Объекты и классы

При создании программы на PHP и отдельных ее блоков нам вполне может хватить той функциональности, которую представляют функции. Однако PHP имеет и другие возможности по созданию программ, которые представляет объектно-ориентированное программирование. В ряде случаев программы, использующие ООП, проще в понимании, их легче поддерживать и изменять.

Ключевыми понятиями парадигмы ООП являются понятия "класс" и "объект". Описанием объекта является класс, а объект представляет экземпляр этого класса. Можно провести следующую аналогию: например, у каждого человека есть имя, определенный возраст, вес, какие-то другие параметры. То есть некоторый шаблон, который содержит набор параметров человека - этот шаблон можно назвать классом. А реально же существующий человек с конкретным именем, возрастом, весом и т.д. является объектом или экземпляром этого класса.

Для создания класса в PHP используется ключевое слово class, после которого идет название класса и фигурные скобки {} - блок кода класса. Например, новый класс, представляющий пользователя:

<?php

class Person

{}

?>

Чтобы создать объект класса Person, применяется ключевое слово new:

<?php

class Person

{ }

$person = new Person();

print\_r($person);

?>

В данном случае переменная $person является объектом класса Person. С помощью функции print\_r() можно вывести содержимое объекта, как и в случае с массивами.

При этом неважно, определяется класс до или после создания объекта. Например, мы можем сначала определить переменную класса, а потом определить этот класс:

<?php

$person = new Person();

class Person

{ }

?>

Свойства и методы

Класс может содержать переменные, которые описывают какие-то признаки объекта, его состояние и которые еще назывют свойствами или атрибутам. И также класс класс может содержать функции, которые еще назвают методами и которые определяют его поведение.

Так, добавим в класс Person несколько свойств и методов:

<?php

class Person

{

    public $name, $age;

    function hello()

    {

        echo "Hello!<br>";

    }

}

$tom = new Person();

$tom->name = "Tom"; // установка свойства $name

$tom->age = 36; // установка свойства $age

$personName = $tom->name;    // получение значения свойства $name

echo "Имя пользователя: " . $personName . "<br>";

$tom->hello(); // вызов метода hello()

print\_r($tom);

?>

Здесь класс Person содержит два свойства: $name и $age. Свойства объявляются как обычные переменные, перед которыми стоит модификатор доступа - в данном случае модификатор public.

Методы представляют обычные функции, которые выполняют определенные действия. Здесь функция hello() просто выводит приветствие.

После создания объекта класса Person:

<?php

class Person

{

    public $name, $age;

    public function hello()

    {

        echo "Hello!<br>";

    }

}

$tom = new Person();

$tom->name = "Tom"; // установка свойства $name

$tom->age = 36; // установка свойства $age

$personName = $tom->name;    // получение значения свойства $name

echo "Имя пользователя: " . $personName . "<br>";

$tom->hello(); // вызов метода hello()

print\_r($tom);

?>

<?php

$tom = new Person();

?>

Мы можем через имя переменной класса обращаться к его свойствам и методам. Чтобы обратиться к свойствам и методам объекта применяется оператор доступа ->. Например, установить значения свойств:

<?php

$tom->name = "Tom"; // установка свойства $name

$tom->age = 36; // установка свойства $age

?>

Или получить значение (например, присвоить его переменной):

<?php

$personName = $tom->name;    // получение значения свойства $name

?>

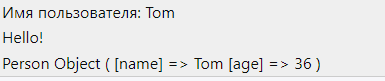
Или вызвать методы объекта:

<?php

$tom->hello(); // вызов метода hello()

?>

В итоге мы получим следующий вывод браузера:



При этом свойствам можно задать в классе некоторые начальные значения:

<?php

class Person

{

    public $name = "Undefined", $age = 18;

    function hello()

    {

        echo "Hello!<br>";

    }

}

$tom = new Person();

$tom->age = 36; // установка свойства $age

echo "Имя пользователя: " . $tom->name . "<br>";

echo "Возраст пользователя: " . $tom->age . "<br>";

?>

Вывод брузера:



Ключевое слово this

Для обращения к свойствам и методам объекта внутри его класса применяется ключевое слово this. Например, определим в классе метод для вывода информации об объекте:

<?php

class Person

{

    public $name = "Undefined", $age = 18;

    function displayInfo()

    {

        echo "Name: " . $this->name ."; Age: " . $this->age . "<br>";

        // также можно написать

        // echo "Name: $this->name; Age: $this->age<br>";

    }

}

$tom = new Person();

$tom -> name = "Tom";

$tom -> displayInfo();   // Name: Tom; Age: 18

?>

<?php

class Person

{

    public $name = "Undefined", $age = 18;

    function displayInfo()

    {

        echo "Name: $this->name; Age: $this->age<br>";

    }

}

$tom = new Person();

$tom -> name = "Tom";

$tom -> displayInfo();

$bob = new Person();

$bob -> name = "Bob";

$bob -> age = 25;

$bob -> displayInfo();

?>

При вызове

<?php

$tom -> displayInfo();

?>

$this фактически будет указывать на переменную $tom. Тогда как при вызове

<?php

$bob -> displayInfo();

?>

$this будет указывать на переменную $bob.

Вывод браузера:



Сравнение объектов

При сравнении объектов классов следует принимать во внимание ряд особенностей. В частности, при использовании оператора равенства == два объекта считаются равными, если они представляют один и тот же класс и их свойства имеют одинаковые значения.

А при использовании оператора эквивалентности === оба объекта считаются равными, если обе переменных классах указывают на один и тот же экземпляр класса.

Рассмотрим на примере:

<?php

class Person

{

    public $name, $age;

    function displayInfo()

    {

        echo "Name: $this->name; Age: $this->age<br>";

    }

}

$tom = new Person();

$tom -> name = "Tom";

$tom -> age = 36;

$tomas = new Person();

$tomas -> name = "Tom";

$tomas -> age = 36;

if($tom == $tomas) echo "переменные tom и tomas равны<br>";

else echo "переменные tom и tomas НЕ равны<br>";

if($tom === $tomas) echo "переменные tom и tomas эквивалентны";

else echo "переменные tom и tomas НЕ эквивалентны";

?>

Здесь сравниваются две переменных - $tom и $tomas. Они представляют один и тот же класс Person, и их свойства имеют одни и те же значения. Однако они представляют разные объекты. Поэтому при сравнении оператор == возвратит true, а оператор === - false:



Возьмем другой пример, когда обе переменных представляют один и тот же объект:

<?php

$person = new Person();

$tom = $person;

$tom -> name = "Tom";

$tom -> age = 36;

$tomas = $person;

if($tom == $tomas) echo "переменные tom и tomas равны<br>";

else echo "переменные tom и tomas НЕ равны<br>";

if($tom === $tomas) echo "переменные tom и tomas эквивалентны";

else echo "переменные tom и tomas НЕ эквивалентны";

?>

Здесь объект класса Person создается только один раз: $person = new Person();. И затем обе переменных $tom и $tomas будут указывать на этот объект. При этом не имеет значения, для какой именно переменной мы устанавливаем свойства. Так как в реальности это будет один и тот же объект. В итоге и оператор ==, и оператор === при сравнении возвратят true



Конструкторы и деструкторы

Конструкторы представляют специальные методы, которые выполняются при создании объекта и служат для начальной инициализации его свойств. Для создания конструктора надо объявить функцию с именем \_\_construct (с двумя подчеркиваниями впереди):

<?php

class Person

{

    public $name, $age;

    function \_\_construct($name, $age)

    {

        $this->name = $name;

        $this->age = $age;

    }

    function displayInfo()

    {

        echo "Name: $this->name; Age: $this->age<br>";

    }

}

$tom = new Person("Tom", 36);

$tom -> displayInfo();

$bob = new Person("Bob", 41);

$bob -> displayInfo();

?>

Функция конструктора в данном случае принимает два параметра. Их значения передаются свойствам класса. И теперь чтобы создать объект, нам надо передать значения для соответствующих параметров:

<?php

$tom = new Person("Tom", 36);

?>

Вывод браузера:



Параметры по умолчанию

Чтобы сделать конструктор более гибким, мы можем обозначить один или несколько параметров в качестве необязательных. Тогда при создании объекта необязательно указывать все параметры. Например, изменим конструктор следующим образом:

<?php

function \_\_construct($name="Том", $age=36)

{

    $this->name = $name;

    $this->age = $age;

}

?>

Таким образом, если не будут заданы параметры, вместо них будут использоваться значения "Том" и 36. И теперь мы можем создать объект Person несколькими способами:

<?php

class Person

{

    public $name, $age;

    function \_\_construct($name="Том", $age=36)

    {

        $this->name = $name;

        $this->age = $age;

    }

    function displayInfo()

    {

        echo "Name: $this->name; Age: $this->age<br>";

    }

}

$tom = new Person();

$tom -> displayInfo();

$bob = new Person("Bob");

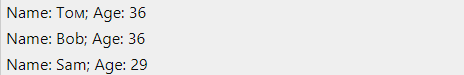
$bob -> displayInfo();

$sam = new Person("Sam", 29);

$sam -> displayInfo();

?>

Вывод браузера:



Объявление свойств через конструктор

Начиная с версии PHP 8 в языке появилась возможность определить свойства через список параметров конструктора. Любой параметр конструктора, который имеет модификатор доступа, например, public, будет автоматически представлять новое свойство.

Например, определим свойства name и age напрямую через параметры конструктора:

<?php

class Person

{

    function \_\_construct(public $name, public $age)

    {

        $this->name = $name;

        $this->age = $age;

    }

    function displayInfo()

    {

        echo "Name: $this->name; Age: $this->age<br>";

    }

}

$tom = new Person("Tom", 36);

$tom -> displayInfo();

?>

Хотя в данном случае в классе явным образом не определены свойства name и age, но поскольку в списке параметров конструктора заданы параметры с такими именами и с модификатором доступа (в данном случае public), то у класса автоматически создаются подобные переменные.

Можно сочетать оба способа объявления переменных:

<?php

class Person

{

    public $name;

    function \_\_construct($name, public $age)

    {

        $this->name = $name;

        $this->age = $age;

    }

    function displayInfo()

    {

        echo "Name: $this->name; Age: $this->age<br>";

    }

}

?>

При подобном объявлении свойств также можно передавать им значения по умолчанию:

<?php

class Person

{

    public $name;

    function \_\_construct($name = "Sam", public $age = 33)

    {

        $this->name = $name;

        $this->age = $age;

    }

    function displayInfo()

    {

        echo "Name: $this->name; Age: $this->age<br>";

    }

}

?>

Деструкторы

Деструкторы служат для освобождения ресурсов, используемых программой - для освобождения открытых файлов, открытых подключений к базам данных и т.д. Деструктор объекта вызывается самим интерпретатором PHP после потери последней ссылки на данный объект в программе.

Деструктор определяется с помощью функции \_\_destruct (два подчеркивания впереди):

<?php

class Person

{

    public $name, $age;

    function \_\_construct($name, $age)

    {

        $this->name = $name;

        $this->age = $age;

    }

    function getInfo()

    {

        echo "Имя: $this->name ; Возраст: $this->age <br>";

    }

    function \_\_destruct()

    {

        echo "Вызов деструктора";

    }

}

?>

Функция деструктора определяется без параметров, и когда на объект не останется ссылок в программе, он будет уничтожен, и при этом будет вызван деструктор.

Анонимные классы

Анонимные классы - это классы, которые не имеют имени. Обычно такие классы полезны, если нам необходимо один раз создать объект подобного класса. И больше этот класс не будет использоваться. Например:

<?php

$person = new class {};

?>

Для определения переменной аонимного класса, как в случае с обычным классом применяется оператор new, после которого идет ключевое слово class и затем - тело класса в фигурных скобках.

