Глава 9

ООП в PHP

В процессе своей эволюции языки программирования обогащались все новыми подходами к описанию различных предметных областей. Основной задачей создателей языков программирования было повышение читабельности кода и упрощение поддержки и расширения функциональности программ.

Одним из важнейших этапов на этом пути стало изобретение *объектно-ориентированного подхода* (ООП). Дело в том, что программы представляли собой последовательности инструкций, т.е. они были ориентированы на операции над некоторыми значениями. При этом было довольно трудно отследить взаимосвязь между ними, и понять логику программы было намного сложнее. Поэтому были придуманы способы описания сложных объектов и правил взаимодействия с ними. Оказалось, что такой подход позволяет программистам работать совершенно на другом уровне абстракции и в своих программах оперировать понятиями соответствующей предметной области: описав класс объектов один раз, можно забыть о том, как он устроен.

Использование объектно-ориентированного подхода также заметно упрощает написание и чтение программ. В частности, часто оказывается, что в программе группа функций использует одни и те же переменные. Приходится каждый раз передавать эти переменные в качестве параметров (что само по себе утомительно) или объявлять их как глобальные, рискуя возникновением конфликтов имен, когда различные функции используют одну и ту же переменную в различных целях и перезаписывают в нее свои значения, мешая друг другу.

Кроме того, классы объектов могут наследовать свойства друг друга, что избавляет программистов от дублирования кода — если в грамотно разработанной программе находят ошибку, то она исправляется в одном месте.

Впервые объектно-ориентированный подход был реализован в знаменитом языке Smalltalk 80, разработанном в лаборатории Xerox PARC. Кстати, на нем был реализован первый графический оконный интерфейс.

В PHP, как и большинстве современных языков программирования, также имеются средства объектно-ориентированного программирования. С ними мы и познакомимся в данной главе.

9.1. Основные концепции ООП

9.1.1. *Инкапсуляция*

Как уже было сказано, объектно-ориентированные языки программирования позволяют описывать классы объектов, что делает возможным абстрагирование от их внутреннего устройства.

Например, далеко не все водители знают, как работает двигатель внутреннего сгорания — им достаточно знать, как его завести, заглушить и как поведет себя машина, если нажать на педаль газа. Таким образом программист может в своей программе использовать класс, написанный другим программистом, также не вникая в то, как он реализован, — это называют *инкапсуляцией*.

Описание класса включает в себя перечень *свойств* (или *атрибутов*) и *методов* — функций, относящихся к этому объекту и имеющих прямой доступ к его свойствам. Кроме того, обычно описываются специальные методы, позволяющие получать и изменять значения атрибутов объектов, а также специальные служебные методы, отвечающие за создание и удаление объектов из памяти: *конструкторы* и *деструкторы*.

Объекты (или *экземпляры класса*) во многом похожи на обычные переменные. В частности, они могут передаваться функциям в качестве параметров. Главное, чтобы функция “знала”, как с этими объектами работать.

9.1.2. *Наследование*

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

Переопределение свойств и методов называют *перегрузкой*. Для того чтобы изменить поведение метода, унаследованного из базового класса, достаточно переопределить этот метод в классе-наследнике. При этом имеется возможность вызова аналогичного метода из базового класса. Поэтому для добавления нескольких строк в перегружаемый метод достаточно вызвать этот метод из базового класса, а затем дописать необходимую функциональность.

Некоторые языки программирования (например, C++ и Python) предусматривают *множественное наследование* (multipleinheritance), т.е. возможность создания нового класса в результате наследования от нескольких базовых классов одновременно.

В языках, не поддерживающих множественное наследование, обычно применяется “обходной маневр” — *агрегирование*. Идея агрегирования состоит в создании цепочки классов, последовательно наследующих друг у друга свойства и методы. Правда, это не всегда удобно, так как в результате получается несколько классов, которые, скорее всего, не будут использоваться в программе.

9.1.3. *Абстрактные методы и классы, полиморфизм*

Абстрактными методами называют методы, не имеющие реализации в классе, в котором они объявляются. Классы, имеющие хотя бы один такой метод, называют *абстрактными классами*. Экземпляры таких классов создать невозможно — они предназначены для создания других классов с помощью наследования.

