UNDERSCORE.PHP

Underscore.php 

The PHP manipulation toolbet

First off : Underscore.php is **not** a PHP port of Underscore.js (well ok I mean it was at first). It's doesn't aim to blatantly port its methods, but more port its philosophy.

It's a full-on PHP manipulation toolbet sugar-coated by an elegant syntax directly inspired by the Laravel framework. Out through the window went the infamous `__()`, replaced by methods and class names that are meant to be read like sentences *à la* Rails : `Arrays::from($article)->sortBy('author')->toJSON()`.

It features a good hundred of methods for all kinds of types : strings, objects, arrays, functions, integers, etc., and provides a parsing class that help switching from one type to the other mid-course. Oh also it's growing all the time. The cherry on top ? It wraps nicely around native PHP functions meaning `String::replace` is actually a dynamic call to `str_replace` but with the benefit of allowed chaining and a **finally** consistant argument order (**all** functions in *Underscore* put the subject as the first argument, NO MATTER WHAT).

It works both as a stand-alone via *Composer* or as a bundle for the Laravel framework. So you know, you don't really have any excuse.

Install Underscore

To install Underscore.php you can either add it via Composer :

```
"anahkiasen/underscore-php" : "dev-master"
```

Or if you're using the Laravel framework, via the Artisan CLI :

```
artisan bundle:install underscore
```

And then adding the following to your bundles files :

```
'underscore' => array('auto' => true),
```

Note that Underscore's type classes (Arrays/String/etc) are by default namespaced in the `Types` folder, so to use Arrays, you would do the following :

```
use Underscore\Types\Arrays;
```

Using Underscore

It can be used both as a static class, and an Object-Oriented class, so both the following are valid :

```
$array = array(1, 2, 3);

// One-off calls to helpers
Arrays::each($array, function($value) { return $value * $value; }) // Square root the array
Function::once($myFunction) // Only allow the function to be called once
Number::paddingLeft(5, 5) // Returns '00005'
Object::methods($myObject) // Return the object's methods
String::length('foobar') // Returns 6

// Or chain calls with the 'from' method
Arrays::from($array)->filter(...)->sort(...)->get(2)

// Which does this in the background
$array = new Arrays($array);
$array->filter(...)->sort(...)->get(2)
```

For those nostalgic of ye old `__()` a generic `Underscore` class is provided that is able to go and fetch methods in all of Underscore.php's methods. For this it looks into the methods it knows and analyzes the subject of the method (meaning if you do `Underscore::contains('foobar', 'foo')` it knows you're not looking for `Arrays::contains`).

On types : it's important to note that using a specific type class to create an Underscore repository will convert the type of provided subject. Say you have an object and do `new Arrays($myObject)` – this will convert the object to an array and allow you to use Arrays methods on it. For this Underscore uses its **Parse** class's methods that for the most part just type cast and return (like this `(array) $object`) but it also sometimes go the extra mile to understand what you want to do : if you convert an array to a string, it will transform it to JSON, it you transform an array into an integer, it returns the size of the array, etc.

The core concept is this : static calls return values from their methods, while chained calls update the value of the object they're working on. Which means that an Underscore object don't return its value until you call the `->obtain` method on it — until then you can

chain as long as you want, it will remain an object. The exception are certains properties that are considered *breakers* and that will return the object's value. An example is `Arrays->get`.

Note that since all data passed to Underscore is transformed into an object, you can do this sort of things, plus the power of chained methods, it all makes the manipulation of data a breeze.

```
$array = new Arrays(['foo' => 'bar']);

echo $array->foo // Returns 'bar'

$array->bis = 'ter'
$array->obtain() // Returns array('foo' => 'bar', 'bis' => 'ter')
```

Customizing Underscore

Underscore.php provides the ability to extend any class with custom functions so go crazy. Don't forget that if you think you have a function anybody could enjoy, do a pull request, let everyone enjoy it !

```
String::extend('summary', function($string) {
  return String::limitWords($string, 10, '... — click to read more');
});

String::from($article->content)->summary()->title()
```

You can also give custom aliases to all of Underscore's methods, in the provided config file. Just add entries to the `aliases` option, the key being the alias, and the value being the method to point to.

