8.1 8.1 Teoria

Step 1

Programowanie zorientowane obiektowo uznaje podstawowe połączenie między danymi a kodem, który na nich działa, i umożliwia projektowanie i wdrażanie programów związanych z tym połączeniem. Na przykład system tablic ogłoszeń zwykle śledzi wielu użytkowników. W **proceduralnym języku programowania** każdy użytkownik jest reprezentowany przez strukturę danych i prawdopodobnie istniałby zestaw funkcji, które współpracują z tymi strukturami danych (w celu tworzenia nowych użytkowników, uzyskiwania ich informacji itp.). W języku **OOP** każdy użytkownik jest reprezentowany przez **obiekt** - strukturę danych z dołączonym kodem. Dane i kod wciąż tam są, ale są traktowani jako nierozłączna całość. Obiekt, będący połączeniem kodu i danych, jest modułową jednostką służącą do tworzenia aplikacji i ponownego wykorzystania kodu. W tym projekcie tablicy ogłoszeń obiekty mogą reprezentować nie tylko użytkowników, ale także wiadomości i wątki. Obiekt użytkownika ma nazwę użytkownika i hasło dla tego użytkownika oraz kod identyfikujący wszystkie wiadomości tego autora. Obiekt wiadomości wie, do którego wątku należy, i ma kod do opublikowania nowej wiadomości, odpowiedzi na istniejącą wiadomość i wyświetlenia wiadomości. Obiekt wątku jest zbiorem obiektów wiadomości i zawiera kod wyświetlić indeks wątku. Jest to jednak tylko jeden sposób na podzielenie niezbędnej funkcjonalności na obiekty. Na przykład w alternatywnym projekcie kod wysyłający nową wiadomość znajduje się w obiekcie użytkownika, a nie w obiekcie wiadomości. Projektowanie systemów obiektowych to złożony temat i napisano na ten temat wiele książek. Dobra wiadomość jest taka, że bez względu na to, jak zaprojektujemy swój system, można go zaimplementować w PHP.

Obiekt jest instancją (lub wystąpieniem) **klasy**. W tym przypadku jest to rzeczywista struktura danych użytkownika z dołączonym kodem. Obiekty i klasy są trochę podobne do wartości i typów danych. Dane powiązane z obiektem nazywane są jego **właściwościami**. Funkcje skojarzone z obiektem nazywane są jego **metodami**. Definiując klasę, definiowane są nazwy, właściwości oraz metody. Debugowanie i konserwacja programów jest znacznie łatwiejsze, podczas używania OOP, ponieważ stosowany jest mechanizm **hermetyzacji**. Chodzi o to, że klasa udostępnia określone metody (interfejs) kodowi, który używa jej obiektów, więc zewnętrzny kod nie ma bezpośredniego dostępu do struktur danych tych obiektów. Debugowanie jest więc łatwiejsze, ponieważ wiemy, gdzie szukać błędów - jedyny kod, który zmienia strukturę danych obiektu, znajduje się w klasie - a konserwacja jest łatwiejsza, ponieważ możemy wymieniać implementacje klasy bez zmiany kodu który używa tej klasy, o ile utrzymujemy ten sam interfejs. Każdy nietrywialny projekt zorientowany obiektowo prawdopodobnie obejmuje mechanizm **dziedziczenia**. Jest to sposób definiowania nowej klasy przez stwierdzenie, że przypomina ona istniejącą klasę, ale ma pewne nowe właściwości i metody. Oryginalna klasa jest nazywana **nadklasą** (lub klasą nadrzędną lub podstawową), a nowa klasa jest nazywana **podklasą** (lub klasą pochodną). Dziedziczenie jest formą ponownego wykorzystania kodu - kod nadklasy jest używany ponownie zamiast kopiowania i wklejania do podklasy. Wszelkie ulepszenia lub modyfikacje nadklasy są automatycznie przekazywane do podklasy.

