**Magic Methods**

Magic Methodsهي مجموعة من الأساليب الخاصة التي تبدأ بـ اسمين متسلسلين (Double underscores) **\_\_** وتستخدم لتعريف سلوك مخصص للعمليات المحددة في اللغة. هذه الأساليب تسمح لك بتخصيص سلوك الكائنات والتعامل مع العديد من الأحداث المختلفة أثناء تشغيل البرنامج.

\_\_constructتُستدعى هذه الأساليب تلقائيًا عند إنشاء كائن جديد من الفئة، وتُستخدم عادة لتنفيذ المهام اللازمة للبدء.

\_\_destructتُستدعى هذه الأساليب تلقائيًا عندما يتم تدمير كائن، وتُستخدم عادة لإتمام المهام النهائية مثل إغلاق الاتصالات أو الافراج عن الموارد.

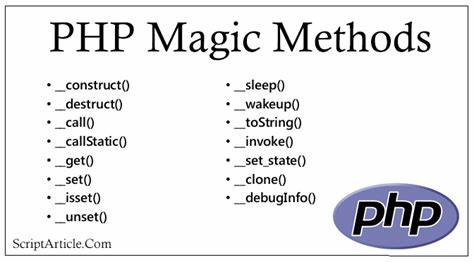
\_\_toStringتُستدعى هذه الأساليب عند محاولة تحويل الكائن إلى سلسلة نصية.

\_\_get و \_\_set تُستدعى هذه الأساليب عندما يتم الوصول إلى خاصية غير موجودة أو تغييرها على التوالي.

\_\_call و \_\_callStatic تُستخدم للتعامل مع استدعاء الأساليب التي غير معرفة.

\_\_isset و \_\_unset تُستخدم لفحص ما إذا كانت خاصية موجودة أو إزالتها بشكل مباشر.

\_\_clone: تُستدعى هذه الأساليب عند استنساخ الكائن



**Difference Between abstract and interface**

Abstract Class

الكلاس المجرد هو كلاس لا يمكن إنشاؤه بمفرده ، بل يستخدم كقالب لكلاسات أخرى للتوريث منه.

يمكن أن يحتوي على طرق مجردة (مطالب بالتطبيق بدون جسم) وطرق ملموسة (تحتوي على تنفيذ).

يمكن أن يحتوي الكلاس المجرد على خصائص.

يمكن للكلاس أن يرث من كلاس مجرد واحد فقط باستخدام الكلمة المفتاحية extends.

يتم استخدام الكلاسات المجردة عندما تريد توفير تنفيذ قاعدة مشتركة مع إمكانية للفصول الفرعية للكلاسات لتجاوز بعض الطرق.

abstract class Animal {

protected $name;

abstract public function makeSound();

public function setName($name) {

$this->name = $name;

}

}

Interface:

الواجهة هي عقد يحدد مجموعة من الطرق التي يجب أن ينفذها الكلاس.

لا يمكن أن تحتوي على خصائص.

يمكن للكلاس تنفيذ العديد من الواجهات باستخدام الكلمة المفتاحية implements.

تستخدم الواجهات عندما تريد تعريف عقد يمكن للكلاسات المتعددة تنفيذه ، لتعزيز إعادة استخدام الكود والتعددية.

interface Logger {

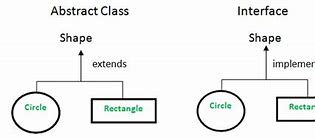
public function log($message);

}

يُستخدم لتعريف عقد (contract)، أو واجهة عامة، تحدد الخدمات التي يجب توفيرها من قِبل الكائنات التي تنفذ هذه الواجهة.

يُستخدم لتحقيق التعددية (polymorphism) وتوفير أساليب مشتركة بين كائنات مختلفة.

باختصار، Abstract يمكن أن يحتوي على تنفيذ وأساليب مجردة، بينما Interface تحتوي فقط على أساليب مجردة ولا يمكنها أن تحتوي على أي تنفيذ.



