The Next Generation

Onward and upward

Of all the programming languages to have emerged in recent years, PHP is one of the great success stories. It has fostered a loyal community of developers and, despite some well documented issues, still maintains its popularity. BY STEVEN GOODWIN

www.space-center-bremen.de

Does its latest incarnation, PHP 5, do justice to its heritage, or is it destined to become a skeleton in the closet? Steven Goodwin finds out.

You're History

The first version of PHP was written in 1994 by Rasmus Lerdorf and constituted a number of Perl scripts for building web pages. The term PHP/FI was coined in 1995, and was followed by PHP/FI 2.0 in 1997, with PHP/FI standing for *Personal Home Page / Forms Interpreter*. Even though some of the functionality that we know today was present in these early versions, beyond a small cult group of developers, it was largely unheard of. Curious readers and history students can find the original documentation lurking at [1].

PHP as we know it really began with version 3, released in June 1998. Unlike the upgrade status to PHP/FI 2.0, PHP 3 was a complete rewrite by Rasmus, along with Andi Gutmans and Zeev Suraski. From a technical standpoint, it was very much improved, and more capable of dealing with high volume web sites and e-commerce applications. It also featured the first signs of object orientation (see Box: Uh Oh!), and a change of acronym to the still-in-use, PHP: Hypertext Preprocessor.

Version 4 was essentially an extension of 3, using the same syntax and semantics. It featured output buffering, improved object orientation, and various new features (like references) and instructions, such as the *foreach* control structure. Behind the scenes too, a lot had changed, as the PHP code base had been modularized and optimized, with some code running over 200 times faster.

This was largely thanks to a new engine, known as *Zend*, which was a complete re-write of the original. The PHP community was also growing by leaps and bounds: a number of third party databases were now supported, the PEAR repository came into being, and the documentation had grown into a comprehensive, and very usable, online manual [2], currently available in 26 different languages.

PHP5 is claimed to be the latest and best version of PHP, utilizing another new code engine (Zend 2), an impressive new object model and a host of new fea-

THE AUTHOR

When builders go down the pub they talk about football. Presumably therefore, when footballers go down the pub they talk about builders! When Steven Goodwin goes down the pub he doesn't talk about football. Or builders. He talks about computers. Constantly...

Uh Oh!

Object Orientation is more a paradigm than it is any particular type of programming. When used correctly, its benefits include code re-use, abstraction of detail, data encapsulation and polymorphism. The implementation of an OO solution involves *objects*, a much over-used term that (in this instance) indicates that both the data, and the code that manipulates it, are bound together in one unit. Each *object* is often represented by a class. The data held within this class represents its current state, and the code that affects it are called *methods*, or *member functions*.

tures. It has gone through four beta versions with, at the time of writing, the current version standing at Release Candidate 2, made available on the 25th April 2004. It is claimed stable, but *"still not recommended for mission-critical use"*. True stability will come with time, although the experiments of this author have shown it perfectly usable for small to medium scale work.

Little Fluffy Clouds

The source code for PHP 5 can be downloaded from [3] as a five MByte tarred gzip file, although the only official binaries available are for Windows. Brave Debian users can grab the *.debs* from [4] by adding the line,

```
deb http://packages➚
.dotdeb.org ./
```

to their */etc/apt/sources.list*, although this will not be guaranteed to work. Assuming you're compiling from source the traditional trio is employed thus,

```
$ ./configure --with-mysql ➚
--with-apxs
$ make
# make install
```

The options for *./configure* may need to be adapted, depending on whether you have (or want) MySQL support. The primary dependency essential for PHP 5 is a recent version of *libxml2* (version 2.5.10 or above), but once that's available, a basic compile and install can be accomplished with a fairly minimal time outlay. Documentation is included in the package, which also details how to run PHP 4 and 5 alongside each other. If you're compiling PHP 5 as an Apache module, then the traditional rules for building DSO's (Dynamic Shared Object) apply.

A Grand Don't Come For Free

The first, most obvious, question with such a new system is a consideration of old code. Does it break anything? Is this a backwards step? Fortunately, the answer appears to be no. Most of the changes in this release have paid lip service to the backwards compatibility issue. The core language looks no different to that of PHP 4 and although there have been a few backwards *in*compatible changes, these have been in the name of stricter standards, and are detailed in [5].

