Programowanie funkcyjne w PHP

Maciek Godek

{ **fido** : *labs* }

godek.maciek@gmail.com

PHP3City Meetup#4, 06.12.2013

Czym jest programowanie funkcyjne?

Programowanie funkcyjne (FP) polega na:

- unikaniu skutków ubocznych,
- stosowaniu funkcji wyższego rzędu.

Czym jest programowanie funkcyjne?

Programowanie funkcyjne (FP) polega na:

- unikaniu skutków ubocznych,
- stosowaniu funkcji wyższego rzędu.

Czym jest programowanie funkcyjne?

Programowanie funkcyjne (FP) polega na:

- unikaniu skutków ubocznych,
- stosowaniu funkcji wyższego rzędu.

Sformułowanie problemu

Jaki problem chcemy rozwiązać?

- Uczynić programowanie jak maksymalnie efektywnym, czyli:
  - musieć jak najmniej pamiętać, oraz
  - móc jak najwięcej wnioskować.

Sformułowanie problemu

Jaki problem chcemy rozwiązać?

- Uczynić programowanie jak maksymalnie efektywnym, czyli:
  - musieć jak najmniej pamiętać, oraz
  - móc jak najwięcej wnioskować.

Sformułowanie problemu

Jaki problem chcemy rozwiązać?

- Uczynić programowanie jak maksymalnie efektywnym, czyli:
  - musieć jak najmniej pamiętać, oraz
  - móc jak najwięcej wnioskować.

Sformułowanie problemu

Jaki problem chcemy rozwiązać?

- Uczynić programowanie jak maksymalnie efektywnym, czyli:
  - musieć jak najmniej pamiętać, oraz
  - móc jak najwięcej wnioskować.

Sformułowanie problemu

Innymi słowy: **uczynić program jak najbardziej czytelnym**

Sformułowanie problemu

Innymi słowy: **uczynić program jak najbardziej czytelnym**

Sposoby na usprawnienie pracy z kodem

Pomysły:

- korzystanie z Integrated Development Environment
- używanie statycznej kontroli typów
- stosowanie konwencji w nazewnictwie obiektów
- ogólniej: **przestrzeganie konwencji w sposobie użycja języka**

Sposoby na usprawnienie pracy z kodem

Pomysły:

- korzystanie z Integrated Development Environment
- używanie statycznej kontroli typów
- stosowanie konwencji w nazewnictwie obiektów
- ogólniej: **przestrzeganie konwencji w sposobie użycja języka**

Sposoby na usprawnienie pracy z kodem

Pomysły:

- korzystanie z Integrated Development Environment
- używanie statycznej kontroli typów
- stosowanie konwencji w nazewnictwie obiektów
- ogólniej: **przestrzeganie konwencji w sposobie użycja języka**

Sposoby na usprawnienie pracy z kodem

Pomysły:

- korzystanie z Integrated Development Environment
- używanie statycznej kontroli typów
- stosowanie konwencji w nazewnictwie obiektów
- ogólniej: **przestrzeganie konwencji w sposobie użycja języka**

Sposoby na usprawnienie pracy z kodem

Pomysły:

- korzystanie z Integrated Development Environment
- używanie statycznej kontroli typów
- stosowanie konwencji w nazewnictwie obiektów
- ogólniej: **przestrzeganie konwencji w sposobie użycja języka**

Konwencje w sposobie użycia języka

Kilka popularnych konwencji:

- notacja węgierska
- metodologia OOP (hermetyzacja, enkapsulacja, polimorfizm, dziedziczenie, ...)
- **programowanie funkcyjne**

Konwencje w sposobie użycia języka

Kilka popularnych konwencji:

- notacja węgierska
- metodologia OOP (hermetyzacja, enkapsulacja, polimorfizm, dziedziczenie, ...)
- **programowanie funkcyjne**

Konwencje w sposobie użycia języka

Kilka popularnych konwencji:

