Magic methods in PHP are special methods that start with double underscores (\_\_) and allow developers to define how objects behave in certain situations, such as accessing properties, calling methods dynamically, or serializing objects. These methods are part of PHP's **object-oriented programming (OOP)** features.

**Common Magic Methods in PHP**

Here are some of the most commonly used magic methods:

1. **\_\_construct()** – Called when an object is created.

class User {

public function \_\_construct($name) {

echo "User $name created!";

}

}

$user = new User("John"); // Output: User John created!

1. **\_\_destruct()** – Called when an object is destroyed or script execution ends.

class Test {

public function \_\_destruct() {

echo "Object destroyed!";

}

}

$obj = new Test();

unset($obj); // Output: Object destroyed!

1. **\_\_get($name)** – Called when trying to access a non-existent or private property.

class Demo {

private $data = "Hidden Value";

public function \_\_get($name) {

return "Property '$name' does not exist or is private!";

}

}

$obj = new Demo();

echo $obj->data; // Output: Property 'data' does not exist or is private!

1. **\_\_set($name, $value)** – Called when trying to set a value to a non-existent or private property.

class Demo {

private $data = [];

public function \_\_set($name, $value) {

$this->data[$name] = $value;

echo "Property '$name' set to '$value'!";

}

}

$obj = new Demo();

$obj->age = 25; // Output: Property 'age' set to '25'!

1. **\_\_isset($name)** – Called when using isset() or empty() on a non-existent or private property.

php

CopyEdit

class Demo {

private $data = "Hidden";

public function \_\_isset($name) {

return isset($this->data);

}

}

$obj = new Demo();

var\_dump(isset($obj->data)); // Output: bool(true)

1. **\_\_unset($name)** – Called when unset() is used on a non-existent or private property.

php

CopyEdit

class Demo {

private $data = "Hidden";

public function \_\_unset($name) {

echo "Cannot unset property '$name'!";

}

}

$obj = new Demo();

unset($obj->data); // Output: Cannot unset property 'data'!

1. **\_\_call($name, $arguments)** – Called when invoking a non-existent or private method.

php

CopyEdit

class Demo {

public function \_\_call($name, $arguments) {

echo "Method '$name' does not exist!";

}

}

$obj = new Demo();

$obj->hello(); // Output: Method 'hello' does not exist!

1. **\_\_callStatic($name, $arguments)** – Called when invoking a non-existent or private static method.

php

CopyEdit

class Demo {

public static function \_\_callStatic($name, $arguments) {

echo "Static method '$name' does not exist!";

}

}

Demo::test(); // Output: Static method 'test' does not exist!

1. **\_\_toString()** – Defines how an object behaves when treated as a string.

php

CopyEdit

class Demo {

public function \_\_toString() {

return "This is a Demo object!";

}

}

$obj = new Demo();

echo $obj; // Output: This is a Demo object!

1. **\_\_invoke()** – Allows an object to be used as a function.

php

CopyEdit

class Demo {

public function \_\_invoke($message) {

return "You called the object with message: $message";

}

}

$obj = new Demo();

echo $obj("Hello!"); // Output: You called the object with message: Hello!

1. **\_\_clone()** – Called when cloning an object using clone.

php

CopyEdit

class Demo {

public $name;

public function \_\_clone() {

$this->name = "Cloned Object";

}

}

$obj1 = new Demo();

$obj2 = clone $obj1;

echo $obj2->name; // Output: Cloned Object

1. **\_\_sleep() and \_\_wakeup()** – Used for serialization and deserialization.

php

CopyEdit

class Demo {

private $data;

public function \_\_sleep() {

return ['data'];

}

public function \_\_wakeup() {

echo "Object unserialized!";

}

}

$obj = new Demo();

$serialized = serialize($obj);

unserialize($serialized); // Output: Object unserialized!

1. **\_\_serialize() and \_\_unserialize()** – Introduced in PHP 7.4 as improved versions of \_\_sleep() and \_\_wakeup().

php

CopyEdit

class Demo {

private $data = "Secret";

public function \_\_serialize() {

return ['data' => $this->data];

}

public function \_\_unserialize($data) {

$this->data = $data['data'];

}

}

$obj = new Demo();

$serialized = serialize($obj);

$unserialized = unserialize($serialized);

**Conclusion**

Magic methods in PHP help developers control object behavior dynamically. They allow for advanced OOP techniques like overloading, serialization, and function invocation on objects. However, they should be used carefully to maintain code readability and avoid unexpected behaviors.

