

# Programowanie obiektowe w PHP

v 1.2

# Plan

- Idea programowania obiektowego
- Obiekty i klasy



# Idea programowania obiektowego

# Programowanie proceduralne

Skrypt wywołuje funkcje, a osobne kawałki kodu mają za zadanie wykonać jakąś czynność na podstawie danych wejściowych. To jest właśnie programowanie proceduralne.

Ogólnie możemy opisać to jako proces składający się z następujących punktów:

- uruchom funkcję A, podaj funkcji – w postaci argumentów – wszystkie potrzebne dane,
- pobierz wynik funkcji A,
- uruchom funkcję B, przekaż funkcji – w postaci argumentów – wszystkie potrzebne dane.

```
function logOnUser($userName, $password) {  
    //...  
    return true;  
}
```

```
function logOffUser($userName) {  
}
```

```
function getAllFriendsOfUser($userName) {  
    return $allFriends;  
}
```

```
function addFriend($userName, $friendsTable) {  
}
```

# Rozwinięcie programowania proceduralnego

## Struktury danych

- Łatwo można zauważyć, że pewne składowe funkcji zawsze się powtarzają (w naszym przykładzie **\$userName** jest potrzebne w każdej funkcji).
- Rozwiązaniem tego problemu są **struktury danych**.
- Są to np. tablice zawierające komórki z odpowiednimi informacjami.

W naszym przykładzie moglibyśmy założyć następująco:

- w pierwszej komórce znajduje się nazwa,
- w drugiej – hasło,
- w trzeciej – tablica wszystkich przyjaciół.

# Struktury + funkcje = obiekty

Następnym krokiem jest zgrupowanie wszystkich funkcji bazujących na jednej strukturze i połączenie ich z tą strukturą. Taki twór nazywamy obiektem.

- Obiekty najczęściej pochodzą z rzeczywistości np. obiekt – samochód.
- Taki obiekt ma swoje **właściwości** i **zachowania**.

właściwości	zachowania
kolor	przyspiesz
moc silnika	zwolnij
liczba miejsc	skręć w lewo
waga	zatrzymaj się
napęd	zredukuj bieg

# Klasa a obiekt

## Klasa

Jest to schemat opisujący atrybuty i funkcje, które można wywołać na jakimś obiekcie. Można powiedzieć, że jest to szablon, względem którego tworzymy później poszczególne obiekty.

Na przykład klasa samochód definiuje, że każdy samochód musi mieć markę, liczbę drzwi, moc silnika oraz funkcje przyspieszania, hamowania i skręcania.

**W programowaniu obiektowym rozdzielamy dwie główne idee: obiekty i klasy.**

## Obiekt

Jest to dokładna implementacja danej klasy uzupełniająca wszystkie atrybuty, na której można wywoływać funkcje.

Na przykład obiekt **\$redCar** ma markę Honda, czworo drzwi i moc silnika 120 KM.

# Cztery założenia programowania obiektowego

Oto cztery główne założenia programowania obiektowego:

- abstrakcja,
- dziedziczenie,
- hermetyzacja,
- polimorfizm.



# Cztery założenia programowania obiektowego

## Abstrakcja

Odwołujemy się tutaj do założenia, że klasa ma „chować” logikę, dzięki której otrzymujemy jakąś funkcjonalność.

Na przykład przy klasie samochód – kierowca nie interesuje się silnikiem – wie, że po wciśnięciu odpowiedniego pedału samochód zwiększy prędkość.

## Dziedziczenie

Obiekty mogą po sobie dziedziczyć, przejmują swoje właściwości i funkcjonalności.

Na przykład obiekt klasy amfibia dziedziczy po klasie samochód.

Dzięki temu wiemy, że obiekt amfibia będzie umiał jeździć (odziedziczył to po klasie samochód), i rozszerzamy go jeszcze o umiejętność pływania.

# Cztery założenia programowania obiektowego

## Hermetyzacja

Założenie to mówi, że klasa powinna chować kod i dane przed niezaplanowanym użyciem.

Klasa powinna w pełni kontrolować swój stan.

Na przykład klasa samochód nie powinna pozwalać nam usuwać kół, hamulców, itp.

## Polimorfizm

Polimorfizm to możliwość podszywania się pod inne klasy.

Oznacza to, że pracując na jakimś interfejsie (zbiorze funkcjonalności), nie przejmujemy się dokładnym typem klasy, tylko tym, żeby miała te funkcjonalności.

Np. amfibia może podszywać się pod samochód (implementuje wszystkie jego funkcje).

# Cztery założenia programowania obiektowego

Dwa założenia dopisane później mówią nam o zależnościach między obiektami.

## Generalizacja

Opisuje relacje **jest** między obiektami.

Na przykład owoc jest generalizacją zbioru: jabłko, gruszka, pomarańcza.