- notacja węgierska
- metodologia OOP (hermetyzacja, enkapsulacja, polimorfizm, dziedziczenie, ...)
- **programowanie funkcyjne**

Konwencje w sposobie użycia języka

Kilka popularnych konwencji:

- notacja węgierska
- metodologia OOP (hermetyzacja, enkapsulacja, polimorfizm, dziedziczenie, ...)
- **programowanie funkcyjne**

Zagadka: co robi poniższy kod?

```
function złoncz($słowa) {
    $wynik = '';
    $n = count($słowa);
    if($n < 1) { return $wynik; }
    $bieżące = reset($słowa);
    for($i = 0; $i < $n; $i++) {
        $wynik .= $bieżące;
        if($i < $n-2) { $wynik .= ', '; }
        elseif($i == $n-2) { $wynik .= ' i '; }
        $bieżące = next($słowa);
    }
    return $wynik;
}
```

Zagadka: co robi poniższy kod?

```
function złoncz($słowa) {
    $wynik = '';
    $n = count($słowa);
    if($n < 1) { return $wynik; }
    $bieżące = reset($słowa);
    for($i = 0; $i < $n; $i++) {
        $wynik .= $bieżące;
        if($i < $n-2) { $wynik .= ', '; }
        elseif($i == $n-2) { $wynik .= ' i '; }
        $bieżące = next($słowa);
    }
    return $wynik;
}
```

Rozwiązanie: łączenie słów naturalnymi spójnikami

Rozwiązanie: złącza słowa spójnikami, w taki sposób, w jaki łączymy słowa w mowie potocznej, tzn. sklejamy wszystkie słowa przecinkami, za wyjątkiem dwóch ostatnich, które sklejamy spójnikiem ’i’.

Przykład użycia

```
złączenie([]) = '';
złączenie(['Ola']) = 'Ola';
złączenie(['Ola', 'Ala']) = 'Ola i Ala';
złączenie(['Ola', 'Ala', 'Ula']) = 'Ola, Ala i Ula'
złączenie(['Ola', 'Ala', 'Ula', 'Ela'])
    = 'Ola, Ala, Ula i Ela';
```

Abstrakcja!

```
złączenie([]) = '';
złączenie([$a]) = $a;
złączenie([$a, $b]) = $a . ' i ' .$b;
złączenie([$a, $b, $c, ...])
    = $a . ', ' . złączenie([$b, $c, ...]);
```

W stronę PHP

```
function złączone($słowa) { // marzeń
    match($słowa) {
    case []:
        return '';
    case [$a]:
        return $a;
    case [$a, $b]:
        return $a . ' i ' . $b;
    case [$a, $b, $c, ...]:
        return $a . ', '
            . złączone([$b, $c, ...]);
    }
}
```

Faktyczna implementacja w PHP

```
function złączone($słowa) {
    switch(count($słowa)) {
    case 0:
        return '';
    case 1:
        return array_pop($a);
    default:
        $ostatnie = array_pop($słowa);
        return implode(', ', $słowa)
            . ' i ' . $ostatnie;
    }
}
```

Kolejna zagadka!

Jak posortować tablicę stringów
w PHP według ich długości?

Kolejna zagadka!

Jak posortować tablicę stringów
w PHP według ich długości?

Pierwsza próba:

```
usort($tablica, function($a, $b) {
    return length($a) - length($b);
});
```

Pytanie: **czy powyższe wywołanie posortuje tablicę rosnąco, czy malejąco?**

Pierwsza próba:

```
usort($tablica, function($a, $b) {
    return length($a) - length($b);
});
```

Pytanie: **czy powyższe wywołanie posortuje tablicę rosnąco, czy malejąco?**

Druga próba:

```
usort($tablica, ascending('length'));
```

gdzie

```
function ascending($property) {
    return function($a, $b) use($property) {
        return $property($a) - $property($b);
    };
}
```

Druga próba:

```
usort($tablica, ascending('length'));
```

gdzie

```
function ascending($property) {
    return function($a, $b) use($property) {
        return $property($a) - $property($b);
    };
}
```

