Php7: Guide and References to All The Changes Between Version 5.X and 7 of Php

PHP 7 was released on December 3rd, 2015. It comes with a number of new features, changes, and backwards compatibility breakages that are outlined below.

Performance

Features

- Combined Comparison Operator
- Null Coalesce Operator
- Scalar Type Declarations
- Return Type Declarations
- Anonymous Classes
- Unicode Codepoint Escape Syntax
- Closure `call()` Method
- Filtered `unserialize()`
- `IntlChar` Class
- Expectations
- Group `use` Declarations
- Generator Return Expressions
- Generator Delegation
- Integer Division with `intdiv()`
- `session_start()` Options
- `preg_replace_callback_array()` Function
- CSPRNG Functions
- Support for Array Constants in `define()`
- Reflection Additions

Changes

- Loosening Reserved Word Restrictions
- Uniform Variable Syntax
- Exceptions in the Engine
- Throwable Interface
- Integer Semantics
- JSON Extension Replaced with JSOND
- ZPP Failure on Overflow
- Fixes to `foreach()`'s Behaviour
- Changes to `list()`'s Behaviour
- Changes to Division by Zero Semantics
- Fixes to Custom Session Handler Return Values
- Deprecation of PHP 4-Style Constructors
- Removal of date.timezone Warning
- Removal of Alternative PHP Tags

- Removal of Multiple Default Blocks in Switch Statements
- Removal of Redefinition of Parameters with Duplicate Names
- Removal of Dead Server APIs
- Removal of Hex Support in Numerical Strings
- Removal of Deprecated Functionality
- Reclassification and Removal of E_STRICT Notices
- Deprecation of Salt Option for `password_hash()`
- Error on Invalid Octal Literals
- `substr()` Return Value Change

FAQ

- What happened to PHP 6?

Performance

Unarguably the greatest part about PHP 7 is the incredible performance boosts it provides to applications. This is a result of refactoring the Zend Engine to use more compact data structures and less heap allocations/deallocations.

The performance gains on real world applications will vary, though many applications seem to receive a ~100% performance boost - with lower memory consumption too!

The refactored codebase provides further opportunities for future optimisations as well (such as JIT compilation). So it looks like future PHP versions will continue to see performance enhancements too.

PHP 7 performance chart comparisons:

- Turbocharging the Web with PHP 7
- Benchmarks from Rasmus's Sydney Talk

Features

Combined Comparison Operator

The combined comparison operator (or spaceship operator) is a shorthand notation for performing three-way comparisons from two operands. It has an integer return value that can be either:

- a positive integer (if the left-hand operand is greater than the right-hand operand)
- 0 (if both operands are equal)
- a negative integer (if the right-hand operand is greater than the left-hand operand)

The operator has the same precedence as the equality operators (`==`, `!=`, `===`, `!==`) and has the exact same behaviour as the other loose comparison operators (`<`, `>=`, etc). It is also non-associative like them too, so chaining of the operands (like `1 <=> 2 <=> 3`) is not allowed.

```
   // compares strings lexically

var_dump('PHP' <=> 'Node'); // int(1)
```

```
// compares numbers by size

var_dump(123 <=> 456); // int(-1)



// compares corresponding array elements with one-another

    var_dump(['a', 'b'] <=> ['a', 'b']); // int(0)
```

Objects are not comparable, and so using them as operands with this operator will result in undefined behaviour.

RFC: Combined Comparison Operator

Null Coalesce Operator

The null coalesce operator (or isset ternary operator) is a shorthand notation for performing `isset()` checks in the ternary operator. This is a common thing to do in applications, and so a new syntax has been introduced for this exact purpose.

```
    // Pre PHP 7 code

$route = isset($_GET['route']) ? $_GET['route'] : 'index';



// PHP 7+ code

    $route = $_GET['route'] ?? 'index';
```

RFC: Null Coalesce Operator

Scalar Type Declarations

Scalar type declarations come in two flavours: **coercive** (default) and **strict**. The following types for parameters can now be enforced (either coercively or strictly): strings (`string`), integers (`int`), floating-point numbers (`float`), and booleans (`bool`). They augment the other types introduced in the PHP 5.x versions: class names, interfaces, `array` and `callable`.

```
    // Coercive mode

function sumOfInts(int ...$ints)

{

    return array_sum($ints);

}



    var_dump(sumOfInts(2, '3', 4.1)); // int(9)
```

To enable strict mode, a single `declare()` directive must be placed at the top of the file. This means that the strictness of typing for scalars is configured on a per-file basis. This directive not only affects the type declarations of parameters, but also a function's return type (see Return Type Declarations), built-in PHP functions, and functions from loaded extensions.