## Specjalizacja

Oznacza, że każda klasa obiektów powinna się w czymś specjalizować.

Jeżeli klasa obiektów nie ma swojej specjalizacji, powinna zostać wchłonięta przez klasę obiektów znajdujących się wyżej w hierarchii.

# Zalety programowania obiektowego

Programowanie obiektowe ma wiele zalet, m.in.:

- możliwość użycia naszego kodu w różnych projektach,
- kod jest łatwiejszy w utrzymaniu,
- lepsza abstrakcja kodu od problemu, dająca większą możliwość podmiany kodu przy zmianie założeń biznesowych,
- modularność kodu (umożliwiająca łatwiejszą podmianę części kodu przy zmianie systemów).

# Obiekty i klasy

# Nasza pierwsza klasa

Klasę w PHP definiujemy za pomocą słowa kluczowego **class**\*, po którym musi nastąpić nazwa naszej klasy.

Nazwa musi zaczynać się od litery, ale zawierać może litery, numery i niektóre znaki specjalne (np. \_).

Kod tworzący klasę znajduje się w nawiasach klamrowych {...}. Jest to tak zwane ciało klasy.

\* Wyjątek stanowią klasy anonimowe  
<http://php.net/manual/en/language.oop5.anonymous.php>

```
class Book {
```

```
//tutaj znajdzie się ciało klasy
```

```
}
```



Definicja klasy o nazwie Book

# Nasze pierwsze obiekty

```
$book1 = new Book();
```

Stworzenie  
nowego obiektu klasy Book

```
$className = "Book";
```

```
$book2 = new $className();
```

Stworzenie nowego obiektu klasy  
o nazwie podanej w zmiennej \$className

```
var_dump($book1);
```

```
var_dump($book2);
```

## Wynik

```
object(Book)#1 (0) { }
```

```
object(Book)#2 (0) { }
```

Zwróćcie uwagę na id obiektu  
(wartość podana po #)



# Nasze pierwsze obiekty

## Operator new

Do tworzenia nowych obiektów używamy operatora **new**.

Co dokładnie się dzieje podczas wywołania tego operatora?

1. Nowy obiekt jest tworzony (nowa pamięć jest alokowana i inicjalizowana).
2. Wołany jest konstruktor.
3. Obiekt jest wkładany do listy i nadawany jest mu unikalny numer identyfikacyjny.
4. Tworzony jest nowy kontener **zval**, a nowo stworzony obiekt jest do niego przypisywany.

<http://php.net/manual/en/features.gc.refcounting-basics.php>



# Atrybuty klas

Klasy mogą mieć wewnętrzne zmienne (zwane atrybutami).

Atrybuty służą nam do trzymania danych w naszym obiekcie.

W PHP 5 atrybuty poprzedzamy jednym z modyfikatorów dostępu do takiego atrybutu.

Mamy do wyboru:

- **private**
- **protected**
- **public**

# Atrybuty klas

- **private** – atrybut prywatny jest dostępny tylko z wnętrza danej klasy,
- **protected** – atrybut jest widoczny tylko z wnętrza danej klasy i klas dziedziczących (zostanie dokładnie omówiony drugiego dnia zajęć z OOP),
- **public** – atrybut publiczny jest dostępny z każdego miejsca.

```
class Book {
```

```
    public $name;
```

```
    public $price;
```

```
    public $author;
```

```
    private $catalogNumber;
```

```
}
```

Te trzy atrybut są dostępne wszędzie

Ten atrybut będzie dostępny tylko z wnętrza klasy

# Atrybuty klas – wartości domyślne

## Kod

```
$book1 = new Book();  
$book1->name = "Rok 1984";  
$book1->price = 21.50;  
$book1->author = "George Orwell";  
$book1->catalogNumber = "12345";
```



Błąd. Odwołanie się  
do prywatnego atrybutu klasy.

```
var_dump($book1);
```

## Wynik

```
object(Book)#1 (4) {  
    ["name"]=> string(8) "Rok 1984"  
    ["price"]=> float(21.5)  
    ["author"]=> string(13) "George Orwell"  
    ["catalogNumber":"Book":private]=> NULL  
}
```

# Atrybuty klas – wartości domyślne

## Kod

```
class Book {  
    public $name = "Unknown";  
    public $price = 0.0;  
    public $author = "No-name";  
    private $catalogNumber = -1;  
}
```

Atrybuty klas mogą mieć przypisane wartości domyślne. Takie wartości będą przypisywane podczas tworzenia obiektu danej klasy.

