



# Reference Guide

OpenL Tablets BRMS  
Release 5.20



Document number: TP\_OpenL\_RG\_2.0\_LSh

Revised: 11-07-2018



*OpenL Tablets Documentation is licensed under a [Creative Commons Attribution 3.0 United States License](https://creativecommons.org/licenses/by/3.0/).*

# Table of Contents

---

<b>1</b>	<b>Preface.....</b>	<b>6</b>
1.1	Audience.....	6
1.2	Related Information .....	6
1.3	Typographic Conventions.....	6
<b>2</b>	<b>Introducing OpenL Tablets .....</b>	<b>8</b>
2.1	What Is OpenL Tablets?.....	8
2.2	Basic Concepts.....	8
	Rules.....	9
	Tables.....	9
	Projects .....	9
2.3	System Overview.....	9
2.4	Installing OpenL Tablets .....	10
2.5	Tutorials and Examples.....	10
	Tutorials .....	10
	Examples .....	12
<b>3</b>	<b>Creating Tables for OpenL Tablets .....</b>	<b>13</b>
3.1	Table Recognition Algorithm .....	13
3.2	Table Properties .....	14
	Category and Module Level Properties.....	15
	Default Value.....	15
	System Properties .....	15
	Properties for a Particular Table Type .....	15
	Table Versioning.....	16
	Info Properties .....	28
	Dev Properties.....	28
	Properties Defined in the File Name .....	34
	Property State with the Countrywide Value Defined in the File Name .....	38
3.3	Table Types.....	40
	Decision Table .....	40
	Datatype Table.....	63
	Data Table .....	67
	Test Table.....	72
	Run Table .....	76
	Method Table.....	77
	Configuration Table.....	77
	Properties Table .....	80
	Spreadsheet Table.....	81
	Column Match Table .....	87
	TBasic Table.....	90
	Table Part .....	91
<b>4</b>	<b>OpenL Tablets Functions and Supported Data Types.....</b>	<b>94</b>
4.1	Working with Arrays.....	94
	Working with Arrays from Rules .....	94
	Array Index Operators.....	95

Functions to Work with Arrays .....	98
Rules Applied to Array.....	99
4.2 Working with Data Types .....	100
Simple Data Types.....	100
Value Data Types.....	101
Range Data Types.....	102
4.3 Working with Functions.....	104
Understanding OpenL Tablets Function Syntax .....	104
Math Functions .....	105
Date Functions .....	109
Special Functions and Operators .....	111
Null Elements Usage in Calculations .....	112
<b>5 OpenL Tablets Business Expression Language .....</b>	<b>114</b>
5.1 Java Business Object Model as a Basis for OpenL Tablets Business Vocabulary.....	114
5.2 New Keywords and Avoiding Possible Naming Conflicts .....	114
5.3 Simplifying Expressions with Explanatory Variables .....	115
5.4 Simplifying Expressions by Using the Unique in Scope Concept.....	115
5.5 OpenL Tablets Programming Language Framework .....	116
OpenL Tablets Grammars.....	116
Context, Variables and Types.....	117
OpenL Tablets Type System .....	118
OpenL Tablets as OpenL Tablets Type Extension .....	118
Operators .....	118
Binary Operators Semantic Map .....	119
Unary Operators .....	119
Cast Operators .....	119
Strict Equality and Relation Operators.....	119
List of org.openl.j Operators .....	120
List of org.openl.j Operator Properties .....	121
<b>6 Working with Projects .....</b>	<b>123</b>
6.1 Project Structure .....	123
Multi Module Project .....	123
Creating a Project.....	123
Project Sources .....	124
6.2 Rules Runtime Context.....	124
Managing Rules Runtime Context from Rules .....	125
6.3 Project and Module Dependencies .....	127
Dependencies Description .....	127
Dependencies Configuration.....	129
Import Configuration .....	130
Components Behavior.....	131
<b>7 Appendix A: BEX Language Overview .....</b>	<b>132</b>
7.1 Introduction to BEX .....	132
7.2 Keywords .....	132
7.3 Simplifying Expressions .....	133
Notation of Explanatory Variables .....	133
Uniqueness of Scope .....	133
7.4 Operators Used in OpenL Tablets.....	133

<b>8</b>	<b>Appendix B: Functions Used in OpenL Tablets.....</b>	<b>137</b>
8.1	Math Functions.....	137
8.2	Array Functions.....	138
8.3	Date Functions.....	140
8.4	String Functions.....	140
8.5	Special Functions.....	142
<b>9</b>	<b>Index.....</b>	<b>143</b>

# 1 Preface

---

This preface is an introduction to the *OpenL Tablets Reference Guide*. The following topics are included in this preface:

- [Audience](#)
- [Related Information](#)
- [Typographic Conventions](#)

## 1.1 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 required to use this guide effectively. Basic knowledge of Java is required to follow the development related sections.

## 1.2 Related Information

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

Related information	
Title	Description
<a href="#">[OpenL Tablets WebStudio User Guide]</a>	Document describing OpenL Tablets WebStudio, a web application for managing OpenL Tablets projects through a web browser.
<a href="http://openl-tablets.org/">http://openl-tablets.org/</a>	OpenL Tablets open source project website.

## 1.3 Typographic Conventions

The following styles and conventions are used in this guide:

Typographic styles and conventions	
Convention	Description
<b>Bold</b>	<ul style="list-style-type: none"> <li>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.</li> <li>Represents keys, such as <b>F9</b> or <b>CTRL+A</b>.</li> <li>Represents a term the first time it is defined.</li> </ul>
<code>Courier</code>	Represents file and directory names, code, system messages, and command-line commands.
<b>Courier Bold</b>	Represents emphasized text in code.
Select <b>File &gt; Save As</b>	Represents a command to perform, such as opening the <b>File</b> menu and selecting <b>Save As</b> .
<i>Italic</i>	<ul style="list-style-type: none"> <li>Represents any information to be entered in a field.</li> <li>Represents documentation titles.</li> </ul>
< >	Represents placeholder values to be substituted with user specific values.
<a href="#">Hyperlink</a>	Represents a hyperlink. Clicking a hyperlink displays the information topic or external source.
<b>[name of guide]</b>	Reference to another guide that contains additional information on a specific feature.

## 2 Introducing OpenL Tablets

---

This chapter 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](#)

### 2.1 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`, and `.xlsm` file formats.

### 2.2 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 more 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, and other components. For more information on projects, see [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.

## 2.3 System Overview

The following diagram displays how OpenL Tablets is used by different types of users.

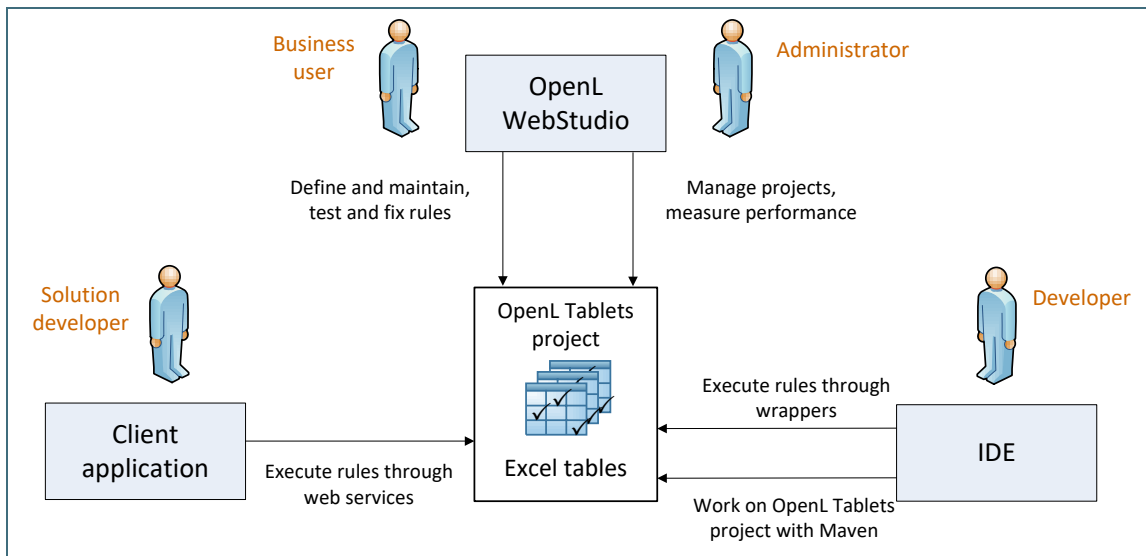


Figure 1: OpenL Tablets overview

A typical lifecycle of an OpenL Tablets project is as follows:

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

3. The business analyst 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.
4. 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.
5. Development team creates other parts of the solution and employs business rules directly through the OpenL Tablets engine or remotely through web services.
6. Whenever required, a business user updates or adds new rules to project tables.  
OpenL Tablets business rules management applications, such as OpenL Tablets WebStudio, Rules Repository, and OpenL Tablets Web Services, can be set up to provide self-service environment for business user changes.

## 2.4 Installing OpenL Tablets

OpenL Tablets installation instructions are provided in [[OpenL Tablets Installation Guide](#)].

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

## 2.5 Tutorials and Examples

OpenL Tablets provides a number of preconfigured projects developed for new users who want to learn working with OpenL Tablets 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. To open Repository Editor, in OpenL Tablets WebStudio, in the top line menu, click the **Repository** item.
2. Click the **Create Project** button .
3. In the **Create Project from** window, click the required tutorial name.
4. Click **Create** to complete.

The project appears in the **Projects** list of Repository Editor.

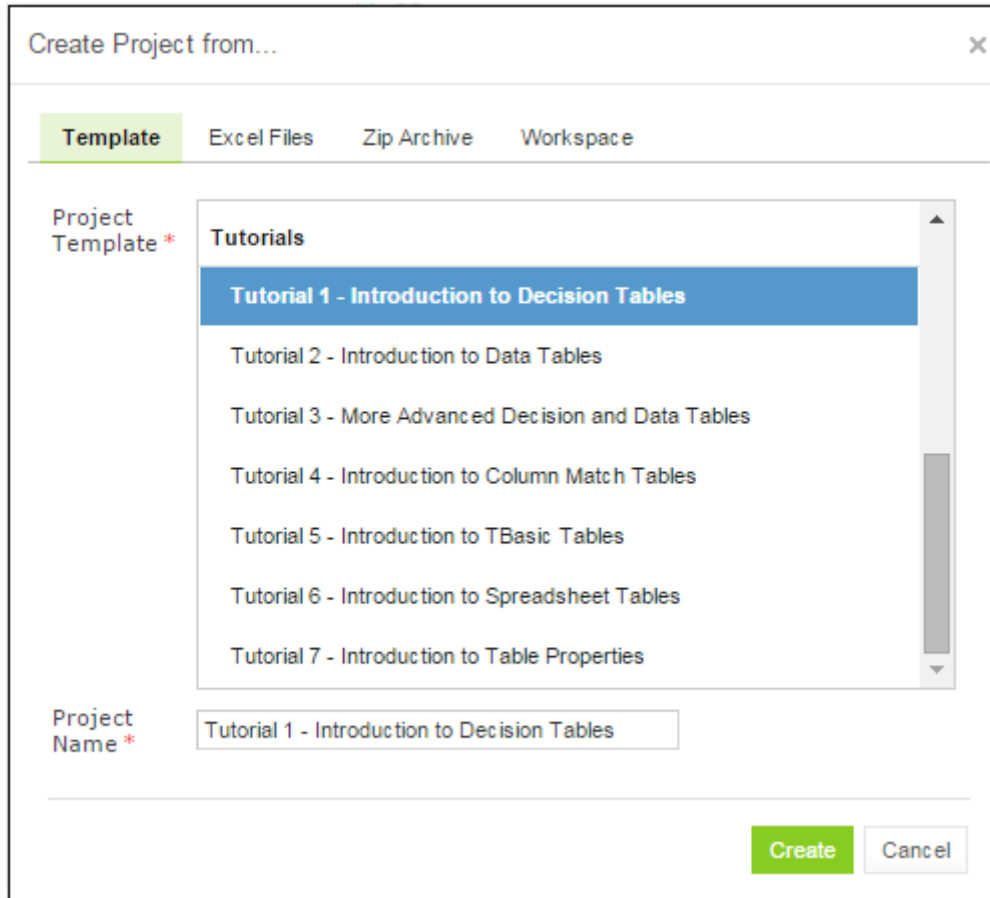


Figure 2: Creating tutorial projects

5. In the top line menu, click **Rules Editor**.  
The project is displayed in the **Projects** list and available for usage. It is highly recommended to start from reading Excel files for examples and tutorials which provide clear explanations for every step involved.

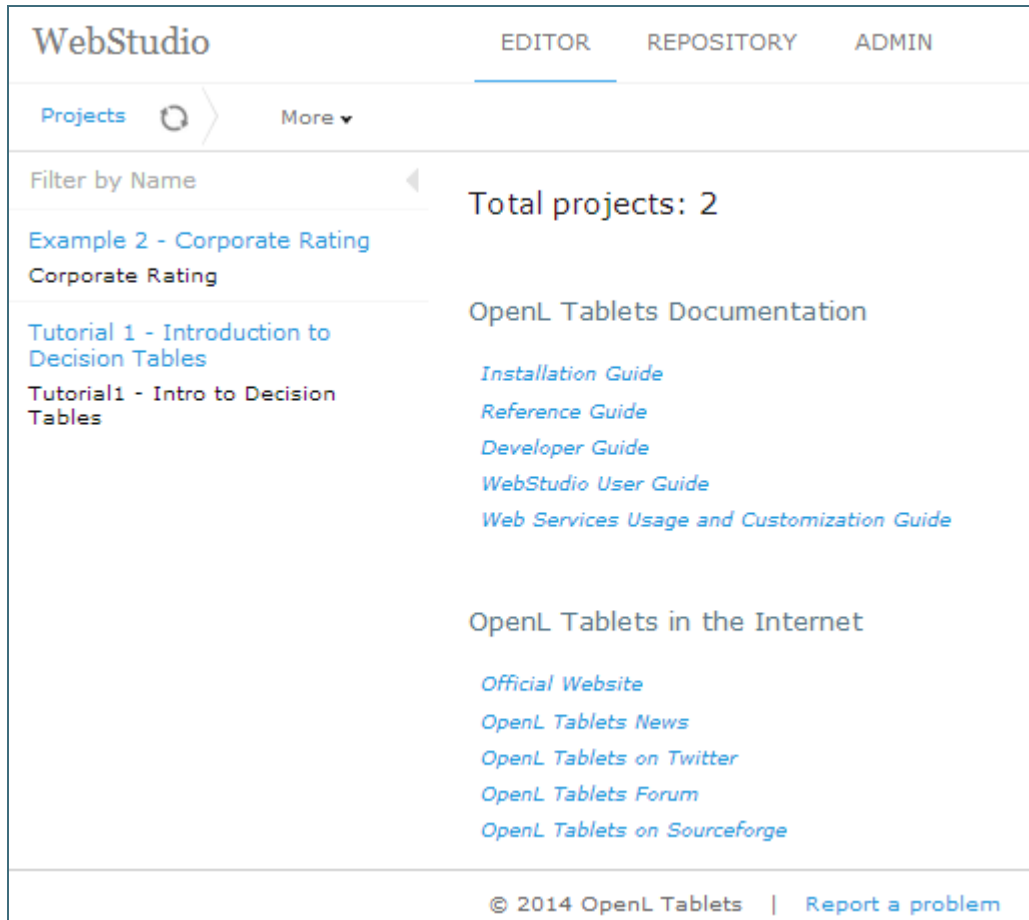


Figure 3: Tutorial project in the OpenL Tablets WebStudio

## 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 [Tutorials](#), and in the **Create Project from** dialog, select an example to explore. When completed, the example appears in the OpenL Tablets WebStudio Rules Editor as displayed in the Figure 3.

## 3 Creating Tables for OpenL Tablets

---

This chapter 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 chapter:

- [Table Recognition Algorithm](#)
- [Table Properties](#)
- [Table Types](#)

### 3.1 Table Recognition Algorithm

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

OpenL Tablets utilizes Excel concepts of workbooks and worksheets, which 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 its turn, is comprised of one or more tables. Workbooks can include tables of different types, each one supporting different underlying logic.

The general table recognition algorithm is as follows:

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, such table is recognized as an OpenL Tablets table.

The following are the supported keywords:

Table type keywords	
Keyword	Table type
Rules	<a href="#">Decision Table</a>
Data	<a href="#">Data Table</a>
Datatype	<a href="#">Datatype Table</a>
Test	<a href="#">Test Table</a>
Run	<a href="#">Run Table</a>
Method	<a href="#">Method Table</a>
Environment	<a href="#">Configuration Table</a>
Properties	<a href="#">Properties Table</a>
Spreadsheet	<a href="#">Spreadsheet Table</a>
ColumnMatch	<a href="#">Column Match Table</a>
TBasic or Algorithm	<a href="#">TBasic Table</a>

Table type keywords	
Keyword	Table type
SimpleRules	<a href="#">Simple Rules Table</a>
SimpleLookup	<a href="#">Simple Lookup Table</a>
TablePart	<a href="#">Table Part</a>

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.

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

It is a 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**.

**Note:** 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.

### 3.2 Table Properties

For all OpenL Tablets table types, except for [Properties Table](#), [Configuration Table](#) and the **Other** type tables, that is, non-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 the **properties** keyword 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 the 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, that is, second and third columns.

SimpleRules DriverType DriverAgeType (Gender gender, Integer age)		
	expirationDate	5/16/16
	effectiveDate	5/5/15
properties	category	Define age of Driver
<b>Gender</b>	<b>Age</b>	<b>Driver Status</b>
Male	<25	Young Driver

Figure 4: Table properties example

The following topics are included in this section:

- [Category and Module Level Properties](#)
- [Default Value](#)
- [System Properties](#)
- [Properties for a Particular Table Type](#)
- [Business Dimension Properties](#)
- [Table Versioning](#)
- [Info Properties](#)
- [Dev Properties](#)

- [Properties Defined in the File Name](#)
- [Property State with the Countrywide Value Defined in the File Name](#)

## Category and Module Level Properties

Table properties can be defined not only for each table separately, but for all tables in a specific category or a whole module. A separate [Properties Table](#) is designed to define this kind of properties. Only properties allowed to be inherited from the category or module level can be defined in this table. Some properties, such as description, can only be defined for a table.

Besides the **Properties** table, the module level properties can also be defined in a name of the Excel file corresponding to the module. For more information on defining properties in the Excel file name, see [Properties Defined in the File Name](#).

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

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

**Note:** The OpenL Tablets engine allows changing property values via the application code when loading rules.

## Default Value

Some properties can have default values. A **default value** is a predefined value that can be changed only in the 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 **Properties** pane on the right side of OpenL Tablets WebStudio Rules Editor.

