Zephir Documentation

Release 0.10.9

Zephir Team

May 13, 2018

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 their alphabetic characters:

```
namespace MyLibrary;

/**
 * Filter
 */
class Filter
{
    /**
     * Filters a string, returning its alpha characters
     *
     * @param string str
```

(continues on next page)

(continued from previous page)

```
     */
    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 many other applications.

These common components are libraries, frameworks, or a combination of the two. Once installed, frameworks rarely change, and 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, 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 present in almost 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 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 development down; you will need a bit more patience to compile your code before running it. On the other hand, interpretation tends to reduce code performance in favor of developer productivity. That said, in some cases, there is not any noticeable difference between the speed of interpreted and compiled code.

Zephir requires compilation of your code, but functionality is used from PHP, which is interpreted.

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

1.2.4 Statically Typed Versus Dynamically Typed Languages

Generally speaking, in a statically 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 this 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 one type, then be reassigned later to a value of an unrelated type. Javascript/PHP are examples of a dynamically typed languages:

```
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 attached to variables themselves to make decisions. In a dynamic language, fewer such clues are available for the optimizer, making optimization choices harder.

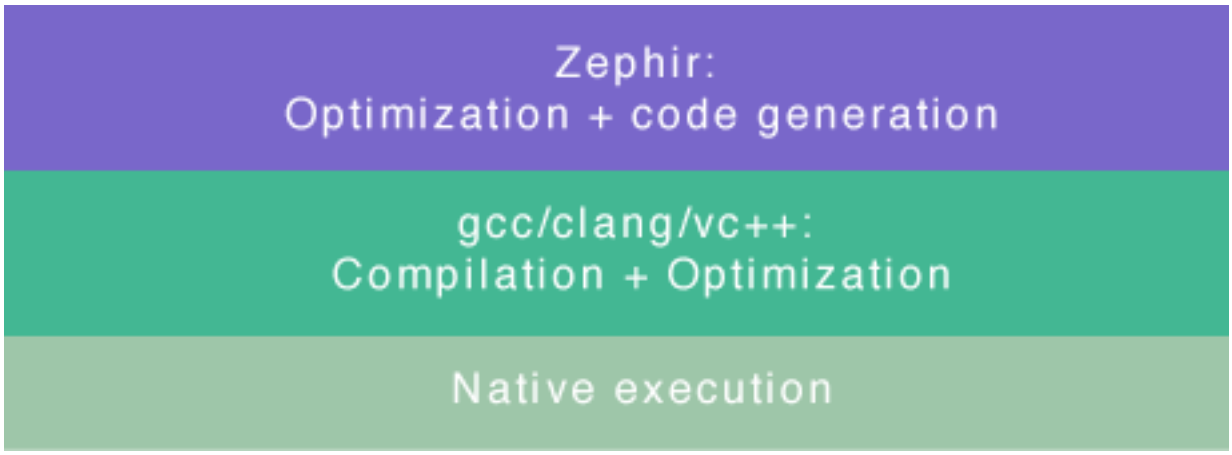
While recent advancements in optimizations for dynamic languages are promising (like JIT compilation), they lag behind the state of the art for static languages. So, if you require very high performance, static languages are probably a safer choice.

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 of both approaches 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 implemented and matured a number of optimizations that improve the performance of compiled applications:

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

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 portion(s) 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 original code to users or customers.

1.2.6 Conclusion

Zephir was not created to replace PHP or C. Instead, we think it is a complement to them, allowing PHP developers to venture into code compilation and static typing. Zephir is an attempt to join good things from the C and PHP worlds, looking for opportunities to make 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 dynamically/statically typed, and some of its features will be familiar to PHP developers.

The name Zephir is a contraction of the words Z(end) E(ngine)/PH(P)/I(nte)r(mediate). 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 the language.

Code in Zephir must be placed in classes. The language is intended to create object-oriented libraries/frameworks, so code outside of a class is not allowed. Additionally, 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 will 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, that will provide a clearer understanding of what's going on internally when working with Zephir.

1.3.2 A Taste of Zephir

In the following examples, we'll describe just enough of the details to 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. There 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 warn you about mistyped variables, or about the use of variables out of 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, the 'int' variables are statically typed integer variables, that can only have integer values 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 will 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 dynamically typed like in PHP - they may contain values of different types. Functions from the PHP userland can be called in Zephir code. In the next example, the function 'count' is called, but the compiler can perform optimizations like avoiding 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 use them if you feel more comfortable doing so, but you aren't required to.

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

Since PHP only works with dynamic variables, methods always return dynamic variables. This means that if a statically 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, similarly to how PHP does it, so you don't need to allocate or free memory like in C.

