BACK-END

Object-Oriented PHP for Beginners

by Jason Lengstorf *23 Dec 2011* *672 Comments* English

For many PHP programmers, object-oriented programming is a frightening concept, full of complicated syntax and other roadblocks. As detailed in my book, *Pro PHP and jQuery,* you'll learn the concepts behind **object-oriented programming** (OOP), a style of coding in which related actions are grouped into classes to aid in creating more-compact, effective code.

Understanding Object-Oriented Programming

Object-oriented programming is a style of coding that allows developers to group similar tasks into **classes**. This helps keep code following the tenet "don't repeat yourself" (DRY) and easy-to-maintain.

> "*Object-oriented programming is a style of coding that allows developers to group similar tasks into **classes**.*"

One of the major benefits of DRY programming is that, if a piece of information changes in your program, usually **only one change is required to update the code.** One of the biggest nightmares for developers is maintaining code where data is declared over and over again, meaning any changes to the program become an infinitely more frustrating game of *Where's Waldo?* as they hunt for duplicated data and functionality.

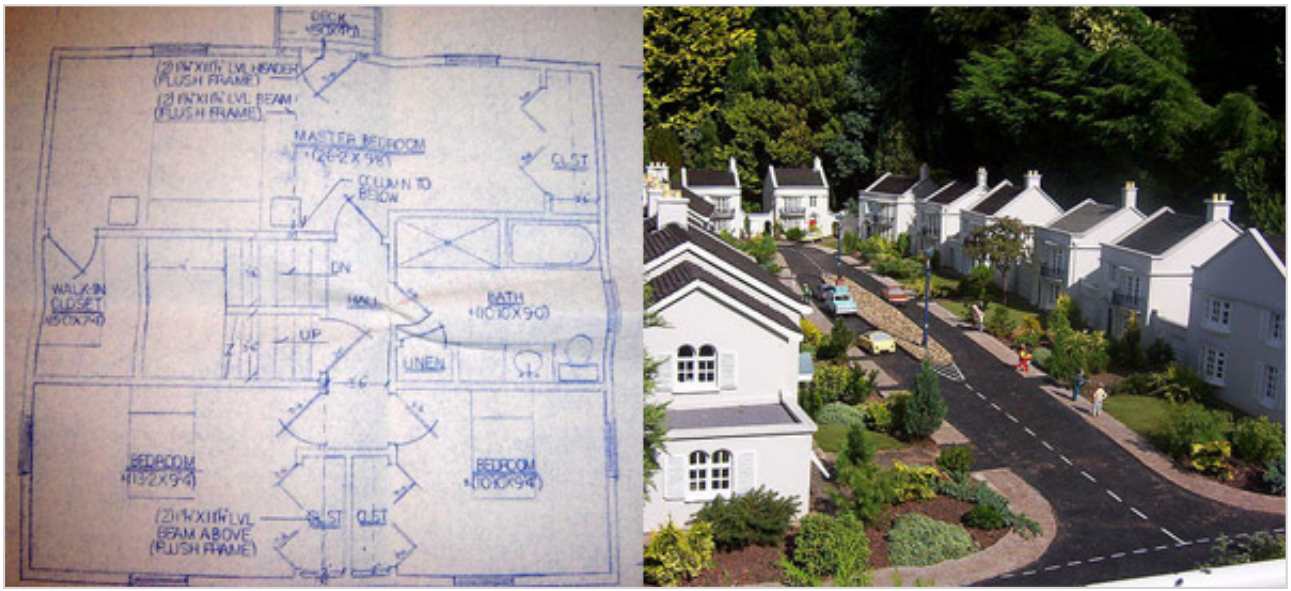
OOP is intimidating to a lot of developers because it introduces new syntax and, at a

glance, appears to be far more complex than simple procedural, or inline, code. However, upon closer inspection, OOP is actually a very straightforward and ultimately simpler approach to programming.

Understanding Objects and Classes

Before you can get too deep into the finer points of OOP, a basic understanding of the differences between **objects** and **classes** is necessary. This section will go over the building blocks of classes, their different capabilities, and some of their uses.

Recognizing the Differences Between Objects and Classes



Photos by Instant Jefferson and John Wardell

> *Developers start talking about objects and classes, and they appear to be interchangeable terms. This is not the case, however.*

Right off the bat, there's confusion in OOP: seasoned developers start talking about objects and classes, and they appear to be interchangeable terms. This is not the case, however, though the difference can be tough to wrap your head around at first.

