Podstawy programowania III

WYKŁAD 3

Jan Kazimirski

PHP techniki zaawansowane

Obsługa błędów

- Dużą częścią każdej większej aplikacji jest obsługa błędów.
- Działanie funkcji/metody może zakończyć się niepowodzeniem z różnych przyczyn (błędne dane, niedostępne zasoby, awaria sprzętowa)
- Problem: **Jak o niepowodzeniu poinformować resztę aplikacji?**

Obsługa błędów c.d.

- Wartość zwracana.
  - Możliwe jeżeli nie wszystkie wartości zwracane mają sens wynikający z działania funkcji
  - Program wołający funkcję musi zapewnić kod identyfikujący i obsługujący błąd w miejscu wywołania funkcji
- Zmienna globalna (errno)
  - Program wołający musi sprawdzać zawartość zmiennej zaraz po powrocie z funkcji
  - Problem z programami wielowątkowymi.

Wyjątki

- Mechanizm wyjątków – obsługa błędów
- Stosowany w wielu językach obiektowych – m.in. Java, C++
- Założenia:
  - w przypadku błędu generowany jest tzw. wyjątek
  - wyjątek powinien być „złapany” tzn. obsłużony przez odpowiednią procedurę obsługi.

Wyjątki i PHP

- Predefiniowana klasa wyjątków - Exception
- Generowanie wyjątku – instrukcja throw
- Wyłapywanie wyjątków – blok try
- Obsługa wyjątków – blok catch.
- Dodatkowe mechanizmy – np. uchwyty do wyłapywania nieobsłużonych wyjątków.

[1] Generowanie wyjątku

```php
<?php

// $arg must be a positive integer
function func($arg) {

  if(!is_int($arg))throw new Exception("Argument is not an int");
  if($arg<=0)throw new Exception("Argument must be positive.");

  // rest of the function
};

$arg = "a";
func($arg);

?>
```

```
PHP Fatal error:  Uncaught exception 'Exception' with
message 'Argument is not an int' in /home/jankazim/ex1.php:6
Stack trace:
#0 /home/jankazim/ex1.php(13): func('a')
#1 {main}
  thrown in /home/jankazim/ex1.php on line 6
```

[2] Konstrukcja try - catch

```php
<?php

// $arg must be a positive integer
function func($arg) {

  // Function body [1]
};

try {
  $arg = "a";
  func($arg);
} catch (Exception $err) {
  echo "Ops!\n";
};

?>
```

```
Ops!
```

[3] Metody klasy Exception

```php
<?php
function func($arg) { // Function body [1]

try {
  $arg = "a";
  func($arg);
} catch (Exception $err) {
  echo "ERROR\n";
  echo "MSG:   ",$err->getMessage(),"\n";
  echo "CODE:  ",$err->getCode(),"\n";
  echo "FILE:  ",$err->getFile(),"\n";
  echo "LINE:  ",$err->getLine(),"\n";
  echo "TRACE: \n",$err->getTraceAsString(),"\n";
};
?>
```

```
ERROR
MSG:   Argument is not an int
CODE:  0
FILE:  /home/jankazim/ex03.php
LINE:  7
TRACE:
#0 /home/jankazim/ex03.php(15):
func('a')
#1 {main}
```

Metody klasy Exception c.d.

- *__construct($msg,$code)* – treść komunikatu i kod błędu są opcjonalne.
- *getMessage()* - zwraca treść komunikatu błędu.
- *getCode()* - zwraca kod błędu.
- *getFile()* - nazwa pliku w którym błąd wystąpił.
- *getLine()* - numer linii w której błąd wystąpił.
- *getTraceAsString()* - tzw. „ślad” (stos wywołań).

Własne wyjątki

- Mechanizm obsługi wyjątków dopuszcza możliwość tworzenia własnych klas wyjątków
- Klasy te muszą wywodzić się z klasy Exception.
- Zróżnicowane klasy wyjątków pozwalają łatwo rozróżniać typy błędów i efektywnie je obsługiwać.

