Chapter 2. Operators

While Chapter 1 introduced the foundational building blocks of PHP—variables to store arbitrary values—these building blocks are useless without some kind of glue to hold them together. This glue is the set of *operators* established by PHP. Operators are the way you tell PHP what to do with certain values—specifically how to change one or more values into a new, discrete value.

In almost every case, an operator in PHP is represented by a single character or by repeated uses of that same character. In a handful of cases, operators can also be represented by literal English words, which helps disambiguate what the operator is trying to accomplish.

This book does not attempt to cover every operator leveraged by PHP; for exhaustive explanations of each, refer to the PHP Manual itself. Instead, the following few sections cover some of the most important logical, bitwise, and comparison operators before diving into more concrete problems, solutions, and examples.

Logical Operators

Logical operations are the components of PHP that create truth tables and define basic and/or/not grouping criteria. Table 2-1 enumerates all of the character-based logical operators supported by PHP.

Table 2-1. Logical operators

Expression	Operator name	Result	Example
$x && $y	and	`true` if both `$x` and `$y` are `true`	`true && true == true`

Expression	Operator name	Result	Example
$x || $y	or	`true` if either `$x` or `$y` is `true`	`true || false == true`
!$x	not	`true` if `$x` is `false` (and vice versa)	`!true == false`

The logical operators `&&` and `||` have English word counterparts: `and` and `or`, respectively. The statement `($x and $y)` is functionally equivalent to `($x && $y)`. The word `or` can likewise be used in place of the `||` operator without changing the functionality of the expression.

The word `xor` can also be used to represent a special *exclusive or* operator in PHP that evaluates to `true` if one of the two values in the expression is `true`, but not when both are `true`. Unfortunately, the logical XOR operation has no character equivalent in PHP.

Bitwise Operators

PHP supports operations against specific bits in an integer, a feature that makes the language quite versatile. Supporting bitwise operations means PHP is not limited to web applications but can operate on binary files and data structures with ease! It's worth mentioning these operators in the same section as the preceding logical operators as they appear somewhat similar in terms of terminology with `and`, `or`, and `xor`.

Whereas logical operators return `true` or `false` based on the comparison between two whole values, bitwise operators actually perform bitwise arithmetic on integers and return the result of that full calculation over the integer or integers provided. For a specific example of how this can be useful, skip ahead to Recipe 2.6.

Table 2-2 illustrates the various bitwise operators in PHP, what they do, and a quick example of how they work on simple integers.

Table 2-2. Bitwise operators

Expression	Operator name	Result	Example
`$x & $y`	and	Returns bits set in both `$x` and `$y`	`5 & 1 == 1`
`$x | $y`	or	Returns bits set in either `$x` or `$y`	`4 | 1 == 5`
`$x ^ $y`	xor	Returns bits set in only `$x` or `$y`	`5 ^ 3 == 6`
`~ $x`	not	Inverts bits that are set in `$x`	`~ 4 == -5`
`$x << $y`	shift left	Shift the bits of `$x` to the left by `$y` steps	`4 << 2 == 16`
`$x >> $y`	shift right	Shift the bits of `$x` to the right by `$y` steps	`4 >> 2 == 1`

In PHP, the largest integer you can have depends on the size of the processor running the application. In any case, the constant `PHP_INT_MAX` will tell you how large integers can be—2147483647 on 32-bit machines and 9223372036854775807 on 64-bit machines. In both cases, this number is represented, in binary, as a long string of 1s equal in length to one less than the bit size. On a 32-bit machine, 2147483647 is represented by 31 1s. The leading bit (a `0` by default) is used to identify the *sign* of the integer. If the bit is `0`, the integer is positive; if the bit is `1`, the integer is negative.

On any machine, the number 4 is represented in binary as `100`, with enough 0s to the left of the most significant digit to fill the bit size of the processor. On a 32-bit system, this would be 29 0s. To make the integer *negative*, you would represent it instead as a 1 followed by 28 0s followed by `100`.

For simplicity, consider a 16-bit system. The integer 4 would be represented as `0000000000000100`. Likewise, a negative 4 would be represented as `1000000000000100`. If you were to apply the bitwise *not* operator (`~`) on

a positive 4 in a 16-bit system, all of the 0s would become 1s and vice versa. This would turn your number into `1111111111111011`, which on a 16-bit system is −5.

Comparison Operators

The core of any programming language is the level of control that language has to branch based on specific conditions. In PHP, much of this branching logic is controlled by comparing two or more values with one another. It is the set of comparison operators provided by PHP, provide most of the advanced branching functionality used to build complex applications.

