Chapter 8. Classes and Objects

The earliest versions of PHP didn't support class definition or object orientation. PHP 4 was the first real attempt at an object interface.[1] It wasn't until PHP 5, though, that developers had the complex object interfaces they know and use today.

Classes are defined using the `class` keyword followed by a full description of the constants, properties, and methods inherent to the class. Example 8-1 introduces a basic class construct in PHP, complete with a scoped constant value, a property, and callable methods.

Example 8-1. Basic PHP class with properties and methods

```
class Foo
{
    const SOME_CONSTANT = 42;

    public string $hello = 'hello';

    public function __construct(public string $world =

    public function greet(): void
    {
        echo sprintf('%s %s', $this->hello, $this->worl
    }
}
```

An object can be instantiated with the `new` keyword and the name of the class; instantiation looks somewhat like a function call. Any parameters passed into this instantiation are transparently passed into the class constructor (the `__construct()` method) to define the object's initial state. Example 8-2 illustrates how the class definition from Example 8-1 could be instantiated either with or without default property values.

Example 8-2. Instantiating a basic PHP class

```
$first = new Foo;                                    ❶

$second = new Foo('universe');                       ❷


$first->greet();                                     ❸

$second->greet();                                    ❹


echo Foo::SOME_CONSTANT;                             ❺
```

❶ Instantiating an object without passing a parameter will still invoke the constructor but will leverage its default parameters. If no defaults are provided in the function signature, this would result in an error.

❷ Passing a parameter during instantiation will provide this parameter to the constructor.

❸ This prints `hello world` by using the constructor's defaults.

❹ This prints `hello universe` to the console.

❺ Constants are referenced directly from the class name. This will print a literal `42` to the console.

Constructors and properties are covered in Recipes 8.1 and 8.2.

Procedural Programming

Most developers' first experience with PHP is through its more procedural interfaces. Example routines, simple scripts, tutorials—all of these typically leverage functions and variables defined within the global scope. This isn't a bad thing, but it does limit the flexibility of the programs you can produce.

Procedural programming often results in stateless applications. There is little to no ability to keep track of what's happened before between function calls, so you pass some reference to the application's state throughout your code. Again, this isn't necessarily a bad thing. The only drawback is that *complex* applications become difficult to analyze or understand.

Object-Oriented Programming

An alternative paradigm is to leverage objects as containers of state. A common practical example is to consider objects as ways to define *things*. A car is an object. So is a bus. And a bicycle. They are discrete *things* that have characteristics (such as color, number of wheels, and type of drive) and capabilities (such as go, stop, and turn).

In the programming world, this is among the easiest ways to describe objects. In PHP, you create objects by first defining a `class` to describe the type of object. A class describes the properties (characteristics) and methods (capabilities) that an object of that type will have.

Like things in the real world, objects in a programming space can inherit from more primitive type descriptions. A car, bus, and bicycle are all types of vehicles, so they can all descend from a specific type. Example 8-3 demonstrates how PHP might construct that kind of object inheritance.

Example 8-3. Class abstraction in PHP

```
abstract class Vehicle
{
    abstract public function go(): void;

    abstract public function stop(): void;

    abstract public function turn(Direction $direction)
}

class Car extends Vehicle
{
    public int $wheels = 4;
    public string $driveType = 'gas';

    public function __construct(public Color $color) {}

    public function go(): void
    {
        // ...
    }

    public function stop(): void
    {
```

```
        // ...
    }

    public function turn(Direction $direction): void
    {
        // ...
    }
}

class Bus extends Vehicle
{
    public int $wheels = 4;
    public string $driveType = 'diesel';

    public function __construct(public Color $color) {}

    public function go(): void
    {
        // ...
    }

    public function stop(): void
    {
        // ...
    }

    public function turn(Direction $direction): void
    {
        // ...
    }
}

class Bicycle extends Vehicle
{
    public int $wheels = 2;
    public string $driveType = 'direct';

    public function __construct(public Color $color) {}

    public function go(): void
    {
        // ...
    }

    public function stop(): void
    {
```

```
        // ...
    }

    public function turn(Direction $direction): void
    {
        // ...
    }
}
```

Instantiating an object creates a typed variable that represents both an initial state and methods for manipulating that state. Object *inheritance* presents the possibility to use one or more types as alternatives to one another in other code. Example 8-4 illustrates how the three types of vehicles introduced in Example 8-2 could be used interchangeably because of inheritance.

Example 8-4. Classes with similar inheritance can be used interchangeably

```
function commute(Vehicle $vehicle)
                          ❶

{
    // ...
}

function exercise(Bicycle $vehicle)
                          ❷

{
    // ...
}
```

❶ All three of the vehicle subtypes can be used as valid replacements for `Vehicle` in function calls. This means you could commute by bus, car, or bicycle, and any choice would be equally valid.

❷ At times you might need to be more precise and use a child type directly. Neither `Bus` nor `Car` nor any other subclass of the `Vehicle` class will be valid for exercise besides a `Bicycle`.

Class inheritance is covered at a deeper level by Recipes 8.6, 8.7, and 8.8.

Multiparadigm Languages

PHP is considered a *multiparadigm* language, as you can write code following either of the preceding paradigms. A valid PHP program can be purely procedural. Or it can be strictly focused on object definitions and custom classes. The program could ultimately use a mix of both paradigms.

The open source WordPress content management system (CMS) is one of the most popular PHP projects on the internet.[2] It's coded to heavily leverage objects for common abstractions like database objects or remote requests. However, WordPress also stems from a long history of procedural programming—much of the codebase is still heavily influenced by that style. WordPress is a key example not just of the success of PHP itself, but of the flexibility of the language's support for multiple paradigms.

There is no single right answer for how an application should be assembled. Most are hybrids of approaches that benefit from PHP's strong support of multiple paradigms. Even in a majority-procedural application, though, you will still likely see a handful of objects as that's how the language's standard library implements much of its functionality.

Chapter 6 illustrated the use of both the functional and object-oriented interfaces for PHP's date system. Error handling, which is covered more in Chapter 13, heavily leverages internal `Exception` and `Error` classes. The `yield` keyword in an otherwise procedural implementation automatically creates instances of the `Generator` class.