If your code is polite and follows the described function behavior to the letter, most of these changes won't affect you, but be warned as *array_merge* now only accepts arrays. This may cause *E_WARNINGs* to appear without apparent reason. It is worthwhile to note that illegal string offsets are now considered errors (not warnings). Note also that,

```
$app.is_object()
```

returns an object and one needs to use the modern form,

```
is_object($app)
```

to get a boolean value. However, most practical code will not (or should not!) have these errors present. For the terminally paranoid there is a compatibility mode. This can be used to determine whether the bug is in your code, or the new PHP 5 interpreter. It can be controlled with the line,

```
zend.ze1_compatibility_mode➚
 = Off
```

in your *php.ini* file. Two sample *ini* files come with PHP 5, a development version and a more secure release version. Although neither sees the need to use the compatibility mode.

Developers working on the bleeding edge will also have realized that studlyCaps (first word in lower case, and the initial letter of every other word being capitalized) are used for the extensions like SQLite and SOAP.

There are a slew of new functions in PHP 5, all detailed in [6]. Most of them provide better support for the core language, although most have been available before in libraries, or as user-added comments in the standard documentation on the main PHP site [2]. These functions break down into five basic groups. See Table 1.

Table 1: New function groups

Area	Number	Notes
Arrays	6	Mostly to determine differences between arrays
InterBase	19	Various functions for Interbase users
iconv	7	Handles internationalization features, and MIME headers
Streams	8	Most supplying useful low level socket handling functionality
Misc	34	General purpose functions, including PHP syntax checker, string conversion and process control

Before we launch into specific details of some of these features, there are two development functions to highlight. The first is *var_dump* which has an improved look, and expands the information inside the class, as does *print_r*.

```
// Example of var_dump
object(MyName)#5 (2) {
  ["firstName"]=>
  string(6) "Steven"
  ["lastName"]=>
  string(7) "Goodwin"
}
```

The second is a new function called *debug_backtrace*, which returns an array showing the call stack. This has been partially back ported into PHP 4, but looks to be a stalwart for all PHP developers looking for quick, light weight, debugging assistant, without resorting to *xdebug*.

The Invisible Man

Among the list of core language features there's a small selection that has been devoted to code that doesn't exist! That is, if a non-existent class method is called, the special *__call* function is invoked.

```
class Reverse {
  function __call($function, ➚
$arguments)
  {
    return strrev($function);
  }
}

$r = new Reverse;
print $r->this_doesnt_exist();
```

Or, if you try to access a member variable that doesn't exist, one of two different functions will be called, depending on whether you tried to read from, or write to, the variable.

```
class Length
{
 function __get($variable_name)
 {
  return strlen($variable_name);
 }

 function __set($variable_name, $value)
 {
  print "I can't set the length of $variable_name to $value!";
 }
}

$len = new Length;
$len->thisname = 10;
print $len->thisname;
```

All three of these functions only work as class methods, so you can not have a global *__set* function to cope with every non-handled case.

On the surface, these might appear as quirky functions, but of little use. However, they do have a number of serious applications. The *__call* function, for example, could be used to create mock objects, while *__set* and *__get* can pass information to and from a database, while remaining completely transparent to the end user.

Another feature of this ilk is *__autoload*, which gets called whenever a non-existent class is instantiated. The classic example for this feature is to load class files on demand. For example,

```
function __autoload($classname)
{
require_once "$classname.inc";
}

$me = new UnknownName;
// Causes UnknownName.inc to be loaded before class creation
```

This allows the script to start and run very quickly, only loading additional class resources when it needs to, allowing larger class hierarchies to be employed without degraded the performance in simpler cases. High performance environments must be careful to consider the load time for all the appropriate class files at the start, compared to sporadic on-demand loading (which may include disk seek times) throughout the script's lifetime.

Be careful when using *__autoload* since file names are case-sensitive, but PHP class names are not. This could cause missing classes to try and load non-existent files from disk. Porting code from a Windows environment will often include at least one of this variety!

The Girl in the Other Room

Of all the new features in PHP 5, one category stands proud: object orientation. OO features have been part of PHP since version 3, although they have always been rather minimal in scope. PHP has never been an OO language, merely a language that happens to support some object oriented features – namely classes.

However, PHP's previous support for OO had very limited encapsulation and data hiding capabilities, making it unsuitable for large projects without requiring a great deal of care. PHP 5 adds such a wealth of OO functionality that in one fell swoop it sits close to OO-Perl and Ruby. As a paradigm, OO is well understood and (despite the bad press attributed it, thanks to the nuances – and nuisances – of C++) is quite suitable for large scale operations. Many toolkits, such as GTK, are based on OO principles, which makes it easier for PHP 5 developers to interact with them.

There are two main areas of object oriented study in PHP 5. The first are the changed features. These are cases where improvements have been made to the language that allow traditional OO design patterns to be used natively within PHP. These were possible previously, albeit with some minor code hacks. The second involves new features that haven't been technically possible before, because of the language design. Let us cover the improvements first.

