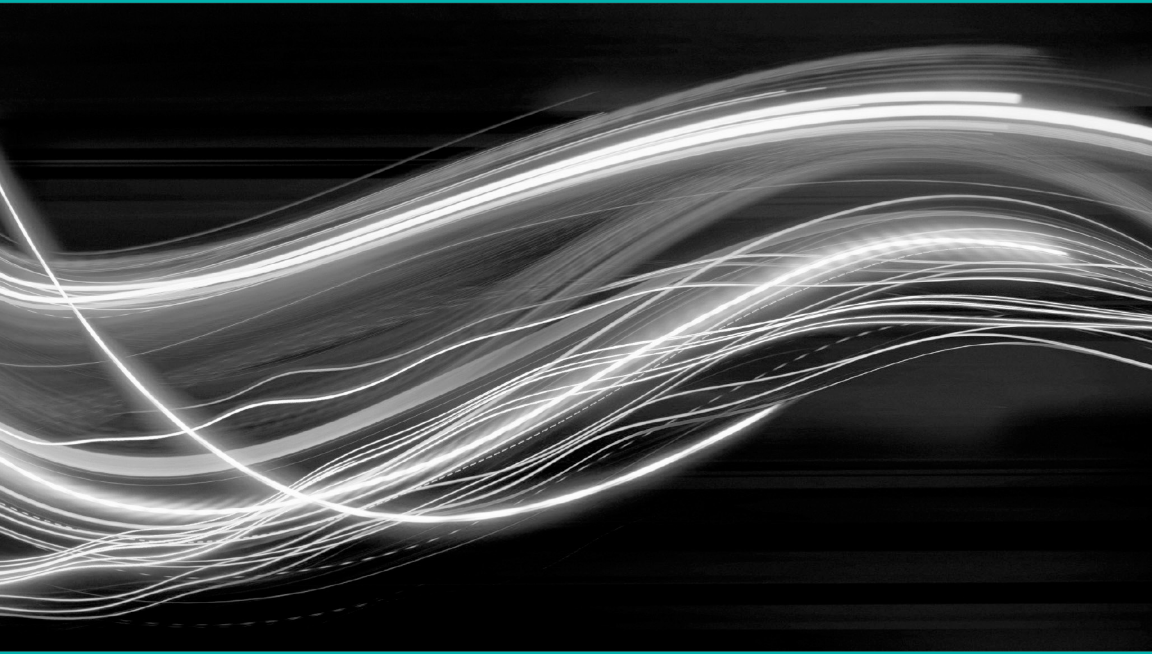
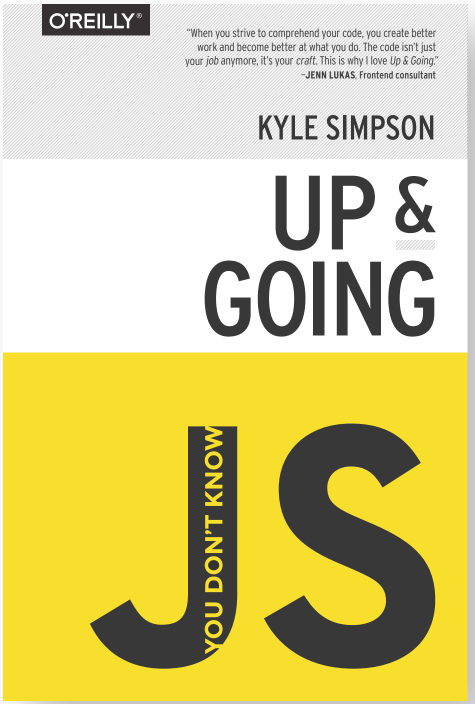
O'REILLY®

Upgrading to PHP 7



Davey Shafik



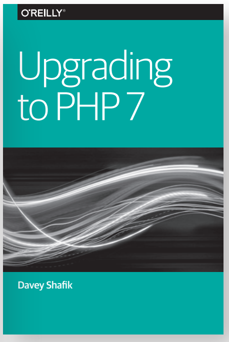
O'REILLY®

"When you strive to comprehend your code, you create better work and become better at what you do. The code isn't just your *job* anymore, it's your *craft*. This is why I love *Up & Going*."
—JENN LUKAS, Frontend consultant

KYLE SIMPSON

UP & GOING

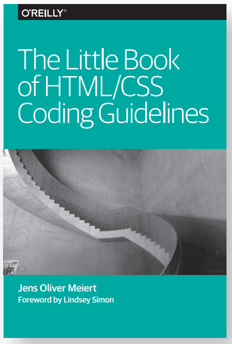
YOU DON'T KNOW JS



O'REILLY®

Upgrading to PHP 7

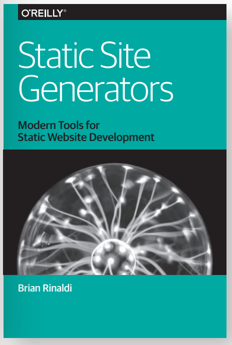
Davey Shafik



O'REILLY®

The Little Book of HTML/CSS Coding Guidelines

Jens Oliver Meiert
Foreword by Lindsey Simon

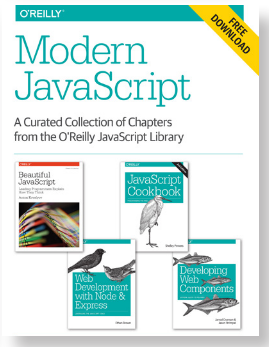


O'REILLY®

Static Site Generators

Modern Tools for Static Website Development

Brian Rinaldi



O'REILLY®

FREE DOWNLOAD

Modern JavaScript

A Curated Collection of Chapters from the O'Reilly JavaScript Library

Beautiful JavaScript

JavaScript Cookbook

Web Development with Node & Express

Developing Web Components

Upgrading to PHP 7

Davey Shafik

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Upgrading to PHP 7
by Davey Shafik

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editor: Allyson MacDonald
Production Editor: Matthew Hacker
Copyeditor: Marta Justak
Interior Designer: David Futato
Cover Designer: Randy Comer
Illustrator: Rebecca Demarest

October 2015: First Edition

Revision History for the First Edition
2015-10-29 First Release

978-1-4919-4009-9

[LSI]

Table of Contents

CHAPTER 1
Upgrading to PHP 7

PHP 7 is here, and it's the most dramatic update to PHP in over a decade. A revamped engine (Zend Engine 3), numerous new features, and lots of language cleanup mean lots of exciting changes to the language that runs the Web.

Bringing with it huge speed improvements and minimal backward incompatibility, there are major benefits to upgrading *today*.

PHP 7 Timeline

With PHP 7.0 now released, we will see the end of life for PHP 5.5 on July 10, 2016, and PHP 5.6 will move to security-only fixes just a few months later on August 28, 2016—with its end of life scheduled to take place a year later.

What this also means, implicitly, is that any PHP version prior to 5.5 has *already* reached its end of life and is *no longer receiving security fixes*.

Given the backward incompatibility issues with moving to PHP 7, you might think that upgrading will be a painful and long process; however, the PHP team has done a fantastic job at minimizing backward incompatibility. In fact, I would go so far as to say that *the upgrade to PHP 7.0 is easier than upgrading from 5.2 to 5.3*.

How We Got Here

Keen-eyed readers will have noticed that we skipped straight from PHP 5 to PHP 7, and if you're curious like me, you might be wondering just *why* that would be the case. While you might be tempted to think we're following in the footsteps of Microsoft® Windows (which skipped version 9 and jumped from 8.1 to 10), in actual fact, it was a subject of much debate filled with intrigue, mystery, and murder. OK, maybe not murder, but there were definitely some ALL CAPS emails on the PHP internals mailing list!

The primary reason behind the jump was that PHP 6 existed as a real project that many people put a lot of hours into—between August 2005 and March 2010 when it was finally killed off, that's almost *five years!*—that would've brought native Unicode support throughout the language.

Unfortunately, it never came to fruition, and to stop the project from stagnating, it was decided to release PHP 5.3 in June 2009 with all the *other* features that were waiting for the Unicode support being completed before they could be released.

Those features included things you might take for granted these days, like closures and namespaces.

Additionally, there were books, many blog posts, and other content produced around the PHP 6 that never was. Between this, and the fact that it *was* a real thing, even if unreleased, it was decided to skip 6.0 and jump straight to 7.0.

Release Cycle

The release cycle timeline for PHP 7 has been incredibly rapid, primarily because the major change (a large rewrite of parts of the Zend Engine) was performed prior to the decision to embark on a new major version, and announced by Zend as php-ng.

The timeline for PHP 7 was formalized in the PHP 7.0 Timeline RFC, which was passed in November 2014, and it was projected for a mid-October release date—just 11 months later.

The timeline called for a feature freeze on March 15, then a further three months to finalize the implementation of the agreed-on features. Finally, between June 15th and the mid-October release date we saw multiple betas and release candidates (Figure 1-1).

