Zephir Documentation

Release 0.4.2

Zephir Team

September 30, 2014

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 management 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.

Getting 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 running it. 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.

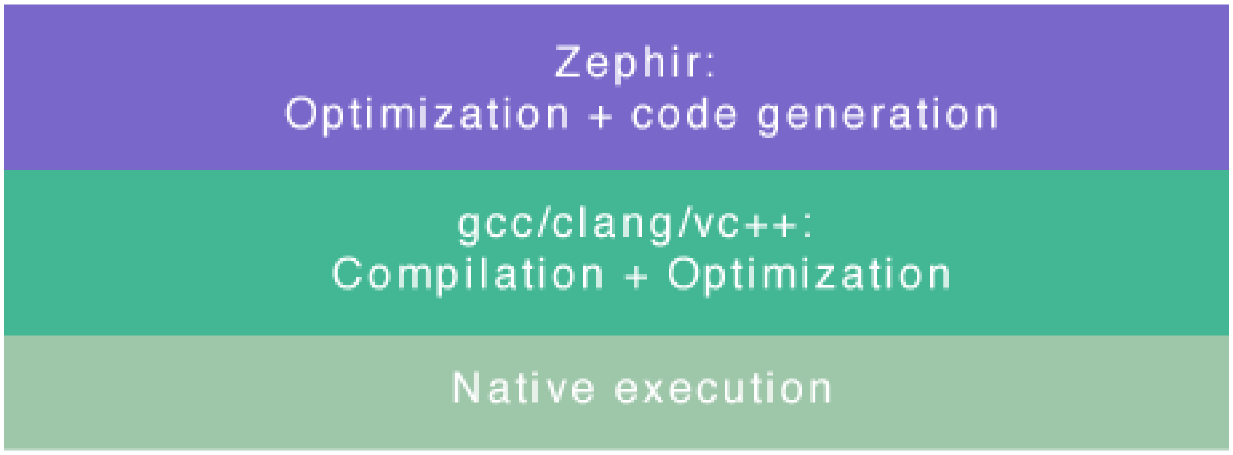
Zephir is both statically and dynamically typed allowing you to take advantage from both sides where possible.

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 Code Protection

In some circumstances, the compilation does not significantly improve performance, this may be because the bottleneck is located in the I/O bound of the application (quite likely) rather than compute/memory bound. However, compiling code could also bring some level of intelectual protection to your application. With Zephir, producing native binaries, you also get the ability to hide the code to users or customers.

1.2.7 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 opportunities 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!" sample, in Zephir this introductory example showcases some important features of this language.

Code in Zephir must be placed in classes. This language 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 uses Zephir must know or even understand 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 a 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 variable in the 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.

In contrast with PHP, you are not required to put a dollar sign ($) in front of variable names.

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 immutable, this means that Zephir expects that most variables stay unchanged. Variables that maintain their initial value can be optimized down by the compiler to static constants. When the variable value 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 dynamic 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 Installation

To install Zephir, please follow these steps:

1.4.1 Prerequisites

To build a PHP extension and use Zephir you need the following requirements:

- gcc >= 4.x/clang >= 3.x
- re2c 0.13 or later
- gnu make 3.81 or later
- autoconf 2.31 or later
- automake 1.14 or later
- libpcre3
- php development headers and tools

If you're using Ubuntu, you can install the required packages this way:

```
$ sudo apt-get update
$ sudo apt-get install git gcc make re2c php5 php5-json php5-dev libpcre3-dev
```

Since Zephir is written in PHP you need to have installed a recent version of PHP and it must be available in your console:

```
$ php -v
PHP 5.5.7 (cli) (built: Dec 14 2013 00:44:43)
Copyright (c) 1997-2013 The PHP Group
Zend Engine v2.5.0, Copyright (c) 1998-2013 Zend Technologies
```

Also, make sure you have also the PHP development libraries installed along with your PHP installation:

```
$ phpize -v
Configuring for:
PHP Api Version:         20121113
Zend Module Api No:      20121212
Zend Extension Api No:   220121212
```

You don't have to necessarely see the exact above output but it's important that these commands are available to start developing with Zephir.

1.4.2 Installing Zephir

Json-C is required by the Zephir parser to process the code, you can install it this way:

```
$ git submodule update --init
$ cd json-c
$ sh autogen.sh
$ ./configure
$ make && sudo make install
```

The Zephir compiler currently must be cloned from Github:

```
$ git clone https://github.com/phalcon/zephir
```

Run the Zephir installer (this compiles/creates the parser):

```
$ cd zephir
$ ./install -c
```

1.4.3 Testing Installation

Check if Zephir is available from any directory by executing:

```
$ zephir help
```

1.5 Tutorial

Zephir and this book are intended for PHP developers which want 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.

