ГЛАВА 3

Основные положения об объектах

В данной книге основное внимание уделяется объектам и классам, поскольку с появлением версии 5 более десяти лет назад они стали основными элементами языка PHP. В этой главе будет заложен прочный фундамент для дальнейшей работы с объектами, описаны подходы к проектированию на основе исследования объектно-ориентированных языковых средств PHP. Если объектно-ориентированное программирование — новая для вас область, постарайтесь очень внимательно прочитать эту главу.

В этой главе рассматриваются следующие вопросы.

- *Классы и объекты*. Объявление классов и создание экземпляров объектов.
- *Методы-конструкторы*. Автоматизация установки начальных значений объектов.
- *Элементарные типы и классы*. Почему тип имеет особое значение.
- *Наследование*. Зачем нужно наследование и как его использовать.
- *Видимость*. Упрощение интерфейсов объектов и защита методов и свойств от вмешательства извне.

Классы и объекты

Первым препятствием для понимания ООП служит странная и удивительная связь между классом и объектом. Для многих людей именно эта связь становится первым моментом откровения, первой искрой интереса к ООП. Поэтому давайте уделим должное внимание самим основам.

Первый класс

Классы часто описывают с помощью объектов. И это весьма любопытно, потому что объекты часто описывают с помощью классов. Такое хождение по

кругу может сильно затруднить первые шаги в ООП. Именно классы определяют объекты, и поэтому начать следует с определения классов.

Короче говоря, *класс* — это шаблон кода, применяемый для создания объектов. Класс объявляется с помощью ключевого слова `class` и произвольного имени класса. В именах классов может использоваться любое сочетание букв и цифр, но они не должны начинаться с цифры. Код, связанный с классом, должен быть заключен в фигурные скобки. Объединим эти элементы вместе, чтобы создать класс следующим образом:

```
// Листинг 03.01

class ShopProduct
{
   // Тело класса
}
```

Класс `ShopProduct` из данного примера уже является полноправным, хотя и не слишком полезным пока еще. Тем не менее мы совершили нечто очень важное, определив тип, или категорию данных, чтобы использовать их в своих сценариях. Важность такого шага станет для вас очевидной по ходу дальнейшего чтения этой главы.

Несколько первых объектов

Если класс — это шаблон для создания объектов, то объект — это данные, которые структурируются по шаблону, определенному в классе. И в этом случае говорят, что объект — это экземпляр класса. Его тип определяется классом.

Итак, воспользуемся классом `ShopProduct` как шаблоном для создания объектов типа `ShopProduct`. Для этого нам потребуется оператор `new`, за которым указывается имя класса, как показано ниже.

```
// Листинг 03.02

$product1 = new ShopProduct();
$product2 = new ShopProduct();
```

После оператора `new` указывается имя класса в качестве его единственного операнда. В итоге создается экземпляр этого класса; в данном примере — объект типа `ShopProduct`.

Мы воспользовались классом `ShopProduct` как шаблоном для создания двух объектов типа `ShopProduct`. И хотя функционально объекты `$product1` и `$product2` идентичны (т.е. пусты), тем не менее, это два разных объекта одного типа, созданные с помощью одного класса.

Если вам все еще непонятно, обратимся к аналогии. Представьте, что класс — это форма для отливки, с помощью которой изготавливают пластмассовые утки, а объекты — отливаемые утки. Тип создаваемых объектов определяется формой отливки. Утки выглядят одинаковыми во всех отношениях, но

все-таки это разные предметы. Иными словами, это разные экземпляры одного и того же типа. У отдельных уток могут быть даже разные серийные номера, подтверждающие их индивидуальность. Каждому объекту, создаваемому в сценарии на PHP, присваивается также идентификатор, однозначный в течение времени существования объекта. Это означает, что в PHP идентификаторы объектов повторно используются даже в пределах одного и того же процесса, где выполняется сценарий. Эту особенность можно продемонстрировать, выведя объекты `$product1` и `$product2` на печать следующим образом:

```
// Листинг 03.03

var_dump($product1);
var_dump($product2);
```

В результате вызовов приведенной выше функции на экран будет выведена следующая информация:

```
object(ShopProduct)#1 (0) {
}
object(ShopProduct)#2 (0) {
}
```

НА ЗАМЕТКУ. В версиях PHP 4 и PHP 5 (до версии 5.1 включительно) объекты можно выводить на печать непосредственно. В итоге объект будет приведен к символьной строке, содержащей его идентификатор. Но, начиная с версии PHP 5.2, такая возможность больше не поддерживается, и любая попытка интерпретировать объект как символьную строку приведет к ошибке, если только в классе этого объекта не будет определен метод **`__toString()`**[1]. Методы будут рассмотрены далее в этой главе, а метод **`__toString()`** — в главе 4.

Передав объекты функции `var_dump()`, можно извлечь полезные сведения о них, включая внутренний идентификатор каждого объекта, указанный после символа `'#'`. Чтобы сделать рассматриваемые здесь объекты более интересными, придется немного изменить определение класса `ShopProduct`, добавив в него специальные поля данных, называемые *свойствами*.

Установка свойств в классе

В классах можно определять специальные переменные, которые называются *свойствами*. Свойство, называемое также *переменной-членом*, содержит данные, которые могут меняться в разных объектах. Так, для объектов типа `ShopProduct` требуется возможность изменять поля, содержащие название товара и его цену.

[1] Обратите внимание на то, что имя метода начинается с двух знаков подчеркивания. — *Примеч. ред.*

Определение свойства в классе похоже на определение обычной переменной, за исключением того, что в операторе объявления перед именем свойства следует указать одно из ключевых слов, характеризующих область его видимости: `public`, `protected` или `private`.

НА ЗАМЕТКУ. Область видимости определяет контекст функции или класса, где можно пользоваться данной переменной (свойством или методом, о чем пойдет речь далее в этой главе). Так, у переменной, определенной в теле функции, имеется локальная область видимости, а у переменной, определенной за пределами функции, — глобальная область видимости. Как правило, доступ к данным, находящимся в более локальных областях видимости, чем текущая область, получить нельзя. Поэтому, определив переменную в теле функции, вряд ли удастся впоследствии получить к ней доступ за пределами этой функции. Объекты в этом смысле более "проницаемы", и к некоторым объектным переменным можно иногда получать доступ из другого контекста. Как поясняется далее, порядок доступа к переменным из конкретного контекста определяется ключевыми словами **`public`**, **`protected`** и **`private`**.

Мы еще вернемся к обсуждению области видимости и определяющим ее ключевым словам далее в этой главе. А до тех пор определим некоторые свойства с помощью ключевого слова `public`.

```
// Листинг 03.04

class ShopProduct
{
   public $title             = "Стандартный товар";
   public $producerMainName  = "Фамилия автора";
   public $producerFirstName = "Имя автора";
   public $price             = 0;
}
```

Как видите, мы определили четыре свойства, присвоив каждому из них стандартное значение. Теперь любым объектам, экземпляры которых получаются с помощью класса `ShopProduct`, будут присваиваться стандартные данные. А ключевое слово `public`, присутствующее в объявлении каждого свойства, обеспечит доступ к этому свойству за пределами контекста его объекта.

К переменным свойств можно обращаться с помощью операции, обозначаемой знаками `'->'`, указав имя объектной переменной и имя свойства, как показано ниже.

```
// Листинг 03.05

$product1 = new ShopProduct();
print $product1->title;
```

В результате выполнения приведенных выше строк кода будет выведено следующее:

```
Стандартный товар
```

Свойства объектов были определены как public, и поэтому их значения можно прочитать, присвоить им новые значения, заменив тем самым набор стандартных значений, устанавливаемых в классе по умолчанию.

```
// Листинг 03.06

$product1 = new ShopProduct();
$product2 = new ShopProduct();

$product1->title = "Собачье сердце";
$product2->title = "Ревизор";
```

Объявляя и устанавливая свойство $title в классе ShopProduct, мы гарантируем, что при создании любого объекта типа ShopProduct это свойство будет присутствовать и его значение будет заранее определено. Это означает, что в том коде, где используется данный класс, можно будет работать с любыми объектами типа ShopProduct. Но поскольку свойство $title можно легко переопределить, то его значение может меняться в зависимости от конкретного объекта.

НА ЗАМЕТКУ. Код, в котором используется класс, функция или метод, обычно называется клиентом класса, функции или метода или просто *клиентским кодом*. Этот термин будет часто встречаться в последующих главах.

