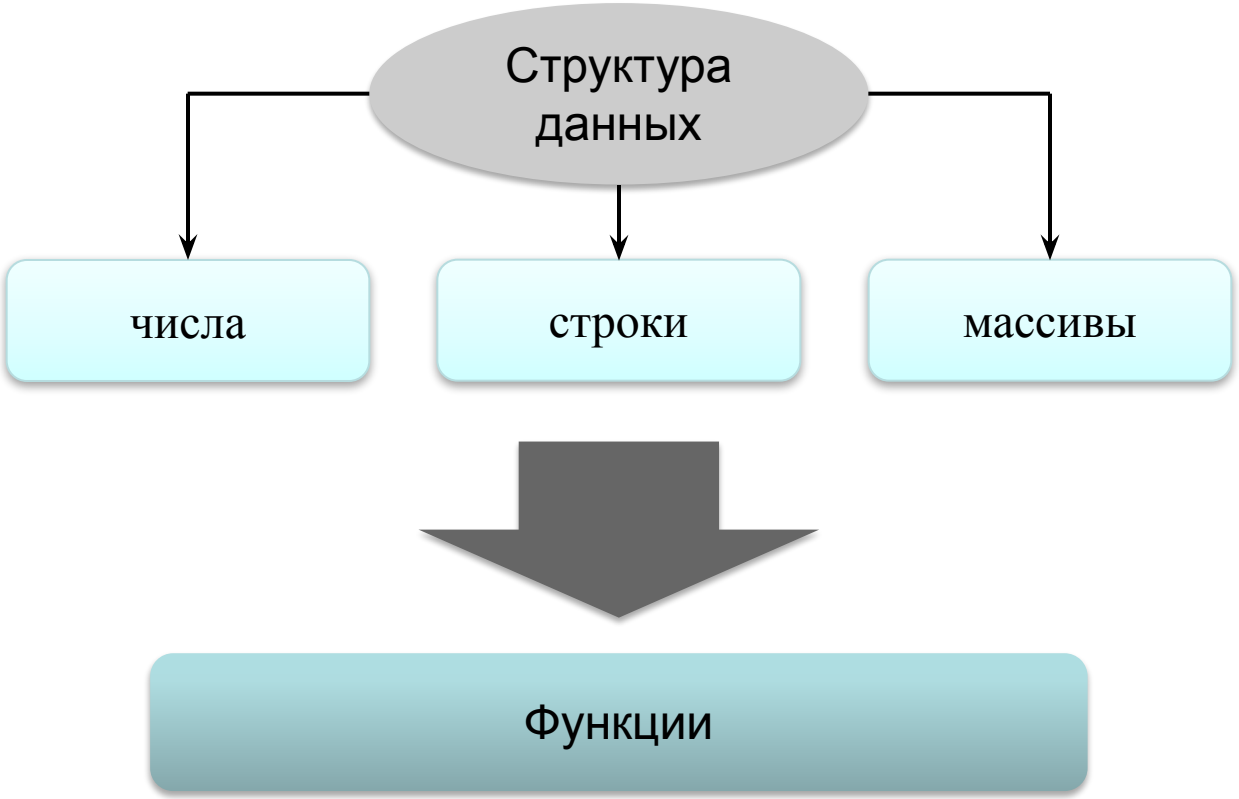
Объектно-ориентированное программирование (ООП)

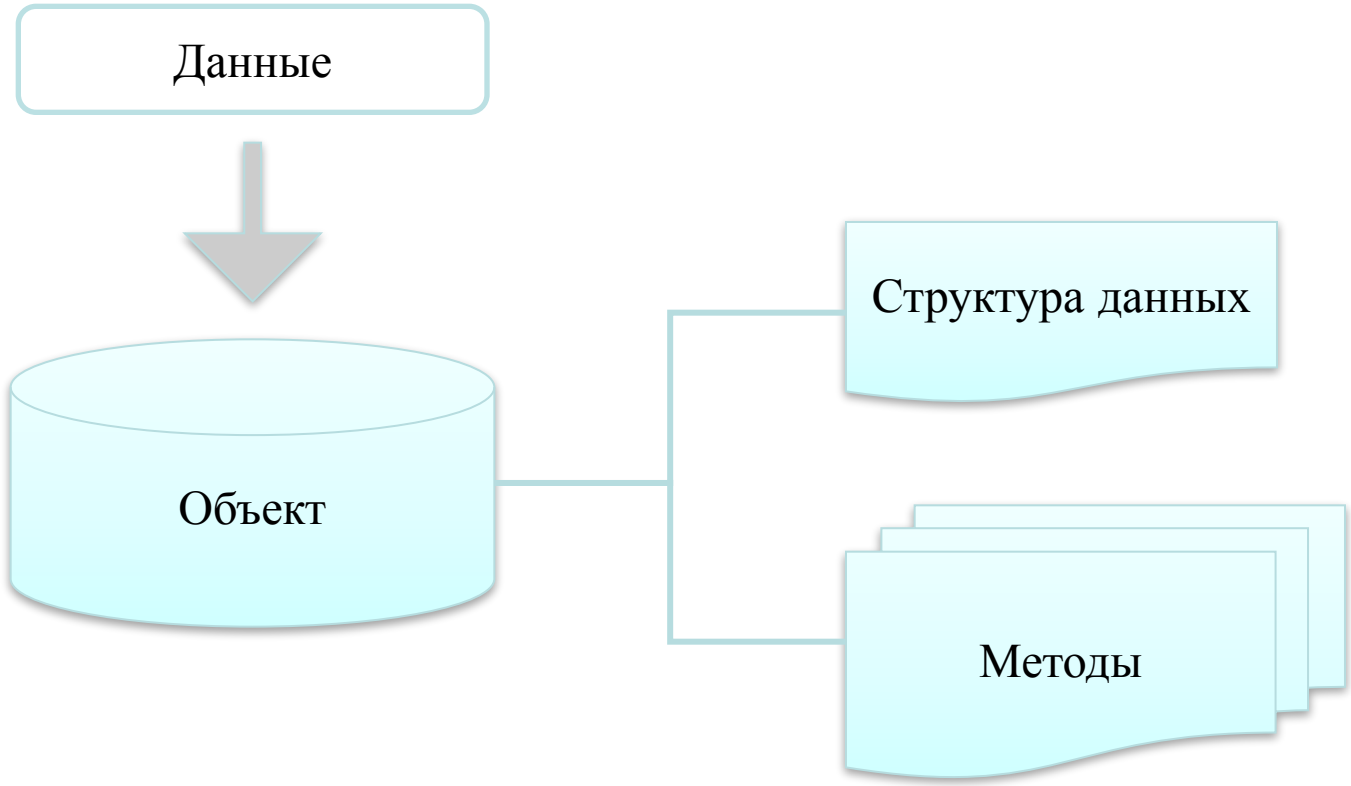
Жукова Г.В.