Sortowanie malejąco

```
function descending ($prop) {
    return function($a, $b) use($prop) {
        return -call_user_func(ascending($prop),
                               $a, $b);
    };
}
```

jeżeli `is_callable($f)`,
to zapis `call_user_func($f, $argumenty ...)`
jest równoważny zapisowi `$f($argumenty ...)`

Sortowanie malejąco

```
function descending ($prop) {
    return function($a, $b) use($prop) {
        return -call_user_func(ascending($prop),
                               $a, $b);
    };
}
```

jeżeli `is_callable($f)`,
to zapis `call_user_func($f, $argumenty ...)`
jest równoważny zapisowi `$f($argumenty ...)`

Popularne funkcje wyższego rzędu

- `call_user_func_array($function, $array)`
- `array_map($callback, $a1, $a2, ...)`
- `array_filter($array, $callback)`
- `array_reduce($callback, $array [, $init])`

Popularne funkcje wyższego rzędu

- `call_user_func_array($function, $array)`
- `array_map($callback, $a1, $a2, ...)`
- `array_filter($array, $callback)`
- `array_reduce($callback, $array [, $init])`

Popularne funkcje wyższego rzędu

- `call_user_func_array($function, $array)`
- `array_map($callback, $a1, $a2, ...)`
- `array_filter($array, $callback)`
- `array_reduce($callback, $array [, $init])`

Popularne funkcje wyższego rzędu

- `call_user_func_array($function, $array)`
- `array_map($callback, $a1, $a2, ...)`
- `array_filter($array, $callback)`
- `array_reduce($callback, $array [, $init])`

`call_user_func_array` – synonimy 1

Zastosowanie `call_user_func_array` – tworzenie synonimów:

```
function apply() {
    return call_user_func_array(
        'call_user_func_array',
        func_get_args()
    );
}
```

`call_user_func_array` – synonimy 1

Zastosowanie `call_user_func_array` – tworzenie synonimów:

```
function apply() {
    return call_user_func_array(
        'call_user_func_array',
        func_get_args()
    );
}
```

`call_user_func_array` – synonimy 2

Zastosowanie `call_user_func_array` – tworzenie synonimów:

```
function map() {
    return apply('array_map', func_get_args());
}


function filter() {
    return apply('array_filter', func_get_args());
}


function reduce() {
    return apply('array_reduce', func_get_args());
}
```

`call_user_func_array` – synonimy 2

Zastosowanie `call_user_func_array` – tworzenie synonimów:

```
function map() {
    return apply('array_map', func_get_args());
}


function filter() {
    return apply('array_filter', func_get_args());
}


function reduce() {
    return apply('array_reduce', func_get_args());
}
```

`call_user_func_array` – synonimy 2

Zastosowanie `call_user_func_array` – tworzenie synonimów:

```
function map() {
    return apply('array_map', func_get_args());
}


function filter() {
    return apply('array_filter', func_get_args());
}


function reduce() {
    return apply('array_reduce', func_get_args());
}
```

`call_user_func_array` – synonimy 2

Zastosowanie `call_user_func_array` – tworzenie synonimów:

```
function map() {
    return apply('array_map', func_get_args());
}


function filter() {
    return apply('array_filter', func_get_args());
}


function reduce() {
    return apply('array_reduce', func_get_args());
}
```

`call_user_func_array` – synonimy 3

Zastosowanie `call_user_func_array` do tworzenia synonimów – **problemy**:

- brak argumentów formalnych – utrudnia współpracę z IDE
- brak obsługi referencji

```
function pop() {
    return apply('array_pop', func_get_args());
} // źle!!!
```

- generuje narzuty wywołań (kod działa wolniej)
- a co z obsługą argumentów domyślnych?

