Sự ra đời của lập trình hướng đối tượng đại diện cho một sự thay đổi mô lớn trong chiến lược phát triển, tái tập trung sự chú ý vào dữ liệu của một ứng dụng hơn là logic của nó. Nói một cách khác, OOP dịch chuyển tập trung từ các sự kiện về thủ tục của chương trình đối với các đối tượng thực tế cuộc sống nó được dự định để mô hình. Kết quả là một ứng dụng gần giống với thế giới xung quanh chúng ta. Phần này xem xét ba khái niệm cơ bản của OOP: đóng gói, thừa kế, đa hình. Cùng với nhau, ba ý tưởng hình thành cơ sở cho mô hình lập trình mạnh mẽ nhất chưa nghĩ ra.

**Tính đóng gói**

Tính đóng gói là tính chất không cho phép người dùng hay đối tượng khác thay đổi dữ liệu thành viên của đối tượng nội tại. Chỉ có các hàm thành viên của đối tượng đó mới có quyền thay đổi trạng thái nội tại của nó mà thôi. Các đối tượng khác muốn thay đổi thuộc tính thành viên của đối tượng nội tại, thì chúng cần truyền thông điệp cho đối tượng, và việc quyết định thay đổi hay không vẫn do đối tượng nội tại quyết định.

Ta có thể hiểu nôm la tính đóng gói là không cho bên ngoài biết được bên trong đối tượng có những gì hay được cài đặt như thế nào. Nếu muốn thay đổi bên trong đối tượng thì phải được sự chấp nhận của đối tượng đó thông qua ba mức độ truy cập private protected và public mà chúng ta đã học ở bài trước.

Ví dụ trong đối tượng kế toán có một chức năng tính lương, chức năng này có những phép tính mà các đối tượng khác không hề biết cách tính nó như thế nào. Đối tượng kế toán sẽ có những chức năng dành cho các hàm kế thừa nó có thể truy xuất vào và thông qua ba mức độ truy cập để giới hạn chúng. Đây cũng chính là bảo mật thông tin cho đối tượng.

**Tính kế thừa**

Tính kế thừa là một trong ba tính chất quan trọng của [lập trình hướng đối tượng](http://freetuts.net/lap-trinh-huong-doi-tuong-la-gi-27.html), nó đòi hỏi sự logic về phân tích các đối tượng trong phần mềm để từ đó đưa ra những mô hình design pattern giúp việc nâng cấp, bảo trì phần mềm dễ dàng hơn. Đây là tính chất được coi là quan trọng nhất bởi hầu hết các Framework trong PHP đều có sử dụng tính kế thừa, vì vậy nếu bạn đã xác định theo con đường lập trình PHP thì ắt phải hiểu về kế thừa.

Nội dung bài học như sau:

**2. Các mức độ truy cập trong kế thừa lồng**

Như ta đã biết có [3 mức độ truy cập](http://freetuts.net/cac-muc-truy-cap-private-protected-va-public-31.html) hay dùng nhất đó là **private, public và protected**. Cũng như trong kế thừa một cấp, tất cả những biến (thuộc tính) và hàm (phương thức) ở dạng public thì ở tất cả các lớp con dù ở cấp độ nào nó cũng có thể truy xuất vào được. Nếu ở dạng **private**thì chỉ dùng trong lớp đó. Nếu ở dạng **protected**thì tất cả các lớp kế thừa nó dù là lồng bao nhiêu lần thì cũng có thể sử dụng. Định nghĩa này chúng ta đã được học ở bài trước và cũng được giữ nguyên cho bài **kế thừa lồng**này.

