PHP. ООП

Основы ООП

- **Инкапсуляция.** Инкапсуляция - это свойство системы, позволяющее объединить данные и методы, работающие с ними, в классе и скрыть детали реализации от пользователя.

  Инкапсуляцию можно сравнить с работой автомобиля с точки зрения типичного водителя. Многие водители не разбираются в подробностях внутреннего устройства машины, но при этом управляют ею именно так, как было задумано. Пусть они не знают, как устроен двигатель, тормоз или рулевое управление, — существует специальный интерфейс, который автоматизирует и упрощает эти сложные операции. Сказанное также относится к инкапсуляции и ООП — многие подробности "внутреннего устройства" скрываются от пользователя, что позволяет ему сосредоточиться на решении конкретных задач. В ООП эта возможность обеспечивается классами, объектами и различными средствами выражения иерархических связей между ними.

- **Полиморфизм.** Полиморфизм позволяет использовать одни и те же имена для похожих, но технически разных задач. Главным в полиморфизме является то, что он позволяет манипулировать объектами путем создания стандартных интерфейсов для схожих действий. Полиморфизм значительно облегчает написание сложных программ.
- **Наследование.** Наследование позволяет одному объекту приобретать свойства другого объекта, не путайте с копированием объектов. При копировании создается точная копия объекта, а при наследовании точная копия дополняется уникальными свойствами, которые характерны только для производного объекта.

Классы и объекты в PHP

Класс - это базовое понятие в объектно-ориентированном программировании (ООП). Экземпляр класса - это **объект**. Свойства и методы называются членами класса. Вообще, объектом является все то, что поддерживает инкапсуляцию. Если класс можно рассматривать как тип данных, то объект — как переменную (по аналогии). По общепринятым правилам имена классов ООП начинаются с прописной буквы.

```
class Имя_класса {
// описание членов класса - свойств и методов для их обработки
}
Объект = new Имя_класса;
```

Пример:

```
class Coor {
// данные (свойства):
public $name;
// методы:
 function get_name() {
 echo "<h3>John</h3>";
 }
}
// Создаем объект класса Coor:
$object = new Coor;
// Получаем доступ к членам класса:
$object->name = "Alex";
echo $object->name; // Выводит 'Alex'
// А теперь получим доступ к методу класса (фактически, к функции внутри класса):
$object-> get_name (); // Выводит 'John' заглавными буквами
```

Чтобы получить доступ к членам класса внутри класса, необходимо использовать указатель $this, который всегда относится к текущему объекту. Модифицированный метод get_name():

```
function get_name () {
echo $this->name;
}
```

Таким же образом, можно написать метод set_name():

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

Ключевое слово **instanceof** используется, чтобы проверить, принадлежит ли объект определенному классу:

```
$apple = new Fruit();
```

```
var_dump($apple instanceof Fruit);
```

Конструкторы и деструкторы

Конструктор - позволяет инициализировать свойства объекта при создании объекта. Если вы создадите функцию __construct(), то PHP автоматически вызовет эту функцию при создании объекта из класса. Пример конструктора:

```
class Fruit {
  public $name;

  function __construct($name) {
    $this->name = $name;
  }
  function get_name() {
    return $this->name;
  }
}
$apple = new Fruit("Apple");
echo $apple->get_name();
```

Деструктор вызывается, когда объект разрушен или скрипт остановлен или завершен. Если вы создадите функцию __destruct(), то PHP автоматически вызовет эту функцию в конце скрипта.

```
class Fruit {
  public $name;

  function __construct($name) {
    $this->name = $name;
  }
  function __destruct() {
    echo "The fruit is {$this->name}.";
  }
}

$apple = new Fruit("Apple");
```

Модификаторы доступа

Свойства и методы могут иметь **модификаторы** доступа, которые контролируют, где они могут быть доступны. Есть три модификатора доступа:

- public - свойство или метод могут быть доступны из любого места. Это по умолчанию;
- protected - свойство или метод могут быть доступны внутри класса и с помощью классов, производных от этого класса;
- private - свойство или метод могут быть доступны ТОЛЬКО внутри класса.

Наследование

Дочерний класс будет наследовать все **открытые** и **защищенные (protected**) свойства и методы от родительского класса. Кроме того, он может иметь свои свойства и методы. Унаследованный класс определяется с помощью ключевого слова **extends**.

```
class Fruit {
  public $name;
  public $color;
  public function __construct($name, $color) {
    $this->name = $name;
    $this->color = $color;
  }
  public function intro() {
    echo "The fruit is {$this->name} and the color is {$this->color}.";
  }
}

// Strawberry наследуется от Fruit
class Strawberry extends Fruit {
  public function message() {
    echo "Am I a fruit or a berry? ";
  }
}
```

```
$strawberry = new Strawberry("Strawberry", "red");
$strawberry->message();
$strawberry->intro();
```

