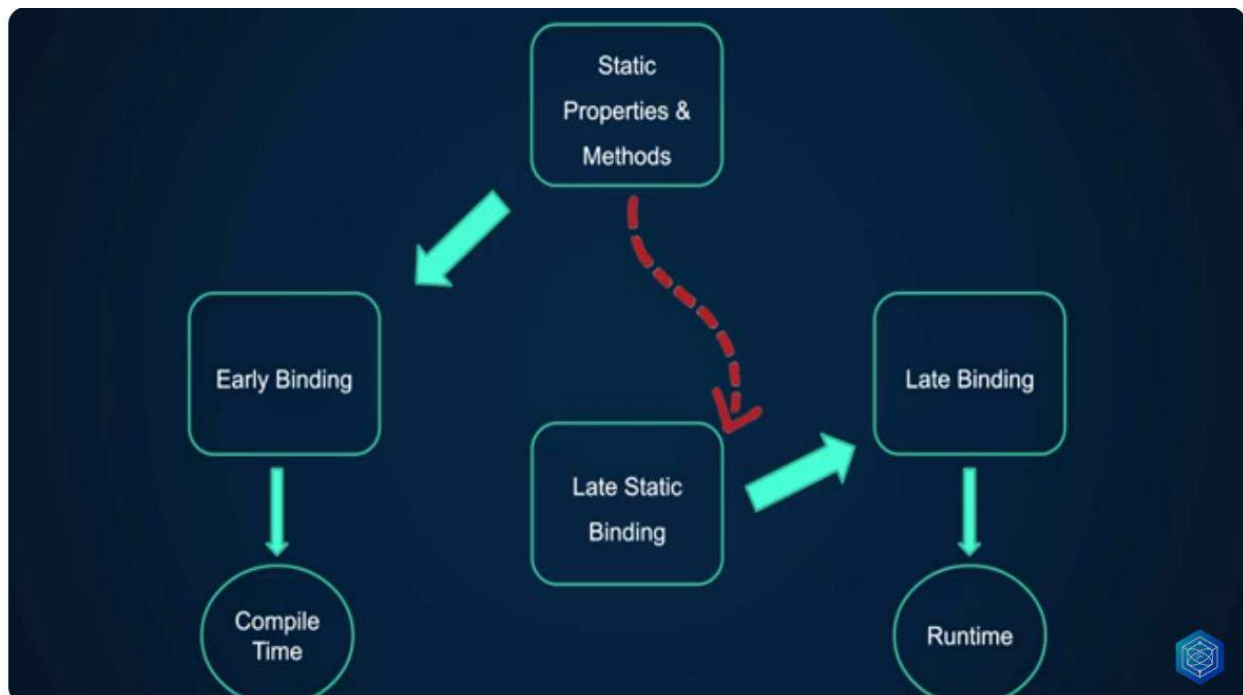


Video 2.13:



+ We covered static properties and methods in lesson 2.7, but we haven't talked about using them with inheritance. Let's get into that.

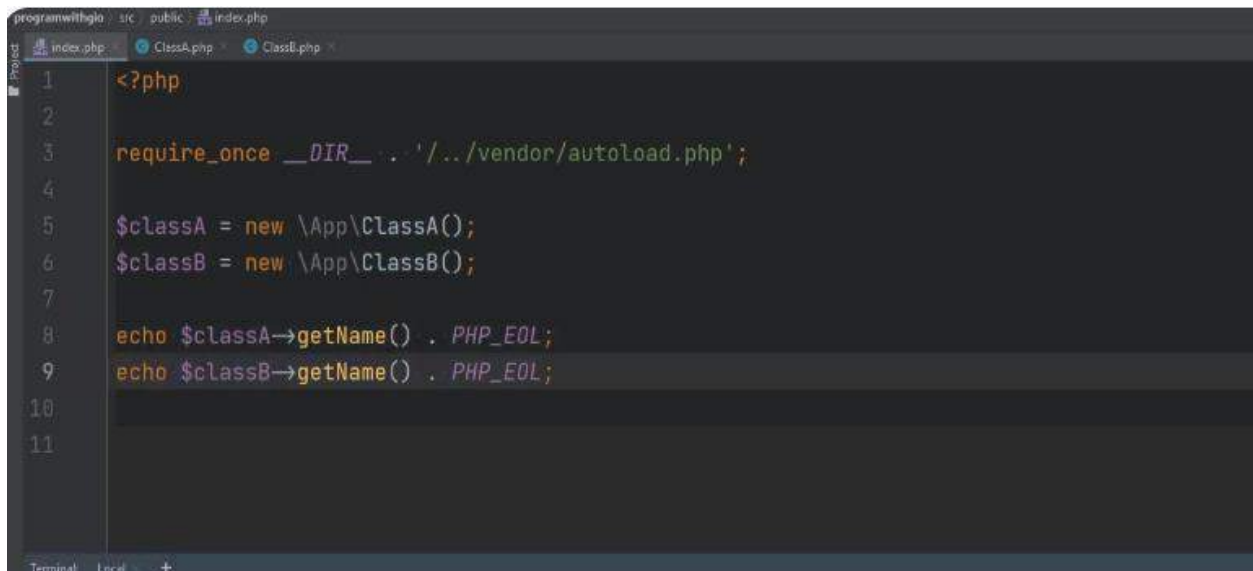
+ Chúng ta đã bàn về các thuộc tính và phương thức tĩnh trong bài học 2.7, nhưng chúng ta chưa thảo luận về cách sử dụng chúng trong kế thừa. Hãy cùng tìm hiểu về điều đó.



+ If you've heard the term late static binding, but sounded confusing or it didn't make sense, stick around and watch this lesson because I hope to make that clear by the end of this video. To better understand what late static binding is and how it works, we need to first understand what the problem is and what problem it solves. There are two types of binding: early binding, which happens at compile time, and late binding, which happens at runtime. The SPHP gets compiled on-demand and even though it is obstructed away from us, it is still there.

+ Nếu bạn đã nghe đến thuật ngữ "late static binding" (ràng buộc tĩnh muộn), nhưng thấy nó khó hiểu hoặc không rõ ràng, hãy tiếp tục theo dõi bài học này, bởi tôi hy vọng sẽ làm rõ điều đó vào cuối video này. Để hiểu rõ hơn về "late static binding" là gì và nó hoạt động như thế nào, trước hết chúng ta cần hiểu vấn đề là gì và vấn đề nó giải quyết. Có hai loại ràng buộc: ràng buộc sớm,

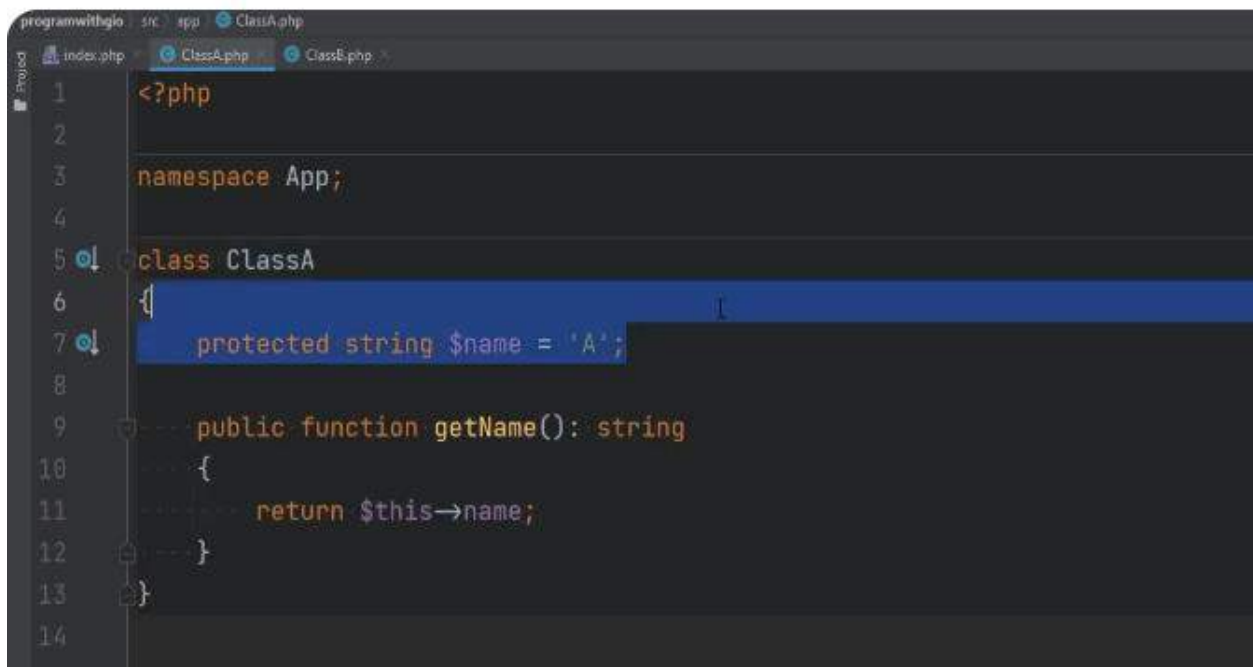
xảy ra trong quá trình biên dịch, và ràng buộc muộn, xảy ra trong thời gian chạy. SPHP được biên dịch khi có yêu cầu và mặc dù nó bị che giấu khỏi chúng ta, nhưng nó vẫn tồn tại.



```
1 <?php
2
3 require_once __DIR__ . '/../vendor/autoload.php';
4
5 $classA = new \App\ClassA();
6 $classB = new \App\ClassB();
7
8 echo $classA->getName() . PHP_EOL;
9 echo $classB->getName() . PHP_EOL;
10
11
```

+ Let's see examples so it makes more sense. I have two classes right here. I have Class A and Class B.

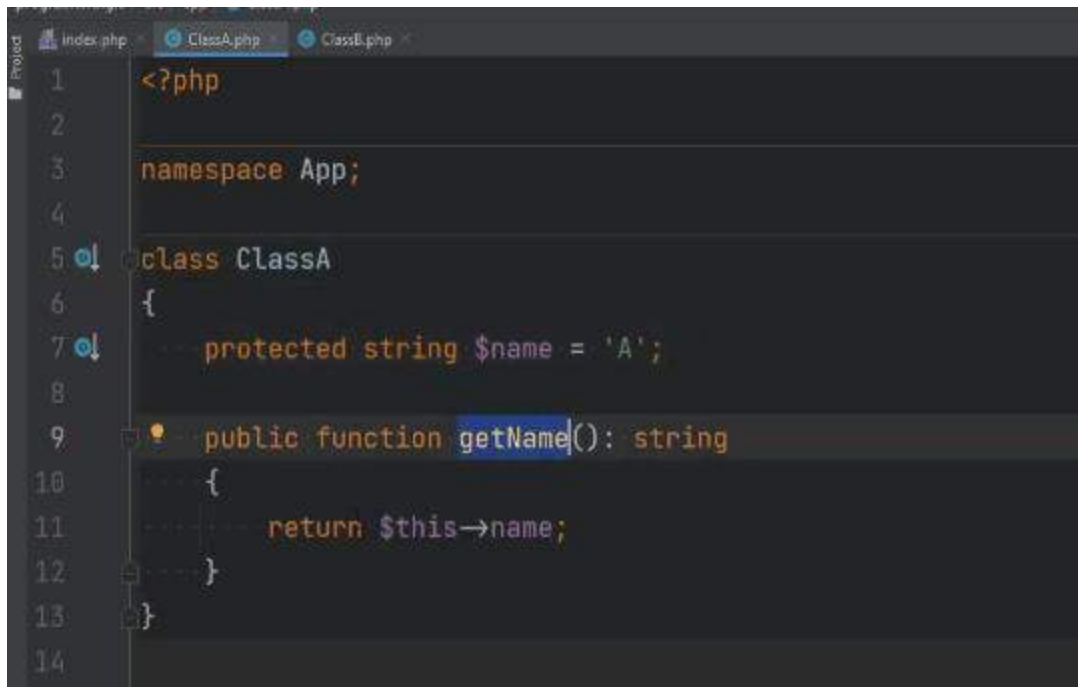
+ Hãy xem ví dụ để hiểu rõ hơn. Tôi có hai lớp ở đây. Tôi có Lớp A và Lớp B.



```
1 <?php
2
3 namespace App;
4
5 class ClassA
6 {
7     protected string $name = 'A';
8
9     public function getName(): string
10     {
11         return $this->name;
12     }
13 }
14
```

+ On the Class A, I have a property called Name with the value A.

+ Trong Lớp A, tôi có một thuộc tính gọi là "Name" với giá trị là "A".



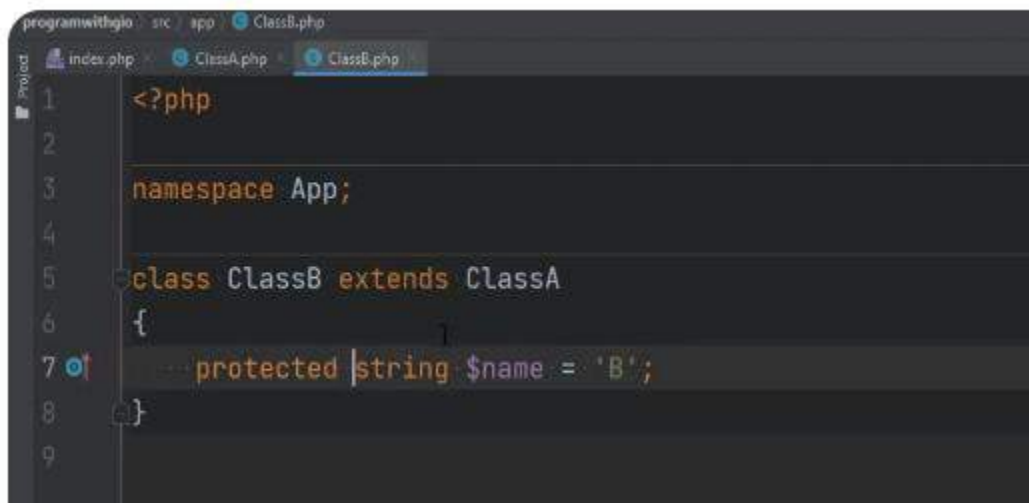
```

1  <?php
2
3  namespace App;
4
5  class ClassA
6  {
7      protected string $name = 'A';
8
9      public function getName(): string
10     {
11         return $this->name;
12     }
13 }
14

```

+I have a single method, getName, which simply returns that property.

+ Tôi có một phương thức duy nhất, "getName", chỉ đơn giản là trả về thuộc tính đó.



```

1  <?php
2
3  namespace App;
4
5  class ClassB extends ClassA
6  {
7      protected string $name = 'B';
8  }
9

```

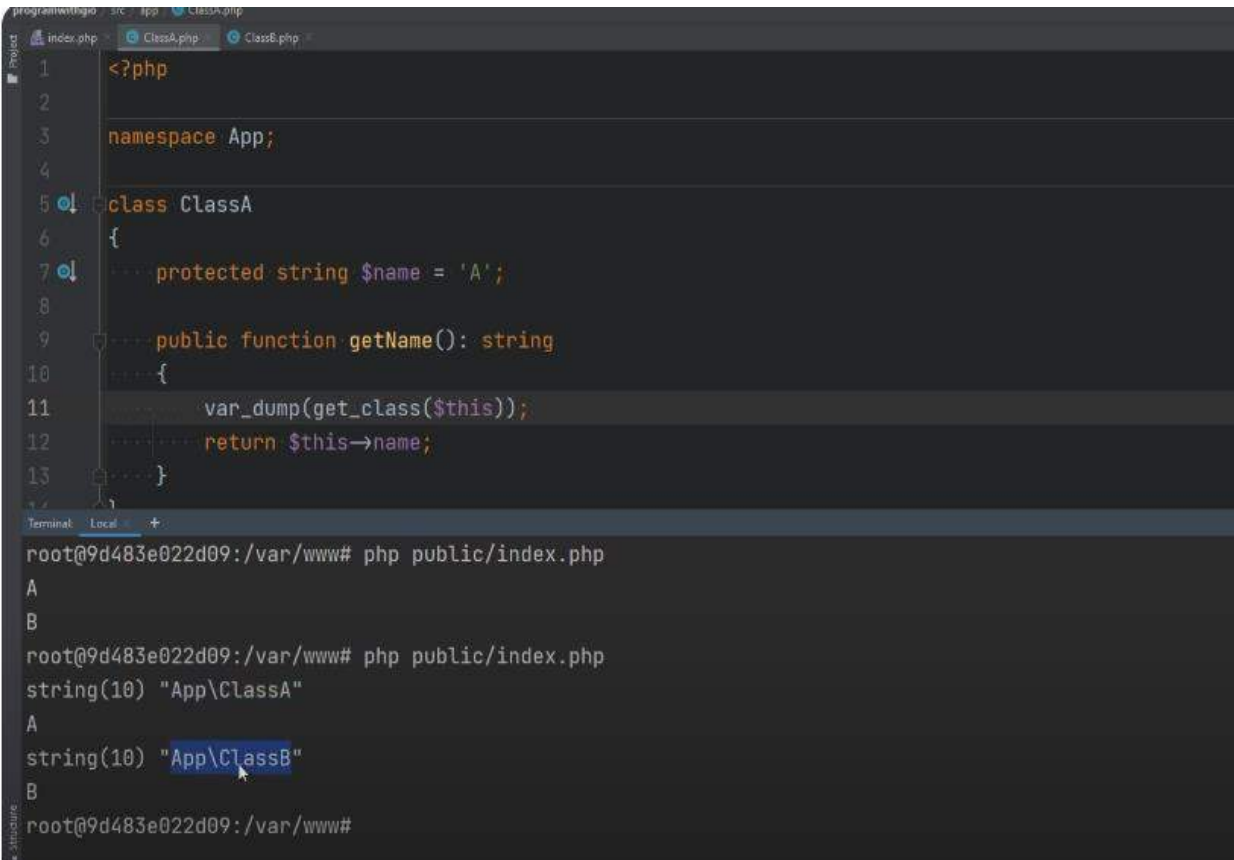
+ and Class B simply extends Class A and overwrites the name property with the value B. Then on the index. Php, I'm instantiating both of those classes and printing out the name properties.

+ và Lớp B chỉ đơn giản là mở rộng từ Lớp A và ghi đè thuộc tính "name" bằng giá trị "B". Sau đó, ở tệp "index.php", tôi khởi tạo cả hai lớp đó và in ra giá trị của thuộc tính "name".

```
Terminal: Local +
root@9d483e022d09:/var/www# php public/index.php
A
B
root@9d483e022d09:/var/www#
```

+ Let's run the code and make sure that this works. It does. It prints A and B, which is expected, right? Because on the first case, we have an object which is an instance of class A and therefore it prints A because the property value is set to A. In the second case, we have an object of class B which overwrites the property with the value of B. This is late binding where the class is resolved at runtime using the runtime information. As you know, this variable right here refers to the calling object. In the case of the second object here, this refers to the class B. The method calls here will depend on the type of the object that we're calling that method on. In the case of class A, we're calling that method on class A, and in the case of class B, we're calling that method on the class B. Because we're inheriting class A in class B, it just calls that method on class A, but this variable still refers to the calling object, which is class B. This class and method resolving and binding happens at runtime because it needs that additional runtime information to figure out on which class to call the method and on which class to access the certain properties and constants and so on.

+ Chúng ta hãy chạy mã và đảm bảo rằng nó hoạt động. Đúng vậy, nó hoạt động. Nó in ra A và B, điều này là mong đợi, phải không? Bởi vì trong trường hợp đầu tiên, chúng ta có một đối tượng là một thể hiện của lớp A và do đó nó in ra A vì giá trị thuộc tính được đặt là A. Trong trường hợp thứ hai, chúng ta có một đối tượng của lớp B mà ghi đè lên thuộc tính bằng giá trị B. Đây là ràng buộc muộn, nơi lớp được giải quyết vào thời gian chạy bằng cách sử dụng thông tin thời gian chạy. Như bạn biết, biến này ở đây tham chiếu đến đối tượng gọi. Trong trường hợp của đối tượng thứ hai ở đây, "this" tham chiếu đến lớp B. Các cuộc gọi phương thức ở đây sẽ phụ thuộc vào loại đối tượng mà chúng ta đang gọi phương thức đó. Trong trường hợp của lớp A, chúng ta đang gọi phương thức đó trên lớp A, và trong trường hợp của lớp B, chúng ta đang gọi phương thức đó trên lớp B. Bởi vì chúng ta kế thừa lớp A trong lớp B, nó chỉ đơn giản là gọi phương thức đó trên lớp A, nhưng biến "this" vẫn tham chiếu đến đối tượng gọi, là lớp B. Quá trình giải quyết và ràng buộc lớp và phương thức này xảy ra trong thời gian chạy vì nó cần thông tin thời gian chạy bổ sung để xác định lớp nào để gọi phương thức và lớp nào để truy cập các thuộc tính và hằng số cụ thể và vân vân.



```
<?php
namespace App;

class ClassA
{
    protected string $name = 'A';

    public function getName(): string
    {
        var_dump(get_class($this));
        return $this->name;
    }
}

class ClassB
{
    use ClassA;
}
```

```
root@9d483e022d09:/var/www# php public/index.php
A
B
root@9d483e022d09:/var/www# php public/index.php
string(10) "App\ClassA"
A
string(10) "App\ClassB"
B
root@9d483e022d09:/var/www#
```

+ This is basically what late binding is. To prove this, we can simply go to class A and Vardump the class of this object and let's see what we get. Let's run the code and we see that first time it's set to class A and the second time it's set to class B because we're calling the method on the object of class B right here.

+ Điều này cơ bản là những gì ràng buộc muộn (late binding) là. Để chứng minh điều này, chúng ta có thể đơn giản đi vào lớp A và sử dụng Vardump để xem lớp của đối tượng này là gì, hãy xem chúng ta thu được kết quả gì. Hãy chạy mã và chúng ta thấy rằng lần đầu tiên nó được thiết lập thành lớp A và lần thứ hai nó được thiết lập thành lớp B vì chúng ta đang gọi phương thức trên đối tượng của lớp B ngay tại đây.



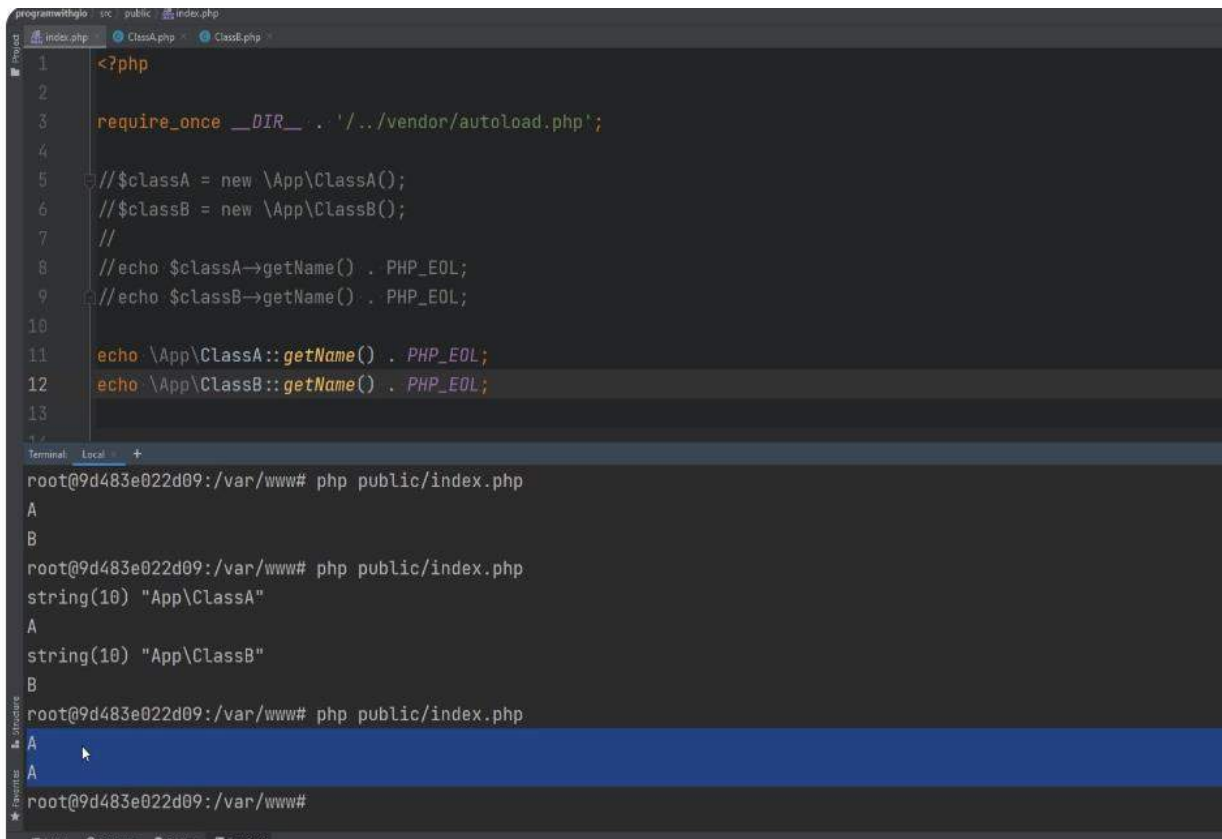
```
1 <?php
2
3 namespace App;
4
5 class ClassA
6 {
7     protected static string $name = 'A';
8
9     public static function getName(): string
10    {
11        var_dump(get_class($this));
12        return self::$name;
13    }
14 }
```

Terminal Local +

root@9d483e022d09:/var/www# php public/index.php

+ Let's make the property and method static and try to call the method statically from both class A and class B. I'm going to change this to static and do the same thing on class B and we need to change this method to static as well. As you know, we cannot use this within the static context. We could use the self keyword instead. I'm going to change this to self and let's get rid of that.

+ Hãy biến thuộc tính và phương thức thành tĩnh và thử gọi phương thức tĩnh từ cả lớp A và lớp B. Tôi sẽ thay đổi thành tĩnh và thực hiện cùng điều đó trên lớp B và chúng ta cần thay đổi phương thức này thành tĩnh luôn. Như bạn biết, chúng ta không thể sử dụng "this" trong ngữ cảnh tĩnh. Chúng ta có thể sử dụng từ khóa "self" thay thế. Tôi sẽ thay đổi thành "self" và loại bỏ điều này.



The screenshot shows a code editor with a file named `index.php` containing the following PHP code:

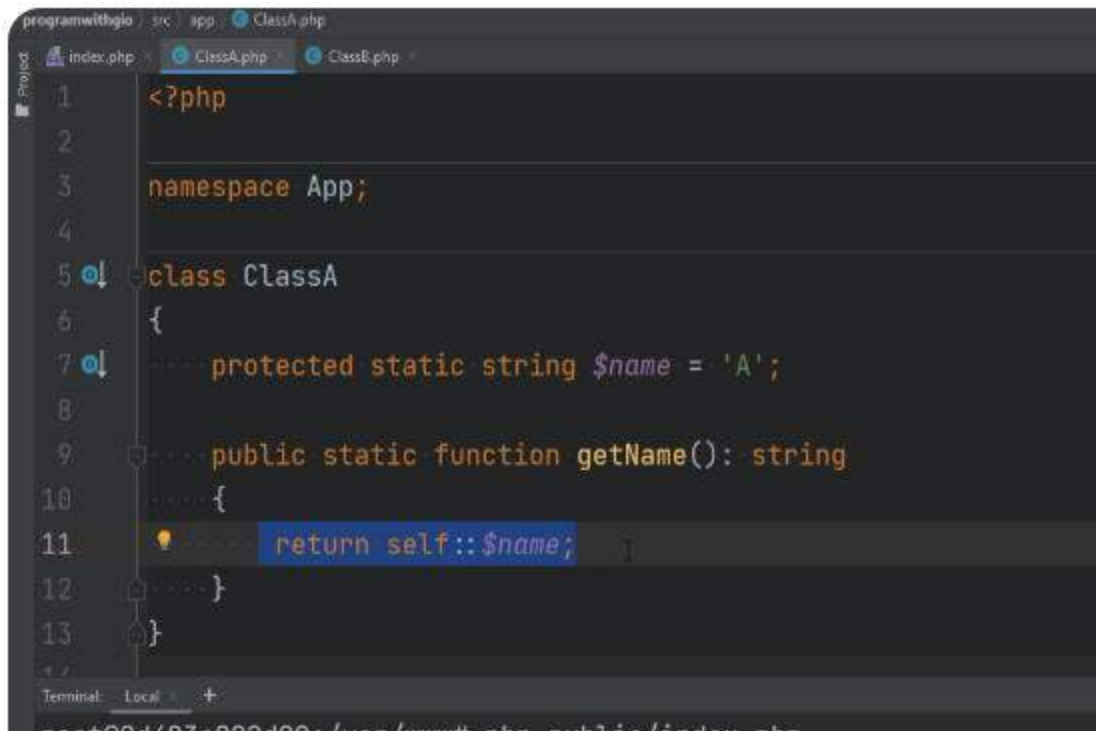
```
1 <?php
2
3 require_once __DIR__ . '/../vendor/autoload.php';
4
5 // $classA = new \App\ClassA();
6 // $classB = new \App\ClassB();
7 //
8 // echo $classA->getName() . PHP_EOL;
9 // echo $classB->getName() . PHP_EOL;
10
11 echo \App\ClassA::getName() . PHP_EOL;
12 echo \App\ClassB::getName() . PHP_EOL;
13
```

Below the code editor is a terminal window showing the execution of the script:

```
root@9d483e022d09:/var/www# php public/index.php
A
B
root@9d483e022d09:/var/www# php public/index.php
string(10) "App\ClassA"
A
string(10) "App\ClassB"
B
root@9d483e022d09:/var/www# php public/index.php
A
A
root@9d483e022d09:/var/www#
```

+ On the `index.php`, I'm simply going to comment these out. Let's call the method statically directly on the class. Let's do the same thing for class B and let's run the code and we get a. A, which is the value from class A. This is the problem because the expected output was AB.

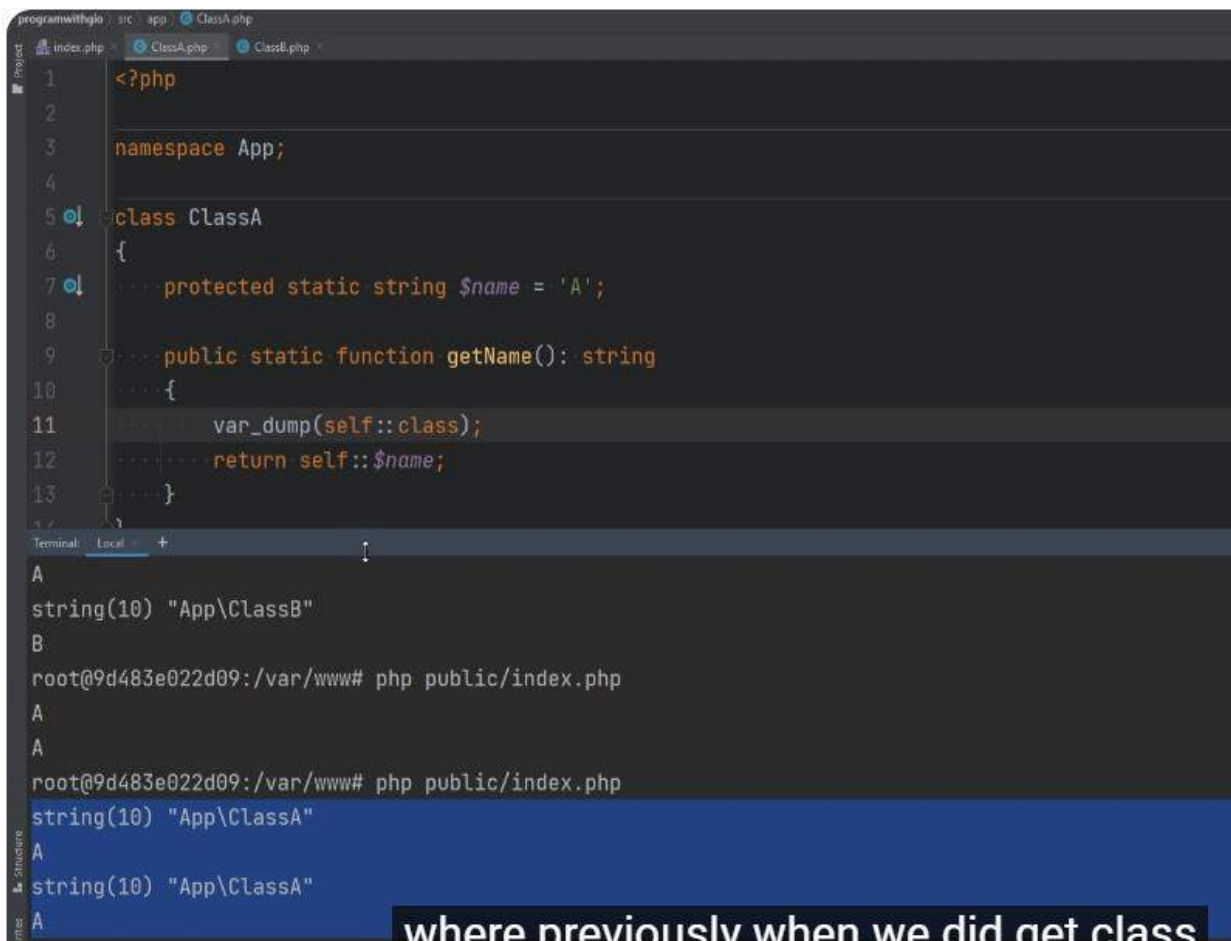
+ Trong tệp "`index.php`", tôi chỉ đơn giản là sẽ comment hai dòng này. Hãy gọi phương thức tĩnh trực tiếp trên lớp. Hãy làm điều tương tự cho lớp B và chúng ta chạy mã và nhận được kết quả là "A", giá trị từ lớp A. Điều này là vấn đề vì kết quả mong đợi là "AB".



```
1 <?php
2
3 namespace App;
4
5 class ClassA
6 {
7     protected static string $name = 'A';
8
9     public static function getName(): string
10    {
11        return self::$name;
12    }
13 }
```

+ That's what it was printing before when we were using this variable. But in the case when we use the self keyword, it's printing AA. This is early binding. Each time this line of code runs here, it will reference the same class. It resolves the class at compile time. This is also the limitation of the self keyword. Unlike this variable, self keyword does not follow the same inheritance rules and it resolves the class to which the method belongs to or where it was defined.

+ Đó là những gì nó đã in ra trước đây khi chúng ta sử dụng biến "this." Nhưng trong trường hợp chúng ta sử dụng từ khoá "self," nó in ra AA. Điều này được gọi là ràng buộc sớm (early binding). Mỗi lần dòng mã này chạy ở đây, nó sẽ tham chiếu đến cùng một lớp. Nó giải quyết lớp tại thời điểm biên dịch. Điều này cũng là giới hạn của từ khoá "self." Không giống như biến "this," từ khoá "self" không tuân theo cùng các quy tắc kế thừa và nó giải quyết lớp mà phương thức thuộc về hoặc nơi nó được định nghĩa.



```
<?php
namespace App;

class ClassA
{
    protected static string $name = 'A';

    public static function getName(): string
    {
        var_dump(self::class);
        return self::$name;
    }
}
```

Terminal: Local

```
A
string(10) "App\ClassB"
B
root@9d483e022d09:/var/www# php public/index.php
A
A
root@9d483e022d09:/var/www# php public/index.php
string(10) "App\ClassA"
A
string(10) "App\ClassA"
A
```

where previously when we did get class

+ We can var dump here self class to see what the class is. If I run the code again, we see that in both cases, it's printing class A, where previously when we did get class under this object, it was printing class B on the second one because we were calling the method on the object of class B. But in this case, even though we're calling the method on class B directly, it still resolves to class A when using the self keyword. This is the problem that late static binding solves, which we'll talk about in a minute. One way to solve this is to overwrite the method in the child class and print self name there.

+ Chúng ta có thể sử dụng "var_dump" để xem lớp của "self" ở đây. Nếu tôi chạy mã lại, chúng ta thấy rằng trong cả hai trường hợp, nó đều in ra "class A," trong khi trước đây khi chúng ta sử dụng "get_class" trên đối tượng này, nó đã in ra "class B" trong trường hợp thứ hai vì chúng ta đã gọi phương thức trên đối tượng của lớp B. Nhưng trong trường hợp này, mặc dù chúng ta gọi phương thức trực tiếp trên lớp B, nó vẫn giải quyết thành lớp A khi sử dụng từ khoá "self." Đây là vấn đề mà ràng buộc tĩnh giải quyết muộn, chúng ta sẽ thảo luận về nó trong một phút. Một cách để giải quyết vấn đề này là ghi đè phương thức trong lớp con và in ra "self name" ở đó.

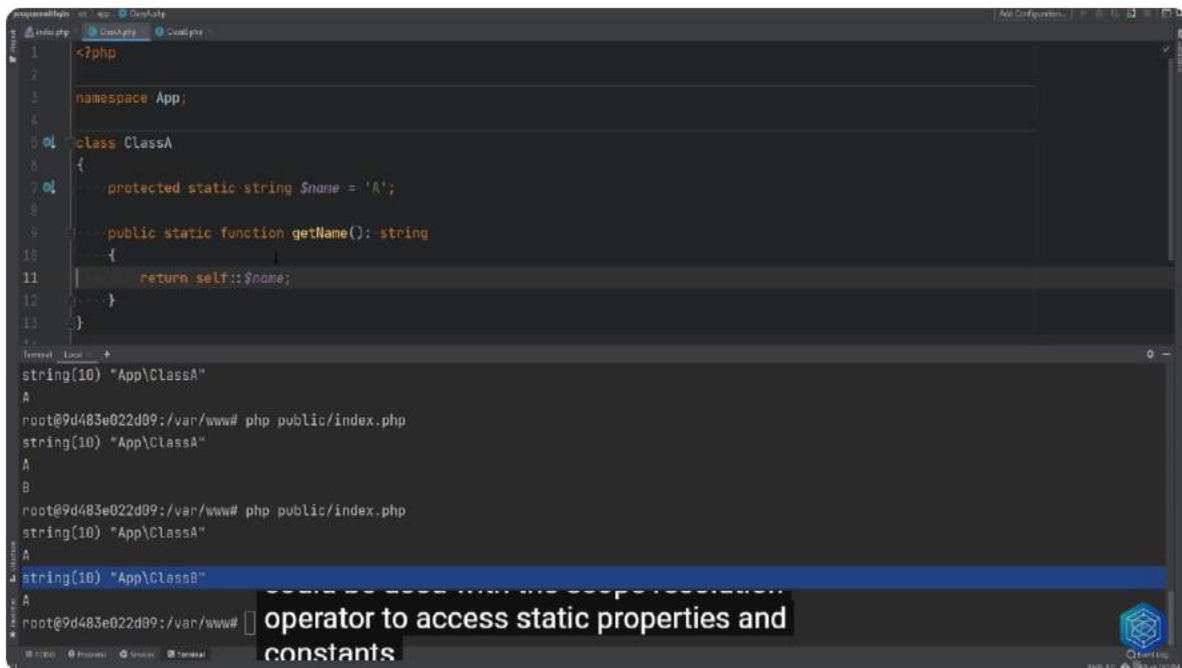
```
2
3 namespace App;
4
5 class ClassB extends ClassA
6 {
7     protected static string $name = 'B';
8
9     public static function getName(): string
10    {
11        return self::$name;
12    }
13 }
14
```

```
Terminal: Local
A
A
root@9d483e022d09:/var/www# php public/index.php
string(10) "App\ClassA"
A
string(10) "App\ClassA"
A
root@9d483e022d09:/var/www# php public/index.php
string(10) "App\ClassA"
A
B
root@9d483e022d09:/var/www#
```

which is expected but this is not ideal because it defeats

+ If we go to class B, we could simply overwrite the getName method here and simply return selfName. Now, if we run the code, we see that now it's returning AB, which is expected. But this is not ideal because it defeats the purpose of inheritance. We don't want to keep on overriding the methods this way. We want to have the base class and base functionality that can be inherited into the child classes. So this is not a great solution.