Процедурный стиль программирования

Проблема процедурного программирования в том, что данные и функции их обработки не связаны между собой

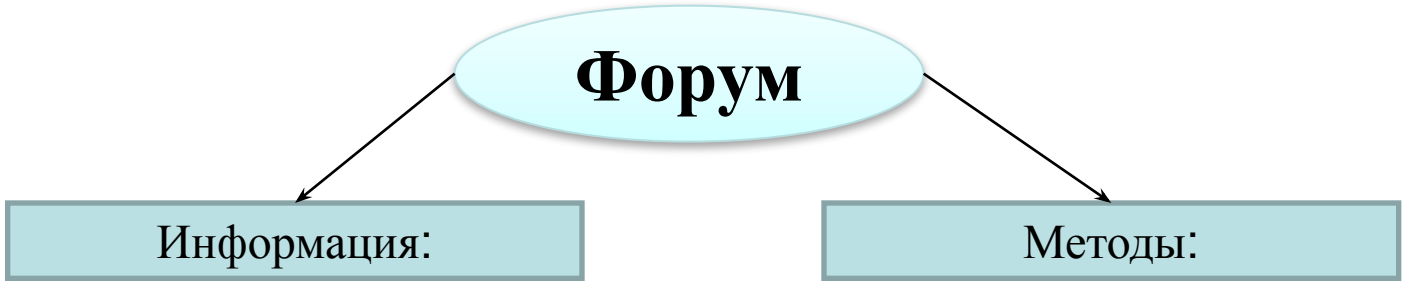


Что такое объектно-ориентированное программирование?



В ООП данные и функции для их обработки (методы) объединены в объекты

Пример



- Имя
- Логин
- Электронный адрес
- Пароль

и т.д

- Регистрация
- Авторизация
- Выход из системы

и т.д.

Объектно-ориентированное программирование (ООП)

Объектно-ориентированное программирование - это стиль кодирования, который позволяет разработчику группировать схожие задачи в классы

Объектно-ориентированное программирование основано на:

Инкапсуляции
Полиморфизме
Наследовании

Основные понятия ООП

Инкапсуляция – это механизм, объединяющий данные и обрабатывающие их функции (обычно называемые методами), как единое целое.

Когда код и данные связываются вместе подобным образом, создается **объект**. Иными словами, **объект** — это элемент, поддерживающий инкапсуляцию. Основной единицей инкапсуляции является **класс**, который описывает данные и методы, т.е. определяет форму объекта.

Полиморфизм - это свойство системы использовать объекты с одинаковым интерфейсом без информации о типе и внутренней структуре объекта. В более общем смысле понятие полиморфизма нередко выражается следующим образом: "один интерфейс — множество методов". Это означает, что для группы взаимосвязанных действий можно разработать общий интерфейс.

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. **Класс** можно рассматривать как своего рода "контейнер" для логически связанных данных и функций обрабатывающих их. Если сказать проще, то класс - это своеобразный тип данных.

Экземпляр класса - это **объект**. **Объект** - это совокупность переменных (свойств) и функций (методов) для их обработки. Свойства и методы называются членами класса. Вообще, объектом является все то, что поддерживает инкапсуляцию.

Если класс можно рассматривать как тип данныхЕсли класс можно рассматривать как тип данных, то объект — как переменную (по аналогии). Скрипт может одновременно работать с несколькими объектами одного класса, как с несколькими переменными.

Структура класса

Описание классов в PHP начинаются служебным словом **class**, за которым следует **имя класса** и **{ }:**

```
class MyClass
{
    // описание членов класса – свойств  и методов класса
}
```

Объект класса (новый экземпляр класса) создается с использованием оператора **new**:

```
$obj1 = new MyClass;
$obj2 = new MyClass;
```

Скрипт может одновременно работать с несколькими объектами одного класса, как с несколькими переменными

Данные описываются с помощью служебного слова **var**.

Пример класса

```
// Создаем новый класс  MyClass :
class MyClass {
// данные (свойства):
var $name;
var $addr;
// методы:
 function Name() {
 echo "<h3>John</h3>";
 }
}
// Создаем объект класса  MyClass:
$object = new  MyClass;
```

Доступ к членам класса

Мы рассмотрели, каким образом описываются классы и создаются объекты. Теперь нам необходимо получить доступ к членам класса, для этого в PHP предназначен оператор ->.

Специальный указатель **$this** применяется для обозначения объекта.

$this - это ссылка на объект, её нельзя использовать в статических методах или для статических свойств, для этого существует ключевые слова self и parent.

Чтобы получить доступ к членам класса (свойствам и методам) внутри класса, необходимо использовать указатель **$this**, который всегда относится к текущему объекту.

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

Обратите внимание на отсутствие знака доллара перед **name**.

Пример

```
// Создаем новый класс Coor:
    class Coor {


// данные (свойства):
    var $name;


// методы:
     function Getname() {          // метод  Getname()
     echo $this -> name;
     }
     function Setname($name) {      // метод  Setname()
     $this -> name = $name;
     }
    }


    $object = new Coor;              // Создаем объект класса Coor:
    $object -> Setname("Nick");      // для изменения имени - метод Setname()
    $object -> Getname();            // для доступа -  Getname()


    // Сценарий выводит 'Nick'
```

Свойства и методы класса живут в разделенных "пространствах имен", так что возможно иметь свойство и метод с одним и тем же именем.

```
class Foo
{
    var $bar = 'свойство';

     function bar() {
        return 'метод';
    }
}

$obj = new Foo();
echo $obj -> bar;
echo "<br>";
echo $obj -> bar();
```

Подведем промежуточные итоги

Объявление класса должно начинаться с ключевого слова **class** (подобно тому, как объявление функции начинается с ключевого слова function).

Каждому объявлению свойства, содержащегося в классе, должно предшествовать ключевое слово **var**. Свойства могут относиться к любому типу данных, поддерживаемых в PHP.

После объявлений свойств следуют объявления методов, очень похожие на типичные объявления пользовательских функций. Методу также можно передавать параметры.

Области видимости свойств и методов

Доступ к свойствам и методам определяется через модификаторы:

public (общедоступные) – доступ как внутри класса, так и вне класса;

protected (защищённые) – доступ только внутри класса или внутри производных классов (классов, которые расширяют базовый класс, содержащий метод с директивой **protected**);

private (частные) – доступ только внутри класса, в котором они определены.

Методы, где определение модификатора отсутствует, определяются как **public**.

Свойства определенные с помощью **var**, будут объявлены как **public**.

static (статические) – доступ без инициализации класса. Статические свойства сохраняют свои значения на протяжении работы всего скрипта.

