Zend Engine

Version 2.0

Feature Overview and Design

Overview 5
Revamped object model using object handles 6
Background 6
Need 6
Overview 6
Functionality 7
Compatibility Notes 9
Dependencies 10
New Object Model Related Changes 11
Improved object returning mechanism in functions 11
Background 11
Need 11
Overview 11
Functionality 12
Compatibility notes 13
Dependencies 13
Improved Object Dereferencing Support 14
Overview 14
Background 14
Need 14
Overview 14
Functionality 15
Compatibility notes 15
Dependencies of feature 15
Object Cloning 16
Background 16
Need 16
Overview 16
Functionality 16
Compatibility notes 16
Dependencies of feature 17
Destructors 18
Background 18
Need 18
Overview 18
Functionality 18
Compatibility notes 19
Dependencies of feature 19
delete statement 20
Background 20
Need 20
Overview 20

Functionality 20
Compatibility notes 20
Dependencies of feature 21
Unified Constructors 22
Background 22
Need 22
Overview 22
Functionality 22
Compatibility notes 24
Dependencies of feature 24
Additional Features 25
Multiple Inheritance 25
Background 25
Need 25
Overview 25
Functionality 25
Compatibility notes 25
Dependencies of feature 26
Private Member Variables 27
Background 27
Need 27
Overview 27
Functionality 27
Compatibility notes 27
Dependencies of feature 28
Static Class Member Variables 29
Background 29
Need 29
Overview 29
Functionality 29
Compatibility notes 30
Dependencies of feature 30
Exception handling (*try*/*throw*/*catch*) 31
Background 31
Need 31
Overview 31
Functionality 31
Compatibility notes 32
Dependencies of feature 33
Revamped OO Syntax Overloading 34
Background 34
Need 34
Overview 35
Functionality 35
Compatibility Notes 35
Dependencies 35

String offset syntax 36
Background 36
Need 36
Overview 36
Functionality 37
Compatibility notes 37
Dependencies of feature 37

Overview

Version 1.0 of the Zend Engine is the heart and brain of PHP 4.0. It provides the infrastructure and services to the function modules, and implements the language syntax. The Zend Engine 1.0 is actually the second revision of the PHP scripting engine; It's still at large based on the same parsing rules as the PHP 3.0 engine (which was basically 'Zend Engine 0.5'). While this allowed for a very easy migration path from PHP 3.0 to 4.0, it also limited the scope of language-level improvements, to the same 'state of mind' as PHP 3.0 was in.

Thanks to the unprecedented success of PHP 3.0 and 4.0, a lot of feedback was received from developers, regarding the kinds of language features that they were either missing, or wanted to see improved. We feel that the time is right to start working towards a revision of the Zend Engine that would incorporate new features, improve existing ones, and provide solutions to some of the most difficult problems that PHP developers experience today. The purpose of this document is to provide the motivations for and in-depth description of each new feature, and is the base for the Zend Engine 2.0.

Note, that the scope of the Zend Engine 2.0 is such that it can be implemented in a reasonable time frame. Future versions of the Zend Engine may and probably will include additional features.

This document is *not* yet final, and may change before the final version of the Zend Engine 2.0. Discussions about the Zend Engine 2.0 will take place on the Engine 2.0 mailing list (mail engine2-subscribe@lists.zend.com to subscribe).

Revamped object model using object handles

Background

In the Zend Engine 1.0 (and its predecessor the PHP 3 scripting engine) the object model's design is that instantiated objects are language values. This means that when programmers are performing operations, such variable assignment and passing parameters to functions, objects are handled very similarly to the way other primitive types are handled such as integers and strings. Semantically this means that the whole object is being copied. The approach Java takes is different where one refers to objects by handle and not by value (one can think of a handle as an objects' ID).

Need

Unfortunately, the approach taken up to now has severely limited the Zend Engine's object oriented model, both feature and simplicity wise. One of the main problems with the former approach is that object instantiation and duplication is very hard to control, a problem which can not only lead to inefficient development but also often to strange run-time behavior. Changing the object model to a handle oriented model will allow the addressing of many needs such as destructors, de-referencing method return values, tight control of object duplication and more.

