
phpReport

Release 3.0.0

Günter Pöhl

Sep 01, 2023

TABLE OF CONTENTS

1	Introduction	3
2	Getting started	5
2.1	Installation	5
3	Actions	9
3.1	Named events	9
4	Prototyping	13
5	Configuration	15
6	Data input	19
6.1	Joining Data	19
7	Data output	21
8	Group changes	23
9	Aggregation	25
9.1	Collector	25
9.2	Default Collectors	25
9.3	Row counter	25
9.4	Group counter	26
9.5	Calculator	29
9.6	Sheet	30
10	Getting values	33
11	Methods returning information	35
12	Overview of methods and properties	37
12.1	Methods to instantiate report class	37
12.2	Methods for data handling	37
12.3	Action methods called from report class	37
12.4	Methods returning information	38
12.5	Public Properties	38
12.6	Prototyping methods	38
13	Indices and tables	39
	Index	41

Edition for phpReport 3.0

by Günter Pöhl

This work is licensed under the GNU Lesser General Public License (LGPL) v3.0

INTRODUCTION

phpReport is a solid foundation for most applications working with data sets having to deal with group changes or to calculate totals and subtotals. Creating reports is the most typical scenario but there are a lot of other use cases.

While **phpReport** will handle most of the repetitive tasks it is wide open to let you use all your business objects and methods. It also integrates seamlessly without any configuration into any framework.

The absence of data input tools permits the use of your own tools. This has the advantage of

- Nothing extra to learn
- Free selection of the most suitable access strategy
- Access to all data sources
- Working with all kind of data like arrays, csv files, Excel tables or data objects

To enhance the data input capabilities you can easily combine data rows with any other data sources and handle them together as a single source. This is implemented by the *join* function and let's you

- Iterate over nested or multidimensional arrays
- Iterate over one to many relations provided by data objects
- Read and iterate over additional data from any source

phpReport also doesn't prepare any output. Actions are invoked on certain events (e.g. call a method *customerHeader* after a group change) which gives you 100% control over what and how something will be done. Create most complex reports, create grid tables, print charts, write data into a file or database, send mails, create pdf files or do whatever else comes into your mind.

Of course there is much more **phpReport** has to offer.

Aggregating values By calling the *compute* method a calculator object will be instantiated. This will aggregate the declared source value and provides for all declared groups totals, subtotals and running totals as well as counters for values being not null or empty. The calculator object might also determine minimum and maximum values.

Sheets Sheets aggregates values vertically. This makes tabular representation of data a snap. Columns can be pre-defined or will be instantiated on demand based on the value of a declared key source.

Grouping, ranging and filtering of sheet columns provides countless combinations for building totals.

Prototyping Prototyping is a way to simulate, replace or extend user actions with prototype actions during development.

Each prototype action generates a html table containing some interesting stuff about the current data row, data group values and computed values.

GETTING STARTED

2.1 Installation

Official installation method is via composer and its packagist package [gpoehl/phpReport](<https://packagist.org/packages/gpoehl/phpReport>).

```
` $ composer require gpoehl/phpReport `
```

After instantiation of the report class call

Then call the group() method for each group that will be controlled.

Use the compute() method to get aggregate functions for an field / attribute. The sheet() method organizes these in an horizontal way like in a spreadsheet.

Then pass your data to the run() method.

At certain events defined actions will be executed. As you can imagine thats the place to do whatever needs to be done.

Note: You have full access to all your existing models. No matter if these are data models or models implementing your business rules.

A report which controls two groups and summarizes totals may look like this example.

```
use gpoehl\phpreport\Report;

require_once(__DIR__ . '/../vendor/autoload.php');

class MyFirstReport {

    // The report object. Use $rep->output to render the output.
    public $rep;

    // @param $data Data usually read by controller object.
    public function __construct(iterable | null $data){
        $this->rep = (new Report($this))
            ->group('customer')
            ->group('invoice', 'invoiceID')
            ->compute('sales', fn($row) => $row->amount * $row->price);
        ->run($data);
    }

    public function init() {
        return "<h1>My very first report</h1>";
    }
}
```

(continues on next page)

(continued from previous page)

```

    }

    public function customerHeader($customer, $row) {
        return "<h2>Customer $customer</h2>" . $customer->address;
    }

    public function invoiceHeader($invoice, $row) {
        return "<h3>Invoice ID = $invoice</h3>";
    }

    // Will be called for each data row
    public function detail($row, $rowKey) {
        return "<br>$rowKey $row->item: $row->description";
    }

    public function invoiceFooter($invoice, $row) {
        return "Total sales for invoice $invoice = " . $this->rep->total->sales->
↪sum();
    }

    public function customerFooter($customer, $row) {
        return "Total sales for customer $customer = " . $this->rep->total->sales->
↪sum();
    }

    public function totalFooter() {
        return
            "<br>Total sales: " . $this->rep->total->sales->sum() .
            "<br>Total number of customers: " . $this->rep->gc->customer->sum() .
            "<br>Total number of invoices: " . $this->rep->gc->invoice->sum() .
            "<br>Total number of rows: " . $this->rep->rc->sum();
    }
}

```

Main features are:

Data handling In the most simple form you will call the run method and pass your dataset to this method. phpReport will iterate over this dataset and execute certain actions.