## System Properties

System properties can only be set and updated by OpenL Tablets, not by users. OpenL Tablets WebStudio defines the following system properties:

- Created By
- Created On
- Modified By
- Modified On

For more information on system properties, see [\[OpenL Tablets WebStudio User Guide\]](#).

## Properties for a Particular Table Type

Some properties are used just for particular types of tables. It means that they make sense just for tables of a special type and can be defined only for those tables. 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](#), the **Auto Type Discovery** property used in [Spreadsheet Tables](#), and the **Precision** property designed for [Test Tables](#).

OpenL Tablets checks applicability of properties and produces an error if the 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:

Properties group list	
Group	Description
Business dimension	<a href="#">Business Dimension Properties</a>
Version	<a href="#">Table Versioning</a>
Info	<a href="#">Info Properties</a>
Dev	<a href="#">Dev Properties</a>

Properties of the **Business Dimension Properties** and **Table Versioning** groups are used for table versioning. They are described in detail further on in this guide.

## Table Versioning

In OpenL Tablets, business rules can be versioned in different ways using properties as described in [Table Properties](#). This section describes the most popular versioning properties:

Versioning properties	
Property	Description
<a href="#">Business Dimension Properties</a>	Targets advanced rules usage when several rule sets are used simultaneously. This versioning mechanism is more extendable and flexible.
<a href="#">Active Table</a>	Is more suitable for “what-if” analysis.

## Business Dimension Properties

This section introduces the **Business Dimension** group properties and includes the following topics:

- [Introducing Business Dimension Properties](#)
- [Using Effective and Expiration Date](#)
- [Using a Request Date](#)
- [Using an Origin Property](#)
- [Overlapping of Properties Values for Versioned Rule Tables](#)
- [Version Validation in Case of the One Rule Table](#)

### Introducing Business Dimension Properties

The properties of the **Business Dimension** group are used to version rules by *property values*. This type of versioning is typically used when there are rules with the same meaning applied under different conditions. In their projects, users can have as many rules with the same name as needed; the system selects and applies the required rule by its properties. For example, calculating employees’ salary for different years can vary by several coefficients, have slight changes in the formula, or both. In this case using the **Business Dimension** properties enables users to apply appropriate rule version and get proper results for every year.

The following table types support versioning by Business Dimension properties:

- Decision tables, including rules, simple rules, and simple lookup table types
- Spreadsheet



- TBasic
- Method
- ColumnMatch

When dealing with almost equal rules of the same structure but with slight differences, for example, with changes in any specific date or state, there is a very simple way to version rule tables by Business Dimension properties. Proceed as follows:

1. Take the original rule table and set Business Dimension properties that indicate by which property the rules must be versioned.  
Multiple Business Dimension properties can be set.
2. Copy the original rule table, set new dimension properties for this table, and make changes in the table data as appropriate.
3. Repeat steps 1 and 2 if more rule versions are required.

Now the rule can be called by its name from any place in the project or application. If there are multiple rules with the same name but different Business Dimension properties, OpenL Tablets reviews all rules and selects the corresponding one according to the specified property values or, in developers' language, by runtime context values.

The following table contains a list of **Business Dimension** properties used in OpenL Tablets:

Business Dimension properties list					
Property	Name to be used in rule tables	Name to be used in context	Level at which a property can be defined	Type	Description
Effective / Expiration dates	<ul style="list-style-type: none"> <li>• effectiveDate</li> <li>• expirationDate</li> </ul>	currentDate	Module Category Table	Date	Time interval within which a rule table is active. The table becomes active on the effective date and inactive after the expiration date. Multiple instances of the same table can exist in the same module with different effective and expiration date ranges.
Start / End Request dates	<ul style="list-style-type: none"> <li>• startRequestDate</li> <li>• endRequestDate</li> </ul>	requestDate	Module Category Table	Date	Time interval within which a rule table is introduced in the system and is available for usage.
LOB (Line of Business)	lob	lob	Module Category Table	String	LOB for a rule table, that is, business area for which the given rule works and must be used.
US Region	usregion	usRegion	Module Category Table	Enum[]	US regions for which the table works and must be used.
Countries	country	country	Module Category Table	Enum[]	Countries for which the table works and must be used.
Currency	currency	currency	Module Category Table	Enum[]	Currencies for which the table works and must be used.
Language	lang	lang	Module Category Table	Enum[]	Languages for which this table works and must be used.

Business Dimension properties list					
Property	Name to be used in rule tables	Name to be used in context	Level at which a property can be defined	Type	Description
US States	state	usState	Module Category Table	Enum[]	US states for which this table works and must be used.
Canada Province	caProvinces	caProvince	Module Category Table	Enum[]	Canada provinces of operation for which the table must be used.
Canada Region	caRegions	caRegion	Module Category Table	Enum[]	Canada regions of operation for which the table must be used.
Region	region	region	Module Category Table	Enum[]	Economic regions for which the table works and must be used.
Origin	origin		Module Category Table	Enum	Origin of rule to enable hierarchy of more generic and more specific rules.

**Note for experienced users:** A particular rule can be called directly regardless of its dimension properties and current runtime context in OpenL Tablets. This feature is supported by setting the ID property as described in [Dev Properties](#), in a specific rule, and using this ID as the name of the function to call. During runtime, direct rule is executed avoiding the mechanism of dispatching between overloaded rules.

Illustrative and very simple examples of how to use Business Dimension properties are provided further in the guide on the example of **Effective/Expiration Date** and **Request Date**.

### Using Effective and Expiration Date

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

Business Dimension properties for versioning on specific dates	
Property	Description
<b>Effective Date</b>	Date as of which a business rule comes into effect and produces required and expected results.
<b>Expiration Date</b>	Date after which the rule is no longer applicable. If <b>Expiration Date</b> is not defined, the rule works at any time on or after the effective date.

The date *for which* the rule is to be performed must fall into the effective and expiration date time interval.

Users can have multiple versions of the same rule table in the same module with different effective and expiration date ranges. However, these dates cannot overlap with each other, that is, if in one version of the rule effective and expiration dates are 1.2.2010 – 31.10.2010, do not create another version of that rule with effective and expiration dates within this dates frame if no other property is applied.

Consider a rule for calculating a car insurance premium quote. The rule is completely the same for different time periods except for a specific coefficient, a Quote Calculation Factor, or **Factor**. This factor is defined for each model of car.

The further examples display how these properties define which rule to apply for a particular date.

The following figure displays a business rule for calculating the quote for 2011. The effective date is 1/1/2011 and the expiration date is 12/31/2011.

SimpleRules Double Factor (String ModelOfCar)		
properties	effectiveDate	1/1/11
	expirationDate	12/31/11
<b>Model of Car</b>	<b>Factor for Quote Calculation</b>	
BMW	20	
Toyota	45	
Bentley	20	

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

However, the rule for calculating the quote for the year 2012 cannot be used because the factors for the cars differ from the previous year.

The rule names and their structure are the same but with the factor values differ. Therefore, it is a good idea to use versioning in the rules.

To create the rule for the year 2012, proceed as follows:

1. To copy the rule table, use the **Copy as New Business Dimension** feature in OpenL Tablets WebStudio as described in [[OpenL Tablets WebStudio User Guide](#)], *Copying Tables* section.
2. Change effective and expiration dates to 1/1/2012 and 12/31/2012 appropriately.
3. Replace the factors as appropriate for the year 2012.

The new table resembles the following:

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

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

To check how the rules work, test them for a certain car model and particular dates, for example, 5/10/2011 and 11/2/2012. The test result for BMW is as follows:

Test Factor FactorTest			
	_context_currentDate	ModelOfCar	_res_
	<b>Current Date</b>	<b>Model of Car</b>	<b>Factor</b>
1	5/10/11	BMW	20
2	11/2/12	BMW	25

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

In this example, the date on which calculation must be performed, per client’s request, is displayed in the **Current Date** column. In the first row for BMW, the current date value is 5/10/2011, and since  $5/10/2011 \geq 1/1/2011$  and  $10/5/2011 \leq 12/31/2011$ , the result factor for this date is **20**.

In the second row, the current date value is 2/11/2012, and since  $2/11/2012 \geq 1/1/2012$  and  $2/11/2012 \leq 12/31/2012$ , the factor is **25**.

**Using a Request Date**

In some cases, it is necessary to define additional time intervals for which user’s business rule is applicable. Table properties related to dates that can be used for selecting applicable rules have different meaning and work with slightly different logic compared to the previous ones.

Request properties used for versioning	
Property	Description
<b>Start Request Date</b>	Date when the rule is introduced in the system and is available for usage.
<b>End Request Date</b>	Date from which the system stops using the rule. If not defined, the rule can be used any time on or after the <b>Start Request Date</b> value.

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

**Note:** Pay attention to the difference between previous two properties: effective and expiration dates identify the date to which user’s rules are applied. In contrast, request dates identify when user’s rules are used, or called from the application.

Users can have multiple rules with different start and end request dates, where dates must intersect. In such cases, priority rules are applied as follows:

1. The system selects the rule with the latest **Start Request** date.

Figure 8: Example of the priority rule applied to rules with intersected Start Request date

2. If there are rules with the same **Start Request** date, OpenL Tablets selects the rule with the earliest **End Request** date.

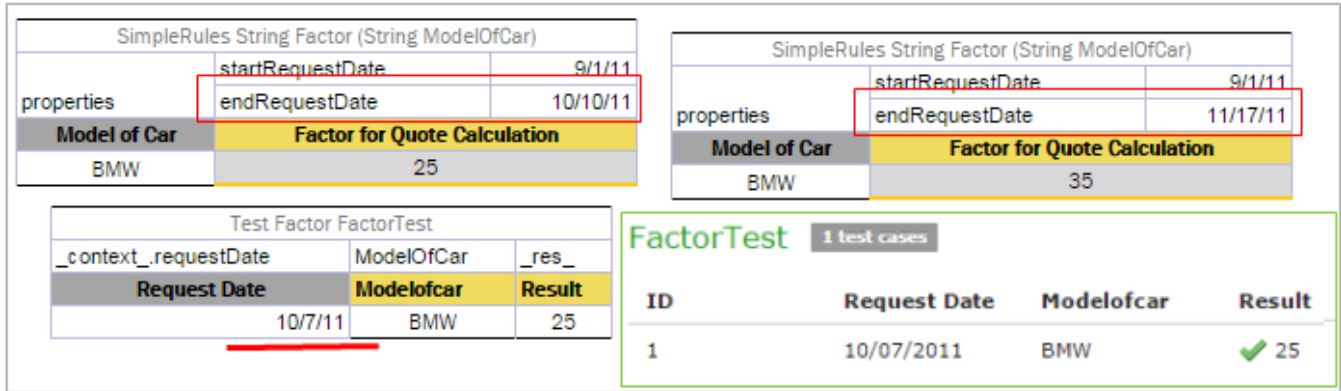


Figure 9: Example of the priority rule applied to the rules with End Request date

If the start and end request dates coincide completely, the system displays an error message saying that such table already exists.

**Note:** A rule table version with exactly the same **Start Request Date** or **End Request Date** cannot be created because it causes an error message.

**Note:** In particular cases, request date is used to define the date when the business rule was called for the very first time.

Consider the same rule for calculating a car insurance quote but add date properties, **Start Request Date** and **End Request Date**, in addition to the effective and expiration dates.

For some reason, the rule for the year 2012 must be entered into the system in advance, for example, from 12/1/2011. For that purpose, add 12/1/2011 as **Start Request Date** to the rule as displayed in the following figure. Adding this property tells OpenL Tablets that the rule is applicable from the specified **Start Request** date.

SimpleRules Double Factor (String ModelOfCar)		
	startRequestDate	12/1/11
	endRequestDate	5/1/12
properties	effectiveDate	1/1/12
	expirationDate	12/31/12
Model of Car	Factor for Quote Calculation	
BMW	25	
Toyota	45	
Bentley	20	

Figure 10: The rule for calculating the quote is introduced from 12/1/2011

Assume that a new rule with different factors from 2/3/2012 is introduced as displayed in the following figure.

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

Figure 11: The rule for calculating the Quote is introduced from 2.3.2011

However, the US legal regulations require that the same rules for premium calculations must be used; therefore users must stick to the previous rules for older policies. In this case, storing a request date in the application helps to solve this issue. By the provided request date, OpenL Tablets will be able to select rules available in the system on the designated date.

The following figure displays results of testing the rules for BMW for particular request dates and effective dates.

Test Factor FactorTest				
	_context_requestDate	_context_currentDate	ModelOfCar	_res_
	Request Date	Current Date	Model of Car	Factor
1	3/10/12	10/5/12	BMW	35
2	12/29/12	10/15/12	BMW	35
3	1/14/12	8/16/12	BMW	25

Figure 12: 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 the rule is run and calculation is performed are displayed in the **Request Date** column.

Pay attention to the row where **Request Date** is 3/10/2012. This date falls in the both start and end Request date intervals displayed in Figure 10 and Figure 11. However, the **Start Request** date in Figure 11 is later than the one defined in the rule in Figure 10. As a result, correct factor value is **35**.

### Using an Origin Property

The **Origin** Business Dimension property indicates the origin of rules used to generate a hierarchy of more generic and more specific rules. This property has two values, **Base** and **Deviation**. A rule with the **Deviation** property value has higher priority than a rule with the **Base** value or a rule without property value. A rule with the **Base** property value has higher priority than a rule without property value. As a result, selecting the correct version of the rule table does not require any specific value to be assigned in the runtime context, and the correct rule table is selected based on the hierarchy.

An example is as follows.

The screenshot displays the OpenL Tablets software interface. At the top, there is a menu bar with options: Edit, Open, Copy, Remove, Run, Trace, Test, and Create Test. Below the menu, there are two main tables and a test results summary.

**Table 1: Rules String Hello (Integer hour)**

properties	origin	Base	
Rule	C1	C2	RET1
	min <= hour	hour <= max	greeting
	Integer min	Integer max	String greeting
Rule	From	To	Greeting
R10	0	11	Good Morning
R20	12	17	Good Afternoon
R30	18	21	Good Evening
R40	22	23	Good Night

**Table 2: Rules Strino Hello (Integer hour)**

properties	origin	Deviation	
Rule	C1	C2	RET1
	min <= hour	hour <= max	greeting
	Integer min	Integer max	String greeting
Rule	From	To	Greeting
R10	0	11	Guten Morgen
R20	12	17	Guten Tag
R30	18	21	Guten Abend
R40	22	23	Gute Nacht

**Table 3: Test Hello HelloTest**

hour	_res_
Hour	Result
1 2	Guten Morgen
2 15	Guten Tag
3 22	Gute Nacht

**Table 4: HelloTest 3 test cases**

ID	Hour	Result
1	2	✓ Guten Morgen
2	15	✓ Guten Tag
3	22	✓ Gute Nacht

Figure 13: Example Rule table with origin property

### Overlapping of Properties Values for Versioned Rule Tables

By using different sets of Business Dimension properties, a user can flexibly apply versioning to rules, keeping all rules in the system. OpenL Tablets runs validation to check gaps and overlaps of properties values for versioned rules.

There are two types of overlaps by Business Dimension properties, “good” and “bad” overlaps. The following diagram illustrates overlap of properties, representing properties value sets of a versioned rule as circles. For simplicity, two sets are displayed.

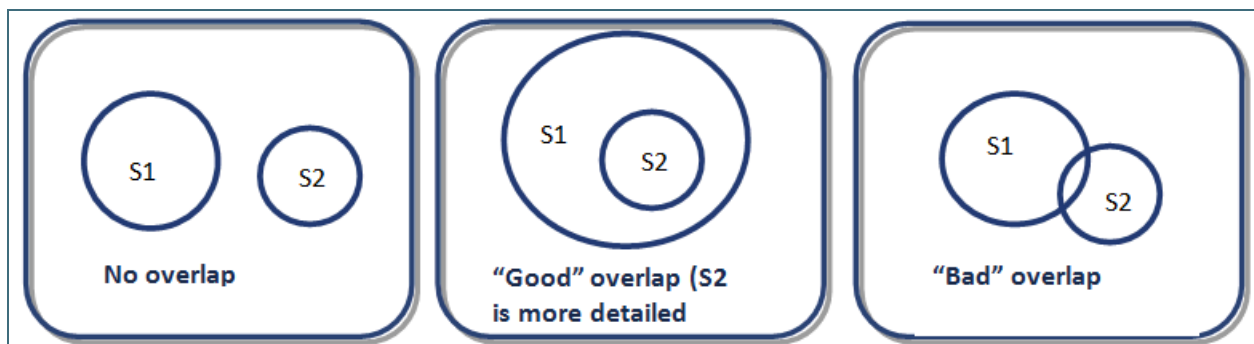


Figure 14: Example of logic for “good” and “bad” overlaps

The **No overlap** case means that property value sets are totally different and the only one rule table can be selected according to the specified client request in runtime context. An example is as follows:

SimpleRules DoubleValue AccidentPremium ()		
properties	state	CA
Per Accident Premium		
\$150		

SimpleRules DoubleValue AccidentPremium ()		
properties	state	NY
Per Accident Premium		
\$145		

Figure 15: Example of No overlap case

The **“Good” overlap** case describes the situation when several rule versions can be selected according to the client request as there are intersections among their sets, but one of the sets completely embeds another one. In this situation, the rule version with the most detailed properties set, that is, the set completely embedded in all other sets, is selected for execution.

**Note:** If a property value is not specified in the table, the property value is all possible values, that is, any value. It also covers the case when a property is defined but its value is not set, that is, the value field is left empty.

**Detailed properties values** mean that all these values are mentioned, or included, or implied in properties values of other tables. Consider the following example.

SimpleRules DoubleValue AccidentPremium ()				
Per Accident Premium				
\$135				
SimpleRules DoubleValue AccidentPremium ()				
properties	state	NY, CA, FL		
Per Accident Premium				
\$145				
SimpleRules DoubleValue AccidentPremium ()				
properties	state	CA		
Per Accident Premium				
\$150				

Test AccidentPremium AccidentPremiumTest		
_context_	_usState_	_res_
US State	Expected Accident Premium	
DE		\$135
NY		\$145
CA		\$150

Figure 16: Example of a rule with “good” overlapping

The first rule table is the most general rule: there are no specified states, so this rule is selected for any client request. It is the same as if the property state is defined with all states listed in the table. The second rule table has several states values set, that is, NY, CA, and FL. The last rule version has the most detailed properties set as it can be selected only if the rule is applied to the California state.

The following diagram illustrates example overlapping.



