final Keyword

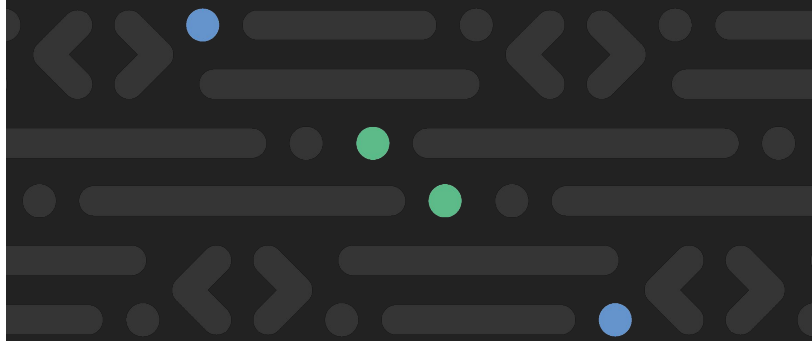
The **final** keyword. You can apply it either to a whole class, or to specific methods within a class. The effect of the final keyword is to prevent classes from being extended or methods from being overridden. The visibility of all final methods is public.

```php
<?php

final class SecureAuthenticator {
   public function authenticate($username, $password) {
       // Security-critical authentication logic
   }

   // Other security-related methods
}

class CustomAuth extends SecureAuthenticator {
   // Custom authentication functionality
}

$auth = new CustomAuth;
// Fatal error: Class CustomAuth cannot extend final class SecureAuthenticator
```

parent Keyword

The **parent** keyword is used to call the overridden method from within the subclass.

```php
<?php
class ParentClass
{
   public function __construct()
   {
       echo "Hello";
   }
}
class ChildClass extends ParentClass
{
   public function __construct()
   {
       parent::__construct();
       echo " World";
   }
}
new ChildClass();
```



Maharah
Code Your Future

Abstract Classes

PHP supports abstract classes, which are classes that contain one or more abstract methods. An abstract method is a method that has been declared but not implemented.

An abstract class cannot be constructed; we cannot create a new object from the class Paintings.

```php
<?php
abstract class Paintings
{
    abstract protected function girlDescendingStairs();
    protected function persistenceOfMemory()
    {
        echo " I have an implementation so this is not an abstract method ";
    }
    public function __construct()
    {
        echo "I cannot be constructed!";
    }
}
```

Maharah
Code Your Future

Abstract Classes (2)

```php
<?php
abstract class Course
{
   abstract protected function maharah();
}

class NewCourse extends Course
{
   public function maharah()
   {
       echo " I have an implementation so this is not an abstract method ";
   }
}
```

Maharah
Code Your Future

Question?

02

What will the output of this code be?

```php
<?php
class World{
  public static function hello() {
      echo "Hello " . __CLASS__;
  }
}
class Maharah extends World {
  public function welcome() {
      echo "I have the world";
  }
}
Maharah::hello();
```

Interfaces

- Interfaces allow you to specify what methods a class must implement without specifying the details of the implementation.
- They are commonly used to define a contract in the service-oriented architecture paradigm, but can also be used whenever you want to stipulate how future classes are expected to interact with your code.
- All methods in an interface must be declared as public and may not have any implementation themselves.
- Interfaces cannot have properties, but they can have constants.

Interfaces Analogy

All devices that can be charged via USB must have a USB port.

Interfaces (2)

```php
<?php
interface PaymentProvider
{
    public function showPaymentPage();
    public function contactGateway(array $messageParameters);
    public function notify(string $email);
}
class CreditCard implements PaymentProvider
{
    public function showPaymentPage()
    {
        // implementation
    }
    public function contactGateway(array $messageParameters)
    {
        // implementation
    }
    public function notify(string $email)
    {
        // implementation
    }
}
```

Magic (__*) Methods

- PHP treats any method with a name prefixed by two underscores __ as magical. PHP calls these methods “magically” (without you needing to call them) at certain times of the object’s lifecycle.

- There are 15 predefined magical functions and it is recommended to avoid naming other functions with the double underscore prefix.

__get and __set

These magic methods are called when PHP tries to read (get) or write (set) inaccessible properties.

```php
<?php
class BankBalance
{
   private $balance;
   public function __get($propertyName)
   {
       return "No value";
   }
   public function __set($propertyName, $value)
   {
       echo "Cannot set $propertyName to $value";
   }
}
$myAccount = new BankBalance();
$myAccount->balance = 100;
// Cannot set balance to 100No value
echo $myAccount->nonExistingProperty;
```

__call and __callStatic

These magic methods are called if you try to call a non-existing method on an object. The only difference is that __callStatic() responds to static calls while __call() responds to non-static calls.

```php
<?php
class Maharah
{
   public function __call($method, $arguments)
   {
       echo __CLASS__ . ' has no ' . $method . ' method';
   }
}
$jacob = new Maharah();
$jacob->honesty();  // Maharah has no honesty method
```

__invoke

The magic method __invoke() is called when you try to execute an object as a function.

