Chapter 3

Using PEAR and PEAR_Error

At the heart of programming with PEAR are two classes: PEAR and PEAR_Error. The two classes combine to offer a wide range of features such as constructors, destructors and error handling. In this chapter we will cover how to utilize these classes to not only build your own classes, but to enhance the way you check for and handle errors in your programs.

The PEAR Class

It is a well known fact that PHP's Object Oriented Programming (OOP) support lacks many of the features that a true OOP language has. The most glaring features missing are private/protected/public variables, constructors and destructors. Because PHP is not a strict typed language there is not a way we can address the problem of private/protected/public variables, but we can code an effective solution for constructors and destructors.

The PEAR's class primary duty is to handle the management of destructors. If your class follows the PEAR destructor standard and is based on the PEAR class it can have destructors that run when the execution of the script has ended.

How PEAR Destructors Work

PEAR makes use of two features in PHP to make destructors work. The first is that all classes must be created and assign by reference. The second is the register_shutdown_function() feature that allows a special PEAR function to run each destructor when your script's execution has ended. The PEAR constructor creates a global list of destructors and then the shutdown function references that list and runs the destructors that have been registered.

Creating a Class using PEAR

There are a number of things to remember when creating classes using the PEAR class.

1. Your class must either be extended directly from PEAR or from a class that is based on PEAR
2. Your class must have a function with the exact same name as your class as well as a function with the exact same name as your class proceeded by an underscore.
3. When you create an instance of your class you must do so by reference.

```
<?php
```

```
require_once('PEAR.php');

/**
* Object
*
* It's always a good idea to comment your code. An even better idea is to
* use PHPDocumentor to markup your comments and then organize them into
* documents.
*
* @author Joe Stump <joe@joestump.net>
* @package Book
* @link http://phpdocu.sourceforge.net
*/
class Object extends PEAR
{
  /**
  * Object
  *
  * This is your object's constructor. Anything in this class will be ran
  * when your object is created. This is a good place to connect to
  * databases, open log files, initialize member variables, etc.
  *
  * @author Joe Stump <joe@joestump.net>
  * @access public
  * @return void
  */
  function Object()
  {
    $this->PEAR();
  }

  /**
  * _Object
  *
  * Our destructor. Code in this function will be ran after your script
  * has finished executing. This is a good place to close database
  * connections, close files and clean up variables.
  *
  * @author Joe Stump <joe@joestump.net>
  * @access protected
  * @return void
  */
  function _Object()
  {
```

```
    $this->_PEAR();
  }
 }

?>
```

The above class is a simple example of creating a class that has both a constructor and a destructor. For the remainder of this chapter we will use the above class in our examples as we attempt to create the ultimate base object class, which we can use to build our application's classes from.

Destructors in Action

To get a better idea of how destructors work we are going to take our newly created Object class and add some debugging lines to it so we can verify that PEAR is in fact working.

```
<?php

 require_once('PEAR.php');

 /**
 * DEBUG
 *
 * @global bool DEBUG
 * @name DEBUG
 */
 define('DEBUG',true);

 /**
 * Object
 *
 * It's always a good idea to comment your code. An even better idea is to
 * use PHPDocumentor to markup your comments and then organize them into
 * documents.
 *
 * @author Joe Stump <joe@joestump.net>
 * @package Book
 * @link http://phpdocu.sourceforge.net
 */
 class Object extends PEAR
 {
  /**
   * Object
```

```
  *
  * This is your object's constructor. Anything in this class will be ran
  * when your object is created. This is a good place to connect to
  * databases, open log files, initialize member variables, etc.
  *
  * @author Joe Stump <joe@joestump.net>
  * @access public
  * @return void
  */
  function Object()
  {
    $this->PEAR();

    if(DEBUG)
    {
      echo 'Object constructor has ran <br />'."\n";
    }
  }

  /**
  * _Object
  *
  * Our destructor. Code in this function will be ran after your script
  * has finished executing. This is a good place to close database
  * connections, close files and clean up variables.
  *
  * @author Joe Stump <joe@joestump.net>
  * @access protected
  * @return void
  */
  function _Object()
  {
    $this->PEAR();

    if(DEBUG)
    {
      echo 'Object destructor has ran <br />'."\n";
    }
  }
}

// Create an instance of our class
$test = & new Object();

?>
```

We've added some debugging code to our class now, which will allow us to turn debugging messages on and off as we code. For production use you may want to strip these extra tests out of your classes for increased performance.

When you run the above code you should see two lines showing you that both the constructor and destructor have successfully ran. The last line of the script, where we actually create an instance of Object, holds two keys to making your PEAR enabled Objects work.