`call_user_func_array` – synonimy 3

Zastosowanie `call_user_func_array` do tworzenia synonimów – **problemy**:

- brak argumentów formalnych – utrudnia współpracę z IDE
- brak obsługi referencji

```
function pop() {
    return apply('array_pop', func_get_args());
} // źle!!!
```

- generuje narzuty wywołań (kod działa wolniej)
- a co z obsługą argumentów domyślnych?

`call_user_func_array` – synonimy 3

Zastosowanie `call_user_func_array` do tworzenia synonimów – **problemy**:

- brak argumentów formalnych – utrudnia współpracę z IDE
- brak obsługi referencji

```
function pop() {
    return apply('array_pop', func_get_args());
} // źle!!!
```

- generuje narzuty wywołań (kod działa wolniej)
- a co z obsługą argumentów domyślnych?

`call_user_func_array` – synonimy 3

Zastosowanie `call_user_func_array` do tworzenia synonimów – **problemy**:

- brak argumentów formalnych – utrudnia współpracę z IDE
- brak obsługi referencji

```
function pop() {
    return apply('array_pop', func_get_args());
} // źle!!!
```

- generuje narzuty wywołań (kod działa wolniej)
- a co z obsługą argumentów domyślnych?

`call_user_func_array` – synonimy 3

Zastosowanie `call_user_func_array` do tworzenia synonimów – **problemy**:

- brak argumentów formalnych – utrudnia współpracę z IDE
- brak obsługi referencji

```
function pop() {
    return apply('array_pop', func_get_args());
} // źle!!!
```

- generuje narzuty wywołań (kod działa wolniej)
- a co z obsługą argumentów domyślnych?

`call_user_func_array` – synonimy 3

Zastosowanie `call_user_func_array` do tworzenia synonimów – **problemy**:

- brak argumentów formalnych – utrudnia współpracę z IDE
- brak obsługi referencji

```
function pop() {
    return apply('array_pop', func_get_args());
} // źle!!!
```

- generuje narzuty wywołań (kod działa wolniej)
- a co z obsługą argumentów domyślnych?

`call_user_func_array` – synonimy 4

Zagadka: **jak stworzyć synonim dla** `func_get_args`?

`array_map` – przykład

`map`: odwzorowanie zbioru funkcją. Na przykład, chcąc usunąć białe znaki ze stringów będących elementami tablicy przy pomocy funkcji trim, możemy napisać:

```
$ogolone_napisy = map('trim', $napisy)
```

`array_map` – przykład

`map`: odwzorowanie zbioru funkcją. Na przykład, chcąc usunąć białe znaki ze stringów będących elementami tablicy przy pomocy funkcji trim, możemy napisać:

```
$ogolone_napisy = map('trim', $napisy)
```

`array_map` – zagadka

Zagadka: **Chcemy wiedzieć, jaka jest największa długość stringu w danej tablicy (stringów).**

`array_map` – rozwiązanie

Rozwiązanie: `max(map('length', $tablica))`

Uwaga! To jest po prostu zdanie „największa spośród długości elementów w `$tablicy`” wyrażone w PHP!

`array_map` – rozwiązanie

Rozwiązanie: `max(map('length', $tablica))`

Uwaga! To jest po prostu zdanie „największa spośród długości elementów w `$tablicy`” wyrażone w PHP!

`array_map` – ciekawostka

Ciekawostka:

`map('f', map('g', $a)) = map('f' ∘ 'g', $a),`

gdzie symbol ∘ oznacza złożenie funkcji, tj. $f(g(x))$

Zadanie: napisz funkcję 'compose', która pobiera jako argumenty funkcje jednoargumentowe i zwraca funkcję będącą ich złożeniem

`array_map` – ciekawostka

Ciekawostka:

`map('f', map('g', $a)) = map('f' ∘ 'g', $a),`

gdzie symbol ∘ oznacza złożenie funkcji, tj. $f(g(x))$

Zadanie: napisz funkcję 'compose', która pobiera jako argumenty funkcje jednoargumentowe i zwraca funkcję będącą ich złożeniem