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 php php-json php-dev libpcre3-dev
```

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

```
$ php -v
PHP 7.0.8 (cli) (built: Jun 26 2016 00:59:31) ( NTS )
Copyright (c) 1997-2016 The PHP Group
Zend Engine v3.0.0, Copyright (c) 1998-2016 Zend Technologies
        with Zend OPcache v7.0.8, Copyright (c) 1999-2016, by Zend Technologies
```

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

```
$ phpize -v
Configuring for:
PHP Api Version:         20151012
Zend Module Api No:      20151012
Zend Extension Api No:   320151012
```

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

1.4.2 Installing Zephir

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 who 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. We'll point out features in Zephir that are similar to these other languages, as well as many features that are new or different. If you are familiar with these specific languages, you'll pick up on these comparisons more quickly.

1.5.1 Checking Installation

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

```
$ zephir help
```

If everything is well, you should see the following help (or something very similar):

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

Zephir version 0.10.9a-dev

Usage:
    command [options]

Available commands:
    stubs               Generates extension PHP stubs
```

(continues on next page)

(continued from previous page)

```
    install             Installs the extension (requires root password)
    version             Shows the Zephir version
    compile             Compile a Zephir extension
    api [--theme-path=/path][--output-directory=/path][--theme-options={json}|/
↪path]Generates a HTML API
    init [namespace]    Initializes a Zephir extension
    fullclean           Cleans the generated object files in compilation
    builddev            Generate/Compile/Install a Zephir extension in development␣
↪mode
    clean               Cleans the generated object files in compilation
    generate            Generates C code from the Zephir code
    help                Displays this help
    build               Generate/Compile/Install a Zephir extension

Options:
    -f([a-z0-9\-]+)     Enables compiler optimizations
    -fno-([a-z0-9\-]+)  Disables compiler optimizations
    -w([a-z0-9\-]+)     Turns a warning on
    -W([a-z0-9\-]+)     Turns a warning off
```

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. Another directory created is "utils" - this directory has the same name as our extension. We will place Zephir code there.

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 the extension itself.

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 contain 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. (NOTE: You can also load it on the command line with `-d extension=utils.so`, but it will only load for that single request, so you'd need to include it every time you want to test your extension in the CLI. Adding the directive to the `php.ini` will ensure it is loaded for every request from then on.)

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" should 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

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

Congratulations!, you have your first extension running in 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 for 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. Now we'll 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 general concepts.

1.6.1 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 the classes and namespaces used. (This is similar to PSR-4 autoloading conventions, except it's enforced by the language itself.)

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 in 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 a 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 it 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 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 separate lines
var a;
var b;
var c;
```

Variables can optionally have an initial compatible default value:

```
//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 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 you to write variables in a very dynamic way:

```
<?php

$b = 100;
```

(continues on next page)

(continued from previous page)

```
$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 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 dynamically and statically typed. In this chapter we highlight the supported types and their behaviors.

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 should 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 one or more digits, followed by a period (.), followed by one 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, Zephir uses the same IEEE 754 double precision format used by PHP, 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 or Go). 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 need to 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 static 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 a series of characters, where a character is the same as a byte.
array	A 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 detailed above, 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:

```
unsigned int a;

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. Assigning unsigned integers to standard (signed) integers may result in 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 with integers, values assigned to long variables are automatically casted to this type:

```
long a;

let a = 50,
    a = -70,
```

(continues on next page)

(continued from previous page)

```
    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; assigning unsigned longs to standard (signed) longs may result in 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 the basic character set (generally 8 bits). A 'char' variable can be used to store any character in a string:

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

let ch = name[2]; // stores 't'
let ch = 'Z'; // char literals must be enclosed in single 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 operators are similar to the ones in PHP, and also inherit 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 set to "x" if a is equal to 1, otherwise "y" is␣
↪assigned as the value
```

1.8.6 Special Operators

The following operators are supported:

Empty

This operator allows checking 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 a 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 reduces 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, and only in that case is 'value' populated.

Typeof

This operator checks a variable's type. 'typeof' can be used with a comparison operator:

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

It can also work like the PHP function 'gettype'.

```
return typeof str;
```

Be careful, if you want to check whether an object is 'callable', you always have to use 'typeof' as a comparison operator, not a function.

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't produce error 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();
```

These "type hints" are weak. This means the program does not check if the value is in fact an instance of the specified class, nor whether it implements the specified interface. If you want it to check this every time in execution, use a strict type:

```
// Always check if the property is an instance
// of MyClass before the assignment
let o = <MyClass!> this->_myObject;
o->myMethod();
```

Branch Prediction Hints

What is branch prediction? Check this article out or refer to the Wikipedia article. 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("Some error message here");
    } 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("Some error message here");
    } 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 by enclosing its 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

It 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 one). 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 implement the usual visibility modifiers available in PHP. Explicity setting 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;
    }

    /**
     * Only 'a' is required, 'b' is optional and it has a default value
     */
```

(continues on next page)

(continued from previous page)

```
    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;
    }
}
```

