

# **ExcelReportGenerator Tutorial**

Introduction .....	3
Templates .....	4
1. Property template .....	4
2. Method call template .....	5
3. Data item template.....	6
4. Aggregate function template.....	7
5. System variable template .....	8
6. System function template .....	9
7. Horizontal page break template.....	10
8. Vertical page break template .....	10
Panels.....	11
1. Simple panel .....	12
2. Data panel.....	13
3. Totals panel.....	20
4. Dynamic panel .....	21
Connection and extension .....	27
1. Extension of system functions.....	27
2. Extension of system variables .....	27
3. Extension of DefaultTypeProvider.....	28
4. Extension of DefaultInstanceProvider.....	28
5. Extension of DefaultPropertyValueProvider.....	29
6. Extension of DefaultMethodCallValueProvider .....	30
7. Extension of DefaultDataItemValueProvider.....	31
8. Extension of DefaultTemplateProcessor.....	31
9. Panels parsing settings .....	33

## Introduction

The library **ExcelReportGenerator** allows you to render data to Microsoft Excel by marking the Excel sheets in a specific way. This library is based on templates and panels. The [ClosedXml](#) library is used to interact with **OpenXml** format. There are several types of templates and panels. Let's look at both in more detail.

## Templates

A template is an expression enclosed in some borders. By default, the template borders are braces (customizable), for example, `{p:Name}` is the property output or `{m:GetData()}` is the method invocation, etc. Any template can occupy a cell entirely:

<code>{p:RenderDate}</code>
-----------------------------

or appear in it inside the text:

Report Date: <code>{p:RenderDate}</code> , Company: <code>{m:GetCompany()}</code>
---

If the template is inside the text, the returned value is always cast to string, that is, `ToString()` method is called on this value. If the template occupies a whole cell, `ToString()` method is not called, which means that the cell will contain the value of the type returned by the template. This is particularly relevant for numeric types, **DateTime**, **Boolean** and some other.

Let's look more closely at the types of templates.

### 1. Property template

The property template in its simplest form looks like:

`{p:PropName}`

where **p** (property) denotes a reference to a property, **PropName** is the name of this property (case sensitive). The property can be static or instance and it must be public. Instead of the property name you can also specify a public field.

This template may have a more complex form. In particular, you can combine the address to properties (or fields) through the `"."`. In this case, the template will take the following form:

`{p:Prop1.Prop2.....PropN}`

It means, that the property (or field) value of **Prop1** is firstly obtained, then the property (or field) **Prop2** is called on received value, and so on. At the same time, if any of the properties (or fields) in the chain is **null** (except the last one), the **NullReferenceException** exception is thrown. The chain can have as many properties (or fields) as needed.

By default, the property (or field) is searched in the report type, an instance of which is passed to the constructor of the **DefaultReportGenerator** class or its descendant. However, you can override this behavior by specifying a type before the property name, such as:

`{p:Company:Name}`

In this case, the property (or field) **Name** will be searched in the type **Company** (the type is separated from the property name with the symbol `":"`). By default, the type itself is searched in the executable assembly. If the type you need is in another assembly, you must specify that assembly in a class derived

from **DefaultReportGenerator** class. In this case, the type **Company** will be searched in all assemblies specified in the descendant of **DefaultReportGenerator**. If only one type with such name is found, the property (or field) **Name** will be searched in it. If this property (or field) is found, the template will return its value and put it in the appropriate place on the sheet. If the property does not exist in the class or does not correspond access modifiers (**public**, **instance**, **static**), the **MemberNotFoundException** exception is thrown.

If the property is not static, an attempt to create an instance of this type will be made. By default, in this case, the type must have a default constructor. But you can override this behavior in the descendant of class **DefaultReportGenerator** by providing your own implementation of **IInstanceProvider** interface. This implementation, for example, can receive object instances from any IoC-container. Note, that in the default implementation, the created instance is the **Singleton** object, that is, when members of this type are accessed many times, they will be called on the object created at the first time.

If many types with such name are found, an **InvalidTemplateException** is thrown. In this case, you can adjust the template by specifying a namespace before the type, for example:

`{p:Reports.Common:Company:Name}`

where **Reports.Common** is the namespace, that is separated from the type name with the ":". Individual namespaces are separated with the ".".

In the case when the type is not found, the **TypeNotFoundException** exception will be thrown.

It is worth mentioning here another feature available for the properties and fields specified in the template. If the property or field is **null** then, by default, the template is simply replaced with an empty string. If you want to replace an empty string to more readable value, you can mark this property or field by **NullValue** attribute, passed to it the required value, for example:

```
[NullValue("Smith")]  
public string Signer { get; set; }
```

In this case, if the property **Signer** will be **null**, the value **Smith** is displayed in place of the template. The type of the attribute value is **object**, that is, you can pass any value that is correct for an attribute.

This template can be used in all types of panels (except dynamic) or simply on a sheet. Also you can pass this template to the method call template as a parameter.

## 2. Method call template

The method call template in its simplest form looks like:

`{m:GetData()}`

where **m** (method) denotes a method invocation, **GetData()** is the call of that method itself. The method can be static or instance and it must be public. You can pass to the method as many parameters as you

want, and those parameters can be either static data hard-coded in the template (strings, numbers, etc) or almost any type of templates. The fragment below is a valid method invocation:

```
{m:GetData(p:Code, EN, m:Namespace:GetCompany(m:GetName(sv:Lang)), 56)}
```

From this it can be seen:

- The template borders wrap only outside method invocation, templates inside this method are already written without borders.
- The method can receive as parameters either static data or any other type of templates including calls of other methods (nesting of methods invocation is not limited).
- As in the case of the property template, before method name you can specify the name of the type (including namespace). The type searching and optionally the instance creation are the same as for the property template.
- You do not necessarily to wrap static data by quotation marks even they are strings. When the method is called, an attempt to convert the passed parameters to the types of the corresponding method parameters will be made. If this conversion fails, the appropriate exception will be thrown.

The method is searched by the name (case sensitive) as well as the number and types of parameters. Methods with variable number of arguments (**params**) are not supported. There are cases (especially with methods overloading), when you have to specify the type of the static parameter explicitly in the template. You can do this in the following way:

```
{m:GetData([string]10, [decimal]56.6)}
```

In this case, the type is explicitly specified before the parameter. It is relevant for all primitive types as well as types **String**, **Decimal** and **DateTime**. It should be noted that for strings instead of specifying the type explicitly you can just wrap the parameter by quotation marks, therefore you can write **"10"** instead of **[string]10**.

It is also possible that there is a comma inside the static parameter. By default, commas are treated as parameters separator. To mark the comma as a plain symbol you need to escape it with another comma, i.e. write **",,"**, for example:

```
{m:GetData(p:Code, Hello,, World!)}
```

This template can be used in all types of panels (except dynamic) or simply on a sheet.

### 3. Data item template

The data item template in its simplest form looks like:

```
{di:Amount}
```

where **di** (data item) denotes a reference to the data item, the meaning of **Amount** can be different depending on the type of the data item. It can be a public field or property name (the data item is an instance of a type), a column name (the data item is a **DataRow**), a dictionary key (the data item is the **IDictionary**). If the data item is an instance of a type then it is possible to combine properties or fields through the ".". Since the panels can be nested inside each other, in this template you can access the current data item of the parent panel as follows:

**{di:parent:Code}**

which means that it is necessary to get the field **Code** from the data item of the parent panel. Also if the data item has a primitive type such as **int** or **string**, and you want to output this element itself, you must address to the element as follows:

**{di:di}**

If the data item is an instance of a type, the **NullValue** attribute can be applied to its properties and fields.

This template can be used only inside the panels that have the **DataSource** property (except dynamic). In the totals panel it can be used only in the aggregate function or when referring to the parent panel. Also this template can be passed to the method call template as a parameter.

#### 4. Aggregate function template

The aggregate function template in its simplest form looks like:

**{Sum(di:Amount)}**

where **Sum** is the name of the aggregate function, **di:Amount** is the data item you want to aggregate. If the data item is an instance of a type then it is possible to combine properties or fields through the ".".