**Ví dụ:**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69 | // Lớp A  class A  {      protected $protected\_A = 'Protected';      private $private\_A = 'Private';      public $public\_A = 'Public';        private function showPrivate()      {          echo $this->private\_A;      }        protected function showProtected()      {          echo $this->protected\_A;      }        public function showPublic()      {          echo $this->public\_A;      }  }    // Lớp B Kế Thừa Lớp A  class B extends A  {      public function ClassB()      {          echo $this->protected\_A;      }  }    // Lớp C Kế Thừa Lớp B  class C extends B  {      public function showInfo()      {          // Lệnh này đúng vì nó truy xuất vào thuộc tính protected          $this->protected\_A = 'Nguyễn Văn A';            // Lệnh này đúng vì nó truy xuất vào thuộc tính public          $this->public\_A = 'Nguyễn Văn B';            // Lệnh này sai vì nó truy xuất vào thuộc tính private          $this->private\_A = 'Lệnh sai';      }  }    // ------------------//  // Chương trình chính//  // ------------------//    // Khởi tạo lớp C  // Lớp C được kế thừa từ lớp B  // mà lớp B kế thừa từ lớp A nên  // suy ra nó kế thừa từ 2 lớp A, B  $c = new C();    // Lệnh này đúng vì gọi đến hàm public của lớp cha A  $c->showPublic();    // Lệnh này sai vì nó gọi hàm protected của lớp cha A  $c->showProtected();    // Lệnh này sai vì nó gọi hàm private của lớp cha A  $c->showPrivate();    // Lệnh này đúng vì nó truy xuất vào hàm public của lớp cha B  $c->ClassB(); |

**3. Khi nào sử dụng cấp độ truy cập private**

Thông thường để an toàn dữ liệu các thuộc tính đều ở dạng private, nhưng điều này rất phiền vì ta phải tạo thêm các hàm SET và GET nên các lập trình viên cũng ít khi sử dụng private. Tuy nhiên có những trường hợp sau ta bắt buộc phải dùng ở dạng private để an toàn cho đối tượng.

* Những thuộc tính có tính biến đổi dữ liệu khi nhập và lấy dữ liệu như hai ví dụ về Username và Password ở trên.
* Những phương thức chỉ dùng trong nội bộ trong lớp đó, không có sử dụng bên ngoài lớp.

Đó là 2 đặc điểm cơ bản để bạn chọn độ truy cập là private.

class A

{

    private $\_\_username;

    private $\_\_password;

    function getUsername()

    {

        return 'Xin chào ' . $this->\_\_username;

    }

    function setUsername($username)

    {

        $this->\_\_username = $username;

    }

    function getPassword()

    {

        return $this->\_\_password;

    }

    function setPassword($password)

    {

        $this->\_\_password = md5($password.'ky\_tu\_muon\_them');

    }

}

// Sử dụng

$a = new A();

$a->setUsername('TheHalfheart');

echo $a->getUsername();

$a->setPassword('matkhau');

echo $a->getPassword();

## 4. Khi nào sử dụng cấp độ truy cập protected

Protected thường được dùng khi bạn biết chắc là có lớp khác sẽ kế thừa lớp này và những phương thức, thuộc tính đó chỉ được dùng trong lớp kế thừa nó.

Giả sử bạn khai báo lớp Động Vật, trong đó có hàm lưu dữ liệu động vật vào database, hàm này dùng chung cho tất cả các lớp con kế thừa lớp Động Vật. Và để bảo mật nên tôi không muốn ở bên ngoài lớp có thể sử dụng được, vì vậy tôi khai báo là protected.

Đó là những ví dụ cơ bản, chứ thực tế thì cũng tùy vào từng bài toán cụ thể mà bạn lựa chọn

## 5. Khi nào sử dụng cấp độ truy cập public

Public là cấp độ thoáng nhất, nó có thể gọi ở mọi nơi từ trong nội bộ của lớp đến lớp kế thừa nó, thậm chí cả bên ngoài lớp cũng gọi được.

Những hàm được khai báo public thường được dùng để lấy dữ liệu và xuất dữ liệu ra bên ngoài, hoặc là những hàm mang tính chất là hàm thao tác cuối cùng với người coder. Ví dụ như các hàm SET và GET được để ở dạng public.

