Podstawy programowania III

Jan Kazimirski

Opis zajęć

- Zastosowanie funkcji i obiektów w PHP – powtórzenie zagadnień.
- Dostęp do bazy danych PostgreSQL z poziomu skryptu PHP.
- Wprowadzenie do języka SQL.
- Przykłady projektów PHP z wykorzystaniem programowania obiektowego i bazy danych PostgreSQL.

Literatura

- http://www.php.net/manual/en/index.php
- **D. Bargieł, „PHP5. Kompendium webmastera", Helion, 2005**
- E. Lecky-Thompson, H. Elde-Goodman, S. D. Nowicki, A. Cove, „PHP5. Zaawansowane programowanie", Helion, 2005

Warunki zaliczenia

- Zaliczenie na podstawie oceny z laboratorium
- Zastrzegam sobie prawo do podniesienia oceny osobom, które uczestniczą w wykładach i są aktywne na wykładach i na platformie e-learningowej).

Funkcje - powtórka

- Funkcja – wydzielony blok kodu wykonujący określone zadanie
- Może działać na zestawie przekazanych do niej parametrów
- Może (ale nie musi – procedura) zwracać określoną wartość
- PHP dysponuje dużą liczbą wbudowanych funkcji, użytkownik może definiować swoje własne

Funkcje – dlaczego używać

- Zmniejszają ilość kodu – szybsze wykonanie programu
- Fragmenty kodu są izolowane – mogą być oddzielnie testowane.
- Kod funkcji występuje tylko w jednym miejscu – łatwiej znaleźć i poprawić błędy.
- Raz napisane mogą być wielokrotnie używane – również w innych projektach.

Definiowanie funkcji

Nazwa funkcji

Parametr lub parametry (może ich nie być)

```
function nazwa($parametr) {
   // lista instrukcji
   return wartość;
}
```

Ciało funkcji – kod zawarty w funkcji

Wartość zwracana

Przykład funkcji (1)

```
<?php
function suma($p1,$p2) {
  $p3 = $p1+$p2;
  return $p3;
}
?>
```

Przykład funkcji (2)

```
<?php
function stopka() {
   echo '</body></html>';
}
?>
```

Wywołanie funkcji

```
<?php

$length = strlen("Ahoj");

$res = sin(asin(1.0));

die("Koniec programu\n");

?>
```

Parametry funkcji

- Funkcja może mieć dowolną liczbę parametrów (lub nie mieć żadnych)
- Parametry zwykle przekazywane są przez wartość.
- Dodając znak & przed nazwę parametru możemy wymusić przekazywanie przez referencję

Parametry funkcji – przekazywanie przez wartość

```php
<?php

function kwadrat($x) {
   $x*=$x;
   echo "$x\n";
}

$x=2;
kwadrat($x);
echo "$x\n";

?>
```

Program wyświetli wartości: 4 i 2

Parametry funkcji – przekazywanie przez referencję (1)

```php
<?php

function kwadrat(&$x) {
   $x*=$x;
   echo "$x\n";
}

$x=2;
kwadrat($x);
echo "$x\n";

?>
```

Program wyświetli wartości: 4 i 4

Parametry funkcji – przekazywanie przez referencję (2)

- Przy przekazywaniu przez referencję parametr musi być zmienną (np. ***kwadrat(5)*** jest błędne).
- Przekazywanie przez referencję pozwala modyfikować zawartość zmiennych na zewnątrz funkcji (**Uwaga! Osłabienie izolacji!**)
- Przekazywanie przez wartość wymaga kopiowania. W przypadku dużych łańcuchów znaków lub obiektów przekazywanie przez referencję jest szybsze.