Even if you never define a class in your program directly, the chances are good that you will leverage one defined either by PHP itself or by a third-party library your program requires.[3]

Visibility

Classes also introduce the concept of *visibility* into PHP. Properties, methods, and even constants can be defined with an optional visibility modifier to change their level of access from other parts of the application. Anything declared `public` is accessible to any other class or function in your application. Both methods and properties can be declared `protected`, making them accessible only to an instance of either the class itself or classes

that descend from it. Finally, the `private` declaration means a member of the class can be accessed only by instances of that class itself.

NOTE

By default, anything not explicitly scoped to be private or protected is automatically public, so you might see some developers skip declaring member visibility entirely.

While member visibility can be directly overridden via reflection,[4] it's typically a sound way to clarify which parts of a class's interface are intended to be used by other code elements. Example 8-5 illustrates how each visibility modifier can be leveraged to build a complex application.

Example 8-5. Class member visibility overview

```
class A
{
    public    string $name  = 'Bob';
    public    string $city  = 'Portland';
    protected int    $year  = 2023;
    private   float  $value = 42.9;

    function hello(): string
    {
        return 'hello';
    }

    public function world(): string
    {
        return 'world';
    }

    protected function universe(): string
    {
        return 'universe';
    }

    private function abyss(): string
    {
        return 'the void';
    }
}
```

```php
class B extends A
{
    public function getName(): string
    {
        return $this->name;
    }

    public function getCity(): string
    {
        return $this->city;
    }

    public function getYear(): int
    {
        return $this->year;
    }

    public function getValue(): float
    {
        return $this->value;
    }
}

$first = new B;
echo $first->getName() . PHP_EOL;
                          ❶

echo $first->getCity() . PHP_EOL;
                          ❷

echo $first->getYear() . PHP_EOL;
                          ❸

echo $first->getValue() . PHP_EOL;
                          ❹


$second = new A;
echo $second->hello() . PHP_EOL;
                          ❺

echo $second->world() . PHP_EOL;
                          ❻

echo $second->universe() . PHP_EOL;
                          ❼

echo $second->abyss() . PHP_EOL;
                          ❽
```

Prints `Bob`.

❶

Prints `Portland`.

❷

Prints `2023`.

❸

Returns a `Warning` as the `::$value` property is private and inaccessible.

Prints `hello`.

❺

Prints `world`.

❻

Throws an `Error` as the `::universe()` method is protected and inaccessible outside the class instance.

This line would not even execute because of the error thrown on the previous line. If the previous line did *not* throw an error, this one would, as the `::abyss()` method is private and inaccessible outside the class instance.

The following recipes further illustrate the preceding concepts and cover some of the most common use cases and implementations of objects in PHP.

8.1 Instantiating Objects from Custom Classes

Problem

You want to define a custom class and create a new object instance from it.

Solution

Define the class and its properties and methods with the `class` keyword, then use `new` to create an instance of it as follows:

```
class Pet
{
    public string $name;
    public string $species;
    public int $happiness = 0;

    public function __construct(string $name, string $s
    {
```

```
            $this->name = $name;
            $this->species = $species;
        }

        public function pet()
        {
            $this->happiness += 1;
        }
    }

    $dog = new Pet('Fido', 'golden retriever');
    $dog->pet();
```

Discussion

The Solution example illustrates several key characteristics of objects:

- Objects can have properties that define the internal state of the object itself.
- These objects can have specific visibility. In the Solution example, objects are `public`, meaning they can be accessed by any code within the application.[5]
- The magic `::__construct()` method can accept parameters only when the object is first instantiated. These parameters can be used to define the initial state of the object.
- Methods can have visibility similar to the properties of the object.

This particular version of class definition is the default many developers have been using since PHP 5, when true object-oriented primitives were first introduced. However, Example 8-6 demonstrates a newer and simpler way to define a simple object like the one in the Solution example. Rather than independently declaring and then directly assigning the properties that carry the object's state, PHP 8 (and later) allows for defining everything within the constructor itself.

Example 8-6. Constructor promotion in PHP 8

```
    class Pet
    {
        public int $happiness = 0;
```

```
        public function __construct(
            public string $name,
            public string $species
        ) {}

        public function pet()
        {
            $this->happiness += 1;
        }
    }

    $dog = new Pet('Fido', 'golden retriever');
    $dog->pet();
```

The Solution example and Example 8-6 are functionally equivalent and will result in objects of the same internal structure being created at runtime. PHP's ability to promote constructor arguments to object properties, however, dramatically reduces the amount of repetitive code you need to type while defining a class.

Each constructor argument also permits the same types of visibility (`public` / `protected` / `private`) that object properties do.[6] The shorthand syntax means you don't need to declare properties, then define parameters, then map the parameters onto those properties upon object instantiation.

See Also

Documentation on classes and objects and the original RFC on constructor promotion.

8.2 Constructing Objects to Define Defaults

Problem

You want to define default values for your object's properties.

Solution

Define default values for the constructor's arguments as follows:

```
class Example
{
    public function __construct(
        public string $someString = 'default',
        public int    $someNumber = 5
    ) {}
}

$first = new Example;
$second = new Example('overridden');
$third = new Example('hitchhiker', 42);
$fourth = new Example(someNumber: 10);
```

Discussion

The constructor function within a class definition behaves more or less like any other function in PHP, except it does not return a value. You can define default arguments similarly to how you would with a standard function. You can even reference the names of the constructor arguments to accept default values for *some* parameters while defining others later in the function signature.

The Solution example explicitly defines the class properties by using constructor promotion for brevity, but the older-style verbose constructor definition is equally valid as follows:

```
class Example
{
    public string $someString;
    public int $someNumber;

    public function __construct(
        string $someString = 'default',
        int    $someNumber = 5
    )
    {
        $this->someString = $someString;
        $this->someNumber = $someNumber;
```

```
        }
    }
```

Similarly, if *not* using constructor promotion, you can initialize object properties directly by assigning a default value when they're defined. When doing so, you would usually leave those parameters off the constructor and manipulate them elsewhere in the program, as shown in the following example:

```
class Example
{
    public string $someString = 'default';
    public int $someNumber = 5;
}

$test = new Example;
$test->someString = 'overridden';
$test->someNumber = 42;
```

WARNING

As will be discussed in Recipe 8.3, you cannot initialize a `readonly` class property directly with a default. This is equivalent to a class constant, and the syntax is therefore disallowed.