> NOTE: The PEAR group warns that some SAPI implementations will not display output during the request for shutdown. If you do not see output from the destructor you should try logging the output to syslog or a file to make sure your destructor is working correctly.

The first is the fact that we are assigning by reference. You will notice that next to the equal sign is an ampersand, which indicates we are assigning by reference. By default PHP creates an instance of the class and then copies it to the variable $test. This default behavior breaks the way PEAR handles destructors because PEAR tracks your class by reference and then uses that reference to run each instance's destructor. Without passing by reference PEAR's reference to our instance of Object will not mach the location of our actual instance of Object. Passing by reference also allows PEAR to run a destructor for each instance of our Object class we create. In other words, no matter how many instances of Object we create it will run each instance's destructor.

> NOTE: When PHP version 5.0 is released it will include enhanced OOP support, which will include support for constructors and destructors. PHP version 5.0 will also pass classes by reference by default, which solves the problem with references I talked about earlier.

The second thing to notice is that we add parenthesis to our class's name, which tells PHP to run Object's constructor when creating an instance of Object. It is at this point that Object::Object() is ran.

The PEAR solution to destructors may not answer all of the lacking OOP problems in PHP, but it is one of the most elegant and portable solutions that is currently available. The reality is that destructors are only a small portion of what the core PEAR classes offer.

Loading PHP Extensions

PHP offers the ability to dynamically load extensions during runtime through the function dl(). The only problem is that extensions are compiled as different binaries on different systems. For instance, an extension compiled for Windows has the extension .dll, while one for Linux has the extension .so.

This discrepancy in file extensions can cause problems for developers wishing to create cross platform code that needs to load dynamic extensions. Thankfully, the PEAR

team has created a function that handles this problem. The function is called PEAR::loadExtension() and is called statically. The function handles suffixes and checks your PHP settings to make sure that dynamic extensions can be loaded during runtime. If dynamic libraries are disabled, safe_mode is turned on or PHP fails to load the desired extension the function will return false.

```
<?php

 if(PEAR::loadExtension('mcal'))
 {
  die("mcal extension could not be loaded!");
 }

?>
```

Persistent Properties

Many times you will create static classes that are never initialized and therefore do not have member variables that methods can access. PEAR offers the function PEAR::getStaticProperty() that handles the management of static properties and ensures they persist within our static class.

```
<?php

 require_once('PEAR.php');


 class staticClass
 {
  function set($set)
  {
   // MUST ASSIGN BY REFERENCE OR VARIABLES WILL NOT PERSIST!
   $foo = & PEAR::getStaticProperty('staticClass','foo');
   $foo = $set;
  }

  function view()
  {
   print PEAR::getStaticProperty('staticClass','foo');
  }
 }

 staticClass::set('Joe Stump');
```

```
staticClass::view();
?>
```

Despite both functions being called statically and no instance of staticClass existing we were able to persist the variable 'foo' across two methods using PEAR's getStaticProperty().

The function does this is by creating a global reference keyed by your class's name. The function then returns that global reference to let you handle the variable however you wish.

Registering Shutdown Functions

As stated before, PEAR handles destructors with a shutdown function, however it should also be noted that PEAR also offers the ability to handle all of your shutdown functions. By registering a shutdown function with PEAR it will make sure that function is appropriately called when your script has finished executing.

You may be wondering why you should use PEAR to manage shutdown functions instead of using PHP's register_shutdown_function(). There are two main reasons why using PEAR is better: PEAR allows you to use class methods as shutdown functions and PEAR allows you to pass parameters to your shutdown functions. PEAR does this by using PHP's call_user_func_array() in its own shutdown function. The simple answer is that PEAR offers more flexibility with respect to shutdown functions.

```
<?php

 require_once('PEAR.php');

 class myclass
 {
  function myclass()
  {
    PEAR::registerShutdownFunc(array('myclass','shutdown'),
                               array('foo','bar'));
  }

  // shutdown function is ran statically!
  function shutdown($param1,$param2)
  {
    echo '<ul>
          <li> '.$param1.'</li>
          <li> '.$param2.'</li>
         </ul>'."\n";
  }
```

```
  }

  $class = & new myclass();

?>
```

