nuGLIF

Les améliorations de

PHP >= 7.0

par Pierrick Charron

NUGLIF

PHP 7.0

3 décembre 2015

Scalar type hinting

- Nouveaux types autorisés : `int float bool string`
- Deux mode de typage ***coercive*** (par défaut), ***strict***

  Le mode est défini par fichier grâce au construit `declare();`

```php
<?php
declare(strict_types=0);

// Coercive mode (declare optional)
function sumOfInts(int ...$ints)
{
        return array_sum($ints);
}

var_dump(sumOfInts(2, '3', 4.1));
// int(9)
```

```php
<?php
declare(strict_types=1);

// Strict mode
function sumOfInts(int ...$ints)
{
        return array_sum($ints);
}

var_dump(sumOfInts(2, '3', 4.1));
// Argument 2 passed to sumOfInts()
must be of the type integer, string
given
```

Coercive type hinting behaviour

Type Déclaration	int	float	string	bool	object
int	yes	yes *	yes †	yes	no
float	yes	yes	yes †	yes	no
string	yes	yes	yes	yes	yes‡
bool	yes	yes	yes	yes	no

* Si compris entre PHP_INT_MIN et PHP_INT_MAX

† Chaînes non numériques non acceptées

‡ Uniquement si __toString() est implémenté

Strict type hinting behaviour

Type Déclaration	int	float	string	bool	object
int	yes	no	no	no	no
float	yes *	yes	no	no	no
string	no	no	yes	no	no
bool	no	no	no	yes	no

* Widening primitive conversion

Déclaration du type de retour

- Les mêmes types que pour le typage de paramètres sont autorisés
- Les deux mode de typage ***coercive*** (par défaut), ***strict*** s’appliquent

  Le même mode que pour le typage de paramètres est utilisé

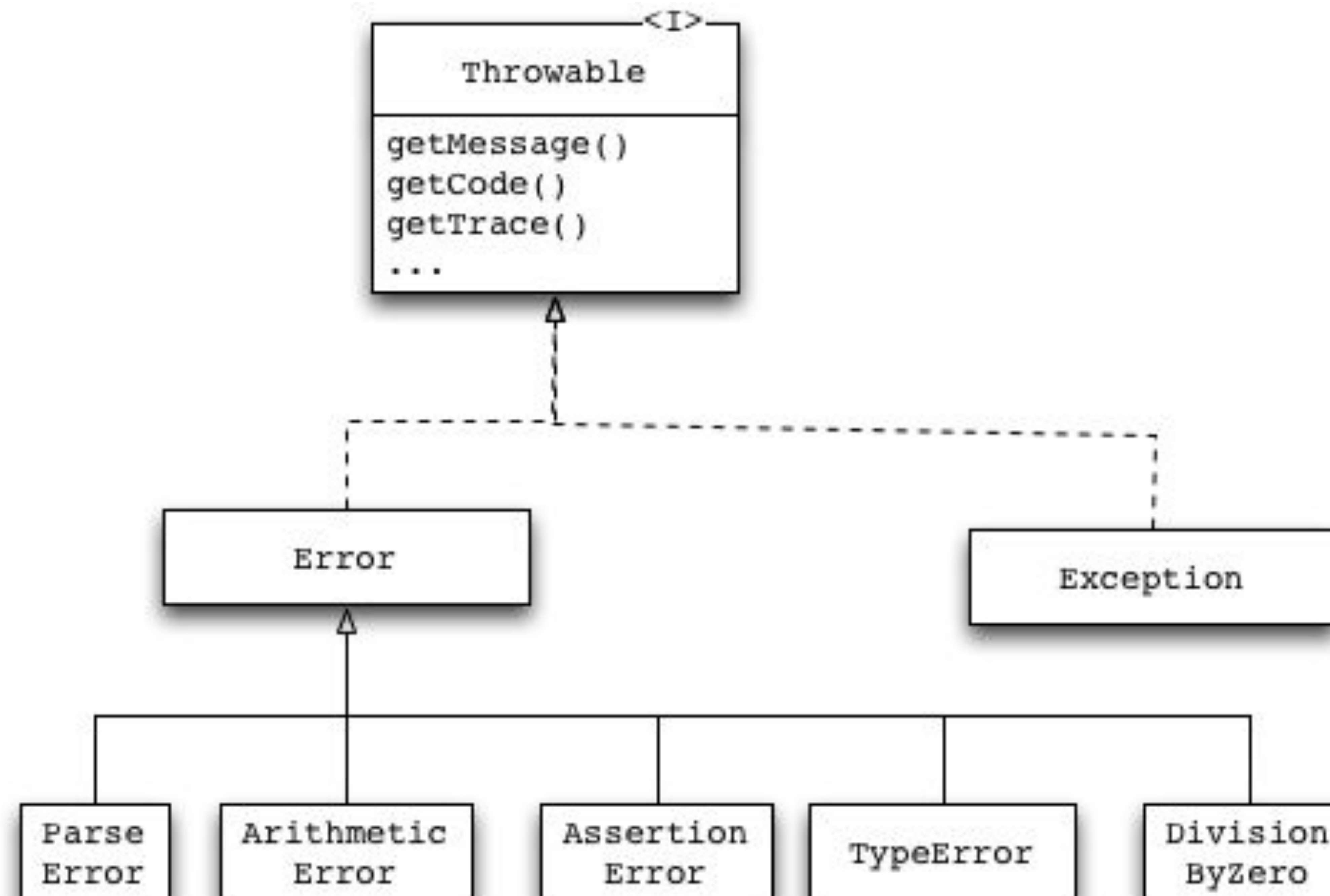
```
function sum($a, $b): float {
    return $a + $b;
}

var_dump(sum(1, 2)); // float(3)
```

