OpenL Tablets Reference Guide

OpenL Tablets 5.12

OpenL Tablets Rules Engine

Document number: TP_OpenL_Ref_2.1_SN
Revised: 04-18-2014

Table of Contents

Preface

This preface is an introduction to the *OpenL Tablets Reference Guide*.

The following topics are included in this preface:

Audience

This guide is mainly intended for analysts and developers who create applications employing the table based decision making mechanisms offered by OpenL Tablets technology. However, other users can also benefit from this guide by learning the basic OpenL Tablets concepts described herein.

Basic knowledge of Excel® is desired to use this guide effectively. Basic knowledge of Java is desired to use some development related sections.

Related Information

The following table lists sources of information related to contents of this guide:

Related information	
Title	**Description**
OpenL Tablets WebStudio User's Guide	Document describing OpenL Tablets WebStudio, a web application for managing OpenL Tablets projects through web browser.
http://openl-tablets.sourceforge.net/	OpenL Tablets open source project website.

Typographic Conventions

The following styles and conventions are used in this guide:

Typographic styles and conventions	
Convention	**Description**
Bold	• Represents user interface items such as check boxes, command buttons, dialog boxes, drop-down list values, field names, menu commands, menus, option buttons, perspectives, tabs, tooltip labels, tree elements, views, and windows. • Represents keys, such as **F9** or **CTRL+A.** • Represents a term the first time it is defined.
`Courier`	Represents file and directory names, code, system messages, and command-line commands.
`Courier Bold`	Represents emphasized text in code.
Select **File > Save As**	Represents a command to perform, such as opening the **File** menu and selecting **Save As.**

Typographic styles and conventions	
Convention	**Description**
Italic	• Represents any information to be entered in a field. • Represents documentation titles.
< >	Represents placeholder values to be substituted with user specific values.
Hyperlink	Represents a hyperlink. Clicking a hyperlink displays the information topic or external source.

Chapter 1: Introducing OpenL Tablets

This section introduces OpenL Tablets and describes its main concepts.

The following topics are included in this section:

- What Is OpenL Tablets?
- Basic Concepts
- System Overview
- Installing OpenL Tablets
- Tutorials and Examples

What Is OpenL Tablets?

OpenL Tablets is a Business Rules Management System (BRMS) and Business Rules Engine (BRE) based on tables presented in Excel documents. Using unique concepts, OpenL Tablets facilitates treating business documents containing business logic specifications as executable source code. Since the format of tables used by OpenL Tablets is familiar to business users, OpenL Tablets bridges a gap between business users and developers, thus reducing costly enterprise software development errors and dramatically shortening the software development cycle.

In a very simplified overview, OpenL Tablets can be considered as a table processor that extracts tables from Excel documents and makes them accessible from software applications.

The major advantages of using OpenL Tablets are as follows:

- OpenL Tablets removes the gap between software implementation and business documents, rules, and policies.
- Business rules become transparent to developers.
- OpenL Tablets verifies syntax and type errors in all project document data, providing convenient and detailed error reporting.
- OpenL Tablets is able to directly point to a problem in an Excel document.
- OpenL Tablets provides calculation explanation capabilities, enabling expansion of any calculation result by pointing to source arguments in the original documents.
- OpenL Tablets provides cross-indexing and search capabilities within all project documents.

OpenL Tablets supports the `.xls, .xlsx, .xlsm` file formats.

Basic Concepts

This section describes the following main OpenL Tablets concepts:

- Rules
- Tables
- Projects

Rules

In OpenL Tablets, a **rule** is a logical statement consisting of conditions and actions. If a rule is called and all its conditions are true then the corresponding actions are executed. Basically, a rule is an IF-THEN statement. The following is an example of a rule expressed in human language:

If a service request costs less than 1,000 dollars and takes less than 8 hours to execute then the service request must be approved automatically.

Instead of executing actions, rules can also return data values to the calling program.

Tables

Basic information OpenL Tablets deals with, such as rules and data, is presented in tables. Different types of tables serve different purposes. For detailed information on table types, see Table Types.

Projects

An **OpenL Tablets project** is a container of all resources required for processing rule related information. Usually, a project contains Excel files, which are called **modules** of the project, and optionally Java code, library dependencies, etc. For detailed information on projects, see Chapter 5: Working with Projects.

There can be situations where OpenL Tablets projects are used in the development environment but not in production, depending on the technical aspects of a solution.

System Overview

The following diagram shows how OpenL Tablets is used by different types of users:

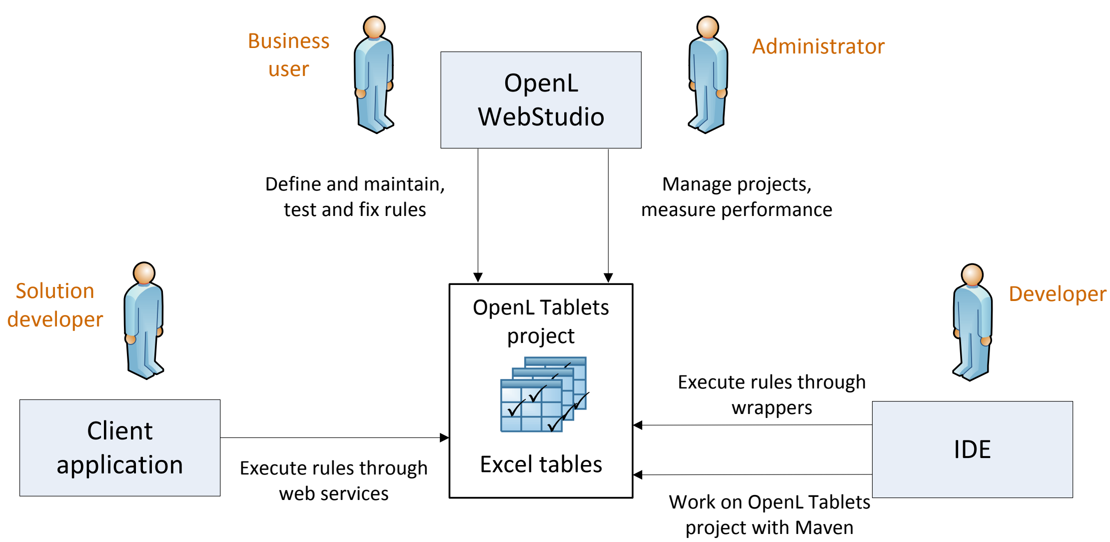


Figure 1: OpenL Tablets overview

The following is a typical lifecycle of an OpenL Tablets project:

1. A business analyst creates a new OpenL Tablets project in WebStudio. Optionally, development team may provide the analyst with a project in case of complex configuration.

   The business analyst also creates correctly structured tables in Excel files based on requirements and includes them in the project. Typically, this task is performed through Excel or OpenL Tablets WebStudio in a web browser.
2. The business analyst performs unit and integration tests by creating test tables and performance tests on rules through OpenL Tablets WebStudio. As a result fully working rules are created and ready to be used.
3. Development team that creates other parts of the solution employs business rules directly through the OpenL Tablets engine or remotely through web services.
4. Whenever required, the business user updates or adds new rules to project tables. OpenL Tablets business rules management applications (OpenL Tablets WebStudio, Rules Repository, Rule Service) can be setuped to provide self-service environment for business user changes.

Installing OpenL Tablets

Refer to OpenL Tablets Installation Guide for installation details.

The development environment is required only for creating OpenL Tablets projects and launching OpenL Tablets WebStudio. If OpenL Tablets projects are accessed through OpenL Tablets WebStudio or web services, no specific software needs to be installed.

Tutorials and Examples

OpenL Tablets provides a number of preconfigured projects intended for new users who want to learn working with it quickly.

These projects are organized into following groups:

- Tutorials
- Examples

Tutorials

OpenL Tablets provides a set of tutorial projects demonstrating basic OpenL Tablets features starting from very simple and following with more advanced projects. Files in the tutorial projects contain detailed comments allowing new users to grasp basic concepts quickly.

To create a tutorial project, proceed as follows:

1. In OpenL Tablets WebStudio, click the **Repository** item in the top line menu to open the Repository Editor**.**
2. Click the **Create Project** button: Create Project.
3. In the **Create Project from** dialog, navigate to the desired tutorial and click its name.
4. Click the **Create** button to complete. The project appears in the **Projects** list of the Repository Editor.

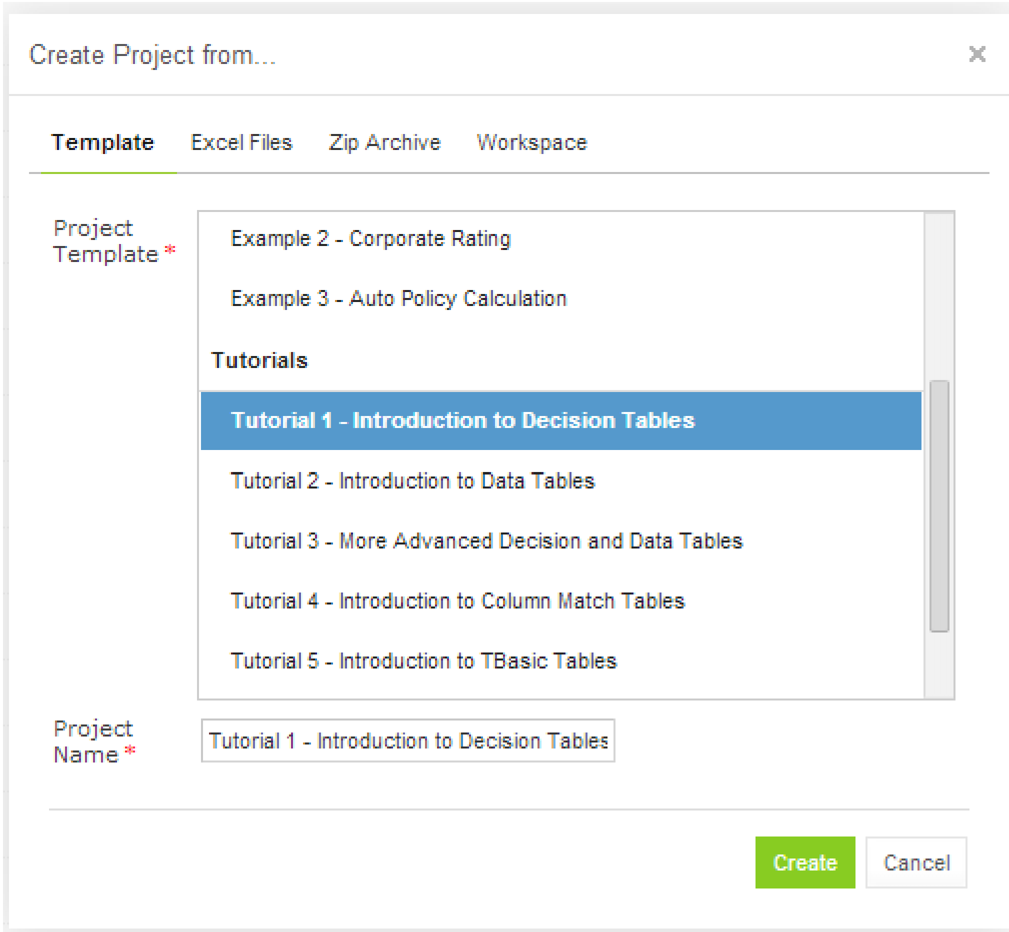


Figure 2: Creating tutorial projects

5.Click **Rules Editor** in the top line menu. The project is displayed in the **Projects** list and available for usage.

We highly recommend you to start from reading Excel files for Examples /Tutorials since they provide clear explanations for every step involved.

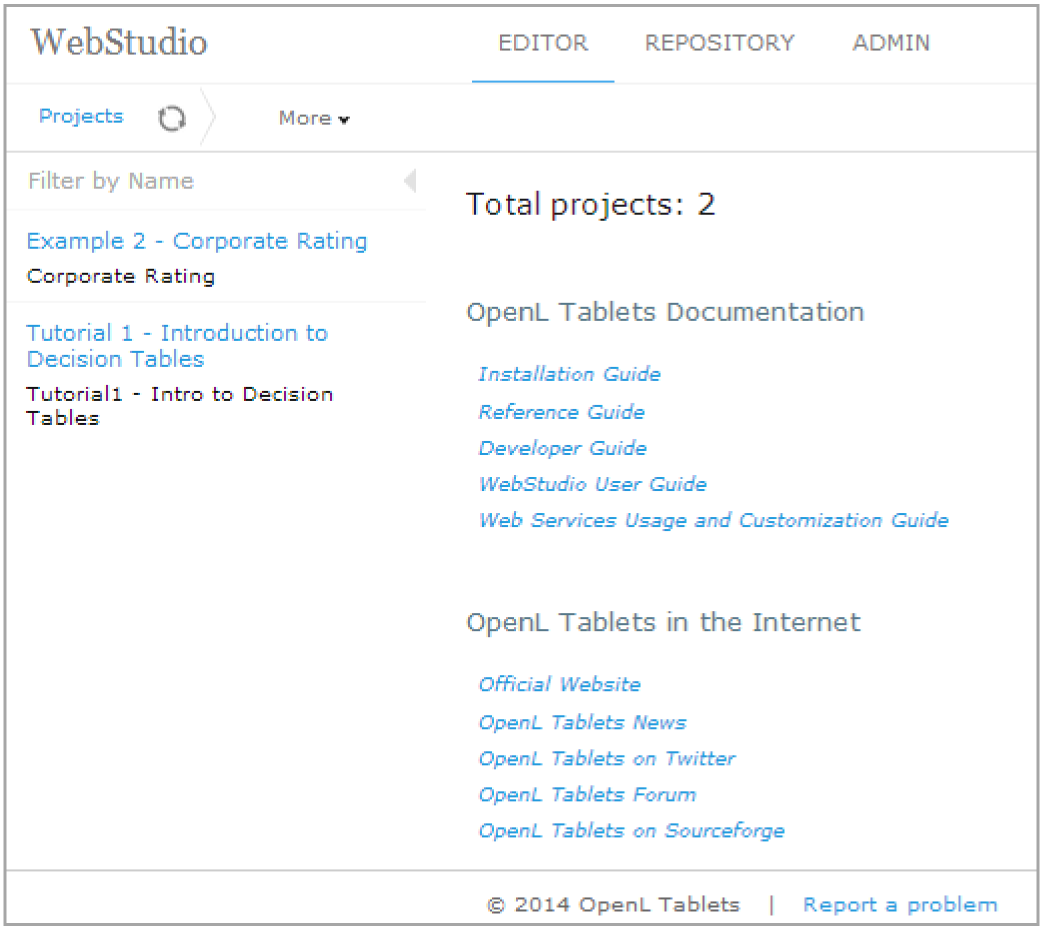


Figure 3: Tutorial project in the OpenL Tablets WebStudio

Examples

In addition to tutorials, OpenL Tablets provides several example projects that demonstrate how OpenL Tablets can be used in various business domains.

To create an example project, follow the steps described in the **Tutorials** section, just in the **Create Project from** dialog choose an example you wish to explore. After you complete, the example appears in the WebStudio Rules Editor as shown in the Figure 3.

Chapter 2: Creating Tables for OpenL Tablets

This section describes how OpenL Tablets processes tables and provides reference information for each table type used in OpenL Tablets.

The following topics are included in this section:

- Table Recognition Algorithm
- Table Properties
- Table Types

Table Recognition Algorithm

This section describes the algorithm of how the OpenL Tablets engine looks for supported tables in Excel files. It is important to build tables according to requirements of this algorithm; otherwise the tables will not be recognized correctly.

OpenL Tablets utilizes Excel concepts of workbooks and worksheets. These can be represented and maintained in multiple Excel files. Each workbook is comprised of one or more worksheets used to separate information by categories. Each worksheet, in turn, is comprised of one or more tables. Workbooks can include tables of different types, each of which can support a different underlying logic.

The following is the general table recognition algorithm:

1. The engine looks into each spreadsheet and tries to identify logical tables.
   Logical tables must be separated by at least one empty row or column or start at the very first row or column. Table parsing is performed from left to right and from top to bottom. The first populated cell that does not belong to a previously parsed table becomes the top-left corner of a new logical table.
2. The engine reads text in the top left cell of a recognized logical table to determine its type.
   If the top left cell of a table starts with a predefined keyword then such table is recognized as an OpenL Tablets table.
   The following are the supported keywords:

Table type keywords	
Keyword	**Table type**
`Rules`	Decision Table
`Data`	Data Table
`Datatype`	Datatype Table
`Test`	Test Table
`Run`	Run Table
`Method`	Method Table
`Environment`	Configuration Table
`Properties`	Properties Table

Table type keywords	
Keyword	**Table type**
`Spreadsheet`	Spreadsheet Table
`ColumnMatch`	Column Match Table
`TBasic or Algorithm`	TBasic Table
`SimpleRules`	SimpleRules Table
`SimpleLookup`	SimlpeLookup Table
`TablePart`	Table Part

All tables that do not have any of the preceding keywords in the top left cell are ignored. They can be used as comments in Excel files.

3. The engine determines the width and height of the table using populated cells as clues.

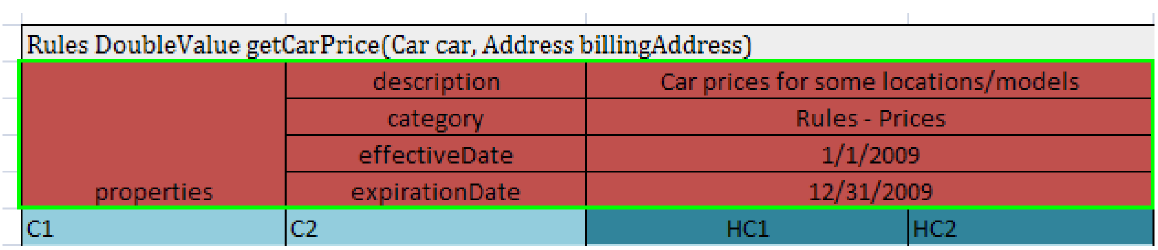
It is good practice to merge all cells in the first table row, so the first row explicitly specifies the table width. The first row is called the table **header.**

***Tip**: To put a table title before the header row, an empty row must be used between the title and the first row of the actual table.*

Table Properties

For all OpenL Tablets table types, except for Properties Table, Configuration Table and the **Other** type tables (not OpenL Tablets tables), properties can be defined as containing information about the table. A list of properties available in OpenL Tablets is predefined, and all values are expected to be of corresponding types. The exact list of available properties can vary between installations depending on OpenL Tablets configuration.

Table properties are displayed in the section which goes immediately after the table **header** and before other table contents. The properties section is optional and can be omitted in the table. The first cell in the properties row contains keyword “**properties**” and is merged across all cells in column if more than one property is defined. The number of rows in the properties section is equal to number of properties defined for the table. Each row in the properties section contains a pair of a property name and a property value in consecutive cells (2nd and 3rd columns).



Rules DoubleValue getCarPrice(Car car, Address billingAddress)			
properties	description	Car prices for some locations/models	
	category	Rules - Prices	
	effectiveDate	1/1/2009	
	expirationDate	12/31/2009	
C1	C2	HC1	HC2

Figure 4: Table properties example

Category and Module Level Properties

Table properties can be defined not only for each table separately, but for all tables in some category or a whole module. Separate Properties Table is designed to define this kind of properties. Only properties which are allowed to be inherited from category and/or module level can be defined in this table. Some properties, e.g., description, can only be defined for a table.

Besides Properties table, module level properties can also be defined in a name of Excel file corresponding to the module. For information how it works, refer to Properties from File Name chapter.

Properties defined at a category or module level can be overridden in tables. The priority of property values is following:

1. Table
2. Category
3. Module
4. Default value

Notice: *OpenL Tablets engine allows changing property values from application code when loading rules.*

Default Value

Some properties can have default values. The value is predefined and can be changed only in OpenL Tablets configuration. The default value is used if no property value is defined in the rule table or in the Properties table.

Properties defined by default are not added to the table's “properties” section and can only be changed in the right side **Properties** pane.

System Properties

System properties can only be set and updated by OpenL Tablets, not by the users. Currently OpenL Tablets WebStudio defines the following system properties: Created By / Created On and Modified By / Modified On (For details refer to *OpenL Tablets WebStudio User's Guide*).

Properties for Particular Table Type

There are properties to be used just for particular types of tables. It means that they make sense just for tables with special type and can be defined only for them. Almost all properties can be defined for Decision Tables except for the ‘Datatype Package’ property intended for Datatype Tables, the ‘Scope’ property used in Properties Tables and ‘Precision’ property designed for Test Tables.

OpenL Tablets checks applicability of properties and will produce error if property value is defined for table not intended to contain the property.

Applications using OpenL Tablets rules can utilize properties for different purposes. All properties are organized into the following groups:

- Business Dimension
- Version
- Info
- Dev

Properties of the Business Dimension and Version groups are used for table versioning. They are described in details in the following sections.

Table Versioning

In OpenL Tablets business rules can be versioned in different ways using Table properties. In this section we will consider the most popular of them:

- Business Dimension properties
- Active table properties

The first way is targeting advanced rules usage when several rule sets are used simultaneously; it is more extendable and flexible. The second way is more suitable for “what-if” analysis. Versioning by Business Dimension properties is described in the next section. Active table properties are discussed further in this document.

Business Dimension Properties

The properties of the “Business Dimension” group are used to version rules by *property values*. You will usually want to use this type of versioning if there are rules with the same “meaning”, but applied in different conditions. You can have in your project as many rules with the same name as you need; the system will select and apply the desired rule by its properties. E.g. calculating employees’ salary for different years can vary by some coefficients, have slight changes in the formula, or the both. In this case using Business Dimension properties enables you for every year apply appropriate rule version and get proper results.

The following table types support versioning by Business Dimension properties

- Decision (including SimpleRules and SimpleLookup)
- Spreadsheet
- TBasic
- Method
- ColumnMatch

If you deal with almost equal rules of the same structure but with slight differences – say, with changes in any specific date or state, – there is a very simple way to version your rule tables by Business Dimension properties. Just follow these steps:

1. Take the original rule table, set any dimension properties that indicate by which property you want to version the rules (it is possible to use more than one property).
2. Copy your original rule table, set new dimension properties for this table, and make changes in the table data as appropriate.
3. Repeat these steps if more rule versions are required.

Now you can call your rule by its name from any place in your project or application. Although you have multiple rules with the same name (but different Business Dimension properties), you should not worry about which rule will work. OpenL Tablets will review all the rules and choose the corresponding one according to the specified property values (or in developers’ language, by runtime context values).

The table below contains a list of Business Dimension properties used in OpenL Tablets.

Property	Name to be used in rule tables	Level at which a property can be defined	Type	Description
Effective / Expiration dates	effectiveDate expirationDate	Module, Category, Table	Date	Defines a time interval within which a rule table is active. The table becomes active on effective date and inactive after the expiration date. You can have multiple instances of the same table in the same module with different effective/expiration date ranges.
Start / End Request dates	startRequestDate endRequestDate	Module, Category, Table	Date	Defines a time interval within which a rule table is introduced in the system and is available for usage.
LOB (Line of Business)	lob	Module, Category, Table	String	Defines a LOB for a rule table - a business area for which the given rule works and should be used.
US Region	usregion	Module, Category, Table	Enum[]	Defines US regions.
Countries	country	Module, Category, Table	Enum[]	Defines countries.
Currency	currency	Module, Category, Table	Enum[]	Defines currencies.
Language	lang	Module, Category, Table	Enum[]	Defines languages.
US States	state	Module, Category, Table	Enum[]	Defines US States.
Region	region	Module, Category	Enum[]	Defines economic regions.

