Więcej o programowaniu obiektowym.

Dziedziczenie

Dziedziczenie to jeden z fundamentów programowania obiektowego. Umożliwia sprawne i łatwe wykorzystywanie napisanego już raz kodu przez budowanie hierarchii klas przejmujących swoje właściwości. Do zaprezentowania, jak tworzy się klasy potomne oraz jakie zależności występują między klasą potomną a bazową utworzymy ogólną klasę ***Osoba***.

```php
<?php
class Osoba
{
  public $imie, $nazwisko;
  public function ustawImie($imie)
  {
    $this->imie = $imie;
  }
  public function ustawNazwisko($nazwisko)
  {
    $this->nazwisko = $nazwisko;
  }
  public function pobierzImie()
  {
    return $this->imie;
  }
  public function pobierzNazwisko()
  {
    return $this->nazwisko;
  }
}
?>
```

Do przechowywania danych dotyczących imienia i nazwiska służą pola o nazwach ***$imie*** i ***$nazwisko***, a ich wartości mogą być ustawiane i odczytywane zarówno bezpośrednio, jak i za pomocą wywołania dostępnych w klasie metod. Zastanówmy się teraz, co należałoby zrobić, gdyby w systemie oprócz danych osób trzeba było zapisywać dane dotyczące użytkowników, a każdy z nich posiadałby imię, nazwisko oraz identyfikator nadawany przez system. Pomysłem jest napisanie dodatkowej klasy np. o nazwie ***Uzytkownik*** w postaci:

```php
class Uzytkownik
{
  public $imie, $nazwisko, $id;
  /*
  w tym miejscu metody klasy
  */
}
```

W której należałoby dopisać pełny zestaw metod takich jak w klasie ***Osoba***, oraz dodatkowe metody obsługujące pole ***$id***. Dlatego klasę ***Uzytkownik*** należy potraktować jako rozszerzenie klasy ***Osoba***. Rozszerzenie o dodatkowe pola i metody. Zamiast więc pisać całkiem od nowa kod klasy ***Uzytkownik***, lepiej spowodować, aby przejęła ona wszystkie możliwości klasy ***Osoba***, wprowadzając dodatkowo swoje własne. To właśnie jest dziedziczenie , które w PHP jest realizowane, za pomocą słowa ***extends*** schematycznie taka konstrukcja ma następującą postać:

```php
class klasa_potomna extends klasa_bazowa
{
  /*
  kod klasy
  */
}
```

Zapis taki oznacza, że klasa potomna dziedziczy z klasy bazowej. W przypadku omawianych klas będzie to wyglądało następująco:

```php
require_once("Osoba.php");

class Uzytkownik extends Osoba
{
  public $id;
  public function ustawId($id)
  {
    $this->id = $id;
  }
  public function pobierzId()
  {
    return $this->id;
  }
}
```

Użycie obiektów klas Osoba i Uzytkownik:

```php
include "osoba.php";
include "uzytkownik.php";

$osoba = new Osoba();

$osoba->ustawImie("Andrzej");
$osoba->ustawNazwisko("Nowak");

echo "Dane obiektu \$osoba\n";
echo $osoba->pobierzImie(), "\n";
echo $osoba->pobierzNazwisko(), "\n";

$uzytkownik = new Uzytkownik();

$uzytkownik->ustawImie("Jan");
$uzytkownik->ustawNazwisko("Kowalski");
$uzytkownik->ustawId("012");

echo "\nDane obiektu \$uzytkownik\n";
echo $uzytkownik->pobierzImie(), "\n";
echo $uzytkownik->pobierzNazwisko(), "\n";
echo $uzytkownik->pobierzId(), "\n";
```

Przesłanianie składowych

Wiadomo, że w trakcie dziedziczenia klasa potomna dziedziczy składowe z klasy bazowej. Co się jednak stanie, jeśli w klasie potomnej zostaną zdefiniowane pola i metody o takich samych nazwach jak w klasie bazowej? Otóż w takiej sytuacji składowa z klasy bazowej zostanie przesłonięta przez tę z klasy potomnej. Wyjaśnijmy to na prostym przykładzie:

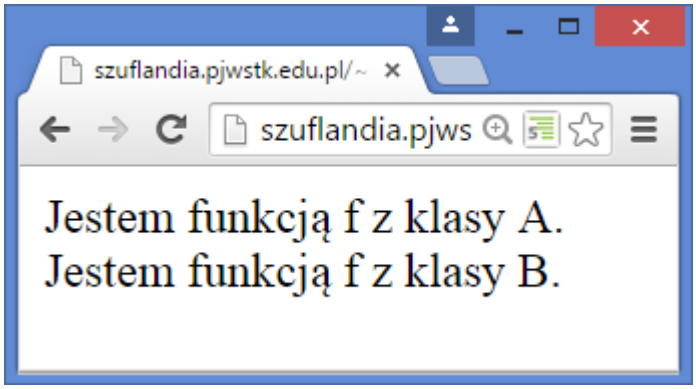
```php
<?php
class A
{
  public function f()
  {
    echo "Jestem funkcją f z klasy A.";
  }
}

class B extends A
{
  public function f()
  {
    echo "Jestem funkcją f z klasy B.";
  }
}

$objA = new A();
$objB = new B();

$objA->f();
echo "\n";
$objB->f();
?>
```

Mamy tu dwie klasy ***A*** i ***B***, obie zawierają metodę o takiej samej nazwie – ***f***. W kodzie tworzone są też dwa obiekty tych klas - ***$objA*** i ***$objB***. Po utworzeniu obiektów wywoływana jest metoda ***f*** każdego z nich. Klasa ***B*** dziedziczy z klasy ***A***. Jak więc zachowa się przedstawiony kod?



Wydawać by się mogło, że powinien wystąpić konflikt nazw. Dlaczego więc konflikt nazw nie występuje? Otóż zasada jest następująca: jeśli w klasie bazowej i pochodnej występuje metoda o tej samej nazwie i argumentach, ta z klasy bazowej jest przesłaniana. Czyli w obiektach klasy bazowej będzie obowiązywała metoda z klasy bazowej, a w obiektach klasy pochodnej ta z klasy pochodnej.

W tym miejscu może pojawić się pytanie, czy jest możliwe wywołanie w klasie pochodnej przesłoniętej metody z klasy bazowej. W tym celu jest wykorzystywane odwołanie ***parent::*** o ogólnej postaci:

parent::nazwa_metody(argumenty);

Dostęp do przesłoniętej metody:

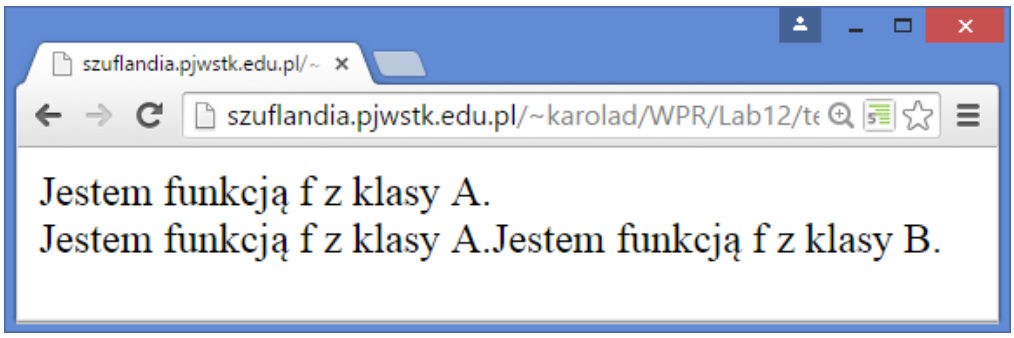
```php
<?php
class A
{
  public function f()
  {
    echo "Jestem funkcj± f z klasy A.";
  }
}

class B extends A
{
  public function f()
  {
    parent::f();
    echo "Jestem funkcj± f z klasy B.";
  }
}

$objA = new A();
$objB = new B();

$objA->f();
echo "\n";
$objB->f();
?>
```

Ilustracja dostępu do przesłoniętej metody:



Klasy i składowe finalne

Wiemy już, że metody klasy bazowej mogą być przeciążane w klasach pochodnych. Taka sytuacja nie zawsze jest jednak pożądana, dlatego też PHP umożliwia deklarowanie metod jako finalne. Tak zadeklarowane składowe (za pomocą słowa kluczowego ***final***) nie mogą być przeciążane. Schematycznie ta konstrukcja będzie miała następującą postać:

class nazwa_klasy

{

final public function nazwa_metody

{

//kod metody

}

//definicja pozostałych składowych

}

Próba przeciążenia metody finalnej:

```php
<?php
class Osoba
{
  public $imie, $nazwisko;
  final public function wyswietl()
  {
    echo "Imie: ", $this->imie, "\n";
    echo "Nazwisko: ", $this->nazwisko, "\n"
  }
}

class Uzytkownik extends Osoba
{
  public $id;
  public function wyswietl()
  {
    parent::wyswietl();
    echo "Id: ", $this->id, "\n";
  }
}
?>
```

Konstruktory i destruktory klas bazowych

Przy dziedziczeniu klas należy zwrócić uwagę na zachowanie konstruktorów i destruktorów. Otóż te, które zostały zdefiniowane w klasach potomnych, zostaną oczywiście wykonane, nie będą natomiast wywoływane konstruktory i destruktory klas bazowych. Z kolei jeśli w klasie potomnej nie ma zdefiniowanego konstruktora, wówczas zostanie wykonany konstruktor z klasy bazowej. To zagadnienie zilustrowano w przykładzie:

```php
<?php
class A
{
  function __construct()
  {
    echo("Jestem konstruktorem z klasy A.\n");
  }
  function __destruct()
  {
    echo("Jestem destruktorem z klasy A.\n");
  }
}

class B extends A
{
  function __construct()
  {
    echo("Jestem konstruktorem z klasy B.\n");
  }
  function __destruct()
  {
    echo("Jestem destruktorem z klasy B.\n");
  }
}

class C extends A
{
}

$objB = new B();
//$objC = new C();
?>
```

Zostały w nim zdefiniowane trzy klasy – ***A***, ***B*** i ***C***, przy czym klasy ***B*** i ***C*** dziedziczą z klasy A. Klasy ***A*** i ***B*** zawierają konstruktory i destruktory mające za zadanie wyświetlenie informacji o tym, z której klasy dana metoda pochodzi natomiast klasa ***C*** jest ich pozbawiona. Choć moglibyśmy się spodziewać, że skoro ***B*** dziedziczy z ***A***, to przy tworzeniu obiektu zostanie wykonany zarówno konstruktor z klasy ***A***, jak i ***B***, tak jednak nie jest. Prawdziwe jest zatem stwierdzenie, że jeśli w klasie potomnej jest zdefiniowany konstruktor, to tylko on zostanie wykonany.

Aby wywołać konstruktora klasy bazowej należy użyć składni ze słowem kluczowym ***parent:***

```php
<?php
class A
{
  function __construct()
  {
    echo("Jestem konstruktorem z klasy A.\n");
  }
  function __destruct()
  {
    echo("Jestem destruktorem z klasy A.\n");
  }
}

class B extends A
{
  function __construct()
  {
    parent::__construct();
    echo("Jestem konstruktorem z klasy B.\n");
  }
  function __destruct()
  {
    parent::__destruct();
    echo("Jestem destruktorem z klasy B.\n");
  }
}

$objB = new B();
?>
```

Specyfikatory dostępu

W PHP składowe klasy muszą mieć określony sposób dostępu, który definiuje się przez tzw. specyfikatory dostępu. Wyróżniamy trzy takie specyfikatory:

- ***public*** – dostęp publiczny,
- ***protected*** – dostęp chroniony,
- ***private*** – dostęp prywatny.

Specyfikator musi wystąpić przed nazwą pola, przed nazwą metody – nie, jest ona wtedy traktowana jako public. Co jednak oznaczają te słowa kluczowe?