На самом деле в PHP необязательно объявлять все свойства в классе. Свойства можно динамически добавлять в объект следующим образом:

```
// Листинг 03.07

$product1->arbitraryAddition = "Дополнительный параметр";
```

Следует, однако, иметь в виду, что такой способ назначения свойств для объектов считается неудачной нормой практики в ООП и почти никогда не применяется. А почему такая норма практики считается неудачной? Дело в том, что при создании класса вы определяете конкретный тип данных. Тем самым вы сообщаете всем, что ваш класс (и любой объект, который является его экземпляром) содержит определенный набор полей и функций. Если в классе ShopProduct определяется свойство $title, то в любом коде, манипулирующем объектами типа ShopProduct, предполагается, что свойство $title определено. Но подобной гарантии в отношении свойств, устанавливаемых динамически, дать нельзя.

Созданные нами объекты пока еще производят довольно тягостное впечатление. Когда нам потребуется манипулировать свойствами объекта, это придется делать за пределами данного объекта. Следовательно, нужно каким-то образом устанавливать и получать значения свойств объекта. Установка нескольких свойств в целом ряде объектов очень быстро становится довольно хлопотным делом, как показано ниже.

```
// Листинг 03.08

$product1->title             = "Собачье сердце";
$product1->producerMainName  = "Булгаков";
$product1->producerFirstName = "Михаил";
$product1->price             = 5.99;
```

Здесь снова используется класс `ShopProduct` и все стандартные значения его свойств переопределяются одно за другим до тех пор, пока не будут заданы все сведения о товаре. А теперь, когда у нас имеются некоторые данные, к ним можно обратиться следующим образом:

```
// Листинг 03.09

print "Автор: {$product1->producerFirstName} "
    . "{$product1->producerMainName}\n";
```

Выполнение данного фрагмента кода приведет к следующему результату:

```
Автор: Михаил Булгаков
```

У такого подхода к определению значений свойств имеется ряд недостатков. В языке PHP допускается определять свойства динамически, и поэтому вы не получите предупреждения, если забудете имя свойства, или сделаете в нем опечатку. Например, вместо следующей строки кода:

```
$product1->producerMainName = "Булгаков";
```

можно случайно ввести такую строку:

```
$product1->producerSecondName = "Булгаков";
```

С точки зрения интерпретатора PHP этот код совершенно корректен, поэтому никакого предупреждения об ошибке мы не получим. Но когда понадобится распечатать имя автора, мы получим неожиданные результаты.

Еще одно затруднение состоит в том, что мы слишком “нестрого” определили свой класс, в котором совсем не обязательно нужно указывать название книги, цену или фамилию автора. Клиентский код может быть уверен, что эти свойства существуют, но зачастую их исходно устанавливаемые стандартные значения вряд ли будут его устраивать. В идеале следовало бы побуждать всякого, кто создает экземпляры объекта типа `ShopProduct`, задавать вполне осмысленные значения его свойств.

И, наконец, нам придется приложить немало усилий, чтобы сделать то, что, вероятнее всего, придется делать очень часто. Как было показано выше, распечатать полное имя автора — дело весьма хлопотное. И было бы неплохо, если бы объект делал это вместо нас. Все эти затруднения можно разрешить, если снабдить объект типа `ShopProduct` собственным набором функций, чтобы пользоваться ими для манипулирования данными свойств в контексте самого объекта.

Работа с методами

Если свойства позволяют объектам сохранять данные, то методы —выполнять конкретные задачи. *Методы* — это специальные функции, объявляемые в классе. Как и следовало ожидать, объявление метода напоминает объявление функции. После ключевого слова `function` указывается имя метода, а вслед за ним — необязательный список переменных-аргументов в круглых скобках. Тело метода заключается в фигурные скобки, как показано ниже.

```
public function myMethod($argument, $another)
{
  // ...
}
```

В отличие от функций, методы следует объявлять в теле класса. При этом можно указывать также ряд спецификаторов, в том числе ключевое слово, определяющее область видимости метода. Как и свойства, методы можно определять как `public`, `protected` или `private`. Объявляя метод как `public`, мы тем самым обеспечиваем возможность его вызова за пределами текущего объекта. Если в определении метода опустить ключевое слово, определяющее область его видимости, то метод будет объявлен неявно как `public`. К модификаторам методов мы еще вернемся далее в этой главе.

```
// Листинг 03.10

class ShopProduct
{
   public $title             = "Стандартный товар";
   public $producerMainName  = "Фамилия автора";
   public $producerFirstName = "Имя автора";
   public $price             = 0;

   public function getProducer()
   {
     return $this->producerFirstName . " "
          . $this->producerMainName;
   }
}
```

Зачастую метод вызывается с помощью объектной переменной, после которой указываются знаки `'->'` и имя метода. При вызове метода следует указывать круглые скобки, как и при вызове функции (даже если методу вообще не передаются аргументы).

```
// Листинг 03.11

$product1 = new ShopProduct();

$product1->title             = "Собачье сердце";
$product1->producerMainName  = "Булгаков";
```

```
$product1->producerFirstName = "Михаил";
$product1->price             = 5.99;

print "Автор: {$product1->getProducer()}\n";
```

Выполнение данного фрагмента кода приведет к следующему результату:

```
Автор: Михаил Булгаков
```

Итак, мы ввели метод `getProducer()` в класс `ShopProduct`. Обратите внимание на то, что этот метод был определен как открытый (`public`), а следовательно, его можно вызывать за пределами класса `ShopProduct`.

В теле метода `getProducer()` мы воспользовались новым средством — псевдопеременной `$this`. С ее помощью реализуется механизм доступа к экземпляру объекта из кода класса. Чтобы легче уяснить принцип действия такого механизма, попробуйте заменить выражение `$this` "текущим экземпляром объекта". Например, оператор

```
$this->producerFirstName
```

означает:

```
Свойство $producerFirstName текущего экземпляра объекта
```

Таким образом, метод `getProducer()` объединяет и возвращает значения свойств `$producerFirstName` и `$producerMainName`, избавляя нас от лишних хлопот всякий раз, когда требуется распечатать полное имя автора.

Хотя нам удалось немного усовершенствовать наш класс, ему по-прежнему присуща излишняя гибкость. Мы все еще рассчитываем на то, что программист будет изменять стандартные значения свойств объекта типа `ShopProduct`. Но это затруднительно в двух отношениях. Во-первых, потребуется пять строк кода, чтобы должным образом инициализировать объект типа `ShopProduct`, и ни один программист не поблагодарит нас за это. И во-вторых, мы никак не можем гарантировать, что какое-нибудь свойство будет определено при инициализации объекта типа `ShopProduct`. Поэтому нам потребуется метод, который будет вызываться автоматически при создании экземпляра объекта из его класса.

Создание метода-конструктора

Метод-конструктор вызывается при создании объекта. Он служит для настройки экземпляра объекта, установки определенных значений его свойств и выполнения всей подготовительной работы к применению объекта.

НА ЗАМЕТКУ. До версии PHP 5 имя метода-конструктора совпадало с именем класса, к которому оно относилось. Так, в качестве конструктора класса **`ShopProduct`** можно было пользоваться методом **`ShopProduct()`**. Теперь такой синтаксис вообще не работает и считается устаревшим, начиная с версии PHP 7. Поэтому метод-конструктор следует называть как **`__construct()`**.

Обратите внимание на то, что имя метода-конструктора начинается с двух символов подчеркивания. Это правило именования распространяется и на многие другие специальные методы в классах PHP. Теперь определим конструктор для класса `ShopProduct` следующим образом:

```
// Листинг 03.12

class ShopProduct
{
   public $title;
   public $producerMainName;
   public $producerFirstName;
   public $price = 0;

  public function __construct(
      $title,
      $firstName,
      $mainName,
      $price
  ) {
      $this->title = $title;
      $this->producerFirstName = $firstName;
      $this->producerMainName = $mainName;
      $this->price = $price;
    }

  public function getProducer()
  {
    return $this->producerFirstName . " "
        . $this->producerMainName;
  }
}
```

И снова мы вводим в класс новые функциональные возможности, стараясь сэкономить время и силы программиста и избавить его от необходимости дублировать код, работающий с классом `ShopProduct`. Метод `__construct()` автоматически вызывается при создании объекта с помощью оператора `new`, как показано ниже.

```
// Листинг 03.13

$product1 = new ShopProduct(
     "Собачье сердце",
     "Михаил",
     "Булгаков",
      5.99
);

print "Автор: {$product1->getProducer()}\n";
```

Выполнение данного фрагмента кода приведет к следующему результату:

```
Автор: Михаил Булгаков
```

Значения всех перечисленных аргументов передаются конструктору. В данном примере конструктору передается название книги, Ф.И.О. автора и цена. Для присвоения значений соответствующим свойствам объекта в методе-конструкторе применяется псевдопеременная `$this`.