Унаследованные методы могут быть переопределены путем переопределения методов (с тем же именем) в дочернем классе.

```
class Fruit {
  public $name;
  public $color;
  public function __construct($name, $color) {
    $this->name = $name;
    $this->color = $color;
  }
  public function intro() {
    echo "The fruit is {$this->name} and the color is {$this->color}.";
  }
}

class Strawberry extends Fruit {
  public $weight;
  public function __construct($name, $color, $weight) {
    $this->name = $name;
    $this->color = $color;
    $this->weight = $weight;
  }
  public function intro() {
    echo "The fruit is {$this->name}, the color is {$this->color}, and the weight is {$this->weight} gram.";
  }
}

$strawberry = new Strawberry("Strawberry", "red", 50);
$strawberry->intro(); // The fruit is Strawberry, the color is red, and the weight is 50 gram.
```

Ключевое слово **final** может использоваться для предотвращения наследования классов или для предотвращения переопределения метода. В следующем примере показано, как предотвратить наследование классов:

```
final class Fruit {
  // какой-то код
}

// приведет к ошибке
class Strawberry extends Fruit {
  // какой-то код
}
```

Константы

Константы нельзя изменить после объявления. Класс константы может быть полезен, если вам нужно определить некоторые постоянные данные внутри класса.Класс константы объявляется внутри класса с помощью ключевого слова const. Классы констант чувствительны к регистру. Тем не менее, рекомендуется называть константы **большими буквами** (в верхнем регистре).Мы можем получить доступ к константе **извне класса**, используя имя класса, за которым следует оператор разрешения области видимости (::), за которым следует имя константы, как здесь:

```
class Goodbye {
  const LEAVING_MESSAGE = "Спасибо за посещение W3Schools!";
}
echo Goodbye::LEAVING_MESSAGE;
```

Или мы можем получить доступ к константе **внутри класса**, используя ключевое слово self, за которым следует оператор разрешения области видимости (::) с последующим именем константы, как здесь:

```
class Goodbye {
  const LEAVING_MESSAGE = "Спасибо за посещение W3Schools!";
  public function byebye() {
    echo self::LEAVING_MESSAGE;
  }
}
```

Абстрактные классы и методы. Интерфейсы.

Абстрактные классы и методы - это когда родительский класс имеет именованный метод, но ему нужен дочерний класс (классы) для выполнения задач. Абстрактный класс - это класс, который содержит хотя бы один абстрактный метод. Абстрактный метод - это метод, который объявлен, но не реализован в коде.Абстрактный класс или метод определяется ключевым словом **abstract**. Синтаксис абстрактного класса (метода)

```
abstract class ParentClass {
  abstract public function someMethod1();
  abstract public function someMethod2($name, $color);
  abstract public function someMethod3() : string;
}
```

Когда дочерний класс **наследуется от абстрактного класса**, у нас есть следующие правила:

- Метод дочернего класса должен быть определен с тем же именем, и он повторно объявляет родительский абстрактный метод
- Метод дочернего класса должен быть определен с таким же или менее ограниченным модификатором доступа
- Количество обязательных аргументов должно быть одинаковым. Тем не менее, дочерний класс может иметь дополнительные аргументы в добавок

```
// Родительский класс
abstract class Car {
  public $name;
  public function __construct($name) {
    $this->name = $name;
  }
  abstract public function intro() : string;
}

// Дочерние классы
class Audi extends Car {
  public function intro() : string {
    return "Choose German quality! I'm an $this->name!";
  }
}

class Volvo extends Car {
  public function intro() : string {
    return "Proud to be Swedish! I'm a $this->name!";
  }
}

// Создать объекты из дочерних классов
$audi = new audi("Audi");
echo $audi->intro();
echo "<br>";

$volvo = new volvo("Volvo");
echo $volvo->intro();
echo "<br>";
```

Классы Audi, Volvo унаследованы от класса Car. Это означает, что классы Audi, Volvo могут использовать публичное свойство $name, а также публичный метод __construct() из класса Car из-за наследования. Но intro() - это абстрактный метод, который должен быть определен **во всех дочерних классах**, и они должны возвращать **строку**.
Интерфейсы позволяют указать, какие методы должен реализовывать класс.

Интерфейсы позволяют легко использовать различные классы одним и тем же способом. Когда один или несколько классов используют один и тот же интерфейс, это называется "полиморфизмом". Интерфейсы объявляются с помощью ключевого слова interface: Синтаксис

```
interface InterfaceName {
  public function someMethod1();
  public function someMethod2($name, $color);
  public function someMethod3() : string;
}
```

Интерфейс похож на абстрактные классы. Разница между интерфейсами и абстрактными классами заключается в следующем:

- Интерфейсы не могут иметь свойств, в то время как абстрактные классы могут

- Все методы интерфейса должны быть общедоступными, а методы абстрактного класса - общедоступными или защищенными
- Все методы в интерфейсе являются абстрактными, поэтому они не могут быть реализованы в коде, и ключевое слово abstract не требуется
- Классы могут реализовывать интерфейс, одновременно наследуя от другого класса

Чтобы реализовать интерфейс, класс должен использовать ключевое слово implements. Класс, реализующий интерфейс, должен реализовывать все методы интерфейса.Пример

```
interface Animal {
  public function makeSound();
}

class Cat implements Animal {
  public function makeSound() {
    echo "Meow";
  }
}

$animal = new Cat();
$animal->makeSound();
```