Extendability

Underscore.php's classes are extendable as well in an OOP sense. You can update an Underscore repository with the `setSubject` method (or directly via `$this->subject =` granted you return `$this` at the end). When creating an Underscore repository, by default it's subject is an empty string, you can change that by returning whatever you want in the `getDefault` method.

```
class Users extends Arrays
{
  public function getDefault()
  {
```

```
    return 'foobar';
  }

  public function getUsers()
  {
    // Fetch data from anywhere

    return $this->setSubject($myArrayOfUsers);
  }
}

$users = new Users; // Users holds 'foobar'
$users->getUsers()->sort('name')->clean()->toCSV()

// Same as above
Users::create()->getUsers()->sort('name')->clean()->toCSV()
```

It is important to not panic about the mass of methods present in Underscore and the dangers extending one of the Types would cause : the methods aren't contained in the classes themselves but in methods repositories. So if you extend the `String` class and want to have a `length` method on your class that has a completely different meaning than `String::length`, it won't cause any signature conflict or anything.

Also note that Underscore method router is dynamic so if your subject is an array and mid course becomes a string, Underscore will always find the right class to call, no matter what you extended in the first place. Try to keep track of your subject though : if your subject becomes a string, calling per example `->map` will return an error.

Call to native methods

Underscore natively extends PHP, so it can automatically reference original PHP functions when the context matches. Now, PHP by itself doesn't have a lot of conventions so `Arrays::` look for `array_` functions, `String::` look for `str_` plus a handful of other hardcoded redirect, but that's it. The advantage is obviously that it allows chaining on a lot of otherwise one-off functions or that only work by reference.

```
Arrays::diff($array, $array2, $array3) // Calls `array_diff`
Arrays::from($array)->diff($array2, $array3)->merge($array4) // Calls `array_diff` then `array_merge` on the result
```

Documentation

You can find a detailed summary of all classes and methods in the repo's wiki or the official page. The changelog is available in the CHANGELOG file.

About Underscore.php

There is technically another port of Underscore.js to PHP available on Github — I first discovered it when I saw it was for a time used on Laravel 4. I quickly grew disapoint of what a mess the code was, the lack of updates, and the 1:1 mentality that went behind it.

This revamped Underscore.php doesn't aim to be a direct port of Underscore.js. It sometimes omits methods that aren't relevant to PHP developers, rename others to match terms that are more common to them, provides a richer syntax, adds a whole lot of methods, and leaves room for future ones to be added all the time — whereas the previous port quickly recoded all JS methods to PHP and left it at that.

If you come from Javascript and are confused by some of the changes, don't put all the blame on me for trying to mess everything up. A basic example is the `map` function : in PHP it has a completely different sense because there exists an `array_map` function that basically does what `__::invoke` does in JS. So `map` is now `Arrays::each`. Always keep in mind this was made for *PHP* developpers first, and differences **do** exist between the two to accomdate the common terms in PHP.

Arrays

Available methods

Informations about an array

average / contains / has / matches / matchesAny / max / min / size / sum /

Array slicers

diff / first / initial / last / rest /

Get from an array

clean / find / get / pluck / random / without /

Generate arrays

range / repeat /

Act upon an array

at /

Alter an array

append / each / filter / flatten / group / implode / invoke / merge / prepend / reject / remove / removeFirst / removeLast / replaceKeys / replaceValue / set / sort / sortKeys /

Methods yet to implement

exclude
intersection
unique

Informations about an array

Arrays::average(array)

Computes the average value of an array

```
Arrays::average(array(1, 2, 3)) // Returns 2
```

Arrays::contains(array, contains)

Check if an item is in an array

```
Arrays::contains(array(1, 2, 3), 2) // Returns true
```

Arrays::has(array, key)

Check if a key exists in an array

```
Arrays::has(array('foo' => 'bar'), 'foo') // Returns true
```

Arrays::matches(array, closure)

Check if all items in an array match a truth test

```
Arrays::matches(array(1, 2, 3), function($value) {
    return $value % 2 == 0; // Returns false
});
```

Arrays::matchesAny(array, closure)