[4] Definiowanie wyjątków

```php
<?php

class NotIntErr extends Exception {};
class NotPosErr extends Exception {};

// $arg must be a positive integer
function func($arg) {
  if(!is_int($arg)) throw new NotIntErr;
  if($arg<=0) throw new NotPosErr;
  // rest of the function
};

?>
```

Klasy wyjątków dziedziczą po klasie bazowej Exception

W przypadku błędu generowany jest wyjątek zdefiniowany przez użytkownika

Definiowanie wyjątków c.d.

```php
<?php

class NotIntErr extends Exception {};
class NotPosErr extends Exception {};

// $arg must be a positive integer
function func($arg) { // Function body };

try {
  $arg = -1;
  func($arg);
} catch (NotIntErr $err) {
  die("ERROR! Argument must be integer.\n");
} catch (NotPosErr $err) {
  die("ERROR! Argument must be positive.\n");
};
?>
```

[5] Definiowanie wyjątków c.d.

```php
<?php

...


try {
   $arg = -1;
   func($arg);
} catch (NotIntErr $err) {
   die("ERROR! Argument must be integer.\n");
} catch (NotPosErr $err) {
   die("ERROR! Argument must be positive.\n");
} catch (Exception $err) {
   die("Something happened!\n");
};
?>
```

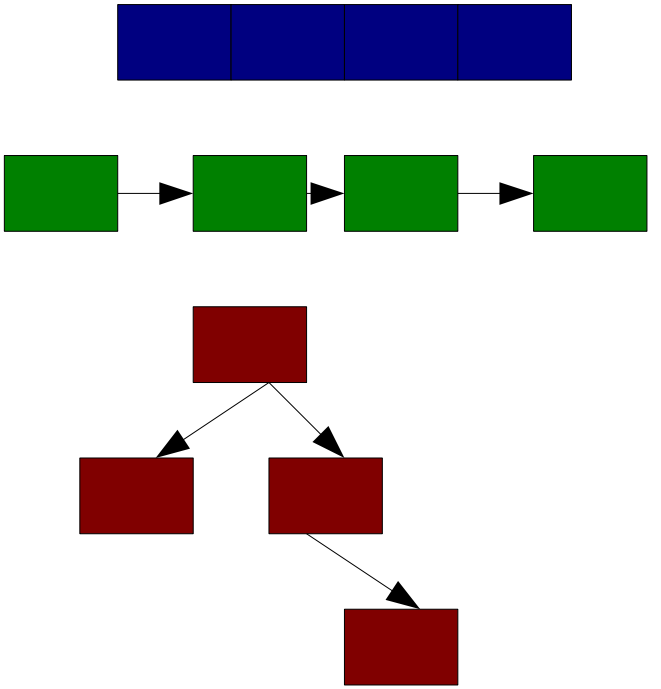
Dodatkowy blok catch wyłapuje inne wyjątki będące klasami pochodnymi od Exception

Zalety wyjątków

- Mogą być stosowane w każdej sytuacji (np. konstruktor, destruktor, funkcja nie zwracająca wartości).
- Uproszczenie kodu (odseparowanie obsługi błędów od reszty programu).
- „Wymuszenie” obsługi błędów

Iteratory

- Typowe struktury danych:
- tablice
- listy
- drzewa
- ...



- Problem: **Jak realizować dostęp sekwencyjny do elementów struktury**?

Wzorzec iteratora

- Wzorzec projektowy – **iterator**
- Zapewnia sekwencyjny dostęp do obiektów pewnej kolekcji bez wiedzy o jej wewnętrznej strukturze (implementacji)
- Podstawowe metody:
  - first – ustawia wewnętrzny wskaźnik na pierwszy obiekt.
  - next – przesuwa wewnętrzny wskaźnik na kolejny obiekt.
  - valid – czy wskaźnik wskazuje na obiekt.
  - current – zwraca obiekt wskazywany przez wewnętrzny wskaźnik.

[6] Implementacja iteratora w PHP

```
class MyCollection {

  public function __construct() {$this->ptr=0; }

  public function rewind() {$this->ptr=0; }

  public function next() {$this->ptr++; }

  public function valid() {
    return isset($this->buf[$this->ptr]);
  }

  public function key() { return $this->ptr; }
  public function current() {
    return $this->buf[$this->ptr];
  }

  private $ptr;
  private $buf = array ("a","b","c","d","e");
};
```

Implementacja iteratora w PHP c.d.

```
$mycoll = new MyCollection;

function func($t) {
  $t->rewind();
  while($t->valid()) {
    echo $t->key()," ",$t->current(),"\n";
    $t->next();
  };
};

func($mycoll);

?>
```

Interfejs - Iterator

- Funkcja ***func*** z przykładu [6] może działać na dowolnej klasie posiadającej odpowiednie metody (rewind, next itd.).
- Aby to zapewnić możemy stworzyć interfejs iteratora tak, aby każda klasa która go implementuje musiała te metody dostarczyć.
- Język PHP dostarcza nam gotowy interfejs o nazwie **Iterator**.