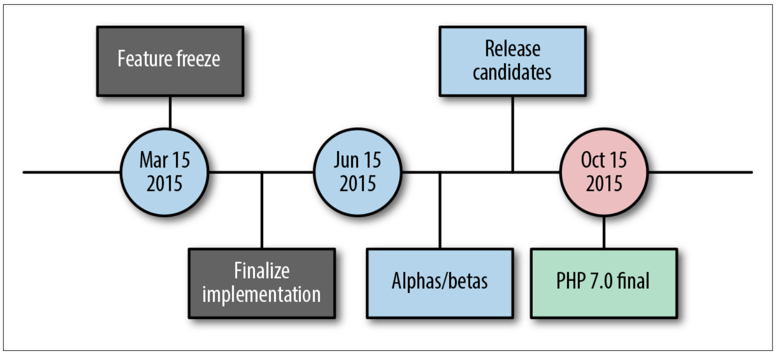


Figure 1-1. PHP 7.0 release timeline

As you will see, despite its relatively short timeline, PHP 7.0 is a very impressive release: bringing many new features to the language that powers most of the Web in a mostly backward-compatible way, while increasing performance at the same time.

CHAPTER 2
Deprecated Features

Over the last few releases of PHP 5.x, we've seen a number of features marked as deprecated, and with PHP 7.0, they have *all* been removed.

NOTE

Deprecated

A feature is marked as deprecated to warn developers that it will be removed in an unspecified future version of the language so that they can start to migrate away from using the feature or avoid using it in the first place. In PHP, using these features will cause an `E_DEPRECATED` error to be emitted.

Alternative PHP Tags

While some developers may not even be aware of it, PHP has alternative open and close tags, both of which have been removed.

These were known as *script tags*, as shown in Example 2-1, and *ASP tags*—which included a short echo tag—as shown in Example 2-2.

Example 2-1. PHP script tags

```
<script language="php">
    // Code here
</script>
```

Example 2-2. PHP ASP tags

```
<%
    // Code here
%>

<%=$varToEcho; %>
```

While I expect that the number of people using these tags is minimal, I have seen the ASP syntax used for templates. If you are using them, you will need to change to using the standard PHP tags, `<?php`, `<?=`, and `?>`.

POSIX-Compatible Regular Expressions

Deprecated in PHP 5.3, *POSIX-compatible regular expressions*, used for string pattern matching, have been removed in PHP 7.0. This means that the entire *ext/ereg* extension has been removed.

This includes the following functions:

- `ereg()`
- `eregi()`
- `ereg_replace()`
- `eregi_replace()`
- `split()`
- `spliti()`
- `sql_regcase()`

Migrating to Perl Compatible Regular Expressions

Due to the lengthy deprecation period (six years!), the usage of the `ereg` extension has declined dramatically. If you have not yet migrated, you will need to switch to the `preg_` family of functions.

Thankfully, POSIX-compatible regular expressions are reasonably compatible with *Perl Compatible Regular Expressions (PCRE)*. The two major changes you will have to make to simple expressions are the addition of delimiters around the expression string (usually a `/`) and using the `i` modifier instead of dedicated case-insensitive functions.

However, there is a more subtle difference that you might run into, which is known as *greediness*.

With POSIX regular expressions, matches are not greedy, which means they will match as much as possible *up until* they reach something matching the next part of the expression.

With PCRE, by default, matches are greedy, meaning they will match as much as possible until the next part of the expression no longer matches.

It is possible to fix this in two ways. The first is to follow quantifiers with a question mark (`?`)—this will make that part of the pattern have the same ungreedy behavior as POSIX regular expressions. The second is to use the `U` modifier, which will invert the greediness, making all quantifiers ungreedy by default and using the `?` to make them greedy.

I personally prefer to use the default behavior, as it *is* the default behavior and is what most developers will expect when reading your code.

Here we take a look at a simple regular expression for matching segments of a URL. As you can see the code is very similar between the POSIX-compatible regular expression and the PCREs.

Example 2-3. Migrating from POSIX- to Perl compatible regular expressions

```
$url = "https://example.org/path/here";

// POSIX
$matches = [];
$regex = "^(http(s?))://(.*)$";
if (eregi($regex, $url, $matches)) { ❶
    // string matched
    var_dump($matches);
    /*
    array(5) {
      [0] =>
      string(29) "https://example.org/path/here"
      [1] =>
      string(5) "https"
      [2] =>
      string(1) "s"
      [3] =>
      string(21) "example.org/path/here"
    }
    */
}

// PCRE
$matches = [];
$regex = "@^(http(s?)):\/\/(.*)$@i"; ❷ ❸
if (preg_match($regex, $url, $matches)) {
    // string matched
    var_dump($matches);
    /*
    the resulting array is the same as with eregi above
    */
}
```

❶ `eregi()` is used for case-insensitive matching.

❷ The `@` delimiter is used to avoid escaping the `/` characters in the URL.

❸ The `i` modifier is used after the closing delimiter to use case-insensitive matching.

In addition to being able to replicate the behavior of POSIX-compatible regular expressions, PCREs bring a host of other new features to the table.

For example, they support Unicode and localization.

My personal favorites are naming capture groups using `(?<NAME>` *`expression`*`)` and accessing the matches using the named key in the resulting matches array. As well as ignoring capture groups using `(:?` *`expression`*`)`, they also support advanced features like look-aheads and look-behinds, and many more.

Another great feature is the `x` modifier, which will ignore nonexplicit whitespace and allow you to add comments using the `#` line comment syntax. This makes it easy to document complex regular expressions as shown in Example 2-4.

Example 2-4. Using the PCRE x modifier

```
$url = "https://example.org/path/here?foo=bar";

$matches = [];
$regex = "@         # Delimiter
^                   # Begining of the string
(?<protocol>        # Name the submatch: protocol
    http              # Match the http protocol
    (?:               # Ignore the subgroup used for https matching
        s?            # Optionally match the s in the https protocol
    )
)
://                   # Match the :// from the protocol
(?<host>              # Name the submatch: host
    .*                # Match any characters
    ?                 # But don't be greedy.
                      # This will stop at the first '/'.
)
(?<path>              # Name the submatch: path
    /                 # Match the /
    [^\?]+            # Match any character that isn't a ?
)
?                     # but only if the path exists
(?:                   # Ignore the subgroup for the ?
    \?                # Match the query string delimiter
    (?<query>         # Name the submatch: query
        .+            # Match the query string itself
    )
)
?                     # but only if it exists
$                     # End of string
@ix";                 # Use the i (case-insentive)
                      # and x (extended) flags ❶

if (preg_match($regex, $url, $matches)) {
    // string matched
    var_dump($matches);
    /*
    array(9) {
      [0] =>
      string(37) "https://example.org/path/here?foo=bar"
      'protocol' => ❷
      string(5) "https" ❸
      [1] =>
      string(5) "https"
      'host' =>
      string(11) "example.org"
      [2] =>
      string(11) "example.org"
      'path' =>
      string(10) "/path/here"
      [3] =>
      string(10) "/path/here"
      'query' =>
      string(7) "foo=bar"
      [4] =>
      string(7) "foo=bar"
    }
    */
}
```

❶ Note that this last comment is a standard PHP comment.

❷ A key exists with the result of each named subexpression, `protocol`, `host`, `path`, and `query`.

❸ The *s* is not returned by itself as the capture group was ignored with `?:`.

I highly recommend O'Reilly's *Mastering Regular Expressions* for an in-depth look at the full power of PCRE.

Multiple Default Cases in Switches

When creating the PHP language spec, a bug was found: you could define multiple `default:` cases inside a `switch`, but only the *last* one would execute, which could lead to potential—hard to find—bugs, as shown in Example 2-5.

Example 2-5. Defining multiple default cases inside a switch in PHP 5.x

```
switch ($expr) {
    default:
        echo "Hello World";
        break;
    default: ❶
        echo "Goodbye Moon!";
        break;
}
```

❶ Only the last `default` case will execute.

To solve this, the ability to define this was removed, and now PHP 7 will throw a fatal error if you try to do so:

```
    Fatal error: Switch statements may only contain one default
    clause
```

Removal of the Original MySQL Extension

Deprecated in PHP 5.5, the original `ext/mysql` extension has been removed; this includes all `mysql_` functions.

This is likely to be the largest change to your existing code if you are using it without any sort of wrapper.

Migrating to Procedural mysqli

The simplest migration path is to the procedural `mysqli_` functions, which are part of the `ext/mysqli` extension. For the most part, they function almost identically to their `mysql_` counterparts, except for the `i` suffix.

In most cases, all you need to do is change the function name, and in about 50 percent of the cases you will need to pass the database handle (returned by `mysqli_connect()`) in as the first argument.

Incompatible functions

The following functions have no direct equivalent in `ext/mysqli`, although `mysql_freeresult()`, `mysql_numrows()`, and `mysql_selectdb()` have similarly named, functionally identical equivalents as noted.

A list of incompatible functions can be seen in Table 2-1.