A class, for example, is like **a blueprint for a house**. It defines the shape of the

house on paper, with relationships between the different parts of the house clearly defined and planned out, even though the house doesn't exist.

An object, then, is like **the actual house** built according to that blueprint. The data stored in the object is like the wood, wires, and concrete that compose the house: without being assembled according to the blueprint, it's just a pile of stuff. However, when it all comes together, it becomes an organized, useful house.

Classes form the structure of data and actions and use that information to build objects. More than one object can be built from the same class at the same time, each one independent of the others. Continuing with our construction analogy, it's similar to the way an entire subdivision can be built from the same blueprint: 150 different houses that all look the same but have different families and decorations inside.

Structuring Classes

The syntax to create a class is pretty straightforward: declare a class using the `class` keyword, followed by the name of the class and a set of curly braces (`{}`):

```
<?php

class MyClass
{
  // Class properties and methods go here
}

?>
```

After creating the class, a new class can be instantiated and stored in a variable using the `new` keyword:

```
$obj = new MyClass;
```

To see the contents of the class, use `var_dump()`:

```
var_dump($obj);
```

Try out this process by putting all the preceding code in a new file called `test.php` in [your local] testing folder:

```
<?php

class MyClass
{
    // Class properties and methods go here
}

$obj = new MyClass;

var_dump($obj);

?>
```

Load the page in your browser at `http://localhost/test.php` and the following should display:

```
object(MyClass)#1 (0) { }
```

In its simplest form, you've just completed your first OOP script.

Defining Class Properties

To add data to a class, **properties**, or class-specific variables, are used. These work exactly like regular variables, except they're bound to the object and therefore can only be accessed using the object.

To add a property to `MyClass`, add the following code to your script:

```
<?php

class MyClass
{
  public $prop1 = "I'm a class property!";
}

$obj = new MyClass;
```

```
09
10  var_dump($obj);
11
12  ?>
```

The keyword `public` determines the visibility of the property, which you'll learn about a little later in this chapter. Next, the property is named using standard variable syntax, and a value is assigned (though class properties do not need an initial value).

To read this property and output it to the browser, reference the object from which to read and the property to be read:

```
1  echo $obj->prop1;
```

Because multiple instances of a class can exist, if the individual object is not referenced, the script would be unable to determine which object to read from. The use of the arrow (`->`) is an OOP construct that accesses the contained properties and methods of a given object.

Modify the script in `test.php` to read out the property rather than dumping the whole class by modifying the code as shown:

```
01  <?php
02
03  class MyClass
04  {
05    public $prop1 = "I'm a class property!";
06  }
07
08  $obj = new MyClass;
09
10  echo $obj->prop1; // Output the property
11
12  ?>
```

Reloading your browser now outputs the following:

```
1  I'm a class property!
```

Defining Class Methods

Methods are class-specific functions. Individual actions that an object will be able to perform are defined within the class as methods.

For instance, to create methods that would set and get the value of the class property `$prop1`, add the following to your code:

```
<?php

class MyClass
{
  public $prop1 = "I'm a class property!";

  public function setProperty($newval)
  {
      $this->prop1 = $newval;
  }

  public function getProperty()
  {
      return $this->prop1 . "<br />";
  }
}

$obj = new MyClass;

echo $obj->prop1;

?>
```

Note — OOP allows objects to reference themselves using `$this`. When working within a method, use `$this` in the same way you would use the object name outside the class.

To use these methods, call them just like regular functions, but first, reference the object they belong to. Read the property from `MyClass`, change its value, and read it out again by making the modifications below:

```
<?php

class MyClass
```

```
{
  public $prop1 = "I'm a class property!";

  public function setProperty($newval)
  {
      $this->prop1 = $newval;
  }

  public function getProperty()
  {
      return $this->prop1 . "<br />";
  }
}

$obj = new MyClass;

echo $obj->getProperty(); // Get the property value

$obj->setProperty("I'm a new property value!"); // Set a new one

echo $obj->getProperty(); // Read it out again to show the change

?>
```

Reload your browser, and you'll see the following:

```
I'm a class property!
I'm a new property value!
```

> *"The power of OOP becomes apparent when using multiple instances of the same class."*

```
<?php

class MyClass
{
  public $prop1 = "I'm a class property!";

  public function setProperty($newval)
  {
      $this->prop1 = $newval;
  }

  public function getProperty()
```

```
    {
        return $this->prop1 . "<br />";
    }
}

// Create two objects
$obj = new MyClass;
$obj2 = new MyClass;

// Get the value of $prop1 from both objects
echo $obj->getProperty();
echo $obj2->getProperty();

// Set new values for both objects
$obj->setProperty("I'm a new property value!");
$obj2->setProperty("I belong to the second instance!");

// Output both objects' $prop1 value
echo $obj->getProperty();
echo $obj2->getProperty();

?>
```

When you load the results in your browser, they read as follows:

```
I'm a class property!
I'm a class property!
I'm a new property value!
I belong to the second instance!
```

As you can see, **OOP keeps objects as separate entities**, which makes for easy separation of different pieces of code into small, related bundles.

