Manual for MolComponents

Matthias Molitor - https://github.com/matthimatiker

13. April 2013

Inhaltsverzeichnis

1 Mol_Mail

1.1 Mol_Mail_Factory

1.1.1 Configuration

The factory expects a list of template configurations as first constructor parameter:

```php
$settings = array(
    'invitation' => array(
        'subject' => 'Please␣visit␣our␣site'
    ),
    'registration' => array(
        'subject' => 'Thank␣you␣for␣registration'
    )
);
$factory = new Mol_Mail_Factory(new Zend_Config($settings), $view);
```

The key equals the name of the template (“invitation” and “registration” this example). The template settings are assigned as value.

Each template configuration may contain the following settings:

- charset (string) - Charset of the email.
- subject (string) - Subject line, will automatically be translated.
- to (array(string)|string) - List of default “to” recipients
- cc (array(string)|string) - List of default “cc” recipients
- bcc (array(string)|string) - List of default “bcc” recipients
- replyTo (string) - A reply-to address
- from (string) - Sender address
- script.text (string) - Template that renders the text part
- script.html (string) - Template that renders the HTML part

Template settings example:

```php
$settings = array(
    'charset' => 'UTF-8',
    'subject' => 'Hello␣world!',
    'to' => array(
        'user@example.org',
        'second-user@example.org'
    ),
    'cc' => array(
        'another.user@example.org'
    ),
    'bcc' => array(
        'archive@example.com'
    ),
    'from'    => 'mailer@example.org',
    'script'  => array(
        'text' => 'hello.text.phtml',
        'html' => 'hello.html.phtml'
    )
);
```

1.2 Mol_Application_Resource_Mailer

1.2.1 Usage

Simple mail creation To create a mail factory without templates it is enough to just activate the resource:

```
resources.mailer = On
```

This configuration might be useful to avoid the manual creation of mail objects.
The following code can be used to create a mail object in the context of an action controller:

```
$mail = $this->getInvokeArg('bootstrap')->getResource('mailer')->create();
```

Configuration and usage of templates Advanced features can be used by configuring mail template configuration files and paths to view scripts:

```
resources.mailer.templates[] = APPLICATION_PATH "/mails/user-related.ini"
resources.mailer.templates[] = APPLICATION_PATH "/mails/notifications.ini"
resources.mailer.scripts[]   = APPLICATION_PATH "/mails/views"
```

Any number of template configurations and script paths can be added. In case of conflict, the later defined template configurations will overwrite the settings of their predecessors. The script paths are searched for mail content templates that are referenced by template configurations. Have a look at {@link Mol_Mail_Factory} to learn about the possible template settings.
Assuming that the template “user-registration” is defined, then the following code (in the context of an action controller) will create a pre-configured mail object:

```
$parameters = array('userName', $name);
$mail       =
    $this->getInvokeArg('bootstrap')->getResource('mailer')->create('user-registration',
    $parameters);
```

The create() method receives a template name and (optionally) a list of parameters that is passed to the configured content view scripts.

2 Mol_Controller

2.1 Mol_Controller_ActionParameter

2.1.1 Usage

To use the controllers functionality add the required request parameters as method arguments to the action:

```
public function myAction($page = 1 ) {
    // my code
}
```

The defined default values are used if the request does not contain a parameter that is named like the argument.
To use the parameters they must be documented in the DocBlock of the action method:

```
/**
 *
 * @param integer $page
 * /
```

The controller uses the DocBlock to determine the expected parameter type and performs a validation. If the validation was succesful the parameter is casted to the required type. Therefore the action method in the example will receive a real integer. If the validation fails an exception is thrown that should be handled by the error controller.
Currently the controller supports the following default types:

- integer
- double
- boolean
- string
- array (with supported types as content)
- mixed (avoids the arguments validation)

2.1.2 Extension

If needed arbitrary types may be added to the controller. Therefore a validator of the type Zend_Validate_Interface must be registered via registerValidator() for the new parameter type:

```
$controller->registerValidator(new MyDateValidator(), 'Datetime');
```

Thats enough to gain basic support for that type. If a action parameter of the type “Datetime” is documented the controller will use the registered validator to perform an argument check. If the parameter is valid it will be passed to the action method. However per default no filtering takes place. Therefore in this example the action method will receive a string as argument.