Анонимные классы, как и обычные классы могут определять свойства и методы. Например:

<?php

$person = new class {

    public $name;

    function sayHello(){

        echo "Hello!<br>";

    }

};

$person->sayHello();

$person -> name = "Sam";

echo "Name: " . $person -> name . "<br>";

?>



Также анонимные классы могут определять конструкторы:

<?php

$person = new class("Bob") {

    public $name;

    function \_\_construct($name)

    {

        $this->name = $name;

    }

    function sayHello(){

        echo "Hello!<br>";

    }

};

echo "Hello, " . $person -> name . "<br>"; // Hello, Bob

?>

При этом вызов конструктора идет сразу после ключевого слова class до открывающей фигурной скобки: new class("Bob") {.

Подобным образом можно определять свойства сразу в конструкторе, сократив тем самым определение класса:

<?php

$person = new class("Bob", 34) {

    function \_\_construct(public $name, public $age)

    {

        $this->name = $name;

    }

    function displayInfo()

    {

        echo "Name: $this->name; Age: $this->age<br>";

    }

};

$person -> displayInfo();

?>



Наследование

Наследование является одним из основных аспектов объектно-ориентированного программирования. Наследование позволяет классу взять функционал уже имеющихся классов и при необходимости переопределить его. Если у нас есть какой-нибудь класс, в котором не хватает пары функций, то гораздо проще переопределить имеющийся класс, написав пару строк, чем создавать новый с нуля, переписывая кучу кода.

Чтобы наследовать один класс от другого, нам надо применить оператор extends. Стоит отметить, что в PHP мы можем унаследовать класс только от одного класса. Множественное наследование не поддерживается.

Например, унаследуем класс Employee от класса Person:

<?php

class Person

{

    public $name;

    function \_\_construct($name)

    {

        $this->name = $name;

    }

    function displayInfo()

    {

        echo "Имя: $this->name<br>";

    }

}

class Employee extends Person

{}

$tom = new Employee("Tom");

$tom -> displayInfo();

?>

В данном случае предположим, что класс Person представляет человека в целом, а класс Employee - работника некого предприятия. В этой связи каждый работник преддставляет человека. И чтобы не дублировать один и тот же функционал, лучше в данном случае унаследовать класс работника - Employee от класа человека - Person. В этой паре класс Person еще называется родительским или базовым классом, а класс - Employee - производным классом или классом-наследником.

И так как класс Employee унаследован от Person, для объектов класса Employee мы можем использовать функционал родительского класса Person. Так, для создания объекта Employee в данном случае вызывается конструктор, который определен в классе Person и который в качестве параметра принимает имя человека:

<?php

$tom = new Employee("Tom");

?>

И также у переменной типа Employee вызывается метод displayInfo, который определен в классе Person:

<?php

$tom -> displayInfo();

?>

Переопределение функционала

Унаследовав функционал от родительского класса класс-наследник может добавить свои свойства и методы или переопредилить унаследованный функционал. Например, изменим класс Employee, добавив в него данные о компании, где работает работник:

<?php

class Person

{

    public $name;

    function \_\_construct($name)

    {

        $this->name = $name;

    }

    function displayInfo()

    {

        echo "Имя: $this->name<br>";

    }

}

class Employee extends Person

{

    public $company;

    function \_\_construct($name, $company)

    {

        $this->name = $name;

        $this->company = $company;

    }

    function displayInfo()

    {

        echo "Имя: $this->name<br>";

        echo "Работает в $this->company<br>";

    }

}

$tom = new Employee("Tom", "Microsoft");

$tom -> displayInfo();

?>

Здесь класс Employee добавляет новое свойство - $company, которое хранит компанию работника. Также класс Employee переопределил конструктор, в который пеередаются данные для имени и компании. А также переопределен метод displayInfo(). Соответственно для создания объекта класса Employee, теперь необходимо использовать переопределенный в классе Employee конструктор:

<?php

$tom = new Employee("Tom", "Microsoft");

?>

Класс-наследник переопределяет конструктор родительского класса, то для создания объекта класса-наследника необходимо использовать переопределенный в нем конструктор.

И также изменится поведение метода displayInfo(), который кроме имени также выведет и компанию работника:



Вызов функционала родительского класса

Если мы посмотрим на код класса-наследника Employee, то можем увидеть части кода, которые повторяют код класса Person. Например, установка имени в конструкторе:

<?php

$this->name = $name;

?>

Также вывод имени работника в методе displayInfo():

<?php

echo "Имя: $this->name<br>";

?>

В обоих случаях речь идет об одной строке кода. Однако что, если конструктор Employee повторяет установку не одного, а десятка свойств. Соответственно что, если метод displayInfo в классе-наследнике повторяет горадо больше действий родительского класса. В этом случае горадо рациональнее не писать повторяющийся код в классе-наследнике, а вызвать в нем соответствующий функционал родительского класса.

Если нам надо обратиться к методу родительского класса, то мы можем использовать ключевое слово parent, после которого используется двойное двоеточие :: и затем вызываемый метод.

Например, перепишем предыдущий пример:

<?php

class Person

{

    public $name;

    function \_\_construct($name)

    {

        $this->name = $name;

    }

    function displayInfo()

    {

        echo "Имя: $this->name<br>";

    }

}

class Employee extends Person

{

    public $company;

    function \_\_construct($name, $company)

    {

        parent::\_\_construct($name);

        $this->company = $company;

    }

    function displayInfo()

    {

        parent::displayInfo();

        echo "Работает в $this->company<br>";

    }

}

$tom = new Employee("Tom", "Microsoft");

$tom -> displayInfo();

?>

Теперь в конструкторе Employee вызывается конструктор базового класса:

<?php

parent::\_\_construct($name);

?>

В нем собственно и происходит установка имени. И подобным образом в методе displayInfo() вызывается реализация метода класса Person:

<?php

parent::displayInfo();

?>

В итоге мы получим тот же самый результат.

Стоит отметить, что в реальности ключевое слово parent заменяет название класса. То есть мы также могли вызывать функционал родительского класса через имя этого класса:

<?php

class Employee extends Person

{

    public $company;

    function \_\_construct($name, $company)

    {

        Person::\_\_construct($name);

        $this->company = $company;

    }

    function displayInfo()

    {

        Person::displayInfo();

        echo "Работает в $this->company<br>";

    }

}

?>

Оператор instanceof

Оператор instanceof позволяет проверить принадлежность объекта определенному класса. Слева от оператора располагается объект, котоый надо проверить, а справа - название класса. И если объект представляет класс, то оператор возвращает true. Например:

<?php

class Person

{

    public $name;

    function \_\_construct($name)

    {

        $this->name = $name;

    }

    function displayInfo()

    {

        echo "Имя: $this->name<br>";

    }

}

class Employee extends Person

{

    public $company;

    function \_\_construct($name, $company)

    {

        Person::\_\_construct($name);

        $this->company = $company;

    }

    function displayInfo()

    {

        Person::displayInfo();

        echo "Работает в $this->company<br>";

    }

}

class Manager{}

$tom = new Employee("Tom", "Microsoft");

$tom instanceof Employee;   // true

$tom instanceof Person;     // true

$tom instanceof Manager;    // false

?>

Здесь переменная $tom представляет класс Employee, поэтому $tom instanceof Employee возвращает true.

Так как класс Employee унаследован от Person, то переменная $tom также представляет класс Person (работник также является человеком).

А вот класс Manager переменная $tom не преддставляет, поэтому выражение $tom instanceof Manager возвращает false.

Запрет наследования и оператор final

В примере выше метод displayInfo() переопределялся классом-наследником. Однако иногда возникают ситуации, когда надо запретить переопределение методов. Для этого в классе-родителе надо указать методы с модификатором final:

<?php

class Person

{

    public $name;

    function \_\_construct($name)

    {

        $this->name = $name;

    }

    final function displayInfo()

    {

        echo "Имя: $this->name<br>";

    }

}

class Employee extends Person

{

    public $company;

    function \_\_construct($name, $company)

    {

        Person::\_\_construct($name);

        $this->company = $company;

    }

    function displayEmployeeInfo()

    {

        Person::displayInfo();

        echo "Работает в $this->company<br>";

    }

}

$tom = new Employee("Tom", "Microsoft");

$tom -> displayEmployeeInfo();

?>

В этом случае во всех классах-наследниках от класса Person мы уже не сможем определить метод с таким же именем. Поэтому в данном случае в классе Employee определен новый метод - displayEmployeeInfo.

Также мы можем вообще запретить наследование от класса. Для этого данный класс надо определить с модификатором final:

<?php

final class Person

{

    public $name;

    function \_\_construct($name)

    {

        $this->name = $name;

    }

    final function displayInfo()

    {

        echo "Имя: $this->name<br>";

    }

}

?>

Теперь мы не сможем унаследовать класс Employee (да и никакой другой класс) от класса Person.

Модификаторы доступа

С помощью специальных модификаторов можно задать область видимости для свойств и методов класса. В PHP есть три таких модификатора:

public: к свойствам и методам, объявленным с данным модификатором, можно обращаться из внешнего кода и из любой части программы

protected: свойства и методы с данным модификатором доступны из текущего класса, а также из классов-наследников

private: свойства и методы с данным модификатором доступны только из текущего класса

Рассмотрим на примере. Например, у нас есть следующий класс:

<?php

class Person

{

    private $privateA ="private";

    public  $publicA = "public";

    protected $protectedA = "protected";

    private function getPrivateMethod()

    {

        echo "private method <br />";

    }

    protected function getProtectedMethod()

    {

        echo "protected method <br />";

    }

    public function getPublicMethod()

    {

        echo "public method <br />";

    }

    function test()

    {

        $this->getPrivateMethod();

        $this->getProtectedMethod();

        $this->getPublicMethod();

        echo "$this->privateA <br />";

        echo "$this->protectedA <br />";

        echo "$this->publicA <br />";

    }

}

?>

Класс определяет три свойства и три метода с разными модификаторами доступа. Из любого метода этого класса мы можем обратиться к любом методу и любому свойству. Если метод не имеет модификатора доступа, то по умолчанию его видимость аналогична модификатору public.

Теперь создадим класс, производный от класса Person:

<?php

class Employee extends Person

{

    function test()

    {

        //echo $this->privateA; // нельзя, так как privateA - private в классе-родителе

        echo $this->protectedA;

        echo $this->publicA;

        //$this->getPrivateMethod(); // нельзя, так как private в классе-родителе

        $this->getProtectedMethod();

        $this->getPublicMethod();

    }

}

?>

Классу-наследнику доступны все свойства и методы с модификаторами public и protected, но недоступны методы и свойства с модификатором private.

Теперь рассмотрим использование класса Person во внешнем коде:

<?php

$person = new Person;

// $person->getPrivateMethod(); // недоступно, так как private

// $person->getProtectedMethod(); // недоступно, так как protected

$person->getPublicMethod();

// echo $person->privateA; // недоступно, так как private

// echo $person->protectedA; // недоступно, так как protected

echo $person->publicA;

?>

При использовании объектов данного класса нам доступны только публичные методы и свойства, а свойства и методы с модификаторами private и protected не доступны.

Доступ к методам и свойствам объекта

Стоит отметить, что в PHP в классе можно обращаться в переменным и методам с модификаторами private и protected объекта этого же класса:

<?php

class Account{

    private $sum = 0;

    function \_\_construct($sum){

        $this->sum = $sum;

    }

    function getSumFrom($otherAccount, $money){

        $otherAccount -> sum -= $money;

        $this -> sum += $money;

    }

    function prinSum(){

        echo "На счете $this->sum у.е.<br>";

    }

}

$acc1 = new Account(100);

$acc2 = new Account(400);

$acc1->getSumFrom($acc2, 200);

$acc1->prinSum();    // На счете 300 у.е.

?>

Класс Account представляет условный класс банковского счета. В приватной переменной $sum хранится сумма на счете.

В методе getSumFrom() в качестве параметра получаем объект этого же класса Account и сумму перевода. И поскольку объект представляет тот же класс Account, то мы можем обратиться к его свойствам и методам с модификаторами private и protected:

<?php

$otherAccount -> sum -= $money;

?>

И таким образом, выполнить условный денежный перевод с счета на счет.

Магический метод get

Следующий магический метод, который мы с вами разберем, называется \_\_get. Этот метод срабатывает при попытке прочитать значение приватного или защищенного свойства.

Если реализовать метод \_\_get в каком-нибудь классе, то все обращения к несуществующим или скрытым свойствам будут обрабатываться этим методом.

