4

Работа с объектами и не только

Объектно-ориентированное программирование (ООП) — одна из наиболее широко применяемых на сегодняшний день парадигм. Изначально язык PHP не задумывался как объектно-ориентированный, но некоторый ограниченный вариант поддержки ООП был добавлен в версию PHP 4 и гораздо больший — в PHP 5.x.

Поскольку для этой книги был выбран формат справочника, мы признаем, что тема ООП — не самая подходящая (по сравнению с другими) для краткого описания. Очень трудно в сжатой форме изложить темы, связанные с ООП, и тогда приходится либо ограничиваться рассмотрением конкретных сценариев, либо, наоборот, использовать общие понятия в расчете на то, что читатель имеет некоторые базовые знания в этой области.

В результате некоторых дискуссий мы решили рассмотреть определенную совокупность разных тем (напрямую или косвенно связанных с ООП), которые демонстрируют полезные, но нетривиальные ООП-возможности языка PHP, разъяснив при этом фундаментальные конструкции ООП. Мы надеемся, что материал, представленный в этой главе, окажется действительно полезным для вас, и вы будете обращаться к нему еще не раз, поскольку при использовании ООП очень важен практический опыт, которым мы и хотели бы с вами поделиться.

Создание классов

```
class MyClass {
}
```

В этом разделе мы предлагаем краткое (и поэтому неполное) изложение основных (и поэтому самых важных) понятий, составляющих суть объектно-ориентированного программирования с точки зрения языка PHP. Центральным элементом идеи ООП является *класс*. Класс может содержать константы, свойства и методы. Для определения области видимости идентификаторов используются ключевые слова `public`, `protected` и `private`,

которые показывают, кто может получить доступ к тому или иному свойству или методу.

Если класс уже создан (с помощью оператора new), ключевое слово $this предоставляет ссылку на вызывающий объект. Если класс содержит метод с именем __construct(), то он будет вызван при создании экземпляра этого класса:

```
<?php
  class MyClass {
    private $prop = null;

    public function setProperty($value) {
      $this->prop = $value;
    }

    public function getProperty() {
      return $this->prop;
    }

    public function __construct($value = null) {
      if ($value !== null) {
        $this->prop = $value;
      }
    }
  }
  $c = new MyClass('abc');
  echo $c->getProperty();
?>
```

Реализация класса (class.php)

ПРИМЕЧАНИЕ. Для определения доступа к свойствам или методам класса используются следующие три уровня видимости:

- public — доступ открыт отовсюду;
- protected — доступ открыт в классе и его экземплярах, а также в производных (унаследованных) классах и их экземплярах;
- private — доступ открыт только в классе и его экземплярах.

Доступ к свойствам или методам класса можно также получить, не имея экземпляра класса, но используя *статический* контекст (и ключевое слово static). Очевидно, что статические методы не имеют доступа к псевдопеременной $this, но с помощью ключевого слова self можно получить доступ к текущему классу:

```
<?php
  class MyStaticClass {
    private static $prop = 'abc';
    public static function getProperty() {
```

```
      return self::$prop;
    }
  }
  echo MyStaticClass::getProperty();
?>
```

Реализация класса с помощью статического метода и статического свойства (static.php)

Понятие наследования

```
class MyDerivedClass extends MyBaseClass {
}
```

Некоторый класс может быть создан из другого класса с использованием ключевого слова extends. Как следствие, новый класс в этом случае будет содержать (т.е. унаследует от базового класса) все его открытые и защищенные, т.е. public- и protected-методы. Разумеется, эти методы в производных классах можно переопределить.

Доступ к родительскому классу можно получить с помощью ключевого слова parent. Связанный с этим синтаксис напоминает использование статического контекста, но таковым не является — поэтому в вызванном методе вы можете использовать псевдопеременную $this.