It is not required to build a dataset upfront. You can optionally call the run method without any data and call the next method once for each data row. This might save a lot of memory and processing time.

phpReport is also able to handle multi-dimensional arrays. Calling the data method tells which element contains the sub-array. phpReport will then iterate over the sub-array. Sub-array can also have elements where you want specific actions when the value changes. So call the group method after the data method to declare this element. Same is true for values to be aggregated.

phpReport might also get related data to a given row. See data section for details. Out of the box phpReport offers row counters.

Group changes phpReport monitors as much groups as you like. As soon as a value changes phpReport executes certain actions like calling group header and group footer methods. See actions section for more details. To let phpReport know which attributes or elements should be monitored call the group method once for each group. Out of the box phpReport offers group counters which lets you know how often a certain value (or group) occurs in another group.

Prototyping Beginners and experienced users of phpReport can benefit from the prototype system. Prototyping lets you know which method would have been called, what data row triggered the actions, what are the values of the

group fields and the values of aggregated fields. [Prototyping](prototype.rst)

ACTIONS

Before we start to learn any details about using **phpReport** we need to know something about actions.

Calling the `run()` method starts the execution and **phpReport** takes control over the program flow. Whenever an important event occurs an action might be invoked.

Typical actions are:

- call a method in the target object (The target parameter of the report class)
- append a string to the output
- execute a callable
- call a method in the prototype class

Within the configuration file you can specify exactly what action will be executed. During instantiation of the report class you might replace some of the configuration parameters.

Events like a group change will trigger individual action for defined groups. To be able to execute different action types the `group()` method also allows replacing the default action.

The default rules are simple: 1) If action is a string this string will be appended to `$output`. 2) If action is a method name the method will only be called when the method exist. The returned value will be appended to `$output`. 3) If action is a callable the returned value will be appended to `$output`.

The above rules can be altered by calling the `setCallOption()`.

3.1 Named events

Each named event is mapped to an action. Below are all events listed. Parameters passed to the action object are shown in parenthesis.

3.1.1 Data independent events

These methods are called even when no data are provided.

init()

First event. Use to initialize application properties independent from the `__construct` method.

close()

Last event. Use to clean up the dishes independent from __destruct method.

totalHeader()

Called once to build the total header page of the report.

totalFooter()

Called once to build the total footer page of the report.

3.1.2 Data driven events

noData()

This event only occurs when the given data set is empty. In this case the following events will never be raised.

beforeGroupHeader(\$groupValue, \$row, \$rowKey)

Raised before the group header action. To suppress any further actions return 'false'.

param mixed \$groupValue The current group value.

param array|object \$row The current row which triggered the group change.

param mixed \$rowKey The rowKey is the key of the current row taken from the input data set or given by calling the next() method.

groupHeader(\$groupValue, \$row, \$rowKey)

Raised when group values between two rows are not equal. Each group has its own groupHeader.

Group headers are executed from the changed group level down to the lowest declared group (within an data dimension).

After executing all headers the detail event will be performed.

param mixed \$groupValue The current group value.

param array|object \$row The current row which triggered the group change.

param mixed \$rowKey The rowKey is the key of the current row taken from the input data set or given by calling the next() method.

groupFooter(\$groupValue, \$row, \$rowKey, \$dimID)

groupFooters are executed like groupHeaders when group values between to rows are not equal.

But the footers are called from the lowest declared group (within a dimension) up to the changed group.

The signature is the same as for groupHeader() methods but the values belongs to the last row within this group and **not** to the latest read row which triggered the group change.

afterGroupFooter(\$groupValue, \$row, \$rowKey)

Raised after groupFooter. Might be used to handle the current output. Examples: Write output to file and reset current output. Generate pdf file. Send output per mail.

beforeDetail(\$row, \$rowKey)

Raised before detail actions. This is after the last afterGroupFooter action. Might be used to create header for details when groupFooter of last group is not suitable or to be more flexible.

detail(\$row, \$rowKey)

Executed for each row of the last data dimension. When the row triggered a group change then the related group footers and group headers will be called before.

param array|object \$row The current row.

param mixed \$rowKey The rowKey is the key of the current row taken from the input data set or given by calling the next() method.

afterDetail(\$row, \$rowKey)

Raised after detail actions.

3.1.3 Methods for multi dimensional data

Following events belongs only to data sources having joined data.

noData_n(\$dimID)

Called when the declared source for the next data dimension doesn't return any data. :param int \$dimID: The ID of data dimension not having related data.

detail_n(\$row, \$rowKey)

Except for the last dimension this event is raised for each data row (See detail method).

When group(s) are declared for this data dimension consider using groupHeader and groupFooter methods instead.

param array|object \$row The current row.

param mixed \$rowKey The rowKey is the key of the current row taken from the input data set or given by calling the next() method.

noGroupChange_n(\$row, \$rowKey, \$dimID)

Raised only for rows not related to the last dimension and when group(s) are declared but current row don't trigger a group change. (Row has the same group values than previous row.) In most cases this is an unexpected behaviour and you might want to trigger an error. That's also the default behaviour.