НА ЗАМЕТКУ. Теперь получать экземпляры и пользоваться объектом типа **`ShopProduct`** стало безопаснее и легче. Ведь получение экземпляров и установка значений свойств выполняются в одном операторе. При написании любого кода, где используется объект типа **`ShopProduct`**, можно быть уверенным, что все свойства этого объекта будут инициализированы.

Очень важным аспектом ООП является предсказуемость. Чтобы сделать объект безопасным, достаточно, например, воспроизвести предсказуемые типы данных, хранящиеся в его свойствах. При этом можно гарантировать, что в свойстве `$name`, например, всегда будут находиться только символьные данные. Но как этого добиться, если данные для инициализации свойств поступают в виде параметров из вне класса? В следующем разделе мы изучим механизм, который можно использовать для явного определения типов объектов при объявлении методов.

Аргументы и типы

Типы определяют порядок оперирования данными в сценариях. Например, строковый тип используется для хранения и отображения символьных данных, а также для выполнения операций над такими данными с помощью строковых функций. Целые числа применяются в математических выражениях, булевы значения — в логических выражениях и т.д. Эти категории называются *элементарными*, или *простыми* типами данных (primitive types). Имя класса также определяет тип данных, но на более высоком уровне. Поэтому объект класса `ShopProduct` относится как к простому типу `object`, так и к типу самого класса `ShopProduct`. В этом разделе мы рассмотрим обе разновидности типов данных по отношению к методам класса.

При определении метода и функции не требуется, чтобы аргумент относился к конкретному типу. Но в этом одновременно заключается преимущество и недостаток. С одной стороны, принадлежность аргумента к любому типу данных доставляет немало удобств. Благодаря этому можно создавать методы, которые будут гибко реагировать на данные различных типов и приспосабливать свои функциональные возможности к меняющимся обстоятельствам. Но, с другой стороны, такая гибкость может стать причиной неопределенности в коде, когда в теле метода ожидается один тип аргумента, а передается другой.

Простые типы данных

PHP является слабо типизированным языком, а это означает, что объявлять тип данных, который должен храниться в переменной, не нужно. Так, в пределах одной и той же области видимости переменная `$number` может содержать как числовое значение `2`, так и символьную строку `"two"` (“два”). В таких

строго типизированных языках программирования, как C или Java, вы обязаны определить тип переменной до присваивания ей значения, и, конечно, это значение должно быть указанного типа.

Но это совсем не означает, что в PHP отсутствует понятие типа. Каждое значение, которое можно присвоить переменной, имеет свой тип данных. В PHP тип значения переменной можно определить с помощью одной из функций для проверки типов. В табл. 3.1 перечислены простые типы данных, существующие в PHP, а также соответствующие им проверочные функции. Каждой функции передается переменная или значение, а она возвращает значение `true` ("истина"), если аргумент относится к соответствующему типу данных. Проверка типа переменной особенно важна при обработке аргументов в методе или функции.

Таблица 3.1. Простые типы данных в PHP и их проверочные функции

Проверочная функция	Тип	Описание
`is_bool()`	`Boolean`	Одно из двух логических значений: **`true`** или **`false`** (истина или ложь)
`is_integer()`	`Integer`	Целое число; является псевдонимом функций **`is_int()`** и **`is_long()`**
`is_double()`	`Double`	Число с плавающей (десятичной) точкой; является псевдонимом функции **`is_float()`**
`is_string()`	`String`	Символьные данные
`is_object()`	`Object`	Объект
`is_array()`	`Array`	Массив
`is_resource()`	`Resource`	Дескриптор, используемый для идентификации и работы с такими внешними ресурсами, как базы данных или файлы
`is_null()`	`Null`	Неопределенное значение

Пример использования простых типов данных

Имейте в виду, что вы должны внимательно следить за типами данных в своем коде. Рассмотрим пример одного из многих затруднений, с которыми вы можете столкнуться, используя типы данных.

Допустим, требуется извлечь параметры конфигурации из XML-файла. Элемент `<resolvedomains>` разметки XML-документа сообщает приложению, следует ли пытаться преобразовывать IP-адреса в доменные имена. Следует заметить, что такое преобразование полезно, хотя и является относительно медленной операцией. Ниже приведен фрагмент кода XML для данного примера.

```
<!-- Листинг 03.14 -->

<settings>
   <resolvedomains>false</resolvedomains>
</settings>
```

Приложение извлекает символьную строку "false" и передает ее в качестве параметра методу outputAddresses(), который выводит информацию об IP-адресах. Ниже приведено определение метода outputAddresses().

```
// Листинг 03.15

class AddressManager
{
  private $addresses = ["209.131.36.159", "216.58.213.174"];

  public function outputAddresses($resolve)
  {
    foreach ($this->addresses as $address) {
      print $address;
      if ($resolve) {
        print " (".gethostbyaddr($address).")";
      }
      print "<br />\n";
    }
  }
}
```

Разумеется, в класс AddressManager можно внести ряд улучшений. В частности, жестко кодировать IP-адреса в виде массива непосредственно в классе не очень удобно. Тем не менее метод outputAddresses() циклически обходит массив IP-адресов и выводит его элементы по очереди. Если значение аргумента $resolve равно true, то данный метод выводит не только IP-адреса, но и соответствующие им доменные имена.

Ниже приведен один из возможных вариантов применения класса Address Manager вместе с дескриптором settings в элементе XML-разметки файла конфигурации. Попробуйте обнаружить ошибку в приведенном ниже фрагменте кода.

```
// Листинг 03.16

$settings = simplexml_load_file(__DIR__."/resolve.xml");
$manager = new AddressManager();
$manager->outputAddresses( (string) $settings->resolvedomains);
```

В этом фрагменте кода для получения значения из элемента разметки resolvedomains применяется интерфейс SimpleXML API. В рассматриваемом здесь примере нам известно, что это значение представляет собой текстовый элемент "false", и поэтому оно приводится к строковому типу, поскольку именно так рекомендуется поступать в документации по SimpleXML API.

Но анализируемый здесь код поведет себя совсем не так, как мы того ожидали. Передавая символьную строку "false" методу outputAddresses(), мы не знаем, какой именно тип аргумента используется в этом методе по умолчанию. Данный метод ожидает получить логическое значение аргумента (true или false). При выполнении условного оператора символьная строка "false" на самом деле преобразуется в логическое значение true. Дело в том, что при

выполнении проверки интерпретатор PHP с готовностью преобразует непустое строковое значение в логическое значение `true`. Поэтому следующий фрагмент кода:

```
if ( "false" ) {
  // ...
}
```

равнозначен такому фрагменту кода:

```
if ( true ) {
  // ...
}
```

Исправить этот недостаток можно по-разному. Во-первых, можно сделать метод `outputAddresses()` менее требовательным, чтобы он распознавал символьную строку и применял некоторые основные правила для ее преобразования в логический эквивалент:

```
// Листинг 03.17

public function outputAddresses($resolve)
{
  if (is_string($resolve)) {
    $resolve = (preg_match("/^(false|no|off)$/i", $resolve))
               ? false : true;
  }
  // ...
}
```

Но с точки зрения проектирования существуют достаточно веские основания избегать приведенного выше решения. Вообще говоря, при проектировании методов или функций лучше предусматривать для них строгий интерфейс, исключающий двусмысленность, вместо неясного и нетребовательного интерфейса. Подобные решения так или иначе вызывают путаницу и порождают ошибки.

Во-вторых, можно оставить метод `outputAddresses()` без изменений, дополнив его комментариями с четкими инструкциями, что аргумент `$resolve` должен содержать логическое значение. Такой подход позволяет предупредить программиста, что нужно внимательно читать инструкции, а иначе пенять на себя.

```
/**
  * Вывести список адресов.
  * Если переменная $resolve содержит истинное
  * значение (true), то адрес преобразуется в
  * эквивалентное имя хоста.
  * @param $resolve Boolean Преобразовать адрес?
  */
function outputAddresses( $resolve ) {
   // ...
}
```

Это вполне разумное решение, если вы точно уверены в том, что программисты, пользующиеся вашим классом, добросовестно прочтут документацию к нему. И наконец, в-третьих, можно сделать метод `outputAddresses()` строгим в отношении типа данных аргумента `$resolve`. Для таких простых типов данных, как логические значения, до выпуска версии PHP 7 это можно было сделать лишь одним способом: написать код для проверки входных данных и предпринять соответствующее действие, если они не отвечают требуемому типу:

```
function outputAddresses($resolve)
{
  if (! is_bool($resolve)) {
     // принять решительные меры
  }
  //...
}
```

Такой подход вынуждает клиентский код предоставить корректный тип данных для аргумента `$resolve`.

НА ЗАМЕТКУ. В следующем далее разделе "Уточнение типов объектов" описывается намного более совершенный способ наложения ограничений на типы аргументов, передаваемых методам и функциям.

