What's new and exciting in PHP 5.3?

@jclermont

Brief history and status of PHP 5.3

- First released in June 2009
- Some people are shy of .0 major releases
  - PHP 5.3 is in its fourth release: 5.3.3
- Much bigger release than a normal 5.x release
  - Many 6.0 features pulled into 5.3
- Becoming available as an option on many shared hosts
  - Not a default anywhere as far as I know
- Several popular frameworks are planning 5.3-only releases
  - Symfony 2, Zend Framework 2, Lithium

Goals

- Cover three new features
  - Anonymous functions / Lambda functions / Closures
  - Namespaces
  - Late Static Binding
- Many more - possibly topics for future meetups
  - Better exceptions
  - New syntax: NOWDOC, ternary shortcut
  - SPL enhancements
  - Garbage collection
- Get you to install 5.3 and use it in production

Warning: This may get geeky

Anonymous Functions

Can you spot the difference between these two functions?

```php
1  <?php
2  function doSomething($arg1) {
3      //function logic
4  
5      //return value
6  }
```

```php
1  <?php
2  function ($arg1) {
3      //function logic
4  
5      //return value
6  }
```

Anonymous Functions

- Just like a normal function, but without a name
- Not a new concept (Javascript, Ruby, C#, Lisp)
- What problem does this solve?
- Where would you use them?

Anonymous Functions: Example

- Very handy anywhere a callback function can be supplied

```php
<?php
$integers = array(1, 2, 3, 4, 5);
$oddIntegers = array_filter(
    $integers,
    function ($i) { return $i % 2 == 1; }
);
print_r($oddIntegers);
```

- Great for one-off functions - no need to declare elsewhere
- NOT the same thing as create_function
  - much worse performance
  - no compilation to opcode

Lambda Functions

- An anonymous function assigned to a variable

```php
<?php
$integers = array(1, 2, 3, 4, 5);
$getOdds = function ($i) {
    return $i % 2 == 1;
};

$oddIntegers = array_filter(
    $integers,
    $getOdds
);
print_r($oddIntegers);
```

- A simplistic example, but it demonstrates the mechanics
  - More interesting examples to come

Closures

- Closures add *context* to lambdas/anonymous functions

```php
<?php
$integers = range(1, 20);
$getMultiples = function ($divisor) {
    return function ($i) use ($divisor) {
        return $i % $divisor == 0;
    };
};

$divisbleBy = array_filter(
    $integers,
    $getMultiples(3)
);
print_r($divisbleBy);
```

- "use" keyword
- function returns a function

Taking it to "11"

- Previous examples could be done pre-5.3 in a hackish or inelegant way
- Let's dive into something far more difficult to implement without PHP 5.3
- Y-combinator
- Memoization
- DEMO

More Use Cases and Tricks

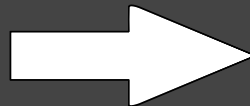
- creating a system of events and event handlers
- passing business logic to a view (MVC) from the controller
- Dependency Injection
- make your own classes behave like closures with __invoke
- new getClosure() method in Reflection classes
  - allows external execution of private methods
  - could be useful for unit testing

Namespaces

- Prior to 5.3, everything lived in the same namespace
- This resulted in collisions between names
    - Solution was to create increasingly lengthy names
- PHP 5.3 namespaces provide a better solution

```php
<?php
namespace Orion;

class Mail
{
    //class implementation
}
```



```php
<?php
$mail = new Orion\Mail;
$mail->doSomething();
```

- Namespace separator was a contentious debate

Namespaces

- Namespaces also provide a way to alias classes
- Saves typing and makes code more readable

```php
<?php
use Zend\Filter\Word\UnderscoreToSeparator as Filter_;

$filter = new Filter_;
```

Namespaces

- Multiple namespaces can be imported into your code

```php
<?php
use Zend\Log;
use Zend\Mail;

//resolves inside Zend\Log
$logger = new Writer\Db;

//resolves inside Zend\Mail
$smtp = new Transport\Smtp;
```

Namespaces

- Make sure you understand the rules of resolution
- Similar to how file paths are resolved
  - absolute versus relative
- Be aware of the global namespace
- Can also be used to override built-in functions/classes
  - "Monkey patching" can be very dangerous though
  - Use wisely

Late Static Binding

- *self* keyword refers to the class it is defined in (compile-time)
- causes difficulty when dealing with inheritance and static calls

```php
<?php
class A {
    public static function who() {
        echo __CLASS__;
    }
    public static function test() {
        self::who();
    }
}

class B extends A {
    public static function who() {
        echo __CLASS__;
    }
}

B::test();
?>
```

http://php.net/manual/en/language.oop5.late-static-bindings.php

Late Static Binding

- Solution in PHP 5.3 - the "static" keyword
- Not a new keyword, but it has a new use case

```
<?php
class A {
    public static function who() {
        echo __CLASS__;
    }
    public static function test() {
        static::who(); // Here comes Late Static Bindings
    }
}

class B extends A {
    public static function who() {
        echo __CLASS__;
    }
}

B::test();
?>
```

Late Static Binding

- Useful when extending framework classes that implement static methods
- Previously, you had to override these functions in your extended class - lots of needless repetition
- Additional discussion on when this technique is useful
  - http://stackoverflow.com/questions/87192/when-would-you-need-to-use-late-static-binding

Questions?

References

- http://fabien.potencier.org/article/17/on-php-5-3-lambda-functions-and-closures
- http://www.slideshare.net/melechi/php-53-part-2-lambda-functions-closures-presentation
- http://www.ibm.com/developerworks/opensource/library/os-php-5.3new2/index.html
- http://php100.wordpress.com/2009/04/13/php-y-combinator/
- http://en.wikipedia.org/wiki/Y_combinator
- http://matt.might.net/articles/implementation-of-recursive-fixed-point-y-combinator-in-javascript-for-memoization/
- http://nerderati.com/2009/11/the-php-5-3-y-combinator/