Note: For experienced users. There is a possibility of direct call of particular rule regardless of its dimension properties and current Runtime Context in OpenL Tablets. This feature is supported by setting the ID property (see the Dev Properties section for description) in some rule and using this identifier as the name of the function to call. During runtime, direct rule will be executed avoiding the mechanism of dispatching between overloaded rules.

Further in this section we provide illustrative and very simple examples of how to use Business Dimension properties.

Effective and Expiration Date

The following Business Dimension properties are intended for versioning business rules depending on some specific dates:

- *Effective Date* is the date as of which a business rule comes into effect and produces desired and expected results.
- *Expiration Date* is the date after which your rule is no longer applicable. If not defined, the rule works at any time on or after the Effective Date.

NOTE: If Expiration Date is not defined, the rule works at any time on or after the Effective Date.

The date *for which* the rule should be performed must fall into the Effective/ Expiration date time interval.

You can have multiple versions of the same rule table in the same module with different Effective/Expiration date ranges. But these dates cannot overlap with each other – i.e. if in one version of the rule your Effective/Expiration dates are 1.2.2010 – 31.10.2010, you should not create another version of that rule with Effective/Expiration dates within this dates frame.

For our example, we will take a rule for calculating a car insurance premium Quote. The rule is completely the same for different time periods except for some coefficient – a Quote Calculation Factor (hereinafter the 'Factor'). This factor is defined for each model of car.

In the examples below we will show how these properties enable you to specify which rule to apply for a particular date.

We will assume that you have a business rule for calculating the Quote for 2011 shown in the Figure 5. The Effective Date is 1/1/2011 and the Expiration Date is 12/31/2011.

SimpleRules DoubleValue Factor(String ModelOfCar)		
properties	effectiveDate	1/1/2011
	expirationDate	12/31/2011
Model of Car	Factor for Quote Calculation	
BMW	20	
Toyota	45	
Bentley	20	

Figure 5: Business rule for calculating a car insurance quote for 2011 year

However, we cannot use this rule for calculating the quote for 2012 year because the Factors for the cars differ from the previous year.

The rules name and their structure are the same but with different values of the Factor. Then it is a good idea to use versioning in the rules.

To create the rule for 2012 year:

1. Copy your rule table. In OpenL Tablets WebStudio, you can do it using the **Copy as New Business Dimension** feature (See the *Copying Tables* section in the *[OpenL Tablets WebStudio User Guide]()*).
2. Change your Effective and Expiration date to 1/1/2012 and 12/31/2012 respectively.
3. Replace the Factors as appropriate for 2012 year.

Your new table will look as shown in the Figure 6.

SimpleRules DoubleValue Factor(String ModelOfCar)		
properties	effectiveDate	1/1/2012
	expirationDate	12/31/2012
Model of Car	Factor for Quote Calculation	
BMW	25	
Toyota	40	
Bentley	15	

Figure 6: Business rule for calculating the same quote for 2012 year

To check how your rules work, let's test them for a certain car model and for particular dates, say, 10/5/2011 and 11/2/2012. The result of the test for BMW is shown in the Figure 7.

Testmethod Factor FactorForQuoteTest		
ModelOfCar	_context_.currentDate	_res_
Model of Car	Current Date	Factor
BMW	5/10/2011	20
BMW	11/2/2012	25

Figure 7: Selection of the Factor based on Effective / Expiration Dates

In this example, the date *on which* you should to perform calculation (on client's requirement) is shown in the **Current Date** column. In the first row for BMW, the Current Date is 10/5/2011, and since 10/5/2011> 1/1/2011 and 10/5/2011< 12/31/2011, the Factor for this date is '20'.

In the second row, the Current Date is 11/2/2012, and since 11/2/2012> 1/1/2012 and 11/2/2012< 12/31/2012, the Factor is 25.

Using Request Date

In some cases it is necessary to define additional time intervals for which your business rule is applicable. There are another two Table properties related to dates which can be used for selecting applicable rules. These properties have different meaning and work with slightly different logic compared to previous ones.

- *Start Request Date* is the date *when* your rule is introduced in the system and is available for usage.
- *End Request Date* is the date from which the system will not use the rule. If not defined, the rule can be used any time on or after the Start Request Date.

The date, *when* the rule is applied must be within the Start / End Request Date time interval. In OpenL Tablets rules this date is defined as a 'Request Date'.

NOTE: Pay attention to the difference between previous two properties: Effective / Expiration dates have meaning for what date your rules are applied. In contrast, Request dates mean when your rules are used (called from application).

You can have multiple rules with different Start /End Request dates, but being intersected in dates. In this case, the system will select the rule with the latest Start Request date. If there are rules with the same Start Request date OpenL Tablets will choose the rule with the earliest End Request date. And only in the case when Start /End Request dates coincide completely the system will show an error message.

NOTE: You cannot create a rule table version with exactly the same Start Request Dates or End Request Dates, because it will cause an error message.

NOTE: In some particular cases, Request Date is used to define the date when your business rule was called at the very first time.

Let's take another example where the additional date properties are defined: Start Request Date and End Request Date.

To demonstrate how these dates work, we will use the same rule for calculating a car insurance quote. But for some reason, you should enter the rule for 2012 year into the system in advance, say, from 12/1/2011. For that you can add to the rule the Start

Request Date = 12/1/2011as shown in the Figure 8. Adding this property tells OpenL Tablets the system can use the rule from the given Start Request date.

SimpleRules DoubleValue Factor(String ModelOfCar)		
properties	startRequestDate	12/1/2011
	endRequestDate	5/1/2012
	effectiveDate	1/1/2012
	expirationDate	12/31/2012
Model of Car	Factor for Quote Calculation	
BMW	25	
Toyota	45	
Bentley	20	

Figure 8: The rule for calculating the Quote is introduced from 12/1/2011

And let's assume that then you introduced a new rule with different Factors from 2/3/2012 as shown in the Figure 9.

SimpleRules DoubleValue Factor(String ModelOfCar)		
properties	startRequestDate	2/3/2012
	effectiveDate	1/1/2012
	expirationDate	12/31/2012
Model of Car	Factor for Quote Calculation	
BMW	35	
Toyota	35	
Bentley	20	

Figure 9: The rule for calculating the Quote is introduced from2.3.2011

However, the US legal regulations require you to use the same rules for premium calculations so you will need to stick to previous rules for older policies. In this case storing Request Date in application helps to solve the issue. By provided Request Date, OpenL Tablets will be able to choose rules available in the system on the designated date.

When you test your rules for BMW for particular request dates and effective dates, you will have the following result shown in the Figure 10:

Testmethod Factor FactorForQuoteTest1			
ModelOfCar	_context_.requestDate	_context_.currentDate	_res_
Model of Car	Request Date	Current Date	Factor
BMW	3/10/2012	10/5/2012	35
BMW	12/29/2012	10/15/2012	35
BMW	1/14/2012	8/16/2012	25

Figure 10: Selection of the Factor based on Start / End Request Dates

In this example, the dates *for which* the calculation is performed, are displayed in the **Current Date** column. The dates *when* you run your rule and perform the calculation is shown in the **Request Date** column.

Please pay attention on the row where Request Date is 3/10/2012 – this date falls in the both Start /End Request date intervals shown in Figures 8 and 9. But Start Request date in Figure 9 is later than the one defined in the rule from Figure 8. As a result, correct Factor value is 35.

So by using different sets of Business Dimension Properties you can flexible version your rules with keeping all of them in the system.

OpenL Tablets runs validation to check gaps and overlaps of properties values for versioned rules.

Active Table

Table versioning enables to store the previous versions of the same table of the rule in the same rules file. The active table versioning mechanism is based on two properties “version” and “active”. The “version” property should be different for each table and only one of them can have **true** as value for the “active” property.

All table versions should have the same identity that is exactly the same signature and dimensional properties values. Also table types should be the same.

An example of an inactive table version is as follows:

Rules DoubleValue driverRiskScore(String driverRisk)		
	version	0.0.1
	active	false
properties	category	Driver-Scoring
C1		**RET1**
risk == driverRisk		score
String risk		DoubleValue score
Driver Risk		**Score**
High Risk Driver		100
		0

Figure 11: An inactive table version

Info Properties

The Info group includes properties that provide any useful information; this group enables users to easily read and understand rule tables.

The table below provides a list of Info properties along with their brief description.

Property	Name to be used in rule tables	Level at which property can be defined and overridden	Type	Description
Category	category	Category, Table	String	The category of the table. By default, is equal to the name of the Excel sheet where the table is located. If the property level is specified as ‘Table’, defines category for the current table. Must be specified if scope is defined as ‘Category’ in a Properties table.

Description	description	Table	String	Description of a table, e.g. 'Car price for a particular Location/Model.' Any additional information to clarify the use of the table.
Tags	tags	Table	String[]	Can be used for search; there can be any number of comma-separated tags.
Created By	createdBy	Table	String	A name of a user created the table in OpenL Tablets WebStudio.
Created On	createdOn	Table	Date	Date of the table creation inOpenL Tablets WebStudio.
Modified By	modifiedBy	Table	String	A name of a user last modified the table in OpenL Tablets WebStudio.
Modified On	modifiedOn	Table	Date	The date of the last table modification in OpenL Tablets WebStudio.

Dev Properties

The Dev group has an impact on the OpenL Tablets features and enables to manage the system behavior depending on a property value. For example, the **Scope** property defines whether the properties are applicable for a particular Category of rules or for the Module. If Scope is defined as Module, the properties will be applied for all tables in the current module. If Scope is defined as Category, then you should use the Category property to specify for which exact category the property is applicable, as shown in the Figure 12.

Properties catPolicyScoring	
scope	Category
category	Policy-Scoring
lob	category_Policy-Scoring_Lob

Figure 12 : The properties are defined for the 'Police-Scoring' category

The Dev group properties are listed in the following table.

Property	Name to be used in rule tables	Type	Table type	Level at which property can be defined	Description
ID	id	Table	All	Table	The property defines unique ID to be used for calling a particular table in a set of overloaded tables without using business dimension properties. **NOTE**: Constraints for the ID value are the same as for any OpenL function.

Build Phase	buildPhase	String	All	Module, Category, Table	The property is used to manage dependencies between build phases. **NOTE**: Will be used in future versions.
Validate DT	validateDT	String	Decision Table	Module, Category, Table	The property specifies the validation mode for Decision Tables. In a wrong case an appropriate warning is issued.Possible values: on – checks if there are any uncovered or overlapped cases; off — the validation is turned off; gap – checks if there are uncovered cases; overlap - checks if there are overlapped cases.
Fail On Miss	failOnMiss	Boolean	Decision Table	Module, Category, Table	The property defines a rule behavior in case no rules were matched. If the property is set to `TRUE` an error occurs along with the corresponding explanation. If `FALSE,` the table output is set to `NULL`.
Scope	scope	String	Properties	Module, Category	The property defines the scope for a Properties table.
Datatype Package	datatypePackage	String	DataType	Table	The property defines the name of the Java package for generating the datatype.
Recalculate	recalculate	Enum		Module, Category, Table	The property defines the way of a table recalculation for a variation. Possible values: Always/Never/Analyze.
Cacheable	cacheable	Boolean		Module, Category, Table	The property defines whether or not to use cache while recalculating the table, depending on the rule input.
Precision	precision	Integer	Test Table	Module, Category, Table	The property specifies precision of comparing the returned results with expected ones while launching Test Tables.

Variation Related Properties

This section provides more information about *variations* and the properties required to work with them, namely — *Recalculate* and *Cacheable*.

A variation means “An additional calculation of the same rule with a modification in its arguments”. Variations are very useful when you need to calculate a rule several times with similar arguments. The idea of this approach is to calculate once the rules for a particular set of arguments and then recalculate only the rules or steps that depend on the specifically modified (by variation) fields in those arguments.

The following Dev properties are designed to manage rules recalculation for variations.

- **Cacheable —** The property switches on/off using cache while recalculating the table. Can be evaluated to true or false. If true, the all calculation results of the rule will be cached and can be used in other variations, otherwise calculation results will not be cached. It’s recommended to set Cacheable to true if you suggest recalculating a rule with the same input parameters. In this case OpenL does not recalculate the rule; instead, it retrieves the results from the cache.
- **Recalculate —** The property explicitly defines the recalculation type of the table for a variation. It can take the following values: *always*, *never* or *analyze*.

If the Recalculate property is set to **Always** for a rule, the rule will be entirely recalculated for a variation. This value is useful for rule tables which are supposed to be recalculated.

If the Recalculate property is set to **Never** for a rule, the system will not recalculate the rule for a variation. You can set it for rules which new results you are not interested in and which are not necessary for a variation.

As for the **Analyze** value, it should be used for top level rule tables to ensure recalculation of the included rules with the Always value. The included table rules with the Never value will be ignored.

By default, the properties are set as follows:

```
recalculate = always;
cacheable = false.
```

To provide an illustrative example of how to use Variation Related Properties, we will take the Spreadsheet rule **DwellPremiumCalculation** (see the figure below) which calculates a home insurance premium Quote. The quote includes calculations of Protection and Key factors which values are dependent on Coverage A limit (see **ProtectionFactor** and **KeyFactor** simple rules). The insurer wants to vary Coverage A limit of the Quote and observe how limit variations impact on Key factor exactly.

The DwellPremiumCalculation is a top level rule and during recalculation of the rule we are interested in some of the results only. That’s why we should define recalculation type (the “recalculate” property) as **Analyze** for this rule.

As the interest of the insurer is to get a new value of Key factor for a new Coverage A limit value, recalculation type of the KeyFactor rule should be determined as **Always**.

On the contrary, Protection factor is not interesting for the insurer, so the ProtectionFactor rule is not required to be recalculated. To optimize the recalculation process, recalculation type of the rule should be set up as **Never**. Moreover, other rules tables like the BaseRate rule which are not required to be recalculated should have “recalculation” property as Never too.

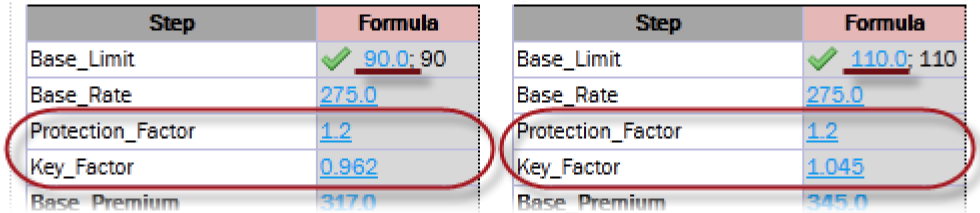
preadsheet SpreadsheetResult **DwellPremiumCalculation** (Policy policy, Dwell dwell	
properties recalculate	analyze
Step	**Formula**
Base_Limit	= coverages[!@ coverageType == "Coverage A"].limit
Base_Rate	= BaseRate (territoryCd, policyForm, policyPlan)
Protection_Factor	= ProtectionFactor (protectionClass, $Base_Limit)
Key_Factor	= KeyFactor ($Base_Limit)
Base_Premium	**= round(product ($Base_Rate:$Key_Factor))**

SimpleLookup DoubleValue **ProtectionFactor** (ProtectionClas		
properties	recalculate	never
Protection Class / Limit	<= 100	> 100
1	0.8	1
2	0.9	1
3	1	1
8B	1.2	1.3
9	1.2	1.4
10	1.5	1.5

SimpleRules DoubleValue **KeyFactor** (DoubleValue lim		
properties	recalculate	always
CoverageA Amount	**Key Factor**	
0 - 75	0.923	
75 - 80	0.933	
80 - 85	0.948	
85 - 90	0.962	
90 - 95	0.981	
95 - 100	1	
100 - 105	1.023	
105 - 110	1.045	
110 - 115	1.072	

Figure 13: Usage of Variation Related Properties

Let Coverage A limit of the quote is 90, Protection Class is 9. A modified value of Coverage A limit for a variation is going to be 110. Then the following Spreadsheet results (after the first calculation and the second recalculation) are obtained:

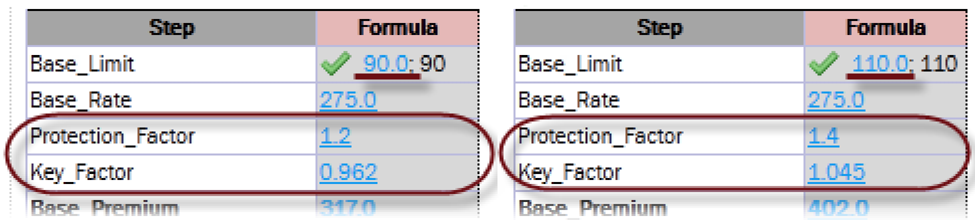


Step	Formula	Step	Formula
Base_Limit	90.0: 90	Base_Limit	110.0: 110
Base_Rate	275.0	Base_Rate	275.0
Protection_Factor	1.2	Protection_Factor	1.2
Key_Factor	0.962	Key_Factor	1.045
Base_Premium	317.0	Base_Premium	345.0

Figure 14: Results of DwellPremiumCalculation with recalculation = Analyze

You can notice that Key factor is recalculated, but Protection factor remains the same (Protection factor of initial Coverage A limit).

If you define recalculation type of DwellPremiumCalculation as Always, OpenL Tablets will ignore (will not analyze) recalculation types of nested rules and recalculate all cells as shown on the figure below:



Step	Formula	Step	Formula
Base_Limit	90.0: 90	Base_Limit	110.0: 110
Base_Rate	275.0	Base_Rate	275.0
Protection_Factor	1.2	Protection_Factor	1.4
Key_Factor	0.962	Key_Factor	1.045
Base_Premium	317.0	Base_Premium	402.0

Figure 15: Results of DwellPremiumCalculation with recalculation = Always

Precision Property usage in Testing

This section provides more information about *how to use precision property*. The property is aimed to be used for testing purpose.

There are some cases when it is impossible or not needed to define exact numeric value of an expected result in Test Tables. For example, non-terminating rational numbers such as π (3.1415926535897…) must be approximated so that it could be written in a cell of a table.

Property Precision is used as a measure of the accuracy of the expected value to the returned value to a certain precision. Let's precision of the expected value *A* is *N*. Then expected value *A* is true only if

$$|A - B| < 1/10^N, \text{ where B – returned value.}$$

It means that if the expected value is close enough to the returned value then the expected value is considered to be true.

Let's look at the following examples. A simple rule FinRatioWeight has 2 tests FinRatioWeightTest1 and FinRatioWeightTest2:

s DoubleValue **FinRatioWeight** (FinancialRatio fin	
Financial Ratio	**Financial Ratio Weight**
Cash Liquidity Ratio	0.111207645
Quick Ratio	0.054117651
Current Ratio	0.420000001
Operating Profit Margin	0.414674703

Figure 16: An example of Simple Rule

The first Test table has Precision property defined with value 5:

Test **FinRatioWeight** FinRatioWeightTest1		
properties	precision	5
financialRatio	_res_	
Financial Ratio	**Financial Ratio Weight**	
Cash Liquidity Ratio	0.11121358	
Quick Ratio	0.05410091	

FinRatioWeightTest1 2 test cases 1

Financial Ratio	Financial Ratio Weight
Cash Liquidity Ratio	✔ 0.111207645
Quick Ratio	✘ 0.054117651 Expected: 0.05410091

Figure 17: An example of Test with precision defined

As a result of launching this Test, the first test case is passed because $|0.11121358 - 0.111207645| = 0.5935*10^{-5} < 0.00001$; but the second is failed because $|0.05410091 - 0.054117651| = 1.6741*10^{-5} > 0.00001$.

OpenL Tablets allows us to specify precision for a particular column which contains expected result values using syntax

`_res_ (N)` or `_res_.$<ColumnName>$<RowName> (N)`

as it's shown in the second test table FinRatioWeightTest2:

Test **FinRatioWeight** FinRatioWeightTest2	
financialRatio	_res_ (2)
Financial Ratio	**Financial Ratio Weight**
Current Ratio	0.42
Operating Profit Margin	0.41

FinRatioWeightTest2 2 test cases

Financial Ratio	Financial Ratio Weight
Current Ratio	0.420000001
Operating Profit Margin	0.414674703

Figure 18: An example of Test with precision for the column defined

This possibility is required in cases when the results of the whole Test are taking into account with one accuracy, but some separate expected results – with others.

A precision defined for the column has higher priority than a precision defined at the Table level.

Precision can be zero or a negative value also (any Integer value).

Properties from File Name

Table properties can be defined for all tables of a module (module level properties) in a file name of the module. For that 2 conditions must be met:

- a **file name pattern** is configured – directly in a rules project descriptor (`rules.xml` file) as *properties-file-name-pattern* tag or via WebStudio as '*Properties pattern for a file name*' on the Project page;
- module file name matches the pattern.

The file name pattern can include:

- text symbols;
- table property names enclosed in '%' marks;
- if a table property value is supposed to be Date then Date format should be also specified right after the property name and colon:

```
...<text>%<property name>%<text>%<property name>:<date format>%...
```

Note: Date formats description and examples can be found in Date and Time Patterns.

Auto Rating project below is configured so that a user can specify values for properties **US State** and **Effective date** via a file name for a whole module:

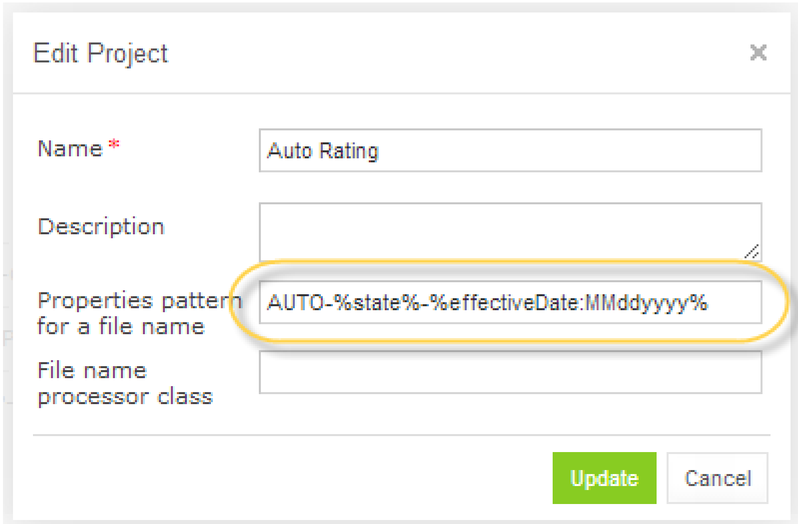


Figure 19: File name pattern configured via WebStudio

```
    </modules>
    <properties-file-name-pattern>AUTO-%state%-%effectiveDate:MMddyyyy%</properties-file-name-pattern>
</project>
```

Figure 20: File name pattern in a rules project descriptor directly

For instance, for the module of *Auto Rating* project with file name **AUTO-FL-01012014.xlsx** the following module properties: **US State= 'Florida'**, **Effective date = 01 Jan 2014** - will be retrieved and inherited by module tables.

If a file name doesn't match the pattern then module properties are not defined.

A user can't specify the same property both in a file name and Properties table of the module.

Note: For experienced users. A default implementation of properties definition in file name is described. A user can redefine this implementation by a custom one specifying their own file name processor class in a rules project descriptor.

Table Types

OpenL Tablets employs the following table types:

- Decision Table
- Datatype Table
- Data Table
- Test Table
- Run Method Table
- Method Table
- Configuration Table
- Properties Table
- Spreadsheet Table
- Column Match Table
- TBasic Table
- Table Part

Decision Table

A **decision table** contains a set of rules describing decision situations where the state of a number of conditions determines the execution of a set of actions. It is the basic table type used in OpenL Tablets decision making.