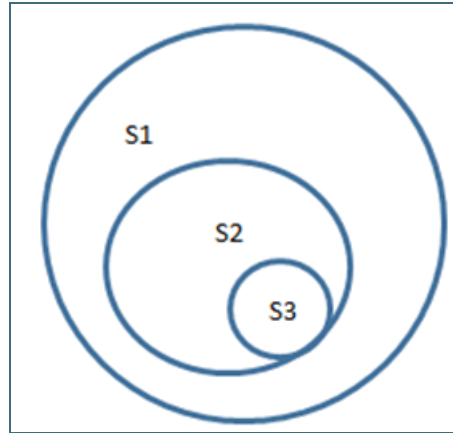


Figure 17: Logic of properties set inclusion

For the Delaware state, the only the first rule is applicable, that is, 135\$ Accident Premium. If the rule is applied to the New York state, then the first and second rule versions are suitable by property values, but according to the “good” overlapping logic, the premium is 145\$ because the second rule table is executed. And, finally, Accident Premium for the California state is 150\$ despite the fact that this property is set in all three rule tables: absence of property state in the first table means the full list of states set.

The “**Bad**” overlap is when there is no certain result variant. “Bad” overlap means that sets  $S_i$  and  $S_j$  have intersections but are not embedded. When a “bad” overlap occurs, the system displays the ambiguous error message.

Consider the following example.

SimpleRules DoubleValue AccidentPremium ()		
properties	state	NY, CA
Per Accident Premium		
\$145		
SimpleRules DoubleValue AccidentPremium ()		
properties	state	FL, CA
Per Accident Premium		
\$150		

Figure 18: Example of a rule with “bad” overlapping

For the California state, there are two possible versions of the rule, and “good” overlapping logic is not applicable. Upon running this test case, an error on ambiguous method dispatch is returned.

**Note:** For the matter of simplicity, only one property, **state**, is defined in examples of this section. A rule table can have any number of properties specified which are analyzed on overlapping.

**Note:** Only properties specified in runtime context are analyzed during execution.

### Version Validation in Case of the One Rule Table

Consider a rule table for which some business dimension properties are set up. There is only one version of this rule table. The following table describes options of versioning functionality behavior for this case depending on the **dispatching.validation** property value located in `webstudio\WEB-INF\conf\`:

Value of <b>dispatching.validation</b> property	
Value	Versioning behavior description
True	Versioning functionality works as for a rule that has only one version. OpenL Tablets reviews properties values of this rule table and executes the rule if the specified properties values match runtime context. Otherwise, the <b>No matching methods for context</b> error message is returned.
False	OpenL Tablets ignores properties of this rule table, and this rule is always executed and returns the result value despite of runtime context.

By default, the **dispatching.validation** value is set to **false** in OpenL Tablets Web Services and to **true** in OpenL Tablets WebStudio.

An example is as follows. Consider a Decision table **Hello** overloaded with the **lob** property.

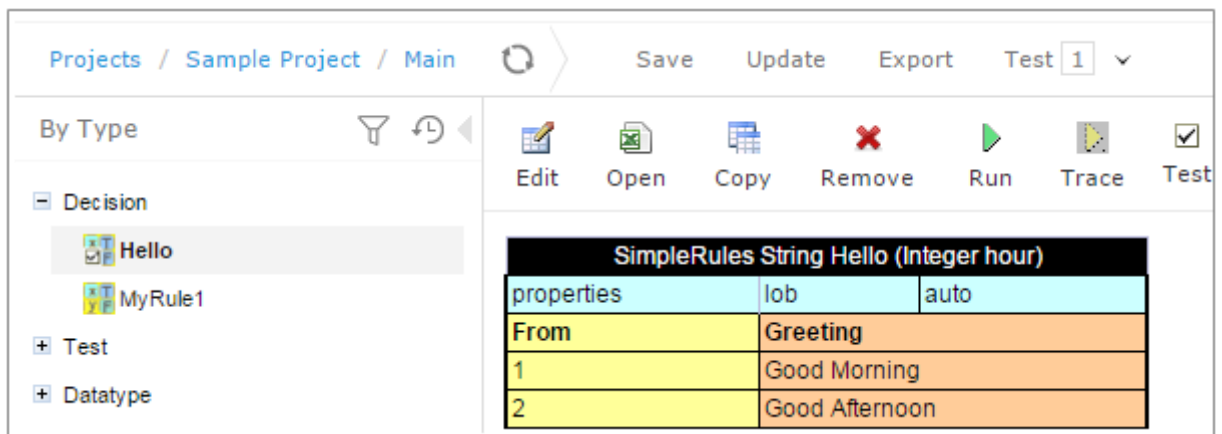


Figure 19: Example single overloaded rule

Create a test table with set context. Define a value of context which does not match the property’s value. The following examples illustrate different system behavior.

An example of `dispatching.validation = false` is as follows.

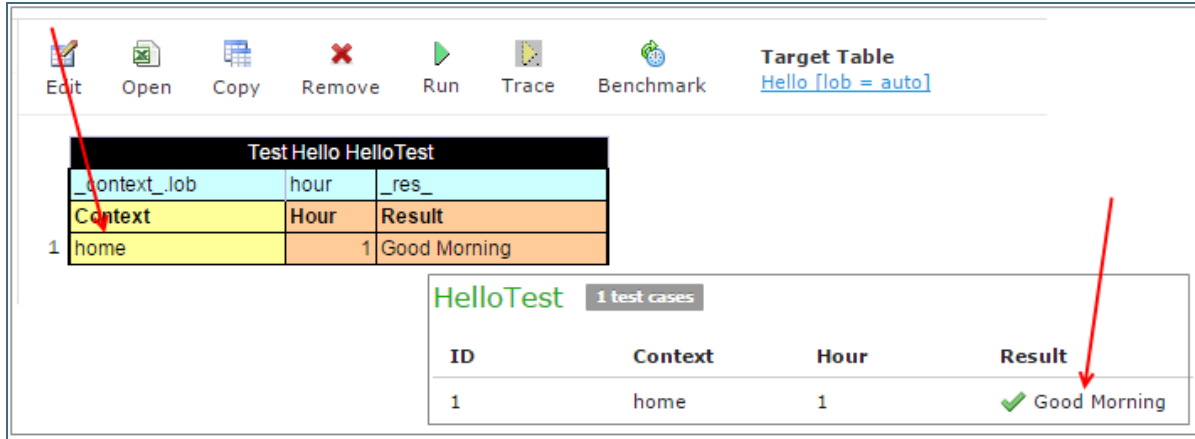


Figure 20: Example `dispatching.validation = false` mode

An example of `dispatching.validation = true` is as follows.

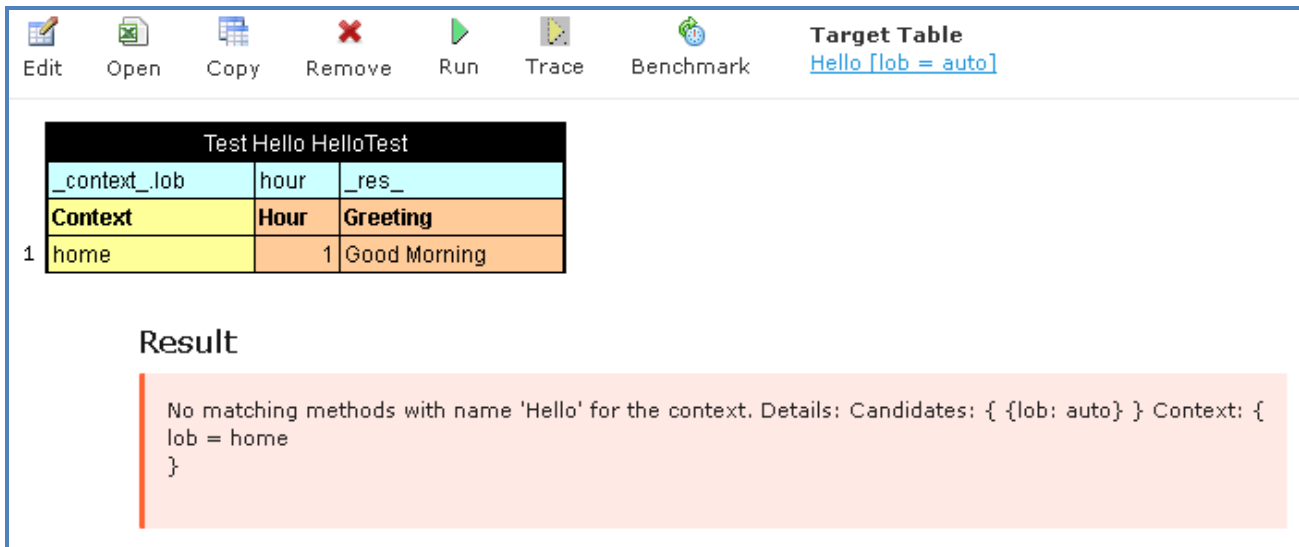


Figure 21: Example `dispatching.validation = true` mode

### Active Table

Table versioning allows storing the previous versions of the same rule table in the same rules file. The active table versioning mechanism is based on two properties, **version** and **active**. The **version** property must be different for each table, and only one of them can have **true** as a value for the **active** property.

All table versions must have the same identity, that is, exactly the same signature and dimensional properties values. Table types also must be the same.

An example of an inactive table version is as follows.

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

Figure 22: An inactive table version

### Info Properties

The **Info** group includes properties that provide useful information. This group enables users to easily read and understand rule tables.

The following table provides a list of **Info** properties along with their brief description:

Info properties list				
Property	Name to be used in rule tables	Level at which property can be defined and overridden	Type	Description
Category	category	Category, Table	String	Category of the table. By default, it is equal to the name of the Excel sheet where the table is located. If the property level is specified as <b>Table</b> , it defines a category for the current table. It must be specified if scope is defined as <b>Category</b> in the <b>Properties</b> table.
Description	description	Table	String	Description of a table that clarifies use of the table. An example is <i>Car price for a particular Location/Model</i> .
Tags	tags	Table	String[]	Tag that can be used for search. The value can consist of any number of comma-separated tags.
Created By	createdBy	Table	String	Name of a user who created the table in OpenL Tablets WebStudio.
Created On	createdOn	Table	Date	Date of table creation in OpenL Tablets WebStudio.
Modified By	modifiedBy	Table	String	Name of a user who last modified the table in OpenL Tablets WebStudio.
Modified On	modifiedOn	Table	Date	Date of the last table modification in OpenL Tablets WebStudio.

### Dev Properties

The **Dev** properties group impacts the OpenL Tablets features and enables system behavior management depending on a property value.

For example, the **Scope** property defines whether properties are applicable to a particular category of rules or for the module. If **Scope** is defined as **Module**, the properties are applied for all tables in the current module. If **Scope** is defined as **Category**, use the **Category** property to specify the exact category to which the property is applicable.

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

Figure 23: The properties are defined for the 'Police-Scoring' category

The following topics are included in this section:

- [Dev Properties List](#)
- [Variation Related Properties](#)
- [Using the Precision Property in Testing](#)

### Dev Properties List

The **Dev** group properties are listed in the following table:

Dev group properties															
Property	Name to be used in rule tables	Type	Table type	Level at which property can be defined	Description										
ID	id	Table	All	Table	Unique ID to be used for calling a particular table in a set of overloaded tables without using business dimension properties. <b>Note:</b> Constraints for the ID value are the same as for any OpenL function.										
Build Phase	buildPhase	String	All	Module, Category, Table	Property used to manage dependencies between build phases. <b>Note:</b> Reserved for future use.										
Validate DT	validateDT	String	Decision Table	Module, Category, Table	Validation mode for decision tables. In the wrong case an appropriate warning is issued. Possible values are as follows: <table border="1" data-bbox="1024 1434 1482 1684"> <thead> <tr> <th>Value</th> <th>Description</th> </tr> </thead> <tbody> <tr> <td>on</td> <td>Checks for uncovered or overlapped cases.</td> </tr> <tr> <td>off</td> <td>Validation is turned off.</td> </tr> <tr> <td>gap</td> <td>Checks for uncovered cases.</td> </tr> <tr> <td>overlap</td> <td>Checks for overlapped cases.</td> </tr> </tbody> </table>	Value	Description	on	Checks for uncovered or overlapped cases.	off	Validation is turned off.	gap	Checks for uncovered cases.	overlap	Checks for overlapped cases.
Value	Description														
on	Checks for uncovered or overlapped cases.														
off	Validation is turned off.														
gap	Checks for uncovered cases.														
overlap	Checks for overlapped cases.														

Dev group properties					
Property	Name to be used in rule tables	Type	Table type	Level at which property can be defined	Description
Fail On Miss	failOnMiss	Boolean	Decision Table	Module, Category, Table	Rule behavior in case no rules were matched: <ul style="list-style-type: none"> <li>If the property is set to <code>TRUE</code>, an error occurs along with the corresponding explanation.</li> <li>If the property is set to <code>FALSE</code>, the table output is set to <code>NULL</code>.</li> </ul>
Scope	scope	String	Properties	Module, Category	Scope for the Properties table.
Datatype Package	datatypePackage	String	Data Type	Table	Name of the Java package for generating the data type.
Recalculate	recalculate	Enum		Module, Category, Table	Way of a table recalculation for a variation. Possible values are <b>Always</b> , <b>Never</b> , and <b>Analyze</b> .
Cacheable	cacheable	Boolean		Module, Category, Table	Identifier of whether to use cache while recalculating the table, depending on the rule input.
Precision	precision	Integer	Test Table	Module, Category, Table	Precision of comparing the returned results with the expected ones while launching test tables.
Auto Type Discovery	autoType	Boolean	Properties Spreadsheet	Module, Category, Table	Auto detection of data type for a value of the <b>Spreadsheet</b> cell with formula. The default value is <code>true</code> . If the value is <code>true</code> , the type can be left undefined.
Concurrent Execution	parallel	Boolean		Module, Category, Table	Controls whether to parallel the execution of a rule when the rule is called for an array instead of a single value as input parameter. Default is <code>false</code> .

## Variation Related Properties

This section describes *variations* and the properties required to work with them, namely *Recalculate* and *Cacheable*.

A **variation** means additional calculation of the same rule with a modification in its arguments. Variations are very useful when calculating 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 fields specifically modified by variation in those arguments.

The following **Dev** properties are used to manage rules recalculation for variations:

Dev properties									
Property	Description								
<b>Cacheable</b>	Switches on or off using cache while recalculating the table. It can be evaluated to <code>true</code> or <code>false</code> . If it is set to <code>true</code> , all calculation results of the rule are cached and can be used in other variations; otherwise calculation results are not cached.  It is recommended to set Cacheable to <code>true</code> if recalculating a rule with the same input parameters is suggested. In this case, OpenL does not recalculate the rule, instead, it retrieves the results from the cache.								
<b>Recalculate</b>	Explicitly defines the recalculation type of the table for a variation. It can take the following values:								
	<table border="1"> <thead> <tr> <th>Value</th> <th>Description</th> </tr> </thead> <tbody> <tr> <td>Always</td> <td>If the Recalculate property is set to <b>Always</b> for a rule, the rule is entirely recalculated for a variation. This value is useful for rule tables which are supposed to be recalculated.</td> </tr> <tr> <td>Never</td> <td>If the Recalculate property is set to <b>Never</b> for a rule, the system does not recalculate the rule for a variation. It can be set for rules which new results users are not interested in and which are not required for a variation.</td> </tr> <tr> <td>Analyze</td> <td>It must be used for the top level rule tables to ensure recalculation of the included rules with the <b>Always</b> value. The included table rules with the <b>Never</b> value are ignored.</td> </tr> </tbody> </table>	Value	Description	Always	If the Recalculate property is set to <b>Always</b> for a rule, the rule is entirely recalculated for a variation. This value is useful for rule tables which are supposed to be recalculated.	Never	If the Recalculate property is set to <b>Never</b> for a rule, the system does not recalculate the rule for a variation. It can be set for rules which new results users are not interested in and which are not required for a variation.	Analyze	It must be used for the top level rule tables to ensure recalculation of the included rules with the <b>Always</b> value. The included table rules with the <b>Never</b> value are ignored.
Value	Description								
Always	If the Recalculate property is set to <b>Always</b> for a rule, the rule is entirely recalculated for a variation. This value is useful for rule tables which are supposed to be recalculated.								
Never	If the Recalculate property is set to <b>Never</b> for a rule, the system does not recalculate the rule for a variation. It can be set for rules which new results users are not interested in and which are not required for a variation.								
Analyze	It must be used for the top level rule tables to ensure recalculation of the included rules with the <b>Always</b> value. The included table rules with the <b>Never</b> value are 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, consider the Spreadsheet rule **DwellPremiumCalculation**, as displayed in the following figure, which calculates a home insurance premium quote. The quote includes calculations of **Protection** and **Key** factors which values are dependent on **Coverage A** limit as defined in the **ProtectionFactor** and **KeyFactor** simple rules. The insurer requests to vary Coverage A limit of the quote to verify how limit variations impact the **Key** factor.

DwellPremiumCalculation is a top level rule and during recalculation of the rule, only some results are of interest. That is why recalculation type, or the **recalculate** property, must be defined as **Analyze** for this rule.

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

On the contrary, the **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 must be set up as **Never**. Moreover, other rules tables, such as the **BaseRate** rule, which are not required to be recalculated, must have the recalculation property set to **Never**.

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 )
<b>Base_Premium</b>	<b>= round(product ( \$Base_Rate;\$Key_Factor ))</b>

Figure 24: Spreadsheet table which contains Recalculate Property

SimpleLookup DoubleValue <b>ProtectionFactor</b> ( ProtectionClass		
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

Figure 25: Decision table with defined Recalculate Property

SimpleRules DoubleValue <b>KeyFactor</b> ( 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 26: Usage of Variation Recalculate Properties

Consider that the **Coverage A** limit of the quote is 90 and **Protection Class** is 9. A modified value of **Coverage A** limit for a variation is going to be 110. 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 27: Results of DwellPremiumCalculation with recalculation = Analyze

Note that the **Key** factor is recalculated, but the **Protection** factor remains the same and the initial value of **Protection Factor** parameter is used.

If the recalculation type of DwellPremiumCalculation is defined as **Always**, OpenL Tablets ignores and does not analyze recalculation types of nested rules and recalculates all cells as displayed in the following figure.



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 28: Results of DwellPremiumCalculation with recalculation = Always

### Using the Precision Property in Testing

This section describes how to use the precision property. The property must be used for testing purpose and is only applicable to the test tables.

There are cases when it is impossible or not needed to define the 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 can be written in a cell of a table.