The biggest improvement is the way references and object copying occur. Previously, the object model was to always copy objects as they were passed into, and out of, functions. This made it difficult to handle objects efficiently, and so references were added. Any PHP code with extensive object use would therefore be full of the & symbol, and omissions could lead to hard-to-spot bugs.

```
class MyName {
  var $name;
}

function ChangeName(&$newname)
{
  $newname->name = "Steev";
}

$me = new MyName;
ChangeName($me);
print $me->name;
```

Table 2: Member Access

Type	Accessible to own class	...to derived classes	..to the world
Public	Yes	Yes	Yes
Protected	Yes	Yes	No
Private	Yes	No	No

Listing 1: London pubs

```
// sample file in boxout: London pubs
$pub_list = simplexml_load_file('londonpubs.xml');

// iterate through each 'pub' element
foreach ($pub_list as $pub) {
  if ($pub->name == $rate_this_pub) { // check name element
    $pub->name['rating'] = $rating_for_pub; // write a rating attribute
    print "Rating changed for $pub->name, to $rating_for_pub";
  }
}

$new_xml_data = $pub_list->asXML();
file_put_contents("newlondonpubs.xml",  $new_xml_data);
```

This simple example, to print the name 'Steev', would fail to work if the & was omitted in line 5, as a brand new copy of the *MyName* class would have been created and assigned to $newname. The name 'Steev' would then have been assigned to this new copy, and not the object passed in from line 11.

With PHP 5, class objects are passed as a reference by default, not value, so the & symbol is no longer needed, making it less error prone. The reference syntax is still maintained for compatibility.

```
05 function ChangeName($newname)
```

The use of this new syntax is encouraged, and should lead to the older explicit references being deprecated in future versions.

The only problem with this feature is that the code appears to be run-time compatible with versions 3 and 4 – but isn't! In the rare cases when your older code relied of the copying of objects, you will now need to *clone* them instead. This is a new feature, made necessary because of the new object model. Calling the *__clone* function will make a bitwise copy of the existing object, as will the more traditional,

```
$new_object = clone $old_object;
```

Furthermore, any class that overloads the *__clone* function can dictate how it should be copied, for cases where reference counting is required, or where the standard copy is not good enough.

Guns Of Brixton

Probably the most visible addition to PHP's OO handling code is represented by three new keywords: public, private and protected. Under PHP 4, every member variable of a class could be read by anyone. From any code. Placed anywhere.

Such variables are termed *public*, in much the same way as permissions in the Linux filesystem. By convention, any variable prefixed with an underscore was considered *private* and should only be accessed by the class itself. However, this is only a convention, and before PHP 5 could not be guaranteed. Now it is possible to prefix a member variable (or function) with the word *private*, and this will be enforced.

```
class MyName {
  private $name;
}

$me = new MyName;
print $me->name;
```

Which produces the error,

```
Fatal error: Cannot access ↵
private property MyName::$name ↵
in somefile.php on line 6
```

Even derived classes will not be able to use *$this->name* to access the variable. Any attempt to do so will cause PHP to create another public variable that belongs to the derived class. This can cause subtle errors, but private members and methods are easy enough to learn, and can add great security to your code, especially when you're developing modules.

Protected is a further extension of private. Any member that is *protected* can be accessed by the class itself (just like *private*), but it can also be used by any derived class, such as *MyFullName*:

```
class MyFullName extends↵
 MyName { ...
```

See Table 2, on the previous page, for a quick breakdown of these security definitions.

In addition, there is a new keyword called *final* which can be added to a member function to prevent it from being derived or extended any further.

All member variables, and methods, are created as *public* by default. This ensures backwards compatibility, and only functions called *public*, *private* or *protected* are likely to have problems.

Cornflake Girl

Another important change has been with the constructors. Instead of creating a function with an identical name to the class,

```
class MyName {
    function Myname()
    {
    // This is the constructor,↵
  called whenever a new ↵
MyName is created!
    }
}
```

a constructor can now be created by using the new name *__construct*. At a first casual glance, this appears, on the surface, to only be a minor change, however it becomes a major time saver if you need to change the class hierarchy, since calling the parent class's constructor now becomes,

```
class MyFullName extends ↵
MyName {
  function __construct()
  {
    parent::__construct();
  }
}
```

In large formal systems, the class hierarchy should not be changed often (if at all) as this can break the design. However in smaller ad-hoc systems this can prevent bugs from creeping in when new classes are added in between a parent and its child.

