

Zaawansowany PHP



v 1.6

Plan

- Interfejsy, klasy abstrakcyjne i finalne
 - Funkcja autoload
 - Zaawansowane działania na stringach
 - Wyrażenia regularne
 - Pliki
 - Wyjątki
- Filtry
 - Mail
 - XML
 - JSON
 - Header



Interfejsy, klasy abstrakcyjne i finalne

Klasy abstrakcyjne

- Klasy abstrakcyjne nie zawierają implementacji części (lub wszystkich) metod.
- Metody abstrakcyjne zawierają jedynie definicję samej metody (widoczność, liczbę i rodzaj argumentów) bez jej kodu.

- Obiekty tworzy się z klas dziedziczących zawierających implementacje metod abstrakcyjnych.

**Z klasy abstrakcyjnej
nie można utworzyć obiektu.**

Klasy abstrakcyjne

Definicja

```
abstract class Vehicle
{
    public function go($speed) {
        $this->startEngine();
    }
    public function stop() {
        $this->stopEngine();
    }
    abstract protected function startEngine();
    abstract protected function stopEngine();
}
```

Wykorzystanie

```
class Car extends Vehicle
{
    protected function startEngine() {
        ...
    }
    protected function stopEngine() {
        ...
    }
}
```

Klasy abstrakcyjne

Zgodność dostępu

- Dziedzicząc po klasie abstrakcyjnej, musimy zachować **zgodność atrybutów dostępu**.
- Jeżeli klasa abstrakcyjna definiuje jakąś metodę jako publiczną, to w naszej klasie ta metoda musi również zostać zdefiniowana jako publiczna.
- Nasze metody muszą mieć widoczność taką samą lub większą niż metody z klasy abstrakcyjnej.

Zgodność argumentów

- Dziedzicząc po klasie abstrakcyjnej, musimy zachować nie tylko zgodność dostępu, lecz także **typów i liczby argumentów**.
- Jeżeli klasa abstrakcyjna określa typ danego argumentu, to metoda w naszej klasie musi również przyjmować taki typ argumentów i zachowywać kolejność argumentów określoną w klasie abstrakcyjnej.
- Można dodać tylko argumenty z wartością domyślną.

Klasy abstrakcyjne

Klasy (metody) finalne

- Metody finalne nie mogą być przeciążane w klasach dziedziczących z danej klasy.
- Zdefiniowanie całej klasy jako finalnej spowoduje, iż inne klasy nie będą mogły z niej dziedziczyć.

Wykorzystanie

```
final class Car extends Vehicle
{
    final public function go($speed) {
        $this->startEngine();
    }

    public function stop() {
        $this->stopEngine();
    }
}
```

Co to jest interfejs?

- Interfejs pozwala określić, jakie metody musi zaimplementować klasa. Nie określa jednak, jak te metody powinny działać (podobnie jak klasa abstrakcyjna).
- Interfejs jest prostszy niż klasa. Zawiera tylko definicje metod (lub stałych), ale nie określa, jak te metody działają.
- W odróżnieniu od klasy abstrakcyjnej – w interfejsie wszystkie metody są abstrakcyjne z definicji.

Co to jest interfejs?

Definicja

interface **Vehicle**

```
{  
    public function go($speed);  
    public function stop();  
}
```

Wykorzystanie

class **Car** implements **Vehicle**

```
{  
    public function go($speed) {  
        $this->startEngine();  
    }  
    public function stop() {  
        $this->stopEngine();  
    }  
}
```

Po co nam interfejsy?

- Dzięki interfejsom różne klasy mogą być wykorzystywane w ten sam sposób, bez konieczności dziedziczenia.
- Przykładowy interfejs **Countable** wymusza na nas implementację metody **count()**.
- Metoda **count()** jest używana później przez funkcję **count()** (tą samą której używacie do otrzymania informacji o długości tablicy).
- Jeżeli nasza klasa implementuje interfejs **Countable** to możemy obiekt tej klasy przekazać do funkcji **count()**.

```
interface Countable {  
    public function count();  
}
```

Wiele interfejsów

Klasa – w przeciwieństwie do dziedziczenia – może implementować wiele różnych interfejsów.

```
class UFO implements Vehicle, Airplane,  
Spaceship {  
    ...  
}
```

Dziedziczenie interfejsów

Interfejsy mogą dziedziczyć z innych interfejsów.

```
Interface Bike {
```

```
}
```

```
Interface Motorbike extends Bike {
```

```
}
```

Zgodność argumentów

- Przy implementacji interfejsu w klasie musimy zachować **zgodność typów i liczby argumentów**.
- Jeżeli interfejs określa typ danego argumentu, to metoda w klasie musi również przyjmować taki typ argumentów i zachowywać kolejność argumentów określoną w interfejsie.