Tworzenie (lub tworzenie instancji) obiektów i używanie ich jest o wiele łatwiejsze niż definiowanie klas obiektów. Aby stworzyć obiekt danej klasy, użyj nowego słowa kluczowego `new`:

```
<?php
  $object = new Nazwa_klasy;
?>
```

Zakładając, że klasa `Person` została zdefiniowana, oto jak utworzyć obiekt `Person`:

```
<?php
  $moana = new Person;
?>
```

Nie należy nazwy_klasy traktować jako ciąg znaków; będzie to traktowane jako błąd kompilacji.

Niektóre klasy pozwalają na przekazywanie argumentów do tworzenia obiektu tej klasy. Dokumentacja klasy powinna określać, czy przyjmuje argumenty. Jeśli tak, to obiekty tworzymy w następujący sposób (na przykładzie klasy `Person`):

```
<?php
  $object = new Person("Mateusz", 35);
?>
```

Nazwa klasy nie musi być zakodowana na stałe w programie. Nazwę klasy możesz podać za pomocą zmiennej:

```php
<?php
    $class = "Person";
    $object = new $class;
    // poniższa linia jest równoważna
    $object = new Person;
?>
```

Zmienne zawierające odniesienia do obiektów są zwykłymi zmiennymi - można ich używać w taki sam sposób, jak innych zmiennych. Należy zwrócić uwagę, że zmienne działają z obiektami, co pokazuje kolejny przykład:

```php
<?php
    $account = new Account;
    $object = "account";
    ${$object}->init(50000, 1.10); // to samo co $account->init
?>
```

Gdy mamy już obiekt, możemy użyć notacji `->` (jak w przykładzie powyżej), aby uzyskać dostęp do metod i właściwości obiektu:

```php
$object->propertyname $object->methodname([arg, ... ])
```

```php
<?php
    echo "Mateusz is {$person->age} years old.\n"; // wywołanie atrybutu danych
    $person->birthday(); // wywołanie metody
    $person->setAge(21); // wywołanie metody z parametrami
?>
```

Metody działają tak samo jak funkcje (tylko w odniesieniu do danego obiektu), więc mogą przyjmować argumenty i zwracać wartość:

```php
<?php
    $clan = $moana->family("extended");
?>
```

W definicji klasy można określić, które metody i właściwości są publicznie dostępne, a które są dostępne tylko z poziomu samej klasy, przy użyciu modyfikatorów dostępu publicznego (`public`) i prywatnego (`private`). Ich użycia zapewnia nam właśnie mechanizm hermetyzacji. **Metoda statyczna** to metoda wywoływana w klasie, a nie w obiekcie. Takie metody nie mają dostępu do właściwości. Nazwa metody statycznej to nazwa klasy, po której następują dwa dwukropki (`::`) i nazwa funkcji.

```php
<?php
    HTML::p("Hello, world");
?>
```

Deklarując klasę, definiuje się, które właściwości i metody są statyczne przy użyciu właściwości dostępu statycznego. Po utworzeniu obiekty są przekazywane przez odniesienie - to znaczy zamiast kopiowania wokół samego obiektu (przedsięwzięcie pochłaniające czas i pamięć), zamiast tego przekazywane jest odniesienie do obiektu. Na przykład:

```php
<?php
    $f = new Person("Pua", 75);
    $b = $f; // $b and $f point at same object
    $b->setName("Hei Hei");
    printf("%s and %s are best friends.\n", $b->getName(), $f->getName()); //Hei Hei and Hei Hei are best
friends.
?>
```

Jeśli chcemy stworzyć prawdziwą kopię obiektu, należy użyć operatora `clone` :

```php
<?php
    $f = new Person("Pua", 35);
    $b = clone $f;
    $b->setName("Hei Hei");
    printf("%s and %s are best friends.\n", $b->getName(), $f->getName()); //Pua and Hei Hei are best
friends.
?>
```

Kiedy używamy operatora `clone` do tworzenia kopii obiektu i ta klasa deklaruje metodę `__clone()`, ta metoda jest wywoływana w nowym obiekcie natychmiast po jego sklonowaniu. Możemy tego użyć w przypadkach, gdy obiekt przechowuje zasoby zewnętrzne (takie jak uchwyty plików) do tworzenia nowych zasobów, zamiast kopiować istniejące.