The following topics are included in this section:

- Decision Table Structure
- Lookup Tables
- Simple Decision Tables
- Decision Table Interpretation
- Local Parameters in Decision Table
- Transposed Decision Tables
- Representing Arrays
- Representing Date Values
- Representing Boolean Values
- Ranges types in OpenL
- Using Calculations in Table Cells

Decision Table Structure

The following is an example of a decision table:

	A	B	C	D	E
1					
2		Rules String Hello (Integer hour)			
3		properties	description	New test Decision table	
4			lang	ENG	
5		Rule	C1	C2	RET1
6			min <= hour	hour <= max	greeting
7			Integer min	Integer max	String greeting
8		Rule	From	To	Greeting
9		R10	0	11	Good Morning
10		R20	12	17	Good Afternoon
11		R30	18	21	Good Evening
12		R40	22	23	Good Night
13					

Figure 21: Decision table

The following table describes its structure:

Decision table structure		
Row number	**Mandatory**	**Description**
1	Yes	Table header, which has the following pattern: `<keyword> <rule header>` where `<keyword>` is either 'Rules' or 'DT' and `<rule header>` is a signature of a method used to access the decision table and provide input parameters. .

<table>
<tr><th colspan="3">Decision table structure</th></tr>
<tr><th>Row number</th><th>Mandatory</th><th>Description</th></tr>
<tr><td>2 and 3</td><td>No</td><td>Rows containing table properties. Each application using OpenL Tablets rules can utilize properties for different purposes.
Although the provided decision table example contains two property rows, there can be any number of property rows in a table.</td></tr>
<tr><td>4</td><td>Yes</td><td>Row consisting of the following cell types:
<table>
<tr><th>Type</th><th>Description</th><th>Examples</th></tr>
<tr><td>Condition column header</td><td>Identifies that the column contains rule condition and its parameters. It must start with character 'C' followed by a number.</td><td><code>C1, C5, C8</code></td></tr>
<tr><td>Horizontal condition column header</td><td>Identifies that the column contains horizontal rule condition and its parameters. It must start with character 'HC' followed by a number.
Horizontal conditions are used in lookup tables only.</td><td><code>HC1, HC5, HC8</code></td></tr>
<tr><td>Action column header</td><td>Identifies that the column contains rule actions. It must start with character 'A' followed by a number.</td><td><code>A1, A2, A5</code></td></tr>
<tr><td>Return value column header</td><td>Identifies that the column contains values to be returned to the calling program. A table can have multiple return columns, however, only the first fired not blank value is returned.</td><td><code>RET1</code></td></tr>
</table>
All other cells in this row are ignored and can be used as comments.
If a table contains action columns, the engine executes actions for all rules whose conditions are true. If a table has a return column, the engine stops processing rules after the first executed rule. If a return column has a blank cell and the rule is executed, the engine does not stop but continues checking rules in the table.</td></tr>
</table>

<table>
<tr><th colspan="3">Decision table structure</th></tr>
<tr><th>Row number</th><th>Mandatory</th><th>Description</th></tr>
<tr><td>5</td><td>Yes</td><td>Row containing cells with expression statements for condition, action, and return value column headers. OpenL Tablets supports Java grammar enhanced with OpenL Business Expression (BEX) grammar features. For information on the BEX language, see Appendix A: BEX Language Overview.
In most cases OpenL Business Expression grammar will cover all the variety of expression statements and an OpenL user will not need to learn Java syntax.
Code in these cells can use any Java objects and methods visible to the OpenL Tablets engine as elsewhere. For information on enabling the OpenL Tablets engine to use custom Java packages, see Configuration Table.
Purpose of each cell in this row depends on the cell above it as follows:
<table>
<tr><th>Cell above</th><th>Purpose</th></tr>
<tr><td>Condition column header</td><td>Specifies the logical expression of the condition. It can reference parameters in the method header and parameters in cells below.
The cell can contain several expressions, but the last expression must return a Boolean value. All condition expressions must be true to execute a rule.</td></tr>
<tr><td>Horizontal condition</td><td>The same to Condition column header.</td></tr>
<tr><td>Action column header</td><td>Specifies expression to be executed if all conditions of the rule are true. The expression can reference parameters in the rule header and parameters in cells below.</td></tr>
<tr><td>Return value column header</td><td>Specifies expression used for calculating the return value. The type of the last expression must match the return value specified in the rule header. The explicit return statement with keyword ‘return’ is also supported.
This cell can reference parameters in the rule header and parameters in cells below.</td></tr>
</table></td></tr>
<tr><td>6</td><td>Yes</td><td>Row containing parameter definition cells. Each cell in this row specifies the type and name of parameters in cells below it.
Parameter name must be one word corresponding to Java identification rules.
Parameter type must be one of the following:
• simple data types
• aggregated data types or Java classes visible to the engine
• one-dimensional arrays of the above types as described in Representing Arrays</td></tr>
<tr><td>7</td><td>Yes</td><td>Descriptive column titles. The rule engine does not use them in calculations but they are intended for business users working with the table. Cells in this row can contain any arbitrary text and be of any layout that does not correspond to other table parts. The height of the row is determined by the first cell in the row.</td></tr>
<tr><td>8 and below</td><td>Yes</td><td>Concrete parameter values. Any cell can contain expression instead of concrete value and calculate the value. This expression can reference parameters in rule header and any parameters of condition columns.</td></tr>
</table>

Lookup Tables

Lookup table is a special modification of Decision table which simultaneously contains vertical and horizontal conditions and returns value on crossroads of matching condition values.

That means condition values could appear either on the left of the lookup table or on the top of it. The values on the left are called "vertical" and values on the top are called "horizontal".

The Horizontal Conditions are marked as HC1, HC2, etc. Every lookup matrix should starts from HC or RET column. The first HC or RET column should go after all vertical conditions (C, Rule, comment, etc columns). RET section can be placed in any place of lookup headers row. HC columns do not have Titles section.

Lookup table must have:

- at least one vertical condition (C)
- at least one horizontal condition (HC)
- exactly one return column (RET)

Lookup table can have:

- Rule column

Lookup table cannot have comment column in horizontal conditions part.

Rules DoubleValue **CarPrice** (Car car, Address billingAddress)				
C1	C2	HC1	HC2	RET1
country	region	brand	model	
Country	String	CarBrand	String	
Country	**Region**	**BMW**		
		Z4 sDrive35i	**Z4 sDrive30i**	
USA	Pacific West	$51,650	$45,750	
USA	West	$52,000	$44,050	
USA	Mid Atlantic	$52,450	$46,550	
GreatBritain	England	$53,650	$47,750	
GreatBritain	Wales	$53,650	$47,750	
GreatBritain	Scotland	$53,650	$47,750	

Figure 22: A lookup table example

Colors identify how values are related to conditions. The same table represented as a decision table follows:

Rules DoubleValue **CarPrice** (Car car, Address billingAddress)				
C1	C2	C3	C4	RET1
country	region	brand	model	
Country	String	CarBrand	String	
Country	**Region**	**Brand**	**Model**	**Price**
USA	Pacific West	BMW	Z4 sDrive35i	$51,650
USA	West	BMW	Z4 sDrive35i	$52,000
USA	Mid Atlantic	BMW	Z4 sDrive35i	$52,450
GreatBritain	England	BMW	Z4 sDrive35i	$53,650
GreatBritain	Wales	BMW	Z4 sDrive35i	$53,650
GreatBritain	Scotland	BMW	Z4 sDrive35i	$53,650
USA	Pacific West	BMW	Z4 sDrive30i	$45,750
USA	West	BMW	Z4 sDrive30i	$44,050
USA	Mid Atlantic	BMW	Z4 sDrive30i	$46,550
GreatBritain	England	BMW	Z4 sDrive30i	$47,750
GreatBritain	Wales	BMW	Z4 sDrive30i	$47,750
GreatBritain	Scotland	BMW	Z4 sDrive30i	$47,750

Figure 23: Lookup table representation as a decision table

Implementation Details

(Just for your information. This passage can be interesting to understand internal OpenL Tablets logic.)

At first the table goes through parsing and validation. On parsing all parts of the table such as header, columns headers, vertical conditions, horizontal conditions, return column and their values are extracted. On validation OpenL checks if the table structure is proper.

Then OpenL will transform Lookup table to regular Decision Table internally and will process it as regular Decision Table.

Simple Decision Tables

Practice shows that most of decision tables have simple structure: there are conditions for each input parameter of a decision table (that check equality of input and condition values) and a return column. Because of this fact, OpenL Tablets have a simplified decision table representation. Simple decision table allows skipping condition and return columns declarations, and the table will consist of a header, properties (optional), column titles and condition/return values. The only restriction for a simple decision table is that condition values must be of the same type or be an array/range of the same type as input parameters and return values must have the type of the return type from the decision table header.

Simple Rules Table

Usual decision table which has simple conditions for each parameter and simple return can be easily represented as SimpleRules table.

Header format:

```
SimpleRules <Return type> RuleName(<Parameter type 1> parameterName1,
(<Parameter type 2> parameterName 2….)
```

SimpleRules Table Example

SimpleRules InjuryRating VehicleInjuryRating (BodyType bodyType, AirbagType airbagType, Boolean hasRollBar)			
Body Type	**Airbags**	**Roll Bar**	**Injury Rating**
Convertible		No	Extremely High
	No		Extremely High
	Driver		High
	Driver&Passenger		Moderate
	Driver&Passenger&Side		Low

Figure 24: SimpleRules table example

SimpleLookup Table

Usual lookup decision table with simple conditions that check equality of an input parameter and a condition value and a simple return can be easy represented as SimpleLookup table. This table is similar to SimpleRules table but has horizontal conditions. Number of parameters that will be associated with horizontal conditions is determined by the height of the first column title cell.

Header format:

```
SimpleLookup <Return type> RuleName(<Parameter type 1> parameterName1,
(<Parameter type 2> parameterName2,….)
```

SimpleLookup Table Example

SimpleLookup DoubleValue getCarPriceSimple(Country countryName, String regionName, CarBrand carBrand, String carModel)					
		BMW	BMW	Porche	Porche
Country	Region	Z4 sDrive35i	Z4 sDrive30i	911 Carrera 4S	911 Targa 4
USA	Pacific West	$51,650	$45,750	$93,200	$90,400
USA	West	$52,000	$44,050	$93,200	$90,400
USA	Mid Atlantic	$52,450	$46,550	$93,200	$90,400
GreatBritain	England	$53,650	$47,750	$94,200	$91,400
GreatBritain	Wales	$53,650	$47,750	$95,200	$92,400
GreatBritain	Scotland	$53,650	$47,750	$96,200	$93,400
Belarus	Minsk	$56,650	$49,750	$93,200	$90,400
Belarus	Vitebsk	$56,650	$49,750	$93,200	$90,400
Belarus	Grodna	$56,650	$49,750	$93,200	$90,400

Figure 25: SimpleLookup table example

Ranges and Arrays in Simple Decision Tables

You can use Range and Array data types in SimpleRule tables and SimpleLookup tables. If a condition is represented as an Array or Range then the rule will be executed for any value from that array or range. As an example, in Figure 17, there is the same Car Price for all regions of Belarus and Great Britain, so, using an array, we can replace three rows for each of these countries by a single one as shown in Figure 18.

SimpleLookup DoubleValue getCarPriceSimpleArray1(Country countryName, String regionName, CarBrand carBrand, String carModel)					
		BMW	BMW	Porche	Porche
Country	Region	Z4 sDrive35i	Z4 sDrive30i	911 Carrera 4S	911 Targa 4
USA	Pacific West	$51,650	$45,750	$93,200	$90,400
USA	West	$52,000	$44,050	$93,200	$90,400
USA	Mid Atlantic	$52,450	$46,550	$93,200	$90,400
GreatBritain	England,Wales,Scotland	$53,650	$47,750	$94,200	$91,400
Belarus	Minsk,Vitebsk,Grodna	$56,650	$49,750	$93,200	$90,400

Figure 26: SimpleLookup table with an array

NOTE: If a string array element contains a comma, the element must be delimited with the backslash (\) separator forwarded by the comma as for **Driver, Passenger, Side** in the example below.

SimpleRules String vehicleInjuryRating(Str	
Body Type	**Airbags**
Convertible	
	No
	Driver
	Driver \, Passenger
	Driver \, Passenger \, Side

Figure 27: Comma within an array element in Simple Rule table

The example in Figure 20 shows how to use a Range in a SimpleRule table.

SimpleRules RegionRisk **Region** (Integer vehicleZip)	
ZIP Code	**Region Risk Value**
10001 .. 10027	1
10598	
21854	
22859	
23401	2
23402 .. 23409	
24603	
24700	
24701	3
24800	
24803	4
25200	10
31200	12

Figure 28: SimpleRules table with a Range

OpenL looks through the Condition column (**ZIP Code**), meets a range (not necessary the first one) and defines that all the data in the column are IntRange, where Integer is defined in the header (Integer vehicleZip).

We cannot use a range and an array in the same Condition column (OpenL issues an exception if so).

Decision Table Interpretation

Rules inside decision tables are processed one by one in the order they are placed in the table. A rule is executed only when all its conditions are true. If at least one condition returns false, all other conditions in the same row are ignored. Absence of a parameter in a condition cell is interpreted as a true value. Blank action and return value cells are ignored.

Local Parameters in Decision Table

When declaring Decision table users have to put next info in header: column type, code snippet, declarations of parameters, titles.

Recent experience shows that in 95% of cases users put very simple logic in code snippet such as just access to some field from input parameters. In this case the parameters declarations for the column are overhead and are useless.

Simplified declarations

Case#1

The following image represents situation when user should provide expression and simple equal operation for condition declaration.

Rules String test4(boolean hadTraining)	
C1	RET1
hadTraining == localParam	eligibility
boolean localParam	String eligibility
Training	Eligibility
No	Not Eligible
	Eligible

Figure 29: Decision Table where user should provide expression and simple equal operation for condition declaration

This code snippet can be simplified as shown on the next image.

Rules String test4(boolean hadTraining)	
C1	RET1
hadTraining	eligibility
	String eligibility
Training	Eligibility
No	Not Eligible
	Eligible

Figure 30: Simplified Decision Table

How it works?

(Just for your information. This passage can be interesting to understand internal OpenL Tablets logic.)

OpenL engine creates required parameter automatically when user omits parameter declaration with the following information:

1. Parameter name will be "P1", where 1 is index of parameter
2. Type of parameter will be the same as expression type (in our case it will be Boolean)

In the next step OpenL will create appropriate condition evaluator.

Case#2

The following image represents situation when user can omit parameter name in declaration.

Rules String test2(String ageType)	
C1	RET1
P1.equals(ageType)	eligibility
String	String eligibility
Driver	Eligibility
Young Driver	Not Eligible
Senior Driver	Not Eligible
	Eligible

Figure 31: Decision Table where user can omit name in declaration

As mentioned in previous case OpenL engine generates parameter name and user can use it in expression but in this case user must provide local parameter type because expression type has different type than parameter.

Transposed Decision Tables

Sometimes decision tables look more convenient in transposed format where columns become rows and rows become columns. For example, a transposed version of the previously shown decision table resembles the following:

Rules String hello(int hour)							
Rule			**Rule**	R10	R20	R30	R40
C1	min <= hour	int min	**From**	0	12	18	22
		int max	**To**	11	17	21	23
A1	System.out.println(greeting+", World!")	String greeting	**Greeting**	Good Morning	Good Afternoon	Good Evening	Good Night

Figure 32: Transposed decision table

OpenL Tablets automatically detects transposed tables and is able to process them correctly.

Representing Arrays

For all tables that have properties of the `enum[]` type or fields of the array type, arrays can be defined as follows:

- horizontally
- vertically
- comma separated arrays

The first option is to arrange array values horizontally using multiple subcolumns. The following is an example of this approach:

String[] set				
Number Set				
1	3	5	7	9
2	4	6	8	

Figure 33: Arranging array values horizontally

In this example, the contents of the `set` variable for the first rule are `[1,3,5,7,9]` and for the second rule `[2,4,6,8]`. Values are read from left to right.

The second option is to present parameter values vertically as follows:

	String[] set
#	**Number Set**
1	1
	3
	5
	7
	9
2	2
	4
	6
	8

Figure 34: Arranging array values vertically

In the second case, the boundaries between rules are determined by the height of the leftmost cell. Therefore, an additional column must be added to the table to specify boundaries between arrays.

In both cases, empty cells are not added to the array.

The third option is to define an array separating values by a comma. If the value itself contains a comma, it must be escaped using back slash symbol “\” by putting it before the comma.

Data Policy policyProfile4			
properties	category	Policy-Data	
name		**Policy**	Policy4
drivers	>driverProfiles3	**Drivers**	test1,test3\,4,test2
vehicles	>autoProfiles3	**Vehicles**	1965 VW Bug
clientTier		**Client Tier**	Elite
clientTerm		**Client Term**	Long Term

Figure 35: Array values separated by comma

In this example, the array consists of the following values:

- test 1
- test 3,4
- test 2

Rules String hello2(String income1, String income2)		
C1	C2	R
array1	contains(array2, income2)	g
String[] array1	String[] array2	S
Array1	**Array2**	G
firstValue	value1, value2, value3	
secondValue		
value1	singleValue	
value2		
value3		

Figure 36: Array values separated by comma. The second example

In this example, the array consists of the following values:

- value1
- value2
- value3

Representing Date Values

To represent date values in table cells either Excel format or the following format for text must be used:

```
'<month>/<date>/<year>
```

The value must always be preceded with an apostrophe to indicate it is text. Excel treats these values as plain text and does not convert to any specific date format.

The following are valid date value examples:

```
'5/7/1981
'10/20/2002
‘10/20/02
```

OpenL Tablets recognizes all Excel date formats.

Representing Boolean Values

OpenL Tablets supports either Excel Boolean format or the following formats of Boolean values as text:

- true, yes, y
- false, no, n

OpenL Tablets recognizes the Excel Boolean value, such as native Excel Boolean value TRUE or FALSE. For more information on Excel Boolean values, see Excel help.

Range types in OpenL

In OpenL there are the following Data types designed to work with ranges:

- IntRange
- DoubleRange

For more information, please see Range Data Types.

Using range types in Decision Tables

For example, we have the next Decision table. The column of type IntRange contains expression statement of type Integer. So the cell may contain value of different types. When using IntRange, for user convenience it is possible to have expression statement cell of following types:

- Byte
- Short
- Int

Rules String ClassifyIncome(String incomeType, short incomeClass)		
C1	C2	RET1
incomeType	incomeClass	
	IntRange	
Type	**Class**	**Rate**
Type 1	< -200	rule1
Type 1	< 100	rule2
Type 2	[-100 .. -20)	rule3
Type 2		rule4

Figure 37: Decision table with IntRange

Be careful with using `Integer.MAX_VALUE` in Decision table. If there is a range with `max_number` equal to `Integer.MAX_VALUE` (e.g. [100; 2147483647]) it won`t be included to range. It is a known limitation.

When using DoubleRange, for user convenience it is possible to have code statement cell of the following types:

- Byte
- Short
- Integer
- Long
- Float
- Double

Using Calculations in Table Cells

OpenL Tablets can perform mathematical calculations involving method input parameters in table cells. For example, instead of returning a concrete number, a rule can return a result of a calculation involving one of the input parameters. Calculation result type must match the type of the cell. Text in cells containing calculations must start with an apostrophe followed by =. Excel treats such values as plain text. Alternatively, OpenL Tablets code can be enclosed by `{ }`.

The following decision table demonstrates calculations in table cells:

Rules int ampmTo24(int ampmHr, String ampm)		
C1	**C2**	**RET1**
range.contains(ampmHr)	suffix.equals(ampm)	result
IntRange range	String suffix	int result
AM/PM hour	**AM or PM**	**24 hour**
12	AM	0
1-11	AM	=ampmHr
12	PM	12
1-11	PM	=ampmHr+12

Figure 38: Decision table with calculations

The table transforms a 12 hour time format into a 24 hour time format. The column `RET1` contains two cells that perform calculations with the input parameter `ampmHr`.

Calculations use regular Java syntax, similar to what is used in conditions and actions.

Note: Excel formulas are not supported by OpenL Tablets. They are used as precalculated values.

Datatype Table

Description

A **Datatype table** defines an OpenL Tablets data structure. Datatype table allows you to:

- create a hierarchical data structure where you combine multiple data elements and their associated datatypes in hierarchy;

-- create vocabulary for data elements.

A data type defined by Datatype Table is called a Custom Data Type. Using Datatype tables a user can create their own data model which is logically suited for usage in particular business domain.

On How to create vocabulary for data elements, please refer to section Alias Data Types.

Datatype Table has the following structure:

1. The first row is the header containing the keyword 'Datatype' followed by the name of the data type.
2. Every row, beginning with the second one, represents one attribute of the data type. The first column contains attribute types; the second column contains corresponding attribute names.
3. There is an optional 3rd column, which defines default values for fields.

For example, let's consider the case when we need to create hierarchical logical data structure. The following is an example of a Datatype table defining a custom data type called **Person**. The table represents a structure of data object Person and combines Person's data elements such as name, social security number, date of birth, gender and address:

Datatype Person	
String	name
String	ssn
Date	dob
Gender	gender
Address	address

Figure 39: Datatype table Person

Note that data attribute (element) address of Person has, by-turn, custom datatype **Address** and consists of zip code, city and street attributes:

Datatype Address	
String	zipCode
String	city
String	street

Figure 40: Datatype table Address

The following example extends the data type Person with default values for some fields:

Datatype Person		
String	name	
String	ssn	
Date	dob	
Gender	gender	Male
Address	address	

Figure 41: Datatype table with default values

Field ‘gender’ will have the given value ‘Male’ for all newly created instances, if other value is not provided.

Note: For experienced users. Java beans can be used as custom data types in OpenL Tablets. If a Java bean is used, the package where the Java bean is located must be imported using a configuration table as described in Configuration Table.

Inheritance in Data types

There is a possibility to inherit one data type from another in OpenL Tablets. New data type that inherits from another one will contain all fields defined in the parent data type. If a child datatype defines fields that are already defined in the parent data type, you will get warnings or errors (in case the same field is declared with different types in the child and the parent data type).

To specify inheritance, the following header format is used in Datatype Table:

```
Datatype <TypeName> extends <ParentTypeName>
```

Alias Data Types

Alias data types allows a user to define a list of possible values for a particular data type, what is in other words – to create a vocabulary for data.

This is how you create Alias datatype:

1. the 1st row is header: starting with keyword “Datatype”, followed by name of the Alias datatype, and having predefined datatype in angle brackets on the basis of which the alias datatype is created at the end
2. the 2nd and following rows list values of the Alias datatype, values can be of the indicated predefined datatype only

You may have noticed on the example above that data type Person has an attribute gender of data type **Gender** which is the following alias data type:

Datatype Gender <String>
Male
Female

So, data of Gender Data type can only be ‘Male’ or ‘Female’.

OpenL Tablets checks each data of alias datatype whether its value is in the defined list of possible values or not. In case the value is outside of the valid domain (vocabulary) OpenL Tablets shows appropriate error. Usage of alias datatypes provides data integrity and ensures a user that he doesn't make any accident mistakes in rules.

Data Table

A **data table** contains relational data that can be referenced by its table name from other OpenL Tablets tables or Java code as an array of data.

