РАЗРАБОТКА WEB-ПРИЛОЖЕНИЙ НА PHP&MYSQL

УРОК 4

ООП в PHP

ВВЕДЕНИЕ

Эволюция языков программирования обогащала их все новыми методами описания различных предметных областей. Повышение сложности и объема кода неизбежно приводило к снижению читабельности и понимания взаимосвязей между данными, с которыми оперируют программы.

Изобретение **объектно-ориентированного программирования (ООП)** позволило решить эту проблему, заставив программистов действовать на совершенно другом уровне абстракции. Применение принципов **ООП** к написанию программ приводит к воссозданию в коде той организационной структуры, которая существует в данной предметной области и в реальном мире.

Сегодня уже можно говорить о том, что технология программирования, не реализующая **ООП**, не может быть конкурентоспособной.

Поддержка **ООП** в **PHP** началась с версии **4**, но его реализация была сильно ограничена. В версии **PHP 5** объектная модель была полностью пересмотрена для более точного соответствия академическому определению **ООП**. Несмотря на то, что **PHP** не является объектно-ориентированным языком изначально, т.е. таким как, например **Ruby** или даже **JavaScript**, в современном **PHP** без **ООП** уже не обойтись никак. Практически все серьёзные **web**-приложения работают исключительно с объектной моделью, которая предварительно создаётся средствами, реализованными в **PHP**.

Изучая технологии программирования, вы уже сталкивались с концепцией **ООП**, поэтому подробным изложением её теоретических основ мы заниматься не будем, а займёмся разбором специфики реализации **ООП** именно в языке **PHP**.

РЕАЛИЗАЦИЯ КОНЦЕПЦИЙ ООП В PHP

Инкапсуляция

Инкапсуляция – это абстрагирование (обеспечение независимости) от внутреннего устройства объектов, что позволяет многократно использовать классы в разных проектах, а также использовать классы, написанные другими программистами.

Концепция инкапсуляции является основной в идеологии **ООП** и реализована в **PHP** полностью еще с версии 4. В **PHP 5** инкапсуляцию слегка доработали, введя деструкторы и изменив работу с конструкторами.

Наследование

Наследование (**inheritance**) позволяет создавать классы объектов на основе других классов, расширяя и частично изменяя их функциональность и набор членов. Это позволяет повторно использовать код, так как изменения в классе, на основе которого были созданы другие классы, затронут всех его наследников.

Переопределение свойств и методов называют **перегрузкой**, а класс, на основе которого создаются классы-наследники – **базовым классом**.

Некоторые языки программирования предусматривают **множественное наследование**, т.е. возможность создания нового класса в результате наследования от нескольких базовых.

PHP поддерживает обычное наследование, но совершенно не поддерживает множественного.

Полиморфизм, абстрактные методы и классы.

Полиморфизм заключается в том, что реализуется единообразие работы с объектами различных классов, но схожими методами работы. Это осуществляется при помощи **абстрактных классов**. Это класс, у которого есть хотя бы один **абстрактный метод**, т.е. метод, не имеющий реализации в классе, в котором он объявлен.

СИНТАКСИС ОПИСАНИЯ КЛАССОВ.

Создание классов в PHP 4

В **PHP 4** базовый класс определялся так:

Пример 2.1.1

```
<?php
class myPHP4Class {
    var $my_variable;
    function my_method($param) {
          echo "Вызван метод my_method($param)!\n";
          echo "Значение внутренней переменной: ";
          echo "{$this->my_variable}\n";
          }
    } ?>
```

После того как класс будет определен, можно создать экземпляр (**instance**) класса. Чтобы создать экземпляр класса `myPHP4Class`, используется оператор `new`. А после того, как будет создан экземпляр класса, свойства и методы, определенные в исходном классе, будут доступны для экземпляра с помощью операции `->`.

Пример 2.1.2

```
<?php
$myinstance = new myPHP4Class();
$anotherinstance = new myPHP4class();
$myinstance->my_variable = 10;
$anotherinstance->my_variable = 20;
$myinstance->my_method("MyParam");
?>
    /* Будет выведено: Вызван метод my_method(MyParam)! Зна-
чение внутренней переменной: 10 */
```

Здесь переменные `$myinstance` и `$anotherinstance` представляют объекты, имеющие тип `myPHP4Class`. Хотя они и были созданы на основе одного и того же определения класса, они совершенно не зависят друг от друга.