Концепция абстрактных классов восходит к языками программирования со строгой типизацией. Дело в том, что строгая типизация, например, не позволяла создавать массивы из объектов различных типов (классов), даже если они имели одинаковый набор методов с небольшими отличиями в реализации. Поэтому сначала создавался абстрактный класс, затем в результате наследования создавались классы-наследники, в которых перегружались абстрактные методы. При объявлении массива в качестве типа его элементов указывался абстрактный класс, но хранить он мог любые объекты его классов-наследников. Таким образом было реализовано единообразие работы с объектами различных классов, но схожими методами работы, или полиморфизм.

9.2. Реализация ООП в PHP

Коротко рассмотрев основные концепции, перейдем к изучению их реализации в PHP 5 и заодно разберемся с ними подробнее. В качестве примера создадим набор классов для Интернет-магазина, занимающегося продажей книг и лазерных дисков.

9.2.1. *Создание классов и объектов*

Для описания класса используется специальная синтаксическая структура, начинающаяся с ключевого слова `class`. Создадим класс товара.

```
<?php
class Commodity {
    public $name;           // Название товара
    public $category;       // Категория товара
    public $price;          // Цена
    public $availability;   // Наличие на складе
}
?>
```

Сохраним данный класс в отдельном файле с именем `commodity.php`, чтобы потом подключать его к тестовым скриптам. Объекты данного класса будут иметь четыре свойства: “Название товара” (`$name`), “Категория товара” (`$category`), “Цена” (`$price`) и “Наличие на складе” (`$availability`). Создание объекта данного класса происходит с помощью оператора `new`.

```
<?php
require_once 'commodity.php';

$obj = new Commodity();
```

```
$obj->name = 'Разработка Web-приложений на PHP 5';
$obj->category = 'book';
$obj->availability = False;
?>
```

Для создания объекта класса достаточно первой строчки приведенного примера, но мы также определили значения некоторых его атрибутов, обращение к которым происходит с помощью конструкции ->.

Удаление объекта происходит так же, как и обычных переменных с помощью функции unset().

```
unset($obj);
```

9.2.2. *Методы класса*

Создадим в классе товара несколько методов.

```
<?php
class Commodity {
    public $name;           // Название товара
    public $category;       // Категория товара
    public $price;          // Цена
    public $availability;   // Наличие на складе


    // Конструктор
    function __construct($name, $category, $price = null,
                         $availability = False) {
        echo 'Запущен конструктор...<br/>';
        $this->name = $name;
        $this->category = $category;
        $this->price = $price;
        $this->vailability = $availability;
    }

    // Деструктор
    function __destruct() {
        echo 'Запущен деструктор...<br/>';
    }

    function getPrice() {
        return (is_null($this->price) ? 'N/A' : $this->price);
    }

    function setPrice($new_price) {
        $this->price = $new_price;
    }
}
?>
```

Первый метод __construct()[1] является служебным. Он отвечает за создание экземпляра класса — он вызывается автоматически при каждом создании объекта. Конструктору могут переданы некоторые параметры, так же как и обычной функции. В нашем примере конструктор используется для задания начальных значений

[1] *Название метода начинается с двух знаков подчеркивания.*

атрибутов создаваемого экземпляра класса. При этом обращение к атрибутам объекта из методов происходит через указатель $this.

При уничтожении объекта так же вызывается специальный метод __destruct() — деструктор класса.

Понять, как работают методы getPrice() и setPrice(), скорее всего, не составит труда. Теперь проверим его работу.

```
<?php
require_once 'commodity.php';


$obj = new Commodity('Разработка Web-приложений на PHP 5',
                     'book');

echo $obj->name . ' - ' . $obj->getPrice() . '<br/>';
$obj->setPrice(230.0);
echo $obj->name . ' - ' . $obj->getPrice() . '<br/>';
?>
```

В результате выполнения наша программа выведет следующее.

```
Запущен конструктор...
Разработка Web-приложений на PHP 5 - N/A
Разработка Web-приложений на PHP 5 - 230
Запущен деструктор...
```

Сначала создается объект $obj класса Commodity; при этом с помощью конструктора задаются значения атрибутов $obj->name и $obj->category. О том, что конструктор выполнился, мы можем судить по тестовой строке "Запущен конструктор... ", которую мы добавили в его тело.

Далее мы вывели название созданного объекта и цену, используя метод getPrice(). Так как цена не была задана, метод getPrice() вернул строку N/A. После этого мы задали цену, вызвав метод setPrice(), и повторили вывод — на этот раз метод getPrice() вернул только что заданное значение.