Magic Methods in OOP

To make the use of objects easier, PHP also provides a number of **magic methods**, or special methods that are called when certain common actions occur within objects. This allows developers to perform a number of useful tasks with relative ease.

Using Constructors and Destructors

When an object is instantiated, it's often desirable to set a few things right off the bat. To handle this, PHP provides the magic method `__construct()`, which is called automatically whenever a new object is created.

For the purpose of illustrating the concept of constructors, add a constructor to `MyClass` that will output a message whenever a new instance of the class is created:

```
01  <?php
02
03  class MyClass
04  {
05    public $prop1 = "I'm a class property!";
06
07    public function __construct()
08    {
09        echo 'The class "', __CLASS__, '" was initiated!<br />';
10    }
11
12    public function setProperty($newval)
13    {
14        $this->prop1 = $newval;
15    }
16
17    public function getProperty()
18    {
19        return $this->prop1 . "<br />";
20    }
21  }
22
23  // Create a new object
24  $obj = new MyClass;
25
26  // Get the value of $prop1
27  echo $obj->getProperty();
28
29  // Output a message at the end of the file
30  echo "End of file.<br />";
31
32  ?>
```

Note — `__CLASS__` returns the name of the class in which it is called; this is what is known as a magic constant. There are several available magic constants, which you

can read more about in the PHP manual.

Reloading the file in your browser will produce the following result:

```
The class "MyClass" was initiated!
I'm a class property!
End of file.
```

To call a function when the object is destroyed, the `__destruct()` magic method is available. This is useful for class cleanup (closing a database connection, for instance).

Output a message when the object is destroyed by defining the magic method `__destruct()` in `MyClass`:

```
<?php

class MyClass
{
  public $prop1 = "I'm a class property!";

  public function __construct()
  {
      echo 'The class "', __CLASS__, '" was initiated!<br />';
  }

  public function __destruct()
  {
      echo 'The class "', __CLASS__, '" was destroyed.<br />';
  }

  public function setProperty($newval)
  {
      $this->prop1 = $newval;
  }

  public function getProperty()
  {
      return $this->prop1 . "<br />";
  }
}

// Create a new object
$obj = new MyClass;

```

```
// Get the value of $prop1
echo $obj->getProperty();

// Output a message at the end of the file
echo "End of file.<br />";

?>
```

With a destructor defined, reloading the test file results in the following output:

```
The class "MyClass" was initiated!
I'm a class property!
End of file.
The class "MyClass" was destroyed.
```

> "*When the end of a file is reached, PHP automatically releases all resources.*"

To explicitly trigger the destructor, you can destroy the object using the function `unset()`:

```
<?php

class MyClass
{
  public $prop1 = "I'm a class property!";

  public function __construct()
  {
      echo 'The class "', __CLASS__, '" was initiated!<br />';
  }

  public function __destruct()
  {
      echo 'The class "', __CLASS__, '" was destroyed.<br />';
  }

  public function setProperty($newval)
  {
      $this->prop1 = $newval;
  }

  public function getProperty()
```

```
    {
        return $this->prop1 . "<br />";
    }
}

// Create a new object
$obj = new MyClass;

// Get the value of $prop1
echo $obj->getProperty();

// Destroy the object
unset($obj);

// Output a message at the end of the file
echo "End of file.<br />";

?>
```

Now the result changes to the following when loaded in your browser:

```
The class "MyClass" was initiated!
I'm a class property!
The class "MyClass" was destroyed.
End of file.
```

Converting to a String

To avoid an error if a script attempts to output `MyClass` as a string, another magic method is used called `__toString()`.