## Wynik

```
object(Book)#1 (4) {  
    ["name"]=> string(7) "Unknown"  
    ["price"]=> float(0)  
    ["author"]=> string(7) "No-name"  
    ["catalogNumber":"Book":private]=> int(-1)  
}
```

# Atrybuty klas – operator ->

- Do atrybutów klas możemy się odwołać, używając operatora ->
- Operator ten może również służyć do dynamicznego dopisywania publicznych atrybutów do klas – **overloading**.

**Uwaga! Overloading w PHP znaczy coś innego niż w innych językach programowania!**

- Atrybut dodany dynamicznie jest dostępny tylko i wyłącznie dla obiektu, do którego go dodaliśmy.
- Jeżeli będziemy chcieli się odwołać do niego w innym obiekcie PHP, zwróci nam NULL (oraz zostanie wyświetlona odpowiednia notyfikacja).

# Atrybuty klas – operator ->

## Kod

```
$book1 = new Book();
```

```
$book2 = new Book();
```

```
$book1->cover = "Hard";
```

```
var_dump($book1);
```

```
var_dump($book1->cover);
```

```
var_dump($book2->cover);
```

## Wynik

```
object(Book)#1 (5) {
```

```
...
```

```
["cover"]=> string(4) "Hard"
```

```
}
```

```
string(4) "Hard"
```

```
Notice: Undefined property: Book::$cover in  
index.php on line 20
```

```
NULL
```

# Metody klas

Klasy mogą mieć wewnętrzne funkcje (zwane metodami).

Zarówno metody, jak i atrybuty poprzedzamy jednym z modyfikatorów dostępu do takiego atrybutu.

Mamy do wyboru:

- **private**,
- **protected**,
- **public**,

```
class Book {
```

```
...
```

```
public function printInfo () {
```

```
...
```

```
}
```

```
}
```

Publiczna metoda naszej klasy Book która drukuje informacje o książce

Jeżeli nie nadamy metodzie żadnego modyfikatora dostępu, wtedy nadawany jest jej modyfikator publiczny.



# Wywoływanie metod

Metody naszej klasy możemy potem wywołać na obiekcie tej klasy.

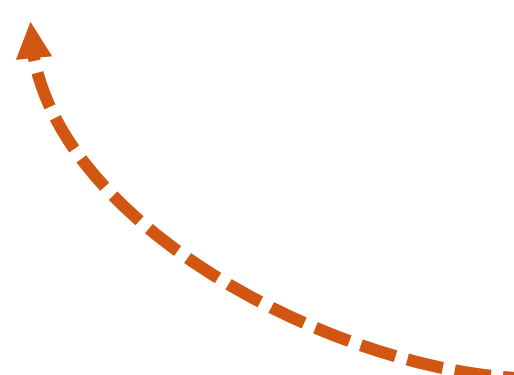
Robimy to poprzez użycie operatora ->

Należy jednak pamiętać że metoda jest wywoływana na danym obiekcie i zawsze będzie miała dostęp do wszystkich danych które zawierają się w tym obiekcie.

```
$book1 = new Book();
```

```
$book1->printInfo();
```

Wywołanie metody **printInfo()** na obiekcie **\$book1**



**Jeżeli nie nadamy metodzie żadnego modyfikatora dostępu, wtedy nadawany jest jej modyfikator publiczny.**



# Metody klas – słowo kluczowe \$this

W metodach mamy od dyspozycji nową pseudozmienną **\$this**.

Zmienna ta jest mechanizmem, dzięki któremu możemy się odnosić do obiektu, na którym wywołaliśmy daną metodę. Możemy o niej myśleć jak o zmiennej w której znajduje się obiekt na którym wywołaliśmy daną metodę.

## Kod

```
class Book {  
    public function printInfo () {  
        var_dump($this);  
    }  
}
```

## Wynik

```
object(Book)#1 (4) {  
    ["name"]=> string(7) "Unknown"  
    ["price"]=> float(0)  
    ["author"]=> string(7) "No-name"  
    ["catalogNumber":"Book":private]=> int(-1)  
}
```

# Magiczne metody

- PHP ma zaimplementowane tzw. magiczne metody. Są one używane w ściśle określonych przypadkach.
- Wszystkie magiczne metody zaczynają się od dwóch podkreśleń (\_\_), dlatego nigdy nie tworzymy własnych funkcji o tym przedrostku.

<http://php.net/manual/en/language.oop5.magic.php>

<http://www.techflirt.com/tutorials/oop-in-php/magic-methods-in-php.html>

- **\_\_construct()**
- **\_\_destruct()**
- **\_\_call()**
- **\_\_callStatic()**
- **\_\_get()**
- **\_\_set()**
- **\_\_isset()**
- **\_\_unset()**
- **\_\_sleep()**
- **\_\_wakeup()**
- **\_\_toString()**
- **\_\_invoke()**
- **\_\_set\_state()**
- **\_\_clone()**
- **\_\_debugInfo()**

# \_\_construct()

- Konstruktor to funkcja magiczna, która jest wywoływana podczas tworzenia nowego obiektu.
- W PHP 5 ma nazwę \_\_construct, a w PHP 4 miała taką samą nazwę jak klasa (ta konstrukcja działa nadal w PHP 5).
- Dobrą praktyką jest przypisywanie wartości domyślnie w konstruktorze, a nie w ciele klasy.
- Jeżeli nie zdefiniujemy żadnego konstruktora, PHP stworzy pusty konstruktor dla naszej klasy.