But sometimes it's deliberate (e.g. From a date field only the year or month is declared as a group) and you want to handle this non unique rows.

param array/object \$row The current row which triggered the group change.

param mixed \$rowKey The rowKey is the key of the current row taken from the input data set or given by calling the next() method.

param int \$dimID The ID of the current data dimension.

PROTOTYPING

Before you start writing any code you might want to use the prototyping system to generate a report which shows some data of the currently processed row, names of methods which will be called in real life applications, the value of group fields and some values out of the aggregated fields.

Prototype tells also what the real action would be (e.g. Call method xy or append string 'foobarbaz').

It's also a good idea to use prototyping before you start tracing or debugging your application.

You can call the prototype function at any time by just calling

```
$rep->prototype();
```

This will return an html table for the current action key.

The other way is setting the call method parameter by calling the setCallAction() method with one of the following constants as parameter.

CALL_EXISTING = 0 Call methods in owner class only when implemented. Default.

CALL_ALWAYS = 1 Call also not existing methods in owner class. This allows using magic function calls.

CALL_PROTOTYPE = 2 Call prototype for methods not implemented in owner class. Very useful for incremental developing of reports.

CALL_ALWAYS_PROTOTYPE = 3 Call always prototype even when method exists in owner class.

CALL_ALL_PROTOTYPE = 4 Call prototype for all actions which are not callables and action is not false.

Usually the method is called once before calling the run() method. But it is also possible to alter the call action at any time.

```
// $rep has a reference to phpReport object
$rep->setCallAction(Report::CALL_EXISTING);
$rep->setCallAction(Report::CALL_ALWAYS);
$rep->setCallAction(Report::CALL_PROTOTYPE);
$rep->setCallAction(Report::CALL_ALWAYS_PROTOTYPE);
```

The prototyping class is a good example how flexible a report can be.

CONFIGURATION

The purpose of configuration is to map named events to actions. Most likely an action is a method to be called in the application object but can be anything else. The `config.php` file is the place to override system wide default action types and action method names to meet your naming conventions.

phpReport lets you configure the real action to be performed for events which triggers one or more actions.

Whenever an event occurs the action defined to an action key will be performed.

For each possible action to be executed you can declare what should be done. Each action key has a default action assigned. Within the **config.php** file the default actions might be adjusted to meet your personal favorites or to follow business rules of your organisation.

Table 1: Action types

Action typ	Action defined by
Call a method in target class	Action is a legal method name
Call a method in any class	Action is an array ([class, method])
Ouput a string	Action is not a legal method name or begins with an :
Throw an error	Action begins with error:
Raise a warning	Action begins with warning:
Do nothing	Action equals false

The following table lists all possible actions, assigned defaults and notices how the % sign the will be replaced during run time.

Table 2: Action mapping

action key	default value	% replaced by
init	init	
close	close	
totalHeader	%Header	\$grandTotalName
totalFooter	%Footer	\$grandTotalName
groupHeader	%Header	group name or group level
groupFooter	%Footer	group name or group level
detail	detail	
noData	 No data found 	
noData_n	noDataDim%	dimension id
noGroupChange	error:Current row in dimension % didn't trigger a group change.	dimension id

Next to the actions parameter you can also declare the following parameters:

property grandTotalName

The name for the grand total group. Will also be used to build actions for action keys totalHeader and totalFooter.

Note: You can always access grand totals by group level of 0.

property buildMethodsByGroupName

When true the % sign actions related to groupHeader and groupFooter will be replaced by the group names. When false by the numeric group level.

That's the place where you define the type of actions to be performed or the method names.

Actions have a key which will translated to real action names.

Per default the action 'init' will call a method called 'init'. That's fairly simple. The same is true also for the 'detail' and 'close' actions.

The default action 'totalHeader' is '%Header'. The % sign will be replaced by the value of the 'grandTotalName'. Assuming 'grandTotalName' is unchanged and has the value of 'total' the action to be performed is the 'totalHeader' method. The same rule applies to the 'totalFooter' action.

Similar rules are true for the 'groupHeader' and 'groupFooter' actions. But as we need an action for each defined group the % sign will be replaced by the group name. Example: You defined group changes for 'region', 'customer' and 'invoice'.

region regionHeader regionFooter customer customerHeader customerFooter invoi invoiceHeader and invoiceFooter

You like it the other way. Then name the groupHeader action 'head_%' and footerAction 'foot_%'.

region head_region foot_region customer head_customer foot_customer invoice head_invoice foot_invoice

The same rule with 'buildMethodsByGroupName' set to 'ucfirst' will result in these actions: region head_Region foot_Region customer head_Customer foot_Customer invoice head_Invoice foot_Invoice

The rules above are very simple to follow so that's nothing really sophisticated.

Setting the 'buildMethodsByGroupName' set to 'false' will replace the % sign by the group level. The last example would call those methods

region head_1 foot_1 customer head_2 foot_2 invoice head_3 foot_3

What might look very strange at the first sight is very powerful when you want to write very flexible apps. Changing the second group from 'customer' to 'branch' doesn't change the action group names. While knowing that the group value is passed to the method as a parameter and that you have easy access to the group names gives you nearly unlimited choices.

You are not limited to declare how method names are build.

