Chapter 1. Variables

The foundation of a flexible application is *variability*—the capability of the program to serve multiple purposes in different contexts. *Variables* are a common mechanism to build such flexibility in any programming language. These named placeholders reference a specific value that a program wants to use. This could be a number, a raw string, or even a more complex object with its own properties and methods. The point is that a variable is the way a program (and the developer) references that value and passes it along from one part of the program to another.

Variables do not need to be set by default—it is perfectly reasonable to define a placeholder variable without assigning any value to it. Think of this like having an empty box on the shelf, ready and waiting to receive a gift for Christmas. You can easily find the box—the variable—but because nothing is inside it, you can't do much with it.

For example, assume the variable is called `$giftbox`. If you were to try to check the value of this variable right now, it would be empty, as it has not yet been set. In fact, `empty($giftbox)` will return `true`, and `isset($giftbox)` will return `false`. The box is both empty and not yet set.

NOTE

It's important to remember that any variable that has not been explicitly defined (or set) will be treated as `empty()` by PHP. An actually defined (or set) variable can either be empty or non-empty depending on its value, as any real value that evaluates to `false` will be treated as empty.

Broadly speaking, programming languages can either be strongly or loosely typed. A strongly typed language requires explicit identification of all variable, parameter, and function return types and enforces that the type of each value absolutely matches expectations. With a loosely typed language—like PHP—values are typed *dynamically* when they're used. For example, developers can store an integer (like `42`) in a variable and then use that variable as a string elsewhere (i.e., `"42"`), and PHP will transparently cast that variable from an integer to a string at runtime.

The advantage of loose typing is that developers don't need to identify how they will use a variable when it's defined, as the interpreter can do that identification well enough at runtime. A key disadvantage is that it's not always clear how certain values will be treated when the interpreter coerces them from one type to another.

PHP is well known as a loosely typed language. This sets the language apart as developers are not required to identify the type of a specific variable when it's created or even when it's called. The interpreter behind PHP will identify the right type when the variable is used and, in many cases, transparently cast the variable as a different type at runtime. Table 1-1 illustrates various expressions that, as of PHP 8.0, are evaluated as "empty" regardless of their underlying type.

Table 1-1. PHP empty expressions

Expression	`empty($x)`
`$x = ""`	`true`
`$x = null`	`true`
`$x = []`	`true`
`$x = false`	`true`
`$x = 0`	`true`
`$x = "0"`	`true`

Note that some of these expressions are not truly empty but are treated as such by PHP. In common conversation, they're considered `falsey` because they are treated to be equivalent to `false` although they're not identical to `false`. It's therefore important to *explicitly* check for expected values like `null` or `false` or `0` in an application rather than relying on language constructs like `empty()` to do the check for you. In such cases, you might want to check for emptiness of a variable *and* make an explicit equality check against a known, fixed value.[1]

The recipes in this chapter handle the basics of variable definition, management, and utilization in PHP.

1.1 Defining Constants

Problem

You want to define a specific variable in your program to have a fixed value that cannot be mutated or changed by any other code.

Solution

The following block of code uses `define()` to explicitly define the value of a globally scoped constant that cannot be changed by other code:

```
if (!defined('MY_CONSTANT')) {
    define('MY_CONSTANT', 5);
}
```

As an alternative approach, the following block of code uses the `const` directive within a class to define a constant scoped to that class itself:[2]

```
class MyClass
{
    const MY_CONSTANT = 5;
}
```

Discussion

If a constant is defined in an application, the function `defined()` will return `true` and let you know that you can access that constant directly within your code. If the constant is not yet defined, PHP tries to guess at what you're doing and instead converts the reference to the constant into a string literal.

TIP

Writing constant names in all caps is not required. However, this convention, defined in the Basic Coding Standard (PHP Standard Recommendation 1, or PSR-1) as published by the PHP Framework Interoperability Group (PHP-FIG), is strongly encouraged by documented standards.

For example, the following block of code will assign the value of `MY_CONSTANT` to the variable `$x` only when the constant is defined. Prior to PHP 8.0, an undefined constant would lead `$x` to hold the literal string `"MY_CONSTANT"` instead:

```
$x = MY_CONSTANT;
```

If the expected value of `MY_CONSTANT` is anything other than a string, the fallback behavior of PHP to provide a string literal could introduce unexpected side effects into your application. The interpreter won't necessarily crash, but having `"MY_CONSTANT"` floating around where an integer is expected will cause problems. As of PHP 8.0, referencing an as-yet-undefined constant results in a fatal error.

