# PHP 7.4

What's new and what's old

Thanks for:
Nikita Popov
Remi

Gabriel Ferreira Rosalino
https://gabrielf.com/php74

# What's new!

*Cool new features and possibilities*

# Typed Properties

```php
<?php

class User {
    public int $id;
    public string $name;
}

class Order {
    public int $id;
    public User $buyer;
    public string $address;
}

$user = new User;
$user->id = 1;
$user->name = "Linio";

$order = new Order;
$order->id = 1;
$order->buyer = $user;
$order->address = false; // should throw an Unacught TypeError exception
```

# Nullability Initialization

```php
<?php

class User {
    public int $id; // no null default
    public ?string $name; // also no null on default (!)
}
```

# Nullability Initialization

```php
<?php

class User {
    public int $id; // no null default
    public ?string $name; // also no null on default (!)
}

$user = new User;
var_dump($user->name); // Also should throw an error:
// Type property User::$name must not be accessed before initialization
```

# Nullability Initialization

```php
<?php

class User {
    public int $id;
    public ?string $name = null;
}

$user = new User;
var_dump($user->name); // NULL
```

# References

```php
<?php

class User {
    public int $id = 42;
    public ?string $name = "linio";
}

$user = new User;
$id =& $user->id;
$id = "other than int"; // Should throw an error:
// Cannot assign string to reference held by property User::$id of type int
```

# Poor man's intersection types

```php
<?php

class Test {
    public ?Traversable $traversable;
    public ?Countable $countable;
    public $both = null;
}

$test = new Test;
$test->traversable =& $test->both;
$test->countable =& $test->both;
$test->both = new ArrayIterator; // Effectively ?(Traversable & Countable)
// Because: ArrayIterator <inherits=> Countable
// And also: ArrayIterator <inherits=> SeekableIterator <inherits=> Iterator
<inherits=> Traversable
```

# Poor man's intersection types

```php
<?php

class Test {
    public ?Traversable $traversable;
    public ?Countable $countable;
    public $both = null;
}

$test = new Test;
$test->traversable =& $test->both;
$test->countable =& $test->both;
$test->both = new AppendIterator; // Should throw an error:
// Uncaught TypeError: Cannot assign AppendIterator to reference held by property
Test::$countable
// of type ?Countable

// Although we have the inheritation:
//  AppendIterator <inherits=>  IteratorIterator <inherits=> OuterIterator
<inherits=> Iterator <inherits=> Traversable
// We don´t have a proper implementation of Countable too
```

# Poor man's typed variable

```php
<?php

// We should also be able to define type to simple variables:
int $i = 0;
$i = "foobar"; // This ideally should throw an error
```

# Poor man's typed variable

```php
<?php

// We can emulate this behavior by creating a function that always work by reference
through mocking an anonymous class
// property. Downside: we have to keep the reference in memory so it doesnt gets
destroyed
function &int(int $i) {
    $obj = new class { public int $prop; };
    $obj->prop = $i;
    $GLOBALS['huge_memory_leak'][] = $obj;
    return $obj->prop;
}

$i =& int(0);
$i = "foobar"; // Should throw an error:
// Uncaught TypeError: Cannot assign string to reference held by property
class@anonymous::$prop of type int
```

# Public typed properties (future)

```php
<?php

class User {
    private $name;

    public function getName() : string
    {
        return $this->name;
    }

    public function setName(string $name) : void
    {
        $this->name = $name;
    }
}
```

# Public typed properties (future)

```php
<?php

class User {
    private string $name;
}

// What if validation will be needed?
```

# Public typed properties (future)

```php
<?php

// Future: Property accessors

class User {
    public string $name {
        get;
        set ($name) {
            if (strlen($name) == 0)
                throw new Exception('Name cannot be empty');
            $this->name = $name;
        }
    };
}
```

# Arrow functions

```php
<?php

$values = [1, 2, 3];
$allowedValues = [2];

array_filter(
    $values,
    function ($v) use ($allowedValues) {
        return in_array($v, $allowedValues);
    }
);
```

