Zephir Documentation

Release 0.2.0a

Zephir Team

August 26, 2013

Contents

CHAPTER 1

Reference

1.1 Welcome!

Welcome to Zephir, an open source, high level/domain specific language designed to ease the creation and maintainability of extensions for PHP with a focus on type and memory safety.

1.1.1 Some features

Zephir's main features are:

Type system	dynamic/static
Memory safety	pointers or direct memory managament aren't allowed
Compilation model	ahead of time
Memory model	task-local garbage collection

1.1.2 A small taste

The following code registers a class with a method that filters variables returning its alphabetic characters:

```
namespace MyLibrary;

/**
 * Filter
 */
class Filter
{
    /**
     * Filters a string returning its alpha characters
     *
     * @param string str
     */
    public function alpha(string str)
    {
        char ch; string filtered = "";

        for ch in str {
           if (ch >= 'a' && ch <= 'z') || (ch >= 'A' && ch <= 'Z') {
                let filtered .= ch;
            }
        }

        return filtered;
    }
}
```

The class can be used from PHP as follows:

```
<?php

$filter = new MyLibrary\Filter();
echo $filter->alpha("01he$l.lo?/1"); // prints hello
```

1.2 Why Zephir?

Today's PHP applications must balance a number of concerns including stability, performance and functionality. Every PHP application is based on a set of common components that are also base for most of the application.

These common components are libraries/frameworks or a combination of them. Once installed, frameworks rarely change, being the foundation of the application they must be highly functional and also very fast.

Gettting fast and robust libraries can be complicated due to high levels of abstraction that are typically implemented on them. Given the condition that base libraries or frameworks rarely change, there is an opportunity to build extensions that provide this functionality taking advantage of the compilation improving performance and resource consumption.

With Zephir you can implement object-oriented libraries/frameworks/applications that can be used from PHP gaining important seconds that can make your application faster while improving the user experience.

1.2.1 If You Are a PHP Programmer...

PHP is one of the most popular languages in use for the development of web applications. Dynamically typed and interpreted languages like PHP offer very high productivity due to their flexibility.

Since version 4 and then 5, PHP is based on the Zend Engine implementation. This is a virtual machine that executes the PHP code from its bytecode representation. Zend Engine is almost present in every PHP installation in the world. With Zephir you can create extensions for PHP running under the Zend Engine.

PHP is hosting Zephir, so they obviously have a lot of similarities, however, they have important differences that give Zephir its own personality. For example, Zephir is more strict and it could be make you less productive compared to PHP due to the compilation step.

1.2.2 If You Are a C Programmer...

C is one of the most powerful and popular languages ever created. In fact, PHP is written in C, which is one of the reasons why PHP extensions are available for it. C gives you the freedom to manage memory, use low level types and even inline assembly routines.

However, developing big applications in C can take much longer than expected compared to PHP or Zephir and some errors can be tricky to find if you aren't an experienced developer.

Zephir was designed to be safe, so it doesn't implement pointers or manual memory management, so if you're a C programmer, you will feel Zephir less powerful but more friendly than C.

1.2.3 Compilation vs Interpretation

Compilation usually slows the development down, you will need a bit more of patience to make your code compiled before see it running. Moreover, the interpretation tends to reduce the performance in favor of productivity. In some cases there is no any noticeable difference between the speed of the interpreted and compiled code.

Zephir requires compilation of your code, however the functionality is used from PHP that is interpreted.

Once the code is compiled is not necessary to do so, however, interpreted code is interpreted each time it is run. A developer can decide which parts of your application should be in Zephir and which not.

1.2.4 Statically Typed Versus Dynamically Typed Languages

In general speaking, in a static typed language, a variable is bound to a particular type for its lifetime. Its type can't be changed and it can only reference type-compatible instances and operations. Languages like C/C++ were implemented with the scheme:

```
int a = 0;
a = "hello"; // not allowed
```

In dynamic typing, the type is bound to the value, not the variable. So, a variable might refer to a value of a type, then be reassigned later to a value of an unrelated type. Javascript/PHP are examples of a dynamic typed language:

```
var a = 0;
a = "hello"; // allowed
```

Despite their productivity advantages, dynamic languages may not be the best choices for all applications, particularly for very large code bases and high-performance applications.