For some actions it is suitable to define a string. Then the string will be appended to the \$output variable. A good example is the 'noData' action. In many cases you won't instantiate **phpReport** when your data query doesn't return any data. But you might also in this cases a report with nice header and footer and in between just print a message like "Sorry, we couldn't find any data". So it's not worth to create a method in each report which returns such a string. The solution is to declare this string as a default. A : sign at the beginning makes sure that this string is always treated as a string and can not be mixed up with a method name.

The % sign in 'noData_n' actions will be replaced by the number of current dimension when the current dimension is greater than 0. See multi dimensional data for more details.

The very last parameter is called 'userConfig'. This is an optional way how you can pass data around. **phpReport** itself doesn't use this values.

Note: All directives can be altered when initializing a new **phpReport**. Some even when calling a method.

DATA INPUT

Data input is completely decoupled from your application. Use your own data access methods, data models and components as well as data access features from any PHP framework or ORM (Object Relation Mapper).

phpReport accepts data rows being an array, an object or even a string. These data rows can be passed to phpReport all at once, row by row or in batches of any size. This gives greatest flexibility and more control over memory usage when working with large amount of data.

It also allows complete separation between a controller class and the application. The controller might feed the report object with the required data.

Choose the access strategy which seems most suitable for your current application and change this strategy on demand without touching the application.

Between reading and feeding data to phpReport you can modify or filter the input. So it's easy to work with any data format (like csv files, excel sheets and json strings).

phpReport accepts data in three different ways.

- **Passing all data within an iterable to the run() method.** **phpReport** iterates over the data set and calls the next() method for each entry.
- **Passing chunks of data within an iterable by calling the run() method** for each chunk while setting the parameter *finalize to false* or by calling the nextSet method. **phpReport** iterates over the data set and calls the next() method for each entry. To finalize the job either call the end() method after the last chunk or set the finalize parameter for the last chunk to true.
- **Passing single rows by calling the next() method for each data row.** **This** is in many cases the most efficient way as you don't have to collect your data into an array. To finalize the job just call the end() method after processing the last row.

6.1 Joining Data

Joining data in phpReport can be used for different purposes.

First you can join any data row with any other source. Data sources don't have to be the same kind. So you can for example easily combine a row from a database with rows from an excel sheet. Use the same data input methods for joined data as for the primary data.

Another way to join data comes into place when your data row is a data model. This model usually has methods or properties providing related data. Declare the relationship by calling the join() method and phpReport will iterate over these related data.

To iterate over an multi-dimensional array the join() method is used to declare which array element holds the next dimension.

To the application joining data is largely invisible. Grouping and computing values behave like data would have been served as a flat record.

join(\$value, \$noDataAction, \$dataAction, \$noGroupChangeAction, ...\$params): Report

Parameters

- **\$value** (*mixed*) – The source for the joined data. When source is a callable just return the whole the data set or return false and call the nextSet() method or the next() method for each data row.
- **\$noDataAction** (*mixed*) – The action to be executed when \$source doesn't return any data.
- **\$dataAction** (*mixed*) – The action to be executed for each row returned by \$source.
- **\$noGroupChangeAction** (*mixed*) – The action to be executed when two consecutive rows don't trigger a group change.
- **\$params** (*mixed*) – Variadic parameters to be passed to \$source.

DATA OUTPUT

The basic idea behind data input is also true for data output. **phpReport** doesn't create any output itself.

Actions described before are the places where you can build your output. Using external components like mPDF, TCPDF, phpSpreadsheet, PHPWord will help to format the output as you like.

phpReport assists you collecting your output by appending the return values of the action methods or defined action strings to the `$output` property.

`$output` is used for nothing else. So you can

- Alter the content
- Write content to disc
- Delete content
- Append data to content
- ...

Your action methods don't need to return anything. You can also keep desired output in any other variable.

Note: `$content` will be returned from the `run()` and the `end()` method.

GROUP CHANGES

Complexity in applications usually grows exceptionally with every additional group which needs to be managed.

With **phpReport** you only need to call the `group()` method once for every group. Groups can be controlled in every data dimension. Just call the `group()` method after calling the related `data()` method.

A group change occurs when within a data dimension when group values of the previous row don't equal those of the current row.

Tip: Data don't need to be sorted by groups. But make sure that rows are grouped by the group field or you might raise unwanted group changes.

Once a group change has been detected the appropriate action methods (group headers and group footers) will be executed.

group (*\$name*, *\$value* = null, *\$headerAction* = null, *\$footerAction* = null, ...*params*)
Declare a data group.

Parameters

- **\$name** (*string*) – The name to be used for this group. This name will be used to build method names for group headers and footers (depending on configuration parameters). Must be unique between all dimensions. All group related values (including cumulated values from sum or sheet methods as well as row and group counters) can be retrieved by this group name or the group level.
- **\$value** (*mixed*) – Source of the group value. Use the attribute name when data row is an object or the key name when data row is an array. It's also possible to use a callable (a closure or an array having class and method parameters) expecting `$row` and `$rowKey` as parameters. When the `$value` parameter is null it defaults to the content of `$name` parameter.
- **\$headerAction** (*mixed*) – Set individual group header action. Null to execute the default action. False to deny any action.
- **\$footerAction** (*mixed*) – Set individual group footer action. Null to execute the default action. False to deny any action.
- **\$params** (*mixed*) – Variadic parameters to be passed to callables declared with value parameter.

