9.1 9.1 Teoria

Step 1

Interfejsy umożliwiają definiowanie kontraktów, do których stosuje się klasa; interfejs zawiera prototypy i stałe metod, a każda klasa implementująca interfejs musi zapewniać implementacje dla wszystkich metod w interfejsie. W PHP składnia interfejsu wygląda następująco:

```
interface interfacename {
  [ function functionname();
    ...
  ]
}
```

Aby zadeklarować, że klasa implementuje interfejs, należy dołączyć słowo kluczowe `implements` i **dowolną liczbę interfejsów**, oddzielając je przecinkami:

```
<?php
    interface Printable {
        function printOutput();
    }
    class ImageComponent implements Printable {
        function printOutput() {
            echo "Printing an image...";
        }
    }
    $obj = new ImageComponent();
    $obj->printOutput();
?>
```

Interfejs może **dziedziczyć z innych interfejsów** (w tym wielu interfejsów), o ile żaden z interfejsów nie dziedziczy z deklarowanych metod o takiej samej nazwie, jak te zadeklarowane w interfejsie podrzędnym.

Trait (cecha) zapewniają mechanizm ponownego wykorzystania kodu poza hierarchią klas. Traity umożliwiają udostępnianie funkcji różnym klasom, które nie mają (i nie powinny) mieć wspólnego przodka w hierarchii klas. W PHP składnia `trait` wygląda następująco:

```
trait traitname [ extends baseclass ] {
  [ use traitname, [ traitname, ... ]; ]
  [ visibility $property [ = value ]; ... ]
  [ function functionname (args) {
    // code
  }
    ...
  ]
}
```

Aby zadeklarować, że klasa powinna zawierać metody `traita`, należy dołączyć słowo kluczowe `use` i **dowolną liczbę cech** oddzielonych przecinkami:

```
<?php
    trait Logger {
        public function log($logString) {
            $className = __CLASS__;
            echo date("Y-m-d h:i:s", time()) . ": [{$className}] {$logString}";
        }
    }
    class User {
        use Logger;
        public $name;
        function __construct($name = '') {
            $this->name = $name;
            $this->log("Created user '{$this->name}'");
        }
        function __toString() {
            return $this->name;
        }
    }
    class UserGroup {
        use Logger;
        public $users = array();
        public function addUser(User $user) {
            if (!in_array($this->users, (array)$user)) {
                $this->users[] = $user;
                $this->log("Added user '{$user}' to group");
            }
        }
    }
    $group = new UserGroup;
    $group->addUser(new User("Franklin"));
?>
```

Metody zdefiniowane przez `trait` `Logger` są dostępne dla wystąpień klasy `UserGroup` tak, jakby zostały zdefiniowane w tej klasie.

Aby zadeklarować, że `trait` powinien składać się z innych `trait`, należy dołączyć instrukcję `use` w deklaracji `trait`, a następnie jedną lub więcej nazw `trait` oddzielonych przecinkami, jak pokazano poniżej:

```
<?php
    trait First {
        public function doFirst() {
            echo "first\n";
        }
    }
    trait Second {
        public function doSecond() {
            echo "second\n";
        }
    }
    trait Third {
        use First, Second;
        public function doAll() {
            $this->doFirst();
            $this->doSecond();
        }
    }
    class Combined {
        use Third;
    }
    $object = new Combined;
    $object->doAll();
?>
```

`Trait` mogą deklarować metody abstrakcyjne. Jeśli klasa używa wielu `trait` definiujących tę samą metodę, PHP podaje błąd krytyczny. Możesz jednak przesłonić to zachowanie, informując kompilator konkretnie, której implementacji danej metody chcemy użyć. Definiując traity, które zawierają klasę, należy użyć słowa kluczowego `insteadof` dla każdego konfliktu:

```
<?php
    trait Command {
        function run() {
            echo "Executing a command\n";
        }
    }
    trait Marathon {
        function run() {
            echo "Running a marathon\n";
        }
    }
    class Person {
        use Command, Marathon {
            Marathon::run insteadof Command;
        }
    }
    $person = new Person;
    $person->run();
?>
```

Zamiast wybierać tylko jedną metodę do uwzględnienia, możemy użyć słowa kluczowego `as`, aby aliasować metodę `trait` w klasie, włączając ją do innej nazwy. Nadal jednak musimy wyraźnie rozwiązać wszelkie konflikty dotyczące dołączonych `trait`. Na przykład:

```
<?php
    trait Command {
        function run() {
            echo "Executing a command";
        }
    }trait Marathon {
        function run() {
            echo "Running a marathon";
        }
    }
    class Person {
        use Command, Marathon {
            Command::run as runCommand;
            Marathon::run insteadof Command;
        }
    }
    $person = new Person;
    $person->run();
    $person->runCommand();
?>
```