Пример

```
class MyClass
{
    public      $public = 'Public';
    protected $protected = 'Protected';
    private     $private = 'Private';

    public function  printHello()
    {
        echo $this->public;
        echo $this->protected;
        echo $this->private;
    }
}
$obj = new MyClass();
echo $obj -> public;    // Работает
echo "<br>";
$obj -> printHello();   // Выводит Public, Protected и Private
```

Определение свойств класса

Определение свойств класса или инициализация объекта - это присвоение свойствам объекта первоначальные значения.

```
class MyClass
{
    $prop1="Свойство объекта";
}

$obj=new MyClass;
echo $obj->prop1;    // Выводим свойство
```

Инициализация объектов

Имя класса Coor и он содержит два свойства: имя человека и город его проживания. Можно написать метод который будет выполнять инициализацию объекта, например Init():

```php
<?php
// Создаем новый класс Coor:
class Coor {
// данные (свойства):
var $name;
var $city;

// Инициализирующий метод:
 function Init($name) {
 $this->name = $name;
 $this->city = "London";
 }

}

// Создаем объект класса Coor:
$object = new Coor;
// Для инициализации объекта сразу вызываем метод:
$object->Init();
?>
```

Конструкторы

Довольно часто при создании объекта требуется задать значения некоторых свойств. К счастью, разработчики технологии ООП учли это обстоятельство и реализовали его в концепции конструкторов.

Конструктор представляет собой метод, который задает значения некоторых свойств (а также может вызывать другие методы). Конструкторы вызываются автоматически при создании новых объектов.

В версиях до PHP5 имя метода конструктора совпадало с именем класса к которому он относится, а начиная с версии PHP5 имя метода конструктора необходимо называть **__construct()** (**это 2 подчеркивания перед словом __construct()**).

Раньше создание объекта и инициализация свойств выполнялись раздельно. Конструкторы позволяют выполнить эти действия за один этап.

Конструктор автоматически вызывается при создании объекта. Давайте попробуем создать класс, который будет содержать метод **__construct()**:

```
class MyClass
{
public  function  __construct()
   {
     echo  "Я только что был создан!";
   }
}
  $myObject = new MyClass();  // выведет "Я только что был создан!"
```

Пример Конструктора

```
Class Product{

  private $title;
  private $price;
  private $discount;


  public function __construct($title,
       $price, $discount)
  {
    $this->title = $title;
    $this->price = $price;
    $this->discount = $discount;
  }

  public function getProduct()
  {
    echo  'Название товара: '.$this->title.'<br>';
    echo  'Цена товара: '.$this->price .' $<br>';
    echo  'Скидка: '.$this->discount .'%<br>';
  }
}
$product = new Product('Мастер создания
       форм', 20, 25);
$product->getProduct();
```

Деструкторы

Подобно конструкторам в PHP существуют деструкторы, которые вызываются строго перед тем, как объект удаляется из памяти.

Это очень полезный метод для корректной очистки свойств класса (например, для правильного закрытия соединения с базой данных).

Примечание: PHP автоматически удаляет объект из памяти, когда не остается ни одной переменной, указывающей на него.

Например, если вы создадите новый объект и сохраните его в переменной **$myObject**, а затем удалите ее с помощью метода **unset($myObject)**, то сам объект также удалится. Также, если вы создали локальную переменную в какой-либо функции, она (вместе с объектом) удалится, когда функция завершит работу.

В отличии от конструкторов, в деструкторы нельзя передавать никакие параметры!

Пример Деструктора

Чтобы создать деструктор, добавьте в класс метод ___destruct().

```
class MyClass
{
 public function __destruct()
  {
   echo "Я деструктор. Объект был удален. Пока!";
   }
}
$myObject = new MyClass();

// для явного вызова деструктора и удаления объекта можно использовать функцию unset( )

unset($myObject);    // отобразит "Я деструктор. Объект был удален. Пока!"

echo “А теперь завершается работа сценария”;
```

Необходимость в вызове деструкторов возникает лишь при работе с объектами, использующими большой объем ресурсов, поскольку все переменные и объекты автоматически уничтожаются по завершении сценария.

```
class MyClass
{
 public function __destruct()
{
   echo "Я деструктор. Объект был удален. Пока!";
 }
}
$myObject = new MyClass();

exit(); // отобразит  "Я деструктор. Объект был удален. Пока!"
```

Магические методы в PHP

Конструктор и деструктор – это так называемые «магические методы».

Магические методы - это специальные методы, которые вызываются, когда над объектом производятся определенные действия.

Все методы начинающиеся со знака __ (два подчеркивания перед именем метода), PHP считает «магическими».

Полный список магических методов смотрите в руководстве по PHP

Вложенные объекты

Свойства объектов сами могут быть объектами.
Тогда говорят, что объект вложен в другой объект.

```
class Room
{
  public $name;
  function __construct($name=‘безымянная’)
  {
    $this -> name = $name;
  }
}
class House
{
  public $room;
}
$home = new House;
$home -> room[] = new Room(‘спальня’);
$home -> room[] = new Room(‘кухня’);
print($home -> room[1] -> name);
```

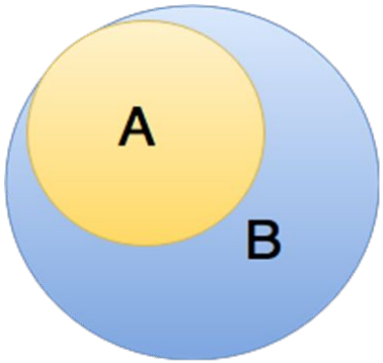
В этом примере объект $home содержит массив вложенных объектов room.

Наследование классов в PHP

Это механизм языка, позволяющий описать новый класс на основе уже существующего (родительского, базового) класса. Потомок может добавить собственные методы и свойства, а также пользоваться родительскими методами и свойствами.

Наследование - это не просто создание точной копии класса, а расширение уже существующего класса, чтобы потомок мог выполнять какие-нибудь новые, характерные только ему функции.

Схематичное изображение наследования классом B класса A



Чтобы создать новый класс, наследующий поведение существующего класса, надо использовать ключевое слово **extends** в его объявлении. Например:

```
class классA extends классB
  {
  .......
  }
```

Ключевое слово **extends** говорит о том, что создаваемый класс является лишь "расширением" класса **A**, и не более того. То есть **B** содержит те же самые свойства и методы, что и **A**, но, помимо них и еще некоторые дополнительные, "свои".

Теперь "часть A" находится прямо внутри класса **B** и может быть легко доступна, наравне с методами и свойствами самого класса **B**.

Обратите также внимание: мы можем теперь забыть, что B унаследовал от A некоторые свойства или методы — снаружи все выглядит так, будто класс B реализует их самостоятельно.

Немного о терминологии: родительский класс A принято называть базовым классом, а дочерний класс B - производным от A. Иногда базовый класс также называют суперклассом, а производный - подклассом.

Имейте в виду, что производный класс имеет только одного родителя.

Класс-наследник (подкласс) может переопределить свойства и методы родителя – это называется **перегрузкой**.

Пример наследования

```php
<?php
class Parent {
  function parent_funct() { echo "<h1>Это родительская функция</h1>"; }
  function test () { echo "<h1>Это родительский класс</h1>"; }
}

class Child extends Parent {
  function child_funct() { echo "<h2>Это дочерняя функция</h2>"; }
  function test () { echo "<h2>Это дочерний класс</h2>"; }
}

$object = new Parent;
$object = new Child;

$object->parent_funct(); // Выводит 'Это родительская функция'
$object->child_funct(); // Выводит 'Это дочерняя функция'
$object->test(); // Выводит 'Это дочерний класс'
?>
```

Дочерний класс (подкласс) Child наследует все методы и свойства суперкласса Parent

Обращение к элементам классов. Оператор разрешения области видимости (::)

Оператор разрешения области видимости или просто "двойное двоеточие" - позволяет обращаться к константам, к статическим свойствам и методам класса, к переопределенным свойствам и методам класса.