Table 2-3 lists the scalar comparison operators considered to be the most vital to understand PHP. The other operators (greater than, less than, and variants) are somewhat standard among programming languages and are not necessary to any of the recipes in this chapter.

Table 2-3. Comparison operators

Expression	Operation	Result
`$x == $y`	Equal	Returns `true` if both values are the same after coercing into the same type
`$x === $y`	Identical	Returns `true` if both values are the same *and* are of the same type
`$x <=> $y`	Spaceship	Returns `0` is both values are equal, `1` if `$x` is greater, or `-1` if `$y` is greater

When dealing with objects, the equality and identity operators work somewhat differently. Two objects are considered equal (`==`) if they have the same internal structure (same attributes and values) and are of the same type (class). Objects are considered identical (`===`) if and only if they are references to the same instance of a class. These are stricter requirements than those for comparing scalar values.

Type Casting

While the name of a type is not formally an operator, you can use it to explicitly cast a value as that type. Simply write the name of the type within parentheses before the value to force a conversion. Example 2-1 converts a simple integer value to various other types prior to using the value.

Example 2-1. Casting values as other types

```
$value = 1;

$bool = (bool) $value;
$float = (float) $value;
$string = (string) $value;

var_dump([$bool, $float, $string]);

// array(3) {
//   [0]=>
//   bool(true)
//   [1]=>
//   float(1)
//   [2]=>
//   string(1) "1"
// }
```

PHP supports the following type casts:

`(int)`
: Cast to `int`

`(bool)`
: Cast to `bool`

`(float)`
: Cast to `float`

`(string)`
: Cast to `string`

`(array)`
: Cast to `array`

`(object)`
: Cast to `object`

It's also possible to use `(integer)` as an alias of `(int)`, `(boolean)` as an alias of `(bool)`, `(real)` or `(double)` as aliases of `(float)`, and `(binary)` as an alias of `(string)`. These aliases will make the same type casts as in the preceding list, but given that they don't use the name of the type to which you're casting, this approach is not recommended.

The recipes in this chapter introduce ways to leverage PHP's most important comparison and logical operators.

2.1 Using a Ternary Operator Instead of an If-Else Block

Problem

You want to provide an either-or branching condition to assign a specific value to a variable in a single line of code.

Solution

Using a *ternary operator* (`a ? b : c`) allows nesting an either-or condition and both possible branched values in a single statement. The following example shows how to define a variable with a value from the `$_GET` superglobal and fall back on a default if it is empty:

```
$username = isset($_GET['username']) ? $_GET['username'
```

Discussion

A ternary expression has three arguments and is evaluated from left to right, checking the *truthiness* of the leftmost statement (whether it evaluates to `true` regardless of the types involved in the expression) and returning the next value if `true` or the final value if `false`. You can visualize this logical flow with the following illustration:

```
$_value_ = (_expression to evaluate_) ? (if true) : (if
```

The ternary pattern is a simple way to return a default value when checking either system values or even parameters from a web request (those stored in the `$_GET` or `$_POST` superglobals). It is also a powerful way to switch logic in page templates based on the return of a particular function call.

The following example assumes a web application that welcomes logged-in users by name (checking their authentication state with a call to `is_logged_in()`) or welcomes a guest if the user has yet to authenticate. As this example is coded directly into the HTML markup of a web page, using a longer `if` / `else` statement would be inappropriate:

```
<h1>Welcome, <?php echo is_logged_in() ? $_SESSION['use
```

Ternary operations can also be simplified if the value being checked is both *truthy* (evaluates to `true` when coerced into a Boolean value) and is the value you want by default. The Solution example checks that a username is set *and* assigns that value to a given variable if so. Since non-empty strings evaluate to `true` , you can shorten the solution to the following:

```
$username = $_GET['username'] ?: 'default';
```

When a ternary is shortened from its *a* `?` *b* `:` *c* format to a simple *a* `?:` *c* , PHP will evaluate the expression to check *a* as if it were a Boolean value. If it's true, PHP merely returns the expression itself. If it's false, PHP returns the fallback value *c* instead.

NOTE

PHP compares truthiness similarly to the way it compares emptiness, as discussed in Chapter 1. Strings that are set (not empty or `null`), integers that are nonzero, and arrays that are non-empty are all generally considered truthy, which is to say they evaluate to `true` when cast as a Boolean. You can read more about the ways types are intermixed and considered equivalent in the PHP Manual section on type comparisons.

