# 10. Programowanie obiektowe w PHP5

**Dlaczego PHP5?** Pierwsze nieśmiałe próby implementacji technik obiektowych mieliśmy już w PHP4, ale były one niekonsekwentne i niekompletne, PHP4 było językiem proceduralnym z obiektowością doklejoną z boku. Dopiero w PHP5 pojawiła się obiektowość z prawdziwego zdarzenia. Ponieważ optymistycznie zakładam, że słuchacze pamiętają jeszcze co nieco z kursu C++, podzielę techniki obiektowe w PHP na trzy grupy:

- rzeczy, które wyglądają jak w C++ i działają tak samo,
- rzeczy, które wyglądają jak w C++, ale działają inaczej,
- rzeczy, których nie ma w C++.

Zaczniemy od tych pierwszych, a więc od samej definicji klasy. Oto prosty przykład:

## 10.1. Definicja klasy

```
class NaszaKlasaPl
{
    $element = 6;
    $drugi = 'a kto umarł, ten nie żyje';
    $bezwartosci;
```

Ogólnie definicja klasy wygląda jak w C++. Oczywiście elementy składowe klasy są zmiennymi PHP, stąd nieśmiertelne „$”. Warto zauważyć, że mogą one mieć wartość

domyślną, co jest pewną nowością. Nadanie wartości domyślnej określa nam typ składowej (ale zgodnie z zasadami PHP, może on później ulec zmianie). Element bez wartości domyślnej jest typu *NULL*. Trzeba pamiętać o tym, że wartość domyślna **musi** być wyrażeniem stałym, nie może więc zawierać operatorów, ani nazw zmiennych.

```
    function JakasMetoda($parametr)
    {
        echo $parametr.$drugi;
    }
}
```

Definicje metod są analogiczne do C++, oczywiście w PHP parametry nie mają typów, co między innymi skutkuje tym, że nie można przeciążać metod (nie można mieć kilku metod o takich samych nazwach a różnych typach parametrów). Na zakończenie zauważmy, że definicja klasy w PHP nie jest zakończona średnikiem.

## 10.2. Tworzenie nowych obiektów i operator *new*

Podobnie jak w C++ operator *new* służy nam do tworzenia nowych obiektów. Natomiast inaczej, niż w C++, operator *new* jest jedynym sposobem ich tworzenia, obiekty globalne i automatyczne (lokalne) nie mają w PHP miejsca ze względu na strukturę języka. Oto przykład:

```
$szybki_serwer = new NaszaKlasaPl();
```

## 10.3. Klasy pochodne

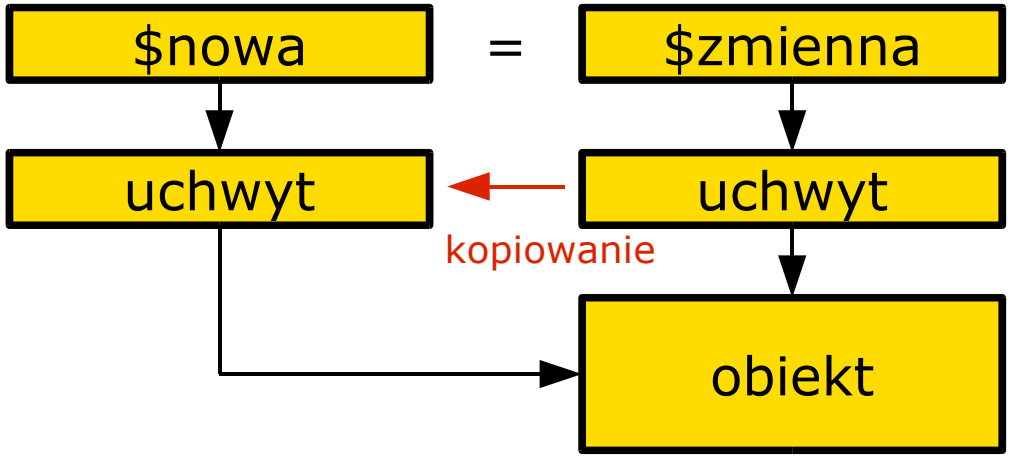
Trudno mówić o obiektowości języka bez klas pochodnych. W PHP5 tworzymy klasy pochodne używając słowa kluczowego *extends*. Nie jest możliwe dziedziczenie wielokrotne (po kilku klasach jednocześnie).

```
class DaneOsobowe extends NaszaKlasaPl
{
    $nowy_element = 'bzz';

    function NowaMetoda()
    {
        echo 'Nowa';
    }

    function JakasMetoda($parametr)
    {
        parent::JakasMetoda($parametr);
        echo 'I dodatek.';
    }
}
```