If an automatic conversion is desired an additional filter of the type Zend_Filter_Interface must be registered for the new type:

```
$controller->registerFilter(new MyDateFilter(), 'Datetime');
```

After successful validation the filter will be applied to the parameter. For example our filter could convert the string to a real Datetime object that will be passed to the action.

The registration of validators and filters may either take place inside the controller class (for example in its init() or preDispatch() method) or from the outside (for example by an action helper).

2.2 Mol_Controller_Exception_ParameterMissing

Exception that is used if a required parameter is not available.

2.3 Mol_Controller_Exception_ActionParameter

Basic class for all exceptions that are raised by the ActionParameter controller.

2.4 Mol_Controller_Exception_ParameterTagMissing

Exception that is used if an action methods DocBlock is defined, but does not contain enough information for a declared parameter.

2.5 Mol_Controller_Exception_ParameterNotValid

Exception that is used if provided parameter is not of the expected type.

2.6 Mol_Controller_Exception_DocBlockMissing

Exception that is used of an action is not documented.

3 Mol_Bootstrap

3.1 Mol_Application_Bootstrap_LazyLoad_ResourceDecorator

An application resource decorator that delays initialization by using a LazyLoader instance.

3.2 Mol_Application_Bootstrap_Injector

3.2.1 Usage

Create an injector:

```
$injector = new Mol_Application_Bootstrap_Injector($myBootstrapper);
```

Inject the bootstrapper into an object:

```
$injector->inject($object);
```

If the given value is not bootstrap aware, then the injector will just ignore it.
The injector will also ignore non-objects:

```
$value = $injector->inject(null);
```

The method inject() returns the provided value afterwards.

3.3 Mol_Application_Bootstrap_LazyLoader

Class that uses a callback to lazy load resources.

3.4 Mol_Application_Bootstrap

3.4.1 Usage

Bootstrapper configuration Use this bootstrapper as base class for your own bootstrap class:

```
class My_Bootstrap extends Mol_Application_Bootstrap
{
}
```

Add the bootstrap configuration to your application.ini to ensure that the bootstrapper is used:

```
bootstrap.path  = APPLICATION_PATH "/Bootstrap.php"
bootstrap.class = "My_Bootstrap"
```

Lazy loading configuration Now it is possible to activate lazy loading for any configured resource by setting the lazyLoad option:

```
resources.log.stream.writerName          = "Stream"
resources.log.stream.writerParams.stream = APPLICATION_PATH
    "/storage/logs/application.log"
resources.log.stream.writerParams.mode   = "a"
resources.log.lazyLoad                   = On
```

To load a resource the getResource() method of the bootstrapper is used as usual. The following code retrieves the lazy loaded logger in context of an action controller:

```
$bootstrap = $this->getInvokeArg('bootstrap');
$logger = $bootstrap->getResource('log');
```

Keep in mind that some resources must be executed early, as they modify the global state of the application and will not be retrieved explicitly via getResource().

4 Mol_Form

4.1 Mol_Application_Resource_Form

4.1.1 Usage

Activation The following line is enough to activate the form factory:

```
resources.form = On
```

Adding aliases Additionally aliases can be configured:

```
resources.form.aliases.login        = "My_Login_Form"
resources.form.aliases.registration = "My_Registration_Form"
```

The alias must be used as key, the form class that it points to as value.

Configuring plugins To enhance form that are created by the factory, plugins can be used.
The following line activates a plugin without providing further plugin options:

```
resources.form.plugins.myPlugin = "My_Form_Factory_Plugin"
```

Any key can be used for plugin registration, but the value must be the name of a class that implements the interface Mol_Form_Factory_Plugin.
To provide additional plugin options, the configuration must use the “class” and “options” keys:

```
resources.form.plugins.myPlugin.class = "My_Form_Factory_Plugin"
resources.form.plugins.myPlugin.options.name    = "Earl"
resources.form.plugins.myPlugin.options.filter = On
```

In this case the name of the plugin class is assigned to the “class” key. The “options” key must be any map or array of plugin options. These options will be directly passed to the plugin constructor.

Bootstrapper injection into plugins If the plugin class implements Mol_Application_Bootstrap_Aware, then the resource will inject the bootstrapper into the created plugin.