Tính đa hình(Polymorphism)

Tính đa hình cho phép các chức năng (method) khác nhau được thực thi khác nhau trên các đối tượng khác nhau. Nói nôm na đơn giản hơn là bạn có 3 lớp A,B,C kế thừa nhau và trong 3 lớp này có ba method cùng tên là show(); . Khi ta tạo mới đối tượng cho các lớp trên thì những đối tượng này gọi tới method show() nằm ở lớp nào thì thực thi chức năng phương thức ở lớp đó. Bởi vì chúng kế thừa nên sử dụng rõ tính đa hình này.

Tính đa hình được dử dụng tùy thuộc vào bối cảnh cụ thể

Cụ thể ta có đa hình tĩnh và đa hình động.  
  
 1. Đa hình tĩnh là việc dùng Overloading gồm các phương thức có trùng tên trong một class. Nhưng các phương thức này được thực thi khác nhau, tùy vào cài đặt. Bạn xem lại bài trước . . . Ở đây chúng ta không đề cập tới nữa  
  
2. Đa hình động lại gồm có Abstract và Virtual. Trong đó abstract bạn có thể xem lại bài trước, chúng ta sẽ nói rõ về phương thức Virtual.  
  
Thực ra virtual cũng giống như abstract, điểm khác biệt là bạn có thể viết các tuyên bố lệnh cho nó còn abstract method thì không. Và khi sử dụng phương thức abstract thì class đó cũng là abstract class còn Virtual thì không.  
  
// Virtual method  
public virtual void abc()  
{  
// Do something here !  
}  
  
// Abstract method  
public abstract void abc();

**Abstract**

Nếu bạn đã từng học qua lập trình hướng đối tượng một ngôn ngữ bất kỳ khác như C++, Java thì abstract khá quen thuộc, và trong php đây cũng là một tính chất của lập trình hướng đối tượng nhưng nó không được xếp vào tính chất quan trọng, tính chất này giống như tính kế thừa theo tầm nhìn bề ngoài. Lớp Abstract sẽ định nghĩa các hàm (phương thức) mà từ đó các lớp con sẽ kế thừa nó và Overwrite lại (tính đa hình). Tất cả các phương thức của lớp abstract đều phải được khai báo là abstract và phải ở mức protected và public, không được ở mức private. Lớp Abstract có thể có [thuộc tính](http://freetuts.net/lop-thuoc-tinh-phuong-thuc-cua-doi-tuong-28.html)nhưng thuộc tính không được khai báo là abstract, và bạn không thể khởi tạo một biến của lớp Abstract được.

Example:

abstract class BaseClass

{

    abstract protected function hello();

}

// Sai vì BaseClass là lớp Abstract nên không

// khởi tạo mới được

$base = new BaseClass();

Mức truy cập các hàm của Abstract phải ở public hoặc protected để lớp kế thừa có thể định nghĩa lại và các thuộc tính của lớp Abstract không được khai báo Abstract. Các bạn xem ví dụ dưới đây:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | abstract class BaseClass  {      // Đúng      public $name;        // Sai vì các thuộc tính không được để ở dạng abstract      abstract public $title;        // Đúng      abstract protected function hello();        // Sai vì hàm abstract không thể ở private      abstract private function hi\_there();  } |

**Lớp kế thừa từ lớp Abstracth phải Rewrite lại tất cả các hàm Abstract trong lớp Abstract, nếu không sẽ bị báo sai. Ví** **dụ:**

