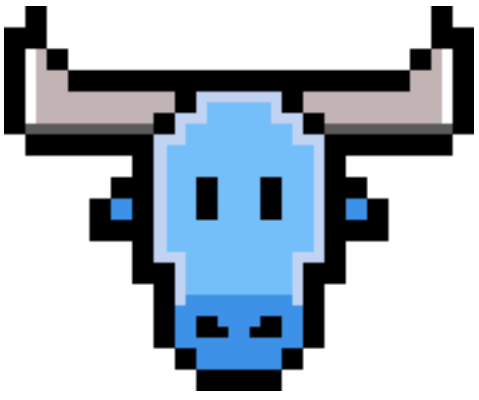



‹ Introduction (chapitre1.html) Lancement des tests (chapitre3.html) ›

# 2. Écrire ses tests

## 2.1. Assertions

### 2.1.1. variable

C'est l'assertion de base de toutes les variables. Elle contient les tests nécessaires à n'importe quel type de variable.

#### 2.1.1.1. isCallable

`isCallable` vérifie que la variable peut être appelée comme fonction.

```
 1.  $f = function() {
 2.      // code
 3.  };
 4.
 5.  $this
 6.      ->variable($f)
 7.          ->isCallable()  // passe
 8.
 9.      ->variable('\Vendor\Project\foobar')
10.          ->isCallable()
11.
12.      ->variable(array (http://www.php.net/array)('\Vendor\Project\Foo'
13.          ->isCallable()
14.
15.      ->variable('\Vendor\Project\Foo::bar')
16.          ->isCallable()
17.  ;
```

#### 2.1.1.2. isEqualTo

`isEqualTo` vérifie que la variable est égale à une certaine donnée.

```
1.  $a = 'a';
2.
3.  $this
4.      ->variable($a)
5.          ->isEqualTo('a')    // passe
6.  ;
```

> `isEqualTo` *ne teste pas le type de la variable. Si vous souhaitez vérifier également son type, utilisez* `isIdenticalTo` *.*

#### 2.1.1.3. isIdenticalTo

`isIdenticalTo` vérifie que la variable a la même valeur et le même type qu'une certaine donnée. Dans le cas d'objets, `isIdenticalTo` vérifie que les données pointent sur la même instance.

```
 1.  $a = '1';
 2.
 3.  $this
 4.      ->variable($a)
 5.          ->isIdenticalTo(1)          // échoue
 6.  ;
 7.
 8.  $stdClass1 = new \StdClass();
 9.  $stdClass2 = new \StdClass();
10.  $stdClass3 = $stdClass1;
11.
12.  $this
13.      ->variable($stdClass1)
14.          ->isIdenticalTo(stdClass3)  // passe
15.          ->isIdenticalTo(stdClass2)  // échoue
16.  ;
```

> `isIdenticalTo` *teste le type de la variable. Si vous ne souhaitez pas vérifier son type, utilisez* `isEqualTo` *.*

#### 2.1.1.4. isNotCallable

`isNotCallable` vérifie que la variable ne peut pas être appelée comme fonction.

```
 1.  $f = function() {
 2.      // code
 3.  };
 4.  $int    = 1;
 5.  $string = 'nonExistingMethod';
 6.
 7.  $this
 8.      ->variable($f)
 9.          ->isNotCallable()   // échoue
10.
11.      ->variable($int)
12.          ->isNotCallable()   // passe
13.
14.      ->variable($string)
15.          ->isNotCallable()   // passe
16.
17.      ->variable(new StdClass)
18.          ->isNotCallable()   // passe
19.  ;
```

#### 2.1.1.5. isNotEqualTo

`isNotEqualTo` vérifie que la variable n'a pas la même valeur qu'une certaine donnée.

```
 1.  $a       = 'a';
 2.  $aString = '1';
 3.
 4.  $this
 5.      ->variable($a)
 6.          ->isNotEqualTo('b')     // passe
 7.          ->isNotEqualTo('a')     // échoue
 8.
 9.      ->variable($aString)
10.          ->isNotEqualTo($1)      // échoue
11.  ;
```

> `isNotEqualTo` *ne teste pas le type de la variable. Si vous souhaitez vérifier également son type, utilisez* `isNotIdenticalTo` *.*

#### 2.1.1.6. isNotIdenticalTo

`isNotIdenticalTo` vérifie que la variable n'a ni le même type ni la même valeur qu'une certaine donnée.

Dans le cas d'objets, `isNotIdenticalTo` vérifie que les données ne pointent pas sur la même instance.

```
 1.  $a = '1';
 2.
 3.  $this
 4.      ->variable($a)
 5.          ->isNotIdenticalTo(1)           // passe
 6.  ;
 7.
 8.  $stdClass1 = new \StdClass();
 9.  $stdClass2 = new \StdClass();
10.  $stdClass3 = $stdClass1;
11.
12.  $this
13.      ->variable($stdClass1)
14.          ->isNotIdenticalTo(stdClass2)   // passe
15.          ->isNotIdenticalTo(stdClass3)   // échoue
16.  ;
```

> `isNotIdenticalTo` *teste le type de la variable. Si vous ne souhaitez pas vérifier son type, utilisez* `isNotEqualTo` *.*

#### 2.1.1.7. isNull

`isNull` vérifie que la variable est nulle.

```
 1.  $emptyString = '';
 2.  $null        = null;
 3.
 4.  $this
 5.      ->variable($emptyString)
 6.          ->isNull()              // échoue
 7.                                  // (c'est vide mais pas null)
 8.
 9.      ->variable($null)
10.          ->isNull()              // passe
11.  ;
```

#### 2.1.1.8. isNotNull

`isNotNull` vérifie que la variable n'est pas nulle.

```
 1.  $emptyString = '';
 2.  $null        = null;
 3.
 4.  $this
 5.      ->variable($emptyString)
 6.          ->isNotNull()           // passe (c'est vide mais pas null)
 7.
 8.      ->variable($null)
 9.          ->isNotNull()           // échoue
10.  ;
```

### 2.1.2. boolean

C'est l'assertion dédiée aux booléens.

Si vous essayez de tester une variable qui n'est pas un booléen avec cette assertion, cela échouera.

> `null` *n'est pas un booléen. Reportez-vous au manuel de PHP pour savoir ce que* `is_bool (http://php.net/is_bool)` *considère ou non comme un booléen.*

#### 2.1.2.1. isEqualTo

> `isEqualTo` *est une méthode héritée de l'asserter* `variable`*.*
> *Pour plus d'informations, reportez-vous à la documentation de* `variable::isEqualTo`

#### 2.1.2.2. isFalse

`isFalse` vérifie que le booléen est strictement égal à `false`.

```
 1.  $true  = true;
 2.  $false = false;
 3.
 4.  $this
 5.      ->boolean($true)
 6.          ->isFalse()     // échoue
 7.
 8.      ->boolean($false)
 9.          ->isFalse()     // passe
10.  ;
```

### 2.1.2.3. isIdenticalTo

`isIdenticalTo` *est une méthode héritée de l'asserter* `variable` *.*
*Pour plus d'informations, reportez-vous à la documentation de*
`variable::isIdenticalTo`

### 2.1.2.4. isNotEqualTo

`isNotEqualTo` *est une méthode héritée de l'asserter* `variable` *.*
*Pour plus d'informations, reportez-vous à la documentation de*
`variable::isNotEqualTo`

### 2.1.2.5. isNotIdenticalTo

`isNotIdenticalTo` *est une méthode héritée de l'asserter* `variable` *.*
*Pour plus d'informations, reportez-vous à la documentation de*
`variable::isNotIdenticalTo`

### 2.1.2.6. isTrue

`isTrue` vérifie que le booléen est strictement égal à `true` .

```
 1.  $true  = true;
 2.  $false = false;
 3.
 4.  $this
 5.      ->boolean($true)
 6.          ->isTrue()      // passe
 7.
 8.      ->boolean($false)
 9.          ->isTrue()      // échoue
10.  ;
```

## 2.1.3. integer

C'est l'assertion dédiée aux entiers.

Si vous essayez de tester une variable qui n'est pas un entier avec cette assertion, cela échouera.

`null` *n'est pas un entier. Reportez-vous au manuel de PHP pour savoir ce que* `is_int (http://php.net/is_int)` *considère ou non comme un entier.*

#### 2.1.3.1. isEqualTo

`isEqualTo` *est une méthode héritée de l'asserter* `variable`*. Pour plus d'informations, reportez-vous à la documentation de* `variable::isEqualTo`

#### 2.1.3.2. isGreaterThan

`isGreaterThan` vérifie que l'entier est strictement supérieur à une certaine donnée.

```
1.  $zero = 0;
2.
3.  $this
4.      ->integer($zero)
5.          ->isGreaterThan(-1)     // passe
6.          ->isGreaterThan('-1')   // échoue car "-1"
7.                                  // n'est pas un entier
8.          ->isGreaterThan(0)      // échoue
9.  ;
```

#### 2.1.3.3. isGreaterThanOrEqualTo

`isGreaterThanOrEqualTo` vérifie que l'entier est supérieur ou égal à une certaine donnée.

```
1.  $zero = 0;
2.
3.  $this
4.      ->integer($zero)
5.          ->isGreaterThanOrEqualTo(-1)    // passe
6.          ->isGreaterThanOrEqualTo(0)     // passe
7.          ->isGreaterThanOrEqualTo('-1')  // échoue car "-1"
8.                                          // n'est pas un entier
9.  ;
```

#### 2.1.3.4. isIdenticalTo

`isIdenticalTo` *est une méthode héritée de l'asserter* `variable`.

*Pour plus d'informations, reportez-vous à la documentation de* `variable::isIdenticalTo`

### 2.1.3.5. isLessThan

`isLessThan` vérifie que l'entier est strictement inférieur à une certaine donnée.

```
1.  $zero = 0;
2.
3.  $this
4.      ->integer($zero)
5.          ->isLessThan(10)    // passe
6.          ->isLessThan('10')  // échoue car "10" n'est pas un entier
7.          ->isLessThan(0)     // échoue
8.  ;
```

### 2.1.3.6. isLessThanOrEqualTo

`isLessThanOrEqualTo` vérifie que l'entier est inférieur ou égal à une certaine donnée.

```
1.  $zero = 0;
2.
3.  $this
4.      ->integer($zero)
5.          ->isLessThanOrEqualTo(10)       // passe
6.          ->isLessThanOrEqualTo(0)        // passe
7.          ->isLessThanOrEqualTo('10')     // échoue car "10"
8.                                          // n'est pas un entier
9.  ;
```

### 2.1.3.7. isNotEqualTo

`isNotEqualTo` *est une méthode héritée de l'asserter* `variable`.

*Pour plus d'informations, reportez-vous à la documentation de* `variable::isNotEqualTo`

### 2.1.3.8. isNotIdenticalTo

`isNotIdenticalTo` *est une méthode héritée de l'asserter* `variable`*.*
*Pour plus d'informations, reportez-vous à la documentation de* `variable::isNotIdenticalTo`

#### 2.1.3.9. isZero

`isZero` vérifie que l'entier est égal à 0.

```
 1.  $zero    = 0;
 2.  $notZero = -1;
 3.
 4.  $this
 5.      ->integer($zero)
 6.          ->isZero()          // passe
 7.
 8.      ->integer($notZero)
 9.          ->isZero()          // échoue
10.  ;
```

`isZero` *est équivalent à* `isEqualTo(0)`*.*

### 2.1.4. float

C'est l'assertion dédiée aux nombres décimaux.

Si vous essayez de tester une variable qui n'est pas un nombre décimal avec cette assertion, cela échouera.

`null` *n'est pas un nombre décimal. Reportez-vous au manuel de PHP pour savoir ce que* `is_float (http://php.net/is_float)` *considère ou non comme un nombre décimal.*

#### 2.1.4.1. isEqualTo

`isEqualTo` *est une méthode héritée de l'asserter* `variable`*.*
*Pour plus d'informations, reportez-vous à la documentation de* `variable::isEqualTo`

#### 2.1.4.2. isGreaterThan

`isGreaterThan` *est une méthode héritée de l'asserter* `integer` *.*

*Pour plus d'informations, reportez-vous à la documentation de* `integer::isGreaterThan`

### 2.1.4.3. isGreaterThanOrEqualTo

`isGreaterThanOrEqualTo` *est une méthode héritée de l'asserter* `integer` *.*

*Pour plus d'informations, reportez-vous à la documentation de* `integer::isGreaterThanOrEqualTo`

### 2.1.4.4. isIdenticalTo

`isIdenticalTo` *est une méthode héritée de l'asserter* `variable` *.*

*Pour plus d'informations, reportez-vous à la documentation de* `variable::isIdenticalTo`

### 2.1.4.5. isLessThan

`isLessThan` *est une méthode héritée de l'asserter* `integer` *.*

*Pour plus d'informations, reportez-vous à la documentation de* `integer::isLessThan`

### 2.1.4.6. isLessThanOrEqualTo

`isLessThanOrEqualTo` *est une méthode héritée de l'asserter* `integer` *.*

*Pour plus d'informations, reportez-vous à la documentation de* `integer::isLessThanOrEqualoo`

### 2.1.4.7. isNearlyEqualTo

`isNearlyEqualTo` vérifie que le nombre décimal est approximativement égal à la valeur qu'elle reçoit en argument.

En effet, en informatique, les nombres décimaux sont gérées d'une façon qui ne permet pas d'effectuer des comparaisons précises sans recourir à des outils spécialisés. Essayez par exemple d'exécuter la commande suivante:

```
$ php -r 'var_dump(1 - 0.97 === 0.03);'
bool(false)
```

Le résultat devrait pourtant être `true` .