**Polymorphism**

(Polymorphism) في البرمجة القدرة على استخدام الكائنات التي تنتمي إلى صنف معين بطرق متعددة. بمعنى آخر، يمكن للكائنات من نوع واحد أن تتصرف بطرق مختلفة استنادًا إلى السياق الذي تم استخدامه فيه.

على سبيل المثال، لنفترض أن لدينا صنف اسمه "شخص"، ويحتوي هذا الصنف على طريقتين مختلفتين للطباعة: "طباعة\_معلومات()" و "طباعة\_تحية()". يمكن لكائنات "شخص" المختلفة مثل "طالب"، "مدرس"، و "موظف" أن تحتوي على تعريفات مختلفة لهذه الطرق.

عندما نستخدم التعددية، يمكننا استخدام كائن من نوع "شخص" واستدعاء طريقة "طباعة\_معلومات()"، وسيتم تنفيذ تعريف الطباعة الخاص بكائن معين مثل "طالب" إذا كان الكائن المستخدم من هذا النوع. وبالمثل، إذا استدعينا "طباعة\_تحية()"، سيتم تنفيذ تعريف الطباعة الخاص بالكائن المستخدم.

هذا النوع من التعددية يسمح لنا بتجريبية البرمجة (Polymorphic Programming)، حيث يمكننا استخدام نفس الكود لمعالجة أنواع مختلفة من الكائنات دون الحاجة إلى معرفة النوع الفعلي للكائن في وقت التنفيذ.

A diagram of animals with words

Description automatically generated

**What is instanceof keyword in PHP**

في لغة PHP، كلمة الرئيسية "instanceof" تستخدم لفحص ما إذا كانت متغير معين يشير إلى نموذج معين من مثيل محدد من الفئة. فمثلا، يمكنك استخدامها للتحقق ما إذا كان كائن معين هو مثيل من فئة معينة قبل تنفيذ أوامر معينة.

بالتأكيد، إليك بعض الأمثلة على كيفية استخدام كلمة الرئيسية "instanceof" في PHP:

php

class Animal {

public function speak() {

echo "Animal speaks";

}

}

class Dog extends Animal {

public function bark() {

echo "Woof!";

}

}

$animal = new Animal();

$dog = new Dog();

// التحقق ما إذا كان الكائن $dog يمثل نموذجاً من الفئة Dog

if ($dog instanceof Dog) {

echo "This is a Dog object.";

} else {

echo "This is not a Dog object.";

}

// التحقق ما إذا كان الكائن $animal يمثل نموذجاً من الفئة Dog

if ($animal instanceof Dog) {

echo "This is a Dog object.";

} else {

echo "This is not a Dog object.";

}

الناتج:

This is a Dog object.

This is not a Dog object.

في هذا المثال، تم استخدام "instanceof" للتحقق ما إذا كان كل كائن من الكائنات $dog و $animal يمثل نموذجاً من الفئة Dog.

**can we define a constructor as a private constructor? if it yes when could we do this?**

نعم، يُمكن تعريف البناء كبناء خاص. يتم ذلك عادة عندما نريد ضمان أن الكائنات لا يمكن إنشاؤها مباشرة من خارج الفئة. على سبيل المثال، قد نريد تحديد نقطة الوصول لإنشاء كائن جديد والتحقق من شروط معينة قبل السماح بذلك.

باستخدام البناء الخاص، يمكننا تحقيق هذا الهدف. البناء الخاص يمنع الكود الخارجي من استدعاء البناء مباشرة، مما يجعله فقط متاحًا داخل الفئة نفسها. هذا يعني أنه يمكننا استخدام البناء الخاص لضمان أنه لا يمكن إنشاء كائنات من الفئة دون اجتياز مراقبة أو شروط معينة.

مثلا، إذا كنت تريد تنفيذ نمط Singleton حيث يمكن إنشاء كائن واحد فقط من الفئة، يمكنك استخدام بناء خاص لتحقيق ذلك، حيث يتحقق البناء من وجود كائن مسبق وإرجاعه إذا كان موجودًا، أو إنشاء وإرجاع كائن جديد إذا لم يكن هناك كائن مسبق.