The following built-in aggregate functions are supported: **Sum**, **Count**, **Avg**, **Min**, **Max**. Also it is possible to create your own aggregate function. In this case you should specify the function **Custom** and as the second parameter the name of this function. The function itself must be placed in the instance of the report class, which is passed to the constructor of the **DefaultReportGenerator** class or its descendant. The template in this case will take the form:

**{Custom(di:Amount, CustomAggregationFunc)}**

The **CustomAggregationFunc** must have the following signature:

```
public TResult CustomAggregationFunc(TAccumulation accumulation, TItem currentItem, int itemNumber)
```

where **TResult** is the type of the return result, **TAccumulation** is the type of the accumulated result, **TItem** is the type of the data item, **itemNumber** is the current data item number (the first element will have a value of "1"). That is, on each iteration the function receive the accumulated result, the current

data item and the number of the current data item. For example, the calculation of multiplication may look like this:

```
public decimal CustomAggregation(decimal? accumulation, decimal currentItem, int itemNumber)
{
    return (accumulation ?? 1) * currentItem;
}
```

It is also possible to specify the function as the third parameter. This function is fulfilled after aggregation:

`{Custom(di:Amount, CustomAggregationFunc, PostAggregationFunc)}`

where **PostAggregationFunc** is the name of this function. The function must be placed in the instance of the report class, which is passed to the constructor of the **DefaultReportGenerator** class or its descendant. This function must have the following signature:

```
public TResult PostAggregationFunc(TAggregationResult aggregationResult, int itemsCount)
```

where **TResult** is the type of the return result, **TAggregationResult** is the type of the aggregation result, **aggregationResult** is the aggregation result, **itemsCount** is the total items count, for example:

```
public string PostAggregation(decimal result, int itemsCount)
{
    return (result / itemsCount).ToString("P");
}
```

If you want to apply the post-aggregation function to a built-in aggregate function, you can simply omit the second parameter as follows:

`{Min(di:Amount, , PostAggregationFunc)}`

This template can be used only inside the totals panels. Also this template can be passed to the method call template as a parameter.

## 5. System variable template

The system variable template looks like:

`{sv:SheetName}`

where **sv** (system variable) denotes a call of system variable, **SheetName** is the name of this variable (case sensitive). The following predefined system variables are currently available:

- **RenderDate** (DateTime);
- **SheetName** (string);
- **SheetNumber** (int).

It is possible to extend system variables. This functionality will be described in a separate chapter.



This template can be used in all types of panels (except dynamic) or simply on a sheet. Also you can pass this template to the method call template as a parameter.

## 6. System function template

The system function template looks like:

```
{sf:Format(p:value, "C")}
```

where **sf** (system function) denotes the system function call, **Format(p:value, "C")** is the call of that function itself. The following predefined system functions are currently available:

```
public static object GetDictVal(object dictionary, object key)
```

This function allows you to get the value from the dictionary by key. The parameter **dictionary** must implement the **IDictionary** interface. If any of these parameters is **null** or the parameter **dictionary** does not implement the **IDictionary** interface or **key** is not present in the dictionary, the corresponding exception is thrown.

```
public static object TryGetDictVal(object dictionary, object key)
```

This function is completely similar to the function **GetDictVal** except that it never throws an exception. In case of any error, the function returns **null**.

```
public static object GetByIndex(object list, int index)
```

This function allows you to get an item of a list by index. The **list** parameter must implement the **IList** interface. If the **list** parameter is **null** or does not implement the **IList** interface or the index is out of range, an exception is thrown.

```
public static object TryGetByIndex(object list, int index)
```

This function is completely similar to the function **GetByIndex** except that it never throws an exception. In case of any error, the function returns **null**.

```
public static string Format(object input, string format, object formatProvider = null)
```

This function converts the input value to a string with the specified formatting. The parameter **input** must implement the **IFormattable** interface, otherwise, the exception will be thrown. There is also a third optional parameter for the ability to specify a culture. This parameter can receive an object of one of three types:

- **IFormatProvider**. This type is applied as is.
- **string**. In this case, the culture name must be passed as a parameter ("en", "en-US", "fr", etc.). An attempt to create a **CultureInfo** object with this name will be made. If no culture is found, the **CultureNotFoundException** exception is thrown.

- **int**. In this case, the culture code must be passed as a parameter, for example, **9** is the code of "en" culture or **1049** is the code of "ru-RU" culture. Similarly, an attempt to create a **CultureInfo** object with this code will be made. If no culture is found, the **CultureNotFoundException** exception is thrown. It is worth noting that when passing the culture code directly from Excel, you must explicitly specify the type **int** in the template, for example:

`{sf:Format(p:value, "C", [int]1033)}`

It is possible to extend system functions. This functionality will be described in a separate chapter.

This template can be used in all types of panels (except dynamic) or simply on a sheet. Also you can pass this template to the method call template as a parameter.

## 7. Horizontal page break template

The horizontal page break template looks like:

`{HorizPageBreak}`

Use this template to insert the page break after the row that contains the cell with the template. This can be useful when you nest data panels to each other. In this case, each block of data is printed on a separate page.

## 8. Vertical page break template

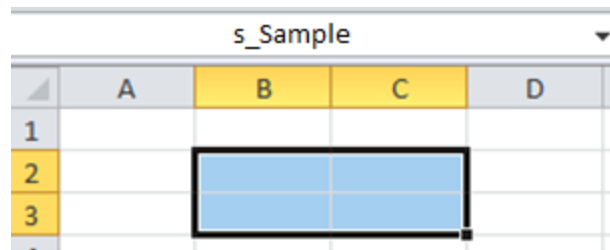
The vertical page break template looks like:

`{VertPageBreak}`

Use this template to insert the page break after the column that contains the cell with the template.

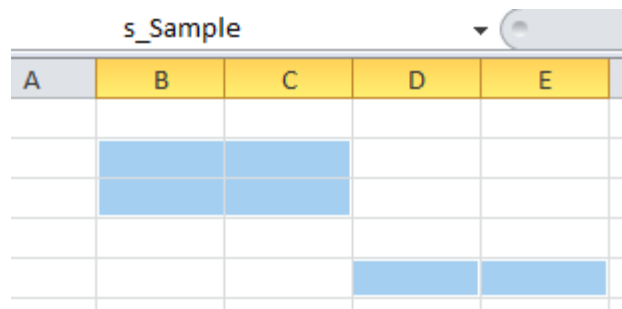
## Panels

Panels are the named ranges of cells in Microsoft Excel that are named in a specific way. For example, the image below shows a simple panel with the name **s\_Sample** and coordinates (**B2, C3**).



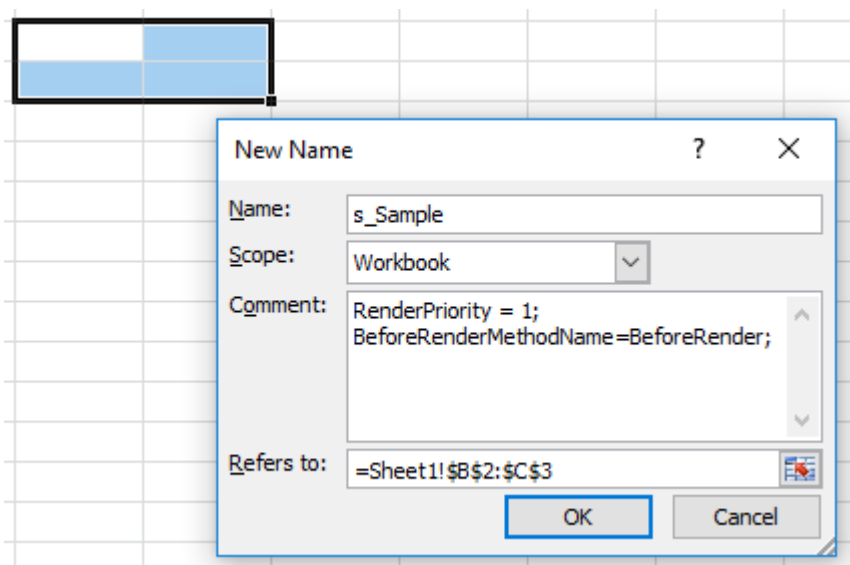
The named range must follow the **<prefix>\_<name>** principle to become a panel, where **<prefix>** is the panel type, **<name>** is the panel name which can be anything. Which prefixes have different types of panels will be described below.