В следующем коде (помимо вызова конструктора родительского класса) продемонстрирована возможность переопределения конструктора в производном классе. Обратите внимание на различие сигнатур конструкторов родительского и производного классов. Здесь также показан пример вызова метода, определенного в базовом классе:

```
<?php
  class MyBaseClass {
    protected $value1 = null;
    protected $value2 = null;

    protected function __construct($value = null) {
      if ($value !== null) {
        $this->value1 = $value;
      }
    }

    public function getValue1() {
      return $this->value1;
    }
  }
```

```
  class MyDerivedClass extends MyBaseClass {
    protected $value2 = null;

    public function __construct($value1 = null,
➥$value2 = null) {
      if ($value1 !== null) {
        parent::__construct($value1);
        if ($value2 !== null) {
          $this->value2 = $value2;
        }
      }
    }

    public function getValue2() {
      return $this->value2;
    }
  }

  $c = new MyDerivedClass('abc', 'def');
  echo '1: ', $c->getValue1(), ', 2:', $c-
➥>getValue2();
?>
```

Использование механизма наследования в языке PHP (`extends.php`*)*

ПРИМЕЧАНИЕ. В PHP не реализована поддержка множественного наследования, поэтому некоторый класс может быть создан только на основе *единственного* другого класса. Безусловно, при необходимости вы можете добиться желаемой "множественности": достаточно создать класс B из класса A, а класс C — из класса B, и тогда класс C также унаследует `public`- и `protected`-методы класса A.

Использование абстрактных классов и интерфейсов

```
abstract class MyAbstractBaseClass {
}
```

PHP поддерживает два дополнительных способа наследования: посредством абстрактных классов и интерфейсов. Эти способы имеют как сходства, так и различия, причем существенные. У них общая цель — определить соглашения или шаблон для производного класса: базовый класс (или классы, как показано ниже) определяет, какие методы войдут в новый класс.

При использовании абстрактного класса наследование реализуется все с тем же ключевым словом `extends`. Но при этом базовый класс определяется как абстрактный, т.е. с помощью ключевого слова `abstract`. Некоторые методы абстрактного класса могут

быть также определены как абстрактные, но при этом они не содержат никакого кода:

```
protected abstract function myMethod();
```

Теперь производный класс должен обязательно реализовать такие методы, используя в точности такие же сигнатуры и такой же (или хотя бы менее ограничительный) уровень видимости. Например, если абстрактный базовый класс определяет абстрактный защищенный метод, производный класс должен реализовать этот метод как защищенный (`protected`) или открытый (`public`). Если вы используете указание типов (type hinting), оно тоже должно быть идентичным заданному в абстрактном классе.

Если абстрактный метод содержит дополнительные методы, которые не объявлены как абстрактные, они наследуются так же, как в случае обычных классов, если они являются защищенными или открытыми:

```
<?php
  abstract class MyAbstractBaseClass {
    protected $value1 = null;
    protected $value2 = null;
    protected function getValue1() {
      return $this->value1;
    }

    protected function getValue2() {
      return $this->value2;
    }

    protected abstract function dumpData();
  }

  class MyAbstractClass extends MyAbstractBaseClass
➥{
    public function __construct($value1 = null,
➥$value2 = null) {
      if ($value1 !== null) {
        $this->value1 = $value1;
        if ($value2 !== null) {
          $this->value2 = $value2;
        }
      }
    }

    public function dumpData() {
      echo '1: ', $this->getValue1(), ', 2:',
➥$this->getValue2();
    }
  }
```

```
  $c = new MyAbstractClass('ghi', 'jkl');
  $c->dumpData();
?>
```

Использование абстрактного класса (abstract.php)

Подобно обычным классам, абстрактные классы поддерживают только единичное наследование.

В "интерфейсном" варианте используется подход, который можно определить как "такой же, да не совсем". Интерфейс внешне напоминает обычный PHP-класс, но вместо ключевого слова `class` в нем используется ключевое слово `interface`. Интерфейс совсем не содержит никакой реализации, а только сигнатуры функций. И все эти функции должны быть открытыми (т.е. определены как `public`-функции).

Реальный класс в этом случае "принимает наследство" не из класса, а из интерфейса; в этом контексте мы говорим, что класс реализует интерфейс. Этот нюанс подчеркивается ключевым словом `implements`, которое используется вместо слова `extends`.

