli np. tworzymy klasę abstrakcyjną Planeta po której dziedziczą konkretne klasy planet (np Ziemia, Mars). Interfejs natomiast nie musi być już tak mocno związany z daną klasą, on określa jej cechy, np możesz stoworzyć interfejs Zniszczalny, który mówi że dany obiekt może zostać zniszczony. Taki interfejs możesz nadać zarówno klasą Planeta, Gwiazda, Budynek itp.

Zadania

Napisz klasę abstrakcyjną Tabela i dziedziczącą po niej klasę TabelaHTML

Cechy klasy Tabela

- posiada wysokość i szerokość
- na podstawie wysokości i szerokości tworzy tablicę wielowymiarową o rozmiarach - wysokość na szerokość, wstępnie wszystie elementy posiadają wartość 0
- posiada abstrakcyjną metodę rysujSie
- posiada finalną metodą służącą do wpisywania wartości w odpowienim miejscu w tabeli.

Cechy klasy TabelaHTML

- dodaje ceche - szerokość kolumny
- nadpisuje metodę rysujSie, rysując tabelę o zadanych rozmiarach, w odpowiednich polach wpisując odpowiednie wartości z tablicy

Podpowiedź - chcąc narysować tabelę o rozmiarach 2 x 3 i szerokości każdej z kolumn 17px użyjemy kodu:

```
<table>
        <tr>
                <td style="width:17px" >komórka1</td>
                <td style="width:17px">komórka2</td>
                <td style="width:17px">komórka3</td>
        </tr>
        <tr>
```

```
                <td style="width:17px">komórka1</td>
                <td style="width:17px">komórka2</td>
                <td style="width:17px">komórka3/td>
        </tr>
</table>
```

Schowaj rozwiązanie

```
<?php
abstract class Tabela
{
protected $wysokosc;
protected $szerokosc;

protected $tablica = array();
protected function __construct($szerokosc, $wysokosc )
{
    $this->szerokoscKomorek = $szerKom;
    $this->szerokosc = $szerokosc;
    $this->wysokosc = $wysokosc;

    // tworzymy tymczasową tablicę, będzie ona rozmiaru
    // takiego jak przesłana szerokość
    $tymczasowa = array();
    for($i = 0; $i < $szerokosc; ++$i){
        // do każdego elementu wpisujemy wartość 0
        $tymczasowa[]= 0;
    }

    /* to tutaj tworzyć będziemy naszą
    tablicę dwuwymiarową,
    tworzymy tablicę o ilości elementów
    jak przesłana wysokość,
    do każdego elementu dopinamy tablicę tymczasową
    wynikiem tego powstanie tablica o wymiarach:
    $wysokosc x $szerokosc */
    for($j = 0; $j < $wysokosc; ++$j ){
        $this->tablica[] = $tymczasowa;
    }

}

// abstrakcyjna metoda rysujSie
abstract public function rysujSie();

// finalna metoda wpiszPole, za jej pomocą wpisujemy
// daną wartość do określonego elementu tablicy
final public function wpiszPole($y, $x, $element){
    $this->tablica[$y][$x] = $element;
}
}

class TabelaHTML extends Tabela
{
protected $szerokoscKomorek;

// w konstruktorze wywołujemy konstruktor rodzica
// po czym ustawiamy szerokoscKomorek
public function __construct(
            $szerokosc, $wysokosc , $szerKom)
```

```
{
    parent::__construct($szerokosc, $wysokosc);
    $this->szerokoscKomorek = $szerKom;
}

    /* nadpisana metoda rysujSie, gdyby nasza klasa
    nie posiadala tej metody, podczas uruchomienia
    nastapilby blad gdyż musimy
    nadpisac metode abstakcyjna */
public function rysujSie()
{
    echo '<table>';
    // główną pętlą budować będziemy wiersze
    for($i = 0 ; $i < $this->wysokosc; ++$i){
        // tworzymy wiersz
        echo '<tr>';
        // tą petla tworzyc bedziemy kolumny
        // w poszczegolnym wierszu
        for($j = 0 ; $j < $this->szerokosc; ++$j){
            // tworzymy kolumne o ustalonej szerokosci
            // i wpisujemy w nią daną wartość z tablicy
            echo '<td style="width:
            '.$this->szerokoscKomorek.'px">'.
            $this->tablica[$i][$j].
            '</td>';
        }

        /* sygnalizujemy koniec wiersza, nastepnie
        wykonany zostanie kolejny obieg petli,
        wiec dopisany kolejny wiersz
        lub zakonczenie petli w wypadku narysowania
        wystarczajacej ilosci wierszy */
        echo '</tr>';

    }

// konczymy tablice
echo '</table>';
}

}
?>
```

Instanceof, argumenty funkcji

Operator instanceof sprawdza czy dany obiekt jest obiektem danej klasy lub klasy która dziedziczy po danej klasie. W wypadku powodzenia zwraca wartość logiczną true.

