New Language Features

Lambda Function | Closure | Functor | Trait

Sebastian Bergmann

March 4th 2009

Who I Am

- Sebastian Bergmann
- Involved in the PHP project since 2000
- Creator of PHPUnit
- Co-Founder and Principal Consultant with thePHP.cc

New PHP Language Features

- PHP 5.3
  - Lambda Functions
  - Closures
  - Functors
  - Namespaces
  - `goto`
- Future
  - Traits

Lambda Functions

- Anonymous functions that
- are declared on-the-fly
- can be assigned to a variable
- and passed to other functions

Lambda Functions

Usage

```
<?php
$lambda = function() { print 'Hello World!'; };
?>
```

Lambda Functions

Usage

```
<?php
$lambda = function() { print 'Hello World!'; };

$lambda();
?>
```

```
Hello World!
```

Lambda Functions

Usage

```
<?php
$lambda = function() { print 'Hello World!'; };

call_user_func($lambda);
?>
```

```
Hello World!
```

Lambda Functions

Usage

```php
<?php
$lambda = function() { print 'Hello World!'; };

call_user_func_array($lambda, array());
?>
```

```
Hello World!
```

Lambda Functions

Usage Scenarios

```
<?php
$list = array(22, 4, 19, 78);

usort(
  $list,
  function ($a, $b) {
      if ($a == $b) return 0;
      return ($a < $b) ? -1 : 1;
  }
);

print_r($list);
?>

Array
(
    [0] => 4
    [1] => 19
    [2] => 22
    [3] => 78
)
```

Lambda Functions

Usage Scenarios

```
<?php
print_r(
  array_map(
    function ($n) { return($n * $n * $n); },
    array(1, 2, 3, 4, 5)
  )
);
?>

Array
(
    [0] => 1
    [1] => 8
    [2] => 27
    [3] => 64
    [4] => 125
)
```

Lambda Functions

Prior to PHP 5.3

```
<?php
$lambda = create_function('', 'print "Hello World!";');

$lambda();
?>
```

```
Hello World!
```

Lambda Functions

Prior to PHP 5.3

```
<?php
$lambda = create_function('', 'print "Hello World!";');

call_user_func($lambda);
?>
```

```
Hello World!
```

Lambda Functions

Prior to PHP 5.3

```
<?php
$lambda = create_function('', 'print "Hello World!";');

call_user_func_array($lambda, array());
?>
```

```
Hello World!
```

Closures

- Anonymous functions that
- are declared on-the-fly
- can be assigned to a variable,
- passed to other functions,
- and remember what happens around them

Closures

Lexical Variables

```php
<?php
$string  = 'Hello World!';
$closure = function() use ($string) { print $string; };

$closure();
?>
```

```
Hello World!
```

Closures

Lexical Variables

```php
<?php
function getClosure()
{
    $string  = 'Hello World!';
    return function() use ($string) { print $string; };
}

$closure = getClosure();

$closure();
?>
```

```
Hello World!
```

Closures

Lexical Variables with reference

```
<?php
$x = 1;
$closure = function() use ($x) { print $x . "\n"; };
$closure();
$x = 2;
$closure();

print "\n";

$x = 1;
$closure = function() use (&$x) { print $x . "\n"; };
$closure();
$x = 2;
$closure();
?>
```

```
1
1

1
2
```

Closures

Reflection API

```
<?php
$closure   = function($a, $b) { return $a + $b; };
$reflector = new ReflectionFunction($closure);
print $reflector;
?>

Closure [ <user> function {closure} ] {
  @@ /home/sb/closure_reflection.php 2 - 2

  - Parameters [2] {
    Parameter #0 [ <required> $a ]
    Parameter #1 [ <required> $b ]
  }
}
```

Functors

- Allow an object to be invoked or called as if it were an ordinary function
- Also called *function objects*, *functionals* or *functionoids*

Functors

Usage

```
<?php
class Example {
    public function __invoke() {
        print __METHOD__ . "\n";
    }
}

$object = new Example;
$object();
?>
```

```
Example::__invoke
```

Traits

- New unit of reuse
- Provide structure, modularity and reusability within classes
- Less complex than Multiple Inheritance

Inheritance

- Single Inheritance
  - A class can inherit interface and implementation from a single parent class

Inheritance

- Single Inheritance
  - A class can inherit interface and implementation from a single parent class
- Multiple Inheritance
  - A class can inherit interface and implementation from multiple parent classes

Inheritance

- Single Inheritance
  - A class can inherit interface and implementation from a single parent class
- Multiple Inheritance
  - A class can inherit interface and implementation from multiple parent classes
- Interfaces
  - A class can inherit interface and implementation from a single parent class
  - A class can implement multiple interfaces