With PEAR's registerShutdownFunc() you can also register an instance of a class and one of the instance's methods or a regular function as shutdown functions. When passing an instance of a class you do it the same as a static class with one exception; you pass the first value of the first argument as a reference to your instance. When adding regular functions you simple pass the name of the function as a string in the first argument. Below is an example of each of the types of shutdown functions you can register with PEAR.

```
<?php

  require_once('PEAR.php');

  function my_shutdown($param1,$param2)
  {
    echo '<ul>';
    echo '<li> Regular function!</li>';
    echo '<li> '.$param1.'</li>'."\n";
    echo '<li> '.$param2.'</li>'."\n";
    echo '</ul>'."\n";
  }

  PEAR::registerShutdownFunc('my_shutdown',
                             array('foo','bar'));

  class myclass
  {
    function myclass()
    {
      PEAR::registerShutdownFunc(array('myclass','shutdown'),
                                 array('foo','bar'));
    }

    function shutdown($param1,$param2)
    {
      echo '<ul>';
      if(is_object($this))
      {
        // If $this is an object the shutdown function was registered
        // with an instance of this class and not simply statically
```

```
      echo '<li> Ran by reference!</li>';
    }
    else
    {
      echo '<li> Ran statically!</li>';
    }

    echo '<li> '.$param1.'</li>'."\n";
    echo '<li> '.$param2.'</li>'."\n";
    echo '</ul>'."\n";
  }
}

$class = & new myclass();

PEAR::registerShutdownFunc(array(&$class,'shutdown'),
                           array('foo','bar'));

?>
```

It is important to remember that PEAR runs the each shutdown function after PEAR destructors have been ran and that the functions registered with PEAR to be called during shutdown are ran in the order in which they were registered. In the above example the function my_shutdown() would be ran, then myclass::shutdown() would be ran statically followed by myclass::shutdown() being ran by reference.

The PEAR_Error Class

PEAR offers feature rich error handling. The two base classes PEAR and PEAR_Error work together to offer error messaging, error numbers, callback functions and other error handling.

Calling PEAR_Error

Before you can actively use PEAR_Error in your applications it is important to understand how it works and how we will use it to handle different types of errors we may come across.

The PEAR_Error's constructor takes five arguments that define how the error should be handled. Below is a reproduction of the PEAR_Error's constructor with a brief description of each argument.

```
/**
 * PEAR_Error constructor
```

```
     *
     * @param string $message  message
     *
     * @param int $code     (optional) error code
     *
     * @param int $mode     (optional) error mode, one of: PEAR_ERROR_RETURN,
     * PEAR_ERROR_PRINT, PEAR_ERROR_DIE, PEAR_ERROR_TRIGGER,
     * PEAR_ERROR_CALLBACK or PEAR_ERROR_EXCEPTION
     *
     * @param mixed $options   (optional) error level, _OR_ in the case of
     * PEAR_ERROR_CALLBACK, the callback function or object/method
     * tuple.
     *
     * @param string $userinfo (optional) additional user/debug info
     *
     * @access public
     *
     */
    function PEAR_Error($message = 'unknown error', $code = null,
                $mode = null, $options = null, $userinfo = null)
```

The only required argument is a message, which can later be retrieved and used for logging, output, etc. The second argument is a custom error code, which can be referenced by your application for its own purposes. The PEAR_Error class does not use this code for anything other than to assign it to the instance of the error for use by your application. The third argument is the error mode, which defines how PEAR_Error will handle the error. Currently, PEAR_Error supports a number of options, which we will discuss in more detail shortly. The fourth argument is options for your error. Depending on your error mode this could be anything from the error message format to a callback function. The final argument is called user info and is basically a variable that you can put whatever you want into it for debugging purposes. For instance, this could be used to pass information to a callback function.

PEAR_Error handles different types of errors with error modes. It currently supports seven different error modes, which should adequately handle any error type you may encounter. Before we move on it is extremely important to understand how each of the error modes behave and what types of errors should use each error mode.

PEAR_ERROR_RETURN

The error mode PEAR_ERROR_RETURN is the default error mode for PEAR_Error. This error mode is designated for expected errors where little or nothing needs to be done. If the error does not require outputting an error message, halting the application or the use of a callback function then this mode would be the appropriate choice.

```
<?php

 require_once(‘PEAR.php’);

 $error = & new PEAR_Error('An error has occurred!');

?>
```

In the above example an instance of PEAR_Error would simply be assigned to the variable $error and the script would continue to process.

PEAR_ERROR_PRINT

PEAR_ERROR_PRINT is used for errors in which you wish to print the error to the screen. If an error occurs and is assigned PEAR_ERROR_PRINT as its error mode it will simple print an error and continue processing your application. If you need to halt your application you will want to use the PEAR_ERROR_DIE instead.

```
<?php

 require_once('PEAR.php');

 // MY_ERROR_CODE can be anything. If your application does not use error codes, but
 // you wish to use this feature simply pass null
 $error = & new PEAR_Error('An error has occurred!',
                 MY_ERROR_CODE,
                 PEAR_ERROR_PRINT,
                 '<b>ERR:</b> %s at line '.__LINE__.' in '.__FILE__);

?>
```

The above example would not only assign an instance of PEAR_Error to $error, but it would also print out an error. You’ll notice the fourth argument, designated for options, is set to a string. That string is the format of the error and is used by a call to PHP’s printf() function with the string being substituted for %s. When the above script is ran it would produce the following output.