Члены класса могут вызываться не только из внешней программы, но и из самого класса. Для того, чтобы обратиться к свойству или методу внутри класса, их вызов необходимо предварить

конструкцией `$this->`. Переменная `$this`, которая неявно присутствует в каждом классе, является ссылкой на текущий объект класса и сообщает интерпретатору, что происходит обращение к объекту данного класса, анне создание нового.

Иногда бывает необходимо обратиться к членам базового класса или к членам класса, экземпляров которого не существует. Для этого применяется оператор `::`.

Уничтожение объектов класса – экземпляров – осуществляется при помощи функции `unset()`.

Создание класса в PHP 5

В **PHP 5** порядок определения и способы использования классов не претерпели существенных изменений. В действительности, тот код, который приведен выше, будет работать в **PHP 5** так, как положено. Однако такой способ определения класса уже не применяется.

Пример 2.2.1

```
<?php
class myPHP5Class {
    public $my_variable;
    public function my_method($param) {
          echo "Вызван метод my_method($param)!\n";
          echo "Значение внутренней переменной: ";
          echo "{$this->my_variable}\n";
    }
}
?>
```

Отличие заключается в использовании новой важной особенности в модели объектного ориентирования **PHP 5** – контроле доступа.

Спецификаторы контроля доступа к членам класса

Если сторонний разработчик использовал класс `myPHP4Class`, можно было свободно изменять или считывать значение переменной `$my_variable`. С другой стороны, в **PHP 5** объектная модель предусматривает три уровня доступа к членам класса, ограничивающие

данные, которые могут извлекаться в сценариях. Это уровни public, private и protected; их можно применять и к методам, и к свойствам класса.

- public (общедоступные). Доступ может быть осуществлен из любого места в пределах сценария. С помощью объекта их можно вызывать или видоизменять либо изнутри самого объекта, либо за его пределами.
- private (закрытые). Доступ может быть осуществлен только из экземпляра этого класса с помощью переменной $this. Если вместо public $my_variable; в описании класса указать private $my_variable;, то при обращении к свойству $my_variable извне объекта возникнет ошибка PHP:

```
Fatal Error: Cannot access private property myPHP5Class::my_variable in …
Неисправимая ошибка: Невозможен доступ к закрытому свойству myPHP5Class::my_variable в …
```

- protected (защищенный). Этот уровень подобен уровню private, поскольку он запрещает внешний доступ к члену класса. Однако в отличие от уровня private, ограничивающего доступ только к тому классу, в котором он определен, уровень protected разрешает доступ как из него самого, так и из любых дочерних классов (дочерние классы образуются в результате наследования (см. **3.3**).

Указание типов

Начиная с версии **PHP 5**, появляется указание типов (**type hinting**). Поскольку методы внутри объектов могут принимать параметры, которые являются экземплярами других объектов, **PHP 5** позволяет ограничивать типы данных для параметров методов:

Пример 2.4.1

```
<?php
class Integer {
    private $number;
    public function getInt() {
        return (int)$this->number;
    }
    public function setInt($num) {
        $this->number = (int)$num;
    }
```

```
}
class Float {
    private $number;
    public function getFloat() {
        return (float)$this->number;
    }
    public function setFloat($num) {
        $this->number = (float)$num;
    }
}
?>
```

Определяются два класса, `Integer` и `Float`, которые реализуют простые оболочки для этих типов данных в **PHP**. А если бы нужно было реализовать класс для сложения всего двух чисел в формате с плавающей точкой?

Пример 2.4.2

```
<?php
class Math {
    public function add($op1, $op2) {
        return $op1->getFloat() + $op2->getFloat();
    }
}
?>
```

Однако мы не можем гарантировать, что параметры `$op1` и `$op2` будут являться экземплярами класса `Float`. Даже если бы мы точно знали, что они являются объектами, все равно у нас нет способа узнать, имеют ли они подходящий тип.