Table 2-1. List of incompatible functions between ext/mysql and ext/mysqli

`mysql_client_encoding()`	`mysql_list_dbs()` (use `SHOW DATABASES` query)
`mysql_db_name()`	`mysql_list_fields()`
`mysql_db_query()`	`mysql_list_processes()` (use `SHOW PROCESSLIST` query)
`mysql_dbname()`	`mysql_list_tables()` (use `SHOW TABLES` query)
`mysql_field_flags()`	`mysql_listdbs()` (use `SHOW DATABASES` query)
`mysql_field_len()`	`mysql_listfields()`

`mysql_field_name()`	`mysql_listtables()` (use `SHOW TABLES` query)
`mysql_field_table()`	`mysql_numfields()`
`mysql_field_type()`	`mysql_numrows()` (use `mysqli_num_rows()` instead)
`mysql_fieldflags()`	`mysql_pconnect()` (append `p:` to the hostname passed to `mysqli_connect()`)
`mysql_fieldlen()`	`mysql_result()`
`mysql_fieldname()`	`mysql_selectdb()` (use `mysqli_select_db()` instead)
`mysql_fieldtable()`	`mysql_table_name()`
`mysql_fieldtype()`	`mysql_tablename()`
`mysql_freeresult()` (use `mysqli_free_result()` instead)	`mysql_unbuffered_query()`

It should be reasonably easy to write a compatibility layer that wraps `ext/mysqli` using the old function names, as they no longer exist in PHP and can be defined in your own code.

Migrating to an Object-Oriented API

There are two options if you want to make the jump to an object-oriented API at the same time that you are migrating away from `ext/mysql`. The first is again, `ext/mysqli`, which provides both a procedural and object-oriented API, and the second is the *PHP Data Objects* (better known as *PDO*), or `ext/pdo` with `ext/pdo_mysql`.

My personal preference is PDO, but either one of the extensions will bring better security and more features, such as prepared statements and calling stored procedures.

Using PDO

The PDO extension allows you to connect to a variety of databases using a (mostly) consistent API. While it is not an abstraction layer for the SQL queries, it allows you to only have to learn one API for working with many databases, rather than a different API for each. You can see an example of using MySQL via PDO in Example 2-6.

Example 2-6. Using PDO in place of ext/mysql

```
$email = \filter_var($_POST['email'], FILTER_SANITIZE_EMAIL);

try {
    $pdo = new \PDO("mysql:host=localhost;dbname=test", "test"); ❶
} catch (\PDOException $e) {
    // Connection failure
}

$sql = "
SELECT
    first_name, last_name, email
FROM users
WHERE
    email = :email"; ❷

$values = [":email" => $email];

try {
    $query = $pdo->prepare($sql); ❸
    $result = $query->execute($values); ❹

    if (!$result || $query->rowCount() == 0) {
        return false; ❺
    }

    foreach ($query->fetch(\PDO::FETCH_OBJ) as $row) {
        ❻
    }
} catch (\PDOException $e) {
    // Something went wrong
}
```

❶ PDO connections use a *Data Source Name* (DSN), a string that denotes the driver to use, the host, and the database to connect to, as well as additional optional connection settings.

❷ Use a placeholder of `:email` to denote your condition value for your prepared query.

❸ Prepare the query, returning an instance of `\PDOStatement`.

❹ Execute the query passing in the values for all placeholders.

❺ Query failed, or no results were found.

❻ Using a fetch mode of `\PDO::FETCH_OBJ` will mean that `$row` contains an object whose properties are named after the columns selected, and that they contain the appropriate values.

Fetch Modes

In the previous example, we return an object for each row selected; however, PDO supports many different types of data structures for returned rows. You can also use `\PDO::FETCH_ASSOC` to return an associative array indexed by column name, `\PDO::FETCH_NUM` to return a numerically keyed array index on column position, or `\PDO::FETCH_BOTH`, which will return an array with both associative and numeric keys.

Additionally, you can do other things such as fetch into an existing object, or use custom objects with the result data injected into it for each result.

Using Mysqli

The mysqli extensions object-oriented interface is quite different from PDO. Mysqli only supports anonymous placeholders (denoted by a question mark: `?`) rather than named placeholders, and it requires you to bind variables explicitly to each placeholder. It also does not throw exceptions on errors. If you wish to use `ext/mysqli` instead of PDO, the previous example would be rewritten as Example 2-7.

Example 2-7. Using ext/mysqli in place of ext/mysql

```
$email = \filter_var($_POST['email'], FILTER_SANITIZE_EMAIL);

$mysqli = new \mysqli('localhost', 'test', null, 'test');

if (\mysqli_connect_errno()) {
    ❶
}

$sql = "
SELECT
    first_name, last_name, email
FROM users
WHERE
    email = ?"; ❷
$query = $mysqli->prepare($sql); ❸
$query->bind_param('s', $email); ❹
```

```
$result = $query->execute(); ❺

if (!$result) {
    return false; ❻
}

$result = $query->fetch_result(); ❼

while ($row = $result->fetch_object()) {
    ❽
}
```

❶ Because `ext/mysqli` does not throw exceptions, you must check for connection errors manually.

❷ Use a placeholder `?` to denote your condition value for your prepared query.

❸ Prepare the query, returning an instance of `\mysqli_stmt`.

❹ Bind the `$email` variable to the first placeholder as a string.

❺ Execute the query.

❻ An error occurred executing the query.

❼ Fetch the result set, returning an instance of `\mysqli_result`.

❽ Using `\mysqli_result->fetch_object()` will mean that `$row` contains an object whose properties correspond to each selected column, containing their values. As with PDO, there are many other ways to retrieve results.

Summary

While this isn't *every* backward-incompatible change, it is what will likely trip you up when migrating to PHP 7.0.

Moving away from these deprecated features may not be quick or easy, but doing so will bring more performance, enhanced security, and better code to your projects.

If you aren't using any of these deprecated features, your transition to PHP 7 will largely be painless, so congratulations!

CHAPTER 3

Uniform Variable Syntax

Until the Uniform Variable Syntax RFC was proposed, I had never considered just how inconsistent PHP's variable syntax was, particular around *variable-variables* and *variable-properties*.

For example, given the syntax `$object->$array[key];`, as developers we are just expected to *know* that PHP will first resolve `$array[key]` to a string and then access the property named by that string on the `$object`.

With Uniform Variable Syntax, all of this inconsistency is fixed, and while it is a backward-incompatible change, it is fairly trivial to change your code to be both forward- and backward-compatible, but it is also a very difficult change to spot.

Consistency Fixes

With uniform variable syntax *all* variables are evaluated from left to right.

This is primarily only an issue when constructing complex dynamic variables and properties. As you can see in Example 3-1 we are moving from the inconsistent PHP 5.x behavior to a new left to right consistent behavior. Also, as shown in Example 3-1, you can achieve the same PHP 5.x behavior in PHP 7, or the new PHP 7 behavior in PHP 5.x by explicitly specifying the order of operations with the addition of appropriate parentheses `()` and braces `{}`.

Example 3-1. Examples of left-to-right consistency changes

```
// Syntax
$$var['key1']['key2'];
// PHP 5.x:
// Using a multidimensional array value as variable name
${$var['key1']['key2']};
// PHP 7:
// Accessing a multidimensional array within a variable-variable
($$var)['key1']['key2'];

// Syntax
$var->$prop['key'];
// PHP 5.x:
// Using an array value as a property name
$var->{$prop['key']};
// PHP 7:
// Accessing an array within a variable-property
($var->$prop)['key'];


// Syntax
$var->$prop['key']();
// PHP 5.x:
// Using an array value as a method name
$var->{$prop['key']}();
// PHP 7:
// Calling a closure within an array in a variable-property
($var->$prop)['key']();

// Syntax
ClassName::$var['key']();
// PHP 5.x:
// Using an array value as a static method name
ClassName::{$var['key']}();
// PHP 7:
// Calling a closure within an array in a static variable
(ClassName::$var)['key']();
```

New Syntax

One of the primary benefits of uniform variable syntax, other than consistency, is that it enables many new combinations of syntax—including some that are not yet supported by the language but that are no longer blocked by a parser that can't handle the syntax.

New Combinations

There are many new combinations of existing syntax that are now available to use, as shown in Example 3-2, including the ability to dereference characters within strings returned by functions.

Example 3-2. Newly supported syntax combinations

```
// Call a closure inside an array returned by another closure
$foo()['bar']();

// Call a property by dereferencing an array literal
[$obj1, $obj2][0]->prop;

// Access a character by index in a returned string
getStr(){0};
```

Nested double colons

Additionally, PHP 7 now supports nested double colons, `::`, at least in some cases, as shown in Example 3-3.

Example 3-3. Some examples of nested double colons

```
// Access a static property on a string class name
// or object inside an array
$foo['bar']::$baz;

// Access a static property on a string class name or object
// returned by a static method call on a string class name
// or object
$foo::bar()::$baz;

// Call a static method on a string class or object returned by
// an instance method call
$foo->bar()::baz();
```

There are still a number of ambiguous cases, however, that cannot be resolved, even with uniform variable syntax, and even when adding parentheses and braces, as shown in Example 3-4.

Example 3-4. Unsupported ambiguous nested double colons

```
$foo = 'Foo';
$class = 'CLASS';
$constant = 'BAR';
```

```
echo $foo::$class::$constant;
echo $foo::{$class}::$constant;
echo $foo::{"$class"}::$constant;
echo $foo::{"$class"}::{"$constant"};
echo $foo::CLASS::$constant;
echo $foo::CLASS::{"$constant"};
echo $foo::($class)::($constant);
```

Nested Method and Function Calls

Furthermore, you can now nest method and function calls—or any callables—by doubling up on parentheses, as shown in Example 3-5.