1.5.1 Checking Installation

If you have successfully installed Zephir, you must be able to execute the following command in your console:

```
$ zephir help
```

If everything is well, you should see the following help in your screen:

```
 _____              __    _
/__  /  ___  ____  / /_  (_)____
  / /  / _ \/ __ \/ __ \/ / ___/
 / /__/  __/ /_/ / / / / / /
/____/\___/ .___/_/ /_/_/_/
         /_/

Zephir version 0.4.5a

Usage:
    command [options]

Available commands:
    build               Generate/Compile/Install a Zephir extension
    clean               Cleans the generated object files in compilation
    compile             Compile a Zephir extension
    full-clean          Cleans the generated object files in compilation
    generate            Generates C code from the Zephir code
    help                Displays this help
    init [namespace]    Initializes a Zephir extension
    install             Installs the extension (requires root password)
    version             Shows Zephir version
```

1.5.2 Extension Skeleton

The first thing we have to do is generate an extension skeleton, this will provide to our extension the basic structure we need to start working. In our case, we're going to create an extension called "utils":

```
$ zephir init utils
```

After this, a directory called "utils" is created on the current working directory:

```
utils/
   ext/
   utils/
```

The directory "ext/" (inside utils) contains the code that is going to be used by the compiler to produce the extension. The another directory created is "utils", this directory has the same as our extension. We will place Zephir code in this directory.

We need to change the working directory to "utils" to start compiling our code:

```
$ cd utils
$ ls
ext/ utils/ config.json
```

The directory listing will also show us a file called "config.json", this file contains configuration settings we can use to alter the behavior of Zephir and/or this extension.

1.5.3 Adding our first class

Zephir is designed to generate object-oriented extensions. To start developing functionality we need to add our first class to the extension.

As in many languages/tools, the first thing we want to do is see a "hello world" generated by Zephir and check that everything is well. So our first class will be called "Utils\Greeting" and it contains a method printing "hello world!".

The code for this class must be placed in "utils/utils/greeting.zep":

```
namespace Utils;

class Greeting
{

    public static function say()
    {
        echo "hello world!";
    }

}
```

Now, we need to tell Zephir that our project must be compiled and the extension generated:

```
$ zephir build
```

Initially, and only for the first time, a number of internal commands are executed producing the necessary code and configurations to export this class to the PHP extension, if everything goes well you will see the following message at the end of the output:

```
...
Extension installed!
Add extension=utils.so to your php.ini
Don't forget to restart your web server
```

At the above step, it's likely that you would need to supply your root password in order to install the extension. Finally, the extension must be added to the php.ini in order to be loaded by PHP. This is achieved by adding the initialization directive: extension=utils.so to it.

1.5.4 Initial Testing

Now that the extension was added to your php.ini, check whether the extension is being loaded properly by executing the following:

```
$ php -m
[PHP Modules]
Core
date
libxml
pcre
Reflection
session
SPL
standard
tokenizer
utils
xdebug
xml
```

Extension "utils" must be part of the output indicating that the extension was loaded correctly. Now, let's see our "hello world" directly executed by PHP. To accomplish this, you can create a simple PHP file calling the static method we have just created:

```php
<?php

echo Utils\Greeting::say(), "\n";
```

Congratulations!, you have your first extension running on PHP.

1.5.5 A useful class

The "hello world" class was fine to check if our enviroment was right, now, let's create some more useful classes.

The first useful class we are going to add to this extension will provide filtering facilities to users. This class is called "Utils\Filter" and its code must be placed in "utils/utils/filter.zep":

A basic skeleton to this class is the following:

```
namespace Utils;

class Filter
{

}
```

The class contains filtering methods that help users to filter unwanted caracters from strings. The first method is called "alpha" and its purpose is to filter only those characters that are ascii basic letters. To begin, we are just going to traverse the string printing every byte to the standard output:

```
namespace Utils;

class Filter
```

```
{

    public function alpha(string str)
    {
        char ch;

        for ch in str {
            echo ch, "\n";
        }

    }

}
```

When invoking this method:

```
<?php

$f = new Utils\Filter();
$f->alpha("hello");
```

You will see:

```
h
e
l
l
o
```

Checking every character in the string is straightforward, we now just could create another string with the right filtered characters:

```
class Filter
{

    public function alpha(string str) -> string
    {
        char ch; string filtered = "";

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

        return filtered;
    }
}
```

The complete method can be tested as before:

```
<?php

$f = new Utils\Filter();
echo $f->alpha("!he#02l3'121lo."); // prints "hello"
```

In the following screencast you can watch how to create the extension explained in this tutorial:

1.5.6 Conclusion

This is a very simple tutorial and as you can see, it's easy to start building extensions using Zephir. We invite you to continue reading the manual so that you can discover additional features offered by Zephir!

1.6 Basic Syntax

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

1.6.1 Organizing Code in Files and Namespaces

In PHP, you can place the 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 Exception extends \Exception
{

}
```

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

1.6.2 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";
```

1.6.3 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!.

If a docblock is not located where is expected, the compiler will throw an exception.

1.6.4 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;
```

1.6.5 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;
    }
}
```

1.6.6 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"];
```

1.6.7 Local Symbol Table

Every method or context in PHP has a symbol table that allows to write variables in a very dynamic 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. If you want to create a variable in the current PHP symbol table, you can use the following syntax:

```
//Set variable $name in PHP
let {"name"} = "hello";

//Set variable $price in PHP
let name = "price";
let {name} = 10.2;
```

1.7 Types

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

1.7.1 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 check 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";
```

1.7.2 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.
array	An structure that can be used as hash, map, dictionary, collection, stack, etc.