Without `__toString()`, *attempting to output the object as a string results in a fatal error*. Attempt to use `echo` to output the object without a magic method in place:

```
<?php

class MyClass
{
    public $prop1 = "I'm a class property!";

    public function __construct()
    {
        echo 'The class "', __CLASS__, '" was initiated!<br />';
    }
```

```
    public function __destruct()
    {
        echo 'The class "', __CLASS__, '" was destroyed.<br />';
    }

    public function setProperty($newval)
    {
        $this->prop1 = $newval;
    }

    public function getProperty()
    {
        return $this->prop1 . "<br />";
    }
}

// Create a new object
$obj = new MyClass;

// Output the object as a string
echo $obj;

// Destroy the object
unset($obj);

// Output a message at the end of the file
echo "End of file.<br />";

?>
```

This results in the following:

```
The class "MyClass" was initiated!

Catchable fatal error: Object of class MyClass could not be conver
```

To avoid this error, add a `__toString()` method:

```
<?php

class MyClass
{
    public $prop1 = "I'm a class property!";

    public function __construct()
```

```
    public function __construct()
    {
        echo 'The class "', __CLASS__, '" was initiated!<br />';
    }

    public function __destruct()
    {
        echo 'The class "', __CLASS__, '" was destroyed.<br />';
    }

    public function __toString()
    {
        echo "Using the toString method: ";
        return $this->getProperty();
    }

    public function setProperty($newval)
    {
        $this->prop1 = $newval;
    }

    public function getProperty()
    {
        return $this->prop1 . "<br />";
    }
}

// Create a new object
$obj = new MyClass;

// Output the object as a string
echo $obj;

// Destroy the object
unset($obj);

// Output a message at the end of the file
echo "End of file.<br />";

?>
```

In this case, attempting to convert the object to a string results in a call to the `getProperty()` method. Load the test script in your browser to see the result:

```
The class "MyClass" was initiated!
Using the toString method: I'm a class property!
The class "MyClass" was destroyed.
End of file.
```

Tip — In addition to the magic methods discussed in this section, several others are available. For a complete list of magic methods, see the PHP manual page.

Using Class Inheritance

Classes can inherit the methods and properties of another class using the `extends` keyword. For instance, to create a second class that extends `MyClass` and adds a method, you would add the following to your test file:

```
01  <?php
02
03  class MyClass
04  {
05    public $prop1 = "I'm a class property!";
06
07    public function __construct()
08    {
09        echo 'The class "', __CLASS__, '" was initiated!<br />';
10    }
11
12    public function __destruct()
13    {
14        echo 'The class "', __CLASS__, '" was destroyed.<br />';
15    }
16
17    public function __toString()
18    {
19        echo "Using the toString method: ";
20        return $this->getProperty();
21    }
22
23    public function setProperty($newval)
24    {
25        $this->prop1 = $newval;
26    }
27
28    public function getProperty()
29    {
30        return $this->prop1 . "<br />";
31    }
32  }
33
34  class MyOtherClass extends MyClass
```

```
{
  public function newMethod()
  {
      echo "From a new method in " . __CLASS__ . ".<br />";
  }
}

// Create a new object
$newobj = new MyOtherClass;

// Output the object as a string
echo $newobj->newMethod();

// Use a method from the parent class
echo $newobj->getProperty();

?>
```

Upon reloading the test file in your browser, the following is output:

```
The class "MyClass" was initiated!
From a new method in MyOtherClass.
I'm a class property!
The class "MyClass" was destroyed.
```

Overwriting Inherited Properties and Methods

To change the behavior of an existing property or method in the new class, you can simply overwrite it by declaring it again in the new class:

```
<?php

class MyClass
{
  public $prop1 = "I'm a class property!";

  public function __construct()
  {
      echo 'The class "', __CLASS__, '" was initiated!<br />';
  }

  public function __destruct()
  {
      echo 'The class "', __CLASS__, '" was destroyed.<br />';
  }

```

```
    public function __toString()
    {
        echo "Using the toString method: ";
        return $this->getProperty();
    }

    public function setProperty($newval)
    {
        $this->prop1 = $newval;
    }

    public function getProperty()
    {
        return $this->prop1 . "<br />";
    }
}

class MyOtherClass extends MyClass
{
    public function __construct()
    {
        echo "A new constructor in " . __CLASS__ . ".<br />";
    }

    public function newMethod()
    {
        echo "From a new method in " . __CLASS__ . ".<br />";
    }
}

// Create a new object
$newobj = new MyOtherClass;

// Output the object as a string
echo $newobj->newMethod();

// Use a method from the parent class
echo $newobj->getProperty();

?>
```

This changes the output in the browser to:

```
A new constructor in MyOtherClass.
From a new method in MyOtherClass.
I'm a class property!
The class "MyClass" was destroyed.
```