4.2 Mol_Form_Factory_Plugin_AutoCompleteOff

4.2.1 Usage

Configuration Activate Autocomplete plugin:

```
resources.form.plugins.autoComplete = "Mol_Form_Factory_Plugin_AutoCompleteOff"
```

4.3 Mol_Form_Factory_Plugin_Null

Form plugin that does nothing. Can be used to disable plugins via configuration or for testing.

4.4 Mol_Form_Factory_Plugin_Csrf

4.4.1 Usage

Configuration Activate CSRF plugin without further configuration:

```
resources.form.plugins.csrf = "Mol_Form_Factory_Plugin_Csrf"
```

Configure added element in detail:

```
resources.form.plugins.csrf.class = "Mol_Form_Factory_Plugin_Csrf"
resources.form.plugins.csrf.options.element.name    = "my_csrf_token"
resources.form.plugins.csrf.options.element.salt    = "secret-salt"
resources.form.plugins.csrf.options.element.timeout = 1800
```

4.5 Mol_Form_Factory_Plugin_AbstractPlugin

Optional base class for form factory plugins.

4.6 Mol_Form_Element_EmailAddress

4.6.1 Usage

The element can simply be added to any Zend_Form, no further configuration is required:

```
$email = new Mol_Form_Element_EmailAddress('email');
$email->setLabel('Email␣address');
$form->addElement($email);
```

Without further configuration the element will accept only valid email addresses:

```
// Returns true:
$element->isValid('matthias@matthimatiker.de');
// Returns false:
$element->isValid('hello');
```

Optionally the accepted hostnames can be restricted by providing a whitelist:

```
$element->setAcceptedHostnames(array('matthimatiker.de'));
// Returns true:
$element->isValid('matthias@matthimatiker.de');
// Returns false:
$element->isValid('matthias@another-hostname.org');
```

The rendered element contains a data attribute that holds a comma-separated list of allowed hostnames (if at least one hostname is available):

```
<input type="text" name="email" id="email" value=""
    data-allowed-hostnames="example.org,example.com" />"
```

That information may be used for client-side validation via JavaScript.

4.7 Mol_Form_Factory

4.7.1 Usage

Creation The form factory does not require any constructor arguments. Therefore, the following line is enough to create a new factory:

```
$factory = new Mol_Form_Factory();
```

Creating forms Without further configuration the factory is able to create forms by class name:

```
// Creates a new Zend_Form instance.
$form = factory->create('Zend_Form');
```

Each call to create() instantiates a new form, created instances are not cached.

Aliases The method addAlias() can be used to register a form alias.
Aliases point to form class names and can be passed to create() instead of the full class name:

```
$factory->addAlias('login', 'My_Login_Form');
// Creates an instance of My_Login_Form.
$form = $factory->create('login');
```

Names of form classes can also be used as alias. That allows mostly transparently switching of form types by configuration:

```
// Creates an instance of My_Login_Form.
$form = $factory->create('My_Login_Form');
$factory->addAlias('My_Login_Form', 'Another_Login_Form');
// Creates an instance of Another_Login_Form.
$form = $factory->create('My_Login_Form');
```

Plugins Plugins are used to improve just created forms.
Any number of plugins can be added to the factory. Instantiated forms are passed to each plugin.
The method registerPlugin() is used to add a plugin to the factory:

```
$factory->registerPlugin($myPluginInstance);
```

Plugins must implement the Mol_Form_Factory_Plugin interface.
Plugins are applied in registration order.
If an already existing form shall be enhanced by plugins, then the form instance can be passed to create():

```
$form = new Zend_Form();
$form = $factory->create($form);
```

5 Mol_Test

5.1 Mol_Test_Assertions_HttpResponse

Encapsulates assertions regarding the response object. Test cases may return an instance of Mol_Test_Assertions_HttpResponse
to support speaking method calls:

```
// assertResponse() returns an instance of Mol_Test_Assertions_Response
$this->assertResponse()->contains('Hello!');
```

5.2 Mol_Test_Controller_Action_Helper_ViewRenderer

A ViewRenderer that avoids global dependencies and is used for testing.

5.3 Mol_Test_Controller_Action_Helper_Redirector

Redirector helper that is used for testing. Avoids connections to global dependencies and prevents calls to exit(), which would lead to untestable code.