При обращении к этим элементам извне класса, необходимо использовать имя этого класса.

Начиная с версии PHP 5.3.0, стало возможным обратиться к классу с помощью переменной.

Для обращения к свойствам и методам внутри самого класса используются ключевые слова **SELF, PARENT, STATIC.**

Использование :: вне объявления класса

При обращении к этим элементам извне класса, необходимо использовать имя этого класса.

Начиная с версии PHP 5.3.0, стало возможным обратиться к классу с помощью переменной.

```
class MyClass
{
    const CONST_VALUE = 'Значение константы';
}

$classname = 'MyClass';
echo $classname :: CONST_VALUE;  // Начиная с версии PHP 5.3.0
echo MyClass :: CONST_VALUE;
```

Использование :: внутри объявления класса

Для обращения к свойствам и методам внутри самого класса используются ключевые слова ***self***, ***parent*** и ***static***.

```
class OtherClass extends MyClass
{
    public static $my_static = 'статическая переменная';

    public static function doubleColon() {
        echo parent :: CONST_VALUE . "\n";
        echo self :: $my_static . "\n";
    }
}

$classname = 'OtherClass';
echo $classname :: doubleColon();   // Начиная с версии PHP 5.3.0
OtherClass :: doubleColon();
```

Пример: Обращение к элементам классов

Используя "двойное двоеточие", можно обращаться к методам классов. При обращении к методам классов, программист должен использовать имена этих классов.

```
class A {
    function example() {
        echo "Это первоначальная функция A :: example(). <br>";
    }
}

class B extends A {
    function example() {
        echo "Это переопределенная функция B :: example().<br>";
        A :: example();
    }
}

// Не нужно создавать объект класса A.

$b = new B;          // Создаем объект класса B

$b->example();     // Выводит следующее:
```

Это переопределенная функция B :: example().
Это первоначальная функция A :: example().

Обращение к методу в родительском классе

Когда дочерний класс перегружает методы, объявленные в классе-родителе, PHP не будет осуществлять автоматический вызов методов, принадлежащих классу-родителю. Этот функционал возлагается на метод, перегружаемый в дочернем классе. Данное правило распространяется на конструкторы и деструкторы, перегруженные и "магические" методы.

```
class MyClass {
    protected function myFunc() {
        echo "MyClass :: myFunc()\n";
    }
}
class OtherClass extends MyClass {
    public function myFunc() {  // Переопределение  родительского метода
        parent :: myFunc();          // Вызов родительского метода
        echo "OtherClass :: myFunc()\n";
    }
}
$class  =  new OtherClass();
$class -> myFunc();
```

Замечание

Если производный класс не содержит собственного конструктора, то при создании его объекта используется конструктор родительского класса. Если в производном класса существует собственный конструктор, то конструктор родительского класса не вызывается.

Чтобы вызвать конструктор, объявленный в родительском классе, следует обратиться к методу **parent::__construct()** внутри конструктора класса-потомка.

Если в классе-потомке не определен конструктор, то он может наследоваться от родительского класса как обычный метод (если он не определен как приватный).

Как и в случае с конструкторами, деструкторы, объявленные в родительском классе, не будут вызваны автоматически. Для вызова деструктора, объявленном в классе-родителе, следует обратиться к методу **parent::__destruct()** в теле деструктора-потомка.

Деструктор будет вызван даже в том случае, если скрипт был остановлен с помощью функции exit() Деструктор будет вызван даже в том случае, если скрипт был остановлен с помощью функции exit(). Вызов exit() в деструкторе предотвратит запуск всех последующих функций завершения.

Статические методы и свойства классов

Методы и свойства можно также использовать, определяя их как **статические** данные класса. Термин статические означает, что мы можем получать доступ и к свойствам, и к методам в контексте класса, а не объекта. Т.е. статические свойства едины для всего класса и не могут принадлежать ни одному из объектов класса. Такие методы и свойства помечаются ключевым словом **static**. К статическому свойству, можно обратиться не создавая объекта:

```
echo mammal::$feeding;
```

Аналогично можно определить статический метод и использовать его без создания объекта такого класса

```
class mammal{
    static  $feeding=‘молоко’;
    static function moving() {
    echo “Все звери двигаются  “;
    }
}
echo mammal::moving();
```

Т.е. из-за пределов класса мы обращаемся к статическим данным с помощью имени класса.

Статические методы и свойства классов

При вызове статического метода используется имя класса и оператор **::**, а не **->**, так как статический метод относится ко всему классу, а не к конкретному объекту этого класса. Поэтому в статическом методе становится невозможным использовать указатель **$this**, т.к. при вызове статического метода неизвестно, в контексте какого объекта он вызывается.

Пример

Внутри класса StaticExample можно использовать ключевое слово **self**.

```
class StaticExample {
    static public $aNum = 0;
    static public function sayHello()
    {
        self::$aNum++;
        print "Привет! (" . self::$aNum . ")\n";
    }
}
```

Обратите внимание: кроме случаев обращения к переопределенному методу родительского класса, конструкция "**::**" должна всегда использоваться только для доступа к статическим методам или свойствам.

По определению статические методы не вызываются в контексте объекта. По этой причине статические методы и свойства часто называют переменными и свойствами класса.

Константы

Кроме обычных свойств класс может иметь константы. Константы представляют поля, значения которых нельзя изменить. Для определения констант используется инструкция **const,** перед именем константы знак **$** не ставится, этим константы отличаются от обычных свойств:

```
class User
{
   const EMPLOYEE = 1;
   const MANAGER = 2;
   const ADMIN = 3;
   public function SayGoods() {
      echo self::ADMIN; //результат будет равен 3
   }
}
echo User::MANAGER; //2
```

В примере доступ к этой константе был получен, с помощью использования ключевого слова self и оператора "**::**". Обращение к константам идет также, как и статическим свойствам: User**::**MANAGER.

Попытка обратиться к константе используя псевдопеременную $this ($this->ADMIN), приведёт к ошибке. Константы недоступны объектам.

Копирование объектов

Так уж устроен PHP, что в нем все **переменные**, в том числе и **объекты**, всегда рассматриваются как простой набор значений и копируются целиком.

Например, если у нас есть объект **$a** и мы выполняем оператор **$b=$a**, то все содержимое **$a** будет скопировано в **$b** один-в-один.