Zły przykład

Interface Alien {

public function **abduct**(Human \$obj, \$purpose);

}

class Martian implements Alien

{

public function **abduct**(Creature \$obj) {

.....

}

}

Typ i liczba argumentów
niezgodne z interfejsem



Interfejsy vs klasy abstrakcyjne

Klasy abstrakcyjne

Klas abstrakcyjnych używamy, gdy klasy są ze sobą mocno powiązane.

Na przykład jest taka sama część kodu i wspólne zmienne dla wszystkich klas dziedziczących z danej klasy, którą możemy zaimplementować w klasie nadrzędnej.

Interfejsy

Interfejsy wykorzystujemy do zapewnienia tego samego zestawu metody dla klas, które nie są ze sobą mocno powiązane.

Przykład

Interfejs Iterator

Pozwala na użycie obiektu w pętli foreach.
Dzięki niemu możemy bez użycia dodatkowych metod przeiterować np. po własności obiektu będącej tablicą.

Metody iteratora są wywoływane automatycznie przy kolejnych iteracjach, do nas należy napisanie ich implementacji.

Interfejs Iterator nie powoduje zamiany obiektu w tablicę a więc nie można takiego obiektu użyć np. w funkcjach tablic.

Iterator

```
Iterator extends Traversable {  
    /* Metody */  
    abstract public mixed current(void)  
    abstract public scalar key(void)  
    abstract public void next(void)  
    abstract public void rewind(void)  
    abstract public boolean valid(void)  
}
```


Przykład

Interfejs `ArrayAccess`

Pozwala na użycie obiektu jak tablicy.
Dzięki niemu możemy odwołać się do obiektu jak do tablicy:

```
$obj = new ExampleObject();  
$obj['foo'] = 'bar'; //wywoła metode offsetSet  
echo $obj['foo']; //wywoła metode offsetGet
```

Metody `ArrayAccess` są wywoływane automatycznie przy próbie dostępu do obiektu jak tablicy.

Interfejs `ArrayAccess` nie powoduje zamiany obiektu w tablicę a więc nie można takiego obiektu użyć np. w funkcjach tablic.

`ArrayAccess`

```
ArrayAccess {  
    /* Metody */  
  
    abstract public boolean offsetExists(mixed $offset)  
  
    abstract public mixed offsetGet(mixed $offset)  
  
    abstract public void offsetSet(mixed $offset,  
                                    mixed $value)  
  
    abstract public void offsetUnset(mixed $offset)  
}
```

Przykład

Interfejs JsonSerializable

Pozwala na użycie obiektu w funkcji **serialize()**.
W metodzie **jsonSerialize()** implementujemy co ma być zwrócone i przekazane do funkcji **serialize()** np. String lub Array

```
$obj = new ExampleObject();  
serialize($obj);
```

//wywoła metode jsonSerialize() i
przekaze jej rezultat do funkcji serialize()

Metoda JsonSerializable jest wywoływana automatycznie przy przekazaniu obiektu do funkcji **serialize()**.