Support des classes anonymes

```
interface Logger {
    public function log($format, ...$args);
}

class Util {
    public function setLogger(Logger $logger) { /* … */ }
}

$util->setLogger(

  new class('file.log') implements Logger {
    private $file;
    public function __construct($file) {
      $this->file = $file;
    }
    public function log($format, ...$args) {
      file_put_contents($this->file, sprintf($format, ...$args));
    }
  }

);
```

“*Exceptions*” dans l’engin

- ➢ Remplacement de plusieurs erreurs par des “*Exceptions like*” `Error`
- ➢ Les erreurs PHP sont des objets de type `Error`
- ➢ Pour catcher les `Error` et les `Exception` l’utilisation de `Throwable` est nécessaire



“*Exceptions*” dans l’engin

```
try {
   bar();
} catch(\Error $e) {
   var_dump($e->getMessage());
}
// Call to undefined function
bar()
```

- L’engin permet de catcher les Erreurs

```
bar();
// Fatal error: Uncaught Error:
Call to undefined function bar()
```

- Si l’erreur n’est pas catchée, elle est transformée en Fatal Error

Context sensitive Lexer

- Certains keywords deviennent semi réservés

```
callable class trait extends implements static abstract final public
protected private const enddeclare endfor endforeach endif endwhile and
global goto instanceof insteadof interface namespace new or xor try
use var exit list clone include include_once throw array print echo
require require_once return else elseif default break continue switch
yield function if endswitch finally for foreach declare case do while
as catch die self parent
```

```
class Collection {
    public function forEach(callable $callback) { /* */ }
    public function list() { /* */ }
}
```

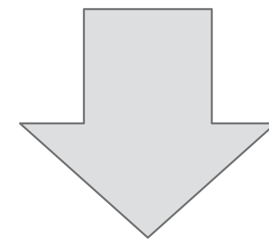
Grouper les déclarations use

Les classes, fonctions et constantes d'un même namespace peuvent êtres groupées dans un même **use** statement

```
// Before PHP 7 code
use some\namespace\ClassA;
use some\namespace\ClassB;
use some\namespace\ClassC as C;

use function some\namespace\fn_a;
use function some\namespace\fn_b;
use function some\namespace\fn_c;

use const some\namespace\ConstA;
use const some\namespace\ConstB;
use const some\namespace\ConstC;

// PHP 7+ code
use some\namespace\{ClassA, ClassB, ClassC as C};
use function some\namespace\{fn_a, fn_b, fn_c};
use const some\namespace\{ConstA, ConstB, ConstC};
```

Null coallescing operator ??

```
$username = isset($_GET['user']) ? $_GET['user'] : 'nobody';
```



```
$username = $_GET['user'] ?? 'nobody';
```

➢ Les coalescences peuvent aussi se chaîner

```
$username = $_GET['user'] ?? $_POST['user'] ?? 'nobody';
```

Zero cost assertions

- `assert` est maintenant un construit du langage avec la signature
  `assert(expression [, message]);`