Wykorzystanie interfejsu Iterator

```
class MyCollection implements Iterator {

  ...

};

$mycoll = new MyCollection;

function func(Iterator $t) {
  $t->rewind();
  while($t->valid()) {
    echo $t->key()," ",$t->current(),"\n";
    $t->next();
  };
};

func($mycoll);
```

Klasa MyCollection implemntuje interfejs Iterator – oznacza to, że musi ona dostarczyć wszystkie metody które interfejs definiuje (jako abstrakcyjne)

Funkcja func jako argument wejściowy akceptuje wyłącznie klasy implementujące interfejs iterator.
Takie klasy gwarantują posiadanie odpowiednich metod dostępu

Iterator i pętla foreach

- Programista może utworzyć własny interfejs iteratora i wykorzystywać go w swoich programach.
- Korzystanie z predefiniowanego interfejsu Iterator daje nam dodatkowy bonus – możliwość korzystania z pętli foreach dla naszych własnych klas.
- Pętla foreach korzysta z iteratora (dokładnie, interfejsu Traversable po którym dziedziczy interfejs Iterator

[8] Pętla foreach i iterator

```php
<?php

class myCollection implements Iterator {

    ...

};

$mycoll = new MyCollection;

foreach($mycoll as $k=>$v) echo "$k => $v\n";

?>
```

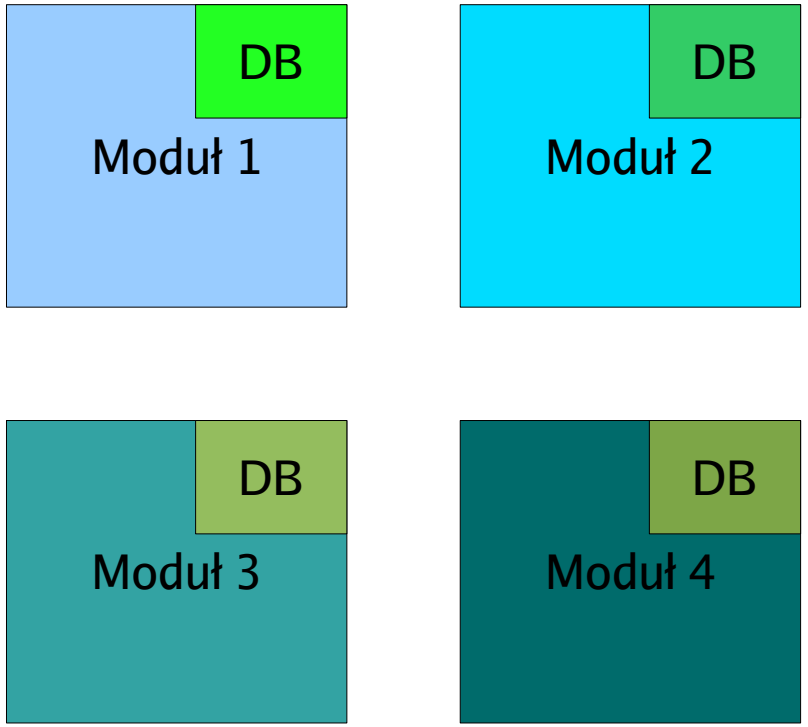
Zastosowanie klasy implementującej interfejs Iterator pozwala użyć jej w pętli foreach

Problem

- Realizujemy dużą (wielo-modułową) aplikację korzystającą z bazy danych.
- Dostęp do bazy danych – blok kodu realizujący połączenie z bazą, weryfikację połączenia, realizację kwerendy itd.
- Problem:
  Jak zaprojektować dostęp do bazy danych w poszczególnych modułach?

```
$conn = pg_connect("host=localhost
                    dbname=jankazim
                    user=jankazim");
if(!$conn)
  die("Blad polaczenia z baza.\n");
```

Rozwiązanie I



W każdym module jest kod realizujący dostęp do bazy danych.