The ternary operator is an advanced form of comparison operator that, while it provides for concise code, can sometimes be overused to create logic that is too difficult to follow. Consider Example 2-2, which nests one ternary operation within another.

Example 2-2. Nested ternary expression

```
$val = isset($_GET['username']) ? $_GET['username'] : (
       ? $_GET['user_id'] : null);
```

This example should be rewritten as a simple `if` / `else` statement instead to provide more clarity as to how the code branches. Nothing is *functionally* wrong with the code, but nested ternaries can be difficult to read or reason about and often lead to logic errors down the road. The preceding ternary could be rewritten as shown in Example 2-3:

Example 2-3. Multiple `if/else` statements

```
if (isset($_GET['username'])) {
    $val = $_GET['username'];
} elseif (isset($_GET['userid'])) {
    $val = $_GET['userid'];
} else {
    $val = null;
}
```

While Example 2-3 is more verbose than Example 2-2, you can more easily track where the logic needs to branch. The code is also more maintainable, as new branching logic can be added where necessary. Adding another logical branch to Example 2-2 would further complicate the already complex ternary and make the program even harder to maintain in the long run.

See Also

Documentation on the ternary operator and its variations.

2.2 Coalescing Potentially Null Values

Problem

You want to assign a specific value to a variable only if it's set and not `null` and otherwise use a static default value.

Solution

Using a null-coalescing operator (`??`) as follows will use the first value only if it is set and not `null` :

```
$username = $_GET['username'] ?? 'not logged in';
```

Discussion

PHP's null-coalescing operator is a newer feature introduced in PHP 7.0. It's been referred to as *syntactic sugar* to replace the shorthand version of PHP's ternary operator, `?:` , discussed in Recipe 2.1.

NOTE

Syntactic sugar is shorthand for performing a common yet verbose operation in code. The developers of languages introduce such features to save keystrokes and render routine, oft-repeated blocks of code via simpler and more concise syntax.

Both of the following lines of code are functionally equivalent, but the ternary form will trigger a notice if the expression being evaluated is undefined:

```
$a = $b ?: $c;
$a = $b ?? $c;
```

While these preceding two examples are *functionally* identical, a notable difference in their behavior occurs if the value being evaluated (`$b`) is not defined. With the null-coalescing operator, everything is golden. With the ternary shorthand, PHP will trigger a notice during execution that the value is undefined before returning the fallback value.

With discrete variables, the differing functionality of these operators isn't entirely obvious, but when the evaluated component is, perhaps, an indexed array, the potential impact becomes more apparent. Assume that, instead of a discrete variable, you are trying to extract an element from the superglobal `$_GET` variable that holds request parameters. In the following example, both the ternary and the null-coalescing operators will return the fallback value, but the ternary version will complain about an undefined index:

```
$username = $_GET['username'] ?? 'anonymous';
$username = $_GET['username'] ?: 'anonymous'; // Notice
```

If errors and notices are suppressed during execution,[1] there is no functional difference between either operator option. It is, however, best practice to avoid writing code that triggers errors or notices, as these can accidentally raise alerts in production or potentially fill system logs and make it more difficult to find legitimate issues with your code. While the shorthand ternary operator is remarkably useful, the null-coalescing operator is purpose-built for this kind of operation and should nearly always be used instead.

See Also

The announcement of the new operator when it was first added to PHP 7.0.

2.3 Comparing Identical Values

Problem

You want to compare two values of the same type to ensure that they're identical.

Solution

Use three equals signs to compare values without dynamically casting their types:

```
if ($a === $b) {
    // ...
```

```
}
```

Discussion

In PHP, the equals sign has three functions. A single equals sign (`=`) is used for *assignment*, which is setting the value of a variable. Two equals signs (`==`) are used in an expression to determine whether the values on either side are equal. Table 2-4 shows how certain values are considered equal because PHP coerces one type into another while evaluating the statement. Finally, three equals signs (`===`) are used in an expression to determine whether the values on either side are *identical*.

Table 2-4. Value equality in PHP

Expression	Result	Explanation
`0 == "a"`	`false`	(Only for PHP 8.0 and above) The string `"a"` is cast as an integer, which means it's cast to `0`.
`"1" == "01"`	`true`	Both sides of the expression are cast to integers, and `1 == 1`.
`100 == "1e2"`	`true`	The right side of the expression is evaluated as an exponential representation of `100` and cast as an integer.