JsonSerializable

```
JsonSerializable{
```

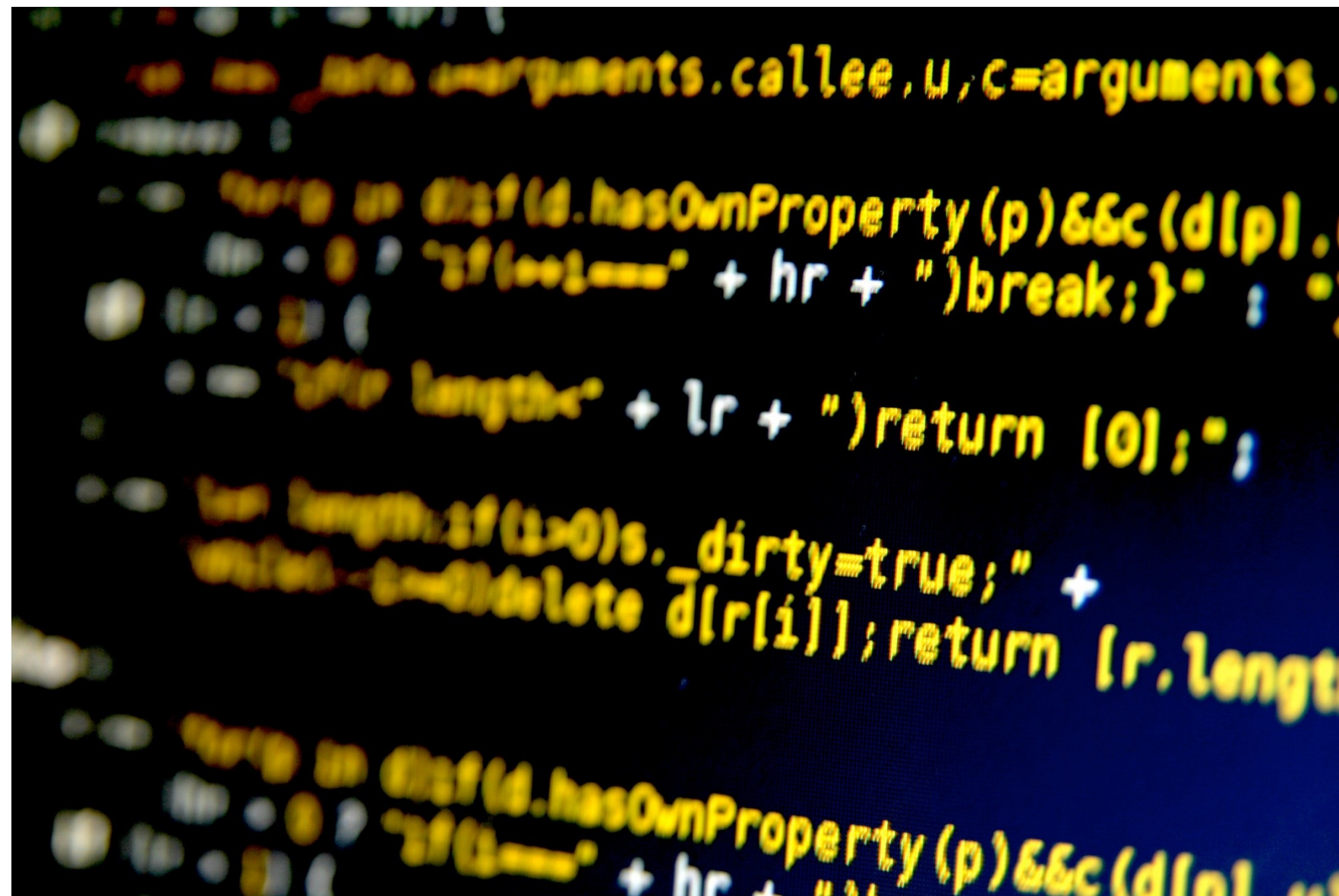
```
/* Metody */
```

```
abstract public mixed jsonSerialize(void)
```

```
}
```

O tym interfejsie wspominaliśmy na zajęciach z REST

Czas na zadania



Wykonajcie zadania z działu:

**Interfejsy, klasy
abstrakcyjne i finalne**

Funkcja autoload

Problem

- W dużych projektach składających się z setek albo tysięcy klas zapisanych w różnych plikach trudno jest programiście kontrolować, które pliki powinny zostać dołączone.
- Nie opłaca się dołączać wszystkich plików tylko dlatego, że jest szansa, iż mogą zostać użyte.
- Taka lista plików byłaby trudna w aktualizacji.

Rozwiązanie

- **Funkcja autoload** pozwala na automatyczne załadowanie odpowiedniego pliku, gdy interpreter stwierdza, że nie ma definicji do utworzenia obiektu.
- W tym celu każdą klasę zapisuje się w oddzielnym pliku.
- Nazwa tego pliku to nazwa klasy, którą ten plik zawiera.

Funkcja `__autoload` na podstawie nazwy klasy/interfejsu, który nie został wcześniej znaleziony, może doładować odpowiedni plik zawierający tę klasę.