В прототипе метода нужно определить требуемый тип:

Пример 2.4.3

```
<?php
class Math {
    public function add(Float $op1, Float $op2) {
        return $op1->getFloat() + $op2->getFloat();
    }
}
?>
```

Конструкторы и деструкторы

Конструкторы и деструкторы представляют собой функции, вызываемые во время создания экземпляра объекта (**конструкторы**) и/или удаления (**деструкторы**). Их основное назначение заключается в инициализации объектов и их удалении и освобождении занимаемой ими памяти. В **PHP 4** были доступны только конструкторы; они создавались посредством определения функции, имя которой было точно таким же, как и имя самого класса:

Пример 2.5.1

```
<?php
class SimpleClass {
    function SimpleClass($param) {
        echo "Создан новый экземпляр SimpleClass!";
    }
}
$myinstance = new SimpleClass;
?>
```

В **PHP 5** эта идея была существенным образом доработана и улучшена. Теперь используется единая функция конструкторов с именем `__construct()` и функция для деструкторов `__destruct()`.

Пример 2.5.2

```
<?php
class SimpleClass {
    function __construct($param) {
        echo "Создан новый экземпляр SimpleClass!";
    }
    function __destruct() {
        echo "Разрушен данный экземпляр SimpleClass";
    }
}
$myinstance = new SimpleClass("value");
unset($myinstance);
?>
```

Обратите внимание, что названия методов `__construct()` и `__destruct()` начинаются с двух символов подчёркивания. Чуть позже мы столкнёмся с еще некоторыми специальными методами, которые оформляются в подобном ключе.

Конструкторы полезны для инициализации свойств класса. А комбинированное использование конструкторов и деструкторов точно так же полезно во всех других случаях. Одним из классических примеров является класс для доступа к серверной базе данных, где конструктор может отвечать за организацию соединения с базой данных, а деструктор – за его закрытие.

Конструктор так же применяется для помещения объектов других классов в качестве члена класса. Если класс содержит в качестве члена объект другого класса, то обращение к членам такого объекта также осуществляется при помощи `->`: `$obj->object->member`.

Число цепочек из `->` не ограниченно, т.е. можно создавать любое количество вложенных объектов.

Пример 2.5.3

```
<?php
class Inner // Внутренний класс Inner
{
    public $member = "Член member класса Inner";
    function method(){ // Объявляем метод
          echo "Метод method() класса Inner";
    }
}

class Outer // Внешний класс Outer
{
    function __construct() //Конструктор
    {
          $this->objt = new Inner;
    }
}
// Объявляем объект класса Outer
$obj = new Outer;
// Обратимся к члену member объекта $object класса Inner
echo "{$obj->objt->member}<br>";
// Обратимся к методу method() объекта $object класса Inner
echo "($obj->objt->method()}<br>";
```

Давайте разберем пример описания простейшего класса товара для интернет-магазина:

Пример 2.5.4

```
<?php
class commodity {
    public $name; //название товара
    public $category; //категория товара
    public $price; //цена
    public $availability; //наличие на складе

    function __construct($name, $category, $price=null,
$availability=false) { //конструктор, который
инициализирует все свойства класса
        echo "запущен конструктор…<br />";
        $this->name=$name;
        $this->category=$category;
        $this->price=$price;
        $this->availability =$availability;
    }

    function __destruct() { //деструктор
        echo "запущен деструктор…<br />";
    }

    function getPrice() { //метод, позволяющий получить ин-
формацию о цене.
        return(is_null($this->price)?'N/A':$this->price);
    }

    function setPrice($new_price) { //метод, изменяющий зна-
чение свойства с ценной на новое.
        $this->price=$new_price;
    }
}
?>
```

РЕАЛИЗАЦИЯ КЛАССОВ И РАБОТА С НИМИ.

Клонирование

В версии **PHP 4** представление объектов не осуществлялось по ссылке. То есть, при передаче объекта в вызов метода или функции

создавалась копия этого объекта. Мало того, что это было связано с трудностями, такой вариант мог порождать серьезные проблемы с отслеживанием ошибок. Поскольку создавалась копия объекта, любые изменения в экземпляре этого объекта, произведенные в методе или функции, влияли только на копию объекта в функции. В версии **PHP 5** такое нелогичное поведение было устранено. Теперь все объекты представляются по ссылке. Несмотря на серьезность изменения, прямые копии экземпляра объекта больше не создаются:

Пример 3.1.1

```
<?php
$class_one = new MyClass();
$class_one_copy = $class_one;
?>
```

В **PHP 4** `$class_one_copy` на самом деле является независимым экземпляром класса `MyClass`, имеющим все особенности экземпляра `$class_one`. В **PHP 5** `$class_one` и `$class_one_copy` представляют один и тот же объект – любые модификации, произведенные в одном из экземпляров, приведут к такому же изменению в другом.