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'
let ch = 'Z'; // char literals must be enclosed in simple quotes
```

String

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

When a variable is declared string it never changes its type:

```
string a;

let a = "",
    a = "hello", //string literals must be enclosed in double quotes
    a = 'A', // converted to string "A"
    a = null; // automatically casted to ""
```

1.8 Operators

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

1.8.1 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

1.8.2 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.

Example:

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

1.8.3 Logical Operators

The following operators are supported:

Operation	Example
And	a && b
Or	a || b
Not	!a

Example:

```
if a && b || !c {
    return -1;
}
return 1;
```

1.8.4 Bitwise Operators

The following operators are supported:

Operation	Example
And	a & b
Or (inclusive or)	a | b
Xor (exclusive or)	a ^ b
Not	~a
Shift left	a << b
Shift right	a >> b

Example:

```
if a & SOME_FLAG {
    echo "has some flag";
}
```

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

1.8.5 Ternary Operator

Zephir supports the ternary operator available in C or PHP:

```
let b = a == 1 ? "x" : "y"; // b is assigned with "x" if a is equals to 1 otherwise "y" is assigned
```

1.8.6 Special Operators

The following operators are supported:

Empty

This operator allows to check whether an expression is empty. ‘Empty’ means the expression is null, is an empty string or an empty array:

```
let someVar = "";
if empty someVar {
    echo "is empty!";
}

let someVar = "hello";
if !empty someVar {
    echo "is not empty!";
}
```

Isset

This operator checks whether a property or index has been defined in an array or object:

```
let someArray = ["a": 1, "b": 2, "c": 3];
if isset someArray["b"] { // check if the array has an index "b"
    echo "yes, it has an index 'b'\n";
}
```

Using 'isset' as return expression:

```
return isset this->{someProperty};
```

Note that 'isset' in Zephir works more like PHP's function array_key_exists, 'isset' in Zephir returns true even if the array index or property is null.

Fetch

'Fetch' is an operator that reduce a common operation in PHP into a single instruction:

```
<?php

if (isset($myArray[$key])) {
    $value = $myArray[$key];
    echo $value;
}
```

In Zephir, you can write the same code as:

```
if fetch value, myArray[key] {
    echo value;
}
```

'Fetch' only returns true if the 'key' is a valid item in the array, only in that case, 'value' is populated.

Typeof

This operator check variable type. 'Typeof' could works like comparison operator:

```
if (typeof str == "string") { // or !=
    echo str;
}
```

Also typeof can works like PHP function 'gettype'.

```
return typeof str;
```

Be careful, if you want to check is object 'callable' you allways have to use 'typeof' as comparison operator.

Type Hints

Zephir always tries to check whether an object implements methods and properties called/accessed on a variable that is inferred to be an object:

```
let o = new MyObject();

// Zephir checks if "myMethod" is implemented on MyObject
o->myMethod();
```

However, due to the dynamism inherited from PHP, sometimes it is not easy to know the class of an object so Zephir can not produce errors reports effectively. A type hint tells the compiler which class is related to a dynamic variable allowing the compiler to perform more compilation checks:

```
// Tell the compiler that "o"
// is an instance of class MyClass
let o = <MyClass> this->_myObject;
o->myMethod();
```

Branch Prediction Hints

What is branch prediction? Check this article out. In environments where performance is very important, it may be useful to introduce these hints.

Consider the following example:

```
let allPaths = [];
for path in this->_paths {
    if path->isAllowed() == false {
        throw new App\Exception("error!!");
    } else {
        let allPaths[] = path;
    }
}
```

The authors of the above code, know in advance that the condition that throws the exception is unlikely to happen. This means that 99.9% of the time, our method executes that condition, but it is probably never evaluated as true. For the processor, this could be hard to know, so we could introduce a hint there:

```
let allPaths = [];
for path in this->_paths {
    if unlikely path->isAllowed() == false {
        throw new App\Exception("error!!");
    } else {
        let allPaths[] = path;
    }
}
```

1.9 Arrays

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

1.9.1 Declaring Array Variables

Array variables can be declared using the keywords ‘var’ or ‘array’:

```
var a = []; // array variable, its type can be changed
array b = []; // array variable, its type cannot be changed across execution
```

1.9.2 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", "foo": 8];
```

1.9.3 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";

//Updating multi-dimensional array
let elements[0]["foo"] = "bar";
let elements["foo"][0] = "bar";
```

1.9.4 Appending elements

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

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

1.9.5 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];
```

1.10 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.

1.10.1 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
{

}
```

Class Modifiers

The following class modifiers are supported:

Final: If a class has this modifier it cannot be extended

```
namespace Test;

/**
 * This class cannot be extended by another class
 */
final class MyClass
{

}
```

Abstract: If a class has this modifier it cannot be instantiated

```
namespace Test;

/**
 * This class cannot be instantiated
 */
abstract class MyClass
{
}
```

1.10.2 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;
    }

}
```

Supported Visibilities

- Public: 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: 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: 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.

Supported Modifiers

- Deprecated: Methods marked as "deprecated" throwing an E_DEPRECATED error when they are called.