Когда речь идёт о **переменных** всё просто:

```
$firstVar = 123;
$lastVar = $firstVar;     // теперь в $lastVar тоже будет значение 123
```

(Данный пример может работать даже с объектами, но в PHP версий 4)

Пример копирования объектов в PHP4

```
class A {
// Создаем новый метод:
 function Test() {
 echo "<h1>Привет!</h1>";
 }
}
// Создаем объект класса A:
$a=new A();
// Копируем объект $a:
$b=$a;
// Теперь работаем с новым объектом $b
$b->Test();      // Выводит ‘Привет!'
```

В PHP5

При создании копии объекта с помощью оператора присваивания ($obj2 = $obj), создается ссылка на объект $obj, а не копия всех свойств и методов объекта $obj.

См. пример

НЕУДАЧНЫЙ ПРИМЕР КОПИРОВАНИЯ ОБЪЕКТОВ

```
class User {
    public $name;
    public function __construct($name) {
     $this->name = $name;
    }
}
$user = new User("Иван");
echo $user->name;      // Вывод: Иван

$user2 = $user;
$user2->name = "Маша";
echo $user->name;      // Вывод: Маша, не смотря на то что
    слово "Маша"   мы поставили в объект $user2
```

Чтобы скопировать свойства и методы объекта, надо применить **клонирование** $user

К объекта в PHP

```
class Book {                                    //Создаем простой класс "Книга"
 private $title = 'Мастер и Маргарита';         //закрытое свойство названия
 public function setTitle($newTitle)            //Метод установки нового названия
 {
      $this->title = $newTitle;
 }
 public function getTitle()                     //Метод получения названия книги
 {
      return $this->title;
 }
}
//Функция изменения названия книги
function changeBookName($bookObject) {
$bookObject->setTitle('Новое название');
}
$book = new Book();                 //Создаем объект и проводим эксперимент
echo $book->getTitle();             //Результат: Мастер и Маргарита
changeBookName($book);
echo $book->getTitle();              //Результат: Новое название
```

Чтобы скопировать свойства и методы объекта, надо применить **клонирование**

В PHP5 процесс копирования **объектов** также называют клонированием. PHP позволяет клонировать объект с помощью оператора **clone**. Выражение с его использованием можно записать следующим образом:

$копия_объекта = clone $обЪект;

При выполнении клонирования создается полная копия объекта, не имеющая никакой связи со своим оригиналом. Она будет принадлежать тому же классу, иметь те же методы, а свойства содержать аналогичные значения.

```
//Создаем новый объект и клонируем его
$origin = new Book();
$clone = clone $origin;

//Клон имеет тоже состояние что и оригинал
echo $origin->getTitle(); //Результат: Мастер и Маргарита
echo $clone->getTitle(); //Результат: Мастер и Маргарита

//Меняем название книги клона
$clone->setTitle('Война и мир');
echo $origin->getTitle(); //Результат: Мастер и Маргарита
echo $clone->getTitle(); //Результат: Война и мир
```

```
class User {
    public $name;
    public function __construct($name) {
        $this->name = $name;
    }

    public function __clone() {
        echo "Клон сработал!";
    }
}

$user = new User("Иван");

$user2 = clone $user;  // Вывод: Клон сработал! ( потому что сработал метод __clone)
```

Магический метод __clone()

Заметка: *Нужно отметить, что клонирование объектов в PHP относится к одной из редко используемых операций. Её повсеместное применение может вызвать снижение производительности программы. Это, безусловно, удобный инструмент, но использовать его нужно по необходимости.*

Класс может определить реализацию специального метода **__clone()**. Он работает как конструктор клонируемого объекта. Единственное его отличие от реального конструктора в том, что он не может принимать параметры. Обратите внимание, что имя метода содержит **два нижних подчеркивания**.

Принцип действия магического метода **__clone()** очень прост. Если он определен в классе, PHP вызовет его автоматически для клонированного объекта. Это может быть необходимо при изменении его состояния. Например, если вы хотите, чтобы копия объекта немного отличалась от оригинала.

Итак, если вам нужно сделать какие-то действия при клонировании объекта, то в PHP есть метод __clone.

Код клонирования объектов. Пример

```
class Student {          //Создаем простой класс "Студент университета“
public $name, $surname, $age;      //Свойства имени, фамилии и возраста
//Конструктор инициализирует свойства
public function __construct($name, $surname, $age) {
$this->surname = $surname;
$this->name = $name;
$this->age = $age;
}
public function __clone() {          //Метод обнуляет значение свойств клона
$this->surname = 'Неизвестно...';
$this->name = 'Неизвестно...';
$this->age = 'Неизвестно...';
}
public function getInfo() {          //Вывод информации о студенте в браузер
echo 'Информация о студенте<br>';
echo "Имя: {$this->name}<br>";
echo "Фамилия: {$this->surname}<br>";
echo "Возраст: {$this->age}<br>";
}
}
```

Код клонирования объектов. Пример. Продолжение

```
//Создаем новый объект студента, заполняем его  свойства, а затем клонируем
$student = new Student('Денис', 'Демидов', 20);
$student->getInfo();
                    /*      Результат:
                    Информация о студенте
                    Имя:  Денис
                    Фамилия:  Демидов
                    Возраст:  20  */
$clone = clone $student;
$clone->getInfo();
                    /*      Результат:
                    Информация о студенте
                    Имя:  Неизвестно...
                    Фамилия:  Неизвестно...
                    Возраст:  Неизвестно...   */
```

Код клонирования объектов. Пример 2

```php
<?php
class MethodClone {

    public $_newElement;

    public function __construct() {
        $this->_newElement = __CLASS__;
    }

    public function __clone() {
        $this->_newElement = '<div>Клон</div>';
    }

}

$_Class_MethodClone = new MethodClone();
$_new_MethodClone = clone $_Class_MethodClone;

echo $_new_MethodClone->_newElement; # Результат: "Клон";
echo $_Class_MethodClone->_newElement; # Результат: "MethodClone";
?>
```

Ещё кому-то может быть полезно знать как запретить клонирование, для этого нам просто нужно ставить спецификатор доступа **private** возле метода **__clone**.

```
class User {
	private function __clone() {
		echo "Клон сработал!";
	}
}

$user = new User;

$user2 = clone $user;	// тут будет остановлен скрипт и будет показана ошибка
```

Сравнение объектов

В PHP 4 объекты сравниваются очень просто: по именам.

В PHP5 сравнение объектов является более сложным процессом, чем в PHP4, а также процессом, более соответствующим идеологии объектно-ориентированного языка.

При использовании оператора сравнения (==), свойства объектов 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.

Сравнение объектов

При этом равенство свойств так же проверяется оператором «**==**».

Сравнение объектов с помощью тождественного оператора «**===**» имеет более строгие ограничения, но легче поддается объяснению. Две переменные, содержащие ссылки, равны между собой только когда они ссылаются на один и тот же объект.