Data tables are widely used during testing rules process when a user defines all input test data in data tables and reuse them in several test tables of a project (by referencing to the data table from test tables). As a result, different tests use the same data tables (to define input parameter values, for example) avoiding duplicating data.

Data tables can contain data types, supported by OpenL Tablets, or types loaded in OpenL Tablets from other sources. For information on data types, see Datatype Table and Working with Data Types.

The following topics are included in this section:

- Using Simple Data Tables
- Using Advanced Data Tables
- Specifying Data for Aggregated Objects
- Ensuring Data Integrity

Using Simple Data Tables

Simple data tables are intended to define a list of values of data types that have simple structure.

1. The first row is the header at the following format:

```
Data <data type> <data table name>
```

   where `data type` – is a type of data the table contains, it can be any predefined or alias data type (for more information on predefines and alias data types, refer to Working with Data Types and Datatype Table sections accordingly).

2. The second row is keyword 'this'.

3. The third row is a descriptive table name intended for business users.

4. In the fourth and following row are values of data provided.

The following is an example of a data table containing an array of numbers:

Data Integer **numbers**
this
Numbers
10
20
30
40
50

Figure 42: Simple data table

Using Advanced Data Tables

Advanced data tables are used for storing information of complex structure such as custom data types. For information on data types, see Datatype Table.

1. The first row of an advanced data table contains text in the following format:

   `Data <data type> <data table name>`

2. Each cell in the second row contains an attribute name of the data type.
3. The third row contains attribute display names.
4. Each row starting from the fourth one contains values for specific data rows.

The following diagram shows a datatype table and a corresponding data table with concrete values below it:

Datatype Person	
String	name
String	ssn

Data Person p1	
name	ssn
Name	**SSN**
Jonh	555-55-0001
Paul	555-55-0002
Peter	555-55-0003
Mary	555-55-0004

Figure 43: Datatype table and a corresponding data table

Specifying Data for Aggregated Objects

Let data, which values are to be specified and stored in a data table, is an object of complex structure with an attribute that is another complex object as well. Then the object that includes another object is called an **aggregated object.** To specify an attribute of an aggregated object in the data table, the following name chain format must be used in the row containing data table attribute names:

`<attribute name of aggregated object>.<attribute name of object>`

To illustrate this approach, assume there are two data types `ZipCode` and `Address` defined:

Datatype ZipCode	
String	zip1
String	zip2

Datatype Address	
String	street
String	city
ZipCode	zip

Figure 44: Complex data types defined by Datatype tables

As can be seen from the data types' structure, the `Address` data type contains a reference to the `ZipCode` data type as its attribute `zip`. A data table can be created that specifies values for both data types at the same time, for example:

Data Address addresses			
street	city	zip.zip1	zip.zip2
Street1	**City**	**Zip1**	**Zip2**
1600 Pennsylvania Avenue	Washington	20500	
1085 Summit Dr	Beverly Hills	90210	2814

Figure 45: Specifying values for aggregated objects

In the preceding example, columns **Zip1** and **Zip2** contain values for data type `ZipCode` referenced by aggregated data type `Address`.

Note: The attribute name chain can be of any arbitrary depth, for example, `account.person.address.street`.

In case a data table is to store information for an array of objects, OpenL Tablets allows us to specify attribute values for each element of an array. Then the following format should be used in the row of data table attribute names:

```
<attribute name of aggregated object>[i].<attribute name of object>
```

where `i` - sequence number of an element, starts from 0.

The next example illustrates this approach:

Data Policy policies					
name	driver	vehicles[0].model	vehicles[0].price	vehicles[1].model	vehicles[1].price
Policy	**Driver**	**Vehicle Model**	**Vehicle Price**	**Vehicle Model**	**Vehicle Price**
Policy1	Sara	Honda Odyssey	$ 39,000	Ford C-Max	
Policy2	Shane	Toyota Camry	$ 12,000		
Policy3	Spencer	VW Bug	$ 1,500	Mazda 3	$ 40,000

Figure 46: Specifying values for an array of aggregated objects

The first policy, **Policy1,** contains two vehicles: **Honda Odyssey** and **Ford C-Max**; the second policy, **Policy2,** – the only vehicle is **Toyota Camry**; the third policy, **Policy3,** contains two vehicles: **VW Bug** and **Mazda 3**.

Note: The approach is valid for simple cases with an array of simple Datatype values, and for complex cases with a nested array of an array, for example, `policy.vehicles[0].coverages[2].limit`.

Note: All mentioned formats of specifying data for aggregated objects are applicable in defining input values or expected result values in Test and Run Tables Types too.

Ensuring Data Integrity

If a data table contains values defined in another data table, it is important to specify this relationship. The relationship between two data tables is defined using foreign keys, a concept that is used in database management systems. Reference to another data table must be specified in an additional row below the row where attribute names are entered. The following format must be used:

```
> <referenced data table name> <column name of the referenced data table>
```

In the following example, the data table **cities** contains values from the table **states.** To ensure users enter correct values, a reference to the **code** column in the **states** table is defined.

Data City cities	
city	state
	>states code
City	**State**
Fairbanks	AK
Beverly Hills	CA

Data SupportedState states	
name	code
State/Possession	**Abbreviation**
ALABAMA	AL
ALASKA	AK
AMERICAN	AS
ARIZONA	AZ
ARKANSAS	AR
CALIFORNIA	CA
COLORADO	CO
CONNECTICUT	CT
DELAWARE	DE

Figure 47: Defining a reference to another data table

In case user enters invalid state abbreviation in the table **cities,** OpenL Tablets reports an error.

The target column does not have to be specified if it is the first column in the referenced data table. For example, if a reference was made to the column **name** in the table **states,** the following simplified reference could be used:

```
>states
```

In case a data table contains values which are defined as part of another data table, the following format can be used:

```
> <referenced data table name>.<attribute name> <column name>
```

The difference with the previous format is that you additionally specify an attribute name of the referenced data table whose corresponding values are included in the other data table.

If <column name> is omitted then, by default, the reference is constructed using the first column of the referenced data table.

In the following diagram, the data table **claims** contains values defined in the table **policies** and related to the **vehicle** attribute. A reference to the column **name** of the table **policies** is omitted as this is the first column in the table.

Data Policy policies				
name	driver	vehicle.model	vehicle.year	vehicle.price
Policy	**Driver**	**Vehicle Model**	**Vehicle Year**	**Vehicle Price**
Policy1	Sara	Honda Odyssey	2005	$39,000
Policy2	Shane	Toyota Camry	2002	$12,000
Policy3	Spencer	VW Bug	1965	$1,500

Data Claim claims				
id	lossDate	vehicle	damage	payment
		>policies.vehicle		
Policy	**Date of Loss**	**Vehicle of Policy**	**Damage Description**	**Payment**
Claim1	02 July 2012	Policy2	broken side window	$350
Claim2	14 March 2013	Policy3	damaged bumper	$200

Figure 48: Defining a reference to another data table

Note: To ensure users enter correct values, cell data validation lists can be used in Excel limiting the range of values users can type in.

Note: The syntax of data integration is applicable in defining input values or expected result values in Test and Run Tables Types too.

Note: The attribute path can be of any arbitrary depth, for example `>policies.coverage.limit`.

Test Table

A **test table** is used to perform unit and integration tests on executable rule tables, such as decision tables, spreadsheet tables, method tables etc. It calls a particular table, provides test input values, and checks whether the returned value matches the expected value.

For example, in the following diagram, the table on the left is a decision table but the table on the right is a unit test table that tests data of the decision table:

Rules DoubleValue RiskFactor3 (Date MyDate)		
C1 dayOfWeek(MyDate) IntRange	RET1	
Day of Week	**Risk Factor (%)**	**Comments**
[2 .. 5]	75%	Monday-to-Wednesday
6	85%	Friday RF
	100%	Week-end RF

Test RiskFactor3 RiskFactor3Test	
MyDate	_res_
Date	**Result**
12/21/2012	0.85
12/22/2012	1.0
12/19/2012	0.75

Figure 49: Decision table and its unit test table

A test table has the following structure:

1. The first row is the table header, which has the following format:

```
Test <rule table name> <test table name>
```

'Test' is a keyword that identifies a test table. The second parameter is the name of the rule table to be tested. The third parameter is the name of the test table.

2. The second row provides a separate cell for each input parameter of the rule table followed by column **_res_,** which typically contains the expected test result values.
3. The third row contains display values intended for business users.
4. Starting with the fourth row, each row is an individual test case.

How to specify values of input parameters and expected test results which have complex constructions, refer to "Specifying Data for Aggregated Objects" and "Ensuring Data Integrity".

Note: For experienced users. Test tables can be used to execute any Java method but in that case a method table must be used as a proxy.

When a test table is called, the OpenL Tablets engine calls the specified rule table for every row in the test table and passes the corresponding input parameters to it.

Application run-time context values are defined in the run-time environment. Test tables for a table, overloaded by business dimension properties, must provide values for the run-time context significant for the tested table. Run-time context values are accessed in the test table through the **_context_** prefix. An example of a test table with the context value Lob follows:

Test driverAgeType driverAgeTypeTest		
driver	_context_lob	_res_
>testDrivers1		
Driver	**Lob**	**Expected Age Type**
Sara	Home	Standard Driver
Spencer, Sara's Son	Home	Old Driver
Sara	Auto	High Risk Driver
Spencer, Sara's Son	Auto	Young Driver

Figure 50: An example of a test table with a context value

Please refer to Context Variables Available in Test Tables section for a full list of runtime context variables available, their description and related Business Dimension (versioning) properties.

You can also use the **_description_** column to enter any useful information.

User can use the **_error_** column of Test table to test algorithm where *error* function is used. OpenL engine compares error message and value of **_error_** column to decide is test passed or not.

Test driverRiskScoreTest driverRiskTest		
driverRisk	_res_	_error_
Driver Risk	**Expected Risk**	**Expected Error**
High Risk Driver	100	
		My Exception

Figure 51: An example of a test table with an expected error column

If OpenL Tablets projects are accessed and modified through OpenL Tablets WebStudio, the user interface provides convenient utilities for running tests and viewing test results. For information on using OpenL Tablets WebStudio, see *OpenL Tablets WebStudio User's Guide*.

Context Variables Available in Test Tables

The following runtime context variables are used in OpenL Tablets and their values can be specified in OpenL test tables using syntax `_context_.<context name>` in a column header:

Context	Context Name used in rule tables	Type	Related Versioning Properties	Property Names used in rule tables	Description
Current Date	`currentDate`	`Date`	Effective / Expiration dates	`effectiveDate, expirationDate`	The date on which the rule is performed.
Request Date	`requestDate`	`Date`	Start / End Request dates	`startRequestDate, endRequestDate`	The date when the rule is applied.
Line of Business	`lob`	`String`	LOB (Line of Business)	`lob`	Lime of Business for which the rule is applied.
US State	`usState`	`Enum`	US States	`state`	US State where the rule is applied.
Country	`country`	`Enum`	Countries	`country`	Country where the rule is applied.
US Region	`usRegion`	`Enum`	US Region	`usregion`	US Region where the rule is applied.
Currency	`currency`	`Enum`	Currency	`currency`	Currency with which the rule is applied.
Language	`lang`	`Enum`	Language	`lang`	Language on which the rule is applied.
Region	`region`	`Enum`	Region	`region`	Economic region where the rule is applied.

Figure 52: Context variables of OpenL Tablets

More information on how property values relate to runtime context values and what rule table is executed you can find in Business Dimension Properties section.

Testing Spreadsheet Result

You can test cells of a Spreadsheet result which is returned by your rule table in the following way. Let's use the Spreadsheet table shown in the Figure below.

Spreadsheet SpreadsheetResult test (String coverageId, int coveredProperty, double koef)				
Step	**Code**	**Formula**	**Value**	**Text : String**
Coverage_Id	COVERAGE_ID			= coverageId
Covered_Property	COVERED_PROPERTY_COVERAGE	= $Value	= coveredProperty + 1	
Final_Premium	FINAL_PREMIUM	= $Value	= koef * $Formula$Covered_Property	

Figure 53: A sample Spreadsheet table

For testing purposes, standard Test table is used. Cells of the Spreadsheet are accessed by using the `_res_.$<column name>$<row name>` expression.

Testmethod test TestSpr				
coverageId	coveredProperty	koef	_res_.$Formula$Final_Premium	_res_.$Text$Coverage_Id
Income Coverage Id	Income Covered Property	Income Koefficient	Result of Final Premium	Result of Coverage Id
myTestCoverage	4	1,23	6,15	myTestCoverage

Figure 54: Test for the sample Spreadsheet table

Columns marked with green color determine the income values, and the columns marked with lilac determine the expected values for some number of cells. It is possible to test as much cells as you need.

As a result of running this test in the WebStudio you will see the next output table.

TestSprTestAll

Input Parameters

Coverage Id	coveredPropertyfff	
myTestCoverage	4	1.23

Result

Step	Code	Formula	Value	Text : String
Coverage_Id	COVERAGE_ID			myTestCoverage; myTestCoverage
Covered_Property	COVERED_PROPERTY_COVERAGE	5.0	5.0	
Final_Premium	FINAL_PREMIUM	6.15; 6.15	6.15	

Figure 55: The sample Spreadsheet test results

If the Custom Spreadsheet result feature is activated, it is even possible to test cells of the resulting Spreadsheet which contain values of complex types such as

-array of values

-custom data type with several attributes

-another Spreadsheet(s) nested in the current one.

For this purpose, the same syntax described in "Specifying Data for Aggregated Objects" can be used, namely:

```
_res_.$<column name>$<row name>[i]

_res_.$<column name>$<row name>.<attribute name>

_res_.$<column of Main Spreadsheet>$<row of Main Spreadsheet>.$<column of Nested
     Spreadsheet>$<row of Nested Spreadsheet>

_res_.$<column of Main Spreadsheet>$<row of Main Spreadsheet>[i].$<column of
Nested Spreadsheet>$<row of Nested Spreadsheet>
```

where `i` - sequence number of an element, starts from 0.

Let's look at the advanced example below. **PolicyCalculation** spreadsheet table performs lots of calculations regarding an insurance policy, including specific calculations for vehicles and a main driver of the policy. In order to evaluate vehicle and drivers (for example, calculate their score, premium etc.) **VehicleCalculation** and **DriverCalculation** spreadsheet tables are invoked in cells of PolicyCalculation rule table.

Spreadsheet SpreadsheetResult PolicyCalculation (Policy policy)	
	Value
Vehicles : SpreadsheetResult[]	= VehicleCalculation (vehicles)
MainDriver : SpreadsheetResult	= DriverCalculation (drivers[0])
Score	= sum(GetScore($Vehicles)) + GetScore($MainDriver) + ClientTierScore(clientTier)
Eligibility : EligibilityType	= PolicyEligibility (clientTerm, $Score)
Premium	= sum(GetPremium($Vehicles)) + GetPremium($MainDriver) - ClientDiscount (clientTier)

Spreadsheet SpreadsheetResult VehicleCalculation (Vehicle vehicle)	
	Value
Age : Integer	= CurrentYear() - year
TheftRating : TheftRating	= VehicleTheftRating (bodyType, price, onHighTheftProbabilityList)
InjuryRating : InjuryRating	= VehicleInjuryRating (bodyType, airbagType, hasRollBar)

Spreadsheet SpreadsheetResult DriverCalculation (Driver driver)	
	Value
DriverType : DriverType	= DriverAgeType (gender, age)
Eligibility : EligibilityType	= DriverEligibility ($DriverType, hadTraining)
DriverRisk : DriverRisk	= DriverRisk (numDUI, numAccidents, numMovingViolations)
Score	= DriverTypeScore ($DriverType, $Eligibility) + DriverRiskScore ($DriverRisk)
Premium	= DriverPremium ($DriverType, maritalStatus, state)+ DriverRiskPremium ($DriverRisk) + AccidentPremium () * numAccidents

Figure 56: The advanced sample Spreadsheet table

So as you may notice that structure of resulting **PolicyCalculation** spreadsheet is rather complex. But you can test any cell of the result as illustrated in **PolicyCalculationTest** test table:

Test PolicyCalculation PolicyCalculationTest			
policy	_res_.$Value$Premium	_res_.$Value$MainDriver.$Value$Score	_res_.$Value$Vehicles[0].$Value$Age
> testPolicy1			
Policy	**Expected Premium**	**Expected Driver Score**	**Expected Vehicle 1 Age**
Policy1	827.5	0	9
Policy2	2550	130	49

Figure 57: Test for the advanced sample Spreadsheet table

Run Table

A **run table** calls a particular rule table multiple times and provides input values for each individual call. Therefore, run tables are similar to test tables, except they do not perform a check of values returned by the called method.

Note: For experienced users. Run tables can be used to execute any Java method.

The following is an example of a run method table:

Run append appendRun	
firstWord	secondWord
First Word	**Second Word**
Hi,	John!
Hello,	Mary!
Good morning,	Bob!

Figure 582: Run table

This example assumes there is a rule `append` defined with two input parameters, `firstWord` and `secondWord`. The run table calls this rule three times with three different sets of input values.

A run table has the following structure:

1. The first row is a table header, which has the following format:
   `Run <name of rule table to call> <run table name>`
2. The second row contains cells with rule input parameter names.
3. The third row contains display values intended for business users.
4. Starting with the fourth row, each row is a set of input parameters to be passed to the called rule table.

How to specify values of input parameters which have complex constructions, refer to "Specifying Data for Aggregated Objects" and "Ensuring Data Integrity".

Method Table

A **method table** is a Java method described within a table. The following is an example of a method table:

Method String getGreeting(String name)
return "Hi, "+name;

Figure 59: Method table

The first row is a table header, which has the following format:

```
<keyword> <return type> <table name> (<input parameters>)
```

where `<keyword>` is either 'Method' or 'Code'.

The second row, and the following rows, is the actual code to be executed. It can reference parameters passed to the method and all Java objects and tables visible to the OpenL Tablets engine. This table type is intended for users having experience in programming to develop rules of any logic and complexity.

Configuration Table

OpenL Tablets allows externalizing business logic into Excel files (modules). And there exist cases when rule tables of one module need to call rule tables placed in another module. In order to indicate module dependency, a configuration table is used. Another common purpose of a configuration table is when OpenL Tablets rules need to use objects and methods defined in the Java environment. To enable use of Java objects and methods in Excel tables, the module must have a configuration table. A

configuration table provides information to the OpenL Tablets engine about available Java packages.

A configuration table is identified by the keyword 'Environment' in the first row. No additional parameters are required. Starting with the second row, a configuration table must have two columns. The first column contains commands and the second column contains input strings for commands.

The following commands are supported in configuration tables:

Configuration table commands	
Command	**Description**
`dependency`	Adds the dependency module by its name. All data from this module will be accessible in the current one. Dependency module can be located in the current project or its dependency projects.
`import`	Imports the specified Java package so that its objects and methods can be used in tables.
`include`	Includes another Excel file so that its tables and data can be referenced in tables of the current file.
`language`	Language import functionality.
`extension`	External set of rules for expanding OpenL Tablets capabilities. After adding, external rules are complied with OpenL Tablets rules and work jointly.
`vocabulary`	Ability to use user created dynamic classes in OpenL Tablets.

The following is an example of a configuration table:

Environment	
dependency	Rating Common
	Rating Domain Model
import	org.apache.commons.lang

Figure 60: Configuration table

Properties Table

Description

A **properties** table is used to define the module and category level properties inherited by tables. The properties table has the following structure:

Properties table elements	
Element	**Description**
Properties	Reserved word that defines the type of the table. It can be followed by a Java identifier. In this case, the properties table value becomes accessible in rules as a field of such name and of the **TableProperties** type.

<table>
<tr><th colspan="2">Properties table elements</th></tr>
<tr><th>Element</th><th>Description</th></tr>
<tr><td>scope</td><td>Identifies levels on which the property inheritance is defined. Available values are as follows:
<table>
<tr><th>Scope level</th><th>Description</th></tr>
<tr><td>Module</td><td>Identifies properties defined for the whole module and inherited by all tables in it. There can be only one table with the Module scope in one module.
<table>
<tr><th colspan="2">Properties property_test1</th></tr>
<tr><td>scope</td><td>Module</td></tr>
<tr><td>effectiveDate</td><td>4/7/10</td></tr>
<tr><td>expirationDate</td><td>4/28/11</td></tr>
<tr><td>lang</td><td>EN</td></tr>
<tr><td>currency</td><td>USD</td></tr>
<tr><td>state</td><td>CA</td></tr>
</table>
Figure 61: A properties table with the Module level scope</td></tr>
<tr><td>Category</td><td>Identifies properties applied to all tables where the category name equals the name specified in the category element.
<table>
<tr><th colspan="2">Properties property_test2</th></tr>
<tr><td>scope</td><td>Category</td></tr>
<tr><td>category</td><td>Testing</td></tr>
<tr><td>country</td><td>CA,CH,DE,FR</td></tr>
<tr><td>lob</td><td>Home</td></tr>
<tr><td>lang</td><td>GER</td></tr>
<tr><td>currency</td><td>CAD</td></tr>
</table>
Figure 62: A properties table with the Category level scope</td></tr>
</table></td></tr>
<tr><td>category</td><td>Defines the category if the scope element is set to Category. If no value is specified, the category name is retrieved from the sheet name.</td></tr>
<tr><td>Module</td><td>Identifies that properties can be overridden and inherited on the Module level.</td></tr>
</table>

Spreadsheet Table

A **spreadsheet** table, in OpenL Tablets, is an analogue of the Excel table with rows, columns, formulas and calculations as contents. Spreadsheets can also call decision tables or other executable tables to make decisions on values, and based on those, make calculations.

The format of the spreadsheet table header is as follows:

```
Spreadsheet SpreadsheetResult <table name> (<input parameters>)
```

or

```
Spreadsheet <return type> <table name> (<input parameters>)
```

The following table describes the spreadsheet table header syntax:

Spreadsheet table header syntax	
Element	**Description**
Spreadsheet	Reserved word that defines the type of the table.

Spreadsheet table header syntax	
Element	**Description**
SpreadsheetResult	Type of the return value. SpreadsheetResult returns the calculated content of the whole table.
<return type>	Data type of the returned value. If only a single value is required, its type must be defined here as a return datatype and calculated in the row or column named **RETURN**
<table name>	Valid name of the table as for any executable table.
<input parameters>	Input parameters as for any executable table.

The first column and row of a spreadsheet table (after header) make the table **column and row names**. Values in other cells are the table values. An example follows:

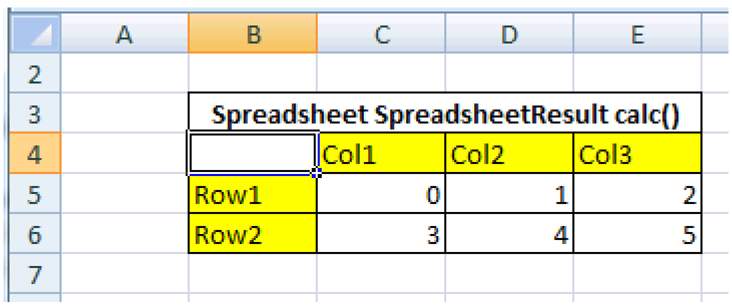


Figure 63: Spreadsheet table organization

A spreadsheet table cell can contain:

- simple values, such as a string, numeric;
- values of other data types. In this case the datatype should be defined explicitly. You can specify the data type for a whole row or column of the cell using syntax

```
<column name or row name> : <data type>;
```

Note: if both column and row of the cell have datatype specified, then the datatype of the column is taken.