The Solution example demonstrates the two patterns used to define constants: `define()` or `const`. Using `define()` will create a global constant that is available anywhere in your application by using just the name of the constant itself. Defining a constant by way of `const` within a class definition will scope the constant to that class. Instead of referencing `MY_CONSTANT` as in the first solution, the class-scoped constant is referenced as `MyClass::MY_CONSTANT`.

WARNING

PHP defines several default constants that cannot be overwritten by user code. Constants in general are fixed and cannot be modified or replaced, so *always* check that a constant is not defined before attempting to define it. Attempts to redefine a constant will result in a notice. See Chapter 12 for more information on handling errors and notices.

Class constants are publicly visible by default, meaning any code in the application that can reference `MyClass` can reference its public constants as well. However, it is possible as of PHP 7.1.0 to apply a visibility modifier to a class constant and make it private to instances of the class.

See Also

Documentation on constants in PHP, `defined()`, `define()`, and class constants.

1.2 Creating Variable Variables

Problem

You want to reference a specific variable dynamically without knowing ahead of time which of several related variables the program will need.

Solution

PHP's variable syntax starts with a `$` followed by the name of the variable you want to reference. You can make the name of a variable *itself* a variable. The following program will print `#f00` by using a variable variable:

```
$red = '#f00';
$color = 'red';

echo $$color;
```

Discussion

When PHP is interpreting your code, it sees a leading `$` character as identifying a variable, and the immediate next section of text to represent that variable's name. In the Solution example, that text is itself, a variable. PHP will evaluate variable variables from right to left, passing the result of one evaluation as the name used for the left evaluation before printing any data to the screen.

Said another way, Example 1-1 shows two lines of code that are functionally equivalent, except the second uses curly braces to explicitly identify the code evaluated first.

Example 1-1. Evaluating variable variables

```
$$color;
${$color};
```

The rightmost `$color` is first evaluated to a literal `"red"`, which in turn means `$$color` and `$red` ultimately reference the same value. The introduction of curly braces as explicit evaluation delimiters suggests even more complicated applications.

Example 1-2 assumes an application wants to A/B test a headline for search engine optimization (SEO) purposes. Two options are provided by the marketing team, and the developers want to return different headlines for different visitors—but return the *same* headline when a visitor returns to the site. You can do so by leveraging the visitor's IP address and creating a variable variable that chooses a headline based on the visitor IP address.

Example 1-2. A/B testing headlines

```
$headline0 = 'Ten Tips for Writing Great Headlines';
$headline1 = 'The Step-by-Step to Writing Powerful Head

echo ${'headline' . (crc32($_SERVER['REMOTE_ADDR']) % 2
```

The `crc32()` function in the preceding example is a handy utility that calculates a 32-bit checksum of a given string—it turns a string into an integer in a deterministic fashion. The `%` operator performs a modulo operation on the resulting integer, returning `0` if the checksum is even and `1` if it is odd. The result is then concatenated to the string `headline` within your dynamic variable to allow the function to choose one or the other headline.

NOTE

The `$_SERVER` array is a system-defined superglobal variable that contains useful information about the server running your code and the incoming request that triggered PHP to run in the first place. The exact contents of this particular array will differ from server to server, particularly based on whether you used NGINX or Apache HTTP Server (or another web server) in front of PHP, but it usually contains helpful information like request headers, request paths, and the filename of the currently executing script.

Example 1-3 presents a `crc32()` utilization line by line to further illustrate how a user-associated value like an IP address can be leveraged to deterministically identify a headline used for SEO purposes.

Example 1-3. Walk-through of checksums against visitor IP addresses

```
$_SERVER['REMOTE_ADDR'] = '127.0.0.1';
                                                ❶
crc32('127.0.0.1') = 3619153832;
                                                ❷
3619153832 % 2 = 0;
                                                ❸
'headline' . 0 = 'headline0'
                                                ❹
${'headline0'} = 'Ten Tips for Writing Great Headlines'
                                                ❺
```

❶ The IP address is extracted from the `$_SERVER` superglobal variable. Note also that the `REMOTE_ADDR` key will only be present when using PHP from a web server and not through the CLI.

❷ `crc32()` converts the string IP address to an integer checksum.

❸ The modulo operator (`%`) determines whether the checksum is even or odd.

❹ The result of this modulo operation is appended to `headline`.

