Delegates

The concept of a delegate is one of the key concepts of framework ClickBlocks. The delegate is a further development of **callable** objects php (see more details here), providing more opportunities for the provision of references to functions or class methods.

In general, the delegate can be represented as a string representation of calling of some function or class method. Delegate is also an object of class **ClickBlocks\Core\Delegate**. Objects of this class are **callable**. This means you can treat them as functions. Thus, in reality, a function or class method on which the delegate refers will be invoked. In other words, the delegate is kind of an alias of a function or a class method.

The class constructor takes a single argument of the delegate - a reference to the **callable** object, which can be any value. Here are some examples of non-string type delegates:

```
// Suppose we have methods of a class for which we want to set delegates.
class Test
{
  public static function a() {...}

  public function b($p1, $p2) {...}
}

// Creates the delegate of method "a".
$d1 = new Delegate(['Test', 'a']);
// Creates the delegate of method "b".
$d2 = new Delegate([new Test, 'b']);
// Creates the delegate of constructor of class "Test"
$d3 = new Delegate(new Test);
// Creates the delegate that refers to another delegate.
$d4 = new Delegate($d3);
// Creates the delegate of anonymous function.
$d5 = new Delegate(function() {...});

// Now we can invoke delegated methods.
// Calls method "a".
$d1();
// Calls method "b".
$d2(1, 2);
// Calls constructor of class "Test".
// Delegate returns an object of class "Test".
$test = $d3(); // the same as $test = $d4();
// Calls anonymous function.
$d5();
```

String delegates allow more flexibility to associate a delegate with a function or class method.

There are 7 general types of string delegates:

1. **function** - function call.
2. **class::method** - call of a class static method.
3. **class->method** - call of a non-static method of a class. The class has the constructor without parameters.
4. **class[]** - call of a class constructor without parameters.
5. **class[n]** - call of a class constructor that takes **n** parameters.
6. **class[n]->method** - call of a non-static method of a class. The class constructor takes **n** parameters.
7. **class@cid->method** - call of a non-static method of some web-control.

Here **function** - function name, **class** - class name, **method** - method name, **n** - number of parameters of a class constructor, **cid** - unique or logical identifier of a web-control.

Besides of these basic types of string delegates, there are two cases of context dependent delegates:

- **::method** - if a page class is defined (variable `ClickBlocks\MVC\Page::$page` is not empty), then such delegate is similar to calling of a static method of the page class. If the page class is not defined the appropriate method of class **CB** will be called.
- **->method** - similar to the first case, but only applied to non-static methods.

Some examples:

```
// Defines a class whose methods will be delegated.
class Test
{
  public static function a() {...}

  public function __construct($p1, $p2) {...}

  public function b($p) {...}
}

// Simple function.
function c($p1, $p2 = null) {...}

// Creates the delegate of method "a".
$d1 = new Delegate('Test::a');
// Creates the delegate of method "b".
$d2 = new Delegate('Test[2]->b');
// Creates the delegate of constructor of class "Test".
$d3 = new Delegate('Test[2]');
// Creates the delegate of function "c".
$d4 = new Delegate('c');

// Calls method "a".
$d1();
// Calls method "b".
// The first two parameters are passed to the class constructor,
// and the third parameter is passed to the method "b".
$d2('a', 'b', 'foo');
// Calls the class constructor and gets the class object.
$test = $d3('a', 'b');
// Calls the simple function.
$d4('a');
```

Sometimes it is more convenient to pass parameters to the delegate as an array. You can do this by using method **call()**. In this case calls of delegates from the example above look like this:

```
// Calls method "a".
$d1->call();
// Calls method "b".
// The first two parameters are passed to the class constructor,
// and the third parameter is passed to the method "b".
$d2->call(['a', 'b', 'foo']);
// Calls the class constructor and gets the class object.
$test = $d3->call(['a', 'b']);
// Calls the simple function.
$d4->call(['a']);
```

Each delegate object can be easily converted to its string representation:

```
// Creates a delegate.
$d = new Delegate([new Test('a', 'b'), 'foo']);
// Gets string representation of the delegate.
echo (string)$d; // displays Test[2]->foo
```

This transformation is reversible (i.e. you can use the string representation of the delegate as a reference to a function or method) except if the delegate refers to a closure.

Class **CB** has the static method that allows to create and simultaneously call a delegate:

```
CB::delegate('Test[1]->foo', 'a', 1, 2);
```

Getting information about delegate

After creating a delegate, you can get various information about it. For example, you can check whether the delegated function or class method are callable, that is, whether a function, class or method exists:

```
// Creates a delegate.
$d = new Delegate('Test->foo');
// Checks whether the delegate is callable.
echo (int)$d->isCallable();
```

Method **isCallable** returns TRUE if there is not any obstacles to call a delegate and FALSE otherwise. Also this method takes boolean parameter (be default it is TRUE) which is responsible for autoloading of a delegate class.

There are some methods returning information about delegate type, delegated class or method name:

```
// Creates a delegate.
$d = new Delegate('Test[1]->foo');
// Gets name of the delegated class.
echo $d->getClass();
// Gets name of class method or function.
echo $d->getMethod();
// Gets the delegate type.
// Possible values: "closure", "function", "class" and "control".
echo $d->getType();
// Gets full information about delegate.
print_r($d->getInfo());
```

In some cases, you may need to get the object of a delegated class or parameters of a delegated function or method. There are the appropriate delegate methods for this purpose:

```
// Creates a delegate.
$d = new Delegate('Test[2]->foo');
// Gets an object of class "Test".
// Passes two parameters to the constructor.
$test = $d->getClassObject(['a', 'b']);
// Gets an array of parameters of method "foo".
$params = $d->getParameters();
```

Delegate permissions

It is often necessary to check whether a delegate belongs to the specified namespace, class, or a particular method of the class. This verification is needed, in particular, in the control system in order to prevent calls of arbitrary delegates on the client side.

To check the validity of the delegate to one or more permits you should use the method **isPermitted()**. The single parameter of this method is array of regular expressions (PCRE compatible) that restrict the area of permitted delegates. This array has the following structure:

```
[
  'permitted' => ['regexp1', 'regexp2', ... ],
  'forbidden' => ['regexp1', 'regexp2', ...]
]
```

If string representation of the delegate matches at least one of **permitted** regular expressions and none of **forbidden** regular expressions, the method returns TRUE. Otherwise it returns FALSE. Example:

```
// Creates permissions.
$permissions = ['permitted' => ['/^ClickBlocks\\\\(MVC|Web\\\POM)\\\\[^\\\\]*$/i'],
                'forbidden' => ['/^ClickBlocks\\\Web\\\POM\\\[^\\\\]*\[\d*\]->offset(Set|Get|Unset|Exists)$/i']];

// Creates delegates.
$d1 = new Delegate('ClickBlocks\MVC\MyPage->test');
$d2 = new Delegate('ClickBlocks\Web\POM\Control@myctrl->offsetSet');
$d3 = new Delegate('ClickBlocks\Web\POM\Control@myctrl->__set');

// Checks permissions.
```

```
echo (int)$d1->isPermitted($permissions); // will output 1.
echo (int)$d2->isPermitted($permissions); // will output 0.
echo (int)$d3->isPermitted($permissions); // will output 1.
```

In all these cases method **isPermitted()** returns TRUE.