+ Nếu chúng ta vào lớp B, chúng ta có thể đơn giản ghi đè phương thức "getName" ở đây và chỉ đơn giản trả về "selfName". Bây giờ, nếu chúng ta chạy mã, chúng ta thấy rằng bây giờ nó trả về AB, điều này là mong đợi. Nhưng điều này không phải là giải pháp tốt vì nó mất đi ý nghĩa của việc kế thừa. Chúng ta không muốn tiếp tục ghi đè các phương thức theo cách này. Chúng ta muốn có lớp cơ sở và chức năng cơ bản có thể được kế thừa vào các lớp con. Vì vậy, đây không phải là một giải pháp tốt.



```
<?php
namespace App;

class ClassA
{
    protected static string $name = 'A';

    public static function getName(): string
    {
        return self::$name;
    }
}
```

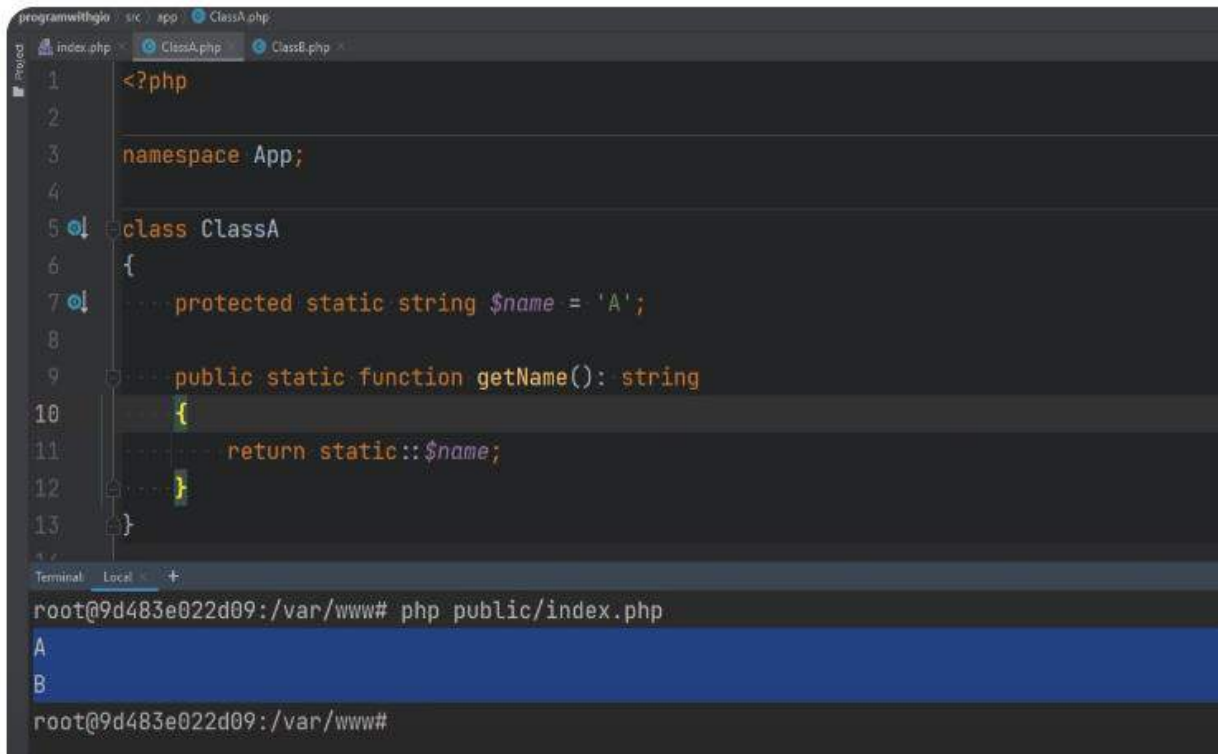
```
string(10) "App\ClassA"
A
root@9d483e022d89:/var/www# php public/index.php
string(10) "App\ClassA"
A
B
root@9d483e022d89:/var/www# php public/index.php
string(10) "App\ClassA"
A
string(10) "App\ClassB"
A
```

operator to access static properties and constants

What Is Late Static Binding & How It Works In PHP - Full PHP 8 Tutorial

+ In old school PHP, before the proper solution was added, a function called `getCalledClass` was used to figure out which is the calling class and then forward the static calls to that. So if we `VarDump` this right here instead of the `self`-class and we run the code, we see that it's giving us class A for the first time and it's giving us class B for the second time. Developers used to use this function to figure out which was the calling class and then they would forward the calls to that class. But then in PHB 5.3, a better solution was introduced called late static binding where the class is resolved using late binding at runtime instead of early binding at compile time.

+ Trong PHP cũ, trước khi giải pháp thích hợp được thêm vào, người ta đã sử dụng một hàm gọi là "`get_called_class`" để xác định lớp gọi và sau đó chuyển tiếp các cuộc gọi tĩnh đến đó. Vì vậy, nếu chúng ta sử dụng "`var_dump`" ở đây thay vì "`self class`" và chúng ta chạy mã, chúng ta thấy rằng nó đang trả về "`class A`" lần đầu và "`class B`" lần thứ hai. Những người phát triển trước đây đã sử dụng hàm này để xác định lớp gọi và sau đó chuyển tiếp cuộc gọi đến lớp đó. Nhưng sau đó, trong PHP 5.3, đã được giới thiệu một giải pháp tốt hơn gọi là "`late static binding`," nơi lớp được giải quyết bằng cách sử dụng ràng buộc tĩnh muộn ở thời gian chạy thay vì ràng buộc sớm ở thời gian biên dịch.



```
1 <?php
2
3 namespace App;
4
5 class ClassA
6 {
7     protected static string $name = 'A';
8
9     public static function getName(): string
10    {
11        return static::$name;
12    }
13 }

```

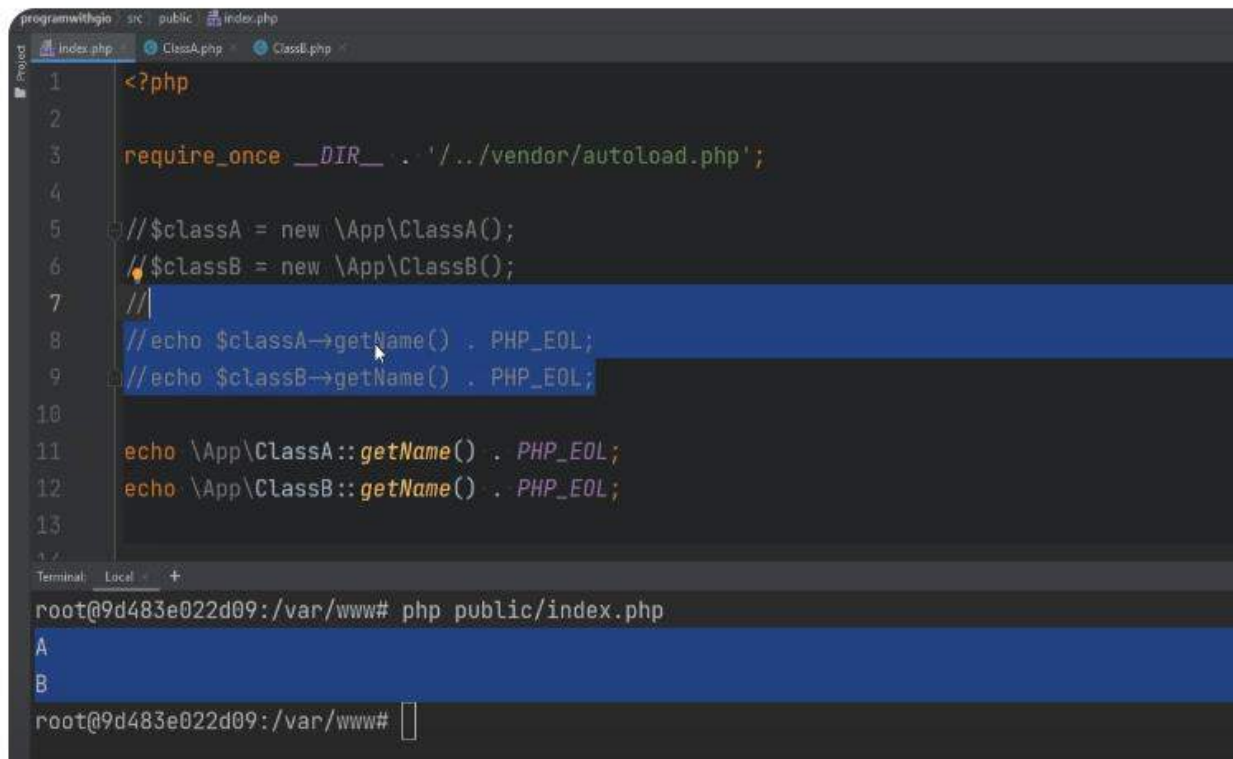
Terminal: Local

```
root@9d483e022d09:/var/www# php public/index.php
A
B
root@9d483e022d09:/var/www#
```

+ Instead of adding a new keyword, PHB and maintainers decided to use an already existing reserved keyword called static that could be used with the scope resolution operator to access static properties and constants and call static methods using late static binding. We can replace self keyword here with the static keyword, and let's clear this out and run the code again and we're getting AB, which is the expected output. Late static binding basically uses runtime information to determine how to call the method or how to access the property or the constant. The way this works is that when the method is called right here, PHP stores the original class name of the last non-forward and call. Then when it encounters the static keyword, it resolves to the original class that it had stored before. I mentioned non-forwarding call, and the example of non-forward and call is this when you directly specify the class name and it's usually before the scope resolution operator. However, when you use keywords like self and parents to access properties or call methods, those are called forward and calls because it might forward call to the parent class. You could also use static keyword in a non-static context.

+ Thay vì thêm từ khoá mới, PHP và người duy trì đã quyết định sử dụng một từ khoá đã tồn tại là "static" có thể được sử dụng với toán tử phạm vi để truy cập các thuộc tính tĩnh và hằng số và gọi các phương thức tĩnh bằng cách sử dụng ràng buộc tĩnh muộn. Chúng ta có thể thay thế từ khoá "self" ở đây bằng từ khoá "static," và hãy xóa phần này và chạy mã lại, chúng ta nhận được AB, đó là kết quả mong đợi. Ràng buộc tĩnh muộn cơ bản sử dụng thông tin thời gian chạy để xác định cách gọi phương thức hoặc cách truy cập thuộc tính hoặc hằng số. Cách hoạt động của nó là khi phương thức được gọi ở đây, PHP lưu trữ tên lớp gốc của cuộc gọi cuối cùng không chuyển tiếp. Sau đó, khi gặp từ khoá "static," nó được giải quyết thành lớp gốc mà nó đã lưu trữ trước đó. Tôi đã đề cập đến cuộc gọi không chuyển tiếp, và ví dụ về cuộc gọi không chuyển tiếp là khi bạn chỉ định trực tiếp tên lớp và thường xảy ra trước toán tử phạm vi. Tuy nhiên, khi bạn

sử dụng các từ khoá như "self" và "parent" để truy cập thuộc tính hoặc gọi phương thức, đó được gọi là cuộc gọi chuyển tiếp, vì nó có thể chuyển tiếp cuộc gọi đến lớp cha. Bạn cũng có thể sử dụng từ khoá "static" trong ngữ cảnh không tĩnh.



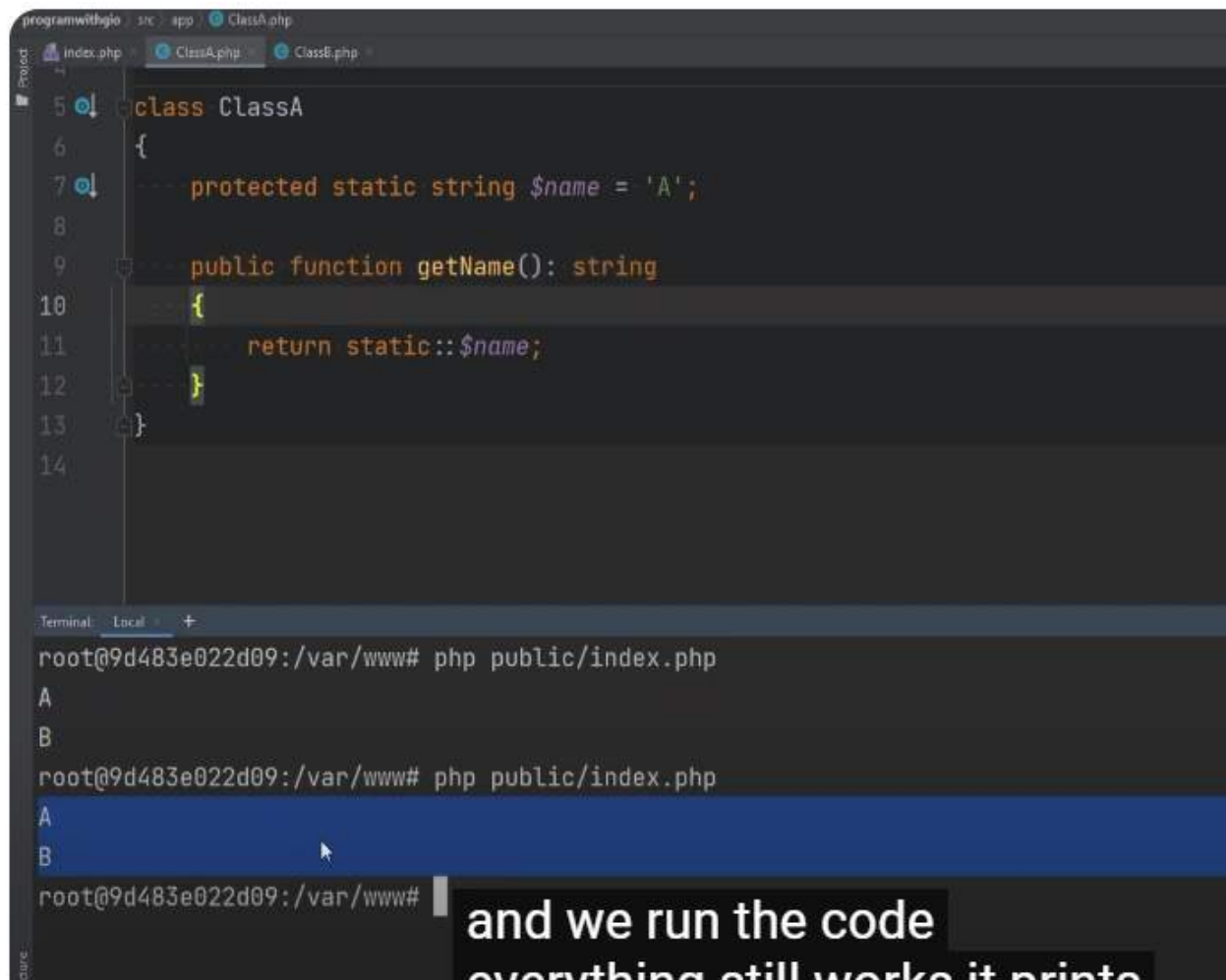
```
1 <?php
2
3 require_once __DIR__ . '/../vendor/autoload.php';
4
5 // $classA = new \App\ClassA();
6 // $classB = new \App\ClassB();
7 //
8 // echo $classA->getName() . PHP_EOL;
9 // echo $classB->getName() . PHP_EOL;
10
11 echo \App\ClassA::getName() . PHP_EOL;
12 echo \App\ClassB::getName() . PHP_EOL;
13
```

Terminal: Local +

```
root@9d483e022d09:/var/www# php public/index.php
A
B
root@9d483e022d09:/var/www#
```

+ This is the static context here, right? Because we are calling this method statically. However, you could also use the static keyword when you are in a non-static context, for example, in this case where we have the object and we're calling the methods on the object. The difference is that, as I mentioned before, in the static context, the class name is explicitly specified on the left side of the scope resolution operator, and that's the class name that gets stored by PHB and then used at runtime when static keyword is encountered. But in the context of objects when we're not within static calls, so if we do something like this, it will resolve to the class of the calling object.

+ Đúng, đây là ngữ cảnh tĩnh (static context), phải không? Bởi vì chúng ta đang gọi phương thức này tĩnh. Tuy nhiên, bạn cũng có thể sử dụng từ khoá "static" khi bạn đang ở trong ngữ cảnh không tĩnh, ví dụ, trong trường hợp này khi chúng ta có đối tượng và chúng ta đang gọi phương thức trên đối tượng. Sự khác biệt là, như tôi đã đề cập trước đó, trong ngữ cảnh tĩnh, tên lớp được xác định rõ ràng ở phía trái của toán tử phạm vi phạm vi, và đó là tên lớp được lưu trữ bởi PHP và sau đó được sử dụng vào thời gian chạy khi gặp từ khoá "static." Nhưng trong ngữ cảnh của đối tượng khi chúng ta không ở trong cuộc gọi tĩnh, vì vậy nếu chúng ta làm điều gì đó như thế này, nó sẽ được giải quyết thành lớp của đối tượng gọi.



The screenshot shows a code editor with a file named `ClassA.php`. The code defines a class `ClassA` with a protected static property `$name` set to 'A' and a public static method `getName()` that returns `static::$name`. Below the code, a terminal window shows the command `php public/index.php` being executed twice, resulting in the output 'A' and 'B' respectively. A text overlay at the bottom right of the terminal area reads: "and we run the code everything still works it prints".

```
5 class ClassA
6 {
7     protected static string $name = 'A';
8
9     public function getName(): string
10    {
11        return static::$name;
12    }
13 }
14
```

```
Terminal: Local +
root@9d483e022d09:/var/www# php public/index.php
A
B
root@9d483e022d09:/var/www# php public/index.php
A
B
root@9d483e022d09:/var/www#
```

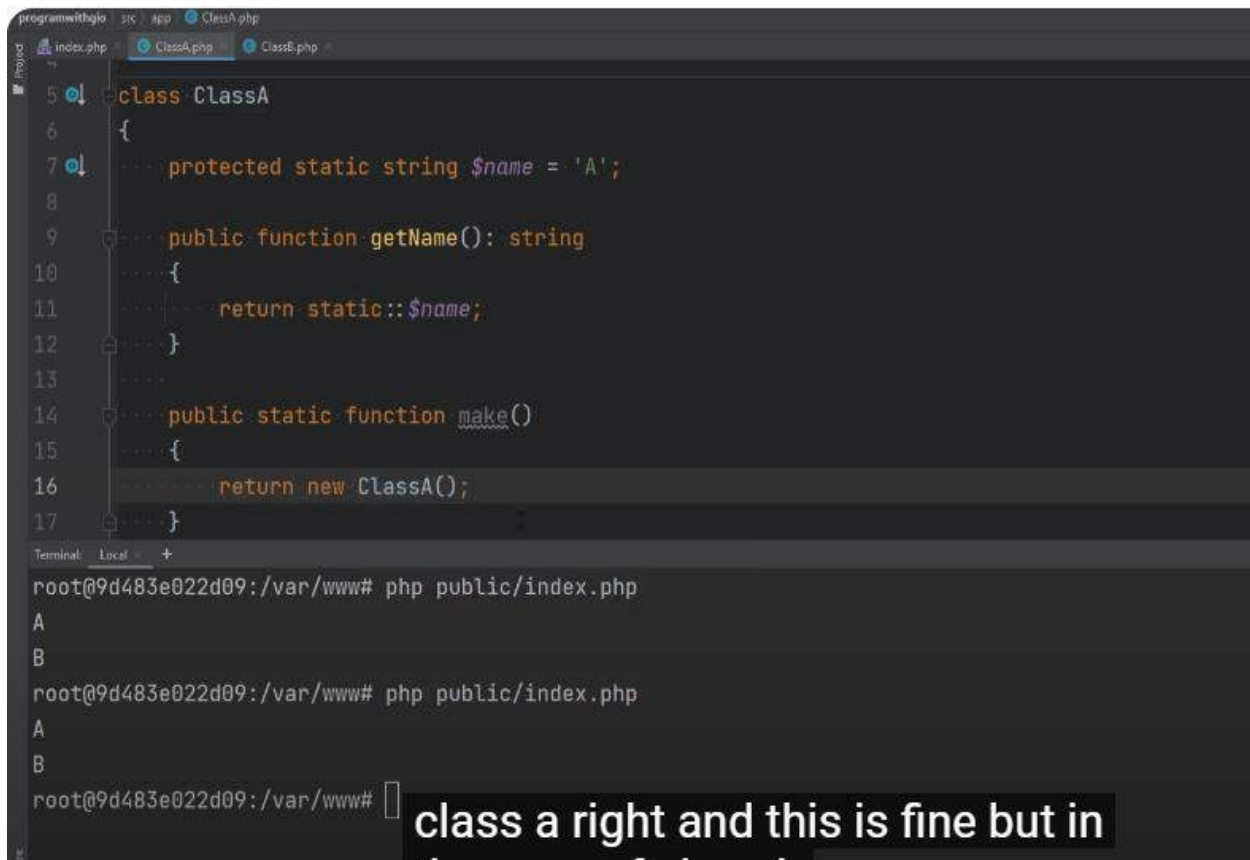
and we run the code
everything still works it prints

+ If we go back to get name and we change this to be non-static and we run the code, everything still works. It prints A and B. Now you might be saying, why can't you simply just use this variable here? And what do you need? The static keyword? The answer to that is, first of all, we cannot use this variable here because we're working with the static properties. But even if we were not working with the static properties, the difference between the static and this variable in a non-static context is that if you call the method using this variable, it could call a private method from the same scope while using static could result in a different method call.

+ Nếu chúng ta quay lại phương thức "getName" và chúng ta thay đổi nó thành không tĩnh và chạy mã, mọi thứ vẫn hoạt động. Nó in ra A và B. Bây giờ bạn có thể đặt câu hỏi, tại sao bạn không thể đơn giản sử dụng biến "this" ở đây? Và tại sao bạn cần từ khóa "static"? Câu trả lời cho điều đó là, trước hết, chúng ta không thể sử dụng biến "this" ở đây vì chúng ta đang làm việc với các thuộc tính tĩnh. Nhưng ngay cả khi chúng ta không làm việc với các thuộc tính tĩnh, sự khác biệt giữa từ khóa "static" và biến "this" trong ngữ cảnh không tĩnh là nếu bạn gọi phương thức bằng biến "this," nó có thể gọi một phương thức riêng tư từ cùng phạm vi trong khi sử dụng "static" có thể dẫn đến cuộc gọi phương thức khác.

+ Another difference is that you cannot use static keyword to access non-static properties. The same rules apply to constants. You could use static keyword to use late static binding to access overwritten constants as well. We wrap up this video, I want to mention that as of PHBA, you can also use the static keyword as a return type. You could already use the self or parent as return types before, but since PHPA, you could also use the static keyword. This can be useful when you're creating a new instance using the static keyword and returning that, for example, to implement something like a factory pattern.

+ + Một sự khác biệt khác là bạn không thể sử dụng từ khoá "static" để truy cập các thuộc tính không tĩnh. Cùng quy tắc tương tự áp dụng cho hằng số. Bạn cũng có thể sử dụng từ khoá "static" để sử dụng ràng buộc tĩnh muộn để truy cập các hằng số bị ghi đè. Trước khi kết thúc video này, tôi muốn nhắc đến rằng từ PHP 8, bạn cũng có thể sử dụng từ khoá "static" làm kiểu trả về (return type). Bạn đã có thể sử dụng "self" hoặc "parent" làm kiểu trả về trước đây, nhưng từ PHP 8, bạn cũng có thể sử dụng từ khoá "static." Điều này có thể hữu ích khi bạn đang tạo một thể hiện mới bằng từ khoá "static" và trả về nó, ví dụ, để thực hiện một mô hình như một mẫu factory.



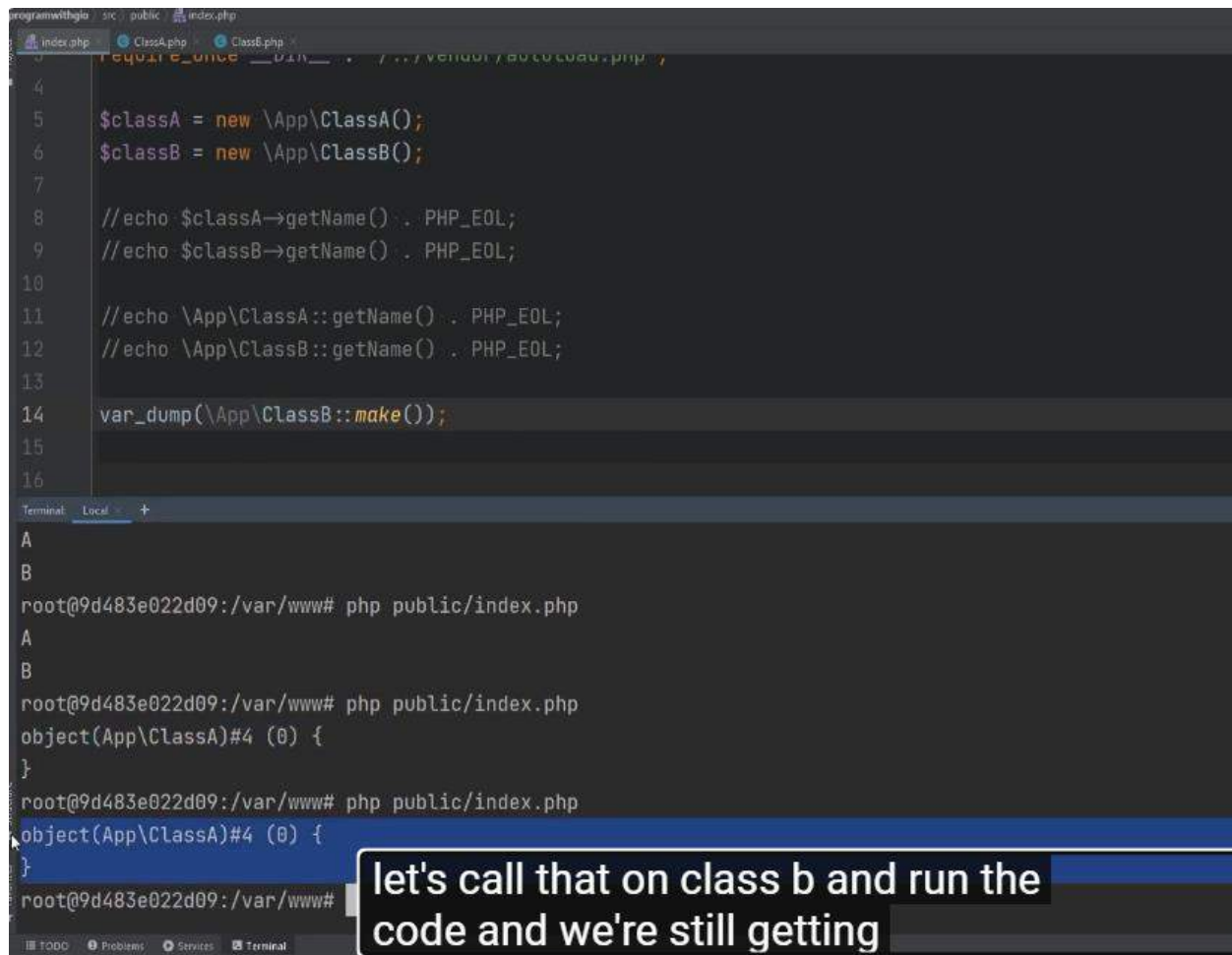
```
5 class ClassA
6 {
7     protected static string $name = 'A';
8
9     public function getName(): string
10    {
11        return static::$name;
12    }
13
14    public static function make()
15    {
16        return new ClassA();
17    }
18 }
```

```
Terminal: Local
root@9d483e022d09:/var/www# php public/index.php
A
B
root@9d483e022d09:/var/www# php public/index.php
A
B
root@9d483e022d09:/var/www#
```

class a right and this is fine but in

+ Let's do an example quick. Let's say that we had a public static function make here that simply returned the new object of class A. One way we would do this would be return new class A. This is fine, but in the case of class B, this would not be fine.

+ Hãy làm một ví dụ nhanh chóng. Giả sử chúng ta có một hàm tĩnh công khai make ở đây chỉ cần trả về đối tượng mới của lớp A. Một cách chúng ta sẽ làm điều này là return new class A. Điều này cũng ổn, nhưng trong trường hợp của lớp B, điều này sẽ không ổn.



The screenshot shows a code editor with a file named `index.php` containing the following PHP code:

```
4
5 $classA = new \App\ClassA();
6 $classB = new \App\ClassB();
7
8 //echo $classA->getName() . PHP_EOL;
9 //echo $classB->getName() . PHP_EOL;
10
11 //echo \App\ClassA::getName() . PHP_EOL;
12 //echo \App\ClassB::getName() . PHP_EOL;
13
14 var_dump(\App\ClassB::make());
15
16
```

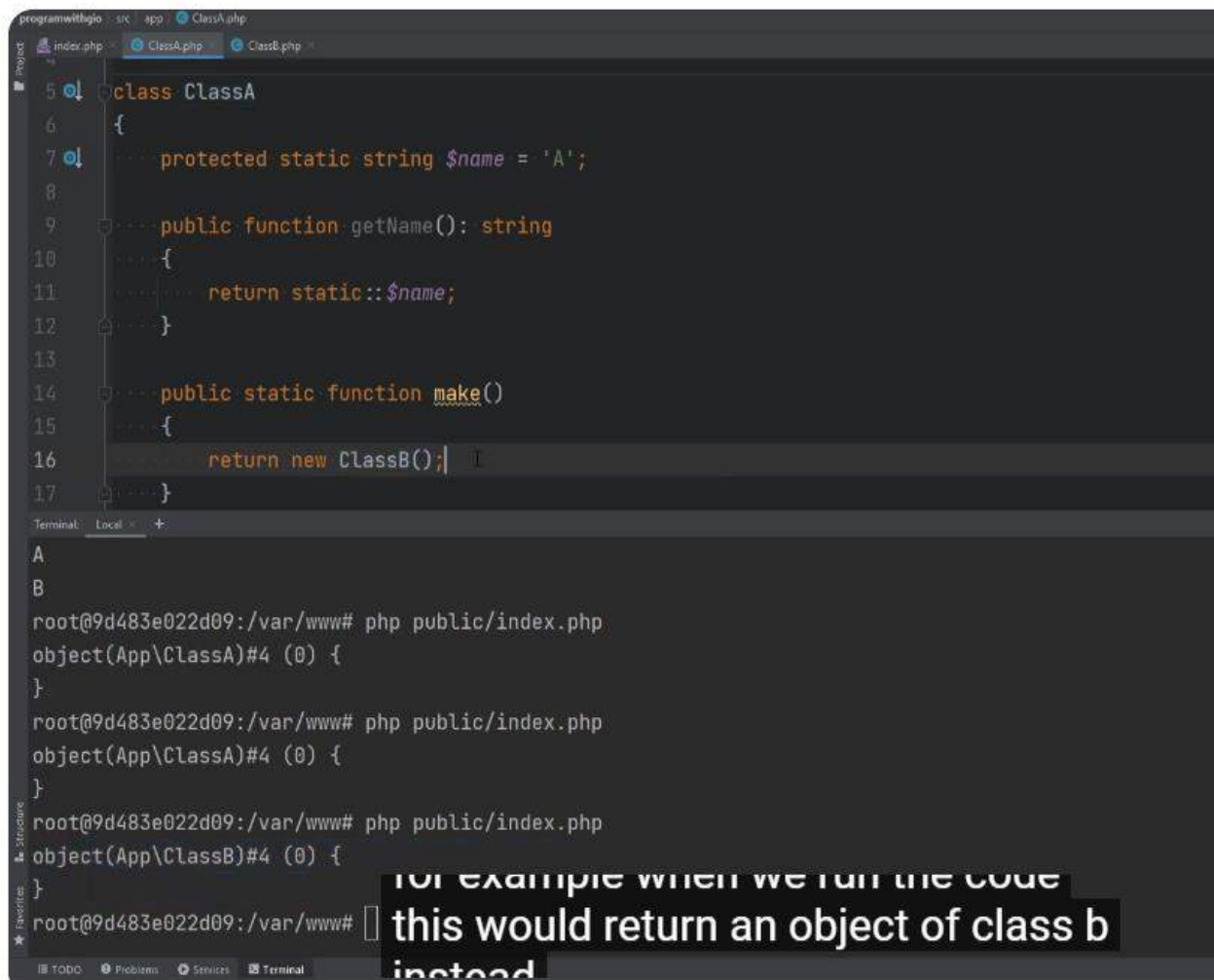
Below the code editor is a terminal window showing the execution of the script. The terminal output is as follows:

```
root@9d483e022d09:/var/www# php public/index.php
A
B
root@9d483e022d09:/var/www# php public/index.php
object(App\ClassA)#4 (0) {
}
root@9d483e022d09:/var/www# php public/index.php
object(App\ClassA)#4 (0) {
}
root@9d483e022d09:/var/www#
```

A text box with a blue background and white text is overlaid on the terminal output, stating: "let's call that on class b and run the code and we're still getting".

+ If we go to index. Php and let's comment this out. Let's simply Vardump class A make and see what we get. We run the code and we get an object of A, which is fine. Let's call that on class B and run the code and we're still getting the object of class A.

+ Nếu chúng ta vào index.php và hãy comment phần này đi. Chúng ta chỉ đơn giản "var_dump(class A::make())" và xem kết quả chúng ta thu được. Chúng ta chạy mã và chúng ta nhận được một đối tượng của A, điều này ổn. Hãy gọi hàm này trên lớp B và chạy mã, chúng ta vẫn nhận được một đối tượng của lớp A.



```
5 class ClassA
6 {
7     protected static string $name = 'A';
8
9     public function getName(): string
10    {
11        return static::$name;
12    }
13
14    public static function make()
15    {
16        return new ClassB();
17    }
18 }
```

```
Terminal: Local x +
A
B
root@9d483e022d09:/var/www# php public/index.php
object(App\ClassA)#4 (0) {
}
root@9d483e022d09:/var/www# php public/index.php
object(App\ClassA)#4 (0) {
}
root@9d483e022d09:/var/www# php public/index.php
object(App\ClassB)#4 (0) {
}
root@9d483e022d09:/var/www#
```

For example when we run the code this would return an object of class b instead

+ That's because we're being explicit right here. We're returning a new object of class A. If we returned a new object of class B, for example, when we run the code, this would return an object of class B.

+ Điều đó là vì chúng ta đang rõ ràng ở đây. Chúng ta đang trả về một đối tượng mới của lớp A. Nếu chúng ta trả về một đối tượng mới của lớp B, ví dụ, khi chúng ta chạy mã, điều này sẽ trả về một đối tượng của lớp B.

```
5 class ClassA
6 {
7     protected static string $name = 'A';
8
9     public function getName(): string
10    {
11        return static::$name;
12    }
13
14    public static function make(): self
15    {
16        return new static();
17    }
18 }
```

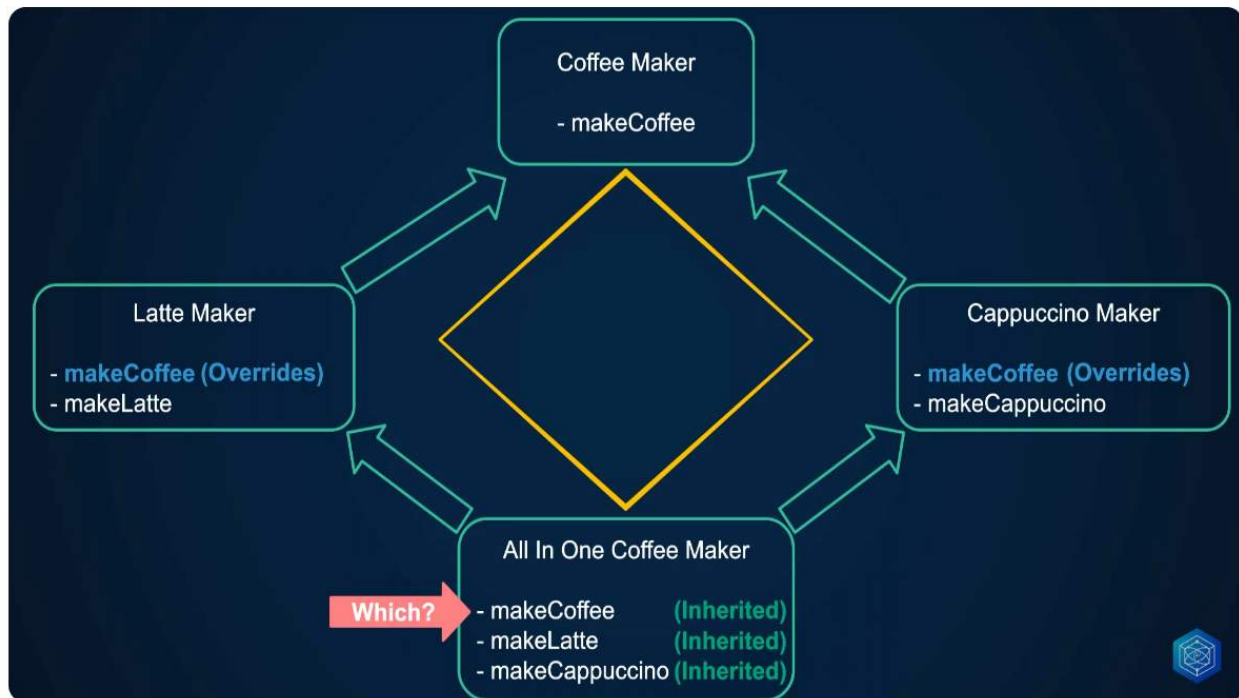
```
Terminal Local +
object(App\ClassA)#4 (0) {
}
root@9d483e022d09:/var/www# php public/index.php
object(App\ClassB)#4 (0) {
}
root@9d483e022d09:/var/www# php public/index.php
object(App\ClassA)#4 (0) {
}
root@9d483e022d09:/var/www# php public/index.php
object(App\ClassB)#4 (0) {
}
root@9d483e022d09:/var/www#
```

self but as you know self would always resolve to class a

+Instead of being specific, you could also do new self, or you could do new parent if you were in a child class and that would also work. But again, this would result in the same problem. If I run the code, we're still going to get the object of class A even when we're calling the make method on class B. To solve that problem is to use static keyword. When we run the code, now it returns an object of B. What was added in PHB is that in the return type, you were before able to specify self. But as you know, self would always resolve to class A. But since PHB 8, now you could specify static, and now this will match the actual returned class type. If we run the code, everything still works. If we change this back to class A and run the code, it returns the object of class A. All right, that's it for this video.

+Thay vì cụ thể, bạn cũng có thể làm new self hoặc bạn có thể làm new parent nếu bạn đang ở trong một lớp con và điều đó cũng sẽ hoạt động. Nhưng một lần nữa, điều này sẽ dẫn đến vấn đề tương tự. Nếu tôi chạy mã, chúng ta vẫn sẽ nhận được một đối tượng của lớp A, ngay cả khi chúng ta gọi phương thức "make" trên lớp B. Để giải quyết vấn đề đó là sử dụng từ khóa "static." Khi chúng ta chạy mã, bây giờ nó trả về một đối tượng của B. Điều được thêm vào trong PHP 8 là trong kiểu trả về, bạn trước đây có thể chỉ định "self." Nhưng như bạn biết, "self" luôn giải quyết thành lớp A. Nhưng từ PHP 8 trở đi, bây giờ bạn có thể chỉ định "static," và bây giờ nó sẽ khớp với kiểu lớp thực tế được trả về. Nếu chúng ta chạy mã, mọi thứ vẫn hoạt động. Nếu chúng ta thay đổi thành lớp A và chạy mã, nó trả về đối tượng của lớp A. Đó là tất cả cho video này.

