Курс:

«Создание web–приложений, исполняемых на стороне сервера при помощи языка программирования PHP, СУБД MySQL и технологии Ajax»

Модуль 02

ТЕМА: РАБОТА С ООП

План занятия

- Что такое ООП?
- Классы
- Парадигмы

Что такое ООП?

Объектно-ориентированное программирование (ООП) — это подход, помогающий разрабатывать более сложные приложения, сохраняя при этом простоту ориентации в коде для продолжения разработки. Людям проще воспринимать окружающий мир в виде с помощью взаимодействий его частей друг с другой и разделять их на виды или классы.

Без использования ООП применялся *процедурный* подход: все программы представляли собой набор функций, которые выполняли отдельные участки кода. Такой способ разработки подходит для простых и небольших программ или проектов. Но чем более размер участка внутри функции и количество таких функций,

тем сложнее их редактировать и поддерживать между ними логическую связь. Из-за этого разные действия лишаются понятной группировки и выглядят беспорядочно.

Класс

В ООП существует понятие **класса**, используемое для представления понятий из реального мира с описанием шаблоном данных (*свойств*) и функциональных возможностей или **методов** — функций и процедур, которые принадлежат классу.

С технической точки зрения, **объект** — это экземпляр класса, и вы можете создать несколько экземпляров одного и того же класса, а сам класс — это шаблон.

При создании класса важно учитывать следующие правила *стандартов разработки на PHP*:

- Классам следует давать описательные имена. По возможности избегайте использования сокращений.
- Каждое слово в имени класса следует писать с заглавной буквы, без использования символа «_».
- Имя класса не должно быть зарезервированным словом интерпретатора PHP.
- Каждый класс следует хранить в отдельном файле, который должен называться, как класс.

Синтаксис создания класса не отличается от других языков программирования и выглядит так:

```
class User {
    function methodExample() {
        // method code
    }
}
```

Для создания экземпляра класса используется переменная, которой присваивается название класса с использованием конструкции `new`:

```
$user = new User;
```

После все методы и свойства класса могут быть доступны экземпляру через **оператор объекта** — символы –>:

```
$user->methodExample();
```

Также с помощью этого оператора и конструкции `$this` (ссылка на сам объект) внутри метода создаются переменные класса:

```
function methodExample() {
    $this->varName = "value";
}
```

Примечание: конструкция `$this` может быть вызвана **только внутри метода**, её использование в теле класса является недопустимым. Однако, объявление переменной в теле класса возможно, но только с использованием *модификатора доступа.*

Конструктор класса

Конструктор — это необязательный метод, который вызывается при создании экземпляра класса; при его создании PHP ищет метод `__construct()` и автоматически вызывает его при наличии[1]. Также конструктор может принимать аргументы, что значительно упрощает работу с классами.

```
class User {
    function __construct(String $string) {
        echo $string;
    }
}
```

Такой код при создании экземпляра класса `User` будет требовать передачи строки и выводить её на страницу:

```
$user = new User("Hello, World!");
```

Следует учесть, что аргументы, принимаемые конструкторов, необходимо указывать при каждом вызове класса.

Примечание: если у класса нет конструктора, или его конструктор не имеет обязательных параметров, скобки после имени класса можно не писать.

[1] До PHP 8.0.0 можно было создать метод, имя которого могло совпадать с именем класса вместо конструктора, однако сейчас такой синтаксис считается устаревшим и игнорируется.

Принципы

Разработка с помощью объектно-ориентированного подхода строится на четырёх основных **принципах**:

- Абстракция
- Инкапсуляция
- Полиморфизм
- Наследование

В ходе первой части введения в ООП-разработку будут рассмотрены первые две парадигмы.

Абстракция

Абстракция подразумевает выделение общих характеристик объектов без лишней информации.