- formulas that start with an apostrophe followed by = or, alternatively, are enclosed by { };
- in a cell formula you may reference another cell value or range of another cell values. The following table describes how cell value can be referenced in a spreadsheet table:

Referencing another cell		
Notation (cell name)	**Reference**	**Description**
`$columnName`	By column name	Used to refer to the value of another column in the same row.
`$rowName`	By row name	Used to refer to the value of another row in the same column.
`$columnName$rowName`	Full reference	Used to refer to the value of another row and column.

On how to specify range of cells, go to Using Ranges in Spreadsheet Table.

Below is an example of spreadsheet table with different calculations for an auto insurance policy (table cells contain simple values, formulas, references to the value of another cell etc.):

Spreadsheet SpreadsheetResult VehicleCalculation (Vehicle vehicle)	
	Value
Vehicle : Vehicle	= vehicle
Age	= CurrentYear() - year
BasePremium	= BasePremium (carType)
VehicleDiscount	= VehicleDiscount (airbagType, hasAlarm)
PreliminaryPremium	= $BasePremium * (1 - $VehicleDiscount)
MinPremium	180
FinalPremium	= max ($PreliminaryPremium, $MinPremium)

Figure 64: Spreadsheet table with calculations as content

Parsing of Spreadsheet Table

OpenL Tablets processes spreadsheet tables in two different ways depending on the return type:

1. Spreadsheet returns *SpreadsheetResult* datatype.
2. Spreadsheet returns any other datatype different from the first point.

In the first case you will get the SpreadsheetResult type that is an analog of result matrix. All the calculated cells of the Spreadsheet table will be accessible through this result. The following example shows Spreadsheet table of this type.

Spreadsheet SpreadsheetResult processDriver(Driver driver)	
	Value
Driver:Driver	{driver}
Age Type:String	{driverAgeType($Driver)}
Eligibility:String	{driverEligibility($Driver, $Age Type)}
Driver Risk:String	{driverRisk($Driver)}
Score	{driverTypeScore($Age Type, $Eligibility) + driverRiskScore($Driver Risk)}
Premium	{driverPremium($Driver, $Age Type) + driverRiskPremium($Driver Risk) + driverAccidentPremium($Driver, $Driver Risk)}

Figure 65: Spreadsheet table returns SpreadsheetResult datatype

In the second case the returned result is a datatype as in all other rule tables (you don't have 'SpreadsheetResult' in the rule table header). The cell with the RETURN key word for a row will be returned. OpenL will calculate the cells that are needed just for that result calculation. In the following example the 'License_Points' cell is not included in the 'Tier Factor' calculation, so it will simply be skipped.

Spreadsheet DoubleValue **TierFactor** (Policy policy)		
Step	**Formula**	**Value**
Credit_Rating_Points	=CreditRatingPoints (creditRating)	
Violations_Points	= ViolationPoints (drivers)	
License_Points	= sum (LicensedYearsPoints (drivers, policyEffectiveDate))	
Total_Points	= sum ($Credit_Rating_Points:$Violatione_Points)	
Tier_Factor	= tierFactor = mapTierPointsToFactor ($Total_Points)	
RETURN	= $Tier_Factor	

Figure 66: Spreadsheet table returns a single value

Accessing Spreadsheet Result cells

A value of SpreadsheetResult type means that this is actually a table (matrix) of values which can be different types. A cell is defined by its table column and row. Therefore, a value of particular spreadsheet cell can be accessed by cell's column and row names and indicating its datatype:

```
(<data type>) <spreadsheet result variable>.$<column name>$<row name>
```

The example below demonstrates how to get a value of **FinancialRatingCalculation** spreadsheet result that is calculated in **Value** column and **FinancialRating** row of the spreadsheet:

Spreadsheet SpreadsheetResult CorporateRatingCalculation (Corporate corporate)		
properties	description	Corporate Rating is the level of company's creditworthiness (Financial Rating) corrected by the level of Risk of Work with it.
Step		**Value**
CheckCurrentFinancialData : void		= SetNonZeroValues(financialData)
FinancialRatingCalculation : SpreadsheetResult		= FinancialRatingCalculation(financialData, industry)
FinancialRating		= (DoubleValue) $FinancialRatingCalculation.$Value$FinancialRating
RiskOfProfile : String		= RiskOfProfile (corporate)
RiskOfOperations : String		= RiskOfOperations (qualityIndicators)
RiskOfGeography : String		= RiskOfGeography (qualityIndicators)

Figure 67: Accessing Spreadsheet Result cell value

You can also access the spreadsheet cell using the `getFieldValue(String <cell name>)` function (for instance, `(DoubleValue) $FinancialRatingCalculation.getFieldValue ("$Value$FinancialRating")`), but it is more complicated option.

Note: There are some limitations:

If the cell name in columns or rows contains not allowed symbols (space, percentage, etc – see not allowed symbols for Java methods), it is impossible to access the cell.

Using Ranges in Spreadsheet Table

When you work with a range in a spreadsheet table, you can use the following syntax: ($FirstValue:$LastValue) to specify your range. In the figure below, there is an example of using range this way (the **TotalAmount** column).

Spreadsheet SpreadsheetResult **IncomeForecast** (Double bonusRate, Double sharePrice)				
	Year1	**Year2**	**Year3**	**TotalAmount**
Salary	45,000	= round ($Year1$Salary * 1.10)	=round ($Year1$Salary * 1.20)	= sum($Year1:$Year3)
Shares	0	0	1,000	= sum($Year1:$Year3)
Bonus	=$Salary * bonusRate	= $Salary * bonusRate	= $Salary * bonusRate	= sum($Year1:$Year3)
bonusRate	=bonusRate	=bonusRate	=bonusRate	
sharePrice	=sharePrice	=sharePrice	=sharePrice	
MinSalary	= $Salary	= $Salary	= $Salary	= sum($Year1:$Year3)
MaxSalary	= $Salary + $Bonus + $Shares * sharePrice	= $Salary + $Bonus + $Shares * sharePrice	= $Salary + $Bonus + $Shares * sharePrice	= sum($Year1:$Year3)

Figure 68: Using ranges of Spreadsheet table in functions

NOTE: In expression like 'min/max($FirstValue:$LastValue)', there should be no space before and after the colon (:) operator.

Custom Spreadsheet Result

It is possible to improve the usage of spreadsheet tables that return the SpreadsheetResult type. Now there is a possibility to have a separate type for each Spreadsheet table – Custom Spreadsheet Result data type - which is determined as

```
SpreadsheetResult<Spreadsheet table name>
```

This feature gives you the following advantages:

- A possibility to explicitly define the type of the returned value. In other words, you don't need to indicate a datatype when accessing the cell.
- Test Spreadsheet cell of any complex type (see Testing Spreadsheet result for details).

By default, the feature is turned on in WebStudio and turned off in Web Services. Please refer to the *System Settings* section in OpenL Tablets WebStudio User Guide for information on how to turn it off.

To understand how it works, let's have a look at the next spreadsheet.

Spreadsheet SpreadsheetResult test (String coverageId, int coveredProperty, double koef)				
Step	**Code**	**Formula**	**Value**	**Text : String**
Coverage_Id	COVERAGE_ID			= coverageId
Covered_Property	COVERED_PROPERTY_COVERAGE	= $Value	= coveredProperty + 1	
Final_Premium	FINAL_PREMIUM	= $Value	= koef * $Formula$Covered_Property	

Figure 69: An example of the Spreadsheet

The return type is **SpreadsheetResult**, but with Custom Spreadsheet result feature turned on, it becomes **SpreadsheetResulttest** datatype. Now it is possible to access any calculated cell in a very simplified way without indicating its datatype, for example, as it is shown in the figure below:

Rules String **CheckFinalPremium** (String coverageId, int coveredProperty, double koef)	
C1	RET1
test(coverageId, coveredPremium, koef).$Value$Final_Premium < upperBound	
DoubleValue upperBound	String
Is Final Premium less than upper bound?	**Message**
1000	Final premium is less than 1000
2000	Final premium is less than 2000
3000	Final premium is less than 3000
	Final premium is more than 3000

Figure 70: Calling Spreadsheet cell

In this example we are accessing the Spreadsheet table cell from the returned Custom Spreadsheet Result.

Column Match Table

A **column match** table has an attached algorithm. The algorithm denotes the table content and how the return value is calculated. Usually this type of table is referred to as a **Decision Tree.**

The format of the column match table header is as follows:

```
ColumnMatch <ALGORITHM> <return type> <table name> (<input parameters>)
```

The following table describes the column match table header syntax:

Column match table header syntax	
Element	**Description**
ColumnMatch	Reserved word that defines the type of the table.
<ALGORITHM>	Name of the algorithm. This value is optional.
<return type>	Type of the return value.
<table name>	Valid name of the table.
<input parameters>	Input parameters as for any executable table.

The following predefined algorithms are available:

Predefined algorithms	
Element	**Reference**
MATCH	MATCH Algorithm
SCORE	SCORE Algorithm
WEIGHTED	WEIGHTED Algorithm

Each algorithm has the following mandatory columns:

<table>
<tr><th colspan="2">Algorithm mandatory columns</th></tr>
<tr><th>Column</th><th>Description</th></tr>
<tr><td>Names</td><td>Names refer to the table or method arguments and bind an argument to a particular row. The same argument can be referred in multiple rows.
Arguments are referenced by their short names. For example, if an argument in a table is a Java Bean with the some property, it is enough to specify some in the names column.</td></tr>
<tr><td>Operations</td><td>The operations column defines how to match or check arguments to values in a table. The following operations are available:
<table>
<tr><th>Operation</th><th>Checks for</th><th>Description</th></tr>
<tr><td>match</td><td>equality or belonging to a range</td><td>The argument value must be equal to or within a range of check values.</td></tr>
<tr><td>min</td><td>minimally required value</td><td>The argument must not be less than the check value.</td></tr>
<tr><td>max</td><td>maximally allowed value</td><td>The argument must not be greater than the check value.</td></tr>
</table>
The min and max operations work with numeric and date types only.
The min and max operations can be replaced with the match operation and ranges. This approach adds more flexibility because it enables the checking of all cases within one row.</td></tr>
<tr><td>Values</td><td>The values column typically has multiple sub columns containing table values.</td></tr>
</table>

MATCH Algorithm

The **MATCH** algorithm enables the user in mapping a set of conditions to a single return value.

Besides the mandatory columns, which are names, operations, and values, the **MATCH** table expects that the first data row contains **Return Values**, one of which is returned as a result of the ColumnMatch table execution.

ColumnMatch <MATCH> Boolean needApproval(Expense expense)							
names	operation	values					
Name	**Operation**	**Values**					
Return Values		**YES**	**YES**	**YES**	**YES**	**NO**	**NO**
area	match	Hardware	Software	Hardware	Software		
money	min	50000	20000	100000	40000		
paysCompany	match	TRUE	TRUE	FALSE	FALSE		
area	match					Hardware	Software
money	max					20000	10000

Figure 71: An example of the MATCH algorithm table

The MATCH algorithm works up to down and left to right. It takes an argument from the upper row and matches it against check values from left to right. If they match, the algorithm returns the corresponding return value, which is the one in the same column as the check value. If values do not match, the algorithm switches to the next row. If no match is found in the whole table, the **null** object is returned.

If the return type is primitive, such as **int, double,** or **Boolean,** a run-time exception is thrown.

The MATCH algorithm supports **AND** conditions. In this case, it checks whether all arguments from a group match the corresponding check values, and checks values in the same value sub column each time. The **AND** group of arguments is created by indenting two or more arguments. The name of the first argument in a group must be left unintended.

SCORE Algorithm

The **SCORE** algorithm calculates the sum of weighted ratings or scores for all matched cases. The **SCORE** algorithm has the following mandatory columns:

- names
- operations
- weight
- values

The algorithm expects that the first row contains **Score,** which is a list of scores or ratings added to the result sum if an argument matches the check value in the corresponding sub column.

ColumnMatch <SCORE> int scoreIssue(Issue issue)								
names	operation	weight	values					
Name	**Operation**	**Weight**	**Values**					
Score			10	5	3	3	2	1
area	match	1	Loss	Profit	Budget	Expenses	HR	
mundane	match	2	FALSE					
money	match	3	1000000+	100000+	25000+		10000+	200+

Figure 72: An example of the SCORE algorithm table

The SCORE algorithm works up to down and left to right. It takes the argument value in the first row and checks it against values from left to right until a match is found. When a match is found, the algorithm takes the score value in the corresponding sub column and multiples it by the weight of that row. The product is added to the result sum. After that, the next row is checked., The rest of the check values on the same row are ignored after the first match. The `0` value is returned if no match is found.

The following limitations apply:

- Only one score can be defined for each row.
- AND groups are not supported.
- Any amount of rows can refer to the same argument.
- The SCORE algorithm return type is always integer.

WEIGHTED Algorithm

The **WEIGHTED** algorithm combines the SCORE and simple MATCH algorithms. The result of the SCORE algorithm is passed to the MATCH algorithm as an input value. The MATCH algorithm result is returned as the WEIGHTED algorithm result.

The WEIGHTED algorithm requires the same columns as the SCORE algorithm. Yet it expects that first three rows are **Return Values, Total Score,** and **Score. Return**

Values and **Total Score** represent the MATCH algorithm, and the **Score** row is the beginning of the SCORE part.

ColumnMatch <WEIGHTED> String scoreIssueImportance(Issue issue)								
names	operation	weight	values					
Name	**Operation**	**Weight**	**Values**					
Return Values			**CRITICAL**	**HIGH**	**Moderate**	**Low**		
Total Score	**min**		30	20	10	0		
Score			10	5	3	3	2	1
area	match	1	Loss	Profit	Budget	Expenses	HR	
mundane	match	2	FALSE					
money	match	3	1000000+	100000+	25000+		10000+	200+

Figure 73: An example of the WEIGHTED algorithm table

The WEIGHTED algorithm requires the use of an extra Method table that joins the SCORE and MATCH algorithm. Testing the SCORE part can become difficult in this case. Splitting the WEIGHTED table into separate SCORE and MATCH algorithm tables is recommended.

TBasic Table

A **TBasic** table is used for code development in more convenient and structured way rather than using Java or Business User Language (BUL). It has several clearly defined structural components. Using Excel cells, fonts, and named code column segments provides clearer definition of complex algorithms.

In a definite UI, it can be used as a workflow component.

The format of the TBasic table header is as follows:

```
TBasic <ReturnType> <TechnicalName> (ARGUMENTS)
```

The following table describes the TBasic table header syntax:

Tbasic table header syntax	
Element	**Description**
TBasic	Reserved word that defines the type of the table.
ReturnType	Type of the return value.
TechnicalName	Algorithm name.
ARGUMENTS	Input arguments as for any executable table.

The following table explains the recommended parts of the structured algorithm:

Algorithm parts	
Element	**Description**
Algorithm precondition or preprocessing	Executed when the component starts execution.
Algorithm steps	Represents the main logic of the component.
Postprocess	Identifies a part executed when the algorithm part is executed.

Algorithm parts	
Element	**Description**
User functions and subroutines	Contains user functions definition and subroutines.

Table Part

A **Table Part** functionality enables the user to split a large table into smaller parts (partial tables). Physically in the Excel workbook the table is represented as several Table Parts but logically it's processed as one rules table.

This functionality is suitable for cases when the user is dealing with .xls file format and a rules table exists with more than 256 columns or 65,536 rows. To create such rule table, the user can split the table into several parts and place each part on a separate worksheet.

Splitting can be vertical or horizontal. In vertical case, the first N1 rows of an original rule table are placed in the 1*st* Table Part, the next N2 rows - in the 2nd and so on. In horizontal case, the first N1 columns of the rule table are placed in the 1st Table Part, the next N2 columns – in the 2nd and so on. The header of the original rule table and its properties definition should be copied in each Table Part in case of horizontal splitting. Merging of Table Parts into the rule table is processed as depicted on Figure 74 and Figure 75.

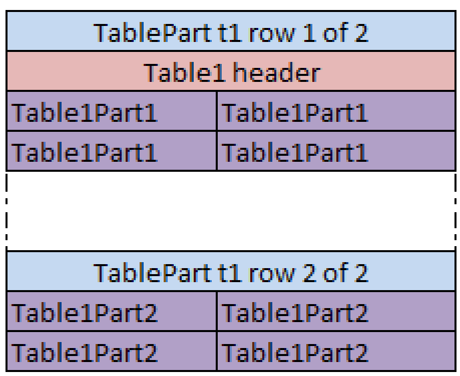


Figure 74: Vertical merging of Table Parts

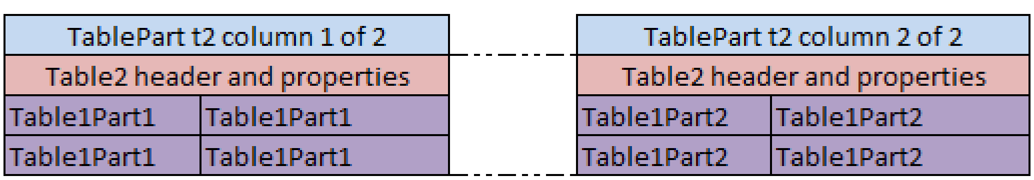


Figure 75: Horizontal merging of Table Parts

All Table Parts should be located within the one Excel file.

Splitting can be applied for any tables of Decision, Data, Test and Run types.

The format of the TablePart header is as follows:

```
TablePart <table id> <split type> {M} of {N}
```

The following table describes the TablePart header syntax:

Table Part header syntax	
Element	**Description**
TablePart	Reserved word that defines the type of the table.
<table id>	A unique name of the rules table. Can be the same as the rules table name if the rules table is not overloaded by properties.
<split type>	Type of splitting. Set to “row” for vertical splitting; “column” – for horizontal splitting.
{M}	Sequential number of the Table Part: 1, 2, and so on.
{N}	Total number of Table Parts of the rule table.

Examples below illustrate vertical and horizontal splitting of decision rule **RiskOfWorkWithCorporate**:

TablePart *RiskOfWorkWithCorporate* row 1 of 2

SimpleRules String **RiskOfWorkWithCorporate** (String ri

Risk of Profile	Risk of Operations	Risk of Geography	Total Risk
LOW	LOW	LOW	LOW
LOW	LOW	MIDDLE	LOW
LOW	LOW	HIGH	LOW
LOW	MIDDLE	LOW	LOW
LOW	MIDDLE	MIDDLE	LOW

TablePart *RiskOfWorkWithCorporate* row 2 of 2

LOW	MIDDLE	HIGH	MIDDLE
LOW	HIGH	LOW	LOW
LOW	HIGH	MIDDLE	MIDDLE
LOW	HIGH	HIGH	MIDDLE
MIDDLE	LOW	LOW	LOW
MIDDLE	LOW	MIDDLE	MIDDLE

Figure 76: Table Parts example. Vertical splitting

TablePart *RiskOfWorkWithCorporate* column 1 of 2	
SimpleRules String **RiskOfWorkWithCorporate** (St	
Risk of Profile	**Risk of Operations**
LOW	LOW
LOW	LOW
LOW	LOW
LOW	MIDDLE
LOW	MIDDLE

TablePart *RiskOfWorkWithCorporate* column 2 of 2	
SimpleRules String **RiskOfWorkWithCorporate** (St	
Risk of Geography	**Total Risk**
LOW	LOW
MIDDLE	LOW
HIGH	LOW
LOW	LOW
MIDDLE	LOW

Figure 77: Table Parts example. Horizontal splitting

Chapter 3: OpenL Tablets Functions and Supported Data Types

This section is intended for OpenL Tablets users to help them better understand how their business rules are processed in the OpenL Tablets system.

To implement your business rules logic you need to instruct OpenL Tablets what you want to do. For this you will create one or several rule tablets which will contain description of your rules logic.

Usually rules operate with some data (from your domain) to perform certain actions or return some results. The actions are performed using *functions* which, in turn, support particular *Data Types*.

This section describes Data Types and the functions that you will use to manage your business rules in the system. Basic principles of the use of Arrays are provided as well.

The section includes the following topics:

- Arrays in OpenL Tablets
- Working with Data Types
- Working with Functions

Arrays in OpenL Tablets

An array is a collection of values of the same type. Separate values in this case are called array *elements*. An array element is a value of any Data Type available in the system: IntValue, Double, Boolean, String, etc. (For more information on OpenL Tablets Data Types refer to the *Working with Data Types* section.) The square brackets in the name of Data Type indicate that you are dealing with an array of values in your rule. For example, you can use the String[] expression to represent an array of text elements of the 'String' Data Type, such as US state names – CA, NJ, VA, etc. In your rules you will use arrays for different purposes such as calculating statistics, representing multiple rates, and so on.

Working with Arrays from rules

There are two possibilities to work with Datatype arrays from rules:

- **by numeric index, starting from 0**

  In this case by call `drivers[5]` you will get the 6th element of the Datatype array.
- **by user defined index**

  The second case is a little more complicated. The first field of datatype is considered to be the user defined index. For example if we have a Datatype Driver with first String field name, we can create a Data Table, initializing two instances of Driver with names: John and David. Then in rules we can call the instance we need by `drivers["David"]`. You can use all Java types (including primitives) and Datatypes for your indexes. When the first field of Datatype is of int type called id, to call the instance from array, wrap it with quotes: e.g. `drivers["7"]`, in this case you won`t get the 8th element in the array, but the Driver with id equals to 7.

- **by conditional index**

  Another case is to use conditions that will consider which element (or elements) should be selected. For this purpose SELECT operators are used which specify conditions for selection. For details how to use SELECT operators refer to the Array Index Operators section

- **by other array index operators and functions**

  To an array in your rules you can apply any index operator which is listed in the Array Index Operators section, or a function designed to work with arrays. The full list of OpenL Tablets array functions is provided in the Appendix B: Functions Used in OpenL Tablets.

Array Index Operators

Array index operators are operators which facilitate working with arrays in rules. Index operators are specified in square brackets of the array and apply particular actions to array elements.

For better understanding of index operators usage, let us provide a detailed description of them along with examples. OpenL Tablets supports the following index operators:

- **SELECT Operators**

  There are cases when it's required to use conditions that will consider which element (or elements) of the array should be selected. For example, if we have a Datatype Driver with fields; *name* (of type String), *age* (of type Integer), etc and we need to select all drivers with name John with age under 20. Then we can use the SELECT operator realizing conditional index like `arrayOfDrivers[select all having name == "John" and age < 20]`.

  There are two different types of SELECT operators:

  - Index operator that **returns the first element satisfying the condition**.

    Returns the first matching element or null if there is no such element.

    Syntax: `array[!@ <condition>]` or `array[select first having <condition>]`

    Example: `arrayOfDrivers[!@ name == "John" and age < 20]`

  - Index operator that **returns all elements satisfying the condition**.

    Returns the array of matching elements or empty array if there are no such elements.

    Syntax: `array[@ <condition>]` or `array [select all having <condition>]`

    Example: `arrayOfDrivers[@ numAccidents > 3]`

- **ORDER BY Operators**

  These operators are intended to sort elements of the array. For example, we have a Datatype *Claim* with fields; *lossDate* (of type Date), *paymentAmount* (of type Double), etc and we need to sort all claims by loss date starting with the earliest one. Then we should use ORDER BY operator like `claims[order by lossDate]`.
  There are two ways of sorting:

  - **sort elements with increasing ordering**.

Syntax: `array[^@ <expression>]` or `array[order by <expression>]` or `array[order increasing by <expression>]`

Example: `claims[^@ lossDate]`

- **sort elements with decreasing ordering**.