Optional nullable parameters

Zephir ensures that the value of a variable remains of the type the variable was declared as. This makes Zephir convert the null value to the closest approximate value:

```
public function foo(int a = null)
{
    echo a; // if "a" is not passed it prints 0
}

public function foo(boolean a = null)
{
    echo a; // if "a" is not passed it prints false
}

public function foo(string a = null)
{
    echo a; // if "a" is not passed it prints an empty string
}

public function foo(array a = null)
{
    var_dump(a); // if "a" is not passed it prints an empty array
}
```

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

- Static: Methods with this modifier can only be called in a static context (from the class, not an object).
- Final: If a method has this modifier it cannot be overriden.
- Deprecated: Methods marked as "deprecated" throw 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 you to easily write setters and getters for properties, without explictly implementing those methods as such.

For example, without shortcuts we would need code like:

```
namespace Test;

class MyClass
{
    protected myProperty;

    protected someProperty = 10;

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

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

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

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

    public function __toString()
    {
```

(continues on next page)

(continued from previous page)

```
        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 manually.

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
```

(continues on next page)

(continued from previous page)

```
        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 return value from these methods, and produce a compiler exception:

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

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 a 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 often lead to bugs. Improper 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 can 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 the same as PHP properties. Properties are exported to the PHP extension, and are visible from PHP code. Properties implement the usual visibility modifiers available in PHP, and explicity setting 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):

```
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 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/app/myclass.zep on line 62

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

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

Classes 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;
```

(continues on next page)

(continued from previous page)

```
    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)
    {
```

(continues on next page)

(continued from previous page)

```
        let this->adapter = adapter;
    }

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

Parameters by Name

Zephir supports calling method parameters by name or keyword arguments. Named parameters can be useful if you want to pass parameters in an arbitrary order, document the meaning of parameters, or specify parameters in a more elegant way.

Consider the following example. A class called "Image" has a method that receives four parameters:

```
namespace Test;

class Image
{
    public function chop(width = 600, height = 400, x = 0, y = 0)
    {
        //...
    }
}
```

Using the standard method calling approach:

```
i->chop(100); // width=100, height=400, x=0, y=0
i->chop(100, 50, 10, 20); // width=100, height=50, x=10, y=20
```

Using named parameters, you can:

```
i->chop(width: 100); // width=100, height=400, x=0, y=0
i->chop(height: 200); // width=600, height=200, x=0, y=0
i->chop(height: 200, width: 100); // width=100, height=200, x=0, y=0
i->chop(x: 20, y: 30); // width=600, height=400, x=20, y=30
```

When the compiler (at compile time) does not know the correct order of these parameters, they must be resolved at runtime. In this case, there could be a minimum additional extra overhead:

```
let i = new {someClass}();
i->chop(y: 30, x: 20);
```

1.11 Built-In Methods

As mentioned before, Zephir promotes object-oriented programming. Variables related to static types can also be 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->trimleft()	ltrim(s)	Strip whitespace (or other characters) from the beginning of a string
s->trimright()	rtrim(s)	Strip whitespace (or other characters) from the 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
s->lowerfirst()	lcfirst(s)	Make a string's first character lowercase
s->upperfirst()	ucfirst(s)	Make a string's first character uppercase
s->format()	sprintf(s, "%s", x)	Return a formatted string
s->md5()	md5(s)	Calculate the md5 hash of a string
s->sha1()	sha1(s)	Calculate the sha1 hash of a string

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->rev()	array_reverse(a)	Return an array with elements in reverse order
a->reversed()	array_reverse(a)	Return an array with elements in reverse order
a->diff()	array_diff(a)	Computes the difference of arrays
a->flip()	array_flip(a)	Exchanges all keys with their associated values in an array
a->walk()	array_walk(a)	Apply a user supplied function to every member of an array
a->hasKey()	array_key_exists(a)	Checks if the given key or index exists in the array
a->keys()	array_keys(a)	Return all the keys or a subset of the keys of an array
a->values()	array_values(a)	Return all the values of an array
a->split()	array_chunk(a)	Split an array into chunks
a->combine(b)	array_combine(a, b)	Creates an array by using one array for keys and another for its values
a->intersect(b)	array_intersect(a, b)	Computes the intersection of arrays
a->merge(b)	array_merge(a, b)	Merge one or more arrays
a->pad()	array_pad(a, b)	Pad array to the specified length with a value

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 evaluate an expression, executing the following block if the evaluation is true. Braces are required. 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";
    }
}
```

'elseif' clauses are also available:

```
if a > 100 {
    echo "to big";
} elseif a < 0 {
    echo "to small";
} elseif a == 50 {
    echo "perfect!";
} else {
    echo "ok";
}
```

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;
```

(continues on next page)

(continued from previous page)

```
    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 by providing a variable for both the key and value:

```
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' loop 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";
}
```

To avoid warnings about unused variables, you can use anonymous variables in 'for' statements, by replacing a variable name with the placeholder "_":

```
// Use the key but ignore the value
for key, _ in data {
    echo key, "\n";
}
```

Break Statement

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

```
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 Zephdr code to include other Zephir files at runtime.