Parametry domyślne funkcji

```php
<?php

function ustaw_kolor($kolor = "bialy") {
  echo "Ustawiam kolor na: $kolor \n";
}

ustaw_kolor("zielony");


ustaw_kolor();

?>
```

Użyje wartości "zielony"

Użyje wartości "biały"

Funkcja ze zmienną liczbą parametrów (1)

- PHP dopuszcza stosowanie funkcji ze zmienną liczbą parametrów – taką funkcję deklaruje się jak funkcję bez parametrów.
- W ciele funkcji można użyć funkcji wbudowanych PHP:
  - **func_get_args()** - zwraca tablice z parametrami
  - **func_num_args()** - liczba przekazanych parametrów
  - **func_get_arg(N)** – zwraca wartość N-tego parametru

Funkcja ze zmienną liczbą parametrów (2)

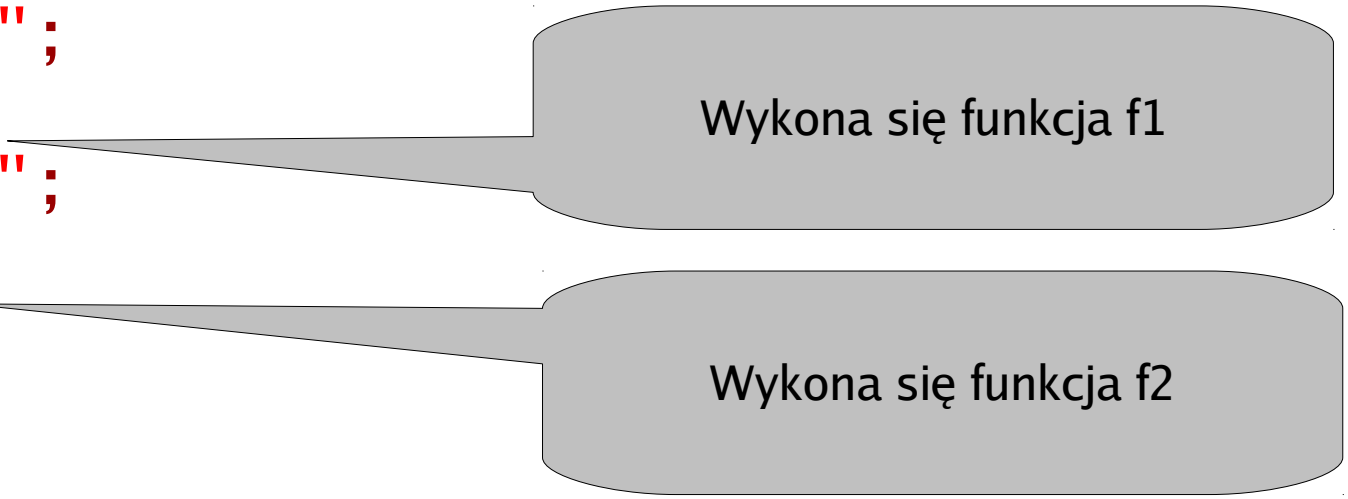
```php
<?php

function wyswietl() {
  if(func_num_args()==0) return;
  for($i=0;$i<func_num_args();$i++)
    echo func_get_arg($i);
}

wyswietl(1,2,3,4,5,"\n");
wyswietl(1,2,3,4,5,6,7,8,9,"\n");

?>
```

Brakujące parametry

- Jeżeli funkcję wywoła się bez podania wszystkich wymaganych parametrów, to PHP generuje ostrzeżenie
- W takim przypadku brakujące parametry nie są ustawiane.
- Ustawienie parametru przekazywanego do funkcji (jego obecność) można sprawdzić za pomocą funkcji wbudowanej **isset(*zmienna*)**

Funkcje zmienne (1)

```
<?php

function f1() {  echo "f1\n";  }

function f2() {  echo "f2\n"; }

$x="f1";
$x();
$x="f2";
$x();

?>
```



Funkcje zmienne (2)

- Próba wykonania nieistniejącej funkcji wygeneruje błąd PHP
- Aby tego uniknąć, warto użyć wbudowanej funkcji **function_exists(*funkcja*)** do sprawdzenia istnienia funkcji
- Jako funkcji zmiennych nie można użyć wbudowanych funkcji PHP, np. echo lub isset

Programowanie Obiektowe - Powtórzenie

Deklarowanie klasy i tworzenie obiektu

```php
<?php

class moja {
  var $atrybut;
  function metoda() { }
};

$m = new moja;

?>
```

Deklaracja klasy

Tworzenie obiektu klasy

Dostęp do składowych obiektu i klasy

- Dostęp do składowych obiektu uzyskuje się za pomocą operatora -> tzn:

```
$obiekt->atrybut
$obiekt->metoda()
```