Step 2

PHP zapewnia również mechanizm deklarowania, że pewne metody w klasie muszą być implementowane przez podklasy - implementacja tych metod nie jest zdefiniowana w klasie nadrzędnej. W takich przypadkach mówimy, że są to **metody abstrakcyjne**; dodatkowo, jeśli klasa zawiera metody zdefiniowane jako abstrakcyjne, to musimy również zadeklarować klasę jako klasę abstrakcyjną:

```
<?php
    abstract class Component {
        abstract function printOutput();
    }
    class ImageComponent extends Component {
        function printOutput() {
            echo "Pretty picture";
        }
    }
    $obj = new ImageComponent();
    $obj->printOutput();
?>
```

Nie można utworzyć instancji klas abstrakcyjnych. W przeciwieństwie do niektórych języków, PHP nie pozwala na zapewnienie domyślnej implementacji metod abstrakcyjnych. `Trait` mogą również deklarować metody abstrakcyjne. Klasy zawierające trait definiującą metodę abstrakcyjną muszą implementować tę metodę:

```
<?php
    trait Sortable {
        abstract function uniqueId();
        function compareById($object) {
            return ($object->uniqueId() < $this->uniqueId()) ? -1 : 1;
        }
    }
    class Bird {
        use Sortable;
        function uniqueId() {
            return 2;
        }
    }
    // to wywoła błąd kompilacji
//    class Car {
//        use Sortable;
//    }
    class Car {
        use Sortable;
        public function uniqueId() {
            return 10;
        }
    }
    $bird = new Bird;
    $car = new Car;
    $comparison = $bird->compareById($car);
    echo $comparison;
?>
```

Podczas implementowania metody abstrakcyjnej w klasie potomnej sygnatury metod muszą być zgodne - to znaczy muszą przyjmować taką samą liczbę wymaganych parametrów, a jeśli którykolwiek z parametrów ma wskazówki dotyczące typu, te wskazówki muszą być zgodne. Ponadto metoda musi mieć taką samą lub mniej ograniczoną widoczność.

Podczas tworzenia pozorowanych obiektów (ang. mocks) do testów przydatne jest tworzenie **anonimowych klas**. Klasa anonimowa zachowuje się tak samo, jak każda inna klasa, z wyjątkiem tego, że nie posiada ona nazwy (co oznacza, że nie można jej bezpośrednio utworzyć):

```
<?php
    class Person {
        public $name = '';
        function getName() {
            return $this->name;
        }
    }
    $anonymous = new class() extends Person {
        public function getName() {
            return "Moana";
        }
    };
    echo $anonymous->getName();
?>
```

Step 3

Introspekcja (ang. introspection) to zdolność programu do zbadania cech obiektu, takich jak jego nazwa, klasa nadrzędna (jeśli istnieje), właściwości i metody. Dzięki introspekcji możemy pisać kod działający na dowolnej klasie lub obiekcie. Nie musimy wiedzieć, które metody lub właściwości są zdefiniowane podczas pisania kodu; zamiast tego możemy odkryć te informacje w czasie wykonywania, co umożliwia pisanie ogólnych debugerów, serializatorów, profilerów i tym podobnych.

Aby określić, czy klasa istnieje, należy użyć funkcji `class_exists()`, która pobiera ciąg i zwraca wartość logiczną. Alternatywnie możemy również użyć funkcji `get_declared_classes()`, która zwraca tablicę zdefiniowanych klas i sprawdza, czy nazwa klasy znajduje się w zwróconej tablicy:

```
<?php
    class Person{

    }
    class Employee extends Person{

    }
    echo class_exists("Person") . "\n";
    $classes = get_declared_classes();
    var_dump($classes);
    $doesClassExist = in_array("Employee", (array)$classes) . "\n";
?>
```

Możemy pobrać metody i właściwości, które istnieją w klasie (w tym te, które są dziedziczone z nadklas) za pomocą funkcji `get_class_methods()` i `get_class_vars()`. Te funkcje pobierają nazwę klasy i zwracają tablicę:

```
<?php
    class Person{
        private $name = '';

        public function getName(): string
        {
            return $this->name;
        }

        public function setName(string $name): void
        {
            $this->name = $name;
        }
    }
    class Employee extends Person{
        private $salary = 0;
        public $yob = 0;
        public function getSalary(): int
        {
            return $this->salary;
        }

        public function setSalary(int $salary): void
        {
            $this->salary = $salary;
        }
    }
    var_dump(get_class_methods("Employee"));
    var_dump(get_class_vars("Employee"));
?>
```