Preserving Original Method Functionality While Overwriting Methods

To add new functionality to an inherited method while keeping the original method intact, use the `parent` keyword with the **scope resolution operator** (`::`):

```
01  <?php
02
03  class MyClass
04  {
05    public $prop1 = "I'm a class property!";
06
07    public function __construct()
08    {
09        echo 'The class "', __CLASS__, '" was initiated!<br />';
10    }
11
12    public function __destruct()
13    {
14        echo 'The class "', __CLASS__, '" was destroyed.<br />';
15    }
16
17    public function __toString()
18    {
19        echo "Using the toString method: ";
20        return $this->getProperty();
21    }
22
23    public function setProperty($newval)
24    {
25        $this->prop1 = $newval;
26    }
27
28    public function getProperty()
29    {
30        return $this->prop1 . "<br />";
31    }
32  }
33
34  class MyOtherClass extends MyClass
35  {
36    public function __construct()
37    {
38        parent::__construct(); // Call the parent class's construct
39        echo "A new constructor in " . __CLASS__ . ".<br />";
40    }
41
42    public function newMethod()
43    {
```

```
44          echo "From a new method in " . __CLASS__ . ".<br />";
45      }
46  }
47
48  // Create a new object
49  $newobj = new MyOtherClass;
50
51  // Output the object as a string
52  echo $newobj->newMethod();
53
54  // Use a method from the parent class
55  echo $newobj->getProperty();
56
57  ?>
```

This outputs the result of both the parent constructor and the new class's constructor:

```
1  The class "MyClass" was initiated!
2  A new constructor in MyOtherClass.
3  From a new method in MyOtherClass.
4  I'm a class property!
5  The class "MyClass" was destroyed.
```

Assigning the Visibility of Properties and Methods

For added control over objects, methods and properties are assigned visibility. This controls how and from where properties and methods can be accessed. There are three visibility keywords: `public`, `protected`, and `private`. In addition to its visibility, a method or property can be declared as `static`, which allows them to be accessed without an instantiation of the class.

> *"For added control over objects, methods and properties are assigned visibility."*

Note — Visibility is a new feature as of PHP 5. For information on OOP compatibility with PHP 4, see the PHP manual page.

Public Properties and Methods

All the methods and properties you've used so far have been public. This means that they can be accessed anywhere, both within the class and externally.

Protected Properties and Methods

When a property or method is declared `protected`, **it can only be accessed within the class itself or in descendant classes** (classes that extend the class containing the protected method).

Declare the `getProperty()` method as protected in `MyClass` and try to access it directly from outside the class:

```
01  <?php
02
03  class MyClass
04  {
05    public $prop1 = "I'm a class property!";
06
07    public function __construct()
08    {
09        echo 'The class "', __CLASS__, '" was initiated!<br />';
10    }
11
12    public function __destruct()
13    {
14        echo 'The class "', __CLASS__, '" was destroyed.<br />';
15    }
16
17    public function __toString()
18    {
19        echo "Using the toString method: ";
20        return $this->getProperty();
21    }
22
23    public function setProperty($newval)
24    {
25        $this->prop1 = $newval;
26    }
27
28    protected function getProperty()
29    {
30        return $this->prop1 . "<br />";
31    }
32  }
33
```

```
class MyOtherClass extends MyClass
{
  public function __construct()
  {
      parent::__construct();
echo "A new constructor in " . __CLASS__ . ".<br />";
  }

  public function newMethod()
  {
      echo "From a new method in " . __CLASS__ . ".<br />";
  }
}

// Create a new object
$newobj = new MyOtherClass;

// Attempt to call a protected method
echo $newobj->getProperty();

?>
```

Upon attempting to run this script, the following error shows up:

```
The class "MyClass" was initiated!
A new constructor in MyOtherClass.

Fatal error: Call to protected method MyClass::getProperty() from
```

Now, create a new method in `MyOtherClass` to call the `getProperty()` method:

```
<?php

class MyClass
{
  public $prop1 = "I'm a class property!";

  public function __construct()
  {
      echo 'The class "', __CLASS__, '" was initiated!<br />';
  }

  public function __destruct()
  {
      echo 'The class "', __CLASS__, '" was destroyed.<br />';
  }
```

```
    }

    public function __toString()
    {
        echo "Using the toString method: ";
        return $this->getProperty();
    }

    public function setProperty($newval)
    {
        $this->prop1 = $newval;
    }

    protected function getProperty()
    {
        return $this->prop1 . "<br />";
    }
}

class MyOtherClass extends MyClass
{
    public function __construct()
    {
        parent::__construct();
echo "A new constructor in " . __CLASS__ . ".<br />";
    }

    public function newMethod()
    {
        echo "From a new method in " . __CLASS__ . ".<br />";
    }

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

// Create a new object
$newobj = new MyOtherClass;

// Call the protected method from within a public method
echo $newobj->callProtected();

?>
```

This generates the desired result:

```
The class "MyClass" was initiated!
A new constructor in MyOtherClass.
I'm a class property!
The class "MyClass" was destroyed.
```

Private Properties and Methods

A property or method declared `private` is accessible **only from within the class that defines it**. This means that *even if a new class extends the class that defines a private property,* that property or method will not be available at all within the child class.