Returns `$this` which allows method call chaining.

Example

```
$rep = (new Report ($this))
->group ('region')
->group ('year', fn($row) => substr($row->saleDate, 0, 4))
->group ('month', fn($row) => substr($row->saleDate, 5, 2))
->group ('customer', 'customerID', null, '</table>')
```

The above example declares four groups (group level 1 to 4) to be monitored. Instead of executing the customer footer action the string ‘</table>’ will be appended to the \$output property.

Note: Group level 0 is always the top level which is called grandTotal.

For each declared group a group counter will be instantiated and incremented when a group change occurs. See ‘Calculation’ for details.

AGGREGATION

9.1 Collector

A collector class is designed to hold and manage multiple items. An item can be an other collector or an calculator object.

9.2 Default Collectors

phpReport has a very powerful feature to aggregate values and increment counters. Each value and counter is implemented as a calculator object. These objects will be assigned to one collector.

phpReport instantiates the following collectors.

Note: All entities assigned to one of those collectors are cumulated to higher group levels.

9.2.1 Row counter collector

The row counter collector is named **rc**. For each data dimension one CalculatorXS object will be instantiated.

9.2.2 Group counter collector

The group counter collector is named **gc**. For each defined group one CalculatorXS object will be instantiated.

9.2.3 Total collector

The total collector is named **total**. The collector is used to hold all calculator objects instantiated by the compute() method and the sheet objects instantiated by the sheet() method.

You can assign further collectors, sheets or calculators to this total collector or any other collector in this tree. So it's possible to build a hirachichal structure of aggregated values.

9.3 Row counter

phpReport always counts all read data rows.

For each data dimension one CalculatorXS object will be instantiated as an item within an collector named **rc**.

Using the same objects for incrementing counters as calculating any values provides you the same methods for accessing the results. This includes also that row counters are available at each group level.

Note: Only the last data dimension has counters for all group levels. First data dimensions serve counters down to the group(s) declared in it.

```
$this->rep = new Report();  
//  
// Some code here ...  
//  
$this->rep->rc->items[*dimensionID*]->sum(*level*);
```

Example:

```
$this->rep = new Report();  
$rc = $this->rep->rc;  
// sum of rows read in all data dimensions at the current group level  
$rc->sum();  
// sum of rows read in all data dimensions at group level = 2  
$rc->sum(2);  
// sum of rows read in data dimensions 0 at current group level  
$rep->rc->{0}->sum();  
$rep->rc->items[0]->sum(); // That's the same  
// sum of rows read in data dimensions 1 at group level = 2  
$rep->rc->{1}->sum(2);  
$rep->rc->items[1]->sum(2); // That's the same
```

9.4 Group counter

As soon as you declare a group **phpReport** counts how often this group occurs in the higher group level. In other words we count how many different values of a certain group has been processed.

For each group one CalculatorXS object will be instantiated as an item within an collector named **gc**.

Group counters are, like any other aggregation methods, available at each group level.

Attention: Make sure you access the correct gc item. To get the counter of the current group use the item of the previous group.

Tip: Use a negative value for the group level. -1 means the level above the current level.

```
$this->rep = new Report();  
//  
// Some code here ...  
//  
$this->rep->gc->items[*group*]->sum(*level*);
```

Example:

```

$this->rep = new Report();
$gc = $this->rep->gc;
// Sum of group in the group above the current group.
$gc->sum(-1);
// sum of group data dimensions at group level = 2
$rc->sum(2);
// sum of rows read in data dimensions 0 at current group level
$rep->rc->{0}->sum();
$rep->rc->items[0]->sum(); // That's the same
// sum of rows read in data dimensions 1 at group level = 2
$rep->rc->{1}->sum(2);
$rep->rc->items[1]->sum(2); // That's the same

```

The main responsibility of a collector is to call methods in assigned items. To make sure that the computation of calculated values works correct calculator objects must be registered to the **total** collector or to one of his child collectors.

The collector class has the `ArrayAccess` interface and the magic `__get` method implemented which allows a broad range of access options.

The visibility of the `$items` array is public. This allows maximum speed when accessing an item and gives the opportunity to apply all php array methods.

9.4.1 Add items

Items will be added (or registered) to a collector usually by calling the `calculate()` or `sheet()` methods of the **phpReport** class.

You may also add items by the `addItem()` method. This is especially desired when you want to group multiple item objects.

Adding items to a sheet collector is hidden and will be internally handled based on key values of the `add()` method.

9.4.2 Alternate item keys

Items are stored in the `$item` array indexed by the name given when calling the `addItem()` method. You can also apply an alternate key to allow accessing an item by the alternate key as well.

The `setAltKeys()` method will set many alternate keys while the `setAltKey()` method one alternate key.

The group counter collector uses alternate keys to allow accessing the counters by the group level and by the group name.

9.4.3 Get items

To access an item you can use the `getItems()` method for all items or the `getItem()` method to get one item.

The magic `__get()` and the array access `get` methods call the `getItem()` method. The item will be returned when it exists either by the item array key or by the alternate key.

```

$this->rep = new phpReport($this);
$this->rep->compute('sales');
// All of the following statements will return the same item.
$item = $this->report->total->getItem('sales');
$item = $this->report->total->items['sales'];

```

(continues on next page)

(continued from previous page)

```
$item = $this->report->total->sales;  
$item = $this->report->total['sales'];
```

9.4.4 Aggregate methods

All aggregate methods implemented in the calculator classes have their counterpart in the collector classes.