Funkcja autoload

File: MyClass1.php

```
class MyClass1 {  
  
}
```

File: MyClass2.php

```
class MyClass2 {  
  
}
```

```
function __autoload($class_name) {  
    include $class_name . '.php';  
}
```

```
$obj1 = new MyClass1();  
$obj2 = new MyClass2();
```

spl_autoload_register

- Istotnym ograniczeniem **autoload** jest to, że w kodzie może być tylko jedna taka funkcja.
- Nowszym mechanizmem do automatycznego ładowania klas jest **spl_autoload_register**, która pozwala na zarejestrowanie wielu różnych funkcji **autoload** co oznacza, iż możemy dodawać klasy np. z różnych katalogów.
- Wszystkie zarejestrowane funkcje są wykonywane w kolejności rejestracji.

Zaawansowane działanie na stringach

Przypomnienie

`$str = "Ala ma kota";`

`$str[0]`

A

`$str[1]`

|

`substr($str,4,2);`

ma

`explode(' ', $str)`

array('Ala','ma','kota')

`implode('_', explode(' ', $str))`

ala_ma_kota

Funkcje

➤ **int strlen (string \$string)**

Zwraca długość łańcucha znaków.

➤ **string trim (string \$str [, string \$character_mask = " \t\n\r\0\x0B"])**

Usuwa niechciane znaki (domyślnie białe znaki) z początku i końca łańcucha znaków.

➤ **int strcmp (string \$str1, string \$str2)**

Porównuje dwa łańcuchy znaków.

Zwraca następujące wartości:

<0 – gdy **\$str1** jest mniejsze od **\$str2**,

>0 – gdy **\$str1** jest większe od **\$str2**,

0 – gdy oba łańcuchy znaków są takie same.

Funkcja jest wykorzystywana do sortowania łańcucha znaków.

Funkcje

- **mixed str_replace (mixed \$search , mixed \$replace , mixed \$subject [, int &\$count])**

Wyszukuje w **\$subject** wystąpienia **\$search** i zmienia je na **\$replace**.

\$search i **\$replace** mogą być również tablicami, wtedy są szukane wszystkie elementy z tablicy **\$search**, następnie zamieniane są na odpowiadające im elementy tablicy **\$replace**.

Funkcja zwraca wynikowy łańcuch znaków.

- **string strstr (string \$haystack , mixed \$needle [, bool \$before_needle = false])**

Znajduje pierwsze wystąpienie **\$needle** w zmiennej **\$haystack**.

Zwraca część zmiennej **\$haystack** od początku wystąpienia **\$needle** do końca.

Jeżeli **\$needle** nie został znaleziony zwraca **false**.

- **mixed strpos (string \$haystack , mixed \$needle [, int \$offset = 0])**

Działa podobnie jak **strstr**, ale zwraca pozycję (indeks) pierwszego wystąpienia **\$needle**.

Jeżeli **\$needle** nie został znaleziony zwraca **false**.

Funkcje

➤ **string strtolower (string \$string)**

Zwraca wejściowy łańcuch znaków po zamianie wszystkich liter na małe.

➤ **string strtoupper (string \$string)**

Zwraca wejściowy łańcuch znaków po zamianie wszystkie liter na wielkie.

➤ **string ucfirst (string \$str)**

Zwraca wejściowy łańcuch znaków po zamianie pierwszej litery na wielką.

➤ **string ucwords (string \$str)**

Zwraca wejściowy łańcuch znaków po zamianie pierwszej litery każdego wyrazu na wielką.

Funkcje

- **string addslashes (string \$str)**

Dodaje odwrotne ukośniki przez znakami, które tego wymagają, przykładowo przy zapytaniach do bazy w celu uniknięcia sql injection

- **string stripslashes (string \$str)**

Usuwa działanie funkcji **addslashes**.

- **string strip_tags (string \$str [, string \$allowable_tags])**

Usuwa tagi HTML z podanego łańcucha znaków.

- **void parse_str (string \$str [, array &\$arr])**

Analizuje wejściowy łańcuch znaków **\$str**, traktuje go jako ciąg parametrów zapytania. Poszczególne parametry zapytania zapisywane są w tablicy **\$arr**.