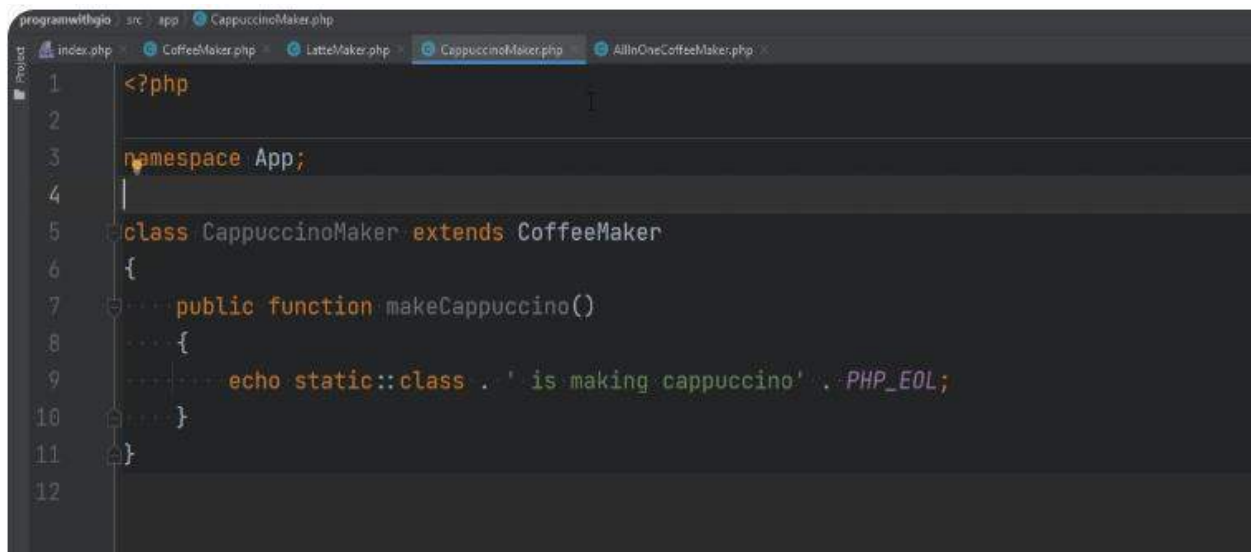
VIDEO 2.14:



+ Php is a single-inheritance language and does not support multiple inheritance, which means that you can only extend a single class. But there are ways you can achieve something similar to multiple inheritance in PHP using traits or interfaces. Before we get into traits, let's see what the problem is with the multiple inheritance and why we would care to have an alternative to multiple inheritance in PHP. Let's say that we have a coffee maker class that has a method called Make Coffee. Then also let's say that we get the specialty coffee makers like Latte Maker and Cappuccino Maker. The thing is these specialty coffee makers can also make regular coffee. In this case, we would leverage the power of inheritance and extend the coffee maker from the latte and Cappuccino Maker and just add additional methods called Make Latte and Make Cappuccino. This is single inheritance and it is well supported and works in PHP. Now, let's say that down the road we got an all-in-one coffee maker which can make regular coffee, but it can also make latte and you can make cappuccino in addition to some other additional features. In some other languages, you could use multiple inheritance and simply extend both latte maker and cappuccino maker in the all-in-one coffee maker class. This is not supported in PHP and multiple inheritance does have its own problems. One such problem is called the diamond problem. As you notice, this forms a shape of a diamond. The problem is that if we override the make coffee method within both latte and cappuccino maker classes and we do not override the make coffee method within the all-in-one coffee maker, the all-in-one coffee maker class would not know which make coffee method to run. This is the problem and some programming languages that support multiple inheritance solve this problem in different ways. However, PHP supports traits that let you share common functionality from multiple classes. Essentially, you're getting some benefits of the multiple inheritance without actually extending classes.

+ PHP là một ngôn ngữ đơn thừa kế (single-inheritance) và không hỗ trợ đa thừa kế (multiple inheritance), điều này có nghĩa là bạn chỉ có thể mở rộng một lớp duy nhất. Tuy nhiên, có cách

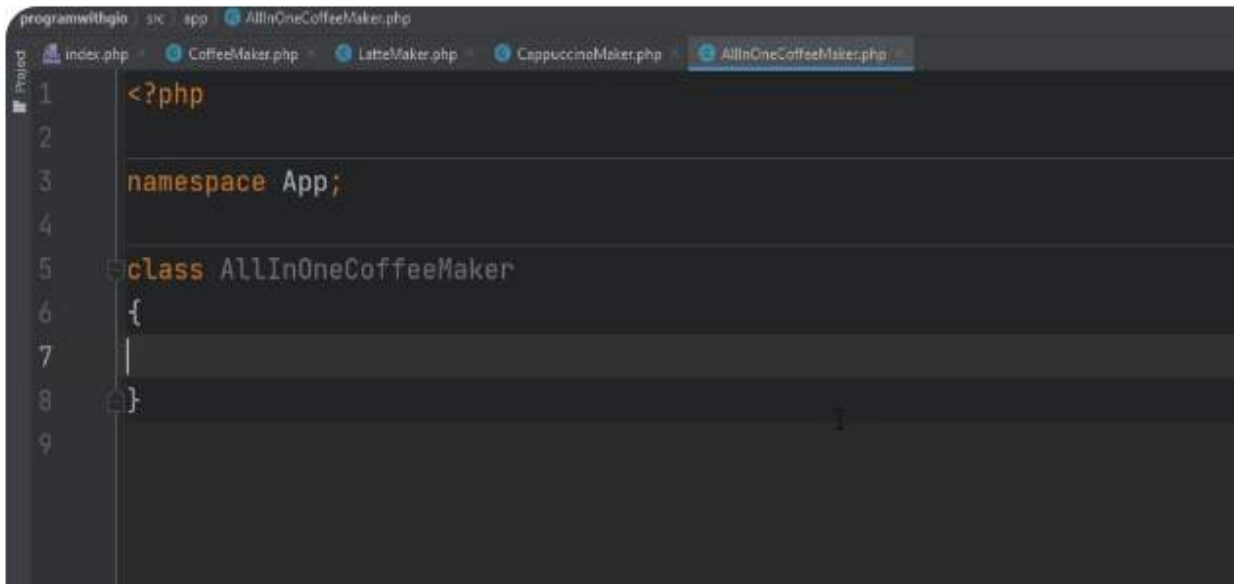
bạn có thể đạt được điều tương tự với đa thừa kế trong PHP bằng cách sử dụng traits hoặc interfaces. Trước khi chúng ta nói về traits, hãy xem vấn đề với đa thừa kế và tại sao chúng ta quan tâm đến việc có một phương thức thay thế cho đa thừa kế trong PHP. Giả sử chúng ta có một lớp "CoffeeMaker" có một phương thức gọi là "MakeCoffee." Tiếp theo, giả sử chúng ta có các loại máy pha cà phê đặc biệt như "LatteMaker" và "CappuccinoMaker." Điều quan trọng là những máy pha cà phê đặc biệt này cũng có thể pha cà phê thông thường. Trong trường hợp này, chúng ta sẽ tận dụng sức mạnh của kế thừa và mở rộng lớp "CoffeeMaker" từ "LatteMaker" và "CappuccinoMaker" và chỉ cần thêm các phương thức bổ sung gọi là "MakeLatte" và "MakeCappuccino." Đây là đơn thừa kế và nó được hỗ trợ tốt và hoạt động trong PHP. Bây giờ, giả sử rằng sau này chúng ta có một máy pha cà phê "AllInOne" có thể pha cà phê thông thường, nhưng nó cũng có thể pha latte và bạn có thể pha cappuccino, cùng với một số tính năng bổ sung khác. Trong một số ngôn ngữ khác, bạn có thể sử dụng đa thừa kế và đơn giản là mở rộng cả "LatteMaker" và "CappuccinoMaker" trong lớp "AllInOne." Điều này không được hỗ trợ trong PHP và đa thừa kế có những vấn đề riêng. Một trong những vấn đề đó được gọi là "vấn đề kim cương" (diamond problem). Như bạn thấy, điều này tạo thành hình dạng của một viên kim cương. Vấn đề ở đây là nếu chúng ta ghi đè phương thức "makeCoffee" trong cả hai lớp "LatteMaker" và "CappuccinoMaker" và chúng ta không ghi đè phương thức "makeCoffee" trong lớp "AllInOneCoffeeMaker," lớp "AllInOneCoffeeMaker" sẽ không biết phải chạy phương thức "makeCoffee" nào. Đây là vấn đề và một số ngôn ngữ lập trình hỗ trợ đa thừa kế giải quyết vấn đề này theo các cách khác nhau. Tuy nhiên, PHP hỗ trợ traits (đặc điểm) cho phép bạn chia sẻ chức năng chung từ nhiều lớp. Theo cách này, bạn đang có một số lợi ích của đa thừa kế mà không cần kế thừa các lớp thực sự.



```
1 <?php
2
3 namespace App;
4
5 class CoffeeMaker {
6     public function makeCoffee()
7     {
8         echo static::class . ' is making coffee' . PHP_EOL;
9     }
10 }
11
12 class LatteMaker extends CoffeeMaker {
13     public function makeLatte()
14     {
15         echo static::class . ' is making latte' . PHP_EOL;
16     }
17 }
18
19 class CappuccinoMaker extends CoffeeMaker {
20     public function makeCappuccino()
21     {
22         echo static::class . ' is making cappuccino' . PHP_EOL;
23     }
24 }
```

+ Let's see this example in code. I have a coffee maker class here with the Make Coffee method that simply echoes out the class name is making coffee. Then we have the latte maker that extends the coffee maker and it has a method called make latte that simply echoes out the class is making latte. We have the same thing in the cappuccino maker, which extends coffee maker and has make cappuccino method, which echoes out the class is making cappuccino.

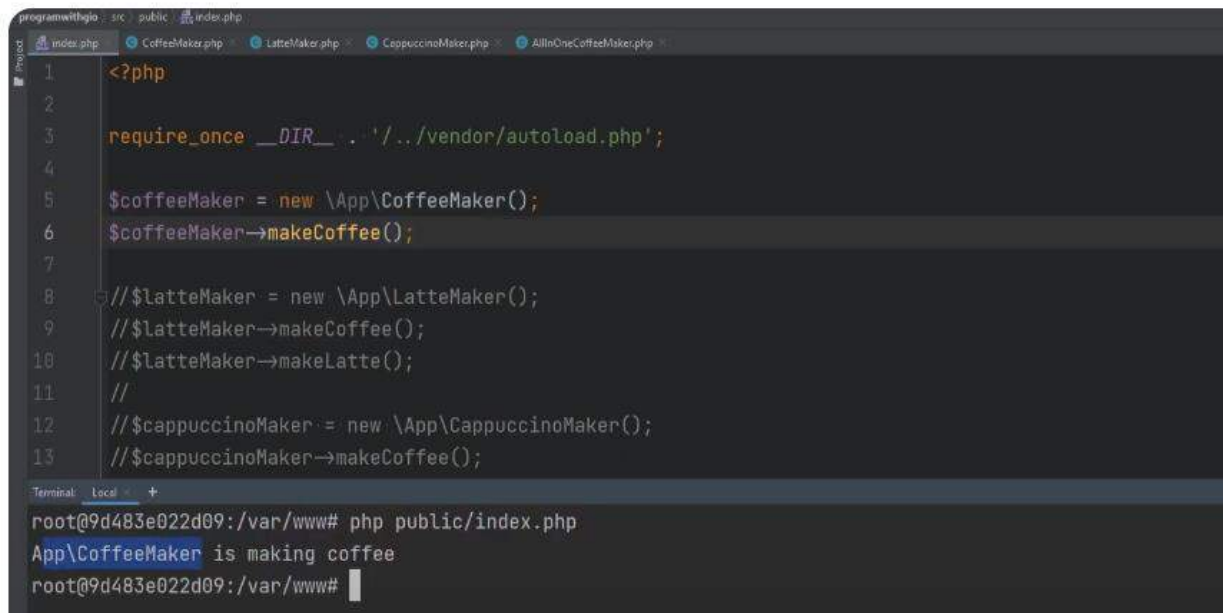
+ Hãy xem ví dụ này trong mã. Tôi có một lớp "CoffeeMaker" ở đây với phương thức "MakeCoffee" đơn giản là in ra tên lớp đang làm cà phê. Sau đó, chúng ta có lớp "LatteMaker" mở rộng từ "CoffeeMaker" và có một phương thức "makeLatte" đơn giản in ra tên lớp đang làm latte. Chúng ta có điều tương tự trong lớp "CappuccinoMaker," mở rộng từ "CoffeeMaker" và có phương thức "makeCappuccino" in ra tên lớp đang làm cappuccino.



```
1 <?php
2
3 namespace App;
4
5 class AllInOneCoffeeMaker
6 {
7
8 }
9
```

+Then we have all-in-one coffee maker, which we're going to fill it in in a minute. Let's test this out so far and make sure that everything is working.

+ Sau đó, chúng ta có lớp "AllInOneCoffeeMaker," chúng ta sẽ điền thông tin vào lớp này trong một phút. Hãy thử nghiệm điều này cho đến nay và đảm bảo rằng mọi thứ đang hoạt động.

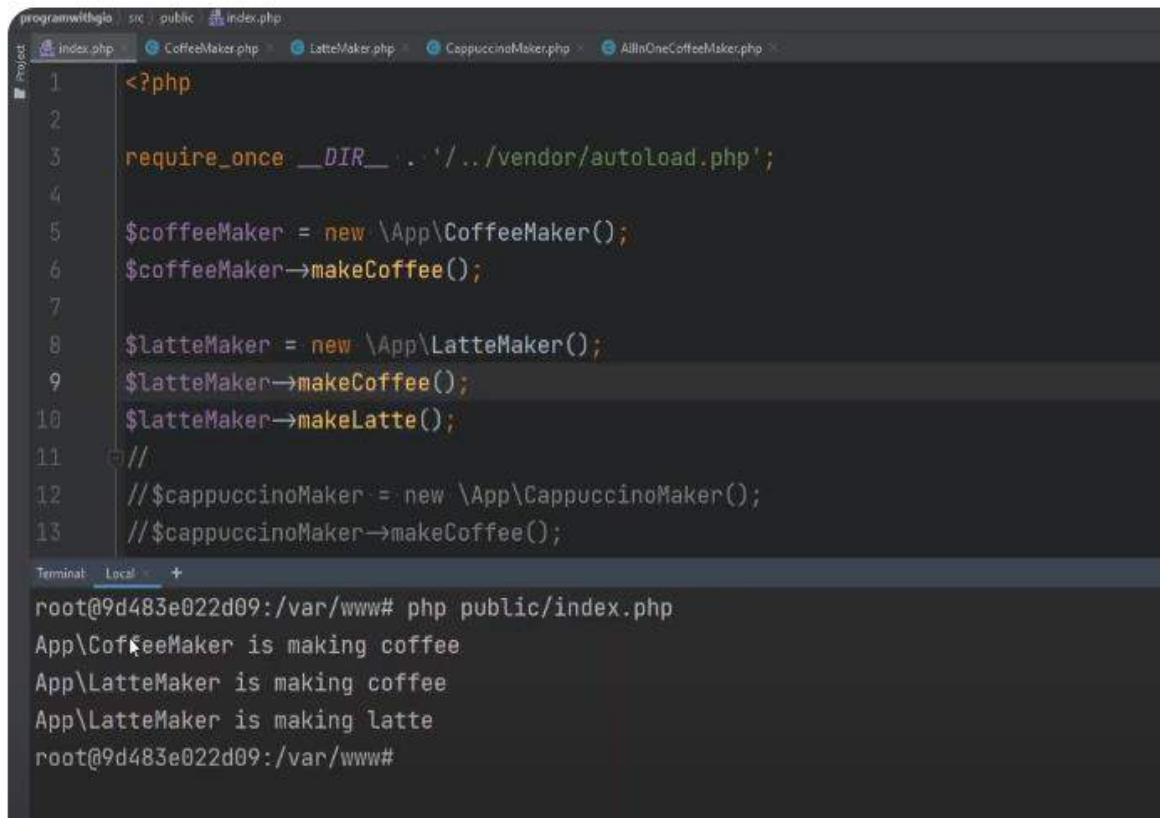


```
1 <?php
2
3 require_once __DIR__ . '/../vendor/autoload.php';
4
5 $coffeeMaker = new \App\CoffeeMaker();
6 $coffeeMaker->makeCoffee();
7
8 // $latteMaker = new \App\LatteMaker();
9 // $latteMaker->makeCoffee();
10 // $latteMaker->makeLatte();
11 //
12 // $cappuccinoMaker = new \App\CappuccinoMaker();
13 // $cappuccinoMaker->makeCoffee();
```

```
root@9d483e022d09:/var/www# php public/index.php
App\CoffeeMaker is making coffee
root@9d483e022d09:/var/www#
```

+ Let's switch over to index. Phb. I have this code written here that's commented out for each individual coffee maker. Let's uncomment it and run it and see what we get. We see that coffee maker is making coffee.

+ Hãy chuyển sang tập tin index.php. Tôi đã viết mã ở đây đã được comment cho từng máy pha cà phê cá nhân. Hãy bỏ comment và chạy nó và xem kết quả. Chúng ta thấy rằng CoffeeMaker đang làm cà phê.

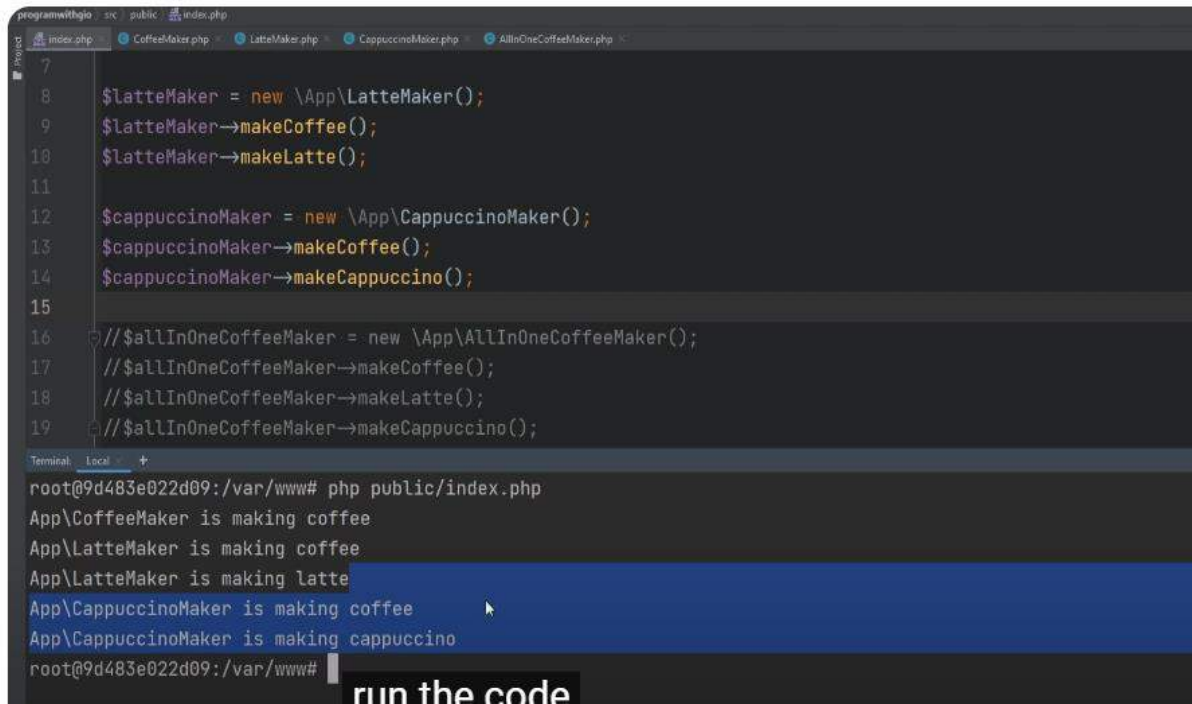


```
1 <?php
2
3 require_once __DIR__ . '/../vendor/autoload.php';
4
5 $coffeeMaker = new \App\CoffeeMaker();
6 $coffeeMaker->makeCoffee();
7
8 $latteMaker = new \App\LatteMaker();
9 $latteMaker->makeCoffee();
10 $latteMaker->makeLatte();
11 //
12 // $cappuccinoMaker = new \App\CappuccinoMaker();
13 // $cappuccinoMaker->makeCoffee();
```

```
Terminal Local +
root@9d483e022d09:/var/www# php public/index.php
App\CoffeeMaker is making coffee
App\LatteMaker is making coffee
App\LatteMaker is making latte
root@9d483e022d09:/var/www#
```

+ Now, let's uncomment the latte maker here and let's clear this out. Let's run the code again and we see that Coffee Maker is making coffee. Then LatteMaker is making coffee maker is making coffee, then latte maker is making coffee, and latte maker is making latte.

+ Bây giờ, hãy bỏ comment lớp "LatteMaker" ở đây và xóa nội dung này. Hãy chạy mã lại và chúng ta thấy rằng CoffeeMaker đang làm cà phê, sau đó LatteMaker đang làm cà phê, và LatteMaker đang làm latte.



The screenshot shows a code editor with a project named 'programwithhilo' and a file 'index.php' in the 'public' directory. The code defines several coffee maker classes and their methods. The terminal output shows the execution of the code, displaying the output of the 'makeCoffee()' and 'makeLatte()' methods for the 'App\CoffeeMaker' and 'App\LatteMaker' classes. The output is as follows:

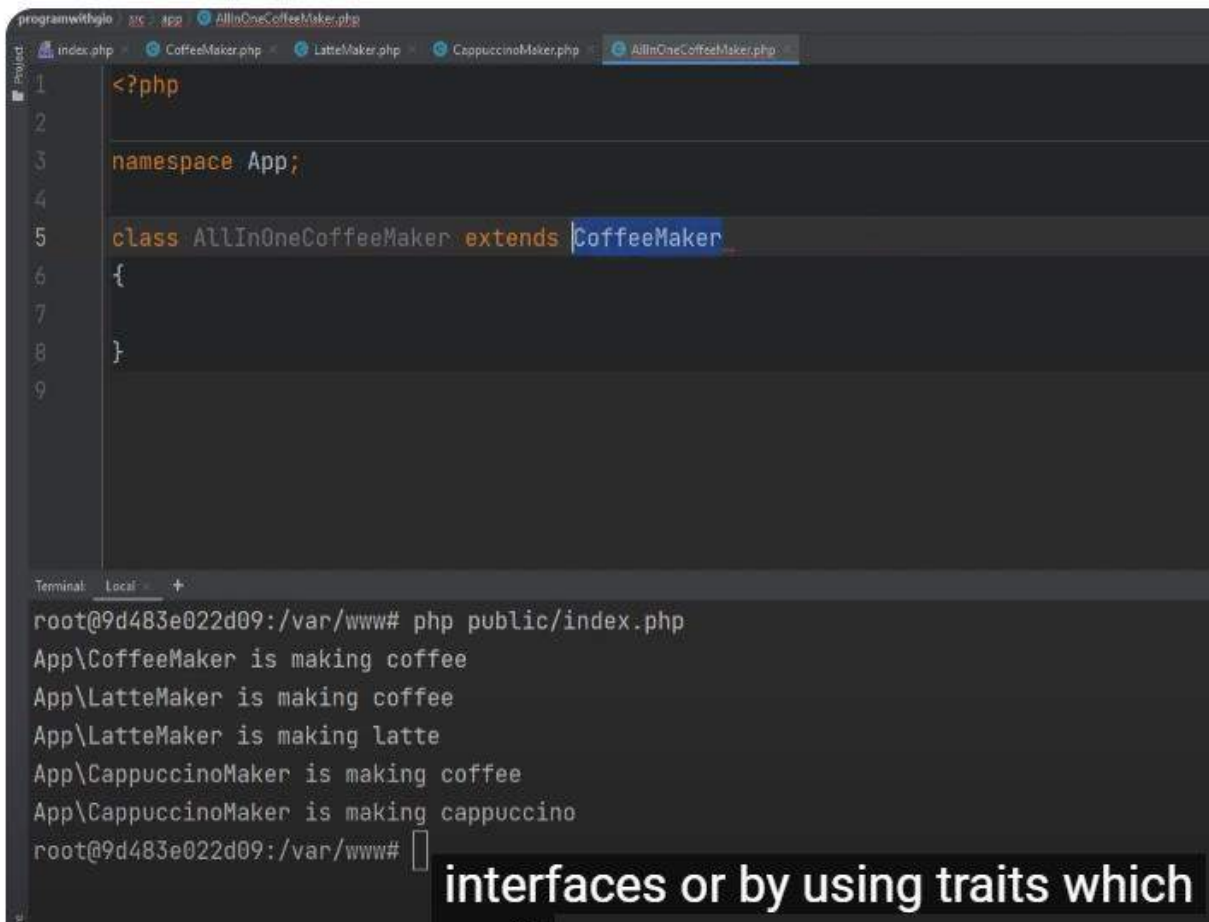
```
root@9d483e022d09:/var/www# php public/index.php
App\CoffeeMaker is making coffee
App\LatteMaker is making coffee
App\LatteMaker is making latte
App\CappuccinoMaker is making coffee
App\CappuccinoMaker is making cappuccino
root@9d483e022d09:/var/www#
```

The code in the editor is as follows:

```
7
8 $latteMaker = new \App\LatteMaker();
9 $latteMaker->makeCoffee();
10 $latteMaker->makeLatte();
11
12 $cappuccinoMaker = new \App\CappuccinoMaker();
13 $cappuccinoMaker->makeCoffee();
14 $cappuccinoMaker->makeCappuccino();
15
16 // $allInOneCoffeeMaker = new \App\AllInOneCoffeeMaker();
17 // $allInOneCoffeeMaker->makeCoffee();
18 // $allInOneCoffeeMaker->makeLatte();
19 // $allInOneCoffeeMaker->makeCappuccino();
```

+ Now, let's uncommment the cappuccino maker. Let's clear this out, run the code and we see that it's working as expected. The Cappuccino Maker is making the regular coffee and it's making cappuccino.

+ Bây giờ, hãy bỏ comment lớp "CappuccinoMaker." Hãy xóa nội dung này và chạy mã, chúng ta thấy rằng nó hoạt động như mong đợi. Lớp "CappuccinoMaker" đang làm cả phê thông thường và nó đang làm cappuccino.



```
<?php
namespace App;

class AllInOneCoffeeMaker extends CoffeeMaker
{
}

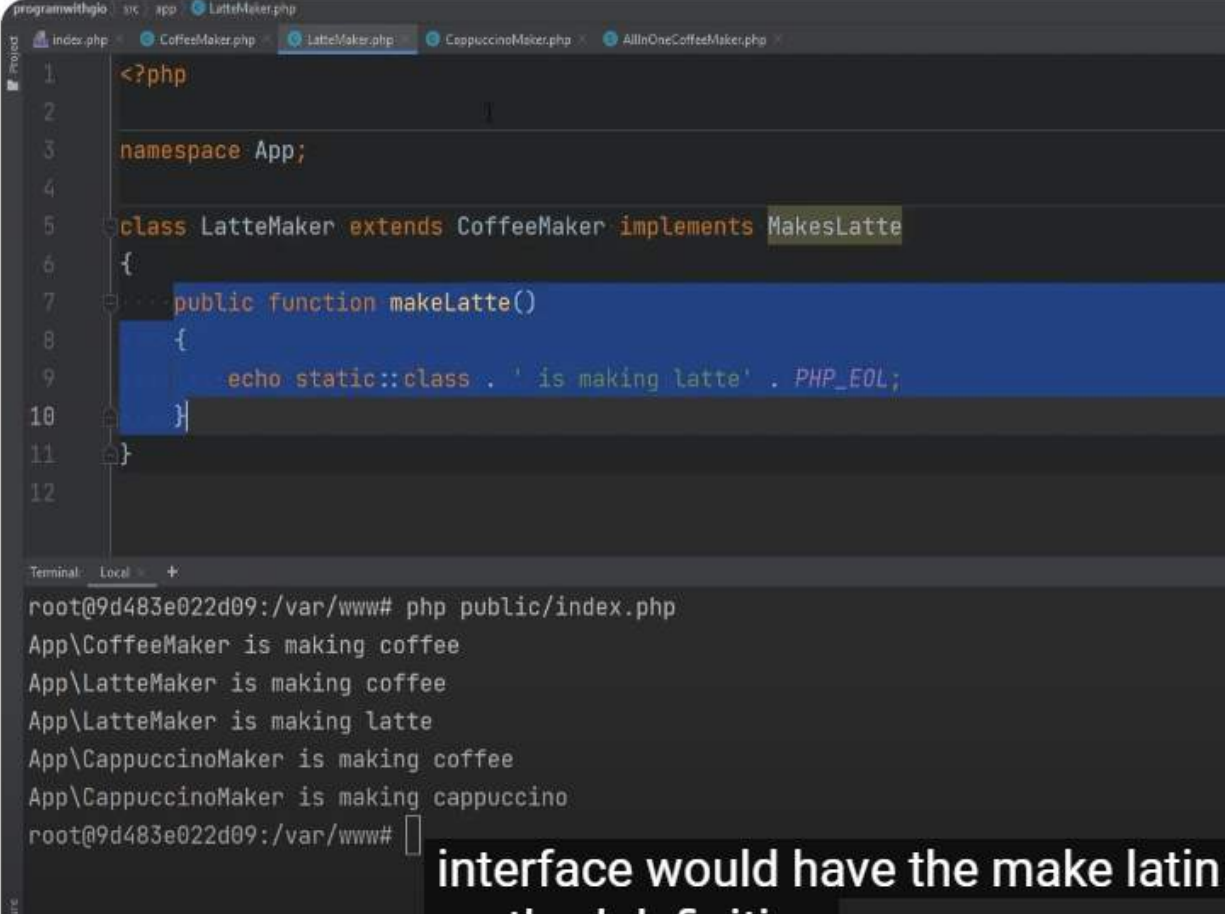
Terminal: Local +
root@9d483e022d09:/var/www# php public/index.php
App\CoffeeMaker is making coffee
App\LatteMaker is making coffee
App\LatteMaker is making latte
App\CappuccinoMaker is making coffee
App\CappuccinoMaker is making cappuccino
root@9d483e022d09:/var/www#
```

interfaces or by using traits which

+Now, how do we get our all-in-one coffee maker class make all kinds of coffee, including the regular coffee, latte, and cappuccino? One way we can do this is by using interfaces. To be honest, that is actually not a bad idea. We could simply extract make latte into an interface and then implement that interface here and define the concrete implementation. We could do the same thing in the all-in-one coffee maker and implement that latte maker interface and then define the concrete implementation here. We would do the same thing for the cappuccino maker. We would implement the cappuccino maker interface and define the concrete implementation here. Also, we could extend the regular coffee maker to be able to make regular coffee. As I mentioned before, in some languages, multiple inheritance is supported and you would be able to do something like extend latte maker, cappuccino maker. But as mentioned before, this is not supported in PHP and instead we are able to implement multiple inheritance by using interfaces or by using traits

+ Bây giờ, làm cách nào để lớp "AllInOneCoffeeMaker" của chúng ta làm tất cả các loại cà phê, bao gồm cà phê thông thường, latte và cappuccino? Một cách chúng ta có thể làm điều này là bằng cách sử dụng interfaces (giao diện). Thực tế, đó không phải là một ý tưởng tồi. Chúng ta có thể đơn giản rút gọn phần "makeLatte" vào một giao diện và sau đó triển khai giao diện đó ở đây và xác định việc triển khai cụ thể. Chúng ta có thể thực hiện tương tự trong lớp

"AllInOneCoffeeMaker" và triển khai giao diện "LatteMaker," sau đó xác định triển khai cụ thể ở đây. Chúng ta có thể làm tương tự cho "CappuccinoMaker." Chúng ta sẽ triển khai giao diện "CappuccinoMaker" và xác định triển khai cụ thể ở đây. Ngoài ra, chúng ta có thể mở rộng lớp "RegularCoffeeMaker" để có khả năng pha cà phê thông thường. Như tôi đã đề cập trước đó, trong một số ngôn ngữ, đa thừa kế được hỗ trợ và bạn có thể làm điều gì đó giống như "extends LatteMaker, CappuccinoMaker." Nhưng như đã đề cập trước đó, điều này không được hỗ trợ trong PHP và thay vào đó, chúng ta có thể triển khai đa thừa kế bằng cách sử dụng giao diện hoặc bằng cách sử dụng traits.



```
<?php

namespace App;

class LatteMaker extends CoffeeMaker implements MakesLatte
{
    public function makeLatte()
    {
        echo static::class . ' is making latte' . PHP_EOL;
    }
}
```

```
root@9d483e022d09:/var/www# php public/index.php
App\CoffeeMaker is making coffee
App\LatteMaker is making coffee
App\LatteMaker is making latte
App\CappuccinoMaker is making coffee
App\CappuccinoMaker is making cappuccino
root@9d483e022d09:/var/www#
```

interface would have the make latin

```
1 <?php
2
3 namespace App;
4
5 class CappuccinoMaker extends CoffeeMaker implements Makes
6 {
7     public function makeCappuccino()
8     {
9         echo static::class . ' is making cappuccino' . PHP_EOL;
10    }
11 }
12
```

Terminal: Local +

```
root@9d483e022d09:/var/www# php public/index.php
App\CoffeeMaker is making coffee
App\LatteMaker is making coffee
App\LatteMaker is making latte
App\CappuccinoMaker is making coffee
App\CappuccinoMaker is making cappuccino
root@9d483e022d09:/var/www#
```

and then this would have implements

+ Which we'll cover in a few minutes. We would essentially be left with something like implement makes latte and makes latte interface would have the make latte method definition and then this would have implement makes cappuccino and that would have the definition of the make cappuccino method.

+ Chúng ta cuối cùng sẽ có một cái gì đó giống như triển khai "makesLatte" và giao diện "makesLatte" sẽ có định nghĩa phương thức "makeLatte," sau đó chúng ta sẽ có triển khai "makesCappuccino" và nó sẽ có định nghĩa phương thức "makeCappuccino."

```
1 <?php
2
3 namespace App;
4
5 class AllInOneCoffeeMaker extends CoffeeMaker implements MakesLatte, MakesCappuccino
6 {
7     public function makeLatte()
8     {
9         echo static::class . ' is making latte' . PHP_EOL;
10    }
11
12     public function makeCappuccino()
13     {
14         echo static::class . ' is making cappuccino' . PHP_EOL;
15    }
16 }
17
```

+ We would essentially need to copy this code and put it in here to provide the implementation for the make cappuccino. The same thing for the latte, we would need to copy this and put it right here.

+Chúng ta sẽ cần sao chép mã này và đặt nó vào đây để cung cấp triển khai cho phương thức "makeCappuccino." Tương tự với phần về latte, chúng ta cần sao chép mã này và đặt nó vào đây.

+ Now, ignore the highlighting here because I actually don't have these interfaces. Just assume that I have the interfaces. Now, this class can make latte, cappuccino, and it can also make regular coffee through inheritance because we're extending the regular coffee maker. The problem with this approach is that if the method body is exactly the same across these classes, then we're duplicating the code. If, however, the concrete implementation were different, meaning that if all-in-one coffee maker was making latte in a different way where you would have slightly different body or implementation of that method, then yeah, the interfaces make sense and it's actually the really good solution because you're coding to interface and you're providing the concrete implementation for each of those methods. In short, interfaces are good and ideal solution when the actual implementation is different. But if we're just duplicating the code, there might be a better way.

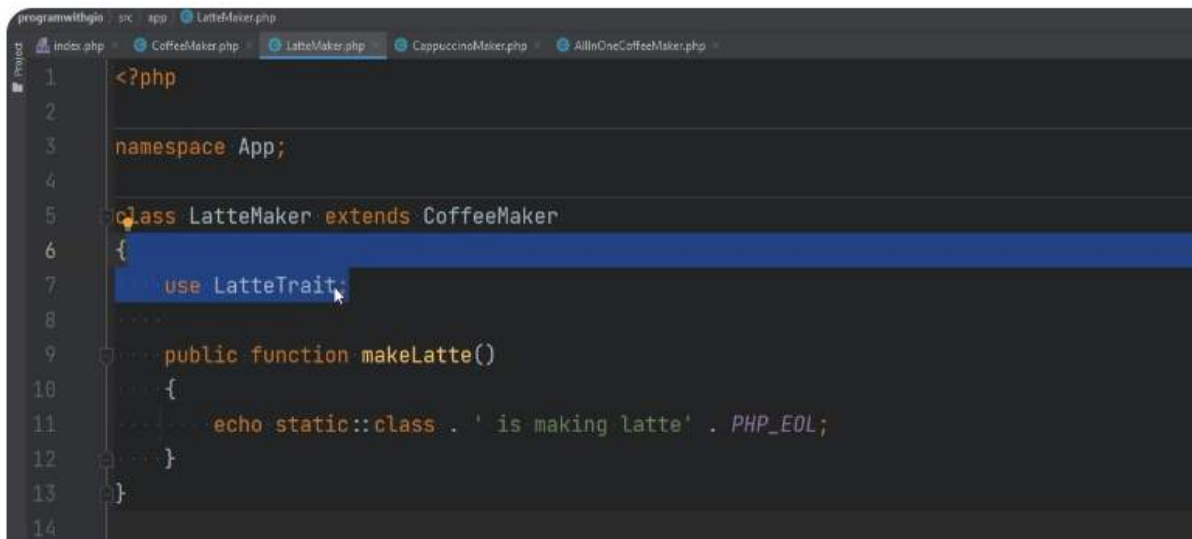
+ Bây giờ, hãy bỏ qua phần được làm nổi bật ở đây vì thực tế tôi không có những giao diện này. Hãy giả định rằng tôi có những giao diện này. Bây giờ, lớp này có thể làm latte, cappuccino và cà phê thông thường thông qua sự kế thừa vì chúng ta đang mở rộng từ lớp "RegularCoffeeMaker." Vấn đề với cách tiếp cận này là nếu nội dung phương thức hoàn toàn giống nhau trong các lớp này, thì chúng ta đang làm trùng mã nguồn. Tuy nhiên, nếu triển khai cụ thể khác nhau, có nghĩa là nếu "AllInOneCoffeeMaker" đang làm latte theo cách khác nhau, có thể bạn sẽ có nội dung hoặc triển khai phương thức đó khác nhau một chút, thì các giao diện sẽ hợp lý và đây thực sự là giải pháp tốt vì bạn đang viết mã theo giao diện và bạn cung cấp triển khai cụ thể cho mỗi phương thức. Tóm lại, giao diện là giải pháp tốt và lý tưởng khi triển khai thực tế khác nhau. Nhưng nếu chúng ta chỉ sao chép mã nguồn, có thể có cách tốt hơn.

```
programwithgio / src / app / CappuccinoMaker.php
index.php > CoffeeMaker.php > LatteMaker.php > CappuccinoMaker.php > AllInOneCoffeeMaker.php
1 <?php
2
3 namespace App;
4
5 class CappuccinoMaker extends CoffeeMaker implements MakesCappuccino
6 {
7     public function makeCappuccino()
8     {
9         echo static::class . ' is making cappuccino' . PHP_EOL;
10    }
11 }
12
```

```
programwithgio / src / app / AllInOneCoffeeMaker.php
index.php > CoffeeMaker.php > LatteMaker.php > CappuccinoMaker.php > AllInOneCoffeeMaker.php
1 <?php
2
3 namespace App;
4
5 class AllInOneCoffeeMaker extends CoffeeMaker implements MakesLatte, MakesCappuccino
6 {
7     public function makeLatte()
```

+This is where traits come in. Traits are mainly used to reduce code duplication and increase code reuse. Let's implement the solution with the traits. Let's get rid of these interfaces from here and from here.