> *Pour avoir plus d'informations sur ce phénomène, reportez-vous au manuel de PHP (http://php.net/types.float).*

Cette méthode cherche donc à minorer ce problème.

```
1.  $float = 1 - 0.97;
2.
3.  $this
4.      ->float($float)
5.          ->isNearlyEqualTo(0.03) // passe
6.          ->isEqualTo(0.03)       // échoue
7.  ;
```

> *Pour avoir plus d'informations sur l'algorithme utilisé, consultez le floating point guide (http://www.floating-point-gui.de/errors/comparison/).*

#### 2.1.4.8. isNotEqualTo

> `isNotEqualTo` *est une méthode héritée de l'asserter* `variable` *.*
> *Pour plus d'informations, reportez-vous à la documentation de* `variable::isNotEqualTo`

#### 2.1.4.9. isNotIdenticalTo

> `isNotIdenticalTo` *est une méthode héritée de l'asserter* `variable` *.*
> *Pour plus d'informations, reportez-vous à la documentation de* `variable::isNotIdenticalTo`

#### 2.1.4.10. isZero

`isZero` *est une méthode héritée de l'asserter* `integer` *.*

*Pour plus d'informations, reportez-vous à la documentation de* `integer::isZero`

### 2.1.5. sizeOf

C'est l'assertion dédiée aux tests sur la taille des tableaux et des objets implémentant l'interface `Countable` .

```
 1.  $array           = array (http://www.php.net/array)(1, 2, 3);
 2.  $countableObject = new GlobIterator('*');
 3.
 4.  $this
 5.      ->sizeOf (http://www.php.net/sizeof)($array)
 6.          ->isEqualTo(3)
 7.
 8.      ->sizeOf (http://www.php.net/sizeof)($countableObject)
 9.          ->isGreaterThan(0)
10.  ;
```

#### 2.1.5.1. isEqualTo

`isEqualTo` *est une méthode héritée de l'asserter* `variable` *.*

*Pour plus d'informations, reportez-vous à la documentation de* `variable::isEqualTo`

#### 2.1.5.2. isGreaterThan

`isGreaterThan` *est une méthode héritée de l'asserter* `integer` *.*

*Pour plus d'informations, reportez-vous à la documentation de* `integer::isGreaterThan`

#### 2.1.5.3. isGreaterThanOrEqualTo

`isGreaterThanOrEqualTo` *est une méthode héritée de l'asserter* `integer` *.*

*Pour plus d'informations, reportez-vous à la documentation de* `integer::isGreaterThanOrEqualTo`

#### 2.1.5.4. isIdenticalTo

`isIdenticalTo` *est une méthode héritée de l'asserter* `variable` *.*

*Pour plus d'informations, reportez-vous à la documentation de* `variable::isIdenticalTo`

#### 2.1.5.5. isLessThan

`isLessThan` *est une méthode héritée de l'asserter* `integer` *.*

*Pour plus d'informations, reportez-vous à la documentation de* `integer::isLessThan`

#### 2.1.5.6. isLessThanOrEqualTo

`isLessThanOrEqualTo` *est une méthode héritée de l'asserter* `integer` *.*

*Pour plus d'informations, reportez-vous à la documentation de* `integer::isLessThanOrEqualoo`

#### 2.1.5.7. isNotEqualTo

`isNotEqualTo` *est une méthode héritée de l'asserter* `variable` *.*

*Pour plus d'informations, reportez-vous à la documentation de* `variable::isNotEqualTo`

#### 2.1.5.8. isNotIdenticalTo

`isNotIdenticalTo` *est une méthode héritée de l'asserter* `variable` *.*

*Pour plus d'informations, reportez-vous à la documentation de* `variable::isNotIdenticalTo`

#### 2.1.5.9. isZero

`isZero` *est une méthode héritée de l'asserter* `integer` *.*

*Pour plus d'informations, reportez-vous à la documentation de* `integer::isZero`

### 2.1.6. object

C'est l'assertion dédiée aux objets.

Si vous essayez de tester une variable qui n'est pas un objet avec cette assertion, cela échouera.

> `null` *n'est pas un objet. Reportez-vous au manuel de PHP pour savoir ce que* `is_object (http://php.net/is_object)` *considère ou non comme un objet.*

#### 2.1.6.1. hasSize

`hasSize` vérifie la taille d'un objet qui implémente l'interface `Countable`.

```
1.  $countableObject = new GlobIterator('*');
2.
3.  $this
4.      ->object($countableObject)
5.          ->hasSize(3)
6.  ;
```

#### 2.1.6.2. isCallable

```
 1.  class foo
 2.  {
 3.      public function __invoke()
 4.      {
 5.          // code
 6.      }
 7.  }
 8.
 9.  $this
10.      ->object(new foo)
11.          ->isCallable()  // passe
12.
13.      ->object(new StdClass)
14.          ->isCallable()  // échoue
15.  ;
```

> *Pour être identifiés comme* `callable`, *vos objets devront être instanciés à partir de classes qui implémentent la méthode magique* `__invoke` *(http://www.php.net/manual/fr/language.oop5.magic.php#object.invoke).*

`isCallable` *est une méthode héritée de l'asserter* `variable`*.*

*Pour plus d'informations, reportez-vous à la documentation de* `variable::isCallable`

### 2.1.6.3. isCloneOf

`isCloneOf` vérifie qu'un objet est le clone d'un objet donné, c'est-à-dire que les objets sont égaux, mais ne pointent pas vers la même instance.

```
 1.  $object1 = new \StdClass;
 2.  $object2 = new \StdClass;
 3.  $object3 = clone($object1);
 4.  $object4 = new \StdClass;
 5.  $object4->foo = 'bar';
 6.
 7.  $this
 8.      ->object($object1)
 9.          ->isCloneOf($object2)   // passe
10.          ->isCloneOf($object3)   // passe
11.          ->isCloneOf($object4)   // échoue
12.  ;
```

*Pour avoir plus de précision sur la comparaison d'objet, reportez-vous au manuel de PHP (http://php.net/language.oop5.object-comparison).*

### 2.1.6.4. isEmpty

`isEmpty` vérifie que la taille d'un objet implémentant l'interface `Countable` est égale à 0.

```
1.  $countableObject = new GlobIterator('atoum.php');
2.
3.  $this
4.      ->object($countableObject)
5.          ->isEmpty()
6.  ;
```

`isEmpty` *est équivalent à* `hasSize(0)`*.*

#### 2.1.6.5. isEqualTo

`isEqualTo` vérifie qu'un objet est égal à un autre.
Deux objets sont considérés égaux lorsqu'ils ont les mêmes attributs et valeurs, et qu'ils sont des instances de la même classe.

> *Pour avoir plus de précision sur la comparaison d'objet, reportez-vous au manuel de PHP (http://php.net/language.oop5.object-comparison).*

> `isEqualTo` *est une méthode héritée de l'asserter* `variable`*.*
> *Pour plus d'informations, reportez-vous à la documentation de* `variable::isEqualTo`

#### 2.1.6.6. isIdenticalTo

`isIdenticalTo` vérifie que deux objets sont identiques.
Deux objets sont considérés identiques lorsqu'ils font référence à la même instance de la même classe.

> *Pour avoir plus de précision sur la comparaison d'objet, reportez-vous au manuel de PHP (http://php.net/language.oop5.object-comparison).*

> `isIdenticalTo` *est une méthode héritée de l'asserter* `variable`*.*
> *Pour plus d'informations, reportez-vous à la documentation de* `variable::isIdenticalTo`

#### 2.1.6.7. isInstanceOf

`isInstanceOf` vérifie qu'un objet est :

- une instance de la classe donnée,
- une sous-classe de la classe donnée (abstraite ou non),
- une instance d'une classe qui implémente l'interface donnée.

```php
$object = new \StdClass();

$this
    ->object($object)
        ->isInstanceOf('\StdClass')     // passe
        ->isInstanceOf('\Iterator')     // échoue
;


interface FooInterface
{
    public function foo();
}

class FooClass implements FooInterface
{
    public function foo()
    {
        echo "foo";
    }
}

class BarClass extends FooClass
{
}

$foo = new FooClass;
$bar = new BarClass;

$this
    ->object($foo)
        ->isInstanceOf('\FooClass')     // passe
        ->isInstanceOf('\FooInterface') // passe
        ->isInstanceOf('\BarClass')     // échoue
        ->isInstanceOf('\StdClass')     // échoue

    ->object($bar)
        ->isInstanceOf('\FooClass')     // passe
        ->isInstanceOf('\FooInterface') // passe
        ->isInstanceOf('\BarClass')     // passe
        ->isInstanceOf('\StdClass')     // échoue
;
```

*Les noms des classes et des interfaces doivent être absolus, car les éventuelles importations d'espace de nommage ne sont pas prises en compte.*

### 2.1.6.8. isNotCallable

```
 1.  class foo
 2.  {
 3.      public function __invoke()
 4.      {
 5.          // code
 6.      }
 7.  }
 8.
 9.  $this
10.      ->variable(new foo)
11.          ->isNotCallable()   // échoue
12.
13.      ->variable(new StdClass)
14.          ->isNotCallable()   // passe
15.  ;
```

`isNotCallable` *est une méthode héritée de l'asserter* `variable` *.*
*Pour plus d'informations, reportez-vous à la documentation de* `variable::isNotCallable`

### 2.1.6.9. isNotEqualTo

`isEqualTo` vérifie qu'un objet n'est pas égal à un autre.
Deux objets sont considérés égaux lorsqu'ils ont les mêmes attributs et valeurs, et qu'ils sont des instances de la même classe.

*Pour avoir plus de précision sur la comparaison d'objet, reportez-vous au manuel de PHP (http://php.net/language.oop5.object-comparison).*

`isNotEqualTo` *est une méthode héritée de l'asserter* `variable` *.*
*Pour plus d'informations, reportez-vous à la documentation de* `variable::isNotEqualTo`

### 2.1.6.10. isNotIdenticalTo

`isIdenticalTo` vérifie que deux objets ne sont pas identiques.
Deux objets sont considérés identiques lorsqu'ils font référence à la même instance de la même classe.

*Pour avoir plus de précision sur la comparaison d'objet, reportez-vous au manuel de PHP (http://php.net/language.oop5.object-comparison).*

`isNotIdenticalTo` *est une méthode héritée de l'asserter* `variable` *.*
*Pour plus d'informations, reportez-vous à la documentation de*
`variable::isNotIdenticalTo`

### 2.1.7. dateInterval

C'est l'assertion dédiée à l'objet `DateInterval (http://php.net/dateinterval)` .

Si vous essayez de tester une variable qui n'est pas un objet `DateInterval` (ou une classe qui l'étend) avec cette assertion, cela échouera.

#### 2.1.7.1. isCloneOf

`isCloneOf` *est une méthode héritée de l'asserter* `object` *.*
*Pour plus d'informations, reportez-vous à la documentation de*
`object::isCloneOf`

#### 2.1.7.2. isEqualTo

`isEqualTo` vérifie que la durée de l'objet `DateInterval` est égale à la durée d'un autre objet `DateInterval` .

```
 1.  $di = new DateInterval('P1D');
 2.
 3.  $this
 4.      ->dateInterval($di)
 5.          ->isEqualTo(                // passe
 6.              new DateInterval('P1D')
 7.          )
 8.          ->isEqualTo(                // échoue
 9.              new DateInterval('P2D')
10.          )
11.  ;
```

#### 2.1.7.3. isGreaterThan

`isGreaterThan` vérifie que la durée de l'objet `DateInterval` est supérieure à la durée d'un autre objet `DateInterval` .

```
$di = new DateInterval('P2D');

$this
    ->dateInterval($di)
        ->isGreaterThan(            // passe
            new DateInterval('P1D')
        )
        ->isGreaterThan(            // échoue
            new DateInterval('P2D')
        )
;
```

#### 2.1.7.4. isGreaterThanOrEqualTo

`isGreaterThanOrEqualTo` vérifie que la durée de l'objet `DateInterval` est supérieure ou égale à la durée d'un autre objet `DateInterval` .

```
$di = new DateInterval('P2D');

$this
    ->dateInterval($di)
        ->isGreaterThanOrEqualTo(   // passe
            new DateInterval('P1D')
        )
        ->isGreaterThanOrEqualTo(   // passe
            new DateInterval('P2D')
        )
        ->isGreaterThanOrEqualTo(   // échoue
            new DateInterval('P3D')
        )
;
```

#### 2.1.7.5. isIdenticalTo

`isIdenticalTo` *est une méthode héritée de l'asserter* `object` *.*

*Pour plus d'informations, reportez-vous à la documentation de* `object::isIdenticalTo`

#### 2.1.7.6. isInstanceOf

`isInstanceOf` *est une méthode héritée de l'asserter* `object` *.*

*Pour plus d'informations, reportez-vous à la documentation de* `object::isInstanceOf`

### 2.1.7.7. isLessThan

`isLessThan` vérifie que la durée de l'objet `DateInterval` est inférieure à la durée d'un autre objet `DateInterval` .

```
 1.  $di = new DateInterval('P1D');
 2.
 3.  $this
 4.      ->dateInterval($di)
 5.          ->isLessThan(                // passe
 6.              new DateInterval('P2D')
 7.          )
 8.          ->isLessThan(                // échoue
 9.              new DateInterval('P1D')
10.          )
11.  ;
```

### 2.1.7.8. isLessThanOrEqualTo

`isLessThanOrEqualTo` vérifie que la durée de l'objet `DateInterval` est inférieure ou égale à la durée d'un autre objet `DateInterval` .

```
 1.  $di = new DateInterval('P2D');
 2.
 3.  $this
 4.      ->dateInterval($di)
 5.          ->isLessThanOrEqualTo(      // passe
 6.              new DateInterval('P3D')
 7.          )
 8.          ->isLessThanOrEqualTo(      // passe
 9.              new DateInterval('P2D')
10.          )
11.          ->isLessThanOrEqualTo(      // échoue
12.              new DateInterval('P1D')
13.          )
14.  ;
```

### 2.1.7.9. isNotEqualTo

`isNotEqualTo` *est une méthode héritée de l'asserter* `object` *.*

*Pour plus d'informations, reportez-vous à la documentation de* `object::isNotEqualTo`

### 2.1.7.10. isNotIdenticalTo

`isNotIdenticalTo` *est une méthode héritée de l'asserter* `object`*.*

*Pour plus d'informations, reportez-vous à la documentation de* `object::isNotIdenticalTo`

#### 2.1.7.11. isZero

`isZero` vérifie que la durée de l'objet `DateInterval` est égale à 0.

```
1.  $di1 = new DateInterval('P0D');
2.  $di2 = new DateInterval('P1D');
3.
4.  $this
5.      ->dateInterval($di1)
6.          ->isZero()      // passe
7.      ->dateInterval($di2)
8.          ->isZero()      // échoue
9.  ;
```

### 2.1.8. dateTime

C'est l'assertion dédiée à l'objet `DateTime (http://php.net/datetime)`.

Si vous essayez de tester une variable qui n'est pas un objet `DateTime` (ou une classe qui l'étend) avec cette assertion, cela échouera.

