

Forum:[Hack Forums](#)

Thread: [\[Infamous/Antagonism\] PHP:OOP - Serialization of Objects and Variable Variables](#)

Keeper - April 2, 2013, 1:39 pm

Introduction:

First off - serialization. Serialization is the conversion of objects or pieces of data into a format that is handling their type and structure and is being stored temporarily so as to called later on in the php code. Whenever the execution of the php code ends, every object is being destroyed (in some cases they are being destroyed even before that). Which actually means that every information is being erased. So in order to get the object in the state that it was at a particular time, we'd need to either, write the whole code again or just serialize the object.

Some of the benefits you could have using serialization is:

A method of persisting objects

- The property for storing objects and maintaining their type

A method for distributing objects

- This is basically a module that works simultaneously with another one but they are both on located on different computers that share the same network.

Usage in remote procedure calls

- The ability to execute a code in a range of virtual addresses sharing the same network

How it works?

Let's say we have a simple class named User.php that has two properties Name and Age & setName and setAge.

PHP Code

```
<?php
```

```
class User {  
    private $name;  
    private $age;  
  
    public function getName() {  
        return $this->name;  
    }  
  
    public function setName($name) {
```

```

    $this->name = $name;
}

public function getAge() {
return $this->age;
}

public function setAge($age) {
    $this->age = $age;
}

}

?>

```

A pretty basic class that we can append something to and call it afterwards - nothing complicated. Now let's get an idea of a script that would append certain values to it:

PHP Code

```
<?php
```

```

include 'User.php';
$a=new User();
$a->setAge(20);
$a->setName('Keeper');

```

```

echo $a->getName();

```

```
?>
```

We make a new instance for the class with the first variable \$a and define the age and name for the class we've included. Now let's set some default values to the variables for name and age in our class User().

PHP Code

```
<?php
```

```

class User {
    private $name='Infamous';
    private $age='99';

    public function getName() {
return $this->name;
}

    public function setName($name) {
    $this->name = $name;
}

    public function getAge() {
return $this->age;
}

    public function setAge($age) {
    $this->age = $age;
}
}

```

```
}
```

```
}
```

```
?>
```

Upon executing index.php which has User.php included, we get the result 'Keeper'. So there is nothing new or awkward. The thing is that the objects are being destroyed and we get returned the last getName(); value which is the string 'Keeper'.

So let's actually see how this goes. When we serialize an object we convert it to a string that we can manipulate in numerous ways. Afterwards, we just unserialize the string and we have returned the value of the object as it was till the time we serialized it.

PHP Code

```
<?php
```

```
include 'User.php';  
$a=new User();  
$a->setAge(20);  
$a->setName('Keeper');
```

```
echo serialize($a);
```

```
?>
```

Echoing out the variable that hold the function, outputs this:

```
Code O:4:"User":2:{s:10:"Username";s:6:"Keeper";s:9:"Userage";i:20;}
```

That is the string value of the object. Every symbol within it has a huge importance to PHP as a whole.

Let's say we serialize it like so and append the output to a variable called \$b. Now we can take that object after 5 minutes, 1 year or a decade and we will still unserialize it successfully.

PHP Code

```
<?php
```

```
include 'User.php';  
$a=new User();  
$a->setAge(20);  
$a->setName('Keeper');
```

```
$b = serialize($a);  
$c = unserialize($b);
```

```
echo $c->getName();
```

```
?>
```

We get 'Keeper' as the output. That indicates that the object has successfully been converted to a string and then again, reconverted to an object. If we get to define the value of the object like that:

PHP Code

```
<?php
```

```
..
```

```
$c = unserialize($b);  
$c->setName('Infamous');
```

```
echo $c->getName();
```

```
?>
```

Logically, we get outputted the string 'Infamous'. Also another important thing to know is that the serialization doesn't work with source code, it uses its data - not itself.