# Arrow functions

```php
<?php

$values = [1, 2, 3];
$allowedValues = [2];

array_filter(
    $values,
    fn ($v) => in_array($v, $allowedValues)
);

// Note: implicit use statement for `$allowedValues`
```

# By-value binding

```php
<?php

$i = 0;
$fn = fn() => $i++;
$fn();
var_dump($i); // int(0)
```

# By-value binding

```php
<?php

$fns = [];
foreach ([1, 2, 3] as $i) {
    $fns[] = fn() => $i;
}

foreach ($fns as $fn) {
    echo $fn() . " ";
}

// Prints: 1 2 3
// Not:    3 3 3 (would happen with by-reference)
```

# Syntax: why **fn**?

```php
<?php

$values = [1, 2, 3];
$allowedValues = [2];

array_filter(
    $values,
    $v => in_array($v, $allowedValues) // ECMAScript Syntax
);
```

# Syntax: why **fn**? Ambiguity!

```php
<?php

[$a, $b, $x, $y] = range(1, 4);

$array = [
    $a => $a + $b,
    $x => $x * $y
];

// Array of arrow functions?
// Or just a key-value map?
```

# Syntax: why **fn**? Ambiguity!

```php
<?php

[$a, $b, $x, $y] = range(1, 4);

$array = [
    $a => $a + $b,
    $x => $x * $y
];

// Array of arrow functions?
// Or just a key-value map?
```

# Syntax: why **fn**? Ambiguity!

```php
<?php

// Looks like: Assignment expression
($x = [42] + ["foobar"]) => $x;

// Looks like: Constant lookup + bitwise and
(Type &$x) => $x;

// Looks like: Ternary operator
$a ? ($b): Type => $c : $d;

// Need special handling starting at (
// But only know it's an arrow function at =>
```

# Block syntax (future)

```php
<?php

$values = [1, 2, 3];
$allowedValues = [2];

array_filter(
    $values,
    fn ($v) => in_array($v, $allowedValues)
);

// In the future we'll have block syntax also:
array_filter(
    $values,
    fn ($v) {
        // More code...
        return in_array($v, $allowedValues);
    }
);

// Basically like normal closure syntax,
// but with implicit variable capture (the famous `use` statement)
```

# Covariant return types

```php
<?php

class A { }
class B extends A { }

class Producer {
    public function produce() : A  {}
}

class ChildProducer extends Producer {
    public function produce() : B {}
}

// It works now!
// The parent class returns something more generic
// The child class returns something more specific
```

# Common case: self

```php
<?php

class Foo {
    public function fluent() : self {}
}

class Bar extends Foo {
    public function fluent() : self {}
}
```

# Ordering issues

```php
<?php

class A {
    public function method() : B { }
}

class B extends A {
    public function method(): C { }
}

class C extends B { }

// Sound, but class C not available when verifying B.
// => No way to reorder classes "correctly".
// => Will only work with autoloading (for now).
```

# Covariant parameter types

```php
<?php

interface Event { }
class SpecificEvent implements Event {  }

interface EventHandler {
    public function handle(Event $e);
}

class SpecificEventHandler implements EventHandler {
    public function handle(SpecificEvent $e) { }
}

// Unsound, will never work!
// Seems correct, but it's not the right way to go with parameters
```

# Generic types (future)

```php
<?php

// The correct way would be to use generic types:

interface Event { }
class SpecificEvent implements Event { }

interface EventHandler <E: Event> {
    public function handle(E $e);
}

public function SpecificEventHandler implements EventHandler<SpecificEvent>
{
    public function handle(SpecificEvent $e) {}
}
```

# Coalesce atribution operator *

```php
<?php

$options = [];

$options['something'] ??= new DefaultObject;

// Equals to:
$options['something'] = $options['something'] ?? new DefaultObject;

// The object is only created in the array if $options['something']
```