Overview

The proposed object model is very much influenced by the Java model. In general, when you create a new object you will be getting a handle to the object instead of the object itself. When this handle is sent to functions, assigned and copied it is only the handle which is copied/sent/assigned. The object itself is never copied nor duplicated. This results in all handles of this object to always point at the same object making it a very consistent solution and saving unnecessary duplication and confusing behavior.

Functionality

After this change the basic use of objects will be almost identical to previous versions of the scripting engine. However, you won't bump into awkward and confusing copying & destructing of objects.

In order to create and use a new object instance you will do the following:

```
$object = new MyClass();
$object->method();
```

The previous code will assign *$object* the handle of a new instance of the class *MyClass* and call one of its methods.

Consider the following code:

```
1    class MyClass
2    {
3         function setMember($value)
4         {
5              $this->member = $value;
6         }
7
8         function getMember()
9         {
10             return $this->member;
11        }
12   }
13
14   function foo($obj)
15   {
16        $obj->setMember(“foo”);
17   }
18
19   $object = new MyClass();
20   $object->setMember(“bar”);
21   foo($object);
22   print $object->getMember();
```

Without the new Java-like handles, at line 20 the objects’ data member *member* is set to the string value of *“bar”*. Because of the internal representation of objects in the Zend Engine 1.0, the object is marked as a reference, and when it is sent by value to the function *foo*, it is duplicated (!). Therefore, the call to *foo()* on line 21 will result in the *$obj->setMember(“foo”)* call being called on a duplicate of *$object*. Line 22 will then result in *“bar”* being printed.

This is how the scripting engine has worked until today. Most developers are probably unaware of the fact that they aren't always talking to the same object but often duplicates; others may have realized this can usually be solved by always passing objects by reference (unless a replica is actually desired, which is uncommon).

The new object model will allow for a much more intuitive implementation of the code. On line 21, the object's handle (ID) is passed to *foo()* by value. Inside *foo()*, the object is fetched according to this handle and, therefore, the *setMember()* method is called on the originally instantiated object and not a copy. Line 22 will therefore result in "*foo*" being printed. This approach gives developers tighter control of when objects are created and duplicated. An additional not-as-important benefit is that the object handle will be passed to *foo()* by value, which most probably will also save unnecessary duplication of the value containing the ID itself and thus additionally improving run-time performance.

This was just a simple description of why the new object model solves awkward behavior and makes object handling much easier, intuitive and efficient. The importance of this change goes far beyond what is mentioned in this section as you will see in further sections which describe new features with a majority of them being based on this change.

Compatibility Notes

Many PHP programmers aren't even aware of the copying quirks of the current object model and, therefore, there is a relatively good chance that the amount of PHP applications that will work out of the box or after a very small amount of modifications would be high.

To simplify migration, version 2.0 will support an optional 'auto-clone' feature, which will perform a cloning of the object whenever it would have been copied in version 1.0. Optionally, it will also be possible to request that the engine will emit an E_NOTICE message whenever such an automatic clone occurs, in order to allow developers to gradually migrate to the version 2.0-style behavior (without automatic clones).

Dependencies

The new object model is not dependent on other features. Many of the other Zend Engine 2.0 features, such as the $foo->bar()->barbara() syntax, destructors and others completely rely on this new object model.

New Object Model Related Changes

Improved object returning mechanism in functions

Background

Because of the underlying implementation, returning objects from functions in the Zend Engine 1.0 is very cumbersome. It is necessary to use special notation, *$foo = &bar()*, if one wishes to have bar() return an object by reference. Additionally, it's also not possible to return an object by reference, by assigning it to a passed-by-reference argument, due to the copying semantics that characterizes the existing object model.

Need

Object oriented design-patterns, such as Factory (creating objects by a centralized function) and others, are becoming more and more popular. Implementing these patterns requires a clean and consistent OO API, and the ability to easily interlink objects to one another. Easy, straightforward syntax for returning object references from functions is crucial to the implementation of such design patterns.