Nazwa klasy może być zmienną zawierającą nazwę klasy, nieosłonięte słowo lub ciąg znaków w cudzysłowie:

```php
<?php
    class Person{
        private $name = '';

        public function getName(): string
        {
            return $this->name;
        }

        public function setName(string $name): void
        {
            $this->name = $name;
        }
    }
    class Employee extends Person{
        private $salary = 0;
        public $yob = 0;
        public function getSalary(): int
        {
            return $this->salary;
        }

        public function setSalary(int $salary): void
        {
            $this->salary = $salary;
        }
    }
    $str = "Employee";
    var_dump(get_class_vars("Employee"));
    var_dump(get_class_vars(Employee::class));
    var_dump(get_class_vars($str));
?>
```

Tablica zwrócona przez funkcje `get_class_methods()` to prosta lista nazw metod. Tablica asocjacyjna zwracana przez `get_class_vars()` odwzorowuje nazwy właściwości na wartości, a także zawiera właściwości dziedziczone.

```php
<?php
    class Person{
        private $name = '';
        public $secondname;
        public function getName(): string
        {
            return $this->name;
        }

        public function setName(string $name): void
        {
            $this->name = $name;
        }
    }
    class Employee extends Person{
        private $salary = 0;
        public $yob = 0;
        public $a;
        public function getSalary(): int
        {
            return $this->salary;
        }

        public function setSalary(int $salary): void
        {
            $this->salary = $salary;
        }
    }
    $str = "Employee";
    var_dump(get_class_vars("Employee"));
?>
```

Aby odnaleźć nazwę klasy nadrzędnej, należy użyć metody `get_parent_class()`.

```
<?php
    class A {}
    class B extends A {}
    $obj = new B;
    echo get_parent_class($obj);
    echo get_parent_class(B::class);
?>
```

Aby pobrać nazwę klasy, do której należy obiekt, należy najpierw upewnić się, że jest to obiekt, korzystając z funkcji `is_object()`:

```
<?php
    class Person{
        private $name = '';
        public $secondName = '';

        public function setName(string $name): void
        {
            $this->name = $name;
        }

        public function getName(): string
        {
            return $this->name;
        }
    }
    $var = new Person();
//    $var = "Person";
    if (is_object($var))
        echo get_class($var);
    else
        echo "\$var is not an object!!!"

?>
```

Przed wywołaniem metody na obiekcie można upewnić się, że ona istnieje, używając funkcji `method_exists()`:

```
<?php
    class Person{
        private $name = '';
        public $secondName = '';

        public function setName(string $name): void
        {
            $this->name = $name;
        }

        public function getName(): string
        {
            return $this->name;
        }
    }
    $var = new Person();
    if (method_exists($var, "getName"))
        echo "Method getName exists";
    else
        echo "Method getName not exists";

?>
```

Tak jak `get_class_vars()` zwraca tablicę właściwości dla klasy, `get_object_vars()` zwraca tablicę właściwości ustawionych w obiekcie:

```php
<?php
    class Person{
        private $name = '';
        public $secondName = '';

        public function setName(string $name): void
        {
            $this->name = $name;
        }

        public function getName(): string
        {
            return $this->name;
        }
    }
    $var = new Person();
    $var->setName("Mateusz");
    $var->secondName = "Miotk";
    var_dump(get_class_vars("Person"));
    var_dump(get_object_vars($var));

?>
```

Poniżej znajduje się kod, który wypisuje wszystkie zadeklarowane klasy, ich metody statyczne i wartości:

```php
<?php
    function displayClasses() {
        $classes = get_declared_classes();
        foreach ($classes as $class) {
            echo "Showing information about {$class}<br />";
            $reflection = new ReflectionClass($class);
            $isAnonymous = $reflection->isAnonymous() ? "yes" : "no";
            echo "Is Anonymous: {$isAnonymous}<br />";
            echo "Class methods:<br />";
            $methods = $reflection->getMethods(ReflectionMethod::IS_STATIC);
            if (!count($methods)) {
                echo "<i>None</i><br />";
            }
            else {
                foreach ($methods as $method) {
                    echo "<b>{$method}</b>()<br />";
                }
            }
            echo "Class properties:<br />";
            $properties = $reflection->getProperties();
            if (!count($properties)) {
                echo "<i>None</i><br />";
            }
            else {
                foreach(array_keys($properties) as $property) {
                    echo "<b>\${$property}</b><br />";
                }
            }
            echo "<hr />";
        }
    }
    displayClasses();
?>
```