NOTE

Callables

In PHP 5.4, `callable` was added as a type hint for any value that could be called dynamically. This includes:

- Closures
- String function names
- Objects that define the `__invoke()` magic method
- An array containing a string class name and a method to call for static method calls (e.g. `[`*`className`*`, 'staticMethod]`)
- An array containing an object and a method to call for instance method calls (e.g., `[$object,` *`method`*`]`)

Example 3-5. Nested method and function calls

```
// Call a callable returned by a function
foo()();

// Call a callable returned by an instance method
$foo->bar()();

// Call a callable returned by a static method
Foo::bar()();

// Call a callable return another callable
$foo()();
```

Arbitrary Expression Dereferencing

Starting in PHP 5.4 with dereferencing arrays returned by methods and functions, and continued in 5.5 with dereferencing literal arrays, in PHP 7, you can now dereference *any* valid expression enclosed with parentheses. You can see some examples of arbitrary expression dereferencing in Example 3-6.

Example 3-6. Arbitrary expression dereferencing

```
// Access an array key
(expression)['foo'];

// Access a property
(expression)->foo;

// Call a method
(expression)->foo();

// Access a static property
(expression)::$foo;

// Call a static method
(expression)::foo();

// Call a callable
(expression)();

// Access a character
(expression){0};
```

This finally allows us to call a closure when we define it, and call a callable within an object property, as you can see in Example 3-7.

Example 3-7. Dereferencing callables

```
// Define and immediately call a closure without assignment
(function() { /* ... */ })();

// Call a callable within an object property
($obj->callable)();
```

Dereferencing scalars

PHP 7 also has some new dereferencing for scalars, in particular the ability to call methods using array-notation callables, as well as the less than useful ability to use scalar strings as class names. Some examples can be seen in Example 3-8.

Example 3-8. Dereferencing scalars

```
// Call a dynamic static method
["className", "staticMethod"]();

// Call a dynamic instance method
[$object, "method"]();

// Use a string scalar as a class name
'className'::staticMethod();
```

Future Syntax

Additionally, as noted previously, the addition of universal variable syntax also allows us to look forward at new possibilities for the language. The one possibility that most people are excited about is the prospect of object scalars—that is, the ability to call methods that act directly upon a scalar value. While they are not yet possible in PHP 7.0, some examples of potential syntax can be seen in Example 3-9.

Example 3-9. Possible future syntax

```
// Object scalars — method calls on scalar values
"string"->toLower();
```

Backward Compatibility Issues

There was one casualty that will now be a parse error, rather than just interpreted differently, and that is when you mix variable-variables and the `global` keyword. With PHP 7, `global` will now only take unambiguous variables. You can see the old unsupported syntax and the unambiguous alternative in Example 3-10.

Example 3-10. Changes to the global keyword

```
global $$foo->bar; // Now a parse error

// instead make sure to add braces to make it unambiguous
global ${$foo->bar};
```

Summary

While Uniform Variable Syntax finally brings some much needed consistency to PHP—even if we didn't know it!—it also introduces what are probably the hardest bugs to detect when upgrading to PHP.

Thankfully, it mostly affects variable-variables and other more complex variable syntax that have long been considered poor practice and are rarely used.

CHAPTER 4

Basic Language Changes

PHP 7.0 introduces numerous small changes to the language, new operators, functions, and changes to existing functions and constructs.

Operators

PHP 7.0 adds two new operators, the null coalesce operator and the combined comparison operator.

Null Coalesce Operator

The new null coalesce operator `??` will return the left operand if it is not null; otherwise, it returns the right operand. The most interesting part of this operator is that it *will not emit a notice* if the operand is a nonexistent variable—similar to `isset()`.

Effectively, this is a shortcut for `isset()` combined with a ternary for assignment. Example 4-1 shows this new, more compact syntax compared to the older syntax.

Example 4-1. Null coalesce operator

```
$foo = isset($bar) ? $bar : $baz;

$foo = $bar ?? $baz;
```

The two lines in the preceding example are functionally identical.

You can also nest the operator, and it will return the first non-null (or the last argument), as shown in Example 4-2.

Example 4-2. Nested null coalesce operators

```
$config =    $config ??
             $this->config ??
             static::$defaultConfig; ❶
```

❶ This shows a common fall-through scenario, whereby a local configuration may be set in `$config`, otherwise fallback to the instance level `$this->config`, or finally, fall back to the static defaults `static::$defaultConfig`.

Combined Comparison Operator

Affectionately called the *spaceship operator*, the combined comparison operator (`<=>`) is the first *trinary operator* in PHP.

This means that rather than return a simple binary `true` or `false`, it can actually return three distinct values:

- `-1` if the left operand is *less* than the right operand
- `0` if the operands are *equal*
- `+1` if the left operand is *greater* than the right operand

This is frequently used for sorting items, for example, using the `usort()` function with a callback.

The following two functions are identical, the first using PHP 5.x syntax, and the second using the new combined comparison operator. You can compare the older, less compact syntax in Example 4-3.

Example 4-3. Sorting with the combined comparison operator

```
// Pre Spacefaring^W PHP 7
function order_func_traditional($a, $b) {
    return ($a < $b) ? -1 : (($a > $b) ? 1 : 0);
}

// Post PHP 7
function order_func_spaceship($a, $b) {
    return $a <=> $b;
}
```

Constant Arrays

Up until now, constants defined with `define()` could only contain scalar values. With PHP 7.0, they have been updated to match constants defined by `const`, allowing you to set them to constant arrays. Example 4-4 demonstrates this new capability.

Example 4-4. Constant arrays using define()

```
define('FOO', [
    'bar' => 'baz',
    'bat' => 'qux'
]);

echo FOO['bar']; ❶
```

❶ This will output `baz`.

Unpacking Objects Using list()

The `list` construct will now allow you to unpack objects that implement the `\ArrayAccess` interface, allowing you to use them just like arrays, as shown in Example 4-5.

Example 4-5. Unpacking objects using list()

```
$object = new \ArrayObject(json_decode($json)); ❶
list($foo, $bar, $baz) = $object;
```

❶ `\ArrayObject` implements `\ArrayAccess` and can take an array, or an object, so no matter which `json_decode()` returns, we can now use it with `list()`.

New Functions

Several new functions have been added with this release.

Integer Division

The new `intdiv()` performs integer division, effectively the inverse of the *modulo* operator (`%`). Example 4-6 shows an example of this new function.

Example 4-6. Integer division

```
intdiv(8, 3); ❶
```

❶ Returns two.

Regular Expressions

As shown in Chapter 2, `ext/ereg` has been removed, but this isn't the only change to regular expressions in PHP 7.0.

This release adds a new `preg_replace_callback_array()` function that makes it much nicer to perform a number of regular expression replacements with different callbacks for each. As shown in Example 4-7, you pass in an array with regular expressions for keys, and closures—or other callables—that accept an array of matches. All matches will be replaced with the return value from the callable.

Example 4-7. Multiple replace callbacks

```
$header = "X-Custom-Header: foo\nbar";

$normalizedHeader = preg_replace_callback_array(
    [
        "/(.*?):/" => function($matches) {
            return strtolower($matches[0]);
        },
        "/\s+/" => function($matches) {
            return "-";
        }
    ],
    $header
); ❶
```

❶ This will transform the input header to `x-custom-header:-foo-bar`.

The other change to the PCRE functions is the removal of the `/e` modifier used with `preg_replace()`. This modifier allowed you to use code that would be evaluated against matches to create the replacement value. You should use `preg_replace_callback()` or the new `preg_replace_callback_array()` instead.

Cryptographically Secure Values

Traditionally, we've resorted to either very poor sources of randomness or openssl to generate cryptographically secure integers or strings.

PHP 7.0 adds two new simple functions, `random_bytes()` and `random_int()` to solve this problem in a platform-independent way.

NOTE

CSPRNG functions

Collectively, these form the *CSPRNG* functions. CSPRNG stands for cryptographically secure psuedorandom number generator.

As you might expect from the name, `random_int()` will return a random integer from a range, while `random_bytes()` will return a random string of a given length as shown in Example 4-8.

Example 4-8. Generating cryptographically secure random values

```
random_bytes(16); ❶
random_int(0, 10000); ❷
```

❶ Returns 16 random bytes in binary format. You will probably want to pass the result to `bin2hex()` before using it.

❷ Returns a random number between `0` and `10000`.

Function Changes

There have also been a few changes to *existing* functions.

Sessions

It is now possible to pass an array of INI settings to `session_start()`.

A new setting `session.lazy_write` has also been added. This new setting, which is *enabled by default*, will mean that session data is only rewritten if there has been a change to it.

You can disable this new setting as shown in Example 4-9.

Example 4-9. Changing session configuration on call

```
session_start([
    'use_strict' => true,
    'lazy_write' => false
]);
```

Filtered unserialize()

In an attempt to enhance security, it is now possible to filter which classes will be instantiated when unserializing using `unserialize()`.

This is done by adding a second argument that takes an array of options, of which there is currently only one, `allowed_classes`.

You can pass in one of three values for the `allowed_classes` option:

- `false` will instantiate *all* objects as `__PHP_Incomplete_Class` object instead.
- An *array* of class names will instantiate those as-is and return `__PHP_Incomplete_Class` for any others.
- `true` will result in the same behavior we've always had, and all objects will be instantiated they are.

Move Up Multiple Levels with dirname()

The `dirname()` function can now accept a second parameter to set how many levels up it will go, meaning you can avoid nesting as shown in Example 4-10.