Остальные правила наследования очень похожи на правила, действующие в отношении абстрактных классов: в новом классе должны быть реализованы все методы из интерфейса, и сигнатуры функций не должны быть изменены (включая уровень видимости `public`). Но здесь есть дополнительный бонус: класс может реализовать несколько интерфейсов одновременно, если все участвующие интерфейсы имеют несовпадающие имена методов.

При выполнении следующего кода создаются два интерфейса, а затем оба они реализуются в одном классе:

```
<?php
  interface MyInterface1 {
    public function getValue1();
    public function getValue2();
  }

  interface MyInterface2 {
    public function dumpData();
  }

  class MyInterfaceClass implements MyInterface1,
➥MyInterface2 {
    protected $value1 = null;
    protected $value2 = null;
    public function __construct($value1 = null,
➥$value2 = null) {
      if ($value1 !== null) {
```

```
        $this->value1 = $value1;
        if ($value2 !== null) {
          $this->value2 = $value2;
        }
      }
    }

    public function getValue1() {
      return $this->value1;
    }

    public function getValue2() {
      return $this->value2;
    }

    public function dumpData() {
      echo '1: ', $this->getValue1(), ', 2:',
?$this->getValue2();
    }
  }

  $c = new MyInterfaceClass('mno', 'pqr');
  $c->dumpData();
?>
```

Использование интерфейса (`interface.php`)

Предотвращение наследования и переопределения

```
final class MyFinalBaseClass {}
public final function myMethod() {}
```

Возможны ситуации, когда вы не хотите, чтобы некоторые классы "давали потомство" (например, в случае, если они содержат критические функции, а вы не желаете, чтобы другие разработчики перезаписали эти жизненно важные методы). Для решения такой задачи в PHP предусмотрено ключевое слово `final`. Вы можете объявить либо `final`-класс (и тогда его невозможно наследовать совсем), либо `final`-метод (и тогда код не сможет его перезаписать). При выполнении следующего кода возникает исключительная ситуация (рис. 4.1):

```
<?php
  class MyFinalBaseClass {
    public final function getCopyrightNotice() {
      return '&copy; by Original Author';
    }
```

```
  }

  class MyFinalClass extends MyFinalBaseClass {
    function getCopyrightNotice() {
      return '&copy; by me!';
    }
  }
?>
```

Предотвращение перезаписи методов с помощью ключевого слова `final` (`final.php`)

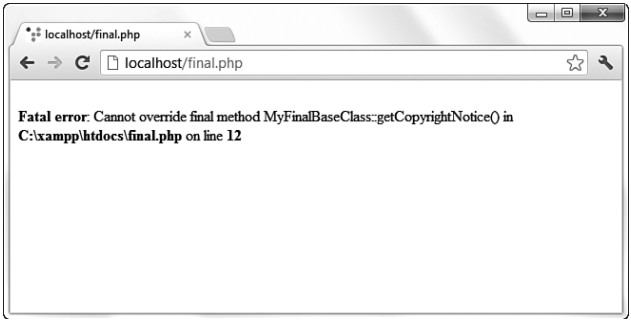

Рис. 4.1. Методы, определенные с помощью ключевого слова `final`, перезаписать невозможно

Если вы — единственный разработчик, то, может показаться, что вам необязательно использовать ключевое слово `final` для класса или метода. Но всегда стоит быть готовым к тому, что в будущем ваша команда может разрастись, или вы решите реорганизовать свой код, превратив его в библиотеку, которой будут пользоваться другие программисты.

Использование автозагрузки

```
spl_autoload_register('myAutoloadFunction');
```

Когда вы используете класс, он должен быть определен; в противном случае PHP сгенерирует сообщение об ошибке. Но можно организовать "второй шанс", если при первой попытке нужный класс оказался недоступным. Функция `spl_autoload_register()` позволяет зарегистрировать функцию, которая будет вызываться в случае, если PHP-код попытается использовать класс, который еще не был зарегистрирован. После обращения к функции регистрации код, который использует неопределенный класс, выполнится снова. И если класс на этот раз окажется доступным, зна-

чит, "жизнь наладилась"; в противном случае будет сгенерировано обычное сообщение об ошибке.