В **PHP 5** вместо прямого присваивания экземпляру объекта новой переменной необходимо использовать оператор `clone`. Он возвращает новый экземпляр текущего объекта, копируя в него значения любого члена и, таким образом, создавая независимый клон.

Пример 3.1.2

```
<?php
$class_one = new MyClass();
$class_one_copy = clone $class_one;
?>
```

Класс также может реализовать специальный метод `__clone()`, позволяющий управлять элементами, которые будут копироваться из одного экземпляра в другой. В этом специальном методе переменная `$this` ссылается на новую копию объекта вместе со всеми значениями из исходного объекта.

Пример 3.1.3

```
<?php
class myObject {
    public $var_one = 10;
```

```
    public $var_two = 20;

    function __clone() {
        /* $var_two в клоне присваивается значение 0 */
        $this->var_two = 0;
    }
}
$inst_one = new myObject();
$inst_two = clone $inst_one;
var_dump($inst_one);
var_dump($inst_two);
?>
```

В данном примере вывод будет выглядеть следующим образом:

```
object(myObject)#1 (2) {
    ["var_one"]=> int(10)
    ["var_two"]=> int(20)
    }
object(myObject)#2 (2) {
    ["var_one"]=> int(10)
    ["var_two"]=> int(0)
}
```

Метод `__clone()` будет полезным во многих ситуациях, особенно если клонируемый объект содержит информацию, характерную только для данного экземпляра (например, уникальный идентификатор объекта). В таких случаях метод `__clone()` можно использовать для копирования только необходимой информации.

Константы классов

Константы классов, являющиеся нововведением **PHP 5**, позволяют определять постоянные значения в пределах класса. Для этого используется ключевое слово `const`, за которым следует имя константы и ее значение:

Пример 3.2.1

```
<?php
class ConstExample {
    private $myvar;
    public $readme;
    const MY_CONSTANT = 10;
```

```
    public function showConstant() {
        echo " Значение : ".MY_CONSTANT;
    }
}
$inst = new ConstExample;
$inst->showConstant();
echo " Значение : ".ConstExample::MY_CONSTANT;
?>
```

Константа `MY_CONSTANT` определяется в классе и имеет целочисленное значение `10`. Обратиться напрямую к этой константе можно из самого класса, как и в случае любой константы, созданной с помощью функции `define()`. А для того чтобы обратиться к константе вне класса, нужно использовать оператор `::`.

Использование констант класса позволяет избежать конфликтов в глобальном пространстве имен в достаточно крупных проектах.

Наследование классов

Нетрудно представить себе ситуацию, когда нужен объект, отличающийся от имеющегося совсем немного – одним или несколькими полями или методами. Самое простое решение в данном случае – это, конечно же, создание копии класса и её модификация по нашим требованиям, но, при необходимости изменения изначального класса, мы будем вынуждены править код в двух местах. помогает решить данную проблему наследование.

Мы уже говорили о том, что такое наследование. Как в версии **PHP4**, так и в **PHP5** ООП построено на основе модели одиночного наследования.

Чтобы один класс мог наследовать другой класс, в его определении ставится ключевое слово `extends`. Рассмотрим пример:

Пример 3.3.1

```
<?php
class ParentClass {
    public $parentvar;
    public function parentOne() {
        echo "Called parentOne()\n";
    }
    private function parentTwo() {
```

```
        echo "Called parentTwo()!\n";
    }
}
class ChildClass extends ParentClass {
    public function childOne() {
        echo "Called childOne()!\n";
    }
}
$v = new ChildClass();
$v->parentOne();/* В определении метода parentOne() нет
необходимости, так как он наследуется от класса ParentClass
*/
?>
```

В этом примере определены два класса: `ParentClass` и `ChildClass` (который расширяет `ParentClass`). Каждый из них реализует свой собственный уникальный набор функций, однако поскольку класс `ChildClass` расширяет класс `ParentClass`, он включает все свойства и методы, к которым имеет доступ. Вот здесь и проявляются уровни доступа `private`, `public` и `protected`. При наследовании методов и свойств в классе-наследнике будут доступны только те члены класса, которые были объявлены как `public` или `protected`.