Same than above but returns true if at least one item matches

```
Arrays::matchesAny(array(1, 2, 3), function($value) {
    return $value % 2 == 0; // Returns true
});
```

Arrays::max(array, [closure])

Get the maximum value from an array. A closure can be passed to evaluate the values a certain way

```
Arrays::max(array(1, 2, 3)) // Returns 3
Arrays::max(array(1, 2, 3), function($value) {
  return $value * -1;
});
```

Arrays::min(array, [closure])

Similar to Arrays::max but for the lowest value

Arrays::size(array)

Get the size of an array

```
Arrays::size(array(1, 2, 3)) // Returns 3
```

Arrays::sum(array)

Computes the sum of an array

```
Arrays::sum(array(1, 2, 3)) // Returns 6
```

Array slicers

Arrays::first(array, [take])

Get the first value from an array. You can also specify a number of elements to return

```
Arrays::first(array(1, 2, 3)) // Returns 1
Arrays::first(array(1, 2, 3), 2) // Returns array(1, 2)
```

Arrays::last(array)

alias : getLast

Get the last value from an array

```
Arrays::last(array(1, 2, 3)) // Returns 3
```

Arrays::initial(array, [ignore])

Exclude the last X elements from an array

```
Arrays::initial(array(1, 2, 3), 1) // Returns array(1, 2)
```

Arrays::rest(array, [from])

Exclude the first X elements from an array

```
Arrays::rest(array(1, 2, 3), 2) // Returns 3
```

Arrays::diff(array*)

Computes the difference between multiple arrays

```
Arrays::diff(array(1, 2, 3), array(1, 5)) // Returns array(2, 3)
```

Get from an array

Arrays::find(array, closure)

alias : select

Find the first value in an array that passes a truth test

```
Arrays::find(array(1, 2, 3), function($value) {
    return $value % 2 == 0; // Returns 2
});
```

Arrays::get(array, key)

Get a value from an array using dot-notation

```
$array = underscore(array('foo' => array('bar' => 'ter')));
$array->get('foo.bar') // Return 'ter'
```

Arrays::pluck(array, column)

Pluck a column from an array

```
$array = array(
    array('foo' => 'bar', 'bis' => 'ter'),
    array('foo' => 'bar', 'bis' => 'ter'),
);
Arrays::pluck($array, 'foo'); // Returns array('bar', 'bar')
```

Arrays::clean(array)

Remove all falsy values from an array

```
Arrays::clean(array(true, false, 0, 1, 'string', '')) // Returns array(true, 1, 'string')
```

Arrays::random(array)

Get a random value from an array

```
Arrays::random(array(1, 2, 3)) // Returns 1, 2 or 3
```

Arrays::without(array, values*)

Returns the array without all instances of the given values

```
$array = array('foo', 'foo', 'bar', 'bis', 'ter')
Arrays::without($array, 'foo', 'bis') // Returns array('bar', 'ter')
```

Generate arrays

Arrays::range([start], stop, [step])

Generate an array from a range

```
Arrays::range(5) // Returns array(1, 2, 3, 4, 5)
Arrays::range(-2, 2) // Returns array(-2, -1, 0, 1, 2)
Arrays::range(1, 10, 2) // Returns array(1, 3, 5, 7, 9)
```

Arrays::repeat(data, times)

Fill an array with $times times the $data

```
Arrays::repeat('foo', 3) // Returns array('foo', 'foo', 'foo')
```

Act upon an array

Arrays::at(array, closure)

Iterate over an array to execute a callback at each loop

```
$multiplier = 3;
Arrays::at(array(1, 2, 3), function($value) use ($multiplier) {
```

```
    echo $value * $multiplier; // Prints out 3, 6, 9
});
```

Alter an array

Arrays::append(array, value)

Append a value to an array

```
Arrays::append(array(1, 2, 3), 4) // Returns array(1, 2, 3, 4)
```

Arrays::each(array, closure)

Iterate over an array and apply a callback to each value

```
Arrays::each(array(1, 2, 3), function($value) {
    return $value * 3; // Return array(3, 6, 9)
});
```

Arrays::filter(array, [closure])

Find all values in an array that passes a truth test If no closure specified, simply cleans the array (see: Arrays::clean)

```
Arrays::filter(array(1, 2, 3), function($value) {
    return $value % 2 != 0; // Returns array(1, 3)
});
```

Arrays::flatten(array, [separator = .])