If the type-check fails, then a `TypeError` exception (see Exceptions in the Engine) is thrown. The only leniency present in strict typing is the automatic conversion of integers to floats (but not vice-versa) when an integer is provided in a float context.

```
declare(strict_types=1);


function multiply(float $x, float $y)

{

    return $x * $y;

}


function add(int $x, int $y)

{

    return $x + $y;

}


var_dump(multiply(2, 3.5)); // float(7)

var_dump(add('2', 3)); // Fatal error: Uncaught TypeError: Argument 1 passed
to add() must be of the type integer, string given...
```

Note that **only** the *invocation context* applies when the type-checking is performed. This means that the strict typing applies only to function/method calls, and not to the function/method definitions. In the above example, the two functions could have been declared in either a strict or coercive file, but so long as they're being called in a strict file, then the strict typing rules will apply.

BC Breaks

- Classes with names `int`, `string`, `float`, and `bool` are now forbidden.

RFC: Scalar Type Declarations

Return Type Declarations

Return type declarations enable for the return type of a function, method, or closure to be specified. The following return types are supported: `string`, `int`, `float`, `bool`, `array`, `callable`,

`self` (methods only), `parent` (methods only), `Closure`, the name of a class, and the name of an interface.

```
    function arraysSum(array ...$arrays): array

{

    return array_map(function(array $array): int {

        return array_sum($array);

    }, $arrays);

}


print_r(arraysSum([1,2,3], [4,5,6], [7,8,9]));

/* Output

Array

(

    [0] => 6

    [1] => 15

    [2] => 24

)

    */
```

With respect to subtyping, **invariance** has been chosen for return types. This simply means that when a method is either overridden in a subtyped class or implemented as defined in a contract, its return type must match exactly the method it is (re)implementing.

```
    class A {}

class B extends A {}


class C

{

    public function test() : A

    {

        return new A;

    }
```

```
}


class D extends C

{

    // overriding method C::test() : A

    public function test() : B // Fatal error due to variance mismatch

    {

        return new B;

    }

   }
```

The overriding method `D::test() : B` causes an `E_COMPILE_ERROR` because covariance is not allowed. In order for this to work, `D::test()` method must have a return type of `A`.

```
class A {}


interface SomeInterface

{

    public function test() : A;

}


class B implements SomeInterface

{

    public function test() : A // all good!

    {

        return null; // Fatal error: Uncaught TypeError: Return value of
B::test() must be an instance of A, null returned...

    }

   }
```

This time, the implemented method causes a `TypeError` exception (see Exceptions in the Engine) to be thrown when executed. This is because `null` is not a valid return type - only an instance of the class `A` can be returned.

RFC: Return Type Declarations

Anonymous Classes

Anonymous classes are useful when simple, one-off objects need to be created.

```
    // Pre PHP 7 code

class Logger

{

    public function log($msg)

    {

        echo $msg;

    }

}


$util->setLogger(new Logger());


// PHP 7+ code

$util->setLogger(new class {

    public function log($msg)

    {

        echo $msg;

    }

   });
```

They can pass arguments through to their constructors, extend other classes, implement interfaces, and use traits just like a normal class can:

```
    class SomeClass {}

interface SomeInterface {}

trait SomeTrait {}
```

```
var_dump(new class(10) extends SomeClass implements SomeInterface {
    private $num;

    public function __construct($num)
    {
        $this->num = $num;
    }

    use SomeTrait;
});

/** Output:
object(class@anonymous)#1 (1) {
  ["Command line code0x104c5b612":"class@anonymous":private]=>
  int(10)
}
   */
```

Nesting an anonymous class within another class does not give it access to any private or protected methods or properties of that outer class. In order to use the outer class' protected properties or methods, the anonymous class can extend the outer class. To use the private or protected properties of the outer class in the anonymous class, they must be passed through its constructor:

```
<?php

class Outer
{
    private $prop = 1;
    protected $prop2 = 2;

    protected function func1()
```

```
    {
        return 3;
    }

    public function func2()
    {
        return new class($this->prop) extends Outer {
            private $prop3;

            public function __construct($prop)
            {
                $this->prop3 = $prop;
            }

            public function func3()
            {
                return $this->prop2 + $this->prop3 + $this->func1();
            }
        };
    }
}

    echo (new Outer)->func2()->func3(); // 6
```

RFC: Anonymous Classes

Unicode Codepoint Escape Syntax

This enables a UTF-8 encoded unicode codepoint to be output in either a double-quoted string or a heredoc. Any valid codepoint is accepted, with leading `0`'s being optional.

```
    echo "\u{aa}"; // ª

echo "\u{0000aa}"; // ª (same as before but with optional leading 0's)
```

```
echo "\u{9999}"; // 香
```

RFC: Unicode Codepoint Escape Syntax

Closure call() Method

The new `call()` method for closures is used as a shorthand way of invoking a closure whilst binding an object scope to it. This creates more perfomant and compact code by removing the need to create an intermediate closure before invoking it.

```
class A {private $x = 1;}


// Pre PHP 7 code

$getXCB = function() {return $this->x;};

$getX = $getXCB->bindTo(new A, 'A'); // intermediate closure

echo $getX(); // 1


// PHP 7+ code

$getX = function() {return $this->x;};

echo $getX->call(new A); // 1
```