See Also

Recipe 3.2 for more on default function parameters, Recipe 3.3 for named function parameters, and documentation on constructors and destructors.

8.3 Defining Read-Only Properties in a Class

Problem

You want to define your class in such a way that properties defined at instantiation cannot be changed after the object exists.

Solution

Use the `readonly` keyword on a typed property:

```
class Book
{
    public readonly string $title;

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

$book = new Book('PHP Cookbook');
```

If using constructor promotion, place the keyword along with the property type within the constructor:

```
class Book
{
    public function __construct(public readonly string
}

$book = new Book('PHP Cookbook');
```

Discussion

The `readonly` keyword was introduced in PHP 8.1 and was aimed at reducing the need for more verbose workarounds originally required to achieve the same functionality. With this keyword, a property can only be initialized with a value *once*, and only while the object is being instantiated.

NOTE

Read-only properties cannot have a default value. This would make them functionally equivalent to class constants, which already exist, so the functionality is unavailable and the syntax is unsupported. Promoted constructor properties can, however, leverage default values within the argument definition as these are evaluated at runtime.

The keyword is also valid only for typed properties. Types are typically optional in PHP (except when strict typing is being used[7]) to aid in flexibility, so it's possible a property on your class *can't* be set to one or another type. In those instances, use the `mixed` type instead so you can set a read-only property without other type constraints.

NOTE

At the time of this writing, read-only declarations are *not* supported on static properties.

As a read-only property can only be instantiated once, it cannot be unset or modified by other subsequent code. All of the code in Example 8-7 will result in the throwing of an `Error` exception.

Example 8-7. Erroneous attempts to modify a read-only property

```
class Example
{
    public readonly string $prop;
}

class Second
{
    public function __construct(public readonly int $cc
}

$first = new Example;
                                    ❶

$first->prop = 'test';
                                    ❷


$test = new Second;
$test->count += 1;
                                    ❸

$test->count++;
                                    ❹

++$test->count;
                                    ❺
```

```
unset($test->count);
```
❻

❶ The `Example` object will have an uninitialized `::$prop` property that cannot be accessed (accessing a property before initialization throws an `Error` exception).

❷ As the object is already instantiated, attempting to write to a read-only property throws an `Error`.

❸ The `::$count` property is read-only, so you cannot assign a new value to it without an `Error`.

❹ As the `::$count` property is read-only, you cannot increment it directly.

❺ You cannot increment in either direction with a read-only property.

❻ You cannot unset a read-only property.

The properties within a class can, however, be other classes themselves. In those situations, a read-only declaration on the property means that property cannot be overwritten or unset, but has no impact on the properties of the child class. For example:

```
class First
{
    public function __construct(public readonly Second
}

class Second
{
    public function __construct(public int $counter = 0
}

$test = new First(new Second);
$test->inner->counter += 1;
```
❶

```
$test->inner = new Second;
```
❷

❶ The increment of the internal counter will succeed as the `::$counter` property is itself not declared as read-only.

❷

The `::$inner` property is read-only and cannot be overridden. Attempts to do so result in an `Error` exception.

See Also

Documentation on read-only properties.

8.4 Deconstructing Objects to Clean Up After the Object Is No Longer Needed

Problem

Your class definition wraps an expensive resource that must be carefully cleaned up when the object goes out of scope.

Solution

Define a class destructor to clean up after the object is removed from memory as follows:

```
class DatabaseHandler
{
    // ...

    public function __destruct()
    {
        dbo_close($this->dbh);
    }
}
```

Discussion

When an object falls out of scope, PHP will automatically garbage-collect any memory or other resources that object used to represent. There might be times, however, when you want to force a specific action when that object goes out of scope. This might be releasing a database handle, as with the Solution example. It could be explicitly logging an event to a file. Or perhaps *deleting* a temporary file from the system, as shown in Example 8-8.

Example 8-8. Removing a temporary file in a destructor

```
class TempLogger
{
    private string $filename;
    private mixed  $handle;

    public function __construct(string $name)
    {
        $this->filename = sprintf('tmp_%s_%s.tmp', $nam
        $this->handle = fopen($this->filename, 'w');
    }

    public function writeLog(string $line): void
    {
        fwrite($this->handle, $line . PHP_EOL);
    }

    public function getLogs(): Generator
    {
        $handle = fopen($this->filename, 'r');
        while(($buffer = fgets($handle, 4096)) !== fals
            yield $buffer;
        }
        fclose($handle);
    }

    public function __destruct()
    {
        fclose($this->handle);
        unlink($this->filename);
    }
}

$logger = new TempLogger('test');
                          ❶

$logger->writeLog('This is a test');
                          ❷

$logger->writeLog('And another');

foreach($logger->getLogs() as $log) {
                          ❸

    echo $log;
}
```

```
unset($logger);
                                                    ❹
```

❶ The object will automatically create a file in the current directory with a name similar to *tmp_test_1650837172.tmp*.

❷ Every new log entry is written as a new line in the temporary log file.

❸ Accessing the logs will create a second handle on the same file, but for reading. The object exposes this handle through a generator that enumerates over every line in the file.

❹ When the logger is removed from scope (or explicitly unset), the destructor will close the open file handle and automatically delete the file as well.

This more sophisticated example demonstrates both how a destructor would be written and how it would be invoked. PHP will look for a `::__destruct()` method on any object when it goes out of scope and will invoke it at that point in time. This destructor explicitly dereferences the object by calling `unset()` to remove it from the program. You could just as easily have set the variable referencing the object to `null` with the same result.

Unlike object constructors, destructors do not accept any parameters. If your object needs to act on any external state while cleaning up after itself, be sure that state is referenced through a property on the object itself. Otherwise, you will not have access to that information.

See Also

Documentation on constructors and destructors.

8.5 Using Magic Methods to Provide Dynamic Properties

Problem

You want to define a custom class without predefining the properties it supports.

Solution

Use magic getters and setters to handle dynamically defined properties as follows:

```
class Magical
{
    private array $_data = [];

    public function __get(string $name): mixed
    {
        if (isset($this->_data[$name])) {
            return $this->_data[$name];
        }

        throw new Error(sprintf('Property `%s` is not c
    }

    public function __set(string $name, mixed $value)
    {
        $this->_data[$name] = $value;
    }
}

$first = new Magical;
$first->custom = 'hello';
$first->another = 'world';

echo $first->custom . ' ' . $first->another . PHP_EOL;

echo $first->unknown; // Error
```

Discussion

When you reference a property on an object that does not exist, PHP falls back on a set of *magic methods* to fill in the blanks on implementation. The getter is used automatically when attempting to reference a property, while the corresponding setter is used when assigning a value to a property.