При этом PHP автоматически будет передавать имя запрошенного свойства в первый параметр этого метода, а возвращаемое этим методом значение будет воспринято как значение свойства, к которому произошло обращение.

Скорее всего пока не очень понятно, как это работает, поэтому давайте посмотрим на практическом примере. Пусть у нас есть вот такой класс Test с приватным и публичным свойствами:

<?php

    class Test

    {

        public $prop1 = 1; // публичное свойство

        private $prop2 = 2; // приватное свойство

    }

?>

Давайте добавим в наш класс магический метод \_\_get, который для начала будет просто возвращать имя свойства, к которому произошло обращение:

<?php

    class Test

    {

        public $prop1 = 1;

        private $prop2 = 2;

        public function \_\_get($property)

        {

            return $property; // просто вернем имя свойства

        }

    }

?>

Давайте проверим работу созданного магического метода. Обратимся к трем типам свойств: к публичному свойству, к приватному и к несуществующему:

<?php

    $test = new Test;

    // Обращаемся к публичному свойству:

    echo $test->prop1; // выведет 1 - то есть значение свойства

    // Обращаемся к приватному свойству:

    echo $test->prop2; // выведет 'prop2' - имя свойства

    // Обращаемся к несуществующему свойству:

    echo $test->prop3; // выведет 'prop3' - имя свойства

?>

Как вы видите, наш магический метод реагирует на обращение к приватным и несуществующим свойствам, но игнорирует обращение к публичным - они работают так, как и работали раньше.

Применение: свойства только для чтения

Пусть теперь в нашем классе все свойства приватные:

<?php

    class Test

    {

        private $prop1 = 1;

        private $prop2 = 2;

    }

?>

Давайте сделаем так, чтобы эти свойства во внешнем мире были доступны только для чтения. Ранее мы такое уже делали, создавая геттеры для каждого свойства и не создавая сеттеры.

Давайте теперь для решения этой задачи воспользуемся магическим методом \_\_get. Будем возвращать в нем значение запрошенного свойства. Как это сделать: имя запрошенного свойства попадает в параметр метода \_\_get, в нашем случае $property.

Это значит, что мы можем прочитать свойство, имя которого хранится в переменной, вот так: $this->$property (имя свойства будет переменной, то есть с долларом вначале, мы это проходили в предыдущих уроках).

Давайте сделаем описанный метод \_\_get:

<?php

    class Test

    {

        private $prop1 = 1;

        private $prop2 = 2;

        public function \_\_get($property)

        {

            return $this->$property;

        }

    }

?>

Воспользуемся им для чтения свойств:

<?php

    $test = new Test;

    echo $test->prop1; // выведет 1

    echo $test->prop2; // выведет 2

?>

Попытка записать что-то в свойство приведет к ошибке:

<?php

    $test = new Test;

    $test->prop1 = 2; // выдаст ошибку

?>

Это именно то, что нам нужно: свойство можно прочитывать, но нельзя записывать.

Попытка прочитать несуществующее свойство выдаст ошибку:

<?php

    $test = new Test;

    echo $test->prop3; // выдаст ошибку

?>

Обратите также внимание на следующий нюанс: когда мы делали свойства только для чтения старым способом, то для того, чтобы прочитать свойство, мы использовали метод-геттер.

В новом способе мы будем обращаться именно к свойствам, будто они публичные. Но записать в них не сможем, будто они приватные.

Магический метод set

Магический метод \_\_set вызывается при попытке изменить значение несуществующего или скрытого свойства. В качестве параметров он принимает имя свойства и значение, которое ему пытаются присвоить.

Давайте посмотрим на практическом примере. Пусть у нас дан вот такой класс Test:

<?php

    class Test

    {

        private $prop1;

        private $prop2;

    }

?>

Давайте сделаем в этом классе магический метод \_\_set, который с помощью функции var\_dump будет выводить имя свойства, к которому произошло обращение, и значение, которое этому свойству пытаются установить:

<?php

    class Test

    {

        private $prop1;

        private $prop2;

        public function \_\_set($property, $value)

        {

            var\_dump($property . ' ' .$value);

        }

    }

?>

Проверим работу нашего класса:

<?php

    $test = new Test;

    $test->prop = 'value'; // var\_dump метода \_\_set выведет 'prop value'

?>

Давайте теперь будем устанавливать значение свойству, имя которого хранится в переменной $property:

<?php

    class Test

    {

        private $prop1;

        private $prop2;

        public function \_\_set($property, $value)

        {

            $this->$property = $value; // устанавливаем значение

        }

    }

?>

Теперь мы сможем записывать в приватные свойства снаружи класса:

<?php

    $test = new Test;

    $test->prop1 = 1; // запишем 1

    $test->prop2 = 2; // запишем 2

?>

Записывать мы можем, однако, проверить, записалось ли туда что-то - нет, так как свойства приватные.

Можно сделать геттер для этих свойств или просто воспользоваться магическим методом \_\_get. Воспользуемся вторым вариантом:

<?php

    class Test

    {

        private $prop1;

        private $prop2;

        public function \_\_set($property, $value)

        {

            $this->$property = $value;

        }

        // Магический геттер свойств:

        public function \_\_get($property)

        {

            return $this->$property;

        }

    }

?>

Вот теперь мы можем проверить работу нашего класса. Проверим:

<?php

    $test = new Test;

    $test->prop1 = 1; // запишем 1

    $test->prop2 = 2; // запишем 2

    echo $test->prop1; // выведет 1

    echo $test->prop2; // выведет 2

?>

На самом деле, конечно же, не стоит разрешать всем подряд записывать в приватные свойства, иначе пропадает суть этих приватных свойств (проще сделать их публичными и все).

Поэтому данный метод следует применять только тогда, когда в этом действительно есть необходимость. Ниже мы еще рассмотрим примеры удачного применения.

Несуществующее свойство

Давайте попробуем записать данные в несуществующее свойство - это будет работать:

<?php

    $test = new Test;

    $test->prop3 = 3; // запишем 3

    echo $test->prop3; // выведет 3

?>

Пусть мы не хотим разрешать записывать в несуществующие свойства. И, вообще, хотим разрешить запись только в свойства prop1 и prop2.

Это легко сделать - достаточно в методе \_\_set добавить соответствующее условие:

<?php

    class Test

    {

        private $prop1;

        private $prop2;

        public function \_\_set($property, $value)

        {

            // Напишем условие:

            if ($property == 'prop1' or $property == 'prop2') {

                $this->$property = $value;

            }

        }

        public function \_\_get($property)

        {

            return $this->$property;

        }

    }

?>

Если таких свойств будет много, то не очень удобно перечислять их все в условии.

Давайте запишем разрешенные для записи свойства в массив и будем проверять наличие свойства в этом массиве с помощью функции in\_array:

<?php

    class Test

    {

        private $prop1;

        private $prop2;

        public function \_\_set($property, $value)

        {

            $properties = ['prop1', 'prop2']; // разрешенные свойства

            if (in\_array($property, $properties)) {

                $this->$property = $value;

            }

        }

        public function \_\_get($property)

        {

            return $this->$property;

        }

    }

?>

Проверка при записи

Давайте будем проверять значения свойств на соответствие определенному условию:

<?php

    class Test

    {

        private $prop1;

        private $prop2;

        public function \_\_set($property, $value)

        {

            switch($property) {

                case 'prop1':

                    // Если prop1 от 0 до 10:

                    if ($value > 0 and $value < 10) {

                        $this->$property = $value;

                    }

                break;

                case 'prop2':

                    // Если prop2 от 10 до 20:

                    if ($value > 10 and $value < 20) {

                        $this->$property = $value;

                    }

                break;

                default:

                    // Такого свойства нет

                break;

            }

        }

        public function \_\_get($property)

        {

            return $this->$property;

        }

    }

?>

Обычные setters and getters:

<?php

    class User

    {

        private $name;

        private $age;

        public function getName()

        {

            return $this->name;

        }

        public function setName($name)

        {

            if ($name != '') { // проверяем имя на непустоту

                $this->name = $name;

            }

        }

        public function getAge()

        {

            return $this->age;

        }

        public function setAge($age)

        {

            if ($age >= 0 and $age <= 70) { // проверяем возраст

                $this->age = $age;

            }

        }

    }

?>

Пример использования строгой типизации:

<?php

class BlogData

{

    /\*\* @var string \*/

    private $title;

    /\*\* @var Status \*/

    private $status;

    /\*\* @var \DateTimeImmutable|null \*/

    private $publishedAt;

    /\*\*

     \* @param string $title

     \* @param Status $status

     \* @param \DateTimeImmutable|null $publishedAt

     \*/

    public function \_\_construct(

        $title,

        $status,

        $publishedAt = null

    ) {

        $this->title = $title;

        $this->status = $status;

        $this->publishedAt = $publishedAt;

    }

    /\*\*

     \* @return string

     \*/

    public function getTitle()

    {

        return $this->title;

    }

    /\*\*

     \* @return Status

     \*/

    public function getStatus()

    {

        return $this->status;

    }

    /\*\*

     \* @return \DateTimeImmutable|null

     \*/

    public function getPublishedAt()

    {

        return $this->publishedAt;

    }

}

Обратите внимание так же на ключевое слово readonly:

<?php

class User {

    public readonly int $uid;

    public function \_\_construct(int $uid) {

        $this->uid = $uid;

    }

}

$user = new User(42);

Статические методы и свойства

Кроме обычных методов и свойств с разными модификаторами доступа класс может содержать статические методы и свойства. Такие методы и свойства помечаются ключевым словом static. Статические методы и свойства содаются один раз для всего класса и относятся ко всему классу, тогда для нестатических свойств и методов создается отдельная копия для каждого объекта.

<?php

class Person

{

    public $name, $age;

    static $retirenmentAge = 65;

    function \_\_construct($name, $age)

    {

        $this->name = $name;

        $this->age = $age;

    }

    function sayHello()

    {

        echo "Привет, меня зовут $this->name<br>";

    }

    static function printPerson($person)

    {

        echo "Имя: $person->name    Возраст: $person->age<br>";

    }

}

?>

Здесь определено статическое свойство retirenmentAge, которое хранит пенсионный возраст. Если возраст - это показатель конкретного человека и может отличаться для разных людей, то пенсионный возраст, как правило, устанавливается общий для всех. Поэтому это свойство можно сделать статическим - оно относится ко всему классу Person, а не устанавливается отдельно для каждого объекта. Для объявления статического свойства перед его именем ставится модификатор static:

<?php

static $retirenmentAge = 65;

?>

Также в классе определен статический метод printPerson(), который выводит информацию о человеке, который в качестве параметра передается в метод. Этот метод также не зависит от конкретного объекта класса Person, а относится ко всему классу Person в целом. Чтобы объявить статический метод, также перед словом function ставится модификатор static:

<?php

static function printPerson($person)

{

    echo "Имя: $person->name    Возраст: $person->age<br>";

}

?>

При обращении к статическим методам и свойствам используется имя класса и оператор ::, вместо операции доступа ->, так как статический метод относится ко всему классу, а не к конкретному объекту этого класса.

