PHP: Массивы

Массив (array) - это переменная специального типа, хранящая много элементов данных. Массив позволяет обратиться отдельно к любому из составляющих его элементов (поскольку внутри массива они хранятся отдельно), а также есть возможность копировать или обрабатывать массив целиком.

Массивы PHP нетипизированы, это означает, что элементы массива могут иметь любой тип, причем разные элементы в массиве могут иметь различные типы. Помимо этого массивы PHP являются динамическими, это означает, что фиксированный размер объявлять не нужно и новые элементы можно добавлять в любое время.

Основные сведения о массивах

Чтобы работать с массивами, вам нужно освоить два новых понятия: элементы и индексы. Элементы - это значения хранящиеся в массиве, значения могут быть абсолютно любого типа. К каждому элементу можно обратиться по его уникальному индексу. В качестве индекса может использоваться целое число или строка.

Массивы можно разделить на два типа: индексные, у которых в качестве значения индекса используется только целое число и ассоциативные, где значением индекса может быть как строка так и число. Часто в ассоциативных массивах индекс называется: «ключ».

Индексные массивы обычно называют просто «массивами», а ассоциативные массивы - «хешами», «ассоциативными» или «словарями».

Создание массива

В PHP есть три способа создания массивов. Первый способ - это создание с помощью специальной функции array(). В качестве аргументов функция принимает любое количество пар ключ => значение (key => value) разделенных запятыми или просто значения, также разделяемые запятыми. Она возвращает массив, который можно присвоить переменной.

```php
<?php

  // Создание массива с числовыми индексами
  $weekdays = array('Понедельник','Вторник','Среда',
                    'Четверг','Пятница','Суббота',
                    'Воскресенье');

?>
```

Так как указывать ключ не обязательно, значения можно добавлять в массив без его указания. Если ключ не указывается, PHP будет использовать числовые индексы. По умолчанию элементы будут нумероваться, начиная с нуля. Массивы с числовыми индексами позволяют просто добавить элемент, а PHP автоматически будет использовать предыдущее наибольшее значение ключа типа integer, увеличенное на 1.

Также можно указывать ключ для отдельных элементов:

```php
<?php
```

```php

    $my_array = array( 'a',
                       'b',
                   7 => 'c',
                       'd');
    var_dump($my_array);

?>
```

Запустив данный пример, вы можете заметить, что последний элемент ('d') был присвоен ключу **8**. Так получилось, потому что самое большое значение ключа целого типа перед ним было **7**.

Теперь рассмотрим создание ассоциативного массива с помощью функции array(). Ассоциативный массив записывается немного по другому: для добавления элемента используется формат ключ => значение.

```php
<?php

  // Создание ассоциативного массива
  $shapes = array('Январь'   => '30',
                  'Февраль'  => '28/29 (29 бывает каждые четыре года)',
                  'Март'     => '31',
                  'Апрель'   => '30',
                  'Май'      => '31',
                  'Июнь'     => '30',
                  'Июль'     => '31',
                  'Август'   => '31',
                  'Сентябрь' => '30',
                  'Октябрь'  => '31',
                  'Ноябрь'   => '30',
                  'Декабрь'  => '31');

?>
```

С отступами, которые вы видите в этом примере, легче добавлять элементы в массив, чем когда они записаны в одну строку.

Теперь рассмотрим второй способ создания массива: использование квадратных скобок [], вместо специальной функции array():

```php
<?php

  $my_array = array('foo' => 'bar',
                    'bar' => 'foo');

  // другой способ создания массива
  $my_array = ['foo' => 'bar',
               'bar' => 'foo'];

?>
```

Разницы между этими массивами никакой нет, кроме различия в написании.

Обратите вниманием, в PHP массивы могут содержать ключи типов int и string одновременно, т.е. PHP не делает различия между индексированными и ассоциативными массивами.

```php
<?php

  $my_array = ['Солнце' => 'яркое',
               'колесо' => 'круглое',
                10      => 'дом',
                -5      =>  290];

?>
```

> **Примечание:** выбирая имя для массива, будьте внимательны, чтобы не использовать имя, совпадающее с именем другой переменной, так как они разделяют общее пространство имен. Создание переменной с тем же именем, что и у существующего массива, приведет к удалению массива без вывода каких-либо предупреждений.

Третий способ создания массивов будет рассмотрен в разделе «Добавление и удаление элементов массива».

Преобразование индексов

Как упоминалось в самом начале главы, ключ может быть одним из двух типов: string или integer. Поэтому ключи несоответствующие одному из этих типов будут преобразованы:

- Если в качестве ключа выступает строка, которая содержит число, то она будет преобразована к типу integer. Однако, если число является некорректным десятичным целым, например '09', то оно не будет преобразовано в тип integer.
- Вещественное число (float), также будет преобразовано в integer - дробная часть в этом случае отбрасывается. Например, если значение ключа 5.4, оно будет интерпретировано как 5.
- Булев тип (bool) также будет преобразован в integer. Например, если значение ключа true, то оно будет преобразовано в 1, а ключ со значением false соответственно будет преобразован в 0.
- Если используется тип null, он будет преобразован в пустую строку.
- Объекты и массивы не могут быть использованы в качестве ключей.