The collector aggregate methods call the same methods for each assigned item. Returned results will be returned either as an array indexed by the item key or as a scalar aggregated value.

9.4.5 Subset of items

To apply aggregate methods only on some items you can build a subset by calling the following filter methods. Each of them will return a cloned collector object with just the filtered items.

To use the cloned collector multiple times hold the reference to the cloned collector in a variable.

range(...\$ranges): AbstractCollector

Extract ranges of items.

Returns ranges of items located between start and end keys.

When a range is an array value1 is the start and value2 the end key. When one of the keys doesn't exist the value of the altKey will be used instead. When the items still doesn't exist an error will be thrown.

If start key equals Null the range begins at the first item. When the end key equals Null the range ends at the last item.

When a range is not an array then the item with the corresponding key or altKey is returned if it exists. If this doesn't exist php raises a notice.

Item keys are preserved. Sort order within ranges is preserved. Ranges are returned in given order. When items belong to multiple ranges only the first occurrence will be returned.

:param array<int|string[]> \$ranges Ranges or item keys for items to be filtered. :return AbstractCollector Cloned collector with items in ranges. :throws InvalidArgumentException When start or end item doesn't exist.

between(...\$ranges): AbstractCollector

Filters items where key is between values.

Iterates over each collector item. If a range is an array and the item key is between value1 and value2 of this range (inclusive) the item is returned.

If the range isn't an array the item with the corresponding key is returned.

If a range matches the key of a named range then the named range value will be used to filter the items.

Item keys and sort order are preserved.

Parameters

- **array|int|string[]** – \$ranges Ranges or item keys for items to be filtered.

Returns AbstractCollector Cloned collector with items in ranges.

filter(callable \$callable):

Filters items using a callback function. Iterates over each item in the array passing key and value to the callback function. If the callback function returns TRUE, the current item is returned into the cloned collector. Item keys are preserved.

Parameters

- **\$callable** (*callable*) – The callback function to use.

Returns AbstractCollector Clone of current collector with filtered items

cmd(callable \$command, ...\$params): AbstractCollector

Alter item collection by executing a php array command.

Parameters

- **\$command** (*callable*) – Any php array command which accepts an array as the first parameter.
- **\$params** (*mixed[]*) – Additional parameters passed to the php command.

Returns AbstractCollector Clone of current collector with applied command on the items array.

9.4.6 More specific groups

To build even more specific groups of calculators instantiate a new collector and add any items (calculator or collector) to this collector. To make sure that the values within the calculators are properly set use only already instanted calculators.

9.5 Calculator

phpReport comes with 3 different calculator classes for calculation with numeric values and 2 classes for calculation using bcmath functions.

Each set of those calculators offer different functionality while classes which names end with 'XS' offers basic functions. Those which names ends with 'XL' offers the maximum functions and without and suffix the medium set of functions.

The classes provided are:

- CalculatorXS (default)
- Calculator
- CalculatorXL

or with bcmath

- CalculatorBcmXS
- CalculatorBcm
- CalculatorBcmXL

The CalculatorXS class has the minimum functionality. It's perfect to cumulate any value or to increment any counter. To increment a value the inc() method does the same as calling 'add(1)' method.

The Calculator class don't have the increment method but counts the not null and not zero values.

The CalculatorXL class extends the Calculator class and detects the minimum and maximum values of an attribute. Use the min() and max() methods to get these values.

Note: The bcm classes are working with the scale parameter. Scale works exactly like in the original bcmath functions. So digits after the scale parameter will be truncated (same as the floor function).

The compute() method instantiates a calculator object which provides aggregate functions. The sheet() method instantiates a sheet collector which holds many calculator objects.

Example

```
$rep = (new Report ($this))
->data('object')
->compute ('amount')
->compute ('price', fn($row, $rowKey) => $row->amount * $row->pricePerUnit);
```

9.6 Sheet

Aggregate values in an tabular form like in a spreadsheet.

Sometimes a sheet can eliminate the need for declaring a group or a data dimension. You can also aggregate the same value in different sheets so that results are grouped by different keys.

The sheet() method instantiates a Sheet or a FixedSheet object. Both of them are special variants of the collector object and will be assigned to the total collector (\$total).

For each column in a sheet a calculator object will be instantiated and linked via the \$items array property.

Note: All calculator objects within a Sheet or FixedSheet object are of the same type.

Group levels are like sheet rows and data keys like sheet columns.

sheet (*string \$name, \$value, \$headerAction = null, \$footerAction = null, ...\$params*)

Aggregate attributes in a sheet.

Sheet is a collection of calculators for a horizontal representation of a value. Call this method once for each sheet.

Parameters

- **\$name** (*string*) – Unique name to reference the sheet object. The reference will be hold in \$this->total.
- **\$value** (*mixed*) – Source of the key and value to be aggregated. Must be served in an array with only one entry were key is the array key and value the value.