Getter/Setter shortcuts

Like in C#, you can use get/set/toString shortcuts in Zephir, this feature allows to easily write setters and getters for properties without explictly implementing those methods as such.

For example, without shortcuts we could find code like:

```
namespace Test;

class MyClass
{
    protected myProperty;

    protected someProperty = 10;

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

    public function setSomeProperty(someProperty)
    {
        this->someProperty = someProperty;
    }

    public function getSomeProperty()
    {
        return this->someProperty;
    }

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

}
```

You can write the same code using shortcuts as follows:

```
namespace App;

class MyClass
{
    protected myProperty {
        set, get, toString
    };

    protected someProperty = 10 {
        set, get
    };

}
```

When the code is compiled those methods are exported as real methods but you don't have to write them one by one.

Return Type Hints

Methods in classes and interfaces can have return type hints, these will provide useful extra information to the compiler to inform you about errors in your application. Consider the following example:

```
namespace App;

class MyClass
{
    public function getSomeData() -> string
    {
        // this will throw a compiler exception
        // since the returned value (boolean) does not match
        // the expected returned type string
        return false;
    }
    public function getSomeOther() -> <App\MyInterface>
    {
        // this will throw a compiler exception
        // if the returned object does not implement
        // the expected interface App\MyInterface
        return new App\MyObject;
    }

    public function process()
    {
        var myObject;

        // the type-hint will tell the compiler that
        // myObject is an instance of a class
        // that implement App\MyInterface
        let myObject = this->getSomeOther();

        // the compiler will check if App\MyInterface
        // implements a method called "someMethod"
        echo myObject->someMethod();
    }

}
```

A method can have more than one return type. When multiple types are defined, the operator | must be used to separate those types.

```
namespace App;

class MyClass
{
    public function getSomeData(a) -> string|bool
    {
        if a == false {
            return false;
        }
        return "error";
    }

}
```

Return Type: Void

Methods can also be marked as 'void'. This means that a method is not allowed to return any data:

```
public function setConnection(connection) -> void
{
    let this->_connection = connection;
}
```

Why is this useful? Because the compiler can detect if the program is expecting a returning value from these methods and produce a compiler exception:

```
let myDb = db->setConnection(connection);
myDb->execute("SELECT * FROM robots"); // this will produce an exception
```

Strict/Flexible Parameter Data-Types

In Zephir, you can specify the data type of each parameter of a method. By default, these data-types are flexible, this means that if a value with wrong (but compatible) data-type is passed, Zephir will try to transparently convert it to the expected one:

```
public function filterText(string text, boolean escape=false)
{
    //...
}
```

Above method will work with the following calls:

```
<?php

$o->filterText(1111, 1); // OK
$o->filterText("some text", null); // OK
$o->filterText(null, true); // OK
$o->filterText("some text", true); // OK
$o->filterText(array(1, 2, 3), true); // FAIL
```

However, passing a wrong type could be often lead to bugs, a bad use of a specific API would produce unexpected results. You can disallow the automatic conversion by setting the parameter with a strict data-type:

```
public function filterText(string! text, boolean escape=false)
{
    //...
}
```

Now, most of the calls with a wrong type will cause an exception due to the invalid data types passed:

```
<?php

$o->filterText(1111, 1); // FAIL
$o->filterText("some text", null); // OK
$o->filterText(null, true); // FAIL
$o->filterText("some text", true); // OK
$o->filterText(array(1, 2, 3), true); // FAIL
```

By specifying what parameters are strict and what must be flexible, a developer can set the specific behavior he/she really wants.

Read-Only Parameters

Using the keyword 'const' you can mark parameters as read-only, this helps to respect const-correctness. Parameters marked with this attribute cannot be modified inside the method:

```
namespace App;

class MyClass
{
    // "a" is read-only
    public function getSomeData(const string a)
    {
        // this will throw a compiler exception
        let a = "hello";
    }
}
```

When a parameter is declared as read-only the compiler can make safe assumptions and perform further optimizations over these variables.

1.10.3 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";
}
```

Updating Properties

Properties can be updated by accesing them using the '->' operator:

```
let this->myProperty = 100;
```

Zephir checks that properties do exist when a program is accesing them, if a property is not declared you will get a compiler exception:

```
CompilerException: Property '_optionsx' is not defined on class 'App\MyClass' in /Users/scott/utils/a

      this->_optionsx = options;
      ------------^
```

If you want to avoid this compiler validation or just create a property dynamically, you can enclose the property name using string quotes:

```
let this->{"myProperty"} = 100;
```

You can also use a simple variable to update a property, the property name will be taken from the variable:

```
let someProperty = "myProperty";
let this->{someProperty} = 100;
```

Reading Properties

Properties can be read by accesing them using the '->' operator:

```
echo this->myProperty;
```

As when updating, properties can be dynamically read this way:

```
//Avoid compiler check or read a dynamic user defined property
echo this->{"myProperty"};

//Read using a variable name
let someProperty = "myProperty";
echo this->{someProperty}
```

1.10.4 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;
    }

}
```