Если в объявлении массива несколько элементов используют одинаковый ключ, то использоваться будет только последний из них, а все другие будут перезаписаны.

```php
<?php

  $my_array = array( 1    => 'a',
                    '1'   => 'b',  // ключи преобразуются в число 1
                     1.5  => 'c',
                     true => 'd');

  var_dump($my_array);

?>
```

В приведенном примере все ключи будут преобразованы в единицу, основываясь на этом, массив будет содержать всего один элемент, содержание которого будет перезаписано 3 раза, в итоге, его значением станет 'd'.

Доступ к элементам массива

Доступ к элементам массива осуществляется с помощью квадратных скобок в которых указывается индекс/ключ: **array[index/key]**.

```php
<?php

$my_array = array('Шоколад' => 'молочный',
                  2         => 'foo');

echo $my_array['Шоколад'], "<br>";
echo $my_array[2];

?>
```

Еще один способ доступа к элементам массива заключается в использовании прямого разыменования массива.

```php
<?php

  function foo() {
    return array(1, 'hello world!', 3);
  }

  echo foo()[1]; // => hello world!

?>
```

Данный пример показывает, что можно обращаться к индексу массива, возвращаемого в качестве результата вызова функции или метода.

Добавление и удаление элементов массива

Теперь, когда вы получили основные понятия о массивах, рассмотрим способы записи значений в массив. Существующий массив может быть изменен явной установкой в нем значений. Это выполняется с помощью присваивания значений массиву.

Операция присваивания значения элементу массива выглядит так же, как операция присваивания значения переменной, за исключением квадратных скобок ([]), которые добавляются после имени переменной массива. В квадратных скобках указывается индекс/ключ элемента. Если индекс/ключ не указан, PHP автоматически выберет наименьший незанятый числовой индекс.

```php
<?php

  $my_arr = array( 0 => 'ноль',
                   1 => 'один');

  $my_arr[2] = 'два';
  $my_arr[3] = 'три';

  var_dump($my_arr);

  // присваивание без указания индекса/ключа
  $my_arr[] = 'четыре';
  $my_arr[] = 'пять';

```

```php
  echo "<br>";
  var_dump($my_arr);

?>
```

Для изменения определенного значения, нужно просто присвоить новое значение уже существующему элементу. Чтобы удалить какой-либо элемент массива с его индексом/ключом или удалить полностью сам массив, используется функция unset():

```php
<?php

  $my_arr = array(10, 15, 20);

  $my_arr[0] = 'радуга'; // изменяем значение первого элемента

  unset($my_arr[1]);     // Удаляем полностью второй элемент (ключ/значен

  var_dump($my_arr);

  unset($my_arr);           // Полностью удалили массив

?>
```

> **Примечание:** как уже упоминалось выше, если элемент добавляется в массив без указания ключа, PHP автоматически будет использовать предыдущее наибольшее значение ключа типа integer, увеличенное на 1. Если целочисленных индексов в массиве еще нет, то ключом будет 0 (ноль).

Учтите, что наибольшее целое значение ключа **не обязательно существует в массиве в данный момент**, такое может быть по причине удаления элементов массива. После того как были удалены элементы, переиндексация массива не происходит. Приведем следующий пример, чтобы стало понятнее:

```php
<?php

  // Создаем простой массив с числовыми индексами.
  $my_arr = array(1, 2, 3);
  print_r($my_arr);

  // Теперь удаляем все элементы, но сам массив оставляем нетронутым:
  unset($my_arr[0]);
  unset($my_arr[1]);
  unset($my_arr[2]);

  echo "<br>";
  print_r($my_arr);

  // Добавляем элемент (обратите внимание, что новым ключом будет 3, вмес
  $my_arr[] = 6;

  echo "<br>";
  print_r($my_arr);

  // Делаем переиндексацию:
```

```php
  $my_arr = array_values($my_arr);
  $my_arr[] = 7;

  echo "<br>";
  print_r($my_arr);

?>
```

В этом примере использовались две новые функции, print_r() и array_values(). Функция array_values() возвращает индексированный массив (заново индексирует возвращаемый массив числовыми индексами), а функция print_r работает наподобие var_dump, но выводит массивы в более удобочитаемом виде.

Теперь мы можем рассмотреть третий способ создания массивов:

```php
<?php

  // следующая запись создает массив
  $weekdays[] = 'Понедельник';
  $weekdays[] = 'Вторник';

  // тоже самое, но с указанием индекса
  $weekdays[0] = 'Понедельник';
  $weekdays[1] = 'Вторник';

?>
```

В примере был показан третий способ создания массива. Если массив $weekdays еще не был создан, то он будет создан. Однако такой вид создания массива не рекомендуется применять, так как если переменная $weekdays уже была ранее создана и содержит значение, это может привести к неожиданным результатам работы сценария.