Optimizing the performance of a dynamic language like PHP is more challenging than for a static language like C. In a static language, optimizers can exploit the type information to make decisions. In a dynamic language, fewer such clues are available for the optimizer, making optimization choices harder.

While recent advancements in optimizations for dynamic languages are promising (like JIT compilation), they lag behind the state of the art for static languages. So, if you require very high performance, static languages are probably a safer choice.

Another small benefit of static languages is the extra checking the compiler performs. A compiler can't find logic errors, which are far more significant, but a compiler can find errors in advance that in a dynamic language only can be found in runtime.

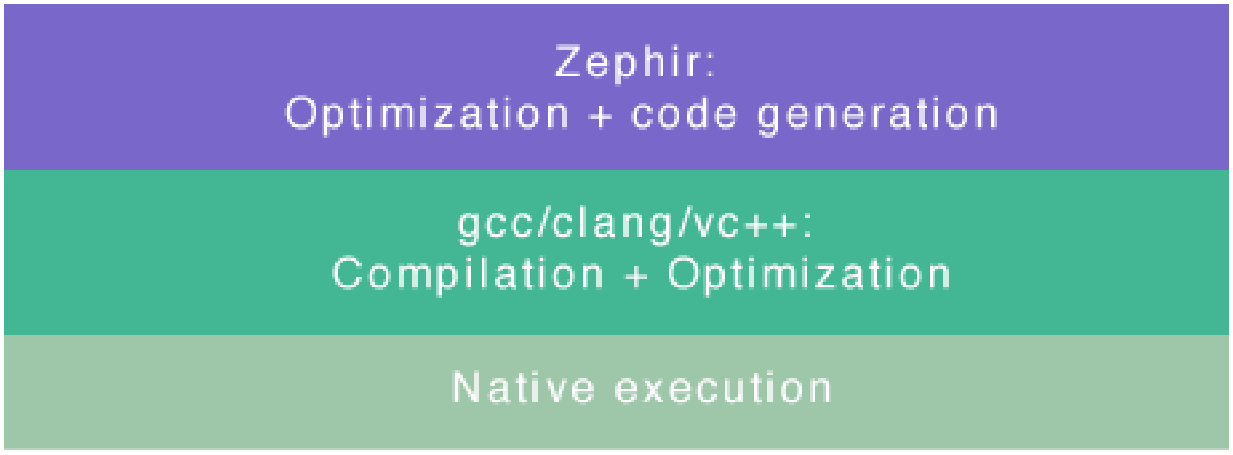
Zephir is both statically and dynamically typed allowing you to take advantage from both sides where possible.

1.2.5 Compilation Scheme

Zephir offers native code generation (currently via compilation to C), a compiler like gcc/clang/vc++ optimizes and compiles the code down to machine code. The following graph shows how the process works:

In addition to the ones provided by Zephir, Over time, compilers have been implemented and matured a number of optimizations that improve the performance of compiled applications:

- GCC optimizations
- LLVM passes
- Visual C/C++ optimizations



1.2.6 Conclusion

Zephir was not created to replace PHP or C, instead of this, we think it is a complement to them, allowing developers to venture into code compilation and static typing. Zephir is precisely an attempt to join good things from the C and PHP worlds, looking for oportunities to make their applications faster.

1.3 Introducing Zephir

Zephir is a language that addresses the major needs of a PHP developer trying to write and compile code that can be executed by PHP. It is a dynamically/statically typed, some of its features can be familiar to PHP developers.

The name Zephir is a contraction of the words Zend Engine/PHP/Intermediate. While this suggests that the pronunciation should be zephyr, the creators of Zephir actually pronounce it zaefire.

1.3.1 Hello World!

Every language has its own “Hello World!”, in Zephir this introductory example showcase some important features of this language.

Code in Zephir must be placed in classes, Zephir is intended to create object-oriented libraries/frameworks, so code out of a class is not allowed. Also a namespace is required:

```
namespace Test;

/**
 * This is a sample class
 */
class Hello
{
    /**
     * This is a sample method
     */
    public function say()
    {
        echo "Hello World!";
    }
}
```

Once this class is compiled it produce the following code that is transparently compiled by gcc/clang/vc++:

```
#ifdef HAVE_CONFIG_H
#include "config.h"
#endif

#include "php.h"
#include "php_test.h"
#include "test.h"

#include "kernel/main.h"

/**
 * This is a sample class
 */
ZEPHIR_INIT_CLASS(Test_Hello) {
    ZEPHIR_REGISTER_CLASS(Test, Hello, hello, test_hello_method_entry, 0);
    return SUCCESS;
}

/**
 * This is a sample method
 */
PHP_METHOD(Test_Hello, say) {
    php_printf("%s", "Hello World!");
}
```

Actually, it is not expected that a developer that use Zephir must understand or know C, however if you have any experience with compilers, php internals or the C language itself, it would provide a more clear sceneario to the developer when working with Zephir.