1.10.5 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}();
    }

}
```

1.11 Built-In Methods

As mentioned before, Zephir promotes object-oriented programming, variables related to static types can be also handled as objects.

Compare these two methods:

```
public function binaryToHex(string! s) -> string
{
    var o = "", n; char ch;

    for ch in range(0, strlen(s)) {
        let n = sprintf("%X", ch);
        if strlen(n) < 2 {
            let o .= "0" . n;
        } else {
            let o .= n;
        }
    }
    return o;
}
```

And:

```
public function binaryToHex(string! s) -> string
{
    var o = "", n; char ch;

    for ch in range(0, s->length()) {
        let n = ch->toHex();
        if n->length() < 2 {
            let o .= "0" . n;
        } else {
            let o .= n;
        }
    }
    return o;
}
```

They both have the same functionality, but the second one uses object-oriented programming. Calling methods on static-typed variables does not have any impact on performance since Zephir internally transforms the code from the object-oriented version to the procedural version.

1.11.1 String

The following string built-in methods are available:

OO	Procedural	Description
s->length()	strlen(s)	Get string length
s->trim()	trim(s)	Strip whitespace (or other characters) from the beginning and end of a string
s->index("foo")	strpos(s, "foo")	Find the position of the first occurrence of a substring in a string
s->lower()	strtolower(s)	Make a string lowercase
s->upper()	strtoupper(s)	Make a string uppercase

1.11.2 Array

The following array built-in methods are available:

OO	Procedural	Description
a->join(" ")	join(" ", a)	Join array elements with a string
a->reverse()	array_reverse(a)	Return an array with elements in reverse order

1.11.3 Char

The following char built-in methods are available:

OO	Procedural
ch->toHex()	sprintf("%X", ch)

1.11.4 Integer

The following integer built-in methods are available:

OO	Procedural
i->abs()	abs(i)

1.12 Control Structures

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

1.12.1 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, 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 literal 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";
}
```

1.12.2 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";
}
```

1.12.3 Require

The 'require' statement dynamically includes and evaluates a specified PHP file. Note that files included 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;
}
```

1.12.4 Let

'Let' statement is used to mutate variables, properties and arrays. Variables are by default inmutable and this instruction makes them mutable:

```
let name = "Tony";             // simple variable
let this->name = "Tony";       // object property
let data["name"] = "Tony";     // array index
let self::_name = "Tony";      // static property
```

Also this instruction must be used to increment/decrement variables:

```
let number++;             // increment simple variable
let number--;             // decrement simple variable
let this->number++;       // increment object property
let this->number--;       // decrement object property
```

Multiple mutations can be performed in a single 'let' operation:

```
let price = 1.00, realPrice = price, status = false;
```

1.13 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 a parameter, some temporary dynamic variable will be used as a 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);
    }

}
```

1.14 Custom optimizers

Most common functions in Zephir use internal optimizers. An 'optimizer' works like an interceptor for function calls. An 'optimizer' replaces the call for the function in the PHP userland by direct C-calls which are faster and have a lower overhead improving performance.

To create an optimizer you have to create a class in the 'optimizers' directory, the following convention must be used:

Function in Zephir	Optimizer Class Name	Optimizer Path	Function in C
calculate_pi	CalculatePiOptimizer	optimizers/CalculatePiOptimizer.php	my_calculate_pi

This is the basic structure for an 'optimizer':

```
<?php

namespace Zephir\Optimizers\FunctionCall;

use Zephir\Call;
use Zephir\CompilerException;
use Zephir\CompilationContext;
use Zephir\Optimizers\OptimizerAbstract;

class CalculatePiOptimizer extends OptimizerAbstract
{

    public function optimize(array $expression, Call $call, CompilationContext $context)
    {
        //...
    }
}
```

Implementation of optimizers highly depends on the kind of code you want to generate. In our example, we're going to replace the call to this function by a call to a c-function. In Zephir, the code used to call this function is:

```
let pi = calculate_pi(1000);
```

So, the optimizer will expect just one parameter, we have to validate that to avoid problems later:

```
<?php

public function optimize(array $expression, Call $call, CompilationContext $context)
{

    if (!isset($expression['parameters'])) {
        throw new CompilerException("'calculate_pi' requires one parameter", $expression);
    }

    if (count($expression['parameters']) > 1) {
        throw new CompilerException("'calculate_pi' requires one parameter", $expression);
    }

    //...
}
```

There are functions that are just called and they don't return any value, our function returns a value that is the calculated PI value. So we need to be aware that the type of the variable used to received this calculated value is OK:

```
<?php

public function optimize(array $expression, Call $call, CompilationContext $context)
{

    if (!isset($expression['parameters'])) {
        throw new CompilerException("'calculate_pi' requires one parameter", $expression);
    }

    if (count($expression['parameters']) > 1) {
        throw new CompilerException("'calculate_pi' requires one parameter", $expression);
    }

    /**
     * Process the expected symbol to be returned
     */
    $call->processExpectedReturn($context);

    $symbolVariable = $call->getSymbolVariable();
    if (!$symbolVariable->isDouble()) {
        throw new CompilerException("Calculated PI values only can be stored in double variables", $e
    }