* Not the official name – given by me

# Array spread operator

```php
<?php

$array = [3, 4, 5];

// You can use the operator on function/method calls:
return call(1, 2, ...$array); // call(1, 2, 3, 4, 5)

// Or in another array:
return [1, 2, ...$array];
```

# Array spread operator

```php
<?php

$array = new ArrayIterator([3, 4, 5]);
return call(1, 2, ...$array);

return [1, 2, ...$array];
```

# Array spread operator

```php
<?php

$array = ['y' => 'b', 'z' => 'c'];
return ['x' => 'a', ...$array];

// Uncaught Error: Cannot unpack array with string keys
```

# Array spread operator in destructing (future)

```php
<?php

$array = [1, 2, 3, 4];
[$head, ...$tail] = $array;

// Or either:

$array = [2 => 2, 1 => 1, 0 => 0];
[$head, ...$tail] = $array;

// What happens? Not exactly very clear for this feature in the end yet
```

# Weak reference

```php
<?php

$object = new stdClass;
$weakRef = WeakReference::create($object);

var_dump($weakRef->get()); // object(stdClass)#1
unset($object);
var_dump($weakRef->get()); // null
```

# Weak map reference (future)

```php
<?php

// People usually like to associate extra data to a object that may happens to be in
a week ref
// Usually this approach is used in cache structures
// To keep track of arrays/maps of cache we could do something like below:

// The array key is the object id, which will be reused once object is destroyed
$this->data[spl_object_id($object)] = [WeakReference::create($object), $data];
// Check $checkRef->get() != null
// to know whether the entry is still valid

// The problem with this approach:
// Doesn't keep $object alive, but keeps dead cache entry

// Instead, in the future we may have:

$this->data = new WeakMap();
$this->data[$object] = $data;

// Does not keep $object alive; immediately drops cache entry when object destroyed
```

# Preload & Foreign Function Interface (FFI)


```
#define FFI_SCOPE "_REMI_ZSTD_"
#define FFI_LIB   "libzstd.so.1"

size_t ZSTD_compress(void* dst, size_t dstCapacity, const void* src, size_t srcSize,
int compressionLevel);
size_t ZSTD_decompress(void* dst, size_t dstCapacity, const void* src, size_t
compressedSize);
size_t ZSTD_compressBound(size_t srcSize);
unsigned long long ZSTD_decompressBound(const void* src, size_t srcSize);
unsigned ZSTD_isError(size_t code);
```

# Preload & Foreign Function Interface (FFI) wrapper

```php
<?php

namespace Remi;

class Zstd {
    static private $ffi = null;

    private static function init() {
        if (self::$ffi) {
            return;
        }
        // Try if preloaded
        try {
            self::$ffi = \FFI::scope("_REMI_ZSTD_");
            echo "Using FFI::scope OK\n";
        } catch (\FFI\Exception $e) {
            // Try direct load
            self::$ffi = \FFI::load(__DIR__ . '/preload-zstd.h');
            echo "Using FFI::load OK\n";
        }
        if (!self::$ffi) {
            throw new \RuntimeException("FFI parse fails");
        }
    }

    public static function compress(string $src): string {
        self::init();

        $len = strlen($src);
        $max = self::$ffi->ZSTD_compressBound($len);

        $comp = str_repeat(' ', $max);
        $clen = self::$ffi->ZSTD_compress($comp, $max, $src, $len, 6);
        if (self::$ffi->ZSTD_isError($clen)) {
            throw new \RuntimeException("Compression fails");
        }

        return substr($comp, 0, $clen);
    }

    public static function decompress(string $comp): string {
        self::init();

        $clen = strlen($comp);
        $max = self::$ffi->ZSTD_decompressBound($comp, $clen);

        $unco = str_repeat(' ', $max);
        $ulen = self::$ffi->ZSTD_decompress($unco, $max, $comp, $clen);
        if (self::$ffi->ZSTD_isError($clen)) {
            throw new \RuntimeException("Decompression fails");
        }

        return substr($unco, 0, $ulen);
    }
}
```