NOTE

The first example in Table 2-4 evaluates as `true` in PHP versions below 8.0. In those earlier versions, comparing the equality of a string (or numeric string) to a number would convert the string first to a number (in this case, converting `"a"` to `0`). This behavior changed in PHP 8.0 such that only numeric strings are cast to numbers, so the result of that first expression is now `false`.

PHP's ability to dynamically convert between types at runtime can be useful, but in some cases it is not what you want to have happen at all. The Boolean literal `false` is returned by some methods to represent an error or failure, while an integer `0` might be a valid return of a function. Consider the

function in Example 2-4 that returns a count of books of a specific category, or `false` if a connection to the database holding that data fails.

Example 2-4. Count items in a database or return `false`

```
function count_books_of_type($category)
{
    $sql = "SELECT COUNT(*) FROM books WHERE category =

    try {
        $dbh = new PDO(DB_CONNECTION_STRING, DB_LOGIN,
        $statement = $dbh->prepare($sql);

        $statement->execute(array(':category' => $categ
        return $statement->fetchColumn();
    } catch (PDOException $e) {
        return false;
    }
}
```

If everything in Example 2-4 runs as expected, the code will then return an integer count of the number of books in a particular category. Example 2-5 might leverage this function to print a headline on a web page.

Example 2-5. Using the results of a database-bound function

```
$books_found = count_books_of_type('fiction');

switch ($books_found) {
    case 0:
        echo 'No fiction books found';
        break;
    case 1:
        echo 'Found one fiction book';
        break;
    default:
        echo 'Found ' . $books_found . ' fiction books'
}
```

Internally, PHP's `switch` statement is using a loose type comparison (our `==` operator). If `count_books_of_type()` returns `false` instead of an

actual result, this `switch` statement will print out that no fiction books were found rather than reporting an error. In this particular use case, that might be acceptable behavior—but when your application needs to reflect a material difference between `false` and `0`, loose equality comparisons are inadequate.

Instead, PHP permits the use of *three* equals signs (`===`) to check whether both values under evaluation are identical—that is, they are both the same value and the same type. Even though the integer `5` and the string `"5"` have the same value, evaluating `5 === "5"` will result in `false` because the two values are not the same type. Thus, while `0 == false` evaluates to `true`, `0 === false` will always evaluate to `false`.

WARNING

Determining whether two values are identical becomes more complicated when dealing with objects, either defined with custom classes or PHP-provided ones. In the case of two objects, `$obj1` and `$obj2`, they will only evaluate as identical if they are actually the same *instance* of a class. For more on object instantiation and classes, see Chapter 8.

See Also

PHP documentation on comparison operators.

2.4 Using the Spaceship Operator to Sort Values

Problem

You want to provide a custom ordering function to sort an arbitrary list of objects by using PHP's native `usort()`.

Solution

Assuming you want to sort by multiple properties of the list of objects, use PHP's spaceship operator (`<=>`) to define a custom sorting function and supply that as the callback to `usort()`.

Consider the following class definition for a person in your application that allows creating records with just first and last names:

```
class Person {
    public $firstName;
    public $lastName;

    public function __construct($first, $last)
    {
        $this->firstName = $first;
        $this->lastName = $last;
    }
};
```

You can then create a list of people, perhaps US presidents, using this class and adding each person to your list in turn, as in Example 2-6.

Example 2-6. Adding multiple object instances to a list

```
$presidents = [];

$presidents[] = new Person('George', 'Washington');
$presidents[] = new Person('John', 'Adams');
$presidents[] = new Person('Thomas', 'Jefferson');
// ...
$presidents[] = new Person('Barack', 'Obama');
$presidents[] = new Person('Donald', 'Trump');
$presidents[] = new Person('Joseph', 'Biden');
```

The spaceship operator can then be leveraged to identify how to sort this data, assuming you want to order by last name first, then by first name, as shown in Example 2-7.

Example 2-7. Sorting presidents with the spaceship operator

```
function presidential_sorter($left, $right)
{
    return [$left->lastName, $left->firstName]
        <=>
        [$right->lastName, $right->firstName];
}
```

```
usort($presidents, 'presidential_sorter');
```

The result of the preceding call to `usort()` is that the `$presidents` array will be properly sorted in place and ready for use.

Discussion

The spaceship operator is a special addition as of PHP 7.0 that helps identify the relationship between the values on either side of it:

- If the first value is less than the second, the expression evaluates to `-1`.
- If the first value is greater than the second, the expression evaluates to `+1`.
- If both values are the same, the expression evaluates to `0`.