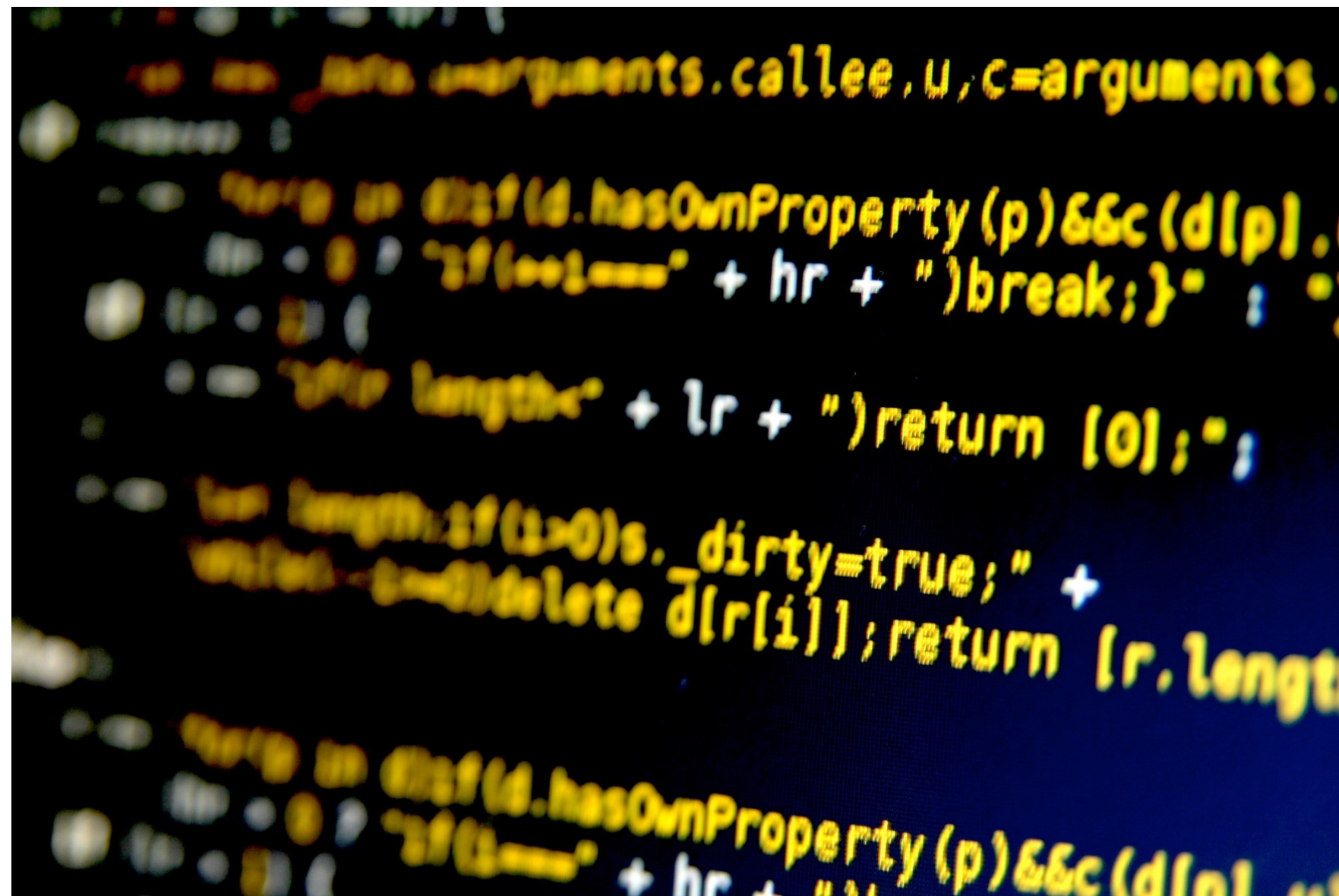
Funkcje niewrażliwe na wielkość liter

- Wiele funkcji wyszukiujących ma swoje odpowiedniki nierozróżniające wielkich i małych liter.
- Te funkcje mają dodane w swojej nazwie literę **i**. Na przykład odpowiednikiem **str_replace** jest **str_ireplace**.
- Te funkcje działają tak samo jak ich odpowiedniki wrażliwe na wielkość znaków.