As confusing as it may seem, there is such thing. That's the property of PHP to append the value of a variable to another variable. I'll write the classes again and start from scratch, meanwhile I'll be explaining everything there is to know. Subsequently, we'll get to the variable variables, but I first wanna make sure you know what we are talking about, the concept of variable variables and how it could come in handy for a project development or whatever. We'll write a class for a validator of a non-existent form and output default results just as a proof of concept.

Let's give it some data and define a couple of rules based on which it will validate the input. First making a method for the data that should be validated. And because we gonna use it inside of the object, let's make a property which would be an array and the place where we'll store the data. Also a method which will set the rules.

PHP Code

```
<?php
```

```
class Validator {
```

```
    private $data=array();  
    private $rules=array();  
    public function setData($params){  
$this->data=$params;  
    }  
  
    public function setRules($field,$type,$param){  
$this->rules[$field][]=array('type'=>$type,'param'=>$param);  
return $this;  
    }
```

```
    public function validate(){  
  
    }  
}
```

```
?>
```

Now to the index.php file. In it we'll include the Validator.php, make an instance for it and set the rules for the validation. Also notice this method and the returning of itself from the upper code:

PHP Code

```
<?php
```

```
public function setRules($field,$type,$param){  
$this->rules[$field][]=array('type'=>$type,'param'=>$param);  
return $this;
```

```
}
```

It's a method called chaining and it happens when we call a method from the same method from the same object. Okay let's put an example to make it more clear.

PHP Code

```
<?php
```

```
    return $this;
```

```
?>
```

Now, the idea is not to call the variable that holds the reference to the object every time. Instead of this, we call the method which calls an instance for it and so on. So the below example is that of a multiple variable defining and not the chain method.

PHP Code

```
<?php
```

```
include 'Validator.php';
$v=new Validator();
$v->setData(array('username'=>'Keeper','pass'=>'Antagonism'));
$v->setRules('username', 'min_length', 4);
$v->setRules('username', 'max_length', 8);
$v->setRules('pass', 'min_length', 4);
```

```
$v->validate();
```

```
?>
```

This is an example of the chain method where we call an object and afterwards the instance for it.

PHP Code

```
<?php
```

```
include 'Validator.php';
$v=new Validator();
$v->setData(array('username'=>'Keeper','pass'=>'Antagonism'));
$v->setRules('username', 'min_length', 4)->
    setRules('username', 'max_length', 8)->
    setRules('pass', 'min_length', 4);
```

```
$v->validate();
```

```
?>
```

Technically, there is no difference whether you write your code like so or use multiple variable defining. I personally prefer the chain method cuz it's tidier and the code is better structured. Now let's see what we have as the current rules we've applied. I'm gonna simply dump the array like so:

PHP Code

```
<?php
```

```
class Validator {
```

```
    private $data=array();
    private $rules=array();
```

```

    public function setData($params){
$this->data=$params;
    }

    public function setRules($field,$type,$param){
$this->rules[$field][]=array('type'=>$type,'param'=>$param);
return $this;
    }

    public function validate(){
echo '<pre>'.print_r($this->rules, true).'

```

?>

We get this:

Code Array

```

(
  [username] => Array
  (
    [0] => Array
    (
      [type] => min_length
      [param] => 4
    )

    [1] => Array
    (
      [type] => min_length
      [param] => 8
    )

  )

  [pass] => Array
  (
    [0] => Array
    (
      [type] => min_length
      [param] => 4
    )

  )

)

```

In the field of username, it has to check for a minimum length of 4 symbols and a maximum length of 8 symbols and the field for password should be just checked for the minimum length of 4. Outputting the array value gets you more aware of what we currently have as a code and the result.