Klasa pochodna może dodawać nowe elementy i metody, może też nadpisywać stare metody. Jeżeli metoda jest redefiniowana, to PHP nie wykona automatycznie metody klasy nadrzędnej (tak samo jak w C++), jeżeli chcemy to uzyskać, posługujemy się słowem kluczowym *parent* i operatorem zakresu („::”).

# 10.4. Zachowanie się obiektów przy przypisaniu

Obiekty zachowują się przy przypisaniu inaczej niż pozostałe typy zmiennych, mianowicie operator przypisania nie kopiuje obiektów. Wynika to z faktu, że zmienna typu *object* nie przechowuje samego obiektu, a jedynie jego uchwyt. Ilustruje to rysunek.




Jeżeli chcemy mieć kopię obiektu, musimy ją stworzyć jawnie poprzez klonowanie, służy do tego operator *clone()*. Jego standardowa wersja kopiuje obiekt składowa po składowej, ale można napisać własną metodę *__clone()*, jeżeli kopiowanie ma się odbyć w jakiś specyficzny sposób.

## 10.5. Konstrukcja i destrukcja obiektów

Tak samo jak C++, PHP pozwala nam na zdefiniowanie w klasie konstruktora (ale tyko jednego, przeciążanie odpada), oraz destruktora. Te dwie metody możemy pominąć, wtedy zostaną użyte metody domyślne, które nic nie robią. W PHP nazwy tych dwóch specjalnych metod nie mają nic wspólnego z nazwą klasy, są to *__construct()* oraz *__destruct()*.Konstruktor jest wywoływany zaraz po stworzeniu obiektu, destruktor tuż przed jego zniszczeniem. Tak jak dla zwykłych metod, konstruktor i destruktor klasy nadrzędnej nie są wywoływane automatycznie i w razie potrzeby robimy to ręcznie używając *parent::*. Konstruktor i destruktor nie zwracają żadnej wartości, destruktor nie ma żadnych parametrów.

```
class Student
{
    $imie;
    $nazwisko;

    function __construct($i, $n)
    {
        $this->imie = $i;
        $this->nazwisko = $n;
    }

    function __destruct()
    {
        print("Z przykrością zawiadamiamy, że $this->imie $this->nazwisko
        został zlikwidowany.");
    }
}
```

## 10.6. *$this*, *self::* i konieczność ich użycia

Ze względu na zasięg zmiennych w PHP, do elementów klasy odwołujemy się zawsze poprzez pseudo-zmienną *$this*, której znaczenie jest takie samo jak w C++, z tym że bez posiadania dosłownego znaczenia wskaźnika. Warto zauważyć, że nazwa składowej nie jest już poprzedzana dolarem, można traktować to, w ten sposób, że nazwą zmiennej jest „this->zmienna” jako całość.

```
class Iks
{
    $x = 'jestem x';

    function Opisz() { echo $this->x; }
```

O ile *$this* odnosi się do konkretnego obiektu, *self::* odnosi się do klasy i służy do odwoływania się do jej elementów statycznych, a więc metod, oraz składowych statycznych, o których dalej. Oto objaśniający sprawę przykład (kontynuujemy definicję klasy *Iks*):

```
    function Witam()
    {
        $b = $this->x;          // dobrze
        $b = self::x;           // źle, x nie jest statyczny
        Opisz();                // źle wbrew pozorom
        $this->Opisz();         // dobrze, również wbrew pozorom
        self::Opisz();          // ale tak jest przejrzyściej
    }
```

