Functional Programming in PHP

Albert Krewinkel

Functional programming

This gives a brief overview over functional programming in PHP, including its principles, merits, challenges, and language support.

Generic properties

- Pure functions (in the mathematical sense)
- No global state
- No *side effects*

Why should we care?

Functional programming

- simplifies some design patterns,
- allows for straight-forward parallelization, and
- makes it simpler to reason about code.

- Parallelization is not important in PHP, but e.g. in JS.
- Reasoning is simplified as there are no **hidden** states. Everything is explicit.

Example: Strategy Pattern

```
interface RequestHandler { handleRequest; }
class DummyRequestHandler extends RequestHandler {…}
class HttpRequestHandler extends RequestHandler {…}

function handlePath($path, RequestHandler $handler) {…}
handlePath("/app", new HttpRequestHandler());
```

- It's a classic pattern described by the gang of four.
- Plays nicely with the open/closed principle of SOLID.

Functions in PHP

Lambda functions

```
function (int $x) {
    return 2 * $x;
};
```

Lambda functions (a.k.a. anonymous functions) were introduced in PHP 5.3.

Assigning functions to variables

```
$doubleInt = function (int x) {
    return 2 * $x;
}
```

Using functions

```
$oneDoubled = call_user_func($doubleInt, 1);
// $oneDoubled == 2
```

Closures

```
$multiplier = 3;
$scale = function ($x) use ($multiplier) {
    return $x * $multiplier;
};

echo call_user_func($scale, 4);
// 12
```

Callables

```
class Foo
{
    static function frob(int $x)
    {
        return $x * 2;
    }
}

$foo = new Foo();
```

```
$bar1 = call_user_func([$foo, 'frob'],  11.5);
$bar2 = call_user_func(['Foo', 'frob'], 11.5);
$bar3 = call_user_func(['Foo::frob'],   11.5);
```

Second and third versions only work because `frob` is static.

Callable objects

```
class Greeter {
    private $msg;
    function __construct($msg) { $this->msg = $msg; }
    function __invoke() { echo $this->msg; }
}

$greeter = new Greeter("Hello, World!");
call_user_func($greeter);
// prints "Hello, World!"
```

Working with functions in PHP

Functions as return values

```
function createScaler(int $multiplier) {
    return function ($x) use ($multiplier) {
        return $x * $multiplier;
    };
}

echo call_user_func(createScaler(3), 4);
// 12
```

`array_*` functions

- process a set of values all at once,
- allow to clearly state ones intend, and
- shield against unwanted side-effects.

All `array_*` functions can be written using `foreach`.

```
function array_map($fn, $arr) {
    $res = [];
    foreach ($arr as $a) {
        $res[] = call_user_func($fn, $a);
    }
    return $res;
}
```

`array_map`

```
$doubleInt = function(int $x) {
    return $x << 1;
};
$doubled = array_map($doubleInt, [1, 2, 3, 4, 5]);
// $doubled == [2, 4, 6, 8, 10]
```

`array_filter`

```
$even = array_filter([1, 2, 3, 4], function (int $x) {
    return ($x % 2 == 0);
});
// array_values($even) == [2, 4]
```

Removes values not satisfying the property.

Note the inverted argument order of `array_filter` compared to `array_map`.

Actual value of `$even` is `[1 => 2, 3 => 4]`.

`array_column`

Not really functional programming, but shortens common usecase of `array_map`:

```
$elements = [
  ['name' => 'Hydrogen', 'electrons' => '1s¹'],
  ['name' => 'Helium',   'electrons' => '1s²'],
  ['name' => 'Lithium',  'electrons' => '2s¹'],
];
$names1 = array_column($elements, 'name');
$names2 = array_map(
  function($e) { return $e['name']; },
  $elements
);
echo $names1 == $names2;
// 1
```

Examples

Readable code

This is the most important slide of this talk. It demonstrates sensible use cases for functional programming in PHP.

```
// Convert all strings to lowercase
array_map('strtolower', $strings);

// Remove empty or whitespace-only strings
array_filter(array_map('trim', $strings));

// Sort countries by name, using the sort-order
// of a given locale.
\uksort(
    $countries,
    [\Collator::create($language), 'compare']
);
```

Somewhat readable code

```
private function orderByUids($uids, $contactPersons)
{
    $uidIndices = \array_flip($uids);
    $cmp = function($a, $b) use ($uidIndices) {
        return ($uidIndices[$a->getUid()] -
                $uidIndices[$b->getUid()])
    };
    \usort($contactPersons, $cmp);
    return $contactPersons;
}
```

Readability can be argued.