|  |  |  |  |
| --- | --- | --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | abstract class Person  {      protected $ten;      protected $cmnd;      protected $namsinh;        abstract public function showInfo();  }    // Lớp này sai vì chưa viết lại hàm showInfo  class CongNhan extends Person  {    }    // Lớp này đúng vì ta đã khai báo, viết lại  // đầy đủ các hàm abstract  class SinhVien extends Person  {      public function showInfo(){        }  } 2. Hàm và lớp final Lớp Final là lớp được khai báo là lớp cuối cùng, không một lớp nào có thể kế thừa nó. Tương tự như hàm Final trong Abstract hoặc trong kế thừa chỉ để gọi sử dụng, không được viết lại (Override).  Để dễ hình dung các bạn xem ví dụ sau đây và thông qua phần ghi chú tôi đã giải thích. **Ví dụ lớp final:**  // Lớp Filnal  final class Person  {      protected $ten;      protected $cmnd;      protected $namsinh;      public function showInfo()      {          echo 'freetuts.net';      }  }    // Hàm này sẽ bị báo lỗi vì lớp SinhVien  // đã kế thừa một lớp Final, điều này là không thể  class SinhVien extends Person {  }    // Đoạn code này đúng vì lớp Final được  // sử dụng bình thường như các lớp khác  // chỉ có điều là không được kế thừa  $person = new Person;  $person->showInfo();  **Ví dụ hàm final:**   |  |  | | --- | --- | | 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | class Person  {      protected $ten;      protected $cmnd;      protected $namsinh;      final public function showInfo()      {          echo 'freetuts.net';      }  }    // Lớp này đúng vì lớp Person không phải  // là một lớp final  class SinhVien extends Person {        // Hàm này sai vì hàm showInfo      // là hàm final trong lớp Person      // nên không thể Override lại      public function showInfo(){        }        public function Go()      {          // Đoạn code này đúng vì hàm final được          // sử dụng bình thường          $this->showInfo();      }  } | |

## 1. Interface là gGì ?

Interface là một Temlate (khuôn mẫu), nó không phải là một lớp đối tượng mà chỉ là một bề nhìn bên ngoài mà nhìn vào đó ta có thể biết được tất cả các hàm của đối tượng implement nó.

Để khai báo một Interface ta dùng từ khóa **interface**để thay cho từ khóa **class**. Tất cả các hàm trong interface đểu ở dạng khai báo và **không được định nghĩa** (giống lớp abstract). Nếu một đối tượng implement một interface thì nó phải khai báo và định nghĩa tất cả các hàm trong Interface.

Thoạt nhìn qua các bạn thấy Interface rất giống với Abstract trong php đúng không nào? Câu trả lời là bản chất bên trong hoàn toàn khác nhau. Interface không phải là một lớp cụ thể mà là một khuôn mẫu để cho **một** đối tượng implement nó, và đương nhiên là ta không thể tạo một biến Interface. Ngược lại [lớp Abstract](http://freetuts.net/lop-truu-tuong-abstract-trong-php-oop-37.html) là một lớp cụ thể, có đầy đủ các tính chất của một đối tượng, có thể gọi, định nghĩa các hàm trong nó. Đối với hằng số ở lớp implement không được định nghĩa lại.  
  
**Ví dụ**: Định nghĩa hằng số bị sai

interface A

{

    const ConstA = 'Freetuts.net';

}

// Lớp này sai vì không thể định nghĩa lại hằng

class B implements A

{

    const ConstA = 'Other Name';

}

// Lớp này đúng

class D implements A

{

}

**Ví dụ**: Định nghĩa hàm trong template bị sai

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | // Khai báo một Interface  interface DogTemplate  {      // Hàm này đúng vì ta chỉ khai báo mà không có định nghĩa      public function Run();        // Hàm này sai vì ta đã định nghĩa cho nó      public function Eat(){        }  } |

**Example**

// Template

interface DogTemplate

{

    public function Run();

    public function Eat();

}

// Lớp Dog

class Dog implements DogTemplate

{

    // Hàm này sai vì cấp độ truy cập

    // của hàm run bên template là public

    // mà trong hàm này ta lại khai báo là private

    private function Run(){

    }

    // Hàm này đúng

    public function Eat(){

    }

}

**Ví dụ**: Bị thiếu hàm

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | // Template  interface DogTemplate  {      public function Run();        public function Eat();  }    // Lớp Dog  // Sai vì thiếu hàm Run  class Dog implements DogTemplate  {      public function Eat(){        }  } |

