Luke Welling
Laura Thomson

New PHP 7 Coverage

PHP and MySQL® Web Development

Fifth Edition

The **Aqueduct of Segovia** is one of the greatest surviving monuments of Roman engineering. Built during the time of the emperor Trajan in the first century A.D., it was designed to bring water from the foothills of the Sierra de Guadarrama to the city of Segovia, Spain, some 18 kilometers away.

The structure is built with over 20,000 roughhewn granite blocks, held together entirely without cement or clamps. The 278 meter-long section of the aqueduct that winds through the center of the city has two levels of arches, forming an elegant latticework of stone 34 meters above the city's streets.

After 2,000 years of weathering both natural and man-made calamities, this timeless example of human ingenuity is still in use today providing a supplemental water supply for the city.

```
class webPage implements Displayable
{
  function display()
  {
   // ...
  }
}
```

This example illustrates a roundabout kind of multiple inheritance because the `webPage` class can inherit from one class and implement one or more interfaces.

If you do not implement the methods specified in the interface (in this case, `display()`), you will get a fatal error.

Using Traits

Traits are a way to get the best aspects of multiple inheritance without the associated pain. In a trait, you can group together functionality that may be reused in multiple classes. A class can combine multiple traits, and traits can inherit from one another. Traits are an excellent set of building blocks for code re-use.

The key difference between interfaces and traits is that traits include an implementation, as opposed to merely specifying an interface that must be implemented.

You create a trait the same way as a class, but using the keyword trait instead, for example,

```
trait logger
{
  public function logmessage($message, $level='DEBUG')
  {
    // write $message to a log
  }
}
```

To use this trait, you could write code as follows:

```
class fileStorage
{
  use logger;

  function store($data) {
    // ...
    $this->logmessage($msg);
  }
}
```

The `fileStorage` class could override the `logmessage()` method by declaring its own, if needed. However, you should note that if the `fileStorage` class had inherited a `logmessage()` method from a parent, by default the trait `logmessage()` method would override it. That is, a trait's methods override inherited methods, but the current's class methods override a trait's methods.

One nice thing about traits is that you can combine multiple traits and when there are methods with the same names, you can explicitly specify which trait's functionality you wish to use. Consider the following example:

```
<?php
trait fileLogger
{
  public function logmessage($message, $level='DEBUG')
  {
    // write $message to a log file
}
}

trait sysLogger
{
  public function logmessage($message, $level='ERROR')
  {
    // write $message to the syslog
}
}

class fileStorage
{
  use fileLogger, sysLogger
  {
    fileLogger::logmessage insteadof sysLogger;
    sysLogger::logmessage as private logsysmessage;
  }

  function store($data)
  {
    // ...
    $this->logmessage($message);
    $this->logsysmessage($message);
  }
}
?>
```

We use the two different logging traits by listing them in the use clause. Because each of the traits implements the same `logmessage()` method, we must specify which one to use. If you don't specify this, PHP will generate a fatal error as it will not be able to resolve the conflict.

You can specify which one to use by using the `insteadof` keyword, as in the following example:

```
fileLogger::logmessage insteadof sysLogger;
```

This line explicitly tells PHP to use the `logmessage()` method from the `fileLogger` trait. However, in this example, we'd also like access to the `logmessage()` method from the `sysLogger` trait. In order to do so, we rename it using the `as` keyword, as follows:

```
sysLogger::logmessage as private logsysmessage;
```

This method will now be available as the `logsysmessage()` method. Note that in this particular example, we actually also changed the visibility of the method. This is not required, but is shown here so that you can see that this is possible.

You can even take it a step further, and build traits that include or consist entirely of other traits. This allows for true horizontal composability. To do this, include a `use` statement inside a trait, just as you would inside a class.

Designing Classes

Now that you know some of the concepts behind objects and classes and the syntax to implement them in PHP, it is time to look at how to design useful classes.

Many classes in your code will represent classes or categories of real-world objects. Examples of some classes you might use in Web development include pages, user interface components, shopping carts, error handling, product categories, or customers.

Objects in your code can also represent specific instances of the previously mentioned classes—for example, the home page, a particular button, or the shopping cart in use by Fred Smith at a particular time. Fred Smith himself can be represented by an object of type `customer`. Each item that Fred purchases can be represented as an object, belonging to a category or class.

In the preceding chapter, you used simple include files to give the fictional company TLA Consulting a consistent look and feel across the different pages of its website. Using classes and the timesaving power of inheritance, you can create a more advanced version of the same site.

Now you want to be able to quickly create pages for TLA that look and behave in the same way. You should be able to modify those pages to suit the different parts of the site.

For purposes of this example, you are going to create a `Page` class. The main goal of this class is to limit the amount of HTML needed to create a new page. It should allow you to alter the parts that change from page to page, while automatically generating the elements that stay the same. The class should provide a flexible framework for creating new pages and should not compromise your freedom.

Because you are generating the page from a script rather than with static HTML, you can add any number of clever things including functionality to

- Enable you to alter page elements in only one place. If you change the copyright notice or add an extra button, you should need to make the change in only a single place.
- Have default content for most parts of the page but be able to modify each element where required, setting custom values for elements such as the title and metatags.