Рассмотрим несколько примеров:

Сравнение объектов. Пример

```
<?php
class A {
// Создаем новый метод:


 function Test() {
 echo "<h1>Hello!</h1>";
 }}
// Создаем объект класса A:
$a=new A();
// Создаем объект класса A:
$b=new A();
// Выводит 'Объекты равны':
if ($a===$b) echo "<h2>Объекты равны</h2>";
?>
```

Сравнение объектов. Пример

Оператор эквивалентности **===** (тройное равенство) не только сравнивает два выражения, но также их типы.

```
$a=0;                    // ноль
$b="";                   // пустая строка

if($a===$b) echo "a и b равны";    // Ничего не выводит
```

Сравнение объектов. Пример

```
//Определяем класс "Кошка"
class Cat
{
    public $nickname;       //Свойство имени кошки
}
//Создаем объекты, назначаем разные свойства и сравниваем между собой
$firstCat = new Cat();
$firstCat->nickname = 'Снежок';

$secondCat = new Cat();
$secondCat->nickname = 'Уголек';
var_dump($firstCat = = $secondCat);          //Результат: bool(false)

//Устанавливаем одинаковые свойства и повторяем обычное сравнение
$firstCat->nickname = 'Мурзик';
$secondCat->nickname = 'Мурзик';
var_dump($firstCat = = $secondCat);          //Результат: bool(true)
```

Сравнение объектов. Пример

Обычное сравнение производит неявное преобразование типов свойств

```
$firstCat->nickname = false;
$secondCat->nickname = 0;
var_dump($firstCat == $secondCat);  //Результат: bool(true)

 //Производим тождественное сравнение разных объектов
var_dump($firstCat === $secondCat);       //Результат: bool(false)

//Производим тождественное сравнение одного объекта
$secondCat = $firstCat;
var_dump($firstCat === $secondCat);       //Результат: bool(true)
```

var_dump() - функция отображает структурированную информацию об одном или нескольких выражениях, включая их тип и значение.

Что же такое ссылки в PHP?

Ссылка — это способ обратиться к переменной с помощью другого имени. Ссылки в PHP - это средство доступа к содержимому одной переменной под разными именами. В PHP имя переменной и её содержимое - это разные вещи, поэтому одно содержимое может иметь разные имена.

```
$a =& $b;   //  $a указывает на то же значение что и $b.
```

Присвоение ссылки в PHP:

```
$myVar = "Привет!";
$Anothervar = $myVar;
$Anothervar = "Увидимся позже";
echo $myVar;          // Выведет "Привет!"
echo $anotherVar;   // Выведет "Увидимся позже"
```

В примере мы создали переменную *$myVar* со значением «Привет!». Затем мы присвоили значение другой переменной *$anotherVar*. Это *копия* значения первой переменной во вторую. Затем мы изменим значение, сохраненное в *$anotherVar* на «Увидимся позже».

Поскольку две переменные являются независимыми, *$myVar* по-прежнему сохраняет свою первоначальное значение («Привет!»), которое будет выедено на странице. Пока всё идёт хорошо. А теперь давайте изменим пример, чтобы присвоить переменной *$myVar* значение *$anotherVar*, используя *ссыл ку*, а не значение. Чтобы сделать это, мы просто напишем знак амперсанда («&») после знака равенства :

```
$myVar = "Привет!";
$Anothervar =&$myVar;  //$Anothervar   указывает на то же значение
    что и  $myVar.

$Anothervar= "Увидимся позже";
echo $myVar;        // Выведет "Увидимся позже"
echo $anotherVar;       // Выведет "Увидимся позже"
```

Теперь вы можете видеть, что *$myVar* также изменен на «Увидимся позже»! Почему это произошло?

Вместо того, чтобы присвоить значение переменной *$myVar* переменной *$anotherVar* — которые просто создают две независимых копии одного и того же значения — мы сделали переменную *$anotherVar ссылкой* на значение *$myVar*. Другими словами, *$myVar* и *$anotherVar* оба указывают на одно и то же значение. Таким образом, когда мы присвоили новое значение переменной, *$anotherVar* значение переменной *$myVar* также изменилось.

Обратите внимание на то, что мы могли бы изменить значение переменной *$myVar* на «Увидимся позже» вместо изменения переменной *$anotherVar* и результат был бы точно такой же. Две переменных, по сути, являются идентичными.

Ссылки в PHP, когда используется ключевое слово *$this*

При написании объектно-ориентированного кода часто используется ключевое слово *$this*. При использовании *$this* в пределах метода объекта, выполняется указание на текущий объект. Стоит запомнить, что ***$this*** всегда ссылается на объект, а не на его копию.

Например:

```
class MyClass {
    var $aProperty = 123;
    function aMethod() {
     $This- >aProperty = 456; // $это ссылка на объект
    }
}
$myObject = new MyClass();
$myObject->aMethod();
echo $myObject->aProperty; // Выведет "456"
```

В примере приведенном выше ***$this*** — это ссылка на объект. Метод может изменять свойство объекта на новое значение в пределах этого объекта.

Использование оператора проверки типа **instanceof**

В PHP существует 12 предопределенных типов данных, включая простые типы, например, целые и с плавающей точкой, комплексные типы (массивы и объекты) и специальные типы (ресурсы и NULL).

Поскольку PHP поддерживает классы и объекты, у вас также есть возможность определять собственные типы. На самом деле, каждый раз при создании класса вы определяете новый тип данных. Если вы знаете тип объекта, вы знаете его характеристики и возможности.

В PHP 5 появился новый оператор **instanceof**, при помощи которого можно проверить тип объекта. Оператор **instanceof** принимает два параметра. Первый - это тестируемый объект, второй - тип, на принадлежность к которому проверяется объект.

Использование оператора **instanceof** с классами

Оператор **instanceof** используется для определения того, является ли текущий объект экземпляром определенного класса

```
class Country {    …      }
class myCountry {    …      }
obj1=new Country();
if ($obj1  instanceof  Country) {
echo  “\ $obj1  - объект класса Country”;
}
// или
var_dump($obj1   instanceof  Country);      // bool(true)
var_dump($obj1   instanceof  myCountry);  // bool(false)
var_dump(!($obj1 instanceof  myCountry));// bool(true)
```

Использование оператора **instanceof** с наследуемыми классами

Оператор **instanceof** используется для определения того, является ли текущий объект экземпляром класса наследника

```
class City extends Country {  …  }
obj2=new City();
if ($obj2  instanceof  Country) {
echo “\ $obj2  - объект класса,  производного от Country”;
}
// или
 var_dump($obj2   instanceof  City);      // bool(true)
 var_dump($obj2   instanceof  Country);     // bool(true)
```

var_dump() - функция отображает структурированную информацию об одном или нескольких выражениях, включая их тип и значение.

Использование instanceof с интерфейсами

instanceof может быть также использован для проверки реализации объектом некоторогоинтерфейса:

```
<?php
interface MyInterface
{
}
class MyClass implements MyIn
terface
{
}
$a = new MyClass;
var_dump($a instanceof MyClas
s);
var_dump($a instanceof MyInte
rface);
?>
```

Результат:

bool(true)

bool(true)

Абстрактные классы

Представим, что у нас есть несколько классов со схожим поведением. У них есть идентичные реализации методов и свойств. Чтобы не «раздувать» программный код и избавиться от его повторяемости, можно применить наследование классов.

В PHP для построения более понятной программной архитектуры существуют **абстрактные классы**. От них нельзя создавать объекты, что делает их идеальным хранилищем наследуемого функционала.