There only appears to be one primary limitation with constructors and even that limitation is fairly inconsequential in practice. That is, you can not overload the constructor as you can in other Object Orientated languages. This limits the number of ways you can create the object:

London Pubs

A sample XML file used in the example might appear thus:

```
<pubs>
        <pub><name>All Bar One</name><address>26-30 London Bridge ↵
        Street, London, SE1 2SZ </address></pub>
        <pub><name>The Banana Store</name><address>1 Cathedral Street,↵
        London, SE1 9DE </address></pub>
        <pub><name>The Barrowboy and Banker</name><address>6-8, ↵
        Borough High Street, London, SE1 9QQ </address></pub>
</pubs>
```

```
//Not directly available in PHP5
$me = new MyName("Steev");
$full_me = new MyName("Steven", "Goodwin");
```

If you need this functionality, I suggest using default parameters in the constructor and patching the data accordingly.

```
function __construct($name1="", $name2="")
{
 if ($name2 == "") {
  $this->name = $name1; // no surname
 } else {
  $this->name = "$name1 $name2";
 }
}
```

It is also possible for constructors to be made private. This trick prevents other classes from creating specific objects, as is required when implementing singletons, as we'll see shortly.

Destructors are a welcome addition to PHP. They are called automatically whenever an object is destroyed, which traditionally occurs when the code has finished executing. Although the memory that a class uses is always reclaimed automatically, any external resources (such as database locks or file handles) may not. Such functions are declared using the name *__destruct*.

Backwards compatibility is still an issue in OO, and fortunately, there are precious few cases that could fail. However, any member function called *__construct* or *__destruct* will cause the code to behave in an undetermined manner (especially if the function name doesn't reflect its traditional purpose). As will any functions using the other new keywords.

Stuck On You

Another very welcome improvement is the ability to use static methods and member variables. A static member is one that is identical across every instance of the class. Previously, it was possible to write code that *looked* like static members were available, but the popular method of doing so (given below) requires extra memory.

```
class PHP4FakeStatic
{
  var $local;

  function PHP4FakeStatic()
  {
    static $single_var=0;

    $this->local =& $single_var;
  }
}
```

So, if you wanted to know how many *MyName* classes had been created, in PHP 5 you could write:

```
01 class MyName {
02   static $class_count = 0;
03
04   function MyName()
05   {
06     MyName::$class_count++;
07   }
08 }
09
10 $you = new MyName;
11 $me = new MyName;
12 $them = new MyName;
13 $everybody = new MyName;
14
15 print MyName::$class_count;
```

Static members do not have a specific instance of the class, so they *must* be set (and read) without *$this*, as seen in line 6. Similarly, member functions can be called directly using the *class name::function name()* syntax. However, be warned that because static members exist outside of any specific instance, it is not possible to access any non-static data from within the class. Any static function that tries to use *$this* will be greeted with the error,

```
Fatal error: Using $this when not in object context in somefile.php on line 6
```

This construct can also be used to create a singleton, as mentioned above. The singleton a design pattern whereby only one instance of a class can ever be created, and is useful for handling log files and other resources where it doesn't make sense to have two.

```
class Singleton
{
  static private $instance = NULL;

  private function __construct() { }
```

PHP 5 Features in Brief

Feature	Notes
Object references	Removes need for &
Constructors and destructors	Now have unified name, regardless of class name
Public/private/protected	For data hiding
Static members	One copy of a variable for all instances of call
Final keyword	Prevents further derivation of classes
Class constants	Use *const cvar = 3.1415* instead of define
Objects copied by __clone function	Needed now objects are passed by reference
Type hinting	Indicates the required type for a parameter. A nod to a more strongly-typed PHP
Instance of	It is possible to determine the type of a dynamic object with *$foo instanceof SomeClass*
Indirect referencing	Allows *$foo->a->b;* in some cases
Automagic class loading	Use *__autoload* to bring in appropriate files
Setters and Getters	Overload member variable access with *__get* and *__set*
Runtime method overload	*__call* will be invoked if the original function doesn't exist
Abstract classes and interfaces	For more complex OO code
Exceptions	Exceptions can be caught and thrown as per other languages
SimpleXML	Supports loading and saving of well formed XML
SQLite	An available database without running extra services. Not recommended for high loads
Filters	Flexible text processing, with built in filters for rot13
Streams	To load data resources from arbitrary locations, using various protocols (e.g. ftp, http). Improved socket handling too
Namespaces	These have been removed!
Extensions	Various extensions for SOAP, SimpleXML and MySQL.

```
  static function get()
  {
    if (self::$instance == NULL) {
      self::$instance = new Singleton();
    }
    return self::$instance;
  }

  function hello()
  {
    print "Hello, singletons!";
  }
}
```