The **Precision** property is used as a measure of accuracy of the expected value to the returned value to a certain precision. Assume the precision of the expected value A is N. The expected value A is true only if

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

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

Consider the following examples. A simple rule FinRatioWeight has two tests, FinRatioWeightTest1 and FinRatioWeightTest2:

DoubleValue <b>FinRatioWeight</b> (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 29: An example of Simple Rule

The first test table has the **Precision** property defined with value 5:

Test <b>FinRatioWeight</b> FinRatioWeightTest1		
properties	precision	5
financialRatio	_res_	
Financial Ratio	Financial Ratio Weight	
Cash Liquidity Ratio	0.11121358	
Quick Ratio	0.05410091	

Figure 30: An Example of Test table with Precision Dev property

FinRatioWeightTest1		2 test cases	1
Financial Ratio	Financial Ratio Weight		
Cash Liquidity Ratio	✓	0.111207645	
Quick Ratio	✗	0.054117651	Expected: 0.05410091

Figure 31: An example of Test with precision defined

When this test is launched, the first test case is passed because  $|0.11121358 - 0.111207645| = 0.5935 \times 10^{-5} < 0.00001$ ; but the second is failed because  $|0.05410091 - 0.054117651| = 1.6741 \times 10^{-5} > 0.00001$ .

OpenL Tablets allows specifying precision for a particular column which contains expected result values using the following syntax:

```

_res_ (N)
_res_.$<ColumnName>$<RowName> (N)
_res_.<attribute name> (N)
    
```

An example of the table using shortcut definition is as follows.

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

Figure 32: Example of using shortcut definition of Precision Property

FinRatioWeightTest2		2 test cases
Financial Ratio	Financial Ratio Weight	
Current Ratio	✓	0.420000001
Operating Profit Margin	✓	0.414674703

Figure 33: An example of Test with precision for the column defined

Precision property shortcut definition is required when results of the whole test are considered with one level of rounding, and some expected result columns are rounded to another number of figures to the right of a decimal point.

Precision defined for the column has higher priority than precision defined at the table level.

Precision can be zero or a negative value, Integer numbers only.

## Properties Defined in the File Name

**Module level properties**, or table properties applied to all tables of a module, can be defined in the module file name. The following conditions must be met for such properties definition:

- A file name pattern is configured directly in a rules project descriptor, in the `rules.xml` file, as the `properties-file-name-pattern` tag, or via OpenL Tablets WebStudio as **Properties pattern for a file name** in the **Project** page.
- The module file name matches the pattern.

The file name pattern can include the following:

- text symbols
- table property names enclosed in ‘%’ marks
- wildcards, or characters that may be substituted for any of a defined subset of all possible characters

For more information on wildcards that can be used in a pattern as regular expressions, see <http://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html>.

If a table property value is supposed to be a date, the **Date** format must be specified right after the property name and colon as follows:

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

For more information on date formats description and examples, see <http://docs.oracle.com/javase/7/docs/api/java/text/SimpleDateFormat.html>.

File name pattern definition can use wildcards. For example, the `AUTO-%effectiveDate:MMddyyyy%-.*` pattern is defined. Then for the `AUTO-01012013-01012013.xls` file name, the module property **Effective date = 01 Jan 2013** is retrieved and the last part of the file name with the date is ignored as `.*` stands for any symbols.

In the following example, the **Auto Rating** project is configured in the way so that a user can specify the **US State** and **Effective date** properties values using the module file name:

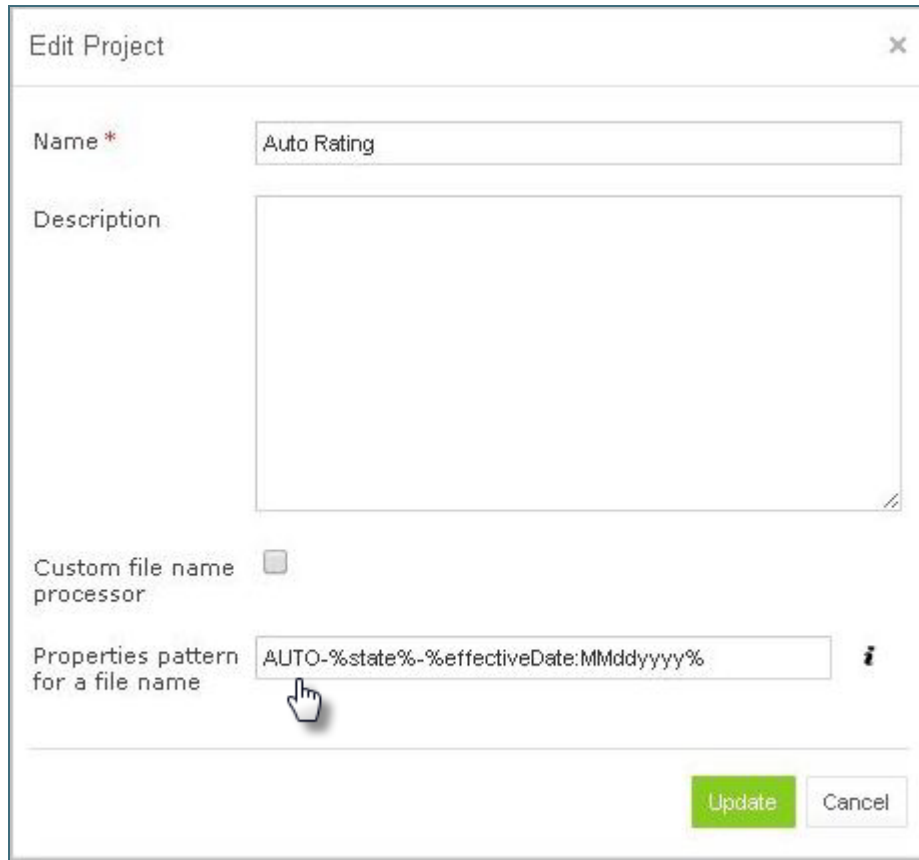


Figure 34: File name pattern configured via OpenL Tablets WebStudio

```
<properties-file-name-pattern>AUTO-%state%-%startRequestDate:MMddyyyy%</properties-file-name-pattern>
```

Figure 35: File name pattern in a rules project descriptor directly

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

If a file name does not match the pattern, module properties are not defined.

To view detailed information about the properties added to the file name pattern, click information icon next to the **Properties pattern for a file name** field.

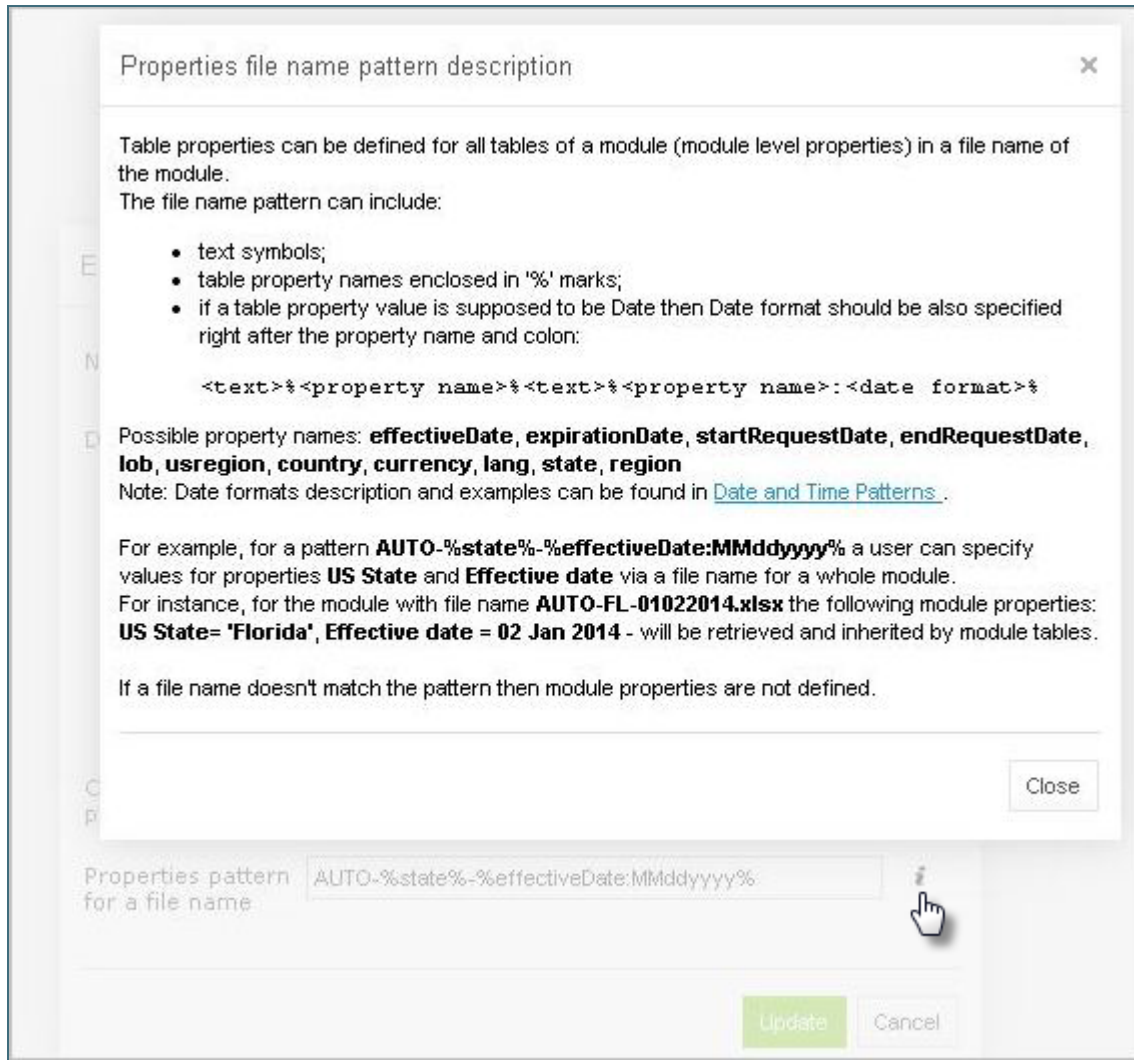


Figure 36: Properties file name pattern description

The same property cannot be defined both in a file name and **Properties** table of the module.

**Note for experienced users:** This section describes default implementation of properties definition in the file name. To use a custom implementation, specify the required file name processor class in a rules project descriptor. When the **Custom file name processor** check box is selected, the **File name processor class** field is displayed.

The screenshot shows the 'Edit Project' dialog box with the following fields and values:

- Name \***: Auto Rating
- Description**: (Empty text area)
- Custom file name processor**:
- File name processor class**: com.exigen.ipb.policy.preconfig.rating.auto.ModuleInitializingLister
- Properties pattern for a file name**: (Empty text field)

At the bottom right, there are two buttons: 'Update' (highlighted in green) and 'Cancel'.

Figure 37: Custom file name processor class

## Property State with the Countrywide Value Defined in the File Name

If the CW value, which stands for **country wide**, is set to the **Property State** in a file name, the rules of the corresponding module work for any state. Usually, only one value can be indicated in the file name, and listing all values in a filename is not available. This feature enables listing all values for property state in a file name by defining the **CW** value instead. It is useful when, for instance, there are particular files with rules for particular states, and a file with rules common for all states.

To enable the feature, the following conditions must be met:

- Define a file name processor class  
`org.openl.rules.project.resolving.CWPropertyFileNameProcessor.`  
 A file name processor class is configured directly in a rules project descriptor, in the `rules.xml` file, as the `properties-file-name-processor` tag, or via OpenL Tablets WebStudio as **File name processor class** in the **Project** page.
- Define the **Properties** pattern for a file name as described in [Properties Defined in the File Name](#).

The screenshot shows a dialog box titled "Edit Project" with a close button (X) in the top right corner. The dialog contains the following fields and controls:

- OpenL version compatibility:** 5.16+ (Latest version)
- Name \*:** Corporate Bank Calculation project
- Description:** A large empty text area.
- Custom file name processor:** A checkbox that is checked.
- File name processor class:** org.openl.rules.project.resolving.CWPropertyFileNameProcessor
- Properties pattern for a file name:** Corporate-%state%-%lob% (with an information icon 'i' to the right)

At the bottom right of the dialog, there are two buttons: "Update" (highlighted in green) and "Cancel".

Figure 38: Enabling CW value for state property from file name feature via OpenL Tablets WebStudio

For instance, consider the **Corporate Bank Calculation** project configured as displayed in the previous figure. The project module with the `CORPORATE-CW-TEST.xlsx` file name has the following property values:

- US State is any state
- lob = test

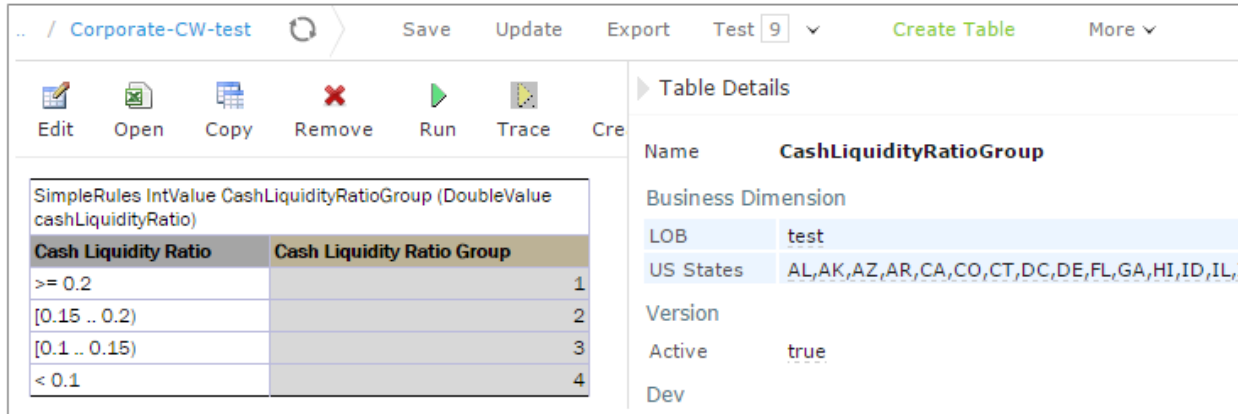


Figure 39: Decision table overloaded with all states

### 3.3 Table Types

OpenL Tablets supports the following table types:

- [Decision Table](#)
- [Datatype Table](#)
- [Data Table](#)
- [Test Table](#)
- [Run 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 execution of a set of actions and returned value. It is a basic table type used in OpenL Tablets decision making.

The following topics are included in this section:

- [Decision Table Structure](#)
- [Decision Table Interpretation](#)
- [Rules Tables](#)
- [Lookup Tables](#)
- [Smart and Simple Decision Tables](#)
- [Local Parameters in Decision Table](#)
- [Transposed Decision Tables](#)
- [Representing Arrays](#)
- [Representing Date Values](#)



- [Representing Boolean Values](#)
- [Representing Range Types](#)
- [Using Calculations in Table Cells](#)
- [Using Referents from Return Column Cells](#)

### Decision Table Structure

An example of a decision table is as follows:

Rules String Hello (Integer hour)		
C1	C2	RET1
min <= hour	hour <= max	greeting
Integer min	Integer max	String greeting
From	To	Greeting
0	11	Good Morning
12	17	Good Afternoon
18	21	Good Evening
22	23	Good Night

Figure 40: 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 table with names and types of the rule and its parameters used to access the decision table and provide input parameters.
2 and 3	No	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, including no rows at all.

Decision table structure																	
Row number	Mandatory	Description															
4	Yes	<p>Row consisting of the following cell types:</p> <table border="1"> <thead> <tr> <th>Type</th> <th>Description</th> <th>Examples</th> </tr> </thead> <tbody> <tr> <td>Condition column header</td> <td>Identifies that the column contains a rule condition and its parameters. It must start with the "C" character followed by a number, or be "MC1" for the 1<sup>st</sup> column with merged rows. If the condition has several parameters, the cell must be merged on all its parameter columns.</td> <td>C1, C5, C8 MC1</td> </tr> <tr> <td>Horizontal condition column header</td> <td>Identifies that the column contains a horizontal rule condition and its parameter (horizontal condition can have only one parameter). It must start with the "HC" character followed by a number.  Horizontal conditions are used in lookup tables only.</td> <td>HC1, HC5, HC8</td> </tr> <tr> <td>Action column header</td> <td>Identifies that the column contains rule actions. It must start with the "A" character followed by a number.</td> <td>A1, A2, A5</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 non-empty value is returned.</td> <td>RET1</td> </tr> </tbody> </table> <p>All other cells in this row are ignored and can be used as comments.</p> <p>If a table contains action columns, the engine executes actions for all rules with true conditions. If a table has a return column, the engine stops processing rules after the first executed rule with true conditions and non-empty result found</p>	Type	Description	Examples	Condition column header	Identifies that the column contains a rule condition and its parameters. It must start with the "C" character followed by a number, or be "MC1" for the 1 <sup>st</sup> column with merged rows. If the condition has several parameters, the cell must be merged on all its parameter columns.	C1, C5, C8 MC1	Horizontal condition column header	Identifies that the column contains a horizontal rule condition and its parameter (horizontal condition can have only one parameter). It must start with the "HC" character followed by a number.  Horizontal conditions are used in lookup tables only.	HC1, HC5, HC8	Action column header	Identifies that the column contains rule actions. It must start with the "A" character followed by a number.	A1, A2, A5	Return value column header	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 non-empty value is returned.	RET1
Type	Description	Examples															
Condition column header	Identifies that the column contains a rule condition and its parameters. It must start with the "C" character followed by a number, or be "MC1" for the 1 <sup>st</sup> column with merged rows. If the condition has several parameters, the cell must be merged on all its parameter columns.	C1, C5, C8 MC1															
Horizontal condition column header	Identifies that the column contains a horizontal rule condition and its parameter (horizontal condition can have only one parameter). It must start with the "HC" character followed by a number.  Horizontal conditions are used in lookup tables only.	HC1, HC5, HC8															
Action column header	Identifies that the column contains rule actions. It must start with the "A" character followed by a number.	A1, A2, A5															
Return value column header	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 non-empty value is returned.	RET1															

Decision table structure												
Row number	Mandatory	Description										
5	Yes	<p>Row containing cells with expression statements for condition, action, and return value column headers. OpenL Tablets supports Java grammar enhanced with OpenL Tablets Business Expression (BEX) grammar features. For more information on the BEX language, see <a href="#">Appendix A: BEX Language Overview</a>.</p> <p>In most cases, OpenL Tablets Business Expression grammar covers all the variety of expression statements and an OpenL user does not need to learn Java syntax. The code in these cells can use any objects and functions visible to the OpenL Tablets engine as elsewhere. For more information on enabling the OpenL Tablets engine to use custom Java packages, see <a href="#">Configuration Table</a>.</p> <p>Purpose of each cell in this row depends on the cell above is as follows:</p> <table border="1"> <thead> <tr> <th>Cell above</th> <th>Purpose</th> </tr> </thead> <tbody> <tr> <td>Condition column header</td> <td>Specifies the logical expression of the condition. It can reference parameters in the table 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 as <b>Condition column header</b>.</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 the 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 the keyword “return” is also supported. This cell can reference parameters in the rule header and parameters in the cells below.</td> </tr> </tbody> </table>	Cell above	Purpose	Condition column header	Specifies the logical expression of the condition. It can reference parameters in the table 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.	Horizontal condition	The same as <b>Condition column header</b> .	Action column header	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 the cells below.	Return value column header	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 the keyword “return” is also supported. This cell can reference parameters in the rule header and parameters in the cells below.
Cell above	Purpose											
Condition column header	Specifies the logical expression of the condition. It can reference parameters in the table 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.											
Horizontal condition	The same as <b>Condition column header</b> .											
Action column header	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 the cells below.											
Return value column header	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 the keyword “return” is also supported. This cell can reference parameters in the rule header and parameters in the cells below.											
6	Yes	<p>Row containing parameter definition cells. Each cell in this row specifies the type and name of parameters in the cells below it. Parameter name must be one word long. Parameter type must be one of the following:</p> <ul style="list-style-type: none"> <li>• simple data types</li> <li>• aggregated data types or Java classes visible to the engine</li> <li>• one-dimensional arrays of the above types as described in <a href="#">Representing Arrays</a></li> </ul>										
7	Yes	<p>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.</p>										
8 and below	Yes	<p>Concrete parameter values. Any cell can contain expression instead of concrete value and calculate the value. This expression can reference parameters in the rule header and any parameters of condition columns.</p>										

A user can merge cells of parameter values to substitute multiple single cells when the same value needs to be defined in these single cells. During rule execution, OpenL Tables unmerges these cells.