+ Đây là nơi các traits (đặc điểm) xuất hiện. Traits chủ yếu được sử dụng để giảm sự trùng lặp mã nguồn và tăng khả năng tái sử dụng mã. Hãy triển khai giải pháp với các traits. Hãy loại bỏ các giao diện này từ đây và từ đây.

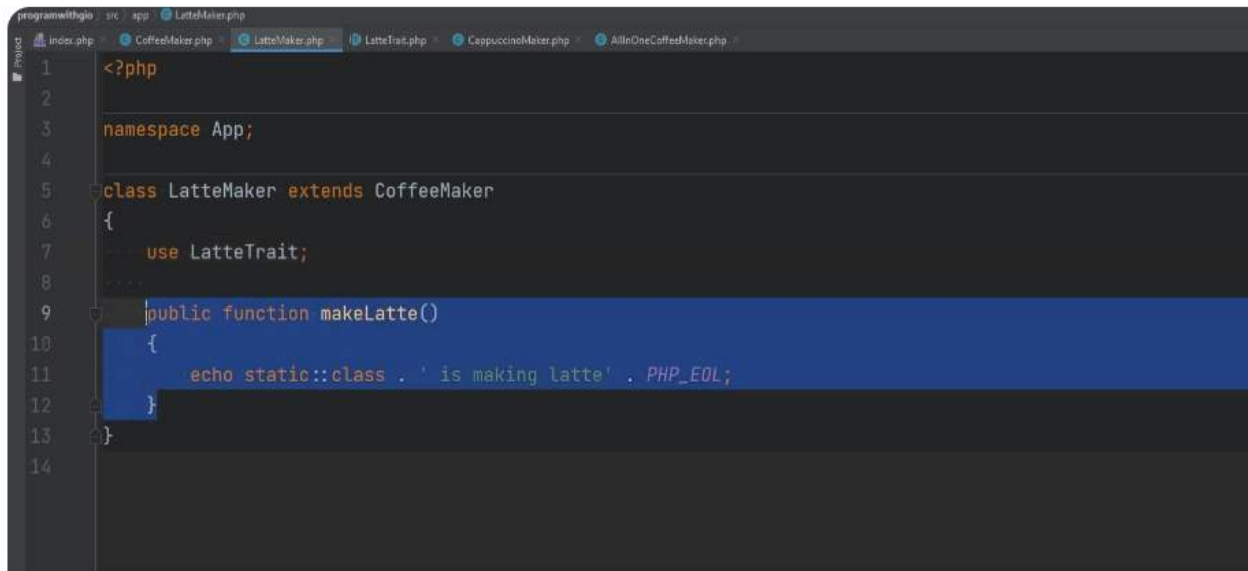
A screenshot of a code editor window showing a PHP file named 'LatteMaker.php'. The code is as follows:

```
1 <?php
2
3 namespace App;
4
5 class LatteMaker extends CoffeeMaker
6 {
7     use LatteTrait;
8     ...
9     public function makeLatte()
10    {
11        echo static::class . ' is making latte' . PHP_EOL;
12    }
13 }
14
```

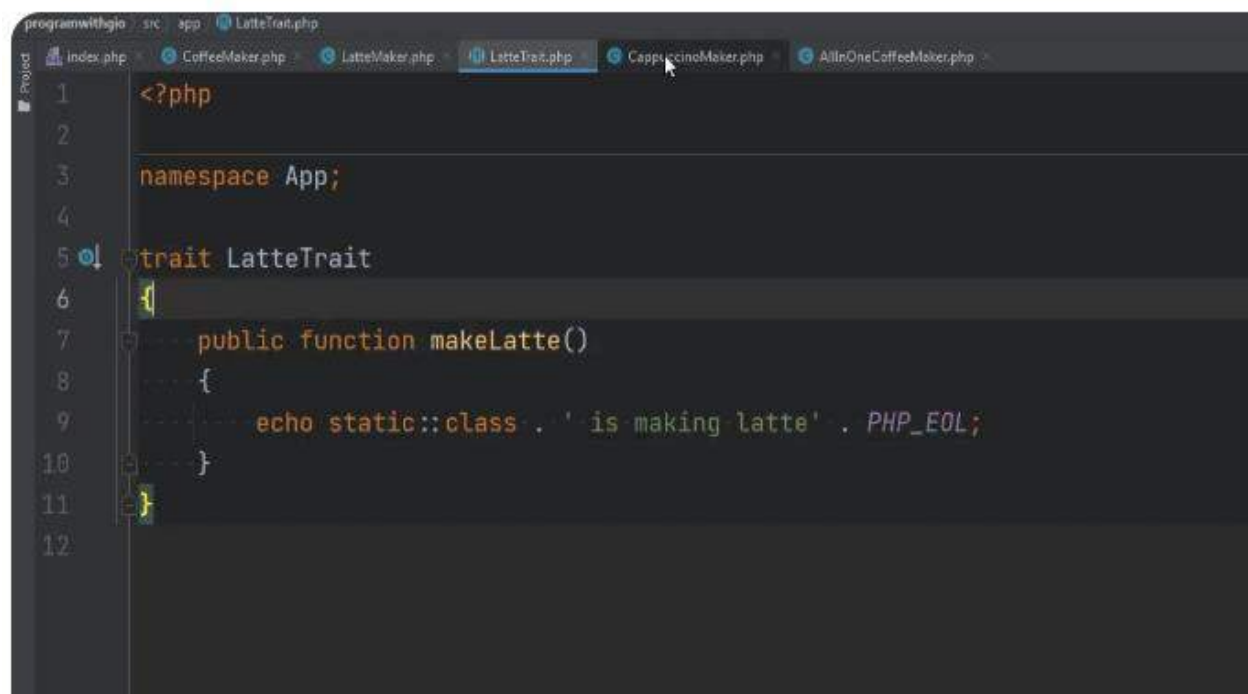
The editor has a dark theme. The 'LatteMaker.php' file is selected in the top tab bar. The code is written in a standard PHP syntax with proper indentation.

+ We can also get rid of these methods for now. Let's simply extract the make latte method into a trait. Let's create a trait called latte trait. The way we create trait is the same way we create classes, but instead of the class keyword, we use the trait keyword. Then to use the trait, we use the use statement. For example, in this latte maker, if we wanted to use a trait called latte trait, we would do something like use latte trait, and this would import the code from the trait into this class at compile time.

+ Chúng ta cũng có thể bỏ đi những phương thức này tạm thời. Hãy đơn giản là trích xuất phương thức "makeLatte" vào một trait. Hãy tạo một trait có tên là "LatteTrait." Cách chúng ta tạo trait tương tự như cách chúng ta tạo lớp, nhưng thay vì từ khóa "class," chúng ta sử dụng từ khóa "trait." Sau đó, để sử dụng trait, chúng ta sử dụng câu lệnh "use." Ví dụ, trong lớp "LatteMaker" này, nếu chúng ta muốn sử dụng một trait có tên là "LatteTrait," chúng ta sẽ làm một cái gì đó giống như "use LatteTrait," và điều này sẽ nhập mã từ trait vào lớp này vào thời điểm biên dịch.



```
1 <?php
2
3 namespace App;
4
5 class LatteMaker extends CoffeeMaker
6 {
7     use LatteTrait;
8
9     public function makeLatte()
10    {
11        echo static::class . ' is making latte' . PHP_EOL;
12    }
13 }
14
```



```
1 <?php
2
3 namespace App;
4
5 trait LatteTrait
6 {
7     public function makeLatte()
8     {
9         echo static::class . ' is making latte' . PHP_EOL;
10    }
11 }
12
```

+ Let's create this trait. I'm going to create that. As you can see, we are using the trait keyword instead of the class keyword. The next thing we need to do is we need to move this code from here into the trait.

+ Hãy tạo trait này. Tôi sẽ tạo ra nó. Như bạn có thể thấy, chúng ta đang sử dụng từ khóa "trait" thay vì từ khóa "class." Điều tiếp theo chúng ta cần làm là chuyển mã từ đây vào trait.

```
programwithgio: src / app / CappuccinoTrait.php
Project index.php CoffeeMaker.php LatteMaker.php LatteTrait.php CappuccinoMaker.php CappuccinoTrait.php AllInOneCoffeeMaker.php
1 <?php
2
3 namespace App;
4
5 trait CappuccinoTrait
6 {
7     public function makeCappuccino()
8     {
9         echo static::class . ' is making cappuccino' . PHP_EOL;
10    }
11 }
12
```

+ Let's do the same thing for a cappuccino. Let's do use cappuccino trait and let's create that trait and let's move this code to that trait. There are many features that trait offers and there are also some rules that you need to be aware of when working with traits and we're going to go over them in a minute.

+ Hãy làm tương tự cho cappuccino. Hãy sử dụng "use CappuccinoTrait" và hãy tạo trait đó và chuyển mã này vào trait đó. Trait cung cấp nhiều tính năng và cũng có một số quy tắc bạn cần phải biết khi làm việc với trait, chúng ta sẽ đi qua chúng trong một phút.

```
programwithgio: src / public / index.php
Project index.php CoffeeMaker.php LatteMaker.php LatteTrait.php CappuccinoMaker.php CappuccinoTrait.php AllInOneCoffeeMaker.php
1 <?php
2
3 require_once __DIR__ . '/../vendor/autoload.php';
4
5 $coffeeMaker = new \App\CoffeeMaker();
6 $coffeeMaker->makeCoffee();
7
8 $latteMaker = new \App\LatteMaker();
9 $latteMaker->makeCoffee();
10 $latteMaker->makeLatte();
11
12 $cappuccinoMaker = new \App\CappuccinoMaker();
13 $cappuccinoMaker->makeCoffee();

```

Terminal: Local +

```
root@9d483e022d09:/var/www# php public/index.php
App\CoffeeMaker is making coffee
App\LatteMaker is making coffee
App\LatteMaker is making latte
App\CappuccinoMaker is making coffee
App\CappuccinoMaker is making cappuccino
root@9d483e022d09:/var/www#
```


+ The first thing you need to know is that you cannot instantiate objects of traits. Instead, you need to use the traits within other traits or within classes. The same way we did it within the LatteMaker and within the CappuccinoMaker. Let's open the terminal and run the code again and we see that everything is still working. We see that it says, CoffeeMaker is making coffee through the regular coffee maker, which does not use any traits. Then we have the Latte maker is making coffee and Latte maker is making latte, and that uses the trait. Same for the cappuccino maker, the make cappuccino method is coming from the trait as well and everything is working.

+ Điều đầu tiên bạn cần biết là bạn không thể khởi tạo đối tượng từ các trait. Thay vào đó, bạn cần sử dụng các trait bên trong các trait khác hoặc bên trong các lớp. Cách làm tương tự như chúng ta đã làm trong LatteMaker và CappuccinoMaker. Hãy mở terminal và chạy mã nguồn lại và chúng ta thấy rằng mọi thứ vẫn hoạt động. Chúng ta thấy nó nói, CoffeeMaker đang làm cà phê thông qua máy làm cà phê thông thường, không sử dụng bất kỳ trait nào. Sau đó, chúng ta có Latte maker đang làm cà phê và Latte maker đang làm latte, và đó sử dụng trait. Tương tự cho cappuccino maker, phương thức make cappuccino cũng đến từ trait và mọi thứ đều hoạt động.

```
10 $latteMaker->makeLatte();
11
12 $cappuccinoMaker = new \App\CappuccinoMaker();
13 $cappuccinoMaker->makeCoffee();
14 $cappuccinoMaker->makeCappuccino();
15
16 $allInOneCoffeeMaker = new \App\AllInOneCoffeeMaker();
17 $allInOneCoffeeMaker->makeCoffee();
18 $allInOneCoffeeMaker->makeLatte();
19 $allInOneCoffeeMaker->makeCappuccino();
20
21
22
```

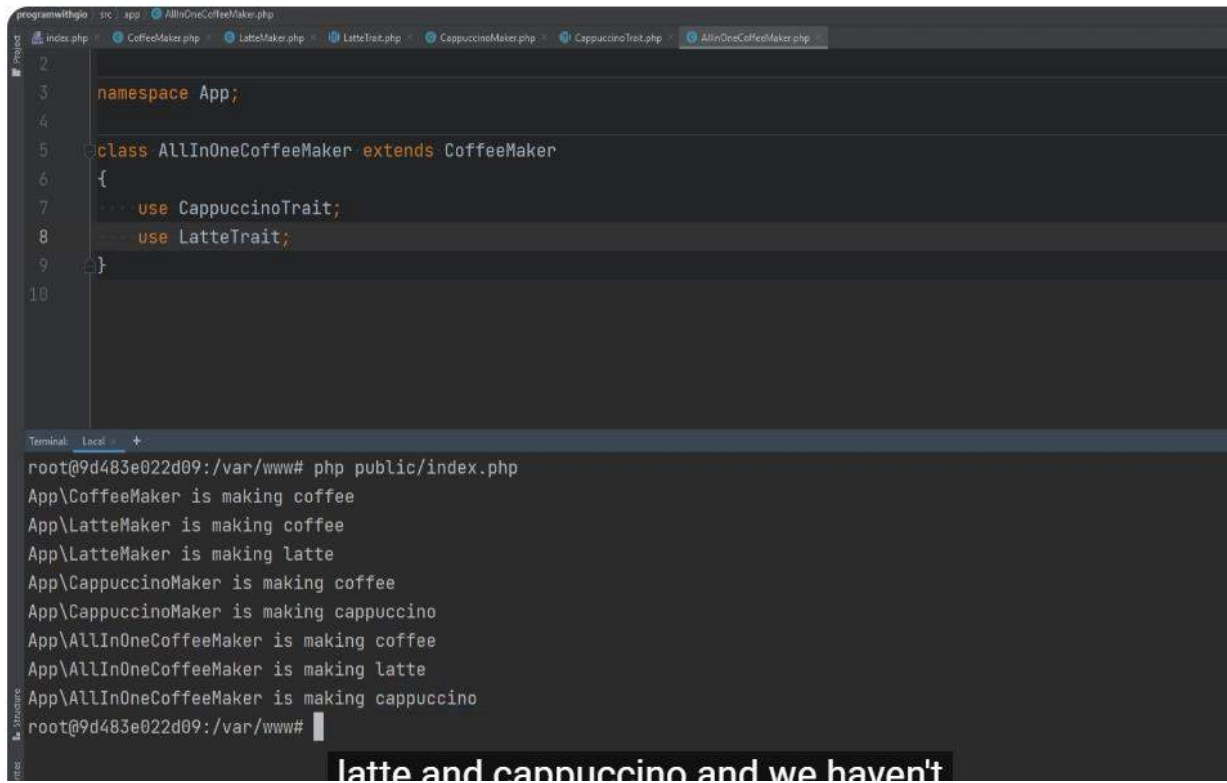
```
Terminal Local x +
root@9d483e022d09:/var/www# php public/index.php
App\CoffeeMaker is making coffee
App\LatteMaker is making coffee
App\LatteMaker is making latte
App\CappuccinoMaker is making coffee
App\CappuccinoMaker is making cappuccino
root@9d483e022d09:/var/www#
```

don't have to make latte and makeup

+ Now, let's uncomment this. Well, we don't have the make latte and make cappuccino method yet because we haven't used those traits in this class. Let's go ahead and pull in those traits here as well. You can pull in multiple traits by comma separating them. Or if you're following the PSR 12, then you would need to put them on its own line.

+ Bây giờ, hãy gỡ bỏ phần chú thích này. Tuy nhiên, chúng ta chưa có phương thức make latte và make cappuccino vì chúng ta chưa sử dụng các trait đó trong lớp này. Hãy tiếp tục sử dụng

các trait đó ở đây. Bạn có thể sử dụng nhiều trait bằng cách phân tách chúng bằng dấu phẩy. Hoặc nếu bạn tuân theo PSR 12, thì bạn cần đặt chúng trên một dòng riêng.



```
2
3 namespace App;
4
5 class AllInOneCoffeeMaker extends CoffeeMaker
6 {
7     use CappuccinoTrait;
8     use LatteTrait;
9 }
10
```

```
Terminal: Local +
root@9d483e022d09:/var/www# php public/index.php
App\CoffeeMaker is making coffee
App\LatteMaker is making coffee
App\LatteMaker is making latte
App\CappuccinoMaker is making coffee
App\CappuccinoMaker is making cappuccino
App\AllInOneCoffeeMaker is making coffee
App\AllInOneCoffeeMaker is making latte
App\AllInOneCoffeeMaker is making cappuccino
root@9d483e022d09:/var/www#
```

latte and cappuccino and we haven't

+We could do use latte trait, cappuccino trait. My ID is underlining this because I have it set to follow PSR 12. If I follow that, it will split it into its own line. Now, if I run the code, we see that everything is still working.

+Chúng ta có thể làm như sau: `use LatteTrait, CappuccinoTrait`. Công cụ lập trình của tôi đang gạch chân dòng này vì tôi đã cài đặt nó để tuân theo PSR 12. Nếu tôi tuân theo PSR 12, nó sẽ chia thành các dòng riêng biệt. Bây giờ, nếu tôi chạy mã, chúng ta thấy rằng mọi thứ vẫn hoạt động.

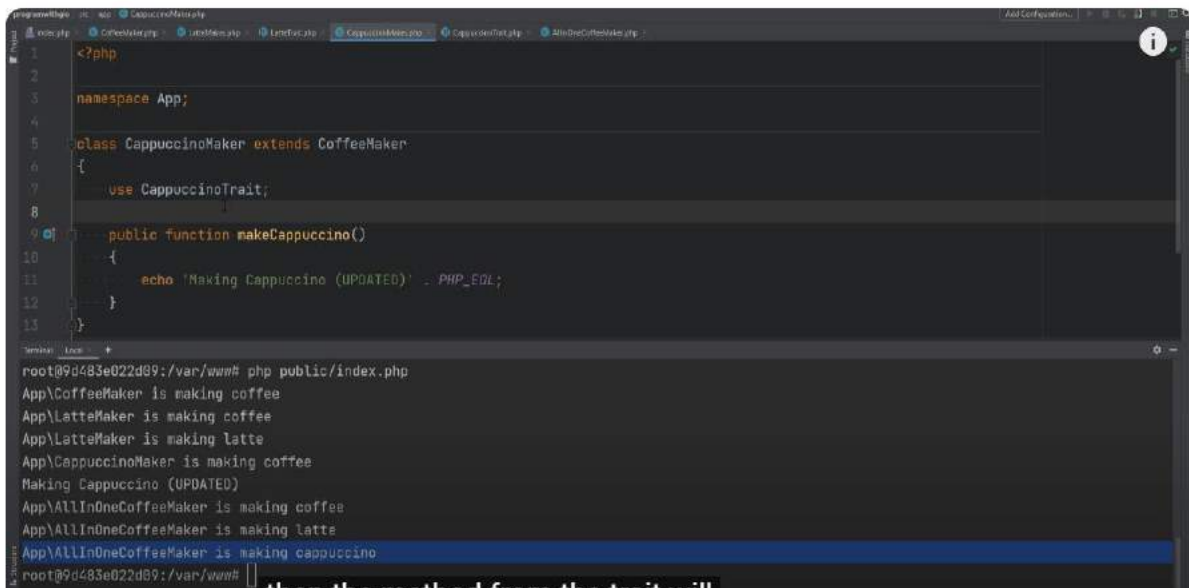
```
programwithgio src / app / CappuccinoMaker.php
index.php < CoffeeMaker.php < LatteMaker.php < LatteTrait.php < CappuccinoMaker.php < CappuccinoTrait.php < AllInOneCoffeeMaker.php
1 <?php
2
3 namespace App;
4
5 class CappuccinoMaker extends CoffeeMaker
6 {
7     use CappuccinoTrait;
8     use LatteTrait;
9 }
10
```

```
programwithgio src / app / CappuccinoMaker.php
index.php < CoffeeMaker.php < LatteMaker.php < LatteTrait.php < CappuccinoMaker.php < CappuccinoTrait.php < AllInOneCoffeeMaker.php
1 <?php
2
3 namespace App;
4
5 class CappuccinoMaker extends CoffeeMaker
6 {
7     use CappuccinoTrait;
8 }
9
```

+ Now we have that all-in-one coffee maker that is able to make regular coffee, latte, and cappuccino. We haven't duplicated any code. Now, if down the road, Cappuccino Maker was able to make latte as well, we could simply pull in the latte trait here and we would get the functionality to make latte. How does this actually work? Think of traits as copy and paste. It simply takes the code that's written in the trait and pastes in the class that uses the trait at compile time. Now, let's talk about some of the rules and the things that you need to know about when working with traits. The first thing we're going to talk about is the precedence and the method overwriting. You are able to redefine the method that is defined in the trait. If the class that uses the trait defines the same method, then when that method is called, the method defined directly in the class takes precedence and it will be used instead of the method from the trait.

+ Bây giờ chúng ta có máy pha cà phê all-in-one có thể làm cà phê thông thường, latte và cappuccino. Chúng ta không sao chép mã nguồn nào. Bây giờ, nếu sau này, Cappuccino Maker có thể làm latte, chúng ta có thể đơn giản là kéo vào trait latte ở đây và chúng ta sẽ có khả năng làm latte. Làm thế nào thực sự nó hoạt động? Hãy nghĩ về trait như việc sao chép và dán. Nó chỉ cần lấy mã nguồn đã viết trong trait và dán vào lớp sử dụng trait vào thời điểm biên dịch. Bây giờ, hãy nói về một số quy tắc và điều bạn cần biết khi làm việc với trait. Điều đầu tiên chúng ta sẽ nói về là ưu tiên và việc định nghĩa lại phương thức. Bạn có thể định nghĩa lại phương thức đã

được định nghĩa trong trait. Nếu lớp sử dụng trait định nghĩa phương thức giống như phương thức trong trait, khi phương thức đó được gọi, phương thức được định nghĩa trực tiếp trong lớp sẽ có ưu tiên và nó sẽ được sử dụng thay vì phương thức từ trait.



```
<?php
namespace App;

class CappuccinoMaker extends CoffeeMaker
{
    use CappuccinoTrait;

    public function makeCappuccino()
    {
        echo 'Making Cappuccino (UPDATED)' . PHP_EOL;
    }
}
```

```
root@9d483e022d89:/var/www# php public/index.php
App\CoffeeMaker is making coffee
App\LatteMaker is making coffee
App\LatteMaker is making latte
App\CappuccinoMaker is making coffee
Making Cappuccino (UPDATED)
App\AllInOneCoffeeMaker is making coffee
App\AllInOneCoffeeMaker is making latte
App\AllInOneCoffeeMaker is making cappuccino
root@9d483e022d89:/var/www#
```

+ For example, Cappuccino Maker is pulling in Cappuccino trait, which has the method Make Cappuccino. If we were to redefine the same method and simply echo out Making Cappuccino, Updated and run the code, we see that when cappuccino is being made, it is using the redefined method here. But when we're using it within the All-in-One Coffee Maker, it's still using the trait method because we haven't redefined it in this class. We are redefining in the Cappuccino Maker class. However, if the method was defined in the base class but is not overridden in the child class that uses the trait with the same method definition, then the method from the trait will override the method defined in the base class.

+ Ví dụ, Cappuccino Maker sử dụng Cappuccino trait, trong đó có phương thức Make Cappuccino. Nếu chúng ta định nghĩa lại cùng một phương thức và chỉ đơn giản in ra Making Cappuccino, Updated và chạy mã, chúng ta thấy rằng khi cappuccino được làm, nó sử dụng phương thức được định nghĩa lại ở đây. Nhưng khi chúng ta sử dụng nó trong All-in-One Coffee Maker, nó vẫn sử dụng phương thức từ trait vì chúng ta chưa định nghĩa lại nó trong lớp này. Chúng ta đang định nghĩa lại trong lớp Cappuccino Maker. Tuy nhiên, nếu phương thức đã được định nghĩa trong lớp cơ sở nhưng không được ghi đè trong lớp con sử dụng trait với cùng định nghĩa phương thức, thì phương thức từ trait sẽ ghi đè phương thức được định nghĩa trong lớp cơ sở.

```
3 namespace App;
4
5 trait CappuccinoTrait
6 {
7     public function makeCappuccino()
8     {
9         echo static::class . ' is making cappuccino' . PHP_EOL;
10    }
11
12    public function makeCoffee()
13    {
14        echo 'Making Coffee (UPDATED)' . PHP_EOL;
15    }
16 }
```

```
root@9d483e022d09:/var/www# php public/index.php
App\CoffeeMaker is making coffee
App\LatteMaker is making coffee
App\LatteMaker is making latte
App\CappuccinoMaker is making coffee
Making Cappuccino (UPDATED)
App\AllInOneCoffeeMaker is making coffee
App\AllInOneCoffeeMaker is making latte
App\AllInOneCoffeeMaker is making cappuccino
root@9d483e022d09:/var/www#
```

+ For example, let's get rid of this. In the Cappuccino Maker, we are extending the Coffee Maker. Coffee Maker defines the method Make Coffee. If for whatever reason we had the Make Coffee method defined in the trait that is used in the same class, so if we were to define a method called here, public function, Make Coffee, and echo out making coffee updated, then this method would take the higher precedence and this is the method that would get executed instead of the method on the base class.

+ Ví dụ, hãy loại bỏ phần này. Trong Cappuccino Maker, chúng ta đang mở rộng Coffee Maker. Coffee Maker định nghĩa phương thức Make Coffee. Nếu vì bất kỳ lý do nào đó chúng ta đã định nghĩa lại phương thức Make Coffee trong trait được sử dụng trong cùng một lớp, vì vậy nếu chúng ta định nghĩa một phương thức có tên là ở đây, public function, Make Coffee, và in ra making coffee updated, thì phương thức này sẽ có ưu tiên cao hơn và đây là phương thức sẽ được thực thi thay vì phương thức trên lớp cơ sở.

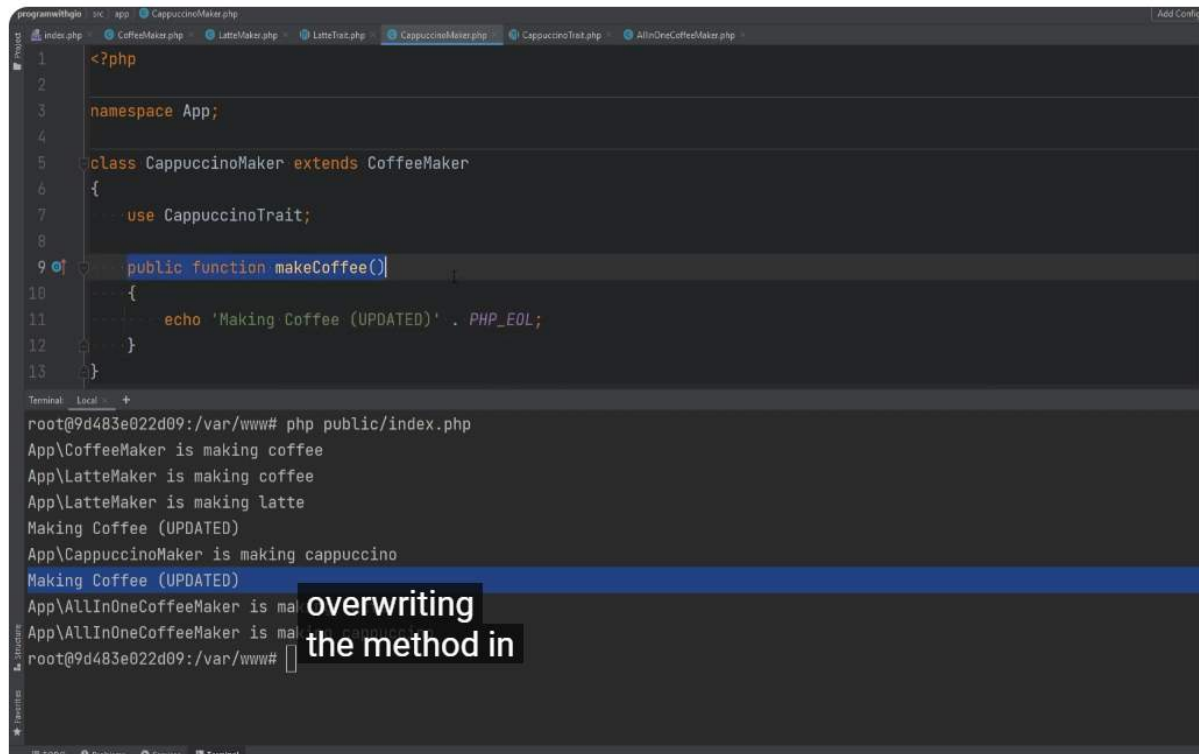
```
1 <?php
2
3 namespace App;
4
5 class CappuccinoMaker extends CoffeeMaker
6 {
7     use CappuccinoTrait;
8 }
9
```

```
Terminal: Local +
root@9d483e022d09:/var/www# php public/index.php
App\CoffeeMaker is making coffee
App\LatteMaker is making coffee
App\LatteMaker is making latte
Making Coffee (UPDATED)
App\CappuccinoMaker is making
Making Coffee (UPDATED)
App\AllInOneCoffeeMaker is making latte
App\AllInOneCoffeeMaker is making cappuccino
root@9d483e022d09:/var/www#
```

base class so if we run the
we see that when the make

+ If we run the code now, we see that when the make coffee is called on the cappuccino maker object or the all-in-one maker object, it is using the updated method from the trait because both of the classes use that cappuccino trait, which overwrites that method. This makes sense because we said that the way traits work is that it copies and paste the code into the underlying class that uses that trait.

+ Nếu chúng ta chạy mã ngay bây giờ, chúng ta thấy rằng khi phương thức make coffee được gọi trên đối tượng cappuccino maker hoặc đối tượng all-in-one maker, nó sử dụng phương thức đã được cập nhật từ trait vì cả hai lớp đều sử dụng trait cappuccino, ghi đè phương thức đó. Điều này có lý vì chúng ta nói cách traits hoạt động là nó sao chép và dán mã vào lớp cơ sở sử dụng trait đó.



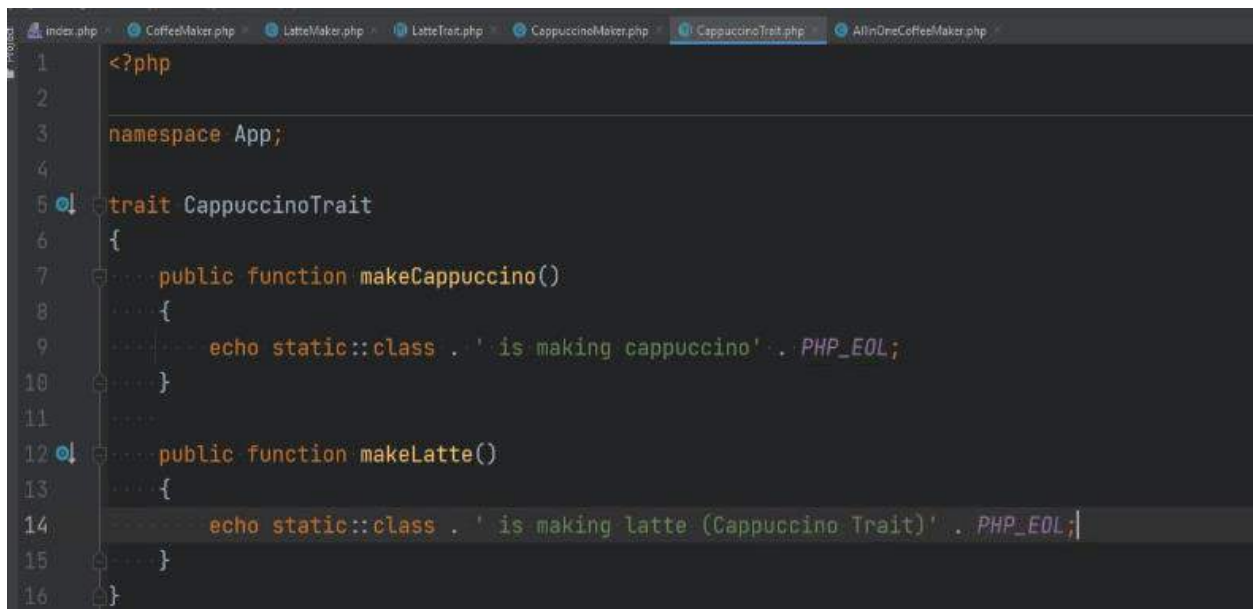
```
1 <?php
2
3 namespace App;
4
5 class CappuccinoMaker extends CoffeeMaker
6 {
7     use CappuccinoTrait;
8
9     public function makeCoffee()
10     {
11         echo 'Making Coffee (UPDATED)' . PHP_EOL;
12     }
13 }
```

```
Terminal Local +
root@9d483e022d09:/var/www# php public/index.php
App\CoffeeMaker is making coffee
App\LatteMaker is making coffee
App\LatteMaker is making latte
Making Coffee (UPDATED)
App\CappuccinoMaker is making cappuccino
Making Coffee (UPDATED)
App\AllInOneCoffeeMaker is making cappuccino
App\AllInOneCoffeeMaker is making cappuccino
root@9d483e022d09:/var/www#
```

overwriting
the method in

+ If it essentially is copying this code and is pasting it into the cappuccino maker here, then this is just a regular rule of inheritance. We're essentially overriding the method in the coffee maker with the method that's defined in the child class. To sum this up, the precedence is the class method that is defined directly on the class, then the trait method that's defined in the trait, and then the base method, which is defined in the parent class if there is any. Let's get rid of this and let's talk about the conflict resolution. The similar problem that exists in the multiple inheritance exists with the traits as well. That happens when the two methods are conflicting because of the same name.

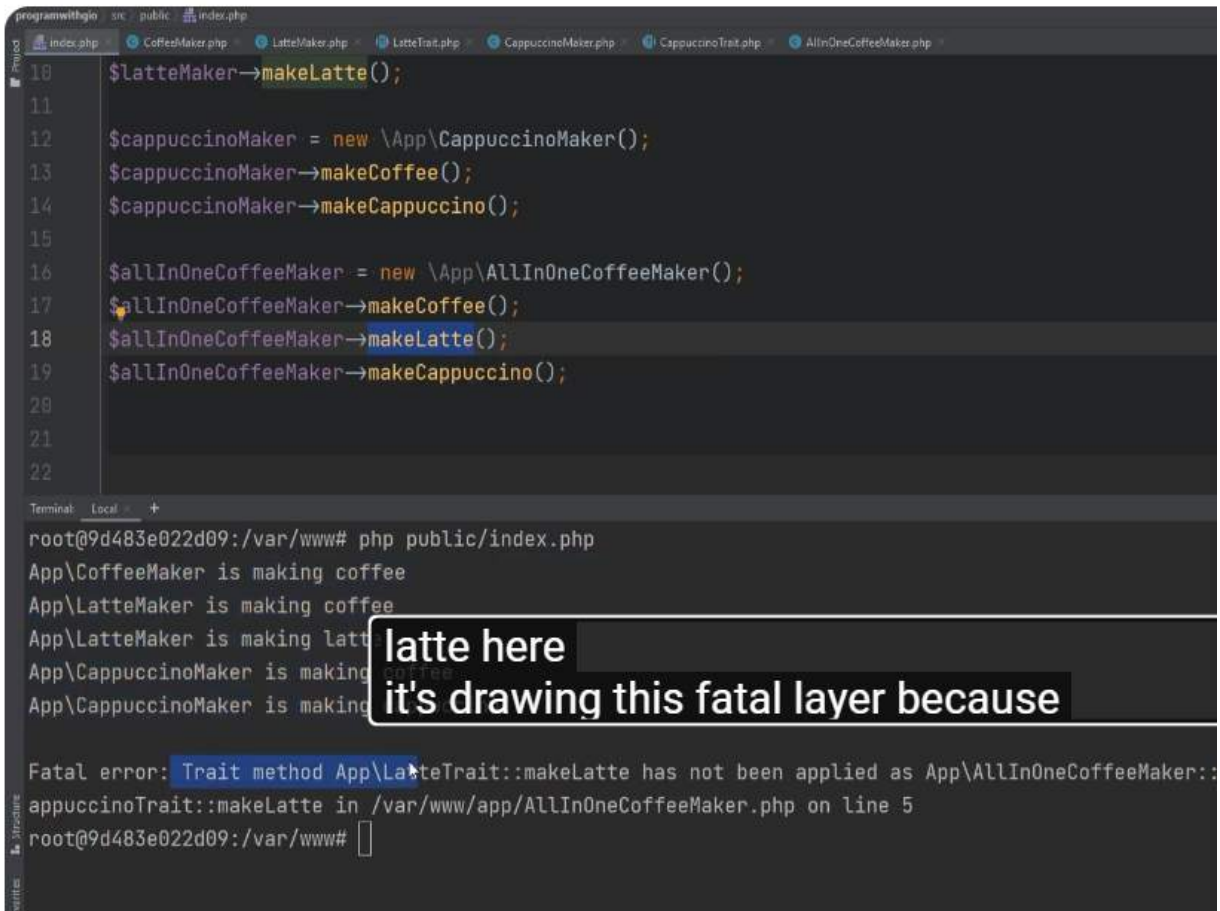
+ Nếu nó thực sự sao chép mã này và dán vào cappuccino maker ở đây, thì đây chỉ là một quy tắc thừa kế thông thường. Thực chất, chúng ta đang ghi đè phương thức trong coffee maker bằng phương thức được định nghĩa trong lớp con. Để tổng kết, ưu tiên là phương thức của lớp được định nghĩa trực tiếp trên lớp, sau đó là phương thức của trait được định nghĩa trong trait và sau đó là phương thức cơ sở, được định nghĩa trong lớp cha nếu có. Hãy xóa điều này và chúng ta sẽ nói về việc giải quyết xung đột. Vấn đề tương tự tồn tại trong kế thừa đa hình với traits cũng tồn tại. Điều đó xảy ra khi hai phương thức xung đột vì cùng tên.

A screenshot of a code editor with a dark theme. The editor shows a PHP file named 'CappuccinoTrait.php'. The code defines a trait 'CappuccinoTrait' with two public methods: 'makeCappuccino()' and 'makeLatte()'. The 'makeCappuccino()' method prints 'is making cappuccino'. The 'makeLatte()' method prints 'is making latte (Cappuccino Trait)'. The code is as follows:

```
1 <?php
2
3 namespace App;
4
5 trait CappuccinoTrait
6 {
7     public function makeCappuccino()
8     {
9         echo static::class . ' is making cappuccino' . PHP_EOL;
10    }
11
12    public function makeLatte()
13    {
14        echo static::class . ' is making latte (Cappuccino Trait)' . PHP_EOL;
15    }
16 }
```

+ For example, let's say the cappuccino trait also had a method to make latte. Let's define public function make latte, and let's say that it's printing the same thing that the class is making latte, but we'll say that it's making from the cappuccino trait. That way we can identify which method runs.