- Dostęp do składowych publicznych jest nieograniczony, co oznacza że można się do nich do nich dowolnie odwoływać.
- Dostęp do składowych chronionych jest ograniczony do klasy, w której są one zdefiniowane, oraz do klas bazowych i pochodnych.
- Dostęp do składowych prywatnych jest ograniczony tylko do klasy, w której są zdefiniowane.

```php
<?php
class A
{
  public $wartosc_1 = 10;
  protected $wartosc_2 = 20;
  private $wartosc_3 = 30;
  public function wyswietl()
  {
    echo "wartosc_1 = ", $this->wartosc_1, "\n";
    echo "wartosc_2 = ", $this->wartosc_2, "\n";
    echo "wartosc_3 = ", $this->wartosc_3, "\n";
  }
}
class B extends A
{
  public function wyswietl()
  {
    echo "wartosc_1 = ", $this->wartosc_1, "\n";
    echo "wartosc_2 = ", $this->wartosc_2, "\n";
    //echo "wartosc_3 = ", $this->wartosc_3, "\n";
  }
}
$objA = new A();
$objB = new B();

echo "Zawarto¶ć obiektu \$objA:\n";
echo $objA->wartosc_1, "\n";
//echo $objA->wartosc_2, "\n";
//echo $objA->wartosc_3, "\n";
$objA->wyswietl();

echo "\nZawarto¶ć obiektu \$objB:\n";
$objB->wyswietl();

?>
```

W kodzie skryptu zostały utworzone dwa obiekty - $objA klasy A oraz $objB klasy B. po utworzeniu obiektów wykonywana jest seria instrukcji obrazujących prawidłowe i nieprawidłowe odwołania:

- ***echo $objA->wartosc_1***: jest prawidłowa, gdyż pole ***wartość_1*** jest publiczne i można się do niego swobodnie odwoływać.
- ***echo $objA->wartosc_2***: jest nieprawidłowa, gdyż pole ***wartość_2*** jest chronione i można się do niego odwoływać jedynie z wnętrza klasy A lub klasy pochodnej B.
- ***echo $objA->wartosc_3***: jestnie prawidłowa, gdyż pole ***wartość_3*** jest prywatne i można się do niego odwołać jedynie z wnętrza klasy A.
- ***echo $objA->wartosc_1***: jest prawidłowa, gdyż pole ***wartość_1*** jest publiczne i można się do niego swobodnie odwoływać.
- ***$objA->wyswietl()***: jest prawidłowa, gdyż metoda ***wyswietl*** klasy A jest publiczna a ponieważ znajduje się wewnątrz klasy, ma pełny dostęp do wszystkich pól i może wyświetlać ich zawartość

- ***$objB->wyswietl()***: jest prawidłowa, gdyż metoda ***wyswietl*** klasy B jest publiczna , jednak zawarta w niej instrukcja ***echo " wartość_3=", $this->wartość_3,"\n***"; jest nieprawidłowa, gdyż próbuje się odwołać do prywatnej składowej z klasy A.

Składowe statyczne

Są to takie składowe, które istnieją nawet wtedy, gdy nie ma żadnego obiektu danej klasy. Są one deklarowane za pomocą słowa kluczowego ***static***, które należy umieścić po specyfikatorze dostępu. Schematyczna postać w przypadku pól:

Specyfikator_dostępu static $nazwa_pola;

W przypadku metod:

Specyfikator_dostępu static function nazwa_metody(argumenty);

Aby dostać się do składowej statycznej, czyli zapisać lub odczytać wartość, bądź też wywołać metodę, należy użyć konstrukcji dla pól w postaci:

Nazwa_klasy::nazwa_pola

Oraz dla metod w postaci:

Nazwa_klasy::nazwa_metody(argumenty)

Zobrazowano to w kodzie:

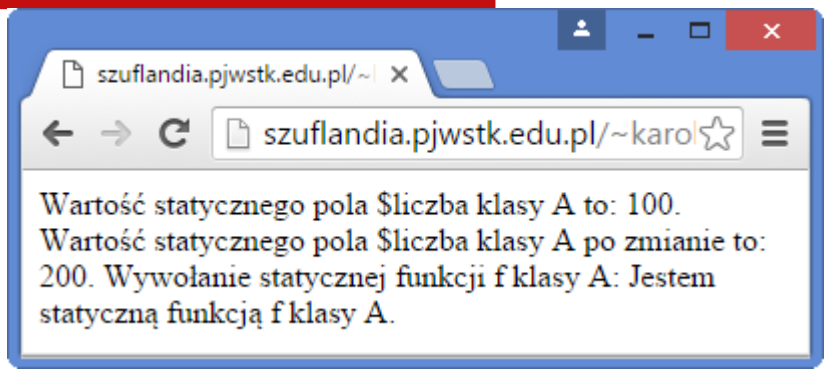
```
<?php
class A
{
  public static $liczba = 100;
  public static function f()
  {
    echo "Jestem statyczn± funkcj± f klasy A.";
  }
}

echo "Warto¶ć statycznego pola \$liczba klasy A to: ";
echo A::$liczba, ".\n";

A::$liczba = 200;

echo "Warto¶ć statycznego pola \$liczba klasy A po zmianie to: ";
echo A::$liczba, ".\n";

echo "Wywołanie statycznej funkcji f klasy A:\n";
A::f();
?>
```



Odwołanie do składowej statycznej przez bieżący obiekt wymaga użycia słowa kluczowego ***self*** i operatora ***::***. Schematycznie dla pól:

Self::nazwa_statycznego pola

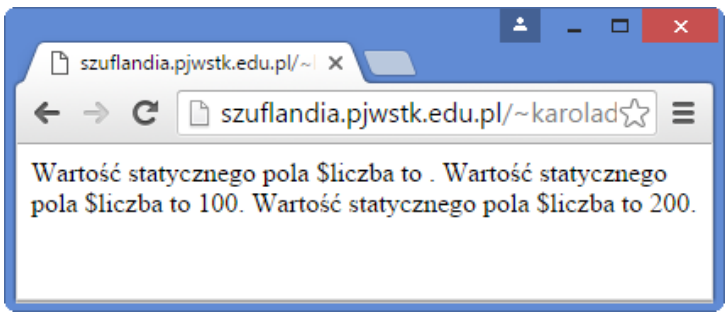
A dla metod:

Self::nazwa_statycznej_metody(argumenty)

```php
<?php
class A
{
  public static $liczba = 100;
  public static function f()
  {
    echo "Warto¶ć statycznego pola \$liczba to ";
    echo A::$liczba, ".\n";
  }
  public function g()
  {
    self::$liczba = 200;
    self::f();
  }
}

$objA = new A();

echo "Warto¶ć statycznego pola \$liczba to {$objA->liczba}.\n";
$objA->f();
$objA->g();
?>
```



Aplikacja nowe auto

Zaczniemy od zaprojektowania klas oraz ich zawartości, czyli pól, właściwości i metod.

Klasą bazową będzie klasa **NoweAuto** zawierająca informacje, takie jak:

- model auta,
- cena w EURO,
- aktualny kurs EURO.

Klasa ta będzie obliczała cenę auta po aktualnym kursie za pomocą metody o nazwie **ObliczCene**.

Klasa potomna w stosunku do **NoweAuto** to **AutoZDodatkami**.

Będzie ona dziedziczyć wszystkie elementy klasy bazowej oraz dodatkowo posiadać nowe właściwości:

- alarm,
- radioodtwarzacz,
- klimatyzacja.

Będzie także obliczała cenę auta z dodatkami. Ponieważ nie chcemy mieć różnych nazw procedur, zastosujemy tu przesłanianie metod. W ten sposób funkcja obliczająca cenę będzie się tak samo nazywać jak w klasie bazowej – **ObliczCene**.

Ostatnią klasą będzie **Ubezpieczenie**, klasa potomna w stosunku do klasy **AutoZDodatkami**.

Nowymi elementami będą właściwości:

- pierwsze auto,
- liczba lat.

Na ich podstawie będziemy mogli dzięki metodzie **ObliczCene** wyliczać ceny aut z uwzględnionymi dodatkami oraz wartością ubezpieczenia. Obliczenie ceny z ubezpieczeniem, za każdy rok posiadania auta jest odliczany 1% wartości auta.

Zadanie domowe

Zaprojektuj formularz do obliczania ceny auta z wykorzystaniem zdefiniowanych klas.