**2.Tính kế thừa interface trong PHP**

**Interface trong php**tuy không phải là một [lớp](http://freetuts.net/lop-thuoc-tinh-phuong-thuc-cua-doi-tuong-28.html)chính hiệu nhưng nó cũng có một tính chất đó là [tính kế thừa](http://freetuts.net/tinh-ke-thua-trong-php-30.html), nghĩa là một Interface A có thể kế thừa một Interface B thì lúc này đối tượng nào implement lớp A thì nó phải định nghĩa tất cả các hàm mà cả hai lớp A và B đã khai báo.  
  
**Ví dụ:**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32 | interface A {      public function funcA();  }    interface B extends A  {      public function funcB();  }    // Lớp này đúng vì nó khai báo đầy  // đủ các hàm trong A và B  class C implements B  {      public function funcA()      {        }        public function funcB()      {        }  }    // Lớp này sai vì nó khai báo mỗi hàm funcA  class D implements B  {      public function funcA()      {        }  } |

Sự khác nhau giữa self và $this trong PHP

Posted: Tháng Tám 25, 2013 in [**PHP - Immediate**](https://bienuit.wordpress.com/category/web-programming/php-programming/php-immediate/)   
Tags: [**non-static**](https://bienuit.wordpress.com/tag/non-static/), [**oop**](https://bienuit.wordpress.com/tag/oop/), [**PHP**](https://bienuit.wordpress.com/tag/php/), [**property**](https://bienuit.wordpress.com/tag/property/), [**sự khác nhau**](https://bienuit.wordpress.com/tag/su-khac-nhau/), [**self**](https://bienuit.wordpress.com/tag/self/), [**static member**](https://bienuit.wordpress.com/tag/static-member/), [**this**](https://bienuit.wordpress.com/tag/this/)

*Thông thường để truy nhập vào một thuộc tính của một đối tượng, ta thường dùng $this. Nhưng khi truy xuất vào những thuộc tính static, có thể đối tượng* đó chưa được khởi tạo nên ta không thể truy xuất qua $this được, bởi vậy ta cần dùng self để truy xuất.

Cú pháp: self::property

Tóm lại, $this sẽ trỏ tới đối tượng hiện thời còn self sẽ trỏ tới class hiện thời. Tức là sử dụng this->member cho non-static member và dùng self::$member cho static member.

Ví dụ:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | <?php  class X {      private $non\_static\_member = 1;      private static $static\_member = 2;        function \_\_construct() {          echo $this->non\_static\_member . ' '             . self::$static\_member;      }  }    new X();  ?> |

Tuy nhiên, khi tạo lập tính đa hình (polymorphism) cho đối tượng, ta không thể dùng self để thay thế cho $this, ta có thể thấy rõ điều này trong ví dụ sau:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | <?php  class X {      function foo() {          echo 'X::foo()';      }        function bar() {          $this->foo();      }  }    class Y extends X {      function foo() {          echo 'Y::foo()';      }  }    $x = new Y();  $x->bar();  ?> |

Ở ví dụ này, khi ta dùng $this, class Y được kế thừa từ class X. Kết quả là nếu đối tượng được khởi tạo là X nó sẽ xuất ra kết quả là X::foo và nếu đối tượng được khởi tạo là Y nó sẽ xuất ra kết quả là Y::foo.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | <?php  class X {      function foo() {          echo 'X::foo()';      }        function bar() {          self::foo();      }  }    class Y extends X {      function foo() {          echo 'Y::foo()';      }  }    $x = new Y();  $x->bar();  ?> |

Còn khi dùng self để thay thế cho $this trong ví dụ trên, tính đa hình sẽ bị loại bỏ hoàn toàn bởi X::foo luôn luôn được gọi.