Трейты

PHP поддерживает только одно наследование: дочерний класс может наследовать только от одного родителя. А что, если класс должен унаследовать несколько типов поведения? ООП трейты решают эту проблему. **Трейты** используются для объявления методов, которые могут использоваться в нескольких классах. Трейты могут иметь методы и абстрактные методы, которые могут использоваться в нескольких классах, а методы могут иметь любой модификатор доступа (открытый, закрытый или защищенный). Трейты объявляются с помощью ключевого слова trait: Синтаксис

```
trait TraitName {
  // какой-то код...
}
```

Чтобы использовать трейт в классе, используйте ключевое слово use: Синтаксис

```
class MyClass {
  use TraitName;
}
```

Статические методы и свойства

Статические методы и свойства могут быть вызваны напрямую - без создания экземпляра класса. Статические методы и свойства объявляются с помощью ключевого слова static: Синтаксис

```
class ClassName {
  public static function staticMethod() {
    echo "Hello World!";
  }
}
```

Для доступа к статическому методу или свойству используйте имя класса, двойное двоеточие (::), и имя метода (свойства): Синтаксис

```
ClassName::staticMethod();
```

Класс может иметь как статические, так и нестатические методы (свойства). Доступ к статическому методу или свойству можно получить из метода или свойства того же класса, используя ключевое слово self и двойное двоеточие (::) Пример

```
class greeting {
  public static function welcome() {
    echo "Hello World!";
  }

  public function __construct() {
    self::welcome();
  }
}
new greeting();
```

Статические методы (или свойства) также можно вызывать из методов других классов. Для этого статический метод должен быть **public** (публичный). Чтобы вызвать статический метод из дочернего класса, используйте ключевое слово parent внутри дочернего класса. Здесь статический метод может быть **public** или **protected** (публичный или защищенный). Пример

```
class domain {
  protected static function getWebsiteName() {
    return "W3Schools.com";
  }
```

```
}

class domainW3 extends domain {
  public $websiteName;
  public function __construct() {
    $this->websiteName = parent::getWebsiteName();
  }
}

$domainW3 = new domainW3;
echo $domainW3 -> websiteName;
```

parent в PHP

parent в PHP — это ключевое слово, которое используется в дочернем классе для доступа к методу родительского класса.

```
class Dad // Dad по американски — отец
{
   public $name = 'Nic', $age = 45;
   public function printInfo()
   {
      echo "Имя: $this->name, возраст: $this->age, ";
   }
}

class Son extends Dad // Son по американски — сын
{
   public $user = 'Nicer', $password = 'secretNic';
   public function printInfo()  // метод можно назвать и printUserInfo(), как угодно
   {
      parent::printInfo();
      echo "логин: $this->user, пароль: $this->password.";
   }
}

$obj = new Son();
$obj->printInfo();
```

В этом примере мы переопределяли метод printInfo(). Но нам нужно в нём использовать функционал родительского метода, расширив его новыми возможностями. Поэтому мы подключили родительский метод конструкцией **parent::printInfo**(), а потом дописали расширяющий функционал.Кстати, обратите внимание, что вместе со словом parent используется оператор ::, который обычно используется для работы со статическими свойствами и методами. Но в случае переопределения метода при наследовании мы работаем не со статическими методами, и использование слова parent — это единственный случай, когда следует использовать **статическую ссылку на нестатический метод**.

Пространства имён

Пространства имён - это квалификаторы, которые решают две разные проблемы:

- Они позволяют улучшить организацию, группируя классы, которые работают вместе для выполнения задачи
- Они позволяют использовать одно и то же имя для нескольких классов

Например, у вас может быть набор классов, описывающих HTML таблицу, например Table, Row и Cell, а также другой набор классов для описания мебели, например Table, Chair и Bed. Пространства имен можно использовать для организации классов в две разные группы, а также для предотвращения смешивания двух классов Table и Table.

Пространства имен объявляются в начале файла с помощью ключевого слова namespace: Синтаксис
Объявите пространство имен под названием Html:

```
namespace Html;
```

Примечание: Объявление namespace должно быть **первым делом** в файле PHP. Константы, классы и функции, объявленные в этом файле, будут принадлежать пространству имен Html.
Для дальнейшей организации можно иметь **вложенные** пространства имен: Синтаксис
Объявите пространство имен Html внутри пространства имен Code:

```
namespace Code\Html;
```

Любой код, следующий за объявлением namespace, работает внутри пространства имен, поэтому классы, принадлежащие к пространству имен, могут быть созданы без каких-либо квалификаторов. Чтобы получить доступ к классам из-за пределов пространства имен, к классу необходимо присоединить пространство имен. Пример: Используйте классы из пространства имен Html:

```
$table = new Html\Table()
$row = new Html\Row();
```

Может быть полезно присвоить пространству имен или классу **псевдоним**, чтобы упростить запись. Это делается с помощью ключевого слова use: Пример: Дайте пространству имен псевдоним:

```
use Html as H;
$table = new H\Table();
```