

New features in the ZE2's OO API

Part 1: Overview



Harald Radi <harald.radi@nme.at>

PHP – Conference 2003 Spring Edition

Agenda



- New features in ZE2
- Constructing an object
- zend_object_value structure
- zend_object_handlers structure
- zend_class_entry structure
- Related Zend macros and functions
- The object store
- Future development

New Features in ZE2 (1)



- Improved object handling
 - Objects are no more passed by value
 - Objects are referenced by a handle
 - Objects are explicitly cloneable
- No serious implications known
- More natural when having other OO languages in mind

New Features in ZE2 (2)

- Improved object dereferencing
 - Objects returned by method or function calls can now be directly dereferenced

```
$object->method() ->method();  
$object->method() ->member->method();  
function() ->method();
```

- Prevents several programming errors
- Common syntax in other languages

New Features in ZE2 (3)

- Explicit object cloning
 - Objects will never be cloned implicitly (like in earlier PHP versions)
 - Explicit cloning gives more control to the user

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

- `__clone()` can be overloaded

```
function __clone() {  
    $that->foo = $this->foo;  
}
```

New Features in ZE2 (4)

- Object destructors
 - Are called when an object is completely dereferenced (unlike PEAR destructors)
 - But there might be exceptions ...
(fatal errors, engine bailouts, ...)

```
function __destruct() {  
    xyz_free($this->id);  
}
```

New Features in ZE2 (5)

- Interfaces

- Exposing multiple functionalities without multiple inheritance
- subtyping vs. subclassing

```
class Foo implements Bar {  
    function foobar() { print „hello world“; }  
}
```

- Type hints

```
function foo(Bar $bar) { ... }
```

New Features in ZE2 (6)

- Namespaces
 - May contain classes, functions, variables and constants
 - Reduce naming clashes
 - No support for nested namespaces

```
namespace Foo {  
    function bar() { ... }  
}
```

- ,:‘ as syntactical sugar ;-)

New Features in ZE2 (7)

- Exception handling

```
try {  
    throw new Exception(„foo“);  
} catch (Exception $ex) {  
    print $ex->name;  
}
```

- Exceptions should not (yet) be used to control the program flow
 - Unfortunately memory leaks prevent you from using exceptions in cli or gtk applications



Constructing an object

- Two kinds of objects
 - Ad-Hoc objects
 - Objects that aren't instances of a given class.
 - Ad-Hoc objects cannot be instantiated with `new`.
 - Class objects
 - Instances of a declared class.
 - Fully reflexible and instantiatable with `new`.

zend_object_value

- Internal representation of an object

```
struct _zend_object_value {
    zend_object_handle handle;
    zend_object_handlers *handlers;
};
```

- zval value member

```
typedef union _zvalue_value {
    long lval;
    double dval;
    struct {
        char *val;
        int len;
    } str;
    HashTable *ht;
    zend_object_value obj;
} zvalue_value;
```

zval container

- Internal representation of a variable

```
struct _zval_struct {  
    /* Variable information */  
    zvalue_value value;  
    zend_uint refcount;  
    zend_uchar type;  
    zend_uchar is_ref;  
};
```

Each variable in PHP has a correspondent zval container in C

zend_object_handlers (1)



- Hooks to react to object manipulations in userland
- Most of them are optional
- Default implementations are available

The new OO overloading provides the most flexibility, from basic overloading as in PHP4 down to hooking the engine inwards.

zend_object_handlers (2)

```
typedef struct _zend_object_handlers {
    /* general object functions */
    zend_object_add_ref_t add_ref;
    zend_object_del_ref_t del_ref;
    zend_object_delete_obj_t delete_obj;
    zend_object_clone_obj_t clone_obj;
    /* individual object functions */
    zend_object_read_property_t read_property;
    zend_object_write_property_t write_property;
    zend_object_get_property_ptr_t get_property_ptr;
    zend_object_get_property_zval_ptr_t get_property_zval_ptr;
    zend_object_get_t get;
    zend_object_set_t set;
    zend_object_has_property_t has_property;
    zend_object_unset_property_t unset_property;
    zend_object_get_properties_t get_properties;
    zend_object_get_method_t get_method;
    zend_object_call_method_t call_method;
    zend_object_get_constructor_t get_constructor;
    zend_object_get_class_entry_t get_class_entry;
    zend_object_get_class_name_t get_class_name;
    zend_object_compare_t compare_objects;
} zend_object_handlers;
```

add_ref

is called whenever a new reference to the object is created

```
typedef void (*zend_object_add_ref_t)(zval *object TSRMLS_DC);
```

Example

```
<?php  
$foo = new Bar();  
$baz = $foo;  
?>
```

del_ref

is called whenever a reference to the object is released, the object will not be destroyed

```
typedef void (*zend_object_del_ref_t)(zval *object TSRMLS_DC);
```