1.3.2 A Taste of Zephir

In the following examples, we'll describe just enough of the details so you understand what's going on. The goal is to give you a sense of what programming in Zephir is like. We'll explore the details of the features in subsequent chapters.

The following example is very simple it implements a class and a method with an small program that checks the types of an array

Let's examine the code in detail, so we can begin to learn Zephir syntax. There are a lot of details in just a few lines of code! We'll explain the general ideas here:

```
namespace Test;

/**
 * MyTest (test/mytest.zep)
 */
class MyTest
{
    public function someMethod()
    {
        /* Variables must be declared */
        var myArray;
        int i = 0, length;

        /* Create an array */
        let myArray = ["hello", 0, 100.25, false, null];
        /* Count the array into a 'int' variable */
        let length = count(myArray);

        /* Print value types */
        while i < length {
            echo typeof myArray[i], "\n";
            let i++;
        }

        return myArray;
    }
}
```

In the method, the first lines use the 'var' and 'int' keywords are used to declare a variables in a local scope. Every variable used in a method must be declared with its respective type. This declaration is not optional, it helps the compiler to report you about mistyped variables or about the use of variables out of their scope which usually ends in runtime errors.

Dynamic variables are declared with the keyword 'var'. These variables can be assigned and reassigned to different types. On the other hand, we have 'i' and 'length' integer static typed variables that can only have values of this type in the entire program execution.

Compared to PHP you don't require a dollar sign ($) in front of variables.

By the way, that's a comment on the first line (with the name of the source file for the code example). Zephir follows the same comment conventions as Java, C#, C++, etc. A //comment goes to the end of a line, while a /* comment */ can cross line boundaries.

Variables are by default inmutable, this means that Zephir expects that most variables stand unchanged. Variables that maintain their initial value can be optimized down by the compiler to static constants. When the value of a variable needs to be changed, the keyword 'let' must be used:

```
/* Create an array */
let myArray = ["hello", 0, 100.25, false, null];
```

By default, arrays are dynamical like in PHP, they may contain values of different types. Functions from the PHP userland can be called in Zephir code, in the example the function 'count' was called, the compiler can performs optimizations like avoid this call because it already knows the size of the array:

```
/* Count the array into a 'int' variable */
let length = count(myArray);
```

Parentheses in control flow statements are optional, you can also use them if you feel more confortable.

```
while i < length {
    echo typeof myArray[i], "\n";
    let i++;
}
```

PHP only works with dynamic variables, methods always return dynamic variables, this means that if a static typed variable is returned, in the PHP side, you will get a dynamic variable that can be used in PHP code. Note that memory is automatically managed by the compiler, so you don't need to allocate or free memory like in C, working in a similar way than PHP.

1.4 Tutorial

Zephir and this book are intended for PHP developers which wants to create C-extensions with a lower complexity.

We assume that you are experienced in one or more other programming languages. We draw parallels to features in PHP, C, Javascript, and other languages. If you know any of these languages, we'll point out similar features in Zephir, as well as many features that are new or different.

1.5 Basic Syntax

In this chapter, we'll discuss organization of files and namespaces, variable declarations, miscellaneous syntax conventions, and a few other concepts.

1.5.1 Organizing Code in Files and Namespaces

In PHP, you can place code in any file without a specific structure. In Zephir every file must contain a class (and just one class). Every class must have a namespace and the directory structure must match the names of classes and namespaces used.

For example given the following structure the classes in each file must be:

```
mylibrary/
        router/
                exception.zep # MyLibrary\Router\Exception
        router.zep # MyLibrary\Router
```

Class in mylibrary/router.zep:

```
namespace MyLibrary;

class Router
{

}
```

Class in mylibrary/router/exception.zep:

```
namespace MyLibrary\Router;

class Router extends Exception
{

}
```

Zephir will raise a compiler exception if a file or class is not located at the expected file or viceversa.