Inheritance

Classes have contradicting goals

- Generator of Instances
  - Must be complete
  - Must have a unique place in the class hierarchy

- Unit of Reuse
  - Should be small
  - Should be applicable at arbitrary places

Multiple Inheritance

The Diamond Problem

```
class A {
    public function method() {
    }
}
```

- Class A declares `method()`

Multiple Inheritance

The Diamond Problem

```
class A {
    public function method() {
    }
}

class B extends A {
    public function method() {
    }
}

class C extends A {
    public function method() {
    }
}
```

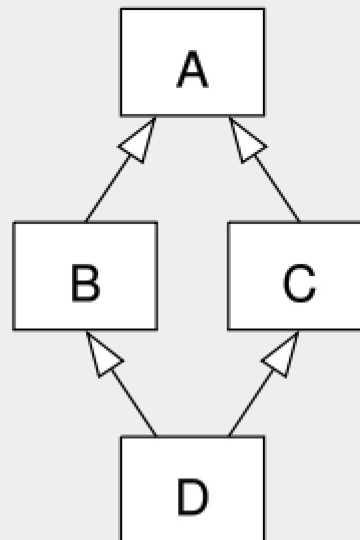
- Class A declares `method()`
- Classes B and C inherit from A and override `method()` differently

Multiple Inheritance

The Diamond Problem

```
class A {
    public function method() {
    }
}

class B extends A {
    public function method() {
    }
}

class C extends A {
    public function method() {
    }
}

class D extends B, C {
}
```

- Class A declares `method()`
- Classes B and C inherit from A and override `method()` differently
- Class D inherits from both B and C and does not override `method()`

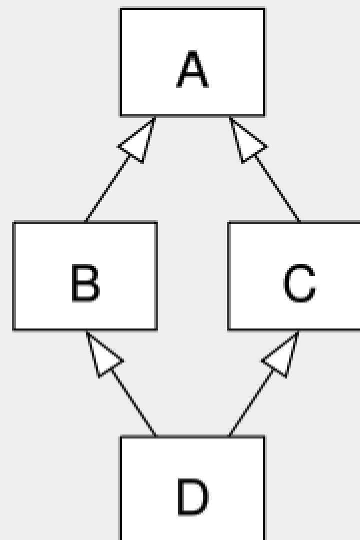


Multiple Inheritance

The Diamond Problem

```
class A {
    public function method() {
    }
}

class B extends A {
    public function method() {
    }
}

class C extends A {
    public function method() {
    }
}

class D extends B, C {
}

$d = new D;
$d->method();
```

- Class A declares `method()`
- Classes B and C inherit from A and override `method()` differently
- Class D inherits from both B and C and does not override `method()`
- When we call `method()` on an object of D, which implementation should be called?



Traits

- New unit of reuse
  - Provide structure, modularity and reusability within classes
  - Can be incomplete

Traits

- New unit of reuse
  - Provide structure, modularity and reusability within classes
  - Can be incomplete
- Less complex than Multiple Inheritance
  - The typical problems associated with multiple inheritance (as well as mixins) are avoided

Traits

- New unit of reuse
  - Provide structure, modularity and reusability within classes
  - Can be incomplete
- Less complex than Multiple Inheritance
  - The typical problems associated with multiple inheritance (as well as mixins) are avoided
- Flattening Property
  - Traits are flattened into classes at compile-time
  - No notion of traits at runtime

Traits

- Lightweight
- Stateless
- Flexible composition of behaviour (methods) into classes
  - Overcome some limitations of single inheritance by enabling the reuse of method sets in several independent classes across different class hierarchies
- Existing implementations
  - Perl 6, Squeak, Scala, Self, Slate, Fortress

Traits

Example

```
<?php
trait T {
    public function method() {
    }
}

class B extends A {
    use T;
}

class D extends C {
    use T;
}
?>
```

```
<?php
class B extends A {
    public function method() {
    }
}

class D extends C {
    public function method() {
    }
}
?>
```

Traits

Classes

- can implement multiple interfaces
- can extend a single parent class
- can use multiple traits

Traits

- can be composed from other traits
- can express requirements to the using class via the abstract mechanism

The End

Thank you for your interest!

These slides will be linked soon from http://sebastian-bergmann.de/

You can vote for this talk on http://joind.in/109

License

This presentation material is published under the Attribution-Share Alike 3.0 Unported license.

You are free:

- **to Share** – to copy, distribute and transmit the work.
- **to Remix** – to adapt the work.

Under the following conditions:

- **Attribution.** You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).
- **Share Alike.** If you alter, transform, or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license.

For any reuse or distribution, you must make clear to others the license terms of this work.

Any of the above conditions can be waived if you get permission from the copyright holder.

Nothing in this license impairs or restricts the author's moral rights.