ERR: An error has occurred! at line 8 in /home/jstump/public_html/error.php

PEAR_ERROR_TRIGGER

PEAR_ERROR_TRIGGER is simply an interface to PHP's trigger_error() function. Before using this feature it would be wise to read the documentation on the function trigger_error(). PEAR passes E_USER_NOTICE as the default error level, but this can be changed.

```
<?php

 require_once('PEAR.php');

 // For the fourth argument you could pass either E_USER_WARNING,
 // E_USER_ERROR or E_USER_NOTICE. The default is E_USER_NOTICE.
 $error = & new PEAR_Error('An error has occurred!',
                 MY_ERROR_CODE,
                 PEAR_ERROR_TRIGGER,
                 E_USER_WARNING);

?>
```

This example illustrates how to change the error level passed to trigger_error() by PEAR_Error. It should be noted that changing the error level passed to trigger_error() does not drastically change it's behavior. It merely changes the wording of the error. Below is an example of the output of the above script.

Warning: An error has occurred! in **/usr/share/php/PEAR.php** on line **750**

PEAR_ERROR_DIE

The PEAR_ERROR_DIE error mode works exactly like the PEAR_ERROR_PRINT error mode with the exception that it halts the processing of your application with a call to PHP's die() function. Errors which are catastrophic and require the instance shutdown of your application should use this error mode. Such an error could be a missing table or database, which would render you application unusable.

```
<?php
 require_once('PEAR.php');
 $error = & new PEAR_Error('An error has occurred!',
                 MY_ERROR_CODE,
                 PEAR_ERROR_DIE,
                 '<b>ERR:</b> %s at line '.__LINE__.' in '.__FILE__);

?>
```

In the example above your application would be halted and the error would be printed to the screen.

PEAR_ERROR_CALLBACK

PEAR_ERROR_CALLBACK is a great tool for handling errors that might be able to be fixed during runtime. Maybe your script requires default configuration values to be set before it can process. With PEAR_ERROR_CALLBACK you could set those variables with a callback function and continue processing your application. You could also use the callback function to log SQL statements that have failed or other tasks that the webserver was not able to process.

The main difference between PEAR_ERROR_CALLBACK and the other error modes is that the fourth argument takes the name of the callback function. Alternatively you could pass it an array where the first value is an instance of a class and the second value is the method to use as a callback function. Whether you use a regular function or a class method as your callback PEAR_Error will pass a single argument to your function or method; an instance of the PEAR_Error calling the callback function.

```
<?php

 require_once('PEAR.php');

 function my_error($error)
 {
  echo $error->getMessage();
 }

 $error = & new PEAR_Error('An error has occurred!',
                  MY_ERROR_CODE,
                  PEAR_ERROR_CALLBACK,
                  'my_error');

?>
```

The above example has PEAR_Error call the function ‘my_error’, which is passed a copy of our instance of $error. The function then just prints out the error message. There is a small problem with the above example and that is that we only get an instance of $error with no other information. If, for example, it was a failed SQL statement we may wish to log that statement for later scrutiny. By using a class method when we trigger an error we have all of the class’s member variables at our fingertips, which we can then use in our callback method.

```
<?php
```

```
require_once('PEAR.php');

class App extends PEAR
{
  var $name;

  function App()
  {
    $this->PEAR();
  }

  function checkName()
  {
    if($this->name != 'Joe Stump')
    {
      return new PEAR_Error('Invalid name: '.$this->name,
                            FILE_INVALID,
                            PEAR_ERROR_CALLBACK,
                            array(&$this,'badName'));
    }
  }

  function badName($error)
  {
    echo 'I do not know anyone named '.$this->name.'!';
  }

  function _App()
  {
    $this->_PEAR();
  }
}

$foo = & new App();
$foo->name = 'Paul Barton';
$foo->checkName();

?>
```

By using a class method as a callback and then calling PEAR_Error for errors called from within that class we can easily reference member variables in our class. The above example could easily be modified to use PEAR's Log package to log information, send alert emails to system administrators or attempt to run a query against a backup database server.