- Dostęp do składowych statycznych klasy uzyskuje się za pomocą operatora :: tzn:

```
klasa::atrybut
klasa::metoda()
```

Widoczność składowych klasy (1)

- Składowe publiczne – **public** – dostępne z wnętrza klasy oraz poza klasą
- Składowe chronione – **protected** – dostępne z wnętrza klasy i klas dziedziczących. Niedostępne poza klasą i jej pochodnymi
- Składowe prywatne – **private** – dostępne tylko z wnętrza klasy

Widoczność składowych klasy (2)

```php
<?php

class moja {
  private $prywatny;
  protected $chroniony;
  public function publiczna() { }
};

?>
```

Wartość domyślna atrybutu

- Atrybuty mogą mieć wartość domyślną. Przyjmują ją już w momencie tworzenia obiektu.
- Składnia

```
// ...
private $prywatny = 1;
// ...
```

- Wartości domyślne atrybutów muszą być zmiennymi prostymi.

Odwoływanie się do składowych z wnętrza klasy

- Aby odwołać się do składowej klasy z jej wnętrza trzeba użyć słowa kluczowego **$this**, np.

  $this->metoda();

- Wskaźnik **$this** istnieje tylko dla obiektu (tzn. musi być utworzony za pomocą operatora new)
- W przypadku składowych statycznych składnia jest następująca:

  self::metoda()

- Zamiast **self** można użyć nazwy klasy.

Składowe statyczne (1)

- Są własnością samej klasy, a nie jej obiektów, tzn. że mogą być używane bez konieczności tworzenia obiektu
- Składowe statyczne deklaruje się za pomocą słowa **static**
- Statyczne metody nie mogą odwoływać się do klasy poprzez **$this**, mogą co najwyżej korzystać z innych składowych statycznych.

Składowe statyczne (2)

```php
<?php

class moja {
  public static $atr = 5;
  public static function atr2() {
    return self::$atr*2;
  }
};

echo moja::$atr,"\n";
echo moja::atr2(),"\n";

?>
```

Konstruktor klasy (1)

- Specjalna metoda wywoływana automatycznie w czasie tworzenia obiektu
- Nazwa: __construct
- Może mieć argumenty, nie zwraca wartości
- Może być wykorzystania do inicjalizacji obiektu (np. otwarcie połączenia do bazy danych, otwarcie pliku, przydział zasobu).

Konstruktor klasy (2)

```php
<?php

class moja {
  public function __construct() {
    echo "Inicjalizacja\n";
  }
};

$m=new moja;

?>
```

Konstruktor klasy (3)

```php
<?php

class moja {
  public function __construct($arg) {
    echo "Inicjalizacja z wartoscia $arg\n";
  }
};

$m=new moja(5);

?>
```

Destruktor klasy (1)

- Specjalna metoda wywoływana automatycznie w czasie niszczenia klasy
- Nazwa: __destruct
- Brak argumentów i wartości zwracanej
- Może być wykorzystana do “sprzątania” zasobów – np. zamykania plików lub połączenia z bazą danych.

Destruktor klasy (2)

```php
<?php

class moja {
  public function __construct() {
    echo "Inicjalizacja\n";
  }
  public function __destruct() {
    echo "Sprzatanie\n";
  }
};

$m=new moja();

?>
```

Dziedziczenie - składnia

```
class bazowa {
  public $arg;
  public function f1() { }
};

class pochodna extends bazowa {
  public function f2() { }
};

?>
```

Dziedziczenie (2)

- Klasa potomna dziedziczy wszystkie publiczne i chronione składowe klasy bazowej
- Jeżeli w klasie potomnej występuje taka sama składowa co w bazowej, to składowa klasy bazowej jest przesłaniana.
- Uwaga! Dotyczy to również konstruktora i destruktora! Przesłoniony konstruktor można wywołać jawnie (używając konstrukcji **parent::**)

Dziedziczenie (3)

```php
<?php
class bazowa {
  public function __construct() { echo "C-bazowa\n";
}
  public function f1() { echo "F1 bazowa\n"; }
}
class pochodna extends bazowa {
  public function __construct() {
    parent::__construct();
    echo "C-pochodna\n"; }
  public function f1() { echo "F1 pochodna\n"; }
};

$p = new pochodna;
$p->f1();
?>
```

Metody typu final

- Metoda zdefiniowana w klasie może być przesłonięta (nadpisana nową wersją) dzięki dziedziczeniu
- Możliwość przesłonięcia metody można zablokować.
- Użycie słowa kluczowego **final** przed definicją metody, oznacza że jest ona ostateczna i nie może być zmieniana. Próba przesłonięcia spowoduje błąd.