Как правило, используют фиксированную схему присваивания имен для файлов, содержащих классы, и помещают эти классы в определенный каталог. Например, класс `MyClass` помещают в файл `/classes/MyClass.class.php`. В этом случае следующая функция автозагрузки загрузит этот класс по первому требованию:

```
<?php
  function myAutoloadFunction($classname) {
    if (preg_match('/^[a-zA-Z_\x7f-\xff][a-zA-Z0-
➥9_\x7f-\xff]*$/', $classname)) {
      //проверка на допустимое имя класса
      require_once
➥"/path/to/classes/$classname.class.php";
    }
  }
  spl_autoload_register('myAutoloadFunction');
?>
```

Автозагрузка классов (`autoload.php`)

Так, например, следующий вызов

```
$c = new UnknownClass();
```

в свою очередь вызовет функцию `myAutoloadFunction()`, передав ей значение `UnknownClass` в качестве параметра `$classname`.

Если вызвать функцию `spl_autoload_register()` несколько раз, все функции автозагрузки будут зарегистрированы, и PHP будет вызывать каждую из них до тех пор, пока, наконец, желаемый класс не будет определен, — или до тех пор, пока все такие функции не будут вызваны.

ПРИМЕЧАНИЕ. Функция `spl_autoload_register()` была введена в версию PHP 5.1.2 как часть стандартной библиотеки SPL (Standard PHP Library). Раньше PHP-программистам приходилось писать функцию `__autoload()` и помещать в нее код автозагрузки. В скором будущем, вероятно, функция `__autoload()` будет удалена из PHP, поэтому вместо нее стоит использовать функцию `spl_autoload_register()`.

ПРИМЕЧАНИЕ. В PHP также предусмотрена возможность доступа к методам и свойствам класса, которые не существуют:

- если вызываемый метод класса не существует, выполняется метод `__call()`, если таковой существует;
- если происходит попытка доступа к свойству, которое не существует, выполняются методы класса `__get()` и `__set()` (для чтения и записи), если таковые существуют.

Клонирование объектов

```
$b = clone $a;
```

Предположим, у вас есть экземпляр класса, который хранится в переменной `$a`. В результате присваивания `$b = $a` не будет создана копия переменной `$a`, но теперь переменная `$b` будет содержать ссылку на `$a`, и у вас по-прежнему будет только один экземпляр класса.

В качестве альтернативы копированию в PHP предусмотрено ключевое слово `clone`, которое позволяет создавать копию класса:

```
$b = clone $a;
```

Но в некоторых случаях вам может понадобиться именно копия. Например, представьте себе, что каждый экземпляр класса имеет свойство, которое содержит глобально уникальный идентификатор (globally unique identifier — GUID). Несмотря на то что вы создаете клон (копию) экземпляра класса, вам нужно, чтобы его идентификатор (GUID) все-таки был уникальным (насколько это возможно).

PHP предоставляет способ внесения кода в объект после его клонирования. Если класс содержит метод `__clone()`, то он будет выполнен для клонированного экземпляра класса после завершения процесса клонирования:

```
<?php
  class MyCloneableClass {
    private $guid = null;
    public function __construct() {
      $this->guid = uniqid();
    }

    public function __clone() {
      $this->guid = uniqid();
    }

    public function getGuid() {
      return $this->guid;
    }
  }

  $c1 = new MyCloneableClass();
  $c2 = clone $c1;
  echo '1: ', $c1->getGuid(), ', 2:', $c2-
➥>getGuid();
?>
```

Клонирование объектов (с последующим их изменением) (`clone.php`)

После выполнения приведенного выше кода был получен результат, подобный представленному на рис. 4.2: два экземпляра класса имеют различные GUID-значения.