Named ranges in Microsoft Excel can have more than one range of cells. For example, the image below shows the named range **s\_Sample** with coordinates (**B2, C3; D5, E5**).



The **ExcelReportGenerator** library does not support multiple ranges, that is, in this case, only the first range of cells will be processed and the second will be ignored.

When you create a named range, in addition to the name and coordinates, you can also specify an area and a comment, as shown in the figure below:



The area does not matter, but the comment field is necessary to set the properties of panels. By default, properties can be separated either by a newline, a semicolon or a tab character. The property name and value are separated by the "=" symbol. Which properties can have each panel will be discussed in more detail in the appropriate chapters.

Panels can be nested inside each other, that is, they can have a parent-child relationship. In this case, the cell range of the child panel must be within the cell range of the parent panel, and the child panel in the comment box must have the property **ParentPanel** equals to the name of the parent panel.

The sheet itself is also a panel, so templates that are non-specific to any particular panel can be written on the sheet directly. The worksheet panel is the parent of all others and is always treated with the highest priority.

## 1. Simple panel

The simple panel name looks like:

**s\_<name>**

where **s** (simple) denotes that this is the simple panel, **<name>** is the panel name, which can be anything, for example, **s\_ReportHeader** is the correct name for the simple panel. Most often, simple panels should not appear explicitly on the sheet, since these panels are not processed in any special way. All templates are simply displayed as is, and therefore they can be written on the sheet itself, which is also, in fact, a simple panel. This panel makes sense if you want to set some additional properties for it. The following properties are relevant for this panel:

1. **RenderPriority (int)** sets the priority in which the panel is processed. By default, all panels have the same priority of "0", but you can increase it if needed, for example, when one panel is dependent on the other. However, parent panels are always processed before descendants (regardless of the priority of the child). So the sheet panel is always processed first, because it is the parent of all others, and you cannot decrease its priority. That is, this property is not end-to-end, but affects the processing of panels of the same nesting level.
2. **BeforeRenderMethodName (string)** allows you to specify the name of the method that will be executed immediately before panel rendering. The method itself must be placed in an instance of the report class, which is passed to the constructor of the **DefaultReportGenerator** class or its descendant. The signature of this method should look as follows:

```
public void BeforeRender(PanelBeforeRenderEventArgs args)
```

where **BeforeRender** is the method name that you specify in the comments box for the property **BeforeRenderMethodName**, **args** is the method parameter which has the **PanelBeforeRenderEventArgs** type. This type contains the following fields:

- **Range** (**ClosedXML.Excel.IXLRange**) is the range of cells that the panel covers before rendering.
  - **IsCanceled** (**bool**) is a flag that allows you to cancel the panel rendering. If it is set to true inside this method, the panel as well as all its descendants will not be processed by the library.
3. **AfterRenderMethodName** (**string**) allows you to specify the name of the method that will be executed immediately after panel rendering. The method itself must be placed in an instance of the report class, which is passed to the constructor of the **DefaultReportGenerator** class or its descendant. The signature of this method should look as follows:
- ```
public void AfterRender(PanelEventArgs args)
```
- where **AfterRender** is the method name that you specify in the comments box for the property **AfterRenderMethodName**, **args** is the method parameter which has the **PanelEventArgs** type. This type contains the following fields:
- **Range** (**ClosedXML.Excel.IXLRange**) is the range of cells that the panel covers after rendering.
4. **ParentPanel** (**string**) allows you to specify the name of the parent panel. In this case, the cells range of the parent panel must cover the cells range of the child panel, otherwise, an exception will be thrown. It rarely makes sense to make a simple panel nested. The meaning of this field is fully revealed when considering data panels.

This panel can contain all types of templates. However, the data item template will only work correctly if the panel has a data panel as its parent.

## 2. Data panel

The data panel name looks like:

**d\_<name>**

where **d** (data) denotes that this is the data panel, **<name>** is the panel name, which can be anything, for example, **d\_ReportData** is the correct name for the data panel. This panel can contain almost any type of templates, but it is specific for the data item template, which has the form **{di:Field}**, where **Field** can mean different depending on the type of data source. The data item template itself is described in more detail in the relevant chapter. This panel has a required property **DataSource** which must have any template that returns a data source. This panel is displayed on the sheet as many times as there are items in the data source. If the data source does not contain items, the panel will be deleted (deletion occurs with the shift specified in field **ShiftType**). The following types can be used as a data source:

1. **DataTable**. The data item is a **DataRow** object, the name of the column is specified in the template.

2. **DataSet**. The data item is a **DataRow** object, because the first **DataTable** is retrieved from the **DataSet**, therefore, the template specifies the column name of the first table of the **DataSet**.
3. **IDataReader**. The data item is also a **DataRow** object, because behind the scenes the **IDataReader** is read into the **DataTable** and immediately closed. To close a database connection when the **DataReader** is closed you must specify it as follows:

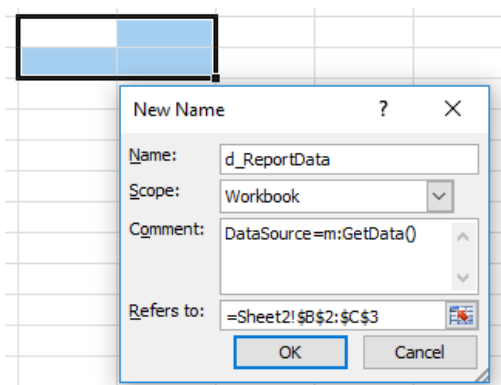
```
IDataReader dataReader = cmd.ExecuteReader(CommandBehavior.CloseConnection);
```

The template specifies the name of the column returned by the SQL query.

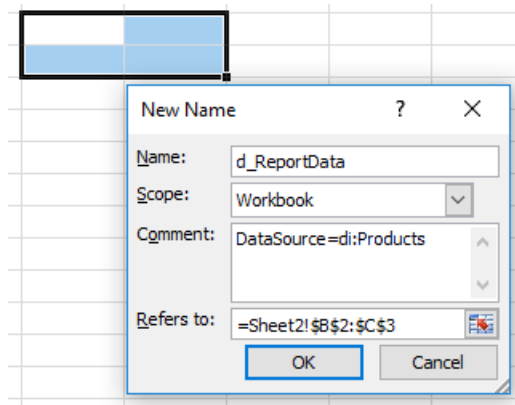
4. **IDictionary<TKey, TValue>**. The data item is a **KeyValuePair<TKey, TValue>** object, in the template you can refer to either the **Key** property or the **Value** property of this structure (you can combine properties through the ".").
5. **IEnumerable<IDictionary<string, TValue>>**. The data item is a **IDictionary<string, TValue>** object. All dictionaries in the sequence must be identical in structure. The dictionary key is specified in the template.
6. **IEnumerable** or **IEnumerable<T>**. The data item is the object contained in this sequence. Public fields or public properties of this object are specified in the template (you can combine fields or properties through the ".").
7. An object of any type. The data item is that object itself. Public fields or public properties of this object are specified in the template (you can combine fields or properties through the ".").

The following properties are relevant for this panel:

1. **DataSource** (required, the type is any template relevant in this context). The template must return one of the data sources described above. The template in this case is set without borders, for example, in the figure below, the method call template is specified as the data source:

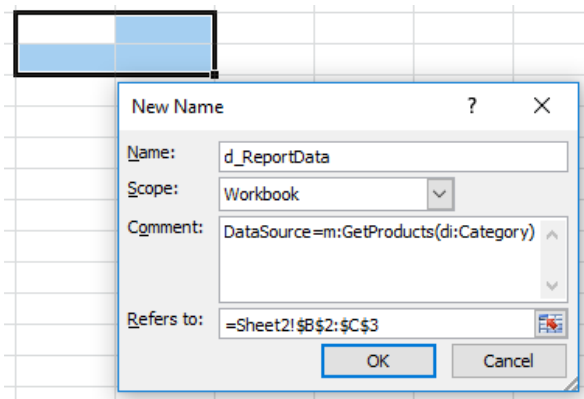


If this panel is a child panel, it has access to the parent panel context, that is, it can use the data items of the parent panel in the **DataSource** property. For example, in the image below, the data item template of the parent panel is specified as the data source for the child panel:



Here, the **di:Products** template can be, for example, a property of an object that contains a collection of products.