Обратите внимание, что в конце программы мы не уничтожали явно наш объект — за нас это сделал диспетчер памяти, предварительно запустив деструктор. Об этом свидетельствует строка "Запущен деструктор... ".

> **Примечание.** Средства ООП появились еще в PHP 3, но в PHP 5 были изменены правила именования конструктора. В более ранних версиях PHP роль конструктора выполнял метод, названный именем класса.
> Для совместимости с предыдущими версиями интерпретатор PHP 5 ведет себя следующим образом: если при создании объекта в классе не найдет конструктор __construct(), то происходит попытка выполнить метод, имя которого совпадает с именем класса.

9.2.3. *Клонирование объектов*

Одним из важнейших отличий PHP 5 от более ранних версий является то, что теперь присвоение объекта или его передача в качестве параметра функции происходит по умолчанию по ссылке, а не по значению, как в предыдущей версии.

И если в PHP 4 объекты обрабатывались так же, как и простые типы данных, что часто приводило к появлению нескольких копий одного и того же объекта, то в PHP 5 такого не происходит, так как каждый объект получает свой собственный числовой идентификатор (handle), который и используется при обращении к объекту. Это изменение было сделано для увеличения производительности PHP-сценариев, где активно используется работа с объектами.

Данная программа, запущенная в PHP 4 и PHP 5, выдаст различные результаты.

```
<?php
require_once 'commodity.php';

$obj1 = new Commodity('Разработка Web-приложений на PHP 5',
                      'book');
$obj1->price = 150.0;
$obj2 = $obj1;
$obj2->price = 230.0;
echo $obj1->price . '<br/>';
echo $obj2->price . '<br/>';
?>
```

В PHP 4 `$obj2` представляет собой копию объекта `$obj1`, а в PHP 5 `$obj1` и `$obj2` указывают на один и тот же объект, так как оператор `$obj2 = $obj1` копирует не сам объект, а только его идентификатор.

Тем не менее в некоторых случаях необходимо создать именно копию объекта. Для этого используется конструктор копирования, вызываемый с помощью ключевого слова `clone`. Например:

```
$obj2 = clone $ob1;
```

При этом объект копируется со всеми своими методами, свойствами и их значениями. Конструктор копирования можно перегрузить, добавив в класс свою функцию `__clone()`. При этом метод `__clone()` не может принимать никакие аргументы, однако позволяет обратиться к исходному объекту через указатель `$this` и к получаемому в результате копирования объекту через указатель `$that`.

9.2.4. *Наследование и перегрузка методов*

Итак, имея класс, мы можем создавать объекты этого класса. Нетрудно представить ситуацию, когда возникает потребность в создании объектов, которые от них немного отличаются: например, количеством или набором полей или одним или несколькими методами.

Самым простым решением может стать создание копии исходного класса с другим именем и модификация его для других потребностей. Недостаток этого подхода кроется в том, что при необходимости изменить функциональность, которая

повторяется в обоих классах, придется вносить правки в двух местах. В результате возрастает вероятность допустить ошибку. Оптимальным решением в данной ситуации является наследование.

Рассмотрим данную концепцию на примере класса товара. Предположим, что появилась вполне логичная потребность конкретизировать свойства продаваемых через Интернет-магазин товаров. Например, если мы продаем книги, то объект товара должен хранить в себе заголовок книги, имя автора и, возможно, другую информацию. При этом такие свойства, как категория товара, его цена и наличие на складе нам также пригодятся (как и методы, позволяющие ими управлять).

Поэтому расширим функциональность класса `Commodity`, создав с помощью наследования класс `Book`. Для начала в заголовке нового класса укажем, что он является наследником класса `Commodity`.

```
<?php
require_once'commodity.php';

class Book extends Commodity {
}


// Посмотрим, что получилось:
$book1 = new Book('Разработка Web-приложений на PHP 5',
                  'Программирование');
echo '<pre>';
print_r($book1);
echo '</pre>';
?>
```

Выполнив данный пример, вы увидите, что класс `Book` полностью идентичен классу `Commodity`.