❺ The final string `headline0` is used as a variable variable to identify the correct SEO headline value.

It's even possible to nest variable variables more than two layers deep. Using three `$` characters—as with `$$$name`—is just as valid, as would be `$$${some_function()}`. It's a good idea, both for the simplicity of code review and for general maintenance, to limit the levels of variability within your variable names. The use cases for variable variables are rare enough to begin with, but multiple levels of indirection will render your code difficult to follow, understand, test, or maintain if something ever breaks.

See Also

Documentation on variable variables.

1.3 Swapping Variables in Place

Problem

You want to exchange the values stored in two variables without defining any additional variables.

Solution

The following block of code uses the `list()` language construct to reassign the values of the variables in place:

```
list($blue, $green) = array($green, $blue);
```

An even more concise version of the preceding solution is to use both the short list and short array syntaxes available since PHP 7.1 as follows:

```
[$blue, $green] = [$green, $blue];
```

Discussion

The `list` keyword in PHP doesn't refer to a function, although it looks like one. It's a *language construct* used to assign values to a list of variables rather than to one variable at a time. This enables developers to set multiple variables all at once from another list-like collection of values (like an array). It also permits the destructuring of arrays into individual variables.

Modern PHP leverages square brackets (`[` and `]`) for a short array syntax, allowing for more concise array literals. Writing `[1, 4, 5]` is functionally equivalent to `array(1, 4, 5)` , but is sometimes clearer depending on the context in which it is used.

NOTE

Like `list` , the `array` keyword refers to a language construct within PHP. Language constructs are hardcoded into the language and are the keywords that make the system work. Keywords like `if` and `else` or `echo` are easy to distinguish from userland code. Language constructs like `list` and `array` and `exit` look like functions, but like keyword-style constructs, they are built into the language and behave slightly differently than typical functions do. The PHP Manual's list of reserved keywords better illustrates existing constructs and cross-references with how each is used in practice.

As of PHP 7.1, developers can use the same short square bracket syntax to replace the usage of `list()` , creating more concise and readable code. Given that a solution to this problem is to assign values from an array to an array of variables, using similar syntax on both sides of the assignment operator (`=`) both makes sense and clarifies your intent.

The Solution example explicitly swaps the values stored in the variables `$green` and `$blue` . This is something that an engineer might do while deploying an application to switch from one version of an API to another. Rolling deployments often refer to the current live environment as the *green* deployment and a new, potential replacement as *blue*, instructing load balancers and other reliant applications to swap from green/blue and verify connectivity and functionality before confirming that the deployment is healthy.

In a more verbose example (Example 1-4), assume that the application consumes an API prefixed by the date of deployment. The application keeps track of which version of the API it is using (`$green`) and attempts to swap to a new environment to verify connectivity. If the connectivity check fails, the application will automatically switch back to the old environment.

Example 1-4. Blue/green environment cutover

```
$green = 'https://2021_11.api.application.example/v1';
$blue = 'https://2021_12.api.application.example/v1';

[$green, $blue] = [$blue, $green];

if (connection_fails(check_api($green))) {
    [$green, $blue] = [$blue, $green];
}
```

The `list()` construct can also be used to extract certain values from an arbitrary group of elements. Example 1-5 illustrates how an address, stored as an array, can be used in different contexts to extract just specific values as needed.

Example 1-5. Using `list()` to extract elements of an array

```
$address = ['123 S Main St.', 'Anywhere', 'NY', '10001'

// Extracting each element as named variables
[$street, $city, $state, $zip, $country] = $address;

// Extracting and naming only the city
[,,$state,,] = $address;

// Extracting only the country
[,,,,$country] = $address;
```

Each extraction in the preceding example is independent and sets only the variables that are necessary.[3] For a trivial illustration such as this, there is no need to worry about extracting each element and setting a variable, but for more complex applications manipulating data that is significantly larger, setting unnecessary variables can lead to performance issues. While `list()` is a powerful tool for destructuring array-like collections, it is only appropriate for simple cases like those discussed in the preceding examples.

See Also

Documentation on `list()`, `array()`, and the PHP RFC on short `list()` syntax.

[1] Equality operators are covered in Recipe 2.3, which provides both an example and a thorough discussion of equality checks.

[2] Read more about classes and objects in Chapter 8.

[3] Recall in this chapter's introduction the explanation that variable references not explicitly being set will evaluate as "empty." This means you can set only the values and variables you need to use.