Klasy i metody abstrakcyjne (1)

- PHP wprowadza możliwość definiowania tzw. klas abstrakcyjnych – ich obiekty nie mogą być tworzone. Służą one jako klasy bazowe w dziedziczeniu.
- Klasa abstrakcyjna może mieć metody abstrakcyjne. Muszą one zostać przesłonięte przez metody klasy potomnej, inaczej ona sama również jest klasą abstrakcyjną.

Klasy i metody abstrakcyjne (2)

```php
<?php

abstract class A {
  abstract public function f();
}

class B extends A {
  public function f() { }
}

$k = new B();

?>
```

Klasa abstrakcyjna

Klasa konkretna

Abstrakcyjna metoda f musi być przesłonięta

Interfejsy (1)

- Niektóre języki wspierające obiektowość umożliwiają wielokrotne dziedziczenie – tzn. klasa pochodna może dziedziczyć po więcej niż jednej klasie bazowej
- Wielokrotne dziedziczenie, pomimo wygody, wprowadza też pewne problemy i dlatego w wielu językach obiektowych zrezygnowano z tej możliwości
- W PHP zamiast wielokrotnego dziedziczenia można wykorzystać interfejsy.

Interfejsy (2)

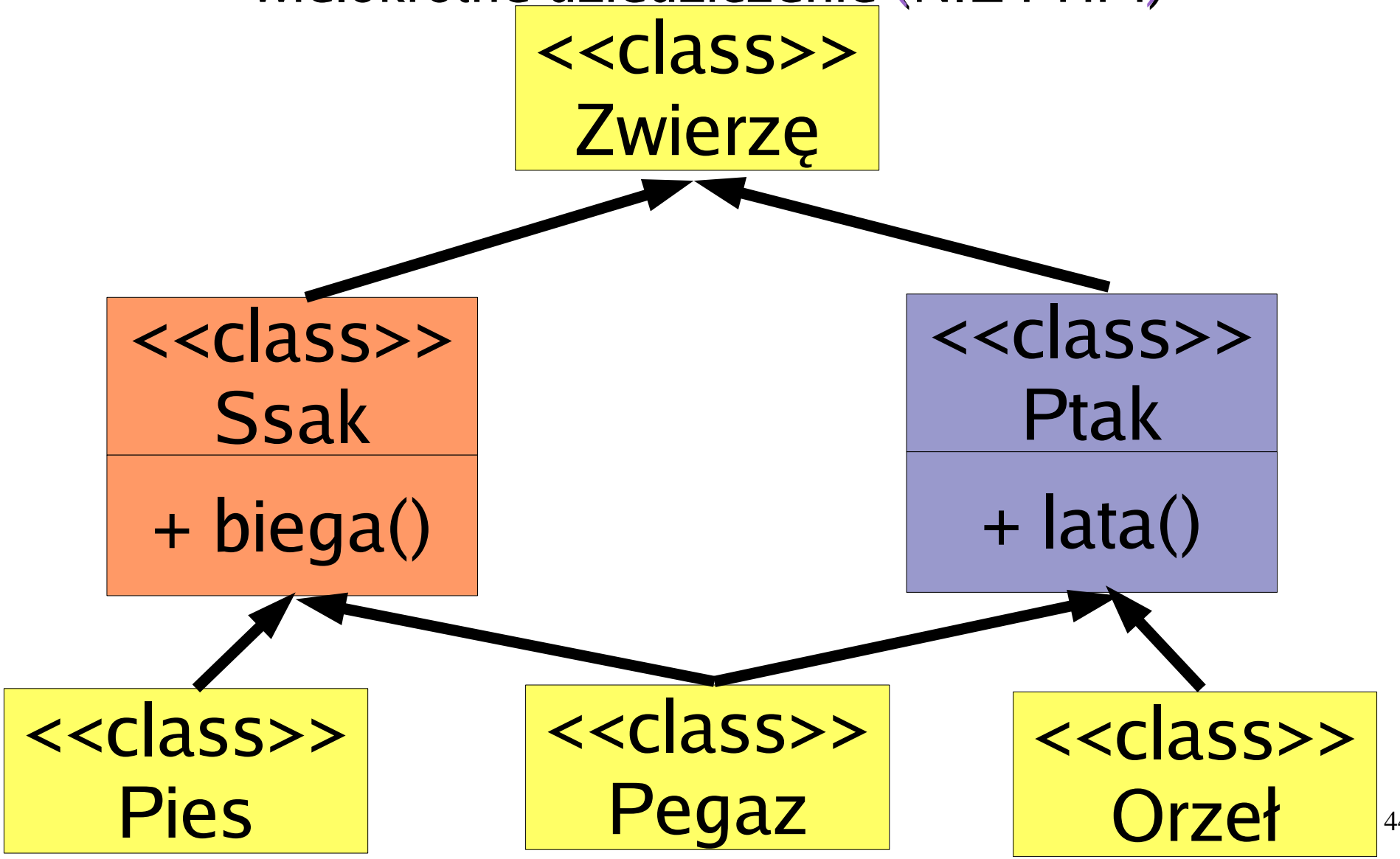
- Interfejs – pewnego rodzaju klasa abstrakcyjna
- Zawiera deklaracje metod abstrakcyjnych o dostępie publicznym. Nie zawiera innych składowych.
- Klasa może implementować dowolną liczbę interfejsów.
- Implementacja interfejsu polega na przesłonięciu wszystkich metod abstrakcyjnych deklarowanych w interfejsie.