5.4 Mol_Test_Bootstrap_Mock

Class that may be used to mock bootstrappers. It does not implement the extensive bootstrapper interfaces, but it supports the most important methods that are used by controllers to load resources and options.

5.5 Mol_Test_Exception

Exception that is used by the testing helper classes, for example to indicate restrictions.

5.6 Mol_Test_Http_Client_Adapter

This class is an improved test adapter for Zend_Http_Client objects. In addition to iterate over a set of responses it is also possible to register responses that are only delivered if the requested url matches a specific pattern.
Per default a “400 Bad Request” response is registered. It may be removed by using the following code:

```
$adapter->setResponse(array());
```

Specific responses can be registered for static urls:

```
$adapter->addResponse($myResponse, 'http://www.matthimatiker.de/index.php');
```

If multiple responses are registered for the same url then the adapter will iterate over that response set as usual:

```
$adapter->addResponse($myResponse, 'http://www.matthimatiker.de/index.php');
$adapter->addResponse($anotherResponse, 'http://www.matthimatiker.de/index.php');
```

The first request for “http://www.matthimatiker.de/index.php” returns $myResponse. The next request to the same url will return $anotherResponse. For following requests the adapter will start from the beginning.

The character “*” may be used as wildcard when registering responses for urls. Therefore it is possible to register a single response for multiple urls. In the following exampple a response is registered for all html pages at matthimatiker.de:

```
$adapter->addResponse($myResponse, 'http://www.matthimatiker.de/*.html');
```

In this configuration the adapter will return $myResponse for “http://www.matthimatiker.de/index.html” as well as for “http://www.matthimatiker.de/login/start.html”.

If multiple url patterns match the same url the responses for the pattern that was registered first will be returned:

```
$adapter->addResponse($myResponse, 'http://www.matthimatiker.de/*.html');
$adapter->addResponse($anotherResponse, 'http://www.matthimatiker.de/demo.*');
```

Requesting “http://www.matthimatiker.de/demo.html” will return $myResponse in this case.

5.7 Mol_Test_WebControllerTestCase

5.7.1 Requirements

Naming and path conventions The WebControllerTestCase detects and loads the tested class automatically. Therefore it is important to name and place the TestCase correctly.

The name of the test class must equal the name of the tested controller class plus the suffix “Test”: Controller class: IndexController Test class: IndexControllerTest

Also the PHP file with the test class must be placed analog to the controller file: Controller file: /application/controllers/IndexController.php Test file: /tests/application/controllers/IndexControllerTest.php

Customize controller loading behavior If the name of the test case and the name of the controller class do not fit together, then `getControllerClass()` must be overwritten by the subclass. The method should return the class name of the tested controller:

```
public function getControllerClass()
{
    return 'My_Custom_Controller';
}
```

Most probably the test case is not able to locate the controller if the naming conventions are not fulfilled. If the class loader is not able to load the tested controller, then the subclass also has to overwrite the method `getControllerPath()`. It must return the path to the controller file:

```
public function getControllerPath()
{
    return APPLICATION_PATH . '/modules/custom/MyController.php';
}
```

The file must contain the controller class that is provided by getControllerClass(), otherwise an exception will be thrown.

5.7.2 Usage

Prepare the environment

Simulate configuration options The prepared bootstrapper is used to simulate configuration options. The following configuration. . .

```
$options = array('name' => 'Dori', 'mail' => 'dori@demo.com');
$this->bootstrapper->setOptions('demo' => $options);
```

. . . is equal to this ini file configuration entries:

```
demo.name = "Dori"
demo.mail = "dori@demo.com"
```

Simulate resources Resources (for example database connections) may also be simulated via bootstrapper:

```
$this->bootstrapper->simulateResource('Locale', new Zend_Locale('en'));
```

The following resources are simulated per default to reduce manual setup work:

- Log
- Layout
- View

Simulate parameters The request parameters are simulated via setGet() and setPost():

```
$this->setGet(array('page' => '1'));
$this->setPost(array('search' => 'hello'));
```

Multiple calls to setGet() or setPost() will not clear parameters that were provided previously, instead the new paramters will be added:

```
$this->setGet(array('page' => '1'));
$this->setGet(array('name' => 'Al'));
```

In this example the controller will receive the GET parameters “page” and “name”.
User parameters may be simulated via setUserParams():

```
$this->setUserParams(array('target' => 'stats'));
```

Usually user parameters are passed via forwarding.