NOTE

Property overloading by way of magic methods only works on instantiated objects. It does not work on the static class definition.

Internally, you then control the behavior of getting and setting data in its entirety. The Solution example stores its data in a private associative array. You can further flesh out this example by fully implementing magic methods for handling `isset()` and `unset()`. Example 8-9 demonstrates how magic methods can be used to fully replicate a standard class definition, but without the need to predeclare all properties.

Example 8-9. Full object definition with magic methods

```
class Basic
{
    public function __construct(
        public string $word,
        public int $number
    ) {}
}

class Magic
{
    private array $_data = [];

    public function __get(string $name): mixed
    {
        if (isset($this->_data[$name])) {
            return $this->_data[$name];
        }

        throw new Error(sprintf('Property `%s` is not c
    }

    public function __set(string $name, mixed $value)
    {
        $this->_data[$name] = $value;
    }

    public function __isset(string $name): bool
    {
        return array_key_exists($name, $this->_data);
```

```
        }

        public function __unset(string $name): void
        {
            unset($this->_data[$name]);
        }
    }

    $basic = new Basic('test', 22);

    $magic = new Magic;
    $magic->word = 'test';
    $magic->number = 22;
```

In Example 8-9, the two objects are functionally equivalent if and only if the only dynamic properties used on a `Magic` instance are those already defined by `Basic`. This dynamic nature is what makes the approach so valuable even if the class definitions are painfully verbose. You might choose to wrap a remote API in a class implementing magic methods in order to expose that API's data to your application in an object-oriented manner.

See Also

Documentation on magic methods.

8.6 Extending Classes to Define Additional Functionality

Problem

You want to define a class that adds functionality to an existing class definition.

Solution

Use the `extends` keyword to define additional methods or override existing functionality as follows:

```
class A
{
    public function hello(): string
    {
        return 'hello';
    }
}

class B extends A
{
    public function world(): string
    {
        return 'world';
    }
}

$instance = new B();
echo "{$instance->hello()} {$instance->world()}";
```

Discussion

Object inheritance is a common concept for any high-level language; it's a way you build new objects on top of other, often simpler object definitions. The Solution example illustrates how a class can *inherit* method definitions from a parent class, which is the core functionality of PHP's inheritance model.

WARNING

PHP does not support inheriting from multiple parent classes. To pull in code implementations from multiple sources, PHP leverages *traits*, which are covered by Recipe 8.13.

In fact, a child class inherits every public and protected method, property, and constant from its parent class (the class it's extending). Private methods, properties, and constants are *never* inherited by a child class.[8]

A child class can also override its parent's implementation of a particular method. In practice, you would do this to change the internal logic of a particular method, but the method signature exposed by the child class must

match that defined by the parent. Example 8-10 demonstrates how a child class would override its parent's implementation of a method.

Example 8-10. Overriding a parent method implementation

```
class A
{
    public function greet(string $name): string
    {
        return 'Good morning, ' . $name;
    }
}

class B extends A
{
    public function greet(string $name): string
    {
        return 'Howdy, ' . $name;
    }
}

$first = new A();
echo $first->greet('Alice');
                           ❶

$second = new B();
echo $second->greet('Bob');
                           ❷
```

❶ Prints `Good morning, Alice`

❷ Prints `Howdy, Bob`

An overridden child method does not lose all sense of the parent's implementation, though. Inside a class, you reference the `$this` variable to refer to that particular instance of the object. Likewise, you can reference the `parent` keyword to refer to the parent implementation of a function. For example:

```
class A
{
    public function hello(): string
    {
        return 'hello';
```

```
    }
}

class B extends A
{
    public function hello(): string
    {
        return parent::hello() . ' world';
    }
}

$instance = new B();
echo $instance->hello();
```

See Also

Documentation and discussion of PHP's object inheritance model.

8.7 Forcing Classes to Exhibit Specific Behavior

Problem

You want to define the methods on a class that will be used elsewhere in your application while leaving the actual method implementations up to someone else.

Solution

Define an object interface and leverage that interface in your application as follows:

```
interface ArtifactRepository
{
    public function create(Artifact $artifact): bool;
    public function get(int $artifactId): ?Artifact;
    public function getAll(): array;
    public function update(Artifact $artifact): bool;
    public function delete(int $artifactId): bool;
}
```

```
class Museum
{
    public function __construct(
        protected ArtifactRepository $repository
    ) {}

    public function enumerateArtifacts(): Generator
    {
        foreach($this->repository->getAll() as $artifac
            yield $artifact;
        }
    }
}
```

Discussion

An interface looks similar to a class definition, except it only defines the *signatures* of specific methods rather than their implementations. The interface does, however, define a type that can be used elsewhere in your application—so long as a class directly implements a given interface, an instance of that class can be used as if it were the same type as the interface itself.

WARNING

There are several situations in which you might have two classes that implement the same methods and expose the same signatures to your application. However, unless these classes explicitly implement the same interface (as evidenced by the `implements` keyword), they cannot be used interchangeably in a strictly typed application.

An implementation must use the `implements` keyword to tell the PHP compiler what's going on. The Solution example illustrates how an interface is defined and how another part of the code can leverage that interface. Example 8-11 demonstrates how the `ArtifactRepository` interface might be implemented using an in-memory array for data storage.

Example 8-11. Explicit interface implementation

```
class MemoryRepository implements ArtifactRepository
{
    private array $_collection = [];

    private function nextKey(): int
    {
        $keys = array_keys($this->_collection);
        $max = array_reduce($keys, function($c, $i) {
            return max($c, $i);
        }, 0);

        return $max + 1;
    }

    public function create(Artifact $artifact): bool
    {
        if ($artifact->id === null) {
            $artifact->id = $this->nextKey();
        }

        if (array_key_exists($artifact->id, $this->_col
            return false;
        }

        $this->_collection[$artifact->id] = $artifact;
        return true;
    }
    public function get(int $artifactId): ?Artifact
    {
        return $this->_collection[$artifactId] ?? null;
    }
    public function getAll(): array
    {
        return array_values($this->_collection);
    }
    public function update(Artifact $artifact): bool
    {
        if (array_key_exists($artifact->id, $this->_col
            $this->_collection[$artifact->id] = $artifa
            return true;
        }

        return false;
```

```
        }
        public function delete(int $artifactId): bool
        {
            if (array_key_exists($artifactId, $this->_colle
                unset($this->_collection[$artifactId]);
                return true;
            }

            return false;
        }
    }
```

Throughout your application, any method can declare a type on a parameter by using the interface itself. The Solution example's `Museum` class takes a concrete implementation of the `ArtifactRepository` as its only parameter. This class can then operate knowing what the exposed API of the repository will look like. The code doesn't care *how* each method is implemented, only that it matches the interface's defined signature exactly.