- Lors de l'execution, si le résultat de l'expression est **`false`**, alors une `AssertionError` sera **`throw`**

- Les assertions peuvent être désactivée via le setting zend.assertions
  - `1 - generate and execute code (development mode)`
  - `0 - generate code and jump around at it at runtime`
  - `-1 - don't generate any code (zero-cost, production mode)`

Zero cost assertions

```php
public function setResponseCode($code) {
    assert(
      $code < 550 && $code > 100,
      "Invalid response code provided: {$code}");
    );

    $this->code = $code;
}
```

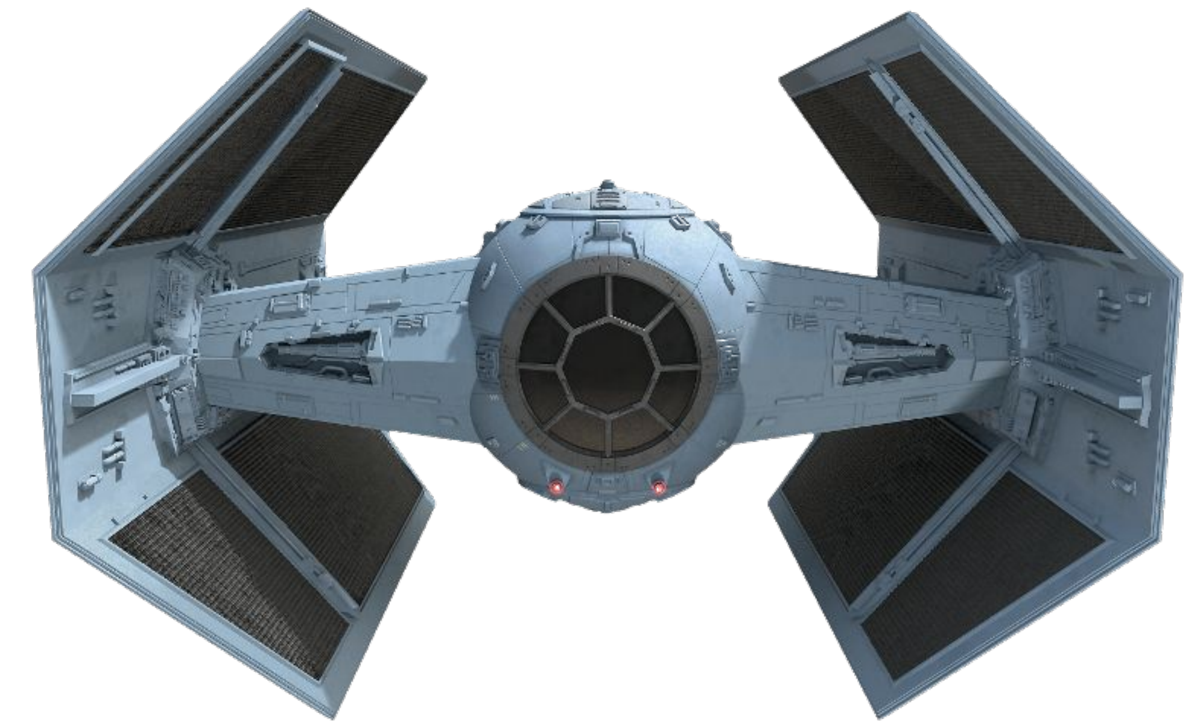
```php
public function getResponseCode() {
    assert($this->code,"The response code is not yet set");

    return $this->code;
}
```

Assertions vs Exceptions

- Exceptions :
  - vérifier les paramètres passés à des fonctions public ou protected.
  - interaction avec un utilisateur ou quand vous vous attendez à ce que le code client se récupère d'une situation exceptionnelle.
  - gérer des problèmes qui pourraient se produire lors du flot d'exécution

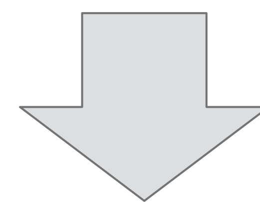
- Assertions
  - vérifier des pre-conditions, post-conditions et invariantes de code privé/interne.
  - donner du feedback à vous ou vos collègues développeurs (documentation).
  - vérifier certaines choses qui ne devraient JAMAIS arriver en production
  - constater des choses que vous (soi-disant) savez être vraies.
  - une assertion ne devrait JAMAIS être nécessaire au code pour fonctionner