Step 4

Serializacja (ang. serializing) obiektu oznacza konwersję go do reprezentacji strumienia bajtowego, która może być przechowywana w pliku. Jest to przydatne w przypadku trwałych danych; na przykład sesje PHP automatycznie zapisują i przywracają obiekty. Serializacja w PHP jest w większości automatyczna - wymaga niewielkiej dodatkowej pracy, poza wywołaniem funkcji `serialize()` oraz `unserialize()` :

```
$encoded = serialize(something);
$something = unserialize(encoded);
```

Serializacja jest najczęściej używana w sesjach PHP, którą ją obsługują. Wystarczy, że wskażemy PHP, które zmienne mają być śledzone, a zostaną one automatycznie zachowane między wizytami na stronach w witrynie. Jednak sesje nie są jedynym zastosowaniem serializacji - jeśli chcemy zaimplementować własną formę trwałych obiektów, `serialize()` i `unserialize()` są naturalnym wyborem. Klasa obiektu musi zostać zdefiniowana, zanim będzie można przeprowadzić odserializację. Próba odserializacji obiektu, którego klasa nie została jeszcze zdefiniowana, umieszcza obiekt w `stdClass`, co czyni go prawie bezużytecznym. Jedną z praktycznych konsekwencji tego jest to, że jeśli używamy sesji PHP do automatycznej serializacji i odserializacji obiektów, to musimy dołączyć plik zawierający definicję klasy obiektu na każdej stronie w serwisie.

```
include "object_definitions.php"; // load object definitions
session_start(); // load persistent variables
?>
<html>...
```

PHP ma posiada dwie metody podczas procesu serializacji i odserializacji: `__sleep()` i `__wakeup()`. Te metody służą do powiadamiania obiektów, że są serializowane lub nieserializowane. Obiekty można serializować, jeśli nie mają tych metod; jednak nie zostaną powiadomieni o tym procesie. Metoda `__sleep()` jest wywoływana na obiekcie tuż przed serializacją; może wykonać dowolne czyszczenie niezbędne do zachowania stanu obiektu, takie jak zamykanie połączeń z bazą danych, wypisywanie niezapisanych trwałych danych i tak dalej. Powinien zwrócić tablicę zawierającą nazwy członków danych, które mają zostać zapisane w strumieniu bajtowym. Jeśli zwrócimy pustą tablicę, żadne dane nie zostaną zapisane. Metoda `__wakeup()` jest wywoływana na obiekcie natychmiast po utworzeniu obiektu ze strumienia bajtowego. Metoda może wykonywać dowolne działania, których wymaga, na przykład ponownie otwieranie połączenia z bazą danych i wykonywać inne zadania inicjujące. Poniżej znajduje się definicja klasy `Log`, która udostępnia dwie przydatne metody: `write()` do dołączania komunikatu do pliku dziennika i `read()` do pobrania aktualnej zawartości pliku dziennika. Używa `__wakeup()`, aby ponownie otworzyć plik dziennika i `__sleep()`, aby zamknąć plik dziennika.

```
//log.php
<?php
    class Log {
        private $filename;
        private $fh;
        function __construct($filename) {
            $this->filename = $filename;
            $this->open();
        }
        function open() {
            $this->fh = fopen($this->filename, 'a') or die("Can't open {$this->filename}");
        }
        function write($note) {
            fwrite($this->fh, "{$note}\n");
        }
        function read() {
            return join('', file($this->filename));
        }
        function __wakeup(): void {
            $this->filename = 'persistent_log';
            $this->open();
        }
        function __sleep() {
            fclose($this->fh);
            return ["filename" => $this->filename];
        }
    }
?>
```

Strona główna HTML poniżej używa klasy `Log` i sesji PHP do utworzenia trwałej zmiennej dziennika, `$logger`.

```
//front.php
<?php
    include_once "log.php";
    session_start();
?>
<html><head><title>Front Page</title></head>
<body>
<?php
    $now = strftime("%c");
    if (!isset($_SESSION['logger'])) {
        $logger = new Log("persistent_log");
        $_SESSION['logger'] = $logger;
        $logger->write("Created $now");
        echo("<p>Created session and persistent log object.</p>");
    }
    else {
        $logger = $_SESSION['logger'];
    }
    $logger->write("Viewed first page {$now}");
    echo "<p>The log contains:</p>";
    echo nl2br($logger->read());
?>
<a href="next.php">Move to the next page</a>
</body></html>
```

Plik `next.php`, jest stroną HTML. Podążając za linkiem ze strony głównej do tej strony wyzwala wczytywanie trwałego obiektu `$logger`. Wywołanie `___wakeup()` ponownie otwiera plik dziennika, dzięki czemu obiekt jest gotowy do użycia.