Simulate identity The controller can use Zend_Auth to determine the currently logged in user.
Use setIdentity() to simulate that user:

```
$this->setIdentity('user@example.org');
```

Pass null as argument to simulate a guest (not logged in):

```
$this->setIdentity(null);
```

This is also the default state after initial setup.
Use getIdentity() as a shortcut to retrieve the current identity:

```
$currentIdentity = $this->getIdentity();
```

Simulate invoke args The controller is created in the set up phase, therefore changing the invoke args afterwards is not possible.
There are several ways to overcome this issue. To change the invoke args for all created controllers, the createInvokeArgs() method can be overwritten:

```
protected function createInvokeArgs()
{
    // Create and returns customized invoke args here.
    return array();
}
```

To test a specific invoke arg combination there is also the possibility to re-create the controller within a test method:

```
$invokeArgs = array();
$this->controller = $this->createController($invokeArgs);
```

Afterwards the new controller can be tested as usual.

Testing

Single controller methods Single controller methods can be executed directly:

```
$this->controller->myAction();
```

Afterwards, the for example the state of the response can be checked to verify the controller behavior:

```
$this->assertResponse()->hasHeader('Expires');
```

Action tests including lifecycle More complex tests might rely on the controller lifecycle. The `dispatch()` method helps to to simulate the controller lifecycle as it occurs in the dispatch loop.

The lifecycle includes:

- calling pre-dispatch hooks of action helpers
- calling controller preDispatch()
- executing the requested action
- calling controller postDispatch()
- calling post-dispatch hooks of action helpers

To execute an action its name (not the name of the action method) is passed to `dispatch()`:

```
$this->dispatch('my-action');
```

Please note: During dispatching the environment is modified and another call to dispatch() will not re-initialize it properly, therefore dispatch() should be called only once per test.

After executing an action the provided assertions are used to check the results just like in a single method test.

5.8 Mol_Test_View_Helper_Url

View helper that is used to simulate the original view helper from Zend. Avoids dependencies to the front controller und caches call parameters for later analysis.

5.9 Mol_Test_Bootstrap

5.9.1 Usage

The static create() method may be used to instantiate a new bootstrapper:

```
$bootstrapper = Mol_Test_Bootstrap::create();
```

This helper method automatically creates an Zend_Application object that is required as constructor argument for bootstrappers.

The method simulateResource() is used to simulated arbitrary resources:

```
$bootstrapper->simulateResource('view', new Zend_View());
```

After resource injection the usual boostrapper methods may be used to retrieve the simulated values:

```
$view = $bootstrapper->getResource('view');
```

6 Mol_Filter

6.1 Mol_Filter_Cast

Filter that casts values to the type that was specified in the constructor. Example:

```
$filter = new Mol_Filter_Cast('integer');
// Returns the integer 42.
$filter->filter('42');
```

6.2 Mol_Filter_Null

Null object for filters. Returns the argument value without doing anything.

Example:

```
$filter = new Mol_Filter_Null();
// Returns the integer 42.
$filter->filter(42);
// Returns the string "Hello!".
$filter->filter('Hello!');
```

7 Mol_View

7.1 Mol_View_Helper_Favicon

Helper that generates a link tag that defines the favicon to use. The following examples demonstrate the usage of this helper in views.

The icon url is given to the entry method favicon():

```
<!-- Generates the markup for the icon "/favicon.ico" -->
<?= $this->favicon('/favicon.ico'); ?>
```

If the url is omitted the icon that was provided before will be used:

```
<?php $this->favicon('/favicon.ico'); ?>
<!-- Generates the markup for the icon "/favicon.ico" -->
<?= $this->favicon(); ?>
```

If a icon url was provided multiple times the one that was given last will be used:

```
<?php $this->favicon('/favicon.ico'); ?>
<?php $this->favicon('/another_favicon.ico'); ?>
<!-- Generates the markup for the icon "/another_favicon.ico" -->
<?= $this->favicon(); ?>
```