You can also use the data item template of the parent panel as a parameter of a method call template, for example:



Here, the data item template **di:Category** of the parent panel is passed as a parameter to the method call template. All this allows you to do quite complex grouping of data. The functionality of child data panels will be described in more detail later.

2. **RenderPriority (int)**. It is completely similar to the same name property of the simple panel.
3. **Type (enum)** allows you to set the panel type to one of two values: **Vertical** (default) or **Horizontal**. This property determines the direction in which the data panel will expand (down or right). By default, this expansion occurs downwards, that is, each new item appears below the previous one. This behavior can be changed by setting this property to the value **Horizontal**. In this case, the expansion will occur to the right, that is, each new item will appear to the right of the previous one.
4. **ShiftType (enum)** allows you to specify the type of shift while the data panel expanding. It can take one of three values: **Cells** (default), **Row** and **NoShift**. If the **Cells** value is set then new cells will be allocated for each subsequent data item within the panel range. Therefore, for the vertical panel, all cells under this range will be shifted downwards the rest will remain in place. For the horizontal panel the same is true except that the cells will be shifted to the right. For example, suppose you have the vertical data panel with coordinates (**B2**, **C2**). At the same time, some text is placed under this panel and to the right of it:

|   | A | B             | C        | D          |
|---|---|---------------|----------|------------|
| 1 |   |               |          |            |
| 2 |   | {di:Customer} | {di:Sum} |            |
| 3 |   |               |          | Right text |
| 4 |   | Bottom text   |          |            |

In the case of **Cells** shift, after rendering, the text "**Bottom Text**" will move down by the number of data items minus one, and the text "**Right text**" will remain in its place. If we assume that the data source has three rows, then the result will be the following:

|   | A | B           | C     | D          |
|---|---|-------------|-------|------------|
| 1 |   |             |       |            |
| 2 |   | Customer_1  | 100\$ |            |
| 3 |   | Customer_2  | 200\$ | Right text |
| 4 |   | Customer_3  | 300\$ |            |
| 5 |   |             |       |            |
| 6 |   | Bottom text |       |            |

If the **Row** value is set then for the vertical panel new cells will be allocated for each subsequent data item for whole row regardless of how many cells the panel occupies. The same is true for the horizontal panel except that columns are allocated instead of rows. In this case, for the vertical panel, anything below will be shifted down by the number of items minus one. For the previous example, the result will be the following:

|   | A | B           | C     | D          |
|---|---|-------------|-------|------------|
| 1 |   |             |       |            |
| 2 |   | Customer_1  | 100\$ |            |
| 3 |   | Customer_2  | 200\$ |            |
| 4 |   | Customer_3  | 300\$ |            |
| 5 |   |             |       | Right text |
| 6 |   | Bottom text |       |            |

If the **NoShift** value is set then additional cells or rows are not allocated, therefore there are no shifts occur. The data panel simply overwrite the cells from the bottom (for the vertical panel) or to the right (for the horizontal panel). For the previous example, the result will be the following:

|   | A | B          | C     | D          |
|---|---|------------|-------|------------|
| 1 |   |            |       |            |
| 2 |   | Customer_1 | 100\$ |            |
| 3 |   | Customer_2 | 200\$ | Right text |
| 4 |   | Customer_3 | 300\$ |            |
| 5 |   |            |       |            |
| 6 |   |            |       |            |
| 7 |   |            |       |            |

Note that the **NoShift** value cannot be set for child data panels, because in this case the parent and child panels will overlap during rendering.

- GroupBy (string)** allows you to specify column numbers (for a vertical panel) or row numbers (for a horizontal panel) which will be grouped, that is, cells with the same value in a row will be merged. A number is the ordinal number of a column or row within a panel range that starts with one. You can list several numbers of columns or rows, separated by a comma, for example: **2,4**.
- GroupBlankCells (bool)** specifies if blank cells should be merged or not while grouping cells (the **GroupBy** property). Default value is **true**.



7. **BeforeRenderMethodName (string)**. It is completely similar to the same name property of the simple panel except that the method signature must be the following:

```
public void BeforeRender(DataSourcePanelBeforeRenderEventArgs args)
```

where **DataSourcePanelBeforeRenderEventArgs** in addition to the properties **Range** and **IsCanceled** also has a property **Data (object)**. That is, before panel rendering, you can perform some additional operations with the data source.

8. **AfterRenderMethodName (string)**. It is completely similar to the same name property of the simple panel except that the method signature must be the following:

```
public void AfterRender(DataSourcePanelEventArgs args)
```

where **DataSourcePanelEventArgs** in addition to the properties **Range** also has a property **Data (object)**.

9. **BeforeDataItemRenderMethodName (string)** allows you to specify the name of the method that will be executed immediately before rendering each panel data item. The method itself must be placed in an instance of the report class, which is passed to the constructor of the **DefaultReportGenerator** class or its descendant. The signature of this method should look as follows:

```
public void BeforeRenderDataItem(DataItemPanelBeforeRenderEventArgs args)
```

where **BeforeRenderDataItem** is the method name that you specify in the comments box for the property **BeforeDataItemRenderMethodName**, **args** is the method parameter which has the **DataItemPanelBeforeRenderEventArgs** type. This type contains the following fields:

- **Range (ClosedXML.Excel.IXLRange)** is the range of cells that the panel covers before rendering.
- **IsCanceled (bool)** is a flag that allows you to cancel the panel rendering. If it is set to true inside this method, the panel as well as all its descendants will not be processed by the library.
- **DataItem (HierarchicalDataItem)** is the data item itself (given hierarchy if panels are nested inside each other).

10. **AfterDataItemRenderMethodName (string)** allows you to specify the name of the method that will be executed immediately after rendering each panel data item. The method itself must be placed in an instance of the report class, which is passed to the constructor of the **DefaultReportGenerator** class or its descendant. The signature of this method should look as follows:

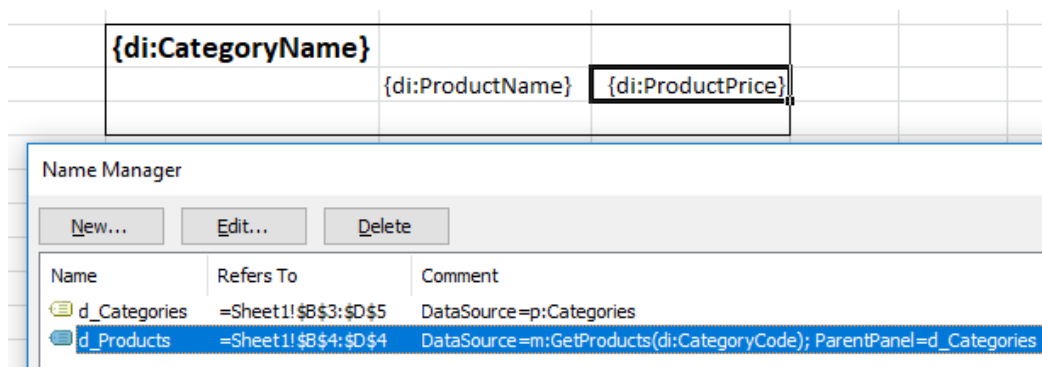
```
public void AfterRenderDataItem(DataItemPanelEventArgs args)
```

where **AfterRenderDataItem** is the method name that you specify in the comments box for the property **AfterDataItemRenderMethodName**, **args** is the method parameter which has the **DataItemPanelEventArgs** type. This type contains the following fields:

- **Range** (**ClosedXML.Excel.IXLRange**) is the range of cells that the panel covers after rendering.
- **DataItem** (**HierarchicalDataItem**) is the data item itself (given hierarchy if panels are nested inside each other).