# Preload & Foreign Function Interface (FFI) usage example

```php
<?php declare(strict_types=1);

if (PHP_VERSION_ID < 70400 || !extension_loaded("ffi")) {
    die("PHP 7.4 with FFI required\n");
}
if (PHP_SAPI != "cli") {
    Header('Content-Type: text/plain');
}
printf("PHP version %s\n", PHP_VERSION);

if (PHP_SAPI == "cli" && !class_exists("\\Remi\\Zstd")) {
    printf("Fallback on manual load\n");
    require_once __DIR__ . '/35_ffi_lzstd_example/preload-zstd.inc';
} else {
    printf("Use preloaded class\n");
}
if (class_exists("\\Remi\\Zstd")) {
    $t = microtime(true);

    $src = file_get_contents(PHP_BINARY);
    $comp = \Remi\Zstd::compress($src);
    printf("\nSrc length           = %d\n", \strlen($src));
    printf("ZSTD_compress        = %d\n", \strlen($comp));
    file_put_contents("/tmp/testffi.zstd", $comp);

    $comp = file_get_contents("/tmp/testffi.zstd");
    $out = \Remi\Zstd::decompress($comp);
    printf("Src length           = %d\n", \strlen($comp));
    printf("ZSTD_decompress      = %d\n", \strlen($out));
    file_put_contents("/tmp/testffi.orig", $out);

    printf("Check                = %s\n", $src === $out ? "OK" : "KO");

    $t = microtime(true) - $t;
    printf("Using FFI  extension = %.3f\"\n\n", $t);
} else {
    printf("\\Remi\\Zstd missing\n\n");
}
```

```
# Most simple usage

$ php 34_foreign_function_interfaces.php

# With preloading the wrapper

$ php -d opcache.preload=35_ffi_lzstd_example/preload-zstd.inc
34_foreign_function_interfaces.php

# With preloading the wrapper and the native C library

$ php d ffi.preload=35_ffi_lzstd_example/preload-zstd.h \
      -d opcache.preload=35_ffi_lzstd_example/preload-zstd.inc \
      34_foreign_function_interfaces.php
```

# Performance comparison with the lzstd use case

FFI wrapper
vs
PHP native extension

```
PHP version 7.4.0RC4
Use preloaded class
Using FFI::scope OK

# FFI Wrapper
Src length              = 8673632
ZSTD_compress           = 1828461
Src length              = 1828461
ZSTD_decompress         = 8673632
Check                   = OK
Using FFI  extension = 0,09"

# PHP extension
Src length              = 8673632
ZSTD_compress           = 1828461
Src length              = 1828461
ZSTD_decompress         = 8673632
Check                   = OK
Using ZSTD extension = 0,09"
```

# What's old!

*Some deprecations and language behavior changes*

# Ternary associativity

```php
<?php

return $a == 1 ? 'one'          // Deprecated in PHP 7.4.
     : $a == 1 ? 'two'          // Compile error in 8.0.
               : 'other';

// was intended as:
return $a == 1 ? 'one'
     : ($a == 2 ? 'two'
                : 'other');

// but PHP interprets it as:
return ($a == 1 ? 'one'
     : $a == 2) ? 'two'
                : 'other';
```

# Concatenation precedence

```php
<?php

$a = 1;
$b = 2;

echo "Sum: " . $a + $b; // Deprecated in PHP 7.4

// Was intended as:
echo "Sum: " . ($a + $b); // New behavior in PHP 8.0

// But PHP interprets it as:
echo ("Sum: " . $a) + $b;
```

# Short open tags

```php
<?php

/*
 * <? deprecated in 7.4, removed in 8.0
 * Only <?php and <?= supported
 * (???) short_open_tag default value from On to Off in 7.4
 */

// Disclaimer: RFC accepted, but much push-back after voting.
```