+Ví dụ, giả sử trait cappuccino cũng có một phương thức để pha latte. Hãy định nghĩa một hàm công khai là `makeLatte`, và chúng ta hãy nói rằng nó in ra điều này, là lớp đang làm latte, nhưng chúng ta sẽ nói rằng nó đang làm từ cappuccino trait. Điều này giúp chúng ta xác định được phương thức nào đang chạy.



```
10 $latteMaker->makeLatte();
11
12 $cappuccinoMaker = new \App\CappuccinoMaker();
13 $cappuccinoMaker->makeCoffee();
14 $cappuccinoMaker->makeCappuccino();
15
16 $allInOneCoffeeMaker = new \App\AllInOneCoffeeMaker();
17 $allInOneCoffeeMaker->makeCoffee();
18 $allInOneCoffeeMaker->makeLatte();
19 $allInOneCoffeeMaker->makeCappuccino();
20
21
22
```

```
Terminal: Local +
root@9d483e022d09:/var/www# php public/index.php
App\CoffeeMaker is making coffee
App\LatteMaker is making coffee
App\LatteMaker is making latte
App\CappuccinoMaker is making coffee
App\CappuccinoMaker is making cappuccino
Fatal error: Trait method App\LatteTrait::makeLatte has not been applied as App\AllInOneCoffeeMaker::
appuccinoTrait::makeLatte in /var/www/app/AllInOneCoffeeMaker.php on line 5
root@9d483e022d09:/var/www#
```

latte here
it's drawing this fatal layer because

+Now, let's try to run this code. I'm going to clear this up and I'm going to run the code and everything is running up until the all-in-one coffee maker, which is right here. When we're calling make latte here, it's throwing this fatal error because it does not know which make latte method to call. If we're going to the all-in-one coffee maker, we see that we are using both cappuccino trait and the latte trait. Both cappuccino trait and latte trait have a method with the same name called Make Latte. This can actually be easily solved in PHP by using instead of Operator. Instead of operator, let's you specify which method to run when there is a conflict between the method names.

+ Bây giờ, hãy thử chạy mã này. Tôi sẽ xóa điều này và chạy mã và mọi thứ đều chạy cho đến khi đến lượt all-in-one coffee maker, nằm ở đây. Khi chúng ta gọi `makeLatte` ở đây, nó sẽ ném ra lỗi nghiêm trọng vì không biết phải gọi phương thức `makeLatte` nào. Nếu chúng ta xem xét all-in-one coffee maker, chúng ta thấy rằng chúng ta đang sử dụng cả `cappuccino trait` và `latte trait`. Cả `cappuccino trait` và `latte trait` đều có một phương thức có tên là `makeLatte`. Vấn đề này có thể dễ dàng được giải quyết trong PHP bằng cách sử dụng toán tử `insteadof`. Toán tử `insteadof` cho phép bạn chỉ định phương thức nào sẽ được chạy khi có xung đột giữa tên phương thức.


```
programwithgio src > app > AllInOneCoffeeMaker.php
index.php > CoffeeMaker.php > LatteMaker.php > LatteTrait.php > CappuccinoMaker.php > CappuccinoTrait.php > AllInOneCoffeeMaker.php
1 <?php
2
3 namespace App;
4
5 class AllInOneCoffeeMaker extends CoffeeMaker
6 {
7     use CappuccinoTrait;
8     use LatteTrait {
9         LatteTrait::makeLatte insteadof CappuccinoTrait;
10    }
11 }
12
Terminal: Local +
root@9d483e022d09:/var/www#
```

+ For example, let's say that we wanted to run the Make Latte from the latte trait and not from the cappuccino trait in this all-in-one coffee maker class. Then we would do something like this. Instead of the semicolon here, we would open and close the curly braces and say that use the latte trait, make latte instead of cappuccino trait.

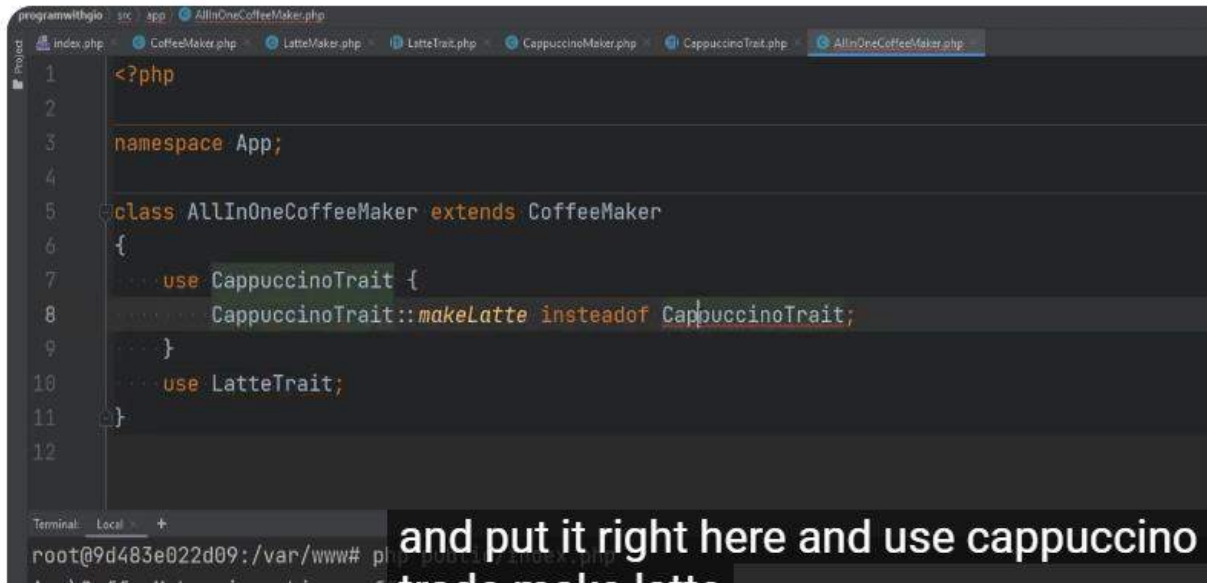
+ Ví dụ, giả sử chúng ta muốn chạy `makeLatte` từ `latte trait` và không phải từ `cappuccino trait` trong lớp `all-in-one coffee maker`. Sau đó, chúng ta sẽ làm như sau. Thay vì dấu chấm phẩy ở đây, chúng ta sẽ mở và đóng dấu ngoặc nhọn và nói rằng sử dụng `latte trait`, `makeLatte` thay vì `cappuccino trait`.

```
index.php > CoffeeMaker.php > LatteMaker.php > LatteTrait.php > CappuccinoMaker.php > CappuccinoTrait.php > AllInOneCoffeeMaker.php
1 <?php
2
3 namespace App;
4
5 class AllInOneCoffeeMaker extends CoffeeMaker
6 {
7     use CappuccinoTrait;
8     use LatteTrait {
9         LatteTrait::makeLatte insteadof CappuccinoTrait;
10    }
11 }
12
Terminal: Local +
root@9d483e022d09:/var/www# pl
App\CoffeeMaker is making coffee
App\LatteMaker is making coffee
App\LatteMaker is making latte
App\CappuccinoMaker is making coffee
App\CappuccinoMaker is making cappuccino
App\AllInOneCoffeeMaker is making coffee
App\AllInOneCoffeeMaker is making latte
App\AllInOneCoffeeMaker is making cappuccino
root@9d483e022d09:/var/www#
```

latte trade if we wanted to use the make latte

+ Let's clear this up and run the code again and we see that everything is working and it's calling the Latte from the Latte trait.

+ Hãy làm sạch và chạy mã nguồn lại, chúng ta sẽ thấy mọi thứ hoạt động và nó gọi Make Latte từ Latte trait.

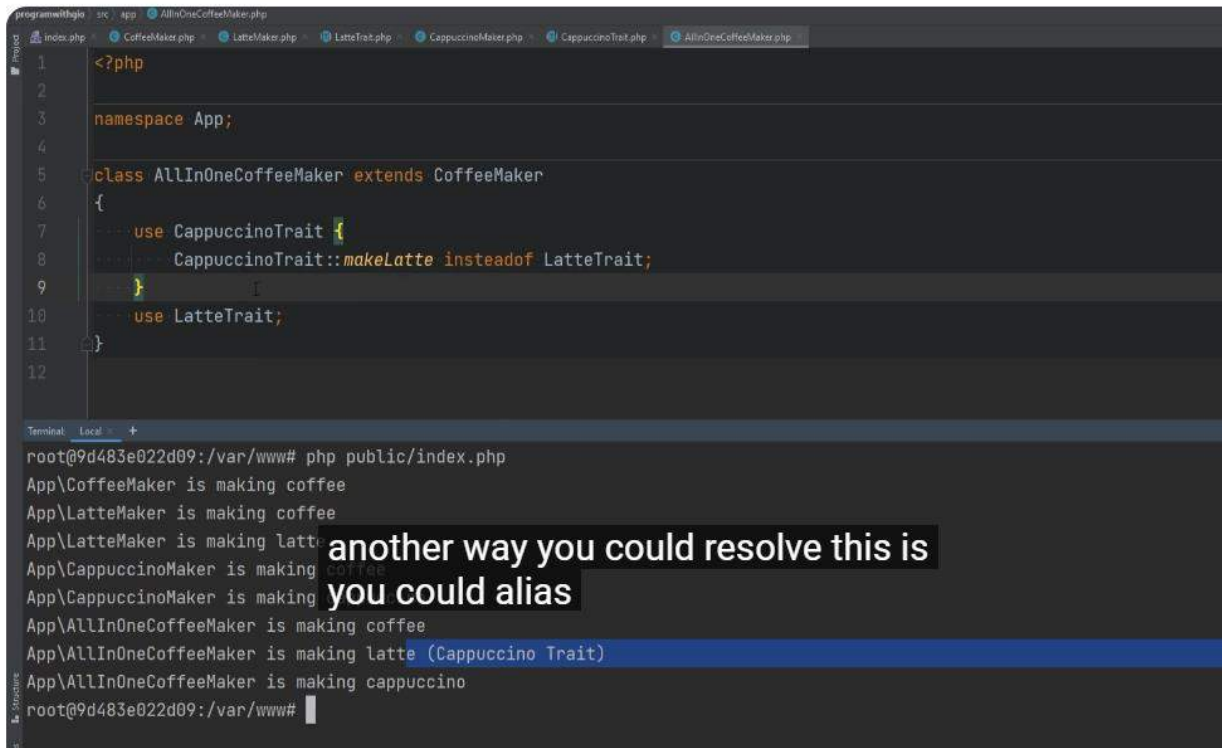


```
1 <?php
2
3 namespace App;
4
5 class AllInOneCoffeeMaker extends CoffeeMaker
6 {
7     use CappuccinoTrait {
8         CappuccinoTrait::makeLatte insteadof CappuccinoTrait;
9     }
10    use LatteTrait;
11 }
12
```

Terminal: Local +
root@9d483e022d09: /var/www# php -d error_reporting=E_ALL index.php
and put it right here and use cappuccino

+ If we wanted to use the Make Latte from the Cappuccino trait, we could simply move this up and put it right here and use Cappuccino trait, Make Latte instead of Latte trait. Now it's going to call the Make Latte on the Cappuccino trait.

+ Nếu chúng ta muốn sử dụng Make Latte từ Cappuccino trait, chúng ta có thể đơn giản di chuyển nó lên đây và đặt nó ngay dưới đây và sử dụng Cappuccino trait, Make Latte thay vì Latte trait. Bây giờ nó sẽ gọi Make Latte trên Cappuccino trait.



```
<?php
namespace App;

class AllInOneCoffeeMaker extends CoffeeMaker
{
    use CappuccinoTrait {
        CappuccinoTrait::makeLatte insteadof LatteTrait;
    }
    use LatteTrait;
}
```

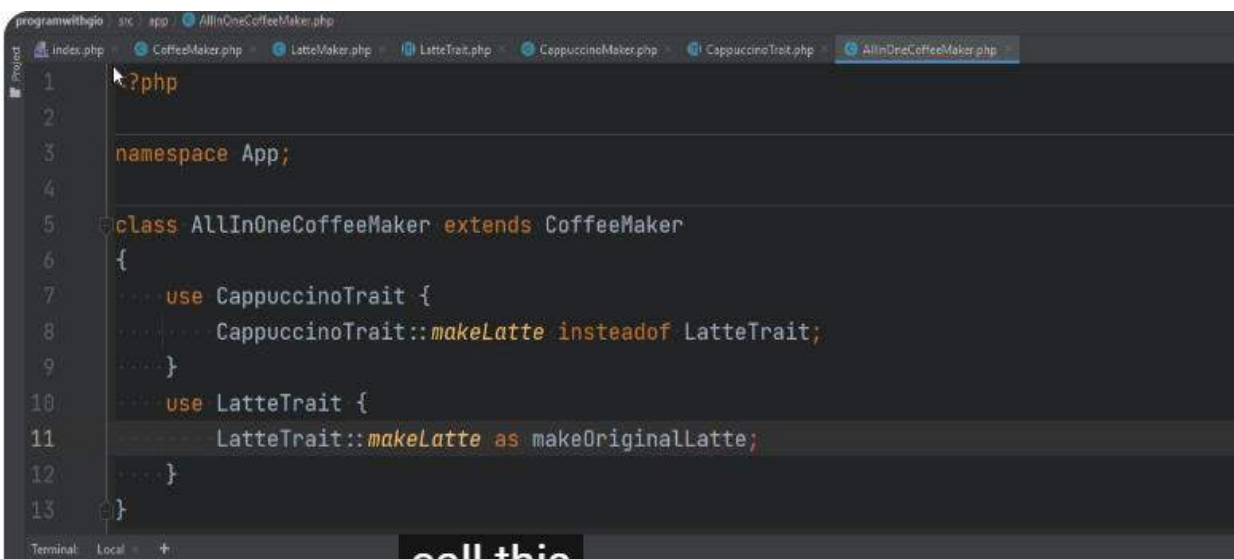
Terminal: Local

```
root@9d483e022d09:/var/www# php public/index.php
App\CoffeeMaker is making coffee
App\LatteMaker is making coffee
App\LatteMaker is making latte
App\CappuccinoMaker is making coffee
App\CappuccinoMaker is making cappuccino
App\AllInOneCoffeeMaker is making coffee
App\AllInOneCoffeeMaker is making latte (Cappuccino Trait)
App\AllInOneCoffeeMaker is making cappuccino
root@9d483e022d09:/var/www#
```

another way you could resolve this is
you could alias

+ If we clear this up and run the code again, we see that it's making latte from the cappuccino trait. Another way you could resolve this is you could alias the methods to a different method name. Let's say that you wanted to make the latte from the latte trait as well, but with a different name.

+ Nếu chúng ta làm sạch nó lên và chạy mã nguồn lại, chúng ta thấy nó đang làm latte từ Cappuccino trait. Một cách khác để giải quyết vấn đề này là bạn có thể đặt tên alias cho các phương thức với một tên phương thức khác. Hãy nói rằng bạn muốn làm latte từ Latte trait cũng, nhưng với một tên khác.



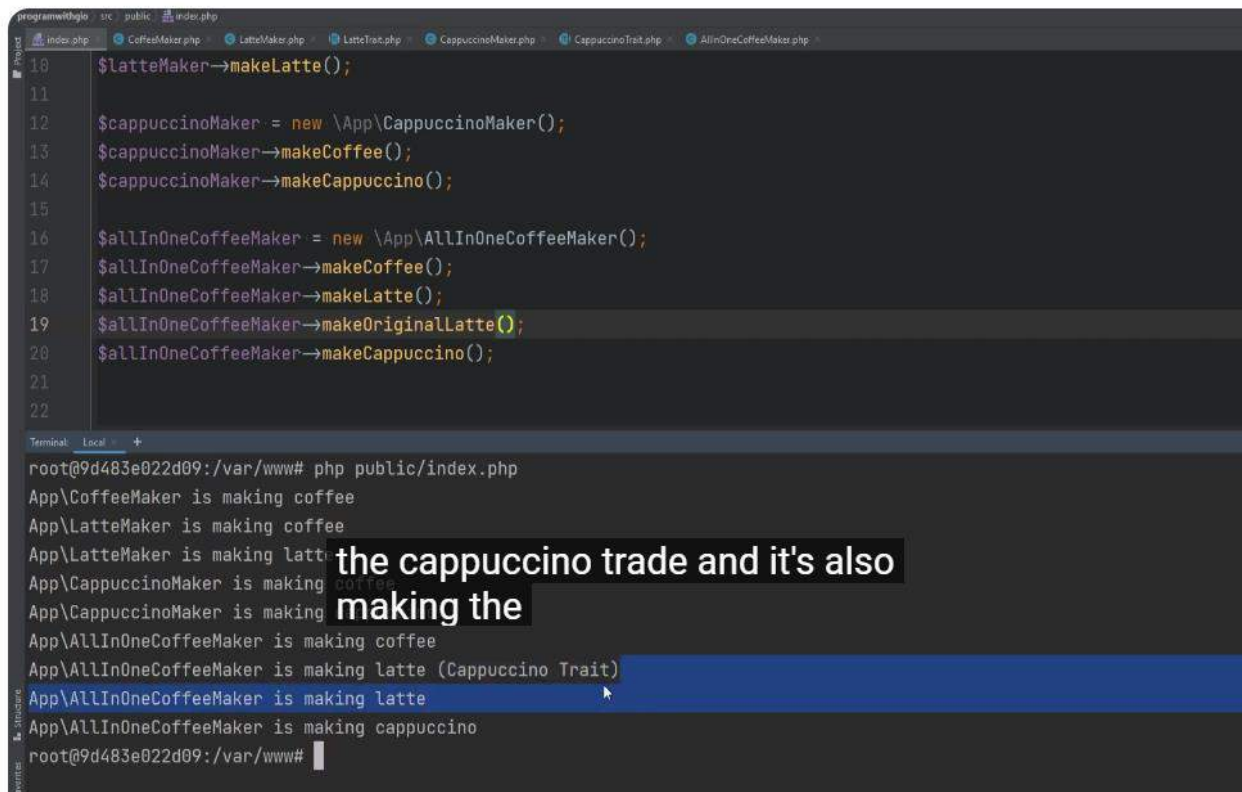
```
<?php
namespace App;

class AllInOneCoffeeMaker extends CoffeeMaker
{
    use CappuccinoTrait {
        CappuccinoTrait::makeLatte insteadof LatteTrait;
    }
    use LatteTrait {
        LatteTrait::makeLatte as makeOriginalLatte;
    }
}
```

call this

+ You could do something like this. You could open Curly braces and you could do latte trait, make latte, and then use the as operator and specify the alias name. We could call this make original latte.

+ Bạn có thể làm như sau. Bạn có thể mở dấu ngoặc nhọn và bạn có thể làm latte trait, make latte, và sau đó sử dụng toán tử as và chỉ định tên alias. Chúng ta có thể gọi điều này là make latte ban đầu.



The screenshot shows a code editor with the following PHP code in `index.php`:

```
10 $latteMaker->makeLatte();
11
12 $cappuccinoMaker = new \App\CappuccinoMaker();
13 $cappuccinoMaker->makeCoffee();
14 $cappuccinoMaker->makeCappuccino();
15
16 $allInOneCoffeeMaker = new \App\AllInOneCoffeeMaker();
17 $allInOneCoffeeMaker->makeCoffee();
18 $allInOneCoffeeMaker->makeLatte();
19 $allInOneCoffeeMaker->makeOriginalLatte();
20 $allInOneCoffeeMaker->makeCappuccino();
21
22
```

Below the code editor is a terminal window showing the output of running `php public/index.php`:

```
root@9d483e022d09:/var/www# php public/index.php
App\CoffeeMaker is making coffee
App\LatteMaker is making coffee
App\LatteMaker is making latte
App\CappuccinoMaker is making coffee
App\CappuccinoMaker is making cappuccino
App\AllInOneCoffeeMaker is making coffee
App\AllInOneCoffeeMaker is making latte (Cappuccino Trait)
App\AllInOneCoffeeMaker is making latte
App\AllInOneCoffeeMaker is making cappuccino
root@9d483e022d09:/var/www#
```

A text overlay on the terminal output reads: "the cappuccino trait and it's also making the" with a cursor pointing to the line "App\AllInOneCoffeeMaker is making latte (Cappuccino Trait)".

+ Now if we go to index. Php, we can call make latte, but we could also call make original latte. If we clear this up and run the code, we see that it's making the latte from the cappuccino trait, and it's also making the regular original latte from the latte trait.

+ Bây giờ nếu chúng ta vào index. Php, chúng ta có thể gọi make latte, nhưng chúng ta cũng có thể gọi make latte ban đầu. Nếu chúng ta làm sạch điều này lên và chạy mã nguồn lại, chúng ta thấy nó đang làm latte từ Cappuccino trait và nó cũng đang làm latte bình thường từ latte trait.

```
programwithgio src / app AllInOneCoffeeMaker.php
index.php CoffeeMaker.php LatteMaker.php LatteTrait.php CappuccinoMaker.php CappuccinoTrait.php AllInOneCoffeeMaker.php
1 <?php
2
3 namespace App;
4
5 class AllInOneCoffeeMaker extends CoffeeMaker
6 {
7     use CappuccinoTrait;
8     use LatteTrait;
9 }
10
```

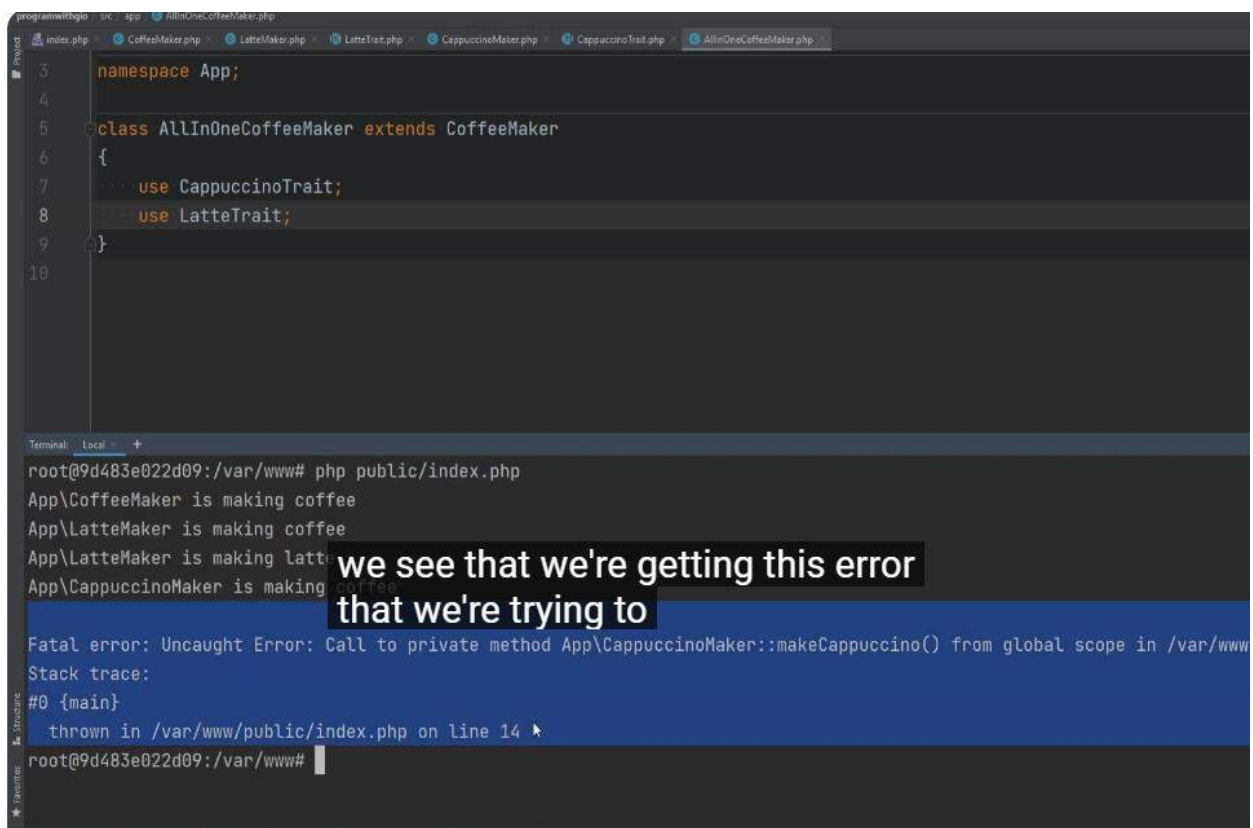
```
index.php CoffeeMaker.php LatteMaker.php LatteTrait.php CappuccinoMaker.php CappuccinoTrait.php AllInOneCoffeeMaker.php
1 <?php
2
3 namespace App;
4
5 trait CappuccinoTrait
6 {
7     private function makeCappuccino()
8     {
9         echo static::class . ' is making cappuccino' . PHP_EOL;
10     }
11 }
12
```

```
programwithgio src / app AllInOneCoffeeMaker.php
index.php CoffeeMaker.php LatteMaker.php LatteTrait.php CappuccinoMaker.php CappuccinoTrait.php AllInOneCoffeeMaker.php
1 <?php
2
3 namespace App;
4
5 class AllInOneCoffeeMaker extends CoffeeMaker
6 {
7     use CappuccinoTrait;
8     use LatteTrait;
9
10     public function foo()
11     {
12         $this->makeCappuccino();
13     }
14 }
Terminal: Local +
root@9d483e022d09:/var/www#
```

make cappuccino even though the make

+ You could also change the visibility of the methods. Let's get rid of these from here for now, and let's get rid of this as well. Let's remove the make latte from the cappuccino trait. We keep things simple. Let's say that make cappuccino method is private. Now, because this is private, you can still access this method within the class that uses the trait. For example, if we had some method here called FU, we could still access the method make cappuccino, even though the make cappuccino method is private. However, you would not be able to access this method outside of this class.

+ Bạn cũng có thể thay đổi tính riêng tư của các phương thức. Hãy xóa chúng khỏi đây trước tiên, và hãy xóa điều này luôn. Hãy loại bỏ make latte từ cappuccino trait. Chúng ta giữ mọi thứ đơn giản. Hãy nói rằng phương thức make cappuccino là riêng tư. Bây giờ, vì đây là riêng tư, bạn vẫn có thể truy cập phương thức này trong lớp sử dụng trait. Ví dụ, nếu chúng ta có một phương thức ở đây gọi là FU, chúng ta vẫn có thể truy cập phương thức make cappuccino, mặc dù phương thức make cappuccino là riêng tư. Tuy nhiên, bạn sẽ không thể truy cập phương thức này bên ngoài lớp này



The screenshot shows a code editor with a PHP file named `AllInOneCoffeeMaker.php`. The code defines a namespace `App` and a class `AllInOneCoffeeMaker` that extends `CoffeeMaker`. It uses two traits: `CappuccinoTrait` and `LatteTrait`. Below the code, a terminal window shows the output of a PHP script. The output displays the following messages:

```
App\CoffeeMaker is making coffee
App\LatteMaker is making coffee
App\LatteMaker is making latte
App\CappuccinoMaker is making coffee
```

After these messages, a fatal error is thrown:

```
Fatal error: Uncaught Error: Call to private method App\CappuccinoMaker::makeCappuccino() from global scope in /var/www/public/index.php:14
Stack trace:
#0 {main}
thrown in /var/www/public/index.php on line 14
```

A text overlay on the terminal output reads: "we see that we're getting this error that we're trying to".

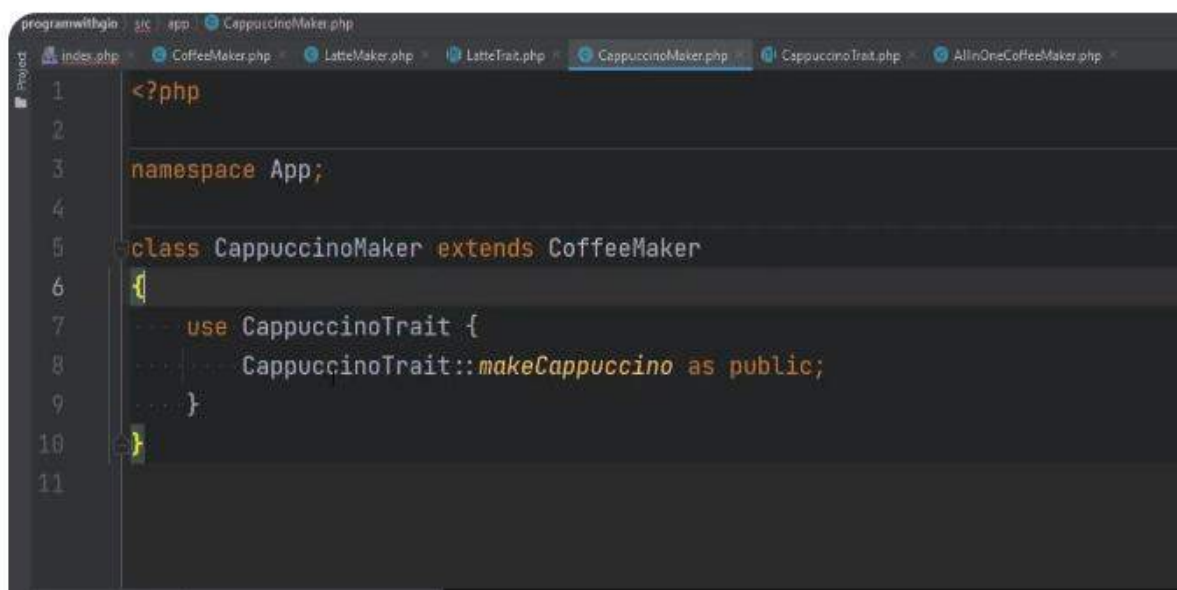

```
programwithgio src public index.php
index.php CoffeeMaker.php LatteMaker.php LatteTrait.php CappuccinoMaker.php CappuccinoTrait.php AllInOneCoffeeMaker.php
10 $latteMaker→makeLatte();
11
12 $cappuccinoMaker = new \App\CappuccinoMaker();
13 $cappuccinoMaker→makeCoffee();
14 $cappuccinoMaker→makeCappuccino();
15
16 $allInOneCoffeeMaker = new \App\AllInOneCoffeeMaker();
17 $allInOneCoffeeMaker→makeCoffee();
18 $allInOneCoffeeMaker→makeLatte();
19 $allInOneCoffeeMaker→makeCappuccino();
20
```

```
Terminal - root
root@9d483e822d09:/var/www# php public/index.php
App\CoffeeMaker is making coffee
App\LatteMaker is making coffee
App\LatteMaker is making latte
App\CappuccinoMaker is making coffee

Fatal error: Uncaught Error: Call to private method App\CappuccinoMaker::makeCappuccino() from global scope in /var/www/public/index.php:14
Stack trace:
#0 {main}
  thrown in /var/www/public/index.php on line 14
root@9d483e822d09:/var/www#
```

+ If we clear the code here and run it again, we see that we're getting this error that we're trying to call a method that is private. Because in index. Phb, we're trying to call make cappuccino right here, and we're trying to call the make cappuccino right here. It's actually failing right here. It doesn't even get to this. We need to remove this make original latte from here. How can we fix this? We can actually change the visibility of make cappuccino when using the trait. Let's do that on the cappuccino maker first.

+ Nếu chúng ta làm sạch mã ở đây và chạy lại, chúng ta thấy chúng ta đang nhận được lỗi rằng chúng ta đang cố gọi một phương thức là riêng tư. Bởi vì trong index. Php, chúng ta đang cố gọi make cappuccino ngay ở đây và chúng ta đang cố gọi make cappuccino ngay ở đây. Nó thậm chí không chạy đến đây. Chúng ta cần loại bỏ make latte ban đầu này khỏi đây. Làm thế nào để chúng ta sửa điều này? Chúng ta có thể thay đổi tính riêng tư của make cappuccino khi sử dụng trait. Hãy làm điều này trên cappuccino maker trước.



```
1 <?php
2
3 namespace App;
4
5 class CappuccinoMaker extends CoffeeMaker
6 {
7     use CappuccinoTrait {
8         CappuccinoTrait::makeCappuccino as public;
9     }
10 }
11
```

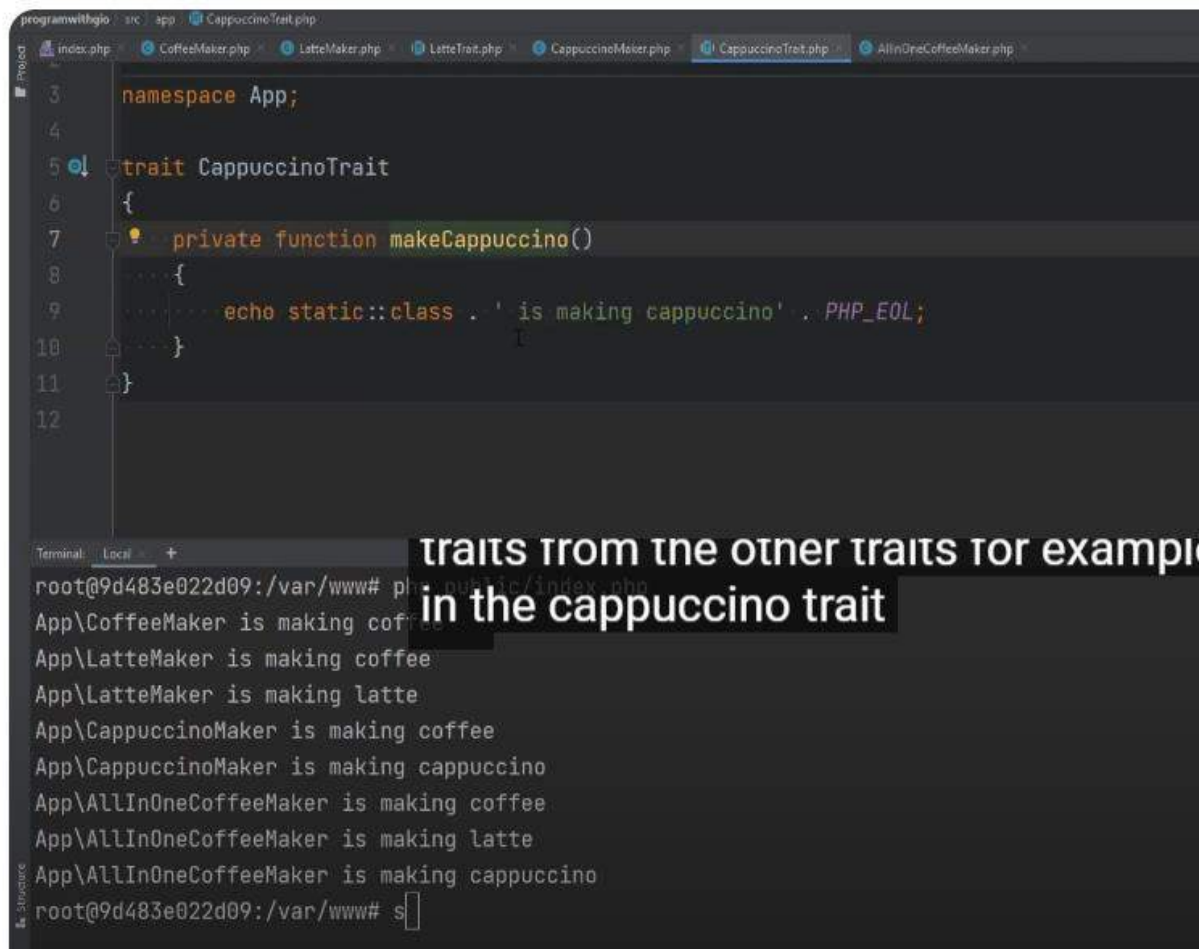
+Let's go here. Instead of the semicolon, we're going to open close Calibrases, and we can use the as operator the same way we did it to alias method name to a different name. But instead of the method name, we can actually choose the visibility keyword, which is either private, public, or protected. We could do a cappuccino trait, make cappuccino as public. Now, make cappuccino method will become public when it's pasted into the cappuccino maker. The make cappuccino method on the cappuccino maker will become public, even though it is set to private inside the cappuccino tray. If I go to index. Php, we see that underlining is gone from here.

+ Hãy làm điều này. Thay vì dấu chấm phẩy, chúng ta sẽ mở đóng dấu Calibrases và chúng ta có thể sử dụng toán tử as theo cách chúng ta đã làm để tạo tên phương thức thành tên khác. Nhưng thay vì tên phương thức, chúng ta thực sự có thể chọn từ khóa tính riêng tư, đó là riêng tư, công cộng hoặc bảo vệ. Chúng ta có thể làm như vậy với cappuccino trait, make cappuccino as public. Bây giờ, phương thức make cappuccino sẽ trở thành công cộng khi nó được dán vào cappuccino maker. Phương thức make cappuccino trên cappuccino maker sẽ trở thành công cộng, mặc dù nó được đặt thành riêng tư trong cappuccino tray. Nếu tôi mở index. Php, chúng ta thấy dòng chân trùng ra khỏi đây.

```
Terminal: Local ~ +
root@9d483e022d09:/var/www# php public/index.php
App\CoffeeMaker is making coffee
App\LatteMaker is making coffee
App\LatteMaker is making latte
App\CappuccinoMaker is making coffee
App\CappuccinoMaker is making cappuccino
App\AllInOneCoffeeMaker is making coffee
App\AllInOneCoffeeMaker is making latte
App\AllInOneCoffeeMaker is making cappuccino
root@9d483e022d09:/var/www#
```

+ If I run the code again, we see that it's working for the cappuccino maker right here, but it's failing on the all-in-one coffee maker because we haven't done the same thing on the all-in-one coffee maker class. Let's open Cappuccino Maker and let's copy this and let's put it in the all-in-one coffee maker instead of this. Let's run the code again and everything is working. No more errors. As you can see, you could make private or protected methods become public. As powerful as this sounds, this actually is not one of the best features that I think trades has.

+ Nếu tôi chạy mã lại, chúng ta thấy nó hoạt động cho cappuccino maker ngay ở đây, nhưng nó gặp sự cố trên all-in-one coffee maker vì chúng ta chưa thực hiện điều tương tự trên lớp all-in-one coffee maker. Hãy mở Cappuccino Maker và hãy sao chép điều này và hãy đặt nó vào all-in-one coffee maker thay vì điều này. Hãy chạy mã lại và không còn lỗi nữa. Như bạn có thể thấy, bạn có thể biến phương thức riêng tư hoặc bảo vệ thành công cộng. Mặc dù có vẻ mạnh mẽ, điều này thực sự không phải là một trong những tính năng tốt nhất mà tôi nghĩ rằng trade có.



The screenshot shows a code editor with a file explorer on the left. The active file is `CappuccinoTrait.php`. The code defines a namespace `App` and a trait `CappuccinoTrait` with a private method `makeCappuccino()` that echoes a message. Below the code, a terminal window shows the output of a script, demonstrating that the trait is used by `CoffeeMaker`, `LatteMaker`, and `AllInOneCoffeeMaker` classes.