A class definition can implement many different interfaces at once. This allows for a complex object to be used in different situations by different pieces of code. Note that, if two or more interfaces define the same method name, their defined signatures must be identical, as illustrated by Example 8-12.

Example 8-12. Implementing multiple interfaces at once

```
interface A
{
    public function foo(): int;
}

interface B
{
    public function foo(): int;
}

interface C
{
    public function foo(): string;
}

class First implements A, B
```

```
{
    public function foo(): int
                                    ❶

    {
        return 1;
    }
}

class Second implements A, C
{
    public function foo(): int|string
                                    ❷

    {
        return 'nope';
    }
}
```

❶ As both `A` and `B` define the same method signature, this implementation is valid.

❷ Since `A` and `C` define different return types, there is no way, even with union types, to define a class that implements both interfaces. Attempting to do so causes a fatal error.

Remember also that interfaces look somewhat like classes so, like classes, they can be extended.[9] This is done through the same `extends` keyword and results in an interface that is a composition of two or more interfaces, as demonstrated in Example 8-13.

Example 8-13. Composite interfaces

```
interface A
                                    ❶

{
    public function foo(): void;
}

interface B extends A
                                    ❷

{
    public function bar(): void;
}
```

```
class C implements B
{
    public function foo(): void
    {
        // ... actual implementation
    }

    public function bar(): void
    {
        // ... actual implementation
    }
}
```

Any class implementing `A` must define the `foo()` method.

Any class implementing `B` must implement both `bar()` and `foo()` from `A`.

See Also

Documentation on object interfaces.

8.8 Creating Abstract Base Classes

Problem

You want a class to implement a specific interface but also want to define some other specific functionality.

Solution

Rather than implementing an interface, define an abstract base class that can be extended as follows:

```
abstract class Base
{
    abstract public function getData(): string;

    public function printData(): void
    {
        echo $this->getData();
    }
```

```
}

class Concrete extends Base
{
    public function getData(): string
    {
        return bin2hex(random_bytes(16));
    }
}

$instance = new Concrete;
$instance->printData();
```

❶

Prints something like `6ec2aff42d5904e0ccef15536d8548dc`

Discussion

An abstract class looks somewhat like an interface and a regular class definition smashed together. It has some unimplemented methods living alongside concrete implementations. As with an interface, you cannot instantiate an abstract class directly—you have to extend it first and implement any abstract methods it defines. As with a class, however, you will automatically have access to any public or protected members of the base class in the child implementation.[10]

One key difference between interfaces and abstract classes is that the latter can bundle properties and method definitions with it. Abstract classes are, in fact, classes that are merely incomplete implementations. An interface cannot have properties—it merely defines the functional interface with which any implementing object must comply.

Another difference is that you can implement multiple interfaces simultaneously, but you can only extend one class at a time. This limitation alone helps characterize when you would leverage an abstract base class versus an interface—but you can also mix and match both!

It's also possible for an abstract class to define *private* members (which are not inherited by any child class) that are otherwise leveraged by accessible methods, as illustrated by the following:

```php
abstract class A
{
    private string $data = 'this is a secret';
                                              ❶


    abstract public function viewData(): void;

    public function getData(): string
    {
        return $this->data;
                                              ❷

    }
}

class B extends A
{
    public function viewData(): void
    {
        echo $this->getData() . PHP_EOL;
                                              ❸

    }
}

$instance = new B();
$instance->viewData();
                                              ❹
```

❶ By making your data private, it is only accessible within the context of `A`.

❷ As `::getData()` is itself defined by `A`, the `$data` property is still accessible.

❸ Though `::viewData()` is defined in the scope of `B`, it is accessing a public method from the scope of `A`. No code in `B` would have direct access to `A`'s private members.

❹ This will print `this is a secret` to the console.

See Also

Documentation and discussion of class abstraction.

8.9 Preventing Changes to Classes and Methods

Problem

You want to prevent anyone from modifying the implementation of your class or extending it with a child class.

Solution

Use the `final` keyword to indicate that a class is not extensible, as follows:

```
final class Immutable
{
    // Class definition
}
```

Or use the `final` keyword to mark a particular method as unchangeable, as follows:

```
class Mutable
{
    final public function fixed(): void
    {
        // Method definition
    }
}
```

Discussion

The `final` keyword is a way to explicitly *prevent* object extension like the mechanisms discussed in the previous two recipes. It's useful when you want to ensure that a specific implementation of a method or an entire class is used throughout a codebase.

Marking a method as `final` means that any class extensions are incapable of overriding that method's implementation. The following example will

throw a fatal error due to the `Child` class's attempt to override a `final` method in the `Base` class:

```
class Base
{
    public function safe()
    {
        echo 'safe() inside Base class' . PHP_EOL;
    }

    final public function unsafe()
    {
        echo 'unsafe() inside Base class' . PHP_EOL;
    }
}

class Child extends Base
{
    public function safe()
    {
        echo 'safe() inside Child class' . PHP_EOL;
    }

    public function unsafe()
    {
        echo 'unsafe() inside Child class' . PHP_EOL;
    }
}
```

In the preceding example, merely omitting the definition of `unsafe()` from the child class will allow the code to execute as expected. If, however, you wanted to prevent any class from extending the base class, you could add the `final` keyword to the class definition itself as follows:

```
final class Base
{
    public function safe()
    {
        echo 'safe() inside Base class' . PHP_EOL;
    }

    public function unsafe()
    {
```

```
            echo 'unsafe() inside Base class' . PHP_EOL;
        }
    }
```

The only time you should leverage `final` in your code is when overriding a specific method or class implementation will break your application. This is somewhat rare in practice but is useful when creating a flexible interface. A specific example would be when your application introduces an interface as well as concrete implementations of that interface.[11] Your API would then be constructed to accept any valid interface implementation, but you might want to prevent subclassing your own concrete implementations (again because doing so might break your application). Example 8-14 demonstrates how these dependencies might be constructed in a real application.