**constructor promotion**

تعتمد فكرة "ترقية البناء" في برمجة الكائنات (OOP) على الفكرة البسيطة لتمرير المعلمات من الفئة المشتقة (الفرعية) إلى الفئة الأساسية (الأب). وهذا يسهل إعادة استخدام الشيفرة ويقلل من تكرار الكود.

في أوب، تقوم الفئة الأساسية بتحديد البناء (المفاهيم المحددة لبدء إنشاء الكائن)، وعادةً ما تحتاج إلى معلمات لضبط حالتها الداخلية. عندما يتم تمديد هذه الفئة لإنشاء فئة فرعية جديدة، يمكن أن تحتوي الفئة الفرعية على خصائص إضافية تتطلب معلمات جديدة للبناء. تتيح ترقية البناء للفئة الفرعية تمرير هذه المعلمات إلى البناء الخاص بالفئة الأساسية، بحيث يمكن تهيئة كلتا الفئتين في وقت واحد.

في PHP، كما رأينا في المثال السابق، يمكن استخدام `parent::\_\_construct()` داخل البناء للفئة الفرعية لاستدعاء البناء المحدد للفئة الأساسية وتمرير المعلمات إليه. هذا يعني أنه يمكن للفئة الفرعية أن ترقي بنائها لتضمين معلمات الفئة الأساسية، وبالتالي تحقيق تعديلات فعالة وتوسيع الوظائف في الشيفرة.

**does php has overloading & how to reach out if exists?**

في PHP، لا يوجد دعم مباشر للتحميل (overloading) في الطريقة التي يحدث بها في بعض اللغات الأخرى مثل Java. في التحميل، يمكنك تعريف عدة دوال بنفس الاسم لكن مع توقيعات مختلفة (أعداد وأنواع المعلمات المختلفة). وعند استدعاء الدالة، يتم تحديد أي دالة يجب استدعاءها استنادًا إلى توقيع المعلمات.

ومع ذلك، يمكنك تحقيق سلوك مشابه باستخدام قيم افتراضية للمعلمات واستخدام دوال متعددة لتنفيذ السلوك المختلف بناءً على القيم التي تم تمريرها. على سبيل المثال:

class MyClass {

public function myFunction($param1, $param2 = null) {

if ($param2 === null) {

// إجراءات للحالة عندما يتم تمرير معلمة واحدة فقط

} else {

// إجراءات للحالة عندما يتم تمرير معلمتين

}

}

}

$obj = new MyClass();

$obj->myFunction(5); // يتم استدعاء الدالة مع قيمة واحدة للمعلمة الثانية

$obj->myFunction(10, 20); // يتم استدعاء الدالة مع معلمتين

في هذا المثال، تم تعريف دالة `myFunction` مع معلمة `$param1` ومعلمة افتراضية `$param2` التي تأخذ قيمة افتراضية `null`. يتم فحص قيمة `$param2` داخل الدالة لتحديد السلوك المناسب بناءً على عدد المعلمات التي تم تمريرها.

هكذا، بالرغم من عدم وجود تحميل مباشر في PHP، إلا أنه يمكنك تحقيق سلوك مشابه باستخدام قيم افتراضية للمعلمات وفحص عدد ونوع المعلمات الممررة.

**does php has multiple constructor or not and how to reach out**

في PHP، لا يوجد دعم مباشر لعدة بناء (constructors) كما هو الحال في بعض لغات البرمجة الأخرى. ومع ذلك، يمكنك تحقيق وظيفة مشابهة من خلال التحميل (overloading) باستخدام الأساليب السحرية (magic methods)، وتحديدًا الطريقة `\_\_construct()`، جنبًا إلى جنب مع الشروط المنطقية داخل البناء.