NOTE

Like PHP's equality operator, the spaceship operator will attempt to coerce the types of each value in the comparison to be the same. It is possible to support a number for one value and a string for the other and get a valid result. Use type coercion with special operators like this at your own risk.

The simplest use of the spaceship operator compares simple types with one another, making it easy to order a simple array or list of primitive values (like characters, integers, floating-point numbers, or dates). This simple case, if using `usort()`, would require a sorting function like the following:

```
function sorter($a, $b) {
    return ($a < $b) ? -1 : (($a > $b) ? 1 : 0);
}
```

The spaceship operator simplifies the nested ternary in the preceding code by replacing the `return` statement entirely with `return $a <=> $b`, but without modifying the functionality of the sorting function at all.

More complex examples, like that used in the Solution to sort based on multiple properties of a custom object definition, would necessitate rather

verbose sorting function definitions. The spaceship operator simplifies comparison logic, empowering developers to specify otherwise complex logic in a single, easy-to-read line.

See Also

The original RFC for PHP's spaceship operator.

2.5 Suppressing Diagnostic Errors with an Operator

Problem

You want to explicitly ignore or suppress errors triggered by a specific expression in your application.

Solution

Prefix the expression with the `@` operator to temporarily set the error reporting level to 0 for that line of code. This might help suppress errors related to missing files when attempting to open them directly, as in the following example:

```
$fp = @fopen('file_that_does_not_exist.txt', 'r');
```

Discussion

The Solution example attempts to open the file *file_that_does_not_exist.txt* for reading. In normal operations, a call to `fopen()` would return `false` because the file does not exist *and* emit a PHP warning for the purposes of diagnosing the issue. Prefixing the expression with the `@` operator doesn't change the return value at all, but it suppresses the emitted warning entirely.

WARNING

The `@` operator suppresses error reporting for the line to which it is applied. If a developer attempts to suppress errors on an `include` statement, they will very easily hide any warnings, notices, or errors caused by the included file not existing (or having improper access controls). The suppression will *also* apply to all lines of code within the included file, meaning any errors (syntax-related or otherwise) in the included code will be ignored. Thus, while `@include('some-file.php')` is perfectly valid code, suppressing errors on `include` statements should be avoided!

This particular operator is useful when suppressing errors or warnings on file access operations (as in the Solution example). It's also useful in suppressing notices in array-access operations, as in the following, where a specific `GET` parameter might not be set in a request:

```
$filename = @$_GET['filename'];
```

The `$filename` variable will be set to the value of the request's `filename` query parameter if it's set. Otherwise, it will be a literal `null` . If a developer were to omit the `@` operator, the value of `$filename` would still be `null` , but PHP would emit a notice that the index of `filename` does not exist in the array.

As of PHP 8.0, this operator will no longer suppress *fatal* errors in PHP that otherwise halt script execution.

See Also

Official PHP documentation on error control operators.

2.6 Comparing Bits Within Integers

Problem

You want to use simple flags to identify state and behavior in your application, where one member might have multiple flags applied.

Solution

Use a bitmask to specify which flags are available and bitwise operators on the subsequent flags to identify which are set. The following example defines four discrete flags by using a binary notation of the integer each represents and combines them to indicate *multiple* flags being set at once. PHP's bitwise operators are then used to identify which flag is set and which branch of conditional logic should be executed:

```
const FLAG_A = 0b0001; // 1
const FLAG_B = 0b0010; // 2
const FLAG_C = 0b0100; // 4
const FLAG_D = 0b1000; // 8

// Set a composite flag for an application
$application = FLAG_A | FLAG_B; // 0b0011 or 3

// Set a composite flag for a user
$user = FLAG_B | FLAG_C | FLAG_D; // 0b1110 or 14

// Switch based on the user's applied flags
if ($user & FLAG_B) {
    // ...
} else {
    // ...
}
```

Discussion

A bitmask is structured by configuring each flag to be a constant integer power of 2. This has the benefit of only setting a single bit in the binary representation of the number such that composite flags are then identified by which bits are set. In the Solution example, each flag is written explicitly as a binary number to illustrate which bits are set (`1`) versus unset (`0`), with the integer representation of the same number in a comment at the end of the line.