11. **ParentPanel** (**string**) allows you to specify the name of the parent panel. In this case, the cells range of the parent panel must cover the cells range of the child panel, otherwise, an exception will be thrown. Note that the child panel must occupy the entire width of the parent if it is vertical or the entire height of the parent if it is horizontal. Otherwise, the panels shift will be incorrect. Unnecessary cells can be left empty. You can also achieve a correct shift even if the child panel does not occupy the entire width or height of the parent, if you set **ShiftType=Row** for it.

Let's look more closely at nested panels. Suppose we want to render products grouped by categories. We can create two nested data panels to achieve this. For the parent panel we specify a collection of all categories as the data source. For the child panel we specify the method which will receive the current category from the parent panel. The figure below shows the layout of the sheet:



At the same time, the report class must have the property that returns all categories as well as the method that returns the products by category code. The result may look something like this:

|                   |            |     |
|-------------------|------------|-----|
| <b>Vegetables</b> |            |     |
|                   | Tomatoes   | 5\$ |
|                   | Cucumbers  | 3\$ |
|                   | Onion      | 1\$ |
| <b>Fruits</b>     |            |     |
|                   | Apples     | 4\$ |
|                   | Pears      | 6\$ |
|                   | Peaches    | 8\$ |
|                   | Oranges    | 7\$ |
| <b>Berries</b>    |            |     |
|                   | Blueberry  | 3\$ |
|                   | Strawberry | 4\$ |

That is, nested panels allow you to display data with grouping.

In this case, for the child panel, we used the method that returned products by category code. If each category had a property containing a set of products, the data source for the child panel could be set to this property, that is, the layout of the sheet would look like this:

|                   |                  |                   |  |
|-------------------|------------------|-------------------|--|
| {di:CategoryName} |                  |                   |  |
|                   | {di:ProductName} | {di:ProductPrice} |  |

| Name         | Refers To             | Comment                                          |
|--------------|-----------------------|--------------------------------------------------|
| d_Categories | =Sheet1!\$B\$3:\$D\$5 | DataSource=p:Categories                          |
| d_Products   | =Sheet1!\$B\$4:\$D\$4 | DataSource=di:Products; ParentPanel=d_Categories |

The result will be the same. Which version to use depends on the specific situation.

On this example we also can see the horizontal page break in action. If you insert {HorizPageBreak} template in the last row of the parent panel, each category will be printed on a separate page. In this case, the layout of the panel will take the following form:

|                   |                  |                   |  |
|-------------------|------------------|-------------------|--|
| {di:CategoryName} |                  |                   |  |
|                   | {di:ProductName} | {di:ProductPrice} |  |
| {HorizPageBreak}  |                  |                   |  |

As mentioned above when considering the data item template, you can access the data item of the parent panel using the **parent** pointer in the template. For the previous example, to access the category code from the child panel, you can write:

{di:parent:CategoryCode}

Since the nesting of panels is not limited, it is possible to access any parent in the hierarchy. In this case, you must repeat the **parent** pointer as many times as necessary.

The previous example essentially illustrates the grouping of data. A similar result can also be achieved by using the data panel property **GroupBy**, although the output will be slightly different. In this case, the child panels are not needed. The single data source should return the entire data set at once sorted by categories. There will be only one panel with following markup:

|                   |  |                  |                   |  |
|-------------------|--|------------------|-------------------|--|
| {di:CategoryName} |  | {di:ProductName} | {di:ProductPrice} |  |
|-------------------|--|------------------|-------------------|--|

| Name                  | Refers To             | Comment                                     |
|-----------------------|-----------------------|---------------------------------------------|
| d_ProductByCategories | =Sheet1!\$B\$2:\$D\$2 | DataSource=p:ProductByCategories; GroupBy=1 |

Here, it is specified that grouping should occur on the first column, that is, in our case on the **CategoryName** field. The result will look something like this:

|                   |            |     |
|-------------------|------------|-----|
| <b>Vegetables</b> | Tomatoes   | 5\$ |
|                   | Cucumbers  | 3\$ |
|                   | Onion      | 1\$ |
| <b>Fruits</b>     | Apples     | 4\$ |
|                   | Pears      | 6\$ |
|                   | Peaches    | 8\$ |
|                   | Oranges    | 7\$ |
| <b>Berries</b>    | Blueberry  | 3\$ |
|                   | Strawberry | 4\$ |

Which version of grouping to choose – through the nested panels or through the **GroupBy** property – depends on what you want to get.

### 3. Totals panel

The totals panel name looks like:

**t\_<name>**

where **t** (totals) denotes that this is the totals panel, **<name>** is the panel name, which can be anything, for example, **t\_ReportTotals** is the correct name for the totals panel. This panel can contain almost any type of templates, but it is specific for the aggregate function template which has the form **{Sum(di:Field)}**, where **Sum** is an aggregate function type, **di:Field** is the data item you want to aggregate. The aggregate function template itself is described in more detail in the corresponding chapter. This panel has a required property **DataSource** which must have any template that returns a data source. Data sources for this panel are completely similar to data sources for the data panel.

This panel retrieves all the aggregate functions specified in it, loops through the data source, and calculates values of these aggregate functions. All aggregate functions are calculated for one pass through the data. The rest templates are calculated in the usual way. Note that the data item template without an aggregate function can only appear in this panel when you access the data item of the parent panel (through the **parent** pointer).

The following properties are relevant for this panel:

1. **DataSource** (required, the type is any template relevant in this context). It is completely similar to the same name property of the data panel.
2. **RenderPriority** (int). It is completely similar to the same name property of the simple panel.
3. **BeforeRenderMethodName** (string). It is completely similar to the same name property of the data panel.
4. **AfterRenderMethodName** (string). It is completely similar to the same name property of the data panel.
5. **ParentPanel** (string). It is completely similar to the same name property of the data panel.

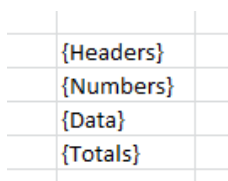
## 4. Dynamic panel

The dynamic panel name looks like:

`dyn_<name>`

where **dyn** (dynamic) denotes that this is the dynamic panel, **<name>** is a panel name, which can be anything, for example, **dyn\_ReportData** is the correct name for the dynamic panel.

The dynamic panel differs from all other panels. It uses its own specific templates. In the full version the panel has the following form:



|           |
|-----------|
| {Headers} |
| {Numbers} |
| {Data}    |
| {Totals}  |

The image above shows all kinds of templates that the dynamic panel can contain. Any of these templates can be omitted and they can appear in any order. This panel has a required property **DataSource**. Let's look at each of the panel template in more detail.

1. The template **{Headers}** turns into column headers (if the panel is vertical) or into row headers (if the panel is horizontal). That is, in the case of the vertical panel, the **{Headers}** template multiplies to the right and, in the case of horizontal panel, it multiplies down. Depending on the data source, column or row names are calculated differently:
  - **DataTable**. Column names are taken from the **Caption** property of the **DataColumn** object. If the **Caption** property is **null**, column names are taken from the **Name** property. The type of columns is determined from the **DataType** property of the **DataColumn** object.
  - **DataSet**. It is similar to the **DataTable** because the first **DataTable** is retrieved from the **DataSet**.
  - **IDataReader**. Column names are the names of the columns returned by the SQL query. The type of columns is determined from the metadata table for the **IDataReader** object.
  - **IDictionary<TKey, TValue>**. The names of columns are always the **Key** and **Value**. The type of columns will correspond to the **TKey** and **TValue** types.
  - **IEnumerable<IDictionary<string, TValue>>**. The names of columns are the dictionary keys. The type of columns will correspond to the **TValue** type.
  - **IEnumerable, IEnumerable<T>**. By default, the column names are the names of public fields and properties of an object contained in a sequence that has primitive data types, enumeration types or **Decimal**, **String**, **DateTime**, and **Guid**. You can override the default behavior by using special attributes that are described below. The type of columns will correspond to the type of the field or property.

- Object of any other type. The same as for **IEnumerable** and **IEnumerable<T>**.