Example

```
<?php  
$foo = new Bar();  
$baz = $foo;  
$baz = null;  
?>
```

delete_obj

is called whenever all references to the object were released, the object should be destroyed now

```
typedef void (*zend_object_delete_obj_t)(zval *object TSRMLS_DC);
```

Example

```
<?php  
$foo = new Bar();  
$foo = null;  
?>
```

clone_obj

is called whenever an object should be cloned

```
typedef zend_object_value (*zend_object_clone_obj_t)(  
    zval *object TSRMLS_DC);
```

Example

```
<?php  
$foo = new Bar();  
$bar = $foo->__clone();  
?>
```

read_property

is called whenever an object's property is requested (read-only)

```
typedef zval *(*zend_object_read_property_t)(  
    zval *object, zval *member TSRMLS_DC);
```

if defined, `get_property_zval_ptr` is used instead

Example

```
<?php  
$foo = new Bar();  
$baz = $foo->baz;  
?>
```

write_property

is called whenever an object's property is changed

```
typedef void (*zend_object_write_property_t)(  
    zval *object, zval *member, zval *value TSRMLS_DC);
```

if defined, `get_property_zval_ptr` is used instead

Example

```
<?php  
$foo = new Bar();  
$foo->baz = 123;  
?>
```

get_property_ptr

is called whenever a reference to a property is requested

```
typedef zval **(*zend_object_get_property_ptr_t)(  
    zval *object, zval *member TSRMLS_DC);
```

Example

```
<?php  
$foo = new Bar();  
$baz = &$foo->baz;  
?>
```

get_property_zval_ptr

is called whenever a property is requested for read or write access

```
typedef zval **(*zend_object_get_property_zval_ptr_t)(  
    zval *object, zval *member TSRMLS_DC);
```

Example

```
<?php  
$foo = new Bar();  
$foo->baz = 123;  
$baz = $foo->baz;  
?>
```



get

is called whenever a proxy object is attempted to be read

```
typedef zval* (*zend_object_get_t)(zval *property TSRMLS_DC);
```

Example

```
<?php
$foo = new Bar();
$baz = &$foo->baz;
$heh = $baz;
?>
```

set

is called whenever a proxy object is attempted to be written

```
typedef void (*zend_object_set_t)( zval **property, zval *value TSRMLS_DC );
```

Example

```
<?php  
$foo = new Bar();  
$baz = &$foo->baz;  
$baz = 123;  
?>
```

has_property

is called to check if a specific property is defined for that object

```
typedef int (*zend_object_has_property_t)(
    zval *object, zval *member, int check_empty TSRMLS_DC);
```

Example

```
<?php
$foo = new Bar();
if (isset($foo->baz)) {
    /* do something */
}
?>
```

unset_property

is called when a property should be undefined for that object

```
typedef void (*zend_object_unset_property_t)(  
    zval *object, zval *member TSRMLS_DC);
```

Example

```
<?php  
$foo = new Bar();  
unset($foo->baz);  
?>
```

get_properties

is called when a list of defined properties is requested to reflect that object

```
typedef HashTable *(*zend_object_get_properties_t)(zval *object TSRMLS_DC);
```

Example

```
<?php
$foo = new Bar();
get_object_vars($foo);    /* ? */
?>
```

get_method

is called to check if a method is defined or not

```
typedef union _zend_function *(*zend_object_get_method_t)(  
    zval *object, char *method, int method_len TSRMLS_DC);
```

if NULL is returned call_method is called

Example

```
<?php  
$foo = new Bar();  
$foo->bar();  
?>
```

call_method

is called when an overloaded (undefined) method is called

```
typedef int (*zend_object_call_method_t)(  
    char *method, INTERNAL_FUNCTION_PARAMETERS);
```

Example

```
<?php  
$foo = new Bar();  
$foo->bar();  
?>
```

get_constructor

this function is used to retrieve the objects constructor function

```
typedef union _zend_function *(*zend_object_get_constructor_t)(  
    zval *object TSRMLS_DC);
```

Example

```
<?php  
$foo = new Bar();  
?>
```

get_class_entry

is called when the class definition is required to reflect the object

```
typedef zend_class_entry *(*zend_object_get_class_entry_t)(  
    zval *object TSRMLS_DC);
```

Example

```
<?php  
$foo = new Bar();  
if (is_subclass_of($foo, „Foo“)) {  
    /* do something */  
}  
?>
```

get_class_name

is called to retrieve the class name or the name of the parent class

```
typedef int (*zend_object_get_class_name_t)(  
    zval *object, char **class_name, zend_uint *class_name_len,  
    int parent TSRMLS_DC);
```