RFC: Closure::call

Filtered `unserialize()`

This feature seeks to provide better security when unserializing objects on untrusted data. It prevents possible code injections by enabling the developer to whitelist classes that can be unserialized.

```
// converts all objects into __PHP_Incomplete_Class object

$data = unserialize($foo, ["allowed_classes" => false]);


// converts all objects into __PHP_Incomplete_Class object except those of
MyClass and MyClass2

$data = unserialize($foo, ["allowed_classes" => ["MyClass", "MyClass2"]);


// default behaviour (same as omitting the second argument) that accepts all
classes

$data = unserialize($foo, ["allowed_classes" => true]);
```

RFC: Filtered unserialize()

`IntlChar` Class

The new `IntlChar` class seeks to expose additional ICU functionality. The class itself defines a number of static methods and constants that can be used to manipulate unicode characters.

```
printf('%x', IntlChar::CODEPOINT_MAX); // 10ffff

echo IntlChar::charName('@'); // COMMERCIAL AT

var_dump(IntlChar::ispunct('!')); // bool(true)
```

In order to use this class, the `Intl` extension must be installed.

BC Breaks

- Classes in the global namespace must not be called `IntlChar`.

RFC: IntlChar class

Expectations

Expectations are backwards compatible enhancement to the older `assert()` function. They enable for zero-cost assertions in production code, and provide the ability to throw custom exceptions on error.

The `assert()` function's prototype is as follows:

```
void assert (mixed $expression [, mixed $message]);
```

As with the old API, if `$expression` is a string, then it will be evaluated. If the first argument is falsy, then the assertion fails. The second argument can either be a plain string (causing an AssertionError to be triggered), or a custom exception object containing an error message.

```
ini_set('assert.exception', 1);

class CustomError extends AssertionError {}

assert(false, new CustomError('Some error message'));
```

With this feature comes two PHP.ini settings (along with their default values):

- zend.assertions = 1
- assert.exception = 0

zend.assertions has three values:

- **1** = generate and execute code (development mode)

- **0** = generate code and jump around at it at runtime
- **-1** = don't generate any code (zero-cost, production mode)

assert.exception means that an exception is thrown when an assertion fails. This is switched off by default to remain compatible with the old `assert()` function.

RFC: Expectations

Group `use` Declarations

This gives the ability to group multiple `use` declarations according to the parent namespace. This seeks to remove code verbosity when importing multiple classes, functions, or constants that come under the same namespace.

```
    // Pre PHP 7 code

use some\namespace\ClassA;

use some\namespace\ClassB;

use some\namespace\ClassC as C;


use function some\namespace\fn_a;

use function some\namespace\fn_b;

use function some\namespace\fn_c;


use const some\namespace\ConstA;

use const some\namespace\ConstB;

use const some\namespace\ConstC;


// PHP 7+ code

use some\namespace\{ClassA, ClassB, ClassC as C};

use function some\namespace\{fn_a, fn_b, fn_c};

    use const some\namespace\{ConstA, ConstB, ConstC};
```

RFC: Group use Declarations

Generator Return Expressions

This feature builds upon the generator functionality introduced into PHP 5.5. It enables for a `return` statement to be used within a generator to enable for a final *expression* to be returned (return by reference is not allowed). This value can be fetched using the new

`Generator::getReturn()` method, which may only be used once the generator has finishing yielding values.

```
    // IIFE syntax now possible - see the Uniform Variable Syntax subsection in
    the Changes section

$gen = (function() {

    yield 1;

    yield 2;



    return 3;

})();



foreach ($gen as $val) {

    echo $val, PHP_EOL;

}



echo $gen->getReturn(), PHP_EOL;



// output:

// 1

// 2

    // 3
```

> Being able to explicitly return a final value from a generator is a handy ability to have. This is because it enables for a final value to be returned by a generator (from perhaps some form of coroutine computation) that can be specifically handled by the client code executing the generator. This is far simpler than forcing the client code to firstly check whether the final value has been yielded, and then if so, to handle that value specifically.

RFC: Generator Return Expressions

Generator Delegation

Generator delegation builds upon the ability of being able to return expressions from generators. It does this by using an new syntax of `yield from <expr>`, where can be any `Traversable` object or array. This will be advanced until no longer valid, and then execution will continue in the calling generator. This feature enables `yield` statements to be broken down into smaller operations, thereby promoting cleaner code that has greater reusability.

```
    function gen()
{
    yield 1;
    yield 2;

    return yield from gen2();
}

function gen2()
{
    yield 3;

    return 4;
}

$gen = gen();

foreach ($gen as $val)
{
    echo $val, PHP_EOL;
}

echo $gen->getReturn();

// output
// 1
// 2
// 3
    // 4
```

RFC: Generator Delegation

Integer Division with `intdiv()`

The `intdiv()` function has been introduced to handle division where an integer is to be returned.

```
var_dump(intdiv(10, 3)); // int(3)
```

BC Breaks

- Functions in the global namespace must not be called `intdiv`.