If the data source is an **IEnumerable** or **IEnumerable<T>** sequence (or perhaps a single object), then, as mentioned above, by default, the public fields and properties that have the certain data types are treated as columns. You can adjust this behavior by using the following attributes:

- **NoExcelColumn**. A field or property marked with this attribute is no longer considered as a column and is not displayed in the dynamic panel.
- **ExcelColumn**. This attribute allows you to specify some parameters of the column in Excel.

It has the following properties:

- **Caption (string)**. By default, the name of a column in Excel is the name of a property or field, but you can use this property to specify a more appropriate name.
- **DisplayFormat (string)** allows you to specify the output format for this column. This property is relevant for numeric data and for dates. For the **Decimal** type the default format is **#,0.00**.
- **IgnoreDisplayFormat (bool)**. This attribute specifies that the format should be ignored even it is specified in the **DisplayFormat** property. It can be useful, for example, to reset the default format for the **Decimal** type if it does not suit you.
- **AggregateFunction (AggregateFunction)** allows you to specify an aggregate function that will be applied to this column in the **{Totals}** template. This property has an enumeration type that contains all the standard aggregate functions. The default aggregate function for the **Decimal** type is **Sum**.
- **NoAggregate (bool)** specifies that no aggregation should be applied to this column even it is specified in the **AggregateFunction** field. This can be useful, for example, to reset the default aggregation for the **Decimal** type if you do not need it.
- **Width (int)** allows you to set the width of the column in the case of a vertical panel or the height of the row in the case of a horizontal panel.
- **AdjustToContent (bool)** specifies that the column width should be adjusted to the content.
- **Order (int)** specifies the order of the column in which it appears on the sheet. By default this field is equal to "0", so the columns appear in the order of the declaration of fields and properties in the type.

2. The **{Numbers}** template is used to display the column number in the case of a vertical panel or a row number in the case of a horizontal panel. By default, the number starts with one, but this template can have an extended version, where you can specify the number from which the numbering starts. In that case, for example, it may look like this: **{Numbers(5)}**. This will mean that the numbering of columns (rows) will start with five.

3. The template **{Data}** basically is not very remarkable. It just displays data from a data source via a data panel according to the columns that were previously defined. Note that even if the **{Headers}** template is not defined in the panel, the columns for output will still be defined correctly.
4. The **{Totals}** template is used to display totals. By default, totals are counted only for the **Decimal** type by using the aggregate function **Sum**. You can specify aggregate functions for any other fields through the **AggregateFunction** property of the **ExcelColumn** attribute or through the lifecycle functions of a dynamic panel. These functions will be described in more detail below in this chapter.

The following properties are relevant for this panel:

1. **DataSource** (required, the type is any template relevant in this context). It is completely similar to the same name property of the data panel.
2. **RenderPriority** (**int**). It is completely similar to the same name property of the simple panel.
3. **Type** (**enum**). It is completely similar to the same name property of the data panel.
4. **ShiftType** (**enum**). It is completely similar to the same name property of the data panel.
5. **GroupBy** (**string**). It is completely similar to the same name property of the data panel.
6. **BeforeRenderMethodName** (**string**). It is completely similar to the same name property of the data panel.
7. **AfterRenderMethodName** (**string**). It is completely similar to the same name property of the data panel.
8. **BeforeHeadersRenderMethodName** (**string**) allows you to specify the name of the method that will be executed immediately before column or row headers rendering. The method itself must be placed in an instance of the report class, which is passed to the constructor of the **DefaultReportGenerator** class or its descendant. The signature of this method should look as follows:

```
public void BeforeHeadersRender(DataSourceDynamicPanelBeforeRenderEventArgs args)
```

where **BeforeHeadersRender** is the method name that you specify in the comments box for the property **BeforeHeadersRenderMethodName**, **args** is the method parameter which has the **DataSourceDynamicPanelBeforeRenderEventArgs** type. This type contains the following fields:

- **Range** (**ClosedXML.Excel.IXLRange**) is the range of cells that the panel covers before rendering.
- **IsCanceled** (**bool**) is a flag that allows you to cancel the panel rendering. If it is set to true inside this method, the panel as well as all its descendants will not be processed by the library.

- **Data (object)** is the data source for this panel.
- **Columns (IList<ExcelDynamicColumn>)** are data columns extracted from the data source. The **ExcelDynamicColumn** has the following fields:
  - **Name (string)** is the column (row) name in the data source.
  - **Caption (string)** is the column (row) name to be displayed in Excel.
  - **Width (double?)** is the column width (or row height if the panel is horizontal).
  - **DataType (Type)** is the column type.
  - **AggregateFunction (AggregateFunction)** is an aggregate function that will be applied to this column.
  - **DisplayFormat (string)** is the format that will be applied to this column.
  - **AdjustToContent (bool)** is specifies that the column width should be adjusted to the content.

That is, in this method you can configure any data column by setting the necessary properties. You can even add a new column to a collection or delete an existing column, but in this case you will also need to adjust the data source, which is also possible in this method. All other lifecycle methods will accept the adjusted column collection and data source.

9. **AfterHeadersRenderMethodName (string)** allows you to specify the name of the method that will be executed immediately after column or row headers rendering. The method itself must be placed in an instance of the report class, which is passed to the constructor of the **DefaultReportGenerator** class or its descendant. The signature of this method should look as follows:

```
public void AfterHeadersRender(DataSourceDynamicPanelEventArgs args)
```

where **AfterHeadersRender** is the method name that you specify in the comments box for the property **AfterHeadersRenderMethodName**, **args** is the method parameter which has the **DataSourceDynamicPanelEventArgs** that contains all of the same fields as the **DataSourceDynamicPanelBeforeRenderEventArgs** that was described earlier, except for the **IsCanceled** field. In this case, the **Range** field will contain a range of cells that the panel covers after rendering.

10. **BeforeNumbersRenderMethodName (string)** allows you to specify the name of the method that will be executed immediately before column or row numbers rendering. In all other respects this method is completely similar to the method **BeforeHeadersRenderMethodName**.
11. **AfterNumbersRenderMethodName (string)** allows you to specify the name of the method that will be executed immediately after column or row numbers rendering. In all other respects this method is completely similar to the method **AfterHeadersRenderMethodName**.



12. **BeforeDataTemplatesRenderMethodName** (**string**) allows you to specify the name of the method that will be executed immediately before data templates rendering. The dynamic panel renders data through the data panel, so the **{Data}** template is first converted to data item templates, and then the resulting range with these templates is passed to the data panel for rendering. In all other respects this method is completely similar to the method **BeforeHeadersRenderMethodName**.
13. **AfterDataTemplatesRenderMethodName** (**string**) allows you to specify the name of the method that will be executed immediately after data templates rendering. In all other respects this method is completely similar to the method **AfterHeadersRenderMethodName**. You can change here the formed template of the data item in any way.
14. **BeforeDataRenderMethodName** (**string**) allows you to specify the name of the method that will be executed immediately before data rendering. This method corresponds to the **BeforeRenderMethodName** method of the data panel.
15. **AfterDataRenderMethodName** (**string**) allows you to specify the name of the method that will be executed immediately after data rendering. This method corresponds to the **AfterRenderMethodName** method of the data panel.
16. **BeforeDataItemRenderMethodName** (**string**). This method corresponds to the same name method of the data panel.
17. **AfterDataItemRenderMethodName** (**string**). This method corresponds to the same name method of the data panel.
18. **BeforeTotalsTemplatesRenderMethodName** (**string**) allows you to specify the name of the method that will be executed immediately before aggregate functions templates rendering. The dynamic panel renders totals through the totals panel, so the **{Totals}** template is first converted to aggregate functions templates, and then the resulting range with these templates is passed to the totals panel for rendering. In all other respects this method is completely similar to the method **BeforeHeadersRenderMethodName**.
19. **AfterTotalsTemplatesRenderMethodName** (**string**) allows you to specify the name of the method that will be executed immediately after aggregate functions templates rendering. In all other respects this method is completely similar to the method **AfterHeadersRenderMethodName**. You can change here the formed template of the aggregate function in any way, for example, add the post-aggregation function.
20. **BeforeTotalsRenderMethodName** (**string**) allows you to specify the name of the method that will be executed immediately before totals rendering. This method corresponds to the **BeforeRenderMethodName** method of the totals panel.

