**Magic Methods**

The "magic" methods are ones with special names, starting with two underscores, which denote methods which will be triggered in response to particular PHP events. That might sound slightly automagical but actually it's pretty straightforward.

### **\_\_construct**

The constructor is a magic method that gets called when the object is instantiated. It is usually the first thing in the class declaration but it does not need to be, it a method like any other and can be declared anywhere in the class. Constructors also inherit like any other method. So if we consider our previous inheritance example from the Introduction to OOP, we could add a constructor to the Animal class like this:

class Animal{

public function \_\_construct() {

$this->created = time();

$this->logfile\_handle = fopen('/tmp/log.txt', 'w');

}

}

Now we can create a class which inherits from the Animal class - a Penguin! Without adding anything into the Penguin class, we can declare it and have it inherit from Animal, like this:

class Penguin extends Animal {

}

$tux = new Penguin;

echo $tux->created;

If we define a \_\_construct method in the Penguin class, then Penguin objects will run that instead when they are instantiated. Since there isn't one, PHP looks to the parent class definition for information and uses that. So we can override, or not, in our new class - very handy.

### **\_\_destruct**

Did you spot the file handle that was also part of the constructor? We don't really want to leave things like that lying around when we finish using an object and so the \_\_destruct method does the opposite of the constructor. It gets run when the object is destroyed, either expressly by us or when we're not using it any more and PHP cleans it up for us. For the Animal, our \_\_destruct method might look something like this:

class Animal{

public function \_\_construct() {

$this->created = time();

$this->logfile\_handle = fopen('/tmp/log.txt', 'w');

}

public function \_\_destruct() {

fclose($this->logfile\_handle);

}

}

The destructor lets us close up any external resources that were being used by the object. In PHP since we have such short running scripts (and look out for greatly improved garbage collection in newer versions), often issues such as memory leaks aren't a problem. However it's good practice to clean up and will give you a more efficient application overall!

### **\_\_get**

This next magic method is a very neat little trick to use - it makes properties which actually don't exist appear as if they do. Let's take our little penguin:

class Penguin extends Animal {

public function \_\_construct($id) {

$this->getPenguinFromDb($id);

}

public function getPenguinFromDb($id) {

// elegant and robust database code goes here

}

}

Now if our penguin has the properties "name" and "age" after it is loaded, we'd be able to do:

$tux = new Penguin(3);

echo $tux->name . " is " . $tux->age . " years old\n";

However imagine something changed about the backend database or information provider, so instead of "name", the property was called "username". And imagine this is a complex application which refers to the "name" property in too many places for us to change. We can use the \_\_get method to pretend that the "name" property still exists:

class Penguin extends Animal {

public function \_\_construct($id) {

$this->getPenguinFromDb($id);

}

public function getPenguinFromDb($id) {

// elegant and robust database code goes here

}

public function \_\_get($field) {

if($field == 'name') {

return $this->username;

}

}

This technique isn't really a good way to write whole systems, because it makes code hard to debug, but it is a very valuable tool. It can also be used to only load properties on demand or show calculated fields as properties, and a hundred other applications that I haven't even thought of!

### **\_\_set**

So we updated all the calls to $this->name to return $this->username but what about when we want to set that value, perhaps we have an account screen where users can change their name? Help is at hand in the form of the \_\_setmethod, and easiest to illustrate with an example.

class Penguin extends Animal {

public function \_\_construct($id) {

$this->getPenguinFromDb($id);

}

public function getPenguinFromDb($id) {

// elegant and robust database code goes here

}

public function \_\_get($field) {

if($field == 'name') {

return $this->username;

}

}

public function \_\_set($field, $value) {

if($field == 'name') {

$this->username = $value;

}

}

}

In this way we can falsify properties of objects, for any one of a number of uses. As I said, not a way to build a whole system, but a very useful trick to know.

### **\_\_call**

There are actually two methods which are similar enough that they don't get their own title in this post! The first is the \_\_call method, which gets called, if defined, when an undefined method is called on this object. The second is \_\_callStatic which behaves in exactly the same way but responds to undefined static method calls instead (this was added in PHP 5.3). Probably the most common thing I use \_\_call for is polite error handling, and this is especially useful in library code where other people might need to be integrating with your methods. So for example if a script had a Penguin object called $penguin and it contained $penguin->speak() ... the speak() method isn't defined so under normal circumstances we'd see:

PHP Fatal error: Call to undefined method Penguin::speak() in ...

What we can do is add something to cope more nicely with this kind of failure than the PHP fatal error you see here, by declaring a method \_\_call. For example:

class Animal {

}

class Penguin extends Animal {

public function \_\_construct($id) {

$this->getPenguinFromDb($id);

}

public function getPenguinFromDb($id) {

// elegant and robust database code goes here

}

public function \_\_get($field) {

if($field == 'name') {

return $this->username;

}

}

public function \_\_set($field, $value) {

if($field == 'name') {

$this->username = $value;

}

}

public function \_\_call($method, $args) {

echo "unknown method " . $method;

return false;

}

}

This will catch the error and echo it. In a practical application it might be more appropriate to log a message, redirect a user, or throw an exception, depending on what you are working on - but the concept is the same. Any misdirected method calls can be handled here however you need to, you can detect the name of the method and respond differently accordingly - for example you could handle method renaming in a similar way to how we handled the property renaming above.

### **\_\_sleep**

The \_\_sleep() method is called when the object is serialised, and allows you to control what gets serialised. There are all sorts of applications for this, a good example is if an object contains some kind of pointer, for example a file handle or a reference to another object. When the object is serialised and then unserialised then these types of references are useless since the target may no longer be present or valid. Therefore it is better to unset these before you store them.

### **\_\_wakeup**

This is the opposite of the \_\_sleep() method and allows you to alter the behaviour of the unserialisation of the object. Used in tandem with \_\_sleep(), this can be used to reinstate handles and object references which were removed when the object was serialised. A good example application could be a database handle which gets unset when the item is serialised, and then reinstated by referring to the current configuration settings when the item is unserialised.

### **\_\_clone**

We looked at an example of using the clone keyword in the [second part](http://lornajane.net/posts/2012/a-little-more-oop-in-php) of my introduction to OOP in PHP, to make a copy of an object rather than have two variables pointing to the same actual data. By overriding this method in a class, we can affect what happens when the clone keyword is used on this object. While this isn't something we come across every day, a nice use case is to create a true singleton by adding a private access modifier to the method.

### **\_\_toString**

Definitely saving the best until last, the \_\_toString method is a very handy addition to our toolkit. This method can be declared to override the behaviour of an object which is output as a string, for example when it is echoed. For example if you wanted to just be able to echo an object in a template, you can use this method to control what that output would look like. Let's look at our Penguin again:

class Penguin {

public function \_\_construct($name) {

$this->species = 'Penguin';

$this->name = $name;

}

public function \_\_toString() {

return $this->name . " (" . $this->species . ")\n";

}

}

With this in place, we can literally output the object by calling echo on it, like this:

$tux = new Penguin('tux');

echo $tux;

I don't use this shortcut often but it's useful to know that it is there.