# Error handling

The framework provides the ability to handle code errors or exceptions and also provides the means for getting detailed information about errors.

The error and exception handling turns on during the framework's initialization. You can also disable it if necessary, or to set the desired level of sensitivity to errors:

```
// Turns off the error handling.
CB::errorHandling(false);
// Turns on the error handling,
// and also sets the sensitivity level to errors.
CB::errorHandling(true, E_ALL & ~E_NOTICE);
```

The default sensitivity level is E_ALL.

You can check whether the error handling is turned on by calling method **isErrorHandlingEnabled()**:

```
echo (int)CB::isErrorHandlingEnabled();
```

This method returns TRUE if the error handling is turned on and FALSE otherwise.

As previously noted in Chapter 3, the framework supports the debug mode in which comprehensive information about errors or exceptions is displayed. The boolean configuration parameter **debugging** is responsible for the debug mode. The default value of this parameter is FALSE.

To get full information about an error the framework uses static method **analyzeException** of class **CB**. This method takes as a single parameter an instance of class **Exception** and returns an associative array of detailed data of the error or exception.

```
print_r(CB::analyzeException(new RuntimeException('Error Message')));
```

To throw an exception method **exception()** is used:

```
CB::exception(new LogicException('Error Message'));
```

Difference between this method and operator **throw** is that it works in all magic methods. For example, you can not throw an exception in the magic method **__toString**, but using the **exception()** it is possible:

```
...
public function __toString()
{
  try
  {
    return $this->build();
  }
  catch (Exception $e)
  {
    CB::exception($e);
  }
}
...
```

# Class ClickBlocks\Core\Exception and error messages

There is agreement on the formation and location of error messages generated by the framework. All framework error messages are stored as constants of classes which generate these messages. The base class **CB** stores common error messages.

Constant name defining a template of error message should follow the following pattern ERR*{class_name}*{index}. The very name of the constant is called the error token.

Error templates can contain replaceable sequence of characters that is called template variables. Template variable symbol is "[{var}]". To form a complete error message static method **error()** of class **CB** is used.

```
// Class that contains error templates.
class Foo
{
  const ERR_FOO_1 = 'My [{var}] error [{var}]';
  ...
}

// Forms the error message by its token.
echo CB::error('Foo::ERR_FOO_1', 'first', 'test');
// Displays 'My first error test (Token: Foo::ERR_FOO_1)'
```

Class **ClickBlocks\Core\Exception** is intended for throwing exceptions with messages in the framework style. Constructor arguments of this class are similar to arguments of method **error**. Also with this class you can easily identify the error and to obtain information about it.

```
// Defines class throwing exceptions.
class Foo
{
  const ERR_FOO_1 = '[{var}] error message';

  // Method that throws the exception.
  public function run()
  {
    throw new Core\Exception($this, 'ERR_FOO_1', 'My');
  }

  ...
}

// Throws the exception and catches it.
try
{
  (new Foo())->run();
}
catch (Core\Exception $e)
{
  // Displays the class name in which the error is occurred.
  echo 'Class: ' . $e->getClass();
  // Displays the error token.
  echo 'Token: ' . $e->getToken();
  // Displays comprehensive error information.
  print_r($e->getInfo());
}
```

# Your own error handler

You can define a custom error handler. To do this configuration variable **customDebugMethod** is used. The value of this variable is a delegate that is invoked when some error is occurred. Delegate takes as parameters an instance of exception and an associative array containing detailed information about exception.

Delegate has to return boolean value. If this value is TRUE then the framework takes over the error processing. In other case the error processing is a task of the framework.

```
// Sets the custom error handler.
CB::getInstance()['customDebugMethod'] = $handler;

// Defines the custom handler.
$handler = function(Exception $e, array $info)
{
  echo $e->getMessage();
  // Forbids subsequent error processing by the framework.
  return false;
}
```

> Note that the custom error handler can be called only if the error handling is turned on (method **CB::isErrorHandlingEnabled()** returns

> TRUE). Otherwise you should turn on the error handling via method **CB::errorHandling()**.

Besides of setting of the own error handler the framework allows to set own debug template for displaying error debug information (**debugging** is TRUE) or specify a template to print messages about the fact of an error in production mode (**debugging** is FALSE). For this you can use configuration properties **templateDebug** and **templateBug**.

```
$cb = CB::getInstance();
// Specifies path to the debug template.
$cb['templateDebug'] = '/path/to/new/debug/template';
// Specifies path to the error template for the production mode.
$cb['templateBug'] = '/path/to/new/bug/template';
```

> If error templates are not defined, the default templates will be used. These default templates are respectively placed in constants **CB::TEMPLATE_DEBUG** and **CB::TEMPLATE_BUG**.

# Error logging

Configuration parameter **logging** is responsible for error logging. If this boolean parameter equals TRUE then all information about catched errors or exceptions will be stored in log files. Configuration variable **dirs['logs']** determines the directory of log files.

You can also store in the logs any other information. Example of writing data to a log file:

```
CB::log(['foo' => 'some data']);
CB::log('another data');
CB::log(new MyClass());
```

The names of the log files are auto-generated. You will be able to view the contents of the log files in the configurator.

As in the case of the error handler you can specify own error logger. Configuration variable **customLogMethod** determines a delegate of the custom error logger. This delegate takes as a single parameter an array of error information.

```
// Sets own logger function.
CB::getInstance()['customLogMethod'] = $logger;

// Defines the logger function.
$handler = function(array $info)
{
  ...
  // Here we stores the error information.
  ...
}
```