#### 2.1.8.1. hasDate

`hasDate` vérifie la partie date de l'objet `DateTime`.

```
1.  $dt = new DateTime('1981-02-13');
2.
3.  $this
4.      ->dateTime($dt)
5.          ->hasDate('1981', '02', '13')   // passe
6.          ->hasDate('1981', '2',  '13')   // passe
7.          ->hasDate(1981,   2,    13)     // passe
8.  ;
```

#### 2.1.8.2. hasDateAndTime

`hasDateAndTime` vérifie la date et l'horaire de l'objet `DateTime`

```php
$dt = new DateTime('1981-02-13 01:02:03');

$this
    ->dateTime($dt)
        // passe
        ->hasDateAndTime('1981', '02', '13', '01', '02', '03')
        // passe
        ->hasDateAndTime('1981', '2',  '13', '1',  '2',  '3')
        // passe
        ->hasDateAndTime(1981,   2,    13,   1,    2,    3)
;
```

#### 2.1.8.3. hasDay

`hasDay` vérifie le jour de l'objet `DateTime`.

```php
$dt = new DateTime('1981-02-13');

$this
    ->dateTime($dt)
        ->hasDay(13)        // passe
;
```

#### 2.1.8.4. hasHours

`hasHours` vérifie les heures de l'objet `DateTime`.

```php
$dt = new DateTime('01:02:03');

$this
    ->dateTime($dt)
        ->hasHours('01')    // passe
        ->hasHours('1')     // passe
        ->hasHours(1)       // passe
;
```

#### 2.1.8.5. hasMinutes

`hasMinutes` vérifie les minutes de l'objet `DateTime`.

```
1.  $dt = new DateTime('01:02:03');
2.
3.  $this
4.      ->dateTime($dt)
5.          ->hasMinutes('02')  // passe
6.          ->hasMinutes('2')   // passe
7.          ->hasMinutes(2)     // passe
8.  ;
```

#### 2.1.8.6. hasMonth

`hasMonth` vérifie le mois de l'objet `DateTime` .

```
1.  $dt = new DateTime('1981-02-13');
2.
3.  $this
4.      ->dateTime($dt)
5.          ->hasMonth(2)       // passe
6.  ;
```

#### 2.1.8.7. hasSeconds

`hasSeconds` vérifie les secondes de l'objet `DateTime` .

```
1.  $dt = new DateTime('01:02:03');
2.
3.  $this
4.      ->dateTime($dt)
5.          ->hasSeconds('03')    // passe
6.          ->hasSeconds('3')     // passe
7.          ->hasSeconds(3)       // passe
8.  ;
```

#### 2.1.8.8. hasTime

`hasTime` vérifie la partie horaire de l'objet `DateTime`

```
1.  $dt = new DateTime('01:02:03');
2.
3.  $this
4.      ->dateTime($dt)
5.          ->hasTime('01', '02', '03')     // passe
6.          ->hasTime('1',  '2',  '3')      // passe
7.          ->hasTime(1,    2,    3)        // passe
8.  ;
```

#### 2.1.8.9. hasTimezone

`hasTimezone` vérifie le fuseau horaire de l'objet `DateTime` .

```
1.  $dt = new DateTime();
2.
3.  $this
4.      ->dateTime($dt)
5.          ->hasTimezone('Europe/Paris')
6.  ;
```

#### 2.1.8.10. hasYear

`hasYear` vérifie l'année de l'objet `DateTime` .

```
1.  $dt = new DateTime('1981-02-13');
2.
3.  $this
4.      ->dateTime($dt)
5.          ->hasYear(1981)     // passe
6.  ;
```

#### 2.1.8.11. isCloneOf

`isCloneOf` *est une méthode héritée de l'asserter* `object` *.*

*Pour plus d'informations, reportez-vous à la documentation de* `object::isCloneOf`

#### 2.1.8.12. isEqualTo

`isEqualTo` *est une méthode héritée de l'asserter* `object` *.*

*Pour plus d'informations, reportez-vous à la documentation de* `object::isEqualTo`

#### 2.1.8.13. isIdenticalTo

`isIdenticalTo` *est une méthode héritée de l'asserter* `object` *.*

*Pour plus d'informations, reportez-vous à la documentation de* `object::isIdenticalTo`

#### 2.1.8.14. isInstanceOf

`isInstanceOf` *est une méthode héritée de l'asserter* `object` *.*

*Pour plus d'informations, reportez-vous à la documentation de* `object::isInstanceOf`

#### 2.1.8.15. isNotEqualTo

`isNotEqualTo` *est une méthode héritée de l'asserter* `object` *.*

*Pour plus d'informations, reportez-vous à la documentation de* `object::isNotEqualTo`

#### 2.1.8.16. isNotIdenticalTo

`isNotIdenticalTo` *est une méthode héritée de l'asserter* `object` *.*

*Pour plus d'informations, reportez-vous à la documentation de* `object::isNotIdenticalTo`

### 2.1.9. mysqlDateTime

C'est l'assertion dédiée aux objets décrivant une date MySQL et basée sur l'objet `DateTime (http://php.net/datetime)` .

Les dates doivent utiliser un format compatible avec MySQL et de nombreux autre SGBD (Système de gestion de base de données)), à savoir « Y-m-d H:i:s » (reportez-vous à la documentation de la fonction `date() (http://php.net/date)` du manuel de PHP pour plus d'information).

Si vous essayez de tester une variable qui n'est pas un objet `DateTime` (ou une classe qui l'étend) avec cette assertion, cela échouera.

#### 2.1.9.1. hasDate

`hasDate` *est une méthode héritée de l'asserter* `dateTime` *.*
*Pour plus d'informations, reportez-vous à la documentation de* `dateTime::hasDate`

#### 2.1.9.2. hasDateAndTime

`hasDateAndTime` *est une méthode héritée de l'asserter* `dateTime` *.*
*Pour plus d'informations, reportez-vous à la documentation de* `dateTime::hasDateAndTime`

#### 2.1.9.3. hasDay

`hasDay` *est une méthode héritée de l'asserter* `dateTime` *.*
*Pour plus d'informations, reportez-vous à la documentation de* `dateTime::hasDay`

#### 2.1.9.4. hasHours

`hasHours` *est une méthode héritée de l'asserter* `dateTime` *.*
*Pour plus d'informations, reportez-vous à la documentation de* `dateTime::hasHours`

#### 2.1.9.5. hasMinutes

`hasMinutes` *est une méthode héritée de l'asserter* `dateTime` *.*
*Pour plus d'informations, reportez-vous à la documentation de* `dateTime::hasMinutes`

#### 2.1.9.6. hasMonth

`hasMonth` *est une méthode héritée de l'asserter* `dateTime` *.*

*Pour plus d'informations, reportez-vous à la documentation de* `dateTime::hasMonth`

### 2.1.9.7. hasSeconds

`hasSeconds` *est une méthode héritée de l'asserter* `dateTime` *.*

*Pour plus d'informations, reportez-vous à la documentation de* `dateTime::hasSeconds`

### 2.1.9.8. hasTime

`hasTime` *est une méthode héritée de l'asserter* `dateTime` *.*

*Pour plus d'informations, reportez-vous à la documentation de* `dateTime::hasTime`

### 2.1.9.9. hasTimezone

`hasTimezone` *est une méthode héritée de l'asserter* `dateTime` *.*

*Pour plus d'informations, reportez-vous à la documentation de* `dateTime::hasTimezone`

### 2.1.9.10. hasYear

`hasYear` *est une méthode héritée de l'asserter* `dateTime` *.*

*Pour plus d'informations, reportez-vous à la documentation de* `dateTime::hasYear`

### 2.1.9.11. isCloneOf

`isCloneOf` *est une méthode héritée de l'asserter* `object` *.*

*Pour plus d'informations, reportez-vous à la documentation de* `object::isCloneOf`

### 2.1.9.12. isEqualTo

`isEqualTo` *est une méthode héritée de l'asserter* `object` *.*

*Pour plus d'informations, reportez-vous à la documentation de* `object::isEqualTo`

#### 2.1.9.13. isIdenticalTo

`isIdenticalTo` *est une méthode héritée de l'asserter* `object` *.*

*Pour plus d'informations, reportez-vous à la documentation de* `object::isIdenticalTo`

#### 2.1.9.14. isInstanceOf

`isInstanceOf` *est une méthode héritée de l'asserter* `object` *.*

*Pour plus d'informations, reportez-vous à la documentation de* `object::isInstanceOf`

#### 2.1.9.15. isNotEqualTo

`isNotEqualTo` *est une méthode héritée de l'asserter* `object` *.*

*Pour plus d'informations, reportez-vous à la documentation de* `object::isNotEqualTo`

#### 2.1.9.16. isNotIdenticalTo

`isNotIdenticalTo` *est une méthode héritée de l'asserter* `object` *.*

*Pour plus d'informations, reportez-vous à la documentation de* `object::isNotIdenticalTo`

### 2.1.10. exception

C'est l'assertion dédiée aux exceptions.

```
$this
    ->exception(
        function() use($myObject) {
            // ce code lève une exception: throw new \Exception;
            $myObject->doOneThing('wrongParameter');
        }
    )
;
```

*La syntaxe utilise les fonctions anonymes (aussi appelées fermetures ou closures) introduites en PHP 5.3. Reportez-vous au manuel de PHP (http://php.net/functions.anonymous) pour avoir plus d'informations sur le sujet.*

#### 2.1.10.1. hasCode

`hasCode` vérifie le code de l'exception.

```
$this
    ->exception(
        function() use($myObject) {
            // ce code lève une exception: throw new \Exception('Mess
            $myObject->doOneThing('wrongParameter');
        }
    )
        ->hasCode(42)
;
```

#### 2.1.10.2. hasDefaultCode

`hasDefaultCode` vérifie que le code de l'exception est la valeur par défaut, c'est-à-dire 0.

```
$this
    ->exception(
        function() use($myObject) {
            // ce code lève une exception: throw new \Exception;
            $myObject->doOneThing('wrongParameter');
        }
    )
        ->hasDefaultCode()
;
```

`hasDefaultCode` *est équivalent à* `hasCode(0)` *.*

### 2.1.10.3. hasMessage

`hasMessage` vérifie le message de l'exception.

```
$this
    ->exception(
        function() use($myObject) {
            // ce code lève une exception: throw new \Exception('Mess
            $myObject->doOneThing('wrongParameter');
        }
    )
        ->hasMessage('Message')     // passe
        ->hasMessage('message')     // échoue
;
```

### 2.1.10.4. hasNestedException

`hasNestedException` vérifie que l'exception contient une référence vers l'exception précédente. Si l'exception est précisée, cela va également vérifier la classe de l'exception.

```
$this
    ->exception(
        function() use($myObject) {
            // ce code lève une exception: throw new \Exception('Mess
            $myObject->doOneThing('wrongParameter');
        }
    )
        ->hasNestedException()      // échoue

    ->exception(
        function() use($myObject) {
            try {
                // ce code lève une exception: throw new \FirstExcept
                $myObject->doOneThing('wrongParameter');
            }
            // ... l'exception est attrapée...
            catch(\FirstException $e) {
                // ... puis relancée, encapsulée dans une seconde exc
                throw new \SecondException('Message 2', 24, $e);
            }
        }
    )
        ->isInstanceOf('\FirstException')           // échoue
        ->isInstanceOf('\SecondException')          // passe

        ->hasNestedException()                      // passe
        ->hasNestedException(new \FirstException)   // passe
        ->hasNestedException(new \SecondException)  // échoue
;
```

#### 2.1.10.5. isCloneOf

`isCloneOf` *est une méthode héritée de l'asserter* `object` *.*

*Pour plus d'informations, reportez-vous à la documentation de* `object::isCloneOf`

#### 2.1.10.6. isEqualTo

`isEqualTo` *est une méthode héritée de l'asserter* `object` *.*

*Pour plus d'informations, reportez-vous à la documentation de* `object::isEqualTo`

#### 2.1.10.7. isIdenticalTo

`isIdenticalTo` *est une méthode héritée de l'asserter* `object` *.*

*Pour plus d'informations, reportez-vous à la documentation de* `object::isIdenticalTo`

### 2.1.10.8. isInstanceOf

`isInstanceOf` *est une méthode héritée de l'asserter* `object` *.*

*Pour plus d'informations, reportez-vous à la documentation de* `object::isInstanceOf`

### 2.1.10.9. isNotEqualTo

`isNotEqualTo` *est une méthode héritée de l'asserter* `object` *.*

*Pour plus d'informations, reportez-vous à la documentation de* `object::isNotEqualTo`

### 2.1.10.10. isNotIdenticalTo

`isNotIdenticalTo` *est une méthode héritée de l'asserter* `object` *.*

*Pour plus d'informations, reportez-vous à la documentation de* `object::isNotIdenticalTo`

### 2.1.10.11. message

`message` vous permet de récupérer un asserter de type `string` contenant le message de l'exception testée.

```
1.  $this
2.      ->exception(
3.          function() {
4.              throw new \Exception('My custom message to test');
5.          }
6.      )
7.          ->message
8.              ->contains('message')
9.  ;
```

## 2.1.11. array

C’est l’assertion dédiée aux tableaux.

> `array` *étant un mot réservé en PHP, il n’a pas été possible de créer une assertion* `array` *. Elle s’appelle donc* `phpArray` *et un alias* `array` *a été créé. Vous pourrez donc rencontrer des* `->phpArray()` *ou des* `->array()` *.*

Il est conseillé d’utiliser exclusivement `->array()` afin de simplifier la lecture des tests.

#### 2.1.11.1. contains

`contains` vérifie qu’un tableau contient une certaine donnée.

```
1.  $fibonacci = array (http://www.php.net/array)('1', 2, '3', 5, '8', 13
2.
3.  $this
4.      ->array (http://www.php.net/array)($fibonacci)
5.          ->contains('1')     // passe
6.          ->contains(1)       // passe, ne vérifie pas...
7.          ->contains('2')     // ... le type de la donnée
8.          ->contains(10)      // échoue
9.  ;
```

> `contains` *ne fait pas de recherche récursive.*

> `contains` *ne teste pas le type de la donnée. Si vous souhaitez vérifier également son type, utilisez* `strictlyContains` *.*

#### 2.1.11.2. containsValues

`containsValues` vérifie qu’un tableau contient toutes les données fournies dans un tableau.

```
1.  $fibonacci = array (http://www.php.net/array)('1', 2, '3', 5, '8', 13
2.
3.  $this
4.      ->array (http://www.php.net/array)($array)
5.          ->containsValues(array (http://www.php.net/array)(1, 2, 3))
6.          ->containsValues(array (http://www.php.net/array)('5', '8', '
7.          ->containsValues(array (http://www.php.net/array)(0, 1, 2))
8.  ;
```

`containsValues` *ne fait pas de recherche récursive.*

`containsValues` *ne teste pas le type des données. Si vous souhaitez vérifier également leurs types, utilisez* `strictlyContainsValues` *.*

### 2.1.11.3. hasKey

`hasKey` vérifie qu'un tableau contient une certaine clef.

```
 1.  $fibonacci = array (http://www.php.net/array)('1', 2, '3', 5, '8', 13
 2.  $atoum     = array (http://www.php.net/array)(
 3.      'name'         => 'atoum',
 4.      'owner'        => 'mageekguy',
 5.  );
 6.
 7.  $this
 8.      ->array (http://www.php.net/array)($fibonacci)
 9.          ->hasKey(0)          // passe
10.          ->hasKey(1)          // passe
11.          ->hasKey('1')        // passe
12.          ->hasKey(10)         // échoue
13.
14.      ->array (http://www.php.net/array)($atoum)
15.          ->hasKey('name')     // passe
16.          ->hasKey('price')    // échoue
17.  ;
```