    //...
}
```

We're checking if the value returned will be stored in a variable type 'double', if not a compiler exception is thrown.

The next thing we need to do is process the parameters passed to the function:

```php
<?php

$resolvedParams = $call->getReadOnlyResolvedParams($expression['parameters'], $context, $expression);
```

As a good practice with Zephir is important to create functions that don't modify their parameters, if you are changing the parameters passed, Zephir will need to allocate memory for constants passed and you have to use getResolvedParams instead of getReadOnlyResolvedParams.

Code returned by these methods is valid C-code that can be used in the code printer to generate the c-function call:

```php
<?php

//Generate the C-code
return new CompiledExpression('double', 'calculate_pi( ' . $resolvedParams[0] . ')', $expression);
```

All optimizers must return a CompiledExpression instance, this will tell the compiler the type returned by the code and its related C-code.

The complete optimizer code is:

```php
<?php

namespace Zephir\Optimizers\FunctionCall;

use Zephir\Call;
use Zephir\CompilerException;
use Zephir\CompilationContext;
use Zephir\CompiledExpression;
use Zephir\Optimizers\OptimizerAbstract;

class CalculatePiOptimizer extends OptimizerAbstract
{

    public function optimize(array $expression, Call $call, CompilationContext $context)
    {

        if (!isset($expression['parameters'])) {
            throw new CompilerException("'calculate_pi' requires one parameter", $expression);
        }

        if (count($expression['parameters']) > 1) {
            throw new CompilerException("'calculate_pi' requires one parameter", $expression);
        }

        /**
         * Process the expected symbol to be returned
         */
        $call->processExpectedReturn($context);

        $symbolVariable = $call->getSymbolVariable();
        if (!$symbolVariable->isDouble()) {
            throw new CompilerException("Calculated PI values only can be stored in double variables'
        }

        $resolvedParams = $call->getReadOnlyResolvedParams($expression['parameters'], $context, $expr

        return new CompiledExpression('double', 'my_calculate_pi(' . $resolvedParams[0] . ')', $expre
    }
}
```

The code that implements the function "my_calculate_pi" is written in C and must be compiled along with the extension.

This code must be placed in the ext/ directory where you find appropiate, just check that those files do not conflict with the files generated by Zephir.

This file must contain the Zend Engine headers and C implementation of the function:

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

#include "php.h"
#include "php_ext.h"

double my_calculate_pi(zval *accuracy) {
    return 0.0;
}
```

This file must be added at a special section in the *config.json* file:

```
"extra-sources": [
    "utils/pi.c"
]
```

Check the complete source code of this example *here <https://github.com/phalcon/zephir-samples/tree/master/ext-optimizers>*.

1.15 Configuration File

Every Zephir extension has a configuration file called config.json. This file is read by Zephir everytime you build or generate the extension and it allows the developer to modify the extension or compiler behavior.

This file use JSON as configuration format which is very known and friendly:

```
{
    "namespace": "test",
    "name": "Test Extension",
    "description": "My amazing extension",
    "author": "Tony Hawk",
    "version": "1.2.0"
}
```

Settings defined in this file override any factory default setting provided by Zephir.

The following settings are supported:

1.15.1 namespace

The namespace of the extension, it must be a simple identifier respecting the regular expression: [a-zA-Z0-9_]+:

```
{
    "namespace": "test"
}
```

1.15.2 name

Extension name, only can contain ascii characters:

```
{
    "namespace": "test"
}
```

1.15.3 description

Extension description, any text describing your extension:

```
{
    "description": "My amazing extension"
}
```

1.15.4 author

Company, developer, institution, etc. that have developed the extension:

```
{
    "author": "Tony Hawk"
}
```

1.15.5 version

Extension version, must follow the regular expression: [0-9]+.[0-9]+.[0-9]+:

```
{
    "version": "1.2.0"
}
```

1.15.6 warnings

Compiler warnings enabled or disabled in the current project:

```
{
    "warnings": {
        "unused-variable": true,
        "unused-variable-external": false,
        "possible-wrong-parameter": true,
        "possible-wrong-parameter-undefined": false,
        "nonexistent-function": true,
        "nonexistent-class": true
    }
}
```

1.15.7 optimizations

Compiler optimizations enabled or disabled in the current project:

```
{
    "optimizations": {
        "static-type-inference": true,
        "static-type-inference-second-pass": true,
        "local-context-pass": false
    }
}
```

1.15.8 globals

Extension globals available. Check the *extension globals* chapter for more information.

```
{
    "globals": {
        "my_setting_1": {
            "type": "bool",
            "default": true
        },
        "my_setting_2": {
            "type": "int",
            "default": 10
        }
}
```

1.15.9 info

phpinfo() sections. Check the *phpinfo()* chapter for more information.

```
{
    "info": [
        {
            "header": ["Directive", "Value"],
            "rows": [
                ["setting1", "value1"],
                ["setting2", "value2"]
            ]
        }
    ]
}
```

1.16 Extension Globals

PHP extensions provide a way to define globals within an extension. Reading/Writing globals should be faster than any other global mechanisms (like static members). You can use extension globals to set up configuration options that change the behavior of your library.