Key and attribute name are attribute names when data row is an object or or when row is an array the element keys. It's also possiblbe to use a closure which returns an array [key => value]. False to just instantiate and reference the sheet. To execute the calculation call the add() method of the sheet object. This is very useful when getting the key or value to be calculated is complicated and / or you need these data on the detail level.

Tip: Using the array_column function might declaring the latest data dimension redundant.

- **\$typ** (*int | null*) – The calculator type. Typ is used to choose between a calculator class. Options are XS, REGULAR and XL. Defaults to XS. Typ belongs to all sheet items.
- **\$fromKey** (*mixed*) – To use a fixed sheet declare the first calculator name. Pass an array when sheet names are not in an sequence. Example: ['young', 'mid-aged', 'old'] Null for sheets where calculators are instantiated for each key value.
- **\$toKey** (*mixed*) – The last calculator name for fixed sheet. FromKey will be icrementd until \$toKey is reached.

- **\$maxLevel** (*int / null*) – The group level at which the value will be added. Defaults to the maximum level of the dimension. Might be less when aggregated data are only needed on higher levels.
- **\$params** (*mixed*) – Variadic parameters to be passed to callables declared with value parameter.

Returns \$this which allows method call chaining.

9.6.1 FixedSheet class

FixedSheet classes are best used when you know which columns you need and want them to be instantiated all at once. To do so pass an array of column names or the starting and ending name to the sheet() method. When starting name is a string make sure that the ending name can be reached by incrementing the starting name.

```
$rep = new Report ($this);
// Declare season names as columns
$rep->sheet ('sales', ['regionID' => 'amount'], ['spring', 'summer', 'autumn', 'winter'
↪])

// Declare columns 1 to 12 (e.g. to represent month)
$rep->sheet ('sales', ['regionID' => 'amount'], 1, 12)

// Declare columns a, b, c, d, e and f
$rep->sheet ('sales', ['regionID' => 'amount'], 'a', 'f')
```

When you try to add a value to an not existing column an exception will be thrown.

9.6.2 Sheet class

Sheet classes don't have a fixed number of columns. Columns will be instantiated whenever a data row delivers a new column name.

If you need columns in a sorted order you should sort your data accordingly. If this is not suitable ksort the \$items property of the sheet object.

```
$rep = (new Report ($this))
->data('array')
->sheet ('sales', ['regionID'=> 'amount'])
```

phpReport manages aggregation by using calculator classes organized in collector classes.

Three different calculator classes provides different sets of aggregation methods. Sheets are special variants of an collector. They are used to hold multiple calculator objects of the same type.

GETTING VALUES

All initialisation methods have parameters called **\$source** and **\$params** to declare how related actions like evaluating a group change will get the required data.

The parameter *\$params* may hold variadic arguments which will be passed unpacked (after \$row and \$rowKey) to callables specified in *\$source*.

Specialized getter classes are available to get the requested values. You may pass a getter object of your choice to the source parameter or let **phpReport** choose the right one. The latter option is usually the easiest one.

Source might get a scalar value or an array with up to three parameters where scalar values might be:

- An array key or property name depending of the type of data data row.
- A closure which expects \$row and \$rowKey as the first two parameters.
- A getter object which implements the GetValueInterface.

For all other sources PHP has to offer *\$source* must be an array with up to 3 elements.

Table 1: Source array parameters

Key	Content
0	The object member name or a closure.
1	The class or object where the object member belongs to. <ul style="list-style-type: none"> • Null - The current row object (default) • True - The default target (Value of the target parameter of report class). • Object or class name. Object might also be used to access a constant.
2	Kind of object member <ul style="list-style-type: none"> • Null - Sets the most logical defaults. For closures same as <i>False</i> (in contrast to the closure as a scalar source). Object property when key[1] is not null. Method of the current row object. • Bool - Name is a callable (method, static method or closure). <i>True</i> will pass \$row and \$rowKey as the first parameters to this callable. <i>False</i> will not pass \$row and \$rowKey to this callable. • stat - Name is a static class property. • const - Name is a class constant.

For method chaining use closures, method calls or dedicated getter objects.

The following examples shows all combinations:

Table 2: Source parameter examples

Source	Description
'abc' 'a b c' 2	Property of row object or array item.
['abc', true]	Property in default target.
['abc', \$object]	Property in \$object.
<pre>fn(\$r, \$k) => (...) fn(\$r, \$k, ... \$params) => (...) [fn(\$r, \$k) => (...), null, true] [fn(\$r, \$k, ... \$p) => (...), null, true]</pre>	Closure getting \$row and \$rowKey.
<pre>[fn(... \$params) => (...)] [fn(... \$params) => (...), null, false]</pre>	Closure not getting \$row and \$rowKey
['abc'] ['abc', null] ['abc', null, null] ['abc', null, false]	Method of current row object.
['abc', null, true]	Static method in current row class getting \$row and \$rowKey.
['abc', true, false] ['abc', true, true]	Method or static method in default target. Same as before but getting \$row and \$rowKey.
['abc', \$obj, false] ['abc', 'xyzClass', false] ['abc', \$obj, true] ['abc', 'xyzClass', true]	Method or static method in \$obj object. Static method in 'xyzClass' class. (Static) method in \$obj getting \$row and \$rowKey. Static method in 'xyzClass' class getting \$row and \$rowKey.