`hasKey` *ne fait pas de recherche récursive.*

`hasKey` *ne teste pas le type des clefs.*

### 2.1.11.4. hasKeys

`hasKeys` vérifie qu'un tableau contient toutes les clefs fournies dans un tableau.

```
 1.  $fibonacci = array (http://www.php.net/array)('1', 2, '3', 5, '8', 13
 2.  $atoum     = array (http://www.php.net/array)(
 3.      'name'         => 'atoum',
 4.      'owner'        => 'mageekguy',
 5.  );
 6.
 7.  $this
 8.      ->array (http://www.php.net/array)($fibonacci)
 9.          ->hasKeys(array (http://www.php.net/array)(0, 2, 4))
10.          ->hasKeys(array (http://www.php.net/array)('0', 2))
11.          ->hasKeys(array (http://www.php.net/array)('4', 0, 3))
12.          ->hasKeys(array (http://www.php.net/array)(0, 3, 10))
13.
14.      ->array (http://www.php.net/array)($atoum)
15.          ->hasKeys(array (http://www.php.net/array)('name', 'owner'))
16.          ->hasKeys(array (http://www.php.net/array)('name', 'price'))
17.  ;
```

`hasKeys` *ne fait pas de recherche récursive.*

`hasKeys` *ne teste pas le type des clefs.*

### 2.1.11.5. hasSize

`hasSize` vérifie la taille d'un tableau.

```
1.  $fibonacci = array (http://www.php.net/array)('1', 2, '3', 5, '8', 13
2.
3.  $this
4.      ->array (http://www.php.net/array)($fibonacci)
5.          ->hasSize(7)        // passe
6.          ->hasSize(10)       // échoue
7.  ;
```

`hasSize` *n'est pas récursif.*

### 2.1.11.6. isEmpty

`isEmpty` vérifie qu'un tableau est vide.

```
 1.  $emptyArray    = array (http://www.php.net/array)();
 2.  $nonEmptyArray = array (http://www.php.net/array)(null, null);
 3.
 4.  $this
 5.      ->array (http://www.php.net/array)($emptyArray)
 6.          ->isEmpty()          // passe
 7.
 8.      ->array (http://www.php.net/array)($nonEmptyArray)
 9.          ->isEmpty()          // échoue
10.  ;
```

### 2.1.11.7. isEqualTo

`isEqualTo` *est une méthode héritée de l'asserter* `variable` *.*

*Pour plus d'informations, reportez-vous à la documentation de* `variable::isEqualTo`

### 2.1.11.8. isIdenticalTo

`isIdenticalTo` *est une méthode héritée de l'asserter* `variable` *.*

*Pour plus d'informations, reportez-vous à la documentation de* `variable::isIdenticalTo`

### 2.1.11.9. isNotEmpty

`isNotEmpty` vérifie qu'un tableau n'est pas vide.

```
 1.  $emptyArray    = array (http://www.php.net/array)();
 2.  $nonEmptyArray = array (http://www.php.net/array)(null, null);
 3.
 4.  $this
 5.      ->array (http://www.php.net/array)($emptyArray)
 6.          ->isNotEmpty()      // échoue
 7.
 8.      ->array (http://www.php.net/array)($nonEmptyArray)
 9.          ->isNotEmpty()      // passe
10.  ;
```

### 2.1.11.10. isNotEqualTo

`isNotEqualTo` *est une méthode héritée de l'asserter* `variable` *.*

*Pour plus d'informations, reportez-vous à la documentation de* `variable::isNotEqualTo`

### 2.1.11.11. isNotIdenticalTo

`isNotIdenticalTo` *est une méthode héritée de l'asserter* `variable` *.*

*Pour plus d'informations, reportez-vous à la documentation de* `variable::isNotIdenticalTo`

### 2.1.11.12. keys

`keys` vous permet de récupérer un asserter de type `array` contenant les clefs du tableau testé.

```
 1.  $atoum = array (http://www.php.net/array)(
 2.      'name'  => 'atoum',
 3.      'owner' => 'mageekguy',
 4.  );
 5.
 6.  $this
 7.      ->array (http://www.php.net/array)($atoum)
 8.          ->keys
 9.              ->isEqualTo(
10.                  array (http://www.php.net/array)(
11.                      'name',
12.                      'owner',
13.                  )
14.              )
15.  ;
```

### 2.1.11.13. notContains

`notContains` vérifie qu'un tableau ne contient pas une donnée.

```
1.  $fibonacci = array (http://www.php.net/array)('1', 2, '3', 5, '8', 13
2.
3.  $this
4.      ->array (http://www.php.net/array)($fibonacci)
5.          ->notContains(null)          // passe
6.          ->notContains(1)             // échoue
7.          ->notContains(10)            // passe
8.  ;
```

`notContains` *ne fait pas de recherche récursive.*

`notContains` *ne teste pas le type de la donnée. Si vous souhaitez vérifier également son type, utilisez* `strictlyNotContains` *.*

### 2.1.11.14. notContainsValues

`notContainsValues` vérifie qu'un tableau ne contient aucune des données fournies dans un tableau.

```
1.  $fibonacci = array (http://www.php.net/array)('1', 2, '3', 5, '8', 13
2.
3.  $this
4.      ->array (http://www.php.net/array)($array)
5.          ->notContainsValues(array (http://www.php.net/array)(1, 4, 10
6.          ->notContainsValues(array (http://www.php.net/array)(4, 10, 3
7.          ->notContainsValues(array (http://www.php.net/array)(1, '2',
8.  ;
```

`notContainsValues` *ne fait pas de recherche récursive.*

`notContainsValues` *ne teste pas le type des données. Si vous souhaitez vérifier également leurs types, utilisez* `strictlyNotContainsValues` *.*

### 2.1.11.15. notHasKey

`notHasKey` vérifie qu'un tableau ne contient pas une certaine clef.

```
 1.  $fibonacci = array (http://www.php.net/array)('1', 2, '3', 5, '8', 13
 2.  $atoum     = array (http://www.php.net/array)(
 3.      'name'  => 'atoum',
 4.      'owner' => 'mageekguy',
 5.  );
 6.
 7.  $this
 8.      ->array (http://www.php.net/array)($fibonacci)
 9.          ->notHasKey(0)          // échoue
10.          ->notHasKey(1)          // échoue
11.          ->notHasKey('1')        // échoue
12.          ->notHasKey(10)         // passe
13.
14.      ->array (http://www.php.net/array)($atoum)
15.          ->notHasKey('name')     // échoue
16.          ->notHasKey('price')    // passe
17.  ;
```

`notHasKey` *ne fait pas de recherche récursive.*

`notHasKey` *ne teste pas le type des clefs.*

### 2.1.11.16. notHasKeys

`notHasKeys` vérifie qu'un tableau ne contient aucune des clefs fournies dans un tableau.

```
 1.  $fibonacci = array (http://www.php.net/array)('1', 2, '3', 5, '8', 13
 2.  $atoum     = array (http://www.php.net/array)(
 3.      'name'         => 'atoum',
 4.      'owner'        => 'mageekguy',
 5.  );
 6.
 7.  $this
 8.      ->array (http://www.php.net/array)($fibonacci)
 9.          ->notHasKeys(array (http://www.php.net/array)(0, 2, 4))
10.          ->notHasKeys(array (http://www.php.net/array)('0', 2))
11.          ->notHasKeys(array (http://www.php.net/array)('4', 0, 3))
12.          ->notHasKeys(array (http://www.php.net/array)(10, 11, 12))
13.
14.      ->array (http://www.php.net/array)($atoum)
15.          ->notHasKeys(array (http://www.php.net/array)('name', 'owner'
16.          ->notHasKeys(array (http://www.php.net/array)('foo', 'price')
17.  ;
```

`notHasKeys` *ne fait pas de recherche récursive.*

`notHasKeys` *ne teste pas le type des clefs.*

### 2.1.11.17. size

`size` vous permet de récupérer un asserter de type `integer` contenant la taille du tableau testé.

```
1.  $fibonacci = array (http://www.php.net/array)('1', 2, '3', 5, '8', 13
2.
3.  $this
4.      ->array (http://www.php.net/array)($fibonacci)
5.          ->size
6.              ->isGreaterThan(5)
7.  ;
```

### 2.1.11.18. strictlyContains

`strictlyContains` vérifie qu'un tableau contient une certaine donnée (même valeur et même type).

```
 1.  $fibonacci = array (http://www.php.net/array)('1', 2, '3', 5, '8', 13
 2.
 3.  $this
 4.      ->array (http://www.php.net/array)($fibonacci)
 5.          ->strictlyContains('1')     // passe
 6.          ->strictlyContains(1)       // échoue
 7.          ->strictlyContains('2')     // échoue
 8.          ->strictlyContains(2)       // passe
 9.          ->strictlyContains(10)      // échoue
10.  ;
```

`strictlyContains` *ne fait pas de recherche récursive.*

`strictlyContains` *teste le type de la donnée. Si vous ne souhaitez pas vérifier son type, utilisez* `contains` *.*

### 2.1.11.19. strictlyContainsValues

`strictlyContainsValues` vérifie qu'un tableau contient toutes les données fournies dans un tableau (même valeur et même type).

```
 1.  $fibonacci = array (http://www.php.net/array)('1', 2, '3', 5, '8', 13
 2.
 3.  $this
 4.      ->array (http://www.php.net/array)($array)
 5.          ->strictlyContainsValues(array (http://www.php.net/array)('1'
 6.          ->strictlyContainsValues(array (http://www.php.net/array)(1,
 7.          ->strictlyContainsValues(array (http://www.php.net/array)(5,
 8.          ->strictlyContainsValues(array (http://www.php.net/array)('5'
 9.          ->strictlyContainsValues(array (http://www.php.net/array)(0,
10.  ;
```

`strictlyContainsValues` *ne fait pas de recherche récursive.*

`strictlyContainsValues` *teste le type des données. Si vous ne souhaitez pas vérifier leurs types, utilisez* `containsValues` *.*

### 2.1.11.20. strictlyNotContains

`strictlyNotContains` vérifie qu'un tableau ne contient pas une donnée (même valeur et même type).

```
1.  $fibonacci = array (http://www.php.net/array)('1', 2, '3', 5, '8', 13
2.
3.  $this
4.      ->array (http://www.php.net/array)($fibonacci)
5.          ->strictlyNotContains(null)          // passe
6.          ->strictlyNotContains('1')           // échoue
7.          ->strictlyNotContains(1)             // passe
8.          ->strictlyNotContains(10)            // passe
9.  ;
```

`strictlyNotContains` *ne fait pas de recherche récursive.*

`strictlyNotContains` *teste le type de la donnée. Si vous ne souhaitez pas vérifier son type, utilisez* `notContains` *.*

### 2.1.11.21. strictlyNotContainsValues

`strictlyNotContainsValues` vérifie qu'un tableau ne contient aucune des données fournies dans un tableau (même valeur et même type).

```
 1.  $fibonacci = array (http://www.php.net/array)('1', 2, '3', 5, '8', 13
 2.
 3.  $this
 4.      ->array (http://www.php.net/array)($array)
 5.          ->strictlyNotContainsValues(array (http://www.php.net/array)(
 6.          ->strictlyNotContainsValues(array (http://www.php.net/array)(
 7.          ->strictlyNotContainsValues(array (http://www.php.net/array)(
 8.          ->strictlyNotContainsValues(array (http://www.php.net/array)(
 9.          ->strictlyNotContainsValues(array (http://www.php.net/array)(
10.  ;
```

`strictlyNotContainsValues` *ne fait pas de recherche récursive.*

`strictlyNotContainsValues` *teste le type des données. Si vous ne souhaitez pas vérifier leurs types, utilisez* `notContainsValues` *.*

## 2.1.12. string

C'est l'assertion dédiée aux chaînes de caractères.

### 2.1.12.1. contains

`contains` vérifie qu'une chaîne de caractère contient une autre chaîne de caractère donnée.

```
1.  $string = 'Hello world';
2.
3.  $this
4.      ->string($string)
5.          ->contains('ll')    // passe
6.          ->contains(' ')     // passe
7.          ->contains('php')   // échoue
8.  ;
```

### 2.1.12.2. hasLength

`hasLength` vérifie la taille d'une chaîne de caractères.

```
1.  $string = 'Hello world';
2.
3.  $this
4.      ->string($string)
5.          ->hasLength(11)     // passe
6.          ->hasLength(20)     // échoue
7.  ;
```

### 2.1.12.3. hasLengthGreaterThan

`hasLengthGreaterThan` vérifie que la taille d'une chaîne de caractères est plus grande qu'une valeur donnée.

```
1.  $string = 'Hello world';
2.
3.  $this
4.      ->string($string)
5.          ->hasLengthGreaterThan(10)      // passe
6.          ->hasLengthGreaterThan(20)      // échoue
7.  ;
```

#### 2.1.12.4. hasLengthLessThan

`hasLengthLessThan` vérifie que la taille d'une chaîne de caractères est plus petite qu'une valeur donnée.

```
1.  $string = 'Hello world';
2.
3.  $this
4.      ->string($string)
5.          ->hasLengthLessThan(20)      // passe
6.          ->hasLengthLessThan(10)      // échoue
7.  ;
```

#### 2.1.12.5. isEmpty

`isEmpty` vérifie qu'une chaîne de caractères est vide.

```
 1.  $emptyString    = '';
 2.  $nonEmptyString = 'atoum';
 3.
 4.  $this
 5.      ->string($emptyString)
 6.          ->isEmpty()              // passe
 7.
 8.      ->string($nonEmptyString)
 9.          ->isEmpty()              // échoue
10.  ;
```

#### 2.1.12.6. isEqualTo

`isEqualTo` *est une méthode héritée de l'asserter* `variable` *.*

*Pour plus d'informations, reportez-vous à la documentation de* `variable::isEqualTo`

### 2.1.12.7. isEqualToContentsOfFile

`isEqualToContentsOfFile` vérifie qu'une chaîne de caractère est égale au contenu d'un fichier donné par son chemin.

```
1.  $this
2.      ->string($string)
3.          ->isEqualToContentsOfFile('/path/to/file')
4.  ;
```

*si le fichier n'existe pas, le test échoue.*

### 2.1.12.8. isIdenticalTo

`isIdenticalTo` *est une méthode héritée de l'asserter* `variable`*.*

*Pour plus d'informations, reportez-vous à la documentation de* `variable::isIdenticalTo`

### 2.1.12.9. isNotEmpty

`isNotEmpty` vérifie qu'une chaîne de caractères n'est pas vide.

```
 1.  $emptyString    = '';
 2.  $nonEmptyString = 'atoum';
 3.
 4.  $this
 5.      ->string($emptyString)
 6.          ->isNotEmpty()          // échoue
 7.
 8.      ->string($nonEmptyString)
 9.          ->isNotEmpty()          // passe
10.  ;
```

### 2.1.12.10. isNotEqualTo

`isNotEqualTo` *est une méthode héritée de l'asserter* `variable`*.*

*Pour plus d'informations, reportez-vous à la documentation de* `variable::isNotEqualTo`