To demonstrate this, declare `getProperty()` as private in `MyClass`, and attempt to call `callProtected()` from `MyOtherClass`:

```php
<?php

class MyClass
{
  public $prop1 = "I'm a class property!";

  public function __construct()
  {
      echo 'The class "', __CLASS__, '" was initiated!<br />';
  }

  public function __destruct()
  {
      echo 'The class "', __CLASS__, '" was destroyed.<br />';
  }

  public function __toString()
  {
      echo "Using the toString method: ";
      return $this->getProperty();
  }

  public function setProperty($newval)
  {
      $this->prop1 = $newval;
  }

  private function getProperty()
  {
      return $this->prop1 . "<br />";
```

```
  }
}

class MyOtherClass extends MyClass
{
  public function __construct()
  {
      parent::__construct();
      echo "A new constructor in " . __CLASS__ . ".<br />";
  }

  public function newMethod()
  {
      echo "From a new method in " . __CLASS__ . ".<br />";
  }

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

// Create a new object
$newobj = new MyOtherClass;

// Use a method from the parent class
echo $newobj->callProtected();

?>
```

Reload your browser, and the following error appears:

```
The class "MyClass" was initiated!
A new constructor in MyOtherClass.

Fatal error: Call to private method MyClass::getProperty() from co
```

Static Properties and Methods

A method or property declared `static` can be accessed without first instantiating the class; you simply supply the class name, scope resolution operator, and the property or method name.

> *"One of the major benefits to using static properties is that they keep their stored values for the duration of the script."*

To demonstrate this, add a static property called `$count` and a static method called `plusOne()` to `MyClass`. Then set up a `do...while` loop to output the incremented value of `$count` as long as the value is less than 10:

```
01  <?php
02
03  class MyClass
04  {
05    public $prop1 = "I'm a class property!";
06
07    public static $count = 0;
08
09    public function __construct()
10    {
11        echo 'The class "', __CLASS__, '" was initiated!<br />';
12    }
13
14    public function __destruct()
15    {
16        echo 'The class "', __CLASS__, '" was destroyed.<br />';
17    }
18
19    public function __toString()
20    {
21        echo "Using the toString method: ";
22        return $this->getProperty();
23    }
24
25    public function setProperty($newval)
26    {
27        $this->prop1 = $newval;
28    }
29
30    private function getProperty()
31    {
32        return $this->prop1 . "<br />";
33    }
34
35    public static function plusOne()
36    {
37        return "The count is " . ++self::$count . ".<br />";
38    }
39  }
```

```php

class MyOtherClass extends MyClass
{
  public function __construct()
  {
      parent::__construct();
      echo "A new constructor in " . __CLASS__ . ".<br />";
  }

  public function newMethod()
  {
      echo "From a new method in " . __CLASS__ . ".<br />";
  }

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

do
{
  // Call plusOne without instantiating MyClass
  echo MyClass::plusOne();
} while ( MyClass::$count < 10 );

?>
```

Note — When accessing static properties, the dollar sign (`$`) comes *after the scope resolution operator.*

When you load this script in your browser, the following is output:

```
The count is 1.
The count is 2.
The count is 3.
The count is 4.
The count is 5.
The count is 6.
The count is 7.
The count is 8.
The count is 9.
The count is 10.
```

Commenting with DocBlocks

> *"The DocBlock commenting style is a widely accepted method of documenting classes."*

While not an official part of OOP, the DocBlock commenting style is a widely accepted method of documenting classes. Aside from providing a standard for developers to use when writing code, it has also been adopted by many of the most popular software development kits (SDKs), such as Eclipse and NetBeans, and will be used to generate code hints.