1.5.2 Instruction separation

You may have already noticed that there were very few semicolons in the code examples in the previous chapter. You can use semicolons to separate statements and expressions, as in Java, C/C++, PHP, and similar languages:

```
myObject->myMethod(1, 2, 3); echo "world";
```

1.5.3 Comments

Zephir supports 'C'/'C++' comments, these are one line comments with // and multi line comments with /* ... */:

```
// this is one line comment

/**
 * multi-line comment
 */
```

In most languages comments are simply text ignored by the compiler/interpreter. In Zephir, multi-line comments are also used as docblocks and they're exported to the generated code, so they're part of the language!.

The compiler would throw an exception if a docblock is not located where is expected.

1.5.4 Variable Declarations

In Zephir, all variables used in a given scope must be declared, this process gives important information to the compiler to perform optimizations and validations. Variables must be unique identifiers and they cannot be reserved words.

```
//Declaring variables for the same type in the same instruction
var a, b c;

//Declaring each variable in different lines
var a;
var b;
var c;
```

Variables can optionally have an initial compatible default value, you can assign a new value to a variable as often as you want:

```
//Declaring variables with default values
var a = "hello", b = 0, c = 1.0;
int d = 50; bool some = true;
```

Variable names are case-sensitive, the following variables are different:

```
//Different variables
var somevalue, someValue, SomeValue;
```

1.5.5 Variable Scope

All variables declared are locally scoped to the method where they were declared:

```
namespace Test;

class MyClass
{

    public function someMethod1()
    {
        int a = 1, b = 2;
        return a + b;
    }

    public function someMethod2()
    {
        int a = 3, b = 4;
        return a + b;
    }
}
```

1.5.6 Super Globals

Zephir doesn't support global variables, accessing global variables from the PHP userland is not allowed. However, you can access the PHP's super-globals as follows:

```
//Getting a value from _POST
let price = _POST['price'];

//Read a value from _SERVER
let requestMethod = _SERVER['REQUEST_METHOD'];
```

1.5.7 Local Symbol Table

Every method or context in PHP has a symbol table that allows to write variables in a very dynamical way:

```
<?php

$b = 100;
$a = "b";
echo $$a; // prints 100
```

Zephir does not implement this feature since all variables are compiled down to low level variables, and there is no way to know which variables do exist in a specific context.

1.6 Types

Zephir is both dynamic and static typed. In this chapter we highlight the supported types and its behavior:

1.6.1 Dynamic Type

Dynamic variables are exactly like the ones in PHP, they can be assigned and reassigned to different types without restriction.

A dynamic variable must be declared with the keyword 'var', the behavior is nearly the same as in PHP:

```
var a, b, c;

// Initialize variables
let a = "hello", b = false;

// Change their values
let a = 10, b = "140";

// Perform operations between them
let c = a + b;
```

They can have eight types:

Type	Description
boolean	A boolean expresses a truth value. It can be either 'true' or 'false'.
integer	Integer numbers. The size of an integer is platform-dependent.
float/double	Floating point numbers. The size of a float is platform-dependent.
string	A string is series of characters, where a character is the same as a byte.
array	An array is an ordered map. A map is a type that associates values to keys
object	Object abstraction like in PHP
resource	A resource holds a reference to an external resource
null	The special NULL value represents a variable with no value

Check more info about these types in the PHP manual

Boolean

A boolean expresses a truth value. It can be either 'true' or 'false':

```
var a = false, b = true;
```

Integer

Integer numbers. The size of an integer is platform-dependent, although a maximum value of about two billion is the usual value (that's 32 bits signed). 64-bit platforms usually have a maximum value of about 9E18. PHP does not support unsigned integers so Zephir has this restriction too:

```
var a = 5, b = 10050;
```

Integer overflow

Contrary to PHP, Zephir does not automatically checks for integer overflows, like in C if you are doing operations that may return a big number you can use types such as 'unsigned long' or 'float' to store them:

```
unsigned long my_number = 2147483648;
```

Float/Double

Floating-point numbers (also known as "floats", "doubles", or "real numbers"). Floating-point literals are expressions with zero or more digits, followed by a period (.), followed by zero or more digits. The size of a float is platform-dependent, although a maximum of ~1.8e308 with a precision of roughly 14 decimal digits is a common value (the 64 bit IEEE format).

```
var number = 5.0, b = 0.014;
```

Floating point numbers have limited precision. Although it depends on the system, as PHP, Zephir uses the IEEE 754 double precision format, which will give a maximum relative error due to rounding in the order of 1.11e-16.