### 2.1.12.11. isNotIdenticalTo

`isNotIdenticalTo` *est une méthode héritée de l'asserter* `variable` *.*
*Pour plus d'informations, reportez-vous à la documentation de* `variable::isNotIdenticalTo`

### 2.1.12.12. length

`length` vous permet de récupérer un asserter de type `integer` contenant la taille de la chaîne de caractères testée.

```
1.  $string = 'atoum'
2.
3.  $this
4.      ->string($string)
5.          ->length
6.              ->isGreaterThanOrEqualTo(5)
7.  ;
```

### 2.1.12.13. match

`match` vérifie qu'une expression régulière correspond à la chaîne de caractères.

```
 1.  $phone = '0102030405';
 2.  $vdm   = "Aujourd'hui, à 57 ans, mon père s'est fait tatouer une lico
 3.
 4.  $this
 5.      ->string($phone)
 6.          ->match('#^0[1-9]\d{8}$#')
 7.
 8.      ->string($vdm)
 9.          ->match("#^Aujourd'hui.*VDM$#")
10.  ;
```

### 2.1.12.14. notContains

`notContains` vérifie qu'une chaîne de caractère ne contient pas une autre chaîne de caractère donnée.

```
1.  $string = 'Hello world';
2.
3.  $this
4.      ->string($string)
5.          ->notContains('php')    // passe
6.          ->notContains(';')      // passe
7.          ->notContains('ll')     // échoue
8.          ->notContains(' ')      // échoue
9.  ;
```

## 2.1.13. castToString

C’est l’assertion dédiée aux tests sur le transtypage d’objets en chaîne de caractères.

```
1.  class AtoumVersion {
2.      private $version = '1.0';
3.
4.      public function __toString() {
5.          return 'atoum v' . $this->version;
6.      }
7.  }
8.
9.  $this
10.     ->castToString(new AtoumVersion())
11.         ->isEqualTo('atoum v1.0')
12. ;
```

### 2.1.13.1. contains

`contains` *est une méthode héritée de l’asserter* `string` *.*

*Pour plus d’informations, reportez-vous à la documentation de* `string::contains`

### 2.1.13.2. notContains

`notContains` *est une méthode héritée de l’asserter* `string` *.*

*Pour plus d’informations, reportez-vous à la documentation de* `string::notContains`

### 2.1.13.3. hasLength

`hasLength` *est une méthode héritée de l'asserter* `string` *.*

*Pour plus d'informations, reportez-vous à la documentation de* `string::hasLength`

#### 2.1.13.4. hasLengthGreaterThan

`hasLengthGreaterThan` *est une méthode héritée de l'asserter* `string` *.*

*Pour plus d'informations, reportez-vous à la documentation de* `string::hasLengthGreaterThan`

#### 2.1.13.5. hasLengthLessThan

`hasLengthLessThan` *est une méthode héritée de l'asserter* `string` *.*

*Pour plus d'informations, reportez-vous à la documentation de* `string::hasLengthLessThan`

#### 2.1.13.6. isEmpty

`isEmpty` *est une méthode héritée de l'asserter* `string` *.*

*Pour plus d'informations, reportez-vous à la documentation de* `string::isEmpty`

#### 2.1.13.7. isEqualTo

`isEqualTo` *est une méthode héritée de l'asserter* `variable` *.*

*Pour plus d'informations, reportez-vous à la documentation de* `variable::isEqualTo`

#### 2.1.13.8. isEqualToContentsOfFile

`isEqualToContentsOfFile` *est une méthode héritée de l'asserter* `string` *.*

*Pour plus d'informations, reportez-vous à la documentation de* `string::isEqualToContentsOfFile`

#### 2.1.13.9. isIdenticalTo

`isIdenticalTo` *est une méthode héritée de l'asserter* `variable` *.*

*Pour plus d'informations, reportez-vous à la documentation de* `variable::isIdenticalTo`

### 2.1.13.10. isNotEmpty

`isNotEmpty` *est une méthode héritée de l'asserter* `string` *.*

*Pour plus d'informations, reportez-vous à la documentation de* `string::isNotEmpty`

### 2.1.13.11. isNotEqualTo

`isNotEqualTo` *est une méthode héritée de l'asserter* `variable` *.*

*Pour plus d'informations, reportez-vous à la documentation de* `variable::isNotEqualTo`

### 2.1.13.12. isNotIdenticalTo

`isNotIdenticalTo` *est une méthode héritée de l'asserter* `variable` *.*

*Pour plus d'informations, reportez-vous à la documentation de* `variable::isNotIdenticalTo`

### 2.1.13.13. match

`match` *est une méthode héritée de l'asserter* `string` *.*

*Pour plus d'informations, reportez-vous à la documentation de* `string::match`

## 2.1.14. hash

C'est l'assertion dédiée aux tests sur les hashs (empreintes numériques).

### 2.1.14.1. contains

`contains` *est une méthode héritée de l'asserter* `string` *.*

*Pour plus d'informations, reportez-vous à la documentation de* `string::contains`

### 2.1.14.2. isEqualTo

`isEqualTo` *est une méthode héritée de l'asserter* `variable` *.*

*Pour plus d'informations, reportez-vous à la documentation de* `variable::isEqualTo`

### 2.1.14.3. isEqualToContentsOfFile

`isEqualToContentsOfFile` *est une méthode héritée de l'asserter* `string` *.*

*Pour plus d'informations, reportez-vous à la documentation de* `string::isEqualToContentsOfFile`

### 2.1.14.4. isIdenticalTo

`isIdenticalTo` *est une méthode héritée de l'asserter* `variable` *.*

*Pour plus d'informations, reportez-vous à la documentation de* `variable::isIdenticalTo`

### 2.1.14.5. isMd5

`isMd5` vérifie que la chaîne de caractère est au format `md5` , c'est-à-dire une chaîne hexadécimale de 32 caractères.

```
1.  $hash    = hash (http://www.php.net/hash)('md5', 'atoum');
2.  $notHash = 'atoum';
3.
4.  $this
5.      ->hash (http://www.php.net/hash)($hash)
6.          ->isMd5()       // passe
7.      ->hash (http://www.php.net/hash)($notHash)
8.          ->isMd5()       // échoue
9.  ;
```

### 2.1.14.6. isNotEqualTo

`isNotEqualTo` *est une méthode héritée de l'asserter* `variable` *.*

*Pour plus d'informations, reportez-vous à la documentation de* `variable::isNotEqualTo`

### 2.1.14.7. isNotIdenticalTo

`isNotIdenticalTo` *est une méthode héritée de l'asserter* `variable` *.*
*Pour plus d'informations, reportez-vous à la documentation de*
`variable::isNotIdenticalTo`

### 2.1.14.8. isSha1

`isSha1` vérifie que la chaîne de caractère est au format `sha1` , c'est-à-dire une chaîne hexadécimale de 40 caractères.

```
1.  $hash    = hash (http://www.php.net/hash)('sha1', 'atoum');
2.  $notHash = 'atoum';
3.
4.  $this
5.      ->hash (http://www.php.net/hash)($hash)
6.          ->isSha1()      // passe
7.      ->hash (http://www.php.net/hash)($notHash)
8.          ->isSha1()      // échoue
9.  ;
```

### 2.1.14.9. isSha256

`isSha256` vérifie que la chaîne de caractère est au format `sha256` , c'est-à-dire une chaîne hexadécimale de 64 caractères.

```
1.  $hash    = hash (http://www.php.net/hash)('sha256', 'atoum');
2.  $notHash = 'atoum';
3.
4.  $this
5.      ->hash (http://www.php.net/hash)($hash)
6.          ->isSha256()    // passe
7.      ->hash (http://www.php.net/hash)($notHash)
8.          ->isSha256()    // échoue
9.  ;
```

### 2.1.14.10. isSha512

`isSha512` vérifie que la chaîne de caractère est au format `sha512` , c'est-à-dire une chaîne hexadécimale de 128 caractères.

```
1.  $hash    = hash (http://www.php.net/hash)('sha512', 'atoum');
2.  $notHash = 'atoum';
3.
4.  $this
5.      ->hash (http://www.php.net/hash)($hash)
6.          ->isSha512()    // passe
7.      ->hash (http://www.php.net/hash)($notHash)
8.          ->isSha512()    // échoue
9.  ;
```

#### 2.1.14.11. notContains

`notContains` *est une méthode héritée de l'asserter* `string` *.*
*Pour plus d'informations, reportez-vous à la documentation de* `string::notContains`

### 2.1.15. output

C'est l'assertion dédiée aux tests sur les sorties, c'est-à-dire tout ce qui est censé être affiché à l'écran.

```
1.  $this
2.      ->output(
3.          function() {
4.              echo 'Hello world';
5.          }
6.      )
7.  ;
```

*La syntaxe utilise les fonctions anonymes (aussi appelées fermetures ou closures) introduites en PHP 5.3. Reportez-vous au manuel de PHP (http://php.net/functions.anonymous) pour avoir plus d'informations sur le sujet.*

#### 2.1.15.1. contains

`contains` *est une méthode héritée de l'asserter* `string` *.*
*Pour plus d'informations, reportez-vous à la documentation de* `string::contains`

### 2.1.15.2. hasLength

`hasLength` *est une méthode héritée de l'asserter* `string` *.*

*Pour plus d'informations, reportez-vous à la documentation de* `string::hasLength`

### 2.1.15.3. hasLengthGreaterThan

`hasLengthGreaterThan` *est une méthode héritée de l'asserter* `string` *.*

*Pour plus d'informations, reportez-vous à la documentation de* `string::hasLengthGreaterThan`

### 2.1.15.4. hasLengthLessThan

`hasLengthLessThan` *est une méthode héritée de l'asserter* `string` *.*

*Pour plus d'informations, reportez-vous à la documentation de* `string::hasLengthLessThan`

### 2.1.15.5. isEmpty

`isEmpty` *est une méthode héritée de l'asserter* `string` *.*

*Pour plus d'informations, reportez-vous à la documentation de* `string::isEmpty`

### 2.1.15.6. isEqualTo

`isEqualTo` *est une méthode héritée de l'asserter* `variable` *.*

*Pour plus d'informations, reportez-vous à la documentation de* `variable::isEqualTo`

### 2.1.15.7. isEqualToContentsOfFile

`isEqualToContentsOfFile` *est une méthode héritée de l'asserter* `string` *.*

*Pour plus d'informations, reportez-vous à la documentation de* `string::isEqualToContentsOfFile`

### 2.1.15.8. isIdenticalTo

`isIdenticalTo` *est une méthode héritée de l'asserter* `variable` *.*

*Pour plus d'informations, reportez-vous à la documentation de* `variable::isIdenticalTo`

### 2.1.15.9. isNotEmpty

`isNotEmpty` *est une méthode héritée de l'asserter* `string` *.*

*Pour plus d'informations, reportez-vous à la documentation de* `string::isNotEmpty`

### 2.1.15.10. isNotEqualTo

`isNotEqualTo` *est une méthode héritée de l'asserter* `variable` *.*

*Pour plus d'informations, reportez-vous à la documentation de* `variable::isNotEqualTo`

### 2.1.15.11. isNotIdenticalTo

`isNotIdenticalTo` *est une méthode héritée de l'asserter* `variable` *.*

*Pour plus d'informations, reportez-vous à la documentation de* `variable::isNotIdenticalTo`

### 2.1.15.12. match

`match` *est une méthode héritée de l'asserter* `string` *.*

*Pour plus d'informations, reportez-vous à la documentation de* `string::match`

### 2.1.15.13. notContains

`notContains` *est une méthode héritée de l'asserter* `string` *.*

*Pour plus d'informations, reportez-vous à la documentation de* `string::notContains`

## 2.1.16. utf8String

C’est l’assertion dédiée aux chaînes de caractères UTF-8.

> `utf8Strings` *utilise les fonctions* `mb_*` *pour gérer les chaînes multi-octets. Reportez-vous au manuel de PHP pour avoir plus d’information sur l’extension* `mbstring (http://php.net/mbstring)` *.*

#### 2.1.16.1. contains

> `contains` *est une méthode héritée de l’asserter* `string` *.*
> *Pour plus d’informations, reportez-vous à la documentation de* `string::contains`

#### 2.1.16.2. hasLength

> `hasLength` *est une méthode héritée de l’asserter* `string` *.*
> *Pour plus d’informations, reportez-vous à la documentation de* `string::hasLength`

#### 2.1.16.3. hasLengthGreaterThan

> `hasLengthGreaterThan` *est une méthode héritée de l’asserter* `string` *.*
> *Pour plus d’informations, reportez-vous à la documentation de* `string::hasLengthGreaterThan`

#### 2.1.16.4. hasLengthLessThan

> `hasLengthLessThan` *est une méthode héritée de l’asserter* `string` *.*
> *Pour plus d’informations, reportez-vous à la documentation de* `string::hasLengthLessThan`

#### 2.1.16.5. isEmpty

> `isEmpty` *est une méthode héritée de l’asserter* `string` *.*
> *Pour plus d’informations, reportez-vous à la documentation de* `string::isEmpty`

#### 2.1.16.6. isEqualTo

`isEqualTo` *est une méthode héritée de l'asserter* `variable` *.*

*Pour plus d'informations, reportez-vous à la documentation de* `variable::isEqualTo`

### 2.1.16.7. isEqualToContentsOfFile

`isEqualToContentsOfFile` *est une méthode héritée de l'asserter* `string` *.*

*Pour plus d'informations, reportez-vous à la documentation de* `string::isEqualToContentsOfFile`

### 2.1.16.8. isIdenticalTo

`isIdenticalTo` *est une méthode héritée de l'asserter* `variable` *.*

*Pour plus d'informations, reportez-vous à la documentation de* `variable::isIdenticalTo`

### 2.1.16.9. isNotEmpty

`isNotEmpty` *est une méthode héritée de l'asserter* `string` *.*

*Pour plus d'informations, reportez-vous à la documentation de* `string::isNotEmpty`

### 2.1.16.10. isNotEqualTo

`isNotEqualTo` *est une méthode héritée de l'asserter* `variable` *.*

*Pour plus d'informations, reportez-vous à la documentation de* `variable::isNotEqualTo`

### 2.1.16.11. isNotIdenticalTo

`isNotIdenticalTo` *est une méthode héritée de l'asserter* `variable` *.*

*Pour plus d'informations, reportez-vous à la documentation de* `variable::isNotIdenticalTo`

### 2.1.16.12. match

`match` *est une méthode héritée de l'asserter* `string` *.*
*Pour plus d'informations, reportez-vous à la documentation de* `string::match`

*Pensez à bien ajouter* `u` *comme option de recherche dans votre expression régulière. Reportez-vous au manuel de PHP (http://php.net/reference.pcre.pattern.modifiers) pour avoir plus d'informations sur le sujet.*

```
1.  $vdm = "Aujourd'hui, à 57 ans, mon père s'est fait tatouer une licorn
2.
3.  $this
4.      ->utf8String($vdm)
5.          ->match("#^Aujourd'hui.*VDM$#u")
6.  ;
```

### 2.1.16.13. notContains

`notContains` *est une méthode héritée de l'asserter* `string` *.*
*Pour plus d'informations, reportez-vous à la documentation de* `string::notContains`

## 2.1.17. afterDestructionOf

C'est l'assertion dédiée à la destruction des objets.