Flattens an array to dot notation or with $separator

```
$array = array('foo' => array('bar' => array('bis' => 'ter')))
Arrays::flatten($array) // Returns array('foo.bar.bis' => 'ter')
Arrays::flatten($array, '/') // Returns array('foo/bar/bis' => 'ter')
```

Arrays::group(array, grouper)

Group an array by the results of a closure Instead of a closure a property name can be passed to group by it

```
Arrays::group(array(1, 2, 3, 4, 5), function($value) {
  return $value % 2 == 0; // Returns array(array(1, 3, 5), array(2, 4))
```

```
})
```

Arrays::invoke(array, function)

Invoke a function on all of an array's values

```
Arrays::invoke(array('   foo'), 'trim'); // Returns array('foo')
```

Arrays::implode(array, with)

Implodes an array into a string

```
Arrays::implode(array('foo', 'bar'), ', '); // Returns "foo, bar"
```

Arrays::merge(array*)

Merge on or more arrays together

```
Arrays::merge(array(1, 2), array('foo' => 'bar')) // Returns array(1, 2, 'foo' => 'bar')
Arrays::from(array(1, 2))->merge(array('foo' => 'bar')) // Same thing
```

Arrays::reject(array, closure)

Find all values in an array that are rejected by a truth test

```
Arrays::filter(array(1, 2, 3), function($value) {
    return $value % 2 != 0; // Returns array(2)
});
```

Arrays::remove(array, key)

Remove values from arrays using dot notation

```
Arrays::remove($articles, '1.author.name') // Will unset $articles[1]['author']['name']
});
```

Arrays::removeFirst(array)

Remove the first value from an array

```
Arrays::removeFirst(array(1, 2, 3)) // Will return [2, 3]
```

Arrays::removeLast(array)

Remove the last value from an array

```
Arrays::removeLast(array(1, 2, 3)) // Will return [1, 2]
```

Arrays::replaceKeys(array, keys)

Replace the keys of an array

```
Arrays::replaceKeys(['foo' => 'foo', 'bar' => 'bar'], ['newFoo', 'newBar']) // Will return ['newFoo' => 'foo', 'newBar' => 'bar']
```

Arrays::replaceValue(array, replace, with)

Replace in the array values

```
Arrays::replaceValue(array('foo', 'foo', 'bar'), 'foo', 'bar') // Will return ['foo, 'foo', 'foo']
```

Arrays::prepend(array, value)

Prepend a value to an array

```
Arrays::prepend(array(1, 2, 3), 4) // Returns array(4, 1, 2, 3)
```

Arrays::set(array, key, value)

Set an value in an array using dot notation

```
Arrays::set(array(), 'foo.bar', 'bis') // Returns array('foo' => array('bar' => 'bis'))
```

Arrays::sort(array, [sorter], [direction])

alias : sortBy

Sort an array The second argument can be a closure returning what to sort by, or a property name The third argument is the direction (asc or desc)

```
Arrays::sort(array(5, 3, 1, 2 ,4), null, 'desc') // Returns array(5, 4, 3, 2, 1)
Arrays::sort($articles, function($article) {
  return $article->name; // Returns article sorted by name
});
Arrays::sort($articles, 'author.name', 'desc') // Returns articles sorted by author name, DESC
```

Arrays::sortKeys(array, [direction])

Sort an array by keys

```
Arrays::sortKeys(array('z' => 0, 'b' => 1, 'r' => 2)) // Returns array('b' => 1, 'r' => 2, 'z' => 0)
Arrays::sortKeys(array('z' => 0, 'b' => 1, 'r' => 2), 'desc') // Returns array('z' => 0, 'r' => 2, 'b' => 1)
```

Object

Available methods

Informations about an object

keys / methods / values /

Alter an object

group / pluck / remove / set / sort / unpack /

Informations about an object

Object::keys(object)

Get the keys from an object

Object::values(object)

Get the values from an object

Object::methods(object)

List all methods of an object

Alter an object

Object::set(array, key, value)