String

A string is series of characters, where a character is the same as a byte. As PHP, Zephir only supports a 256-character set, and hence does not offer native Unicode support.

```
var today = "friday";
```

In Zephir, string literals can only be specified using double quotes (like in C), single quotes are reserved for chars.

The following escape sequences are supported in strings:

Sequence	Description
\t	Horizontal tab
\n	Line feed
\r	Carriage return
\ \	Backslash
\”	double-quote

```
var today = "\tfriday\n\r",
    tomorrow = "\tsaturday";
```

In Zephir, strings don’t support variable parsing like in PHP, you can use concatenation instead:

```
var name = "peter";

echo "hello: " . name;
```

Arrays

The array implementation in Zephir is basically the same as in PHP: Ordered maps optimized for several different uses; it can be treated as an array, list (vector), hash table (an implementation of a map), dictionary, collection, stack, queue, and probably more. As array values can be other arrays, trees and multidimensional arrays are also possible.

The syntax to define arrays is slightly different than in PHP:

```
//Square braces must be used to define arrays
let myArray = [1, 2, 3];

//Double colon must be used to define hashes' keys
let myHash = ["first": 1, "second": 2, "third": 3];
```

Only long and string values can be used as keys:

```
let myHash = [0: "first", 1: true, 2: null];
let myHash = ["first": 7.0, "second": "some string", "third": false];
```

Objects

Zephir allows to instantiate, manipulate, call methods, read class constants, etc from PHP objects:

```
let myObject = new stdClass(),
    myObject->someProperty = "my value";
```

1.6.2 Static Types

Static typing allows the developer to declare and use some variable types available in C. Variables can’t change their type once they’re declared as dynamic types. However, they allow the compiler to do a better optimization job. The following types are supported:

Type	Description
boolean	A boolean expresses a truth value. It can be either 'true' or 'false'.
integer	Signed integers. At least 16 bits in size.
unsigned integer	Unsigned integers. At least 16 bits in size.
char	Smallest addressable unit of the machine that can contain basic character set.
unsigned char	Same size as char, but guaranteed to be unsigned.
long	Long signed integer type. At least 32 bits in size.
unsigned long	Same as long, but unsigned.
float/double	Double precision floating-point type. The size is platform-dependent.
string	A string is series of characters, where a character is the same as a byte.

Boolean

A boolean expresses a truth value. It can be either 'true' or 'false'. Contrary to the dynamic behavior static boolean types remain boolean (true or false) no mater what value is assigned to them:

```
boolean a;

let a = true,
    a = 100, // automatically casted to true
    a = null, // automatically casted to false
    a = "hello"; // throws a compiler exception
```

Integer/Unsigned Integer

Integer values are like the integer member in dynamic values. Values assigned to integer variables remain integer:

```
int a;

let a = 50,
    a = -70,
    a = 100.25, // automatically casted to 100
    a = null, // automatically casted to 0
    a = false, // automatically casted to 0
    a = "hello"; // throws a compiler exception
```

Unsigned integers are like integers but they don't have sign, this means you can't store negative numbers in these sort of variables:

```
let a = 50,
    a = -70, // automatically casted to 70
    a = 100.25, // automatically casted to 100
    a = null, // automatically casted to 0
    a = false, // automatically casted to 0
    a = "hello"; // throws a compiler exception
```

Unsigned integers are twice bigger than standard integers, assign unsigned integers to integers may represent loss of data:

```
uint a, int b;

let a = 2147483648,
    b = a, // possible loss of data
```

Long/Unsigned Long

Long variables are twice bigger than integer variables, thus they can store bigger numbers, As integers values assigned to long variables are automatically casted to this type:

```
long a;

let a = 50,
    a = -70,
    a = 100.25, // automatically casted to 100
    a = null, // automatically casted to 0
    a = false, // automatically casted to 0
    a = "hello"; // throws a compiler exception
```