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

1.12.4 Let

The 'let' statement is used to mutate variables, properties and arrays. Variables are by default immutable and this instruction makes them mutable for the duration of the statement:

```
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 Exceptions

Zephir implements exceptions at a very low level, providing similar behavior and functionality to PHP.

When an exception is thrown, a 'catch' block can be used to capture the exception and allow the developer to provide proper handling:

```
var e;
try {

    // exceptions can be thrown here
```

(continues on next page)

(continued from previous page)

```
    throw new \Exception("This is an exception");

} catch \Exception, e {

    // handle exception
    echo e->getMessage();
}
```

Zephir also provides a "silent" 'try' block, that simply ignores any exceptions produced within that block:

```
try {
    throw new \Exception("This is an exception");
}
```

If you don't need an exception variable when 'catch'ing, then you can safely not provide it:

```
try {

    // exceptions can be thrown here
    throw new \Exception("This is an exception");

} catch \Exception {

    // handle exception
    echo "An exception occur!";
}
```

A single 'catch' block can be used to catch multiple types of exception:

```
var e;
try {

    // exceptions can be thrown here
    throw new \Exception("This is an exception");

} catch \RuntimeException|\Exception, e {

    // handle exception
    echo e->getMessage();
}
```

Zephir allows you to throw literals or static typed variables as if they were the message of the exception:

```
throw "Test"; // throw new \Exception("Test");
throw 't'; // throw new \Exception((string) 't');
throw 123; // throw new \Exception((string) 123);
throw 123.123; // throw new \Exception((string) 123.123);
```

Zephir's exceptions provide the same methods to know where the exception happened that PHP's exceptions do. That is, Exception::getFile() and Exception::getLine() return the location in the Zephir code where the exception was thrown:

```
Exception: The static method 'someMethod' doesn't exist on model 'Robots'
File=phalcon/mvc/model.zep Line=4042
#0 /home/scott/test.php(64): Phalcon\Mvc\Model::__callStatic('someMethod', Array)
#1 /home/scott/test.php(64): Robots::someMethod()
#2 {main}
```

1.14 Calling Functions

PHP has a rich library of functions that you can use within your extensions. To call a PHP function you simply use it as normal within your Zephir code:

```
namespace MyLibrary;

class Encoder
{

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

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

```
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, a temporary dynamic variable will automatically be used as a bridge in order to call the function:

```
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, which cannot be directly assigned to static variables without the appropriate explciit 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;
    }
}
```

Zephir also provides a way for you to call functions dynamically, such as:

```
namespace MyLibrary;

class Encoder
{

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

1.15 Closures

You can use closures (a.k.a. anonymous functions) in Zephir; these are PHP compatible and can be returned to the PHP userland:

```
namespace MyLibrary;

class Functional
{

    public function map(array! data)
    {
        return function(number) {
            return number * number;
        };
    }
}
```

It also can be executed directly within Zephir, and passed as a parameter to other functions/methods:

```
namespace MyLibrary;

```

(continues on next page)

(continued from previous page)

```
class Functional
{

    public function map(array! data)
    {
        return data->map(function(number) {
            return number * number;
        });
    }
}
```

A short syntax is also available to define closures:

```
namespace MyLibrary;

class Functional
{

    public function map(array! data)
    {
        return data->map(number => number * number);
    }
}
```

1.16 Custom optimizers

Most common functions in Zephir use internal optimizers. An 'optimizer' works like an interceptor for function calls. An 'optimizer' replaces calls to a function normally defined 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 (you can configure this directory's name in `config.json`; see below). The following naming convention must be used:

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

Note that an optimizer is written in PHP, not Zephir. It is used during compilation to programmatically generate the appropriate C code for your extension to call. It is respensible for checking that arguments and return types match what the C function actually requires, preventing Zephir from generating invalid C code.

This is the basic structure for an 'optimizer':

```
<?php

namespace Zephir\Optimizers\FunctionCall;

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

class CalculatePiOptimizer extends OptimizerAbstract
{
```

(continues on next page)

(continued from previous page)

```
    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 don't return any value. Our function returns a value that is the calculated PI value. So we need to check that the type of the variable used to receive 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);
```

(continues on next page)

(continued from previous page)

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

    //...
}
```

We're checking if the value returned will be stored in a variable of 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

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

A good practice with Zephir is to create functions that don't modify their parameters. If you are changing the parameters passed, Zephir will need to allocate memory for them, 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

//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

namespace Zephir\Optimizers\FunctionCall;

use Zephir\Call;
use Zephir\CompilationContext;
use Zephir\CompiledExpression;
use Zephir\Compiler\CompilerException;
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);
        }
```

(continues on next page)

(continued from previous page)

```
        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", $expression);
        }

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

        return new CompiledExpression('double', 'my_calculate_pi(' .
↪$resolvedParams[0] . ')', $expression);
    }

}
```

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 wherever 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 the C implementation of the function:

```
#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"
]
```

Lastly you will have to specify where Zephir can find your optimizer by using the `optimizer-dirs` configuration option.

```
"optimizer-dirs": [
    "optimizers"
]
```

Check the complete source code of this example here.