<?php

class Person

{

    public $name, $age;

    static $retirenmentAge = 65;

    function \_\_construct($name, $age)

    {

        $this->name = $name;

        $this->age = $age;

    }

    function sayHello()

    {

        echo "Привет, меня зовут $this->name<br>";

    }

    static function printPerson($person)

    {

        echo "Имя: $person->name    Возраст: $person->age<br>";

    }

}

$tom = new Person("Tom", 36);

// вызов нестатического метода

$tom->sayHello();

// вызов статического метода

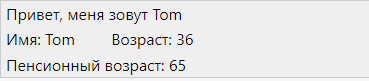
Person::printPerson($tom);

// обращение к статическому свойству

echo "Пенсионный возраст: " . Person::$retirenmentAge . "<br>";

?>

Вывод программы:



Для обращения к статическим свойствам и методам внутри класса вместо имени класса может применяться ключевое слово self. Например, добавим в класс человека метод, который будет вычислять, сколько человеку осталось до пенсии:

<?php

class Person

{

    public $name, $age;

    static $retirenmentAge = 65;

    function \_\_construct($name, $age)

    {

        $this->name = $name;

        $this->age = $age;

    }

    function sayHello()

    {

        echo "Привет, меня зовут $this->name<br>";

    }

    static function printPerson($person)

    {

        echo "Имя: $person->name Возраст: $person->age<br>";

    }

    function checkAge()

    {

        if($this->age >= self::$retirenmentAge)

            echo "Пора на пенсию<br>";

        else

            echo "До пенсии еще " . (Person::$retirenmentAge - $this->age) . " лет<br>";

    }

}

$tom = new Person("Tom", 36);

$tom->checkAge();

?>

Вывод программы:



Стоит отметить, что в статических методах мы можем обращаться только к статическим свойствам и методам. Но не можем обращаться к НЕстатическим свойствам и методам через $this. Например:

<?php

static function printPerson($person)

{

    echo "Имя: $person->name Возраст: $person->age<br>";

    // в статических методах можно обращаться к статическим методам и свойствам

    echo "Пенсионный возраст: " . self::$retirenmentAge . "<br>";

    // но нельзя обращаться к нестатическим методам и свойствам

    // echo "Имя: " . $this->name . "<br>";    // так нельзя

    // $this->sayHello();    // так нельзя

}

?>

Здесь в статическом методе printPerson() мы можем обратиться к статической переменной retirenmentAge:

<?php

echo "Пенсионный возраст: " . self::$retirenmentAge . "<br>";

?>

Но не можем обратиться к нестатическим свойствам и методам:

<?php

// echo "Имя: " . $this->name . "<br>";    // так нельзя

// $this->sayHello();    // так нельзя

?>

При этом статические методы и свойства также могут иметь модификаторы доступа. Например, изменим класс Person следующим образом:

<?php

class Person

{

    public $name;

    private $id;

    private static $counter=0;

    function \_\_construct($name)

    {

        self::$counter++;

        $this->id = self::$counter;

        $this->name = $name;

    }

    static function getCounter()

    {

        return self::$counter;

    }

    function getId()

    {

        return $this->id;

    }

}

$tom = new Person("Tom");

$bob = new Person("Bob");

echo "$tom->name имеет id: " . $tom->getId() . "<br>";

echo "$bob->name имеет id: " . $bob->getId() . "<br>";

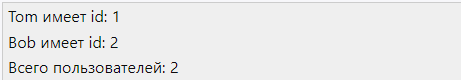
echo "Всего пользователей: " .  Person::getCounter();

?>

В классе Person определено свойство $id, которое представляет идентификатор пользователя. И также определено приватное сттическое свойство $counter, которое хранит общее количество созданных пользователей. В конструкторе мы увеличиваем статическую переменную на единицу, а затем устанавливаем ее значение для свойства $id.

Количество созданных пользователей - это атрибут, общий для всего класса, который независит от конкретного объекта. Однако в то же время нежелательно, чтобы его извне могли произвольно изменить. Поэтому свойство $counter определено как приватное. А чтобы увидеть его значение, определен статический метод getCounter().

Вывод программы:



Интерфейсы

Интерфейс определяет абстрактный дизайн, которому должен соответствовать применяющий его класс. Интерфейс определяет методы без реализации. А класс затем применяет интерфейс и реализует эти методы. Применение интерфейса гарантирует, что класс имеет определенный функционал, описываемый интерфейсом.

Интерфейс определяется с помощью ключевого слова interface, за которым следует имя интерфейса и блок кода интерфейса в фигурных скобках:

<?php

interface Messenger {}

?>

Здесь определен интерфейс Messenger. Внутри блока интерфейса в фигурных скобках определяются сигнатуры методов. Причем все эти методы могут быть только публичными, то есть с модификатором public, либо без модификатора доступа (что аналоично модификатору public):

<?php

interface Messenger

{

    function send();

}

?>

Интерфейсы могут содержать лишь сигнатуры методов - наазвания функций и список параметров в скобках, после которого идет точка с запятой. Так, в данном случае объявлен метод send(). Он не имеет реализации - конкретную реализацию определит класс, который реализует этот интерфейс.

Для реализации классом интерфейса применяется ключевое слово implements, после которого указывается имя применяемого интерфейса:

<?php

interface Messenger

{

    function send();

}

class EmailMessenger implements Messenger

{

    function send()

    {

        echo "Отправка сообщения на e-mail";

    }

}

$outlook = new EmailMessenger();

$outlook->send();

?>

В данном случае класс EmailMessenger реализует интерфейс Messenger. Если класс применяет интерфейс, то он должен реализовать все методы этого интерфейса. Так, в данном случае класс EmailMessenger определяет метод send() с некоторой реализацией.

Интерфейсы также могут наследоваться от других интерфейсов:

<?php

interface Messenger

{

    function send();

}

interface EmailMessenger extends Messenger

{

}

class SimpleEmailMessenger implements EmailMessenger

{

    function send()

    {

        echo "Отправка сообщения на email.";

    }

}

$outlook = new SimpleEmailMessenger();

$outlook->send();

?>

Когда нам могут понадобиться интерфейсы?

Интерфейс - это контракт, который говорит, что класс обязательно реализует определенный функционал. И мы можем использовать это в своей программе. Например, определим следующий код:

<?php

interface Messenger

{

    function send($message);

}

function sendMessage(Messenger $messenger, $text)

{

    $messenger->send($text);

}

class EmailMessenger implements Messenger

{

    function send($message)

    {

        echo "Отправка сообщения на email: $message";

    }

}

$outlook = new EmailMessenger();

sendMessage($outlook, "Hello World");

?>

Для отправки сообщения здесь определена функция sendMessage(), которая в качестве первого параметра принимает объект мессандера, а в качестве второго параметра - отправляемый текст. Причем определение первого параметра говорит, что передаваемое этому параметру значение должно реализовать интерфейс Messenger. В самой функции мы знаем, что первый параметр - это объект, который обязательно реализует интерфейс Messenger, поэтому мы можем вызвать у него метод send() для отправки сообщения:

<?php

function sendMessage(Messenger $messenger, $text)

{

    $messenger->send($text);

}

?>

Множественное применение интерфейсов

Класс может одновременно применять сразу несколько интерфейсов. В этом случае все интерфейсы перечисляются через запятую после слова implements. А класс должен реализовать методы всех применяемых интерфейсов:

<?php

interface Camera

{

    function makeVideo();

    function makePhoto();

}

interface Messenger

{

    function sendMessage($message);

}

class Mobile implements Camera, Messenger

{

    function makeVideo(){ echo "Запись видео";}

    function makePhoto(){ echo "Съемка фото";}

    function sendMessage($message) {echo "Отправка сообщения $message";}

}

$iphone12 = new Mobile();

$iphone12->makePhoto();

?>

Практика интерфейс:

Описать класс, реализующий десятичный счетчик, который может увеличивать или уменьшать свое значение на единицу в заданном диапазоне. Предусмотреть инициализацию счетчика значениями по умолчанию и произвольными значениями. Счетчик имеет два метода: увеличения и уменьшения, — и свойство, позволяющее получить его текущее состояние. Написать программу, демонстрирующую все возможности класса.

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>oop 1</title>

</head>

<body>

    <?php

    /\*\*

     \*

     \*/

    // Define an interface for a counter

    interface CounterInterface

    {

        public function increment(): void;

        public function decrement(): void;

        public function getValue(): int;

    }

    // Implement the CounterInterface

    class DecimalCounter implements CounterInterface

    {

        private int $value;

        private int $min;

        private int $max;

        // Constructor with default values

        public function \_\_construct(int $value = 0, int $min = 0, int $max = 100)

        {

            if ($min > $max) {

                throw new InvalidArgumentException('Minimum value cannot be greater than maximum value.');

            }

            $this->value = $value;

            $this->min = $min;

            $this->max = $max;

        }

        // Method to increment the counter value

        public function increment(): void

        {

            if ($this->value < $this->max) {

                $this->value++;

            }

        }

        // Method to decrement the counter value

        public function decrement(): void

        {

            if ($this->value > $this->min) {

                $this->value--;

            }

        }

        // Method to get the current counter value

        public function getValue(): int

        {

            return $this->value;

        }

    }

    // Function to demonstrate the capabilities of the counter

    function demonstrateCounter(CounterInterface $counter): void

    {

        echo "Initial value: " . $counter->getValue() . "\n";

        $counter->increment();

        echo "After increment: " . $counter->getValue() . "\n";

        $counter->decrement();

        echo "After decrement: " . $counter->getValue() . "\n";

    }

    // Create a counter with default values

    $defaultCounter = new DecimalCounter();

    demonstrateCounter($defaultCounter);

    // Create a counter with custom values

    $customCounter = new DecimalCounter(10, 0, 20);

    demonstrateCounter($customCounter);

    // Attempt to decrement below the minimum value

    for ($i = 0; $i < 15; $i++) {

        $customCounter->decrement();

    }

    echo "After excessive decrements: " . $customCounter->getValue() . "\n";

    // Attempt to increment above the maximum value

    for ($i = 0; $i < 15; $i++) {

        $customCounter->increment();

    }

    echo "After excessive increments: " . $customCounter->getValue() . "\n";

    ?>

    ?>

</body>

</html>

Абстрактные классы и методы

Абстрактный класс представляет частичную реализацию для классов-наследников.

Абстрактный класс определяется с помощью модификатора abstract, который ставится перед именем класса:

<?php

abstract class Messenger { }

?>

Одной из ключевых особенностей абстрактных классов является то, что мы не можем напрямую создать объекты абстрактного класса с помощью вызова его конструктора:

<?php

abstract class Messenger { }

$telegram = new Messenger();    // эта строка не будет работать

?>

Абстрактные классы, как и обычные классы, могут определять переменные и константы, методы и конструкторы.

Но другой особенностью абстрактных классов является то, что они могут содержать абстрактные методы. Это методы, которые не имеют реализации. Реализацию для них предоставляют классы-наследники. Например:

<?php

    abstract class Messenger {

        protected $name;

        abstract function send();

        function \_\_construct($name) {

            $this->name = $name;

        }

        function close() {

            echo "closed<hr>";

        }

    }

    class EmailMessenger extends Messenger {

        function send() {

            echo "message EmailMessenger"."<hr>";

        }

        function printName() {

            echo $this->name."<hr>";

        }

    }

    class UserMessenger extends Messenger {

        function send() {

            echo "message UserMessenger"."<hr>";

        }

    }

    class SomeObject extends UserMessenger {

        function send() {

            echo "message SomeObject"."<hr>";

        }

    }

    // $email = new EmailMessenger("gmail");

    // $email->send("Hello, World!");

    // $email->printName();

    // $email->close();

    function Sender(Messenger &$object) {

        $object->send();

    }

    $email = new EmailMessenger("gmail");

    Sender($email);

    $user = new UserMessenger("Alex");

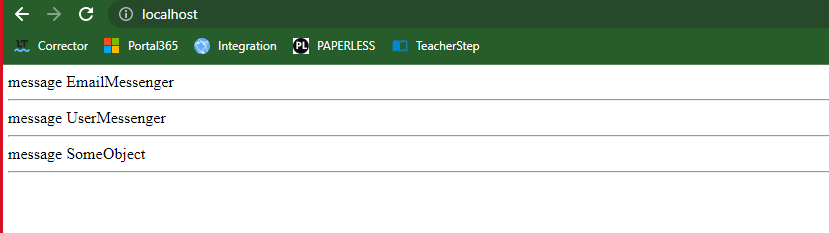
    Sender($user);

    $some = new SomeObject("Some");

    Sender($some);

?>

Result:



Если не определить send у SomeObject, то тогда сработает базовый send

<?php

abstract class Messenger

{

    abstract function send($message);   // абстрактный метод

}

?>

При определении абстрактного метода перед словом function ставится модификатор abstract. А после списка параметров метода - точка с запятой.

Абстрактные методы могут размещаться только в абстрактных классах. Обычный неабстрактный класс не может иметь абстрактных методов.