Kodowanie wielobajtowe

- Kodowanie wielobajtowe jest to wykorzystanie więcej niż jednego bajta do zapisania znaku (np. UTF-8, UTF-32).
 - W takim przypadku zwykłe funkcje operujące na łańcuchach znaków nie sprawdzają się, bo nie potrafią rozróżnić, czy znak jest zakodowany przy użyciu jednego bajta, czy kilku bajtów.
- W PHP istnieje zestaw funkcji **mb_** działających tak samo jak omówione, ale potrafią wykonywać te operacje na łańcuchach znaków z kodowaniem wielobajtowym.
 - Więcej na temat funkcji wielobajtowych w PHP: <http://php.net/manual/en/ref.mbstring.php>

Czas na zadania



Wykonajcie zadania z działu:

**Zaawansowane działania
na stringach**

Wyrażenia regularne

Definicja

- **Wyrażenia regularne (regex)** to wzorce opisujące łańcuchy symboli.
 - W informatyce teoretycznej ciągi znaków pozwalające opisywać języki regularne.
 - W praktyce znalazły bardzo szerokie zastosowanie, pozwalają bowiem w łatwy sposób opisywać wzorce tekstu.
- Dwie najpopularniejsze składnie wyrażen regularnych to **składnia uniksowa** i **składnia perlowa**.
 - W większości zastosowań stosuje się **składnię perlową**.

Podstawowe elementy

- Każdy znak, oprócz znaków specjalnych, określa sam siebie, np. **a** oznacza znak **a**.
- Kolejne symbole oznaczają, że w łańcuchu muszą wystąpić dokładnie te symbole w dokładnie takiej samej kolejności, np. **ab** oznacza, że łańcuch musi składać się ze znaku **a** poprzedzającego znak **b**.

Znaki specjalne

ZNAK	ZNACZENIE
.	Dowolny znak z wyjątkiem znaku nowego wiersza
[]	Jeden dowolny znak ze znaków znajdujących się między nawiasami, np. [abc] – oznacza a, b lub c
[a–c]	Jeden znak z przedziału od a do c
[^...]	Jeden dowolny znak nieznajdujący się między nawiasami np.: [^abc] – oznacza jeden znak z wyjątkiem a, b i c.

ZNAK	ZNACZENIE
()	To grupa symboli do późniejszego wykorzystania (przechwytywanie).
	To odpowiednik słowa lub, oznacza wystąpienie jednego podanych wyrażeń, np.: a b c oznacza a lub b lub c.
^	Oznacza początek wiersza.
\$	Oznacza koniec wiersza.

Znaki specjalne

Znaki specjalne jako zwykłe znaki

Jeżeli chcemy, aby znak specjalny został potraktowany jako zwykły, to musimy go poprzedzić ukośnikiem wstecznym \.

Np.:

- \. – to kropka a nie dowolny znak,
- \.+ – to jedna kropka lub więcej.

ZNAK	ZNACZENIE
*	To zero lub więcej wystąpień poprzedzającego wyrażenia, np.: [abc]* – oznacza zero lub więcej znaków ze zbioru a, b, c.
+	To jedno wystąpienie lub więcej poprzedzającego wyrażenia.
?	To najwyżej jedno wystąpienie (może być zero) poprzedzającego wyrażenia.

Rozszerzenia

KOD	ZNACZENIE
<code>\d</code>	Dowolna cyfra
<code>\D</code>	Dowolny znak niebędący cyfrą
<code>\s</code>	Dowolny znak biały (np. spacja, tabulator)
<code>\S</code>	Dowolny znak niebędący znakiem białym
<code>\w</code>	Dowolny znak wyrazu
<code>\W</code>	Dowolny znak niebędący znakiem wyrazu
<code>\n</code>	Nowa linia
<code>\t</code>	Tabulator
<code>\r</code>	Powrót karetki
<code>[:digit:]</code>	Dowolna cyfra
<code>[:alpha:]</code>	Dowolna litera
<code>[:alnum:]</code>	Dowolna cyfra i litera
<code>{N}</code>	Dokładnie N wystąpień
<code>{N,}</code>	Co najmniej N wystąpień
<code>{N,M}</code>	Od N do M wystąpień

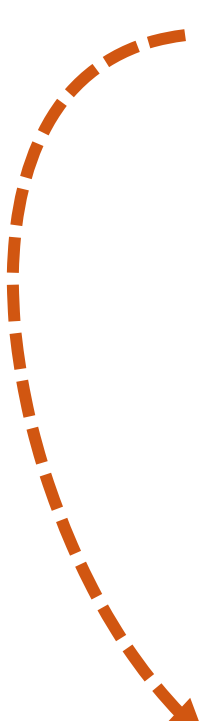
Odwołania i przechwytywanie

Odwołania wsteczne

Przechwycony wcześniej fragment może być wykorzystany później w wyrażeniu za pomocą odwołania wstecznego.