Преобразование строкового аргумента внутри метода — более дружественный подход с точки зрения клиента, но, вероятно, он может вызвать другие проблемы. Обеспечивая механизм преобразования в методе, мы предугадываем контекст его использования и намерение клиента. С другой стороны, соблюдая логический тип данных, мы предоставляем клиенту право выбора: преобразовывать ли символьные строки в логические значения и решить, какое слово должно соответствовать логическому значению **`true`** или **`false`**. А между тем метод **`outputAddresses()`** разрабатывался с целью решить конкретную задачу. Такой акцент на выполнении конкретной задачи при *намеренном игнорировании более широкого контекста* является важным принципом ООП, к которому мы будем еще не раз возвращаться в данной книге.

На самом деле стратегии обращения с типами аргументов зависят от степени серьезности возможных ошибок. Большинство значений простых типов данных в PHP автоматически приводятся к нужному типу в зависимости от конкретного контекста. Так, если числа, присутствующие в символьных строках, используются в математических выражениях, они автоматически преобразуются в эквивалентные целые значения или же значения с плавающей точкой. В итоге прикладной код может быть нетребовательным к ошибкам несоответствия типов данных. Но если в качестве одного из аргументов метода предполагается массив, то следует проявить бо́льшую, чем обычно, внимательность. Передача значения, не являющегося массивом, одной из функций обработки массивов в PHP не приведет ни к чему хорошему и вызовет ряд ошибок в разрабатываемом методе. Поэтому вам придется найти нечто среднее между проверкой типа,

преобразованием одного типа данных в другой и опорой на понятную документацию, которую вы обязаны предоставить независимо от выбранного вами способа.

Каким бы способом вы ни решали подобного рода затруднения в своем коде, можете быть уверены лишь в одном: тип аргумента всегда имеет особое значение! Это значение становится еще более весомым в связи с тем, что PHP не является строго типизированным языком. Здесь нельзя полагаться на компилятор в вопросах предотвращения ошибок, связанных с нарушением типов данных. Поэтому вы должны предусмотреть возможные последствия того, что типы аргументов окажутся не такими, как предполагалось. Не стоит надеяться, что программисты клиентского кода угадают ход ваших мыслей. Вы должны всегда учитывать, что в разрабатываемые вами методы будут передаваться входные данные некорректного типа.

Уточнение типов объектов

Как упоминалось ранее, переменная аргумента может иметь любой простой тип данных, но по умолчанию ее тип не оговаривается, и поэтому она может содержать объект любого типа. С одной стороны, такая гибкость удобна, а с другой — она может стать причиной осложнений при определении метода.

Рассмотрим в качестве примера метод, предназначенный для работы с объектом типа `ShopProduct`.

```
// Листинг 03.18

class ShopProductWriter
{
  public function write($shopProduct)
  {
    $str = $shopProduct->title . ": "
         . $shopProduct->getProducer()
         . " (" . $shopProduct->price . ")\n";
    print $str;
  }
}
```

Мы можем протестировать работу этого класса следующим образом:

```
// Листинг 03.19

$product1 = new ShopProduct("Собачье сердце",
                            "Михаил", "Булгаков", 5.99);
$writer = new ShopProductWriter();
$writer->write($product1);
```

В итоге получим следующий результат:

```
Собачье сердце: Михаил Булгаков (5.99)
```

Класс ShopProductWriter содержит единственный метод write(). Методу write() передается объект типа ShopProduct. Свойства и методы последнего используются в нем для создания и вывода результирующей строки с описанием товара. Мы используем имя переменной аргумента $shopProduct как напоминание программисту, что методу $write() следует передать объект типа ShopProduct, хотя соблюдать это требование необязательно. Это означает, что программист может передать методу $write() некорректный объект или вообще данные простого типа и ничего об этом не узнать до момента обращения к аргументу $shopProduct. К тому времени в нашем коде уже могут быть выполнены какие-либо действия, поскольку предполагалось, что методу write() был передан настоящий объект типа ShopProduct.

НА ЗАМЕТКУ. В связи с изложенным выше у вас может возникнуть вопрос: почему мы не ввели метод **write()** непосредственно в класс **ShopProduct**? Все дело в ответственности. Класс **ShopProduct** отвечает за хранение данных о товаре, а класс **ShopProductWriter** — за вывод этих данных. По мере чтения этой главы вы начнете постепенно понимать, в чем польза такого разделения ответственности.

Для решения упомянутой выше проблемы в PHP 5 были добавлены объявления типов классов, называвшиеся уточнениями типов аргументов. Чтобы добавить объявление типа класса к аргументу метода, достаточно указать перед ним имя класса. Поэтому в метод write() можно внести следующие коррективы:

```
// Листинг 03.20

public function write(ShopProduct $shopProduct)
{
  // ...
}
```

Теперь методу write() можно передавать аргумент $shopProduct, содержащий только объект типа ShopProduct. В следующем фрагменте кода предпринимается попытка "перехитрить" метод write(), передав ему объект другого типа:

```
// Листинг 03.21

class Wrong
{
}

$writer = new ShopProductWriter();
$writer->write(new Wrong());
```

Метод write() содержит объявление типа класса, и поэтому передача ему объекта типа Wrong приведет к неустранимой ошибке, как показано ниже.

```
TypeError: Argument 1 passed to ShopProductWriter::write() must be an instance of
ShopProduct, instance of Wrong given, called in Runner.php on ...[2]
```

Теперь, вызывая метод `write()`, совсем не обязательно проверять тип передаваемого ему аргумента. Это делает сигнатуру метода намного более понятной для программиста клиентского кода. Он сразу же увидит требования, предъявляемые к вызову метода `write()`. Ему не нужно будет беспокоиться по поводу скрытых ошибок, возникающих в результате несоответствия типа аргумента, поскольку объявление типа класса соблюдается строго.

Несмотря на то что автоматическая проверка типов данных служит превосходным средством для предотвращения ошибок, важно понимать, что объявления типов классов проверяются во время выполнения программы. Это означает, что проверяемое объявление типа класса сообщит об ошибке только тогда, когда методу будет передан нежелательный объект. И если вызов метода `write()` глубоко скрыт в условном операторе, который выполняется только на Рождество, то вам придется поработать на праздники, если вы тщательно не проверите свой код.

Имея теперь в своем распоряжении объявления скалярных типов, можно наложить некоторые ограничения на класс `ShopProduct` следующим образом:

```
// Листинг 03.22

class ShopProduct
{
  public $title;
  public $producerMainName;
  public $producerFirstName;
  public $price = 0;

  public function __construct(
      string $title,
      string $firstName,
      string $mainName,
      float $price
  ) {
      $this->title = $title;
      $this->producerFirstName = $firstName;
      $this->producerMainName = $mainName;
      $this->price = $price;
    }

    // ...
}
```

Укрепив подобным образом метод-конструктор, можно гарантировать, что аргументы `$title`, `$firstName`, `$mainName` будут всегда содержать строковые

[2] Ошибка нарушения типа данных: аргумент 1, переданный методу `ShopProduct Writer::write()`, должен быть экземпляром класса `ShopProduct`, а переданный экземпляр класса `Wrong` некорректен, вызвано из исходного файла `Runner.php` ...

данные, тогда как аргумент `$price` — числовое значение с плавающей точкой. В этом можно убедиться, попытавшись получить экземпляр класса `ShopProduct` с неверными данными:

```
// Листинг 03.23

// Не сработает!
$product = new ShopProduct("Название", "Имя", "Фамилия", []);
```

В данном случае предпринимается попытка получить экземпляр класса `ShopProduct`. Конструктору этого класса передаются три символьные строки и пустой массив вместо требующегося числового значения с плавающей точкой, что в конечном итоге приведет к неудачному исходу. Благодаря объявлениям типов данных интерпретатор PHP не допустит такого, выдав следующее сообщение об ошибке:

```
TypeError: Argument 4 passed to ShopProductWriter::write() must be of the type
float, array given, called in ...[3]
```

По умолчанию интерпретатор PHP выполнит там, где это возможно, неявное приведение значений аргументов к требующимся типам данных. И это характерный пример упоминавшегося ранее противоречия между безопасностью и гибкостью прикладного кода. Так, в новой реализации класса `ShopProduct` символьная строка будет автоматически преобразована в числовое значение с плавающей точкой, и поэтому следующая попытка получить экземпляр данного класса, где символьная строка `"4.22"` преобразуется внутренним образом в числовое значение `4.22`, не вызовет проблем:

```
// Листинг 03.24

$product = new ShopProduct("Название", "Имя", "Фамилия", "4.22");
```

Все это, конечно, замечательно, но вернемся к затруднению, возникшему в связи с применением класса `AddressManager`, где символьная строка `"false"` негласно преобразовывалась в логическое значение. По умолчанию это преобразование будет происходить по-прежнему, если применить объявление логического типа данных в методе `AddressManager::outputAddresses()` следующим образом:

```
// Листинг 03.25

public function outputAddresses(bool $resolve)
{
  // ...
}
```

[3] Ошибка нарушения типа данных: аргумент 4, переданный методу `ShopProduct Writer::write()`, должен быть числовым типом с плавающей точкой, а переданный массив некорректен, вызвано из ...