`array_filter` – przykład

`filter`: odsiewanie elementów spełniających dane kryterium,
np. chcemy znaleźć wszystkie liczby z danej tablicy,
mieszczące się w przedziale (2, 5)

```
$liczby_2_5 = filter($różne_liczby, function($x) {
    return (2 < $x) && ($x < 5);
});
```

`array_filter` – przykład

`filter`: odsiewanie elementów spełniających dane kryterium, np. chcemy znaleźć wszystkie liczby z danej tablicy, mieszczące się w przedziale (2, 5)

```
$liczby_2_5 = filter($różne_liczby, function($x) {
    return (2 < $x) && ($x < 5);
});
```

`array_filter` – przykład

`filter`: odsiewanie elementów spełniających dane kryterium, np. chcemy znaleźć wszystkie liczby z danej tablicy, mieszczące się w przedziale (2, 5)

```
$liczby_2_5 = filter($różne_liczby, function($x) {
    return (2 < $x) && ($x < 5);
});
```

`array_reduce` – wzmianka

`reduce`: funkcyjna bestia

zastosowanie: uogólnianie dwuargumentowych działań łącznych na dowolną ilość argumentów

`array_reduce` – wzmianka

`reduce`: funkcyjna bestia
zastosowanie: uogólnianie dwuargumentowych działań łącznych na dowolną ilość argumentów

Składnia $_

Problem: mamy tablicę obiektów. Chcielibyśmy mieć tablicę rzutów tych obiektów na ich pewne własności, które możemy otrzymać przez wywołanie odpowiedniej metody.

Klasyczne rozwiązanie:

```
$rzuty = [];
foreach($obiekty as $obiekt) {
    $rzuty[] = $obiekt->pobierzWłasność();
}
```

Funkcyjne rozwiązanie:

```
global $_; // drobne oszustwo
$rzuty = map($_->pobierzWłasność(), $obiekty);
```

Składnia $_

Problem: mamy tablicę obiektów. Chcielibyśmy mieć tablicę rzutów tych obiektów na ich pewne własności, które możemy otrzymać przez wywołanie odpowiedniej metody.
Klasyczne rozwiązanie:

```
$rzuty = [];
foreach($obiekty as $obiekt) {
    $rzuty[] = $obiekt->pobierzWłasność();
}
```

Funkcyjne rozwiązanie:

```
global $_; // drobne oszustwo
$rzuty = map($_->pobierzWłasność(), $obiekty);
```

Składnia $_

Problem: mamy tablicę obiektów. Chcielibyśmy mieć tablicę rzutów tych obiektów na ich pewne własności, które możemy otrzymać przez wywołanie odpowiedniej metody.
Klasyczne rozwiązanie:

```
$rzuty = [];
foreach($obiekty as $obiekt) {
    $rzuty[] = $obiekt->pobierzWłasność();
}
```

Funkcyjne rozwiązanie:

```
global $_; // drobne oszustwo
$rzuty = map($_->pobierzWłasność(), $obiekty);
```

Składnia $_

Problem: mamy tablicę obiektów. Chcielibyśmy mieć tablicę rzutów tych obiektów na ich pewne własności, które możemy otrzymać przez wywołanie odpowiedniej metody.
Klasyczne rozwiązanie:

```
$rzuty = [];
foreach($obiekty as $obiekt) {
    $rzuty[] = $obiekt->pobierzWłasność();
}
```

Funkcyjne rozwiązanie:

```
global $_; // drobne oszustwo
$rzuty = map($_->pobierzWłasność(), $obiekty);
```

Składnia $_

Inny problem: sortowanie tablicy tablic względem danej kolumny:

```
usort($array, ascending($_['column']));
```

Składnia $_

Inny problem: sortowanie tablicy tablic względem danej kolumny:

```
usort($array, ascending($_['column']));
```

Podsumowanie

Zalety FP:

- łatwiejsza analiza kodu
- dobra kompozycjonalność (nie trzeba się przejmować kopiowaniem i referencjami)
- bardzo naturalne testowanie
- uproszczona paralelizacja