إليك مثال بسيط على كيفية تحقيق بناء متعدد في PHP:

class MyClass {

public function \_\_construct() {

$args = func\_get\_args();

$numArgs = func\_num\_args();

if ($numArgs == 1) {

// بناء يأخذ معلمة واحدة

$this->constructWithOneArg($args[0]);

} elseif ($numArgs == 2) {

// بناء يأخذ معلمتين

$this->constructWithTwoArgs($args[0], $args[1]);

} else {

// البناء الافتراضي أو التعامل مع حالات أخرى

}

}

private function constructWithOneArg($arg1) {

// معالجة البناء الذي يأخذ معلمة واحدة

}

private function constructWithTwoArgs($arg1, $arg2) {

// معالجة البناء الذي يأخذ معلمتين

}

}

$obj1 = new MyClass(); // البناء الافتراضي أو التعامل مع حالات أخرى

$obj2 = new MyClass("القيمة1"); // بناء يأخذ معلمة واحدة

$obj3 = new MyClass("القيمة1", "القيمة2"); // بناء يأخذ معلمتين

في هذا المثال:

- تُستخدم الطريقة `\_\_construct()` كبناء للفئة.

- داخل البناء، تُستخدم الدوال `func\_get\_args()` لاسترجاع جميع المعلمات الممررة، و`func\_num\_args()` للحصول على عدد المعلمات التي تم تمريرها إلى البناء.

- بناءً على عدد المعلمات، يقوم البناء بتوجيه السياق إلى الدوال الخاصة المناسبة (`constructWithOneArg()` و `constructWithTwoArgs()` في هذا الحال) لمعالجة السيناريوهات المختلفة للبناء.

على الرغم من عدم وجود دعم مدمج لعدة بناء في PHP، إلا أن هذا النهج يسمح لك بتحقيق وظيفة مشابهة عن طريق استخدام الشروط المنطقية لتحديد السلوك المناسب بناءً على المعلمات التي يتم تمريرها إلى البناء.

**differences between interface abstract class**

الاختلافات بين الواجهة (Interface)، (Abstract Class) تتعلق بالطبيعة والاستخدام والقواعد التي تفرضها كل منها. إليك شرحًا موجزًا لكل منها باللغة العربية:

1. الواجهة (Interface):

- الواجهة هي عبارة عن تعاقد بين الكائنات. تعرف الواجهة المعالم (المتغيرات) والسلوكيات (الدوال) التي يجب تنفيذها من قبل أي فئة تقوم بتنفيذها.

- لا يمكن تنفيذ الواجهة مباشرة، بل يجب على الفئات تنفيذها باستخدام الكلمة المفتاحية `implements`.

- يسمح باستخدام الواجهات متعددة في نفس الفئة.

- الواجهة لا تحتوي على أي تنفيذ للدوال؛ فقط تعريفات دوال بدون أجسام داخلية.

2. الفصل الطبقي (Abstract Class):

- الفصل الطبقي هو نوع من الفئات لا يمكن إنشاء كائنات منها مباشرة، ويتضمن دوالًا مجردة (abstract methods) التي يجب تنفيذها في الفئات المشتقة.

- يمكن أن تحتوي الفصول الطبقية على دوال مجردة ودوال عادية مع تنفيذ غير مجرد.

- يمكن استخدام الفصول الطبقية لتوفير تنفيذات جزئية لدوال، بحيث يمكن للفئات المشتقة تجاوز هذه التنفيذات حسب الحاجة.

3. \*\*الاختلافات:\*\*

- \*\*التنفيذ:\*\* يمكن للفئات تنفيذ عدة واجهات، بينما يمكن للفئة أن تمتد من فصيل طبقي واحد فقط.

- \*\*التعداد:\*\* يمكن استخدام الواجهات متعددة في نفس الفئة، بينما لا يمكن توريث أكثر من فصل طبقي واحد في الوقت نفسه.

- \*\*التنفيذات:\*\* الواجهات لا تحتوي على أي تنفيذات، بينما يمكن أن تحتوي الفصول الطبقية على تنفيذات جزئية.

- \*\*الأغراض:\*\* تستخدم الواجهات عادة للتعاقد بين الكائنات، بينما تستخدم الفصول الطبقية لتوفير بنيات أساسية للفئات المشتقة.

- \*\*التطبيق:\*\* يستخدم الواجهات عندما يكون التنفيذ مهمًا أكثر من التعداد، بينما يستخدم الفصل الطبقي عندما يكون التعداد وتوفير التنفيذات الجزئية مهمًا.

A diagram of a computer program

Description automatically generated with medium confidence