Overview

The simplified and improved behavior is a side effect of the new handle-like object model. It will be possible to pass-on objects after creating them without any special considerations by using the *return* statement or by assigning the new object to a function parameter that was passed by reference.

Functionality

Returning objects will be done in exactly the same way as any other primitive types (such as integers or strings) are returned. The returned object will always refer to the specific object the user has instantiated, without any implicit copies made behind the scenes.

Consider the following code:

```
1    function FactoryMethod($class_type)
2    {
3         switch ($class_type) {
4              case “foo”:
5                   $obj = new MyFoo();
6                   break;
7              case “bar”:
8                   $obj = new MyBar();
9                   break;
10        }
11
12        return $obj;
13   }
14
15   $object = FactoryMethod(“foo”);
```

With the new object model this code will return the object itself; It will create an instance of class *MyFoo* and will return that exact instance to the executing Zend Engine without any copying of objects. This is different from the behavior under the Zend Engine 1.0, which would cause a replica of the object to be returned, instead of the object itself.

A more confusing and important aspect of this is if *MyFoo* had in some way created an additional reference to itself. With the new model the object will still be returned correctly. With the previous object model, when returning *$obj* it might have only been partially duplicated and any references to it ended up not referencing the returned object,

but some ‘ghost’ object instead. This confusing behavior was the cause of many very-difficult-to-debug bugs.

The current solution for this today is defining the function as **`function &FactoryMethod(class_$type)`** and changing line 15 to **`$object =& FactoryMethod(“foo”);`**. However, this is a cumbersome approach, and the syntax is very prone to errors.

Taking the same example, if the returned object weren’t returned via the return value but via a function argument you would bump into very similar problems. Consider the following function prototype for the previous function:

```
function FactoryMethod($object_type, &$resulting_object)
```

When trying to assign the created object to *$resulting_object* there might be strange behavior in the same cases as mentioned above. With the new model you can happily assign the object handle to *$resulting_object* and the instantiated object will be able to be referenced on the outside.

Compatibility notes

This aspect of the object model change will probably not have any compatibility issues unless the developer was relying on some weird object copying that was happening during execution. However, this change “might” make the *function &func()* and the =& syntax not very useful and in the least their removal should be considered. If they are removed it won’t be a problem to create a small converter that will find them and warn about them (and possible replace the old syntax).

Dependencies

This feature is dependent upon the new object model of the Zend Engine 2.0.

Improved Object Dereferencing Support

Overview

Support the de-referencing of objects returned from methods, e.g. *$object->method()->method()* (this change will possibly also include the ability to de-reference objects returned from functions, i.e. *func()->method()->method()*). (This feature would also include something more complicated like *$object->method()->member->method()* where a returned object's member is an object itself and then one of its methods is called).

Background

Due to the previous object model it is not possible to de-reference returned objects. This support was not implemented because it was not technically feasible.

Need

Many PHP developers have asked for the possibility to de-reference returned objects. Not only will this often lead to better-looking code but it can also prevent certain programming errors.
With the old object model, the equivalent of *$object->method()->method()->method()* would be:

```
$tmp =& $object->method();
$tmp =& $tmp->method();
$tmp->method();
```

Also, when interfacing with Java or COM objects, this is the natural syntax to use.

Overview

Thanks to the new object model, as well as the improved parser, it will be possible to call methods directly on a object handle that is returned from a function. Due to the nature of the new model, the method will be called on the very same object, and not on a different replica.

Functionality

With this feature one will be able to nicely de-reference returned objects as the following code shows:

```
$object->method()->method()->member = 5;
```

Due to the fact that in the middle of this expression handles to objects are returned and not the objects themselves, if the methods are written correctly the right objects are always manipulated.

Compatibility notes

As this syntax does not exist in previous versions of the Zend Engine there are no compatibility impacts.

Dependencies of feature

This feature is dependent upon the new object model of the Zend Engine 2.0.