Step 2

Aby zaprojektować program lub bibliotekę kodu w sposób obiektowy, musimy zdefiniować własne klasy, używając słowa kluczowego `class`. Definicja klasy zawiera nazwę klasy oraz właściwości i metody klasy. Nazwy klas nie uwzględniają wielkości liter i muszą być zgodne z regułami dotyczącymi identyfikatorów PHP. Między innymi zastrzeżona jest nazwa klasy `stdClass`. Oto składnia definicji klasy:

```
<?php
    class classname [ extends baseclass ] [ implements interfacename , [interfacename, ... ] ] {
            [ use traitname, [ traitname, ... ]; ]
        [ visibility $property [ = value ]; ... ]
        [ function functionname (args) [: type ] {
        // code
            }
        ...
        ]
    }
?>
```

Metoda to funkcja zdefiniowana wewnątrz klasy. Chociaż PHP nie nakłada żadnych specjalnych ograniczeń, większość metod działa tylko na danych w obiekcie, w którym znajduje się metoda. Nazwy metod zaczynające się od dwóch podkreślników (`__`) mogą być używane w przyszłości przez PHP, więc zaleca się nie zaczynać nazw metod od tej sekwencji. W ramach metody zmienna `$this` zawiera odniesienie do obiektu, na którym ta metoda została wywołana. Na przykład, jeśli zostanie wywołana linia `$moana->birthday()`, wewnątrz metody `birthday()`, `$this` ma taką samą wartość jak `$moana`. Metody używające zmienną `$this`, uzyskają dostęp do właściwości bieżącego obiektu i wywołują inne metody na tym obiekcie.

Przejdźmy do wcześniej wspomnianej klasie Person. Jego definicja może wyglądać następująco:

```
<?php
    class Person {
        public $name = '';
        function getName() {
            return $this->name;
        }
        function setName($newName) {
            $this->name = $newName;}
    }
    $object = new Person;
    $object->setName("Mateusz");
    echo $object->getName();
?>
```

Jak widać, metody `getName()` i `setName()` używają `$this` do uzyskiwania dostępu i ustawiania właściwości `$name` bieżącego obiektu.

Aby zadeklarować metodę jako metodę statyczną, użyj słowa kluczowego `static`. W metodach statycznych zmienna `$this` nie jest zdefiniowana. Na przykład:

```
<?php
    class HTMLStuff {
        static function startTable() {
            echo "<table border=\"1\">\n";
        }
        static function endTable() {
            echo "</table>\n";
        }
    }
    HTMLStuff::startTable();
    // print HTML table rows and columns
    HTMLStuff::endTable();
?>
```

Używając modyfikatorów dostępu, możemy zmienić widoczność metod. Metody, które są dostępne poza metodami w obiekcie, powinny być zadeklarowane jako publiczne (`public`); metody instancji, które mogą być wywoływane tylko przez metody należące do tej samej klasy, powinny być zadeklarowane jako prywatne (`private`). Wreszcie metody zadeklarowane jako chronione (`protected`) mogą być wywoływane tylko z poziomu metod klasy obiektu i metod klas dziedziczących po klasie. Definiowanie widoczności metod klas jest opcjonalne; jeśli widoczność nie jest określona, metoda jest publiczna.

```
<?php
    class Person {
        public $age;
        public function __construct() {
            $this->age = 0;
        }
        public function incrementAge() {
            $this->age += 1;
            $this->ageChanged();
        }
        protected function decrementAge() {
            $this->age -= 1;
            $this->ageChanged();
        }
        private function ageChanged() {
            echo "Age changed to {$this->age}";
        }
    }
    class SupernaturalPerson extends Person {
        public function incrementAge() {
    // ages in reverse
            $this->decrementAge();
        }
    }
    $person = new Person;
    $person->incrementAge();
//    $person->decrementAge(); // not allowed
//    $person->ageChanged(); // also not allowed
    $person = new SupernaturalPerson;
    $person->incrementAge(); // calls decrementAge under the hood
?>
```