Теперь добавим поле `"Авторы"` и метод `getAuthors()`, возвращающий массив с именами авторов.

```
<?php
require_once 'commodity.php';

class Book extends Commodity {
    public $authors = Array();

    function getAuthors() {
        return $this->authors;
    }
}

?>
```

Логично было бы передавать список имен авторов в качестве параметра конструктора. При этом новый конструктор должен делать все то же, что и конструктор базового класса, плюс некоторые дополнительные действия. В такой ситуации метод перегружают.

```
<?php
require_once 'commodity.php';

class Book extends Commodity {
    public $authors;

    function __construct($name, $authors, $category,
                         $price = null,
                         $availability = False) {
        parent::__construct($name, $category, price,
                            $availability)
        $this->authors = $authors;
    }

    function getAuthors() {
        return $this->authors;
    }
}

?>
```

Как видите, новый конструктор имеет нужный нам набор полей. При этом из его тела происходит вызов конструктора базового класса с помощью ключевого слова `parent`, а затем выполняются действия, специфические для конструктора класса `Book`. Аналогичным образом могут перегружаться любые методы базового класса в классе-наследнике.

Вообще говоря, в PHP 5 предусмотрено два специальных встроенных метода, которые вызываются при обращении к свойству класса и при изменении его значения, — это методы `__get()` и `__set()`. Например, если бы мы решили хранить все свойства товаров в массиве, то эти методы было бы очень удобно использовать.

```
<?php
class Commodity {
    private $properties;

    function __set($name, $value) {
        echo "задание нового свойства $name = $value";
        $this->properties[$name]=$value;
    }
    function __get($name) {
        echo "чтение значения свойства ", $name;
        return $this->properties[$name];
    }
}
?>
```

При этом работа с товаром выглядела бы так.

```
$book = new Book('Разработка Web-Приложений на PHP 5',
                 'Программирование');
$book->wieght = 100;
              // Выводит "задание нового свойства wight=100"
$weight = $book->weight;
              // Выводит "чтение значения свойства wight"
echo $weight; // выводит 1;
```

Новые методы доступа `__get()` и `__set()` позволяют легко проводить динамическое назначение свойств объектам. В качестве параметров этим методам передаются имена свойств. Кроме того, метод `__set()` также получает и значение, которое присваивается указанному свойству.

9.2.5. *Финальные методы*

В PHP 5 введена новая возможность определять методы класса и сами классы как *финальные*. Перед определением таких методов и классов пишется ключевое слово `final`.

Метод, при определении которого используется ключевое слово `final`, не может быть перегружен в классах-наследниках. Если `final` используется при определении самого класса, то порождение от него других классов становится невозможным. При этом все его методы автоматически становятся финальными, поэтому явно определять их как `final` уже нет необходимости.

Атрибуты класса определять как финальные не допускается.

9.2.6. *Методы __toString() и __call()*

Помимо методов `__get()` и `__set()` в PHP 5 было добавлено еще несколько специальных методов, которые создаются в классе во время выполнения сценария. Эти методы могут перегружаться в любом классе.

Метод `__toString()` вызывается интерпретатором PHP, если ему необходимо преобразовать объект в строку (т.е. при каждом вызове функций `echo()` и `print()`). Перегрузив его, вы можете указать, как объект должен выглядеть в строковом представлении.

Перегрузим метод `__toString()` в классе Book.

```
<?php
require_once 'commodity.php';

class Book extends Commodity {
    public $authors;

    // ...
    function __toString() {
        foreach ($this->authors as $author) {
            echo $author . ', ';
        }
        echo $this->name;
        echo ($availability ? '(Есть на складе)'
                            : '(На складе отсутствует)');
    }
}
?>
```

Такое представление объекта более читабельно, не так ли?

Метод `__call()` вызывается, если вызванный метод в классе не определен.

```
<?php
class Commodity {
    function __call($name, $params) {
        echo "Вызван неизвестный метод $name<br/>";
        echo 'Переданные параметры: <pre>';
        print_r($params);
        echo '</pre>';
    }
}

$book = new Book('Разработка Web-приложений на PHP 5',
                 'Программирование');
$book->method(1, 'test');
                  // Выводит "Вызван неизвестный метод method"
                  // и список переданных параметров
?>
```

Как видно из примера, в качестве параметров метод `__call()` принимает имя вызываемого метода и передаваемые этому методу параметры в виде массива.

> **Примечание.** Существует еще два специальных метода (по-английски magic methods): `__sleep()` и `__wakeup()`. Они применяются для определения поведения объекта при его упаковке и распаковке с помощью функций `serialize()` и `unserialize()`. За подробным описанием данных методов и функций обращайтесь к документации PHP.