Object Cloning

Background

Up to now there is no way a user can decide what copy constructor to run when an object is duplicated. At present during duplication the Zend Engine does a “bitwise” copy making an identical replica of all the objects’ properties.

Need

Creating a copy of an object with fully replicated properties is not always the wanted behavior. A good example of the need for copy constructors, is if you have an object which represents a GTK window and the object holds the resource of this GTK window, when you create a duplicate you might want to create a new window with the same properties and have the new object hold the resource of the new window.
Another example is if your object holds a reference to another object which it uses and when you replicate the parent object you want to create a new instance of this other object so that the replica has its own separate copy.

Overview

When the developer asks to create a new copy of an object, the Zend Engine will check if a __clone() method has been defined or not. If not, it will call a default __clone() which will copy all of the object’s properties. If a __clone() method is defined, then it will be responsible to set the necessary properties in the created object. For convenience, the engine will supply a function that imports all of the properties from the source object, so that they can start with a by-value replica of the source object, and only override properties that need to be changed.

Functionality

The suggested syntax for creating a copy of an object is:

```
$copy_of_object = $object->__clone();
```

Compatibility notes

If by chance an object from an older script already has a method *__clone()* defined then it might be called not in a way the developer had planned. This should be quite easy to detect and work around.

Dependencies of feature

This feature becomes mainly interesting with the new object model although it could possibly be implemented in the old one too.

Destructors

Background

No mechanism for object destructors exist today although PHP has support for registering functions which should be run on request shutdown.

Need

Having the ability to define destructors for objects can be very useful. Destructors can log messages for debugging, clean up temporary files and so on.

Overview

The proposed solution is like in most other OO languages. When the last reference to an object is destroyed the object's destructor is called before the object is freed from memory. Due to the nature of PHP such functionality still needs to be evaluated closely. For example, when fatal errors occur it might not be possible to call object's destructors or objects which are in a referential loop which the reference counting mechanism can't detect might not have their destructor called.

Functionality

The user will define a special method in his class definition (which doesn't receive arguments). This method will be called *__destruct()*.

So it would look something like:

```
class MyClass
{
     function __destruct()
     {
          … // Run destructor code
     }
}
```

Like constructors, parent destructors will not be called implicitly by the engine. In order to run a parent destructor, one would have to explicitly call parent::__destruct() in the destructor body.

Compatibility notes

No compatibility problems as this feature doesn't exist in previous versions of the scripting engine.

Dependencies of feature

This feature is dependent on the new object model.

delete statement

Background

In the original object model there is no way to force deletion of an object if there are still references to it.

Need

With the new centralized objects we believe that in certain cases developers will want to force an object to be destroyed at a certain point in their program. It might initially seem trivial to do *$obj = NULL*, however, as unexpected variables might still be holding a reference to this object the effect of the previous statement might not free the object at the wanted time. There should be a way to force the Zend Engine to destroy and object even if there are still references to it.

Overview

The proposed *delete* statement will address this problem. It will call the object's destructor and free it even if the object is referenced by some other places in the engine. Other references to the deleted object will become stale and trying to access them will result in a fatal error.

Functionality

As mentioned in the overview section, the *delete* statement will force the object's destruction.
The syntax will be:

```
delete $object;
```

Compatibility notes

No compatibility issues, as this feature doesn't exist in previous versions of the scripting engine.

Dependencies of feature

This feature is dependent on the new object model.

Unified Constructors

Background

The Zend Engine allows developers to declare constructor functions for their classes. Classes which have a constructor function call this function on each newly-created object, so it is suitable for any initialization that the object may need before it can be used. In version 1.0, constructor functions are simply class methods that bare the same name as the class itself. If a method exists that has the same name as the class it is contained in – it's automatically regarded as the constructor.

Need

Since it is very common to call parent constructors from derived classes, the way the Zend Engine 1.0 works makes it a bit cumbersome to move classes around in a large class hierarchy. If a class is moved to reside under a different parent, the constructor name of that parent changes as well, and the code in the derived class that calls the parent constructor has to be modified.