Cette assertion ne fait que prendre un objet, vérifier que la méthode `__destruct()` est bien définie puis l'appelle.

Si `__destruct()` existe bien et si son appel se passe sans erreur ni exception, alors le test passe.

```
1.  $this
2.      ->afterDestructionOf($objectWithDestructor)     // passe
3.      ->afterDestructionOf($objectWithoutDestructor)  // échoue
4.  ;
```

## 2.1.18. error

C’est l’assertion dédiée aux erreurs.

```
$this
    ->when(
        function() {
            trigger_error (http://www.php.net/trigger_error)('message
        }
    )
        ->error()
            ->exists() // ou notExists
;
```

> *La syntaxe utilise les fonctions anonymes (aussi appelées fermetures ou closures) introduites en PHP 5.3. Reportez-vous au manuel de PHP (http://php.net/functions.anonymous) pour avoir plus d’informations sur le sujet.*
>
> *Les types d’erreur E_ERROR, E_PARSE, E_CORE_ERROR, E_CORE_WARNING, E_COMPILE_ERROR, E_COMPILE_WARNING ainsi que la plupart des E_STRICT ne peuvent pas être gérés avec cette fonction.*

#### 2.1.18.1. exists

`exists` vérifie qu’une erreur a été levée lors de l’exécution du code précédent.

```
$this
    ->when(
        function() {
            trigger_error (http://www.php.net/trigger_error)('message
        }
    )
        ->error()
            ->exists()      // passe

    ->when(
        function() {
            // code sans erreur
        }
    )
        ->error()
            ->exists()      // échoue
;
```

### 2.1.18.2. notExists

`notExists` vérifie qu'aucune erreur n'a été levée lors de l'exécution du code précédent.

```
 1.  $this
 2.      ->when(
 3.          function() {
 4.              trigger_error (http://www.php.net/trigger_error)('message
 5.          }
 6.      )
 7.          ->error()
 8.              ->notExists()   // échoue
 9.
10.      ->when(
11.          function() {
12.              // code sans erreur
13.          }
14.      )
15.          ->error()
16.              ->notExists()   // passe
17.  ;
```

### 2.1.18.3. withType

`withType` vérifie le type de l'erreur levée.

```
 1.  $this
 2.      ->when(
 3.          function() {
 4.              trigger_error (http://www.php.net/trigger_error)('message
 5.          }
 6.      )
 7.          ->error()
 8.              ->withType(E_USER_NOTICE)   // passe
 9.              ->withType(E_USER_WARNING)  // échoue
10.  ;
```

## 2.1.19. class

C'est l'assertion dédiée aux classes.

```
1.  $object = new \StdClass;
2.
3.  $this
4.      ->class(get_class (http://www.php.net/get_class)($object))
5.
6.      ->class('\StdClass')
7.  ;
```

> *Le mot-clef* `class` *étant réservé en PHP, il n'a pas été possible de créer une assertion* `class` *. Elle s'appelle donc* `phpClass` *et un alias* `class` *a été créé. Vous pourrez donc rencontrer des* `->phpClass()` *ou des* `->class()` *.*

Il est conseillé d'utiliser exclusivement `->class()` .

#### 2.1.19.1. hasInterface

`hasInterface` vérifie que la classe implémente une interface donnée.

```
1.  $this
2.      ->class('\ArrayIterator')
3.          ->hasInterface('Countable')     // passe
4.
5.      ->class('\StdClass')
6.          ->hasInterface('Countable')     // échoue
7.  ;
```

#### 2.1.19.2. hasMethod

`hasMethod` vérifie que la classe contient une méthode donnée.

```
1.  $this
2.      ->class('\ArrayIterator')
3.          ->hasMethod('count')    // passe
4.
5.      ->class('\StdClass')
6.          ->hasMethod('count')    // échoue
7.  ;
```

#### 2.1.19.3. hasNoParent

`hasNoParent` vérifie que la classe n'hérite d'aucune classe.

```
1.  $this
2.      ->class('\StdClass')
3.          ->hasNoParent()     // passe
4.
5.      ->class('\FilesystemIterator')
6.          ->hasNoParent()     // échoue
7.  ;
```

*Une classe peut implémenter une ou plusieurs interfaces et n'hériter d'aucune classe.* `hasNoParent` *ne vérifie pas les interfaces, uniquement les classes héritées.*

### 2.1.19.4. hasParent

`hasParent` vérifie que la classe hérite bien d'une classe.

```
1.  $this
2.      ->class('\StdClass')
3.          ->hasParent()       // échoue
4.
5.      ->class('\FilesystemIterator')
6.          ->hasParent()       // passe
7.  ;
```

*Une classe peut implémenter une ou plusieurs interfaces et n'hériter d'aucune classe.* `hasParent` *ne vérifie pas les interfaces, uniquement les classes héritées.*

### 2.1.19.5. isAbstract

`isAbstract` vérifie que la classe est abstraite.

```
1.  $this
2.      ->class('\StdClass')
3.          ->isAbstract()      // échoue
4.  ;
```

### 2.1.19.6. isSubclassOf

`isSubclassOf` vérifie que la classe hérite de la classe donnée.

```
1.  $this
2.      ->class('\FilesystemIterator')
3.          ->isSubclassOf('\DirectoryIterator')    // passe
4.          ->isSubclassOf('\SplFileInfo')          // passe
5.          ->isSubclassOf('\StdClass')             // échoue
6.  ;
```

## 2.1.20. mock

C’est l’assertion dédiée aux bouchons.

```
1.  $mock = new \mock\MyClass;
2.
3.  $this
4.      ->mock($mock)
5.  ;
```

> *Reportez-vous à la documentation sur les bouchons pour obtenir plus d’informations sur la façon de créer et gérer les bouchons.*

### 2.1.20.1. call

`call` permet de spécifier une méthode du mock à tester

```
1.  $mock = new \mock\MyFirstClass;
2.
3.  $this
4.      ->object(new MySecondClass($mock))
5.
6.      ->mock($mock)
7.          ->call('myMethod')
8.              ->once()
9.  ;
```

#### 2.1.20.1.1. atLeastOnce

`atLeastOnce` vérifie que la méthode testée (voir `call`) du mock testé a été appelée au moins une fois.

```
1.  $mock = new \mock\MyFirstClass;
2.
3.  $this
4.      ->object(new MySecondClass($mock))
5.
6.      ->mock($mock)
7.          ->call('myMethod')
8.              ->atLeastOnce()
9.  ;
```

#### 2.1.20.1.2. exactly

`exactly` vérifie que la méthode testée (voir `call`) du mock testé exactement un certain nombre de fois.

```
1.  $mock = new \mock\MyFirstClass;
2.
3.  $this
4.      ->object(new MySecondClass($mock))
5.
6.      ->mock($mock)
7.          ->call('myMethod')
8.              ->exactly(2)
9.  ;
```

#### 2.1.20.1.3. never

`never` vérifie que la méthode testée (voir `call`) du mock testé n'a jamais été appelée.

```
1.  $mock = new \mock\MyFirstClass;
2.
3.  $this
4.      ->object(new MySecondClass($mock))
5.
6.      ->mock($mock)
7.          ->call('myMethod')
8.              ->never()
9.  ;
```

`never` *est équivalent à* `exactly(0)`.

#### 2.1.20.1.4. once/twice/thrice

Ces assertions vérifient que la méthode testée (voir `call`) du mock testé a été appelée exactement :

- une fois (once)
- deux fois (twice)
- trois fois (thrice)

```
 1.  $mock = new \mock\MyFirstClass;
 2.
 3.  $this
 4.      ->object(new MySecondClass($mock))
 5.
 6.      ->mock($mock)
 7.          ->call('myMethod')
 8.              ->once()
 9.          ->call('mySecondMethod')
10.              ->twice()
11.          ->call('myThirdMethod')
12.              ->thrice()
13.  ;
```

> `once`, `twice` *et* `thrice` *sont respectivement équivalents à un appel à* `exactly(1)`, `exactly(2)` *et* `exactly(3)`.

#### 2.1.20.1.5. withAnyArguments

`withAnyArguments` permet de ne pas spécifier les arguments attendus lors de l'appel à la méthode testée (voir `call`) du mock testé.

Cette méthode est surtout utile pour remettre à zéro les arguments, comme dans l'exemple suivant :

```
 1.  $mock = new \mock\MyFirstClass;
 2.
 3.  $this
 4.      ->object(new MySecondClass($mock))
 5.
 6.      ->mock($mock)
 7.          ->call('myMethod')
 8.              ->withArguments('first')     ->once()
 9.              ->withArguments('second')    ->once()
10.              ->withAnyArguments()->exactly(2)
11.  ;
```

#### 2.1.20.1.6. withArguments

`withArguments` permet de spécifier les paramètres attendus lors de l'appel à la méthode testée (voir `call`) du mock testé.

```
 1.  $mock = new \mock\MyFirstClass;
 2.
 3.  $this
 4.      ->object(new MySecondClass($mock))
 5.
 6.      ->mock($mock)
 7.          ->call('myMethod')
 8.              ->withArguments('first', 'second')->once()
 9.  ;
```

> `withArguments` *ne teste pas le type des arguments. Si vous souhaitez vérifier également leurs types, utilisez* `withIdenticalArguments`.

#### 2.1.20.1.7. withIdenticalArguments

`withIdenticalArguments` permet de spécifier les paramètres attendus lors de l'appel à la méthode testée (voir `call`) du mock testé.

```
1.  $mock = new \mock\MyFirstClass;
2.
3.  $this
4.      ->object(new MySecondClass($mock))
5.
6.      ->mock($mock)
7.          ->call('myMethod')
8.              ->withIdenticalArguments('first', 'second')->once()
9.  ;
```

`withIdenticalArguments` *teste le type des arguments. Si vous ne souhaitez pas vérifier leurs types, utilisez* `withArguments` *.*

### 2.1.20.2. wasCalled

`wasCalled` vérifie qu'au moins une méthode du mock a été appelée au moins une fois.

```
1.  $mock = new \mock\MyFirstClass;
2.
3.  $this
4.      ->object(new MySecondClass($mock))
5.
6.      ->mock($mock)
7.          ->wasCalled()
8.  ;
```

### 2.1.20.3. wasNotCalled

`wasNotCalled` vérifie qu'aucune méthode du mock n'a été appelée.

```
1.  $mock = new \mock\MyFirstClass;
2.
3.  $this
4.      ->object(new MySecondClass($mock))
5.
6.      ->mock($mock)
7.          ->wasNotCalled()
8.  ;
```

## 2.1.21. stream

C'est l'assertion dédiée aux stream.

*Malheureusement, je n'ai aucune espèce d'idée de son fonctionnement, alors n'hésitez pas à compléter cette partie !*

#### 2.1.21.1. isRead

*We need help to write this section !*

#### 2.1.21.2. isWrite

*We need help to write this section !*

## 2.2. Aide à l'écriture

Il est possible d'écrire des tests unitaires avec atoum de plusieurs manières, et l'une d'elle est d'utiliser des mots-clefs tels que `if`, `and` ou bien encore `then`, `when` ou `assert`.

### 2.2.1. if, and, then

L'utilisation de ces mots-clefs est très intuitive :

```
 1. $this
 2.     ->if($computer = new computer())
 3.     ->and($computer->setFirstOperand(2))
 4.     ->and($computer->setSecondOperand(2))
 5.     ->then
 6.         ->object($computer->add())
 7.             ->isIdenticalTo($computer)
 8.         ->integer($computer->getResult())
 9.             ->isEqualTo(4)
10. ;
```

Il est important de noter ces mots-clefs n'apporte rien techniquement ou fonctionnellement parlant, car ils n'ont pas d'autre but que de faciliter la compréhension du test et donc sa maintenance en y ajoutant de la sémantique compréhensible facilement par l'Humain et plus particulièrement un développeur.

Ainsi, `if` et `and` permettent de définir les conditions préalables pour que les assertions qui suivent le mot-clef `then` passent avec succès.

Cependant, il n'y a pas de grammaire régissant l'ordre d'utilisation de ces mots-clefs et aucune vérification syntaxique n'est effectuée par atoum.

En conséquence, il est de la responsabilité du développeur de les utiliser de façon à ce que le test soit signifiant, même s'il est par exemple tout à fait possible d'écrire le test de la manière suivante :

```
 1.  $this
 2.      ->and($computer = new computer())
 3.      ->and($computer->setFirstOperand(2))
 4.      ->then
 5.      ->if($computer->setSecondOperand(2))
 6.          ->object($computer->add())
 7.              ->isIdenticalTo($computer)
 8.          ->integer($computer->getResult())
 9.              ->isEqualTo(4)
10.  ;
```

Pour les mêmes raisons, l'utilisation de `then` est facultative.

Il est également important de noter qu'il est tout à fait possible d'écrire le même test en n'utilisant aucun mot-clef :

```
 1.  $computer = new computer();
 2.  $computer->setFirstOperand(2);
 3.  $computer->setSecondOperand(2);
 4.
 5.  $this
 6.      ->object($computer->add())
 7.          ->isIdenticalTo($computer)
 8.      ->integer($computer->getResult())
 9.          ->isEqualTo(4)
10.  ;
```

Le test ne sera pas plus lent ou plus rapide à exécuter et il n'y a aucun avantage à utiliser une notation ou une autre, l'important étant d'en choisir une et de s'y tenir pour faciliter la maintenance des tests (la problématique est exactement la même que celle des conventions de codage).

### 2.2.2. when

En plus de `if`, `and` et `then`, il existe également d'autres mots-clefs.

L'un d'entre eux est `when`. Il dispose d'une fonctionnalité spécifique introduite pour contourner le fait qu'il est illégal d'écrire en PHP le code suivant :

```
1.  $this
2.      ->if($object = new object($valueAtKey0 = uniqid (http://www.php.n
3.      ->and(unset (http://www.php.net/unset)($object[0]))
4.      ->then
5.          ->sizeOf (http://www.php.net/sizeof)($object)
6.              ->isZero()
7.  ;
```

Le langage génère en effet dans ce cas l'erreur fatale : `Parse error: syntax error, unexpected 'unset' (T_UNSET), expecting »)'`

Il est en effet impossible d'utiliser `unset()` comme argument d'une fonction.