Set an value in an array using dot notation

```
$object = new stdClass;
Object::set($object, 'foo.bar', 'bis') // $object->foo['bar'] = 'bis'
```

Object::pluck(object, column)

Pluck a column from an array of objects

```
Object::pluck($articles, 'title'); // Returns array('title1', 'title2', ...)
```

Object::remove(array, key)

Remove attributes from objects using dot notation

```
Object::remove($articles, '1.author.name') // Will unset $articles[1]->author->name
```

Object::group(object, grouper)

Group an array by the results of a closure Instead of a closure a property name can be passed to group by it

```
Object::group($articles, function($value) {
  return $articles->created_at->format('Y'); // Returns articles sorted by creation year
})
```

Object::sort(object, [sorter], [direction])

Sort an array The second argument can be a closure returning what to sort by, or a property name The third argument is the direction (asc or desc)

```
Object::sort($articles, function($article) {
  return $article->name; // Returns article sorted by name
});
Object::sort($articles, 'author.name', 'desc') // Returns articles sorted by author name, DESC
```

Object::unpack(object)

Unpack an object's attribute (similar to doing object = object->something) but without having to know which key to fetch

```
$object = (object) array('attributes' => array('name' => 'foo', 'age' => 18))
$attributes = Object::unpack($object)
```

Number

Available methods

Alter a number

padding / paddingLeft / paddingRight /

Alter a number

Number::padding(number, padding, direction)

Add 0 padding to a number

```
Number::padding(5, 5, STR_PAD_BOTH) // Returns '00500'
```

Number::paddingLeft(number, padding)

Add 0 padding to the left of a number

```
Number::paddingLeft(5, 5) // Returns '00005'
```

Number::paddingRight(number, padding)

Add 0 padding to the right of a number

```
Number::paddingRight(5, 5) // Returns '50000'
```

String

Available methods

> Note : A lot of those methods are directly taken from Laravel 3's String class (including the incredible mechanics behind the singular/plural methods). Credit goes to Taylor Otwell for them

Generate a string

accord / random /

Informations about a string

endsWith / length / startsWith /

Fetch from a string

find / slice / sliceFrom / sliceTo /

Alter a string

explode / limit / lower / plural / remove / repeat / singular / title / toggle / upper / words /

Case switchers

toPascalCase / toSnakeCase / toCamelCase /

Generate a string

String::accord(integer, many, one, [zero])

Creates a string from a number. You can provide a `%d` placeholder to insert the actual count into the final string.

```
String(15, '%d articles', '%d article', 'no articles') // Returns '15 articles'
String(1, '%d articles', '%d article', 'no articles') // Returns '1 article'
String(0, 'many articles', 'one article', 'no articles') // Returns 'no articles'
```

String::random(length)

Generates a random string

```
String::random(10) // Returns 'anFe48cfRc'
```

Informations about a string

String::endsWith(string, with)

Check if a string ends with another string

```
String::endsWith('foobar', 'bar') // Returns true
```

String::length(string)

Returns the length of a string

```
String::length('foobar') // Returns 6
```

String::startsWith(string, with)

Check if a string starts with another string

```
String::startsWith('foobar', 'foo') // Returns true
```

Fetch from a string

String::find(string, find, [caseSensitive], [absolute])

Find one or more needles in one or more haystacks. The third argument makes the search case sensitive, and the fourth argument if true ensures that all searchs need to match to return a true result

```
String::find('foobar', 'oob') // Returns true
String::find('foobar', array('foo', 'bar')) // Returns true
String::find(array('foo', 'bar'), 'foo') // Returns true

String::find('FOOBAR', 'foobar', false) // Returns false

String::find('foofoo', array('foo', 'bar'), false, true) // Returns false
```

String::slice(string, slice)

Slice a string with another string

```
String::slice('foobar', 'ba') // Returns array('foo', 'bar')
```

String::sliceFrom(string, slice)

Slice a string from a particular point

```
String::sliceFrom('foobar', 'ob') // Returns 'obar'
```

String::sliceTo(string, slice)

Slice a string up to a particular point

```
String::sliceTo('foobar', 'ob') // Returns 'fo'
```