Example 8-14. Interfaces and concrete classes

```
interface DataAbstraction
                              ❶

{
    public function save();
}

final class DBImplementation implements DataAbstractior
                              ❷

{
    public function __construct(string $databaseConnect
    {
        // Connect to a database
    }

    public function save()
    {
        // Save some data
    }
}

final class FileImplementation implements DataAbstracti
                              ❸

{
    public function __construct(string $filename)
    {
        // Open a file for writing
    }
```

```
        public function save()
        {
            // Write to the file
        }
    }

    class Application
    {
        public function __construct(
            protected DataAbstraction $datalayer
                                      ❹

        ) {}
    }
```

❶ The application describes an interface that any data abstraction layer must implement.

❷ One concrete implementation stores data explicitly in a database.

❸ Another implementation uses flat files for data storage.

❹ The application doesn't care what implementation you use so long as it implements the base interface. You can use either the provided (`final`) classes or define your own implementation.

In some situations, you might come across a `final` class you need to extend anyway. In those cases, the only means at your disposal are to leverage a decorator. A *decorator* is a class that takes another class as a constructor property and "decorates" its methods with additional functionality.

NOTE

In some circumstances, decorators will not allow you to sidestep the `final` nature of a class. This happens if type hinting and strict typing require an instance of that exact class be passed to a function or another object within the application.

Assume, for example, that a library in your application defines a `Note` class that implements a `::publish()` method that publishes a given piece of data to social media (say, Twitter). You want this method to *also* produce a static PDF artifact of the given data and would normally extend the class itself, as shown in Example 8-15.

Example 8-15. Typical class extension without the `final` keyword

```
class Note
{
    public function publish()
    {
        // Publish the note's data to Twitter ...
    }
}

class StaticNote extends Note
{
    public function publish()
    {
        parent::publish();

        // Also produce a static PDF of the note's data
    }
}

$note = new StaticNote();
                            ❶

$note->publish();
                            ❷
```

❶ Rather than instantiating a `Note` object, you can instantiate a `StaticNote` directly.

❷ When you call the object's `::publish()` method, *both* class definitions are used.

If the `Note` class is instead `final`, you will be unable to extend the class directly. Example 8-16 demonstrates how a new class can be created that *decorates* the `Note` class and indirectly extends its functionality.

Example 8-16. Customizing hte behavior of a `final` class with a decorator

```
final class Note
{
    public function publish()
    {
        // Publish the note's data to Twitter ...
    }
```

```
}

final class StaticNote
{
    public function __construct(private Note $note) {}

    public function publish()
    {
        $this->note->publish();

        // Also produce a static PDF of the note's data
    }
}

$note = new StaticNote(new Note());
                                    ❶

$note->publish();
                                    ❷
```

Rather than instantiating `StaticNote` directly, you use this class to *wrap* (or *decorate*) a regular `Note` instance.

When you call the object's `::publish()` method, *both* class definitions are used.

See Also

Documentation on the `final` keyword.

8.10 Cloning Objects

Problem

You want to create a distinct copy of an object.

Solution

Use the `clone` keyword to create a second copy of the object—for example:

```
$dolly = clone $roslin;
```

Discussion

By default, PHP will copy objects *by reference* when assigned to a new variable. This reference means the new variable literally points to the same object in memory. Example 8-17 illustrates how, even though it might appear that you've created a copy of an object, you're really dealing with just two references to the same data.

Example 8-17. The assignment operator copies an object by reference

```
$obj1 = (object) [                          ❶
    'propertyOne' => 'some',
    'propertyTwo' => 'data',
];
$obj2 = $obj1;                              ❷

$obj2->propertyTwo = 'changed';             ❸

var_dump($obj1);                            ❹
var_dump($obj2);
```

❶ This particular syntax is shorthand, valid as of PHP 5.4, that dynamically converts a new associative array to an instance of the built-in `stdClass` class.

❷ Attempt to copy the first object to a new instance by using the assignment operator.

❸ Make a change to the internal state of the "copied" object.

❹ Inspecting the original object shows that its internal state has changed. Both `$obj1` and `$obj2` point to the same space in memory; you merely copied a reference to the object, not the object itself!

Rather than copy an object reference, the `clone` keyword copies an object to a new variable *by value*. This means all of the properties are copied to a new instance of the same class that has all of the methods of the original object as

well. Example 8-18 illustrates how the two objects are now entirely decoupled.

Example 8-18. The `clone` keyword copies an object by value

```
$obj1 = (object) [
    'propertyOne' => 'some',
    'propertyTwo' => 'data',
];
$obj2 = clone $obj1;
                                ❶


$obj2->propertyTwo = 'changed';
                                ❷


var_dump($obj1);
                                ❸

var_dump($obj2);
                                ❹
```

❶ Rather than use strict assignment, leverage the `clone` keyword to create a by-value copy of the object.

❷ Again make a change to the internal state of the copy.

❸ Inspecting the state of the original object shows no changes.

❹ The cloned and changed object, however, illustrates the property modification made earlier.

An important caveat here is to understand that, as used in the preceding examples, `clone` is a *shallow clone* of the data. The operation does not traverse down into more complex properties like nested objects. Even with proper `clone` usage, it is possible to be left with two different variables referencing the same object in memory. Example 8-19 illustrates what happens if the object to be copied contains a more complex object itself.

Example 8-19. Shallow clones of complex data structures

```
$child = (object) [
    'name' => 'child',
];
$parent = (object) [
    'name'  => 'parent',
```

```
    'child' => $child
];

$clone = clone $parent;

if ($parent === $clone) {                    ❶

    echo 'The parent and clone are the same object!' .
}

if ($parent == $clone) {                     ❷

    echo 'The parent and clone have the same data!' . P
}

if ($parent->child === $clone->child) {      ❸

    echo 'The parent and the clone have the same child!
}
```

❶ When comparing objects, strict comparison only resolves as `true` when the statements on either side of the comparison reference the same object. In this case, you've properly cloned your object and created an entirely new instance, so this comparison is `false`.