In Zephir, extension globals are restricted to simple scalar types like int/bool/double/char, etc. Complex types such as string/arrays/objects/resources aren't allowed here.

You can enable extension globals by adding the following structure to your config.json:

```
{
    //...
    "globals": {
        "allow_some_feature": {
            "type": "bool",
            "default": true
        },
        "number_times": {
            "type": "int",
            "default": 10
        },
        "some_component.my_setting_1": {
            "type": "bool",
            "default": true
        },
        "some_component.my_setting_2": {
            "type": "int",
            "default": 100
        }
    }
}
```

Each global has the following structure:

```
"<global-name>": {
    "type": "<some-valid-type>",
    "default": <some-compatible-default-value>
}
```

Compound globals has the following structure:

```
"<namespace>.<global-name>": {
    "type": "<some-valid-type>",
    "default": <some-compatible-default-value>
}
```

Inside any method you can read/write extension globals using the built-in functions global_get/global_set:

```
global_set("allow_some_feature", true);
let someFeature = global_get("allow_some_feature");
```

If you want to change these globals from PHP a good option is enable a method aimed at this:

```
namespace Test;

class MyOptions
{

    public static function setOptions(array options)
    {
        boolean someOption, anotherOption;

        if fetch someOption, options["some_option"] {
            globals_set("some_option", someOption);
        }

        if fetch anotherOption, options["another_option"] {
            globals_set("another_option", anotherOption);
        }
    }
}
```

Extension globals cannot be dinamically accessed since the C-code generated by global_get/global_set optimizers must be resolved at compilation time:

```
let myOption = "someOption";

//will throw a compiler exception
let someOption = globals_get(myOption);
```

1.17 Phpinfo() sections

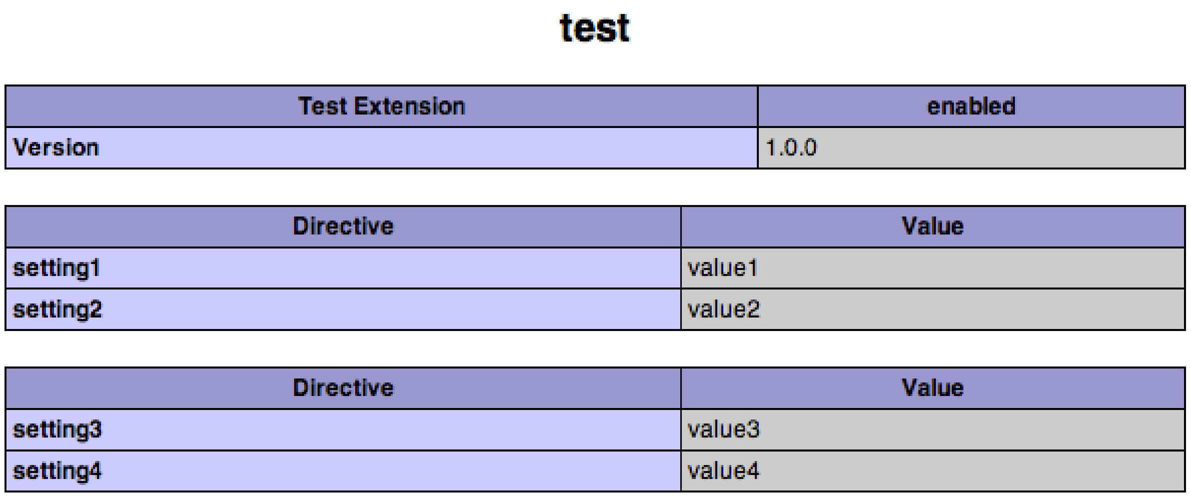
Like most extensions, Zephir extensions are able to show information at the phpinfo() output. These information is usually related to directives, enviroment data, etc.

By default, every Zephir extension automatically adds a basic table to the phpinfo() output showing the extension version.

You can add more directives by adding the following configuration to the config.json file:

```
"info": [
    {
        "header": ["Directive", "Value"],
        "rows": [
            ["setting1", "value1"],
            ["setting2", "value2"]
        ]
    },
    {
        "header": ["Directive", "Value"],
        "rows": [
            ["setting3", "value3"],
            ["setting4", "value4"]
        ]
    }
]
```

These information will be shown as follows:


test

Test Extension	enabled
Version	1.0.0

Directive	Value
setting1	value1
setting2	value2

Directive	Value
setting3	value3
setting4	value4

1.18 Optimizations

Because the code in Zephir is sometimes very high-level, a C compiler might not be in the ability to optimize this code enough.

Zephir, thanks to its AOT compiler, is able to optimize the code at compile time potentially improving its execution time or reducing the memory required by the program.

You can enable optimizations by passing its name prefixed by -f:

```
zephir -fstatic-type-inference -flocal-context-pass
```

Warnings can be disabled by passing its name prefixed by -fno-:

```
zephir -fno-static-type-inference -fno-call-gatherer-pass
```

The following optimizations are supported:

1.18.1 static-type-inference

This compilation pass is very important, since it looks for dynamic variables that can potentially transformed into static/primitive types which are better optimized by the underlying compiler.