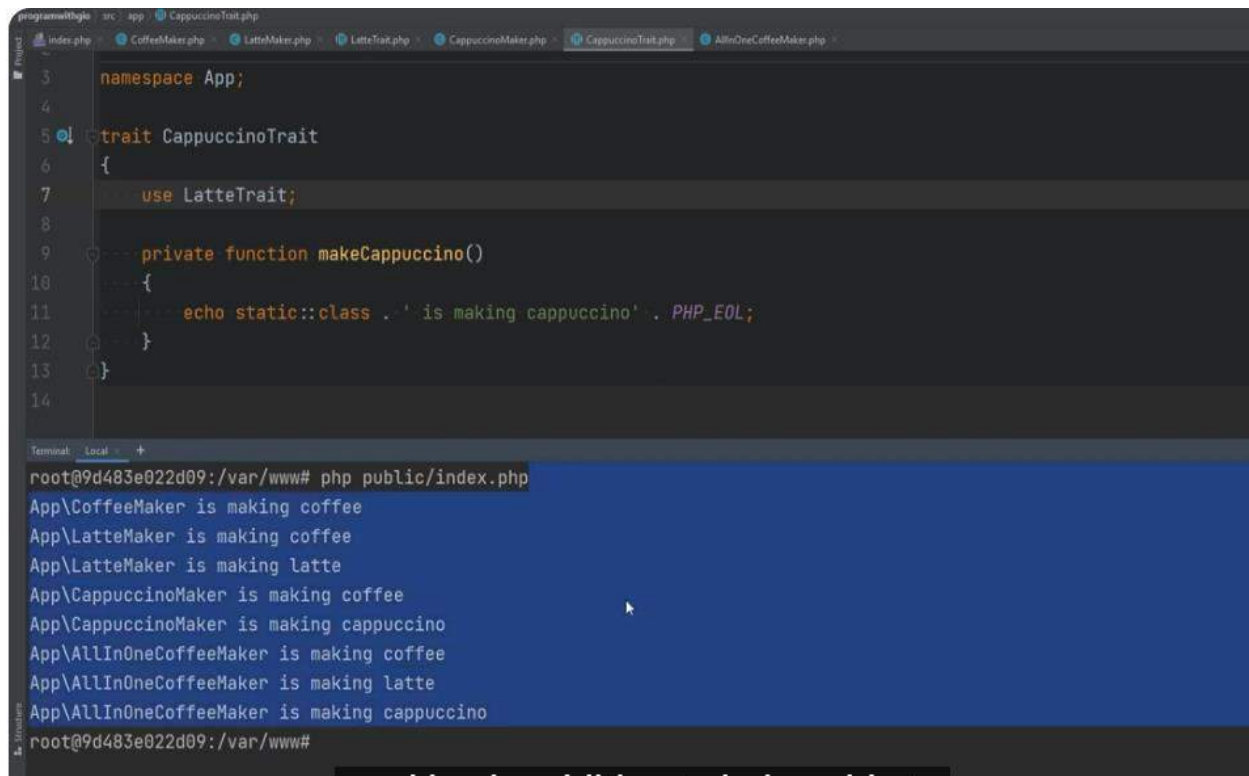
```
3 namespace App;
4
5 trait CappuccinoTrait
6 {
7     private function makeCappuccino()
8     {
9         echo static::class . ' is making cappuccino' . PHP_EOL;
10    }
11 }
12
```

Terminal: Local +

```
root@9d483e022d09:/var/www# php public/index.php
App\CoffeeMaker is making coffee
App\LatteMaker is making coffee
App\LatteMaker is making latte
App\CappuccinoMaker is making coffee
App\CappuccinoMaker is making cappuccino
App\AllInOneCoffeeMaker is making coffee
App\AllInOneCoffeeMaker is making latte
App\AllInOneCoffeeMaker is making cappuccino
root@9d483e022d09:/var/www# s
```

+ In my opinion, this is actually not good and I would not advise using this. We'll talk about my opinion and more about this towards the end of this lesson. Stick around if you want to know my opinion about it. Let's move on. You could use traits within other traits, which gives you the power to compose traits from the other traits. For example, in the cappuccino trait, we said that later we might want to have cappuccino trait be able to make latte. To do that, we defined the make latte method here. But we were essentially duplicating the code again because make latte existed in the latte trait.

+ Theo ý kiến của tôi, điều này thực tế không tốt và tôi không khuyến nghị sử dụng nó. Chúng ta sẽ nói về ý kiến của tôi và nhiều hơn về điều này ở cuối bài học này. Hãy ở lại nếu bạn muốn biết ý kiến của tôi về điều này. Hãy tiến lên. Bạn có thể sử dụng traits bên trong các traits khác, điều này cho phép bạn tạo ra các traits từ các traits khác. Ví dụ, trong traits cappuccino, chúng tôi nói rằng sau này chúng tôi có thể muốn traits cappuccino có khả năng làm latte. Để làm điều đó, chúng tôi đã định nghĩa phương thức làm latte ở đây. Nhưng về cơ bản, chúng tôi đã trùng lặp mã nguồn vì phương thức làm latte đã tồn tại trong traits latte.

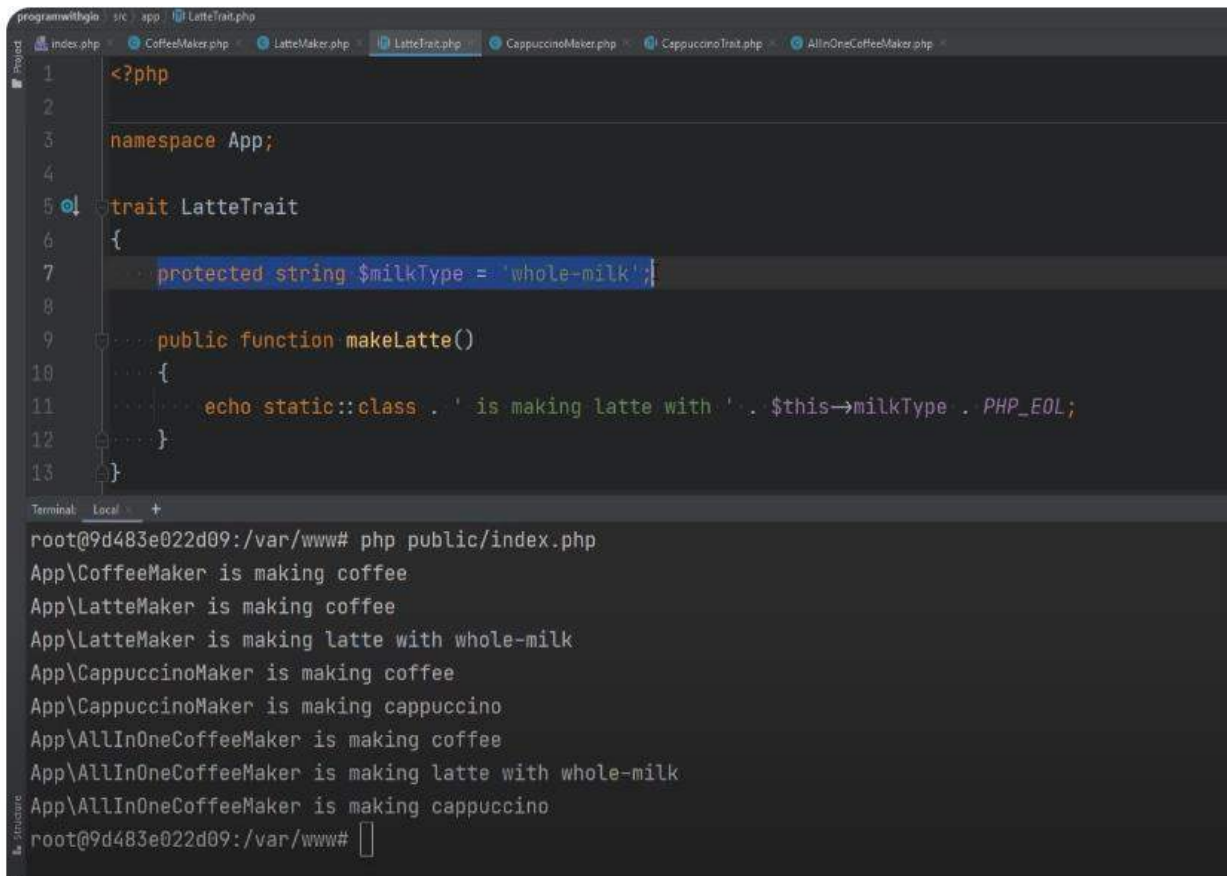


```
3 namespace App;
4
5 trait CappuccinoTrait
6 {
7     use LatteTrait;
8
9     private function makeCappuccino()
10     {
11         echo static::class . ' is making cappuccino' . PHP_EOL;
12     }
13 }
14
```

```
root@9d483e022d09:/var/www# php public/index.php
App\CoffeeMaker is making coffee
App\LatteMaker is making coffee
App\LatteMaker is making latte
App\CappuccinoMaker is making coffee
App\CappuccinoMaker is making cappuccino
App\AllInOneCoffeeMaker is making coffee
App\AllInOneCoffeeMaker is making latte
App\AllInOneCoffeeMaker is making cappuccino
root@9d483e022d09:/var/www#
```

+ I was just doing that to showcase the example. But you could actually use a trait within the trait. We could simply use the latte trait here and now Cappuccino trait would have access to the make latte method. Wherever we would use the cappuccino trait would also include the functionality from the latte trait. If we clear this up and we run the code, everything is still working. In addition to being able to define methods in the traits, you could also define properties and access them in the classes.

+ Chúng tôi chỉ làm điều này để minh họa ví dụ. Nhưng bạn có thể thực sự sử dụng một traits bên trong traits khác. Chúng tôi có thể đơn giản sử dụng traits latte ở đây và bây giờ traits cappuccino sẽ có quyền truy cập vào phương thức làm latte. Bất kỳ nơi nào chúng tôi sử dụng traits cappuccino, chức năng từ traits latte cũng sẽ được bao gồm. Nếu chúng ta xóa nội dung này và chạy mã, mọi thứ vẫn hoạt động. Ngoài việc định nghĩa các phương thức trong traits, bạn cũng có thể định nghĩa các thuộc tính và truy cập chúng trong các lớp.



```
<?php
namespace App;

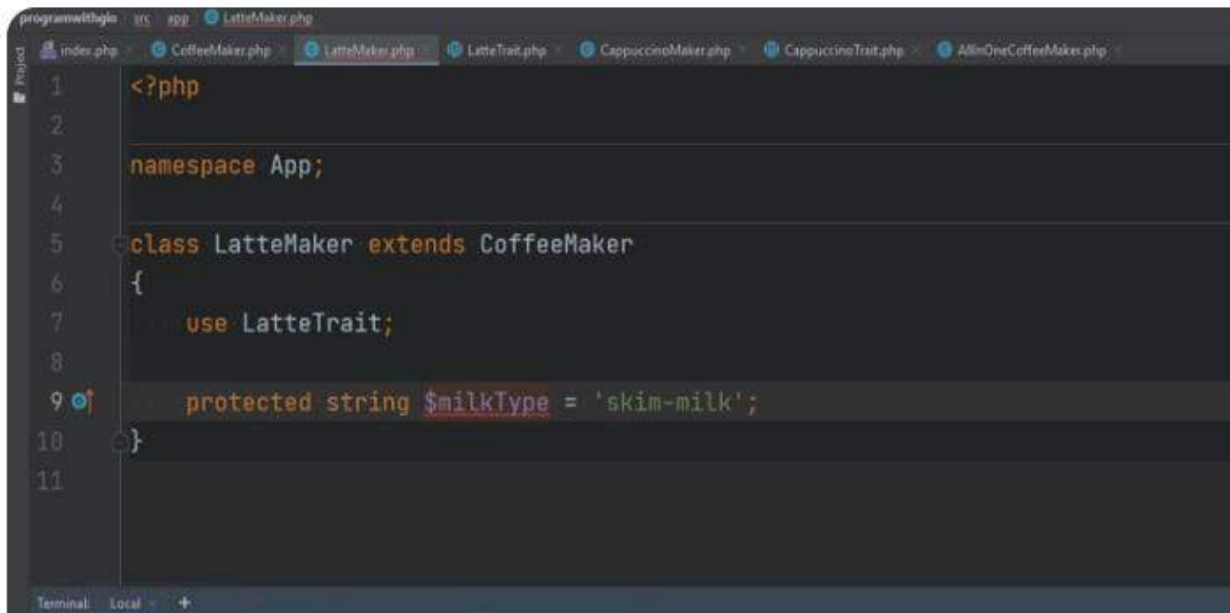
trait LatteTrait
{
    protected string $milkType = 'whole-milk';

    public function makeLatte()
    {
        echo static::class . ' is making latte with ' . $this->milkType . PHP_EOL;
    }
}
```

```
root@9d483e022d09:/var/www# php public/index.php
App\CoffeeMaker is making coffee
App\LatteMaker is making coffee
App\LatteMaker is making latte with whole-milk
App\CappuccinoMaker is making coffee
App\CappuccinoMaker is making cappuccino
App\AllInOneCoffeeMaker is making coffee
App\AllInOneCoffeeMaker is making latte with whole-milk
App\AllInOneCoffeeMaker is making cappuccino
root@9d483e022d09:/var/www#
```

+ For example, let's say that we wanted to specify what type of milk we were using for latte. If we go to latte trait, we are able to define properties here. We could do something like protected string milk type, and you could even set the default value to something like whole milk. Then we could use that within the trait or within the class that uses that trait. We could change this to say that class is making latte with this milk type. If we run the code, we see that whenever we call the make latte, it's using the whole milk. Now, because we've defined the property here, the rule is that the underlying class cannot define the same property. We have to go with the same name unless it is fully compatible. By fully compatible, I mean that the visibility, the type, and even the initial or default value has to match.

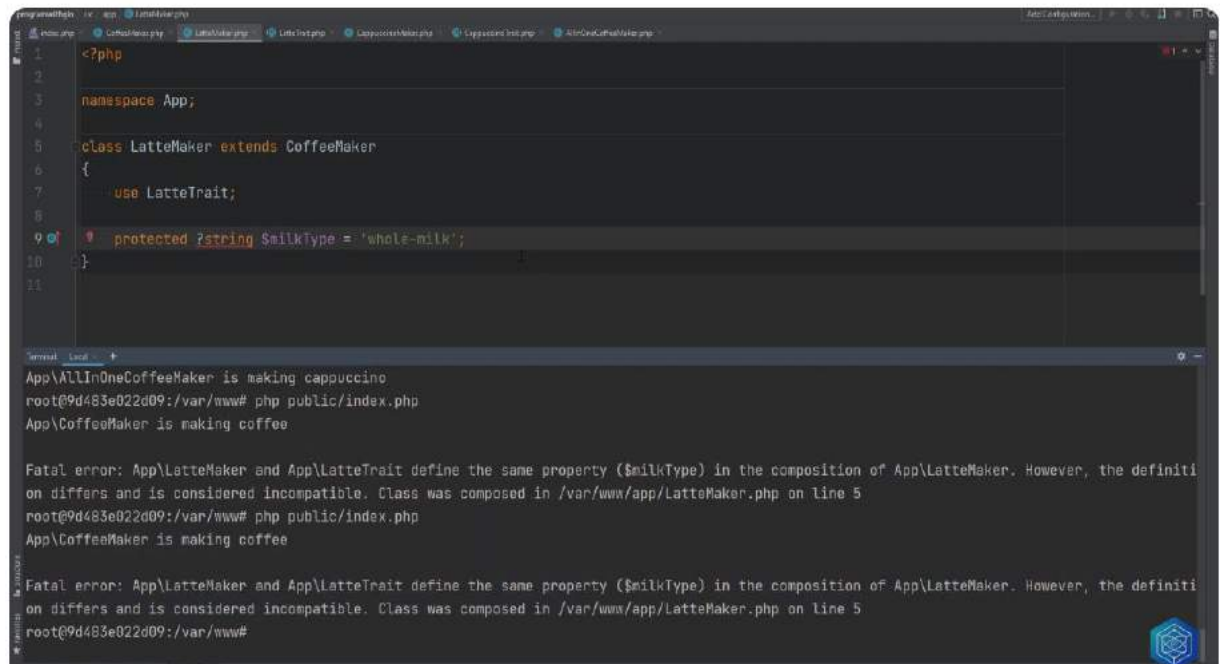
+ Ví dụ, giả sử chúng ta muốn xác định loại sữa đang sử dụng cho latte. Nếu chúng ta đi vào traits latte, chúng ta có thể định nghĩa các thuộc tính ở đây. Chúng ta có thể làm điều gì đó như protected string milk type, và bạn có thể đặt giá trị mặc định là ví dụ whole milk. Sau đó, chúng ta có thể sử dụng nó trong traits hoặc trong lớp sử dụng traits đó. Chúng ta có thể thay đổi cái này để nói rằng lớp đang tạo ra latte với loại sữa này. Nếu chúng ta chạy mã, chúng ta thấy rằng mỗi khi chúng ta gọi phương thức make latte, nó sẽ sử dụng whole milk. Bây giờ, vì chúng ta đã định nghĩa thuộc tính ở đây, quy tắc là lớp cơ sở không thể định nghĩa cùng tên thuộc tính. Chúng ta phải đi với cùng tên trừ khi nó hoàn toàn tương thích. Bằng tương thích hoàn toàn, tôi có nghĩa là phải khớp về khả năng nhìn thấy (visibility), kiểu (type), và thậm chí giá trị ban đầu hoặc giá trị mặc định phải khớp.



```
1 <?php
2
3 namespace App;
4
5 class LatteMaker extends CoffeeMaker
6 {
7     use LatteTrait;
8
9     protected string $milkType = 'skim-milk';
10 }
11
```

+For example, if we go to Latemaker and we try to define the same property, but with a different value, something like skim milk, we run the code and we get this fatal error because it is not compatible. To fix this, we would need to set this to the same value, which is the whole milk.

+ Ví dụ, nếu chúng ta vào Latemaker và chúng ta cố gắng định nghĩa cùng tên thuộc tính, nhưng với giá trị khác, ví dụ skim milk, chúng ta chạy mã và chúng ta nhận được lỗi nghiêm trọng này vì nó không tương thích. Để sửa lỗi này, chúng ta sẽ cần đặt nó thành cùng giá trị, đó là whole milk.



```
1 <?php
2
3 namespace App;
4
5 class LatteMaker extends CoffeeMaker
6 {
7     use LatteTrait;
8
9     protected ?string $milkType = 'whole-milk';
10 }
11
```

App\AllInOneCoffeeMaker is making cappuccino
root@9d483e022d09:/var/www# php public/index.php
App\CoffeeMaker is making coffee

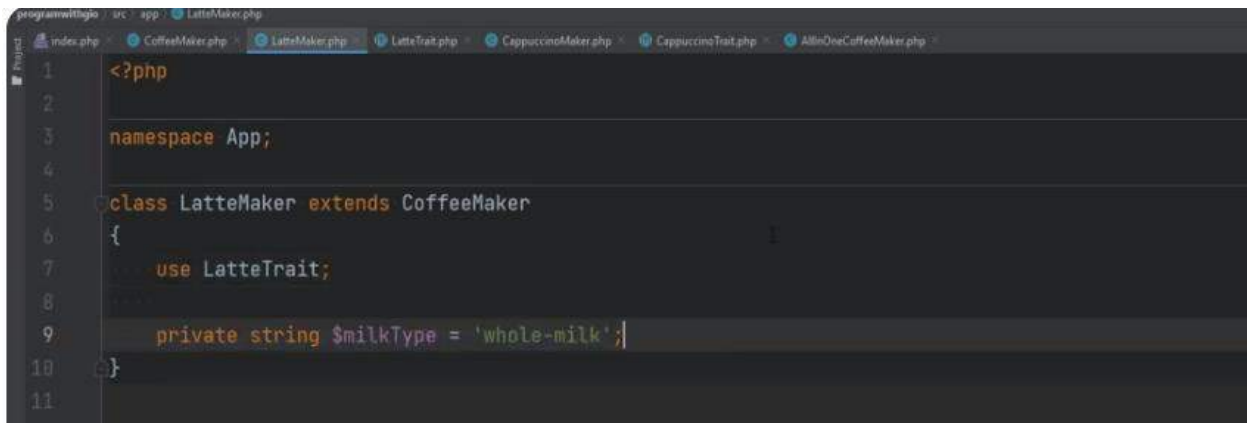
Fatal error: App\LatteMaker and App\LatteTrait define the same property (\$milkType) in the composition of App\LatteMaker. However, the definition differs and is considered incompatible. Class was composed in /var/www/app/LatteMaker.php on line 5
root@9d483e022d09:/var/www# php public/index.php
App\CoffeeMaker is making coffee

Fatal error: App\LatteMaker and App\LatteTrait define the same property (\$milkType) in the composition of App\LatteMaker. However, the definition differs and is considered incompatible. Class was composed in /var/www/app/LatteMaker.php on line 5
root@9d483e022d09:/var/www#

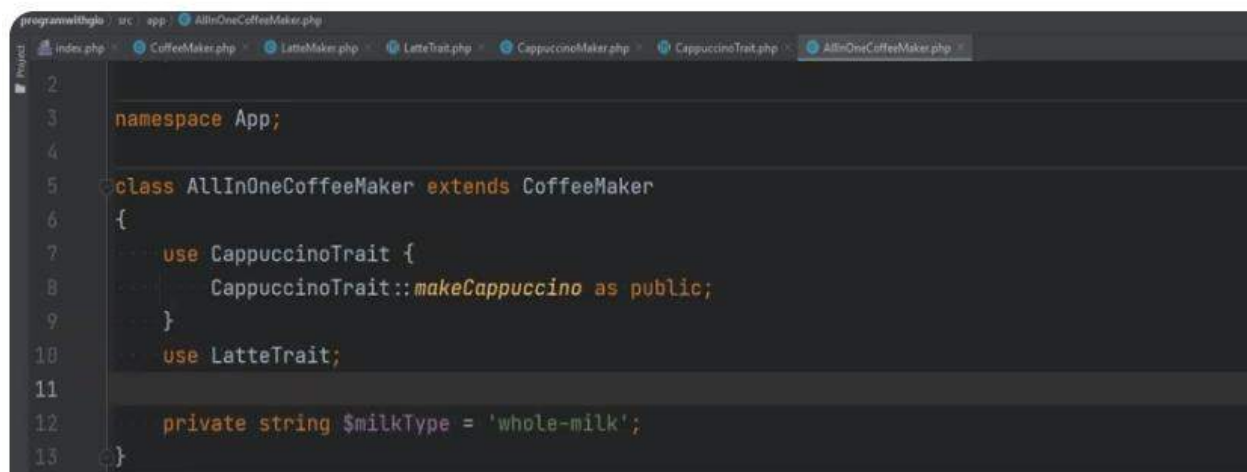
+ Now, if we run the code, everything is back to normal. But even if you change the visibility to something like public, then this would still fail because it's no longer compatible. Of course, if

you change the type, you would also make it not compatible. If you run the code, we see that we get the fatal error. Keep that in mind that if you define properties in the traits, you're not allowed to redefine the same property on the underlying classes unless it's fully compatible. Also, something you might see in frameworks or in somebody else's code is that they make use of this variable without actually having this property on the trait. If I remove this, you see some code that references this magical property that does not exist on the trait, which assumes that this property exists on the underlying class that uses this trait.

+ Bây giờ, nếu chúng ta chạy mã, mọi thứ trở lại bình thường. Nhưng ngay cả khi bạn thay đổi khả năng nhìn thấy (visibility) thành public, sau đó điều này vẫn sẽ thất bại vì nó không còn tương thích. Tất nhiên, nếu bạn thay đổi kiểu, bạn cũng sẽ làm cho nó không tương thích nữa. Nếu bạn chạy mã, chúng ta thấy rằng chúng ta nhận được lỗi nghiêm trọng. Hãy nhớ rằng nếu bạn định nghĩa các thuộc tính trong traits, bạn không được phép định nghĩa lại cùng tên thuộc tính trên các lớp cơ sở trừ khi nó hoàn toàn tương thích. Ngoài ra, điều bạn có thể thấy trong các frameworks hoặc trong mã nguồn của người khác là họ sử dụng biến this mà không có thuộc tính này trên trait. Nếu tôi loại bỏ nó, bạn sẽ thấy một số mã tham chiếu đến thuộc tính "kỳ diệu" này mà không tồn tại trên trait, và giả định rằng thuộc tính này tồn tại trên lớp cơ sở sử dụng trait này.



```
1 <?php
2
3 namespace App;
4
5 class LatteMaker extends CoffeeMaker
6 {
7     use LatteTrait;
8
9     private string $milkType = 'whole-milk';
10 }
11
```



```
2
3 namespace App;
4
5 class AllInOneCoffeeMaker extends CoffeeMaker
6 {
7     use CappuccinoTrait {
8         CappuccinoTrait::makeCappuccino as public;
9     }
10     use LatteTrait;
11
12     private string $milkType = 'whole-milk';
13 }
```



```
terminal Local +
Fatal error: App\LatteMaker and App\LatteTrait define the same property ($milkType) in the composition of App\Latte
on differs and is considered incompatible. Class was composed in /var/www/app/LatteMaker.php on line 5
root@9d483e022d09:/var/www# php public/index.php
App\CoffeeMaker is making coffee
App\LatteMaker is making coffee
App\LatteMaker is making latte with whole-milk
App\CappuccinoMaker is making coffee
App\CappuccinoMaker is making cappuccino
App\AllInOneCoffeeMaker is making coffee
App\AllInOneCoffeeMaker is making latte with whole-milk
App\AllInOneCoffeeMaker is making cappuccino
root@9d483e022d09:/var/www#
```

and run the code and everything is

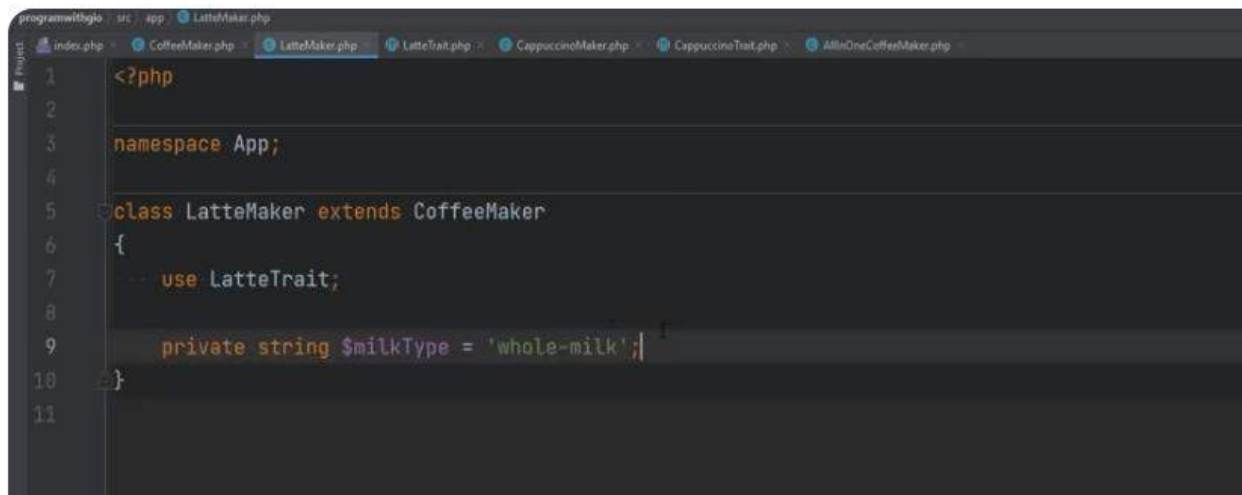
+ For example, if I go to LatteMaker here, we could simply define the milk type and set it to whole milk. We could do the same thing in the all-in-one coffee maker and run the code and everything is working as expected. But the problem with this is that I don't like referencing magical properties like that, that I'm assuming that the underlying class that uses this trait defines that property.

+ Ví dụ, nếu tôi vào LatteMaker ở đây, chúng ta có thể đơn giản định nghĩa milk type và đặt nó thành whole milk. Chúng ta có thể làm điều tương tự trong lớp all-in-one coffee maker và chạy mã, và mọi thứ hoạt động như mong đợi. Nhưng vấn đề với điều này là tôi không thích tham chiếu đến các thuộc tính "kỳ diệu" như vậy, và giả định rằng lớp cơ sở sử dụng trait này định nghĩa thuộc tính đó.

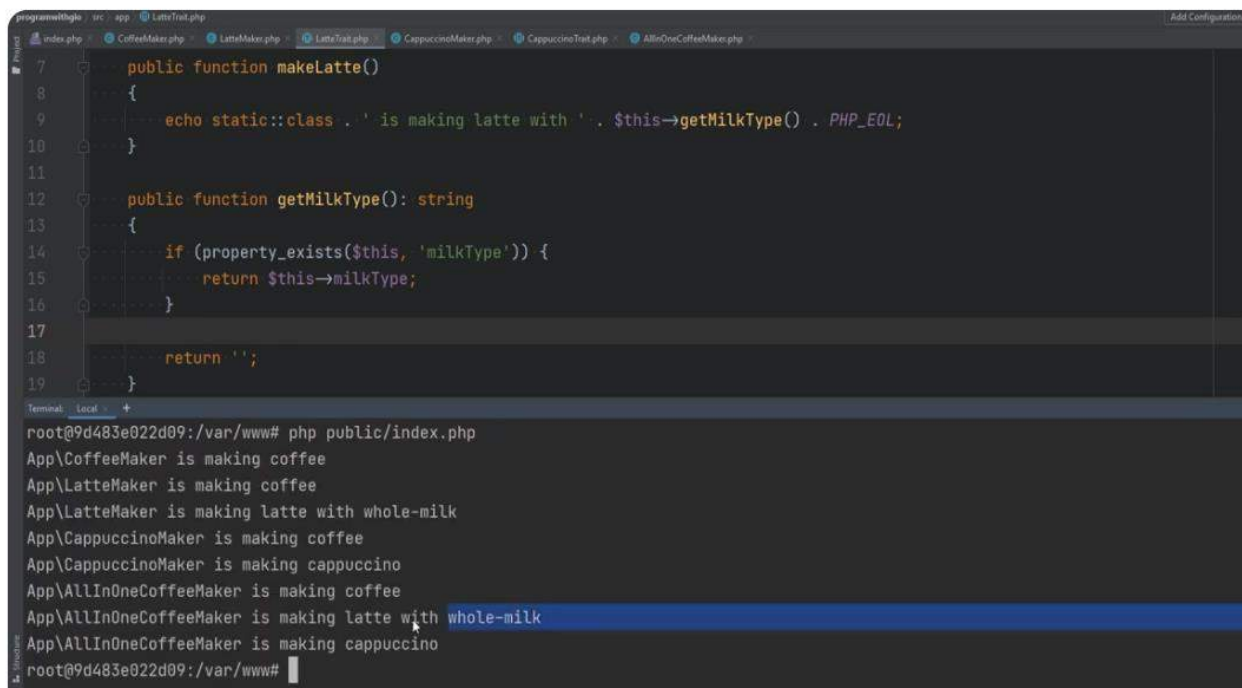
```
programWithLogo src app LatteTrait.php
index.php CoffeeMaker.php LatteMaker.php LatteTrait.php CappuccinoMaker.php CappuccinoTrait.php AllInOneCoffeeMaker.php
3 namespace App;
4
5 trait LatteTrait
6 {
7     public function makeLatte()
8     {
9         echo static::class . ' is making latte with ' . $this->getMilkType() . PHP_EOL;
10    }
11
12    public function getMilkType(): string
13    {
14        return 'whole-milk';
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
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1188
1189
1190
1191
1192
1193
1194
1195
1196
1197
1198
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1289
1290
1291
1292
1293
1294
1295
1296
1297
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1489
1490
1491
1492
1493
1494
1495
1496
1497
1498
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2179
2180
2181
2182
2183
2184
2185
2186
2187
2188
2189
2190
2191
2192
2193
2194
2195
2196
2197
2198
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2210
2211
2212
2213
2214
2215
2216
2217
2218
2219
2220
2221
2222
2223
2224
2225
2226
2227
2228
2229
2230
2231
2232
2233
2234
2235
2236
2237
2238
2239
2240
2241
2242
2243
2244
2245
2246
2247
2248
2249
2250
2251
2252
2253
2254
2255
2256
2257
2258
2259
2260
2261
2262
2263
2264
2265
2266
2267
2268
2269
2270
2271
2272
2273
2274
2275
2276
2277
2278
2279
2280
2281
2282
2283
2284
2285
2286
2287
2288
2289
2290
2291
2292
2293
2294
2295
2296
2297
2298
2299
2300
2301
2302
2303
2304
2305
2306
2307
2308
2309
2310
2311
2312
2313
2314
2315
2316
2317
2318
2319
2320
2321
2322
2323
2324
2325
2326
2327
2328
2329
2330
2331
2332
2333
2334
2335
2336
2337
2338
2339
2340
2341
2342
2343
2344
2345
2346
2347
2348
2349
2350
2351
2352
2353
2354
2355
2356
2357
2358
2359
2360
2361
2362
2363
2364
2365
2366
2367
2368
2369
2370
2371
2372
2373
2374
2375
2376
2377
2378
2379
2380
2381
2382
2383
2384
2385
2386
2387
2388
2389
2390
2391
2392
2393
2394
2395
2396
2397
2398
2399
2400
2401
2402
2403
2404
2405
2406
2407
2408
2409
2410
2411
2412
2413
2414
2415
2416
2417
2418
2419
2420
2421
2422
2423
2424
2425
2426
2427
2428
2429
2430
2431
2432
2433
2434
2435
2436
2437
2438
2439
2440
2441
2442
2443
2444
2445
2446
2447
2448
2449
2450
2451
2452
2453
2454
2455
2456
2457
2458
2459
2460
2461
2462
2463
2464
2465
2466
2467
2468
2469
2470
2471
2472
2473
2474
2475
2476
2477
2478
2479
2480
2481
2482
2483
2484
2485
2486
2487
2488
2489
2490
2491
2492
2493
2494
2495
2496
2497
2498
2499
2500
2501
2502
2503
2504
2505
2506
2507
2508
2509
2510
2511
2512
2513
2514
2515
2516
2517
2518
2519
2520
2521
2522
2523
2524
2525
2526
2527
2528
2529
2530
2531
2532
2533
2534
2535
2536
2537
2538
2539
2540
2541
2542
2543
2544
2545
2546
2547
2548
2549
2550
2551
2552
2553
2554
2555
2556
2557
2558
2559
2560
2561
2562
2563
2564
2565
2566
2567
2568
2569
2
```

trait. You could do public function, get milk type, the return string, and return whole milk. But this is not exactly the proper solution because we're hardcoding the whole milk here.

+ Thay vào đó, bạn có thể yêu cầu lớp cơ sở định nghĩa một phương thức cung cấp cho bạn giá trị của thuộc tính đó. Thay vì truy cập trực tiếp vào thuộc tính này bằng milk type, bạn có thể chỉ cần thực hiện get milk hoặc get milk type. Sau đó, bạn có thể định nghĩa phương thức này trên trait. Bạn có thể làm public function get milk type, trả về string, và trả về whole milk. Nhưng đây không phải là giải pháp chính xác vì chúng ta đang hardcode whole milk ở đây.



```
1 <?php
2
3 namespace App;
4
5 class LatteMaker extends CoffeeMaker
6 {
7     use LatteTrait;
8
9     private string $milkType = 'whole-milk';
10 }
11
```



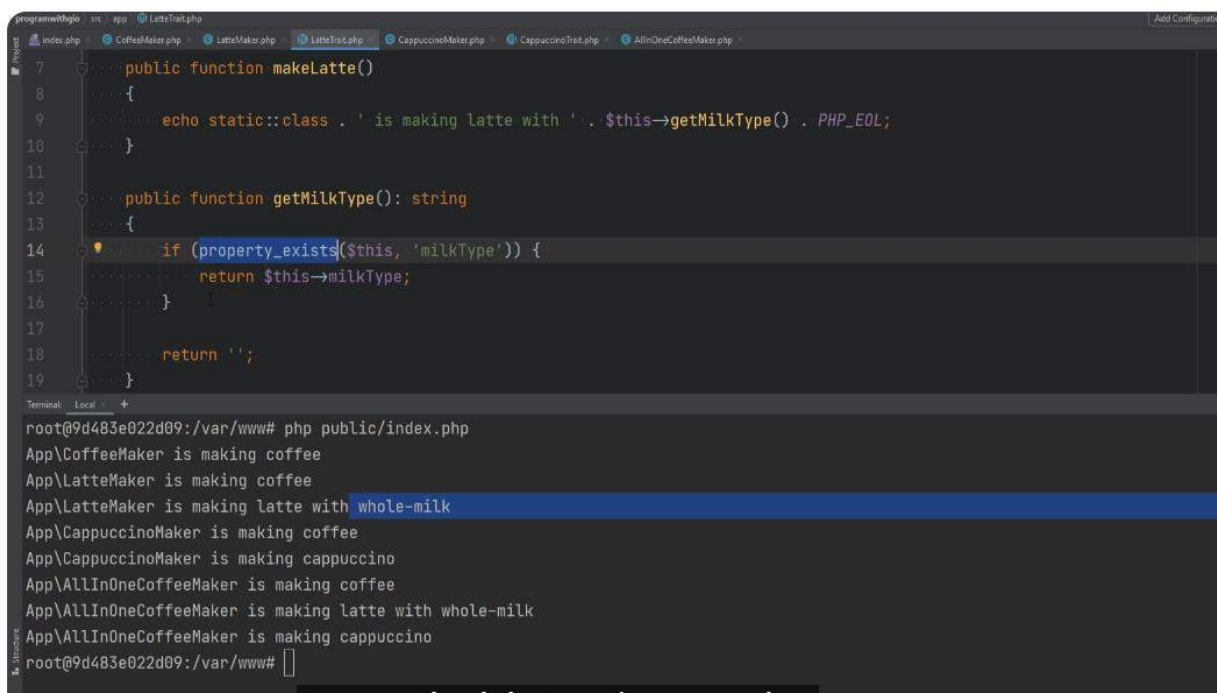
```
7 public function makeLatte()
8 {
9     echo static::class . ' is making latte with ' . $this->getMilkType() . PHP_EOL;
10 }
11
12 public function getMilkType(): string
13 {
14     if (property_exists($this, 'milkType')) {
15         return $this->milkType;
16     }
17
18     return '';
19 }
```

```
Terminal Local +
root@9d483e022d09:/var/www# php public/index.php
App\CoffeeMaker is making coffee
App\LatteMaker is making coffee
App\LatteMaker is making latte with whole-milk
App\CappuccinoMaker is making coffee
App\CappuccinoMaker is making cappuccino
App\AllInOneCoffeeMaker is making coffee
App\AllInOneCoffeeMaker is making latte with whole-milk
App\AllInOneCoffeeMaker is making cappuccino
root@9d483e022d09:/var/www#
```

+ If I clear this out and run the code again, everything is sure working and it's saying that it's making it with whole milk. But then what's the point of these properties here? Because it's not taking the value from these properties, it's taking the value from the hardcoded string that we set

right here. Instead, we want to be able to get the value from the property. You might see some developers do something like this. If property exists on the current object, then return this milk type, otherwise, return some default value. In this case, let's set the default value to blank so we can see the difference. Let's run the code and we see that it's working properly.

+Nếu tôi xóa nó đi và chạy lại mã, mọi thứ vẫn hoạt động và nó đang nói rằng nó đang làm với sữa nguyên kem. Nhưng vậy thì mục đích của các thuộc tính ở đây là gì? Bởi vì nó không lấy giá trị từ các thuộc tính này, mà lấy giá trị từ chuỗi được hardcoded mà chúng ta đã đặt ở đây. Thay vào đó, chúng ta muốn có thể lấy giá trị từ thuộc tính. Bạn có thể thấy một số nhà phát triển làm điều gì đó như thế này. Nếu thuộc tính tồn tại trên đối tượng hiện tại, thì trả về milk type này, nếu không, trả về một giá trị mặc định nào đó. Trong trường hợp này, chúng ta hãy đặt giá trị mặc định thành blank để chúng ta có thể thấy sự khác biệt. Hãy chạy mã và chúng ta thấy rằng nó đang hoạt động bình thường.