Если у вас возникают сомнения по поводу того, является ли переменная массивом, воспользуйтесь функцией *is_array*. Например, проверку можно выполнить следующим образом:

```php
<?php

  $yes = array('это', 'массив');
  echo is_array($yes) ? 'Массив' : 'Не массив';
  echo '<br>';

  $no = 'обычная строка';
  echo is_array($no) ? 'Массив' : 'Не массив';

?>
```

Обход массива в цикле

Оператор цикла foreach осуществляет последовательный перебор всех элементов массива. Он работает только с массивами и объектами, а в случае его использования с переменными других типов или неинициализированными переменными будет сгенерирована ошибка. Есть два вида синтаксиса для данного цикла. Первый вид синтаксиса выглядит следующим образом:

```php
foreach ($array as $value) {
  инструкции
}
```

Цикл будет перебирать заданный массив - $array (вместо $array подставляется название массива). На каждой итерации значение текущего элемента присваивается переменной $value (можно указать любое другое имя переменной). Оператор цикла foreach очень удобен, поскольку сам выполняет обход и чтение всех элементов массива, пока не будет достигнут последний. Он позволяет не держать постоянно в памяти тот факт, что индексация массивов начинается с нуля, и никогда не выходит за пределы массива, что делает конструкцию цикла очень удобной и помогает избежать распространенных ошибок. Посмотрим, как он работает на примере:

```php
<?php

  $my_arr = array(1, 2, 3, 4, 5);

  foreach ($my_arr as $value) {
    echo $value, " ";
}

?>
```

Второй вид синтаксиса foreach, выглядит так:

```php
foreach ($array as $key => $value) {
  инструкции
}
```

При использовании данной формы синтаксиса на каждой итерации дополнительно присваивается значение текущего ключа переменной $key (можно указать любое другое имя переменной):

```php
<?php

  $my_arr = array(1, 2, 3, 4, 5);

  foreach ($my_arr as $key => $value) {
    echo "[$key] => ", $value, "<br>";
}

?>
```

Чтобы можно было напрямую изменять элементы массива внутри цикла, нужно использовать ссылку. В этом случае значение будет присвоено по ссылке.

```php
<?php

  $my_arr = array(1, 2, 3);
  foreach ($my_arr as &$value) {
    $value *= 2;
    echo $value;
  }

  /* это нужно для того, чтобы последующие записи в
  переменную $value не меняли последний элемент массива */
```

```php
  unset($value); // разорвать ссылку на последний элемент

?>
```

Примечание: Ссылка на последний элемент массива остается даже после того, как оператор foreach завершил работу. Поэтому рекомендуется удалять ее с помощью функции unset() как показано в примере выше. Давайте посмотрим что будет, если не использовать unset():

```php
<?php

  $numbers = array(1,2,3,4,5);

  foreach ($numbers as &$num) {
    echo $num, " ";
  }

  // Присваиваем новое значение переменной $num
  $num = '100';
  echo '<br>';

  foreach ($numbers as &$num) {
    echo $num, " ";
  }

?>
```

Стоит отметить следующий момент, ссылку можно использовать только если перебираемый массив является переменной. Следующий код не будет работать:

```php
<?php

  foreach (array(1, 2, 3) as &$value) {
    $value *= 2;
  }

?>
```

PHP: Функции

Функция - это именованный блок кода, в данном случае на языке PHP, который определяется один раз, а затем может вызываться на исполнение сколько угодно раз.

Примеры функций (встроенных в PHP) вы уже видели - это var_dump(), unset(). В этой главе вы узнаете, как создавать свои функции.

Определение и вызов

Определение функции выполняется с помощью ключевого слова function за которым указываются следующие компоненты:

- Идентификатор, определяющий имя функции. Он будет использован для создания новой переменной, которой будет присвоена функция. Имена функций следуют тем же правилам, что и именование переменных. **Замечание**: имена функций не чувствительны к регистру букв.
- Пара круглых скобок вокруг списка из нуля или более идентификаторов, разделенных запятыми. Эти идентификаторы будут определять имена параметров функции. В теле функции они используются в качестве локальных переменных.
- Пара фигурных скобок с нулем или более инструкций. Эти инструкции составляют тело функции: они выполняются при каждом вызове функции.

Синтаксис определения функции выглядит следующим образом:

```
function имя_функции([параметры]) { блок кода }
```

Квадратные скобки ([]) означают необязательность. Теперь приведем простой пример определения функции:

```
<?php

  function hello() {
    echo "Привет из функции";
  }

?>
```

Для вызова функции используется оператор вызова, представляющий из себя пару круглых скобок. Завершается вызов функции как и все инструкции точкой с запятой (;):

```
<?php

  function hello() {
    echo "Привет из функции";
  }

  // вызов функции
  hello();

?>
```

При вызове функции исполняются инструкции расположенные в ее теле.

Функции допускается определять в любом месте программы, это значит, что они не обязаны быть определены до их использования, **исключая** тот случай, когда функции определяются условно:

```php
<?php

  $makefunc = true;

  /* Здесь нельзя вызвать функцию foo(), так как
    она еще не определена (оператор if еще не сработал).
    Но мы можем вызвать здесь функцию bar(),
    так как она определена уже снизу */

  bar();  // вызываем так как функция уже определена

  if ($makefunc) {
    // функция foo определяется только сейчас
    function foo() {
      echo "Я не существую до тех пор, пока выполнение программы меня не
    }
  }

  /* Теперь мы можем вызывать функцию foo(),
   поскольку $makefoo была интерпретирована как true */

  foo();

  function bar()  {
    echo "Я существую сразу с начала старта программы.\n";
  }

?>
```

Все функции в PHP имеют глобальную область видимости - они могут быть вызваны вне функции, даже если были определены внутри и наоборот:

```php
<?php

  function func1() {
    function func2() {
    echo "Я не существую пока не будет вызвана func1().\n";
    }
  }

  /* пока нельзя вызвать func2(),
   поскольку она еще не определена. */

  func1();

  /* Теперь можно вызвать функцию func2(),
   вызов func1() сделал ее доступной. */

  func2();

?>
```

Чаще всего функции работают с каким-либо переданными ей значениями. Для того, чтобы функции можно было передать некоторые значения, в ней должны

быть указаны параметры. Эти параметры по сути являются обычными переменными, которые инициализируются переданными значениями при вызове функции.