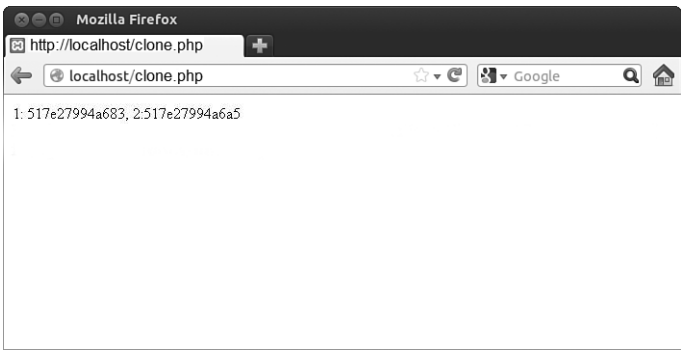

Рис. 4.2. Два экземпляра класса имеют различные GUID-значения

Сериализация и десериализация объектов

```
public function __sleep() {
  $this->db->close();
  return array('guid');
}
public function __wakeup() {
  $this->connectDb();
}
```

Когда вам нужно хранить объекты для использования в будущем, их приходится преобразовывать в формат, который позволяет легко помещать их, скажем, в базу данных. В PHP предусмотрено несколько способов преобразования объектных экземпляров в строки (и обратного преобразования в исходную форму), и самый распространенный состоит в использовании функций `serialize()` и `deserialize()`, о которых мы поговорим и в других главах этой книги.

Интересно то, что с точки зрения объектно-ориентированного программирования процесс сериализации и десериализации можно перехватывать. Предположим, например, что ваш класс содержит свойство, которое не должно (или не может) быть сериализировано (возможно, это дескриптор базы данных). В идеале это свойство следует удалить из класса перед сериализацией и восстановить после десериализации.

Оказывается, в PHP есть методы с "магическими" именами, которые позволят нам справиться с этой задачей. Если ваш класс содержит эти методы, они будут выполнены "как по волшебству":

- метод `__sleep()` — возвращает массив со всеми свойствами, подлежащими сериализации, возможна также очистка объекта;
- метод `__wakeup()` — вызывается после десериализации для воссоздания свойств.

На примере следующего кода показана типичная реализация процесса закрытия и восстановления соединения с базой данных. После сериализации и десериализации GUID-значение остается прежним, но соединение с базой данных восстанавливается:

```
<?php
  class MySerializableClass {
    public $guid = null;
    public $db = null;

    public function __construct() {
      $this->guid = uniqid();
      $this->connectDb();
    }

    private function connectDb() {
      $this->db = new MySQLi('server', 'user',
➥'password', 'db');
    }

    public function __sleep() {
      $this->db->close();
      return array('guid');
    }

    public function __wakeup() {
      $this->connectDb();
    }
  }

  $c1 = new MySerializableClass();
  echo 'Before: ', $c1->guid;
  $s = serialize($c1);
  $c2 = unserialize($s);
  echo '; after: ', $c2->guid;
?>
```

Сериализация и десериализация объектов (`serialize.php`)

ПРИМЕЧАНИЕ. При десериализации объекта необходимо удостовериться в существовании соответствующего класса; в противном случае PHP при формировании результата использует "особенно страшный" класс `__PHP_Incomplete_Class_Name`.

СОВЕТ. PHP поддерживает множество магических методов и функций, с которыми можно ознакомиться в онлайн-справочнике по адресу `http://php.net/oop5.magic`.

Реализация синглтон-объектов

```
if (self::$instance == null) {
  self::$instance = new self;
}
return self::$instance;
```

Вполне обычным, но тем не менее критикуемым (из-за недостатков) шаблоном разработки является шаблон синглтона (singleton pattern). Такой объект не так-то просто реализовать, поэтому мы посвятили его созданию отдельный раздел.

Основная цель использования шаблона синглтона состоит в следующем: вне зависимости от того, насколько часто вы реализуете определенный класс, вы можете получить *только один* объект этого класса. Это может оказаться полезным, например, при создании соединения с базой данных: зачастую необходимо только одно соединение, которое может быть повторно использовано в вашем коде.