RFC: intdiv()

`session_start()` Options

This feature gives the ability to pass in an array of options to the `session_start()` function. This is used to set session-based php.ini options:

```
session_start(['cache_limiter' => 'private']); // sets the
session.cache_limiter option to private
```

This feature also introduces a new php.ini setting (`session.lazy_write`) that is, by default, set to true and means that session data is only rewritten if it changes.

RFC: Introduce session_start() Options

`preg_replace_callback_array()` Function

This new function enables code to be written more cleanly when using the `preg_replace_callback()` function. Prior to PHP 7, callbacks that needed to be executed per regular expression required the callback function (second parameter of `preg_replace_callback()`) to be polluted with lots of branching (a hacky method at best).

Now, callbacks can be registered to each regular expression using an associative array, where the key is a regular expression and the value is a callback.

Function Signature:

```
string preg_replace_callback_array(array $regexesAndCallbacks, string
$input);
```

```
$tokenStream = []; // [tokenName, lexeme] pairs
```

```
$input = <<<'end'

$a = 3; // variable initialisation

end;
```

```
// Pre PHP 7 code
preg_replace_callback(
    [
        '~\$[a-z_][a-z\d_]*~i',
        '~=~',
        '~[\d]+~',
        '~;~',
        '~//.*~'
    ],
    function ($match) use (&$tokenStream) {
        if (strpos($match[0], '$') === 0) {
            $tokenStream[] = ['T_VARIABLE', $match[0]];
        } elseif (strpos($match[0], '=') === 0) {
            $tokenStream[] = ['T_ASSIGN', $match[0]];
        } elseif (ctype_digit($match[0])) {
            $tokenStream[] = ['T_NUM', $match[0]];
        } elseif (strpos($match[0], ';') === 0) {
            $tokenStream[] = ['T_TERMINATE_STMT', $match[0]];
        } elseif (strpos($match[0], '//') === 0) {
            $tokenStream[] = ['T_COMMENT', $match[0]];
        }
    },
    $input
);

// PHP 7+ code
preg_replace_callback_array(
    [
        '~\$[a-z_][a-z\d_]*~i' => function ($match) use (&$tokenStream) {
```

```
            $tokenStream[] = ['T_VARIABLE', $match[0]];
        },
        '~=~' => function ($match) use (&$tokenStream) {
            $tokenStream[] = ['T_ASSIGN', $match[0]];
        },
        '~[\d]+~' => function ($match) use (&$tokenStream) {
            $tokenStream[] = ['T_NUM', $match[0]];
        },
        '~;~' => function ($match) use (&$tokenStream) {
            $tokenStream[] = ['T_TERMINATE_STMT', $match[0]];
        },
        '~//.*~' => function ($match) use (&$tokenStream) {
            $tokenStream[] = ['T_COMMENT', $match[0]];
        }
    ],
    $input
);
```

BC Breaks

- Functions in the global namespace must not be called `preg_replace_callback_array`.

RFC: Add preg_replace_callback_array Function

CSPRNG Functions

This feature introduces two new functions for generating cryptographically secure integers and strings. They expose simple APIs and are platform-independent.

Function signatures:

```
string random_bytes(int length);
int random_int(int min, int max);
```

Both functions will emit an `Error` exception if a source of sufficient randomness cannot be found.

BC Breaks

- Functions in the global namespace must not be called `random_int` or `random_bytes`.

RFC: Easy User-land CSPRNG

Support for Array Constants in `define()`

The ability to define array constants was introduced in PHP 5.6 using the `const` keyword. This ability has now been applied to the `define()` function too:

```
define('ALLOWED_IMAGE_EXTENSIONS', ['jpg', 'jpeg', 'gif', 'png']);
```

RFC: no RFC available

Reflection Additions

Two new reflection classes have been introduced in PHP 7. The first is `ReflectionGenerator`, which is used for introspection on generators:

```
class ReflectionGenerator

{

 public __construct(Generator $gen)

 public array getTrace($options = DEBUG_BACKTRACE_PROVIDE_OBJECT)

 public int getExecutingLine(void)

 public string getExecutingFile(void)

 public ReflectionFunctionAbstract getFunction(void)

 public Object getThis(void)

 public Generator getExecutingGenerator(void)

}
```

The second is `ReflectionType` to better support the scalar and return type declaration features:

```
class ReflectionType

{

 public bool allowsNull(void)

 public bool isBuiltin(void)

 public string __toString(void)

}
```

Also, two new methods have been introduced into `ReflectionParameter`:

```
class ReflectionParameter

{

    // ...

    public bool hasType(void)

    public ReflectionType getType(void)

   }
```

As well as two new methods in `ReflectionFunctionAbstract`:

```
class ReflectionFunctionAbstract

{

    // ...

    public bool hasReturnType(void)

    public ReflectionType getReturnType(void)

   }
```

BC Breaks

- Classes in the global namespace must not be called `ReflectionGenerator` or `ReflectionType`.