Аргументы и параметры

Параметры указываются в определении функции, внутри круглых скобок, и являются ее локальными переменными, т.е. они видны только в ее теле, если параметров несколько, то они указываются через запятую. При вызове функция может получать аргументы, с помощью которых инициализируются параметры.

Что такое параметры мы рассмотрели, теперь узнаем о том, какими значениями они инициализируются. Значения, которые будут присвоены параметрам называются аргументами - это может быть например строковой или целочисленный литерал, переменная или какое-нибудь более сложное выражение состоящее из переменных и операторов, но которое может быть вычислено интерпретатором PHP для получения значения, которым будет инициализирован параметр. Проще говоря, аргумент - это переданное функции значение:

```php
<?php

  // Определение функции
  function my_car($car, $color) {  // Указано два параметра: $car и $colo
    echo "Марка моей машины: $car и она имеет $color цвет";
  }

  $color = 'красный';

  // Вызываем функцию и передаем ей два аргумента
  my_car('BMW', $color);  // Аргументы - строковой литерал и переменная

?>
```

Передача аргументов

PHP поддерживает два способа передачи аргументов функции. Первый - передача аргументов по значению (работает по умолчанию), второй - передача аргументов по ссылке. Также PHP поддерживает значения по умолчанию. Давайте теперь рассмотрим все три варианта подробнее.

По умолчанию аргументы передаются в функцию по значению (это значит, если вы измените значение параметра внутри функции, то вне ее переданное значение останется прежним):

```php
<?php

  function foo($color) {
    $color = 'синий';    // изменили значение параметра
    echo "Внутри функции параметр \$color имеет значение: $color";
  }

  $color = 'красный';

  foo($color);                // Аргумент - значение переменной $color
```

```php

  echo "<br>$color цвет";    // Значение переменной не изменилось

?>
```

Если необходимо разрешить функции изменять переданные аргументы за ее пределами, вы должны передавать их по ссылке. Для того, чтобы аргумент был передан по ссылке, необходимо указать знак & (амперсанд) перед именем параметра в определении функции:

```php
<?php

  function foo(&$my_color) {  // теперь параметр будет ссылаться на ориги
    $my_color = 'синий';      // присваиваем новое значение
  }

  $color = 'красный';

  foo($color);

  echo $color;    // выведет: синий

?>
```

Функции могут определять значения аргументов по умолчанию. Чтобы установить значение по умолчанию, в определении функции нужно всего лишь присвоить параметру желаемое значение:

```php
<?php

  function tea($str = 'зеленый') {
    return "В чашке $str чай<br>\n";
  }

  echo tea();          // выведет значение по умолчанию
  echo tea('черный');

?>
```

Примечание: все параметры, для которых установлены значения аргументов по умолчанию, должны находиться правее аргументов, для которых значения по умолчанию не заданы, так как в противном случае ваш код может работать не так, как вы того ожидали:

```php
<?php

  // данный пример вызовет ошибку
  function my_car($car = 'Mazda', $color) {
    echo "Марка моей машины: $car и она имеет $color цвет";
  }

  my_car('красный');  // Не будет работать так, как мы могли бы

  // корректный пример
  function my_car($color, $car = 'Mazda') {
```

```
13      echo "Марка моей машины: $car и она имеет $color цвет";
14    }
15
16    my_car('красный');
17
18  ?>
```

Значение, возвращаемое функцией

Когда выполнение функции завершается, она может возвратить некоторое значение (результат работы функции) программе, которая ее вызвала. Оператор return внутри функций служит для определения значения, возвращаемого функцией. В качестве возвращаемого значения может быть любой тип. Он имеет следующий синтаксис:

```
1  return выражение;
```

Оператор return может быть расположен в любом месте функции. Когда до него доходит управление, функция возвращает значение (если указано) и завершает свое выполнение. Если оператор return не указан или не указано возвращаемое значение, то функция вернет значение NULL. Для использования возвращаемого значения, результат выполнения функции можно присвоить к примеру переменной:

```
1  <?php
2
3    function sqr($num) {
4      return $num * $num;
5    }
6
7    $x = sqr(4);
8
9    echo "$x <br>";   // => 16.
10
11
12   function foo($num) {
13     if($num === 10)
14       return "$num равно 10";
15     else
16       return "$num не равно 10";
17
18     echo 'hello';  // эта строка кода никогда не выполнится
19   }
20
21   echo foo(6);
22
23 ?>
```

Обращение к функциям через переменные

Функцию можно присвоить переменной, так же как и обычное значение. Для этого имя функции должно быть присвоено переменной в виде строки, но без указания круглых скобок:

```
1  <?php
2
```

```php
  function foo() {
    echo "функция foo()<br>\n";
  }


  $my_func = 'foo';

  // Теперь мы можем запустить функцию foo() при помощи переменной $my_fu
  // которая хранит имя указанной функции в виде строки
  $my_func();        // Вызываем функцию foo()

?>
```

Такая концепция PHP имеет название «переменные-функции». Она заключается в том, что если добавить к переменной в конце круглые скобки, то интерпретатор PHP проверит сначала, не существует ли функции с именем равным **значению** переменной и если такая функция есть - выполнит ее.