❷ Loose type comparison between objects resolves as `true` when the *values* on either side of the operator are the same, even between discrete instances. This statement evaluates to `true`.

❸ Since `clone` is a shallow operation, the `::$child` property on both of your objects points to the same child object in memory. This statement evaluates to `true`!

To support a deeper clone, the class being cloned must implement a `__clone()` magic method that tells PHP what to do when leveraging `clone`. If this method exists, PHP will invoke it automatically when closing an instance of the class. Example 8-20 shows exactly how this might work while still working with dynamic classes.

NOTE

It is not possible to dynamically define methods on instances of `stdClass`. If you want to support deep cloning of objects in your application, you must either define a class directly or leverage an anonymous class, as illustrated by Example 8-20.

Example 8-20. Deep cloning of objects

```
$parent = new class {
    public string $name = 'parent';
    public stdClass $child;

    public function __clone()
    {
        $this->child = clone $this->child;
    }
};
$parent->child = (object) [
    'name' => 'child'
];

$clone = clone $parent;

if ($parent === $clone) {
                            ❶

    echo 'The parent and clone are the same object!' .
}

if ($parent == $clone) {
                            ❷

    echo 'The parent and clone have the same data!' . P
}

if ($parent->child === $clone->child) {
                            ❸

    echo 'The parent and the clone have the same child!
}

if ($parent->child == $clone->child) {
                            ❹
```

```
        echo 'The parent and the clone have the same child
    }
```

❶ The objects are different references; therefore, this evaluates to `false`.

❷ The parent and clone objects have the same data, so this evaluates to `true`.

❸ The `::$child` properties were also cloned internally, so the properties reference different object instances. This evaluates to `false`.

❹ Both `::$child` properties contain the same data, so this evaluates to `true`.

In most applications, you will generally be working with custom class definitions and not anonymous classes. In that case, you can still implement the magic `__clone()` method to instruct PHP on how to clone the more complex properties of your object where necessary.

See Also

Documentation on the `clone` keyword.

8.11 Defining Static Properties and Methods

Problem

You want to define a method or property on a class that is available to all instances of that class.

Solution

Use the `static` keyword to define properties or methods that can be accessed outside an object instance—for example:

```
class Foo
{
```

```
    public static int $counter = 0;

    public static function increment(): void
    {
        self::$counter += 1;
    }
}
```

Discussion

Static members of a class are accessible to any part of your code (assuming proper levels of visibility) directly from the class definition, whether or not an instance of that class exists as an object. Static properties are useful since they behave more or less like global variables but are scoped to a specific class definition. Example 8-21 illustrates the difference in invoking a global variable versus a static class property in another function.

Example 8-21. Static properties versus global variables

```
class Foo
{
    public static string $name = 'Foo';
}

$bar = 'Bar';

function demonstration()
{
    global $bar;
                              ❶


    echo Foo::$name . $bar;
                              ❷

}
```

❶ To access a global variable within another scope, you must explicitly refer to the global scope. Given that you can have separate variables in a narrower scope that match the names of global variables, this can become potentially confusing in practice.

❷ A class-scoped property, however, can be accessed directly based on the name of the class itself.

More usefully, static methods provide ways to invoke utility functionality bound to a class prior to instantiating an object of that class directly. One common example is when constructing value objects that should represent serialized data where it would be difficult to construct an object from scratch directly.

Example 8-22 demonstrates a class that does not allow for direct instantiation. Instead, you must create an instance by unserializing some fixed data. The constructor is inaccessible outside of the class's interior scope, so a static method is the only means of creating an object.

Example 8-22. Static method object instantiation

```
class BinaryString
{
    private function __construct(private string $bits)
                                                    ❶


    public static function fromHex(string $hex): self
    {
        return new self(hex2bin($hex));
                                                    ❷
    }

    public static function fromBase64(string $b64): sel
    {
        return new self(base64_decode($b64));
    }

    public function __toString(): string
                                                    ❸

    {
        return bin2hex($this->bits);
    }
}

$rawData = '48656c6c6f20776f726c6421';
$binary = BinaryString::fromHex($rawData);
                                                    ❹
```

❶ A private constructor can be accessed only from within the class itself.

❷

Within a static method, you can still create a new object instance by leveraging the special `self` keyword to refer to the class. This permits you to access your private constructor.

The magic `__toString()` method is invoked whenever PHP tries to coerce an object into a string directly (i.e., when you try to `echo` it to the console).

Rather than creating an object with the `new` keyword, leverage a purpose-built static deserialization method.

Both static methods and properties are subject to the same visibility constraints as their nonstatic peers. Note that marking either as `private` means they can only be referenced by one another or by nonstatic methods within the class itself.

As static methods and properties aren't tied directly to an object instance, you can't use regular object-bound accessors to reach them. Instead, leverage the class name directly and the scope resolution operator (a double colon, or `::`) —for example, `Foo::$bar` for properties or `Foo::bar()` for methods. Within the class definition itself, you can leverage `self` as a shorthand for the class name or `parent` as a shorthand for the parent class name (if using inheritance).

NOTE

If you have access to an object instance of the class, you can use that object's name rather than the class name to access its static members as well. For example, you can use `$foo::bar()` to access the static `bar()` method on the class definition for the object named `$foo`. While this works, it can make it more difficult for other developers to understand what class definition you're working with, so this syntax should be used sparingly if at all.

See Also

Documentation on the `static` keyword.

8.12 Introspecting Private Properties or

Methods Within an Object

Problem

You want to enumerate the properties or methods of an object and leverage its private members.

Solution

Use PHP's Reflection API to enumerate properties and methods. For example:

```
$reflected = new ReflectionClass('SuperSecretClass');

$methods = $reflected->getMethods();
                              ❶

$properties = $reflected->getProperties();
                              ❷
```

Discussion

PHP's Reflection API grants developers vast power to inspect all elements of their application. You can enumerate methods, properties, constants, function arguments, and more. You can also ignore the privacy afforded to each at will and directly invoke private methods on objects. Example 8-23 illustrates how an explicitly private method can be invoked directly with the Reflection API.