Do kolejnych fragmentów odwołujemy się poprzez \1 \2 \3 itd.

- (\w+)=\1
- Pasuje do:
- Ala=Ala



Sprawdza czy przed znakiem równości w stringu znajduje się słowo \w+, przechwytuje je a następnie sprawdza czy za znakiem równości jest to samo słowo \1

Przechwytywanie nazwane

Przechwytywany fragment tekstu możemy oznaczyć etykietą, dzięki temu łatwiej będzie się do niego odwołać.

- (?P<etykieta>...)

i odwołać się poprzez:

- (?P=etykieta)
- /(?(?P<slowo>\w+)=(?P=slowo)/

Przydatne przykłady wyrażeń regularnych

Wartość HEX

`^#?([a-f0-9]{6}|[a-f0-9]{3})$`

Adres IP

`^(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)$`

Tag HTML

`^<([a-z]+)([^\<]+)*(?:>(.*)<\1>|\s+V>)$`

Kod pocztowy

`[0-9][0-9]-[0-9][0-9][0-9]`

Adres email

`^[_a-zA-Z0-9-]+(\.[_a-zA-Z0-9-]+)*@[a-zA-Z0-9-]+\.[a-zA-Z0-9-]{1,}\.([a-zA-Z]{2,}){1}$`

Godzina w formacie 24h

`([01]?[0-9]|2[0-3]):[0-5][0-9]`

Separatory i wyrażenia zachłanne (greedy)

Separatory

Separatory oddzielają poszczególne części wyrażenia.

Najczęściej stosowane separatory to: / # ~

- `/[0-9][0-9]-[0-9][0-9][0-9] /`
- `/[0-9][0-9]-[0-9][0-9][0-9] /i`

Wyrażenia zachłanne

Kwantyfikatory w wyrażeniach regularnych dopasowują tak wiele znaków jak to możliwe

- `[[Ala ma kota]] [[Ale kot nie ma Ali]]`
`(\[.*\])`

Można tego uniknąć dzięki zdefiniowaniu znaków, które nie powinny wystąpić:

- `(\[^[^]*\])`

Można też dodać? po kwantyfikatorze, który zmienia go na leniwy.

Tryby dopasowania

- **i** – ignoruje wielkość liter w wyrażeniu,
- **m** – wykonuje dopasowanie dla wielu linii,
- **s** – kropka oznacz również znak nowej linii,
- **u** – stosuje UTF-8,
- **U** – domyślnie kwantyfikatory nie są zachłanne.

Tryby dopasowania można łączyć ze sobą:

- **/w+/iu**

Wyrażenia regularne w PHP

- **preg_match, preg_match_all**
– sprawdza, czy wyrażenie pasuje do podanego łańcucha znaków,
- Funkcja zwraca następujące wartości:
 - **1** – jeżeli jest dopasowanie,
 - **0** – jeżeli nie ma dopasowania,
 - **false** – jeżeli wystąpił błąd.

```
$subject = "abcdef";
```


```
$pattern = '/^def/';
```

```
preg_match($pattern, $subject, $matches);
```

Wyrażenia regularne w PHP

preg_grep – wyszukuje elementy tablicy pasujące do wzoru.

```
$fl_array = preg_grep("/^(\d+)?\.\d+$/", $array);
```



Zwraca wszystkie elementy
będące liczbami zmiennoprzecinkowymi.

Wyrażenia regularne w PHP

preg_replace – wyszukuje i zamienia wystąpienia wzorca podanym tekstem.