If no icon url is provided the helper will not generate any markup:

```
<!-- Generates an empty string. -->
<?= $this->favicon(); ?>
```

7.2 Mol_View_Helper_To

View helper that simplifies generating urls within the application. Create a url with the parameter “confirm” within a view:

```
<?= $this->to('my-action', 'my-controller', 'my-module')->withParam('confirm', 1); ?>
```

If the default module is addressed the module argument may be omitted:

```
<?= $this->to('my-action', 'my-controller'); ?>
```

Multiple parameters may be added by using method chaining:

```
<?= $this->to('my-action', 'my-controller')->withParam('action',
    'delete')->withParam('confirm', 1); ?>
```

Special routes may be used via withRoute():

```
<?= $this->to('my-action', 'my-controller')->withRoute('custom'); ?>
```

Add an anchor to the url:

```
<?= $this->to('my-action', 'my-controller')->withAnchor('info'); ?>
```

Append all parameters of the current request:

```
<?= $this->to('my-action', 'my-controller')->keepParams(); ?>
```

7.3 Mol_View_Helper_Value_Url

Contains url data and generates the url if it is casted to a string.

8 Mol_Validate

8.1 Mol_Validate_Form_Relation_NotContains

Validator which ensures that a value does not contain another value. Can be used in combination with Mol_Validate_Form_Ele

8.2 Mol_Validate_Form_Relation_Equal

Validator that checks if two values are equal. Can be used in combination with Mol_Validate_Form_ElementRelation.

8.3 Mol_Validate_Form_Relation_GreaterThanOrEqual

Validator that checks if a value is greater than or equal to a compared value. Can be used in combination with Mol_Validate_Form_ElementRelation.

8.4 Mol_Validate_Form_Relation_LessThanOrEqual

Validator that checks if a value is less than or equal to a compared value. Can be used in combination with Mol_Validate_Form_ElementRelation.

8.5 Mol_Validate_Form_Relation_Contains

Validator which ensures that a value contains another value. Can be used in combination with Mol_Validate_Form_ElementRel

8.6 Mol_Validate_Form_Relation_GreaterThan

Validator that checks if a value is greater than a compared value. Can be used in combination with Mol_Validate_Form_Elemen

8.7 Mol_Validate_Form_Relation_NotEqual

Validator that checks if two values are *not* equal. Can be used in combination with Mol_Validate_Form_ElementRelation.

8.8 Mol_Validate_Form_Relation_LessThan

Validator that checks if a value is less than a compared value. Can be used in combination with Mol_Validate_Form_ElementRe

8.9 Mol_Validate_Form_ElementRelation

8.9.1 Usage

The validator is simply added to a form element. The relation and the compared element are passed to the constructor of the validator.

Example:

```
// Create elements that expect date values in format yyyy-mm-dd:
$from = new Zend_Form_Element_Text('from);
$to␣=␣new␣Zend_Form_Element_Text('to');
//␣Ensure␣that␣the␣provided␣"to"␣date␣is␣only␣valid␣if␣it␣is
//␣beyond␣the␣"from"␣date:
$to->addValidator(new␣Mol_Validate_Form_ElementRelation('>',␣$from));
```

The constructor accepts the relation first, then the compared element. That order makes it easy to read the relation rule. The example above tells: $to > $from

As shown relation identifiers (strings) can be provided if built-in relations are used.

The following relation identifiers are currently supported:

- ==
- !=
- <
-
- <=
-

```
=
```

More specific or custom relations can be provided as object instead of the relation identifier.

8.9.2 Custom relations

The relation validators that are used internally must implement the Zend_Validate_Interface interface.

The isValid() method should accept a second (for compability reasons optional) value:

```
public function isValid($value, $other = null)
{
}
```

The relation object is then passed to the constructor of the ElementRelation validator. For example we could use a custom relation validator that checks for a maximal difference between two values. Adding it could look like this:

```
$relation  = new My_Custom_MaxDiffRelation(42);
$validator = new Mol_Validate_Form_ElementRelation($relation, $comparedElement);
```

8.9.3 Error messages

The ElementRelation validator will pass through the messages that are provided by the internal relation validator.

It will also translate these messages if possible/desired. Therefore the relation does not need to take care of translation itself.