PEAR_ERROR_EXCEPTION

PEAR_ERROR_EXEPTION is something that is currently only available for use with PHP5. It is a simple interface to throw exceptions, which will be supported when PHP5 is released. If you attempt to use this error mode with a version of PHP older than version 5.0 you will get parse errors.

Replacing Boolean

The PEAR_Error class offers a robust option for replacing the tried and true boolean method of testing. The old way involved returning true or false from our function depending on the result of the function.

```
<?

 require_once('PEAR.php');

 class File extends PEAR
 {
  function File()
  {
   $this->PEAR();
  }

  function isValid($file)
  {
   if(file_exists($file))
   {
    if(is_writable($file))
    {
     return true;
    }
   }

   return false;
  }

  function _File()
  {
   $this->_PEAR();
  }
 }

 $file = & new File();
 if($file->isValid('/etc/motd'))
```

```
{
  echo ‘GOOD’;
}
else
  echo ‘BAD’;
}

?>
```

With the old way outlined above you have no way of knowing which test failed. Without knowing which test failed it becomes difficult to adequately handle the problem. For instance, if the file does not exist we may wish to create a blank file for future manipulation. If the file exists, but is not writable we may have more serious problems that require a closer look.

```
<?php

 require_once('PEAR.php');

 // Define our own error codes to pass to PEAR_Error
 define('FILE_INVALID',       1);
 define('FILE_NO_PERMISSIONS', 2);

 class File extends PEAR
 {
  var $file;

  function File()
  {
   $this->PEAR();
  }

  function isValid($file)
  {
   $this->file = $file;
   if(file_exists($this->file))
   {
    if(is_writable($this->file))
    {
     return true;
    }
    else
    {
     return new PEAR_Error('File is not writable: '.$this->file,
```

```
                    FILE_NO_PERMISSIONS,
                    PEAR_ERROR_DIE);
    }
   }
   else
   {
    return new PEAR_Error('File does not exist: '.$this->file,
                  FILE_INVALID,
                  PEAR_ERROR_CALLBACK,
                  array(&$this,'touch'));
   }
  }

  function touch()
  {
   if(!@touch($this->file))
   {
    $this = new PEAR_Error('Permission denied: '.$this->file,
                  FILE_NO_PERMISSIONS,
                  PEAR_ERROR_RETURN);
   }
  }

  function _File()
  {
   $this->_PEAR();
  }
 }

?>
```

The best thing about the error handling that PEAR provides is that we not only can check via simple boolean, but we get an error message and error code. Furthermore we can define an error mode telling PEAR how to handle the error.

Using PEAR_Error

Now that we are comfortable with creating instances of PEAR_Error it is important to learn how to use the instances of PEAR_Error. The PEAR_Error class offers a number of methods that let us interface with the error.

Using the example of the File class above I will go over how to use the PEAR_Error class. Before we get started it is important to note that some of the error handling is out of our hands. Calls to PEAR_Error with PEAR_ERROR_CALLBACK and PEAR_ERROR_DIE are out of our hands and calls to PEAR_ERROR_PRINT and

PEAR_ERROR_TRIGGER print errors to the screen, which we have no control over. In most instances packages use PEAR_ERROR_RETURN and give us an error class to use at our discretion.

```
<?php

 $file = & new File();

 $open = '/path/to/file/that/does/not/exist';
 $result = $file->isValid($open);
 if(File::isError($result))
 {
  echo $result->getMessage();
 }

?>
```

Before we move too far along we should talk about the error handling functions that our File class inherits from the base PEAR class. Probably the most important one is the static method PEAR::isError(), which is a Boolean function that returns true if the class passed to it is an instance of PEAR_Error or a class which is a subclass of PEAR_Error.

Once it is determined that we have an instance of the PEAR_Error class we can use a number of functions to glean error information from the class. Probably the most useful of these is the PEAR_Error::getMessage() function, which returns a textual message of the error. In the above example, since the file we passed does not exist, the message "File does not exist" would be printed on the screen. Since that specific error also triggers a callback function we would see additional errors if we could not touch the file.

Our callback function File::touch() also does error checking. If we cannot touch the file that means that the file does not exist and, additionally, we could not create it either. This error is handled a little bit differently. By using the reserved variables $this we assign an instance of PEAR_Error to $this, effectively turning our instance of $file into an instance of PEAR_Error, which we can then use to further debug any problems.

```
<?php

 $file = & new File();

 $open = '/path/to/file/that/does/not/exist';
 $result = $file->isValid($open);
 if(File::isError($result))
 {
  if(!File::isError($file))
  {
```

```
    // We had an error, but our callback fixed it
   }
   else
   {
    die($file->getMessage());
   }
  }

?>
```