Unsigned longs are like longs but they aren't signed, this means you can't store negative numbers in these sort of variables:

```
let a = 50,
    a = -70, // automatically casted to 70
    a = 100.25, // automatically casted to 100
    a = null, // automatically casted to 0
    a = false, // automatically casted to 0
    a = "hello"; // throws a compiler exception
```

Unsigned longs are twice bigger than standard longs, assign unsigned longs to longs may represent loss of data:

```
ulong a, long b;

let a = 4294967296,
    b = a, // possible loss of data
```

Char/Unsigned Char

Char variables are the smallest addressable unit of the machine that can contain basic character set. Every 'char' variable represents every character in a string:

```
char ch, string name = "peter";

let ch = name[2]; // stores 't'
```

1.7 Operators

Zephir's set of operators are similar to the ones PHP and also inherits some of their behaviors.

1.7.1 Arithmetic Operators

The following operators are supported:

Operation	Example
Negation	-a
Addition	a + b
Substraction	a - b
Multiplication	a * b
Division	a / b
Modulus	a % b

1.7.2 Comparison Operators

Comparison operators depend on the type of variables compared, for example, if both compared operands are dynamic variables the behavior is the same as in PHP:

a == b	Equal	TRUE if a is equal to b after type juggling.
a === b	Identical	TRUE if a is equal to b, and they are of the same type.
a != b	Not equal	TRUE if a is not equal to b after type juggling.
a <> b	Not equal	TRUE if a is not equal to b after type juggling.
a !== b	Not identical	TRUE if a is not equal to b, or they are not of the same type.
a < b	Less than	TRUE if a is strictly less than b.
a > b	Greater than	TRUE if a is strictly greater than b.
a <= b	Less than or equal to	TRUE if a is less than or equal to b.
a >= b	Greater than or equal to	TRUE if a is greater than or equal to b.

Learn more about comparison of dynamic variables in the php manual.

1.8 Arrays

Array manipulation in Zephir provides a way to use PHP arrays. An array is an implementation of a hash table.

1.8.1 Creating Arrays

An array is created enclosing their elements in square brackets:

```
//Creating an empty array
let elements = [];

//Creating an array with elements
let elements = [1, 3, 4];

//Creating an array with elements of different types
let elements = ["first", 2, true];

//A multidimensional array
let elements = [[0, 1], [4, 5], [2, 3]];
```

As PHP, hashes or dictionaries are supported:

```
//Creating a hash with string keys
let elements = ["foo": "bar", "bar": "foo"];

//Creating a hash with numeric keys
let elements = [4: "bar", 8: "foo"];
```

```
//Creating a hash with mixed string and numeric keys
let elements = [4: "bar", 8: "foo"];
```

1.8.2 Updating arrays

Arrays are updated in the same way as PHP using square brackets:

```
//Updating an array with a string key
let elements["foo"] = "bar";

//Updating an array with a numeric key
let elements[0] = "bar";
```

1.8.3 Appending elements

Elements can be appended at the end of the array as follows:

```
//Append an element to the array
let elements[] = "bar";
```

1.8.4 Reading elements from arrays

Is possible to read array elements as follows:

```
//Getting an element using the string key "foo"
let foo = elements["foo"];

//Getting an element using the numeric key 0
let foo = elements[0];
```

1.9 Classes and Objects

Zephir promotes object-oriented programming, this is why you can only export methods and classes in extensions, also you will see that most of the time, runtime errors raise exceptions instead of fatal errors or warnings.

1.9.1 Classes

Every Zephir file must implement a class or an interface (and just once). A class structure is very similar to a PHP class:

```
namespace Test;

/**
 * This is a sample class
 */
class MyClass
{

}
```

1.9.2 Implementing Methods

The "function" keyword introduces a method. Methods implements the usual visibility modifiers available in PHP, explicity set a visibility modifier is mandatory in Zephir:

```
namespace Test;

class MyClass
{

    public function myPublicMethod()
    {
        // ...
    }

    protected function myProtectedMethod()
    {
        // ...
    }

    private function myPrivateMethod()
    {
        // ...
    }
}
```

Methods can receive required and optional parameters:

```
namespace Test;

class MyClass
{

    /**
     * All parameters are required
     */
    public function doSum1(a, b)
    {
        return a + b;
    }

    /**
     * Just 'a' is required, 'b' is optional and it has a default value
     */
    public function doSum2(a, b=3)
    {
        return a + b;
    }

    /**
     * Both parameters are optional
     */
    public function doSum3(a=1, b=2)
    {
        return a + b;
    }

    /**
     * Parameters are required and their values must be integer
     */
    public function doSum4(int a, int b)
    {
        return a + b;
    }

    /**
     * Static typed with default values
     */
    public function doSum4(int a=4, int b=2)
    {
        return a + b;
    }

}
```

Public Visibility

Methods marked as "public" are exported to the PHP extension, this means that public methods are visible to the PHP code as well to the extension itself.