Example

```
<?php  
$foo = new Bar();  
get_class($foo);  
?>
```

compare_objects

is called when two objects are compared with `==`, whereas `===` compares the object handles directly

```
typedef int (*zend_object_compare_t)(
    zval *object1, zval *object2 TSRMLS_DC);
```

Example

```
<?php
$foo = new Bar();
$bar = new Bar();
if ($foo == $bar) {
    /* do something */
}
```

Ad-Hoc objects

- No corresponding `zend_class_entry`
- Can't be reflected using the reflection API
- Can only be created as return values
- Are easy to create

Just assign a unique object handle and a object handlers structure to the `zval` container and set its type to `IS_OBJECT`.

e.g.: `mysql_fetch_object()` never returns the same object twice; Ad-Hoc objects are the best choice then.

Class objects

- More difficult than Ad-Hoc objects
- Fully reflexible by PHP's reflection API
- For each class a `zend_class_entry` structure has to be exported

It contains the class definition containing the parent scope of the class, declared variables and their default values, class methods and constants.

zend_class_entry (1)

```
struct _zend_class_entry {
    char type;
    char *name;
    zend_uint name_length;
    struct _zend_class_entry *parent;
    int refcount;
    zend_bool constants_updated;
    zend_uint ce_flags;
    HashTable function_table;
    HashTable default_properties;
    HashTable properties_info;
    HashTable class_table;
    HashTable *static_members;
    HashTable constants_table;
    zend_function_entry *builtin_functions;
    struct _zend_class_entry *ns;
    union _zend_function *constructor;
    union _zend_function *destructor;
    union _zend_function *clone;
```

zend_class_entry (2)

```
union _zend_function *__get;
union _zend_function *__set;
union _zend_function *__call;
/* handlers */
zend_object_value(*create_object)(zend_class_entry *class_type
TSRMLS_DC);
zend_class_entry **interfaces;
zend_uint num_interfaces;
char *filename;
zend_uint line_start;
zend_uint line_end;
char *doc_comment;
zend_uint doc_comment_len;
};
```

Related Zend macros

- **INIT_CLASS_ENTRY**

```
INIT_CLASS_ENTRY(class_container, class_name, functions)
```

- **INIT_OVERLOADED_CLASS_ENTRY**

```
INIT_OVERLOADED_CLASS_ENTRY(class_container, class_name,  
    functions, handle_fcall, handle_propget,  
    handle_propset)
```

- **INIT_NAMESPACE**

```
INIT_NAMESPACE(ns_container, ns_name)
```

Related Zend functions (1)

- zend_register_internal_class

```
ZEND_API zend_class_entry *zend_register_internal_class (  
    zend_class_entry *class_entry TSRMLS_DC)
```

- zend_register_internal_class_ex

```
ZEND_API zend_class_entry *zend_register_internal_class_ex (  
    zend_class_entry *class_entry, zend_class_entry *parent_ce,  
    char *parent_name TSRMLS_DC)
```

Related Zend functions (2)

- zend_register_internal_ns_class

```
ZEND_API zend_class_entry *zend_register_internal_ns_class (  
    zend_class_entry *class_entry, zend_class_entry *parent_ce,  
    zend_namespace *ns, char *ns_name TSRMLS_DC)
```

- zend_register_internal_namespace

```
ZEND_API zend_namespace *zend_register_internal_namespace (  
    zend_namespace *ns TSRMLS_DC);
```



The object store (1)

- Similar to the resource list
 - Used to store the actual object data
 - Generates unique object handles
 - Fast access to the object data based on the handle
 - Handles reference counting

The object store (2)

- zend_objects_store_put

```
ZEND_API zend_object_handle zend_objects_store_put(  
    void *object, zend_objects_store_dtor_t dtor,  
    zend_objects_store_clone_t clone TSRMLS_DC)
```

- zend_objects_store_delete_obj

```
ZEND_API void zend_objects_store_delete_obj(zval *object TSRMLS_DC)
```

- zend_objects_store_get_object

```
ZEND_API void *zend_object_store_get_object(zval *object TSRMLS_DC)
```

The object store (3)

- **ZEND_OBJECT_STORE_HANDLERS**
default handlers for the zend_object_handlers structure to abstract the reference counting

```
ZEND_OBJECTS_STORE_HANDLERS
```

- **zend_get_std_object_handlers**
get standard object handler callbacks

```
ZEND_API zend_object_handlers *zend_get_std_object_handlers()
```

Future development



- Indexer overloading

overloading the indexer operator [] for members of an object AND the object itself

- Serialization callback

having `__sleep()` and `__wakeup()` available for your overloaded objects

- Built-in interfaces

Serializable, Persistable, Throwable, Enumerable, ...

Break



<http://www.nme.at>