В типичной PHP-реализации шаблона синглтона используются языковые средства ООП, которые позволяют предотвратить создание различных экземпляров одного класса. Прежде всего делаем недоступным конструктор класса благодаря использованию модификатора доступа `private`:

```
private function __construct() {
}
```

Однако путем клонирования можно создать два экземпляра класса и с закрытым конструктором, т.е. можно создать один (разрешенный) экземпляр, а затем клонировать его. Чтобы не допустить этого, сделаем также закрытым и магический метод `__clone()`:

```
private function __clone() {
}
```

Итак, теперь мы не можем создать множество экземпляров класса; но пока мы не можем создать даже один экземпляр.

Поэтому вводим метод, который позволит нам создать один экземпляр класса. Ведь в самом классе мы можем вызвать конструктор класса — уровень видимости private в этом случае нам не помеха. И тогда экземпляр класса будет храниться в статическом и закрытом члене класса. При вызове формальный метод создания экземпляра сначала проверит, не хранится ли уже экземпляр класса в этом статическом свойстве. Если да, метод вернет этот экземпляр; в противном случае код создаст новый (и единственный) экземпляр класса:

```
static private $instance = null;
static public function getInstance() {
  if (self::$instance == null) {
    self::$instance = new self;
  }
  return self::$instance;
}
```

Следующий код представляет собой полный синтаксис создания синглтон-класса:

```
<?php
  class PHPSingleton{
    static private $instance = null;

    static public function getInstance() {
      if (self::$instance == null) {
        self::$instance = new self;
      }
      return self::$instance;
    }

    private function __construct() {
    }
    private function __clone() {
    }
  }
?>
```

Реализация синглтона (singleton.php)

ПРИМЕЧАНИЕ. Как уже упоминалось, синглтон-шаблоны имеют некоторые недостатки. Код с их участием обычно труднее тестировать, поскольку класс синглтона должен быть глобальным, что навязывает это состояние всему приложению. Корректность работы с синглтоном зависит от порядка обращений к нему, что значительно усложняет работу с ним, но, как говорят, в каждом конкретном случае могут быть свои трудности.

Использование пространства имен

Одна важная новинка в версии PHP 5.3 — включение пространств имен. Пространства имен — это инструмент группирования связанных функций, который позволяет избежать конфликтов на уровне имен. Представьте себе, что вы написали функцию с таким распространенным именем, как `showInfo()`. Другой разработчик, участвующий в вашем совместном проекте, пишет другую функцию, но использует то же самое имя. Обычно, чтобы сделать имя функции уникальным, его снабжают префиксом, но это может привести к образованию очень длинных идентификаторов, например `Project_Module_Submodule_showInfo()`. Вот тут-то нас и выручают пространства имен: вы обеспечиваете контекст в форме пространства имен, в котором ваша функция `showInfo()` будет уникальной, а другой разработчик будет использовать другое пространство имен. В конечном счете этот механизм способствует инкапсуляции функциональных средств.

Главным ключевым словом для объявления пространств имен служит `namespace`. Оно должно быть в первой команде на PHP-странице (после необязательных комментариев), и ему не должна предшествовать HTML-разметка. Если вы хотите структурировать пространства имен, то можете создать иерархию с помощью (несколько необычного) символа обратной косой черты в виде разделителя:

```
namespace Project\Module\Submodule;
```

Это пространство имен теперь действует для всего файла. Если в этом файле вы создадите метод `showInfo()` и включите этот файл в какой-нибудь другой PHP-сценарий, то для выполнения этого метода можно использовать следующий код:

```
Project\Module\Submodule\showInfo();
```

Чтобы определить несколько пространств имен в одном файле, можно взять несколько операторов `namespace`, но предпочтительнее использовать фигурные скобки:

```
namespace Namespace1 {
  // ...
}

namespace Namespace2 {
  // ...
}
```

В PHP предусмотрено несколько вариантов использования пространств имен. Вы можете как предоставить полностью определенное имя пространства имен (как в предыдущем примере),