Отдельно следует обратить внимание на функцию `__autoload()`. Большинство разработчиков пишут свои Web-приложения, размещая каждый класс в отдельном файле для удобства управления ими. Но такой подход вынуждает каждый раз перечислять все подключаемые файлы, и список подключаемых модулей может быть довольно большим.

В PHP 5 эта проблема решается очень просто — достаточно описать в функции `__autoload()`, как подключатся файлы с описаниями классов. Например:

```
function __autoload($class_name) {
    require_once MODULES_PATH . $class_name . '.php';
}
```

Таким образом при инициализации неизвестного класса функция `__autoload()`, вызываемая интерпретатором в таких случаях автоматически, будет подключать недостающий модуль.

> **Замечание.** Если в функции `__autoload()` возникнет исключение (о них подробно будет рассказано в следующей главе), то обработать его, к сожалению, будет невозможно. Поэтому интерпретатор в любом случае выдаст сообщение о фатальной ошибке. Будьте внимательны!

9.2.7. *Управление доступом к методам и свойствам*

Вероятно, вы уже обратили внимание на ключевое слово `public`, которое используется при объявлении атрибутов класса. На самом деле таких ключевых слов три — `public`, `protected` и `private`. Они используются для определения правил доступа к атрибутам и методам данного класса.

В более ранних версиях PHP никаких ограничений не было: объявление атрибутов класса начиналось с ключевого слова `var`.

```
<?php
class Test {
    var $test;
}
?>
```

В PHP 5 подобная запись допустима (для совместимости с предыдущими версиями), но вместо нее рекомендуется использовать ключевое слово `public`, обозначающее общедоступные методы и свойства класса.

Защищенные (`protected`) элементы класса доступны внутри класса, в котором они объявлены, и в производных от него классах. Закрытые (`private`) элементы доступны только в классе, в котором они объявлены.

Чтобы лучше понять, как это работает, обратимся к следующему примеру.

```
<?php
     class BaseClass {
         public $public = "общедоступный элемент";
         protected $protected = "защищенный элемент";
         private $private = "закрытый элемент"
         public function printPrivate() {
             echo $this->private;
         }
     }
     $obj1 = new BaseClass;
     echo $obj1->public; // Выводит "общедоступный элемент"

     class ExtendedClass extends BaseClass {
         public function printProtected() {
             echo $this->protected;
         }
     }
     $obj2 = new ExtendedClass();
     $obj2->printProtected(); // Выводит "защищенный элемент"
     $obj1->printPrivate();   // Выводит "закрытый элемент"

     echo $obj1->protected;   // Вызывает ошибку доступа
     echo $obj1->private;     // Вызывает ошибку доступа
?>
```

Если не указывать ни один из спецификаторов, то по умолчанию элемент будет иметь уровень доступа `public`.

Примечание. Не стоит делать все свойства и методы защищенными или закрытыми, так как для обращения к ним извне придется писать специальные методы, возвращающие или изменяющие их значения. Такой подход ухудшит читабельность программы и отрицательно скажется на скорости ее выполнения. Поэтому не стоит стремиться к тому, чтобы программа на 100% соответствовала всем требованиям объектно-ориентированного подхода.

9.2.8. *Полиморфизм*

Основной целью, которая преследовалась при создании 5-й версии PHP, было достижение максимального соответствия концепциям объектно-ориентированного программирования. Поэтому в PHP 5 также появилась возможность создания абстрактных методов и классов. Полиморфизм в языках с нестрогой типизацией может быть сымитирован и стандартными средствами, но введение специального синтаксиса позволило более жестко описать правила использования этой концепции.

Итак, абстрактные методы имеют только объявление, но не имеют реализации. Класс, который содержит такие методы, должен быть обязательно объявлен как абстрактный. В нашем примере с электронным магазином абстрактным, возможно, имеет смысл сделать класс `Commodity`, добавив в него абстрактный метод вывода свойств товара, поскольку, не зная, о каком именно товаре идет речь, мы не может точно сказать, каким набором специфических свойств он обладает.

```
<?php
abstract class Commodity {
    ...
    abstract function printProperties();
}
?>
```

Обратите внимание, что абстрактные классы также могут содержать и обычные (неабстрактные) методы, т.е. методы, которые мы описали в предыдущих разделах, никуда не делись. При этом конструктор и деструктор точно так же будут наследоваться от базового класса.

Класс-наследник абстрактного класса, в котором не был перегружен хотя бы один абстрактный метод, также является абстрактным.