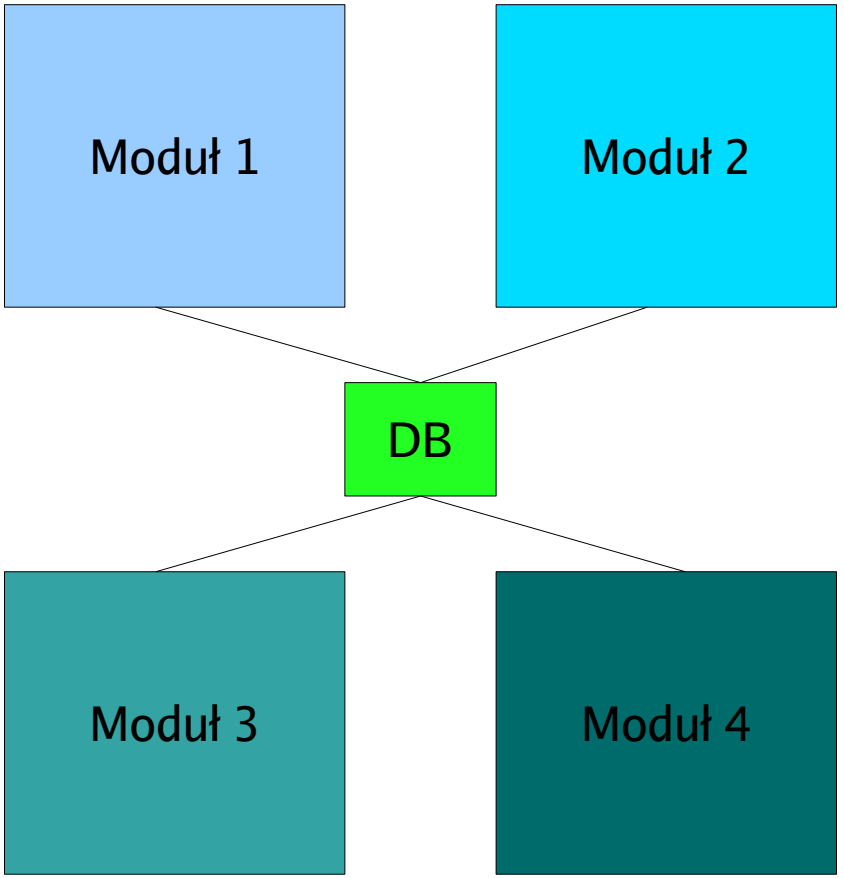
PLUSY

+ Moduły są niezależne

MINUSY

- Ten sam kod jest powtarzany wielokrotnie
- Kod jest trudny do utrzymania (trudno wprowadzać zmiany – muszą być wprowadzane w wielu miejscach – ryzyko desynchronizacji).
- Generowanie wielu niepotrzebnych połączeń z bazą danych

Rozwiązanie II



Dostęp do bazy danych realizowany jest przez osobny obiekt z którym współpracują poszczególne moduły

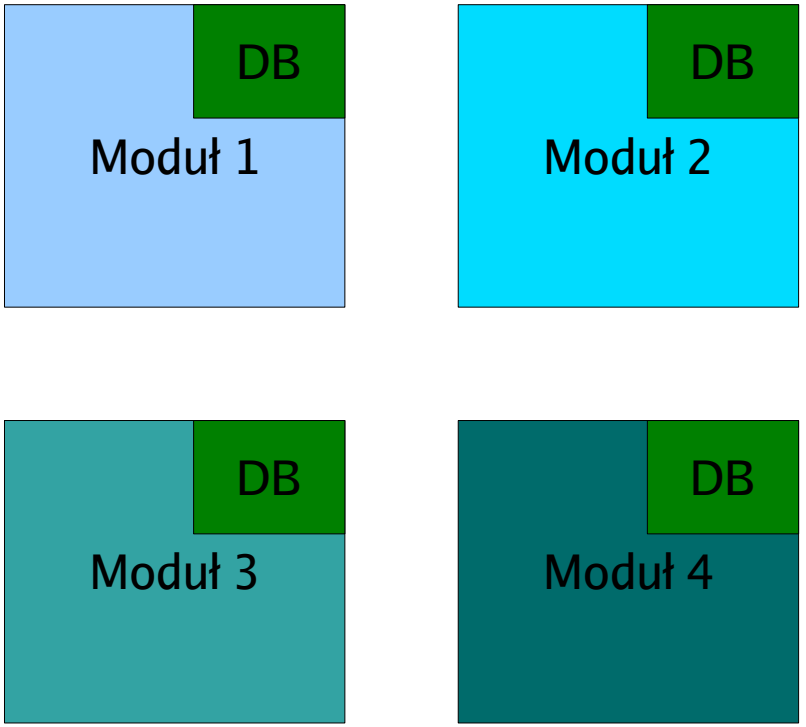
PLUSY

+ Kod obsługi bazy danych jest w jednym miejscu – jest łatwy do testowania i utrzymywania
+ Jedno połączenie obsługuje całą aplikację