Overview

The Zend Engine 2.0 will introduce a standard way of declaring constructor functions – by simply calling them by the name *__construct()*.

Functionality

For example:

```
class Shape {
     function __construct()
     {
          // shape initialization code
          …
     }
     …
};
```

```
class Square extends Shape {
     function __construct()
     {
          parent::__construct();
          // square-specific initialization code
          …
     }
     …
};
```

If we decide to introduce a new Rectangle class in between the Shape and Square class in the hierarchy:

```
class Rectangle extends Shape {
     function __construct()
     {
          parent::__construct();
          // rectangle initialization code
     }
     …
};
```

We can do so without changing any code in the Square class, and only have to change the class it is derived from:

```
class Square extends Rectangle {
     …
};
```

Compatibility notes

For backwards compatibility, if the engine cannot find a __construct() function for a given class, it will search for the old-style constructor function, by the name of the class. Effectively, it means that the only case that would have compatibility issues is if the class had a method named __construct() which was used for different semantics.

Dependencies of feature

No dependencies

Additional Features

Multiple Inheritance

Note: Multiple Inheritance functionality may or may not be a part of the Zend Engine 2.0. Whether it is added to the engine or not will be publicly discussed.

Background

In the Zend Engine 1.0, classes can only inherit the interface and functionality of a single parent class.

Need

Sometimes, one may wish to create a class that uses functionality from several parent classes. This could be done if inheriting from multiple classes were possible.

Overview

The Zend Engine 2.0 will allow classes to inherit from more than one parent class. The resulting class will have the methods of all of its parent classes, in addition to the methods defined in the class itself. As with single inheritance, the child class will be able to override any method from its parent classes. In case two parent classes have methods with identical names, the child class will be forced to override the method (if possible, the Zend Engine will detect whether the two methods are actually the very same method (in case of 'diamond-shaped' inheritance), and will automatically resolve the conflict.

Functionality

Implementing classes that inherit from multiple parent classes will use the following notation:

```
class child extends parent1, parent2, … {
     …
};
```

Compatibility notes

Certain functions, such as *get_parent_class()* will have to change (e.g., to return the list of parent classes, or just one of the parent classes arbitrarily). Otherwise, multiple inheritance is downwards compatible with single inheritance, so it should introduce no compatibility issues.

Dependencies of feature

In order to implement a sound class hierarchy based on multiple inheritance, it will be necessary to use private member variables. In that sense, this functionality depends on the *private member variables* feature.

Private Member Variables

Note: Private Member Variables functionality may or may not be a part of the Zend Engine 2.0. Whether it is added to the engine or not will be publicly discussed.

Background

In the Zend Engine 1.0, all variables within objects are accessible to the outside world, for both reading and writing.

Need

Often, objects hold information that may only be modified by methods of their class, in order to maintain consistency and data protection. In addition, in case *Multiple Inheritance* is introduced to the Zend Engine 2.0, it will become much more important to ensure that member variables from one parent class don't clash with member variables from another parent class, simply for having the same common name.

Overview

The Zend Engine 2.0 will introduce special notation for declaring variables as private. Private member variables will be accessible *only* to methods that belong to the class to which they belong. In case an attempt is made to access a private variable from outside the scope of a valid class method, a fatal error will occur. Due to the proposed implementation, it will not be possible to reference private member variables indirectly (i.e., by the use of *$$varname* notation), but only by using the variable name itself.

Functionality

Private member variables will be declared and manipulated in the following way:

```
class foo {
     private $priv_var;

     function some_method(…)
     {
          $priv_var = …;
     }
};

$obj = new foo;
$obj->priv_var = …;      // will result in a fatal error
```

Compatibility notes

No compatibility notes.

Dependencies of feature

No dependencies.

Static Class Member Variables

Background

The Zend Engine 1.0 introduced the ability to call class methods via the class name, instead of using an object instance (e.g. *class_name::method()*). However, there is no way to access class-specific variables using a similar notation. As a matter of fact, classes in the Zend Engine 1.0 don't have any storage of class-specific variables.