Alter a string

String::explode(string, with, [limit])

Explodes a string by $with

```
String::explode("Joey doesn't share food", ' ') // Returns array('Joey', "doesn't", 'share', 'food')
```

String::limit(string, limit, [end])

Limits a string to X characters and append something

```
String::limit('This is somehow long', 10, '...') // Returns 'This is so...'
```

String::lower(string)

Converts a string to lowercase

```
String::lower('UNDerScoRe') // Returns 'underscore'
String::lower('ΧΙΣΤΗ') // Returns 'χιστη'
```

String::plural(string)

Converts a string to plural

```
String::plural('child') // Returns 'children'
```

String::remove(string, remove)

Remove one or more parts from a string

```
String::remove('foo', 'oo') // Return 'f'
String::remove('foo bar bis', array('bar', 'foo')) // Returns 'bis'
```

String::repeat(string, times)

Repeat a string X times

```
String::repeat('foo', 2) // Returns 'foofoo'
```

String::singular(string)

Converts a string to singular

```
String::singular('dogs') // Returns 'dog'
```

String::title(string)

Converts a string to title case

```
String::title('my little boat') // Returns 'My Little Boat'
```

String::toggle(string, first, second, [loose = false])

Toggles a string between two states. Contains a $loose flag on the last argument to allow the switching of a string that matches neither states

```
String::toggle('foo', 'foo', 'bar') // Returns 'bar'
String::toggle('bis', 'foo', 'bar', true) // Returns 'foo'
```

String::upper(string)

Converts a string to uppercase

```
String::upper('underscore') // Returns 'UNDERSCORE'
String::upper(' χιστη') // Returns ' ΧΙΣΤΗ'
```

String::words(string, limit, [end])

Limits a string to X words and append something

```
String::words('This is somehow long', 3, '...') // Returns 'This is somehow...'
```

Case switchers

String::toPascalCase(string, [limit])

Converts a string to PascalCase, or only the first $limit parts

```
String::toPascalCase('my_super_class') // Returns 'MySuperClass'
String::toPascalCase('my_super_class', 2) // Returns 'MySuper_class'
```

String::toSnakeCase(string, [limit])

Converts a string to snake_case, or only the first $limit parts

```
String::toSnakeCase('MySuperClass') // Returns 'my_super_class'
String::toSnakeCase('MySuperClass', 2) // Returns 'my_superClass'
```

String::toCamelCase(string, [limit])

Converts a string to CamelCase, or only the first $limit parts

```
String::toCamelCase('my_super_class') // Returns 'mySuperClass'
String::toCamelCase('my_super_class', 2) // Returns 'mySuper_class'
```

Functions

Available methods

Function limiters

after / cache / once / only / throttle /

Function limiters

Functions::after(function, $times)

Allow a function to be called only after $times times

```
$number = 0;
$increment = Functions::after(function(&$number) {
  $number++;
}, 2);

$increment(); // $number = 0
$increment(); // $number = 0
$increment(); // $number = 1
$increment(); // $number = 2
```

Functions::cache(function)

Cache the result of a function so that it's only computed once (different arguments are taken into account)

```
$compute = Functions::cache(function() {
  return microtime();
});

$compute(); // return 0.25139300 1138197510
$compute(); // return 0.25139300 1138197510
```

Functions::once(function)

Allow a function to be called only once

```
$number = 0;
```

```
$increment = Functions::once(function(&$number) {
  $number++;
});

$increment(); // $number = 1
$increment(); // $number = 1
```

Functions::only(function, times)

Allow a function to be called only $times times

```
$number = 0;
$increment = Functions::only(function(&$number) {
  $number++;
}, 2);

$increment(); // $number = 1
$increment(); // $number = 2
$increment(); // $number = 2
```

Functions::throttle(function, seconds)

Allow a function to be called only every $seconds seconds

```
$number = 0;
$increment = Functions::only(function(&$number) {
  $number++;
}, 1);

$increment(); // $number = 1
$increment(); // $number = 1
sleep(1)
$increment(); // $number = 2
```

Parse

The Parse class is a general helper from converting from and to various formats. You can either call it directly on content like this `Parse::toJSON($array)` or chain it from existing **Underscore** objects, like this : `Arrays::from($data)->toJSON()`