The following code use a set dynamic variables to perform some mathematical calculations:

```
public function someCalculations(var a, var b)
{
        var i = 0, t = 1;

        while i < 100 {
                if i % 3 == 0 {
                        continue;
                }
                let t += (a - i), i++;
        }

        return i + b;
}
```

Variables 'a', 'b' and 'i' are used all of them in mathematical operations and can thus be transformed into static variables taking advantage of other compilation passes. After this pass, the compiler automatically rewrites this code to:

```
public function someCalculations(int a, int b)
{
        int i = 0, t = 1;

        while i < 100 {
                if i % 3 == 0 {
                        continue;
                }
                let t += (a - i), i++;
        }

        return i + b;
}
```

By disabling this compilation pass, all variables will maintain the type which they were originally declared without optimization.

1.18.2 static-type-inference-second-pass

This enables a second type inference pass which improves the work done based on the data gathered by the first static type inference pass.

1.18.3 local-context-pass

This compilation pass moves variables that will be allocated in the heap to the stack.

1.18.4 constant-folding

Constant folding is the process of simplifying constant expressions at compile time. The following code is simplified when this optimization is enabled:

```
public function getValue()
{
        return (86400 * 30) / 12;
}
```

Is transformed into:

```
public function getValue()
{
        return 216000;
}
```

1.18.5 static-constant-class-folding

This optimization replaces values at class constants in compile time:

```
class MyClass
{

        const SOME_CONSTANT = 100;

        public function getValue()
        {
                return self::SOME_CONSTANT;
        }

}
```

Is transformed into:

```
class MyClass
{

        const SOME_CONSTANT = 100;

        public function getValue()
        {
                return 100;
        }

}
```

1.18.6 call-gatherer-pass

This pass counts how many times a function or method is called within the same method. This allow the compiler to introduce inline caches to avoid method or function lookups.

1.19 Compiler Warnings

The compiler raise warnings when it finds situations where the code can be improved or a potential error can be avoided.

Warnings can be enabled via command line parameters or can be added to the config.json to enable or disable them permanently:

You can enable warnings by passing its name prefixed by -w:

```
zephir -wunused-variable -wnonexistent-function
```

Warnings can be disabled by passing its name prefixed by -W:

```
zephir -Wunused-variable -Wnonexistent-function
```

The following warnings are supported:

1.19.1 unused-variable

Raised when a variable is declared but it is not used within a method. This warning is enabled by default.

```
public function some()
{
    var e; // declared but not used

    return false;
}
```

1.19.2 unused-variable-external

Raised when a parameter is declared but it is not used within a method.

```
public function sum(a, b, c) // c is not used
{
    return a + b;
}
```

1.19.3 possible-wrong-parameter-undefined

Raised when a method is called with a wrong type for a parameter:

```
public function some()
{
    return this->sum("a string", "another");  // wrong parameters passed
}

public function sum(int a, int b)
```

```
{
    return a + b;
}
```

1.19.4 nonexistent-function

Raised when a non-existent function at compile time is called:

```
public function some()
{
    someFunction(); // someFunction does not exist
}
```

1.19.5 nonexistent-class

Raised when a non-existent class is used at compile time:

```
public function some()
{
    var a;

    let a = new \MyClass(); // MyClass does not exist
}
```

1.19.6 non-valid-isset

Raised when the compiler detects that an 'isset' operation is being made on a non array or object value:

```
public function some()
{
    var b = 1.2;
    return isset b[0]; // variable integer 'b' used as array
}
```

1.19.7 non-array-update

Raised when the compiler detects that an array update operation is being made on a non array value:

```
public function some()
{
    var b = 1.2;
    let b[0] = true; // variable 'b' cannot be used as array
}
```

1.19.8 non-valid-objectupdate

Raised when the compiler detects that an object update operation is being made on a non object:

```
public function some()
{
    var b = 1.2;
    let b->name = true; // variable 'b' cannot be used as object
}
```

1.19.9 non-valid-fetch

Raised when the compiler detects that a 'fetch' operation is being made on a non array or object value:

```
public function some()
{
    var b = 1.2, a;
    fetch a, b[0]; // variable integer 'b' used as array
}
```

1.19.10 invalid-array-index

Raised when the compiler detects that an invalid array index is used:

```
public function some(var a)
{
    var b = [];
    let a[b] = true;
}
```

1.19.11 non-array-append

Raised when the compiler detects that an element is being tried to be appended to a non array variable:

```
public function some()
{
    var b = false;
    let b[] = "some value";
}
```

1.19.12 non-array-append

Raised when the compiler detects that an element is being tried to be appended to a non array variable:

```
public function some()
{
    var b = false;
    let b[] = "some value";
}
```

```
        'invalid-return-type'                 => true,
        'unreachable-code'                    => true,
        'nonexistant-constant'                => true,
        'not-supported-magic-constant'        => true,
        'non-valid-decrement'                 => true,
        'non-valid-increment'                 => true,
        'non-valid-clone'                     => true,
        'non-valid-new'                       => true,
        'non-array-access'                    => true,
        'invalid-reference'                   => true,
        'invalid-typeof-comparison'           => true,
        'conditional-initialization'          => true
```

1.20 License

CHAPTER 2

Other Formats

- PDF

Index