# Array & string offset with curly braces

```php
<?php

/*
 * Array and string offset access using curly braces
 *
 * The array and string offset access syntax using curly braces is deprecated.
 * Use $var[$idx] instead of $var{$idx}.
 */

$arr = [1];

// Don't do:
$myVal = $arr{0};

// Do:
$myVal = $arr[0];
```

# Real type

```php
<?php

/*
 * (real) cast and is_real() function
 *
 * The (real) cast is deprecated, use (float) instead.
 * The is_real() function is also deprecated, use is_float() instead.
 *
 */

// Don't do:
$myDecimal = (real) 0.42;
is_real($myDecimal);

// Do:
$myDecimal = (float) 0.42;
is_float($myDecimal);
```

# Unbinding $this from closure

```php
<?php

/*
 * Unbinding $this when $this is used
 *
 * Unbinding $this of a non-static closure that uses $this is deprecated.
 */

// Don't do:
class A {
    public function foo() {}
    public function method() {
        $this->foo();
    }
}

$closure = (new ReflectionMethod('A', 'method'))->getClosure(new A);
$closure = $closure->bindTo(null, 'A');
$closure();

// Do:
class B {
    public static B $instance;
    public function foo() {}
    public static function method() {
        if (!static::$instance) {
            static::$instance = new B;
        }
        static::$instance->foo();
    }
}

$closure = (new ReflectionMethod('B', 'method'))->getClosure(new A);
$closure = $closure->bindTo(null, 'B');
$closure();
```

# Parent keyword without parent class

```php
<?php

/*
 * parent keyword without parent class
 *
 * Using parent inside a class without a parent is deprecated, and will throw a
compile-time error in the future.
 * Currently an error will only be generated if/when the parent is accessed at run-
time.
 */

// Don't do:
class Foo {
    public function bar() : void
    {
        parent::bar();
    }
}

// Do:
class Bar extends Foo {
    public function bar() : void
    {
        parent::bar();
    }
}
```

# *allow_url_include* ini config

```php
<?php

/*
 * allow_url_include INI option
 *
 * The allow_url_include ini directive is deprecated. Enabling it will generate a
deprecation notice at startup.
 *
 */

// Don't do:
include('https://google.com/script.js');

// Do:
function include_url($url) {
    $ch = curl_init();
    curl_setopt($ch, CURLOPT_URL, $url);
    curl_setopt($ch, CURLOPT_RETURNTRANSFER, 1);
    $output = curl_exec($ch);
    curl_close($ch);
    return $output;
}
include_url('https://google.com/script.js');
```

# Invalid characters in base conversion

```php
<?php

/*
 * Invalid characters in base conversion functions
 *
 * Passing invalid characters to base_convert(), bindec(), octdec() and hexdec() will
now generate a deprecation notice.
 * The result will still be computed as if the invalid characters did not exist.
Leading and trailing whitespace,
 * as well as prefixes of type 0x (depending on base) continue to be allowed.
 *
 */
```

# *array_key_exists* with object

```php
<?php

/*
 * Using array_key_exists() on objects
 *
 * Using array_key_exists() on objects is deprecated. Instead either isset() or
property_exists() should be used.
 */

// Don't do:
$obj = new \stdClass;
array_key_exists('foo', $obj);

// Do:
property_exists($obj, 'foo');
```

# Magic quotes functions

```php
<?php

/*
 * Magic quotes functions
 *
 * The get_magic_quotes_gpc() and get_magic_quotes_runtime() functions are
deprecated.
 * They always return FALSE.
 */
```

# *hebrevc* function

```php
<?php

/*
 * hebrevc() function
 *
 * The hebrevc() function is deprecated.
 * It can be replaced with nl2br(hebrev($str)) or, preferably, the use of Unicode RTL
support.
 */

// Don't do:
hebrevc("á çùåï äúùñâ\ná çùåï äúùñâ");

// Do:
nl2br(hebrev("á çùåï äúùñâ\ná çùåï äúùñâ"));
```