Możemy również w klasie deklarować metody następująco:

```
<?php
    class Person {
        function takeJob(Job $job) {
            echo "Now employed as a {$job->title}\n";
        }
    }
?>
```

Gdy metoda zwraca wartość, możemy zadeklarować typ wartości zwracanej metody:

```
<?php
    class Person {
        function bestJob(): Job {
            $job = Job("PHP developer");
            return $job;
        }
    }
?>
```

Step 3

W poprzedniej definicji klasy `Person` została jawnie zadeklarowana właściwość `$name`. Deklaracje własności są opcjonalne i po prostu grzecznościowe dla każdego, kto zajmuje się programem. Deklarowanie właściwości jest dobrym stylem PHP, ale w dowolnym momencie można dodać nowe właściwości. Oto wersja klasy `Person`, która ma niezadeklarowaną właściwość `$name`:

```
<?php
    class Person {
        function getName() {
            return $this->name;
        }
        function setName($newName) {
            $this->name = $newName;
        }
    }
    $obj1 = new Person;
    $obj2 = new Person;
    $obj1->setName("Mateusz");
    $obj2->setName(1234);
    echo $obj1->getName() . "\n";
    echo $obj2->getName() . "\n";
?>
```

Możemy przypisać wartości domyślne do właściwości, ale te wartości domyślne muszą być stałymi prostymi:

```
<?php
    class Person {
        public $name = "Mateusz";
        public $age = 0;
//        public $day = 60 * 60 * hoursInDay(); // nie zadziała
        function getName() {
            return $this->name;
        }
        function setName($newName) {
             $this->name = $newName;
        }
    }
    $obj = new Person;
    echo $obj->getName() . "\n" . $obj->age . "\n";
?>
```

Za pomocą modyfikatorów dostępu można zmienić widoczność właściwości. Właściwości, które są dostępne poza zakresem obiektu, powinny być zadeklarowane jako publiczne (`public`); właściwości instancji, do których można uzyskać dostęp tylko za pomocą metod z tej samej klasy, należy zadeklarować jako prywatne (`private`). Wreszcie, do właściwości zadeklarowanych jako chronione (`protected`) można uzyskać dostęp tylko za pomocą metod klasy obiektu i metod klas dziedziczących po klasie.

```
<?php
    class Person {
        protected $rowId = 0;
        public $username = 'Anyone can see me';
        private $hidden = true;
    }
    $obj = new Person;
    echo $obj->username . "\n";
//    echo $obj->$rowId . "\n";
//    echo $obj->$hidden . "\n";
?>
```

Oprócz właściwości instancji obiektów, PHP umożliwia definiowanie właściwości statycznych, które są zmiennymi w klasie obiektów, do których można uzyskać dostęp, odwołując się do właściwości za pomocą nazwy klasy.

```
<?php
    class Person {
        static $global = 23;
    }
    $localCopy = Person::$global;
    echo $localCopy;
?>
```

Jeśli dostęp do właściwości jest uzyskiwany w obiekcie, który nie istnieje, i jeśli metoda `__get()` lub `__set()` jest zdefiniowana dla klasy obiektu, metoda ta ma możliwość pobrania wartości lub ustawienia wartości dla tej właściwości . Na przykład możemy zadeklarować klasę, która reprezentuje dane pobierane z bazy danych, ale możemy nie chcieć pobierać dużych wartości danych - takich jak duże obiekty binarne (BLOB). Jednym ze sposobów realizacji tego byłoby oczywiście utworzenie metod dostępu do właściwości, które odczytują i zapisują dane na żądanie. Inną metodą może być użycie tych metod korzystając z tzw. mechanizmu przeciążania:

```
<?php
    class Person {
        public function __get($property) {
            if ($property === 'biography') {
                $biography = "long text here..."; // would retrieve from database
                return $biography;
            }
        }
        public function __set($property, $value) {
            if ($property === 'biography') {
    // set the value in the database
            }
        }
    }
?>
```

```php
<?php
    class Person {
        private $name = "";
        private $age = 0;
        public function __get($property) {
            if ($property === 'name') {
                return $this->name;
            }
            if ($property === 'age'){
                return $this->age;
            }
        }
        public function __set($property, $value) {
            if ($property === 'name') {
                $this->name = $value;
            }
            if ($property === 'age'){
                $this->age === $value;
            }
        }
    }
    $obj = new Person;
    $obj->name = "Mateusz";
    echo $obj->name;
?>
```