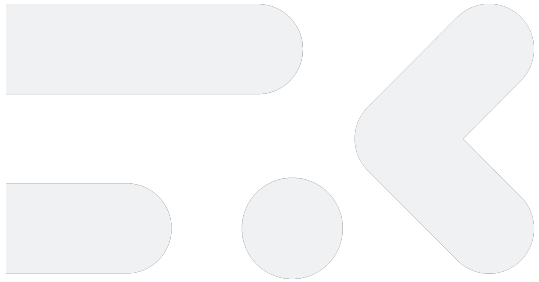
```php
<?php
class Square
{
   public function __invoke($var)
   {
       return $var ** 2;
   }
}
$callableObject = new Square;
echo $callableObject(10); // 100
```

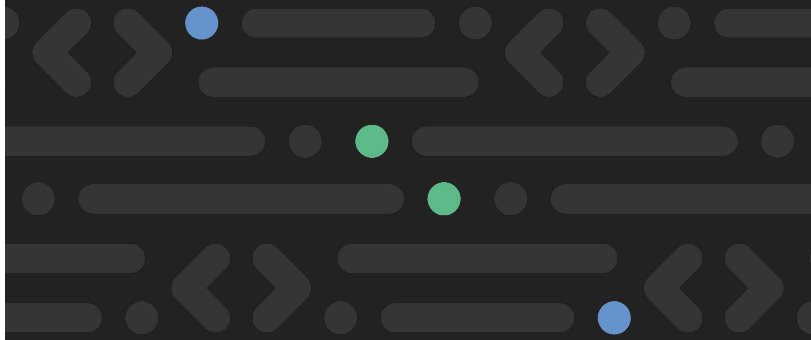
Maharah
Code Your Future

PHP Magic Methods Reference

Visit the official PHP documentation for more information.

https://www.php.net/manual/en/language.oop5.magic.php





Maharah
Code Your Future

Namespaces

Namespaces are designed to solve two problems that authors of libraries and applications encounter when creating re-usable code elements such as classes or functions:

- Name collisions between code you create, and internal PHP classes/functions/constants or third-party classes/functions/constants.
- Ability to alias (or shorten) Extra_Long_Names designed to alleviate the first problem, improving readability of source code.

Declaring a Namespace:

1- Unbracketed namespace

Namespace is declared using the **namespace** keyword, followed by the desired namespace name. This declaration is typically placed at the beginning of a PHP file.

```php
<?php

namespace MaharahNamespace;
// ...
```

Note: Namespace declaration statement has to be the very first statement or after any declare call in the script

Maharah
Code Your Future

Declaring a Namespace:
2- Bracketed namespace

Namespace is declared using the **namespace** keyword, followed by the desired namespace name. This declaration is typically placed at the beginning of a PHP file.

```php
<?php
namespace MyNamespace {
   class MyClass
   { /* ... */
   }
   function myFunction()
   { /* ... */
   }
   const MY_CONSTANT = 'value';
}
```

Note: Cannot mix bracketed namespace declarations with unbracketed namespace declarations, However, you have the flexibility to use bracketed namespaces within the same file.

Using Namespaces

- Importing with **use** keyword: To use classes, functions, or constants from a namespace without having to type the fully qualified name every time, the use keyword is employed.
- Fully qualified vs. relative usage: Developers can use the fully qualified name or employ relative names within a namespace, depending on the context.
- Namespace aliases: For brevity, aliases can be defined using the **as** keyword.

```php
<?php

use MyNamespace\MyClass as AliasClass;
```

Traits

Traits are designed to alleviate some of the limitations of a single inheritance language.

We use the **trait** keyword to declare a trait; to include it in a class, we employ the **use** keyword. A class may use multiple traits.

A trait contains a set of methods and properties just like a class, but cannot be instantiated by itself. Instead, the trait is included into a class and the class can then use its methods and properties as if they were declared in the class itself. In other words, traits are flattened into a class and it doesn't matter if a method is defined in the trait or in the class that uses the trait. You could copy and paste the code from the trait into the class and it would be used in the same manner. The code that is included in a trait is intended to encapsulate reusable properties and methods that can be applied to multiple classes.

Traits: Usage

- A class uses a trait by declaring **use** TraitName; within the class.
- Multiple traits can be used in a single class, offering flexibility.

```php
<?php
trait Logger
{
    public function log($message)
    {
        echo "Logging: $message\n";
    }
}
// Class User
class User
{
    use Logger;
    public function logActivity()
    {
        $this->log("User activity recorded");
    }
}
```

Maharah
Code Your Future

```
$qa = new QA();
$qa->start();
```

This project is funded by the European Union

Maharah
Code Your Future

LibyanSpider