Syntax: `array[v@ <expression>]` or `array[order decreasing by <expression>]`

Example: `claims[v@ paymentAmount]`

The operator returns the array with ordered elements. It saves element order in case of equal elements. `<expression>`, by which ordering performs, must have comparable type (like Date, String, Number).

- **SPLIT BY Operator**

If you need to **split array elements into groups by some criteria**, just use SPLIT BY operator which returns collection of arrays with elements in each array of the same criteria. For example, `codes = {"5000", "2002", "3300", "2113"}; codes[split by substring(0,1)]` will produce 3 collections `{"5000"}, {"2002", "2113"} and {"3300"}` which unite codes with equal first number.

Syntax: `array[~@ <expression>]` or `array[split by <expression>]`

Example: `orders[~@ orderType]`

where orders of *Order*[] Datatype, custom Datatype *Order* has a field *orderType* for defining a category of the *Order*. The operator from the example produces *Order*[][] split by different categories.

So SPLIT BY operator returns 2-dimentional array containing arrays of elements split by equal value of <expression>. Saves relative element order.

- **TRANSFORM TO Operators**

This operator gives us an opportunity to turn source array elements into another transformed array in a quick way. Let us assume that we have the collection of *claims* and we'd like to return *claim ID* and *loss date* information for each *claim* (in form of array of strings). Then we could use TRANSFORM TO operator like `claims[transform to id + " - " + dateToString(lossDate, "dd.MM.YY")]`.

There are two types of transforming:

- Index operator that **transforms elements and returns the whole transformed array**.

Syntax: `array[*@ <expression>]` or `array[transform to <expression>]`

Example: `drivers[*@ name]`

- Index operator that **transforms elements and returns just unique elements of the transformed array**.

Syntax: `array[*!@ <expression>]` or `array[transform unique to <expression>]`

Example: `drivers[*!@ vehicle]`

The example above produces collection of *vehicles* , and in this collection each *vehicle* is listed only once (without identical vehicles).

The operator return array of the `<expression>` Type. It saves the order of the elements.

Any field, method of the collection element, any OpenL Tablets function can be used in `<condition>` / `<expression>`, like in examples: `claims[order by lossDate]`, where `lossDate` is a field of the array element Claim; `arrayOfCarModels[@ contains("Toyota")]`, where `contains` is a method of String element of the array `arrayOfCarModels`.

Advanced usage:

Let us consider a case when the name of an array element needs to be referred explicitly in condition/expression. For example, the policy has a collection of drivers (of Driver[] Datatype) and we want to select all *policy drivers* with *age* less than 19, except for the primary *driver*. The following syntax with explicit definition of the collection element `Driver d` allows us to do that:

```
policy.drivers[(Driver d) @ d != policy.primaryDriver && d.age < 19]
```

The expression could be written without type definition in case when the element type is known:

```
policy.drivers[(d) @ d != policy.primaryDriver && d.age < 19]
```

Note: For experienced users. There is a possibility to apply array index operators for Lists. Usually in this case it's necessary to use named element to define type of the list's components like `List claims = policy.getClaims(); claims[(Claim claim) order by claim.date]` or `List claims = policy.getClaims(); claims[(Claim claim) ^@ date]`.

Rules Applied to Array

Regarding arrays, OpenL Tablets provides a feature that allows you to apply a rule (that is intended for working with one value) to an array of values. The following example demonstrates this feature in very simple way.

Spreadsheet SpreadsheetResult PolicyCalculation (Policy policy)	
	Value
Policy : Policy	= policy
Vehicles : SpreadsheetResult[]	= VehicleCalculation (vehicles)
Premium	= sum (GetPremium ($Vehicles)) - ClientDiscount (clientTier)

Spreadsheet SpreadsheetResult VehicleCalculation (Vehicle vehicle)	
	Value
Age	= CurrentYear() - year
BasePremium	= BasePremium (carType)
Surcharge	= AgeSurcharge ($Age)
VehicleDiscount	= VehicleDiscount (airbagType, hasAlarm)
Premium	= ($BasePremium + $Surcharge) * (1 - $VehicleDiscount)

Figure 78: Applying a rule to an array of values

The rule **VehicleCalculation** is designed for working with one vehicle as input parameter and returns one Spreadsheet as a result. In the example this rule is applied for an array of vehicles which means that it is executed for each vehicle and returns an array of Spreadsheet results.

Working with Data Types

Data in OpenL Tablets should have a type of data defined. Datatype indicates the meaning of the data, their possible values and instructs OpenL Tablets how it can process, what operations, rules can perform and what it should do with that data.

All data types used in OpenL Tablets can be divided into two groups:

- Predefined Data Types
- Custom Data Types.

Predefined Data Types are those existing in OpenL Tablets that you can use but cannot modify. Custom Data Types can be created by a user as described in the Datatype Table section.

This section describes Predefined Data Types that include the following ones:

- Simple Data Types
- Value Data Types
- Range Data Types

Simple Data Types

The following table lists Simple Data Types that you will use in your business rules in OpenL Tablets.

Data Type	Description	Examples	Usage in OpenL Tablets	Notes
Integer	It is used to work with whole numbers (without fraction points)	8; 45; 12; 356; 2011	It is common for representing a variety of numbers such as driver's age, a year, a number of points, mileage, etc.	Not exceeding 2,147,483, 647
Double	It is used for operations with fractional numbers. Can hold very large (or very small) numbers.	8.4; 10.5; 12.8; 12,000.00; 44.41666666666664	It is commonly used for calculating balances or discount values, for representing exchange rates, a monthly income, etc.	
BigInteger	It is used to operate with whole numbers that exceed the values allowed by the Integer data type (The maximum Integer value is 2147483647).	7,832,991,657,779;20,000,000,013	This Data Type is only used for operations on very big values – over two billion, for example, dollar deposit in Bulgarian Leva equivalent.	
BigDecimal	It enables to represent decimal numbers with very high precision. Can be used to work with decimal values that have more than 16 significant digits especially when precise rounding is required.	0,6666666666666666667	This Data Type is often used for currency calculations or in financial reports that require exact mathematical calculations, for example a year bank deposit premium calculation.	
String	The String Data Type is used to represent text rather than numbers. String values are comprised of a set of characters that can also contain spaces and numbers. For example, the word "Chrysler" and the phrase "The Chrysler factory warranty is valid for 3 years" are both Strings.	John Smith, London, Alaska, BMW; Driver is too young.	This Data Type is used for representing cities, states, people names, car models, genders, marital statuses, as well as messages such as warnings, reasons, notes, diagnosis, etc.	
Boolean	This Data Type has only two possible values: `true` and `false`. For example, if a Driver is trained (condition is true) then his or her insurance premium coefficient is 1.5; if the Driver is not trained	true; yes; y; false; no; n	Is used to handle conditions in OpenL Tablets.	The synonym for 'true' is 'yes', 'y'; for 'false' – 'no', 'n'

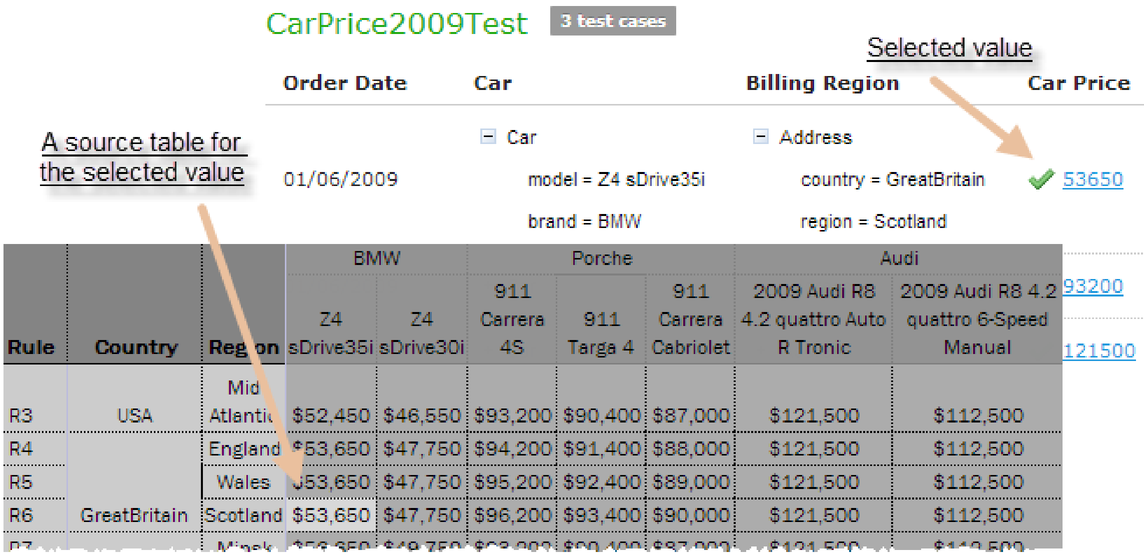
	(the condition is false) then the coefficient is 0.25.			
Date	It is used to operate with dates.	06/05/2010; 01/22/2014; 11/07/95; 1/1/1991.	This Data Type represents any dates such as policy effective date, date of birth, report date etc. If date is defined as text cell value it is expected in <month>/<date>/<year> format.	

Byte, Character, Short, Long, and Float data types are rarely used in OpenL Tablets, so we will only provide here the ranges of the values. For more information about them please refer to appropriate Java portal pages.

Data Type	Min	Max
Byte	-128	127
Character	0	65535
Short	-32768	32767
Long	-9223372036854775808	9223372036854775807
Float	$1.5*10^{-45}$	3.4810^{38}

Value Data Types

In OpenL Tablets *Value Data Types* are completely the same as Simple Data Types except for an *explanation* — a clickable field that you can see in the test results table in OpenL Tablets WebStudio as shown in the illustration below. These data types provide detailed information on results of the rules testing and are useful for working with calculated values to have better debugging capabilities. By clicking the linked value you can view the source table for that value and get information on how the value was calculated.



Rule	Country	Region	BMW Z4 sDrive35i	BMW Z4 sDrive30i	Porche 911 Carrera 4S	Porche 911 Targa 4	Porche 911 Carrera Cabriolet	Audi 2009 Audi R8 4.2 quattro Auto R Tronic	Audi 2009 Audi R8 4.2 quattro 6-Speed Manual
R3	USA	Mid Atlantic	$52,450	$46,550	$93,200	$90,400	$87,000	$121,500	$112,500
R4		England	$53,650	$47,750	$94,200	$91,400	$88,000	$121,500	$112,500
R5		Wales	$53,650	$47,750	$95,200	$92,400	$89,000	$121,500	$112,500
R6	GreatBritain	Scotland	$53,650	$47,750	$96,200	$93,400	$90,000	$121,500	$112,500

Figure 79: Usage of Value Data Type

OpenL Tablets supports the following Value Data types: ByteValue, ShortValue, IntValue, LongValue, FloatValue, DoubleValue, BigIntegerValue, BigDecimalValue.

Range Data Types

You will want to use the *Range Data Types* in cases when your business rule should be applied to a group of values. For example, a driver's insurance premium coefficient is usually the same for all drivers from within a particular age group. So you can define a *range* of ages, and create one rule for all drivers from within that range. All that you need to inform OpenL Tablets that the rule shall be applied to a group of drivers is to declare driver's age as the Range Data Type.

There are two Range Data Types in OpenL Tablets:

IntRange – The **IntRange** Data Type is intended for processing whole numbers within an interval. For example, vehicle or driver age for calculation of insurance compensations, years of service when calculating the annual bonus and so on.

DoubleRange – The **DoubleRange** Data Type is used for operations on fractional numbers within a certain interval. For instance, annual percentage rate in banks depends on amount of deposit which is expressed as intervals: 500 – 9,999.99; 10,000 – 24,999.99 etc.

The illustration below provides very simple example of how to use Range Data Type. The value of discount percentage depends on the number of orders and is the same for 4 to 5 orders and 7 to 8 orders. We have defined amount of cars per order as IntRange Data Type. For number of orders from, for example, 6 to 8 the rule for calculating the discount percentage is the same: The discount percentage is 10.00% for BMW, 4.00% for Porsche, and 6.00% for Audi.

Rules DoubleValue getDiscountPercentage(Car car, int numberOfCars)				
properties	category	Rules - Discounts		
C1	//Description	HC1	RET1	
numberOfCars		brand	discountPercentage	
IntRange amountPerOrder		CarBrand carBrand	DoubleValue discountPercentage	
Number of Orders	**Discount Description**	**BMW**	**Porche**	**Audi**
1	No discount for any brand	0,00%	0,00%	0,00%
2	Min discount applied	1,00%	1,00%	1,00%
3	+1% discount	2,00%	2,00%	2,00%
4 - 5	Depends on car brand	5,00%	3,00%	4,00%
6 - 8	Depends on car brand	10,00%	4,00%	6,00%
> 8	Depends on car brand	15,00%	5,00%	8,00%

Figure 80: Usage of the Range Data Type

Supported range formats:

1. `"<min_number> - <max_number>"` – borders will be included to range.
1. `"<min_number> .. <max_number>"` – the same as 1.
2. `"<min_number> … <max_number>"` – borders will not be included to range. Important: using of ".." and "..." requires spaces between numbers and dots.
3. `"<<max_number>"` – the min_number will be `Integer.MIN_VALUE` and will be included to range. `max_number` will be out of the range.
4. `"<=<max_number>"` – the min_number will be the same as in previous case, `max_number` will be also included to range.
5. `"><min_number>"` – max_number in current case will be the `Integer.MAX_VALUE` and will be included to range. `min_number` will be out of range.
6. `">=<min_number>"` – max_number will be the same as in previous case. `min_number` will be included to range.
7. `"<min_number>+"` – The same to `">= <min_number>"`
8. `"[<min_number>; <max_number>")` – mathematic definition for ranges with using of square brackets for including borders and round brackets for not including.
9. `"[<min_number> .. <max_number>)"` – brackets are used to include or not borders. See 9.
10. `"<min_number> and more"` – the same to 7.
11. `"more than <min_number>"` – the same to 6.
12. `"less than <max_number>"` – the same to 4.

Also numbers can be enhanced with `$` sign as prefix and `K`, `M`, `B` as postfix, e.g. `$1K = 1000`. For using negative values use '`-`' (minus) sign before number (e.g. `-<number>`).

Working with Functions

In the previous section we discussed Data Types that OpenL Tablets uses for representing your data in the system. To implement your business logic in the rules, you will use *functions.* For example, the **Sum** function is used to calculate a sum of values, the **Min/Max** functions enable you to find the minimum or maximum values in a set of values, etc. This section describes OpenL Tablets functions and provides some simple examples of their usage. All the functions can be divided into the following groups:

- Math functions
- Array processing functions

- Date functions
- String functions
- Errors handling functions

Understanding OpenL Function Syntax

This section provides a brief description of how the functions work in OpenL Tablets. Any function is represented by the function name (or identifier) such as **sum**, **sort**, **median**; the function parameter(s); and a value (or values) that the function returns. For example, in the max(value1, value2) expression, 'max' is the rule/function name, (value1, value2) – are the function parameters, i. e. values that take part in the action. If you determine value1 and value2 as 50 and 41, the given function will look as follows: max(50, 41) and returns '50' in result as the biggest number in the couple. If we want an action to be performed in a rule, we should use the corresponding function in the rules table. For example, to calculate the best result for a gamer in the example below, we shall use the **max** function and write *max(score1, score2, score3)* in the C1 column. This expression instructs OpenL Tablets to select the maximum value in the set. The **contains** function can be used then to determine the gamer level.

Subsequent sections provide description for a few often used OpenL Tablets functions. The full list of functions you can find in Appendix B of this Guide.

Math Functions

Math functions serve for performing math operations on numeric data. These functions support all numeric Data Types described in the Working with Data Types section.

The example in the illustration below will help you to better understand how to use functions in OpenL Tablets. The rule in the diagram defines a gamer level depending on his or her best result in three attempts.

Rules String GamerLevelEvaluation(Integer score1, Integer score2, Integer score3)	
C1	RET1
max(score1,score2,score3)	
IntRange	
BestResultEvaluation	**GamerLevel**
0-3	novice
4-6	medium
7-10	senior

Figure 81: En example of using the 'max' function

min/max – returns the smallest or biggest number in a set of numbers (for array, multiple values); the function result is a number.

`min/max(number1, number2, …)`

`min/max(array[])` – you should previously define the array in the given rule table, or in a different table

In the above diagram, the `max(score1,score2,score3)` expression is used to define the highest result for a player. For example, `max(1, 5, 3)` gives us '5' as the result; so the player level will be 'medium' as defined in the `RET1` column.

Sum – adds all numbers in the provided array and returns the result as a number.

`sum (number1, number2, …)`

`sum(array[])`

avg – returns the arithmetic average of array elements. The function result is a number.

`avg(number1, number2, …)`

`avg(array[])`

product – multiplies the numbers from the provided array and returns the product as a number.

`product(number1, number2, …)`

`product(array[])`

mod – returns the remainder after a number is divided by a divisor. The result is a numeric value and has the same sign as the devisor.

`mod(number, divisor)`

`number` is a numeric value whose remainder you wish to find;

`divisor` is the number used to divide the `number`. If the divisor is '0', then the **mod** function returns error

sort – returns values from the provided array in ascending sort; the result is also an array

`sort(array[])`

Date Functions

OpenL Tablets supports a wide range of Date functions that users can apply in their rule tables. The following Date functions return an `int` Data type:

absMonth – returns the number of months since AD.

`absMonth(Date)`

absQuarter – returns the number of quarters since AD as an integer value.

`absQuarter(Date)`

dayOfWeek – takes a Date as input and returns the day of the week on which that date falls; days in a week are numbered from 1 to 7 as follows: 1=Sunday, 2=Monday, 3 = Tuesday, etc.

`dayOfWeek(Date d)`

dayOfMonth – takes a Date as input and returns the day of the month on which that date falls; days in a month are numbered from 1 to 31.

`dayOfMonth(Date d)`

dayOfYear – takes a Date as input and returns the day of the year on which that date falls; days in a year are numbered from 1 to 365.

`dayOfYear(Date d)`

weekOfMonth – takes a Date as input and returns the week of the month within which that date is; weeks in a month are numbered from 1 to 5.

`weekOfMonth(Date d)`

weekOfYear – takes a Date as input and returns the week of the year on which that date falls; weeks in a year are numbered from 1 to 54.

`weekOfYear(Date d)`

second – returns the second (0 to 59) for an input Date

`second(Date d)`

minute – returns the minute (0 to 59) for an input Date

`minute(Date d)`

hour – the hour of the day in 12 hour format for an input Date

`hour(Date d)`

hourOfDay – returns the hour of the day in 24 hour format for an input Date

`hourOfDay(Date d)`

The next Date function returns a String Data type:

amPm(Date d) – returns `Am` or `Pm` value for an input Date

`amPm(Date d)`

The figure below shows values returned by Date functions for a particular input date specified in the MyDate field.

Spreadsheet SpreadsheetResult testDateFunctions(Date MyDate)	
Step	**Value**
Day of week	=dayOfWeek(MyDate)
Day of month	=dayOfMonth(MyDate)
Day of year	=dayOfYear(MyDate)
Week of year	=weekOfYear(MyDate)
Hour of day	=hourOfDay(MyDate)

MyDate	Result	
12/19/2012 07:13	**Step**	**Value**
	Day of week	4.0
	Day of month	19.0
	Day of year	354.0
	Week of year	52.0
	Hour of day	19.0

Figure 82: Date functions in OpenL Tablets

The following Decision table provides a very simple example of how the `dayOfWeek` function can be used when returned value – Risk Factor, – depends on day of week.

Rules DoubleValue RiskFactor3 (Date MyDate)		
C1 dayOfWeek(MyDate) IntRange	RET1	
Day of Week	**Risk Factor (%)**	**Comments**
[2 .. 5]	75%	Monday-to-Wednesday RF
6	85%	Friday RF
	100%	Week-end RF

RiskFactor3Test 3 test cases

Date	Expected	Result
12/21/2012	0.85	✔ 0.85
12/22/2012	1	✔ 1
12/19/2012	0.75	✔ 0.75

Figure 83: A Risk Factor depends on a day of week

ROUND function

The *ROUND function* is used to round a value to a specified number of digits. For example, in financial operations you may want to calculate insurance premium with accuracy up to two decimals. Usually we need to limit a number of digits in long data types such as Double Value or BigDecimal. The function allows you to round a value to an integer number or to a fractional number with limited signs after point.

The ROUND function syntax is:

- `round(DoubleValue)` – rounds to integer number
- `round(DoubleValue, int)` – rounds to fractional number; `int` – a number of digits after point
- `round(DoubleValue, int, int)` – rounds to fractional number and enables you to get results different from usual mathematical rules; in this variant of syntax:
    - the first `int` – a number of digits after point
    - the second `int` – a rounding mode represented by a constant (e.g., `1` – `ROUND_DOWN`, `4`- `ROUND_HALF_UP`)

round(DoubleValue)

Use this syntax to round to an integer number. The illustration below provides an example of the syntax usage.

Rules DoubleValue roundToInteger (DoubleValue value2)	
C1	RET1
true	
Boolean condition	
Condition	**Rate**
	=round(value)

Figure 84: Rounding to integer

Testmethod **roundToInteger** roundToInteger Test		
description	value2	_res_
TestID	**TestType**	**Test Result**
Test1	32.285	32
Test2	42.285	42
Test3	52.285	52
Test4	62.285	62
Test5	72.285	72
Test6	82.285	82
Test7	92.285	92
Test8	102.285	102
Test9	112.285	112

Figure 85: Test table for rounding to integer

round(DoubleValue,int)

You will use this syntax to round to a rounds to fractional number; int – a number of digits after point.

Rules DoubleValue round (DoubleValue value)	
C1	RET1
true	
Boolean condition	
Condition	**Rate**
	=round(value,2)

Figure 86: Rounding to a fractional number

Testmethod **round** roundTest		
description	value	_res_
Test ID	**TestType**	**Test Result**
Test1	32.285	32.29
Test2	42.285	42.29
Test3	52.285	52.29
Test4	62.285	62.29
Test5	72.285	72.29
Test6	82.285	82.29
Test7	92.285	92.29
Test8	102.285	102.29
Test9	112.285	112.29

Figure 87: Test table for rounding to a fractional number

round(DoubleValue,int,int)

Enables you to rounds to fractional number and get results different from usual mathematical rules; in this variant of syntax:

- the first `int` – a number of digits after point
- the second `int` – a rounding mode represented by a constant (e.g., `1`– `ROUND_DOWN`, `4`- `ROUND_HALF_UP`)

The following table contains a list of the constants and their descriptions.

Constant	Name	Description
`0`	`ROUND_UP`	Rounding mode to round away from zero.
`1`	`ROUND_DOWN`	Rounding mode to round towards zero
`2`	`ROUND_CEILING`	Rounding mode to round towards positive infinity
`3`	`ROUND_FLOOR`	Rounding mode to round towards negative infinity.
`4`	`ROUND_HALF_UP`	Rounding mode to round towards "nearest neighbor" unless both neighbors are equidistant, in which case round up.
`5`	`ROUND_HALF_DOWN`	Rounding mode to round towards "nearest neighbor" unless both neighbors are equidistant, in which case round down.
`6`	`ROUND_HALF_EVEN`	Rounding mode to round towards the "nearest neighbor" unless both neighbors are equidistant, in which case, round towards the

		even neighbor.
7	ROUND_UNNECESSARY	Rounding mode to assert that the requested operation has an exact result, hence no rounding is necessary.