9.2.9. *Интерфейсы*

Интерфейсы во многом схожи с абстрактными классами. Но сфера их применения несколько отличается. *Интерфейсы* описывают наборы методов, которыми должен обладать класс, реализующий соответствующий интерфейс. Если этот класс не будет содержать описания хотя бы одного из методов интерфейса, то это приведет к возникновению ошибки интерпретации программы. Если класс, реализующий интерфейс, абстрактный, то наличие в нем определения методов интерфейса не обязательно, при условии, что эти определения будут в его классах-наследниках.

Обратите также внимание, что интерфейсы не могут содержать какой-либо информации об атрибутах объектов. Синтаксис определения интерфейсов очень прост и напоминает определение класса.

```
interface Int1 {
    function funct1();
    function funct2();
    // и так далее
}
```

Для указания того, что класс реализует интерфейс, используется ключевое слово `implements`, после которого следует список интерфейсов, реализуемых классом.

```
class MyClass implements Int1, Int2 {
    public function funct1() {
        echo "Вызов метода 1";
    }
    public function funct2() {
        echo "Вызов метода 2";
    }
}
```

Следует также заметить, что класс может быть наследником одного класса и при этом поддерживать несколько интерфейсов. При этом указание базового класса должно обязательно следовать до указания перечня реализуемых документов.

Вернемся к нашему примеру простейшего электронного магазина. Создадим два интерфейса: один для определения методов работы с базой данных, а другой для определения методов работы с подсистемой статистики. Разумеется, наш пример гипотетический, поэтому мы не станем описывать все методы, которые могут быть в реализуемых нашим классом интерфейсах.

Создадим файл `interfaces.php` с описанием необходимых нам интерфейсов.

```
<?php
interface Database {
    function select();
    function insert();
    function update();
    function delete();
}

interface Stats {
    function increaseViews();
    function increaseOrders();
}
```

Подключим его к файлу с описанием класса книги и реализуем в нем эти интерфейсы.

```
<?php


require_once 'commodity.php';
require_once 'interfaces.php';

class Book extends Commodity implements Database, Stats {
    public $authors;
```

```
    function select() {
        // Тело метода select()
    }

    function insert() {
        // Тело метода insert()
    }

    function update() {
        // Тело метода update()
    }

    function delete() {
        // Тело метода delete()
    }

    function increaseViews() {
        // Тело метода increaseViews()
    }

    function increaseOrders() {
        // Тело метода increaseOrders()
    }
}

?>
```

Мы не стали описать методы полностью в целях экономии места. Логика программы и назначение методов понятны и без комментариев.

9.2.10. *Константы класса*

Еще одно новшество PHP 5 — константы классов. Теперь имеется возможность определять константы внутри классов, а не только в глобальном пространстве имен, что позволяет избежать конфликтов.

Синтаксис определения констант классов следующий.

```
class MyClass {
    const CONSTANT = 'value';
}
```

Обращение к константе класса происходит так.

```
echo MyClass::CONSTANT; // Выводит "value"
```

9.2.11. *Статические свойства и методы*

Статические свойства и методы также являются нововведением в PHP 5. Статические свойства и методы принадлежат всему классу и не относятся к какому-то конкретному объекту. Для определения статических методов и свойств используется ключевое слово `static`.

Статические свойства могут использоваться, например, для счетчиков количества экземпляров объектов одного класса. Добавим такое свойство в класс книги и модифицируем конструктор и деструктор так, чтобы при создании экземпляра элемента счетчик увеличивался на единицу, а при удалении — уменьшался.

```
<?php
class Book extends Commodity implements Database, Stats {
    static $counter;

    function __construct($name, $authors, $category,
                         $price = null,
                         $availability = False) {
        parent::__construct($name, $category, price,
                            $availability)
        $this->authors = $authors;

        // Увеличение счетчика
        Book::$counter++;
    }

    // ...

    function __destruct() {
        parent::__destruct();
        Book::$counter--;
    }
}
?>
```

Таким образом, обратившись к статическому свойству, можно определить количество инициализированных в данный момент экземпляров классов.

```
echo Book::$counter;
```

Обратите внимание, что обращаться к статическим свойствам через указатель `$this` не допускается, так как статические свойства находятся вне контекста какого-то одного объекта.

Статические методы классов в PHP 5, так же как и статические свойства, принадлежат всему классу в целом. Это позволяет использовать такой метод без инициализации объекта такого класса.