Если неабстрактный класс наследуется от абстрактного класса, то он обязан реализовать все его абстрактные методы.

Для наследования классом абстрактного класса, как и в общем случае при наследовании, применяется ключевое слово extends. Например:

<?php

abstract class Messenger

{

    protected $name;

    function \_\_construct($name)

    {

        $this->name = $name;

    }

    abstract function send($message);

    function close()

    {

        echo "Выход из мессенджера...";

    }

}

class EmailMessenger extends Messenger

{

    function send($message)

    {

        echo "$this->name отправляет сообщение: $message<br>";

    }

}

$outlook = new EmailMessenger("Outlook");

$outlook->send("Hello PHP 8");

$outlook -> close();

?>

?>

В данном случае класс EmailMessenger наследуется от абстрактного класса Messenger.

Абстрактный класс определяет абстрактный метод send(), поэтому класс-наследник EmailMessenger должен предоставить реализацию для этого метода.

Так, в данном случае мы получим следующий вывод:



Можно заметить, что абстрактные классы похожи на интерфейсы - и те, и другие могут определять методы без реализации, которые реализуются в других классах. Однако, абстрактные классы, как и обычные классы, могут иметь переменные, неабстрактные методы, конструкторы с реализацией, а интерфейсы нет. Кроме того, в PHP один класс может наследоваться только от одного класса, тогда как один класс может применять сразу несколько интерфейсов.

Traits

Traits представляют группу методов, которые могут быть добавлены в классы. Traits позволяют определять блоки функционала и многократно повторно использовать в классах без необходимости усложнять код классов, которые используют эти методы.

Traits определяются с помощью ключевого словва trait, после которого идет название трейта:

<?php

trait Printer { }

?>

Traits могут содержать только статические и нестатические методы:

<?php

trait Printer

{

    public function printSimpleText($text) { echo "$text<br>"; }

    public function printHeaderText($text) { echo "<h2>$text<h2>"; }

}

?>

Для применения классов трейта применяется оператор use, после которого указывается добавляемый трейт:

<?php

trait Printer

{

    public function printSimpleText($text) { echo "$text<br>"; }

    public function printHeaderText($text) { echo "<h2>$text<h2>"; }

}

class Message

{

    use Printer;

}

$myMessage = new Message();

$myMessage->printSimpleText("Hello World!");

$myMessage->printHeaderText("Hello PHP 8");

?>

После добавления трейта с помощью оператора use класс может использовать его методы, как-будто они определены в самом этом классе.

Следует учитывать, что при наследовании методы трейта переопределяют унаследованные методы с тем же именем:

<?php

class Data

{

    function print() { echo "Print from Data"; }

}

trait Printer

{

    function print() { echo "Print from Printer"; }

}

class Message extends Data

{

    use Printer;

}

$myMessage = new Message();

$myMessage->print();        // Print from Printer

?>

example trainse:

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>oop 2</title>

</head>

<body>

    <?php

    trait PrinterTraits {

        public int $id;

        public function \_\_construct(int $id) {

            $this->id = $id;

        }

        public function send\_image($image) : void {

            echo "Printing the $image<hr>";

        }

        public function send\_text(string $text) : void {

            echo "Printing the $text<hr>";

        }

    }

    trait TestTrait {

        public function send\_text\_2(string $text) : void {

            echo "Printing the $text<hr>";

        }

    }

    class Printer {

        use PrinterTraits;

        use TestTrait;

        public string $vender;

    }

    class Message extends Printer {}

    $message = new Message(5);

    $message->send\_text("Hello");

    $message->send\_text\_2("Hello");

    ?>

</body>

</html>

Копирование объектов классов

При присваивании объекта класса другой переменной создается новая ссылка на тот же объект. Например:

<?php

class Person{

    public $name;

    function \_\_construct($name){

        $this->name = $name;

    }

}

$tom = new Person("Tom");

$bob = $tom;

$bob->name = "Bob";

echo $tom->name;    // Bob

?>

В данном случае после присваивания $bob = $tom; обе переменных будут указывать на один и тот же объект. Поэтому если мы поменяем свойство $name у одной переменной, то это измение затронет и другую переменую. Так как они ссылаются на один и тот же объект.

Однако такое поведение может быть нежелательным, если мы хотим, чтобы после копирования переменные представляли независимые друг от друга объекты. И для этого PHP предоставляет оператор clone:

<?php

class Person{

    public $name;

    function \_\_construct($name){

        $this->name = $name;

    }

}

$tom = new Person("Tom");

$bob = clone $tom;      // копируем объект из $tom в переменную $bob

$bob->name = "Bob";

echo $tom->name;    // Tom

?>

При применении оператора clone все свойства, которые представляют примитивные типы и массивы, копируются в новый объект. Однако, что если у нас свойство класса представляет другой класс:

<?php

class Company{

    public $name;

    function \_\_construct($name){ $this->name = $name; }

}

class Person{

    public $name, $company;

    function \_\_construct($name, $company)

    {

        $this->name = $name;

        $this->company = $company;

    }

}

$microsoft = new Company("Microsoft");

$tom = new Person("Tom", $microsoft);

$bob = clone $tom;      // копируем объект из $tom в переменную $bob

$bob->name = "Bob";

$bob->company->name = "Google"; // изменяем у Боба название компании

$bob->languages[0] = "french";

echo $tom->company->name;   // Google - у Тома тоже изменилась компания

?>

Здесь в принципе мы сталкиваемся с той же ситуацией, что и в первом примере. Класс Person содержит свойство, которое представляет класс Company. При клонировании объекта:

<?php

$bob = clone $tom;

?>

Обе переменных $tom и $bob будут содержать ссылку на один и тот же объект Company. Соответственно если через одну переменную изменить свойства этого объекта:

<?php

$bob->company->name = "Google";

?>

то изменение затронет и другую переменную:

<?php

echo $tom->company->name; // Google

?>

Чтобы выйти из этой ситуации, необходимо в классе определить метод \_\_clone. Он вызывается при клонировании и может применяться для клонирования вложенных объектов:

<?php

class Company{

    public $name;

    function \_\_construct($name){ $this->name = $name; }

}

class Person{

    public $name, $company;

    function \_\_construct($name, $company)

    {

        $this->name = $name;

        $this->company = $company;

    }

    function \_\_clone()

    {

        $this->company = clone $this->company;

    }

}

$microsoft = new Company("Microsoft");

$tom = new Person("Tom", $microsoft);

$bob = clone $tom;      // копируем объект из $tom в переменную $bob

$bob->name = "Bob";

$bob->company->name = "Google";   // изменяем у Боба название компании

$bob->languages[0] = "french";

echo $tom->company->name; // Microsoft - у Тома НЕ изменилась компания

?>

Свойства и классы для чтения

Свойства для чтения

Иногда необходимы свойства, которые не должны менять своего значения. Начиная с версии 8.1 в PHP была добавлена возможность определять свойства для чтения. Подобные свойства предваряются ключевым словом readonly. Это позволяет гарантировать, что значение свойства не изменится. Таким свойствам можно передать значение только один раз внутри класса, в котором они определены (обычно это делается в конструкторе класса).

Рассмотрим небольшой пример:

<?php

class Person

{

    public readonly string $name;

    public $age;

    public function \_\_construct($name, $age)

    {

        $this->name = $name;

        $this->age = $age;

    }

}

$tom = new Person("Tom", 38);

$tom->age = 22;             // значение свойства $age можно поменять

// $tom->name = "Bob";     // значение свойства $name нельзя поменять, так как оно только для чтения

echo "Name: $tom->name";  // получить значение свойства $name можно

?>

Здесь определен класс Person, в котором есть два свойства. При этом свойство $name определено как свойство для чтения с модификатором readonly

<?php

public readonly string $name;

?>

Кроме того, если свойство определяется как свойство только для чтения, то для него необходимо явным образом указать тип данных. Так, в данном случае свойство name представляет тип string, то есть строку.

В конструкторе свойствам $name и $age присваиваются начальные значения. Однако после этого мы не сможем изменить значение свойства $name, даже внутри этого же класса. Мы можем только получить его значение. И если бы мы попробовали бы изменить его значение, например:

<?php

$tom->name = "Bob";

?>

то мы столкнулись бы с ошибкой:



Стоит отметить, что свойствам для чтения даже нельзя установить значение по умолчанию при определении, поскольку в этом случае оно было бы аналогично константе:

<?php

class Person

{

    public readonly string $name = "undefined"; // ! Ошибка

    public $age = 18;                           // Норм

?>

Конструктор - не единственное место, где можно установить свойство для чтения - это может быть любой метод класса. Например, можно сделать так:

<?php

class Person

{

    public readonly string $name;

    public $age;

    public function init(){

        $this->name = "Bob";

    }

}

$tom = new Person();

$tom -> init();         // инициализация свойства $name

echo "Name: $tom->name";  // Name: Bob

?>

Здесь установка значения производится в методе init(). Однако в этом случае мы можем столкнуться с проблемой, что к свойству $name будет обращение до вызова метода init(), и соответственно мы столкнемся с ошибкой. Поэтому конструктор является наиболее оптимальным местом для инициализации свойств для чтенияю.

Классы для чтения

Начиная с версии 8.2 PHP позволяет определять классы для чтения. Такой класс определяется с помощью ключевого слова readonly. Свойства таких классов по умолчанию являются свойствами, доступными только для чтения. Это гарантирует, что никакое из свойств объекта не сможет изменить значение. Например, следующий класс:

<?php

readonly class Person

{

    public string $name;

    public int $age;

    public function \_\_construct($name, $age)

    {

        $this->name = $name;

        $this ->age = $age;

    }

}

?>

будет эквивалентен следующему:

<?php

class Person

{

    public readonly string $name;

    public readonly int $age;

    public function \_\_construct($name, $age)

    {

        $this->name = $name;

        $this ->age = $age;

    }

}

?>

При этом для свойств класса для чтения также надо явным образом указывать тип данных. Кроме того, при наследовании производный класс также должен быть классом только для чтения.

В остальном классы для чтения используются также как и обычные классы:

<?php

readonly class Person

{

    public function \_\_construct(public string $name, public int $age)

    {}

}

$tom = new Person("Tom", 38);

// получить значения свойств можно

echo "Name: $tom->name  Age: $tom->age";  // Name: Tom Age: 38

// изменить значения свойств нельзя

// $tom->name = "Tomas";  // !ошибка

?>

Практика ООП

Что такое объектно-ориентированное программирование (ООП)?

Объектно-ориентированное программирование, обычно называемое ООП - это подход, который вам помогает разрабатывать сложные приложения таким образом, чтобы они легко поддерживались и масштабировались в течение длительного времени. В мире ООП реальные понятия Person, Car или Animal рассматриваются как объекты. В объектно-ориентированном программировании вы взаимодействуете с вашим приложением, используя объекты. Это отличается от процедурного программирования, когда вы, в первую очередь, взаимодействуете с функциями и глобальными переменными.

В ООП существует понятие «class», использываемое для моделирования или сопоставления реального понятия с шаблоном данных (свойств) и функциональных возможностей (методов). «Оbject» - это экземпляр класса, и вы можете создать несколько экземпляров одного и того же класса. Например, существует один класс Person, но многие объекты person могут быть экземплярами этого класса - dan, zainab, hector и т. д.

Например, для класса Person могут быть name, age и phoneNumber. Тогда у каждого объекта person для этих свойств будут свои значения.

Вы также можете определить в классе методы, которые позволяют вам манипулировать значениями свойств объекта и выполнять операции над объектами. В качестве примера вы можете определить метод save, сохраняющий информацию об объекте в базе данных.

Что такое класс PHP?

Класс - это шаблон, который представляет реальное понятие и определяет свойства и методы данного понятия. В этом разделе мы обсудим базовую анатомию типичного класса PHP.

Лучший способ понять новые концепции - показать это на примере. Итак, давайте рассмотрим в коде класс Employee, который представляет объект служащего.