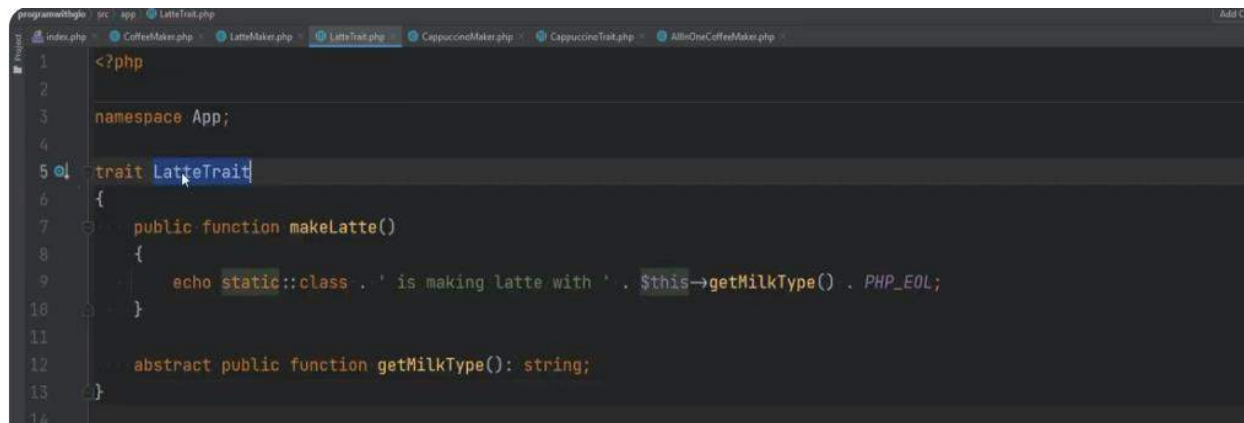


```
7 public function makeLatte()
8 {
9     echo static::class . ' is making latte with ' . $this->getMilkType() . PHP_EOL;
10 }
11
12 public function getMilkType(): string
13 {
14     if (property_exists($this, 'milkType')) {
15         return $this->milkType;
16     }
17
18     return '';
19 }
```

```
root@9d483e022d09:/var/www# php public/index.php
App\CoffeeMaker is making coffee
App\LatteMaker is making coffee
App\LatteMaker is making latte with whole-milk
App\CappuccinoMaker is making coffee
App\CappuccinoMaker is making cappuccino
App\AllInOneCoffeeMaker is making coffee
App\AllInOneCoffeeMaker is making latte with whole-milk
App\AllInOneCoffeeMaker is making cappuccino
root@9d483e022d09:/var/www#
```

+ It's taking this whole milk value from the property and not the default value. But again, this is not ideal because you are forced to do this property exists validation and you might also want to do method exists validation if you're referencing to a method, but here we are assuming it would exist on the underlying class.

+ Nó lấy giá trị sữa nguyên kem này từ thuộc tính chứ không phải giá trị mặc định. Nhưng một lần nữa, điều này không lý tưởng vì bạn buộc phải thực hiện xác thực thuộc tính này và bạn cũng có thể muốn thực hiện xác thực phương thức nếu bạn đang tham chiếu đến một phương thức, nhưng ở đây chúng tôi đang giả định rằng nó sẽ tồn tại trên lớp cơ sở.

A screenshot of a code editor window showing PHP code. The code defines a trait named `LatteTrait` within the `App` namespace. The trait has a public method `makeLatte()` which echoes a message indicating the milk type is being used. It also has an abstract public method `getMilkType(): string;`. The code is as follows:

```
1 <?php
2
3 namespace App;
4
5 trait LatteTrait
6 {
7     public function makeLatte()
8     {
9         echo static::class . ' is making latte with ' . $this->getMilkType() . PHP_EOL;
10    }
11
12    abstract public function getMilkType(): string;
13 }
```

+There is another way of approaching this, and I'm not a big fan of this, but traits has this feature and can be used, that is defining methods abstract within traits. Instead of providing the concrete definition here, we could simply make this abstract. Now we're making sure that any class that uses this trait provides the concrete implementation of this method. But hold on a second. Remember in the inheritance lesson, we said that if you have at least one abstract method, the entire class has to be marked as abstract. That is correct, but not when it comes with traits. When you have an abstract method in the trait, you don't have to mark the class that uses that trait as abstract. If we go to Latemaker, we have to define that method, but we don't have to make this class abstract.

+ Có một cách khác để tiếp cận điều này, và tôi không phải là người hâm mộ lớn của điều này, nhưng các đặc điểm có tính năng này và có thể được sử dụng, đó là định nghĩa các phương thức trừu tượng trong các đặc điểm. Thay vì cung cấp định nghĩa cụ thể ở đây, chúng ta có thể chỉ cần làm cho nó trừu tượng. Bây giờ chúng tôi đang đảm bảo rằng bất kỳ lớp nào sử dụng đặc điểm này đều cung cấp triển khai cụ thể của phương thức này. Nhưng khoan đã. Hãy nhớ trong bài học về kế thừa, chúng ta đã nói rằng nếu bạn có ít nhất một phương thức trừu tượng, thì toàn bộ lớp phải được đánh dấu là trừu tượng. Điều đó đúng, nhưng không phải khi nói đến các đặc điểm. Khi bạn có một phương thức trừu tượng trong đặc điểm, bạn không phải đánh dấu lớp sử dụng đặc điểm đó là trừu tượng. Nếu chúng ta chuyển sang Latemaker, chúng ta phải định nghĩa phương thức đó, nhưng chúng ta không phải làm cho lớp này trở thành trừu tượng.

```
programwithgo  src  app  CappuccinoTrait.php
index.php  CoffeeMaker.php  LatteMaker.php  LatteTrait.php  CappuccinoMaker.php  CappuccinoTrait.php  AllInOneCoffeeMaker.php
1  <?php
2
3  namespace App;
4
5  class AllInOneCoffeeMaker extends CoffeeMaker
6  {
7      use CappuccinoTrait {
8          CappuccinoTrait::makeCappuccino as public;
9      }
10     use LatteTrait;
11
12     private string $milkType = 'whole-milk';
13
14     public function getMilkType(): string
15     {
16         return $this->milkType;
17     }
18 }
19
```

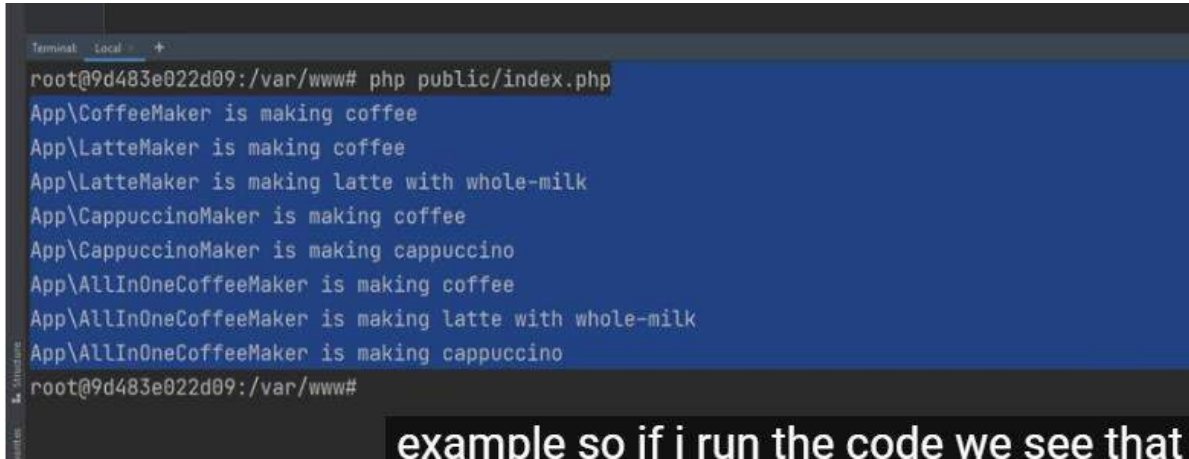
+ This can remain as regular class. Now in here, we have the access to this property and we know that it exists. We can simply return this milk type and this would work. We could do the same thing within the all-in-one coffee maker class.

+ Điều này có thể vẫn là lớp bình thường. Bây giờ ở đây, chúng ta có quyền truy cập vào thuộc tính này và chúng ta biết rằng nó tồn tại. Chúng ta có thể chỉ cần trả về loại sữa này và điều này sẽ hoạt động. Chúng ta có thể làm điều tương tự trong lớp máy pha cà phê tất cả trong một.

```
programwithgo  src  app  CappuccinoTrait.php
index.php  CoffeeMaker.php  LatteMaker.php  LatteTrait.php  CappuccinoMaker.php  CappuccinoTrait.php  AllInOneCoffeeMaker.php
1  <?php
2
3  namespace App;
4
5  trait CappuccinoTrait
6  {
7      private function makeCappuccino()
8      {
9          echo static::class . ' is making cappuccino' . PHP_EOL;
10     }
11 }
12
```

+ Before we run the code, let's remove the use of Latemaker trait within the cappuccino trait because this would cause errors. We would have to either make a cappuccino maker abstract class or provide the implementation of the get milk type method. We don't want to do that for this example.

+ Trước khi chạy mã, hãy xóa việc sử dụng đặc điểm Latemaker trong đặc điểm cappuccino vì điều này sẽ gây ra lỗi. Chúng ta sẽ phải tạo lớp máy pha cappuccino trừu tượng hoặc cung cấp triển khai của phương thức get milk type. Chúng ta không muốn làm điều đó cho ví dụ này.

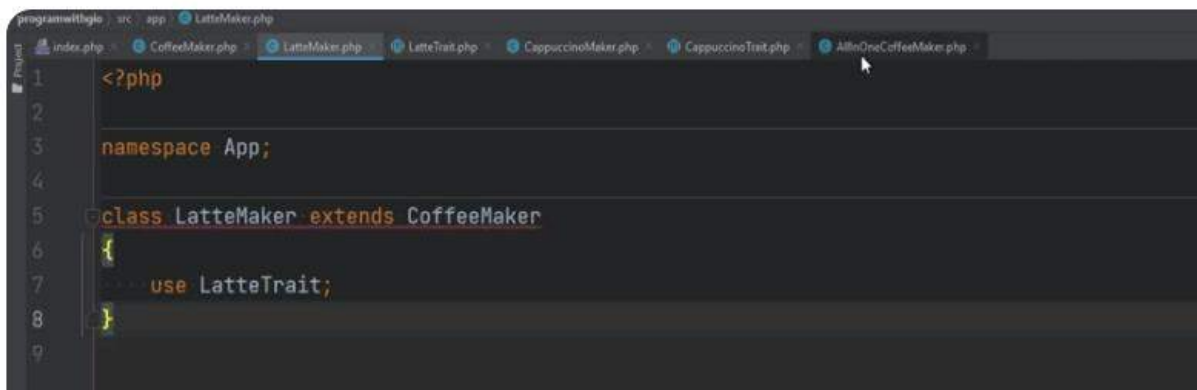


```
Terminat Local +
root@9d483e022d09:/var/www# php public/index.php
App\CoffeeMaker is making coffee
App\LatteMaker is making coffee
App\LatteMaker is making latte with whole-milk
App\CappuccinoMaker is making coffee
App\CappuccinoMaker is making cappuccino
App\AllInOneCoffeeMaker is making coffee
App\AllInOneCoffeeMaker is making latte with whole-milk
App\AllInOneCoffeeMaker is making cappuccino
root@9d483e022d09:/var/www#
```

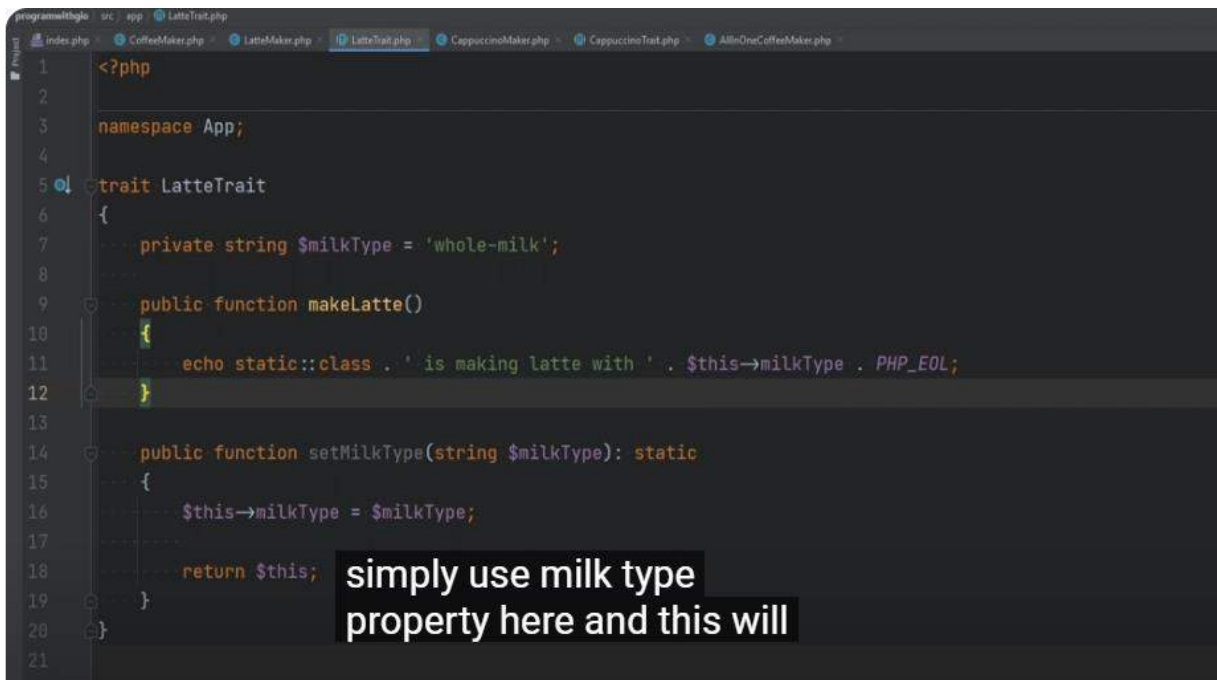
example so if i run the code we see that

+ If I run the code, we see that everything is working and we did not have to mark this class as abstract and we did not have to mark LatteMaker as abstract either. To be honest, I'm not a big fan of abstracts within the traits. But to me, the better way would be to define the property with the default value in the trait.

+ Nếu tôi chạy mã, chúng ta sẽ thấy rằng mọi thứ đều hoạt động và chúng ta không phải đánh dấu lớp này là trừu tượng và chúng ta cũng không phải đánh dấu LatteMaker là trừu tượng. Thành thật mà nói, tôi không phải là người hâm mộ của các khái niệm trừu tượng trong các đặc điểm. Nhưng đối với tôi, cách tốt hơn là định nghĩa thuộc tính với giá trị mặc định trong đặc điểm.



```
programwithgo src > app > LatteMaker.php
index.php > CoffeeMaker.php > LatteMaker.php > LatteTrait.php > CappuccinoMaker.php > CappuccinoTrait.php > AllInOneCoffeeMaker.php
1 <?php
2
3 namespace App;
4
5 class LatteMaker extends CoffeeMaker
6 {
7     use LatteTrait;
8 }
9
```

```
1 <?php
2
3 namespace App;
4
5 trait LatteTrait
6 {
7     private string $milkType = 'whole-milk';
8
9     public function makeLatte()
10    {
11        echo static::class . ' is making latte with ' . $this->milkType . PHP_EOL;
12    }
13
14    public function setMilkType(string $milkType): static
15    {
16        $this->milkType = $milkType;
17
18        return $this;
19    }
20 }
21
```

simply use milk type
property here and this will

+ We would get rid of this and we would get rid of this from here. Within the Latte trait, we would simply define it like we had it before. If we wanted to change this, then we could simply provide the setter method, something like public function, set milk type, where it takes the milk type as the argument and simply updates the milk type to milk type and returns this object. We could set the return type to static. Then if we ever wanted to change the milk type, we could call this method and it will be available on the class because it's using the trait. This is much cleaner way than having abstract method and duplicating the code across different classes. Now, instead of get milk type, we could simply use milk type property here and this will work.

+ Chúng ta sẽ loại bỏ cái này và chúng ta sẽ loại bỏ cái này từ đây. Trong đặc điểm Latte, chúng ta chỉ cần định nghĩa nó như chúng ta đã có trước đây. Nếu chúng ta muốn thay đổi điều này, thì chúng ta có thể chỉ cần cung cấp phương thức setter, thứ gì đó như public function, set milk type, nơi nó lấy loại sữa làm đối số và chỉ cần cập nhật milk type thành milk type và trả về đối tượng này. Chúng ta có thể đặt kiểu trả về thành tĩnh. Sau đó, nếu chúng ta muốn thay đổi loại sữa, chúng ta có thể gọi phương thức này và nó sẽ có sẵn trên lớp vì nó đang sử dụng đặc điểm. Đây là cách sạch hơn nhiều so với việc có phương thức trừu tượng và sao chép mã giữa các lớp khác nhau. Bây giờ, thay vì get milk type, chúng ta có thể chỉ cần sử dụng thuộc tính milk type ở đây và điều này sẽ hoạt động.

A terminal window with a dark background and light blue text. The prompt is 'root@9d483e022d09:/var/www#'. The command 'php public/index.php' has been executed. The output shows several lines of text: 'App\CoffeeMaker is making coffee', 'App\LatteMaker is making coffee', 'App\LatteMaker is making latte', 'App\CappuccinoMaker is making coffee', 'App\CappuccinoMaker is making cappuccino', 'App\AllInOneCoffeeMaker is making coffee', 'App\AllInOneCoffeeMaker is making latte with whole-milk', and 'App\AllInOneCoffeeMaker is making cappuccino'. A semi-transparent black box with white text is overlaid on the right side of the terminal, containing the text 'run the code' and 'everything is working as expected' in two lines.

```
Terminal Local +
root@9d483e022d09:/var/www# php public/index.php
App\CoffeeMaker is making coffee
App\LatteMaker is making coffee
App\LatteMaker is making latte
App\CappuccinoMaker is making coffee
App\CappuccinoMaker is making cappuccino
App\AllInOneCoffeeMaker is making coffee
App\AllInOneCoffeeMaker is making latte with whole-milk
App\AllInOneCoffeeMaker is making cappuccino
root@9d483e022d09:/var/www#
```

+ If I run the code, everything is working as expected. Something to be aware of with the abstract methods within traits is that prior to PHB 8, you were only able to define public or protected abstract methods within the traits. But since PHB 8, you could also define private abstract methods, which, as you know, is not possible with abstract classes when you're defining abstract methods. All right, let's move on to Statics. In addition to having regular properties and methods, you could also have static properties and static methods.

+ Nếu tôi chạy mã, mọi thứ đều hoạt động như mong đợi. Một điều cần lưu ý với các phương thức trừu tượng trong các đặc điểm là trước PHP 8, bạn chỉ có thể định nghĩa các phương thức trừu tượng công khai hoặc được bảo vệ trong các đặc điểm. Nhưng kể từ PHP 8, bạn cũng có thể định nghĩa các phương thức trừu tượng riêng tư, điều này, như bạn biết, không thể thực hiện được với các lớp trừu tượng khi bạn đang định nghĩa các phương thức trừu tượng. Được rồi, chúng ta hãy chuyển sang Statics. Ngoài việc có các thuộc tính và phương thức thông thường, bạn cũng có thể có các thuộc tính và phương thức tĩnh.

```
1 <?php
2
3 namespace App;
4
5 trait LatteTrait
6 {
7     private string $milkType = 'whole-milk';
8
9     public function makeLatte()
10     {
11         echo static::class . ' is making latte with ' . $this->milkType . PHP_EOL;
12     }
13
14     public static function foo()
15     {
16         echo 'Foo Bar';
17     }
18
19     public function setMilkType(string $milkType): static
20     {
21         $this->milkType = $milkType;
22
23         return $this;
24     }
25 }
```

and let's say it just prints out full
bar

```
1 <?php
2
3 require_once __DIR__ . '/../vendor/autoload.php';
4
5 $coffeeMaker = new \App\CoffeeMaker();
6 $coffeeMaker->makeCoffee();
7
8 $latteMaker = new \App\LatteMaker();
9 $latteMaker->makeCoffee();
10 $latteMaker->makeLatte();
11
12 $cappuccinoMaker = new \App\CappuccinoMaker();
13 $cappuccinoMaker->makeCoffee();
14 $cappuccinoMaker->makeCappuccino();
15
16 $allInOneCoffeeMaker = new \App\AllInOneCoffeeMaker();
17 $allInOneCoffeeMaker->makeCoffee();
18 $allInOneCoffeeMaker->makeLatte();
19 $allInOneCoffeeMaker->makeCappuccino();
20
21 \App\LatteMaker::foo();
22
23
```

do
latte maker full if

+ We could simply have public, static function, foo, and let's say it just prints out foobar and within index. Phb, we could simply do LatteMaker foo. If we run the code, we see that foobar gets printed. Same applies to the properties.

+ Chúng ta có thể chỉ cần có public, static function, foo, và giả sử nó chỉ in ra foobar và trong index.php, chúng ta có thể chỉ cần làm LatteMaker foo. Nếu chúng ta chạy mã, chúng ta sẽ thấy rằng foobar được in ra. Điều tương tự cũng áp dụng cho các thuộc tính.

```
1 <?php
2
3 namespace App;
4
5 trait LatteTrait
6 {
7     private string $milkType = 'whole-milk';
8
9     public static int $x = 1;
10
11     public function makeLatte()
12     {
```

```
19 $allInOneCoffeeMaker->makeLatte();
20
21 \App\LatteMaker::$x();
22
23
```

```
Terminal Local
root@9d483e022d09:/var/www# php public/index.php
App\CoffeeMaker is making coffee
App\LatteMaker is making coffee
App\LatteMaker is making latte
App\CappuccinoMaker is making coffee
App\CappuccinoMaker is making cappuccino
App\AllInOneCoffeeMaker is making coffee
App\AllInOneCoffeeMaker is making latte with whole-milk
App\AllInOneCoffeeMaker is making cappuccino
root@9d483e022d09:/var/www#
```

+ We could set public static int x equals to one and we could print that out and run the code and it's working. Now, something to be aware of, when a trait has a static property, each class that uses that trait will have the independent instance of that property. Unlike in Heritance, where the static property is shared. Let's see this in action. We have the Coffee Maker base class here.

+ Chúng ta có thể đặt public static int x bằng một và chúng ta có thể in ra giá trị đó và chạy mã, và nó hoạt động. Bây giờ, điều cần lưu ý, khi một tính chất có một thuộc tính static, mỗi lớp sử dụng tính chất đó sẽ có một phiên bản độc lập của thuộc tính đó. Không giống như trong Kế

thừa, nơi thuộc tính static được chia sẻ. Hãy xem điều này trong thực tế. Chúng ta có lớp cơ sở Coffee Maker ở đây.

```
1 <?php
2
3 namespace App;
4
5 class CoffeeMaker
6 {
7     public static string $foo;
8     public function makeCoffee()
9     {
10         echo static::class . ' is making coffee' . PHP_EOL;
11     }
12 }
13
```

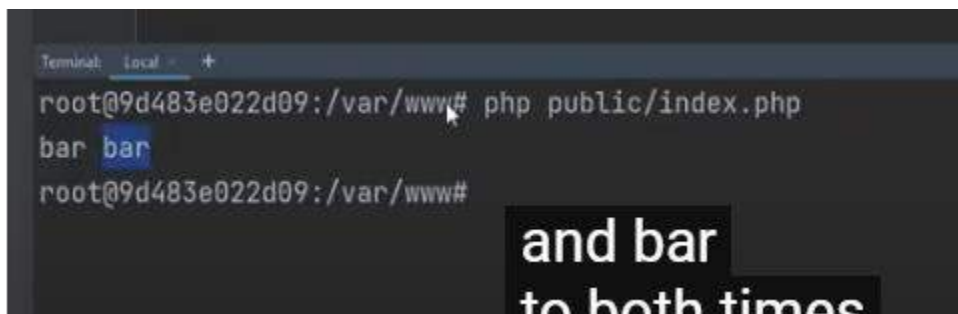
```
1 <?php
2
3 require_once __DIR__ . '/../vendor/autoload.php';
4
5 // $coffeeMaker = new \App\CoffeeMaker();
6 // $coffeeMaker->makeCoffee();
7 //
8 // $latteMaker = new \App\LatteMaker();
9 // $latteMaker->makeCoffee();
10 // $latteMaker->makeLatte();
11 //
12 // $cappuccinoMaker = new \App\CappuccinoMaker();
13 // $cappuccinoMaker->makeCoffee();
14 // $cappuccinoMaker->makeCappuccino();
15 //
16 // $allInOneCoffeeMaker = new \App\AllInOneCoffeeMaker();
17 // $allInOneCoffeeMaker->makeCoffee();
18 // $allInOneCoffeeMaker->makeCappuccino();
19 // $allInOneCoffeeMaker->makeLatte();
20
21 \App\CoffeeMaker::$foo = 'foo';
22 \App\LatteMaker::$foo = 'bar';
23
24 echo \App\CoffeeMaker::$foo . ' ' . \App\LatteMaker::$foo . PHP_EOL;
```

inheritance so let's do this
let's

+ Let's add a public static property here called FU. Then in index. Php, let's simply comment this out for now and let's do something like this. We can do coffee maker foo equals to foo. Then we have the latte maker that extends the coffee maker. We could do latte maker foo equals to bar. Now, let's print out

the foo variable from both coffee maker and latte-maker and see what we get when we use the inheritance.

+Hãy thêm một thuộc tính static public ở đây gọi là FU. Sau đó, trong index.php, hãy tạm thời bỏ chúng ta ra và làm điều này. Chúng ta có thể tạo ra coffee maker foo và gán foo bằng foo. Sau đó, chúng ta có latte maker mà kế thừa từ coffee maker. Chúng ta có thể tạo latte maker foo và gán foo bằng bar. Bây giờ, hãy in ra biến foo từ cả coffee maker và latte maker và xem chúng ta nhận được gì khi sử dụng kế thừa.

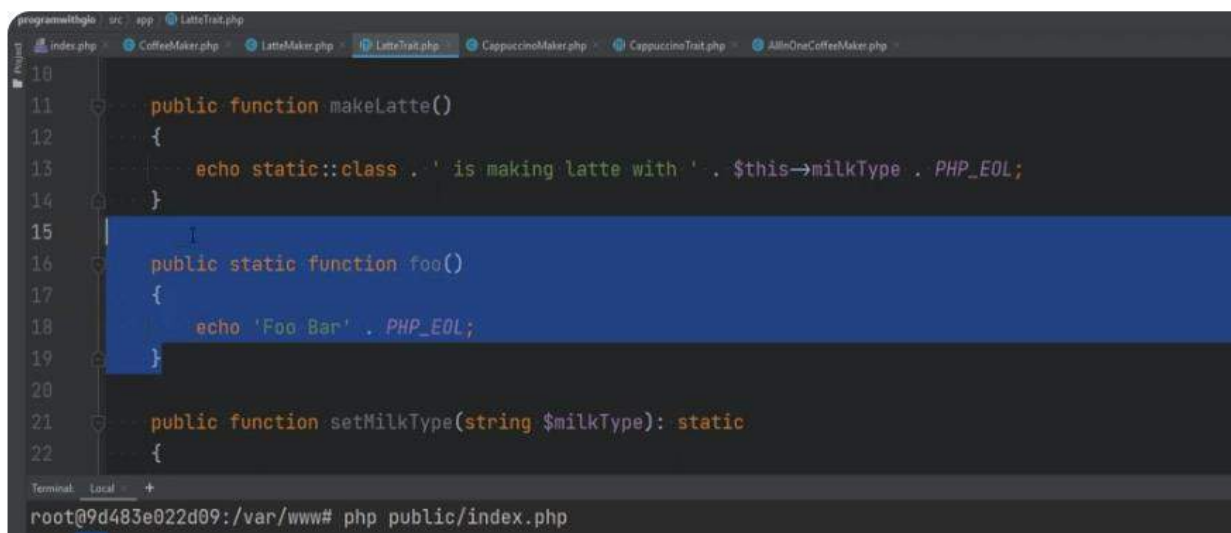


```
Terminal: Local +
root@9d483e022d09:/var/www# php public/index.php
bar bar
root@9d483e022d09:/var/www#
```

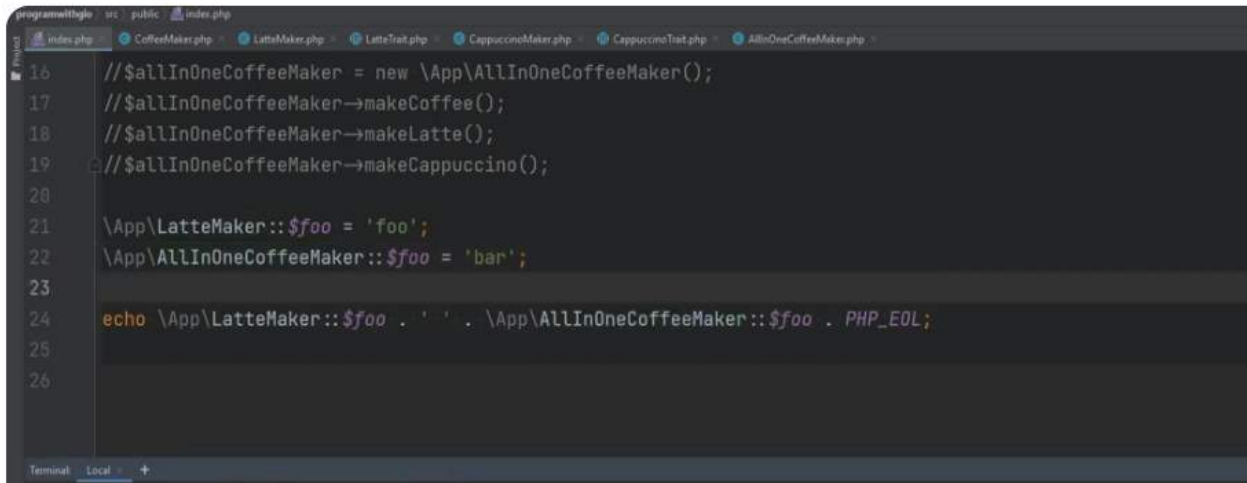
and bar
to both times

+ Let's do this. Let's run the code and we're getting bar and bar the both times. As you can see, when we're using inheritance, the static property is shared. When we're changing the value, it affects the original class as well. This is not the case with the traits. Now, let's get rid of this from here and let's simply put that within here.

+ Hãy làm như sau. Chạy mã và chúng ta nhận được bar và bar cả hai lần. Như bạn có thể thấy, khi chúng ta sử dụng kế thừa, thuộc tính static được chia sẻ. Khi chúng ta thay đổi giá trị, nó ảnh hưởng đến lớp gốc cũng. Điều này không xảy ra với tính chất. Bây giờ, hãy loại bỏ phần này và đặt nó đơn giản trong đây.



```
programmultiglo src app LatteTrait.php
index.php CoffeeMaker.php LatteMaker.php LatteTrait.php CappuccinoMaker.php CappuccinoTrait.php AllInOneCoffeeMaker.php
10
11 public function makeLatte()
12 {
13     echo static::class . ' is making latte with ' . $this->milkType . PHP_EOL;
14 }
15
16 public static function foo()
17 {
18     echo 'Foo Bar' . PHP_EOL;
19 }
20
21 public function setMilkType(string $milkType): static
22 {
23
24 }
25
Terminal: Local +
root@9d483e022d09:/var/www# php public/index.php
```



```
16 // $allInOneCoffeeMaker = new \App\AllInOneCoffeeMaker();
17 // $allInOneCoffeeMaker->makeCoffee();
18 // $allInOneCoffeeMaker->makeLatte();
19 // $allInOneCoffeeMaker->makeCappuccino();
20
21 \App\LatteMaker::$foo = 'foo';
22 \App\AllInOneCoffeeMaker::$foo = 'bar';
23
24 echo \App\LatteMaker::$foo . ' ' . \App\AllInOneCoffeeMaker::$foo . PHP_EOL;
25
26
```

+ Let's run with the property X and the method foo. Now in the index. Phb, we know that the property foo is available on the LatteMaker because LatteMaker uses the Latte trait. We also know that All-in-One Coffee Maker also uses the Latte trait and therefore it also has the property foo. We could do All-in-One Coffee Maker foo equals to bar and we can set this to full.

+ Hãy chạy với thuộc tính X và phương thức foo. Bây giờ, trong index.php, chúng ta biết rằng thuộc tính foo có sẵn trên LatteMaker vì LatteMaker sử dụng tính chất Latte. Chúng ta cũng biết rằng All-in-One Coffee Maker cũng sử dụng tính chất Latte và do đó nó cũng có thuộc tính foo. Chúng ta có thể tạo All-in-One Coffee Maker foo và gán foo bằng bar và chúng ta có thể đặt giá trị này thành full.



```
Terminal Local +
root@9d483e022d09:/var/www# php public/index.php
foo bar
root@9d483e022d09:/var/www#
```

case
are not shared and e

+Let's change this around and let's clear the code and run it again. We see that the first time it's printing foo, which is on the LatteMaker, which is this. Then on the All-in-One Coffee Maker, it's printing bar. As you can see, the static properties in this case are not shared and each individual class that uses the trait has its own version of the static property. Another thing to note is that the magic class constant, when used in a trait, will resolve to the class where the trait is used.

+ Hãy thay đổi điều này và làm sạch mã và chạy lại. Chúng ta thấy rằng lần đầu tiên nó in ra foo, tức là ở LatteMaker, như là này. Sau đó trên All-in-One Coffee Maker, nó in ra bar. Như bạn có thể thấy, các thuộc tính static trong trường hợp này không được chia sẻ và mỗi lớp riêng

lê sử dụng tính chất có phiên bản riêng của thuộc tính static. Một điều cần lưu ý khác là hằng số lớp kỳ diệu, khi sử dụng trong một tính chất, sẽ giải quyết thành lớp mà tính chất đang sử dụng.

```
1 <?php
2
3 namespace App;
4
5 trait LatteTrait
6 {
7     private string $milkType = 'whole-milk';
8
9     public function makeLatte()
10     {
11         echo __CLASS__ . ' is making latte with ' . $this->milkType . PHP_EOL;
12     }
13
14     public function setMilkType(string $milkType): static
15     {
16         $this->milkType = $milkType;
17
18         return $this;
19     }
20 }
```

right if we replace this with the magic constant class

```
1 <?php
2
3 require_once __DIR__ . '/../vendor/autoload.php';
4
5 $coffeeMaker = new \App\CoffeeMaker();
6 $coffeeMaker->makeCoffee();
7
8 $latteMaker = new \App\LatteMaker();
9 $latteMaker->makeCoffee();
10 $latteMaker->makeLatte();
11
12 $cappuccinoMaker = new \App\CappuccinoMaker();
13 $cappuccinoMaker->makeCoffee();
14 $cappuccinoMaker->makeCappuccino();
15
16 $allInOneCoffeeMaker = new \App\AllInOneCoffeeMaker();
17 $allInOneCoffeeMaker->makeCoffee();
18 $allInOneCoffeeMaker->makeCappuccino();
19 $allInOneCoffeeMaker->makeLatte();
20
21
22
```

the code on the index.php and we

```
Terminal Local +
root@9d483e022d09:/var/www# php public/index.php
App\CoffeeMaker is making coffee
App\LatteMaker is making coffee
App\LatteMaker is making latte
App\CappuccinoMaker is making coffee
App\CappuccinoMaker is making
App\AllInOneCoffeeMaker is making coffee
App\AllInOneCoffeeMaker is making latte with whole-milk
App\AllInOneCoffeeMaker is making cappuccino
root@9d483e022d09:/var/www#
```

and we ran it we see
class name where th

+ For example, in the Latta trait, we are using static class. If we replace this with the magic constant class and we uncommented the code on the index. Php and we ran it, we see that it's using the class name where that trait is being used. In this case, it's using LattaMaker, and in this case, it's using All-in-One Coffee Maker. We've talked about what traits are, how to use them, the features of traits, and so on.

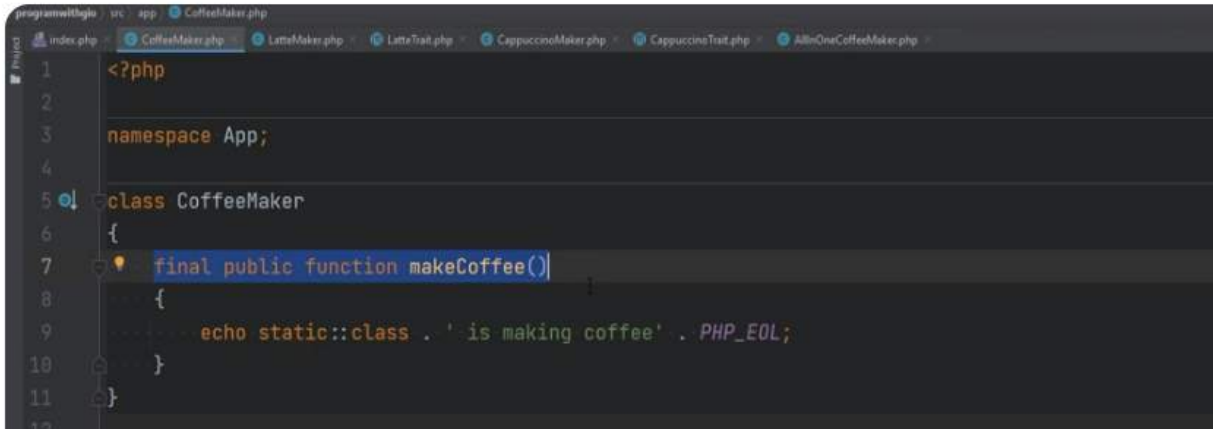
+ Ví dụ, trong tính chất Latta, chúng ta đang sử dụng class static. Nếu chúng ta thay thế điều này bằng hằng số kỳ diệu class và bỏ dấu chú thích trong mã index.php và chạy nó, chúng ta thấy rằng nó sử dụng tên lớp mà tính chất đang được sử dụng. Trong trường hợp này, nó đang sử dụng LattaMaker, và trong trường hợp này, nó đang sử dụng All-in-One Coffee Maker. Chúng ta đã nói về traits là gì, cách sử dụng chúng, các tính năng của traits và vân vân.



+ Now it's time for me to share my personal opinion about traits and just to be clear, traits are great when used properly and as with everything else, when they're actually not overused. One issue with the traits is being able to provide abstract methods within the traits. Traits should not be used for anything other than simple code reuse, in my opinion. When you're defining a method abstract in a trait, you're trying to enforce a contract, right? Enforcing contracts through traits, I think, is pretty bad idea. If you want to enforce contracts, you should use either abstract classes with abstract methods or better than abstract classes with abstract methods is to simply use interfaces. That's what interfaces are for. Interfaces are contracts. Let me show you one example where you can't exactly enforce things with traits because certain rules don't apply like it applies with inheritance.

+ Bây giờ đến lúc tôi chia sẻ ý kiến cá nhân về traits và để rõ ràng, traits tốt khi sử dụng đúng cách và giống với mọi thứ khác, khi chúng không bị lạm dụng. Một vấn đề với traits là khả năng

cung cấp các phương thức trừu tượng trong traits. Theo ý kiến của tôi, traits không nên được sử dụng cho bất cứ điều gì ngoài việc tái sử dụng mã đơn giản. Khi bạn định nghĩa một phương thức trừu tượng trong một trait, bạn đang cố gắng bắt buộc một hợp đồng, phải không? Tôi nghĩ rằng việc bắt buộc các hợp đồng thông qua traits khá tệ. Nếu bạn muốn bắt buộc các hợp đồng, bạn nên sử dụng hoặc các lớp trừu tượng với các phương thức trừu tượng hoặc tốt hơn là các lớp trừu tượng với các phương thức trừu tượng là đơn giản sử dụng giao diện. Đó là mục đích của các giao diện. Hãy để tôi cho bạn xem một ví dụ nơi bạn không thể chính xác áp dụng các quy tắc với traits vì một số quy tắc không áp dụng giống như nó áp dụng với kế thừa.