Чтобы проиллюстрировать эту возможность, добавим в класс книги статический метод `countItems()`, возвращающий количество инициализированных объектов.

```
<?php
class Book extends Commodity implements Database, Stats {
    static $counter;

    // ...

    static function countItems() {
        return Book::$counter;
    }
}
?>
```

Вызываться эта функция будет так.

```
echo Book::countItems();
```

9.2.12. *Сравнение объектов*

В PHP 5 были введены некоторые изменения в процесс сравнения объектов. При использовании оператора сравнения (==) объекты сравниваются как обычно: два *объекта считаются равными*, если они являются экземплярами одного и того же класса, имеют одинаковый набор атрибутов и значения этих атрибутов равны.

Но оператор проверки идентичности (===) теперь работает несколько иначе: *объекты считаются идентичными*, если они ссылаются на один и тот же экземпляр одно класса.

Проиллюстрируем вышесказанное на следующем примере.

```
<?php
function bool2str($bool) {
    return ($bool === False ? 'False' : 'True');
}

function compareObjects(&$obj1, &$obj2) {
    echo '== : ' . bool2str($obj1 == $obj2) . '<br/>';
    echo '=== : ' . bool2str($obj1 === $obj2) . '<br/>';
    echo '<br/>';
}

class Class1 {
    public $flag;

    function Flag($flag = True) {
        $this->flag = $flag;
    }
}

class Class2 {
    public $flag;

    function OtherFlag($flag = True) {
        $this->flag = $flag;
    }
}

$o1 = new Class1();
$o2 = new Class1();
$o3 = $o1;
$o4 = new Class2();

echo 'Два экземпляра одного класса<br/>';
compareObjects($o1, $o2);

echo 'Две ссылки на один объект<br/>';
compareObjects($o1, $o3);

echo 'Экземпляры различных классов<br/>';
compareObjects($o1, $o4);
?>
```

Вывод данного скрипта будет выглядеть так.

```
Два экземпляра одного класса
== : True
=== : False

Две ссылки на один объект
== : True
=== : True

Экземпляры различных классов
== : False
=== : False
```

9.3. Стандартная библиотека PHP

Стандартная библиотека PHP (Standard PHP Library — SPL) представляет собой набор обыкновенных интерфейсов и классов, которые встроены в PHP и обычно не требуют установки или подключения дополнительных расширений и библиотек. Эти классы призваны решить те задачи, на которые разработчик тратит большое количество времени. На данный момент в этом расширении PHP предусмотрена реализация классов для решения целого подмножества проблем — итераторов.

Итераторы используются для решения многих задач, начиная с листинга директорий файловой системы и заканчивая "перегрузкой" синтаксиса PHP. Например, перегрузить можно массивы, создав собственную конструкцию, которая будет подобна массивам. Это может быть очень удобно, так как новые конструкции будут "настроены" в соответствии с вашими потребностями.

По большому счету, итераторы автоматизируют все то, что может быть пройдено в цикле — массивы, строки файла, выборки из базы данных. Это расширение PHP сейчас бурно развивается и в разных релизах PHP 5 может немного отличаться. Чтобы узнать, какие стандартные классы есть в вашей системе, необходимо выполнить несколько строк кода.

```
<?php
print_r(spl_classes());
?>
```

Этот код выведет на экран список наименований встроенных классов, который может выглядеть вот так.

```
Array
(
    [ArrayObject] => ArrayObject
    [ArrayIterator] => ArrayIterator
    [CachingIterator] => CachingIterator
    [CachingRecursiveIterator] => CachingRecursiveIterator
    [DirectoryIterator] => DirectoryIterator
    [FilterIterator] => FilterIterator
    [LimitIterator] => LimitIterator
    [ParentIterator] => ParentIterator
    [RecursiveDirectoryIterator] => RecursiveDirectoryIterator
    [RecursiveIterator] => RecursiveIterator
    [RecursiveIteratorIterator] => RecursiveIteratorIterator
```

```
        [SeekableIterator] => SeekableIterator
        [SimpleXMLIterator] => SimpleXMLIterator
    )
```

9.3.1. *Создание собственных итераторов*

Для написания итератора, который будет наиболее полно отвечать вашим потребностям, нужно воспользоваться интерфейсом `iterator`, входящим в SPL. В собственном итераторе необходимо реализовать несколько методов, которые рассмотрены в табл. 9.1.