Interfejsy (3)

```php
<?php

interface I {
  public function f();
}

class A implements I {
  public function f() { }
}

$obj = new A();

?>
```

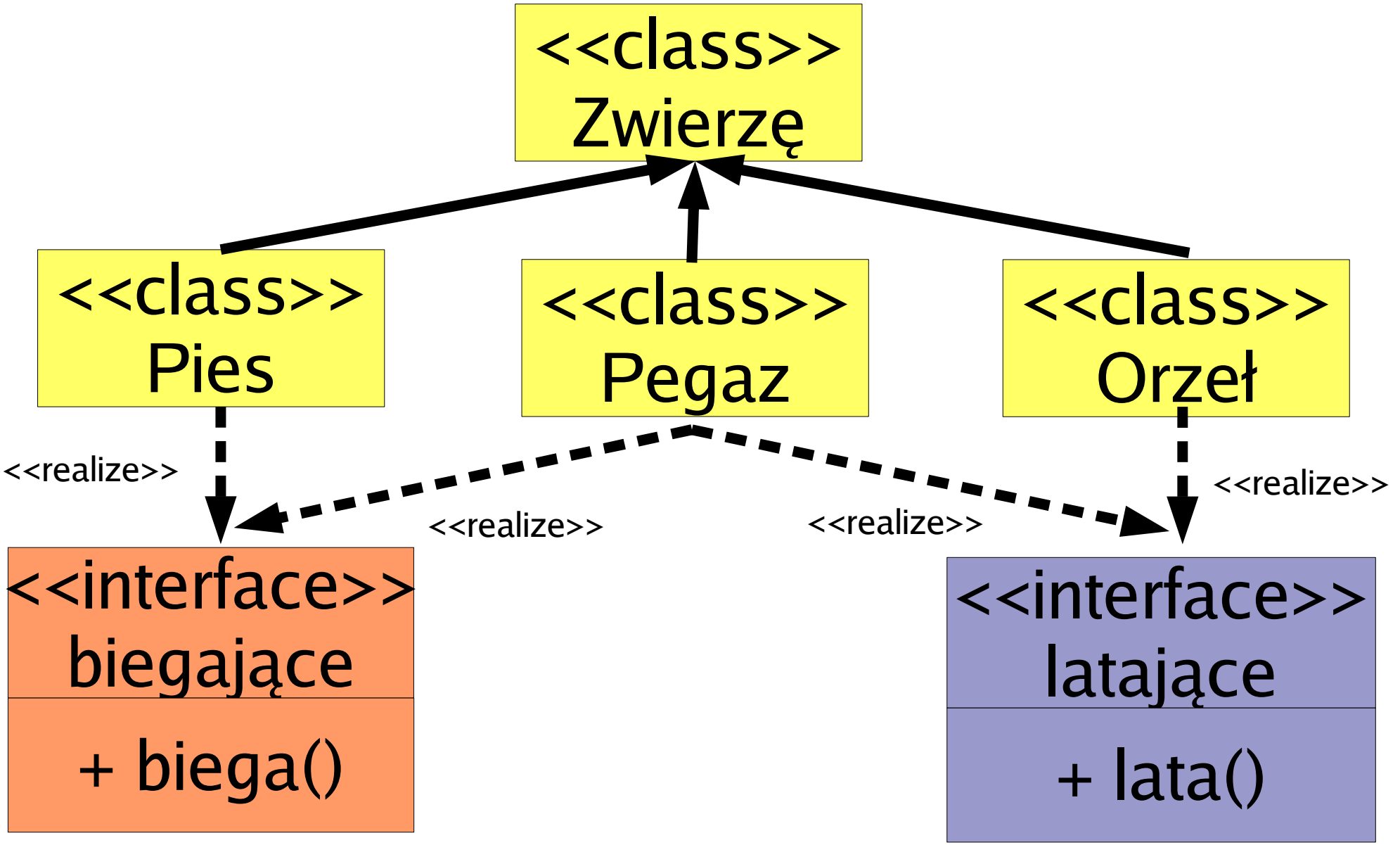
Wielokrotne dziedziczenie vs. interfejsy