A DocBlock is defined by using a block comment that starts with an additional asterisk:

```
1 /**
2  * This is a very basic DocBlock
3  */
```

The real power of DocBlocks comes with the ability to use **tags**, which start with an at symbol (`@`) immediately followed by the tag name and the value of the tag. **DocBlock tags allow developers to define authors of a file, the license for a class, the property or method information, and other useful information.**

The most common tags used follow:

- **@author**: The author of the current element (which might be a class, file, method, or any bit of code) are listed using this tag. Multiple author tags can be used in the same DocBlock if more than one author is credited. The format for the author name is `John Doe <john.doe@email.com>`.
- **@copyright**: This signifies the copyright year and name of the copyright holder for the current element. The format is `2010 Copyright Holder`.
- **@license**: This links to the license for the current element. The format for the license information is `http://www.example.com/path/to/license.txt License Name`.
- **@var**: This holds the type and description of a variable or class property. The format is `type element description`.

- **@param**: This tag shows the type and description of a function or method parameter. The format is `type $element_name element description`.
- **@return**: The type and description of the return value of a function or method are provided in this tag. The format is `type return element description`.

A sample class commented with DocBlocks might look like this:

```
<?php

/**
 * A simple class
 *
 * This is the long description for this class,
 * which can span as many lines as needed. It is
 * not required, whereas the short description is
 * necessary.
 *
 * It can also span multiple paragraphs if the
 * description merits that much verbiage.
 *
 * @author Jason Lengstorf <jason.lengstorf@ennuidesign.com>
 * @copyright 2010 Ennui Design
 * @license http://www.php.net/license/3_01.txt PHP License 3.01
 */
class SimpleClass
{
  /**
   * A public variable
   *
   * @var string stores data for the class
   */
  public $foo;

  /**
   * Sets $foo to a new value upon class instantiation
   *
   * @param string $val a value required for the class
   * @return void
   */
  public function __construct($val)
  {
      $this->foo = $val;
  }

  /**
   * Multiplies two integers
   *
   * Accepts a pair of integers and returns the
```

```
     * product of the two.
     *
     * @param int $bat a number to be multiplied
     * @param int $baz a number to be multiplied
     * @return int the product of the two parameters
     */
    public function bar($bat, $baz)
    {
        return $bat * $baz;
    }
}

?>
```

Once you scan the preceding class, the benefits of DocBlock are apparent: everything is clearly defined so that the next developer can pick up the code and *never have to wonder what a snippet of code does or what it should contain.*

Comparing Object-Oriented and Procedural Code

There's not really a right and wrong way to write code. That being said, **this section outlines a strong argument for adopting an object-oriented approach in software development, especially in large applications.**

Reason 1: Ease of Implementation

> *"While it may be daunting at first, OOP actually provides an easier approach to dealing with data."*

While it may be daunting at first, OOP actually provides an easier approach to dealing with data. Because an object can store data internally, variables don't need to be passed from function to function to work properly.

Also, because *multiple instances of the same class can exist simultaneously*, dealing with large data sets is infinitely easier. For instance, imagine you have two people's

information being processed in a file. They need names, occupations, and ages.

The Procedural Approach

Here's the procedural approach to our example:

```php
<?php

function changeJob($person, $newjob)
{
  $person['job'] = $newjob; // Change the person's job
  return $person;
}

function happyBirthday($person)
{
  ++$person['age']; // Add 1 to the person's age
  return $person;
}

$person1 = array(
  'name' => 'Tom',
  'job' => 'Button-Pusher',
  'age' => 34
);

$person2 = array(
  'name' => 'John',
  'job' => 'Lever-Puller',
  'age' => 41
);

// Output the starting values for the people
echo "<pre>Person 1: ", print_r($person1, TRUE), "</pre>";
echo "<pre>Person 2: ", print_r($person2, TRUE), "</pre>";

// Tom got a promotion and had a birthday
$person1 = changeJob($person1, 'Box-Mover');
$person1 = happyBirthday($person1);

// John just had a birthday
$person2 = happyBirthday($person2);

// Output the new values for the people
echo "<pre>Person 1: ", print_r($person1, TRUE), "</pre>";
echo "<pre>Person 2: ", print_r($person2, TRUE), "</pre>";

?>
```

When executed, the code outputs the following:

```
Person 1: Array
(
  [name] => Tom
  [job] => Button-Pusher
  [age] => 34
)
Person 2: Array
(
  [name] => John
  [job] => Lever-Puller
  [age] => 41
)
Person 1: Array
(
  [name] => Tom
  [job] => Box-Mover
  [age] => 35
)
Person 2: Array
(
  [name] => John
  [job] => Lever-Puller
  [age] => 42
)
```

While this code isn't necessarily bad, there's a lot to keep in mind while coding. **The array of the affected person's attributes must be passed and returned from each function call**, which leaves margin for error.

