Object interfaces allow you to create code which specifies which methods a class must implement, without having to define how these methods are implemented. Interfaces share a namespace with classes and traits, so they may not use the same name.

Interfaces are defined in the same way as a class, but with the interface keyword replacing the class keyword and without any of the methods having their contents defined.

All methods declared in an interface must be public; this is the nature of an interface.

In practice, interfaces serve two complementary purposes:

* To allow developers to create objects of different classes that may be used interchangeably because they implement the same interface or interfaces. A common example is multiple database access services, multiple payment gateways, or different caching strategies. Different implementations may be swapped out without requiring any changes to the code that uses them.
* To allow a function or method to accept and operate on a parameter that conforms to an interface, while not caring what else the object may do or how it is implemented. These interfaces are often named like Iterable, Cacheable, Renderable, or so on to describe the significance of the behavior.

Interfaces may define [magic methods](https://www.php.net/manual/en/language.oop5.magic.php) to require implementing classes to implement those methods.

Note:

Although they are supported, including [constructors](https://www.php.net/manual/en/language.oop5.decon.php#language.oop5.decon.constructor) in interfaces is strongly discouraged. Doing so significantly reduces the flexibility of the object implementing the interface. Additionally, constructors are not enforced by inheritance rules, which can cause inconsistent and unexpected behavior.

implements[¶](https://www.php.net/manual/en/language.oop5.interfaces.php" \l "language.oop5.interfaces.implements)

To implement an interface, the implements operator is used. All methods in the interface must be implemented within a class; failure to do so will result in a fatal error. Classes may implement more than one interface if desired by separating each interface with a comma.

**Warning**

A class can implement two interfaces which define a method with the same name, only if the method declaration in both interfaces is identical.

**Warning**

A class that implements an interface may use a different name for its parameters than the interface. However, as of PHP 8.0 the language supports [named arguments](https://www.php.net/manual/en/functions.arguments.php#functions.named-arguments), which means callers may rely on the parameter name in the interface. For that reason, it is strongly recommended that developers use the same parameter names as the interface being implemented.

Note:

Interfaces can be extended like classes using the [extends](https://www.php.net/manual/en/language.oop5.inheritance.php) operator.

Note:

The class implementing the interface must declare all methods in the interface with a [compatible signature](https://www.php.net/manual/en/language.oop5.basic.php#language.oop.lsp).

Constants[¶](https://www.php.net/manual/en/language.oop5.interfaces.php#language.oop5.interfaces.constants)

It's possible for interfaces to have constants. Interface constants work exactly like [class constants](https://www.php.net/manual/en/language.oop5.constants.php). Prior to PHP 8.1.0, they cannot be overridden by a class/interface that inherits them.

Examples[¶](https://www.php.net/manual/en/language.oop5.interfaces.php#language.oop5.interfaces.examples)

Example #1 Interface example

<?php  
  
// Declare the interface 'Template'  
interface Template  
{  
    public function setVariable($name, $var);  
    public function getHtml($template);  
}  
  
// Implement the interface  
// This will work  
class WorkingTemplate implements Template  
{  
    private $vars = [];  
    
    public function setVariable($name, $var)  
    {  
        $this->vars[$name] = $var;  
    }  
    
    public function getHtml($template)  
    {  
        foreach($this->vars as $name => $value) {  
            $template = str\_replace('{' . $name . '}', $value, $template);  
        }  
   
        return $template;  
    }  
}  
  
// This will not work  
// Fatal error: Class BadTemplate contains 1 abstract methods  
// and must therefore be declared abstract (Template::getHtml)  
class BadTemplate implements Template  
{  
    private $vars = [];  
    
    public function setVariable($name, $var)  
    {  
        $this->vars[$name] = $var;  
    }  
}  
?>

Example #2 Extendable Interfaces

<?php  
interface A  
{  
    public function foo();  
}  
  
interface B extends A  
{  
    public function baz(Baz $baz);  
}  
  
// This will work  
class C implements B  
{  
    public function foo()  
    {  
    }  
  
    public function baz(Baz $baz)  
    {  
    }  
}  
  
// This will not work and result in a fatal error  
class D implements B  
{  
    public function foo()  
    {  
    }  
  
    public function baz(Foo $foo)  
    {  
    }  
}  
?>

Example #3 Multiple interface inheritance

<?php  
interface A  
{  
    public function foo();  
}  
  
interface B  
{  
    public function bar();  
}  
  
interface C extends A, B  
{  
    public function baz();  
}  
  
class D implements C  
{  
    public function foo()  
    {  
    }  
  
    public function bar()  
    {  
    }  
  
    public function baz()  
    {  
    }  
}  
?>

Example #4 Interfaces with constants

<?php  
interface A  
{  
    const B = 'Interface constant';  
}  
  
// Prints: Interface constant  
echo A::B;  
  
  
class B implements A  
{  
    const B = 'Class constant';  
}  
  
// Prints: Class constant  
// Prior to PHP 8.1.0, this will however not work because it was not  
// allowed to override constants.  
echo B::B;  
?>

Example #5 Interfaces with abstract classes

<?php  
interface A  
{  
    public function foo(string $s): string;  
  
    public function bar(int $i): int;  
}  
  
// An abstract class may implement only a portion of an interface.  
// Classes that extend the abstract class must implement the rest.  
abstract class B implements A  
{  
    public function foo(string $s): string  
    {  
        return $s . PHP\_EOL;  
    }  
}  
  
class C extends B  
{  
    public function bar(int $i): int  
    {  
        return $i \* 2;  
    }  
}  
?>

Example #6 Extending and implementing simultaneously

<?php  
  
class One  
{  
    /\* ... \*/  
}  
  
interface Usable  
{  
    /\* ... \*/  
}  
  
interface Updatable  
{  
    /\* ... \*/  
}  
  
// The keyword order here is important. 'extends' must come first.  
class Two extends One implements Usable, Updatable  
{  
    /\* ... \*/  
}  
?>

An interface, together with type declarations, provides a good way to make sure that a particular object contains particular methods. See [instanceof](https://www.php.net/manual/en/language.operators.type.php) operator and [type declarations](https://www.php.net/manual/en/language.types.declarations.php).

[＋add a note](https://www.php.net/manual/add-note.php?sect=language.oop5.interfaces&redirect=https://www.php.net/manual/en/language.oop5.interfaces.php)

User Contributed Notes 4 notes

[up](https://www.php.net/manual/vote-note.php?id=102755&page=language.oop5.interfaces&vote=up)

[down](https://www.php.net/manual/vote-note.php?id=102755&page=language.oop5.interfaces&vote=down)

30

[thanhn2001 at gmail dot com ¶](https://www.php.net/manual/en/language.oop5.interfaces.php#102755)

11 years ago

PHP prevents interface a contant to be overridden by a class/interface that DIRECTLY inherits it.  However, further inheritance allows it.  That means that interface constants are not final as mentioned in a previous comment.  Is this a bug or a feature?  
  
<?php  
  
interface a  
{  
    const b = 'Interface constant';  
}  
  
// Prints: Interface constant  
echo a::b;  
  
class b implements a  
{  
}  
  
// This works!!!  
class c extends b  
{  
    const b = 'Class constant';  
}  
  
echo c::b;  
?>