1.17 Configuration File

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

This file uses JSON as its configuration format:

```
{
    "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:

1.17.1 namespace

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

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

1.17.2 extension-name

The base filename of the extension. It must follow the same rules as the `namespace` setting, which is used as a fallback in case this one isn't given.

```
{
    "extension-name": "test"
}
```

1.17.3 name

Extension name used in compiled C code - can only contain ascii characters:

```
{
    "name": "test"
}
```

1.17.4 description

Extension description - any text describing your extension:

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

1.17.5 author

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

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

1.17.6 version

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

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

1.17.7 warnings

Compiler warnings which should be 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
    }
}
```

1.17.8 optimizations

Compiler optimizations which should be enabled or disabled in the current project:

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

1.17.9 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
        }
    }
}
```

1.17.10 info

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

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

1.17.11 initializers

This setting lets you provide one or more C functions to be executed on certain extension lifecycle events - specifically, `GINIT` (`globals`), `MINIT` (`module`), and `RINIT` (`request`). Check the *lifecycle hooks* chapter for more information.

```
{
    "initializers": [
        {
            "globals": [
                {
                    "include": "my/awesome/library.h",
                    "code": "setup_globals_deps(TSRMLS_C)"
                }
            ],
            "module": [
                {
                    "include": "my/awesome/library.h",
                    "code": "setup_module_deps(TSRMLS_C)"
                }
            ],
            "request": [
                {
                    "include": "my/awesome/library.h",
                    "code": "some_c_function(TSRMLS_C)"
```

(continues on next page)

(continued from previous page)

```
                },
                {
                    "include": "my/awful/library.h",
                    "code": "some_other_c_function(TSRMLS_C)"
                }
            ]
        }
    ]
}
```

1.17.12 destructors

This setting lets you provide one or more C functions to be executed on certain extension lifecycle events - specifically, `RSHUTDOWN` (`request`), `PRSHUTDOWN` (`post-request`), `MSHUTDOWN` (`module`), and `GSHUTDOWN` (`globals`). Check the *lifecycle hooks* chapter for more information.

```
{
    "destructors": [
        {
            "request": [
                {
                    "include": "my/awesome/library.h",
                    "code": "c_function_for_shutting_down(TSRMLS_C)"
                },
                {
                    "include": "my/awful/library.h",
                    "code": "some_other_c_function_than_the_other_ones(TSRMLS_C)"
                }
            ],
            "post-request": [
                {
                    "include": "my/awesome/library.h",
                    "code": "c_function_for_cleaning_up_after_the_response_is_
↪sent(TSRMLS_C)"
                }
            ],
            "module": [
                {
                    "include": "my/awesome/library.h",
                    "code": "release_module_deps(TSRMLS_C)"
                }
            ],
            "globals": [
                {
                    "include": "my/awesome/library.h",
                    "code": "release_globals_deps(TSRMLS_C)"
                }
            ]
        }
    ]
}
```

1.17.13 extra-cflags

Any additional flags you want to add to the compilation process:

```
{
    "extra-cflags": "-I/usr/local/Cellar/libevent/2.0.21_1/include"
}
```

1.17.14 extra-libs

Any additional libraries you want to add to the compilation process:

```
{
    "extra-libs": "-L/usr/local/Cellar/libevent/2.0.21_1/lib -levent"
}
```

1.17.15 extra-sources

Any additional files you want to add to the compilation process - the search directory is relative to the `ext` folder of your project:

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

1.17.16 optimizer-dirs

The directories where your own optimizers can be found - the search directory is relative to the root folder of your project:

```
{
    "optimizer-dirs": [
        "optimizers"
    ]
}
```

1.17.17 constants-sources

To import just the constants in a C source file into your project, list the file's path in this setting:

```
{
    "constants-sources": [
        "utils/math_constants.h"
    ]
}
```

1.17.18 extra-classes

If you already have a PHP class implmented in C, you can include it directly in your extension by configuring it here:

```
{
    "extra-classes": [
        {
            "header": "utls/old_c_class/class.h",
            "source": "utils/old_c_class/class.c",
            "init": "old_c_class",
            "entry": "old_c_class_ce"
        }
    ]
}
```

1.17.19 external-dependencies

You can include a class from another namespace/extension directly in your own extension by configuring it here:

```
{
    "external-dependencies": {
        "My\\Awesome": "my/awesome/class.zep",
        "My\\Awful": "my/awful/class.zep"
    }
}
```

1.17.20 package-dependencies

Declare library dependencies (version constraints will be checked by `pkg-config`, and can use one of the operators =, >=, <=, or *):

```
{
    "package-dependencies": {
        "openssl": "*",
        "libpng": ">= 0.1.0",
        "protobuf": "<= 2.6.1"
    }
}
```

1.17.21 requires

Allows you to list other extensions as required to build/use your own:

```
{
    "requires": {
        "extensions": [
            "igbinary",
            "session"
        ]
    }
}
```

1.17.22 prototype-dir

Allows you to provide prototype files describing other extensions required to build your own, so they don't necessarily need to be installed during the build phase:

```
{
    "prototype-dir": {
        "igbinary": "prototypes",
        "session": "prototypes"
    }
}
```

1.17.23 stubs

This setting allows adjusting the way IDE documentation stubs are generated. `path` sets where the stubs should be created, while `stubs-run-after-generate` sets whether to automatically (re)build the stubs when your code is compiled to C:

```
{
    "stubs": {
        "path": "ide/%version%/%namespace%/",
        "stubs-run-after-generate": false
    }
}
```

1.17.24 api

Used to configure the automatically generated HTML documentation for your extension. `path` specifies where to create the documentation relative to the project root. `base-url` is used to generate a `sitemap.xml` file for your documentation. `theme` is used to set the theme used for the generated documentation (via the `name` setting), and any options the theme supports passing (via the `options` setting). Finally, `theme-directories` is used to provide additional search paths for finding your desired theme.:

```
{
    "api": {
        "path": "doc/%version%",
        "base-url": "http://example.local/api/",
        "theme": {
            "name"   : "zephir",
            "options": {
                "github":           null,
                "analytics":        null,
                "main_color":       "#3E6496",
                "link_color":       "#3E6496",
                "link_hover_color": "#5F9AE7"
            }
        },
        "theme-directories": [
            "my/api/themes"
        ]
    }
}
```

1.17.25 backend

Provides a way to configure the Zend Engine backend used by your extension. At the moment, only the `templatepath`, which lets you select between `ZendEngine2` and `ZendEngine3`, is supported:

```
{
    "backend": {
        "templatepath": "ZendEngine3"
    }
}
```

1.17.26 extra

Contains extra settings that also can be passed, as is, on the command line. Currently, that's `export-clases` (generate headers for accessing your classes from other C code), and `indent` (select between using `tabs` or `spaces` to indent code in generated files):

```
{
    "extra": {
        "export-classes": true,
        "indent": "tabs"
    }
}
```

1.17.27 silent

Suppresses most/all output from `zephir` commands (same as `-w`):

```
{
    "silent": false
}
```

1.17.28 verbose

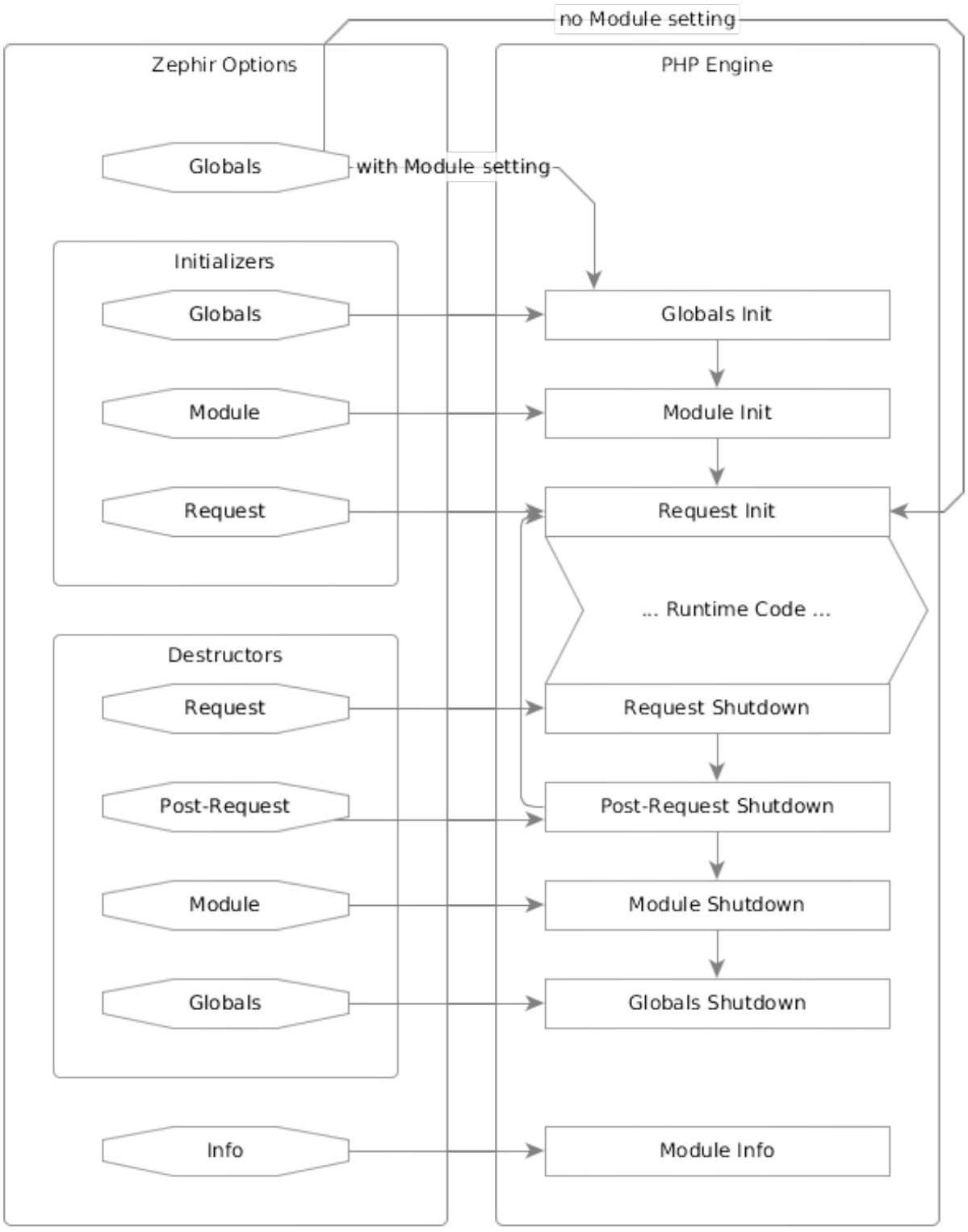
Displays more detail in error messages from exceptions generated by `zephir` commands (can also enable with `-v`, or disable with `-V`):