Need

Very often, it is desirable to store information that is only relevant to a specific class. In version 1.0, there's no good way of doing that, other than using a global variable, usually with an appropriate prefix that would avoid collisions with other global variables.

Overview

Version 2.0 will introduce class-local variables – variables which belong to a specific class. The syntax for accessing these variables will be similar to that of accessing class methods via the class name.

Functionality

For example, if you wish to implement a singleton class:

```
class Logger {
     static $m_Instance = NULL;

     function Instance()
     {
          if (Logger::$m_Instance == NULL) {
               Logger::$m_Instance = new Logger();
          }
          return Logger::$m_Instance;
     }
     function Log()
```

```
    {
        …
    }
};

$Logger = Logger::Instance();
$Logger->Log(…);
```

Compatibility notes

No compatibility issues

Dependencies of feature

No dependencies.

Exception handling (*try*/*throw*/*catch*)

Background

No exception handling mechanism exists today. There are certain errors that can be raised and processed by a registered error handler but there isn't structured exception handling on the language level.

Need

Exception handling is a very nice tool when used correctly (it should be used to catch errors and not to control regular program flow). It is often useful to developers to be able to protect a big block of code with a *try*/*catch* construct so that each error doesn't have to be handled on each line where an error could occur but in general for the complete code block. The reason why this often makes sense is because if not all of the code executes well you might just want to print an error message and it saves writing error handling code on each line where something could go wrong.

Overview

The proposed implementation of *try*/*throw*/*catch* will look similar as other popular programming languages. Any code which is inside a *try*/*catch* block can *throw* an exception passing the exception to the closest exception handler. If this exception handler doesn't want to take care of the exception it can re-throw the exception.
Internal functions (usually written in C) will also be able to raise exceptions, however, they will most probably continue to return error codes as return values and not as exceptions. Most developers will probably not use exceptions and the current error reporting of internal functions is quite good.

Functionality

The currently suggested syntax is as follows:

```
1    try {
2          …code
```

```
3         if (failure) {
4              throw new MyException(“Failure”);
5         }
6         …code
7    } catch ($exception) {
8         … handle exception
9
10        throw $exception; // Re-throw exception.
11   }
```

The code in the *try*/*catch* block starts at line 2 and ends at line 6. Any exception which is raised inside this block is handled by the *catch* statements (unless there is a *try*/*catch* block which is closer in scope. Any kind of value can be thrown and when the exception reaches the *catch* statements it will be accessibly via the *$exception* variable which the developer can define (on line 7 *$exception* could for instance be exchanged with *$foobar*).

When an exception value is thrown it is thrown by-value. The exception will skip all code and unwind the function stack until it finds the closest *catch*. The exception handling code can check $exception, decide if it wants to handle it or not and either handle it or re-throw it at the end in order to make it propagate to the next *catch()* in line.

NOTE: This feature may cause memory leaks in certain situations. Therefore, developers should make sure that they aren’t using it for flow-control but only for structured exception handling in applications which don’t have a long life time (such as web applications).

Compatibility notes

No compatibility problems exist, as this feature doesn’t exist in previous versions of the scripting engine. In order to simplify error handling in the existing code base, the engine will support a mode in which errors (such as E_WARNING and E_NOTICE) will raise exceptions, instead of displaying an error. This will allow users to use one try..catch

statement to recover from any possible errors during the course of a large code block (e.g., establishing a connection to a database server, selecting a database, and issuing a query), without having to add lots of error-handling code.

Dependencies of feature

No dependencies.

Revamped OO Syntax Overloading

Background

PHP 4.0 introduced the concept of overloading the object oriented syntax of the language, to manipulate external objects and/or components, such as COM components or Java objects. This feature allowed to implement the interface to OO facilities in a much more intuitive way, compared to the old procedural interface that was the only option in PHP 3.0.