Pour résoudre ce problème, le mot-clef `when` est capable d'interpréter l'éventuelle fonction anonyme qui lui est passée en argument, ce qui permet d'écrire le test précédent de la manière suivante :

```
 1.  $this
 2.      ->if($object = new object($valueAtKey0 = uniqid (http://www.php.n
 3.      ->when(
 4.          function() use ($object) {
 5.              unset (http://www.php.net/unset)($object[0]);
 6.          }
 7.      )
 8.      ->then
 9.        ->sizeOf (http://www.php.net/sizeof)($object)
10.          ->isZero()
11.  ;
```

Bien évidemment, si `when` ne reçoit pas de fonction anonyme en argument, il se comporte exactement comme if, and et then, à savoir qu'il ne fait absolument rien fonctionnellement parlant.

### 2.2.3. assert

Enfin, il existe le mot-clef `assert` qui a également un fonctionnement un peu particulier.

Pour illustrer son fonctionnement, le test suivant va être utilisé :

```
 1.  $this
 2.      ->if($foo = new \mock\foo())
 3.      ->and($bar = new bar($foo))
 4.      ->and($bar->doSomething())
 5.      ->then
 6.          ->mock($foo)
 7.              ->call('doOtherThing')
 8.                  ->once()
 9.
10.      ->if($bar->setValue(uniqid (http://www.php.net/uniqid)())
11.      ->then
12.          ->mock($foo)
13.              ->call('doOtherThing')
14.                  ->exactly(2)
15.  ;
```

Le test précédent présente un inconvénient en terme de maintenance, car si le développeur a besoin d'intercaler un ou plusieurs nouveaux appels à bar::doOtherThing() entre les deux appels déjà effectués, il sera obligé de mettre à jour en conséquence la valeur de l'argument passé à exactly().

Pour remédier à ce problème, vous pouvez remettre à zéro un mock de 2 manières différentes :

- soit en utilisant $mock->getMockController()->resetCalls() ;
- soit en utilisant $this->resetMock($mock).

```
 1.  $this
 2.      ->if($foo = new \mock\foo())
 3.      ->and($bar = new bar($foo))
 4.      ->and($bar->doSomething())
 5.      ->then
 6.          ->mock($foo)
 7.              ->call('doOtherThing')
 8.                  ->once()
 9.
10.      // 1ère manière
11.      ->if($foo->getMockController()->resetCalls())
12.      ->and($bar->setValue(uniqid (http://www.php.net/uniqid)())
13.      ->then
14.          ->mock($foo)
15.              ->call('doOtherThing')
16.                  ->once()
17.
18.      // 2ème manière
19.      ->if($this->resetMock($foo))
20.      ->and($bar->setValue(uniqid (http://www.php.net/uniqid)())
21.      ->then
22.          ->mock($foo)
23.              ->call('doOtherThing')
24.                  ->once()
25.  ;
```

Ces méthodes effacent la mémoire du contrôleur, il est donc possible d'écrire l'assertion suivante comme si le bouchon n'avait jamais été utilisé.

Le mot-clef `assert` permet de se passer de l'appel explicite à `resetCalls()` et de plus il provoque l'effacement de la mémoire de l'ensemble des adaptateurs et des contrôleurs de bouchon définis au moment de son utilisation.

Grâce à lui, il est donc possible d'écrire le test précédent d'une façon plus simple et plus lisible, d'autant qu'il est possible de passer une chaîne de caractère à assert afin d'expliquer le rôle des assertions suivantes :

```
 1.  $this
 2.      ->assert (http://www.php.net/assert)('Foo est vide')
 3.          ->if($foo = new \mock\foo())
 4.          ->and($bar = new bar($foo))
 5.          ->and($bar->doSomething())
 6.          ->then
 7.              ->mock($foo)
 8.                  ->call('doOtherThing')
 9.                      ->once()
10.
11.      ->assert (http://www.php.net/assert)('Foo a une valeur')
12.          ->if($bar->setValue(uniqid (http://www.php.net/uniqid)()))
13.          ->then
14.              ->mock($foo)
15.                  ->call('doOtherThing')
16.                      ->once()
17.  ;
```

La chaîne de caractères sera de plus reprise dans les messages générés par atoum si l'une des assertions ne passe pas avec succès.

## 2.3. Le mode loop

Lorsqu'un développeur fait du développement piloté par les tests, il travaille de la manière suivante :

1. il commence par créer le test correspondant à ce qu'il veut développer ;
2. il exécute le test qu'il vient de créer ;
3. il écrit le code permettant au test de passer avec succès ;
4. il modifie ou complète son test et repars à l'étape 2.

Concrètement, cela signifie qu'il doit :

- créer son code dans son éditeur favori ;
- quitter son éditeur pour utiliser une console afin d'exécuter son test ;
- revenir à son éditeur pour écrire le code permettant au test de passer avec succès ;
- revenir à la console afin de relancer l'exécution de son test ;
- revenir à son éditeur afin de modifier ou compléter son test ;

Il y a donc bien un cycle qui se répétera tant que la fonctionnalité n'aura pas été développée dans son intégralité.

Cependant, ce cycle est complexe et impose de nombreux allers-retours entre plusieurs logiciels, ainsi que la saisie récurrente d'une même commande dans le terminal afin de lancer l'exécution des tests unitaires.

atoum propose le mode `loop` disponible via les arguments `-l` ou `--loop` , qui permet au développeur de ne pas avoir à relancer manuellement les tests et permet donc de fluidifier le processus de développement.

Dans ce mode, atoum commence par exécuter une première fois les tests qui lui sont demandés.

Une fois les tests terminés, si les tests ont été passés avec succès par le code, atoum se met simplement en attente :

```
$ php tests/units/classes/adapter.php -l
> atoum version DEVELOPMENT by Frédéric Hardy (/Users/fch/Atoum)
> PHP path: /usr/local/bin/php
> PHP version:
=> PHP 5.3.8 (cli) (built: Sep 21 2011 23:14:37)
=> Copyright (c) 1997-2011 The PHP Group
=> Zend Engine v2.3.0, Copyright (c) 1998-2011 Zend Technologies
=>     with Xdebug v2.1.1, Copyright (c) 2002-2011, by Derick Rethans
> mageekguy\atoum\tests\units\adapter...
[S_____________________________________________________________][1/1]
=> Test duration: 0.02 second.
=> Memory usage: 0.25 Mb.
> Total test duration: 0.02 second.
> Total test memory usage: 0.25 Mb.
> Code coverage value: 100.00%
> Running duration: 0.16 second.
Success (1 test, 0 method, 2 assertions, 0 error, 0 exception) !
Press <Enter> to reexecute, press any other key to stop...
```

Si le développeur presse une autre touche que `Enter` , atoum se terminera.

Dans le cas contraire, atoum réexécutera à nouveau les mêmes tests, sans aucune autre action de la part du développeur.

Dans le cas où le code ne passe pas les tests avec succès, c'est-à-dire si des assertions ne sont pas vérifiées ou s'il y a eu des erreurs ou des exceptions, atoum se met également en attente :

```
$ php tests/units/classes/adapter.php -l
> atoum version DEVELOPMENT by Frédéric Hardy (/Users/fch/Atoum)
> PHP path: /usr/local/bin/php
> PHP version:
=> PHP 5.3.8 (cli) (built: Sep 21 2011 23:14:37)
=> Copyright (c) 1997-2011 The PHP Group
=> Zend Engine v2.3.0, Copyright (c) 1998-2011 Zend Technologies
=>     with Xdebug v2.1.1, Copyright (c) 2002-2011, by Derick Rethans
> mageekguy\atoum\tests\units\adapter...
[F_____________________________________________________________][1/1]
=> Test duration: 0.00 second.
=> Memory usage: 0.00 Mb.
> Total test duration: 0.00 second.
> Total test memory usage: 0.00 Mb.
> Running duration: 0.17 second.
Failure (1 test, 0 method, 1 failure, 0 error, 0 exception) !
> There is 1 failure:
=> mageekguy\atoum\tests\units\adapter::test__call():
In file /Users/fch/Atoum/tests/units/classes/adapter.php on line 17, mag
-Reference
+Data
@@ -1 +1 @@
-string(13) "4ea0354cd717c"
+string(32) "19798c230d5462b3bdae194f364feffa"
Press <Enter> to reexecute, press any other key to stop...
```

Tout comme dans le cas ou tout s'est bien passé, si le développeur presse une autre touche que `Enter`, atoum se terminera.

Cependant, s'il presse la touche `Enter`, au lieu de rejouer les mêmes tests comme dans le cas où les tests ont été passés avec succès, atoum n'exécutera que les tests en échec, au lieu de les rejouer dans leur intégralité.

Le développeur pourra alors dépiler les problèmes et rejouer les tests en erreur autant de fois que nécessaire simplement en appuyant sur `Enter`.

De plus, une fois que tous les tests en échec passeront à nouveau avec succès, atoum exécutera automatiquement la totalité de la suite de tests afin de détecter les éventuelles régressions introduites par la ou les corrections effectuées par le développeur.

Bien évidemment, le mode `loop` ne prend en compte que le ou les fichiers de tests unitaires lancés (chapitre3.html#Fichiers-a-executer) par atoum.

## 2.4. Le mode debug

Parfois, un test ne passe pas et il est difficile d'en découvrir la raison.

Dans ce cas, l'une des techniques possibles pour remédier au problème est de tracer le comportement du code concerné, soit directement au cœur de la classe testée à l'aide de fonctions du type de `var_dump()` ou `print_r()`, soit au niveau du test unitaire.

Et il se trouve que atoum dispose d'un certain nombre d'outils pour faciliter la tâche du développeur dans ce dernier contexte.

Ces outils ne sont cependant actif que lorsque atoum est exécuté à l'aide de l'argument `--debug`, afin que l'exécution des tests unitaires ne soit pas perturbée par les instructions relatives au débogage hors de ce contexte.

Lorsque l'argument `--debug` est utilisé, trois méthodes peuvent être activée :

- `dump()` qui permet de connaître le contenu d'une variable ;
- `stop()` qui permet d'arrêter l'exécution d'un test ;
- `executeOnFailure()` qui permet de définir une fonction anonyme qui ne sera exécutée qu'en cas d'échec d'une assertion.

Ces trois méthodes s'intègrent parfaitement dans l'interface fluide qui caractérise atoum.

### 2.4.1. dump

La méthode `dump()` peut s'utiliser de la manière suivante :

```
1.  $this
2.      ->if($foo = new foo())
3.      ->then
4.          ->object($foo->setBar($bar = new bar()))
5.              ->isIdenticalTo($foo)
6.          ->dump($foo->getBar())
7.  ;
```

Lors de l'exécution du test, le retour de la méthode `foo::getBar()` sera affiché sur la sortie standard.

Il est également possible de passer plusieurs arguments à `dump()`, de la manière suivante :

```
1.  $this
2.      ->if($foo = new foo())
3.      ->then
4.          ->object($foo->setBar($bar = new bar()))
5.              ->isIdenticalTo($foo)
6.          ->dump($foo->getBar(), $bar)
7.  ;
```

### 2.4.2. stop

L'utilisation de la méthode `stop()` est également très simple :

```
1.  $this
2.      ->if($foo = new foo())
3.      ->then
4.          ->object($foo->setBar($bar = new bar()))
5.              ->isIdenticalTo($foo)
6.          ->stop() // le test s'arrêtera ici si --debug est utilisé
7.          ->object($foo->getBar())
8.              ->isIdenticalTo($bar)
9.  ;
```

### 2.4.3. executeOnFailure

La méthode `executeOnFailure()` est très puissante et tout aussi simple à utiliser.

Elle prend en effet en argument une fonction anonyme qui sera exécutée si et seulement si l'une des assertions composant le test n'est pas vérifiée. Elle s'utilise de la manière suivante :

```
 1.  $this
 2.      ->if($foo = new foo())
 3.      ->executeOnFailure(
 4.          function() use ($foo) {
 5.              var_dump (http://www.php.net/var_dump)($foo);
 6.          }
 7.      )
 8.      ->then
 9.          ->object($foo->setBar($bar = new bar()))
10.              ->isIdenticalTo($foo)
11.          ->object($foo->getBar())
12.              ->isIdenticalTo($bar)
13.  ;
```

Dans l'exemple précédent, contrairement à `dump()` qui provoque systématiquement l'affichage sur la sortie standard le contenu des variables qui lui sont passées en argument, la fonction anonyme passée en argument ne provoquera l'affichage du contenu de la variable `foo` que si l'une des assertions suivantes est en échec.

Bien évidemment, il est possible de faire appel plusieurs fois à `executeOnFailure()` dans une même méthode de test pour définir plusieurs fonctions anonymes différentes devant être exécutées en cas d'échec du test.

## 2.5. Les méthodes d'initialisation

Lorsqu'il exécute les méthodes de test d'une classe, atoum suit le processus suivant :

1. il exécute la méthode `setUp()` de la classe de test ;
2. il lance un sous-processus PHP pour exécuter chaque méthode de test ;
3. dans le sous-processus PHP, avant d'exécuter la méthode de test, il exécute la méthode `beforeTestMethod()` de la classe de test ;
4. dans le sous-processus PHP, il exécute la méthode de test ;
5. dans le sous-processus PHP, il exécute la méthode `afterTestMethod()` de la classe de test ;
6. une fois le sous-processus PHP terminé, il exécute la méthode `tearDown()` de la classe de test.

Les méthodes `setUp()` et `tearDown()` permettent donc respectivement d'initialiser et de nettoyer l'environnement de test pour l'ensemble des méthodes de test de la classe exécutée, à la différence des méthodes `beforeTestMethod()` et `afterTestMethod()`.

Ces deux méthodes permettent en effet respectivement d'initialiser et de nettoyer l'environnement d'exécution des tests individuellement pour chacune des méthodes de test de la classe, puisqu'elles sont exécutées dans le même sous-processus, au contraire de `setUp()` et `tearDown()`.

C'est d'ailleurs la raison pour laquelle les méthodes `beforeTestMethod()` et `afterTestMethod()` acceptent comme argument le nom de la méthode de test exécutée, afin de pouvoir ajuster les traitements en conséquence.

```
1. <?php
2. namespace vendor\project\tests\units;
3.
4. use
5.     mageekguy\atoum,
```

```
 6.      vendor\project
 7.  ;
 8.
 9.  require __DIR__ . '/mageekguy.atoum.phar';
10.
11.  class bankAccount extends atoum
12.  {
13.      public function setUp()
14.      {
15.          // Exécutée *avant l'ensemble* des méthodes de test.
16.          // Initialisation globale.
17.      }
18.
19.      public function beforeTestMethod($method)
20.      {
21.          // Exécutée *avant chaque* méthode de test.
22.
23.          switch ($method)
24.          {
25.              case 'testGetOwner':
26.                  // Initialisation pour testGetOwner().
27.              break;
28.
29.              case 'testGetOperations':
30.                  // Initialisation pour testGetOperations().
31.              break;
32.          }
33.      }
34.
35.      public function testGetOwner()
36.      {
37.          ...
38.      }
39.
40.      public function testGetOperations()
41.      {
42.          ...
43.      }
44.
45.      public function afterTestMethod($method)
46.      {
47.          // Exécutée *après chaque* méthode de test.
48.
49.          switch ($method)
50.          {
51.              case 'testGetOwner':
52.                  // Nettoyage pour testGetOwner().
53.              break;
54.
55.              case 'testGetOperations':
56.                  // Nettoyage pour testGetOperations().
57.              break;
58.          }
59.      }
60.
61.      public function tearDown()
62.      {
```

```
63.          // Exécutée après l'ensemble des méthodes de test.
64.          // Nettoyage global.
65.      }
66.  }
```

Par défaut, les méthodes `setUp()`, `beforeTestMethod()`, `afterTestMethod()` et `tearDown()` ne font absolument rien.