```
{
    "verbose": false
}
```

1.18 Lifecycle hooks

PHP provides several lifecycle events, which extensions can use to perform common initialization or shutdown tasks. Normally, Zephir's own hooks into these events will cover all the setup and teardown your extension will need, but if you find that you need to do something more, there are a few options you can use to pass your own code into these same hooks.

Consider the following diagram:


no Module setting
Zephir Options
PHP Engine
Globals
with Module setting
Initializers
Globals
Module
Request
Globals Init
Module Init
Request Init
... Runtime Code ...
Destructors
Request
Post-Request
Module
Globals
Request Shutdown
Post-Request Shutdown
Module Shutdown
Globals Shutdown
Info
Module Info

Lifecycle hooks are registered in the `config.json` file. As you can see in the diagram above, there are four types of lifecycle hooks - `globals`, `initializers`, `destructors`, and `info`. Each of these has its own corresponding root-level setting in the configuration, and both *globals* and *info* have their own chapters. This chapter covers the other two settings.

You can register an `include` and a `code` for each group's supported `INIT` and `SHUTDOWN` events. The `code` can be whatever you need/want, but a single function call per hook is recommended, both for clarity in the config, and to keep code in other files as much as possible. You can safely omit either the `include` or `code` option, but duplicate `include` options are removed, so you can safely repeat those, instead. It is recommended to provide both values, to make it easier to see which includes are needed for which hooks, and make it easier to add and remove hooks individually.

1.18.1 initializers

The `initializers` block looks something like this:

```
{
    "initializers": [
        {
            "globals": [
                {
                    "include": "my/awesome/library.h",
                    "code": "setup_globals_deps(TSRMLS_C)"
                }
            ],
            "module": [
                {
                    "include": "my/awesome/library.h",
                    "code": "setup_module_deps(TSRMLS_C)"
                }
            ],
            "request": [
                {
                    "include": "my/awesome/library.h",
                    "code": "some_c_function(TSRMLS_C)"
                },
                {
                    "include": "my/awful/library.h",
                    "code": "some_other_c_function(TSRMLS_C)"
                }
            ]
        }
    ]
}
```

1.18.2 destructors

And the `destructors` block like this:

```
{
    "destructors": [
        {
            "request": [
                {
                    "include": "my/awesome/library.h",
```

(continues on next page)

(continued from previous page)

```
                    "code": "c_function_for_shutting_down(TSRMLS_C)"
                },
                {
                    "include": "my/awful/library.h",
                    "code": "some_other_c_function_than_the_other_ones(TSRMLS_C)"
                }
            ],
            "post-request": [
                {
                    "include": "my/awesome/library.h",
                    "code": "c_function_for_cleaning_up_after_the_response_is_
↪sent(TSRMLS_C)"
                }
            ],
            "module": [
                {
                    "include": "my/awesome/library.h",
                    "code": "release_module_deps(TSRMLS_C)"
                }
            ],
            "globals": [
                {
                    "include": "my/awesome/library.h",
                    "code": "release_globals_deps(TSRMLS_C)"
                }
            ]
        }
    ]
}
```

1.19 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/array/object/resource 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,
            "module": true
        },
        "number_times": {
            "type": "int",
            "default": 10
        },
        "some_component.my_setting_1": {
```