The additional **Rule** column with merged cells is used as the first column when the return value must be a list of values written in multiple rows of the same column, that is, a vertically arranged array. The Rule column determines the height of the result value list.

Rules DoubleValue[] <b>DriverPremiums</b> (DriverType driverType, MaritalS			
Rule	C1	C2	RET1
	driverType	maritalStatus	
	DriverType		
Rule	Driver Age	Marital Status	Driver Premiums
R1	Young Driver	Married	\$700
R2		Single	\$300
R3	Senior Driver		\$500
R4			\$200
			\$0

Figure 41: A table with the Rule column as the first column

Results of running <a href="#">DriverPremiums</a>			
ID	driverType	maritalStatus	Result
1	Young Driver	Married	<input type="checkbox"/> Collection of DoubleValue 700 720

Figure 42: Result in the vertically arranged array format

### 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.

Blank parameter value cell of the condition is interpreted as a true condition. If the condition column has several parameters, the condition with all its parameter cells blank is interpreted as a true condition.

Blank parameter value cell of the return/action column is ignored, the system does not calculate the return/action expression of the current rule and starts processing the next rule. If the return/action column has several parameters, all parameters cells need to be blank to ignore the rule.

If the empty return value is calculated by the expression, the system starts processing the next rule searching for a non-empty result.

The following example contains empty case interpretation. For **Senior Driver**, the marital status of the driver does not matter. Although there is no combination of **Senior Driver** and **Single** mode, the result value is 500 as for an empty marital status value.

SimpleRules DoubleValue DriverPremium (DriverType driverType, MaritalStatus maritalStatus)		
Driver Age	Marital Status	Driver Premium
Young Driver	Married	\$700
Young Driver	Single	\$720
Young Driver	Married	\$300
Young Driver	Single	\$300
Senior Driver		\$500
		\$0

Results of running <a href="#">DriverPremium</a>			
ID	driverType	maritalStatus	Result
1	Senior Driver	Single	<a href="#">500</a>

Figure 43: Empty case interpretation in the Decision table

### Rules Tables

A **rules table** is a regular decision table with vertical conditions only, that is, Cn and MC1 columns.

By default, each row of the decision table is a separate rule. Even if some cells of condition columns are merged, OpenL Tablets treats them as unmerged. This is the most common scenario.

The MC1 column plays the role of the Rule column in a table. It determines the height of the result value list. An example is as follows.

Rules Double[] DeductibleList(String coverageName, String brand)		
MC1	C2	RET1
coverageName	brand	
String	String	
Coverage Name	Brand Name	List of Deductibles
Flood Coverage	Brand X	200
		10000
Earthquake Coverage	Brand Y	500
Earthquake Coverage	Brand X	0
		100
		5000
Removal Coverage	Brand Z	100

Figure 44: A Decision table with merged condition values

Earthquake Coverage for Brand Y and Brand X has a different list of values, so they are not merged although their first condition is the same.

ID	coverageName	brand	Result
1	Removal Coverage	Brand Z	<input type="checkbox"/> Collection of Double 100 5000 100

Figure 45: A list of values as a result

## Lookup Tables

This section introduces lookup tables and includes the following topics:

- [Understanding Lookup Tables](#)
- [Lookup Tables Implementation Details](#)

### Understanding Lookup Tables

A **lookup table** is a special modification of the decision table which simultaneously contains vertical and horizontal conditions and returns value on crossroads of matching condition values.

That means condition values can 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 and so on. Every lookup matrix must start from the HC or RET column. The first HC or RET column must go after all vertical conditions, such as C, Rule, and comment columns. The RET section can be placed in any place of the lookup headers row. HC columns do not have the Titles section.

A lookup table must have the following components:

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

A lookup table can also have a rule column (or MC1 column, for more details refer to the [Rules Tables](#)).

A lookup table cannot have a comment column in the horizontal conditions part.

Rules DoubleValue <b>CarPrice</b> (Car car, Address billingAddress)				
C1	C2	HC1	HC2	RET1
country	region	brand	model	
Country	String	CarBrand	String	
<b>Country</b>	<b>Region</b>	<b>BMW</b>		
		<b>Z4 sDrive35i</b>	<b>Z4 sDrive30i</b>	
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 46: A lookup table example

The first cell of column titles must be merged on all rows that contain horizontal condition values (the height of the titles row is determined by the first cell in the row). See cell **Country** in the example above.

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

Rules DoubleValue <b>CarPrice</b> (Car car, Address billingAddress)				
C1	C2	C3	C4	RET1
country	region	brand	model	
Country	String	CarBrand	String	
<b>Country</b>	<b>Region</b>	<b>Brand</b>	<b>Model</b>	<b>Price</b>
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 47: Lookup table representation as a decision table

### Lookup Tables Implementation Details

This section describes 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 Tablets transforms a lookup table into a regular decision table internally and processes it as a regular decision table.

### Smart and Simple Decision Tables

Practice shows that most of decision tables have a simple structure: there are conditions for input parameters of a decision table that check equality of input and condition values, and a return value. Because of this, OpenL Tablets have simplified decision table representations. A simplified decision table allows skipping condition and return columns declarations, and thus the table consists of a header, column titles and condition and return values, and, optionally, properties. Restrictions for a simplified decision table are as follows:

- Condition values must be of the same type or be an array or range of the same type as corresponding input parameters.
- Return values must have the type of the return type from the decision table header.

The following topics are included in this section:

- [Smart Rules Table](#)
- [Smart Lookup Table](#)
- [Simple Rules Table](#)
- [Simple Lookup Table](#)
- [Ranges and Arrays in Simple Decision Tables](#)
- [Result of Custom Data Type in Smart and Simple Rules Tables](#)

#### Smart Rules Table

A decision table which has simple conditions for input parameters and a direct return (without expression) can be easily represented as a **smart rules table**.

The smart rules table header format is as follows:

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

SmartRules Double <b>DriverPremium</b> (DriverType driverType, MaritalStatus maritalStatus, Double factor)		
Type of Driver	Marital Status	Driver Premium
Young Driver	Married	\$700
	Single	\$720
Senior Driver		\$500
		= \$600 * factor

Figure 48: Smart rules table with simple return value

OpenL Tablets identifies which condition columns correspond to which input parameters by condition titles and parameter names. It finds an input parameter with the name closest to the column title, then the system checks condition and parameter values on equality or inclusion and returns the result from the return column(s) (the last column(s) which are identified as the result).



In the example above, the **driverType** value is compared with values from the **Type of Driver** column, the **maritalStatus** value is compared with the **Marital Status** column values, and the value from the **Driver Premium** column is returned as the result.

**Note:** To insure the system checks a condition with an appropriate input parameter, the user can "hover" with a mouse over the column title and see the hint with this information in WebStudio.

If a string value of the condition contains a comma, the value must be delimited with the backslash (\) separator followed by the comma. Otherwise, it is treated as an array of string elements as described in [Ranges and Arrays in the Simple Decision Tables](#):

Datatype <b>Industry</b> <String>	
trade	
precious metals, stones	
manufacturing	
services	
agriculture	
tourism	
other	

SmartRules DoubleValue <b>IndustryScore</b> (Industry industry)	
Industry	Industry Score
trade	5
precious metals\, stones	4
services, tourism	3
manufacturing	2
	1

Figure 49: Comma within a string value in a Smart table

### Smart Lookup Table

A lookup decision table with simple conditions that check equality or inclusion of an input parameter with a condition value and a direct return (without expression) can be easily represented as a **smart lookup table**. This table is similar to a smart rules table but has horizontal conditions.

The smart lookup table header format is as follows:

SmartLookup <Return type> RuleName(<Parameter type 1> parameterName1, (<Parameter type 2> parameterName2, ...)

SmartLookup Double <b>DriverPremium</b> (Double factor, DriverType driverType, MaritalStatus maritalStatus)		
Type of Driver	Married	Single
Young Driver	\$700	\$720
Senior Driver	\$500	\$500
	= \$600 * factor	

Figure 50: Smart lookup table example

The system identifies which vertical condition columns correspond to which input parameters by condition titles and parameter names in the same way as for smart rules tables. As for horizontal conditions, they are associated with the last input parameters in the same order they are listed. This means that the value of the last horizontal condition is compared with the value of the last input parameter, and so on.

The number of horizontal conditions is determined by the height of the first column title cell. This means that title cells of the vertical conditions must be merged on all rows which go for horizontal conditions.

The following is an example of a smart lookup table with several horizontal conditions:

SmartLookup Double <b>CarPrice</b> (String region, String country, String brand, String model)					
Country	Region	BMW		Porche	
		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		
USA	Mid Atlantic	\$52,450	\$46,550		
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

Figure 51: Smart lookup table with several horizontal conditions

### Simple Rules Table

A simplified decision table which has simple conditions for each parameter and a simple return can be easily represented as a **simple rules table**.

Unlike smart rules, a simple rule table uses all input parameters to associate them with condition columns in strict order, determined by simple logic, and using no titles. The value of the first column is compared with the value of the first input parameter, and so on. The value of the last column (return column) returns as a result. This means that input parameters must be in the same order as the corresponding condition columns, and the number of inputs must be equal to the number of conditions.

The simple rules table header format is as follows:

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

The following is an example of a simple rules table header:

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 52: Simple rules table example

**Note:** If a string value contains a comma, the value must be delimited with the backslash (\) separator followed by a comma. Otherwise, it is treated as an array of string elements as described in [Ranges and Arrays in Simple Decision Tables](#).

### Simple Lookup Table

A lookup decision table with simple conditions that check equality of an input parameter and a condition value and a simple return can be easily represented as **simple lookup table**. This table is similar to simple rules table but has horizontal conditions. The number of parameters to be associated with horizontal conditions is determined by the height of the first column title cell.

The simple lookup table header format is as follows:

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

The following is an example of a simple lookup table.

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

Figure 50: Simple lookup table example

### Ranges and Arrays in Smart and Simple Decision Tables

Range and array data types can be used in simplified rules tables and simplified lookup tables. If a condition is represented as an array or range, the rule is executed for any value from that array or range. As an example, in Figure 50 there is the same Car Price for all regions of Belarus and Great Britain, so, using an array, three rows for each of these countries can be replaced by a single one as displayed in the following table.

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

Figure 54: Simple lookup table with an array

If a string value contains a comma, the value must be delimited with the backslash (\) separator followed by a comma as illustrated for **Driver\, Passenger\, Side** in the following example. Otherwise, it is treated as an array of string elements.

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

Figure 55: Comma within a string value in a Simple Rule table

The following example explains how to use a range in a simple rules table.

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

Figure 56: Simple rules table with a Range

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

A range and array cannot be used in the same **Condition** column. Otherwise, OpenL issues an exception.

**Result of Custom Data Type in Smart and Simple Rules Tables**

A simplified rules table can return the value of compound type (custom data type) – the whole data object. To accomplish this, the user should make return column titles close to the corresponding fields of the object so the system can associate the data from the return columns with the returned object fields correctly.

In the example below, the rule **VehicleDiscount** determines the vehicles’s discount type and rate depending on air bags type and alarm indicator:

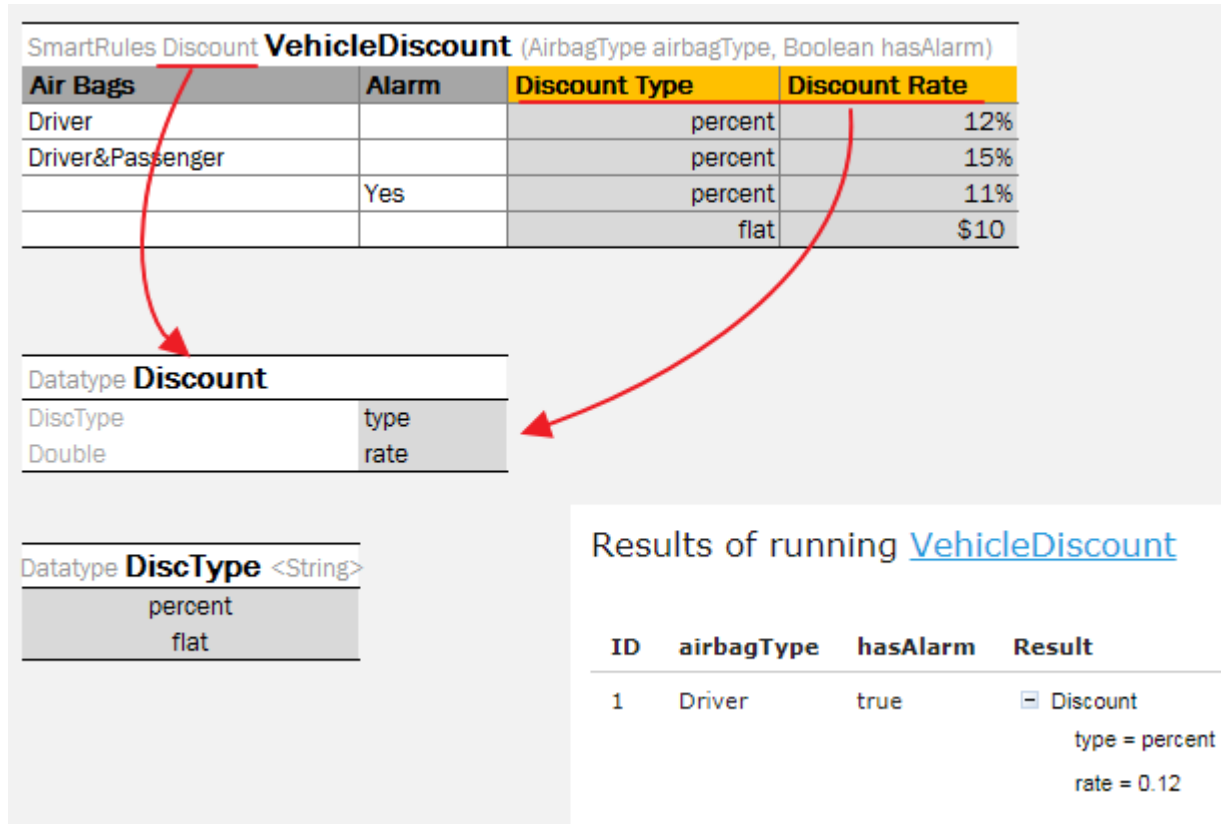


Figure 57: Smart rules table with complex return value

**Note:** To insure the system matches the return column with an appropriate return object field, the user can "hover" over the column title and see the hint with this information in WebStudio.

**Note:** A rule column in a smart rules table can be determined both as a condition column and as a return column at the same time if its name matches. This means that the fields of the return object can be populated with condition values as well.

If the rule returns the result of a very complex object (with nested objects inside), then there are several options for creating column titles:

- titles in one row with names that can be matched to the object fields unambiguously (the previously described approach) as shown in the example below, rule **VehicleDiscount1**;
- titles in several rows to define the hierarchy (structure) of the return object; in this case the user can merge cells associated with fields of a nested object as shown on the example below, rule **VehicleDiscount2**. Using this option, merging condition titles is required.

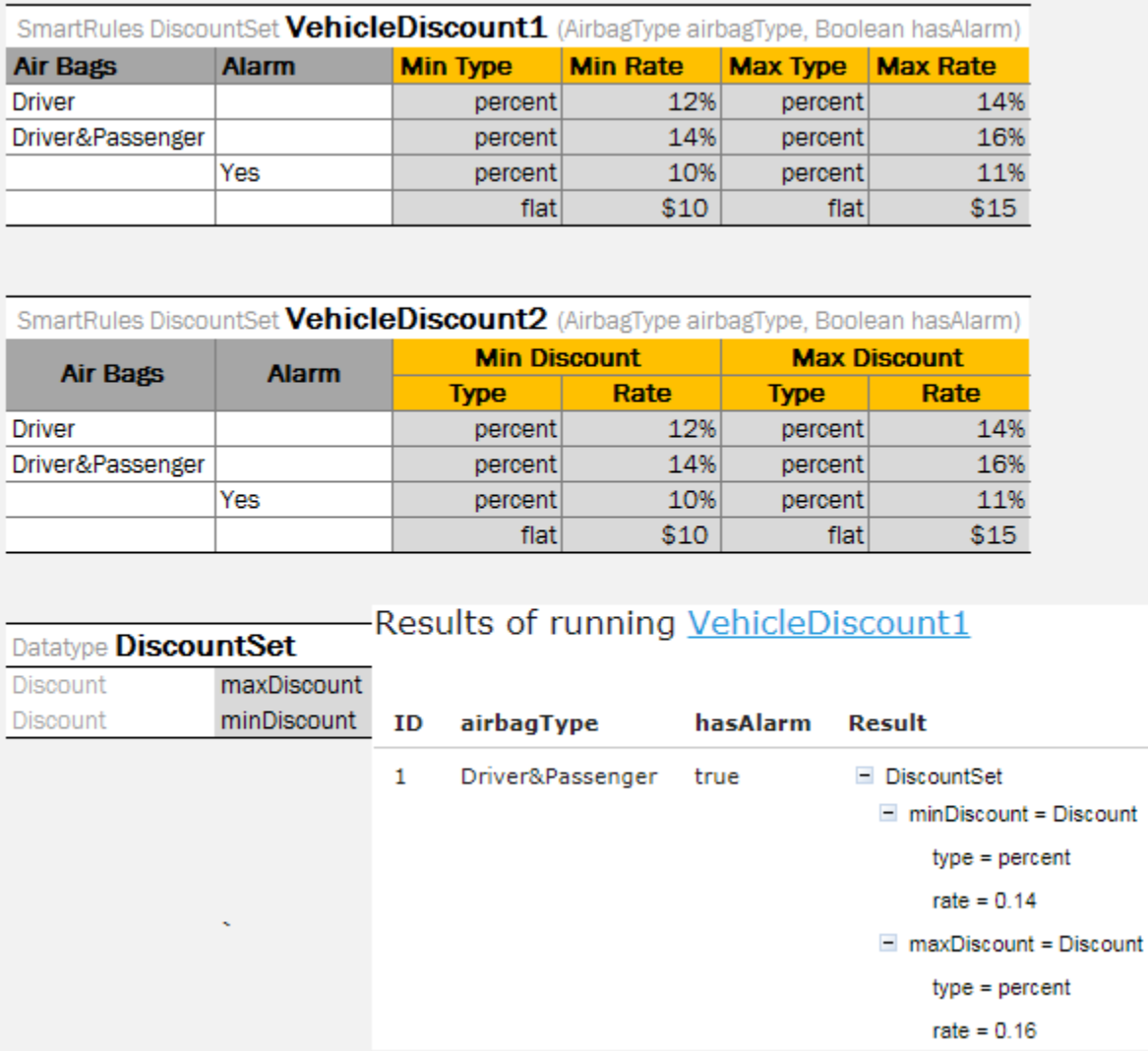


Figure 51: Smart rules tables with complex return value

### Collecting Results in Decision Table

A decision table returns only the first fired, non-empty result in common case. But there are business cases when all rules in a table must be checked and all results found returned. To do so, use:

- Collect keyword right before <Return type> in the table header for Smart and Simple rule table types;
- CRET as the return value column header for a regular decision table type;
- Define <Return type> as an array.