For more details on the constants representing rounding modes see http://docs.oracle.com/javase/6/docs/api/constant-values.html#java.math.BigDecimal.ROUND_HALF_DOWN.

You will find detailed description of the constants with examples at http://docs.oracle.com/javase/6/docs/api/java/math/RoundingMode.html, in the *Enum Constant Details* section.

The following examples show how the rounding works with the `ROUND_DOWN` constant.

Rules DoubleValue **round** (DoubleValue value)	
C1	RET1
true	
Boolean condition	
Condition	**Rate**
	=round(value,2,1)

Figure 88: Usage of the `ROUND_DOWN` constant

Testmethod **round** roundTest		
description	value	_res_
Test ID	**TestType**	**Test Result**
Test1	32.285	32.28
Test2	42.287	42.28
Test3	52.283	52.28
Test4	62.289	62.28

Figure 89: Test table for rounding to fractional number using the `ROUND_DOWN` constant

ERROR function

The *ERROR* function is used to handle exceptional cases in a rule when appropriate valid returned result cannot be defined. The function returns a message containing a problem description instead and stops processing. The message text is specified as the error function parameter.

In the example below, in case value for coverage limit of an insurance policy is more then 1000$, a rule notifies a user about wrong limit value and stops further processing:

SimpleRules Double **CoveragePremium** (Integer limit)	
Coverage Limit	**Premium**
<= 100	$0
101 - 500	$15
501 - 900	$45
901 - 1000	$60
> 1000	= error ("coverage limit can't be more then 1000$"); 0

Figure 90: Usage of ERROR function

The usage of the function is a little bit tricky. Notice that right after error function any arbitrary value of expected return value data type must be defined just for compilation purposes.

Null Elements Usage in Calculations

This section describes how null elements represented as Value Data Types are processed in calculations.

In some calculations e.g. 'a+b' or 'a*b' values ‘a’ and/or ‘b’ can be ‘null’ elements. If one of the calculated values is ‘null’ the following rules are applied.

If one of calculated values is ‘null’ it is recognized as ‘0’ for sum operations or as ‘1’ for multiply operations.

The following diagrams demonstrate these rules.

Rules DoubleValue Operations (String testType, DoubleValue a, DoubleValue b)		
properties	modifiedOn	27.6.2012
	modifiedBy	LOCAL
C1	RET1	
testType		
String		
Checked operations	**Return**	
SUBTRACT	=a - b	
ADD	=a + b	
DIVIDE	=a / b	
MULTIPLY	=a * b	
POW	=a ** b	

Figure 91: Rules for null elements usage in calculations

The next Test table provides examples of calculations with null values.

Testmethod Operations OperationsTest			
	modifiedOn	27.6.2012	
properties	modifiedBy	LOCAL	
testType	a	b	_res_
TestType	**Val1**	**Val2**	**Test Result**
SUBTRACT	5.0	3.0	2.0
SUBTRACT	5.0		5.0
SUBTRACT		3.0	-3.0
SUBTRACT			
ADD	5.0	3.0	8.0
ADD	5.0		5.0
ADD		3.0	3.0
ADD			
DIVIDE	8.0	4.0	2.0
DIVIDE	8.0		8.0
DIVIDE		4.0	0.25
DIVIDE			
MULTIPLY	8.0	4.0	32.0
MULTIPLY	8.0		8.0
MULTIPLY		4.0	4.0
MULTIPLY			
POW	2.0	3.0	8.0
POW		3.0	0.0
POW	2.0		2.0
POW			

Figure 92: Test table for null elements usage in calculations

NOTE: If all values are ‘null’ the result is also ‘null’.

Chapter 4: OpenL Business Expression Language

OpenL language framework has been designed from the ground to allow flexible combination of grammar and semantics. OpenL Business Expression (BEX) language proves this statement on practice by extending existing OpenL Java grammar and semantics presented in org.openl.j configuration by new grammar and semantic concepts that allow users to write "natural language" expressions.

Java Business Object Model as Basis for OpenL Business Vocabulary

As always, OpenL minimizes the necessary effort required to build a Business Vocabulary. Using of BEX does not require any special mapping, the existing Java BOM automatically becomes the basis for OpenL Business Vocabulary (OBV). For example, the following expressions are equivalent

```
driver.age
```

and

```
Age of the Driver
```

Another example:

```
policy.effectiveDate
```

and

```
Effective Date of the Policy
```

As you can see from these examples, if your Java model correctly reflects Business Vocabulary there is no further action needed. In cases where Java Model is not satisfactory you still can apply custom type-safe mapping (renaming) that always have been part of OpenL Framework.

New Keywords, how to avoid possible naming conflicts

In the previous chapter we introduced new ‘of the’ keyword. There are other (self-explanatory) keywords in BEX language:

- is less than
- is more than
- is less or equal
- is no more than
- is more or equal
- is no less than

We plan to add more keywords to OpenL BEX language, and therefore there is a chance of a name clash with Business Vocabulary. The easiest way to avoid this clash is to use upper case notation when referring to the model attributes, because BEX grammar is case-sensitive and all the new keywords will be in the lower case. For example, there is an attribute called isLessThanCoverageLimit. If you refer to it as is less than coverage limit, there is going to be a name clash with the keyword, but if you write Is Less Than Coverage Limit, no clash will appear. The possible direction in extending keywords is to add numerics, measurement units, measure sensitive comparisons, like is longer than or is colder than etc.

Simplifying Expressions with explanatory variables

For example, we have a (not very) complex expression:

In Java:

```
(vehicle.agreedValue - vehicle.marketValue) / vehicle.marketValue >
limitDefinedByUser
```

In BEX language you can re-write the same expression in a "business-friendly" way:

```
(Agreed Value of the vehicle - Market Value of the vehicle) / Market Value of
the vehicle is more than Limit Defined By User
```

Unfortunately, the more complex is the expression, the less comprehensible the "natural language" expression becomes. OpenL BEX offers you an elegant solution for this problem:

```
(A - M) / M > X, where
  A - Agreed Value of the vehicle,
  M - Market Value of the vehicle,
  X - Limit Defined By User
```

The syntax is pretty similar to the one that have been used in scientific publications and is easily understood by anybody. We believe that the syntax provides the best mix of mathematical clarity and business readability.

Simplifying Expressions by Using *Unique in Scope* concept

Humans differ from computers, in particular, by their ability to understand the scope of the language expression. For example, if we discuss an insurance policy and mention "the effective date" we don't have to say every time the fully qualifying expression "the effective date of the policy", because the context of the effective date is clearly understood. On another hand, if we discuss two policies, for example, the old and the new ones, we have to say "the effective date of the new policy" vs. "the effective date of the old policy". This is necessary, because there are 2 different policies in the context of the conversation.

Similarly, when humans write so called "business documents" – files that serve as a reference point to a rule developer, they also often use an "implied context" in mind. Therefore they often use in documentation business terms such as Effective Date, Driver, Account etc. with implied scope in mind. The "scope resolution" is left to a so-called Rules Engineer, who has to do it by manually analyzing BOM and setting appropriate paths from the root objects.

OpenL BEX tries to close this semantic gap or at least make it a bit narrower by letting use "unique in scope" attributes. For example, if there is only one policy in the scope, user can write just "effective date" instead of "effective date of the policy". OpenL BEX will automatically determine the uniqueness of the attribute and either produce a correct path, or will emit error message in case of an ambiguous statement. The level of the resolution can be modified programmatically and by default equals to 1.

OpenL Vocabulary and OpenL BEX

Since version 5.0.5 OpenL introduced the ability to augment existing Java BOM with different kind of meta-information through OpenL Vocabulary. As always, we made it accessible through Java API and as Excel tables, giving business users access to the Vocabulary. While Vocabulary can be used for many other important activities, it has one feature that is significant in the context of OpenL BEX – Business Object Attribute Aliasing – or in layman's words, the ability to name attribute with alternative names. This gives to user an ability to adopt existing Java model names to the business terminology in a case when Java model does not reflect it properly. See OpenL Vocabulary for more details.

Future developments, compatibility etc

OpenL BEX is a fairly new development, it will evolve to provide user with new convenient features. In particular, right now there is no other way to call methods in OpenL except for the old-fashioned Java/C++ style. Nevertheless, we want to state that existing syntax will remain compatible with all future modifications. Also, the ability to use Java style constructs together with "natural-language" extensions will stay in the language. And, finally, the last word about using NL references in this document – BEX is NOT a NL-tool, it is just a syntax extension of the standard Java grammar that allows your expressions in many cases look like normal English phrases. The result will be as good as your Java BOM is, BEX would not be able to fix a bad design or naming conventions. We strongly recommend that you at least try it and send us your feedback, it does not require any additional efforts, because BEX is now a standard part of OpenL Tablets. You can use in any place where you previously used Java expressions.

OpenL Programming Language Framework

As we all know, Business Rules consist of rules. Each Rule has Condition and Action. Condition is a boolean expression (the one that returns true or false). Action can be any sequence (usually simple) of programming statements. What kind of language is most suitable for this task?

Let's take a look at the expression that probably is as ubiquitous in any BR doc as "if customer's level is GOLD":

```
driver.age < 25
```

From the semantic perspective, the expression intends to define the relationship between some value defined by 'driver.age' expression and literal '25'. One might guess that the English semantic of the statement could be any of if age of the driver is less than 25 years or select drivers who are younger than 25 years old or some other.

From the programming language perspective, the semantic part is irrelevant, the statement should only be:

- a valid statement in the language grammar
- a statement should be correct from the type-checking point of view
- if language is compiled, the valid binary code or some other results of compiling (for example bytecode, or even code in some other target language might also be considered as possible results of the compiling) should be produced from the statement
- some kind of runtime system, interpreter or Virtual Machine should be able to execute (interpret) this statement's compiled code and produce a resulting object

OpenL Grammars

When OpenL parser parses an OpenL expression it produces a Syntax Tree. Each Tree Node has a node type, a literal value, a reference to the source code for displaying errors and debugging, and also may contain children nodes. This is similar to what other parsers do, with one notable exception – the OpenL Grammar is not hard-coded, it can be configured, and different can be used. Having said this, we also must admit that for all the practical purposes, as of today, we distribute only the following grammars implemented in OpenL: **org.openl.j –** based on "classic" Java 1.3 grammar (no templates and exception handling) and **org.openl.bex –** which is basically org.openl.j grammar with "business natural language" extensions. The latter is used by default in OpenL Tablets business rules product.

We also have experimental **org.openl.n3 grammar** and we may add **org.openl.sql** grammar in the future.

The Syntax Tree produced by the **org.openl.j** grammar for the expression we started with will look like this:

```
        <
       / \
      .   25
     / \
driver  age
```

The node types of the nodes are

- **op.binary.lt** for '<'
- **literal.integer** for '25'
- **chain** for '.'
- **identifier** for 'driver'
- **identifier** for 'age'

Node type names are significant, as we will see later, but at this point they look rather like random names.

NOTE. It is also important to recognize that the Grammar we use in org.openl.j is similar not only to Java but to any other language in C/C++/Java/C# family. This makes OpenL easily learned and accepted by the huge pool of available Java/Cxx programmers and adds to it's strength. The proliferation of new languages like Ruby, Groovy, multiple proprietary languages used in different Business Rules Engines, CEP Engines etc., introduced not only the new semantics to the programming community, but also a bunch of new grammars that make the acceptance of the new technologies much harder.

We at OpenL work day and night to stay as close to the Java syntax as possible to make sure that the "entities would not be multiplied beyond necessity". Let's keep the world's linguistic entropy down, folks.

Context, Variables and Types

After the Syntax Tree had been created, the next stage of the compilation process, or Binding, binds syntax nodes to its semantic definitions. At this stage, OpenL uses specific Binders for each node type. The modular structure of OpenL allows to define custom Binders for each node type. Once syntax node had been bound into Bound Node, it has been assigned a type, making the process type-safe.

Most of the time, the standard Java approach is used to assign type to the variable – it should be defined somewhere in the context of the OpenL framework. Typical examples include:

- Method parameter
- Local Variable
- Member of surrounding class (in case of OpenL it is usually the implementation of IOpenClass called Module)
- External types accessed as static, mostly Java classes that are imported into OpenL

Fields and Methods in binding context – this is a feature that does not exist in Java; OpenL allows programmatically add custom types, fields and methods into Binding Context; for different examples of how it could be done you need to take a good look at the source code of OpenLBuilder classes in different packages. For example, org.openl.j automatically imports all the classes from the java.util in addition to the standard java.lang package. Since version 5.1.1 java.math is also being imported automatically

OpenL Type System

Everybody knows that Java is a type-safe language. But it's type-safety ends when Java has to deal with types that lie outside of Java type system – like database tables, http requests or XML files. There are two approaches to deal with those "external" types – use API or use code-generation. API approach is inherently not type-safe, it treats attribute as literal strings, therefore even spelling errors will be visible only in runtime. Another problem with API – it is well, API-specific, so unless the standard API exists, your program becomes dependent on the particular API. The approach with code-generation is better, but it also introduce an extra building step and is dependent on particular generator, especially the part where names and name spaces are converted into Java names and packages. Often, the generators introduce dependencies with runtime libraries that also affect the portability of the code. Finally, generators usually require full conversion from external data into Java objects that may incur an unnecessary performance penalty in the case where you need to access only a few attributes. OpenL Open Type system gives you the simple way to add new types into OpenL language, all you need is to define a class object that implements IOpenClass interface and add it to OpenL type system. The implementations can vary, but access to object's attributes and methods will have the same syntax and will provide the same type-checking in all OpenL code throughout your application.

OpenL Tablets as OpenL Type extension

OpenL Tablets is built on top of OpenL type system, and this allows it to integrate naturally into any Java or OpenL environment. Using OpenL methodology, Decision Tables become Methods, and Data Tables become Fields. The similar conversion happens to all the other project artifacts. It allows for easy modular access to any project's component through Java or OpenL code. An OpenL Tablets project itself becomes a "class" and easy Java access to it is provided through a generated JavaWrapper class.

Operators

Operators are just other methods with priorities defined by the Grammar. OpenL has 2 major types of operators: unary and binary. In addition, there are other operator types used in special cases. Here is a complete list of OpenL operators used in **org.openl.j** Grammar – the one that is used by default in OpenL Tablets product.

When we say that OpenL has a modular structure, we not only refer to the fact that OpenL has configurable, high-level separate components like Parser and Binder; it is also, that each node type can have its own NodeBinder. At the same time, we can assign the single NodeBinder to a group of operators, like we do in the case of the prefix **op.binary**.

NOTE: (op.binary.or '||' and op.binary.and '&&' have separate NodeBinders to provide short-circuiting for boolean operands). For all other binary operators OpenL uses a simple algorithm, based on the operator's node type name. For example, if the node type is 'op.binary.add', the algorithm looks for the method named 'add()' in the following order:

- Tx add(T1 p1, T2 p2) in the namespace org.openl.operators in the BindingContext
- public Tx T1.add(T2 p2) in T1
- static public Tx T1.add(T1 p1, T2 p2) in T1
- static public Tx T2.add(T1 p1, T2 p2) in T2

The found method is then being executed in the runtime. So, if you need to override binary operator t1 OP t2 (where t1, t2 are objects of classes T1, T2), you need to do the following steps:

1. Check the Operators Table and find the operator's type name.
2. The last part of the type name will give you the name of the method that you need to implement
3. Now you have the following options for implementing operators:
    - put it into some class YourCustomOperators as the static method and register the class as the library in org.openl.operators namespace (see OpenLBuilder code for more details).
    - implement as method in T1: public Tx name(T2 p2)
    - implement as method in T1: static public Tx name(T1 p1,T2 p2)
    - implement as method in T2: static public Tx name(T1 p1,T2 p2)

Usually, if T1 and T2 are different, you need to define both OP(T1, T2) and OP(T2, T1), unless you can rely on autocast() operator or Binary Operators' Semantic Map. Autocast can help you skip implementation when you already have an operator implemented for the autocasted type. For example, if you have OP(T1, double), you don't have to implement OP(T1, int), because int is autocasted to double. You may incur some performance penalty by doing this though. Binary Operator Semantic Map is described next.

Binary Operators Semantic Map

Since the version 5.1.1 there is a convenient feature that we call *Operator Semantic Map*. It makes implementing of some of the operators easier by describing properties (*symmetrical* and *inverse*) for some operators.

Unary Operators

For unary operators, the same method resolution algorithm is being applied; the only difference is that there is only one parameter to deal with.

Cast Operators

Cast Operators in general correspond to Java guidelines and come in 2 types: **cast** and **autocast**. **T2 autocast (T1 from, T2 to)** methods used to overload implicit cast operators (like from int to long, so that actually no cast operators are required in code), T2 cast(T1 from, T2 to) methods are used with explicit cast operators.

NOTE: It is important to remember that while both **cast()** and **autocast()** methods require 2 parameters, only T1 from parameter will be actually used. The second parameter is needed to avoid ambiguity in Java method resolution

Strict equality and relation operators

Strict operators are same as their original prototypes but they used the strict comparison for float point values. Float point numbers are used in JVM as value with an inaccuracy. The original relation and equality operators are used inaccuracy of float point operations. For example:

```
1.0 == 1.0000000000000002 – returns true value,

1.0 ==== 1.0000000000000002 (1.0 + ulp(1.0)) – returns false value,
```

where `1.0000000000000002 = 1.0 + ulp(1.0).`

The list of org.openl.j Operators

In the order of priority:

Assignment operators	
Operator	**org.openl.j operator**
=	op.assign
+=	op.assign.add
-=	op.assign.subtract

<table>
<tr><th colspan="2">Assignment operators</th></tr>
<tr><th>Operator</th><th>org.openl.j operator</th></tr>
<tr><td>*=</td><td>op.assign.multiply</td></tr>
<tr><td>/=</td><td>op.assign.divide</td></tr>
<tr><td>%=</td><td>op.assign.rem</td></tr>
<tr><td>&=</td><td>op.assign.bitand</td></tr>
<tr><td>|=</td><td>op.assign.bitor</td></tr>
<tr><td>^=</td><td>op.assign.bitxor</td></tr>
</table>

Conditional Ternary	
Operator	**org.openl.j operator**
? :	op.ternary.qmark

Implication	
Operator	**org.openl.j operator**
->	op.binary.impl [*]

<table>
<tr><th colspan="2">Boolean OR</th></tr>
<tr><th>Operator</th><th>org.openl.j operator</th></tr>
<tr><td>|| or "or"</td><td>op.binary.or</td></tr>
</table>

Boolean AND	
Operator	**org.openl.j operator**
&& or "and"	op.binary.and

<table>
<tr><th colspan="2">Bitwise OR</th></tr>
<tr><th>Operator</th><th>org.openl.j operator</th></tr>
<tr><td>|</td><td>op.binary.bitor</td></tr>
</table>

Bitwise XOR	
Operator	**org.openl.j operator**
^	op.binary.bitxor

Bitwise AND	
Operator	**org.openl.j operator**
&	op.binary.bitand

Equality	
Operator	**org.openl.j operator**
==	op.binary.eq
!=	op.binary.ne

Equality	
Operator	**org.openl.j operator**
====	op.binary.strict_eq (*)
!===	op.binary.strict_ne (*)

Relational	
Operator	**org.openl.j operator**
<	op.binary.lt
>	op.binary.gt
<=	op.binary.le
>=	op.binary.ge
<==	op.binary.strict_lt (*)
>==	op.binary.strict_gt (*)
<===	op.binary.strict_le (*)
>===	op.binary.strict_ge (*)

Bitwise Shift	
Operator	**org.openl.j operator**
<<	op.binary.lshift
>>	op.binary.rshift
>>>	op.binary.rshiftu

Additive	
Operator	**org.openl.j operator**
+	op.binary.add
-	op.binary.subtract

Multiplicative	
Operator	**org.openl.j operator**
*	op.binary.multiply
/	op.binary.divide
%	op.binary.rem

Power	
Operator	**org.openl.j operator**
**	op.binary.pow (*)

Unary Operators	
Operator	**org.openl.j operator**
+	op.unary.positive
-	op.unary.negative

Unary Operators	
Operator	**org.openl.j operator**
++x	op.prefix.inc
--x	op.prefix.dec
x++	op.suffix.inc
x--	op.suffix.dec
!	op.unary.not
~	op.unary.bitnot
(cast)	type.cast
|x|	op.unary.abs [(*)]

[(*)] Operators do not exist in Java Standard, only in org.openl.j, but you can use and overload them at will

The list of opg.openl.j Operator Properties

Symmetrical

```
eq(T1,T2) <=> eq(T2, T1)
add(T1,T2) <=> add(T2, T1)
```

Inverse

```
le(T1,T2) <=> gt(T2, T1)
lt(T1,T2) <=> ge(T2, T1)
ge(T1,T2) <=> lt(T2, T1)
gt(T1,T2) <=> le(T2, T1)
```

Chapter 5: Working With Projects

This section describes creating an OpenL Tablets project. For general information on projects, see Projects.

The following topics are included in this section:

- Project Structure
- Rules Runtime Context
- Project and Module dependencies

Project Structure

The best way to use the OpenL Tablets rule technology in a solution is to create an OpenL Tablets project in OpenL Tablets WebStudio. A typical OpenL Tablets project contains just Excel files which are physical storage of rules and data in the form of tables. On logical structure level, **Excel files represent modules of the project**. Additionally, the project can contain rules.xml, Java classes, JAR files (according to developer's needs) and other related documents (guides, instructions etc.).

Thereby, the structure can be adjusted according to the developer's preferences, for example, to comply with the Maven structure.

Note: For experienced users. File `rules.xml` of the project is a rules project descriptor that contains information about a project and configuration details. For instance, a user may redefine a module name there (by default, a module name is a name of corresponding Excel file). If you perform project details changes via WebStudio – `rules.xml` file is automatically created/updated accordingly.
For more information about configuring `rules.xml` see OpenL Tablets – Developer Guide, the *Rules project descriptor* section.

Multi Module Project

All modules inside one project have mutual access to each other's tables. It means that a rule/table of a module of a project is accessible (can be referenced and used) from any rule of any module of the same project. Projects with several rule modules are called as Multi Module Projects.

In order to define compilation order of modules of a project – module dependencies are used. If you need to run a rule table from another project – project dependencies ought to be used. Dependency usage is described in details in Project and Module dependencies chapter.

Creating a Project

The simplest way to create an OpenL Tablets project is to create a project from Template in the installed OpenL Tablets WebStudio.

A new project is created containing simple template files that developers can use as the basis for a custom rule solution.

Rules Runtime Context

OpenL Tablets supports rules overloading by metadata (table properties). What is this? Sometimes user needs business rules that work differently but have the same inputs. Let's imagine that you provide vehicle insurance and have a premium calculation rule for it. For example, the algorithm of premium calculation is following:

```
PREMIUM = RISK_PREMIUM + VEHICLE_PREMIUM + DRIVER_PREMIUM - BONUS
```

For different US states you have different bonus calculation policies. In simple way for all the states you must have different calculations:

```
PREMIUM_1 = RISK_PREMIUM + VEHICLE_PREMIUM + DRIVER_PREMIUM - BONUS_1, for state
#1
PREMIUM_2 = RISK_PREMIUM + VEHICLE_PREMIUM + DRIVER_PREMIUM - BONUS_2, for state
#2
...
PREMIUM_N = RISK_PREMIUM + VEHICLE_PREMIUM + DRIVER_PREMIUM - BONUS_N, for state
#N
```

OpenL Tablets provides more elegant solution for this case.

```
PREMIUM = RISK_PREMIUM + VEHICLE_PREMIUM + DRIVER_PREMIUM - BONUS*, where
BONUS* = BONUS_1, for state #1
BONUS* = BONUS_2, for state #2
...
BONUS* = BONUS_N, for state #N
```

So you have one common premium calculation rule and several different rules for bonus calculation. When you run the premium calculation rule you should provide current state as additional input for OpenL Tablets to choose the appropriate rule. Using this information OpenL Tablets makes decision which bonus method should be used (invoked). This kind of information is called runtime data and should be set into *runtime context* before you run the calculations.