wielokrotne dziedziczenie (NIE PHP!)



Wielokrotne dziedziczenie vs. interfejsy

interfejsy (PHP)



Wielokrotne użycie kodu

- Programowanie strukturalne a szczególnie obiektowe sprzyjają koncepcji wielokrotnego użycia kodu (funkcji i klas)
- W celu uniknięcia pracy związanej z kopiowaniem, definicje funkcji i klas można umieścić w osobnych plikach – biblioteki funkcji i klas.
- Biblioteki funkcji i klas można włączać do głównego skryptu

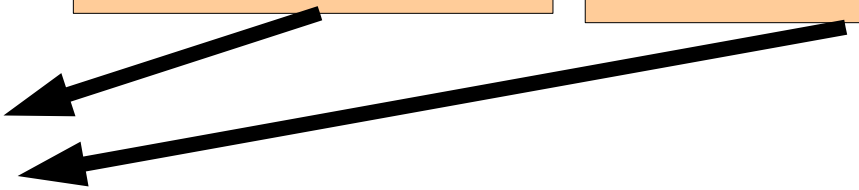
Włączanie plików (1)

- Dwa sposoby włączenia pliku z definicjami funkcji i klas do skryptu:
- **include(“*NAZWA*”)** - włącza w kod zawartość pliku o podanej nazwie. W przypadku braku pliku generuje ostrzeżenie i nie przerywa programu.
- **require(“*NAZWA*”)** - j.w. ale brak pliku powoduje błąd krytyczny (zatrzymanie programu)

Włączanie plików (2)

```php
<?php

class B {
  // ...
}

function f2() {
  // ...
}

?>
```

```php
<?php

class A {
  // ...
}

function f1() {
  // ...
}

?>
```



```php
<?php

include("plik1.php");
require("plik2.php");

// ...

?>
```

Podsumowanie

- Powtórka z zastosowania funkcji
  Funkcje, parametry funkcji, parametry domyślne, funkcje o zmiennej liczbie parametrów, funkcje zmienne
- Programowanie obiektowe
  Klasy i obiekty, widoczność składowych klasy, składowe statyczne, konstruktor, destruktor, dziedziczenie, klasy i metody abstrakcyjne, interfejsy
- Włączanie plików