Wady FP:

- nie wszędzie daje się stosować
- brak wsparcia ze strony narzędzi
- kod może działać niewydajnie

Podsumowanie

Zalety FP:

- łatwiejsza analiza kodu
- dobra kompozycjonalność (nie trzeba się przejmować kopiowaniem i referencjami)
- bardzo naturalne testowanie
- uproszczona paralelizacja

Wady FP:

- nie wszędzie daje się stosować
- brak wsparcia ze strony narzędzi
- kod może działać niewydajnie

Podsumowanie

Zalety FP:

- łatwiejsza analiza kodu
- dobra kompozycjonalność (nie trzeba się przejmować kopiowaniem i referencjami)
- bardzo naturalne testowanie
- uproszczona paralelizacja

Wady FP:

- nie wszędzie daje się stosować
- brak wsparcia ze strony narzędzi
- kod może działać niewydajnie

Podsumowanie

Zalety FP:

- łatwiejsza analiza kodu
- dobra kompozycjonalność (nie trzeba się przejmować kopiowaniem i referencjami)
- bardzo naturalne testowanie
- uproszczona paralelizacja

Wady FP:

- nie wszędzie daje się stosować
- brak wsparcia ze strony narzędzi
- kod może działać niewydajnie

Podsumowanie

Zalety FP:

- łatwiejsza analiza kodu
- dobra kompozycjonalność (nie trzeba się przejmować kopiowaniem i referencjami)
- bardzo naturalne testowanie
- uproszczona paralelizacja

Wady FP:

- nie wszędzie daje się stosować
- brak wsparcia ze strony narzędzi
- kod może działać niewydajnie

Podsumowanie

Zalety FP:

- łatwiejsza analiza kodu
- dobra kompozycjonalność (nie trzeba się przejmować kopiowaniem i referencjami)
- bardzo naturalne testowanie
- uproszczona paralelizacja

Wady FP:

- nie wszędzie daje się stosować
- brak wsparcia ze strony narzędzi
- kod może działać niewydajnie

Podsumowanie

Zalety FP:

- łatwiejsza analiza kodu
- dobra kompozycjonalność (nie trzeba się przejmować kopiowaniem i referencjami)
- bardzo naturalne testowanie
- uproszczona paralelizacja

Wady FP:

- nie wszędzie daje się stosować
- brak wsparcia ze strony narzędzi
- kod może działać niewydajnie

Podsumowanie

Zalety FP:

- łatwiejsza analiza kodu
- dobra kompozycjonalność (nie trzeba się przejmować kopiowaniem i referencjami)
- bardzo naturalne testowanie
- uproszczona paralelizacja

Wady FP:

- nie wszędzie daje się stosować
- brak wsparcia ze strony narzędzi
- kod może działać niewydajnie

Podsumowanie

Zalety FP:

- łatwiejsza analiza kodu
- dobra kompozycjonalność (nie trzeba się przejmować kopiowaniem i referencjami)
- bardzo naturalne testowanie
- uproszczona paralelizacja

Wady FP:

- nie wszędzie daje się stosować
- brak wsparcia ze strony narzędzi
- kod może działać niewydajnie

Podsumowanie

Zalety FP:

- łatwiejsza analiza kodu
- dobra kompozycjonalność (nie trzeba się przejmować kopiowaniem i referencjami)
- bardzo naturalne testowanie
- uproszczona paralelizacja

Wady FP:

- nie wszędzie daje się stosować
- brak wsparcia ze strony narzędzi
- kod może działać niewydajnie

Lektury

Wpiszcie sobie w Google:
John Carmack Functional Programming

Książki:
„Struktura i Interpretacja Programów Komputerowych”,
H. Abelson, G. Sussman

Lektury

Wpiszcie sobie w Google:
John Carmack Functional Programming

Książki:
„Struktura i Interpretacja Programów Komputerowych”,
H. Abelson, G. Sussman

Dziękuję!

https://joind.in/10258

godek.maciek@gmail.com