Так, как показано в примере выше, можно делать только с функциями определенными пользователями. Встроенные языковые конструкции и функции, такие как echo, unset(), isset() и другие подобные им нельзя таким же образом напрямую присвоить переменным. Но можно сделать свою функцию-обертку (wrapper) для того, чтобы встроенные языковые конструкции могли работать подобно пользовательским функциям.

```php
<?php

  // Функция-обертка для echo
  function foo($str) {
    echo $str;
  }


  $my_func = 'foo';
  $my_func('test');  // Вызывает функцию foo()

?>
```

Анонимные функции

PHP позволяет определять анонимные функции. Такое название произошло за счет того, что такие функции создаются без определенного имени, иногда можно встретить и другое название таких функций - лямбда функция. Они определяются и присваиваются переменным, как обычные значения:

```php
<?php

  $my_func = function($str) {
    echo "hello $str";
  };    // точка с запятой обязательна

  $my_func('World!');

?>
```

Обратите внимание на пример, в конце определения функции есть точка с запятой, так как анонимная функция является по своей сути значением, и мы

присваиваем значение переменной, то в конце как и для обычных инструкций ставится точка с запятой.

Анонимные функции отличаются от именованных тем, что создаются только в тот момент, когда до них доходит выполнение, поэтому воспользоваться ими можно только после их определения:

```php
<?php

  // этот пример вызовет ошибку
  $my_func('World!');

  $my_func = function($str) {
    echo "hello $str";
  };

?>
```

PHP: Классы и объекты

Поскольку именно классы описывают объекты, мы начнем описание с определения классов.

Определение класса

Класс - это шаблон кода, который используется для создания объектов. Класс определяется с помощью ключевого слова **class** после которого указывается произвольное имя класса. В имени класса может использоваться любое сочетание букв и цифр, но они не должны начинаться с цифры. Код, связанный с классом должен быть заключен в фигурные скобки, которые указываются после имени. Определение класса описывает, какие элементы будут содержаться в каждом новом экземпляре этого класса. На основе полученных данных давайте посмотрим синтаксис определения класса на примере:

```php
<?php

  class first {
    // Тело класса
  }

?>
```

Класс first из приведенного примера - уже полноправный класс, хотя пока и не слишком полезный. Но тем не менее мы сделали нечто очень важное. Мы определили тип, т.е. создали категорию данных, которые мы можем использовать в своих сценариях. Важность этого станет для вас очевидной по мере дальнейшего чтения главы.

Создание объекта

Так как класс - это шаблон для создания объектов, следовательно, объект - это данные, которые создаются и структурируются в соответствии с шаблоном, определенным в классе. Объект также называют экземпляром класса, тип которого определяется классом. Для создания нового экземпляра класса нам понадобится оператор **new**. Он используется совместно с именем класса следующим образом:

```php
<?php

  // Ключевое слово new сообщает интерпретатору PHP о необходимости
  // создать новый экземпляр класса first
  $obj1 = new first();
  $obj2 = new first();

?>
```

После оператора new указывается имя класса на основе которого будет создан объект. Оператор new создает экземпляр класса и возвращает ссылку на вновь созданный объект. Эта ссылка сохраняется в переменной соответствующего типа. В результате выполнения этого кода будет создано два объект типа first. Хотя функционально они идентичны (т.е. пусты) $obj1 и $obj2 - это два разных объекта одного типа, созданных с помощью одного класса.

Если вам все еще не понятно, давайте приведем аналогию из реальной жизни.

Представьте, что класс - это форма для отливки, с помощью которой изготавливаются пластмассовые машинки. Объекты - это и есть машинки. Тип создаваемых объектов определяется формой отливки. Машинки выглядят одинаковыми во всех отношениях, но все-таки это разные предметы. Другими словами, это разные экземпляры одного и того же типа.

Давайте сделаем эти объекты немного интереснее, изменив класс first, добавив в него специальные поля данных, называемые свойства.

Определение свойств

В классе можно определить переменные. Переменные, которые определены в классе называются свойствами (или полями данных). Они определяются с одним из ключевых слов protected, public, или private, характеризующих управление доступом. Эти ключевы слова мы рассмотрим подробно в следующей главе. А сейчас давайте определим некоторые свойства с помощью ключевого слова public:

```php
<?php

  class first {
    public $num = 0;
    public $str = 'some text';
  }

?>
```

Как видите, мы определили два свойства, присвоив каждому из них значение. Теперь любые объекты, которые мы будем создавать с помощью класса first, будут иметь два свойства с указанными значениями.

> **Примечание:** значения инициализирующие свойства должны быть литералами (константными значениями), инициализировать свойства в классе не обязательно (если значение не указано, по умолчанию это будет NULL).