А теперь рассмотрим следующий вызов, где методу `outputAddresses()` передается символьная строка:

```
// Листинг 03.26

$manager->outputAddresses("false");
```

Вследствие неявного приведения типов этот вызов функционально равнозначен вызову данного метода с логическим значением `true`.

Объявления скалярных типов данных можно сделать строгими, хотя только на уровне отдельных исходных файлов. В следующем примере кода устанавливаются строгие объявления типов данных, а метод `outputAddresses()` снова вызывается с символьной строкой:

```
// Листинг 03.27

declare(strict_types=1);
$manager->outputAddresses("false");
```

Этот вызов приведет к появлению следующей ошибки типа `TypeError` из-за строгих объявлений типов данных:

```
TypeError: Argument 4 passed to AddressManager::outputAddresses() must be of
the type boolean, string given, called in ...[4]
```

НА ЗАМЕТКУ. Объявление **`strict_types`** применяется к тому исходному файлу, откуда делается вызов, а не к исходному файлу, где реализована функция или метод. Поэтому соблюдение строгости такого объявления возлагается на клиентский код.

Иногда аргумент требуется сделать необязательным и, тем не менее, наложить ограничение на его тип, если он все же указывается. Для этого достаточно указать стандартное значение аргумента, устанавливаемое по умолчанию, как показано ниже.

```
// Листинг 03.28

class ConfReader
{
  public function getValues(array $default = null)
  {
    $values = [];

    // Выполнить действия для получения новых значений

    // Объединить полученные значения со стандартными
    // (результат всегда будет находиться в массиве)
```

4 Ошибка нарушения типа данных: аргумент 4, переданный методу `AddressManager::outputAddresses()`, должен быть логическим типом, а переданная символьная строка некорректна, вызвано из ...

```
        $values = array_merge($default, $values);
        return $values;
    }
}
```

Объявления типов, которые поддерживаются в PHP, перечислены в табл. 3.2.

Таблица 3.2. Объявления типов данных в PHP

Объявляемый тип данных	Версия, начиная с которой поддерживается	Описание
`array`	5.1	Массив. По умолчанию может быть пустым значением или массивом
`int`	7.0	Целое значение. По умолчанию может быть пустым или целым значением
`float`	7.0	Числовое значение с плавающей (десятичной) точкой. Допускается целое значение, даже если активизирован строгий режим. По умолчанию может быть пустым, целым или числовым значением с плавающей точкой
`callable`	5.4	Вызываемый код (например, анонимная функция). По умолчанию может быть пустым значением
`bool`	7.0	Логическое значение. По умолчанию может быть пустым или логическим значением
`string`	5.0	Символьные данные. По умолчанию может быть пустым или строковым значением
`self`	5.0	Ссылка на содержащий класс
`[тип класса]`	5.0	Тип класса или интерфейса. По умолчанию может быть пустым значением

При описании объявлений типов классов здесь подразумевается, что типы и классы являются синонимами, хотя у них имеются существенные отличия. При определении класса определяется также тип, но он может описывать целое семейство классов. Механизм, посредством которого различные классы можно группировать под одним типом, называется *наследованием*. О наследовании речь пойдет в следующем разделе.

Наследование

Наследование — это механизм, посредством которого один или несколько классов можно получить из некоторого базового класса. Класс, унаследованный от другого класса, называется его *подклассом*. Эта связь обычно описывается с помощью терминов *родительский* и *дочерний*. В частности, дочерний класс

происходит от родительского и наследует его характеристики, состоящие из свойств и методов. Обычно функциональные возможности родительского класса, который называется также *суперклассом*, дополняются в дочернем классе новыми функциональными возможностями. Поэтому говорят, что дочерний класс расширяет родительский. Прежде чем приступить к исследованию синтаксиса наследования, рассмотрим проблемы, которые оно поможет нам решить.

Проблема наследования

Вернемся к классу `ShopProduct`, который в настоящий момент является достаточно обобщенным. С его помощью можно манипулировать самыми разными товарами, как показано ниже.

```
// Листинг 03.29

$product1 = new ShopProduct( "Собачье сердце",
                             "Михаил", "Булгаков", 5.99 );

$product2 = new ShopProduct( "Классическая музыка. Лучшее",
                             "Антонио", "Вивальди", 10.99 );

print "Автор: "       .$product1->getProducer() . "<br />\n";
print "Исполнитель: " .$product2->getProducer() . "<br />\n";
```

Выполнение данного фрагмента кода приведет к следующему результату:

```
Автор: Михаил Булгаков
Исполнитель: Антонио Вивальди
```

Как видите, разделение имени автора на две части очень хорошо подходит как при работе с книгами, так и с компакт-дисками. В этом случае мы можем отсортировать товары по фамилии автора (т.е. по полю, содержащему строковые значения `"Булгаков"` и `"Вивальди"`), а не по имени, в котором содержатся менее определенные строковые значения `"Михаил"` и `"Антонио"`. Лень — это отличная стратегия проектирования, поэтому на данной стадии разработки не стоит особенно беспокоиться об употреблении класса `ShopProduct` для описания более чем одного вида товара.

Но если добавить в рассматриваемом здесь примере несколько новых требований, то дело сразу же усложнится. Допустим, требуется отобразить данные, характерные только для книг или компакт-дисков. Скажем, для компакт-дисков желательно вывести общую продолжительность звучания, а для книг — количество страниц. Безусловно, могут быть и другие отличия, но даже эти наглядно показывают суть возникшей проблемы.

Как же расширить данный пример, чтобы учесть все эти изменения? На ум сразу приходят два варианта. Во-первых, можно расположить все данные в классе `ShopProduct`, а во-вторых, разделить класс `ShopProduct` на два отдельных класса. Рассмотрим сначала первый вариант. Ниже показано, как объединить все данные о книгах и компакт-дисках в одном классе.

```
// Листинг 03.30

class ShopProduct
{
   public $numPages;
   public $playLength;
   public $title;
   public $producerMainName;
   public $producerFirstName;
   public $price;

   public function __construct(
       string $title,
       string $firstName,
       string $mainName,
       float $price,
       int $numPages = 0,
       int $playLength = 0
   ) {
       $this->title = $title;
       $this->producerFirstName = $firstName;
       $this->producerMainName = $mainName;
       $this->price = $price;
       $this->numPages = $numPages;
       $this->playLength = $playLength;
  }

   public function getNumberOfPages()
   {
     return $this->numPages;
   }

   public function getPlayLength()
   {
     return $this->playLength;
  }

  public function getProducer()
  {
    return $this->producerFirstName . " "
        . $this->producerMainName;
  }
}
```

Чтобы продемонстрировать проявляющиеся противоречивые силы, в данном примере были использованы методы доступа к свойствам `$numPages` и `$playLength`. В итоге объект, экземпляр которого создается с помощью приведенного выше класса, будет всегда содержать избыточные методы. Кроме того, экземпляр объекта, описывающего компакт-диск, приходится создавать с помощью лишнего аргумента конструктора. Таким образом, для компакт-диска будут сохраняться данные и функциональные возможности класса, относящиеся к книгам (количество страниц), а для книг — данные о продолжительности звучания компакт-диска. С этим пока еще можно как-то мириться. Но что, если

добавить больше видов товаров, причем каждый из них с собственными методами обработки, а затем ввести дополнительные методы для каждого вида товара? В конечном счете класс окажется слишком сложным и трудным для применения.

Поэтому принудительное объединение полей, относящихся к разным товарам, в один класс приведет к созданию слишком громоздких объектов с лишними свойствами и методами.