MINUSY

- Każda funkcja korzystająca z bazy danych musi mieć dostęp do obiektu DB.
- Wszystkie moduły są powiązane poprzez klasę DB (niewygodna praca zespołowa)

Rozwiązanie III - Singleton



Wzorzec projektowy – Singleton

Każdy moduł ma własną instancję klasy DB.

Tak naprawdę jednak wszystkie korzystają z jednej i tej samej instancji (obiektu)

Singleton

- Jeden z wzorców projektowych.
- Zapewnia globalny dostęp do obiektu zapewniając jednocześnie, że stworzona będzie tylko jedna instancja.
- Pozwala na „leniwe” inicjowanie obiektu (dopiero kiedy jest potrzebny).
- UWAGA! Niektórzy zalecają unikania Singletonów ze względu na łamanie zasad projektowania obiektowego (odpowiednik zmiennej globalnej)
- UWAGA! Niebezpieczny w aplikacjach wielowątkowych.

[9] Implementacja Singletona w PHP

```
class Singleton {

  private static $mInstance = false;

  private function __construct() {}
  private function __clone() {}

  public static function getInstance() {
    if(self::$mInstance==false)
      self::$mInstance = new Singleton;
    return self::$mInstance;
  }

  public $data = 1;

};
```

Prywatna składowa statyczna (tzn. będąca własnością wspólną wszystkich instancji klasy, a nie indywidualną każdego obiektu)

Implementacja Singletona w PHP

```
class Singleton {

  private static $mInstance = false;

  private function __construct() {}
  private function __clone() {}

  public static function getInstance() {
    if(self::$mInstance==false)
      self::$mInstance = new Singleton;
    return self::$mInstance;
  }

  public $data = 1;

};
```

Prywatny konstruktor oraz metoda __clone powodują, że nie można stworzyć instancji Singletona z użyciem operatora new, ani poprzez kopiowanie go z istniejącego obiektu.

Implementacja Singletona w PHP

```
class Singleton {

  private static $mInstance = false;

  private function __construct() {}
  private function __clone() {}

  public static function getInstance() {
     if(self::$mInstance==false)
       self::$mInstance = new Singleton;
     return self::$mInstance;
   }

  public $data = 1;

};
```

Metoda getInstance – statyczna metoda publiczna zwracająca instancję klasy Singleton.

Przy pierwszym wywołaniu tworzony jest obiekt klasy Singleton, zapamiętywany w zmiennej $mInstance i zwracany do programu wywołującego

Kolejne wywołania metody getInstance zwracają referencję do wcześniej utworzonego obiektu

Implementacja Singletona w PHP

```
class Singleton {

  private static $mInstance = false;

  private function __construct() {}
  private function __clone() {}

  public static function getInstance() {
    if(self::$mInstance==false)
      self::$mInstance = new Singleton;
    return self::$mInstance;
  }

  public $data = 1;

};
```

Właściwa „zawartość” Singletona, zależna od potrzeb, np. kod tworzący połączenie z bazą danych, otwierający plik log itp.

Singleton – przykład działania

```
class Singleton {
  ...
};

function func1() {
  $sing = Singleton::getInstance();
  $sing->data=5;
}

function func2() {
  $sing = Singleton::getInstance();
  echo $sing->data,"\n";
}

func1();
func2();
```

Funkcje func1 i func2 korzystają z instancji Singletona utworzonych za pomocą metody Singleton::getInstance.

Każda funkcja jest niezależna a jednak obie korzystają z tej samej danej (singleton)

Rezultatem wykonania drugiej funkcji będzie wyświetlenie liczby 5.

Podsumowanie

- Obsługa wyjątków w PHP
- Wzorzec Iterator
- Wzorzec Singleton