Kod

```
$string = 'April 15, 2003';  
$pattern = '/(\w+) (\d+), (\d+)/i';  
$replacement = '${1} 1,$3';  
echo(preg_replace($pattern, $replacement, $string));
```

Wynik

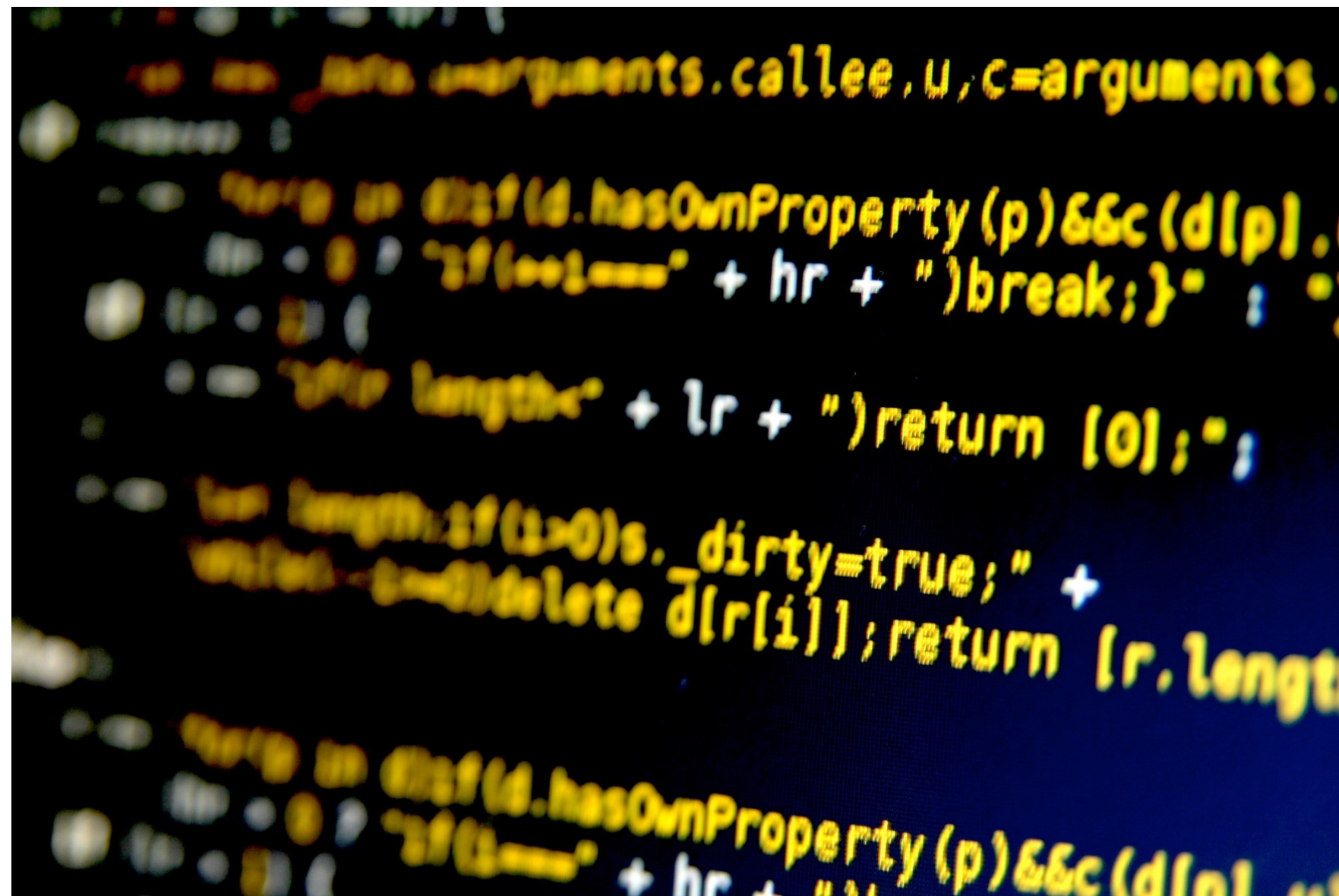
April 1,2003

Wyrażenia regularne w PHP

preg_split – dzieli łańcuch znaków przy użyciu wyrażenia regularnego oraz zwraca tablicę z wyodrębnionymi częściami.

```
$keywords = preg_split("/[\\s,]+/", "hypertext  
language, programming");
```


Czas na zadania



Wykonajcie zadania z działu:

Wyrażenia regularne