['abc', null, 'stat']
 ['abc', true, 'stat']

Static property from row class,
 default target,
 \$obj object,
 'xyzClass'

METHODS RETURNING INFORMATION

The report class has some extra convenience methods to provide you with useful information.

class Report

The report class offers a lot of methods which delivers almost all information you need to create your application.

getRow (*int \$dimID = null*)

Get the active row for the requested dimension.

Parameters

- **\$dimID** (*int | null*) – The data dimension for which you want the current row. Defaults to the current data dimension. If \$dimID is negative the value will be subtracted from the current data dimension.

Returns The active data row for the requested dimension.

getRowKey (*\$dimID*)

Get key of the active row for the requested dimension.

Parameters

- **\$dimID** (*int | null*) – Same as in getRow().

Returns The requested key.

isFirst (*\$level = null*): **bool**

Checks if the current group is the first one within the next higher group. e.g. Is it the first invoice for a customer.

Parameters

- **\$level** (*string | int | null*) – The group level to be checked. Defaults to the next higher group level.

Returns True when the current group is the first one within the given level. False when not.

isLast (*\$level = null*): **bool**

Check if the current group footer action is executed the last time within the group level. The question can only be answered in group footers.

Parameters

- **\$level** (*string | int | null*) – The group level to be checked. Defaults to the next higher group level.

Returns True when the current action is executed the last time within the given group level. False when not.

Throws `InvalidArgumentException` when method is not called in a group footer or asked for group levels not higher than the current one.

getLevel (*\$groupName = null*): **int**

Get the current group level or the level associated with the group name.

Parameters

- **\$groupName** (*string/null*) – The name of the group. Null for the current group level.

Returns The requested group level

getGroupValues(*?int \$dimID = null, bool \$fromFirstLevel = true*): **array**

Get all active group values. Note that in footer methods the row which triggered the group change is not yet active.

Parameters

- **\$dimID** (*int/null*) – The dimension id for / till the group values will be returned. Defaults to the current dimension id.
- **\$fromFirstLevel** (*bool*) – When true all group values from the first dimension to the requested dimension are returned. When false only the group values of the requested dimension are returned.

Returns Array with requested group values indexed by group level.

getGroupValue (*\$group = null*)

Get the current value for the requested group.

Parameters

- **\$group** (*int/null/string*) – String representing the group name or integer representing the group level. When null it defaults to the current group level. Negative values are subtracted from the current level.

Returns Current value of the requested group.

getGroupNames(): **array**

Get all group names.

Returns array Array of all group names.

getGroupName(*int \$groupLevel*): **string**

Get the associated group name of the group level.

Parameters

- **\$groupLevel** (*int*) – The level of the group.

Returns string The associated group name of the level.

getDimId(*mixed \$level*): **int**

Get the dimension ID for a given group level.

Parameters

- **\$groupLevel** (*mixed*) – The level of the group. Defaults to the current level.

Returns int The dimension id.

OVERVIEW OF METHODS AND PROPERTIES

12.1 Methods to instantiate report class

`data` Describe data input

`group` Declare a group to monitor changes between data rows

`aggregate` Declare variable to provide aggregate functions (sum, count, min, max)

`sheet` Declare variable to be aggregated horizontally (having key and value)

12.2 Methods for data handling

`run` Start execution with data

`runPartial` Iterate over a set of data

`next` Take a single data row

`end` Finalize execution

12.3 Action methods called from report class

`init` First called method to initialize application

`close` Last called method to clean up the dishes independent from `__destruct` method.

`totalHeader` Called once after `init()` to build the total header page of the report.

`totalFooter` Called once before `close()` to build the total footer page of the report.

`groupHeader` Called for each new group value(s)

`groupFooter` Called after `detail()` but before activating new group value(s).

`detail` Called for each data row in last data dimension.

`noData` Called when no data was given.

`noData_n` Called when no data was given for dimension 'n'.

`noGroupChange_n` Called when groups for dimension 'n' are declared but row didn't trigger a group change.

12.4 Methods returning information

`getRow` Get the active row for the requested dimension.
`getRowKey` Get the key of active row for the requested dimension.
`getGroupNames` Get names for all declared groups.
`getGroupName` Get name for a requested or current group level.
`getGroupValues` Get current values for all declared groups.
`getGroupValue` Get current value for the requested or current group.
`getLevel` Get the current group level or the level associated with the group name.
`getChangedLevel` Get the level which triggered the group change.
`getDimID` Get the dimension id related to a group level or the current dimension id.
`isFirst` Bool if the action for the current or given level called the first time.
`isLast` Bool if the action for the current or given level called the last time.

12.5 Public Properties

\$out Output object holding the output
\$gc Group count collector
\$rc Row count collector
\$total Collector for calculators, sheets and collectors
\$userConfig Configuration parameter given during instantiation

12.6 Prototyping methods

`prototype` Call prototype method related to current action. `setCallAction` Alter targets for actions to be executed.

INDICES AND TABLES

- `genindex`
- `modindex`
- `search`

Symbols

(*property*), [16](#)
() (*method*), [23](#), [30](#)

G

getGroupValue() (*Report method*), [36](#)
getRow() (*Report method*), [35](#)
getRowKey() (*Report method*), [35](#)

R

Report (*class*), [35](#)