Podobnie jak w przypadku stałych globalnych, przypisywanych za pomocą funkcji `define()`, PHP umożliwia przypisywanie stałych w klasie. Podobnie jak właściwości statyczne, dostęp do stałych można uzyskać bezpośrednio przez klasę lub w metodach obiektów przy użyciu notacji własnej. Po zdefiniowaniu stałej nie można zmienić jej wartości:

```php
<?php
    class PaymentMethod {
        public const TYPE_CREDITCARD = 0;
        public const TYPE_CASH = 1;
    }
    echo PaymentMethod::TYPE_CREDITCARD;
?>
```

Podobnie jak w przypadku stałych globalnych, powszechną praktyką jest definiowanie stałych klas z identyfikatorami wielkimi literami.

Za pomocą modyfikatorów dostępu można zmienić widoczność stałych klas. Stałe klas, które są dostępne poza metodami obiektu, powinny być zadeklarowane jako publiczne; stałe klasy instancji, do której można uzyskać dostęp tylko za pomocą metod z tej samej klasy, należy zadeklarować jako prywatne. Wreszcie, do stałych zadeklarowanych jako chronione można uzyskać dostęp tylko z poziomu metod klasy obiektu i pliku metody klas klas dziedziczących po klasie. Definiowanie widoczności stałych klas jest opcjonalne; jeśli widoczność nie jest określona, metoda jest publiczna.

```php
<?php
    class Person {
        protected const PROTECTED_CONST = false;
        public const DEFAULT_USERNAME = "<unknown>";
        private const INTERNAL_KEY = "ABC1234";
    }
    echo Person::DEFAULT_USERNAME;
?>
```

Step 4

Aby dziedziczyć właściwości i metody z innej klasy, należy użyć słowa kluczowego `extends` w definicji klasy, po którym następuje nazwa klasy bazowej:

```
<?php
    class Person {
        public $name, $address, $age;
    }
    class Employee extends Person {
        public $position, $salary;
    }
?>
```

Klasa `Employee` zawiera właściwości `$position` i `$salary`, a także właściwości `$name`, `$address` i `$age` odziedziczone z klasy `Person`. Jeśli klasa pochodna ma właściwość lub metodę o takiej samej nazwie jak klasa nadrzędna, właściwość lub metoda w klasie pochodnej ma pierwszeństwo przed właściwością lub metodą w klasie nadrzędnej. Odwołanie do właściwości zwraca wartość właściwości w przypadku elementu podrzędnego, podczas gdy odwołanie do metody wywołuje metodę w przypadku elementu podrzędnego. Notacja `parent::method()`, służy do uzyskania dostępu do nadpisanej metody w klasie nadrzędnej obiektu:

```
<?php
    class Person {
        public $name, $address, $age;
        public function birthday(){
            echo "This is a birthday\n";
        }
    }
    class Employee extends Person {
        public $position, $salary;
        public function birthday()
        {
           parent::birthday();
           echo "This is a birthday from Employee\n";
        }
    }
    $obj = new Person;
    $obj->birthday();
    $obj1 = new Employee;
    $obj1->birthday();
?>
```

Częstym błędem jest zakodowanie na stałe nazwy klasy nadrzędnej w wywołaniach nadpisanych metod:

```
<?php
    class Person {
        public $name, $address, $age;
        public function birthday(){
            echo "This is a birthday\n";
        }
    }
    class Employee extends Person {
        public $position, $salary;
        public function birthday()
        {
           Person::birthday();
           echo "This is a birthday from Employee\n";
        }
    }
    $obj = new Person;
    $obj->birthday();
    $obj1 = new Employee;
    $obj1->birthday();
?>
```