The following OpenL table snippets show our sample in action:

SimpleRules DoubleValue **Bonus**()		
properties	state	STATE #1
Bonus Premium		
$100		

SimpleRules DoubleValue **Bonus**()		
properties	state	STATE #2
Bonus Premium		
$150		

SimpleRules DoubleValue **Bonus**()		
properties	state	STATE #N
Bonus Premium		
$200		

Figure 93: The group of Decision Tables overloaded by properties

All tables for bonus calculation have the same header but different *state* property value.

OpenL Tablets has predefined runtime context which already has several properties.

Managing Rules Runtime Context from Rules

There is a possibility to work with runtime context from OpenL Tablets rules. It is provided by the following additional internal methods for modification, retrieving and restoring runtime context:

1. **getContext()**: returns copy of the current runtime context.
2. **emptyContext()** : returns new empty runtime context.
3. **setContext(IRulesRuntimeContext context)** : replaces current runtime context by the specified.

Method DoubleValue calcRateForDate (Policy policy, Date date)
IRulesRuntimeContext context = getContext();
context.currentDate = date;
setContext(context);
return calcRate(policy);

4. **modifyContext(String propertyName, Object propertyValue)** : modifies current context by one property: adds new or replaces by specified if property with such a name already exists in current context. Note: all properties from current context will be available after modification, so it is only one property update.

Rules DoubleValue calcRateForState(int homeIndex, Policy policy)	
A1	RET1
modifyContext("usState",stateToSet)	result
UsStatesEnum stateToSet	DoubleValue result
State	Check
=policy.home[homeIndex].state	=calc(policy)

5. **restoreContext() :** discharges the last changes in runtime context. It means that context will be rolled back to the state before the last **setContext** or **modifyContext**.

Method DoubleValue calcAutoRateForMO (Policy policy)
IRulesRuntimeContext context = emptyContext();
context.lob = "auto";
context.usState = UsStatesEnum.MO;
setContext(context);
DoubleValue res = calcRate(policy);
restoreContext();
return res;

ATTENTION: You should control all changes and rollbacks manually: all changes applied to runtime context will remain after the execution of the rule is completed. So you should make sure that the changed context is restored after the rule had been executed to prevent unexpected behavior of rules caused by unrestored context.

Project and Module dependencies

Dependencies are used for more flexibility and convenience. They may divide rules into different modules and structure them in a project or add other related projects to the current one. For example, a user has several projects with different modules, all his projects share the same domain model or use similar helpers rules, so to avoid rules duplication, user can put his common rules and data to separate module and add this module as dependency for all modules where it is needed.

Dependencies description

Module dependency feature allows making hierarchy of modules when rules of one module depend on rules of another one. As it is mentioned before, all modules of one project have mutual access to each other's tables. Therefore, module dependencies are intended to order them in the project (if it is required for compilation purposes). Module dependencies are commonly established among modules of the same project (there is one exception described below).

The following diagram illustrates a project in which content of *Module_1* and *Module_2* depends on content of *Module_3* (thin black arrows are module dependencies):

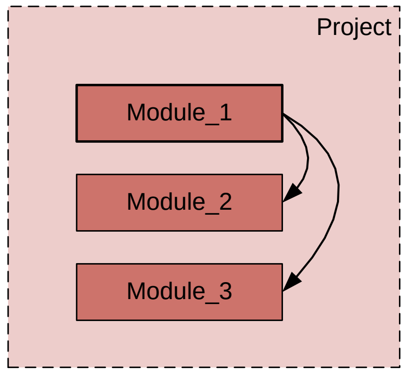


Figure 94: Example of a project with modules' hierarchy

In addition, **Project dependency** enable to access modules of other projects from the current one:

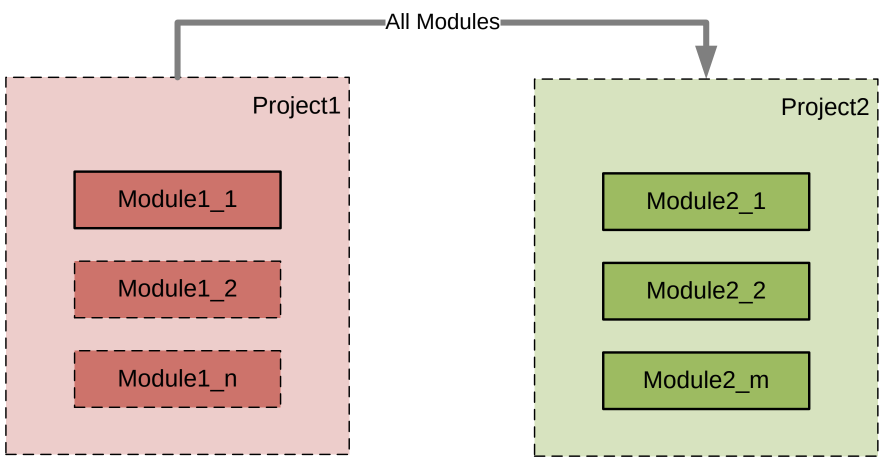


Figure 95: Example of a project dependency (with all modules)

The diagram above shows that any module of *Project1* can execute any table of any module of *Project2* (thick gray arrow with label '**All Modules**' is a project dependency with all dependency project modules included). This is equivalent to the following schema when each module of *Project1* has implicit dependency declaration to each module of *Project2*:

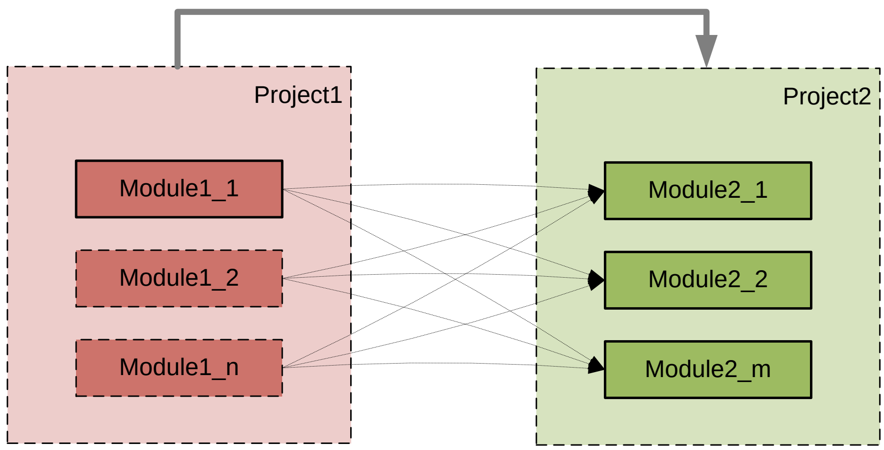


Figure 96: Interpretation of a project dependency (with all modules)

So, project dependency with ‘All Modules’ setting switched on provides an access to any module of a dependency project from the current (root) project.

You may **combine module and project dependencies** in case a particular module of another project needs to be used only:

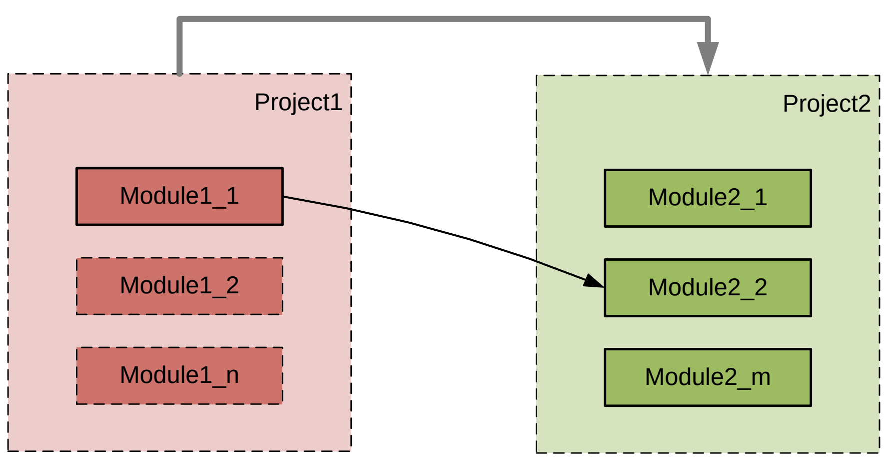


Figure 97: Example of a project and module dependencies combined

In the example, from defined external *Project2*, only the content of *Module2_2 is accessible from Project1*, and no others (thick gray arrow without label is a project dependency which defines other projects where dependency module can be located).

It means that in case project dependency does not have ‘All Modules’ setting enabled, dependencies are determined on module level, and such project dependencies just serves the isolation purpose - to make it possible to get a dependency module from particular external projects.

After adding a dependency, all its rules, data fields, datatypes are accessible from root module. Root module can call dependency rules.

Dependencies configuration

Let's describe in details, how dependencies are configured.

To add dependency to module, simply add the instruction to Environment table, use command *dependency* and the name of the module you want to add. Module can contain any number of dependencies. Dependency modules may also have dependencies. Avoid cyclic dependencies:

Environment	
dependency	Module2_2

Figure 98: Example of configuring module dependencies

Project dependency is configured in a *rules project descriptor* (`rules.xml` file which is created in the project root folder) in *Dependency* section where tag *name* is used for defining the name of a dependency project, tag *autoIncluded* – whether 'All Modules' option is switched on or off:

```
<dependencies>
  <dependency>
    <name>Project Name</name>
    <autoIncluded>false</autoIncluded>
  </dependency>
</dependencies>
```

Figure 99: Example of configuring project dependencies – fragment of rules.xml

For more information about configuring `rules.xml` see OpenL Tablets – Developer Guide, the *Rules project descriptor* section.

By a business user, project dependencies are easily set and updated in WebStudio (for details refer to *OpenL Tablets WebStudio User's Guide*, *Defining Project Dependencies* section).

Project can contain any number of dependencies. Dependency projects may also have dependencies. Avoid cyclic dependencies. Module names of root and dependency projects MUST be unique.

When OpenL Tablets is processing module, if there is any dependency declaration, it tries to load and compile it first before root module. When all required dependencies have been successfully compiled, OpenL compiles root module itself with awareness about rules and data from dependencies.

Components behavior

All OpenL components can be divided into 3 types:

- Rules (Decision table, Spreadsheet, Method, TBasic, etc)
- Data (Data table)
- Datatypes (Datatype table)

The table below describes the behavior of different OpenL components in dependency infrastructure.

Operations \ Components	Rules	Datatypes	Data
Can access components in root module from dependency	yes	yes	yes
Both root and dependency modules contain similar component	1. Rules with the same signature and without dimension properties: duplicate exception. 2. Methods with the same signature and with a number of dimension properties: they will be wrapped by Method Dispatcher. At runtime will be executed method that matches the runtime context properties. 3. Methods with the same signature and with property active: only one table can be set to true (appropriate validation will check this case at compile time).	Duplicate exception	Duplicate exception
None of root and dependency modules contain the component	'There is no such method' exception while compilation	'There is no such datatype' exception while compilation	'There is no such field' exception while compilation

Glossary

Root module – the module that has dependency declaration (explicit via Environment or implicit via project dependency) to other module.

Root project – the project that has dependency declaration to other project.

Dependency module – the module that is used as a dependency.

Dependency project – the project that is used as a dependency.

Appendix A: BEX Language Overview

This section provides a general overview of the BEX language that can be used in OpenL Tablets expressions.

The following topics are included in this section:

- Introduction to BEX
- Keywords
- Simplifying Expressions

Introduction to BEX

BEX language allows a flexible combination of grammar and semantics by extending the existing Java grammar and semantics presented in the `org.openl.j` configuration using new grammar and semantic concepts. It enables users to write expressions similar to natural human language.

BEX does not require any special mapping; the existing Java business object model automatically becomes the basis for open business vocabulary used by BEX. For example, Java expression 'policy.effectiveDate' is equivalent with BEX expression 'Effective Date of the Policy'.

If the Java model correctly reflects business vocabulary, there is no further action required. In case the Java model is not satisfactory, custom type-safe mapping or renaming can be applied.

Keywords

The following table represents BEX keyword equivalents to Java expressions:

BEX keywords	
Java expression	**BEX equivalents**
`==`	• `equals to` • `same as`
`!=`	• `does not equal to` • `different from`
`a.b`	`b of the a`
`<`	`is less than`
`>`	`is more than`
`<=`	• `is less or equal` • `is in`
`!>`	`is no more than`
`>=`	`is more or equal`
`!<`	`is no less than`

Because of these keywords, name clashes with business vocabulary can occur. The easiest way to avoid clashes is to use upper case notation when referring to model attributes because BEX grammar is case sensitive and all keywords are in lower case.

For example, assume there is an attribute called `isLessThanCoverageLimit`. If it is referred as 'is less than coverage limit', a name clash with keywords 'is less than' occurs. The workaround is to refer to the attribute as 'Is Less Than Coverage Limit'.

Simplifying Expressions

Unfortunately, the more complex an expression is, the less comprehensible the natural language expression becomes in BEX. For this purpose, BEX provides the following methods for simplifying expressions:

- Notation of Explanatory Variables
- Uniqueness of Scope

Notation of Explanatory Variables

BEX supports a notation where an expression is written using simple variables followed by the attributes they represent. For example, assume the following expression is used in Java:

```
(Agreed Value of the vehicle - Market Value of the vehicle) / Market Value of
the vehicle is more than Limit Defined By User
```

As can be seen from the example, the expression is hard to read. However, the expression is much simpler if written according to the notion of explanatory variables as follows:

```
(A - M) / M > X, where
  A - Agreed Value of the vehicle,
  M - Market Value of the vehicle,
  X - Limit Defined By User
```

This syntax is similar to the one used in scientific publications and is much easier to read for complex expressions. It provides a good mix of mathematical clarity and business readability.

Uniqueness of Scope

BEX provides another way for simplifying expressions using the concept of unique scope. For example, if there is only one policy in the scope of expression, the user can write 'effective date' instead of 'effective date of the policy'. BEX automatically determines the uniqueness of the attribute and either produces a correct path or emits an error message in case of ambiguous statement. The level of the resolution can be modified programmatically and by default equals 1.

Appendix B: Functions Used in OpenL Tablets

This section provides a complete list of functions available in OpenL Tablets.

Math Functions

Math Functions		
Function	**Description**	**Comment**
abs(a)	Returns the absolute value of a number.	
acos(double a)	Returns the arc cosine of a value; the returned angle is in the range 0.0 through pi.	
asin(double a)	Returns the arc sine of a value; the returned angle is in the range -pi/2 through pi/2.	
atan(double a)	Returns the arc tangent of a value; the returned angle is in the range -pi/2 through pi/2.	
atan2(double y, double x)	Returns the angle theta from the conversion of rectangular coordinates (x, y) to polar coordinates (r, theta).	
cbrt(double a)	Returns the cube root of a double value.	
ceil(double a)	Returns the smallest (closest to negative infinity) double value that is greater than or equal to the argument and is equal to a mathematical integer.	
copySign(magnitude, sign)	Returns the first floating-point argument with the sign of the second floating-point argument.	
cos(double a)	Returns the trigonometric cosine of an angle.	
cosh(double x)	Returns the hyperbolic cosine of a double value.	
cosh(double x)	Returns the hyperbolic cosine of a double value.	
exp(double a)	Returns Euler's number e raised to the power of a double value.	
expm1(double x)	Returns ex -1	
floor(double a)	Returns the largest (closest to positive infinity) double value that is less than or equal to the argument and is equal to a mathematical integer.	
format(double d)	Formats double value.	
format(double d, String fmt)	Formats double value according to Format fmt.	
getExponent(a)	Returns the unbiased exponent used in the representation of a.	
getExponent(double x, double y)	Returns sqrt(x2 +y2) without intermediate overflow or underflow.	
IEEEremainder(double f1, double f2)	Computes the remainder operation on two arguments as prescribed by the IEEE 754 standard.	

Math Functions		
Function	**Description**	**Comment**
log(double a)	Returns the natural logarithm (base e) of a double value.	
log10(double a)	Returns the base 10 logarithm of a double value.	
log1p(double x)	Returns the natural logarithm of the sum of the argument and 1.	
mod(number, divisor)	Returns the remainder after a number is divided by a divisor.	
nextAfter(start, direction)	Returns the floating-point number adjacent to the first argument in the direction of the second argument.	
pow(double a, double b)	Returns the value of the first argument raised to the power of the second argument.	
quotient(number, divisor)	Returns the quotient from division number by divisor.	
random()	Returns a double value with a positive sign, greater than or equal to 0.0 and less than 1.0.	
rint(double a)	Returns the double value that is closest in value to the argument and is equal to a mathematical integer.	
round(value)	Returns the closest value to the argument, with ties rounding up.	
round(value, int scale, int roundingMethod)	Returns a BigDecimal whose scale is the specified value, and whose unscaled value is determined by multiplying or dividing this BigDecimal's unscaled value by the appropriate power of ten to maintain its overall value.	
scalb(a, int scaleFactor)	Return a × 2scaleFactor rounded as if performed by a single correctly rounded floating-point multiply to a member of the double value set.	
signum(double d) / (float f)	Returns the signum function of the argument; zero if the argument is zero, 1.0 if the argument is greater than zero, -1.0 if the argument is less than zero.	
sin(double a)	Returns the trigonometric sine of an angle.	
sinh(double x)	Returns the hyperbolic sine of a double value.	
sqrt(double a)	Returns the correctly rounded positive square root of a double value.	
tan(double a)	Returns the trigonometric tangent of an angle.	
tanh(double x)	Returns the hyperbolic tangent of a double value.	
toDegrees(double angrad)	Converts an angle measured in radians to an approximately equivalent angle measured in degrees.	
toRadians(double angdeg)	Converts an angle measured in degrees to an approximately equivalent angle measured in radians.	

Math Functions		
Function	**Description**	**Comment**
ulp(value)	Returns the size of an ulp of the argument.	

Array Functions

Array Functions		
Function	**Description**	**Comment**
add(array[],element)	Copies the given array and adds the given element at the end of the new array.	
add(array[],index, element)	Inserts the specified element at the specified position in the array.	
addAll(array1[], array2[])	Adds all the elements of the given arrays into a new array.	
addIgnoreNull(array[], element)	Copies the given array and adds the given element at the end of the new array.	
addIgnoreNull(array[], int index, element)	Inserts the specified element at the specified position in the array.	
allTrue(Boolean[] values)	Returns true if all array elements are true.	
anyTrue(Boolean[] values)	Returns true if any array element is true.	
avg(array[])	Returns the arithmetic average of the array of number elements.	
big(array[], int position)	Removes null values from array, sorts an array in descending order and returns the value at position '*position*'.	
contains(array[], elem)	Checks if the value is in the given array.	
indexOf(array[], elem)	Finds the index of the given value in the array.	
intersection(String[] array1, String[] array2)	Returns a new array containing elements common to the two arrays.	
isEmpty(array[])	Checks if an array is empty or null.	
max(array[])	Returns the maximal value in the array of numbers.	
min(array[])	Returns the minimal value in the array of numbers.	
noNulls(array[])	Checks if the array is nonempty and has only nonempty elements.	

Array Functions		
Function	**Description**	**Comment**
product(array [] values)	Multiplies the numbers from the provided array and returns the product as a number.	
remove(array [] , int index)	Removes the element at the specified position from the specified array.	
removeElement(array [], element)	Removes the first occurrence of the specified element from the specified array.	
removeNulls(T[] array)	Return a new array without null elements.	
slice(Array[], int startIndexInclusive, int endIndexExclusive)	Returns a part of array from startIndexInclusive to endIndexExclusive.	
small(Array[], int position)	Removes null values from array, sorts an array in ascending order and returns the value at position 'position'.	
sort(Array[])	Sorts the specified array of values into ascending order, according to the natural ordering of its elements.	
sum(array[])	Returns the sum of numbers in the array,	

Date Functions

Date / Time Functions		
Function	**Description**	**Comment**
absMonth(Date)	Returns the number of months since AD.	
absQuarter(Dat)	Returns the number of quarters since AD as an integer value.	
amPm(Date d)	Returns `Am` or `Pm` value for an input Date as a String	
dateToString(Date date)	Converts a date to the String.	
dateToString(Date date, dateFormat)	Converts a date to the String according dateFormat.	
dayDiff(Date d1, Date d2)	Returns the difference in days between endDate and startDate.	
dayOfMonth(Date d)	Returns the day of month.	
dayOfWeek(Date d)	Returns the day of week.	
dayOfYear(Date d)	Returns the day of year.	
firstDateOfQuarter(int absQuarter)	Returns the first date of quarter.	
hour(Date d)	Returns the hour.	

Date / Time Functions		
Function	**Description**	**Comment**
hourOfDay(Date d)	Return the hour of day.	
lastDateOfQuarter(int absQuarter)	Returns the last date of the quarter.	
lastDayOfMonth(Date d)	Returns the last date of the month.	
minute(Date d)	Returns the minute.	
month(Date d)	Returns the month (0 to 11) of an input date.	
monthDiff(Date d1, Date d2)	Return the difference in months before d1 and d2.	
quarter(Date d)	Returns the quarter (0 to 3) of an input date.	
second(Date d)	Returns the second of an input date.	
weekDiff(Date d1, Date d2)	Returns the difference in weeks between endDate and startDate.	
weekOfMonth(Date d)	Returns the week of the month within which that date is.	
weekOfYear(Date d)	Returns the week of the year on which that date falls.	
yearDiff(Date d1, Date d2)	Returns the difference in years between endDate and startDate.	
year(Date d)	Returns the year (0 to 59) for an input Date.	

String Functions

String Functions		
Function	**Description**	**Comment**
contains(String str, char searchChar)	Checks if String contains a search character, handling null.	
contains(String str, String searchStr)	Checks if String contains a search String, handling null.	
containsAny(String str, char[] chars)	Checks if the String contains any character in the given set of characters.	
containsAny(String str, String searchChars)	Checks if the String contains any character in the given set of characters.	
endsWith(String str, String suffix)	Check if a String ends with a specified suffix.	
isEmpty(String str)	Checks if a String is empty ("") or null.	
lowerCase(String str)	Converts a String to lower case.	
removeEnd(String str, String remove)	Removes a substring only if it is at the end of a source string, otherwise returns the source string.	
removeStart(String str, String remove)	Removes a substring only if it is at the beginning of a source string, otherwise returns the source string.	

String Functions		
Function	**Description**	**Comment**
replace(String str, String searchString, String replacement)	Replaces all occurrences of a String within another String.	
replace(String str, String searchString, String replacement, int max)	Replaces a String with another String inside a larger String, for the first max values of the search String.	
startsWith(String str, String prefix)	Check if a String starts with a specified prefix.	
substring(String str, int beginIndex)	Gets a substring from the specified String.	A negative start position can be used to start n characters from the end of the String.
substring(String str, int beginIndex, int endIndex)	Gets a substring from the specified String.	A negative start position can be used to start/end n characters from the end of the String.
upperCase(String str)	Converts a String to upper case.	

Error Handling Functions

Error Handling Functions		
Function	**Description**	**Comment**
error(String msg)	Shows the error message.	

Index