Главная особенность абстракции заключается в отсутствии излишней детализации. Например, при использовании кофемашины используется вода, зёрна и выбирается тип кофе.

```
class CoffeeMachine {
    function PourWater() {
        // Заливка воды
    }

    function AddBeans() {
        // Засыпка зёрен
    }

    function BrewCoffee() {
        // Заварка кофе. При этом,
        // содержание метода для разных
        // кофемашин будет отличаться
    }
}
```

Дальнейшие детали процесса заваривания скрыты от пользователя, и могут отличаться в зависимости от кофемашины.

Инкапсуляция

Инкапсуляция позволяет вводить ограничения на доступ к участкам кода. Например, возможно сделать метод, который будет доступен только для внутреннего использования в классе.

Для определения доступности существует понятия модификаторов доступа, которые и определяют наличие ограничений. Их существует три вида:

- `public` — доступ к свойствам и методам из любого места.
- `protected` — доступ к *родительскому* и *наследуемому* классу (область класса *наследника*).
- `private` — доступ только из класса, в котором объявлен сам элемент (область самого класса).

Модификатор по умолчанию — `public`. У свойств значения модификатора по умолчанию нет, поэтому он может быть задан при объявлении переменных в теле класса:

```
class User {
    public $id = null;

    public function __construct(Int $id) {
        $this->id = $id;
    }
}
```

Наследование

Наследование позволяет создать класс на основе существующего, используя уже готовые методы или даже конструктор. Так один класс (*parent*) может лежать в основе другого класса (*child*).

Для создания класса-наследника используется ключевое слово `extends`, указывая, что новый класс будет являться расширением основного.

При этом наследник может переопределять родительские методы и свойства, а также реализовывать собственные. Использование наследования имеет ряд преимуществ для программного кода:

- Повторное использование кода — общий функционал реализуется в родителе, а все *подклассы* наследуют его.
- Код становится проще — использование наследования позволяет разбивать большие и сложные классы на более мелкие и управляемые.

Для лучшего понимания наследования можно представить класс `Animal`. Например, у каждого животного может быть кличка и возраст, которые указываются при создании экземпляра класса:

```
class Animal {
    public string $name;
    public int $age;

    public function __construct(string $name,
    int $age) {
        $this->name = $name;
        $this->age = $age;
    }
}
```

Такой класс позволяет представить *абстрактно* любое животное. Однако, некоторые виды животных имеют отличия друг от друга и требуют дополнительной реализации тех или иных возможностей внутри класса.

Например, у кошек можно дополнительно определить породу, унаследовав остальные свойства животных. Благодаря этому не придётся заново определять свойства клички и возраста, но потребует частично переопределить конструктор.

Для переопределения конструктора понадобится создать точно такой же метод внутри класса с новым набором аргументов.

При этом, чтобы не описывать заново всю работу конструктора родителя, его можно вызвать с помощью специального названия `parent` и *оператора разрешения области видимости* (`::`) для обращения к элементам извне класса и передать конструктору родителя важные для него аргументы:

```
class Cat extends Animal {
    public string $breed;

    public function __construct(string $name,
    int $age, string $breed) {
        parent::__construct($name, $age);
        $this->breed = $breed;
    }
}
```

Теперь можно создать абстрактное животное по кличке Кристи возраста двух лет и сиамскую кошку по кличке Бонни возраста четырёх лет:

```
$Christy = new Animal("Кристи", 2);
$Bonnie = new Cat("Бонни", 4, "Сиамская");
```

Принцип наследования можно упростить до следующего примера: «каждая кошка — животное, но не всякое животное — кошка».

Примечание: классы-наследники получают доступ не только к `public`- свойствам и методам, но и к `protected`. Однако, доступ к `private`- свойствам и методам ограничивается **только родительским классом**.

Полиморфизм

Главная особенность полиморфизма — это возможность разным объектам выполнять одни и те же действия. При этом, различия объектов и их устройство значения не имеют.