## Kod

```
class Book {  
    ...  
    public function __construct() {  
        echo("Tworzę nowa książkę. <br>");  
    }  
}
```

# Prywatna metoda \_\_construct()

- Prywatny konstruktor spowoduje sytuację, w której stworzenie waszej klasy spoza metod tej klasy będzie niemożliwe.
- Jest to sytuacja poprawna służąca do implementacji wzorca projektowego Singleton.

```
class Book {  
    ...  
    private function __construct() {  
        echo("Tworzę nowa książkę. <br>");  
    }  
}
```

# \_\_destruct()

- Destruktor jest funkcją magiczną wywoływaną podczas niszczenia waszego obiektu.
- W PHP 5 ma nazwę **\_\_destruct**, wcześniej nie istniała taka metoda.
- Jeżeli nie zdefiniujemy żadnego destruktora, PHP stworzy pusty destruktor dla naszej klasy.
- Destruktor nie powinien przyjmować żadnych parametrów (i tak nie możemy ich przekazać).

# \_\_destruct()

## Kod

```
class Book {  
    ...  
    public function __destruct() {  
        echo("Niszczę książkę. <br>");  
    }  
}  
  
$book1 = new Book();  
$book1 = null;
```

## Wynik

Tworzę nowa książkę.  
Niszczę książkę.

# Setery i getery

Przy programowaniu obiektowym bardzo często widuje się tak zwane **setery** i **getery**.

Są to specjalne funkcje które dają nam dostęp do prywatnych atrybutów naszej klasy.

Służą one do tego żebyśmy kontrolowali dostęp do prywatnych atrybutów klasy. Dzięki temu będziemy spełniać założenia enkapsulacji (czyli jednej z podstaw obiektowości).



# Setery

Setery to metody których celem jest nastawienie jakiegoś atrybutu klasy.

Zazwyczaj pisze się osobne metody do każdego atrybutu klasy który pozwalamy zmieniać (a nie zawsze będziemy pozwalać zmieniać wszystkie z naszych atrybutów).

Setery powinny sprawdzać czy przekazane zostały poprawne dane i czy dane są w poprawnym zakresie.

Np. cena książki musi być liczbą i nie może być mniejsza niż 0.

```
class Book {  
    private $price;  
    ...  
    public function setPrice( $newPrice ) {  
        if( is_numeric( $newPrice ) && $newPrice > 0 ){  
            $this->price = $newPrice;  
        }  
        return $this;  
    }  
}
```

Zwyczajowo setery zaczynają się od słowa **set** a potem mają nazwę atrybutu na który wpływają

Setery zazwyczaj nie zwracają żadnej wartości lub zwracają **\$this**

Główną rolą seterów jest sprawdzenie czy nastawiana wartość jest poprawna



# return \$this?


Na początku może dziwić że funkcje setujące zwracają obiekt na którym wykonały jakąś zmianę.

Jest to konwencja lubiana przez niektórych programistów nazywana **chaining**.

Zwracanie obiektu na którym pracujemy daje nam możliwość wywołania następnej funkcji set w łańcuchu.

Czasami jednak powoduje to trudność w znalezieniu błędu w kodzie (w jednej linijce jest wywołane kilka metod).

```
$book1 = new Book();  
$book1->setPrice(12)->setName("Paragraf 22")  
->setAutor("Joseph Heller");
```



Dzięki zwracaniu **\$this** z seterów możemy po kolei wywoływać je na jednym obiekcie

# Getery

Getery to metody które zwracają jakiś atrybut z obiektu. Dzięki nim możemy przeczytać jakie obiekt ma wartości bez możliwości wpływu na nie.

Zazwyczaj piszemy osobne getery do każdego atrybutu jaki chcemy pokazać na zewnątrz (nie musimy chcieć pokazywać wszystkich atrybutów).

```
class Book {  
    private $price;  
    ...  
    public function getPrice() {  
        return $this->price;  
    }  
}
```

Zwyczajowo getery zaczynają się od słowa get a potem mają nazwę atrybutu który zwracają

Jedynym celem funkcji set jest zwrócenie danego atrybutu

# Czas na zadania

- Przeróbcie ćwiczenia z części A z waszego Repozytorium  
Pierwsze ćwiczenie zróbcie z wykładowcą.