In the example below, rule **InterestTable** returns the list of interest schemes of a particular plan:

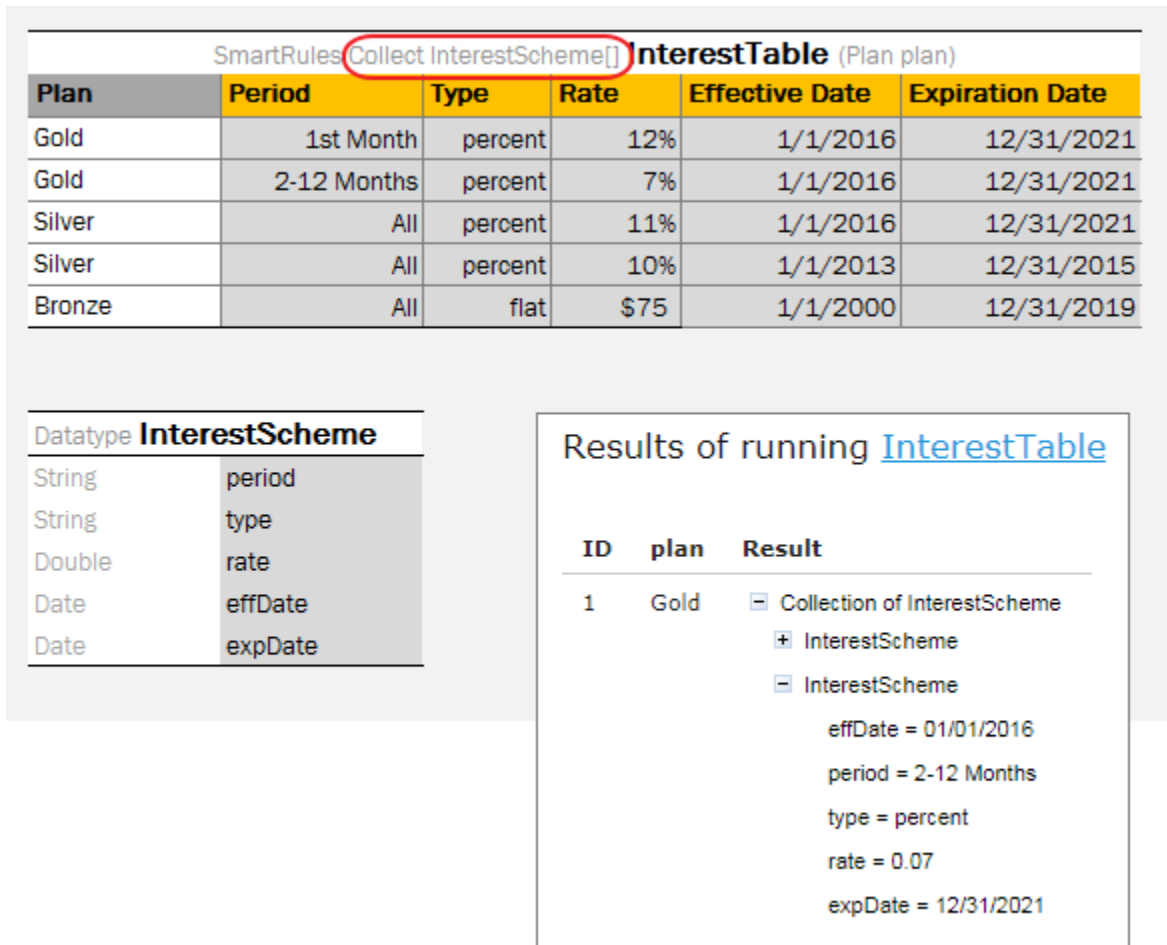
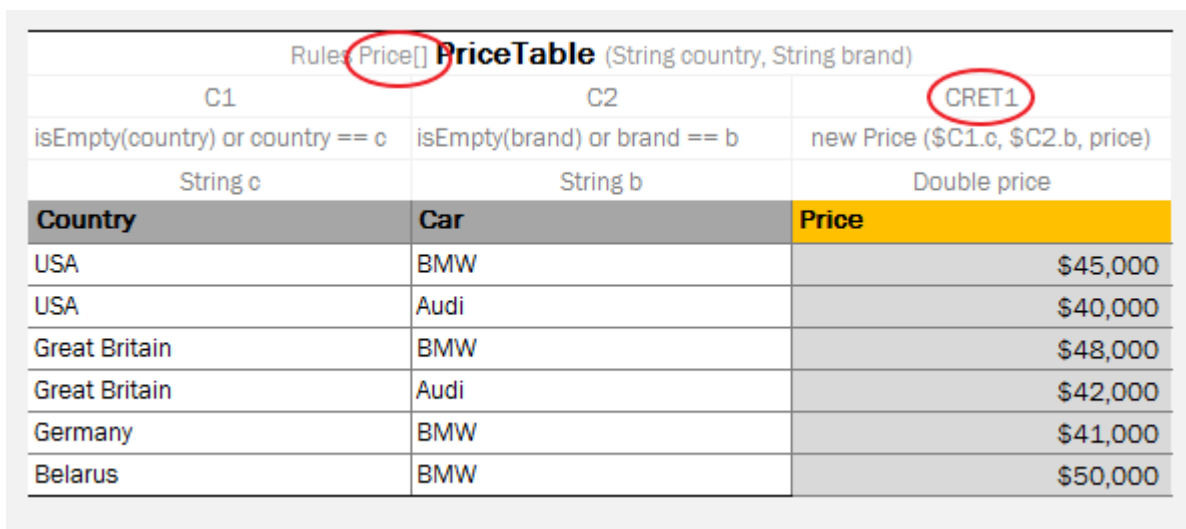


Figure 52: Collecting results in Smart and Simple rule table

In the following example, rule **PriceTable** collects car price information for desired specified country and/or "make" of a car:



## Results of running [PriceTable](#)

ID	country	brand	Result
1	Great Britain	Empty	<ul style="list-style-type: none"> <li>[-] Collection of Price                             <ul style="list-style-type: none"> <li>[-] Price                                     <ul style="list-style-type: none"> <li>country = Great Britain</li> <li>brand = BMW</li> <li>price = 48000</li> </ul> </li> <li>[-] Price                                     <ul style="list-style-type: none"> <li>country = Great Britain</li> <li>brand = Audi</li> <li>price = 42000</li> </ul> </li> </ul> </li> </ul>

Figure 60: Collecting results in regular Decision table

**Note for experienced users:** Smart and Simple rule tables can return the collection of List, Set, or Collection type. To define a type of a collection element, use the following syntax: `Collect as <Element type> <Collection type>` for example, `SmartRules Collect as String List Greeting (Integer hour)`.

### Local Parameters in Decision Table

When declaring a decision table, the header must contain the following information:

- column type
- code snippet
- declarations of parameters
- titles

Recent experience shows that in 95% of cases, users add very simple logic within code snippet, such as just access to a field from input parameters. In this case, parameter declaration for a column is useless and can be skipped.

The following topics are included in this section:

- [Simplified Declarations](#)
- [Performance Tips](#)

#### ***Simplified Declarations***

##### **Case#1**

The following image represents a situation when users must provide an expression and simple equal operation for condition declaration.



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

Figure 53: Decision Table requiring an expression and simple equal operation for condition declaration

This code snippet can be simplified as displayed in the following example.

Rules String test4(boolean hadTraining)	
C1	RET1
hadTraining	eligibility
	String eligibility
<b>Training</b>	<b>Eligibility</b>
No	Not Eligible
	Eligible

Figure 61: Simplified Decision Table

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

1. The parameter name will be **P1**, where 1 is index of the parameter.
2. The type of the parameter will be the same as the expression type.

In this example, it will be Boolean.

In the next step, OpenL Tablets will create an appropriate condition evaluator.

**Case#2**

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

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

Figure 62: Decision Table where user can omit name in declaration

As mentioned in the previous case, the OpenL Tablets engine generates the parameter name and users can use it in the expression, but in this case, users must provide a local parameter type because the expression type differs from the parameter type.

**Case#3**

The following example illustrates the **Greeting** rule with the **min <= value and value < max** condition expression.

Rules String <b>Greeting</b> (Integer hour)		
C1		RET1
min <= hour and hour < max		greeting + ", World!"
Integer min	Integer max	String greeting
From	To	Greeting
0	12	Good Morning
12	18	Good Afternoon
18	22	Good Evening
22	24	Good Night

Figure 63: The Greeting rule

Instead of the full expression **min <= value and value < max**, a user can simply use **value** and OpenL Tablets automatically recognizes the full condition.

Rules String <b>Greeting</b> (Integer hour)		
C1		RET1
hour		greeting + ", World!"
Integer min	Integer max	String greeting
From	To	Greeting
0	12	Good Morning
12	18	Good Afternoon
18	22	Good Evening
22	24	Good Night

Figure 64: Simplified Greeting rule

**Performance Tips**

OpenL Tablets enables users to create and maintain tests to insure reliable work of all rules. A 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.

To speed up rules execution, put simple conditions before more complicated ones.

In the following example, simple condition is located before a more complicated one.

Rules DoubleValue BankLimitIndex (Bank bank, String bank.RatingGroup)		
C1	C2	
bankRatingGroup String[]	(bankRatings[select first having ratingAgency == agency]=null) && (contains(ratingArray, bankRatings[select first having ratingAgency == agency].rating))	
	RatingAgency agency	String[] ratingArray
Bank Rating Group / Country, Financial Data	Agency	Rating of Agency
	Moody's Investors Service	Aaa, Aa1, Aa2, Aa3, A1, A2, A3, Baa1, Baa2, Baa3
	Fitch	AAA, AA+, AA, AA-, A+, A, A-, BBB-, BBB, BBB-
R1	Standard & Poor's	AAA, AA+, AA, AA-, A+, A, A-, BBB-, BBB, BBB-
	Moody's Investors Service	Ba1, Ba2, Ba3, B1, B2, B3
	Fitch	BB-, BB, BB-, B-, B, B-
R1, R2	Standard & Poor's	BB-, BB, BB-, B-, B, B-

Figure 65: Simple condition location


The main benefit of this approach is performance: expected results are found much faster.

Time for executing the OpenL Tablets rules heavily depends on complexity of condition expressions. To improve performance, use simple decision table types and simplified condition declarations.

### Transposed Decision Tables

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

Rules String Hello (Integer hour)			
Rule	C1	C2	RET1
	min <= hour	hour <= max	greeting
	Integer min	Integer max	String greeting
Rule	From	To	Greeting
R10	0	11	Good Morning
R20	12	17	Good Afternoon
R30	18	21	Good Evening
R40	22	23	Good Night



Rules String Hello (Integer hour)							
Rule			Rule	R10	R20	R30	R40
C1	min <= hour	Integer min	From	0	12	18	22
C2	hour <= max	Integer max	To	11	17	21	23
RET1	greeting	String greeting	Greeting	Good Morning	Good Afternoon	Good Evening	Good Night

Figure 66: Transposed decision table

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

### Representing Values of Different Types

The following sections describe how to present some values – list or range of numbers, dates, logical values – in OpenL table cells. The following topics are included in this section:

- [Representing Arrays](#)
- [Representing Date Values](#)
- [Representing Boolean Values](#)

- [Representing Range Types](#)

**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
- as 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 67: 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 68: 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 by 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		<b>Policy</b>	Policy4
drivers	>driverProfiles3	<b>Drivers</b>	test1 ,test3\,4 ,test2
vehicles	>autoProfiles3	<b>Vehicles</b>	1965 VW Bug
clientTier		<b>Client Tier</b>	Elite
clientTerm		<b>Client Term</b>	Long Term

Figure 69: 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
<b>Array1</b>	<b>Array2</b>	G
firstValue		
secondValue	value1, value2, value3	
value1		
value2		
value3	singleValue	

Figure 70: 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 must be used for the text:

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

The value must always be preceded with an apostrophe to indicate that 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 a 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.

### Representing Range Types

In OpenL, the following data types are designed to work with ranges:

- IntRange
- DoubleRange

For more information on these data types used for ranges, see [Range Data Types](#).

SimpleRules DriverType DriverAgeType (Gender gender, Integer age)		
Gender	Age	Driver Status
Male	<25	Young Driver
Female	<20	Young Driver
	71+	Senior Driver
		Standard Driver

Figure 71: Decision table with IntRange

**Note:** Be careful with using `Integer.MAX_VALUE` in a decision table. If there is a range with the border `max_number` equals to `Integer.MAX_VALUE`, for example, `[100; 2147483647]`, it is not included to the range. This is a known limitation.

### 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. The 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 a plain text. Alternatively, OpenL Tablets code can be enclosed by `{ }`.

The following decision table demonstrates calculations in table cells.

SimpleRules Integer AmPmTo24 (Integer ampmHr, String ampm)		
AM/PM hour	AM or PM	24 hour
12	AM	0
1-11	AM	=ampmHr
12	PM	12
1-11	PM	=ampmHr+12

Figure 72: Decision table with calculations

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

Calculations use regular Java syntax, similar to the one used in conditions and actions.

**Note:** Excel formulas are not supported by OpenL Tablets. They are used as pre-calculated values.

## Using Referents from Return Column Cells

When a condition value from a cell in the Return column must be called, specify the value by using `$(C<n> <variable name>` in the **Return** column.

Rules String <b>RiskOfWorkWithCorporate</b> (String riskOfProfile, String riskOfOperations, String riskOfGeography)			
C1	C2	C3	RET1
riskOfProfile	riskOfOperations	riskOfGeography	
String profile	String operations	String geography	String
<b>Risk of Prof</b>	<b>Risk of Operatio</b>	<b>Risk of Geography</b>	<b>Total Risk</b>
LOW	LOW	LOW	LOW
LOW	LOW	MIDDLE	=\$C1.profile
LOW	LOW	HIGH	LOW
LOW	MIDDLE	LOW	=\$C2.operations
LOW	MIDDLE	MIDDLE	LOW

Figure 54: A Decision table with referents inside the Return column

**Detailed trace tree**

- DT String = LOW RiskOfWorkWithCorporate(String
- Indexed condition: C1, Rules: [R1, R2, R3, R4]
  - Indexed condition: C2, Rules: [R1, R2, R3]
    - Indexed condition: C3, Rules: [R2]
      - Returned rule: R2

**Input parameters:** LOW LOW MIDDLE

**Returned result:** LOW

Rules String RiskOfWorkWithCorporate (String riskOfProfile, String riskOfOperations, String riskOfGeography)			
C1	C2	C3	RET1
riskOfProfile	riskOfOperations	riskOfGeography	
String profile	String operations	String geography	String
<b>Risk of Profile</b>	<b>Risk of Operations</b>	<b>Risk of Geography</b>	<b>Total Risk</b>
LOW	LOW	LOW	LOW
LOW	LOW	MIDDLE	=\$C1.profile
LOW	LOW	HIGH	LOW
LOW	MIDDLE	LOW	=\$C2.operations
LOW	MIDDLE	MIDDLE	LOW

Figure 73: Tracing Decision table with referents

## Datatype Table

This section describes datatype tables and includes the following topics:

- [Introducing Datatype Tables](#)
- [Inheritance in Data Types](#)
- [Alias Data Types](#)

### Introducing Datatype Tables

A **Datatype table** defines an OpenL Tablets data structure. A Datatype table is used for the following purposes:

- create a hierarchical data structure combining multiple data elements and their associated data types in hierarchy
- define the default values
- create vocabulary for data elements

A compound data type defined by Datatype table is called a **custom data type**. Datatype tables enable users to create their own data model which is logically suited for usage in a particular business domain.

For more information on creating vocabulary for data elements, see [Alias Data Types](#).

A Datatype table has the following structure:

1. The first row is the header containing the **Datatype** keyword followed by the name of the data type.
2. Every row, starting with the second one, represents one attribute of the data type.

The first column contains attribute types, and the second column contains corresponding attribute names.

3. The third column is optional and defines default values for fields.

Consider the case when a hierarchical logical data structure must be created. The following example of a Datatype table defines a custom data type called **Person**. The table represents a structure of the **Person** data object 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 74: Datatype table Person

Note that data attribute, or element, address of **Person** has, by-turn, custom data type **Address** and consists of zip code, city and street attributes.

Datatype Address	
String	zipCode
String	city
String	street

Figure 75: Datatype table Address

The following example extends the **Person** data type with default values for specific fields.

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

Figure 76: Datatype table with default values

The **Gender** field has 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](#).

The following is an example of a Datatype table defining a custom data type called **Person**. The table represents a structure of the **Person** data object and combines **Person's** data elements, such as name, social security number, date of birth, gender, and address. If necessary, default values can be defined in the Datatype table for the fields of complex type when combination of fields exists with default values.



Datatype Corporate		
String	corporateID	
String	corporateFullName	
Industry	industry	other
Ownership	ownership	private
Integer	numberOfEmployees	1
FinancialData	financialData	_DEFAULT_
QualityIndicators	qualityIndicators	_DEFAULT_

Figure 77: Datatype table containing value \_DEFAULT\_

FinancialData refers to the FinancialData data type for default values.