В PHP полиморфизм реализован использованием одинаковых названий методов в классах-наследниках. Например, в наследниках класса `Animal` может быть реализован метод `feed()`, который позволит накормить зверят.

```
class Cat extends Animal {
    // …
    public function feed() {
        // feed details
    }
}

class Dog extends Animal {
    // …
    public function feed() {
        // feed details
    }
}
```

Для того, чтобы вызвать метод для наследников класса, можно собрать животных в массив и выполнить такой метод для каждого элемента массива:

```
$animals[] = new Cat("Кристи", 2, "Сфинкс");
$animals[] = new Cat("Бонни", 4, "Сиамская");
$animals[] = new Dog("Адам", 6, "Доберман");
$animals[] = new Dog("Патрик", 5, "Чау-чау");

foreach ($animals as $animal) {
    // Если экземпляр — наследник класса Animal:
    if ($animal instanceof Animal) {
        $animal->feed();
    }
}
```

Примечание: важно, чтобы использованный метод был реализован в родительском классе, иначе будет ошибка, так как наследники не будут соответствовать родителю.

Абстрактные классы и интерфейсы

Абстрактный класс

Абстрактный класс — средство разработки классов для повторного использования кода, является «заготовкой» или образцом для класса.

Для создания абстрактного класса используется ключевое слово `abstract`, которое указывается перед его объявлением.

Важное отличие абстрактного класса заключается в том, что на его основе невозможно создать объект — только от класса наследника.

Также внутри абстрактного класса возможно создать абстрактные методы, которые должны быть реализованы в наследниках. При этом абстрактные методы не могут иметь модификатор доступа `private`, он обязательно должен быть или `public`, или `protected`. Однако создание абстрактных свойств в PHP *невозможно*.

Созданный ранее класс `Animal` должен быть абстрактным, чтобы создавать новых животных только на его основе, а не по подобию. Класс может содержать в себе абстрактный метод `getAge()` для получения возраста животного, но реализация получения возраста будет зависеть только от наследников.

```
abstract class Animal {
    abstract public function getAge();
}
```

Например, у кошки один год может быть равен 7 человеческим годам, поэтому при реализации метода в наследнике Cat будет написано следующее:

```
class Cat extends Animal {
    public function getAge() {
        return $this->age * 7;
    }
}
```

У собаки же один год может равняться 11 человеческих, поэтому реализация этого метода будет иметь другой вид:

```
class Dog extends Animal {
    public function getAge() {
        return $this->age * 11;
    }
}
```

Таким образом родительский класс указывает наследникам необходимость реализации метода, но оставляет за ними способ реализации.

Интерфейс

В отличие от абстрактного класса, интерфейс является лишь описанием того, что должен содержать в себе наследник. То есть, интерфейс может содержать названия методов и принимаемые ими аргументы (сигнатуру), но в интерфейсе не может быть реализация таких методов.

Интерфейс объявляется с помощью ключевого слова `interface` и в остальном практически не имеет отличий от описания абстрактных классов.

Наследники интерфейса должны быть созданы с помощью ключевого слова `implements` , а также возможно наследование от нескольких интерфейсов, но тогда необходимо будет реализовать все методы принимаемых родителей.

Также интерфейс не может содержать конструктора, а все методы обязаны быть публичными.

Интерфейс не может быть заменён абстрактным классом, так как интерфейс используется, когда важно передать наследнику информацию о требуемых методах, а не их реализацию.

Различия и сходства

Интерфейс	Абстрактный класс
Наследование происходит через ключевое слово `implements`	Наследование происходит через ключевое слово `extends`
На основе интерфейса не может быть создан объект	На основе абстрактного класса не может быть создан объект
Не содержит конструктор	Может содержать конструктор
Содержит только объявление методов (сигнатуры методов)	Может содержать как сигнатуры методов так и их реализации
Не может иметь модификаторов доступа — все методы по умолчанию публичные	Может иметь модификаторы доступа
Поддерживает множественное наследование	Не поддерживает множественное наследование