21. **AfterTotalsRenderMethodName** (**string**) allows you to specify the name of the method that will be executed immediately after totals rendering. This method corresponds to the **AfterRenderMethodName** method of the totals panel.

22. **ParentPanel** (**string**). This property is similar to the same name property of the data panel. There are some limitations when you nest the dynamic panel to the data panel:

- You can nest the dynamic panel in the data panel if both of them have the same value in the **Type** property (either vertical or both horizontal).
- You cannot access from the nested dynamic panel to the context of the parent panel through the **parent** pointer.

## Connection and extension

It is quite easy to connect the library. To do this use the **DefaultReportGenerator** class whose constructor takes a single parameter – an instance of a report of the **object** type that you want to display in Excel. To start rendering use the **Render** method with the following signature:

```
public XLWorkbook Render(XLWorkbook reportTemplate, IXLWorksheet[] worksheets = null)
```

where **reportTemplate** is an Excel workbook marked with the required panels and templates, **worksheets** is an array of sheets to be processed by the engine. If the array is **null** or empty, all sheets will be processed. The return value is the completed report where all panels and templates are replaced with real data.

The **DefaultReportGenerator** class has the following events:

- **BeforeReportRender** occurs just before rendering the entire Excel workbook. It has the **EventHandler<ReportRenderEventArgs>** type where **ReportRenderEventArgs** contains a single field **Workbook** of type **XLWorkbook**. This is the Excel workbook template that was passed to the **Render** method as the first parameter.
- **BeforeWorksheetRender** occurs just before rendering of each sheet. It has the **EventHandler<WorksheetRenderEventArgs>** type where **WorksheetRenderEventArgs** contains a single field **Worksheet** of type **IXLWorksheet**.
- **AfterWorksheetRender** occurs just after rendering of each sheet. It has the same type as the event **BeforeWorksheetRender**.

The **DefaultReportGenerator** class has the ability for extension and customization.

### 1. Extension of system functions

The **DefaultReportGenerator** class has the property **SystemFunctionsType** of type **Type**. By default, this property is set to an instance of the **SystemFunctions** type. To extend system functions you must create a class inherited from the class **SystemFunctions**, define your custom system functions in it, and then assign the type of that class to the **SystemFunctionsType** property of the **DefaultReportGenerator** object.

### 2. Extension of system variables

The **DefaultReportGenerator** class has the property **SystemVariableProvider** of type **SystemVariableProvider**. By default, this property is set to an instance of the **SystemVariableProvider** type which contains the system variables supplied by the library. To extend system variables you must create a class inherited from the class **SystemVariableProvider**, define your custom system variables in it, and then assign the type of that class to the **SystemVariableProvider** property of the **DefaultReportGenerator** object.

### 3. Extension of DefaultTypeProvider

The **DefaultReportGenerator** class has the property **TypeProvider** of type **ITypeProvider**. By default, this property is set to an instance of the **DefaultTypeProvider** type. The **ITypeProvider** interface provides types specified in the template either explicitly or implicitly. If the type is not specified in the template, for example, **{p:Name}**, **DefaultTypeProvider** returns the type of an instance passed in the constructor of the **DefaultReportGenerator** class, that is, the property **Name** will be searched in this type. If the type is specified explicitly, for example, **{p:Company:Name}**, **DefaultTypeProvider** will look for the **Company** type in the executable assembly. If no type is found, or more than one type with the given name is found, the corresponding exception is thrown. You can extend the type name in the template by the namespace, for example, **{p:Reports.Common:Company:Name}**. In this case, the type **Company** will be searched taking into account the **Reports.Common** namespace. This is true for any template where you can specify a type.

By default, all types are searched in the executable assembly, but you can change this behavior. The constructor of class **DefaultTypeProvider** has two parameters:

- **assemblies (ICollection<Assembly>)** is a collection of assemblies in which types are searched. If the collection is **null** or empty, the types are searched in the executable assembly. The default value is **null**.
- **defaultType (Type)**. The type returned if the template does not explicitly specify a type. Default value is **null**. When an instance of the **DefaultTypeProvider** is created inside the **DefaultReportGenerator** class, this parameter is set to the type of the object passed in the constructor.

To change the default behavior you must create a class that inherits from the **DefaultReportGenerator** class and override the **TypeProvider** property to return the desired implementation. This implementation can be an instance of the **DefaultTypeProvider** class created with the necessary constructor parameters or you can extend the **DefaultTypeProvider** class or implement the **ITypeProvider** interface from scratch. This interface has a single method with the following signature:

```
Type GetType(string typeTemplate);
```

This method accepts a string template of the type and must return the type corresponding to the given template.

### 4. Extension of DefaultInstanceProvider

The **DefaultReportGenerator** class has the property **InstanceProvider** of type **IInstanceProvider**. By default, this property is set to an instance of the **DefaultInstanceProvider** type. The **IInstanceProvider** interface provides instances for types specified in the templates. The type **DefaultInstanceProvider** has a single constructor parameter in which you can pass an instance of an

object that will be returned if the type is not explicitly specified in the template. When an instance of the **DefaultInstanceProvider** is created inside the **DefaultReportGenerator** class, this parameter is set to the type of the object passed in the constructor. Therefore, when the type is not specified explicitly in the template, all properties, methods, etc. are called on the report object. Note that in order to **DefaultInstanceProvider** could create an instance of an object, the type must have a default constructor. All objects supplied by the **DefaultInstanceProvider** class are **Singleton** objects, that is, if multiple templates specify the same type, all references to instance members of that type will occur on the object created at the first access time.

To change the default behavior you must create a class that inherits from the **DefaultReportGenerator** class and override the **InstanceProvider** property to return the desired implementation. This implementation can be an instance of the **DefaultInstanceProvider** class created with the necessary constructor parameters or you can extend the **DefaultInstanceProvider** class or implement the **IInstanceProvider** interface from scratch. This interface has two methods with the following signature:

```
object GetInstance(Type type);  
T GetInstance<T>();
```

Both methods must return an instance of the specified type. The second method is simply a generic version of the first one. For example, you can replace the default implementation with an implementation that will retrieve instances of objects through any IoC-container.

## 5. Extension of DefaultPropertyValueProvider

The **DefaultReportGenerator** class has the property **PropertyValueProvider** of type **IPropertyValueProvider**. By default, this property is set to an instance of the **DefaultPropertyValueProvider** type. The **IPropertyValueProvider** provides values for property template. The constructor of class **DefaultPropertyValueProvider** has two parameters:

- **typeProvider (ITypeProvider)** is an instance of an object that will provide the types specified in the templates. In the **DefaultReportGenerator** class this parameter is set to the value of the **TypeProvider** property described above.
- **instanceProvider (IInstanceProvider)** is an instance of an object that will provide the instances of types specified in the templates. In the **DefaultReportGenerator** class this parameter is set to the value of the **InstanceProvider** property described above.

The **DefaultPropertyValueProvider** implementation supports accessing to public fields and properties both static and instance. It is also possible to combine fields and properties through the ".".

To change the default behavior you must create a class that inherits from the **DefaultReportGenerator** class and override the **PropertyValueProvider** property to return the desired implementation. This

implementation can be an instance of the **DefaultPropertyValueProvider** class created with the necessary constructor parameters or you can extend the **DefaultPropertyValueProvider** class or implement the **IPropertyValueProvider** interface from scratch. This interface has a single method with the following signature:

```
object GetValue(string propertyTemplate);
```

This method accepts a property template and must return the value of this property based on this template.

## 6. Extension of DefaultMethodCallValueProvider

The **DefaultReportGenerator** class has the property **MethodCallValueProvider** of type **IMethodCallValueProvider**. By default, this property is set to an instance of the **DefaultMethodCallValueProvider** type. The **IMethodCallValueProvider** interface provides values for method call templates. The **DefaultMethodCallValueProvider** constructor accepts the same arguments as the **DefaultPropertyValueProvider** constructor.