<?php

class Employee

{

  private $first\_name;

  private $last\_name;

  private $age;

  public function \_\_construct($first\_name, $last\_name, $age)

  {

    $this->first\_name = $first\_name;

    $this->last\_name = $last\_name;

    $this->age = $age;

  }

  public function getFirstName()

  {

    return $this->first\_name;

  }

  public function getLastName()

  {

    return $this->last\_name;

  }

  public function getAge()

  {

    return $this->age;

  }

}

?>

Оператор class Employee в первой строке определяет класс Employee. Затем мы продолжаем объявлять свойства, конструктор и другие методы класса.

Свойства класса в PHP

Вы можете думать о свойствах класса как о переменных, которые используются для хранения информации об объекте. В приведенном выше примере мы определили три свойства - first\_name, last\_name и age. В большинстве случаев доступ к свойствам класса осуществляется через созданные объекты.

Эти private свойства могут быть доступны только внутри класса. Данный подход - самый безопасный уровень доступа к свойствам. Позже в уроке мы обсудим различные уровни доступа к свойствам и методам класса.

Конструкторы для классов PHP

Конструктор - это специальный метод класса, который вызывается автоматически при инстанцинации объекта. В следующих разделах мы увидим, как инстанцировать объекты, но сейчас вам нужно просто знать, что конструктор используется для инициализации свойств объекта при создании объекта.

Вы можете определить конструктор с помощью метода \_\_construct.

Методы для классов PHP

Давайте подумаем о методах класса как о функциях, которые выполняют определенные действия, связанные с объектами. В большинстве случаев они используются для доступа и управления свойствами объекта и выполнения связанных операций.

В приведенном выше примере мы определили метод getLastName, который возвращает фамилию, связанную с объектом.

В следующем разделе мы увидим, как создавать объекты класса Employee.

Что такое объект в PHP?

В предыдущем разделе мы обсудили базовую структуру PHP класса. Теперь, когда вы хотите использовать класс, вам нужно его инстанцировать, конечным результатом чего будет объект. Таким образом, мы можем думать о классе как о проекте, а объект - это реальная вещь, над которой вы можете работать.

В контексте класса Employee, созданного в предыдущем разделе, давайте посмотрим, как создать понятие объекта этого класса.

Полный пример:

<?php

class Employee

{

  private $first\_name;

  private $last\_name;

  private $age;

  public function \_\_construct($first\_name, $last\_name, $age)

  {

    $this->first\_name = $first\_name;

    $this->last\_name = $last\_name;

    $this->age = $age;

  }

  public function getFirstName()

  {

    return $this->first\_name;

  }

  public function getLastName()

  {

    return $this->last\_name;

  }

  public function getAge()

  {

    return $this->age;

  }

}

?>

<?php

$objEmployee = new Employee('Bob', 'Smith', 30);

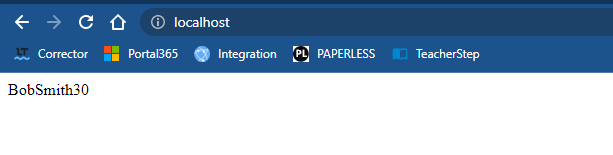
echo $objEmployee->getFirstName(); // print 'Bob'

echo $objEmployee->getLastName(); // prints 'Smith'

echo $objEmployee->getAge(); // prints '30'

?>

Result:



Если вы хотите создать понятие объекта любого класса вместе с его именем, нужно использовать ключевое слово new, и в итоге вы получите новое понятие объекта этого класса.

Если класс определил метод \_\_construct и ему требуются аргументы, вам нужно передать эти аргументы при создании экземпляра объекта. В нашем случае конструктор класса Employee требует три аргумента, и поэтому мы их передали, когда создавали объект $objEmployee. Как мы говорилось ранее, метод \_\_construct вызывается автоматически при инстанциации объекта.

Затем мы вызвали методы класса для объекта $objEmployee, чтобы запечатлить информацию, инициализированную во время создания объекта. Конечно же, вы можете создать несколько объектов одного класса, как это показано в следующем фрагменте.

<?php

class Employee

{

  private $first\_name;

  private $last\_name;

  private $age;

  public function \_\_construct($first\_name, $last\_name, $age)

  {

    $this->first\_name = $first\_name;

    $this->last\_name = $last\_name;

    $this->age = $age;

  }

  public function getFirstName()

  {

    return $this->first\_name;

  }

  public function getLastName()

  {

    return $this->last\_name;

  }

  public function getAge()

  {

    return $this->age;

  }

}

?>

<?php

// $objEmployee = new Employee('Bob', 'Smith', 30);

// echo $objEmployee->getFirstName(); // print 'Bob'

// echo $objEmployee->getLastName(); // prints 'Smith'

// echo $objEmployee->getAge(); // prints '30'

?>

<?php

$objEmployeeOne = new Employee('Bob', 'Smith', 30);

echo $objEmployeeOne->getFirstName(); // prints 'Bob'

echo $objEmployeeOne->getLastName(); // prints 'Smith'

echo $objEmployeeOne->getAge(); // prints '30'

$objEmployeeTwo = new Employee('John', 'Smith', 34);

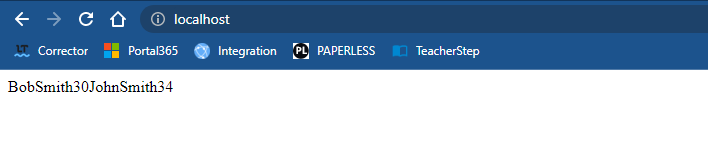
echo $objEmployeeTwo->getFirstName(); // prints 'John'

echo $objEmployeeTwo->getLastName(); // prints 'Smith'

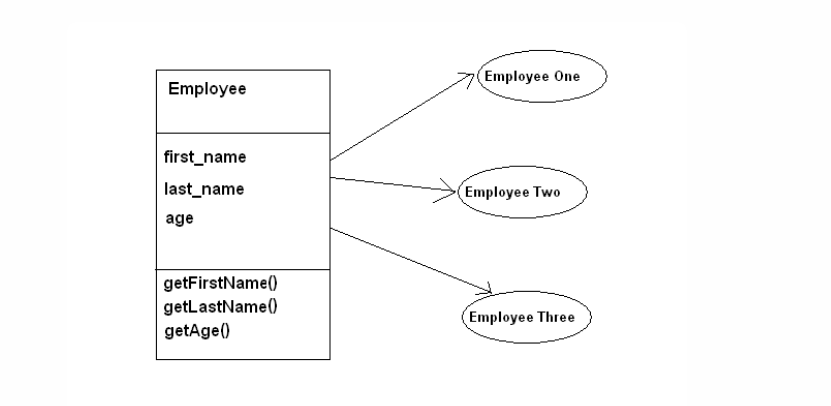
echo $objEmployeeTwo->getAge(); // prints '34'

?>

Result:



Следующее изображение является графическим представлением класса Employee и некоторых его экземпляров.



Проще говоря, класс - это проект, который вы можете использовать для создания структурированных объектов.

Инкапсуляция

В предыдущем разделе мы обсуждали, как создавать экземпляры объектов класса Employee. Интересно отметить, что сам объект $objEmployee объединяет свойства и методы класса. Другими словами, он скрывает эти детали от остальной части программы. В мире ООП это называется инкапсуляцией данных.

Инкапсуляция является важным аспектом ООП, позволяющий ограничить доступ к определенным свойствам или методам объекта. А это побуждает нас обсудить другую тему: уровень доступа.

Уровни доступа

Если вы определяете свойство или метод в классе, тогда вы можете объявить, что он имеет один из этих трех уровней доступа - public, private, или protected.

Доступ public

Когда вы объявляете свойство или метод как public, к нему можно получить доступ из любого места вне класса. Значение открытого свойства можно изменить из любого участка вашего кода.

Давайте рассмотрим на пример, чтобы понять как создать уровень публичного доступа.

<?php

class Person

{

  public $name;

  public function getName()

  {

    return $this->name;

  }

}

$person = new Person();

$person->name = 'Bob Smith';

echo $person->getName(); // prints 'Bob Smith'

?>

Как вы видете в приведенном выше примере, мы объявили общедоступное свойство name. Следовательно, вы можете установить его из любого места вне класса, что мы и сделали.

Доступ private

В случае если вы объявляете свойство или метод private, доступ к ним можно получить только из класса. Это означает, что вам нужно определить методы получения и установки, чтобы получить и установить значение этого свойства.

Опять же, давайте пересмотрим предыдущий пример, чтобы понять уровень частного доступа.

<?php

class Person

{

  private $name;

  public function getName()

  {

    return $this->name;

  }

  public function setName($name)

  {

    $this->name = $name;

  }

}

$person = new Person();

$person->name = 'Bob Smith'; // Throws an error

$person->setName('Bob Smith');

echo $person->getName(); // prints 'Bob Smith'

?>

Если вы захотите получить доступ к private свойству вне класса, он выдаст фатальную ошибку Cannot access private property Person::$name. Таким образом, вам нужно установить значение private свойства с помощью метода setter, как мы это cделали с помощью метода setName.

Могут возникать веские причины, из-за которых вы захотите установить private свойство. Например, возможно, для предпринятия какого-то действия (скажем, обновить базу данных или перерисовать шаблон), если это свойство меняется. В этом случае вы можете определить метод установки и управление любой специальной логикой для изменения свойства.

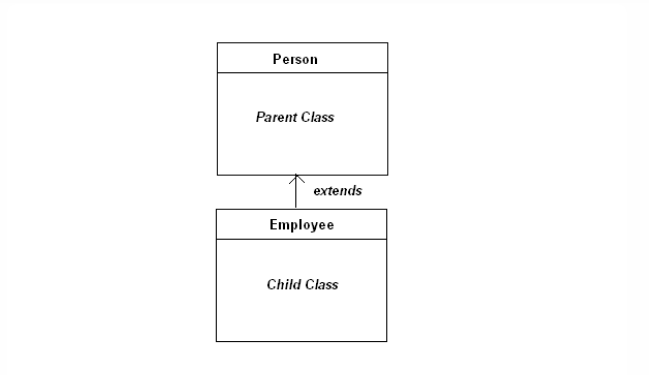
Доступ protected

Наконец, когда вы объявляете свойство или метод protected, к ним может обращаться тот же класс, который их определил, или классы, которые наследуют рассматриваемый класс. Мы обсудим наследование в следующем разделе, поэтому вернемся к уровню защищенного доступа чуть позже.

Наследование

Наследование является важным аспектом парадигмы объектно-ориентированного программирования, которая позволяет наследовать свойства и методы других классов, расширяя их. Класс, который наследуется, называется родительским классом, а класс, который наследует другой класс, называется дочерним классом. Когда вы создаете экземпляр объекта дочернего класса, он также наследует свойства и методы родительского класса.

Давайте посмотрим на следующий скриншот, чтобы понять концепцию наследования.



В примере выше класс Person является родительским классом, а класс Employee расширяет или наследует класс Person, поэтому и называется дочерним классом.

Давайте попробуем разобраться на реальном примере, чтобы понять, как это работает.