Таблица 9.1. Обязательные методы итератора

Метод	*Описание*
`__construct()`	Стандартный конструктор класса
`rewind()`	Перемещает указатель конструкции в начало
`key()`	Возвращает ключ текущего элемента
`next()`	Перемещает указатель на следующий элемент
`valid()`	Говорит о том, существует элемент или нет

Рассмотрим пример написания собственного итератора, задачей которого будет извлечение строк из файла и удаление в них всех HTML-тегов (листинг 9.1).

Листинг 9.1. Пример написания и использования собственного итератора

```
<?php
class FileStack implements Iterator {

//здесь будут храниться данные из файла
private $data = array();

private $valid = FALSE;

function __construct($fpath) {
//при создании объекта данные из файла $path
//конвертируются в массив
$this->data = file($fpath);
}

function rewind(){
$this->valid = (FALSE !== reset($this->data));
}

function current(){
//каждый текущий элемент (строка из файла)
//подвергается очистке от тегов
$stripped = strip_tags($this->data[$this->key()]);
//обратите внимание на то, как может происходить
//обращение к текущему элементу
$this->data[$this->key()] = $stripped;
return current($this->data);
}
```

```
function key(){
return key($this->data);
}

function next(){
$this->valid = (FALSE !== next($this->data));
}

function valid(){
return $this->valid;
}
}
//А теперь используем этот итератор!
// Создаем объект-итератор
$stack = new FileStack("stack.txt");
// Начать итерации!
foreach ( $stack as $line ) {
echo $line ."<br>";
}
?>
```

В результате выполнения данного кода из файла `stack.txt` будет извлечено содержимое, удалены все теги и обработанный вариант выведен на экран. Конечно, эту задачу можно решить и по-другому, но данный пример был приведен для демонстрации способа написания собственного итератора.

Преимуществом данного подхода является то, что программисту не приходится следить за перемещениями курсора внутри итератора и писать собственные функции для обработки его элементов. Программист только указывает, что необходимо выполнить при том или ином событии (проход текущего элемента, переход к следующему и т.д.), и не отвлекается на структуру тех данных, которые перебирает.

9.3.2. *Встроенные классы*

SPL предлагает не только некоторые абстрактные интерфейсы, помогающие разработчику выстраивать свои классы, но и добавляет в арсенал большое количество прикладных классов. В их числе можно указать следующие.

- `ArrayIterator`. Итератор для управления массивами и открытыми свойствами объектов.
- `ArrayObject`. Обертка для массива, которая позволяет рекурсивно перебирать массивы и открытые свойства объектов.
- `DirectoryIterator`. Итератор для прохода по директории в файловой системе.
- `FilterIterator`. Фильтр для итераторов.
- `ParentIterator`. Позволяет отфильтровывать элементы без потомков.
- `RecursiveDirectoryIterator`. Позволяет проходить директории в файловой системе с возможностью захода, т.е. рекурсивной обработки.

Так как SPL это не одна общая тема, а всего лишь набор классов и интерфейсов для решения различных задач, то мы рассмотрим некоторые из классов в соответствующих тематике главах. Например, работа `RecursiveDirectoryIterator` будет рассмотрена в главе по работе с файловой системой.

Подведем итог

Итак, с выходом 5-й версии PHP обрел множество полезных синтаксических конструкций, предназначенных для написания объектно-ориентированных программ. Старый синтаксис был существенно доработан для того, чтобы программы на PHP лучше соответствовали идеологии ООП и стали более читабельными.

В частности, появились модификаторы доступа к свойствам и методам классов, поддержка абстрактных классов и интерфейсов, константы классов и статические свойства. С учетом того, что в PHP 5 нет поддержки множественного наследования (multiple inheritance — одновременного наследования от нескольких классов), можно сказать, что в качестве более-менее адекватного аналога могут выступать интерфейсы, так как один класс одновременно может реализовывать несколько интерфейсов.

К сожалению, в PHP 5 иногда все же придется прибегать к агрегации там, где в других языках применяется множественное наследование. *Агрегация* — создание цепочки классов-наследников, предназначенной для "собирания" (или агрегирования) свойств и методов нескольких классов в одном, стоящем в конце цепочки. Такой подход имеет один существенный недостаток — агрегирование может порождать множество классов, которые впоследствии не будут использоваться в программе, "захламляя" код и отрицательно сказываясь на его качестве.

Пока объектная модель PHP еще далека от совершенства, но в 5-й версии был сделан серьезный шаг для улучшения поддержки ООП.