Объявление абстрактного класса начинается с ключевого слова **abstract**.

Абстрактные классы могут содержать описание абстрактных методов. Для таких методов указывается лишь заголовок с ключевым словом **abstract** и всеми прочими атрибутами, указываемыми при объявлении метода.

Абстрактные методы не имеют тела или реализации, они лишь описывают, что должен уметь делать объект, а как он это будет делать – проблема наследников абстрактного класса.

Экземпляр абстрактного класса создавать нельзя, так как в противном случае могла произойти попытка вызвать от этого экземпляра абстрактный метод, что абсурдно, так как он не имеет

Абстрактные классы

Сделаем наш класс *Animal* и его метод *say* абстрактными. Теперь класс, унаследованный от класса *Animal*, обязан будет содержать реализацию метода *say* или должен быть объявлен абстрактным, в противном случае ещё до начала выполнения скрипта произойдёт ошибка.

В абстрактном классе можно объявлять так же и обычные методы и поля, которые могут быть унаследованы.

Несмотря на то, что абстрактный класс не может иметь экземпляров, он может иметь конструктор, который могут использовать для инициализации его полей потомки.

Добавим свойство $name, хранящее кличку животного, метод getName() для её получения и конструктор в класс *Animal*.

```
abstract class Animal {
  private $name;

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

  abstract public function say();

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

Абстрактные классы

Теперь унаследуем от нашего абстрактного класса *Animal* два класса: класс *Cat* и класс *Dog*, и добавим в них, реализации метода *Say*. Обратите внимание, что в конструкторах этих классов вызывается конструктор абстрактного класса-предка.

```php
class Cat extends Animal {

  public function __construct($name) {
    parent::__construct($name);
  }

  public function say() {
    echo "meow-meow";
  }
}

class Dog extends Animal {

  public function __construct($name) {
    parent::__construct($name);
  }

  public function say() {
    echo "woof-woof!";
  }
}
```

Абстрактные классы

Создать объект, являющийся экземпляром абстрактного класса, невозможно. Но вот после того, как в классе наследнике абстрактный метод будет переопределен, можно уже создавать экземпляры этого класса-наследника:

```
$obj = new Dog;
$obj -> say();
$obj1 = new Cat;
$obj1 -> say();
```

Полиморфизм классов в PHP

Полиморфизм - это следствие наследования. Это свойство *унаследованных классов* иметь одинаковые методы, которые будут работать по-разному в контексте *объектов*. Например, у нас есть класс фигура и классы квадрат, треугольник и трапеция - унаследованные от класса фигуры. Каждая фигура содержит функционал для вычисления площади, но у каждой фигуры он свой.

В более общем смысле, концепцией полиморфизма является идея «*один интерфейс, множество методов*». Это означает, что можно создать общий интерфейс для группы близких по смыслу действий. Преимуществом полиморфизма является то, что он помогает снижать сложность программ, разрешая использование того же интерфейса для задания единого класса действий.

Интерфейсы

В PHP невозможно описать класс, являющийся наследником сразу двух классов, даже абстрактных. Для решения этой проблемы существуют интерфейсы, представляющие собой абстрактные классы, не содержащие ни одного неабстрактного метода.

Класс может наследоваться от двух интерфейсов одновременно, переопределяя их методы. Можно создавать объекты – экземпляры такого класса-наследника.

Для объявления интерфейса используется ключевое слово interface.

В отличие от абстрактных классов про интерфейсы чаще говорят, что классы их не наследуют, а имплементируют или реализуют. Для реализации интерфейса используется оператор implements.

Интерфейсы

Пусть класс Hero, будет имплементировать интерфейсы Voin *и* Artist.

```
interface Voin {
    function shoot ()
}
interface Artist {
    function paint ()
}

class Hero implements Voin, Artist {
    public function shoot () {
      echo “Герой умеет стрелять. “;
    }
    public function paint () {
      echo “Герой умеет рисовать. “;
    }
}
$obj = new Hero;
$obj -> shoot();
$obj -> paint();
```

Финальные методы и классы

Финальный 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**.

Кстати, ключевое слово **final** должно стоять **перед** любыми другими **модификаторами**, такими как, например, *protected* или *static*.

Абстрактные методы не могут быть финальными, потому что они должны быть обязательно переопределены.

Финальные классы

Ключевое слово final перед определением класса означает, что этот класс не может иметь дочерних классов - т.е. является финальным в смысле наследования.

```
final class MyClass {
    ......................
}
```

// Выполнение следующего кода приведет к ошибке: PHP Fatal error: class MyClass1 may not inherit from final class (MyClass) in …

// По-русски это звучит примерно так: Класс MyClass1 не может быть унаследован от финального класса MyClass.

```
class MyClass1 extends MyClass {
    ..................
}
```

Если класс определен как final, то и все методы данного класса автоматически становятся финальными, таким образом, определять их явно как final уже нет необходимости.

Определять свойства класса как финальные – недопустимо.

Финальные методы

Метод, при определении которого используется ключевое слово final, не может быть переопределен в классах, производных от данного класса.

```
class MyClass {

        final public function func() {
            ……..
        }
    }
// Теперь мы можем создать подкласс MyClass1 класса  MyClass

    class MyClass1 extends MyClass {

     public function func() {
            ……..
        }
    }
// Однако любая попытка переопределить метод func() приведет к
    неустранимой ошибке:        Fatal error: cannot override final method
    MyClass ::func() in ...
// Что в переводе на русский означает:      Нельзя переопределить
    финальный метод MyClass ::func()
```

Итераторы объектов

Как известно, важнейшим элементом в языке программирования PHP являются массивы. Самым главным свойством массивов - является возможность перебора (итерации) всех элементов, для получения либо ключа, либо значения, либо того и другого сразу.

Фактически процесс итерации обеспечивает возможность одинаково работать с любым итерируемым (перебираемым) объектом, а не только с массивом.

PHP 5 предоставляет такой способ объявления объектов, который дает возможность пройти по списку элементов данного объекта, например, с помощью оператора **foreach**. По умолчанию, в этом обходе (итерации) будут участвовать все видимые (общедоступные) свойства объекта.

Итераторы объектов. Примеры

```
class MyClass
{
    // общедоступное свойство
    public $public = "Public Property";

    // защищенное свойство
    protected $protected = "Protected Property";


    // закрытое свойство
      private $private = "Private Property";
}

$myclass = new MyClass();
      foreach( $myclass as $prop )
      {
      echo $prop, PHP_EOL;
      }
// Напечатает:    Public Property
```