The **DefaultMethodCallValueProvider** implementation supports calling both static and instance public methods. You can pass to the method as many parameters as you want, and those parameters can be either static data hard-coded in the template (strings, numbers, etc) or almost any type of templates including calls of other methods (nesting of methods invocation is not limited). Methods with variable number of arguments (**params**) are not supported. Methods overloading is partially supported. Sometimes you have to explicitly specify static parameter types so that the overloaded method is determined correctly. If the overloaded method cannot be uniquely determined, the corresponding exception is thrown.

To change the default behavior you must create a class that inherits from the **DefaultReportGenerator** class and override the **MethodCallValueProvider** property to return the desired implementation. This implementation can be an instance of the **DefaultMethodCallValueProvider** class created with the necessary constructor parameters or you can extend the **DefaultMethodCallValueProvider** class or implement the **IMethodCallValueProvider** interface from scratch. This interface has two methods:

```
object CallMethod(string methodCallTemplate, ITemplateProcessor templateProcessor, HierarchicalDataItem dataItem);
```

This method accepts a method call template, a template processor (described below), and a data item if the template is in a data context. It returns the result of a method call.

```
object CallMethod(string methodCallTemplate, Type concreteType, ITemplateProcessor templateProcessor, HierarchicalDataItem dataItem);
```

This method, unlike the previous one, takes another parameter **concreteType** of type **Type**. In this case, the type in which methods are searched is specified explicitly, otherwise it is completely similar to the above.

## 7. Extension of DefaultDataItemValueProvider

The **DefaultReportGenerator** class has the property **DataItemValueProvider** of type **IGenericDataItemValueProvider<HierarchicalDataItem>**. By default, this property is set to an instance of the **DefaultDataItemValueProvider** type. The **IGenericDataItemValueProvider<HierarchicalDataItem>** provides values for data item templates. The **HierarchicalDataItem** class is a hierarchical data item. **DefaultDataItemValueProvider** does not accept any parameters in the constructor.

The implementation of **DefaultDataItemValueProvider** supports rendering of various data items (**DataRow**, **KeyValuePair<TKey, TValue>**, object of any type, etc.). When accessing a data item you can combine fields and properties through the ".". Data hierarchy is also supported through the **parent** pointer.

To change the default behavior you must create a class that inherits from the **DefaultReportGenerator** class and override the **DataItemValueProvider** property to return the desired implementation. In this implementation you can extend the **DefaultDataItemValueProvider** class or implement the **IGenericDataItemValueProvider<HierarchicalDataItem>** interface from scratch. This interface has two methods:

```
object GetValue(string template, object dataItem);
```

This method accepts the data item template and the data item itself. It returns the value from that data item according to the template.

```
object GetValue(string template, T dataItem);
```

This is the generic version of the first method.

## 8. Extension of DefaultTemplateProcessor

The **DefaultReportGenerator** class has the property **TemplateProcessor** of type **ITemplateProcessor**. By default, this property is set to an instance of the **DefaultTemplateProcessor** type. The **ITemplateProcessor** interface is responsible for parsing and processing all templates. The constructor of class **DefaultTemplateProcessor** has following parameters:

- **propertyValueProvider (IPropertyValueProvider)** is an instance of an object that provides values for property templates. In the **DefaultReportGenerator** class this parameter is set to the value of the **PropertyValueProvider** property described above.
- **systemVariableProvider (SystemVariableProvider)** is an instance of an object that provides values for system variables. In the **DefaultReportGenerator** class this parameter is set to the value of the **SystemVariableProvider** property described above.

- **methodCallValueProvider** (**IMethodCallValueProvider**) is an instance of an object that provides values for method call templates. In the **DefaultReportGenerator** class this parameter is set to the value of the **MethodCallValueProvider** property described above.
- **dataItemValueProvider** (**IGenericDataItemValueProvider<HierarchicalDataItem>**) is an instance of an object that provides values for data item templates. In the **DefaultReportGenerator** class this parameter is set to the value of the **DataItemValueProvider** property described above.

To change the default behavior you must create a class that inherits from the **DefaultReportGenerator** class and override the **TemplateProcessor** property to return the desired implementation. This implementation can be an instance of the **DefaultTemplateProcessor** class created with the necessary constructor parameters or you can extend the **DefaultTemplateProcessor** class or implement the **ITemplateProcessor** interface from scratch. This interface has the following read-only properties:

- **LeftTemplateBorder** (**string**) is the left template border. It cannot be null, empty, or whitespace. In the **DefaultTemplateProcessor** class this property has a value "{".
- **RightTemplateBorder** (**string**) is the right template border. It cannot be null, empty, or whitespace. In the **DefaultTemplateProcessor** class this property has a value "}".
- **MemberLabelSeparator** (**string**) is the delimiter that separates a template type (label) from the template itself. It cannot be null or an empty string. In the **DefaultTemplateProcessor** class this property has a value ":".
- **PropertyMemberLabel** (**string**) is the property template label. It cannot be null, empty, or whitespace. In the **DefaultTemplateProcessor** class this property has a value "p".
- **MethodCallMemberLabel** (**string**) is the method call template label. It cannot be null, empty, or whitespace. In the **DefaultTemplateProcessor** class this property has a value "m".
- **DataItemMemberLabel** (**string**) is the data item template label. It cannot be null, empty, or whitespace. In the **DefaultTemplateProcessor** class this property has a value "di".
- **SystemVariableMemberLabel** (**string**) is the system variable template label. It cannot be null, empty, or whitespace. In the **DefaultTemplateProcessor** class this property has a value "sv".
- **SystemFunctionMemberLabel** (**string**) is the system function template label. It cannot be null, empty, or whitespace. In the **DefaultTemplateProcessor** class this property has a value "sf".
- **HorizontalPageBreakLabel** (**string**) is the horizontal page break template label. It cannot be null, empty, or whitespace. In the **DefaultTemplateProcessor** class this property has a value "HorizPageBreak".



- **VerticalPageBreakLabel (string)** is the vertical page break template label. It cannot be null, empty, or whitespace. In the **DefaultTemplateProcessor** class this property has a value "VertPageBreak".
- **DataItemSelfTemplate (string)** is the template is used to return the data item itself, but not any field or property from it (it is useful when a data item has a simple type). It cannot be null, empty, or whitespace. In the **DefaultTemplateProcessor** class this property returns the value of **DataItemMemberLabel** property mentioned above. Therefore, by default, the template to return the data item itself looks like **{di:di}**.

Also, this interface has a single method with the following signature:

```
object GetValue(string template, HierarchicalDataItem dataItem = null);
```

This method accepts a template and a data item if the template is in a data context. It returns the value of the template.

## 9. Panels parsing settings

The **DefaultReportGenerator** class has the property **PanelParsingSettings** of type **PanelParsingSettings**. This class **PanelParsingSettings** has the following properties:

- **PanelPrefixSeparator (string)** is the delimiter between the panel type prefix and the panel name. It cannot be null, empty, or whitespace. The default value is "\_".
- **SimplePanelPrefix (string)** is simple panel prefix. It cannot be null, empty, or whitespace. The default value is "s".
- **DataSourcePanelPrefix (string)** is data panel prefix. It cannot be null, empty, or whitespace. The default value is "d".
- **DynamicDataSourcePanelPrefix (string)** is dynamic panel prefix. It cannot be null, empty, or whitespace. The default value is "dyn".
- **TotalsPanelPrefix (string)** is totals panel prefix. It cannot be null, empty, or whitespace. The default value is "t".
- **PanelPropertiesSeparators (string[])** is an array of delimiters between the panel properties. It cannot be null, or empty. The default value is ["\n", "\t", ";"]. That is, properties can be separated by either a newline character or a tab character or a semicolon.
- **PanelPropertyNameValueSeparator (string)** is the delimiter between the name and value of a panel property. It cannot be null, empty, or whitespace. Default value is "=".

To change the default behavior you must create a class that inherits from the **DefaultReportGenerator** class and override the **PanelParsingSettings** property to return an instance of the **PanelParsingSettings** class with the appropriate settings.