## **Interface in PHP**

An **interface** in PHP defines a **contract** for classes, specifying which methods they must implement. It is similar to an abstract class but **only contains method declarations (no properties or method implementations).**

## **Declaring an Interface**

* Use the interface keyword.
* Methods **must be public** (default).
* No method body (only signatures).
* A class must use implements to follow the interface.

### ****Example: Basic Interface****

interface Animal {

public function makeSound();

}

class Dog implements Animal {

public function makeSound() {

echo "Woof!";

}

}

class Cat implements Animal {

public function makeSound() {

echo "Meow!";

}

}

// Using the interface

$dog = new Dog();

$dog->makeSound(); // Output: Woof!

$cat = new Cat();

$cat->makeSound(); // Output: Meow!

## **Why Use Interfaces?**

✔ **Encapsulation** – Defines a contract without dictating how it's implemented.  
✔ **Code Consistency** – Ensures all implementing classes have the required methods.  
✔ **Multiple Implementations** – Different classes can have different behaviors while following the same structure.  
✔ **Supports Multiple Inheritance** – A class can implement multiple interfaces.

## **Multiple Interfaces in PHP**

Unlike classes (which allow only **single inheritance**), PHP **supports multiple interfaces**.

interface CanFly {

public function fly();

}

interface CanSwim {

public function swim();

}

class Duck implements CanFly, CanSwim {

public function fly() {

echo "Flying high!";

}

public function swim() {

echo "Swimming in the lake!";

}

}

$duck = new Duck();

$duck->fly(); // Output: Flying high!

$duck->swim(); // Output: Swimming in the lake!

👉 The Duck class **implements two interfaces**, making it both a flyer and a swimmer.

## **Interfaces vs Abstract Classes**

| **Feature** | **Interface** | **Abstract Class** |
| --- | --- | --- |
| Methods | Only declarations (no body) | Can have both declared and defined methods |
| Properties | ❌ Not allowed | ✅ Allowed |
| Multiple Inheritance | ✅ Can implement multiple interfaces | ❌ Only one abstract class can be extended |
| Use Case | When multiple unrelated classes need the same method signatures | When classes share behavior but need modifications |

## **Real-World Example: Database Connection Interface**

interface Database {

public function connect();

}

class MySQLDatabase implements Database {

public function connect() {

echo "Connected to MySQL";

}

}

class PostgreSQLDatabase implements Database {

public function connect() {

echo "Connected to PostgreSQL";

}

}

// Using the interface

$db = new MySQLDatabase();

$db->connect(); // Output: Connected to MySQL

👉 This ensures all database classes implement a connect() method.

## **Key Takeaways**

✅ **Interfaces define required methods but not implementations.**  
✅ **Classes implement interfaces using implements.**  
✅ **A class can implement multiple interfaces.**  
✅ **Good for designing reusable and scalable applications.**

\_\_METHOD\_\_ is a **magic constant** in PHP that returns the fully qualified name of the current method, including the class name (if applicable). It is useful for debugging, logging, and tracing method execution.

**Usage of \_\_METHOD\_\_**

* It returns the method's name along with the class name (if inside a class).
* If used inside a function (not a class method), it returns just the function name.

**Example 1: Using \_\_METHOD\_\_ in a Class**

class Test {

public function exampleMethod() {

echo \_\_METHOD\_\_;

}

}

$obj = new Test();

$obj->exampleMethod(); // Output: Test::exampleMethod

**Explanation:**

* Since exampleMethod() is inside the Test class, \_\_METHOD\_\_ returns "Test::exampleMethod".

**Example 2: Using \_\_METHOD\_\_ in a Function**

function myFunction() {

echo \_\_METHOD\_\_;

}

myFunction(); // Output: myFunction

**Explanation:**

* Since this is a standalone function, \_\_METHOD\_\_ simply returns "myFunction".

**Example 3: Using \_\_METHOD\_\_ in a Static Method**

class Demo {

public static function staticMethod() {

echo \_\_METHOD\_\_;

}

}

Demo::staticMethod(); // Output: Demo::staticMethod

**Explanation:**

* It returns "Demo::staticMethod" because it includes the class name and method.

**Use Cases of \_\_METHOD\_\_**

1. **Debugging:** Helps track where a method is being executed.
2. **Logging:** Useful for logging function calls without hardcoding method names.
3. **Error Handling:** When throwing exceptions, you can include \_\_METHOD\_\_ to indicate the source of the error.

Example:

class Logger {

public function logMessage() {

error\_log("Method executed: " . \_\_METHOD\_\_);

}

}

$obj = new Logger();

$obj->logMessage(); // Logs: Method executed: Logger::logMessage