Давайте расширим возможности класса товара для интернет-магазина, созданного нами ранее (в гл. **2.5**). Мы создадим на его основе класс `Book` – товар-книга.

Пример 3.3.2

```
<?php
require_once 'commodity.php'; /* Вспомните материал одного
из прошлых уроков. Мы подключаем в качестве библиотеки
файл, в котором описан класс товара. */
class Book extends Commodity {
    public $authors;
    function __construct($name, $authors,
                         $category, $price=null,
                         $availability=false) {
        parent::__construct($name,
                         $category, $price,
                         $availability);
            $this->authors=$authors;
        }
    function getAuthors() {
```

```
            return $this->authors;
        }
}
?>
```

В данном примере мы создаём новый конструктор, в теле которого происходит вызов конструктора родительского класса при помощи ключевого слова `parent`. Таким образом, мы перегружаем конструктор. Аналогично могут перегружаться и любые другие методы базового класса в классе-наследнике.

Давайте рассмотрим отвлеченный пример перегрузки метода:

Пример 3.3.3

```
<?php
class ParentClass {
    public function callMe() {
            echo "Вызван родительский класс!\n";
    }
}
class ChildClass extends ParentClass {
    public function callMe() {
            echo "Вызван дочерний класс!\n";
    }
}
$child = new ChildClass();
$child->callMe();
?>
/* Вызван дочерний класс!\n */
```

Кроме того, важно помнить о том, что переменная `$this` всегда будет ссылаться на экземпляр класса, который производит вызов метода класса, независимо от того, где находится соответствующий код.

Вот пример, иллюстрирующий это:

Пример 3.3.4

```
<?php
class ParentClass {
    public function callMe() {
            $this->anotherCall();
    }
    public function anotherCall() {
```

```
        echo "Вызван родительский класс!\n";
    }
}

class ChildClass extends ParentClass {
    public function anotherCall() {
        echo "Вызван дочерний класс!\n";
    }
}

$child = new ChildClass();
$child->callMe();
?>

/* Вызван дочерний класс!\n */
```

Что произойдет при вызове метода `callMe()`? Поскольку метод `callMe()` не определен в классе `ChildClass`, будет использоваться метод, определенный в родительском классе `ParentClass`. Если посмотреть на метод `callMe()` в классе `ParentClass`, то можно увидеть, что он вызывает еще один метод — `anotherCall()`. Хотя **PHP** выполняет метод `callMe()` из класса `ParentClass`, выполняться будет функция `anotherCall()` класса `ChildClass`, несмотря даже на то, что она существует в классе `ParentClass`.

Финальные методы

Финальный (**final**) класс или метод используется для того, чтобы предоставить разработчику возможность управлять наследованием. Классы или методы, объявленные как финальные, не могут быть расширены и/или перегружены классами-наследниками. Чтобы запретить перегрузку определенного класса или метода, в определении класса или метода должно стоять ключевое слово `final`:

Пример 3.4.1

```
<?php
final class finalClass {
    public function myFunction() {
        /* Логика функции */
    }
}
```

```
class semiFinalClass {
    final public function anotherFunc() {
        /* Логика функции */
    }
}
class myChild extends semiFinalClass {
    public function thirdFunction() {
        /* Логика функции */
    }
}
?>
```

Здесь определяются три разных класса. Первый из них, класс `finalClass`, никогда не будет родительским классом для класса-наследника, потому что сам по себе весь класс был объявлен как `final`. Все его методы также становятся финальными.

Класс `semiFinalClass` расширяется классом `myChild`, но метод `anotherFunc()`, находящийся в нем, никогда не будет перегружен классом-наследником.

Свойства классов не могут быть объявлены финальными.

Статические методы и свойства

Статическими (**static**) называются свойства и методы, являющиеся частью класса, но созданные для вызова за пределами контекста конкретного экземпляра объекта. Они также были введены в **PHP5**. Поведение этих методов аналогично поведению обычных методов в классе, кроме одной важной особенности – в них нельзя использовать переменную `$this` для ссылки на текущий экземпляр объекта.

Для создания статических методов или свойств используется ключевое слово `static` перед их объявлением в классе.