RFC: no RFC available

Changes

Loosening Reserved Word Restrictions

Globally reserved words as property, constant, and method names within classes, interfaces, and traits are now allowed. This reduces the surface of BC breaks when new keywords are introduced and avoids naming restrictions on APIs.

This is particularly useful when creating internal DSLs with fluent interfaces:

```
// 'new', 'private', and 'for' were previously unusable

Project::new('Project Name')->private()->for('purpose here')->with('username here');
```

The only limitation is that the `class` keyword still cannot be used as a constant name, otherwise it would conflict with the class name resolution syntax (`ClassName::class`).

RFC: Context Sensitive Lexer

Uniform Variable Syntax

This change brings far greater orthogonality to the variable operators in PHP. It enables for a number of new combinations of operators that were previously disallowed, and so introduces new ways to achieve old operations in terser code.

```
   // nesting ::

$foo::$bar::$baz // access the property $baz of the $foo::$bar property



// nesting ()

foo()() // invoke the return of foo()



// operators on expressions enclosed in ()

   (function () {})() // IIFE syntax from JS
```

The ability to arbitrarily combine variable operators came from reversing the evaluation semantics of indirect variable, property, and method references. The new behaviour is more intuitive and always follows a left-to-right evaluation order:

```
                          // old meaning            // new meaning

$$foo['bar']['baz']     ${$foo['bar']['baz']}     ($$foo)['bar']['baz']

$foo->$bar['baz']       $foo->{$bar['baz']}       ($foo->$bar)['baz']

$foo->$bar['baz']()     $foo->{$bar['baz']}()     ($foo->$bar)['baz']()

   Foo::$bar['baz']()      Foo::{$bar['baz']}()      (Foo::$bar)['baz']()
```

BC Breaks

- Code that relied upon the old evaluation order must be rewritten to explicitly use that evaluation order with curly braces (see middle column of the above). This will make the code both forwards compatible with PHP 7.x and backwards compatible with PHP 5.x

RFC: Uniform Variable Syntax

Exceptions in the Engine

Exceptions in the engine converts many fatal and recoverable fatal errors into exceptions. This enables for graceful degradation of applications through custom error handling procedures. It also means that cleanup-driven features such as the `finally` clause and object destructors will now be executed. Furthermore, by using exceptions for application errors, stack traces will be produced for additional debugging information.

```
   function sum(float ...$numbers) : float

{
```

```
    return array_sum($numbers);
}


try {
    $total = sum(3, 4, null);
} catch (TypeError $typeErr) {
    // handle type error here
   }
```

The new exception hierarchy is as follows:

```
interface Throwable
    |- Exception implements Throwable
        |- ...
    |- Error implements Throwable
        |- TypeError extends Error
        |- ParseError extends Error
        |- AssertionError extends Error
        |- ArithmeticError extends Error
            |- DivisionByZeroError extends ArithmeticError
```

See the Throwable Interface subsection in the Changes section for more information on this new exception hierarchy.

BC Breaks

- Custom error handlers used for handling (and typically ignoring) recoverable fatal errors will not longer work since exceptions will now be thrown
- Parse errors occurring in `eval()`ed code will now become exceptions, requiring them to be wrapped in a `try...catch` block

RFC: Exceptions in the Engine

Throwable Interface

This change affects PHP's exception hierarchy due to the introduction of exceptions in the engine. Rather than placing fatal and recoverable fatal errors under the pre-existing `Exception` class

hierarchy, it was decided to implement a new hierarchy of exceptions to prevent PHP 5.x code from catching these new exceptions with catch-all (`catch (Exception $e)`) clauses.

The new exception hierarchy is as follows:

```
interface Throwable

 |- Exception implements Throwable

     |- ...

 |- Error implements Throwable

     |- TypeError extends Error

     |- ParseError extends Error

     |- AssertionError extends Error

     |- ArithmeticError extends Error

         |- DivisionByZeroError extends ArithmeticError
```

The `Throwable` interface is implemented by both `Exception` and `Error` base class hierarchies and defines the following contract:

```
interface Throwable
{
 final public string getMessage ( void )

 final public mixed getCode ( void )

 final public string getFile ( void )

 final public int getLine ( void )

 final public array getTrace ( void )

 final public string getTraceAsString ( void )

 public string __toString ( void )
}
```

`Throwable` cannot be implemented by user-defined classes - instead, a custom exception class should extend one of the pre-existing exceptions classes in PHP.