```
class Mammal
{
      public $blood, $legs;
      public function  __construct($name)
      {
        $this -> name = $name;
        $this -> blood = “теплая”;
        $this -> legs = 4;
      }
}
$Murka =  new Mammal(“Кошка”);
foreach ($Murka as $key => $value)
{
      echo “$key => $value\n”;
}
```

Как видно из примеров, мы можем пройтись по объекту словно по массиву. Но напечатаны будут только значения общедоступных свойств , так как защищенные и закрытые свойства класса не могут быть прочитаны в цикле.

Итераторы объектов

Таким образом, мы можем итерировать объекты в PHP, так как будто, они являются массивами.

Итераторы в PHP - это очень мощная технология, и в стандартной библиотеке PHP есть десятки уже готовых итераторов, которые позволяют буквально в пару строк кода выполнить далеко нетривиальные задачи, такие как, например, рекурсивный обход каталога, фильтрация списка на основании некоторого условия или регулярного выражения и т.д.

Автозагрузка класса

Разрабатывая web-приложение с использованием ООП, очень часто приходится подключать классы, используя инструкции require_once и include_once. При этом, как правило, каждый скрипт начинается длинным списком подключаемых классов.

Начиная с версии PHP 5, появился механизм автозагрузки классов и метод __autoload().

Этот магический метод предназначен для того, чтобы избавить нас от бесконечных include и require в коде сайта.

Магический метод __autoload

Метод __autoload() принимает один параметр – имя класса и подключает файл с определением нужного класса. Метод вызывается каждый раз, во время создания объекта или обращения к экземпляру класса.

```
function __autoload($classname)
{
   $filename = "./". $classname .".php";
   require_once($filename);
}
$a = new A();
```

```
function  __autoload($class)
{
    include  "src/$class.”.php";
}
$Barsik = new Mammal('Кошка');
```

При этом подходе видно, что в одном файле должен находится один класс, а название класса должно совпадать с названием файла.

Замечание: При создании классов и файлов надо помнить о чувствительности к регистру в именах файлов.

Автозагрузка класса

Метод __autoload() используется для автоматической загрузки классов и интерфейсов, но он имеет ряд недостатков:

- нет возможности регистрации нескольких автозагрузчиков,
- нет возможности динамически активировать/деактивировать автозагрузчики.

Поэтому данный метод объявлен УСТАРЕВШИМ начиная с PHP 7.2.0 и его использование не рекомендовано.

Для решения этих проблем в PHP 5.1.2 появился ряд SPL функций.

Предпочтение следует отдать функции spl_autoload_register()
Предпочтение следует отдать функции spl_autoload_register(), потому, что она предоставляет гораздо более гибкую альтернативу функции __autoload(), позволяя регистрировать необходимое количество автозагрузчиков, например, для сторонних библиотек.

spl_autoload_register()

Данная функция позволяет регистрировать любую переданную ей функцию как реализацию механизма автозагрузки классов.

Этот механизм имеет свою очередь, поэтому можно регистрировать более одной функции для разрешения вопросов автозагрузки классов. Все функции будут вызваны в порядке их регистрации.

В случае обращения к несуществующему в данный момент классу, PHP будет вызывать по очереди все зарегистрированные автозагрузчики, передавая им имя класса. Если автозагрузчик знает, где лежит этот класс, он должен подключить файл с ним, PHP увидит, что класс появился, и продолжит выполнение программы. Иначе PHP вызовет следующий автозагрузчик. Если ни один автозагрузчик не подключит файл с классом, то будет выведена ошибка об обращении к несуществующему классу.

Пример spl_autoload_register()

```
function libraryOne($classname) {
    $filename = "./path/one/". $classname .".php";
    require_once($filename);
}

function libraryTwo($classname) {
    $filename = "../../path/two/". $classname .".php";
    require_once($filename);
}

// регистрация
spl_autoload_register('libraryOne');
spl_autoload_register('libraryTwo');


...
// деактивация первой библиотеки
spl_autoload_unregister('libraryOne');
```

Если подключается сторонняя библиотека, то она может зарегистрировать свой автозагрузчик для загрузки своих классов. Таким образом, каждая библиотека может устанавливать свои правила для поиска файлов со своими классами.

SPL - функции

spl_autoload_call — принудительно загружает класс по его имени, используя все доступные в системе автозагрузчики;

spl_autoload_extensions — возвращает/модифицирует расширения файлов, из которых происходит загрузка неинициализированных классов;

spl_autoload_functions — возвращает список всех зарегистрированных автозагрузчиков в системе;

spl_autoload_register — регистрация собственного автозагрузчика в стеке автозагрузки;

spl_autoload_unregister — удаление автозагрузчика из стека автозагрузки;

spl_autoload — основная функция автоматической загрузки классов. Именно она вызывается при обращении к классу, который еще не инициализирован. Данная функция активирует все автоматические загрузчики из стека в порядке их добавления.

Обработка исключений (ошибок) в ООП

Исключение — это некий участок кода, где может произойти ошибка. Ошибка на языке ООП — это и есть исключение.

Эта ошибка не синтаксическая, а скорее логическая. И мы при программировании приложения, заранее можем знать, где она произойдет.

Пример:

```
$a = 1;
$b = 0;
echo $a/$b;
```

В процедурном программировании мы бы написали условие if(){} else{}, и если бы произошло деление на ноль, мы бы дали знать об этом пользователю.

В объектно-ориентированном программировании, мы заключаем данный код в конструкцию try{} catch(exception $e){}:

```
try{
        $a = 1;
        $b = 0;
        echo $a/$b;
}
catch (exception $e)
{ …. }
```

Встроенный класс Exception

Для обработки ошибок в PHP5 встроен **класс Exception**. Он реализует методы, предоставляющие ценную информацию о процессе отладки, в которой содержатся сведения о возникшей ошибке.

```
class Exception {
    protected $message;              // сообщение об ошибке;
    protected $code;            // код ошибки, определяемый пользователем;
    protected $file;            // имя файла, в котором произошла ошибка;
    protected $line;            // номер строки, где произошла ошибка;
    function __construct( $message = "", $code = 0 );
    function __toString();           // создание строки для отображения на экране;
    final function getFile();
    final function getLine();
    final function getMessage();
    final function getCode();
    final function getTrace();
    final function getTraceAsString();
    }
```

В PHP реализована схема обработки исключений с помощью конструкции **try … catch … throw**. Она позволяет весь код обработки ошибок локализовать в одном месте сценария. Код при выполнении которого может появиться ошибка, пишется внутри блока **try**.

Конструкция try … catch … throw

Мы знаем, что в блоке **try{ }** может произойти ошибка. Поэтому мы генерируем исключение следующей конструкцией:

```
try {
        $a = 1;
        $b = 0;
        if($b == 0) throw new   Exception("Деление на 0 запрещено!");
        echo $a/$b;
        }

catch (Exception $e) {
        echo «Ошибка в строке », $e -> getLine();
        echo $e -> getMessage();
}
```

Пишем код, чтобы при возникновении ошибки создавался новый **экземпляр $e класса Exception** оператором **new**. Экземпляр класса, с помощью слова **throw** создается как бы «на лету», то есть не присваивается переменной.

Передадим в конструктор класса Exception в качестве значения свойства **$message** сообщение об исключении ***"Деление на 0 запрещено!".***

Если ошибка возникнет, то она будет перехвачена конструкцией **catch**, которая напечатает ***«Ошибка в строке »,*** затем вызовет метод **getLine()** для объекта **$e** и метод **getMessage()**.