Mutual dependency: problem

```
class HotelController {
    public function showMapAction() {
        $hotels = $this->hotelService->generateJson($language);

    }
}
class HotelService {
    public function generateJsonData($language) {
        foreach ($this->allHotels as $hotel) {

            $url = /* ??? */

         }
    }
}
```

We want a JSON representation of all hotels. The JSON should include the hotels' URLs, but only the controller has all the information to create an URL for the hotels. The controller should not be botherd with the inner structure of the JSON.

Mutual Dependency: resolution

```
class HotelController {
    public function showMapAction() {
        $hotels = $this->hotelService->generateJson(
            $language,
            $this->createUriGenerator());

    }
    private function createUriGenerator() {
        return function ($hotel) {
            return =
                $this->controllerContext()->getUriBuilder()
                     ->reset()
                     ->setTargetPageUid(5)
                     ->uriFor(…, ["id" => $hotel->getUid()], …);
        }
    }
}
```

We pass a closure to the JSON generating function. The closure, created in the controller, knows how to generate a URL for a given hotel.

Drawbacks & Pitfalls

Clunky and unfamiliar

```
foreach ($names as &$name) {
    $name = strtolower($name);
}
```

vs

```
array_walk($names, function (&$name, $index) {
    $name = strtolower($name);
});
```

Inconsistent

```
array_map($callable, $array);
```

vs

```
array_walk($array, $callable);
```

Type-obscuring syntax

Describing a function by name can make code difficult to understand, especially with higher-order functions:

```
frob('Vladimir', 'Iosifovich', 'Levensthein');
```

`Callable` is an unspecific type

```
// from Silex\ControllerCollection
function match(
    $pattern,
    $to = null) {…}
```

vs

```
function match(
    string $pattern,
    RequestHandler $handler) {…}
```

Exploring a codebase with an IDE is much simpler if argument types are clear and can be inspected. A lot of typing information is lost when using callables. There has been a PHP RFC to change this, but it was defeated with 18 votes in favor and 19 votes against.

The above example is from the silex framework.

Not everything that can be called is a `callable`

Some PHP ”functions” are actually language constructs.

```
$arr = ["", "0", "1"];

// fails
array_map('empty', $arr);

// OK
array_map(function ($x) { return empty($x); }, $arr);
// → [true, true, false]
```

Functions in other languages

JavaScript

Higher-order functions are very common:

- Event handlers

  ```
  document.addEventListener('click', closeModalWindow);
  ```

- Array manipulation

  ```
   var doubled = [1, 2, 3, 4, 5].map(function(x) {
     return x * 2;
  })
  // doubled == [2, 4, 6, 8, 10]
  ```

Note that functions are first class objects in JavaScript.

JavaScript cont.

- Callbacks for async operations

  ```
  $.ajax({…}).done(console.log)
             .fail(function(req, text, err) {..})
  ```

- ES6s arrow functions lead to less boilerplate

  ```
  materials.map(material => material.price);
  ```

The latter differs from the old syntax in that `this` is handled differently.

There is an RFC under discussion suggesting arrow function syntax for PHP.

TypeScript

Use of functions similar to JS; functions can be typed:

```
type RequestHandler<R extends Request> =
    (req: R) => Result

match(pattern: string, handler: RequestHandler) {…}
```

Haskell

```haskell
doubled :: Int -> Int
doubled = (* 2)

sumOfDigits :: Int -> Int
sumOfDigits = sum . map (\c -> read (c:"")) . show

matchAny :: RoutePattern -> Handler () -> ApplicationState ()
matchAny p h = do
  …
```

Adapt to your language

> Every language has its own way. Follow its form, do not try to program as if you were using another language.

Summary

General advice

- Make state changes explicit.
- Functional programming can improve code quality.
- The "strategy" pattern can be simplified in presence of first-class functions.

PHP-specific advice

- Consider using `array_` methods instead of `foreach` loops.
- Do so only if it improves code quality.
- Universal sorting functions are worth using.
- Don't overuse callables.

Questions?