Za pomocą operatora instanceof sprawdzić także możemy czy obiekt jest klasy imlemenującej dany interfejs.

```
<?php
interface Interfejs{}
class Bazowa{}

class Dziedzic extends Bazowa implements Interfejs{}
```

```
$obiekt = new Dziedzic;
if( $obiekt instanceof Interfejs ){
    echo 'obiekt implementuje Interfejs<br />';
}

if( $obiekt instanceof Bazowa ){
    echo 'jest to obiekt klasy Bazowa
    lub jest on obiektem klasy która
    po niej dziedziczy<br /> ';
}

if( $obiekt instanceof Dziedzic ){
    echo 'jest to obiekt klasy Dziedzic
    lub jest on obiektem klasy która
    po niej dziedziczy ';
}

?>
```

Na ekranie ujrzymy :

obiekt implementuje Interfejs
jest to obiekt klasy Bazowa lub jest obiektem klasy która po niej dziedziczy
jest to obiekt klasy Dziedzic lub jest obiektem klasy która po niej dziedziczy

Argumenty funkcji

PHP pozwala na podawanie oczekiwanycg typów obiektów przy deklaracji argumentów funkcji.

Podczas wywołania takiej funkji PHP automatycznie sprawdzi poprawność przesłanego obiektu za pomocą operatora instanceof.

Jeśli operator zwróci wartość logiczną false, metoda nie zostanie wykonana a PHP zwróci błąd.

```
<?php
interface miesozerne{}
interface roslinozerne{}
class Slon implements miesozerne,roslinozerne{}
class Zyrafa implements roslinozerne{}
function zjedzMieso(miesozerne $zmierze){
    echo 'mniam mniam, miesko';
}
function zjedzRosline(roslinozerne $zwierze){
echo 'mniam mniam, zieleninka';
}
$zyrafa = new Zyrafa;
$slonik = new Slon;
zjedzRosline($zyrafa);
zjedzRosline($slonik);
//zjedzMieso($zyrafa);
zjedzMieso($slonik);
?>
```

//zjedzMieso($zyrafa); fragment kodu zakomentowany gdyz obiekt klasy Zyrafa nie implementuje interfejsu miesozerne.

Obsługa błędów

Wyjątki

Wyjątki powodują określone zachowanie PHP podczas napotkania błędu w bloku kodu.

Wyjątki są to obiekty klasy Exception bądź klas pochodnych od niej.

Wyjątek wywołujemy pomocą operatora **throw**

Konstruktor klasy Exception pobiera dwa opcjonalne argumenty:

- Text wyjątku
- Numer wyjątku

Wszystkie metody klasy Excepion posiadają modyfikator final, więc nie możemy ich modyfikować w klasach pochodnych. Oto ich lista:

- getCode() - metoda zwraca, numeryczny opcjonalny kod wyjątku
- getFile() - zwraca nazwę pliku w kórym wywołany został wyjątek
- getLine() - zwraca numer wiersza w którym wywołany został wyjątek
- getMassage()-zwraca opcjonalną treść wyjątku
- getTrace()- zwraca ślad stosu - tablice metod wywołanych do momentu wywołania wyjątku
- getTraceAsString()- zwraca ślad stosu jako string

Nieobsłużony wyjątek powoduje przerwanie skryptu i zwraca ślad stosu

```
<?php
throw new Exception('blad',12);
?>
```

Oto co ujrzymy w rezultacie :

Fatal error: Uncaught exception 'Exception' with message 'blad' in /homez.106/czerner/www/pawel/test.php5:2 Stack trace: #0 {main} thrown in **/homez.104/login/www/pawel/test.php5** on line **2**

Wyłapywanie wyjątków

Wyjątki wyłapujemy za pomocą bloku **try - catch**

Dzięki wyłapywaniu wyjątków jesteśmy w stanie określić jak ma się zachować kod w razie napotkania wyjątku

Składnia bloku :

```
try{
// tutaj wywolujemy wyjatki ktore beda przechwycone
}
catch(klasaWyjatku1 $nazwaWblokuCatch){
// tresc funkcji
}
catch(klasaWyjatku2 $nazwaWblokuCatch){
// tresc funkcji
}
...
catch(klasaWyjatku3 $nazwaWblokuCatch){
// tresc funkcji
}
```

- Blok kodu w którym mogą być wywoływane wyjątki oznaczany jest słowem try, ograniczony nawiasem klamrowym.
- Następnie znajdują się bloki catch, może być ich dowolna ilość, są to specjalne funkcje do obsługi wyjątków.W okrągłym nawiasie podajemy nazwę obsługiwanej klasy przez blok catch i zmienną pod która w bloku widoczny będzie obiekt wyjątku.
- Gdy pojawi się wyjątek natychmiast wychodzimy z bloku try, a następnie klasa wywołanego wyjątku poszukiwana jest w blokach catch. Gdy interpreter znajdie blok catch obługujący dany wyjątek wykonuje jego ciało, po czym wychodzi z bloku try-catch.