К свойствам объекта можно обращаться с помощь символов '->', указав объект и имя свойства. Поскольку свойства объектов были определены как public, мы можем считывать их значения, а также присваивать им новые значения, заменяя тем самым начальные значения, определенные в классе:

```php
<?php

  class first {
    public $num = 0;
    public $str = 'some text';
  }

  $obj = new first();

  echo $obj->str;

  // присваиваем свойству объекта новое значение
  $obj->str = 'новая строка';
  echo "<br>$obj->str";

```

```
?>
```

На самом деле в PHP необязательно объявлять все свойства в классе. Свойства можно добавлять к объекту динамически:

```php
<?php

  class first {
    public $str = 'some text';
  }

  $obj = new first();

  // добавляем объекту новое свойство
  $obj->newprop = 'новое свойство';
  echo $obj->newprop;

?>
```

Нужно отметить, что этот способ присваивания свойств объектам считается дурным тоном в объектно-ориентированном программировании и почти никогда не используется.

Работа с методами

Методы - это обычные функции, которые определяются внутри класса, они позволяют объектам выполнять различные задачи. Объявление метода напоминает определение обычной функции, за исключением предваряемого одного из ключевых слов protected, public, или private. Если в определении метода вы опустите ключевое слово, определяющее видимость, то метод будет объявлен неявно как public. К методам объекта можно обращаться с помощь символов '->', указав объект и имя метода. При вызове метода, так же как и при вызове функции нужно использовать круглые скобки.

```php
<?php

  class first {
    public $str = 'some text';

    // определение метода
    function getstr() {
      echo $this->str;
    }
  }

  $obj = new first();

  // вызов метода объекта
  $obj->getstr();

?>
```

Мы добавили метод getstr() к классу first. Обратите внимание на то, что при определении метода мы не использовали ключевое слово, определяющее область видимости. Это означает, что метод getstr() относится к типу public и его можно вызвать за пределами класса.

В определении метода мы воспользовались специальной псевдопеременной ***$this***. Она используется для обращения к методам или свойствам внутри класса и имеет следующий синтаксис:

```
$this->имя переменной или метода
```

Значением переменной $this является ссылка на текущий объект. Чтобы стало понятнее посмотрите на следующий пример:

```
class first {
    public $str = 'some text';

    // при определении метода в классе, переменная $this не имеет никакого
    function getstr() {
      echo $this->str;
    }
  }

  // создаем объект
  $obj = new first();

  // созданный нами объект имеет свойство и метод
  // теперь в методе объекта переменная $this имеет
  // ссылку на текущий объект, а именно на $obj

  // т.е. если в методе заменить $this текущим экземпляром объекта
  $this->str;

  // это будет выглядеть как простое
  // обращение к свойству текущего объекта
  $obj->str;

```

Примечание: переменной $this нельзя ничего присваивать. Помните, что переменная $this всегда ссылается на текущий объект.

Специальный метод - конструктор

У класса может быть определен специальный метод - конструктор, который вызывается каждый раз при создании нового экземпляра класса (объекта) с целью инициализировать его, например установить значения свойств. Конструктор, как и любой другой метод может иметь параметры. Чтобы определить метод в качестве конструктора его необходимо назвать __construct(). Обратите внимание на то, что имя метода должно начинаться с двух символов подчеркивания. Посмотрим, как это работает:

```
<?php

  class first {
    // определяем два свойства
    public $num1 = 0;
    public $num2 = 0;

    // определяем конструктор класса
    function __construct($num1, $num2) {
      $this->num1 = $num1;
```

```php
    $this->num2 = $num2;
  }

  // метод, который складывает два числа
  function summa() {
    return $this->num1 + $this->num2;
  }
}

// создаем объект и передаем два аргумента
$obj = new first(15, 35);

// вызываем метод и сразу выводим результат его работы
echo $obj->summa();

?>
```

Метод __construct вызывается, когда создается объект с помощью оператора new. Указанные в скобках аргументы передаются конструктору. В методе конструктора используется псевдопеременная $this для присвоения значений соответствующим свойствам создаваемого объекта.

> **Примечание:** если конструктор не имеет параметров и при создании новых экземпляров класса не передаются никакие аргументы, круглые скобки () после имени класса можно опустить:
>
> ```php
> $obj = new first;
> ```

Указание типа аргумента в методах

По умолчанию метод может принимать аргументы любого типа, но бывают случаи, когда необходимо сделать так, чтобы метод мог принимать в качестве аргумента только экземпляры определенного класса. Для указания типа принимаемого аргумента, просто поместите в определении метода перед именем параметра название класса:

```php
<?php

  // определяем два пустых класса
  class cat {}
  class wrong {}

  class write {

    // метод, который принимает аргументы только типа cat
    function getobj(cat $getCat) {   // определяем параметр типа cat
      echo 'Получен объект типа cat';
    }

  }

  // создаем экземпляр типа write
  $kitty = new write();

  // работает: передали в качестве аргумента экземпляр типа cat
  $kitty->getobj( new cat() );
  // здесь будет ошибка: передали в качестве аргумента экземпляр типа wrc
```

```
  $kitty->getobj( new wrong() );

?>
```

Теперь в качестве аргумента методу getobj() можно передавать только экземпляры типа cat. Поскольку метод getobj() содержит уточнение типа класса, передача ему объекта типа wrong приведет к ошибке.