Furthermore the ElementRelation supports some additional placeholders in the messages. Currently the following additional placeholders are supported:

- %compareName% - The name of the compared element
- %compareLabel% - The label of the compared element
- %compareValue% - The value that was compared

8.10 Mol_Validate_Boolean

A validator that checks if a value may be interpreted as boolean.

8.11 Mol_Validate_False

A validator that always returns false. May be used as a null or special case object.
The error message is customizable:

```
$validator = new Mol_Validate_False('Invalid');
// Returns false.
$validator->isValid('hello');
// Returns an array with the message "Invalid".
$validator->getMessages();
```

8.12 Mol_Validate_True

A validator that returns always true. May be used as null or special case object.

8.13 Mol_Validate_Suffix

8.13.1 Usage

Multiple allowed suffixes can be passed to the constructor:

```
$allowed   = array('.txt', '.doc');
$validator = new Mol_Validate_Suffix($allowed);
```

A single suffix is also allowed as constructor arguments:

```
$validator = new Mol_Validate_Suffix('.txt');
```

The method setSuffixes() may be used to change the allowed suffixes after creation:

```
$validator = new Mol_Validate_Suffix('.txt');
$validator->setSuffixes('.png', '.jpg');
```

The validator accepts only strings that end with an allowed suffix:

```
$validator = new Mol_Validate_Suffix('.txt');
// Returns true:
$validator->isValid('test.txt');
// Returns false:
$validator->isValid('test.pdf');
```

The validator will accept any string if the list of allowed suffixes is empty:

```
$validator = new Mol_Validate_Suffix(array());
// Returns true:
$validator->isValid('test.pdf');
```

8.14 Mol_Validate_String

A validator that checks if a given value is a string. Example:

```
$validator = Mol_Validate_String();
// Returns false.
$validator->isValid(42);
// Returns true.
$validator->isValid('42');
```

9 Mol_DataType

9.1 Mol_DataType_Map

A hash map implementation that returns a configured standard value if no value is defined for a requested key. Configure a default value:

```
// Returns 0 per default.
$map = new Mol_DataType_Map(array(), 0);
$map['x'] = 5;
// Prints 5.
echo $map['x'];
// Prints the default value 0.
echo $map['y'];
```

Register a value for multiple keys:

```
// Registers 42 for the keys a, b and c.
$map->register(42, array('a', 'b', 'c'));
```

9.2 Mol_DataType_String

9.2.1 Description

Each string is represented by an object that encapsulates the raw string value and the charset.

The content of a string object is not changable, if a modification is performed then a new string object will be created and returned.

If necessary methods take the charset into account. Therefore it is possible to compare string with different charsets and so on.

9.2.2 Usage

String objects are instantiated via create():

```
$stringObject = Mol_DataType_String::create('my␣string');
```

If no charset is provided then UTF-8 is assumed, but you may specify the charset of the provided string too:

```
$stringObject = Mol_DataType_String::create('my␣string', 'latin1');
```

Once a string object is created you may use its methods to inspect the string:

```
$stringObject->endsWith('Test');
```

All methods respect the charset if necessary, so multi-byte characters are handled correctly:

```
$stringObject = Mol_DataType_String::create('äüö', 'UTF-8');
// Returns: array('ä', 'ü', 'ö')
$characters   = $stringObject->toCharacters();
```

10 Mol_Util

10.1 Mol_Util_ObjectBuilder

10.1.1 Usage

Creating a builder The most simple builder can be created without any constructor argument:

```
$builder = new Mol_Util_ObjectBuilder();
```

This builder does not enforce any type when creating objects.

Optionally a type constraint can be passed:

```
$builder = new Mol_Util_ObjectBuilder('Countable');
```

This builder checks the type constraint *before* creating an object and rejects instantiation requests for classes that do not fulfill the type requirement. That is especially useful if the object class was provided by configuration and a common base class or interface is required.

It is even possible to provide multiple type constraints:

```
$builder = new Mol_Util_ObjectBuilder(array('Traversable', 'Countable'));
```

In this case the builder will only instantiate classes that fulfill *all* of the given type constraints.