Protected Visibility

Methods marked as "protected" are exported to the PHP extension, this means that protected methods are visible to the PHP code as well to the extension itself. However, protected methods can only be called in the scope of the class or in classes that inherit them.

Private Visibility

Methods marked as "private" are not exported to the PHP extension, this means that private methods are only visible to the class where they're implemented.

1.9.3 Implementing Properties

Class member variables are called "properties". By default, they act as PHP properties. Properties are exported to the PHP extension and are visibles from PHP code. Properties implement the usual visibility modifiers available in PHP, explicity set a visibility modifier is mandatory in Zephir:

```
namespace Test;

class MyClass
{

    public myProperty1;

    protected myProperty2;

    private myProperty3;

}
```

Within class methods non-static properties may be accessed by using -> (Object Operator): this->property (where property is the name of the property):

```
namespace Test;

class MyClass
{

    protected myProperty;

    public function setMyProperty(var myProperty)
    {
        let this->myProperty = myProperty;
    }

    public function getMyProperty()
    {
        return this->myProperty;
    }

}
```

Properties can have literal compatible default values. These values must be able to be evaluated at compile time and must not depend on run-time information in order to be evaluated:

```
namespace Test;

class MyClass
{

    protected myProperty1 = null;
    protected myProperty2 = false;
    protected myProperty3 = 2.0;
    protected myProperty4 = 5;
    protected myProperty5 = "my value";

}
```

1.9.4 Class Constants

Class may contain class constants that remain the same and unchangeable once the extension is compiled. Class constants are exported to the PHP extension allowing them to be used from PHP.

```
namespace Test;

class MyClass
{

    const MYCONSTANT1 = false;
    const MYCONSTANT2 = 1.0;

}
```

Class constants can be accessed using the class name and the static operator (::):

```
namespace Test;

class MyClass
{

    const MYCONSTANT1 = false;
    const MYCONSTANT2 = 1.0;

    public function someMethod()
    {
        return MyClass::MYCONSTANT1;
    }

}
```

1.9.5 Calling Methods

Methods can be called using the object operator (->) as in PHP:

```
namespace Test;

class MyClass
{

    protected function _someHiddenMethod(a, b)
    {
        return a - b;
    }

    public function someMethod(c, d)
    {
        return this->_someHiddenMethod(c, d);
    }

}
```

Static methods must be called using the static operator (::):

```
namespace Test;

class MyClass
{

    protected static function _someHiddenMethod(a, b)
    {
        return a - b;
    }

    public static function someMethod(c, d)
    {
        return self::someHiddenMethod(c, d);
    }

}
```

You can call methods in a dynamic manner as follows:

```
namespace Test;

class MyClass
{
    protected adapter;

    public function setAdapter(var adapter)
    {
        let this->adapter = adapter;
    }

    public function someMethod(var methodName)
    {
        return $this->adapter->{methodName}();
    }

}
```

1.10 Control Structures

Zephir implements a simplified set of control structures present in similar languages like C, PHP etc.

1.10.1 Conditionals

If Statement

‘if’ statements evaluates an expression executing this trace if the evaluation is true. Braces are compulsory, an ‘if’ can have an optional ‘else’ clause, and multiple ‘if’/’else’ constructs can be chained together:

```
if false {
    echo "false?";
} else if true {
    echo "true!";
} else {
    echo "neither true nor false";
}
```

Parentheses in the evaluated expression are optional:

```
if a < 0 { return -1; } else if a > 0 { return 1; }
```

Switch Statement

A ‘switch’ evalutes an expression against a series of predefined values executing the corresponding ‘case’ block or falling back to the ‘default’ block case:

```
switch count(items) {
    case 1:
    case 3:
        echo 'odd items';
        break;
    case 2:
    case 4:
        echo 'even items';
        break;
    default:
        echo 'unknown items';
}
```