Example 4-10. Moving up two directories using dirname()

```
$path = '/foo/bar/bat/baz';
dirname($path, 2); ❶
```

❶ This will return `/foo/bar`.

Salts Deprecated in password_hash()

The `salt` option for `password_hash()` has been deprecated, and it will now emit an `E_DEPRECATED` when used.

Summary

While these changes are small, they all add up to a nicer developer experience, and make the language more consistent.

CHAPTER 5
Expectations and Assertions

PHP has had an `assert()` function since PHP 4; it gives you the ability to add sanity-checks to your code. `assert()` is intended for *development* use only, and it can be easily enabled and disabled using `assert_options()` or the `assert.active` INI setting.

To use assertions, you pass in either an expression or a string as the first argument. If you pass in a string, it is evaluated by the `assert()` function as code. If the expression result, or the result of evaluating the string evaluates to `false`, then a *warning* is raised.

Example 5-1. Using assertions

```
assert('$user instanceof \MyProject\User');
assert($user instanceof \MyProject\User);
```

Single versus double quotes

Ensure that you use single quotes; otherwise, variables will be interpolated.

In Example 5-1, we see the same assertion using a string and an expression. If either of these evaluates to false, then a warning is raised:

```
Warning: assert(): Assertion "$user instanceof \MyProject
\User" failed in <file> on line <num>
```

TIP

Using string assertions

While `eval()` is typically frowned upon, if you pass a string into `assert()` and assertions are *disabled*, the string is not evaluated at all. *Using strings is considered best practice.*

Expectations

With PHP 7, `assert()` has been expanded by the Expectations RFC, allowing for so-called *zero-cost* assertions.

With this change, you not only *disable* assertions, but you can also remove all overhead entirely. With this setting, assertions are not compiled, regardless of string or expression arguments, and therefore have zero impact on performance/execution. This is different than just disabling assertions, which will still result in expressions being evaluated (potentially affecting execution) and will just skip the call.

This is done by changing the `zend.assertions` INI setting, as shown in Example 5-2.

Example 5-2. Enable/disable assertions

```
zend.assertions  = 1  ❶
zend.assertions  = 0  ❷
zend.assertions  = -1 ❸
```

❶ Enable assertions.

❷ Disable assertions and stop string evaluations.

❸ Zero-cost assertions.

Additionally, you can now have assertions throw an *exception* instead of a warning when the assertion fails. Again, this is an INI setting. Simply set `assert.exceptions` to `1` to throw exceptions or `0` (the default) to emit backward-compatible warnings instead.

NOTE

Further details

For more on assert exceptions, see Chapter 6.

The final change is the addition of a second argument, which allows you to specify a custom error message—as shown in Example 5-3—or an instance of an `Exception`.

Example 5-3. Specifying a custom assertion error message

```
assert(
    '$user instanceof \MyProject\User',
    'user was not a User object'
);
```

When you specify this custom message, it is shown instead of the expression on failure:

```
    Warning: assert(): user was not a User object failed in <file>
    on line <num>
```

If you enable exceptions, the custom message will be used as the exception message, or if you specify an instance of an `Exception`, it will be thrown instead on failure.

Summary

With the addition of zero-cost assertions, you finally have a lightweight way to add sanity checking to your code during development, without impacting production.

While assertions are not for everybody, they can be a first step on the road to adding unit testing to your application.

CHAPTER 6

Error Handling

Errors and error handling in PHP has remained pretty much unchanged since PHP 4, except for the additions of `E_STRICT` in PHP 5.0, `E_RECOVERABLE_ERROR` in PHP 5.2, and `E_DEPRECATED` in PHP 5.3.

Despite adding exceptions to PHP 5 and seeing limited application within PHP (e.g., `ext/pdo` and `ext/spl`), numerous attempts to replace traditional errors with them have failed. In PHP 7.0, this has finally changed, with the "Exceptions in the engine (for PHP7)" RFC.

Exceptions on Constructor Failure

Prior to PHP 7.0, if an *internal* class failed to instantiate properly, either a `null` or an unusable object was returned.

With PHP 7.0, *all* internal classes will throw an exception on `__construct()` failure. The type of exception will differ on the object being instantiated and the reason for failure—as shown in Example 6-1. If you pass in an argument of the incorrect type, a `\TypeError` is thrown (see "\TypeError" on page 42).

Example 6-1. Consistent constructor failure

```
try {
    new MessageFormatter('en_US', null);
} catch (\IntlException $e) {
    ❶
}
```

❶ This will result in an `\IntlException` with the message `Constructor failed`.

Engine Exceptions

With PHP 7.0, almost all fatal errors and catchable fatal errors are now *engine exceptions*. This is possible because an *uncaught* exception still results in a traditional fatal error, ensuring that the change is *mostly* backward compatible.

NOTE

Other error types

Changing other (nonfatal) types of errors to exceptions would mean fatal errors for things like notices and warning when the exceptions are not handled. This would not be backward compatible.

With this change, we get a number of benefits, the most obvious of which is that we can now handle fatal errors in our code using `try…catch` blocks. However, there are a number of other benefits:

- `finally` blocks are called.
- Object destructors (`__destruct()`) are called.
- Callbacks registered with `register_shutdown_function()` are called.
- Catchable-fatal errors are much easier to handle.
- As with all exceptions, it will have a stack trace, making it easier to debug.

Exception Hierarchy

For backward compatibility, we have to ensure that existing catch-all `catch` blocks (`catch (\Exception $e) { }`) do not catch the new engine exceptions and will therefore be a fatal error as before.

To solve this, the new exceptions *do not extend* the original base `\Exception` class. The new exceptions are instead an `\Error` exception, which is a *sibling* to the original `\Exception` class. All other engine exceptions extend from the new `\Error` exception class.

Additionally, a new `\Throwable` interface was added with the passing of the appropriately named "Throwable Interface" RFC, which both `\Exception` and `\Error` implement.

With these new changes, we have a new exception hierarchy as you can see in Example 6-2.

Example 6-2. New exception hierarchy for PHP 7.0+

```
\Throwable ❶
├── \Exception (implements \Throwable) ❷
│   ├── \LogicException
│   │   ├── \BadFunctionCallException
│   │   │   └── \BadMethodCallException
│   │   ├── \DomainException
│   │   ├── \InvalidArgumentException
│   │   ├── \LengthException
│   │   └── \OutOfRangeException
│   └── \RuntimeException
│       ├── \OutOfBoundsException
│       ├── \OverflowException
│       ├── \RangeException
│       ├── \UnderflowException
│       └── \UnexpectedValueException
└── \Error (implements \Throwable) ❸
    ├── \AssertionError
    ├── \ArithmeticError
    ├── \DivisionByZeroError
    ├── \ParseError
    └── \TypeError
```

❶ The new `\Throwable` interface is the top-level parent.

❷ The original base exception `\Exception` now implements `\Throwable`.

❸ Engine exceptions use the new `\Error` exception.

With this change, if you want to do a true catch-all, you have to use `catch (\Throwable $e) { }` instead.

Error Exceptions

As you can see in the exception hierarchy, there are four new error exceptions, each one used for a different purpose.

\Error

Standard PHP fatal and catchable-fatal are now thrown as `\Error` exceptions. These will continue to cause a "traditional" fatal error if they are uncaught. For example, calling a non-existant function will result in an `\Error` exception with the message `Fatal error: Uncaught Error: Call to undefined function non_existant_function()`.

Catching `\Error` exceptions is done the same was as regular exceptions, as shown in Example 6-3.

Example 6-3. Error exceptions

```
try {
    non_existent_function();
} catch (\Error $e) {
    // handle error
}
```

\AssertionError

With the enhancements to assertions (see Chapter 5), if you set `assert.exception` to `1` in your *php.ini* (or via `ini_set()`), an exception is thrown when the assertion fails.

These exceptions are `\AssertionError` exceptions, as shown in Example 6-4.

Example 6-4. AssertionError exceptions

```
try {
    ini_set('assert.exception', 1);
    assert('true === false', 'Assertion failed');
} catch (\AssertionError $e) {
    ❶
}
```

❶ This will result in an `\AssertionError` exception with whatever you pass in as the second argument for `assert()` as the message, in this case: `Assertion failed`.

Assertion messages

If you do not pass a message as the second argument to `assert()`, the `\AssertionError` exception with have *no* message; however, its stack trace will reveal its origins and the failed assertion.

\ArithmeticError and \DivisionByZeroError

Two exceptions have been added to handle issues when performing arithmetic. The first, `\ArithmeticError`, will be thrown whenever an error occurs when performing mathematical operations—for example, if it results in an out-of-bound integer, or if you try to bit shift by a negative amount. You can see this in Example 6-5.

Example 6-5. ArithmeticError exceptions

```
try {
    1 >> -1;
} catch (\ArithmeticError $e) {
    ❶
}
```

❶ This will result in an `\ArithmeticError` exception with a message of `Bit shift by negative number`.

A second, more specific `\DivisionByZeroError` has also been added that is thrown whenever you attempt to divide by zero—this includes using the `/` and `%` operators and the `intdiv()` function, as shown in Example 6-6.