Building objects The create() method is used to instantiate objects of a given class:

```
$array = $builder->create('SplFixedArray');
```

Constructor arguments can be passed as second argument:

```
$array = $builder->create('SplFixedArray', array(100));
```

Object creation will fail with an InvalidArgumentException if the type constraints are not fulfilled:

```
$builder = new Mol_Util_ObjectBuilder('ArrayObject');
// This creation will fail:
$array = $builder->create('SplFixedArray', array(100));
```

10.2 Mol_Util_StringStream

Helper class that simplifies reading from strings via stream functions. The helper may be used to pass string to function that usually support files only.

Example:

```
$data = 'Hello␣World!';
// $content contains "Hello World!"
$content = file_get_contents(new Mol_Util_StringStream($data));
```

10.3 Mol_Util_Math

Provides mathematical helper methods.

10.4 Mol_Util_TypeInspector

10.4.1 Usage

A new type inspector is simply created without any argument:

```
$inspector = new Mol_Util_TypeInspector();
```

Although there is currently no internal class state, an inspector object must be created to use the class. This is intended, as for example a cache for already checked types might be added in the future. Such a feature should not lead to global attributes later.

Checking types The TypeInspector provides several methods to check types by name.

Simple checks test for classes and interfaces:

```
// Returns true.
$isClass = $inspector->isClass('ArrayObject');
// Returns false.
$isInterface = $inspector->isInterface('ArrayObject');
```

The method isType() can be used if it is not important whether a class or interface name is provided:

```
$isClassOrInterface = $inspector->isType('ArrayObject');
```

The is() method can be used to perform complex type checks. It checks if a given type fulfills all provided type constraints:

```
// Returns true, as ArrayObject is Traversable as well as Countable.
$constraintsFulfilled = $inspector->is('ArrayObject', array('Traversable', 'Countable'));
// Returns false, as ArrayObject is Countable, but it is not an instance of SplStack.
$constraintsFulfilled = $inspector->is('ArrayObject', array('Countable', 'SplStack'));
```

For convenience, also a single type constraint can be passed to is(). The following checks are equivalent:

```
$inspector->is('ArrayObject', array('Countable'));
$inspector->is('ArrayObject', 'Countable');
```

Asserting type rules Besides methods for type checks, there are also methods that can be used to assert certain type conditions.

The assertion methods throw an InvalidArgumentException if the requested condition does not hold. Otherwise they will do nothing.

There are several assertion methods to apply simple type rules:

```
// Throws an exception if ArrayObject is not a class.
$inspector->assertClass('ArrayObject');
// Throws an exception if ArrayAccess is not an interface.
$inspector->assertInterface('ArrayAccess');
// Throws an exception if SplStack is neither a class nor an interface.
$inspector->assertType('SplStack');
```

The assertTypes() method can be used to check a list of types at once:

```
// Throws an exception if any of the entries is neither class nor interface.
$inspector->assertTypes(array('ArrayObject', 'ArrayAccess', 'SplStack'));
```

Like is(), assertFulfills() can be used to check complex type rules:

```
// Throws an exception if ArrayObject does not fulfill all of the
// provided type constraints.
$this->assertFulfills('ArrayObject', array('Countable', 'Traversable'));
```

10.5 Mol_Util_MemoryStreamWrapper

Low level class that allows to access MemoryStream objects via stream functions.

10.6 Mol_Util_MemoryStream

Stream helper class that allows writing into memory. May be used with functions that usually write to files only.

Example:

```
$stream = new Mol_Util_MemoryStream();
file_put_contents($stream, 'Hello␣World!');
```

In contrast to the PHP memory stream wrapper this stream allows reading from the stream later, even if it was closed after previous write operations:

```
$stream = new Mol_Util_MemoryStream();
$content = file_get_contents($stream);
```

The stream handle is not required.

The stream content may be pre-filled by passing the initial data to the constructor:

```
$stream = new Mol_Util_MemoryStream('Hello␣World!');
```

10.7 Mol_Util_Stringifier

Class that converts data into simple strings. This class does *not* serialize data, the generated strings are just meant for output.

10.8 Mol_Util_String

Contains helper methods for string handling. This class contains lightweight helper methods that simplify string handling. All operations are independent of the underlying charset of the subject string.

If you need to perform actions that depend on the charset, then try to use Mol_DataType_String as it includes charset handling.

Hint: The helper methods are static and require a string to operate on. That string is called “subject” and its always the first argument in all methods.