так и применить относительный синтаксис. Для локальных пространств имен удобнее использовать псевдоним (альтернативное имя). Синтаксис `use <namespace>` или `use <namespace> as <alias>` может значительно сократить объем кодирования и упростить доступ к пространству имен. Приведем пример импортирования пространства имен:

```
import Project\Module\Submodule;
Submodule\showInfo();
```

При использовании псевдонима код может выглядеть так:

```
import Project\Module\Submodule as Mod;
Mod\showInfo();
```

В пространстве имен доступ к встроенным PHP-классам может показаться проблематичным, поскольку PHP выполняет поиск этих классов в текущем пространстве имен. Но достаточно поставить перед именем класса обратную косую черту — и это заставит PHP "заглянуть" в глобальный список классов:

```
namespace Project\Module\Submodule;
$c = new \SoapClient('file.wsdl');
```

О пространствах имен можно еще много порассказать полезного, но, следуя логике построения справочника, мы ограничимся лишь акцентом на том, как они работают. Больше информации вы найдете в интернет-справочнике по адресу `http://php.net/namespace`.

Использование типажей

```
class MyTraitClass {
  use MyTrait1, MyTrait2 {
    MyTrait1::showTime as now;
    MyTrait2::showCopyright insteadof MyTrait1;
  }
}
```

Одной из новых возможностей в версии PHP 5.4 является поддержка *типажей*, т.е. средства для повторного использования существующего кода. Типаж (trait) представляет собой коллекцию функций (как у класса, но без реализации). Чтобы использовать это новое средство, вам необходимо создать типаж с помощью ключевого слова `trait`.

```
trait MyTrait1 {
  public function showTime() {
    $date = new DateTime('now');
```

```
    echo $date->format('H:i:s');
  }
}
```

Затем в классе вы можете загрузить уже созданный типаж, используя ключевое слово use. После этого все методы, входящие в состав типажа, поступают в полное ваше распоряжение:

```
class MyTraitClass {
  use MyTrait1;
}
$c = new MyTraitClass();
$c->showTime();
```

Вы можете также в одном классе использовать несколько типажей. Но если окажется, что в нескольких типажах определены одни и те же методы, вы должны проявить внимательность к их использованиею. В этом случае вы могли бы ожидать сообщение об ошибке (подобно работе с интерфейсами, в которых определены методы с одинаковыми именами). Но, в отличие от интерфейсов, типажи предлагают способ предотвращения этой ошибки. При загрузке типажа можно разрешить подобные конфликты по совпадению имен с помощью оператора insteadof. Предположим, к примеру, что у вас есть два типажа, MyTrait1 и MyTrait2, которые определяют метод showTime(). Следующий синтаксис гарантирует, что вы не получите сообщения об ошибке и что при выполнении кода будет использована реализация метода showTime() из типажа MyTrait1, а не из типажа MyTrait2:

```
class MyTraitClass {
  use MyTrait1, MyTrait2{
    MyTrait1::showTime insteadof MyTrait2;
  }
}
```

В PHP 5.4 есть также оператор as, который (в отличие от insteadof) позволяет не исключать конфликтующие методы, а, наоборот, включать один из них под другим именем (псевдонимом). С помощью ключевого слова as в типаже можно использовать даже абстрактные методы. В следующем коде продемонстрировано использование двух типажей, оператора insteadof и псевдонима, а результат его выполнения подобен представленному на рис. 4.3:

```
<?php
  trait MyTrait1 {
    public function showTime() {
      $date = new DateTime('now');
      echo $date->format('H:i:s');
    }
```

```
    public function showCopyright() {
      echo '&copy; Trait 1';
    }
  }

  trait MyTrait2 {
    public function showCopyright() {
      echo '&copy; Trait 2';
    }
  }

  class MyTraitClass {
    use MyTrait1, MyTrait2 {
      MyTrait1::showTime as now;
      MyTrait2::showCopyright insteadof MyTrait1;
    }
  }

  $c = new MyTraitClass();
  $c->now();
  echo '<hr />';
  $c->showCopyright();
?>
```

Использование двух типажей (`trait.php`)

Рис. 4.3. Демонстрация использования метода `showCopyright()` из второго типажа