However, the implementation of Zend Engine 1.0's OOSO (as found in PHP 4.0) is limited in several ways, and is relatively difficult to program for.
The limitations and difficulties:

1. Due to the Zend Engine 1.0's parser limitations, it's not always possible to write a valid expression that corresponds to a desired behavior. For instance, **`$foo->bar()->baz = 5;`**
   which corresponds to an expression that makes sense in both COM and Java, is not a valid Zend expression.
2. The performance of the overloading mechanism is relatively low, because the Zend Engine maintains a complete 'parse tree' for every overloaded expression during execution.
3. Implementing an OOSO module is difficult, because it later on has to traverse the above parse tree to perform all of the necessary operations.
4. An OOSO module must overload the entire object; It's impossible to overload only parts of an object, and have the other parts behave as standard Zend objects.

Need

The above deficiencies are mostly notable in PHP 4.0's lacking Java implementation, which is both limited and difficult to debug. It's also very much noticeable in the new PHP-GTK extension, which is gaining popularity rapidly; PHP-GTK is greatly harmed by all of the deficiencies mentioned in section 2.1, especially 2.1.1 and 2.1.4.

Overview

The Zend Engine 2.0 will feature a new interface to OOSO, that will address all of the issues mentioned in section 2.1. The idea is to neglect the approach of developing a parse tree at run-time, and passing it on to the OOSO module for resolution; Instead, the Zend Engine will allow specific mini-operations (fetching, assignment and method calling) to be overloaded separately at each stage. This will:

- Simplify development of OOSO modules
- Allow overloaded object to contain both other overloaded objects, or standard Zend values (zval's).

Significantly improve performance, as overloaded operations will occur just in time, at the same stage as the standard Zend fetching/writing/method calling operations would have occurred.

Functionality

The new OOSO modules will help make overloaded interfaces more intuitive to write for module developers; Perhaps more importantly, relying on the syntax improvements that are scheduled for the Zend Engine 2.0, the new OOSO modules will allow complete syntax compatibility between object models such as COM and Java, and Zend Engine expressions.

Compatibility Notes

End users should experience no compatibility issues; The new API will allow for modules which support the same functionality as the old ones, along with some new functionality (such as $foo->bar()->baz support). It will, however, be necessary to rewrite all of the existing OOSO modules to implement the new API.

Dependencies

This feature is dependent upon the new object model of the Zend Engine 2.0.

String offset syntax

Background

Sometimes, it is desirable to access a specific character from within a string. The Zend Engine 1.0 has no special notation for referring to such string offsets. Instead, strings can be referred to as arrays, and using array offset notation, one can access a specific character within a string.

Need

There are three main reasons why specialized string offsets syntax would be a good addition to the Zend Engine:

a) There is an ambiguity in today's sharing of array offsets syntax for both strings and arrays. In code such as $str[0] = 'a'; where $str is an empty string (due to earlier PHP 4 and PHP 3 behavior) $str is treated as an array and therefore the above statement would result in $str[0] being converted to an array and having its offset 0 being set to the string "a". However, some people would expect the result to be $str as the string value "a".
b) By introducing a specialized syntax for string offsets it will be possible to somewhat optimize the run-time processing, as we will know at compile-time that the user specifically means to use a string offset.
c) Language wise it is much better if developers will be able to tell if the author meant to use array offsets or string offsets in a code snippet.

Overview

The Zend Engine will feature a new syntax for accessing string offsets. Using the array offsets syntax for string offsets will be deprecated initially by a run-time warning (possibly E_NOTICE or maybe a possible E_STRICT).

Functionality

The currently suggested syntax is as follows:

```
$str{2} = ‘a’;
```

An example of the new functionality:

```
$str1 = $str2 = “”;
$str1{0} = ‘a’;
$str2[0] = ‘a’;
```

The result will be $str1 being the string “a” and $str2 being *array(0 => “a”)*.

Compatibility notes

Using the array offset syntax for string offsets will print a warning in order to allow for people to migrate their scripts. Due to backwards compatibility problems it is doubtful if we will ever be able to change the array offset syntax to raise an error if used in a string context.

Dependencies of feature

No dependencies.