Но этим рассматриваемая здесь проблема не ограничивается. Функциональные возможности данного класса также вызывают серьезные трудности. Представьте метод, выводящий краткие сведения о товаре. Допустим, что сведения о товаре требуются отделу продаж для указания в виде одной итоговой строкой в счете-фактуре. Они хотят, чтобы мы включили в нее время звучания компакт-диска и количество страниц книги. Таким образом, нам придется реализовать этот метод отдельно для каждого вида товара. Чтобы отслеживать формат объекта, можно воспользоваться специальным признаком, как демонстрируется в следующем примере:

```
// Листинг 03.31

public function getSummaryLine()
{
  $base  = "{$this->title} ( {$this->producerMainName}, ";
  $base .= "{$this->producerFirstName} )";
  if ($this->type == 'book') {
      $base .= ": {$this->numPages} стр.";
  } elseif ($this->type == 'cd') {
      $base .= ": Время звучания - {$this->playLength}";
  }
  return $base;
}
```

Значение свойства `$type` устанавливается в конструкторе при создании объекта типа `ShopProduct`. Для этого нужно проанализировать значение аргумента `$numPages`. Если он содержит число больше нуля, то это книга, иначе — компакт-диск. В итоге класс `ShopProduct` стал еще более сложным, чем нужно. По мере добавления дополнительных отличий в форматы или ввода новых форматов нам будет все труднее справляться с реализацией функциональных возможностей данного класса. Поэтому для решения рассматриваемой здесь проблемы, по-видимому, придется выбрать другой подход.

В связи с тем что класс `ShopProduct` начинает напоминать “два класса в одном”, нам придется это признать и создать два типа вместо одного. Ниже показано, как это можно сделать.

```
// Листинг 03.32

class CdProduct
{
  public $playLength;
```

```
  public $title;
  public $producerMainName;
  public $producerFirstName;
  public $price;

  public function __construct(
      string $title,
      string $firstName,
      string $mainName,
      float $price,
      int $playLength
  ) {
      $this->title = $title;
      $this->producerFirstName = $firstName;
      $this->producerMainName = $mainName;
      $this->price = $price;
      $this->playLength = $playLength;
  }

  public function getPlayLength()
  {
    return $this->playLength;
  }

  public function getSummaryLine()
  {
    $base = "{$this->title} ( {$this->producerMainName}, ";
    $base .= "{$this->producerFirstName} )";
    $base .= ": Время звучания - {$this->playLength}";
    return $base;
  }

  public function getProducer()
  {
    return $this->producerFirstName . " "
         . $this->producerMainName;
  }
}

// Листинг 03.33

class BookProduct
{
  public $numPages;
  public $title;
  public $producerMainName;
  public $producerFirstName;
  public $price;

  public function __construct(
      string $title,
      string $firstName,
      string $mainName,
      float $price,
      int $numPages
  ) {
```

```
        $this->title = $title;
        $this->producerFirstName = $firstName;
        $this->producerMainName = $mainName;
        $this->price = $price;
        $this->numPages = $numPages;
    }

    public function getNumberOfPages()
    {
      return $this->numPages;
    }

    public function getSummaryLine()
    {
      $base = "{$this->title} ( {$this->producerMainName}, ";
      $base .= "{$this->producerFirstName} )";
      $base .= ": {$this->numPages} стр.";
      return $base;
    }

    public function getProducer()
    {
      return $this->producerFirstName . " "
           . $this->producerMainName;
    }
}
```

Мы постарались справиться с упомянутой выше сложностью, хотя для этого пришлось кое-чем пожертвовать. Теперь метод `getSummaryLine()` можно создать для каждого вида товара, даже не проверяя значение специального признака. И ни один из приведенных выше классов больше не содержит поля или методы, не имеющие к нему никакого отношения.

А жертва состояла в дублировании кода. Методы `getProducer()` абсолютно одинаковы в обоих классах. Конструктор каждого из этих классов одинаково устанавливает ряд сходных свойств. И это еще один признак дурного тона, которого следует всячески избегать в программировании.

Если требуется, чтобы методы `getProducer()` действовали одинаково в каждом классе, любые изменения, внесенные в одну реализацию, должны быть внесены и в другую. Но так мы очень скоро нарушим согласованность обоих классов.

Даже если мы уверены, что можем справиться с дублированием, на этом наши хлопоты не закончатся. Ведь у нас теперь имеются два типа, а не один.

Вспомните класс `ShopProductWriter`. Его метод `write()` предназначен для работы с одним типом `ShopProduct`. Как же исправить это положение, чтобы все работало, как прежде? Мы можем удалить объявление типа класса из сигнатуры метода, но тогда нам остается лишь надеяться, что методу `write()` будет передан объект правильного типа. Для проверки типа мы можем ввести в тело метода собственный код, как показано ниже.

```
// Листинг 03.34

class ShopProductWriter
{
  public function write($shopProduct)
  {
    if (
        ! ($shopProduct instanceof CdProduct  ) &&
        ! ($shopProduct instanceof BookProduct)
    ) {
        die("Передан неверный тип данных ");
    }
    $str = "{$shopProduct->title}: "
       .     $shopProduct->getProducer()
       . " ({$shopProduct->price})<br />\n";
    print $str;
  }
}
```

Обратите внимание, что в данном примере используется выражение с участием оператора `instanceof`. Вместо него подставляется логическое значение `true`, если объект, расположенный слева от оператора `instanceof` относится к типу, указанному справа. И снова мы были вынуждены ввести новый уровень сложности. Ведь нам нужно не только проверять аргумент `$shopProduct` на соответствие двум типам в методе `write()`, но и надеяться, что в каждом типе будут поддерживаться те же самые поля и методы, что и в другом. Согласитесь, что иметь дело только с одним типом было бы намного лучше. Ведь тогда мы могли бы воспользоваться объявлением типа класса в аргументе метода `write()` и были бы уверены, что в классе `ShopProduct` поддерживается конкретный интерфейс.

Свойства класса `ShopProduct`, связанные с книгами и компакт-дисками, не используются одновременно, но они, по-видимому, не могут существовать и по отдельности. Нам нужно оперировать книгами и компакт-дисками как одним типом данных, но в то же время обеспечить отдельную реализацию метода для каждого формата вывода. Следовательно, нам нужно предоставить общие функциональные возможность в одном месте, чтобы избежать дублирования, но в то же время сделать так, чтобы при вызове метода, выводящего краткие сведения о товаре, учитывались особенности этого товара. Одним словом, нам придется воспользоваться наследованием.

Использование наследования

Первый шаг в построении дерева наследования состоит в том, чтобы найти элементы базового класса, которые не соответствуют друг другу или требуют разного обращения с ними. Во-первых, нам известно, что методы `getPlayLength()` и `getNumberOfPages()` не могут находиться в одном классе. И во-вторых, нам известно, что потребуются разные реализации метода `getSummaryLine()`. Мы воспользуемся этими отличиями в качестве основания для создания двух производных классов.

```
// Листинг 03.35

class ShopProduct
{
  public $numPages;
  public $playLength;
  public $title;
  public $producerMainName;
  public $producerFirstName;
  public $price;

  public function __construct(
      string $title,
      string $firstName,
      string $mainName,
      float $price,
      int $numPages = 0,
      int $playLength = 0
  ) {
      $this->title = $title;
      $this->producerFirstName = $firstName;
      $this->producerMainName = $mainName;
      $this->price = $price;
      $this->numPages = $numPages;
      $this->playLength = $playLength;
  }

  public function getProducer()
  {
    return $this->producerFirstName . " "
         . $this->producerMainName;
  }

  public function getSummaryLine()
  {
    $base  = "{$this->title} ( {$this->producerMainName}, ";
    $base .= "{$this->producerFirstName} )";
    return $base;
  }
}

// Листинг 03.36

class CdProduct extends ShopProduct
{
  public function getPlayLength()
  {
    return $this->playLength;
  }

  public function getSummaryLine()
  {
    $base  = "{$this->title} ( {$this->producerMainName}, ";
    $base .= "{$this->producerFirstName} )";
    $base .= ": Время звучания - {$this->playLength}";
    return $base;
  }
}
```

```
// Листинг 03.37

class BookProduct extends ShopProduct
{
  public function getNumberOfPages()
  {
    return $this->numPages;
  }

  public function getSummaryLine()
  {
    $base  = "{$this->title} ( {$this->producerMainName}, ";
    $base .= "{$this->producerFirstName} )";
    $base .= ": {$this->numPages} стр.";
    return $base;
  }
}
```

Чтобы создать производный (или дочерний) класс, достаточно указать в его объявлении ключевое слово `extends`. В данном примере мы создали два новых класса — `BookProduct` и `CDProduct`. Оба они расширяют класс `ShopProduct`.

Поскольку в производных классах конструкторы не определяются, при получении экземпляров этих классов будет автоматически вызываться конструктор родительского класса. Дочерние классы наследуют доступ ко всем методам типа `public` и `protected` из родительского класса, но не к методам и свойствам типа `private`. Это означает, что метод `getProducer()` можно вызвать для созданного экземпляра класса `CDProduct`, как показано ниже, несмотря на то, что данный метод определен в классе `ShopProduct`.

```
// Листинг 03.38

$product2 = new CdProduct(
                              "Классическая музыка. Лучшее",
                              "Антонио",
                              "Вивальди",
                              10.99,
                              0,
                              60.33
                            );

print "Исполнитель: {$product2->getProducer()}\n";
```

Таким образом, оба дочерних класса наследуют поведение общего родительского класса. И мы можем обращаться с объектом типа `CdProduct` так, как будто это объект типа `ShopProduct`. Мы можем также передать объект типа `BookProduct` или `CDProduct` методу `write()` из класса `ShopProductWriter`, и все будет работать как следует.