RFC: Throwable Interface

Integer Semantics

The semantics for some integer-based behaviour has changed in an effort to make them more intuitive and platform-independent. Here is a list of those changes:

- Casting `NAN` and `INF` to an integer will always result in 0
- Bitwise shifting by a negative number of bits is now disallowed (causes a bool(false) return and emits an E_WARNING)
- Left bitwise shifts by a number of bits beyond the bit width of an integer will always result in 0
- Right bitwise shifts by a number of bits beyond the bit width of an integer will always result in 0 or -1 (sign dependent)

BC Breaks

- Any reliance on the old semantics for the above will no longer work

RFC: Integer Semantics

JSON Extension Replaced with JSOND

The licensing of the old JSON extension was regarded as non-free, causing issues for many Linux-based distributions. The extension has since been replaced with JSOND and comes with some performance gains and backward compatibility breakages.

BC Breaks

- A number *must not* end in a decimal point (i.e. `34.` must be changed to either `34.0` or just `34`)
- The `e` exponent *must not* immediately follow the decimal point (i.e. `3.e3` must be changed to either `3.0e3` or just `3e3`)

RFC: Replace current json extension with jsond

ZPP Failure on Overflow

Coercion between floats to integers can occur when a float is passed to an internal function expecting an integer. If the float is too large to represent as an integer, then the value will be silently truncated (which may result in a loss of magnitude and sign). This can introduce hard-to-find bugs. This change therefore seeks to notify the developer when an implicit conversion from a float to an integer has occurred and failed by returning `null` and emitting an E_WARNING.

BC Breaks

- Code that once silently worked will now emit an E_WARNING and may fail if the result of the function invocation is directly passed to another function (since `null` will now be passed in).

RFC: ZPP Failure on Overflow

Fixes to `foreach()`'s Behaviour

PHP's `foreach()` loop had a number of strange edge-cases to it. These were all implementation-driven and caused a lot of undefined and inconsistent behaviour when iterating between copies and references of an array, when using iterator manipulators like `current()` and `reset()`, when modifying the array currently being iterated, and so on.

This change eliminates the undefined behaviour of these edge-cases and makes the semantics more predictable and intuitive.

`foreach()` by value on arrays

```
   $array = [1,2,3];

$array2 = &$array;


foreach($array as $val) {

    unset($array[1]); // modify array being iterated over

    echo "{$val} - ", current($array), PHP_EOL;

}


// Pre PHP 7 result

1 - 3

3 -


// PHP 7+ result

1 - 1

2 - 1

   3 - 1
```

When by-value semantics are used, the array being iterated over is now not modified in-place. `current()` also now has defined behaviour, where it will always begin at the start of the array.

`foreach()` by reference on arrays and objects and by value on objects

```
   $array = [1,2,3];


foreach($array as &$val) {
```

```
    echo "{$val} - ", current($array), PHP_EOL;
}


// Pre PHP 7 result
1 - 2
2 - 3
3 -


// PHP 7+ result
1 - 1
2 - 1
   3 - 1
```

The `current()` function is no longer affected by `foreach()`'s iteration on the array. Also, nested `foreach()`'s using by-reference semantics work independently from each other now:

```
$array = [1,2,3];


foreach($array as &$val) {
    echo $val, PHP_EOL;


    foreach ($array as &$val2) {
        unset($array[1]);
        echo $val, PHP_EOL;
    }
}


// Pre PHP 7 result
1
1
1
```

```
// PHP 7+ result

1

1

1

3

3

   3
```

BC Breaks

- Any reliance on the old (quirky and undocumented) semantics will no longer work.

RFC: Fix "foreach" behavior

Changes to `list()`'s Behaviour

The `list()` function was documented as not supporting strings, however in few cases strings could have been used:

```
   // array dereferencing

$str[0] = 'ab';

list($a, $b) = $str[0];

echo $a; // a

echo $b; // b


// object dereferencing

$obj = new StdClass();

$obj->prop = 'ab';

list($a, $b) = $obj->prop;

echo $a; // a

echo $b; // b


// function return

function func()
```

```
{
    return 'ab';
}


list($a, $b) = func();
var_dump($a, $b);
echo $a; // a
    echo $b; // b
```

This has now been changed making string usage with `list()` forbidden in all cases.

Also, empty `list()`'s are now a fatal error, and the order of assigning variables has been changed to left-to-right:

```
    $a = [1, 2];
list($a, $b) = $a;


// OLD: $a = 1, $b = 2
// NEW: $a = 1, $b = null + "Undefined index 1"


$b = [1, 2];
list($a, $b) = $b;


// OLD: $a = null + "Undefined index 0", $b = 2
    // NEW: $a = 1, $b = 2
```

BC Breaks

- Making `list()` equal to any non-direct string value is no longer possible. `null` will now be the value for the variable `$a` and `$b` in the above examples
- Invoking `list()` without any variables will cause a fatal error
- Reliance upon the old right-to-left assignment order will no longer work

RFC: Fix list() behavior inconsistency

RFC: Abstract syntax tree

Changes to Division by Zero Semantics

Prior to PHP 7, when a divisor was 0 for either the divide (/) or modulus (%) operators, an E_WARNING would be emitted and `false` would be returned. This was nonsensical for an arithmetic operation to return a boolean in some cases, and so the behaviour has been rectified in PHP 7.