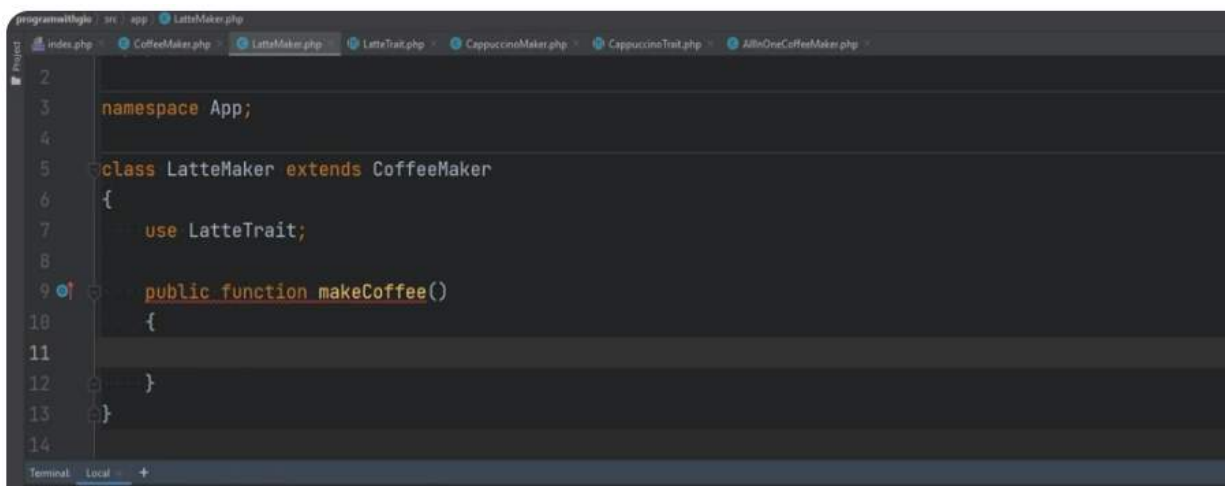


```
<?php
namespace App;

class CoffeeMaker
{
    final public function makeCoffee()
    {
        echo static::class . ' is making coffee' . PHP_EOL;
    }
}
```

+For example, the final keyword. When we're using inheritance in the base class, we can define a method to be final. This makes sure that you're not able to override this method in your child classes.

+ Ví dụ, từ khóa final. Khi chúng ta sử dụng kế thừa trong lớp cơ sở, chúng ta có thể định nghĩa một phương thức là final. Điều này đảm bảo rằng bạn không thể ghi đè phương thức này trong các lớp con của bạn.



```
namespace App;

class LatteMaker extends CoffeeMaker
{
    use LatteTrait;

    public function makeCoffee()
    {
    }
}
```

```
14
Terminal Local
root@9d483e022d09:/var/www# php public/index.php
App\CoffeeMaker is making coffee

Fatal error: Cannot override final method App\CoffeeMaker::makeCoffee() in /var
root@9d483e022d09:/var/www#
```

+ In the LatteMaker class, when we extend the CoffeeMaker, we can no longer override the Make Coffee method. We're going to get this error. If I try to run the code, we're going to get this fatal error that we're trying to override the final method.

+ Trong lớp LatteMaker, khi chúng ta mở rộng từ CoffeeMaker, chúng ta không thể ghi đè phương thức Make Coffee nữa. Chúng ta sẽ nhận được lỗi này. Nếu tôi cố gắng chạy mã, chúng ta sẽ nhận được lỗi chết người rằng chúng ta đang cố gắng ghi đè phương thức cuối cùng.

```
programwithglo . src app LatteMaker.php
index.php CoffeeMaker.php LatteMaker.php LatteTrait.php CappuccinoMaker.php CappuccinoTrait.php AllInOneCoffeeMaker.php
2
3 namespace App;
4
5 class LatteMaker extends CoffeeMaker
6 {
7     use LatteTrait;
8
9     public function makeCoffee()
10    {
11
12    }
13 }
14
Terminal Local
root@9d483e022d09:/var/www# php public/index.php
```

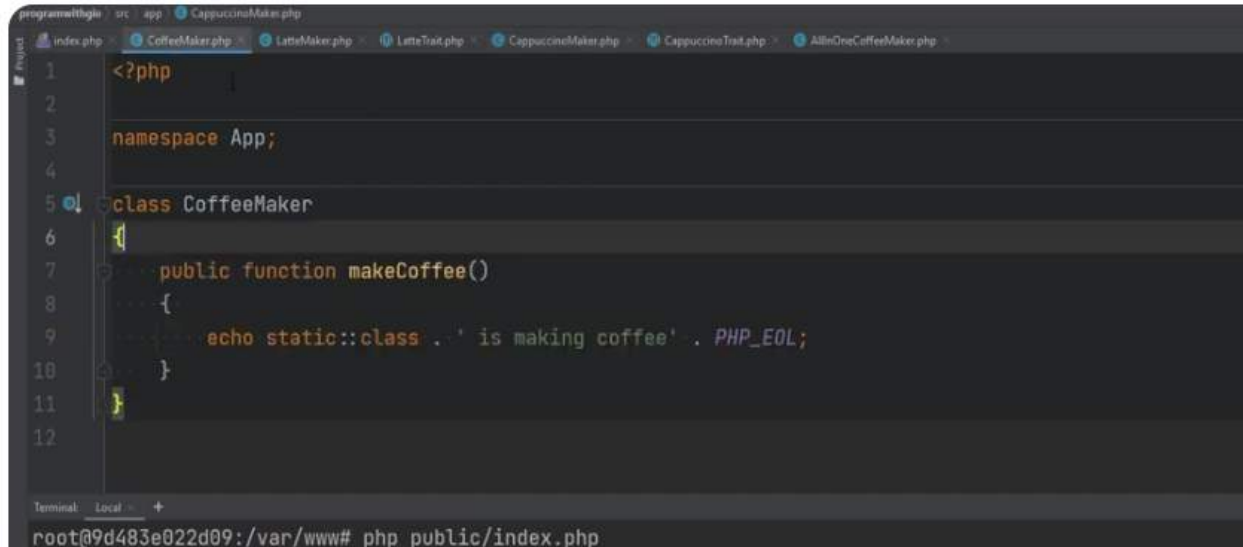
```
programwithglo src app CoffeeMaker.php
Project index.php CoffeeMaker.php LatteMaker.php LatteTrait.php CappuccinoMaker.php CappuccinoTrait.php AllInOneCoffeeMaker.php
1 <?php
2
3 namespace App;
4
5 class CoffeeMaker
6 {
7     public function makeCoffee()
8     {
9         echo static::class . ' is making coffee' . PHP_EOL;
10    }
11 }
12
```

```
programwithglo src app LatteTrait.php
Project index.php CoffeeMaker.php LatteMaker.php LatteTrait.php CappuccinoMaker.php CappuccinoTrait.php AllInOneCoffeeMaker.php
1 <?php
2
3 namespace App;
4
5 trait LatteTrait
6 {
7     private string $milkType = 'whole-milk';
8
9     final public function makeLatte()
10    {
11        echo __CLASS__ . ' is making latte with ' . $this->milkType . PHP_EOL;
12    }
13
```

```
programwithglo src app LatteMaker.php
Project index.php CoffeeMaker.php LatteMaker.php LatteTrait.php CappuccinoMaker.php CappuccinoTrait.php AllInOneCoffeeMaker.php
1 <?php
2
3 namespace App;
4
5 class LatteMaker extends CoffeeMaker
6 {
7     use LatteTrait;
8
9     public function makeLatte()
10    {
11        echo 'MAKING LATTE' . PHP_EOL;
12    }
13 }
```

+ Now, this same restriction does not apply when final method is defined within the trait. Let's remove this from here. Let's remove the final keyword from here, and let's instead make the make latte method final. Within the LatteMaker, we can now actually redefine that make latte method and echo out making latte and run the code and it works. It is allowing us to redefine a final method. As you can see, when we call make latte on a LatteMaker object, it's printing making latte instead of printing latte maker is making latte with whole milk. Another thing I don't like is that you should not be able to change the method visibility. It makes maintenance and testing complicated. Why would you define a method as protected or private only to change its visibility to public? That should not even be allowed, in my opinion, and it should result in fatal error.

+ Tuy nhiên, hạn chế này không áp dụng khi phương thức final được định nghĩa trong trait. Hãy loại bỏ điều này từ đây. Hãy loại bỏ từ khóa final từ đây và thay vào đó hãy làm cho phương thức make latte cuối cùng. Bên trong LatteMaker, chúng ta có thể thực sự định nghĩa lại phương thức make latte đó và in ra making latte và chạy mã và nó hoạt động. Nó cho phép chúng ta định nghĩa lại một phương thức cuối cùng. Như bạn có thể thấy, khi chúng ta gọi make latte trên một đối tượng LatteMaker, nó in ra making latte thay vì in ra latte maker is making latte with whole milk. Một điều tôi không thích là bạn không nên có khả năng thay đổi khả năng hiển thị của phương thức. Điều này làm cho việc bảo trì và kiểm tra phức tạp hơn. Tại sao bạn lại định nghĩa một phương thức là protected hoặc private chỉ để thay đổi khả năng hiển thị của nó thành public? Theo ý kiến của tôi, điều đó không nên được cho phép và nó nên gây ra lỗi chết người.

A screenshot of a code editor with a dark theme. The editor shows a PHP file with the following code:

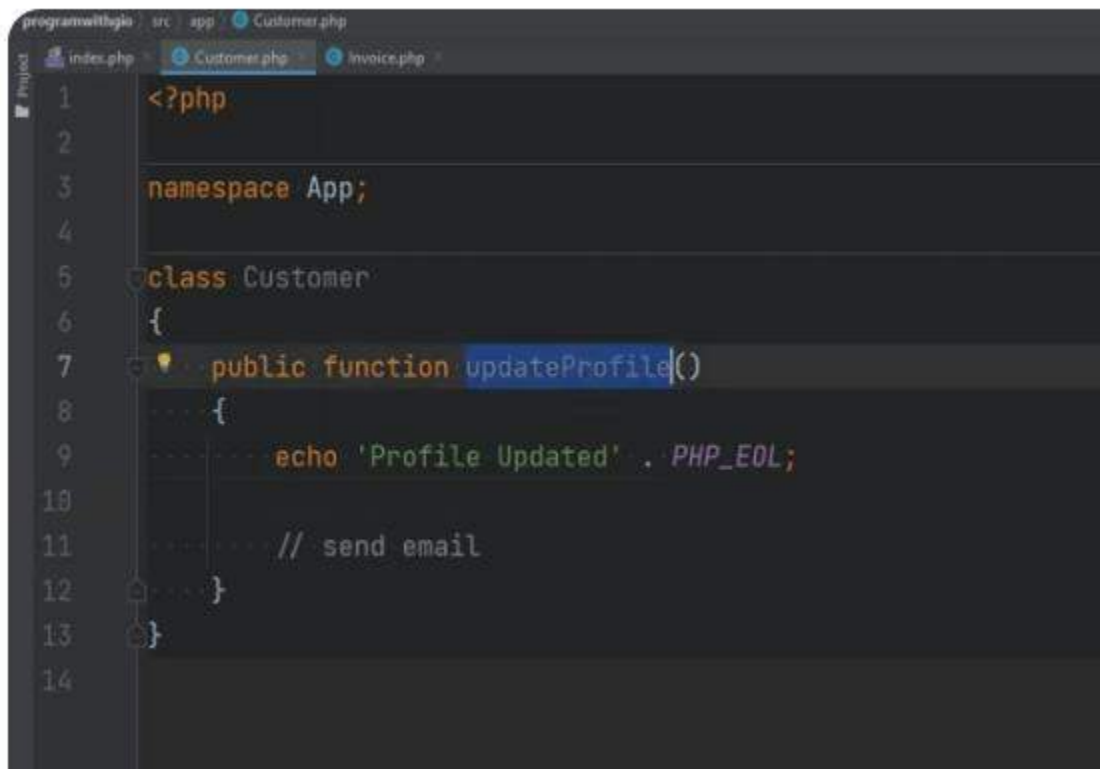
```
1 <?php
2
3 namespace App;
4
5 class CoffeeMaker
6 {
7     public function makeCoffee()
8     {
9         echo static::class . ' is making coffee' . PHP_EOL;
10    }
11 }
12
```

The editor has a sidebar on the left with a file explorer showing several files: index.php, CoffeeMaker.php, LatteMaker.php, LatteTrait.php, CappuccinoMaker.php, CappuccinoTrait.php, and AllInOneCoffeeMaker.php. At the bottom, there is a terminal window showing the command: `root@9d483e022d09:/var/www# php public/index.php`

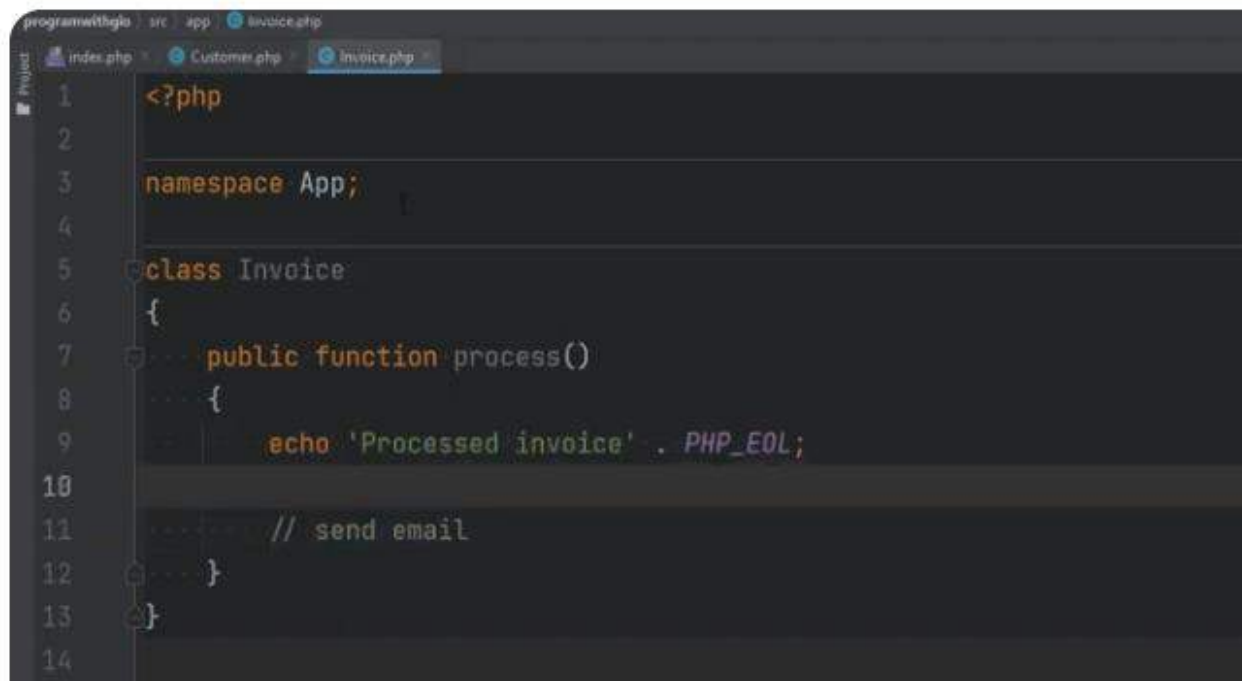
+ For example, in The Cappuccino Maker, I'm able to change the visibility of make cappuccino to public. There are a few other things that I don't like about it, but I don't want to talk about it a lot. I don't want to make this video too long. With all that being said, traits are still very powerful, and with all that power comes great responsibility. In my opinion, you should use traits to simply reduce code duplication. All right, before wrapping up, I want to show you another more useful example than Coffee Makers. In this example where we use the Coffee Makers, we used single inheritance, right? Because we're extending Coffee Maker in the Latte Maker and we're extending Coffee Maker in the Cappuccino Maker and in the

All-in-One Coffee maker. We are making use of single inheritance. All these classes are somewhat related to each other because they're all essentially Coffee Makers. Traits also allow non-related or otherwise independent classes share common functionality.

+Ví dụ, trong lớp CappuccinoMaker, tôi có khả năng thay đổi khả năng hiển thị của make cappuccino thành public. Còn một số điều khác mà tôi không thích về nó, nhưng tôi không muốn nói nhiều về điều đó. Tôi không muốn làm video này quá dài. Với tất cả những điều được nói ở trên, traits vẫn rất mạnh mẽ và với quyền lực đó đi kèm sự trách nhiệm lớn. Theo ý kiến của tôi, bạn nên sử dụng traits để đơn giản hóa việc trùng lặp mã. Đúng rồi, trước khi kết thúc, tôi muốn cho bạn xem một ví dụ hữu ích hơn về việc sử dụng traits so với Coffee Makers. Trong ví dụ này khi chúng ta sử dụng Coffee Makers, chúng ta đã sử dụng kế thừa đơn, phải không? Bởi vì chúng ta mở rộng từ Coffee Maker trong LatteMaker và chúng ta mở rộng từ Coffee Maker trong CappuccinoMaker và trong All-in-One Coffee maker. Chúng ta đang sử dụng kế thừa đơn. Tất cả các lớp này có mối quan hệ với nhau vì chúng đều là Coffee Makers. Traits cũng cho phép các lớp không liên quan hoặc độc lập khác chia sẻ chức năng chung.

A screenshot of a code editor window titled 'programwithglo' with tabs for 'src', 'app', 'Customer.php', 'index.php', 'Customer.php', and 'Invoice.php'. The 'Customer.php' tab is active, showing the following PHP code:

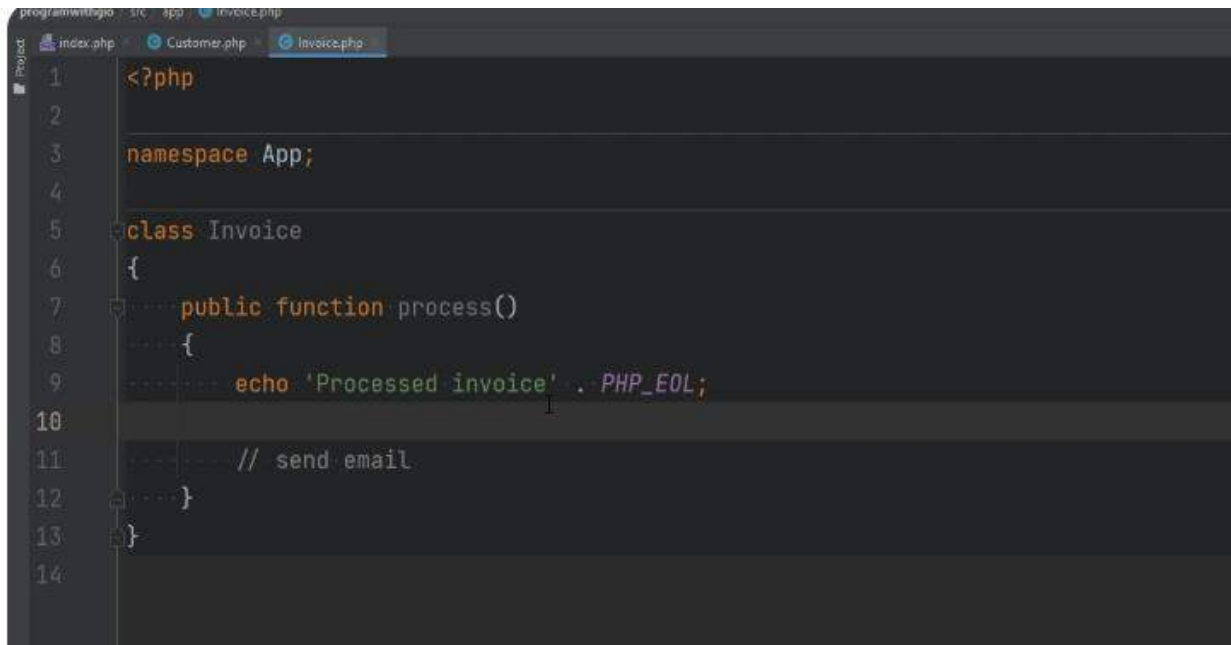
```
1 <?php
2
3 namespace App;
4
5 class Customer
6 {
7     public function updateProfile()
8     {
9         echo 'Profile Updated' . PHP_EOL;
10
11         // send email
12     }
13 }
14
```

A screenshot of a code editor window with a dark theme. The title bar shows 'programerthgio | src | app | Invoice.php'. The editor has three tabs: 'index.php', 'Customer.php', and 'Invoice.php'. The 'Invoice.php' tab is active, showing PHP code. The code starts with a PHP opening tag, followed by a namespace declaration 'namespace App;'. Then, a class 'Invoice' is defined with an opening curly brace. Inside the class, there is a public function 'process()' with its own opening curly brace. The function body contains an 'echo' statement that outputs 'Processed invoice' followed by a newline character (PHP_EOL). Below the echo statement is a comment '// send email'. The function is closed with a closing curly brace, and the class is closed with a final closing curly brace. Line numbers 1 through 14 are visible on the left side of the editor.

```
1 <?php
2
3 namespace App;
4
5 class Invoice
6 {
7     public function process()
8     {
9         echo 'Processed invoice' . PHP_EOL;
10
11         // send email
12     }
13 }
14
```

+ Let's close all of this out and let's say that we had a customer and invoice classes. A customer class has update profile method which simply prints out profile updated and then we need to send an email and then we have an invoice class that processes the invoice, which simply prints out the processed invoice, and then we also need to send the email.

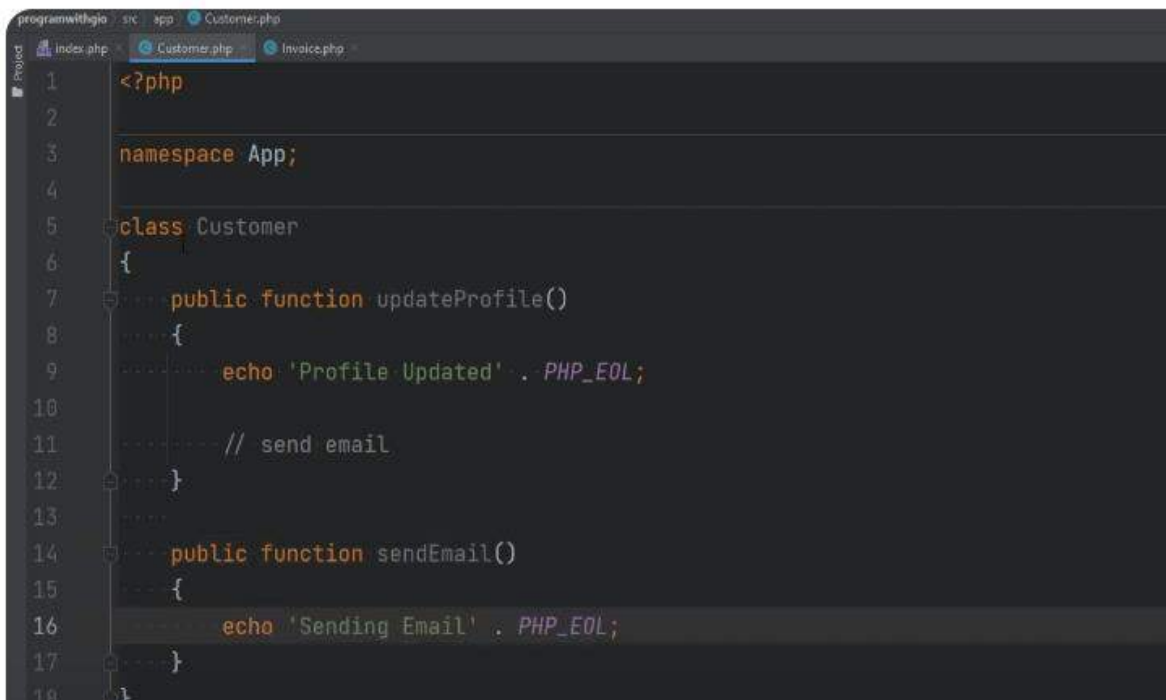
+ Hãy đóng tất cả và hãy nói rằng chúng ta có một lớp Customer và Invoice. Lớp Customer có phương thức update profile đơn giản in ra 'profile updated' và sau đó chúng ta cần gửi email. Sau đó, chúng ta có lớp Invoice xử lý hóa đơn, đơn giản chỉ in ra 'processed invoice' và sau đó chúng ta cũng cần gửi email.



```
1 <?php
2
3 namespace App;
4
5 class Invoice
6 {
7     public function process()
8     {
9         echo 'Processed invoice' . PHP_EOL;
10
11         // send email
12     }
13 }
14
```

+ As you notice, customer and invoice are not exactly related to each other. Invoice is not a customer and customer is not an invoice. But both of these classes have methods that need to send an email. Sending email could be same for both of the methods. One way we would add the send email functionality would be to add the send email code to both of these methods, but that would introduce code duplication again.

+ Như bạn đã nhận thấy, khách hàng và hóa đơn không thực sự liên quan đến nhau. Hóa đơn không phải là khách hàng và khách hàng không phải là hóa đơn. Nhưng cả hai lớp này đều có các phương thức cần gửi email. Việc gửi email có thể giống nhau đối với cả hai phương thức.



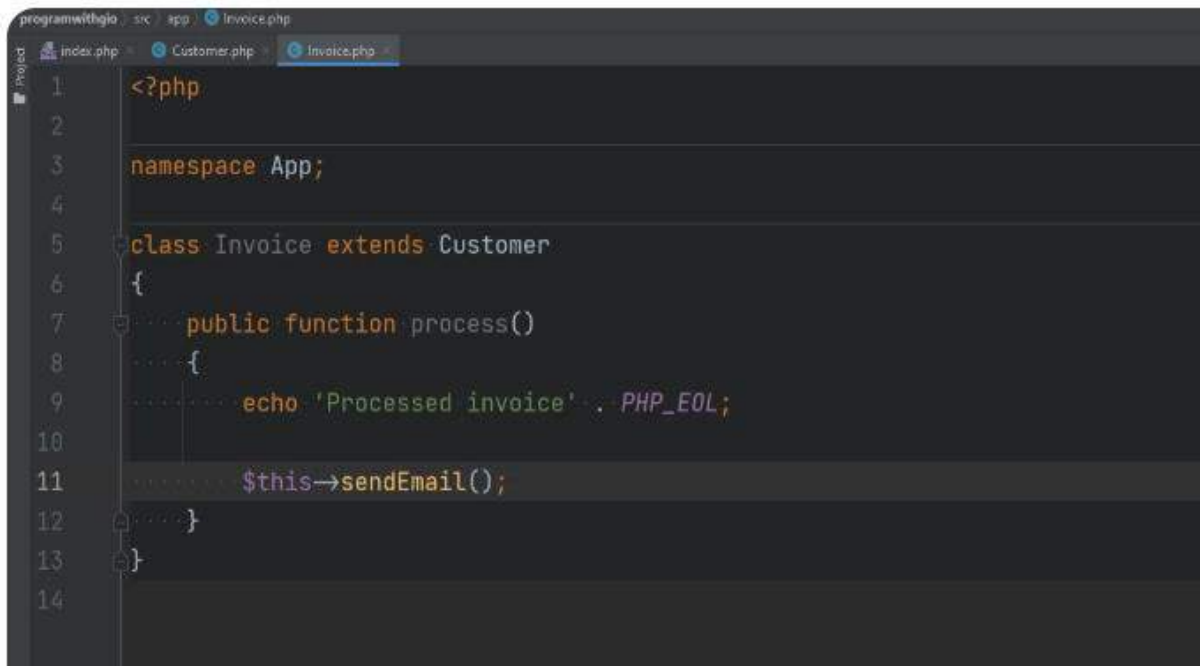
```

1  <?php
2
3  namespace App;
4
5  class Customer
6  {
7      public function updateProfile()
8      {
9          echo 'Profile Updated' . PHP_EOL;
10
11          // send email
12      }
13
14      public function sendEmail()
15      {
16          echo 'Sending Email' . PHP_EOL;
17      }
18  }

```

+Another solution would be to add a method to the customer class, something like public function, send email, echo, sending email, and then extend customer in the invoice.

+ Một cách để thêm chức năng gửi email là thêm mã gửi email vào cả hai phương thức này, nhưng điều đó sẽ lại dẫn đến việc sao chép mã.



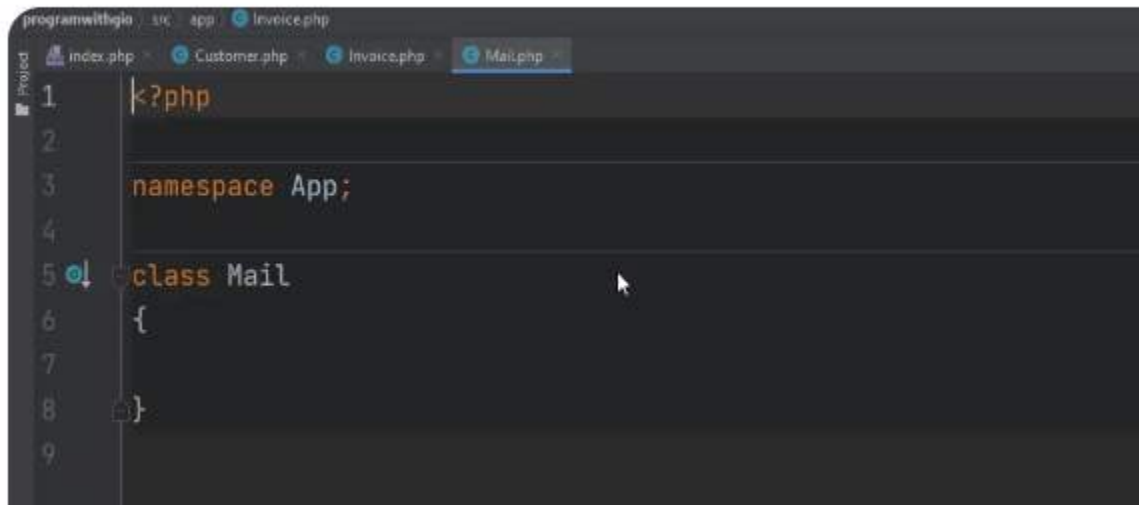
```

1  <?php
2
3  namespace App;
4
5  class Invoice extends Customer
6  {
7      public function process()
8      {
9          echo 'Processed invoice' . PHP_EOL;
10
11          $this->sendEmail();
12      }
13  }
14

```

+ Then within the process, we could simply do this, send email. But this also does not make sense because invoice is not a customer. We are using the inheritance in the wrong way. Notice how it does not pass the is a relationship check that we discussed in the inheritance lesson. This is not a good solution.

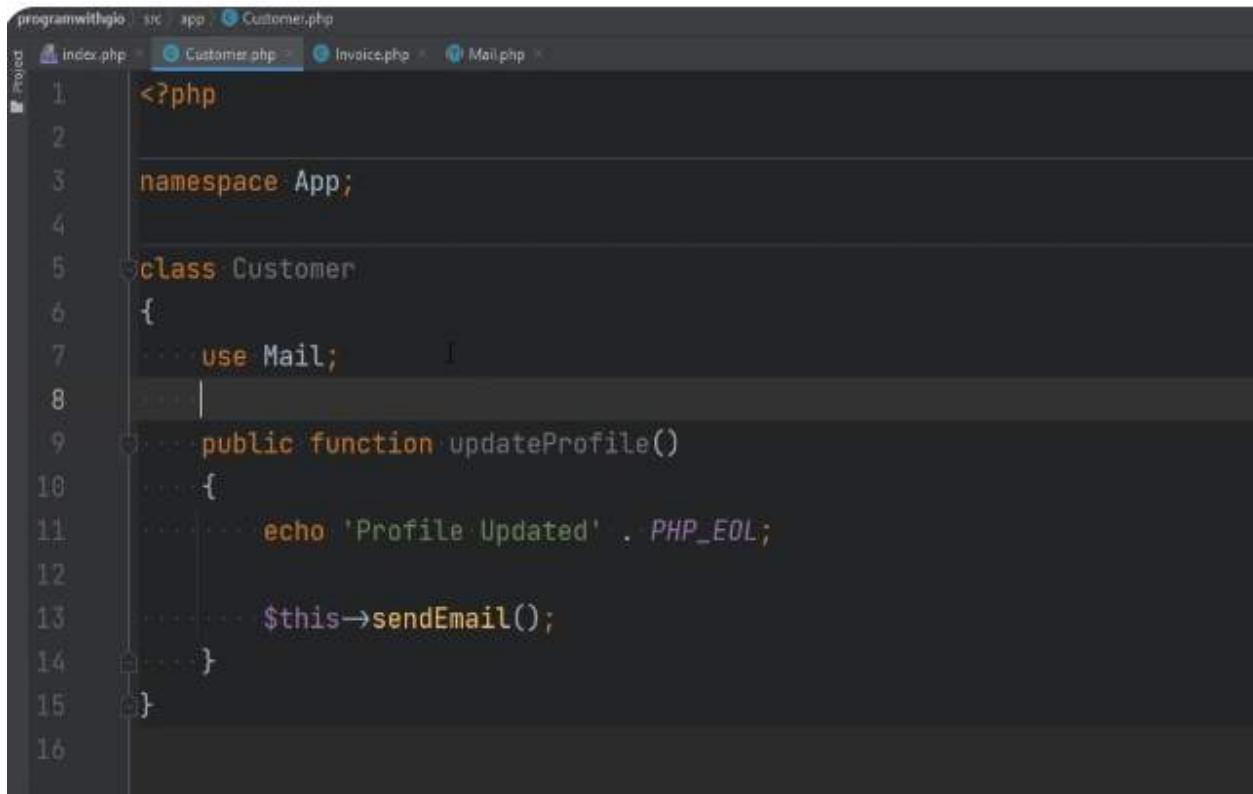
+ Một giải pháp khác là thêm một phương thức vào lớp khách hàng, thứ gì đó như public function, send email, echo, sending email, và sau đó mở rộng khách hàng trong hóa đơn. Sau đó, trong quá trình xử lý, chúng ta có thể chỉ cần thực hiện điều này, gửi email. Nhưng điều này cũng không có ý nghĩa vì hóa đơn không phải là khách hàng. Chúng ta đang sử dụng kế thừa theo cách sai. Hãy chú ý cách nó không vượt qua bài kiểm tra mối quan hệ là một mà chúng ta đã thảo luận trong bài học về kế thừa. Đây không phải là một giải pháp tốt.



```
programwithgin : src : app : Invoice.php
index.php Customer.php Invoice.php Mail.php
1 <?php
2
3 namespace App;
4
5 class Mail
6 {
7
8 }
9
```

+ Another solution would be to create a mail class and move the Send Email method to the Mail class and then extend both invoice and customers with the Mail class. Let's add that class and let's move the method there and let's also extend it in the customer and this would surely work.

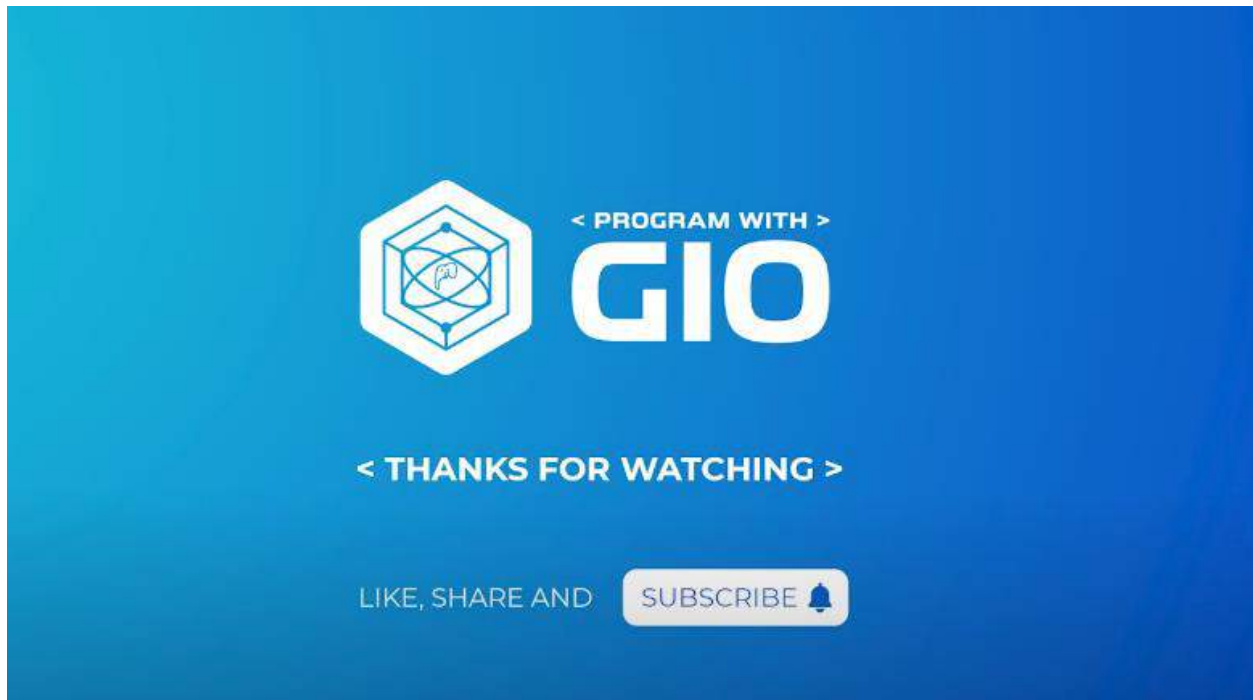
+ Một giải pháp khác là tạo một lớp thư và di chuyển phương thức Gửi email sang lớp Thư và sau đó mở rộng cả hóa đơn và khách hàng với lớp Thư. Hãy thêm lớp đó và di chuyển phương thức đó và chúng ta cũng hãy mở rộng nó trong khách hàng và điều này chắc chắn sẽ hoạt động.



```
1 <?php
2
3 namespace App;
4
5 class Customer
6 {
7     use Mail;
8
9     public function updateProfile()
10    {
11        echo 'Profile Updated' . PHP_EOL;
12
13        $this->sendEmail();
14    }
15 }
16
```

+ We could do this, Send Email here as well and this would work. But this is also the wrong use of inheritance for the same reason as before. It does not pass the is a relationship. Customer is not a male and invoice is not a male either. Instead, we need a way to share common functionality between such two independent classes. This is where traits can help. Instead of the male class, we can have male trait. We can change this from being class to be trait. Then with an invoice instead of extending, we can simply use mail and we can do the same thing here. Now this will work. We have avoided the code duplication and we've also avoided the wrong use of inheritance. Instead, we are making a good use of traits to reduce the code duplication. This is it for this lesson.

+ Chúng ta có thể làm điều này, Gửi email ở đây cũng vậy và điều này sẽ hoạt động. Nhưng đây cũng là cách sử dụng kế thừa sai vì cùng lý do như trước. Nó không vượt qua mối quan hệ là một. Khách hàng không phải là nam và hóa đơn cũng không phải là nam. Thay vào đó, chúng ta cần một cách để chia sẻ chức năng chung giữa hai lớp độc lập như vậy. Đây là nơi mà các đặc điểm có thể giúp đỡ. Thay vì lớp nam, chúng ta có thể có đặc điểm nam. Chúng ta có thể thay đổi điều này từ lớp thành đặc điểm. Sau đó, với hóa đơn thay vì mở rộng, chúng ta có thể chỉ cần sử dụng thư và chúng ta có thể làm điều tương tự ở đây. Bây giờ điều này sẽ hoạt động. Chúng ta đã tránh được việc sao chép mã và chúng ta cũng đã tránh được việc sử dụng kế thừa sai cách. Thay vào đó, chúng ta đang sử dụng các đặc điểm một cách hiệu quả để giảm thiểu việc sao chép mã. Đây là bài học cho bài học này.



+Thank you so much for watching. If you enjoyed this lesson, please give it a thumbs up, share, and subscribe. I'll see you next time.

+ Cảm ơn bạn rất nhiều vì đã xem. Nếu bạn thích bài học này, vui lòng cho nó một lượt thích, chia sẻ và đăng ký. Hẹn gặp lại bạn lần sau.