Example 6-6. DivisionByZeroError exceptions

```
try {
    10 % 0;
} catch (\DivisionByZeroError $e) {
    ❶
}
```

❶ This will result in a `\DivisionByZeroError` exception with the message `Modulo by zero`.

\ParseError

You can now handle parse errors in `include` and `require` statements, and `eval()` parse errors, as both now throw `\ParseError` exceptions as shown in Example 6-7.

Example 6-7. ParseError exceptions

```php
try {
    include 'parse-error.php'; ❶
} catch (\ParseError $e) {
    ❷
}
```

❶ Try to include a file containing a parse error.

❷ This will result in a `\ParseError` exception with a message like `syntax error, unexpected $foo (T_VARIABLE), expecting identifier (T_STRING) or (`.

\TypeError

With the introduction of scalar and (especially) strict types in PHP 7.0 (see Chapter 11), these will also throw exceptions when a type mismatch occurs. It is important to understand that this does not apply just to scalar type hints, but also to traditional type hints, such as class/interface names, `callable` and `array`. Example 6-8 shows the new `\TypeError` in action.

Example 6-8. TypeError exceptions

```php
function example(callable $callback)
{
    return $callback();
}

try {
    example(new stdClass);
} catch (\TypeError $e) {
    ❶
}
```

❶ This will result in a `\TypeError` exception with the message `Argument 1 passed to example() must be callable, object given, called in <file> on line <num>`.

Catchable Fatal Errors

Another important change in PHP 7.0 is with catchable fatal errors. Previously, these could have been caught and handled using `set_error_handler()`. However, with PHP 7.0, they are now `\Error` exceptions, which, because an uncaught exception is still a real fatal error, will no longer be catchable in `set_error_handler()`.

This is a backward-compatible break and means that to work in both PHP 5.x and 7, you need to use both `set_error_handler()` and a `try…catch` block.

\Throwable and Userland

With `\Throwable` and the new exception hierarchy, it would make sense that we could create our own branches in the exception hierarchy for completely custom exceptions by simply implementing the `\Throwable` interface.

Unfortunately, due to the fact that exceptions are magical under the hood, to be able to do things like capture line/file and stack trace information, you must still extend either `\Exception` or `\Error`, and cannot directly implement `\Throwable` alone.

Trying to implement `\Throwable` as shown in Example 6-9 will result in a fatal error.

Example 6-9. Implementing \Throwable

```
Fatal error: Class MyException cannot implement
interface Throwable, extend Exception or Error instead
```

However, this is not the full story. You *can* extend `\Throwable` and then—while still extending `\Error` or `\Exception`—you can implement your extended interface, as shown in Example 6-10.

Example 6-10. Defining a custom interface

```
namespace My\Library;
interface MyExceptionInteface extends \Throwable {
  public function someMethod();

  public function someOtherMethod();
}

use \My\Library\MyExceptionInteface;
class MyException
    extends \Exception implements MyExceptionInterface
{
    // implement interface methods
}
```

Calling Methods on non-objects

Prior to PHP 7.0, if you were to attempt to call a method on a variable that did not contain an object, it would fatal error, with the message `Call to a member function method() on a non-object`.

It was initially decided that this should be a catchable fatal error; however, the addition of engine exceptions supersedes this, and it will now throw an `\Error` exception, which can be handled using `try…catch`.

Summary

While these changes will have a large impact on how you write your code—allowing you to gracefully handle almost all previously fatal errors—it has been implemented in an almost complete backward-compatible way.

In fact, if you are not using `set_error_handler()` to handle catchable fatal errors, you should not need to make any changes at all to your applications.

CHAPTER 7

Unicode Enhancements

While PHP 6 never emerged with its promise of Unicode greatness, that doesn't mean PHP doesn't support Unicode. PHP 7.0 still includes some enhancements to its existing Unicode support.

These additions come from two different RFCs. These are "Unicode Codepoint Escape Syntax" RFC and the "IntlChar class" RFC.

Unicode Codepoint Escape Syntax

A simple, yet powerful change allows us to easily embed complex Unicode characters in (double quoted) strings. Similar to other languages, this uses the `\u` escape sequence; however, in other languages, this is then followed directly by four hexadecimal digits to represent the character. For PHP 7.0, we instead follow it with an arbitrary number of hexadecimal numbers inside curly braces (`{}`).

The reason for this is that using a fixed length of four hexadecimal digits would limit us to the *Basic Multilingual Plane*, or BMP (`U+0000` to `U+FFFF`), while many useful characters—in particular, emojis—reside outside of the BMP and require six hexadecimal digits.

Requiring six digits would mean that the majority of characters used would need to be left padded with two zeros (e.g., `\u001000`). Allowing arbitrary lengths would introduce potential ambiguity for Unicode points followed by other numbers, particularly if you were expecting the same behavior as the other languages with four digits.

With the curly braces, you can specify any number of hexadecimal digits, without the need to left pad, and you can completely avoid ambiguity.

This means that you can use `\u{FF}` or the more traditional `\u{00FF}`. We can also express those non-BNP characters such as "🐢" with `\u{1F422}`.

New Internationalization Features

Most *Internationalization* (I18N) features reside in the `ext/intl` extension, which has a new addition in PHP 7.0. The `IntlChar` class.

NOTE

Internationalization = I18N

Internationalization is often shortened to I18N because there are 18 characters between the first and last characters, and it's a lot easier to write! The same is commonly done with *Localization* (L10N) and also *Accessibility* (a11y).

Working with Unicode Characters

The new `IntlChar` class provides a number of static methods that provide access to information about Unicode characters.

To retrieve a character name, you use the new `IntlChar::charName()` method, passing in the Unicode character, as shown in Example 7-1.

Example 7-1. Retrieving character names

```
IntlChar::charName("\u{1F422}"); ❶
```

❶ Will return `TURTLE`.

You can also use it to detect the type of character you are dealing with, for example, digits or punctuation.

Example 7-2. Detecting character types

```
$char = "\u{F1}"; ❶
IntlChar::isAlpha($char); ❷
```

```
IntlChar::isAlnum($char); ❸
IntlChar::isPunct($char); ❹
```

❶ `\u{F1}` represents the ñ character.

❷ Returns `true`.

❸ Returns `true`.

❹ Returns `false`.

Summary

These changes bring a few of the new Unicode features to PHP 7.0, but this chapter barely scratches the surface of what they provide. However as you can see, the future of Unicode in PHP is alive and well.

CHAPTER 8

Closure Enhancements

Closures have been around since PHP 5.3, and with PHP 5.4, the ability to access an early-bound `$this` within them was added. In addition to this, the `Closure` class went from being an implementation detail to being probably the easiest way to break PHP's object model.

PHP 7.0 makes this even easier.

Bind Closure On Call

With the addition of `$this`, `Closure` gained two methods, the instance method `Closure->bindTo()` and the static method `Closure::bind()`.

Both of these functions do the same thing, but the first is called on the closure instance itself, while the static version must be passed the closure itself as the first argument. They both then take two arguments: the first is an object that `$this` will then point to, and the second is the *scope* from which to access `$this`.

This second one is important, because it means that you can change the scope to that of another class allowing you to call its private and protected methods and properties.

Both of the functions will then return a *clone* of the closure with the new `$this` and scope, rather than modifying the original.

The "Closure::call" RFC gives us an easier way to achieve the most common case, without needing to clone the object.

With PHP 7.0, the `Closure` class has a new instance method, `Closure->call()`, which takes an object as its first argument, to which `$this` is bound, and to which the scope is set, and then it calls the closure passing through any additional arguments passed to `Closure->call()`, as shown in Example 8-1.

Example 8-1. Bind closure on call

```
class HelloWorld {
    private $greeting = "Hello";
}

$closure = function($whom) {
    echo $this->greeting . ' ' . $whom;
};

$obj = new HelloWorld();
$closure->call($obj, 'World');
```

In the previous example, the closure `echo`s a nonexistent `$this->greeting` property.

To resolve this, we call the closure using `Closure->call()` passing in an instance of the `HelloWorld` object. This changes `$this` to be that instance, and the calling scope to the `HelloWorld` class, allowing access to the private `$this->greeting` property.

Summary

This small but powerful change makes working with closures both much more convenient and more dangerous. Proceed with caution!

CHAPTER 9

Generator Enhancements

Generators were my favorite feature in PHP 5.5, and the new changes in PHP 7.0 are definitely close to the top of my list again.

Generator Return Values

In PHP 5.5, if a generator function has a `return` statement followed by an expression (e.g., `return true;`), it would result in a parse error.

With the "Generator Return Expressions" RFC, this has now changed. As with the `Closure` class between 5.3 and 5.4, the `Generator` class that provides the magic for generators has moved from an implementation detail to a concrete implementation with the addition of the `Generator->getReturn()` method.

Not only will it no longer cause a parse error, but the `Generator->getReturn()` method will allow you to retrieve the value returned by any `return` statement inside the generator. Example 9-1 shows this new functionality.

Example 9-1. Retrieving generator return values

```
function helloGoodbye() {
    yield "Hello";
    yield " ";
    yield "World!";

    return "Goodbye Moon!";
}

$gen = helloGoodbye();

foreach ($gen as $value) {
    echo $value; ❶
}

echo $gen->getReturn(); ❷
```

❶ Outputs `Hello` on iteration one, a space on iteration two, and `World!` on iteration three.