To clean up this example, it would be desirable to **leave as few things up to the developer as possible.** Only absolutely essential information for the current operation should need to be passed to the functions.

This is where OOP steps in and helps you clean things up.

The OOP Approach

Here's the OOP approach to our example:

```
<?php

```

```php
class Person
{
  private $_name;
  private $_job;
  private $_age;

  public function __construct($name, $job, $age)
  {
      $this->_name = $name;
      $this->_job = $job;
      $this->_age = $age;
  }

  public function changeJob($newjob)
  {
      $this->_job = $newjob;
  }

  public function happyBirthday()
  {
      ++$this->_age;
  }
}

// Create two new people
$person1 = new Person("Tom", "Button-Pusher", 34);
$person2 = new Person("John", "Lever Puller", 41);

// Output their starting point
echo "<pre>Person 1: ", print_r($person1, TRUE), "</pre>";
echo "<pre>Person 2: ", print_r($person2, TRUE), "</pre>";

// Give Tom a promotion and a birthday
$person1->changeJob("Box-Mover");
$person1->happyBirthday();

// John just gets a year older
$person2->happyBirthday();

// Output the ending values
echo "<pre>Person 1: ", print_r($person1, TRUE), "</pre>";
echo "<pre>Person 2: ", print_r($person2, TRUE), "</pre>";

?>
```

This outputs the following in the browser:

```
Person 1: Person Object
```

```
(
  [_name:private] => Tom
  [_job:private] => Button-Pusher
  [_age:private] => 34
)

Person 2: Person Object
(
  [_name:private] => John
  [_job:private] => Lever Puller
  [_age:private] => 41
)

Person 1: Person Object
(
  [_name:private] => Tom
  [_job:private] => Box-Mover
  [_age:private] => 35
)

Person 2: Person Object
(
  [_name:private] => John
  [_job:private] => Lever Puller
  [_age:private] => 42
)
```

There's a little bit more setup involved to make the approach object oriented, but after the class is defined, creating and modifying people is a breeze; **a person's information does not need to be passed or returned from methods, and only absolutely essential information is passed to each method.**

> *"OOP will significantly reduce your workload if implemented properly."*

On the small scale, this difference may not seem like much, but as your applications grow in size, OOP will significantly reduce your workload if implemented properly.

Tip — *Not everything needs to be object oriented.* A quick function that handles something small in one place inside the application does not necessarily need to be wrapped in a class. Use your best judgment when deciding between object-oriented and procedural approaches.

Reason 2: Better Organization

Another benefit of OOP is how well it lends itself to being **easily packaged and cataloged.** Each class can generally be kept in its own separate file, and if a uniform naming convention is used, accessing the classes is extremely simple.

Assume you've got an application with 150 classes that are called dynamically through a controller file at the root of your application filesystem. All 150 classes follow the naming convention `class.classname.inc.php` and reside in the `inc` folder of your application.

The controller can implement PHP's `__autoload()` function to dynamically pull in only the classes it needs as they are called, rather than including all 150 in the controller file just in case or coming up with some clever way of including the files in your own code:

```
<?php
  function __autoload($class_name)
  {
      include_once 'inc/class.' . $class_name . '.inc.php';
  }
?>
```

Having each class in a separate file also makes code more portable and easier to reuse in new applications without a bunch of copying and pasting.

Reason 3: Easier Maintenance

Due to the more compact nature of OOP when done correctly, **changes in the code are usually much easier to spot** and make than in a long spaghetti code procedural implementation.

If a particular array of information gains a new attribute, a procedural piece of software may require (in a worst-case scenario) that the new attribute be added to each function that uses the array.

An OOP application could potentially be updated as easily adding the new property and then adding the methods that deal with said property.

A lot of the benefits covered in this section are the product of **OOP in combination with DRY programming practices.** It is definitely possible to create easy-to-maintain procedural code that doesn't cause nightmares, and it is equally possible to create awful object-oriented code. [*Pro PHP and jQuery*] will attempt to demonstrate a combination of good coding habits in conjunction with OOP to generate clean code that's easy to read and maintain.

Summary

At this point, you should feel comfortable with the object-oriented programming style. Learning OOP is a great way to take your programming to that next level. When implemented properly, OOP will help you produce easy-to-read, easy-to-maintain, portable code that will save you (and the developers who work with you) hours of extra work. Are you stuck on something that wasn't covered in this article? Are you already using OOP and have some tips for beginners? Share them in the comments!

Author's Note — This tutorial was an excerpt from Pro PHP and jQuery (Apress, 2010).