PHP v5 Gets Serious About Objects: Part Two

By Scott Courtney

In the first installment of this two-part article, we looked at some of the interesting new features for object-oriented (OO) programming in PHP version 5. This new version introduces OO features that put PHP's object support in the same league as Java and other OO languages, although PHP is still (by choice and by design) not a "pure" OO language.

This month, we will look at some of the more advanced OO features in the new PHP—tools that not only help with basic object-oriented programming, but which also help the designer to model the problem in more elegant ways. If you are new to PHP and/or new to OO programming (OOP), you may want to read Part 1 of this article before delving into the material that follows.

INTERFACES

Whereas a class defines and implements the behaviors and data members of an object, an interface simply specifies available functionality without providing details on how it is implemented. Think of an interface as a contract that represents an agreement that any code that claims to implement the interface will provide certain functions.

In PHP, as in Java, interfaces are defined very much like classes except using the **interface** keyword instead of the class keyword. The interface body does not contain any executable code, but rather only the function declarations that are mandated by its contract. Any class

FIGURE 1: INTERFACES ALLOW A CLASS TO DECLARE "I AM A ..." WITHOUT NECESSARILY INHERITING FROM ANY SPECIFIC BASE CLASS

```
interface GoGetterWidget {
    function getSomething($what);

    function getAnotherThingy($what);

    function getDefaultThing();

}

class ClassOne extends YourAppClass implements GoGetterWidget {

    function getSomething($what) {
        return "I am getting " . $what;
    }

    function getAnotherThing($what) {
        return $what . " has turned into something else!";
    }

    function getDefaultThing() {
        return "";
    }
}

class ClassTwo extends SomeOtherClass implements GoGetterWidget {

    function getSomething($what) {
        return "I am getting a better thing: " . $what;
    }

    function getAnotherThing($what) {
        return $what . " has mutated into something new!";
    }

    function getDefaultThing() {
        return "DEFAULT THING";
    }
}
```

can then declare that it implements the interface, but doing so requires that the class declare and implement all of the interface's mandated functions. FIGURE 1 provides an illustration of declaring and using an interface.

Interfaces are often used to specify very generic functionality not related to a specific application. For example, an interface could indicate that collection-type objects know how to sort themselves (by specifying a sort() method) without getting into the details of what sorting algorithm is used, or what the concept of "sorting" means in the context of complex data types that may not have an obvious sequence. An interface with no methods specified can also be used as a true/false flag to indicate an object's suitability for some meta-operation, such as serialization or multi-thread safety. An interface is, to some extent, similar in purpose (but not structure) to the "mix-in" classes supported by other languages.

ABSTRACT CLASSES

Another major new OO feature in PHP is abstract classes. An abstract class is a class with only partial implementation of its behaviors, relying on subclasses to actually implement those functions. This situation often occurs in applications where a class library exposes a generalized superclass to applications, but relies on subclasses of that to actually do the detail work. For example, there are many ways to send a file from one computer to another on the Internet. An abstract class to define this capability might look like the one in FIGURE 2.

The value of an abstract class is that it allows the designer of a class library to create a framework for how a family of objects should behave, without necessarily having the details of their implementation. Generally, the abstract class at the top of an inheritance tree represents an abstract concept in the real world.

Since abstract classes cannot be instantiated, they are also very useful for creating utility "helper" classes that have *only* static methods and static data members. Such classes offer an object-oriented way to build a library of utility functions that contain no stateful data—in other words, to sneak a bit of non-OO code into an application that is otherwise purely objects, without breaking the OO paradigm.

Any class that has at least one abstract method must be declared an abstract class.

CHOOSING BETWEEN INTERFACE AND ABSTRACT CLASS

It can be confusing to choose between defining an interface and defining an abstract class, because the two concepts are very similar. In general, use an interface to indicate functionality or features that exist within a class but which are not its primary purpose. Use an abstract class to define primary features of a class inheritance tree without implementing them at the top level.

As an example, consider a sophisticated network connection management library that supports all kinds of protocols as well as managing connection daemons, pools of open sockets, worker threads, thread pooling, and other concepts often found in large-scale network applications. The designer might use abstract classes (not inheriting from one another) to define the fundamental behaviors of objects in the system, e.g., NetDaemon, NetThreadPool, NetThread, NetConnection, NetServer, NetClient, and so on. Now, suppose for logging purposes the designer decides that these objects need to be able to be assigned names, perhaps from a startup configuration file and/or a command line invocation parameter, or simply generated programmatically (e.g., "Thread001", "Thread002", and so on) from the class name. The designer creates an interface like the one in FIGURE 3 to indicate that an object can be named.

Notice that the interface does not specify how the name is stored, or even whether it is stored. Perhaps some classes implement a setName() function to allow applications to assign a name, whereas others simply return an internally-generated name for the getName() function.

In the network library example, there is also likely to be a need to have certain classes keep track of how many of their instances have been created, and perhaps assign default names (which can be overridden by the application if desired) at the time of instance creation. An interface such as InstanceCountingObject might be created, using static variables and static functions to maintain an internal one-per-class counter of the instances as they are created. Clearly, behaviors such as "this object can be named" or "this class counts its instances" are not related to the goal of the application, as they have nothing to do with network protocols, clients, or servers. Thus, these meta-behaviors are very good candidates to be declared with interfaces.

Another very simple way to decide between an interface and an abstract class is: "Does the thing you are declaring need to implement part of its own logic, or just define what logic will exist?" Since abstract classes can contain *some* concrete method implementations, and interfaces cannot, the need for actual logic in the declaration may force a designer's hand. Likewise, if the thing being declared needs to be applied to multiple classes which are not ancestrally related to one another, an interface may be the only sensible alternative.