- Recognize which page is being viewed and alter navigation elements to suit; there is no point in having a button that takes you to the home page located on the home page.
- Allow you to replace standard elements for particular pages. If, for instance, you want different navigation buttons in sections of the site, you should be able to replace the standard ones.

Writing the Code for Your Class

Having decided what you want the output from your code to look like and a few features you would like for it to have, how do you implement it? Later in the book, we discuss design and project management for large projects. For now, we concentrate on the parts specific to writing object-oriented PHP.

The class needs a logical name. Because it represents a page, you can call it `Page`. To declare a class called `Page`, type

```
class Page
{
}
```

The class needs some attributes. For this example, set elements that you might want changed from page to page as attributes of the class. The main contents of the page, which are a combination of HTML tags and text, are called `$content`. You can declare the content with the following line of code within the class definition:

```
public $content;
```

You can also set attributes to store the page's title. You will probably want to change this title to clearly show what particular page the visitor is looking at. Rather than have blank titles, you can provide a default title with the following declaration:

```
public $title = "TLA Consulting Pty Ltd";
```

Most commercial web pages include metatags to help search engines index them. To be useful, metatags should probably change from page to page. Again, you can provide a default value:

```
public $keywords = "TLA Consulting, Three Letter Abbreviation,
                    some of my best friends are search engines";
```

The navigation buttons shown on the original page in Figure 5.2 (see the preceding chapter) should probably be kept the same from page to page to avoid confusing people, but to change them easily, you can make them an attribute, too. Because the number of buttons might be variable, you can use an array and store both the text for the button and the URL it should point to:

```
public $buttons = array( "Home"     => "home.php",
                         "Contact"  => "contact.php",
                         "Services" => "services.php",
                         "Site Map" => "map.php"
                       );
```

To provide some functionality, the class also needs operations. You can start by providing accessor functions to set and get the values of the attributes you defined:

```
public function __set($name, $value)
{
  $this->$name = $value;
}
```

The `__set()` function does not contain error checking (for brevity), but this capability can be easily added later, as required. Because it is unlikely that you will be requesting any of these values from outside the class, you can elect not to provide a `__get()` function, as done here.

The main purpose of this class is to display a page of HTML, so you need a function. We called ours `Display()`, and it is as follows:

```
  public function Display()
  {
    echo "<html>\n<head>\n";
    $this -> DisplayTitle();
    $this -> DisplayKeywords();
    $this -> DisplayStyles();
    echo "</head>\n<body>\n";
    $this -> DisplayHeader();
    $this -> DisplayMenu($this->buttons);
    echo $this->content;
    $this -> DisplayFooter();
    echo "</body>\n</html>\n";
  }
```

The function includes a few simple `echo` statements to display HTML but mainly consists of calls to other functions in the class. As you have probably guessed from their names, these other functions display parts of the page.

Breaking up functions like this is not compulsory. All these separate functions might simply have been combined into one big function. We separated them out for a number of reasons.

Each function should have a defined task to perform. The simpler this task is, the easier writing and testing the function will be. Don't go too far; if you break up your program into too many small units, it might be hard to read.

Using inheritance, you can override operations. You can replace one large `Display()` function, but it is unlikely that you will want to change the way the entire page is displayed. It will be much better to break up the display functionality into a few self-contained tasks and be able to override only the parts that you want to change.

This `Display()` function calls `DisplayTitle()`, `DisplayKeywords()`, `DisplayStyles()`, `DisplayHeader()`, `DisplayMenu()`, and `DisplayFooter`(). This means that you need to define these operations. You can write operations or functions in this logical order, calling the operation or function before the actual code for the function. In many other languages, you need to write the function or operation before it can be called. Most of the operations are fairly simple and need to display some HTML and perhaps the contents of the attributes.

Listing 6.1 shows the complete class, saved as `page.php`, to include or require into other files.

Listing 6.1 **`page.php`—The Page Class Provides an Easy and Flexible Way to Create TLA Pages**

```
<?php
class Page
{
  // class Page's attributes
  public $content;
  public $title = "TLA Consulting Pty Ltd";
  public $keywords = "TLA Consulting, Three Letter Abbreviation,
                     some of my best friends are search engines";
  public $buttons = array("Home"    => "home.php",
                        "Contact"  => "contact.php",
                        "Services" => "services.php",
                        "Site Map" => "map.php"
                    );

  // class Page's operations
  public function __set($name, $value)
  {
    $this->$name = $value;
  }

  public function Display()
  {
    echo "<html>\n<head>\n";
    $this -> DisplayTitle();
    $this -> DisplayKeywords();
    $this -> DisplayStyles();
    echo "</head>\n<body>\n";
    $this -> DisplayHeader();
    $this -> DisplayMenu($this->buttons);
    echo $this->content;
    $this -> DisplayFooter();
    echo "</body>\n</html>\n";
  }

  public function DisplayTitle()
  {
    echo "<title>".$this->title."</title>";
  }

  public function DisplayKeywords()
  {
    echo "<meta name='keywords' content='".$this->keywords."'/>";
  }
```