Il est donc de la responsabilité du programmeur de les surcharger lorsque c'est nécessaire dans les classes de test concerné.

## 2.6. Fournisseurs de données (data provider)

Pour vous aider à tester efficacement vos classes, atoum met à votre disposition des fournisseurs de données (data provider en anglais).

Un fournisseur de données est une méthode d'une classe de test chargée de générer des arguments pour une méthode de test, arguments qui seront utilisés par ladite méthode pour valider des assertions.

La définition du fournisseur de données qui doit être utilisé par une méthode de test se fait grâce à l'annotation `@dataProvider` appliquée à la méthode de test concernée, de la manière suivante :

```
 1.  class calculator extends atoum
 2.  {
 3.      /**
 4.       * @dataProvider sumDataProvider
 5.       */
 6.      // Veillez à définir le bon nombre d'arguments
 7.      public function testSum($a, $b)
 8.      {
 9.          $this
10.              ->if($calculator = new project\calculator())
11.              ->then
12.                  ->integer($calculator->sum($a, $b))->isEqualTo($a + $
13.          ;
14.      }
15.
16.      ...
17.  }
```

Évidemment, il ne faut pas oublier de définir, au niveau de la méthode de test, les arguments correspondant à ceux qui seront retournés par le fournisseur de données. Si ce n'est pas le cas, atoum générera une erreur lors de l'exécution des tests.

Une fois l'annotation définie, il n'y a plus qu'à créer la méthode correspondante :

```
 1.  class calculator extends atoum
 2.  {
 3.      ...
 4.
 5.      // Fournisseur de données de testSum().
 6.      public function sumDataProvider()
 7.      {
 8.          return array (http://www.php.net/array)(
 9.              array (http://www.php.net/array)( 1, 1),
10.              array (http://www.php.net/array)( 1, 2),
11.              array (http://www.php.net/array)(-1, 1),
12.              array (http://www.php.net/array)(-1, 2),
13.          );
14.      }
15.  }
```

Lors de l'exécution des tests, atoum appellera la méthode de test `testSum()` successivement avec les arguments `(1, 1)`, `(1, 2)`, `(-1, 1)` et `(-1, 2)` renvoyés par la méthode `sumDataProvider()`.

> *L'isolation des tests ne sera pas utilisée dans ce contexte, ce qui veut dire que chacun des appels successifs à la méthode* `testSum()` *sera réalisé dans le même processus PHP.*
>
> *Un fournisseur de données peut au choix retourner un tableau ou bien un itérateur.*

## 2.7. Les bouchons (mock)

atoum dispose d'un système de bouchonnage (mock en anglais) puissant et simple à mettre en œuvre qui vous permettra de générer des mocks à partir de classes (existantes ou inexistantes) ou d'interfaces, mais également à partir de classe abstraite. Grâce à ces bouchons, vous pourrez simuler des comportements en redéfinissant les méthodes publiques de vos classes.

### 2.7.1. Générer un bouchon

Il y a plusieurs manières de créer un bouchon à partir d'une interface ou d'une classe (abstraite ou non).

La plus simple est de créer un objet dont le nom absolu est préfixé par `\mock` :

```
 1.  // création d'un bouchon de l'interface \Countable
 2.  $countableMock = new \mock\Countable;
 3.
 4.  // création d'un bouchon de la classe abstraite
 5.  // \Vendor\Project\AbstractClass
 6.  $vendorAppMock = new \mock\Vendor\Project\AbstractClass;
 7.
 8.  // création d'un bouchon de la classe \StdClass
 9.  $stdObject     = new \mock\StdClass;
10.
11.  // création d'un bouchon à partir d'une classe inexistante
12.  $anonymousMock = new \mock\My\Unknown\Class;
```

### 2.7.2. Le générateur de bouchon

atoum s'appuie sur un composant spécialisé pour générer les bouchons : le `mockGenerator`. Vous avez accès à ce dernier dans vos tests afin de modifier la procédure de génération des mocks.

Par défaut, les bouchons seront générés dans le namespace `mock` et se comporteront exactement de la même manière que les instances de la classe originale (le bouchon hérite directement de la classe originale).

#### 2.7.2.1. Changer le nom de la classe

Si vous désirez changer le nom de la classe ou son espace de nom, vous devez utiliser le `mockGenerator`.

Sa méthode `generate` prend 3 paramètres :

- le nom de l'interface ou de la classe à bouchonner ;
- le nouvel espace de nom, optionnel ;
- le nouveau nom de la classe, optionnel.

```
1.  // création d'un bouchon de l'interface \Countable vers \MyMock\Count
2.  // on ne change que l'espace de nom
3.  $this->mockGenerator->generate('\Countable', '\MyMock');
4.  $countableMock = new \myMock\Countable;
5.
6.  // création d'un bouchon de la classe abstraite
7.  // \Vendor\Project\AbstractClass vers \MyMock\AClass
8.  // on change l'espace de nom et le nom de la classe
9.  $this->mockGenerator->generate('\Vendor\Project\AbstractClass', '\MyM
10. $vendorAppMock = new \myMock\AClass;
11.
12. // création d'un bouchon de la classe \StdClass vers \mock\OneClass
13. // on ne change que le nom de la classe
14. $this->mockGenerator->generate('\StdClass', null, 'OneClass');
15. $stdObject     = new \mock\OneClass;
```

*Si vous n'utilisez que le premier argument et ne changez ni l'espace de nommage ni le nom de la classe, alors la première solution est équivalente.*

```
1.  $countableMock = new \mock\Countable;
2.
3.  // est équivalent à:
4.
5.  $this->mockGenerator->generate('\Countable');   // inutile
6.  $countableMock = new \mock\Countable;
```

#### 2.7.2.2. Shunter les appels aux méthodes parentes

Un bouchon hérite directement de la classe à partir de laquelle il a été généré, ses méthodes se comportent donc exactement de la même manière.

Dans certains cas, il peut être utile de shunter les appels aux méthodes parents afin que leur code ne soit plus exécuté. Le `mockGenerator` met à votre disposition plusieurs méthodes pour y parvenir :

```
1.  $this->mockGenerator->shuntParentClassCalls();
2.  // le bouchon ne fera pas appel à la classe parente
3.  $countableMock = new \mock\OneClass;
4.  $this->mockGenerator->unshuntParentClassCalls();
```

Ici, toutes les méthodes du bouchon se comporteront comme si elles n'avaient pas d'implémentation par contre elles conserveront la signature des méthodes originales. Vous pouvez également préciser les méthodes que vous souhaitez shunter :

```
1.  $this->mockGenerator->shunt('firstMethod');
2.  $this->mockGenerator->shunt('secondMethod');
3.  // le bouchon ne fera pas appel à la classe parente pour les méthodes
4.  $countableMock = new \mock\OneClass;
```

#### 2.7.2.3. Rendre une méthode orpheline

Il peut parfois être intéressant de rendre une méthode orpheline, c'est-à-dire, lui donner une signature et une implémentation vide. Cela peut être particulièrement utile pour générer des bouchons sans avoir à instancier toutes leurs dépendances.

```
1.  class FirstClass {
2.      protected $dep;
3.
4.      public function __construct(SecondClass $dep) {
5.          $this->dep = $dep;
6.      }
7.  }
8.
9.  class SecondClass {
10.     protected $deps;
11.
12.     public function __construct(ThirdClass $a, FourthClass $b) {
13.         $this->deps = array (http://www.php.net/array)($a, $b);
14.     }
15. }
16.
17. $this->mockGenerator->orphanize('__construct');
18. $this->mockGenerator->shuntParentClassCalls();
19.
20. // Nous pouvons instancier le bouchon sans injecter ses dépendances
21. $mock = new \mock\SecondClass();
22.
23. $object = new FirstClass($mock);
```

### 2.7.3. Modifier le comportement d'un bouchon

Une fois le bouchon créé et instancié, il est souvent utile de pouvoir modifier le comportement de ses méthodes.

Pour cela, il faut passer par son contrôleur en utilisant l'une des méthodes suivantes :

```php
$databaseClient = new \mock\Database\Client();

$databaseClient->getMockController()->connect = function() {};
// Équivalent à
$this->calling($databaseClient)->connect = function() {};
```

Le `mockController` vous permet de redéfinir **uniquement les méthodes publiques et abstraites protégées** et met à votre disposition plusieurs méthodes :

```php
$databaseClient = new \mock\Database\Client();

// redéfinie la méthode connect : elle retournera toujours true
$this->calling($databaseClient)->connect = true;

// redéfinie la méthode select
$this->calling($databaseClient)->select = function() {
    return array (http://www.php.net/array)();
};

// redéfinie la méthode query avec des arguments
$result = array (http://www.php.net/array)();
$this->calling($databaseClient)->query = function(Query $query) use($
    switch($query->type) {
        case Query::SELECT:
            return $result

        default;
            return null;
    }
};

// la méthode connect lèvera une exception
$this->calling($databaseClient)->connect->throw = new \Database\Clien
```

Comme vous pouvez le voir, il est possible d'utiliser plusieurs méthodes afin d'obtenir le comportement souhaité :

- Utiliser une valeur statique qui sera retournée par la méthode
- Utilser une implémentation courte grâce aux fonctions anonymes de PHP
- Utiliser le mot-clef `throw` pour lever une exception

Vous pouvez également spécifier plusieurs valeurs en fonction de l'ordre d'appel:

```
1.  // défaut
2.  $this->calling($databaseClient)->count (http://www.php.net/count) = r
3.  // équivalent à
4.  $this->calling($databaseClient)->count (http://www.php.net/count)[0] =
5.
6.  // 1er appel
7.  $this->calling($databaseClient)->count (http://www.php.net/count)[1] =
8.
9.  // 3ème appel
10. $this->calling($databaseClient)->count (http://www.php.net/count)[3] =
```

- Le premier appel retournera 13.
- Le second aura le comportement par défaut, c'est à dire un nombre aléatoire.
- Le troisième appel retournera 42.
- Tous les appels suivants auront le comportement par défaut, c'est à dire des nombres aléatoires.

Si vous souhaitez que plusieurs méthodes du bouchon aient le même comportement, vous pouvez utiliser les méthodes `methods` ou `methodsWhichMatch` .

#### 2.7.3.1. methods

`methods` vous permet, grâce à la fonction anonyme passée en argument, de définir pour quelles méthodes le comportement doit être modifié :

```
// si la méthode a tel ou tel nom,
// on redéfinit son comportement
$this
    ->calling($mock)
        ->methods(
            function($method) {
                return in_array (http://www.php.net/in_array)(
                    $method,
                    array (http://www.php.net/array)(
                        'getOneThing',
                        'getAnOtherThing'
                    )
                );
            }
        )
            ->return = uniqid (http://www.php.net/uniqid)()
;

// on redéfinit le comportement de toutes les méthodes
$this
    ->calling($mock)
        ->methods()
            ->return = null
;

// si la méthode commence par "get",
// on redéfinit son comportement
$this
    ->calling($mock)
        ->methods(
            function($method) {
                return substr (http://www.php.net/substr)($method, 0,
            }
        )
            ->return = uniqid (http://www.php.net/uniqid)()
;
```

Dans le cas du dernier exemple, vous devriez plutôt utiliser `methodsWhichMatch` .

> *La syntaxe utilise les fonctions anonymes (aussi appelées fermetures ou closures) introduites en PHP 5.3. Reportez-vous au manuel de PHP (http://php.net/functions.anonymous) pour avoir plus d'informations sur le sujet.*

#### 2.7.3.2. methodsWhichMatch

`methodsWhichMatch` vous permet de définir les méthodes où le comportement doit être modifié grâce à l'expression rationnelle passée en argument :

```
// si la méthode commence par "is",
// on redéfinit son comportement
$this
    ->calling($mock)
        ->methodsWhichMatch('/^is/')
            ->return = true
;

// si la méthode commence par "get" (insensible à la casse),
// on redéfinit son comportement
$this
    ->calling($mock)
        ->methodsWhichMatch('/^get/i')
            ->throw = new \exception
;
```

`methodsWhichMatch` *utilise* `preg_match` *(http://php.net/preg_match) et les expressions rationnelles. Reportez-vous au manuel de PHP (http://php.net/pcre) pour avoir plus d'informations sur le sujet.*

### 2.7.4. Cas particulier du constructeur

Pour bouchonner le constructeur d'une classe, il faut :

- créer une instance de la classe \atoum\mock\controller avant d'appeler le constructeur du bouchon ;
- définir via ce contrôleur le comportement du constructeur du bouchon à l'aide d'une fonction anonyme ;
- injecter le contrôleur lors de l'instanciation du bouchon.

```
$controller = new \atoum\mock\controller();
$controller->__construct = function() {};

$databaseClient = new \mock\Database\Client($controller);
```

*Dans l'exemple ci-dessus, le constructeur de la classe* `\Database\Client` *n'a aucun argument, nous injectons directement le contrôleur. Dans le cas d'un constructeur ayant des paramètres, il faudra passer le contrôleur en dernier argument, après tous les arguments requis et optionnels.*

### 2.7.5. Tester un bouchon

atoum vous permet de vérifier qu'un bouchon a été utilisé correctement.

```
 1.  $databaseClient = new \mock\Database\Client();
 2.  $databaseClient->getMockController()->connect = function() {};
 3.  $databaseClient->getMockController()->query   = array (http://www.php
 4.
 5.  $bankAccount = new \Vendor\Project\Bank\Account();
 6.  $this
 7.      // utilisation du bouchon via un autre objet
 8.      ->array (http://www.php.net/array)($bankAccount->getOperation($da
 9.          ->isEmpty()
10.
11.      // test du bouchon
12.      ->mock($databaseClient)
13.          ->call('query')
14.              ->once()        // vérifie que la méthode query
15.                              // n'a été appelé qu'une seule fois
16.  ;
```

*Reportez-vous à la documentation sur l'assertion* `mock` *pour obtenir plus d'informations sur les tests des bouchons.*

❮ Introduction (chapitre1.html)  Lancement des tests (chapitre3.html) ❯

Something is wrong in the documentation ? Help us fix this by reporting errors and problems ! (https://github.com/atoum/atoum-documentation/issues)