<?php

class Person

{

  protected $name;

  protected $age;

  public function getName()

  {

    return $this->name;

  }

  public function setName($name)

  {

    $this->name = $name;

  }

  private function callToPrivateNameAndAge()

  {

    return "{$this->name} is {$this->age} years old.";

  }

  protected function callToProtectedNameAndAge()

  {

    return "{$this->name} is {$this->age} years old.";

  }

}

class Employee extends Person

{

  private $designation;

  private $salary;

  public function getAge()

  {

    return $this->age;

  }

  public function setAge($age)

  {

    $this->age = $age;

  }

  public function getDesignation()

  {

    return $this->designation;

  }

  public function setDesignation($designation)

  {

    $this->designation = $designation;

  }

  public function getSalary()

  {

    return $this->salary;

  }

  public function setSalary($salary)

  {

    $this->salary = $salary;

  }

  public function getNameAndAge()

  {

    return $this->callToProtectedNameAndAge();

  }

}

$employee = new Employee();

$employee->setName('Bob Smith');

$employee->setAge(30);

$employee->setDesignation('Software Engineer');

$employee->setSalary('30K');

echo $employee->getName(); // prints 'Bob Smith'

echo $employee->getAge(); // prints '30'

echo $employee->getDesignation(); // prints 'Software Engineer'

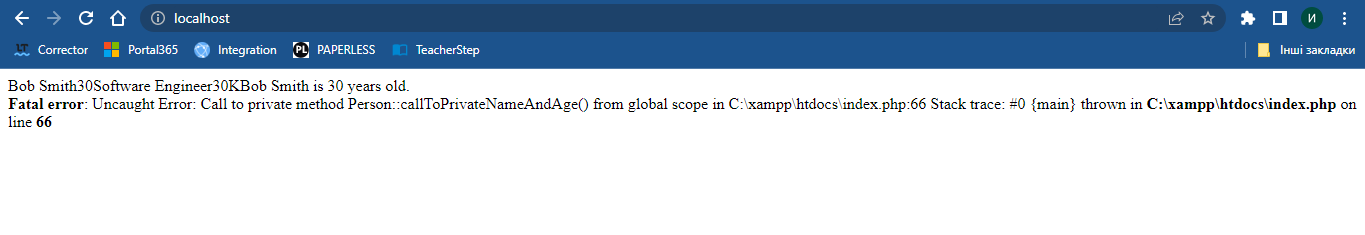
echo $employee->getSalary(); // prints '30K'

echo $employee->getNameAndAge(); // prints 'Bob Smith is 30 years old.'

echo $employee->callToPrivateNameAndAge(); // produces 'Fatal Error'

?>

Result:



Здесь важно отметить, что класс Employee использовал для наследования класса Person ключевое слово extends. Теперь класс Employee может получить доступ ко всем свойствам и методам класса Person, объявленные как public или protected. (Он не может получить доступ к ntv, которые объявлены как private.)

В примере выше объект $employee может получить доступ к методам getName и setName, которые определены в классе Person, поскольку они объявлены как public.

Затем мы обратились к методу callToProtectedNameAndAge, используя метод getNameAndAge, определенный в классе Employee, поскольку он объявлен как protected. Наконец, объект $employee не может получить доступ к методу callToPrivateNameAndAge класса Person, поскольку он объявлен как private.

С другой стороны, вы можете использовать объект $employee для установки свойства age класса Person, как мы это делали в методе setAge, который определен в классе Employee, поскольку свойство age объявлено как protected.

И так, это было краткое введение в наследование. Оно помогает сократить дублирование кода и, следовательно, способствует его повторному использованию.

Полиморфизм

Полиморфизм - еще одна важная концепция в мире объектно-ориентированного программирования, которая относится к способности по-разному обрабатывать объекты в зависимости от их типов данных.

Например, в контексте наследования, если дочерний класс хочет изменить поведение метода родительского класса, он может переопределить этот метод. Это называется переопределением метода. Давайте быстро рассмотрим реальный пример, чтобы понять концепцию переопределения метода.

<?php

class Message

{

  public function formatMessage($message)

  {

    return printf("<i>%s</i>", $message);

  }

}

class BoldMessage extends Message

{

  public function formatMessage($message)

  {

    return printf("<b>%s</b>", $message);

  }

}

$message = new Message();

$message->formatMessage('Hello World'); // prints '<i>Hello World</i>'

$message = new BoldMessage();

$message->formatMessage('Hello World'); // prints '<b>Hello World</b>'

?>

Как видите, мы изменили поведение метода formatMessage, переопределив его в классе BoldMessage. Важно то, что сообщение форматируется по-разному в зависимости от типа объекта, будь то экземпляр родительского или дочернего класса.

(Некоторые объектно-ориентированные языки также имеют своего рода перезагрузку методов, которая позволяет определять несколько методов класса с одним и тем же именем, но с разным количеством аргументов. Это не поддерживается напрямую в PHP, но существуют несколько обходных путей для достижения аналогичной функциональности.)

Полиморфизм в PHP

Язык PHP поддерживает полиморфизм в том смысле, что позволяет использовать вместо экземпляров родительского класса экземпляры подкласса. Ввод в действие требуемого метода осуществляется на этапе прогона. Поддержка перегрузки методов, при которой ввод метода в действие осуществляется с учетом сигнатуры метода, отсутствует. Дело в том, что в каждом классе может присутствовать только один метод с определенным именем. Но благодаря тому, что в языке PHP применяется слабая типизация и поддерживается переменное количество параметров, появляется возможность обойти это ограничение.

Пример полиморфизма в PHP

<?php

abstract class Publication {

    // определяем правило, что все публикации должны печататься, т.е. иметь метод do\_print()

    abstract public function do\_print();

   }

   class News extends Publication {

    // переопределяем абстрактный метод печати

    public function do\_print() {

     echo '<h4>Новость</h4>';

     //...

    }

   }

   class Announcement extends Publication {

    // переопределяем абстрактный метод печати

    public function do\_print() {

     echo '<h4>Объявление</h4>';

     //...

    }

   }

   class Article extends Publication {

    // переопределяем абстрактный метод печати

    public function do\_print() {

     echo '<h4>Статья</h4>';

     //...

    }

   }

   //Наполняем массив публикаций объектами, производными от Publication

   $publications[] = new News();

   $publications[] = new Announcement();

   $publications[] = new Article();

   foreach ($publications as $publication) {

    if ($publication instanceof Publication) { // Если мы работаем с наследниками Publication

     $publication->do\_print(); // то мы можем смело выводить данные на печать

    }

    else {

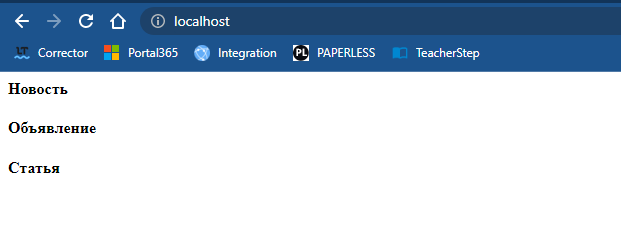
     //исключение или обработка ошибки

    }

   }

?>

Result:



<?php

   interface Machine {

      public function calcTask();

   }

   class Circle implements Machine {

      private $radius;

      public function \_\_construct($radius){

         $this -> radius = $radius;

      }

      public function calcTask(){

         return $this -> radius \* $this -> radius \* pi();

      }

   }

   class Rectangle implements Machine {

      private $width;

      private $height;

      public function \_\_construct($width, $height){

         $this -> width = $width;

         $this -> height = $height;

      }

      public function calcTask(){

         return $this -> width \* $this -> height;

      }

   }

   $mycirc = new Circle(3);

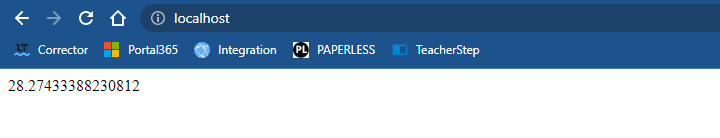
   $myrect = new Rectangle(3,4);

   echo $mycirc->calcTask();

   echo $myrect->calcTask();

?>

Result:



Практика:

Index.php

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta http-equiv="X-UA-Compatible" content="IE=edge">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>test oop</title>

</head>

<body>

    <?php

        include\_once('student.php');

        include\_once('group.php');

        $student = new Student("Alex", 25);

        echo "name: ".$student->name."<hr>";

        echo "age: ".$student->age."<hr>";

        echo "title: ".$student->getGroup()->title."<hr>";

        $client = new class {

            private $balance = 255.50;

            function getBalance() { return $this->balance; }

        };

        echo "client balance: ".$client->getBalance()."<hr>";

        //echo "weight: ".$student->weight."<hr>"; magic \_\_get doesn't work in Child

        echo "weight: ".$student->getWeight()."<hr>";

    ?>

</body>

</html>

Student.php

<?php

    include\_once('person.php');

    include\_once('group.php');

    class Student extends Person {

        private $group = null;

        private $weight = 72;

        function \_\_construct($name, $age) {

            Person::\_\_construct($name, $age); //call base constructor

            $this->group = new Group("PR3587");

            if($this->group instanceof Group) {

                echo "True<hr>";

            } else {

                echo "False<hr>";

            }

            echo "constructor student<br>";

        }

        function getGroup() {

            return $this->group;

        }

        function getWeight() {

            return $this->weight;

        }

    }

?>

Person.php

<?php

class Person {

    private $name = "";

    private $age = "";

    function \_\_construct($name, $age) {

        $this->name = $name;

        $this->age = $age;

        echo "constructor person<br>";

    }

    function \_\_destruct() {

        echo "<h1>destructor</h1>";

    }

    //getter and setters (variant old)

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

    function getAge() { return $this->age; }

    function setName($name) { $this->name = $name; }

    function setAge($age) { $this->age = $age; }

    //magic get and set (variant new)

    function \_\_get($property) {

        // $properties = ['age', 'name']; //permission

        // if(in\_array($property, $properties))

            return $this->$property;

        // return;

    }

    function \_\_set($property, $value) {

        // $properties = ['age']; //permission

        // if(in\_array($property, $properties))

        //     return $this->$property = $value;

        // return;

        $this->$property = $value;

    }

}

?>

Group.php

<?php

class Group {

    public $title = "";

    function \_\_construct($title) {

        $this->title = $title;

    }

}

?>

Example DataStorage and FileStorage:

<?php

interface Logger {

    public function logOperation(string $operation): void;

}

interface DataStorage {

    const STORAGE\_NAME = "Generic Data Storage";

    public function saveData(array $data): bool;

    public function getData(): array;

    public function deleteData(): void;

    public function getInfo(): string;

}

class Storage implements Logger {

    protected array $data = [];

    public function logOperation(string $operation): void {

        echo "Log: {$operation}";

    }

}

class DatabaseStorage extends Storage implements DataStorage {

    public function saveData(array $data): bool {

        parent::logOperation("Saved data");

        $this->data = $data;

        return true;

    }

    public function getData(): array {

        parent::logOperation("Get data");

        return $this->data;

    }

    public function deleteData(): void {

        parent::logOperation("Delete data");

        $this->data = [];

    }

    public function getInfo(): string {

        return "Database storage ".self::STORAGE\_NAME;

    }

}

class FileStorage extends Storage implements DataStorage {

    private string $filePath;

    public function \_\_construct(string $filePath) {

        $this->filePath = $filePath;

    }

    public function saveData(array $data): bool {

        parent::logOperation("Saved data to file");

        $encodedData = json\_encode($data);

        return file\_put\_contents($this->filePath, $encodedData);

    }

    public function getData(): array {

        parent::logOperation("Get data from file");

        $fileContent = file\_get\_contents($this->filePath);

        return json\_decode($fileContent, true) ? : [];

    }

    public function deleteData(): void {

        parent::logOperation("Delete data in file");

        unlink($this->filePath);

    }

    public function getInfo(): string {

        return "File storage ".self::STORAGE\_NAME;

    }

}

?>

Материалы к уроку:

1. <https://www.tutorialspoint.com/explain-polymorphism-in-php>
2. <https://www.c-sharpcorner.com/UploadFile/c63ec5/polymorphism-in-php/>
3. <https://code.tutsplus.com/tutorials/understanding-and-applying-polymorphism-in-php--net-14362>
4. <https://www.phptutorial.net/php-oop/php-polymorphism/>
5. <https://otus.ru/journal/obektno-orientirovannoe-programmirovanie-v-php/>
6. <https://habr.com/ru/articles/511744/>
7. <https://w3schoolsua.github.io/php/php_oop_inheritance.html#gsc.tab=0>

Домашнее задание:

1. Создайте class с именем student, содержащую поля: фамилия и инициалы, номер группы, успеваемость (массив из пяти элементов). Создать массив из десяти элементов такого типа, упорядочить записи по возрастанию среднего балла. Добавить возможность вывода фамилий и номеров групп студентов, имеющих оценки, равные только 4 или 5.

2. Создайте class с именем train, содержащую поля: название пункта назначения, номер поезда, время отправления. Ввести данные в массив из пяти элементов типа train, упорядочить элементы по номерам поездов. Добавить возможность вывода информации о поезде, номер которого введен пользователем. Добавить возможность сортировки массив по пункту назначения, причем поезда с одинаковыми пунктами назначения должны быть упорядочены по времени отправления.