Example 8-23. Using Reflection to violate class privacy

```
class Foo
{
    private int $counter = 0;
                           ❶


    public function increment(): void
    {
        $this->counter += 1;
    }

    public function getCount(): int
```

```
    {
        return $this->counter;
    }
}

$instance = new Foo;
$instance->increment();
                          ❷

$instance->increment();
                          ❸


echo $instance->getCount() . PHP_EOL;
                          ❹


$instance->counter = 0;
                          ❺


$reflectionClass = new ReflectionClass('Foo');
$reflectionClass->getProperty('counter')->setValue($ins
                          ❻


echo $instance->getCount() . PHP_EOL;
                          ❼
```

❶ The example class has a single, private property to maintain an internal counter.

❷ You want to increment the counter a bit past its default. Now it's `1`.

❸ An additional increment sets the counter to `2`.

❹ At this point, printing out the counter's state will confirm it's `2`.

❺ Attempting to interact with the counter *directly* will result in a thrown `Error`, as the property is private.

❻ Through reflection, you can interact with object members regardless of their privacy setting.

❼ Now you demonstrate the counter was truly reset to `0`.

Reflection is a truly powerful way to get around visibility modifiers in the API exposed by a class. However, its use in a production application likely points to a poorly constructed interface or system. If your code needs access to a

private member of a class, either that member should be public to begin with or you need to create an appropriate accessor method.

The only legitimate use of Reflection is in inspecting and modifying the internal state of an object. In an application, this behavior should be limited to the class's exposed API. In *testing*, however, it might be necessary to modify an object's state between test runs in ways the API doesn't support during regular operation.[12] Those rare circumstances might require resetting internal counters or invoking otherwise private cleanup methods housed within the class. It's then that Reflection proves its utility.

In regular application development, though, Reflection paired with functional calls like `var_dump()` helps to disambiguate the internal operation of classes defined in imported vendor code. It might prove useful to introspect serialized objects or third-party integrations, but take care not to ship this kind of introspection to production.

See Also

Overview of PHP's Reflection API.

8.13 Reusing Arbitrary Code Between Classes

Problem

You want to share a particular piece of functionality between multiple classes without leveraging a class extension.

Solution

Import a Trait with a `use` statement—for example:

```
trait Logger
{
    public function log(string $message): void
    {
        error_log($message);
```

```
    }
}

class Account
{
    use Logger;                          ❶


    public function __construct(public int $accountNumb
    {
        $this->log("Created account {$accountNumber}.")
    }
}

class User extends Person
{
    use Logger;                          ❷


    public function authenticate(): bool
    {
        // ...
        $this->log("User {$userId} logged in.");
        // ...
    }
}
```

The `Account` class imports the logging functionality from your `Logger` trait and can use its methods as if they were native to its own definition. ❶

Likewise, the `User` class has native-level access to `Logger`'s methods, even though it extends a base `Person` class with additional functionality. ❷

Discussion

As discussed in Recipe 8.6, a class in PHP can descend from at most one other class. This is referred to as *single inheritance* and is a characteristic of languages other than PHP as well. Luckily, PHP exposes an additional mechanism for code reuse called *Traits*. Traits allow for the encapsulation of some functionality in a separate class-like definition that can be easily imported without breaking single inheritance.

A Trait looks somewhat like a class but cannot be instantiated directly. Instead, the methods defined by a Trait are imported into another class definition by the `use` statement. This permits code reuse between classes that do not share an inheritance tree.

Traits empower you to define common methods (with differing method visibility) and properties that are shared between definitions. You can also override the default visibility from a Trait in the class that imports it. Example 8-24 shows how an otherwise public method defined in a Trait can be imported as a protected or even private method in another class.

Example 8-24. Overriding the visibility of methods defined in a Trait

```
trait Foo
{
    public function bar()
    {
        echo 'Hello World!';
    }
}

class A
{
    use Foo { bar as protected; }
                              ❶

}

class B
{
    use Foo { bar as private; }
                              ❷

}
```

❶ This syntax will import every method defined in `Foo` but will explicitly make its `::bar()` method protected within the scope of class `A`. This means only instances of class `A` (or its descendants) will be able to invoke the method.

❷ Likewise, class `B` changes the visibility of its imported `::foo()` method to private so only instances of `B` can access that method directly.

Traits can be composed together as deeply as you want, meaning a Trait can `use` another Trait just as easily as a class can. Likewise, there is no limit to the number of Traits that can be imported either by other Traits or by class definitions.

If the class importing a Trait (or multiple Traits) defines a method also named in the Trait, then the class's version takes precedence and is used by default. Example 8-25 illustrates how this precedence works by default.

Example 8-25. Method precedence in Traits

```
trait Foo
{
    public function bar(): string
    {
        return 'FooBar';
    }
}

class Bar
{
    use Foo;
    public function bar(): string
    {
        return 'BarFoo';
    }
}

$instance = new Bar;
echo $instance->bar(); // BarFoo
```

In some circumstances, you might import multiple Traits that all define the same method. In those situations, you can explicitly identify which version of a method you want to leverage in your final class when you define your `use` statement as follows:

```
trait A
{
    public function hello(): string
    {
        return 'Hello';
    }
```

```
        public function world(): string
        {
            return 'Universe';
        }
    }

    trait B
    {
        public function world(): string
        {
            return 'World';
        }
    }

    class Demonstration
    {
        use A, B {
            B::world insteadof A;
        }
    }

    $instance = new Demonstration;
    echo "{$instance->hello()} {$instance->world()}!"; // H
```

Like class definitions, Traits can also define properties or even static members. They provide an efficient means by which you can abstract operational logic definitions into reusable code blocks and share that logic between classes in your application.

See Also

Documentation on Traits.

[1] PHP 3 included some primitive object functionality, but the language wasn't really considered object-oriented by most developers until the delivery of 4.0.

[2] At the time of this writing, WordPress was used to power 43% of all websites.

[3] Libraries and extensions are discussed at length in Chapter 15.

[4] See Recipe 8.12 for more on the Reflection API.

[5] Review "Visibility" for more background on visibility within classes.

[6]

[6]
See Recipe 8.3 for details on how these properties could further be made read-only.

[7]
See Recipe 3.4 for more on strict type enforcement.

[8]
For more on property and method visibility, review “Visibility”.

[9]
Review Recipe 8.6 for more on class inheritance and extension.

[10]
See Recipe 8.6 for more on class inheritance.

[11]
See Recipe 8.7 for more on interfaces.

[12]
Both testing and debugging are discussed at length in Chapter 13.