Поскольку статические методы и свойства не связаны с определенным экземпляром объекта, их можно вызывать за пределами контекста этого экземпляра. Для вызова статического метода или свойства используется оператор `::`.

Атрибут класса, объявленный статическим, не может быть доступен посредством экземпляра класса (но статический метод может быть вызван). Доступ к статическим свойствам класса не может

быть получен через оператор ->.

Пример 3.5.1

```
<?php
class test {
    public static $my_static = 'test';
    public function staticValue() {
        return self::$my_static;
    }
}
class child extends test {
    public function nextStatic(){
        return parent::$my_static;
    }
}
echo test::$my_static."\n";
$test = new test();
echo $test->staticValue()."\n";
echo $test->my_static."\n";
        // Не определено свойство my_static
echo $test::$my_static."\n";
$classname = 'test';
echo $classname::$my_static."\n";
        // Начиная только с PHP 5.3.0
echo child::$my_static."\n";
$child = new child();
echo $child->nextStatic()."\n";
?>
```

В целях совместимости с **PHP4**, сделано так, что если не использовалось определение области видимости, то член или метод будет рассматриваться, как если бы он был объявлен как public.

Статические свойства могут использоваться, например, для счётчиков количества экземпляров объектов одного класса.

Сравнение объектов

В **PHP5** сравнение объектов является более сложным процессом, чем в **PHP4**, а также процессом, более соответствующим идеологии ООП.

При использовании оператора сравнения (==), свойства объек-

тов просто сравниваются друг с другом, а именно: два объекта равны, если они содержат одинаковые свойства и одинаковые их значения и являются экземплярами одного и того же класса.

С другой стороны, при использовании оператора идентичности (===), свойства объекта считаются идентичными тогда и только тогда, когда они ссылаются на один и тот же экземпляр одного и того же класса.

Рассмотрим это на примере.

Пример 3.6.1

```
<?php
function bool2str($bool) {
    return ($bool===false ? "FALSE" : "TRUE";
}

function compareObjects(&$o1, &$o2) {
    echo "==: ".bool2str($o1==$o2)."\n";
    echo "===: ".bool2str($o1===$o2)."\n\n";
}

class Flag {
    public $flag;
    function Flag($flag=true) {
        $this->flag=$flag;
    }
}

class OtherFlag {
    public $flag;
    function OtherFlag($flag=true) {
        $this->flag=$flag;
    }
}

$o=new Flag();
$p=new Flag();
$q=$o;
$r=new OtherFlag();

echo "Два экземпляра одного и того же класса\n";
compareObjects($o, $p);
```

```php
echo "Две ссылки на один и тот же экземпляр\n";
compareObjects($o, $q);

echo "Экземпляры двух разных классов\n";
compareObjects($o, $r);
?>
```

Вывод этого кода будет выглядеть так:

```
Два экземпляра одного и того же класса
==: TRUE
===: FALSE

Две ссылки на один и тот же экземпляр
==: TRUE
===: TRUE

Экземпляры двух разных классов
==: FALSE
===: FALSE
```

ДОМАШНЕЕ ЗАДАНИЕ.

Реализовать класс «Органайзер», который должен хранить информацию о планируемых делах. Методы данного класса должны позволять:

- добавить запись в список дел – запланировать. Добавление может осуществляться при помощи формы;
- распечатать список дел запланированных на день, на неделю, на месяц;
- отменять запланированные дела;

Главным при выполнении домашнего задания является разработка необходимых свойств и методов. Хранение промежуточных данных можно осуществить путём вывода их в поля формы при формировании страниц, а затем обратной передачи на сервер для дальнейшей обработки.

ИТОГИ.

В нынешнем уроке мы рассмотрели основу концепции ООП в применении её к **PHP5**, а именно:

- описание классов;
- создание объектов;
- наследование и перегрузка классов;
- клонирование и сравнение объектов.

В следующем уроке мы продолжим знакомство с ООП в **PHP5** и рассмотрим усовершенствованные технологии работы с объектами.

В современном **web**-программировании с использованием **PHP** работа с объектами используется очень широко.

В будущем, познакомившись с другими возможностями PHP, а именно работа с файлами, базами данных и сетевыми протоколами, вы научитесь разрабатывать более прикладные классы, находящие применение для построения сложных **web**-проектов.