Jest to błąd, ponieważ rozpowszechnia wiedzę o nazwie klasy nadrzędnej w całej klasie pochodnej. Użycie `parent::` centralizuje wiedzę o klasie nadrzędnej w klauzuli `extends`. Jeśli metoda może być podklasą i chcemy mieć pewność, że wywołujemy ją w bieżącej klasie, należy użyć notacji `self::method()`:

```
<?php
    class Person {
        public $name, $address, $age;
        public function birthday(){
            echo "This is a birthday\n";
        }
    }
    class Employee extends Person {
        public $position, $salary;
        public function birthday(){
           echo "This is a birthday from Employee\n";
        }
        public function get_birthday(){
            self::birthday();
        }
    }
    $obj = new Person;
    $obj->birthday();
    $obj1 = new Employee;
    $obj1->get_birthday();
?>
```

Aby sprawdzić, czy obiekt jest instancją określonej klasy lub czy implementuje określony interfejs, należy użyć operatora `instanceof`:

```
<?php
    class Person {
        public $name, $address, $age;
        public function birthday(){
            echo "This is a birthday\n";
        }
    }
    class Employee extends Person {
        public $position, $salary;
        public function birthday(){
           echo "This is a birthday from Employee\n";
        }
        public function get_birthday(){
            self::birthday();
        }
    }
    $obj = new Person;
    if ($obj instanceof Person){
        echo "This is a Person object\n";
    }
?>
```

Step 5

Możemy podać listę argumentów występujących po nazwie klasy podczas tworzenia wystąpienia obiektu:

```
$person = new Person("Fred", 35);
```

Te argumenty są przekazywane do konstruktora klasy, specjalnej funkcji, która inicjuje właściwości klasy. **Konstruktor** to funkcja w klasie o nazwie `__construct()`. Oto konstruktor dla klasy `Person`:

```
<?php
    class Person {
        function __construct($name, $age) {
            $this->name = $name;
            $this->age = $age;
        }
    }
    $obj = new Person("Mateusz", 30);
    echo $obj->name . " " . $obj->age;
?>
```

PHP nie zapewnia automatycznego łańcucha konstruktorów; to znaczy, że jeśli utworzymy instancję obiektu klasy pochodnej, tylko konstruktor z tej klasy jest wywoływany automatycznie. Aby można było wywołać konstruktor klasy nadrzędnej, konstruktor w klasie pochodnej musi jawnie wywołać konstruktor. W tym przykładzie konstruktor klasy `Employee` wywołuje konstruktor `Person`:

```
<?php
    class Person {
        public $name, $address, $age;
        function __construct($name, $address, $age) {
            $this->name = $name;
            $this->address = $address;
            $this->age = $age;
        }
    }
    class Employee extends Person {
        public $position, $salary;
        function __construct($name, $address, $age, $position, $salary) {
            parent::__construct($name, $address, $age);
            $this->position = $position;
            $this->salary = $salary;
        }
    }
    $obj = new Person("Mateusz", "Gdańsk", 30);
    $emp = new Employee($obj->name,$obj->address, $obj->age, "Teacher", 2000);
    var_dump($emp);
?>
```

Gdy obiekt zostanie zniszczony, na przykład gdy ostatnie odwołanie do obiektu zostanie usunięte lub osiągnięty zostanie koniec skryptu, wywoływany jest jego **destruktor**. Ponieważ PHP automatycznie czyści wszystkie zasoby, gdy wykraczają poza zakres i pod koniec wykonywania skryptu, ich zastosowanie jest ograniczone. Destruktor to metoda o nazwie `__destruct()`:

```
<?php
    class Building {
        function __destruct() {
           echo "A Building is being destroyed!";
        }
    }
    $obj = new Building;
?>
```