Every member of the singleton class can only be called through the *get* function, as there is no other way to get a reference to the singleton object. This comes from the private constructor, which stops any class (other than itself) creating an object. For example,

```
print Singleton::get()->hello();
```

Let It Grow

PHP 5 features several new, and improved, extensions. MySQLite is certainly one of the most talked about, as is the closely related MySQLi (which is the MySQL *i*mproved version, supporting amongst other things, prepared statements and variable bindings). However, the inclusion of SimpleXML is probably my favorite. While not as feature-full as DOM, it does provide a very efficient way of handling simple XML, such as data and configuration files.

After reading the XML into a SimpleXML object you can modify the data within PHP (using the standard data types), and then save the same object out to disk by serializing it into an XML text stream with *asXML*. See Listing 1. At this point it appears as standard, compliant, XML. See box: London Pubs.

When working with your own XML files, there are few limitations and even fewer problems. It just works! However, in more complex applications the limitation of namespaces within XML can be problematic, as SimpleXML ignores them, acting is if they don't exist.

When SimpleXML is not enough and you need more powerful editing features, don't worry. There's no need to port your code over to DOM, since two conversion functions have been provided.

```
$sxe = simplexml_import_dom($dom);
$dom = dom_import_simplexml($sxe);
```

PHP 5 has better support for various XML technologies, such as XSLT and SOAP, as the library underneath SimpleXML (libxml2) is used throughout.

Martha's Harbour

In general, a lot of good work has found its way into PHP 5. The integration of SQLite will appeal to those wanting to experiment with databases without the hassle of configuring extra daemons, and the Simple XML module will make it so easy to migrate away from flat files that many people will do so. Both should raise the standard of available PHP applications. These are just two of the many good modules available with version 5.

Alas, space does not permit detailed explanation of them all, so a brief run down of all the new features can be found in the box: PHP 5 Features in Brief, and on the web at [7] and [8].

However, the biggest selling point of PHP 5 must be the new object model. Almost everything else has been available before as a third party module, or as library code.

The OO in 5 is very good, and such an improvement on previous versions that many old hands will have to unlearn their various tricks in order to write effective PHP 5 code. For them, this transition will be the hardest part of migration. Not the new syntax rules (as they are simple to learn), but the feeling of loss as once you start using the new syntax you'll never want (or be able) to go back to PHP 4. With this in mind, PHP 5 should be considered completely separate from PHP 4, and you should not try to write code compatible with both.

There may also be some pain during the migration process, as occasional problems with old code will arise. Most of these should be caught with sufficient code auditing. When trialling PHP 5 on my own system for example, there were very few niggles, and my home site (with around 100 pages) took under an hour to get working.

For those coming to the language afresh, especially those looking for good OO capabilities, PHP 5 hits the spot in all the right places and appears as an effortless OO language. It should encourage developers who have been weary about PHP to return with renewed vigor, and may even convert a few programmers from other languages.

Only time will tell… ■

Mr. Big

The penetration of PHP is quite incredible. For instance, according to *SecuritySpace.com* PHP is installed on over half of the Apache servers surveyed in March 2004, and on 33% of the 47,173,415 domains queried by *Netcraft* in the same period. Also, from a study of all the projects in the *SourceForge.net* repository, 11.2% (4,433 of them) are written in PHP, a feat only bettered by C/C++ and Java.

Problems

Despite all the newness present in PHP 5, a number of older problems still remain. There is still no stack protection, for example, and most code that attempts to allocate too much memory (forced over-allocation) will still crash. Fortunately, most of these cases can only happen through malicious intent. There is also a problem when using the (deprecated) *dl* function, whereby an extension loaded twice (one explicitly with *dl*, followed by one implicitly through *new*) can cause a fatal exit as the extension is deleted before the objects created by the library are.

INFO

[1] PHP/FI Version 2.0: *http://www.php.net/manual/phpfi2.php*

[2] PHP manual: *http://www.php.net/manual/en/*

[3] Latest stable version of PHP: *http://www.php.net/downloads.php#5*

[4] Stable Debian repository: *http://www.dotdeb.org/*

[5] Backward Incompatible Changes: *http://www.zend.com/manual/migration5.incompatible.php*

[6] New PHP 5 functions: *http://www.zend.com/manual/migration5.functions.php*

[7] Changes in PHP 5/Zend Engine 2.0: *http://www.php.net/zend-engine-2.php*

[8] Migrating from PHP 4 to PHP 5: *http://www.zend.com/manual/migration5.php*