(continues on next page)

(continued from previous page)

```
            "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 (namespaced) globals have the following structure:

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

The optional `module` key, if present, places that global's initialization process into the module-wide `GINIT` lifecycle event, which just means it will only be set up once per PHP process, rather than being reinitialized for every request, which is the default:

```
{
    //...
    "globals": {
        "allow_some_feature": {  // set up only once, at startup
            "type": "bool",
            "default": true,
            "module": true
        },
        "number_times": {        // set up at the start of each request
            "type": "int",
            "default": 10
        }
    }
}
```

Inside any method, you can read/write extension globals using the built-in functions `globals_get`/`globals_set`:

```
globals_set("allow_some_feature", true);
let someFeature = globals_get("allow_some_feature");
```

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

```
namespace Test;

class MyOptions
{

    public static function setOptions(array options)
```

(continues on next page)

(continued from previous page)

```
    {
        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 dynamically accessed, since the C code generated by the `globals_get`/`globals_set` optimizers must be resolved at compilation time:

```
let myOption = "someOption";

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

1.20 Phpinfo() sections

Like most extensions, Zephir extensions are able to show information in the phpinfo() output. This 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, and any INI options the extension supports.

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"]
        ]
    }
]
```

This 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

1.21 Static Analysis

Zephir's compiler provides static analysis of the compiled code. The idea behind this feature is to help the developer to find potential problems and avoid unexpected behaviors, well before runtime.

1.21.1 Conditional Unassigned Variables

Static Analysis of assignments tries to identify if a variable is used before it's assigned:

```
class Utils
{
        public function someMethod(b)
        {
                string a; char c;

                if b == 10 {
                        let a = "hello";
                }

                //a could be unitialized here
                for c in a {
                        echo c, PHP_EOL;
                }
        }
}
```

The above example illustrates a common situation. The variable "a" is assigned only when "b" is equal to 10, then it's required to use the value of this variable - but it could be uninitialized. Zephir detects this, automatically initializes the variable to an empty string, and generates a warning alerting the developer:

```
Warning: Variable 'a' was assigned for the first time in conditional branch,
consider initialize it in its declaration in
/home/scott/test/test/utils.zep on 21 [conditional-initialization]

        for c in a {
```

Finding such errors is sometimes tricky, however static analysis helps the programmer to find bugs in advance.

1.21.2 Dead Code Elimination

Zephir informs the developer about unreacheable branches in the code and performs dead code elimination, which means it gets rid of all that code from the generated binary, since it cannot be executed anyway:

```
class Utils
{
        public function someMethod(b)
        {
                if false {
                        // This is never executed
                        echo "hello";
                }
        }
}
```

1.22 Optimizations

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

Zephir, thanks to its AOT (ahead-of-time) 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 the name prefixed by -f:

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

Optimizaitons can be disabled by passing the name prefixed by -fno-:

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

With recent versions of zephir-parser, optimizations can be configured in the config file `config.json`.

The following optimizations are supported:

1.22.1 static-type-inference

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

The following code uses a set of 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++;
        }
```

(continues on next page)

(continued from previous page)

```
        return i + b;
}
```

Variables 'a', 'b', and 'i' are used exclusively 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 with which they were originally declared, without optimization.

1.22.2 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.

1.22.3 local-context-pass

This compilation pass moves variables that will be allocated in the heap to the stack. This optimization can reduce the number of memory indirections a program has to do.

1.22.4 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;
}
```

1.22.5 static-constant-class-folding

This optimization replaces values of 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;
        }
}
```

1.22.6 call-gatherer-pass

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

```
  class MyClass extends OtherClass
  {

          public function getValue()
          {
                  this->someMethod();
this->someMethod(); // This method is called faster
          }
  }
```

1.23 Compiler Warnings

The compiler raises 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 more permanently.

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

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

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

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

The following warnings are supported:

1.23.1 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;
}
```

1.23.2 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;
}
```

1.23.3 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;
}
```

1.23.4 nonexistent-function

Raised when a function is called that doesn't exist at compile time:

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

1.23.5 nonexistent-class

Raised when a class is used that doesn't exist at compile time:

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

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

1.23.6 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
}
```

1.23.7 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
}
```

1.23.8 non-valid-objectupdate

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

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

1.23.9 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
}
```

1.23.10 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;
}
```

1.23.11 non-array-append

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

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

1.24 License

CHAPTER 2

Other Formats

- PDF

Index

I