Przykład :

```
<?php
class FunnyException extends Exception{}
class IntException extends FunnyException{}
try{
    $zmienna = 'jasio';
    if( !is_int($zmienna) ){
        throw new IntException(
          'zmienna nie jest typu integer');
    }
    echo 'to sie nie wykona gdy wystapi wyjatek,
    interpreter od razu przejdzie do blokow catch';
}
catch(FunnyException $wyjatek){
    echo 'Stalo sie cos zlego!'.$wyjatek->getMessage();
}
catch(Exception $wyjatek){
     echo 'Standardowy wyjatek'.$wyjatek->getMessage();
}
catch(IntException $wyjatek){
    echo 'Nastapil blad typu zmiennej
         '.$wyjatek->getMessage();
}
?>
```

Powyższy kod jest poprawny, nie jest jednak poprawny logicznie

Na ekranie ujrzymy :

Stalo sie cos zlego!zmienna nie jest typu integer

Stało się tak ponieważ PHP przeszukuje bloki catch zaczynając od tego z najmniejszym numerem wiesza.Poprawność sprawdzana jest podobnie jak przy podawaniu typu obieku przy funkcjach - operatorem istanceof. Klasa IntException dziedziczy po FunnyException tak więc wykonywany jest pierwszy blok catch.

Zapamiętaj klasy w bloku catch należy podawawać począwszy od tych położonych najniżej w hierarhii dziedziczenia

Oto poprawy logicznie kod :

```
<?php
class FunnyException extends Exception{}
class IntException extends FunnyException{}
try{
    $zmienna = 'jasio';
    if( !is_int($zmienna) ){
        throw new IntException(
            'zmienna nie jest typu integer');
    }
    echo 'to sie nie wykona gdy wystapi wyjatek,
    interpreter od razu przejdzie do blokow try';
}
catch(IntException $wyjatek){
    echo 'nastapil blad typu zmiennej
         '.$wyjatek->getMessage();
}
catch(FunnyException $wyjatek){
    echo 'Stalo sie cos zlego!
         '.$wyjatek->getMessage();
}
catch(Exception $wyjatek){
     echo 'Standardowy wyjatek
          '.$wyjatek->getMessage();
}
?>
```

Metody magiczne

__autoload()

Jest to globalna funkcja, która wywoływana jest gdy tworzymy obiekt klasy która jest niedostępna/nieistnieje.

Funkcja ta pobiera jeden argument - nazwę klasy której obiekt chcemy stworzyć.

Gdy tworzysz obiekt PHP wywołuje funkcję __autoload() a następnie próbuje stworzyć obiekt raz jeszcze.

Przykład :

```
<?php
function __autoload($klasa){
    if( is_file('/klasy/'.$klasa.'.php') ){
        require_once '/klasy/'.$klasa.'.php';
    }
}
?>
```

Załóżmy że stworzyliśmy tablicę w której przechowywane są dostępne klasy, ponadto wszystkie znajdują sie w katalogu - klasy. Tak więc sprawdzamy czy klasa której obiekt chcemy stworzyć jest dostępna, w wyniku powodzenia, dołączamy plik w którym znajduje się definicja klasy.

__set()

Funkcja __set() jest uruchamiana, podczas dynamicznego tworzenia własności klas.Stworzone własnoći zawsze uzyskują dostęp publiczny.

Pobiera ona dwa argumenty - nazwę zmienne i wartość.

Możemy nadpisać metodę __set()

```
<?php
class Nieprzyjazna{
public function __set($zmienna,$wartosc){
    echo 'jestem nieprzyjazna nie mam właściwosci
         '.$zmienna. 'tak wiec nie wpisze  wartosci
         '.$wartosc;
    }
}
$obiekt = new Nieprzyjazna;
$obiekt->nowa = 10;

?>
```

__get()

Funkcja get uruchamiana jest gdy chcemy wypisać właściowść która nie ma ustawnionej wartości.

Pobiera jeden argument - nazwę właściwości

```
<?php
class Kontener
{
private $tablica;

function __construct(){
    $this->tablica = array();
}
```

```
function __set($zmienna,$wartosc){
    $this->tablica[$zmienna] = $wartosc;
}

function __get($zmienna){
    return $this->tablica[$zmienna];
}

function przejrzyj(){
    foreach($this->tablica as $klucz => $wartosc){
        echo '<br />'.$klucz.' = '.$wartosc;
    }
}

};
$pojemnik = new Kontener;
$pojemnik->wysokosc = 100;
echo $pojemnik->wysokosc;
$pojemnik->przejrzyj();
?>
```