3. Создать класс с двумя переменными. Добавить функцию вывода на экран и функцию изменения этих переменных. Добавить функцию, которая находит сумму значений этих переменных, и функцию которая находит наибольшее значение из этих двух переменных.

4. Описать класс, реализующий десятичный счетчик, который может увеличивать или уменьшать свое значение на единицу в заданном диапазоне. Предусмотреть инициализацию счетчика значениями по умолчанию и произвольными значениями. Счетчик имеет два метода: увеличения и уменьшения, — и свойство, позволяющее получить его текущее состояние. Написать программу, демонстрирующую все возможности класса.

5. Создать класс с двумя переменными. Добавить конструктор с входными параметрами. Добавить конструктор, инициализирующий члены класса по умолчанию. Добавить деструктор, выводящий на экран сообщение об удалении объекта.

6. Создать класс, содержащий динамический массив и количество элементов в нем. Добавить конструктор, который выделяет память под заданное количество элементов, и деструктор. Добавить методы, позволяющие заполнять массив случайными числами, переставлять в данном массиве элементы в случайном порядке, находить количество различных элементов в массиве, выводить массив на экран.

7. Составить описание класса для определения одномерных массивов строк фиксированной длины. Предусмотреть контроль выхода за пределы массива, возможность обращения к отдельным строкам массива по индексам, выполнения операций поэлементного сцепления двух массивов с образованием нового массива, слияния двух массивов с исключением повторяющихся элементов, а также вывод на экран элемента массива по заданному индексу и всего массива.

8. Составить описание класса многочленов от одной переменной, задаваемых сте¬пенью многочлена и массивом коэффициентов. Предусмотреть методы для вы¬числения значения многочлена для заданного аргумента, операции сложения, вычитания и умножения многочленов с получением нового объекта-многочлена, вывод на экран описания многочлена.

9. Построить три класса (базовый и 3 потомка), описывающих некоторых хищных животных (один из потомков), всеядных(второй потомок) и травоядных (третий потомок). Описать в базовом классе абстрактный метод для расчета количества и типа пищи, необходимого для пропитания животного в зоопарке.

a) Упорядочить всю последовательность животных по убыванию количества пищи. При совпадении значений – упорядочивать данные по алфавиту по имени. Вывести идентификатор животного, имя, тип и количество потребляемой пищи для всех элементов списка.

b) Вывести первые 5 имен животных из полученного в пункте а) списка.

c) Вывести последние 3 идентификатора животных из полученного в пункте а) списка.

d) Организовать запись и чтение коллекции в/из файл.

e) Организовать обработку некорректного формата входного файла.

10. Описать класс «домашняя библиотека». Предусмотреть возможность работы с произвольным числом книг, поиска книги по какому-либо признаку (например, по автору или по году издания), добавления книг в библиотеку, удаления книг из нее, сортировки книг по разным полям.

11. Составить описание класса прямоугольников со сторонами, параллельными осям координат. Предусмотреть возможность перемещения прямоугольников на плоскости, изменение размеров, построение наименьшего прямоугольника, содержащего два заданных прямоугольника, и прямоугольника, являющегося общей частью (пересечением) двух прямоугольников.

12. Создать класс для хранения комплексных чисел. Реализовать операции над комплексными числами: сложение, вычитание, умножение, деление, сопряжение, возведение в степень, извлечение корня. Предусмотреть возможность изменения формы записи комплексного числа: алгебраическая форма, тригонометрическая форма, экспоненциальная форма.

13. Составить описание класса для представления времени. Предусмотреть возможности установки времени и изменения его отдельных полей (час, минута, секунда) с проверкой допустимости вводимых значений. В случае недопустимых значений полей выбрасываются исключения. Создать методы изменения времени на заданное количество часов, минут и секунд.

14. Составить описание класса для вектора, заданного координатами его концов в трехмерном пространстве. Обеспечить операции сложения и вычитания векторов с получением нового вектора (суммы или разности), вычисления скалярного произведения двух векторов, длины вектора, косинуса угла между векторами.

15. Описать класс, представляющий треугольник. Предусмотреть методы для создания объектов, вычисления площади, периметра и точки пересечения медиан. Описать свойства для получения состояния объекта.

16. Создать абстрактный класс Figure с методами вычисления площади и периметра, а также методом, выводящим информацию о фигуре на экран. Создать производные классы: Rectangle (прямоугольник), Circle (круг), Triangle (треугольник) со своими методами вычисления площади и периметра.Создать массив n фигур и вывести полную информацию о фигурах на экран.

17. Класс Покупатель: Фамилия, Имя, Отчество, Адрес, Номер кредитной карточки, Номер банковского счета; Конструктор; Методы: установка значений атрибутов, получение значений атрибутов, вывод информации. Создать массив объектов данного класса. Вывести список покупателей в алфавитном порядке и список покупателей, у которых номер кредитной карточки находится в заданном диапазоне.

18. Класс Абонент: Идентификационный номер, Фамилия, Имя, Отчество, Адрес, Номер кредитной карточки, Дебет, Кредит, Время междугородных и городских переговоров; Конструктор; Методы: установка значений атрибутов, получение значений атрибутов, вывод информации. Создать массив объектов данного класса. Вывести сведения относительно абонентов, у которых время городских переговоров превышает заданное. Сведения относительно абонентов, которые пользовались междугородной связью. Список абонентов в алфавитном порядке.

19. Задача на взаимодействие между классами. Разработать систему «Автобаза». Диспетчер распределяет заявки на Рейсы между Водителями и назначает для этого Автомобиль. Водитель может сделать заявку на ремонт. Диспетчер может отстранить Водителя от работы. Водитель делает отметку о выполнении Рейса и состоянии Автомобиля.

20. Задача на взаимодействие между классами. Разработать систему «Железнодорожная касса». Пассажир делает заявку на станцию назначения, время и дату поездки. Система регистрирует Заявку и осуществляет поиск соответствующего Поезда. Пассажир делает выбор Поезда и получает Счет на оплату. Кассир вводит номера Поездов, промежуточные и конечные станции, цены.

21. Задача на взаимодействие между классами. Разработать систему «Интернет-магазин». Товаровед добавляет информацию о Товаре. Клиент делает и оплачивает Заказ на Товары. Товаровед регистрирует Продажу и может занести неплательщика в «черный список».

22. Задача на взаимодействие между классами. Разработать систему «Платежи». Клиент имеет Счет в банке и Банковскую карту (КК). Клиент может оплатить Заказ, сделать платеж на другой Счет, заблокировать КК и аннулировать Счет. Администратор может заблокировать КК за превышение платежа.

23. Задача на взаимодействие между классами. Разработать систему «Вступительные экзамены». Абитуриент регистрируется на Факультет, сдает Экзамены. Преподаватель выставляет Оценку. Система подсчитывает средний бал и определяет Абитуриента, зачисленного в учебное заведение.

24. Разработать класс «Калькулятор логарифмов»с возможностью сложения, вычитания, умножения, деления, возведения в степень и перехода к другому основанию. Программа должна выполнять ввод данных, проверку правильности введенных данных, выдачу сообщений в случае ошибок.

25. Аквариум

1. Определите объект TFish - аквариумная рыбка. Рыбка имеет координаты, скорость, размер, цвет, направление движения. Методами объекта являются:

Init - устанавливает значения полей объекта и рисует рыбу на экране методом Draw. Draw - рисует рыбу (виртуальный метод).

Look - проверяет несколько точек на линии движения рыбы. Если хоть одна из них отличается по цвету от воды, возвращается её цвет и расстояние до рыбы.

Run - перемещает рыбу в текущем направлении на расстояние, зависящее от текущей скорости рыбы. Иногда случайным образом меняет направление движения рыбы. Если рыба видит препятствие, направление движения меняется, пока препятствие не исчезнет из поля зрения рыбы.

2. Определите объект Taquarium, который является местом обитания рыб. Он представляет собой область экрана, наполненную водой.

Методы:

Init - включает графический режим, заполняет аквариум водой, скалами и рыбами. Run - организует бесконечный цикл, в котором выполняется метод Run всех обитателей аквариума.

Done - выключает графический режим.

3. Определите два объекта Tpike и Tkarp, которые наследуют объект Tfish.

4. Карпы и щуки должны быть объединены в стаи. Стая - это связанный список рыб в динамической памяти. Для связи в объектах Tpike и Tkarp используйте поле Next - указатель на следующую рыбу в стае. Аквариум должен быть владельцем двух стай. Пользователь может пополнять стаи, вводя рыб с клавиатуры.

5. Щуки должны поедать карпов, как только они их увидят. Необходимо установить, какого именно карпа видит щука (ближайший по координатам карп к щуке, найденный карп удаляется из стаи).

26. Разработка приложения для предметной области «Учёт товаров в магазине»

Разработать приложение, позволяющее собирать и накапливать сведения о поступлении и реализации товаров некоторого магазина. Структура приложения обязательно должна включать следующие классы: товар, производитель, документ, поступление товара, реализация товара и др.

27. Разработка приложения для предметной области «Организация учебного процесса в ВУЗе»

Разработать приложение, позволяющее собирать и накапливать сведения об организации и диспетчеризации учебного процесса в ВУЗе. Структура приложения обязательно должна включать следующие классы: академическая группа, специальность, дисциплина, аудитория, преподаватель и др.

28. Транспортное агентство

Разработайте программу, имитирующую работу трансагентства. Трансагентство имеет сеть филиалов в нескольких городах. Транспортировка грузов осуществляется между этими городами тремя видами транспорта: автомобильным, железнодорожным и воздушным. Любой вид транспортировки имеет стоимость единицы веса на единицу пути и скорость доставки. Воздушный транспорт можно использовать только между крупными городами, этот вид самый скоростной и самый дорогой. Кроме того, воздушный транспорт зависит от погоды. Доставить груз воздушным путем можно только при условии хорошей погоды одновременно в городах отправки и назначения. Хорошая или плохая погода задается случайным образом. Железнодорожный транспорт можно использовать между крупными и средними городами, этот вид самый дешевый. Автомобильный транспорт можно использовать между любыми городами. Заказчики через случайные промежутки времени обращаются в один из филиалов трансагентства с заказом на перевозку определенной массы груза и возможным пожеланием о скорости/цене доставки. Трансагентство организует отправку грузов одним из видов транспорта с учетом пожеланий клиента. Оплату трансагенство получает только после успешной доставки груза. Между некоторыми городами для железнодорожного и/или автомобильного транспорта имеются скоростные магистрали, на которых скорость соответствующего вида транспорта увеличивается с заданным коэффициентом. При перевозке грузов могут происходить аварии, при этом вероятность аварии на автотранспорте больше, чем на железнодорожном транспорте, а авиатранспорт имеет аварийность очень низкую. На скоростных магистралях вероятность аварии меньше, чем на обычных дорогах. При аварии трансагентство возвращает заказчику двойную стоимость перевозки.

Процесс имитации может быть остановлен пользователем программы для просмотра параметров объектов:

 Доход трансагенства, в том числе с разбивкой по видам транспорта и городам.

 Среднее время доставки груза, в том числе с разбивкой по видам транспорта и городам.

 Потери, связанные с плохой погодой.

 Потери, связанные с аварийностью, в том числе с разбивкой по видам транспорта и по видам дорог.

 Доход на тонно-километр скоростных магистралей в сравнении с таким же доход на обычных дорогах.

 Список исполняемых заказов с возможность сортировки по городам, видам транспорта, стоимости перевозки.

 Список задерживаемых заказов в связи с плохой погодой.

(Разработать и реализовать классы, которые являются основными в задании. Для сдачи лабораторной работы необходимо создать проект, в котором демонстрируется работа объектов созданных классов. Должны быть продемонстрированы выполнение конструктора, всех доступных методов и деструктора.)

31. Создать класс машина, имеющий марку (строка) , число цилиндров, мощность. Определить конструкторы, деструктор и функцию печати. Создать public- производный класс – грузовики, имеющий грузоподъемность кузова. Определить конструкторы по умолчанию и с разным числом параметров, деструкторы, функцию печати. Определить функции переназначения марки и грузоподъемности.