OVERLOADING

PHP 5 provides a totally new feature called overloading, basically a set of predefined "hooks" into PHP's internal runtime logic. A detailed discussion of these hooks is beyond the scope of this article, but they are documented in the PHP online manual at http://www.php.net/manual/en/language.oop5.overloading.php. Overloading allows an object designer to intercept applications' attempts to access variables or methods that do not exist. Instead of generating an error message, the __get(), __set(), and __call() hooks allow the object to actually implement the functionality programmatically. This can be useful when an object must support a large number of methods or variables that are all cookie-cutter replicas of one another but with different names.

OTHER NEW FEATURES

PHP 5 adds some advanced OO features that may be familiar to developers in Java or other OO languages, but which are beyond the scope of this article to discuss in detail.

The new PHP 5 runtime can iterate through an object's public variables using the **foreach** keyword, as if they were an associative array. Also, a class can declare itself to implement Iterator or IteratorAggregate; in doing so, it is able to provide customized implementation behavior that will automatically be invoked when the **foreach** statement is executed.

PHP 5 also has introspection and reflection, which provides a way for applications to examine object classes *at runtime* to determine their characteristics (Does method so-and-so exist in this object? Is this object an implementer of a certain interface?) Java programmers have used reflection to create very smart classes that can adapt themselves to complex data types at runtime; now a similar capability is available in PHP. Reflection is also extremely useful when writing tools such as performance profilers and debuggers.

The **instanceof** keyword now provides a very easy way to interrogate an object at runtime to see if it is a member of a specified class or one of its subclasses, or implements a specific interface. This keyword is used in a logical expression, such as shown in FIGURE 4.

Polymorphism applies to the **instanceof** logic. If YourClass extends MyClass and $foo is an instance of YourClass, then "$foo instanceof MyClass" will also be true.

If, as is often the case, you want an error like this to be treated as fatal, then the new object type hint syntax can be used in the function header to provide an equivalent result. In FIGURE 4, the declaration "function tweedledum(MyClass $foo)" would have provided essentially equivalent behavior without the need for the **if** logic. The **if** logic and **instanceof** test are more flexible, however, and are needed in case you want to handle the error programmatically rather than allowing the runtime to terminate the program.

FINALLY...

It is only fitting that the last feature discussed in this article should be the **final** keyword. Like its Java counterpart, **final** can be applied to a class or method to indicate that subclasses may not override the declaration. A method that is **final** is inherited by subclasses but cannot be overridden. If an entire class is declared **final**, then that class cannot be subclassed at all.

CONCLUSION

Object oriented programming can, of course, be accomplished even in languages that have no specific support for OO. It's a matter of programmer mindset more than language syntax. On the other hand, having strong OO support in a language certainly eases the job of a programmer who wishes to model the problem using objects. The more work that can be done automatically by the compiler and the runtime environment, the less work is needed from the programmer, and the lower the chance for human-induced error due to tedium.

OO extensions almost always add overhead to the runtime, and if not used sensibly, can hurt performance. It's too early to say how much, if at all, PHP 5's new OO features will impact performance, but the history of PHP suggests that its developers will have worked long and hard to ensure that the new features do not degrade the high performance we have come to expect from the language.

FIGURE 2: A SIMPLE EXAMPLE OF AN ABSTRACT CLASS, WITH ONE ABSTRACT METHOD THAT MUST BE OVERRIDDEN, AND ONE CONCRETE METHOD THAT MAY BE OVERRIDDEN

```
abstract class FileMover {
    protected $src;
    protected $dst;

    public function __construct($sourcefile, $destfile) {
        $this->src = $sourcefile;
        $this->dst = $destfile;
    }

    // Subclasses should override this method to start the
    // file transfer using some underlying network protocol.
    // The method should return 0 for success or an integer
    // error code in case of failure.
    public abstract function startTransfer($max_wait_seconds);

    // Subclasses may override this method to abort a transfer
    // in progress, if the underlying protocol supports that.
    // The default implementation does nothing, and returns
    // FALSE to indicate that the transfer was not stopped.
    public function abortTransfer() {
        return FALSE;
    }
}
```

FIGURE 3: THIS INTERFACE IS APPLIED TO OBJECTS WHOSE INSTANCES CAN BE ASSIGNED A HUMAN-FRIENDLY NAME

```
interface NamedObject {

    // This function returns the instance's name as a string
    public function getName();
}
```

FIGURE 4: USING THE INSTANCEOF KEYWORD TO CHECK A FUNCTION'S ARGUMENT FOR CORRECT TYPE

```
// $foo is supposed to be an instance of MyClass, but we
// can't guarantee what the application might send us, so
// we check it at runtime.
function tweedledum($foo) {
    if ($foo instanceof MyClass) {
        // ... Do whatever we are supposed to do
    } else {
        die("Argument to tweedledum() must be a MyClass object.\n");
    }
}
```

WEBLIOGRAPHY

PHP Home Page http://www.php.net/

PHP Online Manual (English) http://www.php.net/manual/en/

PHP 5 Objects http://www.php.net/manual/en/language.oop5.php

NaSPA member Scott Courtney is a senior engineer with Sine Nomine Associates, an engineering consulting company. His career has included fifteen years in engineering and IT at a large manufacturing company. He also worked as a technical journalist and editor for an online publisher for one year. Scott is an active open source developer in both PHP and Java languages and maintains a number of production Web sites using open source tools.