Spaceship operator



- Nouvel opérateur de three-way comparaison
- Retourne
  - 0 si les deux opérandes sont égales
  - 1 si l'opérande gauche est supérieure
  - -1 si l’opérande droite est supérieure
- Utile pour écrire des fonctions de comparaison (pour usort, ...)

```php
function order_func($a, $b) {
    return ($a < $b) ? -1 : (($a > $b) ? 1 : 0);
}
```



```php
function order_func($a, $b) {
    return $a <=> $b;
}
```

Caractère d’échapement unicode

➢ Utilisation du \u pour définir un caractère unicode

```
echo "\u{1F602}"; // outputs 😂
```

➢ Permet de mieux distinguer deux valeurs dont l’affichage est visuellement similaire mais avec un encodage différent. Par exemple :

```
echo "mañana";
echo "mañana";
```

est moins explicite que :

```
echo "ma\u{00F1}ana";  // pre-composed character
echo "man\u{0303}ana"; // "n" with combining ~ character (U+0303)
```

Délégation de générateurs

```php
function gen()
{
    yield 1;
    yield 2;
    yield from gen2();
    yield from [ 5 , 6 ];
    yield from new ArrayIterator([ 7 , 8 ]);
}

function gen2()
{
    yield 3;
    yield 4;
}

foreach (gen() as $val)
{
    echo $val, PHP_EOL;
}
// 1 2 3 4 5 6 7 8
```

- Permet de déléguer la logique d’un générateur à des objets `Traversable` ou à des `Array`

Autres

- Suppression des tags php alternatifs
  - `<% %>`
  - `<script type="php"></script>`

- Suppression des extensions
  - mssql
  - mysql
  - ereg

- Zend Engine 3.0
  - AST Based parser
  - Meilleur gestion de la mémoire

NUGLIF

PHP 7.1

1 décembre 2016

Nullable type

➢ Possibilité d'accepter null comme valeur d'un certain type

```php
<?php
declare(strict_types=0);

function nullableParam(?string $str)
{
	// $str will be string or null
}

function nullableReturn(): ?string
{
	// This function will only accept string or null
	// as return type
}
```

Void return type

- Signature des méthodes sans retour

```
<?php
declare(strict_types=0);

function doNothing(): void
{
        // Fatal error if the function return something
}
```

Symmetric array destructuring

```
$data = [
    [1, 'Tom'],
    [2, 'Fred'],
];

// list() style
list($id1, $name1) = $data[0];

// [] style
[$id1, $name1] = $data[0];

// list() style
foreach ($data as list($id, $name)) {
    // logic here with $id and $name
}

// [] style
foreach ($data as [$id, $name]) {
    // logic here with $id and $name
}
```

Class constant visibility

- Permet d’ajouter de la visibilité à une constante de classe

```php
<?php
class ConstDemo
{
    const PUBLIC_CONST_A = 1;
    public const PUBLIC_CONST_B = 2;
    protected const PROTECTED_CONST = 3;
    private const PRIVATE_CONST = 4;
}
```

Iterable pseudo type

- Accepte les array ou les objets qui implémentent Iterable
- Peut être utilisé pour les paramètres et les valeurs de retour

```
<?php
function iterator(iterable $iter)
{
    foreach ($iter as $val) {
        //
    }
}
```

Multi-catch exception handling

- Permet de gérer plusieurs exception de hiérarchie différentes de la même façon sans dupliquer le code

```php
<?php
try {
    // some code
} catch (FirstException | SecondException $e) {
    // handle first and second exceptions
}
```

Support for keys in list()

- Permet la déstructuration d'un array non séquentiel ou avec des clés non numériques

```php
$data = [
    ["id" => 1, "name" => 'Tom'],
    ["id" => 2, "name" => 'Fred'],
];

// list() style
list("id" => $id1, "name" => $name1) = $data[0];

// [] style
["id" => $id1, "name" => $name1] = $data[0];

// list() style
foreach ($data as list("id" => $id, "name" => $name)) {
    // logic here with $id and $name
}

// [] style
foreach ($data as ["id" => $id, "name" => $name]) {
    // logic here with $id and $name
}
```

Autres features

- Negative string offset

```
var_dump("abcdef"[-2]);
// string(1) "e"
```

- HTTP/2 server push support in ext/curl
- Asynchronous signal handling
- Support for AEAD in ext/openssl

NUGLIF

Questions ?

Merci :-)