Datatype FinancialData		
Date	reportDate	01/01/2010
Double	cashAndEquivalent	0
Double	inventory	0
Double	currentAssets	0.0001
Double	currentLiabilities	0.0001
Double	equity	0
Double	revenue	0.0001
Double	operatingProfit	0
Double	monthlyCashTurnover	0
Double	monthlyAccountsTurnover	0

Figure 78: Datatype table with defined default values

During execution, the system takes default values from FinancialData data type.

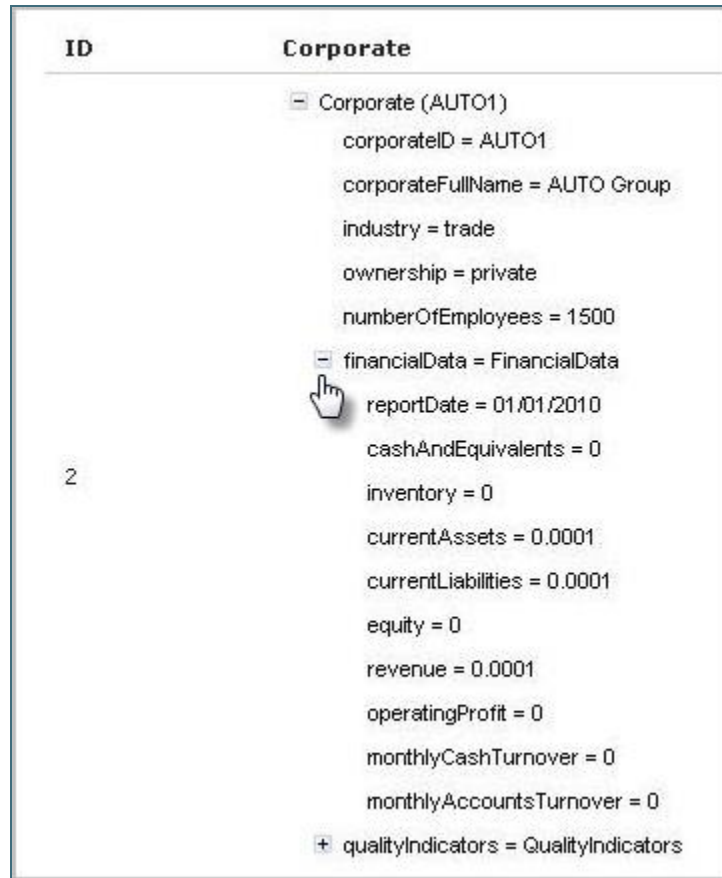


Figure 79: Datatype table with default values

**Note:** For array types `_DEFAULT_` creates an empty array.

**Note:** It is strongly recommended to leave an empty column right after the third column with default values if such column is used. Otherwise, in case the data type has 3 or less attributes, errors occur due to transposed tables support in OpenL Tablets.

			✗					✓	
Datatype <b>Driver</b>				Datatype <b>Driver</b>					
String	driverID		//unique ID	String	driverID		//unique ID		
DriverType	driverType	Principal		DriverType	driverType	Principal			
Integer	age			Integer	age				

Figure 80: Datatype table with comments nearby

### Inheritance in Data Types

In OpenL Tablets, one data type can be inherited from another one.

A new data type that inherits from another one contains all fields defined in the parent data type. If a child data type defines fields that are already defined in the parent data type, warnings or errors, if the same field is declared with different types in the child and the parent data type, are displayed.

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

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

## Alias Data Types

**Alias data types** are used to define a list of possible values for a particular data type, that is, to create a vocabulary for data.

The alias data type is created as follows:

1. The first row is the header.

It starts with the **Datatype** keyword, followed by the alias data type name. The predefined data type is in angle brackets on the basis of which the alias data type is created at the end.

2. The second and following rows list values of the alias data type.

The values can be of the indicated predefined data type only.

In the example described in [Introducing Datatype Tables](#), the data type **Person** has an attribute **gender** of the **Gender** data type which is the following alias data type.

Datatype Gender <String>
Male
Female

Figure 81: Example of Alias Datatype table with String parameters

Thus, data of Gender data type can only be **Male** or **Female**.

OpenL Tablets checks all data of the alias data type one whether its value is in the defined list of possible values. If the value is outside of the valid domain, or defined vocabulary, OpenL Tablets displays an appropriate error. Usage of alias data types provides data integrity and allows users to avoid accidental 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 reuses them in several test tables of a project by referencing the data table from test tables. As a result, different tests use the same data tables to define input parameter values, for example, to avoid duplicating data.

Data tables can contain data types supported by OpenL Tablets or types loaded in OpenL Tablets from other sources. For more 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 define a list of values of data types that have a simple structure.

1. The first row is the header of 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 predefined and alias data types, refer to [Working with Data Types](#) and [Datatype Table](#).

2. The second row is a keyword **this**.
3. The third row is a descriptive table name intended for business users.
4. In the fourth and following rows, values of data are provided.

An example of a data table containing an array of numbers is as follows.

<b>Data Integer numbers</b>
this
<b>Numbers</b>
10
20
30
40
50

Figure 82: Simple data table

### Using Advanced Data Tables

Advanced data tables are used for storing information of a complex structure, such as custom data types. For more 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 displays a datatype table and a corresponding data table with concrete values below it.

<b>Datatype Person</b>	
String	name
String	ssn
<b>Data Person p1</b>	
name	ssn
<b>Name</b>	<b>SSN</b>
Jonh	555-55-0001
Paul	555-55-0002
Peter	555-55-0003
Mary	555-55-0004

Figure 83: Datatype table and a corresponding data table

**Note:** There can be blank cells left in data rows of the table. In this case, OpenL Tablets considers such data as non-existent for the row and does not initialize any value for it, that is, there will be a **null** value for attributes or even **null** for the array of values if all corresponding cells for them are left blank.

There might be a situation when a user needs a Data table column with unique values, while other columns contain values that are not unique. In this case, add a column with the predefined `_PK_` attribute name, standing for the primary key.

Data Person <b>person2</b>				
<code>_PK_</code>	name	dob	gender	maritalStatus
<b>person ID</b>	<b>Name</b>	<b>DOB</b>	<b>Gender</b>	<b>Marital Status</b>
1	Jonh	1/1/1980	Male	Single
2	Peter	5/7/1981	Male	Single
3	Peter	10/20/1982	Male	Single
4	Mary	7/7/1987	Female	Married

Figure 84: A Data table with unique `_PK_` column

If the `_PK_` column is not defined, the first column of the table is used as a primary key.

The screenshot shows a data table 'Data Policy policyProfile4' with columns: name, drivers, vehicles, clientTier, and clientTerm. The 'drivers' column has a dropdown menu open, showing 'Policy4' and 'Drivers' with a 'D1' key. Below the dropdown is a 'Select All' button and a 'Done' button. A secondary table 'Data Driver driverProfiles3' is shown below, with columns: `_PK_`, name, gender, age, maritalStatus, and state. The `_PK_` column has two entries: 'D1' and 'D2'. The 'Name' column has 'Shane' and 'Tom', 'Gender' has 'Male' and 'Male', 'Age' has '21' and '26', 'Marital Status' has 'Single' and 'Married', and 'State' has 'CA' and 'CA'. Red arrows indicate the relationship between the 'D1' key in the dropdown and the 'D1' key in the secondary table.

Figure 85: Referring from one Data table to another using a primary key

### Specifying Data for Aggregated Objects

Assume that the data, which values are to be specified and stored in a data table, is an object of a complex structure with an attribute that is another complex object. The object that includes another object is called an **aggregated object**. To specify an attribute of an aggregated object in a 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 86: Complex data types defined by Datatype tables

In the data types structure, the `Address` data type contains a reference to the `ZipCode` data type as its attribute `zip`. An example of a data table that specifies values for both data types at the same time is as follows.

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 87: Specifying values for aggregated objects

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

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

If a data table must store information for an array of objects, OpenL Tablets allows defining attribute values for each element of an array. The following format must 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 following 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 88: 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**, contains the only vehicle **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 data type 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 to the input values or expected result values definition in the Test and Run tables.

### 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 **cities** data table contains values from the **states** table. To ensure that correct values are entered, a reference to the **code** column in the **states** table is defined.

Data City cities		Data SupportedState states	
city	state	name	code
	>states code	<b>State/Possession</b>	<b>Abbreviation</b>
<b>City</b>	<b>State</b>	ALABAMA	AL
Fairbanks	AK	ALASKA	AK
Beverly Hills	CA	AMERICAN	AS
		ARIZONA	AZ
		ARKANSAS	AR
		CALIFORNIA	CA
		COLORADO	CO
		CONNECTICUT	CT
		DELAWARE	DE

Figure 89: Defining a reference to another data table

If an invalid state abbreviation is entered in the **cities** table, OpenL Tablets reports an error.

The target column definition is not required if it is the first column or **\_PK\_** column in the referenced data table. For example, if a reference is made to the **name** column in the **states table**, the following simplified reference can be used:

```
>states
```

If a data table contains values defined as a part of another data table, the following format can be used:

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

The difference from the previous format is that an attribute name of the referenced data table, which corresponding values are included in the other data table, is specified additionally.

If **<column name>** is omitted, the reference by default is constructed using the first column or **\_PK\_** column of the referenced data table.

In the following diagram, the **claims** data table contains values defined in the **policies** table and related to the **vehicle** attribute. A reference to the **name** column of the **policies** table 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
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 90: Defining a reference to another data table

**Note:** To ensure that correct values are provided, cell data validation lists can be used in Excel, thus limiting the range of values that can be entered.

**Note:** The same syntax of data integration is applicable to the input values or expected result values definition in the Test and Run tables.

**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, and method tables. 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	RET1	
dayOfWeek(MyDate)		
IntRange		
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 91: Decision table and its unit test table

A test table has the following structure:

- 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.
- The second row provides a separate cell for each input parameter of the rule table followed by the **\_res\_** column, which typically contains the expected test result values.
- The third row contains display values intended for business users.



4. Starting with the fourth row, each row is an individual test case.

For more information on how to specify values of input parameters and expected test results of complex constructions, see [Specifying Data for Aggregated Objects](#) and [Ensuring Data Integrity](#).

**Note for experienced users:** Test tables can be used to execute any Java method. In this 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 runtime context values are defined in the runtime environment. Test tables for a table, overloaded by business dimension properties, must provide values for the runtime context significant for the tested table. Runtime 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 92: An example of a test table with a context value

For a full list of runtime context variables available, their description, and related Business Dimension versioning properties, see [Context Variables Available in Test Tables](#).

The **\_description\_** column can be used for entering useful information.

The **\_error\_** column of the test table can be used for a test algorithm where the **error** function is used. The OpenL Tablets Engine compares an error message to the value of the **\_error\_** column to decide if test is passed.

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

Figure 93: An example of a test table with an expected error column

If OpenL Tablets projects are accessed and modified through OpenL Tablets WebStudio, UI provides convenient utilities for running tests and viewing test results. For more information on using OpenL Tablets WebStudio, see [\[OpenL Tablets WebStudio User Guide\]](#).

The following topics are included in this section:

- [Context Variables Available in Test Tables](#)
- [Testing Spreadsheet Result](#)

### 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 variables of OpenL Tablets					
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	Date on which the rule is performed.
Request Date	requestDate	Date	Start / End Request dates	startRequestDate, endRequestDate	Date when the rule is applied.
Line of Business	lob	String	LOB (Line of Business)	lob	Line 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 in which the rule is applied.
Region	region	Enum	Region	region	Economic region where the rule is applied.
Canada Province	caProvince	Enum	Canada Province	caProvinces	Canada province of operation where the rule is applied.
Canada Region	caRegion	Enum	Canada Region	caRegions	Canada region of operation where the rule is applied.

For more information on how property values relate to runtime context values and what rule table is executed, see [Business Dimension Properties](#).

### Testing Spreadsheet Result

Cells of a spreadsheet result, which is returned by the rule table, can be tested as displayed in the following spreadsheet table.

Spreadsheet SpreadsheetResult IncomeForecast (Double bonusRate, Double sharePrice)				
Step	Year1	Year2	Year3	TotalAmount
Salary	45000	= round (\$Year1\$Salary * 1.10)	=round (\$Year1\$Salary * 1.20)	= sum (\$Year1:\$Year3)
Shares	0	0	1000	= sum (\$Year1:\$Year3)
Bonus	=\$Salary * bonusRate	= \$Salary * bonusRate	= \$Salary * bonusRate	= sum (\$Year1:\$Year3)
MinSalary	= \$Salary	= \$Salary	= \$Salary	= sum (\$Year1:\$Year3)
MaxSalary	= \$Salary + \$Bonus + \$Shares * sharePrice	= \$Salary + \$Bonus + \$Shares * sharePrice	= \$Salary + \$Bonus + \$Shares * sharePrice	= sum (\$Year1:\$Year3)

Figure 94: A sample spreadsheet table

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

Test IncomeForecast IncomeForecastTest			
bonusRate	sharePrice	_res_.\$TotalAmount\$MinSalary	_res_.\$TotalAmount\$MaxSalary
Bonus Rate	Share Price	Min Total Salary	Max Total Salary
15%	\$15	\$148,500	\$185,775
10%	\$25	\$148,500	\$188,350
5%	\$35	\$148,500	\$190,925

Figure 95: Test for the sample spreadsheet table

Columns marked with the green color determine income values, and columns marked with lilac determine the expected values for a specific number of cells. It is possible to test as many cells as needed.

The result of running this test in OpenL Tablets WebStudio is provided in the following output table.

Results of running [IncomeForecastTest](#)

**IncomeForecastTest** 3 test cases

ID	Bonus Rate	Share Price	Min Total Salary	Max Total Salary
1	0.15	15	✓ 148500	✓ 185775
2	0.1	25	✓ 148500	✓ 188350
3	0.05	35	✓ 148500	✓ 190925

Figure 96: The sample spreadsheet test results

It is 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 spreadsheets 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.

Consider an advanced example provided in the following figure. The **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 and premium, the

**VehicleCalculation** and **DriverCalculation** spreadsheet tables are invoked in cells of the **PolicyCalculation** rule table.

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

Figure 97: Example of the PolicyCalculation spreadsheet table

Spreadsheet SpreadsheetResult VehicleCalculation (Vehicle vehicle)	
	Value
Age : Integer	= <a href="#">CurrentYear</a> () - year
TheftRating : TheftRating	= <a href="#">VehicleTheftRating</a> (bodyType, price, onHighTheftProbabilityList)
InjuryRating : InjuryRating	= <a href="#">VehicleInjuryRating</a> (bodyType, airbagType, hasRollBar)

Figure 98: Example of the VehicleCalculation spreadsheet table

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

Figure 99: The advanced sample spreadsheet table

The structure of the resulting **PolicyCalculation** spreadsheet is rather complex. Any cell of the result can be tested as illustrated in the **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 100: 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.

An example of a run method table is as follows.

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

Figure 101: 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.

For more information on how to specify values of input parameters which have complex constructions, see [Specifying Data for Aggregated Objects](#) and [Ensuring Data Integrity](#).

## Method Table

A **method table** is a Java method described within a table. An example of a method table is as follows:

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

Figure 102: 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 are the actual code to be executed. They 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 experienced in programming in developing rules of any logic and complexity.

## Configuration Table

This section describes the structure of the **configuration** table and includes the following topics:

- [Configuration Table Description](#)
- [Defining Dependencies between Modules in the Configuration Table](#)

### Configuration Table Description

OpenL Tablets allows splitting business logic into multiple Excel files, or modules. There are cases when rule tables of one module need to call rule tables placed in another module. A **configuration table** is used to indicate module dependency.

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 a dependency module by its name. All data from this module becomes accessible in the current module. A dependency module can be located in the current project or its dependency projects.
import	Imports the specified Java package, class, or library 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. This command is deprecated.
language	Provides language import functionality.
extension	Expands OpenL Tablets capabilities with external set of rules. After adding, external rules are compiled with OpenL Tablets rules and work jointly.
vocabulary	Allows using user created dynamic classes in OpenL Tablets. This command is deprecated.

For more information on dependency and import configurations, see [Project and Module Dependencies](#).

## Defining Dependencies between Modules in the Configuration Table

Often several or even all modules in the project have the same symbols in the beginning of their name. In such case, there are several options how to list several dependency modules in the **Environment** table:

- adding each dependency module by its name
- adding a link to all dependency modules using the common part of their names and the asterisk \* symbol for the varying part
- adding a link to several dependency modules using the question mark ? symbol to replace one symbol anywhere in the name

All modules that have any letter or number at the position of the question mark symbol will be added as dependency.

The second option, that is, using the asterisk symbol after the common part of names, is considered a good practice because of the following reasons:

- Any new version of dependency module is not omitted in future and requires no changes to the configuration table.
- The configuration table looks simpler.

Environment	
dependency	Rating Common
	Rating Domain Model

Figure 103: Configuration table with dependency modules added by their name

Environment	
dependency	Rating *

Figure 104: Configuration table with link to all dependency modules

**Note:** When using the asterisk \* symbol, ensure that the name of the module where dependency is defined does not match the pattern as OpenL Tablets prohibits circular dependencies.

The following example illustrates how displaying dependency modules in the configuration table impacts resulting values calculation. The following modules are defined in the project for an auto insurance policy:

- Auto-Rating Algorithm.xlsx
- Auto-Rating-Domain Model.xlsx
- Auto-Rating-FL-01012016.xlsx
- Auto-Rating-OK-01012016.xlsx
- Auto-Rating Test Data.xlsx

The purpose of this project is to calculate the Vehicle premium. The main algorithm is located in the Auto-Rating Algorithm.xlsx Excel file.

Spreadsheet SpreadsheetResult DetermineVehiclePremium ( Vehicle vehicle )	
Step	Value
Vehicle	= vehicle
Age	= CurrentYear() - year
TheftRating	= VehicleTheftRating ( bodyType, price, onHighTheftProbabilityList )
InjuryRating	= VehicleInjuryRating ( bodyType, airbagType, hasRollBar )
AgeSurcharge	= AgeSurcharge ( \$Age )
InjuryRatingSurcharge	= InjuryRatingSurcharge ( \$InjuryRating )
TheftRatingSurcharge	= TheftRatingSurcharge ( \$TheftRating )
BasePremium	= BasePremium ( carType )
OtherSurcharges	= sum ( \$AgeSurcharge:\$TheftRatingSurcharge )
VehicleDiscount	= VehicleDiscount ( airbagType, hasAlarm )
<b>Premium</b>	<b>= sum ( \$BasePremium:\$OtherSurcharges ) - \$VehicleDiscount</b>

Figure 105: Rule with the algorithm to calculate the Vehicle premium

This file also contains the configuration table with the following dependency modules:

Dependency modules defined in the configuration table	
Module	Description
Auto-Rating-Domain Model.xlsx	Contains the domain model.
Auto-Rating-FL-01012016.xlsx	Contains rules with the FL state specific values used in the premium calculation.
Auto-Rating-OK-01012016.xlsx	Contains rules with the OK state specific values.