❷ Will output `Goodbye Moon!`

Generator Delegation

While generator return values are neat, generator delegation is by far the more exciting of the enhancements being made.

Generator delegation allows a generator to `yield` other generators, objects that implement the `\Traversable` interface, and *arrays*, and they will be iterated on in turn. These are also known as *sub-generators*.

This is done using the new `yield from <thing to iterate>` syntax, and has the effect of *flattening* the subgenerators, so that the iterating mechanism has no knowledge of the delegation.

Example 9-2. Using generator delegation

```
function hello() {
     yield "Hello";
     yield " ";
     yield "World!";

     yield from goodbye(); ❶
}

function goodbye() {
     yield "Goodbye";
     yield " ";
     yield "Moon!";
}

$gen = hello();
foreach ($gen as $value) {
    echo $value; ❷
}
```

❶ We `yield from` another generator, `goodbye()`.

❷ This will output `Hello` on the first iteration, a space on the second, `World!` on the third, `Goodbye` on the fourth, another space on the fifth, and finally `Moon!` on the sixth.

If you wanted to, you could have further `yield` or `yield from` statements after the initial delegation, and any generator that you delegate to can, in turn, delegate, and the structure will be flattened again.

Summary

Despite these additions, generators still manage to be a lightweight alternative to Iterators. If you haven't started using generators yet, now is a great time to start looking into them.

CHAPTER 10

Changes in Object-Oriented Programming

Considering that PHP 5.0 and almost every subsequent release were *primarily* about changes to the object model, it might seem odd that there are hardly any changes to it in PHP 7.0.

But, few though they may be, PHP 7.0 still brings some new powerful OOP features to the table.

Context-Sensitive Lexer

PHP 7.0 introduces the *context-sensitive lexer*, which allows for the use of keywords as names for properties, methods, and constants within classes, interfaces, and traits.

What this means is that PHP goes from having 64 reserved keywords to having just one—`class`—and only in the class constant context.

You may now use any of the following keywords as property, function, and constant names:

`callable`	`and`	`include`	`function`
`trait`	`global`	`include_once`	`if`
`extends`	`goto`	`throw`	`endswitch`

`implements`	`instanceof`	`array`	`finally`
`static`	`insteadof`	`print`	`for`
`abstract`	`interface`	`echo`	`foreach`
`final`	`namespace`	`require`	`declare`
`public`	`new`	`require_once`	`case`
`protected`	`or`	`return`	`do`
`private`	`xor`	`else`	`while`
`const`	`try`	`elseif`	`as`
`enddeclare`	`use`	`default`	`catch`
`endfor`	`var`	`break`	`die`
`endforeach`	`exit`	`continue`	`self`
`endif`	`list`	`switch`	`parent`
`endwhile`	`clone`	`yield`	`class`

The *only* exception is that you still cannot use `const class`, because it clashes with the fully qualified class name magic constant added in PHP 5.5.

PHP 4 Constructors Deprecated

While some called for them to be removed, the old PHP 4 constructors have instead been marked as deprecated in PHP 7.0.

Because removing PHP 4 constructors would be a backward-compatible break, any potential removal would happen no earlier than PHP 8.0, meaning this will probably not affect most people much for quite a while.

As a refresher, PHP 4 constructors have the same name as the class in which they are defined. Meaning a class `foo` has a PHP 4 constructor named `foo()`.

The ability to use PHP 4 constructors was disallowed inside of namespaced classes when namespaces were introduced (PHP 5.3).

Group Use Declarations

In an effort to reduce duplication and simplify your code, the "Group Use Declarations" RFC was one of the more polarizing changes that managed to squeak in—passing with just 67 percent of the vote that required a two-thirds majority.

Group use declarations allow us to deduplicate the common prefixes in our `use` statements and just specify the unique parts within a block (`{}`) instead.

The way that you group statements is up to you. As you can see, Example 10-1 can be written differently, as shown in Example 10-2.

Example 10-1. Using group use statements

```
// Original
use Framework\Component\ClassA;
use Framework\Component\ClassB as ClassC;
use Framework\OtherComponent\ClassD;

// With group use statements
use Framework\{
     Component\ClassA,
     Component\ClassB as ClassC,
     OtherComponent\ClassD
};
```

Example 10-2. Alternative organization of use statements

```
use Framework\Component\{
     Component\ClassA,
     Component\ClassB as ClassC
};
Use Framework\OtherComponent\ClassD;
```

Also, if you want to import functions or constants—a feature that was added in PHP 5.6—you simply prefix the import line with `function` or `const`, respectively, as shown in Example 10-3.

Example 10-3. Importing functions and constants with group use statements

```
use Framework\Component\{
     SubComponent\ClassA,
     function OtherComponent\someFunction,
     const OtherComponent\SOME_CONSTANT
};
```

Anonymous Classes

The addition of anonymous classes gives us closure-like capabilities, but for objects—with all the associated methods and properties that go along with that.

Anonymous classes are probably as close as we will get to an object literal syntax in PHP.

To create an anonymous class, you simple combine the `new class($constructor, $args)` followed by a standard class definition. An anonymous class is always instantiated during creation, giving you an object of that class.

Example 10-4 shows a simple example of creating an anonymous class.

Example 10-4. Creating an anonymous class

```
$object = new class("bar") {
    public $foo;

    public function __construct($arg)
    {
        $this->foo = $arg;
    }
};
```

The preceding example will create an object with a `__construct()` method, which has already been called with the argument `bar`, and a property `$foo`, which has been assigned the value of that argument by the constructor.

```
object(class@anonymous)#1 (1) {
  ["foo"]=>
  string(3) "bar"
}
```

Anonymous classes can be namespaced, and they support inheritance, traits, and interfaces, using the same syntax as regular classes, as shown in Example 10-5.

Example 10-5. Anonymous class feature support

```
namespace MyProject\Component;

$object = new class ($args) extends Foo implements Bar {
        use Bat;
};
```

Anonymous Class Names

It may seem silly to think about "anonymous" classes having names, but internally every single one has a unique name, which is based on the memory address of the operation that created it.

These look something like `class@0x7fa77f271bd0`.

This is important because if you were to create a class within a loop, the instruction has the same memory address each iteration and therefore only one class is defined—but many instances of it are created.

This means that if the resulting object of two iterations have the same property values, they will be equal (== but not identical ===).

However, even if you define another anonymous class with *exactly* the same structure somewhere else in the code, it will have a different name based on its memory address and therefore not be equal. Example 10-6 demonstrates this behavior.

Example 10-6. Anonymous classes created in loops

```
$objects = [];
foreach (["foo", "foo", "bar"] as $value) {
        $objects[] = new class($value) {

            public $value;

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

```
$objects[] = new class("foo") {

    public $value;

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

In Example 10-6, we create three instances of an anonymous class inside a `foreach` loop. The first two are passed `foo` as their constructor, while the third is passed `bar`.

We then create a fourth new anonymous class with the same definition and again pass in `foo` to the constructor.

Because of this, the first and second objects—`$objects[0]` and `$objects[1]`, respectively—are equal, but not identical. However, neither of these two objects is equal to the third object—`$objects[2]`.

They also will not be equal to the fourth object—`$objects[3]`—because it was defined outside of the loop and, despite its identical structure and value, it is a different class with a different name.

Summary

While the focus in PHP 7 is not sweeping changes to the object model, the changes that have been added are not insubstantial—in particular, anonymous classes opens up a number of new architecture possiblities.

These changes will hopefully make for more robust, easier to write, object-oriented code.

CHAPTER 11

Type Hints

The most polarizing—and exciting—new feature added in PHP 7.0 is without a doubt the addition of *scalar type hints*. There have been many RFCs for different variations of this feature over the years, and finally, with PHP 7, we have an RFC that has passed—albeit after numerous revisions and authors. Unfortunately, this issue was so contentious that it even spilled out of the mailing lists into public social networks before the current implementation was accepted.

Scalar type hints aren't the only new feature related to type hints, however. As you will see in this chapter, we also have return types, strict types, and more.

Scalar Type Hints

Alongside the introduction of a robust object model in PHP 5.0, we also got the first type hints: class/interface hints. The ability to require a function/method parameter was an `instanceof` a given class or interface.

PHP 5.1 introduced the `array` type hint, while PHP 5.4 added `callable`—the ability to type hint on anything that was a valid callback (e.g., closures, function names, etc.).

PHP 7.0 adds the ability to hint on scalar types:

- `bool`: Boolean values (i.e., `true`/`false`)
- `float`: Floating point numbers (e.g., `4.35`)

- `int`: Integer numbers (e.g., `123`)
- `string`: Strings (e.g., `foo`)

Example 11-1 shows each of these new hints in use.

Example 11-1. Scalar type hinting a function

```
function hinted(bool $a, float $b, int $c, string $c)
{

}

hinted(true, 4.35, 123, "foo");
```

Coercive Types

By default, PHP 7.0 will use *coercive* type hints, which means that it will *attempt* to convert to the specified type, and, if possible, will do so without complaint. This is the *nonstrict* option. Coercive types come with some gotchas, most importantly, precision loss.

Example 11-2 shows type coercion in use.