The new behaviour causes the divide operator to return a float as either +INF, -INF, or NAN. The modulus operator E_WARNING has been removed and (alongside the new `intdiv()` function) will throw a `DivisionByZeroError` exception. In addition, the `intdiv()` function may also throw an `ArithmeticError` when valid integer arguments are supplied that cause an incorrect result (due to integer overflow).

```
    var_dump(3/0); // float(INF) + E_WARNING

var_dump(0/0); // float(NAN) + E_WARNING



var_dump(0%0); // DivisionByZeroError



    intdiv(PHP_INT_MIN, -1); // ArithmeticError
```

BC Breaks

- The divide operator will no longer return `false` (which could have been silently coerced to 0 in an arithmetic operation)
- The modulus operator will now throw an exception with a 0 divisor instead of returning `false`

RFC: No RFC available

Fixes to Custom Session Handler Return Values

When implementing custom session handlers, predicate functions from the `SessionHandlerInterface` that expect a `true` or `false` return value did not behave as expected. Due to an error in the previous implementation, only a `-1` return value was considered false - meaning that even if the boolean `false` was used to denote a failure, it was taken as a success:

```
    <?php


class FileSessionHandler implements SessionHandlerInterface

{

     private $savePath;
```

```
    function open($savePath, $sessionName)
    {
        return false; // always fail
    }

    function close(){return true;}

    function read($id){}

    function write($id, $data){}

    function destroy($id){}

    function gc($maxlifetime){}
}

session_set_save_handler(new FileSessionHandler());

    session_start(); // doesn't cause an error in pre PHP 7 code
```

Now, the above will fail with a fatal error. Having a `-1` return value will also continue to fail, whilst `0` and `true` will continue to mean success. Any other value returned will now cause a failure and emit an E_WARNING.

BC Breaks

- If boolean `false` is returned, it will actually fail now
- If anything other than a boolean, `0`, or `-1` is returned, it will fail and cause a warning to be emitted

RFC: Fix handling of custom session handler return values

Deprecation of PHP 4-Style Constructors

PHP 4 constructors were preserved in PHP 5 alongside the new `__construct()`. Now, PHP 4-style constructors are being deprecated in favour of having only a single method (`__construct()`) to be invoked on object creation. This is because the conditions upon whether

the PHP 4-style constructor was invoked caused additional cognitive overhead to developers that could also be confusing to the inexperienced.

For example, if the class is defined within a namespace or if an `__construct()` method existed, then a PHP 4-style constructor was recognised as a plain method. If it was defined above an `__construct()` method, then an E_STRICT notice would be emitted, but still recognised as a plain method.

Now in PHP 7, if the class is not in a namespace and there is no `__construct()` method present, the PHP 4-style constructor will be used as a constructor but an E_DEPRECATED will be emitted. In PHP 8, the PHP 4-style constructor will always be recognised as a plain method and the E_DEPRECATED notice will disappear.

BC Breaks

- Custom error handlers may be affected by the raising of E_DEPRECATED warnings. To fix this, simply update the class constructor name to `__construct`.

RFC: Remove PHP 4 Constructors

Removal of date.timezone Warning

When any date- or time-based functions were invoked and a default timezone had not been set, a warning was emitted. The fix was to simply set the `date.timezone` INI setting to a valid timezone, but this forced users to have a php.ini file and to configure it beforehand. Since this was the only setting that had a warning attached to it, and it defaulted to UTC anyway, the warning has now been removed.

RFC: Remove the date.timezone warning

Removal of Alternative PHP Tags

The alternative PHP tags `<%` (and `<%=`), `%>`, `<script language="php">`, and `</script>` have now been removed.

BC Breaks

- Code that relied upon these alternative tags needs to be updated to either the normal or short opening and closing tags. This can either be done manually or automated with this porting script.

RFC: Remove alternative PHP tags

Removal of Multiple Default Blocks in Switch Statements

Previously, it was possible to specify multiple `default` block statements within a switch statement (where the last `default` block was only executed). This (useless) ability has now been removed and causes a fatal error.

BC Breaks

- Any code written (or more likely generated) that created switch statements with multiple `default` blocks will now become a fatal error.

RFC: Make defining multiple default cases in a switch a syntax error

Removal of Redefinition of Parameters with Duplicate Names

Previously, it was possible to specify parameters with duplicate names within a function definition. This ability has now been removed and causes a fatal error.

```
   function foo($version, $version)

{

    return $version;

}



echo foo(5, 7);



// Pre PHP 7 result

7



// PHP 7+ result

   Fatal error: Redefinition of parameter $version in /redefinition-of-
   parameters.php
```

BC Breaks

- Function parameters with duplicate name will now become a fatal error.