# *convert_cyr_string* function

```php
<?php

/*
 * convert_cyr_string() function
 *
 * The convert_cyr_string() function is deprecated.
 * It can be replaced by one of mb_convert_string(), iconv() or UConverter.
 */

// Don't do:
$s = "Welcome! æøå";
convert_cyr_string($s,'w','a');

// Do:
mb_convert_encoding($s, "UCS-2LE", "JIS, eucjp-win, sjis-win");
```

# *money_format* function

```php
<?php

/*
 * money_format() function
 *
 * The money_format() function is deprecated.
 * It can be replaced by the intl NumberFormatter functionality.
 */

// Don't do:
$number = 1234.56;
money_format('%.2n', $number);

// Do:
$formatter = new \NumberFormatter("it-IT", \NumberFormatter::CURRENCY);
$symbol = $formatter->getSymbol(\NumberFormatter::INTL_CURRENCY_SYMBOL);
$formatter->formatCurrency($number,  $symbol);
```

# *ezmlm_hash* function

```php
<?php

/*
 * ezmlm_hash() function
 *
 * The ezmlm_hash() function is deprecated.
 */
```

# *restore_include_path* function

```php
<?php

/*
 * restore_include_path() function
 *
 * The restore_include_path() function is deprecated.
 * It can be replaced by ini_restore('include_path').
 */

// Don't do:
restore_include_path();

// Do:
ini_restore('include_path');
```

# *implode* function with any parameter order

```php
<?php

/*
 * Implode with historical parameter order
 *
 * Passing parameters to implode() in reverse order is deprecated,
 * use implode($glue, $parts) instead of implode($parts, $glue).
 */

// Don't do:
$arr = [1,2,3];
implode($arr, '|');

// Do:
implode('|', $arr);
```

# Deprecation of case insensitive constants

```php
<?php

/*
 * Importing type libraries with case-insensitive constant registering has been
deprecated.
 */

// Don't do:
define('FOO', 42, true); // Deprecated: define(): Declaration of case-insensitive
constants is deprecated
var_dump(FOO); // Ok!
var_dump(foo); // Deprecated: Case-insensitive constants are deprecated. The correct
casing for this constant is "FOO"

// Do:
define('FOO', 42);
var_dump(FOO); // Ok!
var_dump(foo); // Will throw an error of undefined constant
```

# Magic quotes filter deprecated

```php
<?php

/*
 * FILTER_SANITIZE_MAGIC_QUOTES is deprecated, use FILTER_SANITIZE_ADD_SLASHES
instead.
 */

// Don't do:
$dangerousString = "my string ' OR 1=1";
filter_var($dangerousString, FILTER_SANITIZE_MAGIC_QUOTES);

// Do:
filter_var($dangerousString, FILTER_SANITIZE_ADD_SLASHES);
```

# Multibyte string

```php
<?php

/*
 * Multibyte String
 *
 * Passing a non-string pattern to mb_ereg_replace() is deprecated.
 * Currently, non-string patterns are interpreted as ASCII codepoints.
 * In PHP 8, the pattern will be interpreted as a string instead.
 *
 * Passing the encoding as 3rd parameter to mb_strrpos() is deprecated.
 * Instead pass a 0 offset, and encoding as 4th parameter.
 */
```

# LDAP pagination

```php
<?php

/*
 * Lightweight Directory Access Protocol
 *
 * ldap_control_paged_result_response() and ldap_control_paged_result() are
deprecated.
 * Pagination controls can be sent along with ldap_search() instead.
 */
```

# Reflection export methods

```php
<?php

/*
 * The export() methods on all Reflection classes are deprecated.
 * Construct a Reflection object and convert it to string instead:
 */

class Foo {
    public function bar() { }
}

// Don't do:
$str = ReflectionFunction::export('foo');

// Do:
$str = (string) new ReflectionFunction('foo');
```

# Socket flags

```php
<?php

/*
 * The AI_IDN_ALLOW_UNASSIGNED and AI_IDN_USE_STD3_ASCII_RULES flags for
socket_addrinfo_lookup() are deprecated,
 * due to an upstream deprecation in glibc.
 */
```