Our example's `FLAG_B` is the integer 2, which is represented in binary as `0010` (the third bit is set). Likewise, `FLAG_C` is the integer 4 with a binary representation of `0100` (the second bit is set). To specify that *both* flags are set, you add the two together to set both the second and third bits: `0110` or the integer 6.

For this specific example, addition is an easy model to keep in mind, but it's not exactly what is going on. To combine flags, you merely want to combine the bits that are set, not necessarily add them together. Combining `FLAG_A` with itself should result in *only* `FLAG_A`; adding the integer representation (1) to itself would change the meaning of the flag entirely.

Rather than addition, use the bitwise operations *or* (`|`) and *and* (`&`) to both combine bits and filter on assigned flags. Combining two flags together requires using the `|` operator to create a new integer with bits that are set in *either* of the flags being used. Consider Table 2-5 to create a composite of `FLAG_A | FLAG_C`.

Table 2-5. Composite binary flags with bitwise *or*

Flag	Binary representation	Integer representation
`FLAG_A`	`0001`	`1`
`FLAG_C`	`0100`	`4`
`FLAG_A | FLAG_C`	`0101`	`5`

Comparing composite flags against your definitions then requires the `&` operator, which returns a new number that has bits set on *both* sides of the operation. Comparing a flag to itself will always return 1, which is type cast to `true` in conditional checks. Comparing two values that have any of the same bits set will return a value *greater* than 0, which is type cast to `true`. Consider the simple case of evaluating where `FLAG_A & FLAG_C` in Table 2-6.

Table 2-6. Composite binary flags with bitwise *and*

Flag	Binary representation	Integer representation
`FLAG_A`	`0001`	`1`
`FLAG_C`	`0100`	`4`

Flag	Binary representation	Integer representation
`FLAG_A & FLAG_C`	`0000`	`0`

Instead of comparing primitive flags against one another, you can and should build composite values and then compare them to your sets of flags. The following example visualizes the role-based access controls of a content management system for publishing news articles. Users can view articles, create articles, edit articles, or delete articles; their level of access is determined by the program itself and the permissions granted to their user account:

```
const VIEW_ARTICLES   = 0b0001;
const CREATE_ARTICLES = 0b0010;
const EDIT_ARTICLES   = 0b0100;
const DELETE_ARTICLES = 0b1000;
```

A typical, anonymous visitor will never be logged in and will then be granted a default permission of being able to view content. Logged-in users might be able to create articles but not edit them without an editor's permission. Likewise, editors can review and modify content (or delete it) but cannot independently create articles. Finally, administrators might be allowed to do everything. Each of the roles is composited from the preceding permission primitives as follows:

```
const ROLE_ANONYMOUS = VIEW_ARTICLES;
const ROLE_AUTHOR    = VIEW_ARTICLES | CREATE_ARTICLES;
const ROLE_EDITOR    = VIEW_ARTICLES | EDIT_ARTICLES |
const ROLE_ADMIN     = VIEW_ARTICLES | CREATE_ARTICLES
                       | DELETE_ARTICLES;
```

Once composite roles are defined from primitive permissions, the application can structure logic around checking the user's active role. While permissions were composited together with the `|` operator, the `&` operator will allow you to switch based on these flags, as demonstrated by the functions defined in Example 2-8.

Example 2-8. Leveraging bitmask flags for access control

```
function get_article($article_id)
{
    $role = get_user_role();

    if ($role & VIEW_ARTICLES) {
        // ...
    } else {
        throw new UnauthorizedException();
    }
}

function create_article($content)
{
    $role = get_user_role();

    if ($role & CREATE_ARTICLES) {
        // ...
    } else {
        throw new UnauthorizedException();
    }
}

function edit_article($article_id, $content)
{
    $role = get_user_role();

    if ($role & EDIT_ARTICLES) {
        // ...
    } else {
        throw new UnauthorizedException();
    }
}

function delete_article($article_id)
{
    $role = get_user_role();

    if ($role & DELETE_ARTICLES) {
        // ...
    } else {
        throw new UnauthorizedException();
    }
}
```

Bitmasks are a powerful way to implement simple flags in any language. Take caution, though, if the number of flags needed is ever planned to increase, because each new flag represents an additional power of 2, meaning the value of all flags grows rapidly in size. However, bitmasks are commonly used in both PHP applications and by the language itself. PHP's own error reporting setting, discussed further in Chapter 12, leverages bitwise values to identify the level of error reporting used by the engine itself.

See Also

PHP documentation on bitwise operators.

[1] Error handling and suppressing errors, warnings, and notices are discussed at length in Chapter 12.