**Static**

Trong lập trình hướng đối tượng ta có thể hiểu dữ liệu tĩnh là loại dữ liệu được sử dụng ở dạng toàn cục, dù nó được xử lý ở bất kỳ file nào trong cùng một chương trình đều được lưu lại trong lớp, ta có thể gọi chúng là thành viên tĩnh. Mỗi thành viên đều có[các mức truy cập private, public và protected](http://freetuts.net/cac-muc-truy-cap-private-protected-va-public-31.html) bình thường.

**Ví dụ 1:**chương trình ở phần mở đầu.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | // Lớp động vật  class Animal  {      protected $\_name = 'Chưa có tên';        function setName($name){          $this->\_name = $name;      }        function getName(){          return $this->\_name;      }  }    // Phần 1: Con Vịt  $con\_vit = new Animal();  $con\_vit->setName('Con Vịt');  echo $con\_vit->getName();  // Kết quả: Con Vịt    // Phần 2: Con Heo  $con\_heo = new Animal();  echo $con\_heo->getName();  // Kết quả: Chưa có tên |

Ta thấy ở Phần 1 tôi đã tạo object $con\_vit và thiết lập tên cho nó là 'Con Vịt'. Tiếp theo tôi tạo một object $con\_heo và tôi không có thiết lập tên cho nó, khi xuất tên ra màn hình thì Phần 1 xuất hiện chữ 'Con Vịt' còn Phần 2 thì xuất hiện chữ 'Chưa có tên'. Như vậy rõ ràng các thao tác trên biến $con\_vit không ảnh hưởng qua biến $con\_heo.

**Ví dụ 2:** chương trình dùng dạng thành viên tĩnh

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | // Lớp động vật  class Animal  {      protected static $\_name = 'Chưa có tên';        public static function setName($name){          Animal::$\_name = $name;      }        public static function getName(){          return Animal::$\_name;      }  }    // Phần 1: Con Vịt  $con\_vit = new Animal();  $con\_vit->setName('Con Vịt');  echo $con\_vit->getName();  // Kết quả: Con Vịt    // Phần 2: Con Heo  $con\_heo = new Animal();  echo $con\_heo->getName();  // Kết quả: Con Vịt |

Trong ví dụ này khác ở ví dụ 1 là các phương thức và thuộc tính tôi khai báo dạng tĩnh (có từ khóa static) và kết quả hoàn toàn khác với ví dụ 1. Ở cả Phần 1 và Phần 2 đều xuất ra màn hình là 'Con Vịt', lý do là tôi sử dụng dạng tĩnh và khi có thao tác thay đổi dữ liệu thì nó đều lưu vào trong class Animal nên khi khởi tạo thêm biến nó đều bị ảnh hưởng theo.

Qua hai ví dụ này có lẽ các bạn cũng đã hình dung được phần nào rồi, ta sẽ tiếp tục tìm hiểu sâu hơn nữa nhé.

## 3. Các vấn đề thông dụng khi sử dụng thành viên tĩnh

Như đã trình bày ở trên các thành viên tĩnh đều có các mực truy cập bình thường. Và để khai báo các mức truy cập thì ta sử dụng cú pháp sau:

**Cú pháp**:

* Thuộc tính: [private|public|protected] static $name, ví dụ **public static $name**
* Phương thức: [private|public|protected] static function functionname(){}, ví dụ **public static function setName()**

Bây giờ ta sẽ thảo luận đến một số vấn đề khi sử dụng thành viên tĩnh.