Removal of Dead Server APIs

The following SAPIs have been removed from the core (most of which have been moved to PECL):

- sapi/aolserver
- sapi/apache
- sapi/apache_hooks
- sapi/apache2filter
- sapi/caudium
- sapi/continuity
- sapi/isapi
- sapi/milter
- sapi/nsapi
- sapi/phttpd
- sapi/pi3web
- sapi/roxen
- sapi/thttpd
- sapi/tux

- sapi/webjames
- ext/mssql
- ext/mysql
- ext/sybase_ct
- ext/ereg

RFC: Removal of dead or not yet PHP7 ported SAPIs and extensions

Removal of Hex Support in Numerical Strings

A Stringy hexadecimal number is no longer recognised as numerical.

```
   var_dump(is_numeric('0x123'));

var_dump('0x123' == '291');

echo '0x123' + '0x123';


// Pre PHP 7 result

bool(true)

bool(true)

582


// PHP 7+ result

bool(false)

bool(false)

   0
```

The reason for this change is to promote better consistency between the handling of stringy hex numbers across the language. For example, explicit casts do not recognise stringy hex numbers:

```
var_dump((int) '0x123'); // int(0)
```

Instead, stringy hex numbers should be validated and converted using the `filter_var()` function:

```
var_dump(filter_var('0x123', FILTER_VALIDATE_INT, FILTER_FLAG_ALLOW_HEX)); // int(291)
```

BC Breaks

- This change affects the `is_numeric()` function and various operators, including `==`, `+`, `-`, `*`, `/`, `%`, `**`, `++`, and `--`

RFC: Remove hex support in numeric strings

Removal of Deprecated Functionality

All Deprecated functionality has been removed, most notably:

- The original mysql extension (ext/mysql)
- The ereg extension (ext/ereg)
- Assigning `new` by reference
- Scoped calls of non-static methods from an incompatible `$this` context (such as `Foo::bar()` from outside a class, where `bar()` is not a static method)

BC Breaks

- Any code that ran with deprecation warnings in PHP 5 will no longer work (you were warned!)

RFC: Remove deprecated functionality in PHP 7

Reclassification and Removal of E_STRICT Notices

E_STRICT notices have always been a bit of a grey area in their meaning. This changes removes this error category altogether and either: removes the E_STRICT notice, changes it to an E_DEPRECATED if the functionality will be removed in future, changes it to an E_NOTICE, or promotes it to an E_WARNING.

BC Breaks

- Because E_STRICT is in the lowest severity error category, any error promotions to an E_WARNING may break custom error handlers

RFC: Reclassify E_STRICT notices

Deprecation of Salt Option for `password_hash()`

With the introduction of the new password hashing API in PHP 5.5, many began implementing it and generating their own salts. Unfortunately, many of these salts were generated from cryptographically insecure functions like mt_rand(), making the salt far weaker than what would have been generated by default. (Yes, a salt is always used when hashing passwords with this new API!) The option to generate salts have therefore been deprecated to prevent developers from creating insecure salts.

RFC: no RFC available

Error on Invalid Octal Literals

Invalid octal literals will now cause a parse error rather than being truncated and silently ignored.

```
echo 0678; // Parse error:  Invalid numeric literal in...
```

BC Breaks

- Any invalid octal literals in code will now cause parse errors

RFC: no RFC available

`substr()` Return Value Change

`substr()` will now return an empty string instead of `false` when the start position of the truncation is equal to the string length:

```
    var_dump(substr('a', 1));


// Pre PHP 7 result

bool(false)


// PHP 7+ result

    string(0) ""
```

`substr()` may still return `false` in other cases, however.

BC Breaks

- Code that strictly checked for a `bool(false)` return value may now be semantically invalid

RFC: no RFC available

FAQ

What happened to PHP 6?

PHP 6 was the major PHP version that never came to light. It was supposed to feature full support for Unicode in the core, but this effort was too ambitious with too many complications arising. The predominant reasons why version 6 was skipped for this new major version are as follows:

- **To prevent confusion**. Many resources were written about PHP 6 and much of the community knew what was featured in it. PHP 7 is a completely different beast with entirely different focuses (specifically on performance) and entirely different feature sets. Thus, a version has been skipped to prevent any confusion or misconceptions surrounding what PHP 7 is.
- **To let sleeping dogs lie**. PHP 6 was seen as a failure and a large amount of PHP 6 code still remains in the PHP repository. It was therefore seen as best to move past version 6 and start afresh on the next major version, version 7.

RFC: Name of Next Release of PHP

https://www.ma-no.org/en/programming/php/php7-guide-and-references-to-all-the-changes-between-version-5-x-and-7-of-php