# 10.7. Kontrola dostępu do składowych

Mamy tu sytuację analogiczną do C++. Dysponujemy słowami kluczowymi *public*, *protected* i *private*. Ich znaczenie jest podobne do C++, składowe publiczne dostępne są wszędzie, chronione tylko w klasie i klasach pochodnych, prywatne tylko w klasie. Domyślnie, jeżeli brak słowa kluczowego, składowa jest publiczna. Konstruktory i destruktory w PHP **muszą** być publiczne jeżeli są zdefiniowane.

```
class W
{
    public $pub;
    protected $pro;
    private $pri;
}

class X extends W
{
    function M()
{
        $p = $this->pub;        // OK
        $p = $this->pro;        // OK
        $p = $this->pri;        // chała!
    }
}

$obj = new W();

$cokolwiek = $obj->pub;         // OK
```

```
$cokolwiek = $obj->pro;        // to se ne da
$cokolwiek = $obj->pri;        // oczywiście też nie
```

## 10.8. Składowe statyczne

Składowe takie definiuje się używając słowa *static*. Powoduje to dwie rzeczy. Po pierwsze składowa statyczna jest jedna dla wszystkich obiektów (w przypadku zwykłej składowej każdy obiekt ma swoją kopię). Po drugie składowa statyczna istnieje nawet gdy nie ma żadnego obiektu danej klasy.

```
class Beee
{
    public $fooo = 'fooo';
    static public $beee = 'beee';
}

// nie ma żadnych obiektów klasy Beee

echo Beee::$fooo;     // błąd!
echo Beee::$beee;     // dobrze
```

Statyczne mogą być również metody, co prawda w pewnym sensie statyczna jest każda metoda, bo wszystkie obiekty danej klasy mają tę samą kopię jej kodu, niemniej użycie słowa *static* pozwala na wykonanie metody bez posiadania jakiegokolwiek obiektu klasy. Konsekwencją jednak jest niedziałanie w takiej metodzie *$this* (skoro można ją wywołać bez obiektu...), a więc nie ma ona dostępu do składowych niestatycznych.

```
class MetodeMaStatyczna
{
    public $normalna;
    public static $statyczna;

    function Metoda()
    {
        $this->normalna = 'blablabla';
        $this->statyczna = 'adzia!';
    }

    static function Statyczna()
    {
        $this->normalna = 'umc, umc, umc';    // niestety nie przejdzie
        self::$normalna = 'umc, umc, umc';    // padnie przy wywołaniu
        self::$statyczna = 'sesja jedyną miłością studenta';  // OK
    }
}

// obiektów brak

MetodeMaStatyczna::Metoda();      // nie można
MetodeMaStatyczna::Statyczna();   // można

$obj = new MetodeMaStatyczna();

$obj->Metoda();                   // można
$obj->Statyczna();                // też można
```

## 10.9. Klasy abstrakcyjne

To temat również znany z C++, klasy abstrakcyjne to takie, które nie mogą mieć własnych obiektów, mogą mieć tylko obiekty klas pochodnych. Klasa abstrakcyjna powstaje wtedy, gdy co najmniej jedna jej metoda jest abstrakcyjna:

```
abstract class Abstrakcja
{
    abstract function M1();   // brak definicji
    function M2()
    {
        // coś w środku
    }
}
```

Klasa pochodna od takiej ma dwa wyjścia. Albo zaimplementować wszystkie metody abstrakcyjne (wtedy jest już zwykłą klasą, mogącą posiadać obiekty), albo po prostu pominąć którąś z metod abstrakcyjnych (wtedy klasa pochodna również jest abstrakcyjna).

## 10.10. Interfejsy

Interfejs jest to jak gdyby szablon (niewiele mający wspólnego z szablonem w C++), który jest jak gdyby instrukcją budowania własnych klas. Interfejs to po prostu zestaw metod z argumentami, klasa która **implementuje** interfejs musi posiadać wszystkie metody z wykazu. Interfejsów używa się więc do porządkowania kodu, jeżeli narzucimy

kilku klasom jeden interfejs, mamy gwarancję, że do obiektów wszystkich tych klas będziemy się mogli odwoływać przez ten sam zestaw metod.

```
interface Ofiara
{
    public function uderz_go($gdzie);
    public function zaatakuj_go($jak);
    public function przegral();
}

class Bokser implements Ofiara
{
    // tutaj musimy zaimplementować wszystkie metody z interfejsu
}

$Andrzej = new Bokser();

// mamy pewność, że możemy używać metod określonych w interfejsie

$Andrzej->uderz_go('korpus');
$Andrzej->zaatakuj_go('prawy prosty');
$Andrzej->uderz_go('glowa');
$Andrzej->przegral();

// koniec walki na dziś
```