PHP Code

```
<?php
```

```

class Validator {

    private $data=array();
    private $rules=array();
    public function setData($params){
$this->data=$params;
    }

    public function setRules($field,$type,$param){
$this->rules[$field][]=array('type'=>$type,'param'=>$param);
return $this;
    }

    public function validate(){
        foreach ($this->rules as $k => $v) {
            foreach ($v as $vv) {
                if($vv['type']=='min_length'){
$this->min_length($k, $vv['param']);
                }
                else if($vv['type']=='max_length'){
$this->max_length($k, $vv['param']);
                }
            }
        }
    }

    private function min_length($field,$param){
        if(mb_strlen($this->data[$field])>$param){
            echo $field.' min_length is OK<br>';
        }
        else{
            echo $field.' min_length is NOT OK<br>';
        }
    }

    private function max_length($field,$param){
        if(mb_strlen($this->data[$field])<$param){
            echo $field.' max_length is OK<br>';
        }
        else{
            echo $field.' max_length is NOT OK<br>';
        }
    }
}

```

?>

The first foreach() cycle holds the rules and the \$k variable is the field we want to call. Now let's make a checks for all fields. We need to create a method for each and every one of them and loop them in an if() cycle so as to perform the validation. With the first if() we check if the type of the check is minimum length, then we call the method min_length and make it accept the name of the field and what parameters it should accept. Meanwhile, with the second if() cycle we check if the length is larger than the parameter's value and if so we return true, if not - false. We do the same for the maximum length as well. So to check whether this code is in fact functional, let's echo an

example for every check.

Upon execution of this code, we get the following:

Code username min_length is OK
username max_length is NOT OK
pass min_length is OK

Now going through our index.php where we've defined our rules for the validation, we have: 'Keeper' as username and 'Antagonism' as password. Since the string is over 4 characters, the first check returns true. It is also true that the maximum length of the string is not exceeding 8 characters and the same goes for the password value.

Let's go about modifying the function validate() so as to be using a variable variable and to explain you how it goes.

PHP Code

```
<?php
```

```
public function validate(){
    foreach ($this->rules as $k => $v) {
        foreach ($v as $vv) {
            $this->max_length($k, $vv['param']);
        }
    }
}
```

```
?>
```

Which is the same if we make it look like so:

PHP Code

```
<?php
```

```
public function validate(){
    foreach ($this->rules as $k => $v) {
        foreach ($v as $vv) {

            $this->{$vv['type']}($k, $vv['param']);
        }
    }
}
```

```
?>
```

In the first iteration, the line:

PHP Code

```
<?php
```

```
$this->{$vv['type']}
```

hold the string min_length. When appending a dollar sign to it, PHP will execute the object or property with that name. Actually, we don't even know what the variable variable is. Whenever we add a \$, the variable that hold the value, that very value becomes the name of the field. So in every iteration of the script, this will go through min_length, max_length and so on.

The idea behind variable variables is that we can take the name of an object or property from a variable's value and call it the same way.

Conclusion:

This was the second tutorial from my series on object-oriented programming in PHP. It happened to be larger than I expected. Got way deeper than I should but, the more - the better. I hope at least one person reads it and gets something out of it. If anyone has any questions or issues with the class and methods, leave a reply here or PM me. Thanks for reading!

amus3d - April 2, 2013, 1:41 pm

STOP BEING SMART.

Like, seriously, stop. Please.

-?Divine- - April 2, 2013, 1:42 pm

Now that's one big fat tutorial. Good job Keeper.

lcYSeptember - April 2, 2013, 11:00 pm

I already knew about this, but I know a lot of people don't.

Great tutorial! Good job!

Keeper - April 3, 2013, 4:53 am

Thanks for the feedback, I'll probably post the next one from the series today.

Money Maker Mike - April 3, 2013, 4:54 am

Great tutorial! keep it up, Keeper.

Dr. Sheldon Cooper - April 3, 2013, 4:57 am

Great tutorial thanks for the contribution.

?Dante - April 3, 2013, 9:48 am

Nice thread! I don't know PHP though.

Keeper - April 3, 2013, 1:40 pm

Thanks. The 3rd one from the series coming out in some minutes now.

Gee4rce - June 8, 2013, 7:11 pm

Thanks a lot for this well written HQ Tutorial,

eventhough i cant code this yet, i still understand the theory of it and it helps a lot in general understanding!!

Keeper Wrote:"I hope at least one person reads it and gets something out of it."

Yes i did get much out of it, thank you!! ?

Greets, Gee.