Обратите внимание на то, что для обеспечения собственной реализации в классах `CDProduct` и `BookProduct` переопределяется метод `getSummaryLine()`. Производные классы могут расширять и изменять функциональные возможности

родительских классов. И в то же время каждый класс наследует свойства родительского класса.

Реализация метода `getSummaryLine()` в суперклассе может показаться избыточной, поскольку этот метод переопределяется в обоих дочерних классах. Тем не менее мы предоставляем базовый набор функциональных возможностей, который можно будет использовать в любом новом дочернем классе. Наличие этого метода в суперклассе гарантирует также для клиентского кода, что во всех объектах типа `ShopProduct` будет присутствовать метод `getSummaryLine()`. Далее будет показано, как выполнить это обязательство в базовом классе, вообще не предоставляя никакой реализации. Каждый объект дочернего класса `ShopProduct` наследует все свойства своего родительского класса. В собственных реализациях метода `getSummaryLine()` из классов `CDProduct` и `BookProduct` обеспечивается доступ к свойству `$title`.

Усвоить понятие наследования сразу не так-то просто. Объявляя один класс, расширяющий другой, мы гарантируем, что экземпляр его объекта определяется характеристиками сначала дочернего, а затем родительского класса. Чтобы лучше понять наследование, его удобнее рассматривать с точки зрения поиска. Так, если сделать вызов `$product2->getProducer()`, интерпретатор PHP не сможет найти указанный метод в классе `CDProduct`. Поиск завершится неудачно, и поэтому будет использована стандартная реализация данного метода в классе `ShopProduct`. С другой стороны, когда делается вызов `$product2->getSummaryLine()`, интерпретатор PHP находит реализацию метода `getSummaryLine()` в классе `CDProduct` и вызывает его.

Это же относится и к доступу к свойствам. При обращении к свойству `$title` в методе `getSummaryLine()` из класса `BookProduct` интерпретатор PHP не находит определение этого свойства в классе `BookProduct` и поэтому использует определение данного свойства, заданное в родительском классе `ShopProduct`. А поскольку свойство `$title` используется в обоих подклассах, оно должно определяться в суперклассе.

Даже беглого взгляда на конструктор класса `ShopProduct` достаточно, чтобы понять, что в базовом классе по-прежнему выполняется обработка тех данных, которыми должен оперировать дочерний класс. Так, конструктору класса `BookProduct` должен передаваться аргумент `$numPages`, значение которого устанавливается в одноименном свойстве, а конструктор класса `CDProduct` должен обрабатывать аргумент `$playLength` и одноименное свойство. Чтобы добиться этого, определим методы-конструкторы в каждом дочернем классе.

Конструкторы и наследование

Определяя конструктор в дочернем классе, вы берете на себя ответственность за передачу требующихся аргументов родительскому классу. Если же вы этого не сделаете, то получите частично сконструированный объект.

Чтобы вызвать нужный метод из родительского класса, придется обратиться непосредственно к этому классу через дескриптор. Для этой цели в PHP предусмотрено ключевое слово `parent`.

Чтобы обратиться к методу в контексте класса, а не объекта, следует использовать символы `"::"`, а не `"->"`. Следовательно, синтаксическая конструкция `parent::__construct()` означает следующее: “Вызвать метод `__construct()` из родительского класса”. Изменим рассматриваемый здесь пример таким образом, чтобы каждый класс оперировал только теми данными, которые имеют к нему непосредственное отношение.

```
// Листинг 03.39

class ShopProduct
{
  public $title;
  public $producerMainName;
  public $producerFirstName;
  public $price;

function __construct(
      $title,
      $firstName,
      $mainName,
      $price
  ) {
      $this->title = $title;
      $this->producerFirstName = $firstName;
      $this->producerMainName = $mainName;
      $this->price = $price;
  }

  function getProducer()
  {
    return $this->producerFirstName . " "
         . $this->producerMainName;
  }

  function getSummaryLine()
  {
    $base = "{$this->title} ( {$this->producerMainName}, ";
    $base .= "{$this->producerFirstName} )";
    return $base;
  }
}

// Листинг 03.40

class BookProduct extends ShopProduct
{
  public $numPages;
```

```
  public function __construct(
    string $title,
    string $firstName,
    string $mainName,
    float $price,
    int $numPages
) {
    parent::__construct(
      $title,
      $firstName,
      $mainName,
      $price
    );
    $this->numPages = $numPages;
  }

  public function getNumberOfPages()
  {
    return $this->numPages;
  }

  public function getSummaryLine()
  {
    $base = "{$this->title} ( $this->producerMainName, ";
    $base .= "$this->producerFirstName )";
    $base .= ": {$this->numPages} стр.";
    return $base;
  }
}

// Листинг 03.41

class CdProduct extends ShopProduct
{
  public $playLength;

  public function __construct(
    string $title,
    string $firstName,
    string $mainName,
    float $price,
    int $playLength
) {
    parent::__construct(
      $title,
      $firstName,
      $mainName,
      $price
    );
    $this->playLength = $playLength;
  }

  public function getPlayLength()
  {
    return $this->playLength;
  }
```

```
  public function getSummaryLine()
  {
    $base = "{$this->title} ( {$this->producerMainName}, ";
    $base .= "{$this->producerFirstName} )";
    $base .= ": Время звучания - {$this->playLength}";
    return $base;
  }
}
```

В каждом дочернем классе перед определением его собственных свойств вызывается конструктор родительского класса. Теперь в базовом классе известны только его собственные данные. Дочерние классы обычно являются специализированными вариантами родительских классов. Как правило, следует не допускать, чтобы в родительских классах было известно что-нибудь особенное о дочерних классах.

НА ЗАМЕТКУ. До появления PHP 5 имя конструктора совпадало с именем того класса, к которому он относился. В новом варианте обозначения конструкторов для единообразия используется имя **`__construct()`**. При использовании старого синтаксиса вызов конструктора из родительского класса был привязан к имени конкретного класса, например: **`parent::ShopProduct();`**. А в версии PHP 7 старый синтаксис вызова конструкторов признан устаревшим и больше не должен использоваться.

Вызов переопределяемого метода

Ключевое слово `parent` можно использовать в любом методе, переопределяющем свой эквивалент из родительского класса. Когда переопределяется метод, то, вероятнее всего, требуется расширить, а не отменить функциональные возможности родительского класса. Достичь этого можно, вызвав метод из родительского класса в контексте текущего объекта. Если снова проанализировать реализации метода `getSummaryLine()`, то можно заметить, что значительная часть кода в них дублируется. И лучше воспользоваться этим обстоятельством, как показано ниже, чем повторять функциональные возможности, уже имеющиеся в классе `ShopProduct`.

```
// Листинг 03.42

// Класс ShopProduct...

function getSummaryLine()
{
  $base = "{$this->title} ( {$this->producerMainName}, ";
  $base .= "{$this->producerFirstName} )";
  return $base;
}
```

```
// Листинг 03.43

// Класс BookProduct...

public function getSummaryLine()
{
  $base = parent::getSummaryLine();
  $base .= ": {$this->numPages} стр.";
  return $base;
}
```

Итак, мы определили основные функции для метода `getSummaryLine()` в базовом классе `ShopProduct`. Вместо того чтобы повторять их в подклассах `CDProduct` и `BookProduct`, мы просто вызовем родительский метод, прежде чем вводить дополнительные данные в итоговую строку.

А теперь, когда были изложены основы наследования, можно, наконец, рассмотреть область видимости свойств и методов, чтобы получить полную картину происходящего.

Управление доступом к классам: модификаторы `public`, `private` и `protected`

До сих пор мы явно или неявно объявляли все свойства как открытые (`public`). Такой тип доступа задан по умолчанию для всех методов и свойств, объявляемых с помощью устаревшего ключевого слова `var`.

Элементы класса можно объявить как `public` (открытые), `private` (закрытые) или `protected` (защищенные). Это означает следующее.

- К открытым свойствам и методам можно получать доступ из любого контекста.
- К закрытому свойству и методу можно получить доступ только из того класса, в котором они объявлены. Даже подклассы данного класса не имеют доступа к таким свойствам и методам.
- К защищенным свойствам и методам можно получить доступ либо из содержащего их класса, либо из его подкласса. Никакому внешнему коду такой доступ не предоставляется.