Указание типа нельзя использовать для определения параметров элементарных типов, таких как строки, числа и т.д. Для этой цели в теле метода следует использовать функции проверки типов, например is_string(). Также есть возможность определить, что передаваемый аргумент является массивом:

```
<?php

  class write {

    $my_arr;

    // аргументом для метода может быть только массив
    function setArray(array $some_arr) {
      this->my_arr = $some_arr;
    }

  }

?>
```

И последнее о чем осталось сказать: если параметр метода определяется с указанием определенного класса, разрешается указать значение по умолчанию, на случай, если методу не было передано никакого объекта. В качестве значения по умолчанию может быть использовано только значение NULL:

```
function getobj(cat $getCat = null) {
    $this->someVar = $getCat;
  }
```

Если вместо NULL указать какое-либо другое значение по умолчанию, будет выдана ошибка.

PHP: Наследование

Наследование

Наследование - это механизм объектно ориентированного программирования, который позволяет описать новый класс на основе уже существующего (родительского).

Класс, который получается в результате наследования от другого, называется подклассом. Эту связь обычно описывают с помощью терминов «родительский» и «дочерний». Дочерний класс происходит от родительского и наследует его характеристики: свойства и методы. Обычно в подклассе к функциональности родительского класса (который также называют суперклассом) добавляются новые функциональные возможности.

Чтобы создать подкласс, необходимо использовать в объявлении класса ключевое слово extends, и после него указать имя класса, от которого выполняется наследование:

```php
<?php

  class Cat {
    public $age;

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

    function add_age () {
      $this->age++;
    }

  }

  // объявляем наследуемый класс
  class my_Cat extends Cat {
    // определяем собственный метод подкласса
    function sleep() {
      echo '<br>Zzzzz...';
    }
  }

  $kitty = new my_Cat(10);

  // вызываем наследуемый метод
  $kitty->add_age();

  // считываем значение наследуемого свойства
  echo $kitty->age;

  // вызываем собственный метод подкласса
  $kitty->sleep();

?>
```

Подкласс наследует доступ ко всем методам и свойствам родительского класса, так как они имеют тип public. Это означает, что для экземпляров класса my_Cat мы можем вызывать метод add_age() и обращаться к свойству $age не смотря

на то, что они определены в классе cat. Также в приведенном примере подкласс не имеет своего конструктора. Если в подклассе не объявлен свой конструктор, то при создании экземпляров подкласса будет автоматически вызываться конструктор суперкласса.

Обратите внимание на то, что в подклассах могут переопределяться свойства и методы. Определяя подкласс, мы гарантируем, что его экземпляр определяется характеристиками сначала дочернего, а затем родительского класса. Чтобы лучше это понять рассмотрим пример:

```php
<?php

  class Cat {
    public $age = 5;

    function foo() {
      echo "$this->age";
    }
  }

  class my_Cat extends Cat {
    public $age = 10;
  }

  $kitty = new my_Cat;

  $kitty->foo();

?>
```

При вызове $kitty->foo() интерпретатор PHP не может найти такой метод в классе my_Cat, поэтому используется реализация этого метода заданная в классе Cat. Однако в подклассе определено собственное свойство $age, поэтому при обращении к нему в методе $kitty->foo(), интерпретатор PHP находит это свойство в классе my_Cat и использует его.

Так как мы уже рассмотрели тему про указание типа аргументов, осталось сказать о том, что если в качестве типа указан родительский класс, то все потомки для метода будут так же доступны для использования, посмотрите на следующий пример:

```php
<?php

  class Cat {
    function foo(Cat $obj) {}
  }

  class my_Cat extends Cat {}

  $kitty = new Cat;

  // передаем методу экземпляр класса my_Cat
  $kitty->foo( new my_Cat );

?>
```

Мы можем обращаться с экземпляром класса my_Cat так, как будто это объект

типа Cat, т.е. мы можем передать объект типа my_Cat методу foo() класса Cat, и все будет работать, как надо.

Оператор parent

На практике подклассам бывает необходимо расширить функциональность методов родительского класса. Расширяя функциональность за счет переопределения методов суперкласса, в подклассах вы сохраняете возможность сначала выполнить программный код родительского класса, а затем добавить код, который реализует дополнительную функциональность. Давайте разберем как это можно сделать.

Чтобы вызвать нужный метод из родительского класса, вам понадобится обратиться к самому этому классу через дескриптор. Для этой цели в PHP предусмотрено ключевое слово ***parent***. Оператор parent позволяет подклассам обращаться к методам (и конструкторам) родительского класса и дополнять их существующую функциональность. Чтобы обратиться к методу в контексте класса, используются символы "::" (два двоеточия). Синтаксис оператора parent:

```
parent::метод_родительского_класа
```

Эта конструкция вызовет метод, определенный в суперклассе. Вслед за таким вызовом можно поместить свой программный код, который добавит новую функциональность:

```php
<?php

  class book {
    public $title;
    public $price;

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

  class new_book extends book {
    public $pages;

    function __construct($title, $price, $pages) {
      // вызываем метод-конструктор родительского класса
      parent::__construct($title, $price);

      // инициализируем свойство определенное в подклассе
      $this->pages = $pages;
    }
  }

  $obj = new new_book('азбука', 35, 500);

  echo "Книга: $obj->title<br>
        Цена: $obj->price<br>
        Страниц: $obj->pages";

?>
```

Когда в дочернем классе определяется свой конструктор, PHP не вызывает конструктор родительского класса автоматически. Это необходимо сделать вручную в конструкторе подкласса. Подкласс сначала в своем конструкторе вызывает конструктор своего родительского класса, передавая нужные аргументы для инициализации, исполняет его, а затем выполняется код, который реализует дополнительную функциональность, в данном случае инициализирует свойство подкласса.