Example 11-2. Type coercion

```
function sendHttpStatus(int $statusCode, string $message)
{
    header('HTTP/1.0 ' .$statusCode. ' ' .$message);
}


sendHttpStatus(404, "File Not Found"); ❶
sendHttpStatus("403", "OK");
 ❷
```

❶ Integer and string passed, no coercion occurs.

❷ Integer string passed, coerced to `int(403)`, string `OK` is left untouched.

Precision Loss

Because of this coercion, you may unknowingly lose data when passing it into a function. The simplest example is passing a `float` into a function that requires an `int`, as shown in Example 11-3.

Example 11-3. Float passed into integer hinted argument

```
function add(int $a, int $b)
{
    return $a + $b;
}

add(897.23481, 361.53); ❶
```

❶ Both floats are turned into integers, `897` and `361`. This is the same behavior as casting, e.g., `(int) 361.53` will return `361`.

However, there are other precision losses, passing a string that *starts* with a numeric and ends in non-numerics to a `float`/`int` will be cast to that type, dropping the non-numeric characters. A string that starts with (or only contains) non-numerics will cause a `\TypeError` exception to be thrown, as it cannot be coerced.

TIP You can read more on `\TypeError` exceptions in Chapter 6.

Also, passing integers in to `float` hinted arguments *may* cause precision loss as 64-bit integers greater than 2^{53} cannot be represented exactly as a float.

By default, hint coercion will result in the following:

- `int(1)` ⇒ `function(float $foo)` ⇒ `float(1.0)`
- `float(1.5)` ⇒ `function(int $foo)` ⇒ `int(1)`
- `string("100")` ⇒ `function (int $foo)` ⇒ `int(100)`
- `string("100int")` ⇒ `+function (int $foo)` ⇒ `int(100)`
- `string("1.23")` ⇒ `function(float $foo)` ⇒ `float(1.23)`
- `string("1.23float")` ⇒ `function(float $foo)` ⇒ `float(1.23)`

Additionally, `bool` hints will follow the same rules as comparisons: `int(0)`, `float(0.0)`, `null`, `string(0)`, empty strings, empty arrays, and a declared variable with no assigned value will all coerce to `false`. Anything else will be coerced to `true`.

Strict Types

In addition to the default behavior, PHP 7.0 also supports *strict type hints*. This means that rather than coerce, *any* type mismatch will result in a `\TypeError` exception.

Strict type hint behavior can only be enabled at the *calling* site. This means that, for example, a library author cannot dictate that their library is strictly typed: this can only be done in the code that calls it.

This might seem odd, because if you can't enforce strict usage, how can you be sure that your library is used correctly? While this is a valid concern, you need not be worried that the resulting input will ever be the wrong type. Regardless of strict mode, no matter what a value is passed in as, it will always be the correct type *within* the function.

To enable strict types, we use the previously esoteric `declare` construct with the new `strict_types` directive: `declare(strict_types=1);`.

The `declare` construct *must be the first statement* in the file. This means that nothing can be placed between the open PHP tag and the `declare` *except* for comments. This includes the `namespace` declaration, which must be placed *immediately after* any `declare` constructs. Example 11-4 shows how to enable strict types.

Example 11-4. Enabling strict types

```
// Enable strict types
declare(strict_types=1);
namespace MyProject\Component;

hinted("foo", 123, 4.35, true); ❶
```

❶ This would result in a `\TypeError` exception for the first incorrect argument with the message `TypeError: Argument 1 passed to hinted() must be of the type boolean, string given`.

Return Type Hints

In addition to more types of hints being available for arguments, we can now also type hint return values, using all the same types we can use for arguments.

NOTE

It bears repeating that you may use nonscalar type hints for return hints also: classes/interfaces, `array`, and `callable`. This is often overlooked, as return types are seen as an extension of the scalar type hints RFC.

Return hints follow the same rules under coercive or strict type hint settings. With coercive type hints, the `return` will be coerced to the hinted type when possible, while strict type hints will result in a `\TypeError` exception on type mismatch.

To add a type hint, simple follow the argument list with a colon and the return type, as shown in Example 11-5.

Example 11-5. Return type hints

```
function divide(int $a, int $b): int
{
    return $a / $b;
}

divide(4, 2); ❶
divide(5, 2); ❶
```

❶ Both calls will result in `int(2)` in coercive mode, or the second will cause a `\TypeError` exception with the message `Return value of divide() must be of the type integer, float returned` in strict mode.

In the preceding example, we have added an `int` return type hint to the `divide()` function that accepts two `int` arguments.

Despite the fact that all arguments are integers, because division can result in a `float`, you again may see precision loss in coercive mode or `\TypeError` exceptions in strict mode.

Reserved Keywords for Future Types

One final change that was added for type hinting is that the following list of type names can no longer be used as class, interface, or trait names:

- `int`
- `float`
- `bool`
- `string`
- `true`
- `false`
- `null`

This was done to facilitate potential future changes that would otherwise need to wait until PHP 8 to preserve backward compatiblity in PHP 7.x.

Summary

Unlike other languages where it is all or nothing when it comes to strict type hinting, PHP not only has *opt-in* strict hints, but it is up to the *calling* code, rather than the defining code.

This behavior is probably the only way to do scalar type hints "the PHP way."

Regardless of this, you should still be able to reap the full benefits of type hinting: more reliable code, and the ability to do things like statically analyze it to ensure correctness.

However, there are still several features that some feel are missing from the current type hinting, namely nullable types (which would include `null` return types), compound types (e.g., `int|float` or predefined ones like `numeric`), a special `void` return type hint, and custom types. While none of these are yet slated for inclusion in future releases, there are many people working on most of them, so it *could* happen in PHP 7.1 and beyond.

APPENDIX A
Resources

The following is a list of resources. You can find the most up-to-date list at *http://daveyshafik.com/php7*.

Further Reading

- *PHP Manual: Migrating from PHP 5.6.x to PHP 7.0.x*
- Blog post: "What to Expect When You're Expecting: PHP7, Part 1"
- Blog post: "What to Expect When You're Expecting: PHP 7, Part 2"
- Blog post: "An Exceptional Change in PHP 7.0"

Other Resources

- Rasmus Lerdorf's PHP 7 Dev Vagrant Box
- Jan Burkl's PHP 7 Docker container (nightly builds)
- GoPHP7-ext, helping to migrate extensions to PHP 7

RFCs for PHP 7.0

More than 45 RFCs went into PHP 7.0, and you can find them all below ordered by category.

Deprecated Features

See: Chapter 2

- "Removal of dead or not yet PHP7 ported SAPIs and extensions"
- "Make defining multiple default cases in a switch a syntax error"
- "Remove alternative PHP tags"

Uniform Variable Syntax

See: Chapter 3

- "Uniform Variable Syntax"

Basic Language Changes

See: Chapter 4

- "Combined Comparison (Spaceship) Operator"
- "Null Coalesce Operator"
- "intdiv()"
- "Add preg_replace_callback_array function"
- "Easy User-land CSPRNG"
- "Filtered unserialize()"
- "Introduce session_start() options—read_only, unsafe_lock, lazy_write and lazy_destroy"

Other RFCs

The following RFCs were not covered:

- "Remove the date.timezone warning"
- "Fix 'foreach' behavior"
- "Replacing current json extension with jsond"
- "Preserve Fractional Part in JSON encode"
- "Integer Semantics"
- "Fix list() behavior inconsistency"
- "Remove hex support in numeric strings"
- "Fix handling of custom session handler return values"

Expectations and Assertions

See: Chapter 5

- "Expectations"

Error Handling

See: Chapter 6

- "Constructor behaviour of internal classes"
- "Throwable Interface"
- "Exceptions in the engine (for PHP 7)"
- "Catchable 'call to a member function of a non-object'" (superseded by Exceptions in the engine)

Other RFCs

The following RFCs were not covered:

- "Reclassify E_STRICT notices"
- "Continue output buffering despite aborted connection"

Unicode Enhancements

See: Chapter 7

- "Unicode Codepoint Escape Syntax"
- "ICU IntlChar class"

Closures

See: Chapter 8

- “Closure::call”

Generators

See: Chapter 9

- “Generator Return Expressions”
- “Generator Delegation”

Object-Oriented Programming

See: Chapter 10

- “Context Sensitive Lexer”
- “Remove PHP 4 Constructors”
- “Group Use Declarations”
- “Anonymous Classes”

Type Hints

See: Chapter 11

- “Scalar Type Declarations”
- “Return Type Declarations”
- “Reserve More Types in PHP 7”

Internals Changes

As these are mainly changes to the inner workings of PHP, the following were not covered:

- "Abstract syntax tree"
- "Turn gc_collect_cycles into function pointer"
- "Fast Parameter Parsing API"
- "Native TLS"
- "ZPP Failure on Overflow"
- "Move the phpng branch into master"
- "64 bit platform improvements for string length and integer in zval"

About the Author

Davey Shafik is a full-time developer with over 14 years of experience in PHP and related technologies. He is a Developer Evangelist at Akamai Technologies and has written three books, numerous articles, and spoken at conferences the world over. He is best known for his books—the *Zend PHP Certification Study Guide* and *PHP Master: Write Cutting Edge Code*—and as the creator of PHP Archive (PHAR) for PHP 5.3.

Davey is passionate about improving the tech community. He coorganizes the Prompt initiative, which is dedicated to lifting the stigma surrounding mental health discussions, and has worked with PHPWomen since its inception.