### Truy xuất trực tiếp không cần khởi tạo object

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | // Lớp động vật  class Animal  {      protected static $\_name = 'Chưa có tên';        public static function setName($name){          Animal::$\_name = $name;      }        public static function getName(){          return Animal::$\_name;      }  }    Animal::setName('Con Vịt');  echo Animal::getName();  // Kết quả: Con Vịt |

### Gọi các hàm tĩnh trong nội bộ của class

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | // Lớp động vật  class Animal  {      protected static $\_name = 'Chưa có tên';        public static function setName($name){          Animal::$\_name = $name;      }        public static function getName(){          return Animal::$\_name;      }        public static function all($name){          Animal::setName($name);          echo Animal::getName();      }  }    Animal::all('Con Vịt');  // Kết quả: Con Vịt |

Các bạn để ý trong ví dụ này tôi đã gọi các thành viên tĩnh nội bộ lẫn nhau trong chính đối tượng đó thông qua cú pháp hai dấu chấm (::). Tương tự khi sử dụng ở ngoài lớp ta vẫn dùng hai dấu chấm thay vì dấu mũi tên. Có một lưu ý rằng khi gọi đến thuộc tính tĩnh thì thông thường ta dùng $this->name, nhưng trong thành viên tĩnh thì khác ta dùng thêm dấu $ nữa, ví dụ Anmal::**$**name

### Không sử dụng từ khóa $this

Vì các thuộc tính và phương thức tĩnh ở dạng toàn cục, được gọi mà không cần khởi tạo nên nếu bạn dùng từ khóa $this để gọi đến một hàm nào đó trong chính lớp đó thì sẽ bị báo sai. Từ đây ta rút ra kết luận trong phương thức tĩnh chỉ gọi được những thuộc tính và phương thức cùng class ở dạng tĩnh. Ngoài cách gọi trực tiếp tên class ta có thể dùng từ khóa **self** để thay thế, ví dụ self::$\_name

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | // Lớp động vật  class Animal  {      protected $\_age = '';        protected static $\_name = 'Chưa có tên';        public static function setInfo($name, $age)      {          // Đúng          Animal::$\_name = $name;            // Sai vì $this không tồn tại          $this->\_name = $name;            // Sai vì $this không tồn tại          $this->\_age = $age;            // Sai vì thuộc tính $\_age không phải là tĩnh          Animal::$\_age = $age;      }  } |

### Kế thừa khi sử dụng static

Về [tính kế thừa](http://freetuts.net/tinh-ke-thua-trong-php-30.html) thì nó hoàn toàn bình thường không có gì đặc biệt, chỉ khác một điều là dùng hai dấu chấm (::) để truy xuất đến hàm tĩnh của lớp cha.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | // Lớp động vật  class Animal  {      protected static $\_name = '';        public static function setName($name)      {          Animal::$\_name = $name;      }        public static function getName(){          return Animal::$\_name;      }  }    class ConHeo extends Animal  {        public static function setName($name) {          parent::setName($name);      }    }    ConHeo::setName('Con Heo');  echo ConHeo::getName();  // Kết quả: COn Heo |

## 3. Lợi hại khi sử dụng thành viên tĩnh static

Việc sử dụng các hàm và thuộc tính ở dạng tĩnh có ưu điểm là ta có thể thay đổi dữ liệu toàn cục cho đối tượng đó, không cần khởi tạo đối tượng mới vẫn sử dụng được. Tuy nhiên nó có khuyết điểm là nếu ta khai báo tĩnh thì chương trình sẽ xử lý lưu trữ toàn cục nên sẽ tốn bộ nhớ hơn.

**4. khi nào dùng interface và khi nào dùng abstract**

Interface là tạo ra 1 khuôn mẫu chung, 1 interface chung để cho việc giao tiếp giữa các đối tượng trong chương trình một cách thống nhất. Có thể hiểu nó như 1 chuẩn nào đó. Khi một class implements từ nó => instance của class đó đạt chuẩn => các đối tượng của class khác có thể sử dụng các tình năng của đối tượng chuẩn đó mà không lo lắng nó thiếu function này nọ vì đã đạt chuẩn là bắt buộc phải có đủ các phương thức của chuẩn đó.

Abstract class => lớp trừu tượng => được sử dụng để implements các function của interface. Tuy nhiên do có mác trừu tượng nên nó sẽ bao gồm 1 vài function nào đó vẫn còn trừu tượng abstract function và chờ để các lớp thực tế (concrete class) triển khai cụ thể.