Чем же это может быть нам полезным? Ключевые слова, определяющие область видимости, позволяют раскрыть только те элементы класса, которые требуются клиенту. Это дает возможность задать ясный и понятный интерфейс для объекта.

Управление доступом, позволяющее запретить клиенту обращаться к некоторым свойствам и методам класса, помогает также избежать ошибок в прикладном коде. Допустим, требуется организовать поддержку скидок в объектах типа `ShopProduct`. С этой целью можно ввести свойство `$discount` и метод `setDiscount()` в данный класс, как показано ниже.

```
// Листинг 03.44

// Класс ShopProduct...

  public $discount = 0;

//...

  public function setDiscount(int $num)
  {
    $this->discount = $num;
  }
```

Имея на вооружении механизм определения скидки, мы можем создать метод `getPrice()`, где во внимание принимается установленная скидка.

```
public function getPrice()
{
  return ($this->price - $this->discount);
}
```

Но тут возникает затруднение. Нам требуется раскрыть только скорректированную цену, но клиент может легко обойти метод `getPrice()` и получить доступ к свойству `$price` следующим образом:

```
print "Цена товара -  {$product1->price}\n";
```

В итоге будет выведена исходная цена, а не цена со скидкой, которую нам требуется представить. Чтобы избежать этого, достаточно сделать свойство `$price` закрытым (`private`), тем самым запретив клиентам прямой доступ к нему и вынуждая их вызывать метод `getPrice()`. Любая попытка получить доступ к свойству `$price` вне класса `ShopProduct` завершится неудачно. И это свойство перестанет существовать для внешнего мира.

Но объявление свойства как `private` может оказаться излишне строгой мерой, ведь тогда дочерний класс не сможет получить доступ к закрытым свойствам своего родительского класса. А теперь представьте следующие бизнес-правила: скидка не распространяется только на книги. В таком случае можно переопределить метод `getPrice()`, чтобы он возвращал свойство `$price` без учета скидки:

```
// Листинг 03.45

// Класс BookProduct...

  public function getPrice()
  {
    return $this->price;
  }
```

Свойство `$price` объявлено в классе `ShopProduct`, а не в `BookProduct`, и поэтому попытка в приведенном выше фрагменте кода получить доступ к этому свойству завершится неудачно. Чтобы разрешить это затруднение, следует объявить свойство `$price` защищенным (`protected`) и тем самым предоставить доступ к нему дочерним классам. Напомним, что к защищенным свойствам или методам нельзя получить доступ за пределами иерархии того класса, в котором они были объявлены. Доступ к ним можно получить только из исходного класса или его дочерних классов.

Как правило, предпочтение следует отдавать конфиденциальности. Сначала сделайте свойства закрытыми или защищенными, а затем ослабляйте ограничения, накладываемые на доступ к ним, по мере надобности. Многие (если не все) методы в ваших классах будут открытыми, но опять же, если у вас есть сомнения, ограничьте доступ. Метод, предоставляющий локальные функциональные возможности другим методам в классе, вообще не нужен пользователям класса. Поэтому сделайте его закрытым или защищенным.

Методы доступа

Даже если в клиентской программе потребуется обрабатывать значения, хранящиеся в экземпляре вашего класса, как правило, стоит запретить прямой доступ к свойствам данного объекта. Вместо этого создайте методы, которые возвращают или устанавливают нужные значения. Такие методы называют *методами доступа* или методами *получения* и *установки*.

Ранее на примере метода `getPrice()` уже демонстрировалось одно преимущество, которое дают методы доступа: с их помощью можно фильтровать значения свойств в зависимости от обстоятельств. Метод установки можно также использовать для соблюдения типа свойства. Если объявления типов классов накладывают определенные ограничения на аргументы методов, то свойства могут содержать данные любого типа. Вспомните определение класса `ShopProductWriter`, где для вывода списка данных используется объект типа `ShopProduct`? Попробуем пойти дальше и сделать так, чтобы в классе `ShopProductWriter` можно было одновременно выводить данные из любого количества объектов типа `ShopProduct`.

```
// Листинг 03.46

class ShopProductWriter
{
  public $products = [];

  public function addProduct(ShopProduct $shopProduct)
  {
    $this->products[] = $shopProduct;
  }
```

```
  public function write()
  {
    $str = "";
    foreach ($this->products as $shopProduct) {
      $str .= "{$shopProduct->title}: ";
      $str .= $shopProduct->getProducer();
      $str .= " ({$shopProduct->getPrice()})<br />\n";
    }
    print $str;
  }
}
```

Теперь класс `ShopProductWriter` стал намного более полезным. Он может содержать много объектов типа `ShopProduct` и сразу выводить информацию обо всех этих объектах. Но мы все еще должны полагаться на то, что программисты клиентского кода будут строго придерживаться правил работы с классом. И хотя мы предоставили метод `addProduct()`, мы не запретили программистам непосредственно манипулировать свойством `$products`. В итоге можно не только добавить объект неверного типа в массив свойств `$products`, но и затереть весь массив и заменить его значением простого типа. Чтобы не допустить этого, нужно сделать свойство `$products` закрытым.

```
// Листинг 03.47

class ShopProductWriter {
  private $products = [];
//...
```

Теперь внешний код не сможет повредить массив свойств `$products`. Весь доступ к нему должен осуществляться через метод `addProduct()`, а объявление типа класса, которое используется в объявлении этого метода, гарантирует, что в массив свойств могут быть добавлены только объекты типа `ShopProduct`.

Семейство классов `ShopProduct`

И в заключение этой главы внесем коррективы в класс `ShopProduct` и его дочерние классы таким образом, чтобы ограничить доступ к их элементам.

```
// Листинг 03.48

class ShopProduct
{
  private $title;
  private $producerMainName;
  private $producerFirstName;
  protected $price;
  private $discount = 0;

  public function __construct(
      string $title,
      string $firstName,
      string $mainName,
      float $price
  ) {
```

```
        $this->title = $title;
        $this->producerFirstName = $firstName;
        $this->producerMainName = $mainName;
        $this->price = $price;
    }

    public function getProducerFirstName()
    {
      return $this->producerFirstName;
    }

    public function getProducerMainName()
    {
      return $this->producerMainName;
    }

    public function setDiscount($num)
    {
      $this->discount = $num;
    }

    public function getDiscount()
    {
      return $this->discount;
    }

    public function getTitle()
    {
      return $this->title;
    }

    public function getPrice()
    {
      return ($this->price - $this->discount);
    }

    public function getProducer()
    {
      return $this->producerFirstName . " "
          . $this->producerMainName;
    }

    public function getSummaryLine()
    {
      $base = "{$this->title} ( {$this->producerMainName}, ";
      $base .= "{$this->producerFirstName} )";
      return $base;
    }
}

// Листинг 03.49

class CdProduct extends ShopProduct
{
    private $playLength;
```

```
  public function __construct(
    string $title,
    string $firstName,
    string $mainName,
    float $price,
    int $playLength
  ) {
      parent::__construct(
        $title,
        $firstName,
        $mainName,
        $price
      );
      $this->playLength = $playLength;
  }

  public function getPlayLength()
  {
    return $this->playLength;
  }

  public function getSummaryLine()
  {
    $base = "{$this->title} ( {$this->producerMainName}, ";
    $base .= "{$this->producerFirstName} )";
    $base .= ": Время звучания - {$this->playLength}";
    return $base;
  }
}

// Листинг 03.50

class BookProduct extends ShopProduct
{
  private $numPages;

  public function __construct(
    string $title,
    string $firstName,
    string $mainName,
    float $price,
    int $numPages
  ) {
      parent::__construct(
        $title,
        $firstName,
        $mainName,
        $price
      );
      $this->numPages = $numPages;
  }

  public function getNumberOfPages()
  {
    return $this->numPages;
  }
```

```
  public function getSummaryLine()
  {
    $base = parent::getSummaryLine();
    $base .= ": {$this->numPages} стр.";
    return $base;
  }

  public function getPrice()
  {
    return $this->price;
  }
}
```

В этой версии семейства классов `ShopProduct` нет ничего особенно нового. Все свойства теперь стали закрытыми или защищенными. И для завершенности этого семейства классов мы добавили в него ряд методов доступа.

Резюме

В этой главе мы подробно рассмотрели основы ООП на PHP, превратив первоначально пустой класс в полностью функциональную иерархию наследования. Мы разобрались в некоторых вопросах проектирования, особенно касающихся типов данных и наследования. Из этой главы вы также узнали о поддержке видимости элементов кода в PHP и ознакомились с некоторыми примерами ее применения. В следующей главе будут представлены другие объектно-ориентированные возможности PHP.