1.10.2 Loops

While Statement

‘while’ denotes a loop that iterates as long as its given condition evaluates as true:

```
let counter = 5;
while counter {
    let counter -= 1;
}
```

Loop Statement

In addition to ‘while’, ‘loop’ can be used to create infinite loops:

```
let n = 40;
loop {
    let n -= 2;
    if n % 5 == 0 { break; }
    echo x, "\n";
}
```

For Statement

A ‘for’ is a control structure that allows to traverse arrays or strings:

```
for item in ['a', 'b', 'c', 'd'] {
    echo item, "\n";
}
```

Keys in hashes can be obtained in the following way:

```
let items = ['a': 1, 'b': 2, 'c': 3, 'd': 4];

for key, value in items {
    echo key, ' ', value, "\n";
}
```

A ‘for’ loop can also be instructed to traverse an array or string in reverse order:

```
let items = [1, 2, 3, 4, 5];

for value in reverse items {
    echo value, "\n";
}
```

A ‘for’ can be used to traverse string variables:

```
string language = "zephir"; char ch;

for ch in language {
    echo "[", ch "]";
}
```

In reverse order:

```
string language = "zephir"; char ch;

for ch in reverse language {
    echo "[", ch "]";
}
```

A standard 'for' that traverses a range of integer values can be written as follows:

```
for i in range(1, 10) {
    echo i, "\n";
}
```

Break Statement

'break' ends execution of the current 'while', 'for' or 'loop' statements:

```
for item in ['a', 'b', 'c', 'd'] {
    if item == 'c' {
        break; // exit the for
    }
    echo item, "\n";
}
```

Continue Statement

'continue' is used within looping structures to skip the rest of the current loop iteration and continue execution at the condition evaluation and then the beginning of the next iteration.

```
let a = 5;
while a > 0 {
    let a--;
    if a == 3 {
        continue;
    }
    echo a, "\n";
}
```

1.10.3 Require

The 'require' statement dynamically includes and evaluates a specified PHP file. Note that included files via Zephir are interpreted by Zend Engine as normal PHP files. 'require' does not allow to include other zephir files in runtime.

```
if file_exists(path) {
    require path;
}
```

1.11 Calling Functions

PHP has a rich library of functions you can use in your extensions. To call a PHP function you can just refer its name in the Zephir code.

```
namespace MyLibrary;

class Encoder
{

    public function encode(var text)
    {
        if strlen(text) != 0 {
            return base64_encode(text);
        }
        return false;
    }

}
```

You can call also functions that are expected to exist in the PHP userland but they aren't built in with PHP:

```
namespace MyLibrary;

class Encoder
{

    public function encode(var text)
    {
        if strlen(text) != 0 {
            if function_exists("my_custom_encoder") {
                return my_custom_encoder($text);
            } else {
                return base64_encode(text);
            }
        }
        return false;
    }

}
```

Note that all PHP functions only receive and return dynamic variables, if you pass a static typed variable as parameter, some temporary dynamic variable will be used as bridge in order to call them:

```
namespace MyLibrary;

class Encoder
{

    public function encode(string text)
    {
        if strlen(text) != 0 {
            // an implicit dynamic variable is created to
            // pass the static typed 'text' as parameter
            return base64_encode(text);
        }
        return false;
    }

}
```

Similarly, functions return dynamic values that cannot be directly assigned to static variables without the appropriate cast:

```
namespace MyLibrary;

class Encoder
{

    public function encode(string text)
    {
        string encoded = "";

        if strlen(text) != 0 {
            let encoded = (string) base64_encode(text);
            return '(' . encoded . ')';
        }
        return false;
    }

}
```

Sometimes, we would need to call functions in a dynamic way, you can call them as follows:

```
namespace MyLibrary;

class Encoder
{

    public function encode(var callback, string text)
    {
        return {callback}(text);
    }

}
```

1.12 Installation

PHP extensions require a slightly different installation method to a traditional php-based library or framework. You can either download a binary package for the system of your choice or build it from the sources.

1.13 License

1.13.1 The ZephirLang License 1.0

Copyright (c) 2013 ZephirLang Team. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, is permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- The name "ZephirLang" must not be used to endorse or promote products derived from this software without prior written permission. For written permission, please contact team@zephir-lang.com.

CHAPTER 2

Other Formats

- PDF

Index