phpReport

Release 2.1.0

Günter Pöhl

May 25, 2021

TABLE OF CONTENTS

Edition for phpReport 2.0

by Günter Pöhl

CHAPTER

ONE

INTRODUCTION

phpReport is designed to be the backbone for all applications needing control over group changes or when calculation of totals and subtotals is required. This is usually the case for reports but there are a lot of other use cases.

phpReport integrates into any framework without configuration. You'll never have do extend your classes to use **phpReport**. So your application can still extend from any framework class.

Use of **phpReport** is very simple even when you work with very complex data structures.

Data retrieval is done by your own (framework) methods. **phpReport** will work with this data no matter if you provide a data set or an multi dimensional array. Joining data, even between different sources, is also easy to accomplish.

1.1 Getting started

After instantiation of the report class call the data() method to specifiy which the data handler to be used. The data handler is responsible to return values from a data row. Out of the box there are an ArrayDataHandler and an ObjectDataHandler. You can also use the 'array' or 'object' aliases.

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

Use the aggregate() 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;

    /**
    * It might also be a good idea to instantiate the Report in your
```

(continues on next page)

(continued from previous page)

```
     * controller.
     */
    public function __construct($data){
        // initialize report
        $this->rep = (new Report($this))
        ->group('customer')
        ->group('invoice', 'invoiceID')
        ->compute('sales', fn($row) => $row->amount * $row->price);
        // Start execution. $data is an iterable having some data rows
        $this->rep->run($data);
    }

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

    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, $rowKey){
        return "<br>$row->item: $row->description";
    }

    // $row in footer methods is the previous row, not the one which triggered the␣
↪group change.
    public function invoiceFooter($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
            "Total sales = $this->rep->total->sales->sum()" .
            "Total number of customers: $this->rep->gc->customer->sum()" .
            "Total number of invoices:  $this->rep->gc->invoice->sum()" .
            "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 the 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 of 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 getting related data to a given row. See data section for details. Out of the box phpReport offers row counters.

Aggregating values With phpReport it's easy to aggregate values. While calling the aggregate method your values are cumulated. Your might also let phpReport count how often you got a not null or not zero value as well as figure out the min and max value.

Sheets Sheets are a very powerful to aggregate values horizontally. Assume you want to present your calculated data in a table grouped by month. All you need to do is calling the sheet method and tell where to find the key (month) and where to find the value.

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 for each group. Out of the box phpReport offers group counters which lets you know how often a certain value (or group) occurs in an other group.

Prototyping Beginners and experienced users of phpReport can benefit from the prototype system. Prototying lets you know which method would habe 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)

CHAPTER

TWO

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 exectued.

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 prototyp 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 an individual groupHeader 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().

2.1 Named events

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

2.1.1 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.

2.1.2 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.

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

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.

param int $dimID The current data dimension. Initial data dimension equals 0.

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.

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.

2.1.3 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)

Raised when for a data dimension group(s) are declared but current row has the same group values than previous row. In a well designed data model this should not happen. If you can't change the model consider what you have to do in such situations. Ignoring this case, trigger a warning or throw an exception are valid options.

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.

CHAPTER

THREE

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 methods in prototype class when they are not implemented in owner class. Very useful for incremental developing of reports.

CALL_ALWAYS_PROTOTYPE = 3 Call methods in prototype class for any action.

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 prototying class is a good example how flexible a report can be.

CHAPTER

FOUR

CONFIGURATION

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 orgisation.

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 alse for the 'detail' and 'close' actions.

The deaufault 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 sopisticated.

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 sigth 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 parmeter and that you have easy access to the group names gives you nearly unlimited choices.

Your 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 appende to the $output variable. A good example is the 'noData' action. In many cases you won't instantiate **phpReport** when your data query doen't return any data. But you might also in this cases a report with nice header and footer and in between just printint 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** isself does't use this values.

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

CHAPTER

FIVE

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, on 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.

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 efficent 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.

5.1 Joining Data

Joining data in phpReport can be used for different pupurses.

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.

CHAPTER

SIX

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, tcpfd, phpSpredsheet, 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.

CHAPTER

SEVEN

GROUP CHANGES

Compexity in applications usually grows exceptionlly with every addidtional 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 appropiate 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 possiblbe 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 execte the default action. False to deny any action.
- **$footerAction** (*mixed*) – Set individual group footer action. Null to execte the default action. False to deny any action.
- **$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.

CHAPTER

EIGHT

AGGREGATION

8.1 Collector

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

The main responsibilty of an collector is to call methods in assigned items. To make sure that the cumputation 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.

8.1.1 Add items

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

You may can 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.

8.1.2 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 conter collecor uses alternate keys to allow accessing the counters by the group level and by the group name.

8.1.3 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 calls the getItem() method. The item will returned when it exist 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']);
$item = $this->report->total->sales;
$item = $this->report->total['sales'];
```

8.1.4 Aggregate methods

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

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

8.1.5 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): AbstractCollecotor

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 don'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 exist. If this doesn't exist php raise a notice.

Item keys are preserved. Sort order within ranges are preserved. Ranges are returned in given order. When items belong to multiple ranges only the first occurence will 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): AbstractCollecotor

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 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.

8.2 Calculator

phpReport comes with 3 different calculator classes. The reason behind is that only nessessarty operations will be performed.

In most cases you don't need the minimum or maximum value of an attribute. Identifying those values is time consuming. The same is true for the number of rows having a not null or a not zero value.

The classes provided are:

- CalculatorXS (default)
- Calculator
- CalculatorXL

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

The Calculator class don't have the incrememt 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.

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

Example

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

8.3 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 colums.

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'l Null for sheets where calculators are instantiated for each key value.
- **$toKey** (*mixed*) – The last calculator name for fixed sheet. FromKey will be icremented 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.

8.3.1 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.

8.3.2 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 in 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 a different set of aggregation methods. Sheets are special variants of an collector. They are used to hold multiple calculator ojects of the same type.

CHAPTER

NINE

GETTING VALUES

Almost all methods which initializes an **phpReport** applicaton have a parameter called value. Of course you shouldn't pass a real value but the information where to find the value.

The following table shows all options and the differences between objects and arrays.

Table 1: Value sources

value	row is an array	row is an object
string or integer	value of $row[$value]	value of $row->$value
closure	Returned value of closure	Returned value of closure.
Array with one element	Returned value of $value method in target class	Returned value of $value function in row object. Only $params will be passed to the function.
Array with two elements	Returned value of callable. When first element is a classname it is a static call.	Same as for arrays:

When the value source is a closure or a callable the variadic parameters will also passed to the method.

CHAPTER

TEN

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 givel 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 substracted 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.

CHAPTER

ELEVEN

OVERVIEW OF METHODS AND PROPERTIES

11.1 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 functons (sum, count, min, max)

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

11.2 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

11.3 Action methods called from report class

`init` First called method to initiialize 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.

11.4 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.

11.5 Public Properties

$output String with concatenated return values from actions

$gc Group count collector

$rc Row count collector

$total Collector for calculators, sheets and collectors

$userConfig Configuration parameter given during instantiation

11.6 Prototyping methods

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

CHAPTER

TWELVE

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

Symbols

G

R