Available methods

Parse from a format

fromJSON / fromCSV / fromXML /

Parse to a format

toJSON / toCSV /

Type switchers

toArray / toBoolean / toInteger / toObject / toString /

Parse from a format

Parse::fromJSON(data)

Converts data from JSON

```
Parse::fromJSON('{"foo":"bar","bis":"ter"}') // Returns ['foo' => 'bar', 'bis' => 'ter']
```

Parse::fromCSV(data)

Converts data from CSV

```
Parse::fromCSV('foo;bar;bis;ter') // Returns ['foo', 'bar', 'bis', 'ter']
```

Parse::fromXML(data)

Converts data from XML

```
Parse::fromXML('<article><name>Foo</name><content>Bar</content></article>') // Returns ['article' => ['name' => 'F
oo', 'content' => 'Bar']]
```

Parse to a format

Parse::toJSON(data)

Converts data to JSON

```
Parse::toJSON(array(1, 2, 3)) // Returns [1, 2, 3]
```

Parse::toCSV(data, [delimiter = ;], [exportHeaders = true])

Converts data to CSV. You can specify which delimiter is to be used for the rows, and whether an **headers** row should be created

```
$array = array(array('name' => 'foo', 'content' => 'bar'), array('name' => 'bar', 'content' => 'foo'))
Parse::toCSV($array, ';', true) // Returns "name";"content"\n"foo";"bar"\n"bar";"foo"
```

Type switchers

Parse::toArray(data)

Converts data to an array

```
Parse::toArray($article) // Returns array('title' => 'My article', 'content' => ...)
Parse::toArray(15) // Returns array(15)
```

Parse::toBoolean(data)

Converts data to a boolean

```
Parse::toBoolean(15) // Returns true
Parse::toBoolean('foo') // Returns true
Parse::toBoolean(array()) // Returns false
Parse::toBoolean('') // Returns false
Parse::toBoolean(0) // Returns false
```

Parse::toInteger(data)

Converts data to a boolean. Arrays will return their size. String will return their length, excepted if the string contains a number.

```
Parse::toInteger('15') // Returns 15
Parse::toInteger('foo') // Returns 3
Parse::toInteger(array()) // Returns 0
Parse::toInteger(array(1, 2, 3)) // Returns 3
Parse::toInteger('') // Returns 0
```

Parse::toObject(data)

Converts data to an object

Parse::toString(data)

Converts data to a string. Arrays and objects will be converted to JSON.

```
Parse::toString(15) // Returns '15'
Parse::toString(array(1, 2, 3)) // Returns "[1, 2, 3]"
```

Repository
Repositories

Repository is an abstract class extended by all type classes. It's the core of **Underscore.php** and is what allows you to both chain and do static calls of the methods, as well as switching from types to types.

The `Arrays` class wil be used in the following examples but you can replace it with any of the type classes.

Creating a repository

Repositories can be created the following ways :

From nothing

With `Arrays::create()` or `new Arrays`. This will create an empty repository, containing a subject matching the type you called (so `new Arrays` has `array()` as subject).

From existing data

With `Arrays::from($data)` or `new Arrays($data)` – note this will convert the existing data to make it match the type of the class, so objects passed to `Arrays::from` will become arrays.

Special properties

Repositories have special properties that allow you to manipulate their subject. This make working with both arrays and objects smoother by allowing you to use `$repository->property` and `$repository->property = 'something'`, whether the subject is an array or an object.

You can replace the subject of a repository at any time by doing `$repository->setSubject($newSubject)`.

You can also check if a repository's subject is empty by doing `$repository->isEmpty`.

Finally to get the subject of a repository, do `$repository->obtain()`.

Repositories will also use Underscore's `Parse::toString` method when the `__toString()` method is being called on them. Meaning the following :

```
echo Arrays::from(['foo' => 'bar']) // Prints out {"foo":"bar"}
```

Type conversions

Although you can create a repository from any type class, you're not limited to the methods of that class. Underscore will automatically check the type of the subject and call the right class, so if per example you do `Arrays::from()->size()` and thus have an integer as subject, you can then call the `Number` methods on it.