Ключевое слово parent можно использовать не только в конструкторах, но и в любом другом методе, функциональность которого вы хотите расширить, достигнуть этого можно, вызвав метод родительского класса:

```php
<?php

  class Cat {
    public $name = "Арни";

    function getstr() {
      $str = "Имя кота: {$this->name}.";
      return $str;
    }
  }

  class my_Cat extends Cat {
    public $age = 5;

    function getstr() {
      $str = parent::getstr();

      $str .= "<br>Возраст: {$this->age} лет.";
      return $str;
    }
  }

  $obj = new my_Cat;
  echo $obj->getstr();

?>
```

Здесь сначала вызывается метод getstr() из суперкласса, значение которого присваивается переменной, а после этого выполняется остальной код определенный в методе подкласса.

Теперь, когда мы познакомились с основами наследования, можно, наконец, рассмотреть вопрос видимости свойств и методов.

public, protected и private: управление доступом

До этого момента мы явно объявляли все свойства как public (общедоступные). И такой тип доступа задан по умолчанию для всех методов.

Элементы класса можно объявлять как ***public*** (общедоступные), ***protected*** (защищенные) и ***private*** (закрытые). Рассмотрим разницу между ними:

- К **public** (общедоступным) свойствам и методам, можно получить доступ из любого контекста.
- К **protected** (защищенным) свойствам и методам можно получить доступ либо

из содержащего их класса, либо из его подкласса. Никакому внешнему коду доступ к ним не предоставляется.
- Вы можете сделать данные класса недоступными для вызывающей программы с помощью ключевого слова **private** (закрытые). К таким свойствам и методам можно получить доступ только из того класса, в котором они объявлены. Даже подклассы данного класса не имеют доступа к таким данным.

public - открытый доступ:

```php
<?php

  class human {
    public $age = 5;
    public function say() {
      echo "<br>hello";
    }
  }

  $obj = new human;

  // доступ из вызывающей программы
  echo "$obj->age";  // Допустимо
  $obj->say();       // Допустимо

?>
```

private - доступ только из методов класса:

```php
<?php

  class human {
    private $age = 5;
    function say() {
      // внутри класса доступ к закрытым данным есть
      echo "$this->age";
    }
  }

  $obj = new human;

  // напрямую из вызывающей программы доступа к закрытым данным нет
  echo "$obj->age";  // Ошибка! доступ закрыт!

  // однако с помощью метода можно выводить закрытые данные
  $obj->say();       // Допустимо

?>
```

protected - защищенный доступ:

Модификатор protected с точки зрения вызывающей программы выглядит точно так же, как и private: он запрещает доступ к данным объекта извне. Однако в отличие от private он позволяет обращаться к данным не только из методов своего класса, но также и из методов подкласса.

PHP: Статические методы и свойства

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. Определить статические данные класса можно с помощью ключевого слова static:

```php
<?php

  class Test {
    static public $num = 1;

    static public function sayHi() {
      echo 'Привет!';
    }
  }

?>
```

Статические методы сами не могут получать доступ ни к каким обычным свойствам класса, потому что такие свойства принадлежат объектам. Однако из статических методов можно обращаться к статическим свойствам.

Чтобы обратиться к статическому элементу класса нужно указать имя класса после которого указывается два двоеточия, а затем имя статического свойства или метода:

```php
echo test::$num;
test::sayHi();
```

С этим синтаксисом вы познакомились в предыдущей главе. Там мы использовали конструкцию "::" в сочетании с ключевым словом parent, чтобы получить доступ к переопределённому методу родительского класса.

В коде класса можно использовать ключевое слово parent, чтобы получить доступ к суперклассу, не используя имя класса. Чтобы получить доступ к статическому методу или свойству из того же самого класса (не из дочернего), можно использовать ключевое слово ***self***, оно используется для обращения к текущему классу. Поэтому из-за пределов класса мы обращаемся к данным с помощью имени класса:

```php
test::$num;
```

А внутри класса можно использовать ключевое слово self:

```php
<?php

  class Test {
    static public $num = 1;

    static public function sayHi() {
      self::$num++;
      echo 'Число: '. self::$num;
    }
  }

```

```
?>
```

Теперь разберем вопрос о том, зачем вообще использовать статические методы или свойства. У статических элементов есть ряд полезных характеристик. Они доступны из любой точки сценария. Статическое свойство доступно каждому экземпляру этого класса. Поэтому можно определить значения, которые должны быть доступны всем объектам данного типа. И наконец, сам факт, что не нужно иметь экземпляр класса для доступа к его статическому свойству или методу, позволит избежать создания экземпляров исключительно ради вызова простой функции. Давайте посмотрим как обращаться к методам и свойствам из объекта:

```php
<?php

  class Test {
    static public $num = 1;

    static public function sayHi() {
      self::$num++;
      echo '<br>Число: '. self::$num;
    }
  }

  $obj = new Test;

  // доступ к статическому свойству из объекта
  echo $obj::$num;
  // вызов статического метода из объекта
  $obj::sayHi();
  // второй способ вызова статического метода
  $obj->sayHi();  // такой способ не рекомендуется использовать, чтобы не

?>
```