All these modules have a common part at the beginning of the name, Auto-Rating-.

So the configuration table can be defined with a link to all these modules as follows:

Environment	
dependency	Auto-Rating-*

Figure 106: Configuration table in the Auto-Rating Algorithm.xlsx file

**Note:** The dash symbol – added to the dependency modules names in a common part helps to prevent inclusion of dependency on Auto-Rating Algorithm itself.

## Properties Table

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 <b>TableProperties</b> type.

scope Identifies levels on which the property inheritance is defined. Available values are as follows:

Scope level	Description
Module	Identifies properties defined for the whole module and inherited by all tables in it. There can be only one table with the <b>Module</b> scope in one module.

Properties property_test1	
scope	Module
effectiveDate	4/7/10
expirationDate	4/28/11
lang	EN
currency	USD
state	CA

Figure 107: A properties table with the Module level scope

Category Identifies properties applied to all tables where the category name equals the name specified in the **category** element.

Properties property_test2	
scope	Category
category	Testing
country	CA,CH,DE,FR
lob	Home
lang	GER
currency	CAD

Figure 108: A properties table with the Category level scope

category	Defines the category if the <b>scope</b> element is set to <b>Category</b> . If no value is specified, the category name is retrieved from the sheet name.
Module	Identifies that properties can be overridden and inherited on the module level.



## Spreadsheet Table

In OpenL Tablets, a **spreadsheet** table 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.
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 data type and calculated in the row or column named <b>RETURN</b> .
<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 the header, make the table column and row names. Values in other cells are the table values. An example is as follows.

	A	B	C	D	E
2					
3		Spreadsheet SpreadsheetResult calc()			
4			Col1	Col2	Col3
5		Row1	0	1	2
6		Row2	3	4	5
7					

Figure 109: Spreadsheet table organization

A spreadsheet table cell can contain:

- simple values, such as a string or numeric values
- values of other data types
- formulas that start with an apostrophe followed by = or, alternatively, are enclosed by { }
- another cell value or a range of another cell values referenced in a cell formula

The following table describes how a cell value can be referenced in a spreadsheet table.

Referencing another cell		
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.

For more information on how to specify range of cells, see [Using Ranges in Spreadsheet Table](#). Below is an example of a spreadsheet table with different calculations for an auto insurance policy. Table cells contain simple values, formulas, references to the value of another cell, and other information.

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

Figure 110: Spreadsheet table with calculations as content

The data type for each cell can be determined by OpenL Tablets automatically or it can be defined explicitly for each row or column. The data type for a whole row or column can be specified using the following syntax:

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

**Note:** If both column and row of the cell have a data type specified, the data type of the column is taken.

The following topics are included in this section:

- [Parsing a Spreadsheet Table](#)
- [Accessing Spreadsheet Result Cells](#)
- [Using Ranges in Spreadsheet Table](#)
- [Custom Spreadsheet Result](#)

### Parsing a Spreadsheet Table

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

1. A spreadsheet returns the result of **SpreadsheetResult** data type.
2. A spreadsheet returns the result of any other data type different from **SpreadsheetResult**.

In the first case, users get the value of SpreadsheetResult type that is an analog of result matrix. All calculated cells of the spreadsheet table are accessible through this result. The following example displays a spreadsheet table of this type.

Spreadsheet SpreadsheetResult DetermineDriverPremium (Driver driver)	
Step	Value
DriverType	= DriverAgeType ( gender, age )
Eligibility	= DriverEligibility ( \$DriverType, hadTraining )
DriverRisk	= DriverRisk ( numDUI, numAccidents, numMovingViolations )
DriverTypeScore	= DriverTypeScore ( \$DriverType, \$Eligibility )
DriverRiskScore	= DriverRiskScore ( \$DriverRisk )
DriverPremium	= DriverPremium ( \$DriverType, maritalStatus, state )
DriverRiskPremium	= DriverRiskPremium ( \$DriverRisk )
AccidentPremium	= AccidentPremium () * numAccidents
<b>Score</b>	= sum ( \$DriverTypeScore:\$DriverRiskScore )
<b>Premium</b>	= sum ( \$DriverPremium:\$AccidentPremium )

Results of running [DetermineDriverPremium](#)

ID	driver	Result																						
1	<div style="border: 1px solid gray; padding: 5px;">                     Driver (Sara)                      name = Sara                      gender = Female                      age = 25                      maritalStatus = Single                      state = NY                      numAccidents = 1                      numMovingViolations = 0                      numDUI = 0                      hadTraining = false                 </div>	<table border="1"> <thead> <tr> <th>Step</th> <th>Value</th> </tr> </thead> <tbody> <tr> <td>DriverType</td> <td>Standard Driver</td> </tr> <tr> <td>Eligibility</td> <td>Eligible</td> </tr> <tr> <td>DriverRisk</td> <td>Standard Risk Driver</td> </tr> <tr> <td>DriverTypeScore</td> <td>0.0</td> </tr> <tr> <td>DriverRiskScore</td> <td>0.0</td> </tr> <tr> <td>DriverPremium</td> <td>0.0</td> </tr> <tr> <td>DriverRiskPremium</td> <td>0.0</td> </tr> <tr> <td>AccidentPremium</td> <td>150.0</td> </tr> <tr> <td><b>Score</b></td> <td>0.0</td> </tr> <tr> <td><b>Premium</b></td> <td>150.0</td> </tr> </tbody> </table>	Step	Value	DriverType	Standard Driver	Eligibility	Eligible	DriverRisk	Standard Risk Driver	DriverTypeScore	0.0	DriverRiskScore	0.0	DriverPremium	0.0	DriverRiskPremium	0.0	AccidentPremium	150.0	<b>Score</b>	0.0	<b>Premium</b>	150.0
Step	Value																							
DriverType	Standard Driver																							
Eligibility	Eligible																							
DriverRisk	Standard Risk Driver																							
DriverTypeScore	0.0																							
DriverRiskScore	0.0																							
DriverPremium	0.0																							
DriverRiskPremium	0.0																							
AccidentPremium	150.0																							
<b>Score</b>	0.0																							
<b>Premium</b>	150.0																							

Figure 111: Spreadsheet table returns the SpreadsheetResult datatype

In the second case, the returned result type is a data type as in all other rule tables, so there is no need for **SpreadsheetResult** in the rule table header. The cell with the RETURN key word for a row will be returned. OpenL Tablets calculates the cells required just for that result calculation. In the following example, the **License\_Points** cell is not included in the **Tier Factor** calculation and can 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:\$Violations_Points )	
Tier_Factor	= tierFactor = mapTierPointsToFactor ( \$Total_Points )	
<b>RETURN</b>	<b>= \$Tier_Factor</b>	

Figure 112: Spreadsheet table returning a single value

### Accessing Spreadsheet Result Cells

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

```
<spreadsheet result variable>.$<column name>$<row name>
```

or

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

The following example demonstrates how to get a value of the **FinancialRatingCalculation** spreadsheet result that is calculated in the **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	= <a href="#">SetNonZeroValues</a> ( financialData )
FinancialRatingCalculation	= <a href="#">FinancialRatingCalculation</a> ( financialData, industry )
FinancialRating	= <a href="#">\$FinancialRatingCalculation.\$Value\$FinancialRating</a>
RiskOfProfile	= <a href="#">RiskOfProfile</a> ( corporate )
RiskOfOperations	= <a href="#">RiskOfOperations</a> ( qualityIndicators )
RiskOfGeography	= <a href="#">RiskOfGeography</a> ( qualityIndicators )

Figure 113: Accessing Spreadsheet Result cell value

The spreadsheet cell can also be accessed using the `getFieldValue(String <cell name>)` function, for instance, `(DoubleValue) $FinancialRatingCalculation.getFieldValue (" $Value$FinancialRating")`. This is a more complicated option.

**Note:** If the cell name in columns or rows contains forbidden symbols, such as space or percentage, the cell cannot be accessed. For more information on symbols that are not allowed, see Java method documentation.

### Using Ranges in Spreadsheet Table

The following syntax is used to specify a range in a spreadsheet table:

```
$FirstValue:$LastValue
```

An example of using a range this way in the **TotalAmount** column is as follows.

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 114: Using ranges of Spreadsheet table in functions

**Note:** In expressions, such as `min/max($FirstValue:$LastValue)`, there must be no space before and after the colon (:) operator.

**Note:** It is impossible to make math operations under ranges which names are specified with spaces. Please use step names without spaces.

### Auto Type Discovery usage

OpenL Tablets determines the cell data type automatically without its definition for a row or column. A user can turn on or off this behavior using the **autotype** property. If any row or column contains explicit data type definition, it supersedes automatically determined data type. The following example demonstrates that any data type can be correctly determined in auto mode. A user can put the mouse cursor over the “=” symbol to check the type of the cell value in OpenL Tablets WebStudio.

Step	Value
Policy	<b>Cell type: SpreadsheetResultDetermineVehiclePremium[]</b>
Vehicles	= DetermineVehiclePremium ( vehicles )
Drivers	= DetermineDriverPremium ( drivers )
VehiclesPremium	= sum ( \$Value\$Premium ( \$Vehicles ) )
DriversPremium	= sum ( \$Value\$Premium ( \$Drivers ) )
ClientDiscount	= ClientDiscount ( clientTier )
VehiclesScore	= sum ( \$Value\$Score ( \$Vehicles ) )
DriversScore	= sum ( \$Value\$Score ( \$Drivers ) )
ClientTierScore	= ClientTierScore ( clientTier )
Eligibility	= PolicyEligibility ( clientTerm, \$Score )
Score	= sum ( \$VehiclesScore:\$ClientTierScore )
Premium	= sum ( \$VehiclesPremium:\$DriversPremium ) - \$ClientDiscount

Figure 115: Auto Type Discovery Property Usage inside Spreadsheet table

However, there are several limitations of auto type discovering when the system cannot possibly determine the cell data type:

- A user explicitly defines the type of the value as common SpreadsheetResult, for instance, in input parameter definition as follows:

Step	Value
VehicleID	<b>Cell type: Object</b>
MainDriverRisk	driverCalc.\$Value\$DriverRisk
Age	= CurrentYear() - year

Instead, in such cases, the [custom spreadsheet result](#) type must be used to allow correct auto type discovering as follows:

Spreadsheet SpreadsheetResult DetermineVehiclePremium (Vehicle vehicle, SpreadsheetResultDetermineDriverPremium driverCalc)	
Step	Value
VehicleID	Cell type: DriverRisk
MainDriverRisk	driverCalc.\$Value\$DriverRisk
Age	= CurrentYear() - year

- A user explicitly defines the return type of other Rules tables, such as Decision tables, as common SpreadsheetResult as follows:

Rules SpreadsheetResult DriverEligibility (Driver driver)	
C1	RET1
driverType	
DriverType	
Driver Status	Driver Eligibility
Young Driver	= YoungDriverEligibilityCalc (driver)
Senior Driver	= SeniorDriverEligibilityCalc (driver)
	= StandardDriverEligibilityCalc (driver)

Spreadsheet SpreadsheetResult DriverCalc (Driver driver)	
Step	Value
DriverID	Cell type: Object
Eligibility	DriverEligibility (driver).\$Value\$Eligibility

The type of undefined cells must be explicitly defined as a custom spreadsheet result type or any other suitable type to avoid uncertain Object typing.

- There is a circular dependency in a spreadsheet table calling the same spreadsheet rule itself in a cell. This cell type must be explicitly defined to allow correct auto type discovering of the whole spreadsheet table as follows:

Spreadsheet SpreadsheetResult DeterminePremium (Driver driver)	
Step	Value
Low	= LowPremium(driver)
High	= HighPremium(driver)
UpdatedDriver	= UpdateDriver(driver)
Premium : Double	= \$Low <= \$High ? DeterminePremium (\$UpdatedDriver).\$Value\$Premium : (\$Low + \$High)/2

### Custom Spreadsheet Result

Usage of spreadsheet tables that return the SpreadsheetResult type is improved by having a separate type for each such spreadsheet table, that is, custom SpreadsheetResult data type, which is determined as follows:

SpreadsheetResult<Spreadsheet table name>

Custom SpreadsheetResult data type is automatically generated by a system and substitutes common SpreadsheetResult type. This provides the following advantages:

the system understands the structure of the spreadsheet result, i.e., knows names of columns and rows, and data types of cell values. In other words, there is no need to indicate a data type when accessing the cell.

- test spreadsheet cell of any complex type  
For more information on test spreadsheet result, see [Testing Spreadsheet Result](#).

To understand how this works, consider the following spreadsheet.

Spreadsheet SpreadsheetResult CoveragePremium (String coverageId, Integer coveredProperty, Double koef)	
Step	Value
Coverage_Id	= coverageId
Covered_Property	= coveredProperty + 1
Premium	= koef * \$Covered_Property

Figure 116: An example of a spreadsheet

The return type is **SpreadsheetResult**, but it becomes **SpreadsheetResultCoveragePremium** data type. Now it is possible to access any calculated cell in a very simplified way without indicating its data type, for example, as displayed in the following figure.

Spreadsheet SpreadsheetResult FinalPremium (String coverageId, Integer coveredProperty, Double koef)	
Step	Value
Coverage_Id	<i>Cell type: SpreadsheetResultCoveragePremium</i>
Coverage_Calc	= <a href="#">CoveragePremium</a> (coverageId, coveredProperty, koef)
Final_Premium	= <a href="#">\$Coverage_Calc.\$Value\$Premium</a>

Figure 117: Calling Spreadsheet cell

In this example, the spreadsheet table cell is accessed 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	<a href="#">MATCH Algorithm</a>
SCORE	<a href="#">SCORE Algorithm</a>
WEIGHTED	<a href="#">WEIGHTED Algorithm</a>

Each algorithm has the following mandatory columns:

Algorithm mandatory columns													
Column	Description												
Names	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 <b>some</b> property, it is enough to specify <b>some</b> in the names column.												
Operations	The <b>operations</b> column defines how to match or check arguments to values in a table. The following operations are available: <table border="1" data-bbox="321 793 1468 1003"> <thead> <tr> <th>Operation</th> <th>Checks for</th> <th>Description</th> </tr> </thead> <tbody> <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> </tbody> </table> <p>The <b>min</b> and <b>max</b> operations work with numeric and date types only. The <b>min</b> and <b>max</b> operations can be replaced with the <b>match</b> operation and ranges. This approach adds more flexibility because it enables verifying all cases within one row.</p>	Operation	Checks for	Description	match	equality or belonging to a range	The argument value must be equal to or within a range of check values.	min	minimally required value	The argument must not be less than the check value.	max	maximally allowed value	The argument must not be greater than the check value.
Operation	Checks for	Description											
match	equality or belonging to a range	The argument value must be equal to or within a range of check values.											
min	minimally required value	The argument must not be less than the check value.											
max	maximally allowed value	The argument must not be greater than the check value.											
Values	The <b>values</b> column typically has multiple sub columns containing table values.												

The following topics are included in this section:

- [MATCH Algorithm](#)
- [SCORE Algorithm](#)
- [WEIGHTED Algorithm](#)

### MATCH Algorithm

The **MATCH** algorithm allows mapping a set of conditions to a single return value.

Besides the mandatory columns, such as 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 118: An example of the MATCH algorithm table

The MATCH algorithm works from top to bottom 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 runtime 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 119: 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 multiplies 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 120: 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 a 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.

Tbasic table header syntax	
Element	Description
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.
User functions and subroutines	Contains user functions definition and subroutines.

## Table Part

The **Table Part** functionality enables the user to split a large table into smaller parts, or partial tables. Physically, in the Excel workbook, the table is represented as several table parts which logically are processed as one rules table.

This functionality is suitable for cases when a user is dealing with `.xls` file format using a rules table with more than 256 columns or 65,536 rows. To create such a rule table, a 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 first table part, the next N2 rows in the second table part, and so on. In horizontal case, the first N1 columns of the rule table are placed in the first table part, the next N2 columns in the second table part, and so on. The header of the original rule table and its properties definition must be copied to each table part in case of horizontal splitting. Merging of table parts into the rule table is processed as depicted in the following figures.

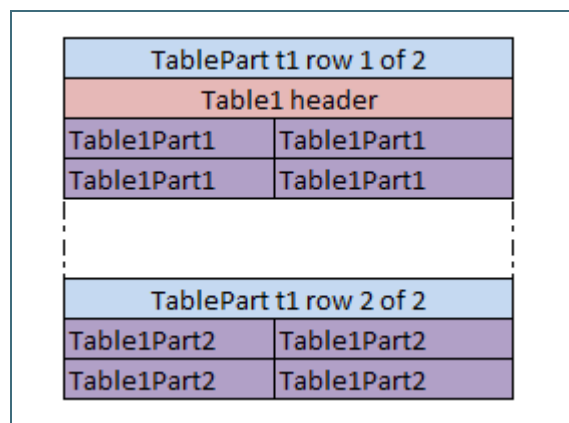


Figure 121: Vertical merging of table parts

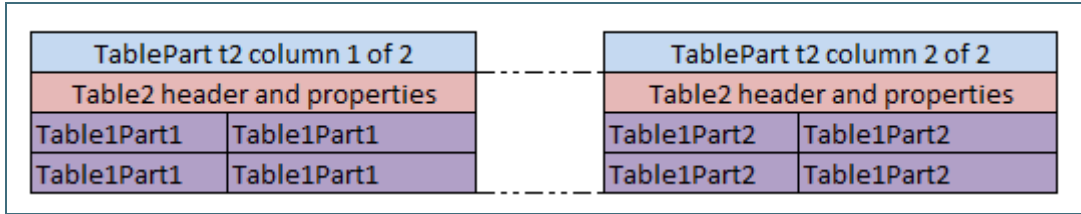


Figure 112: Horizontal merging of table parts

All table parts must be located within one Excel file.

Splitting can be applied to 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>	Unique name of the rules table. It can be the same as the rules table name if the rules table is not overloaded by properties.
<split type>	Type of splitting. It is set to <b>row</b> for vertical splitting and <b>column</b> 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.

The following examples illustrate vertical and horizontal splitting of the **RiskOfWorkWithCorporate** decision rule.

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

Figure 123: Table Parts example. Vertical splitting part 1

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 124: Table Parts example. Vertical splitting part2

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

Figure 125: Table Part example. Horizontal splitting part 1

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 126: Table Parts example. Horizontal splitting part 2

# 4 OpenL Tablets Functions and Supported Data Types

---

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

To implement business rules logic, users need to instruct OpenL Tablets what they want to do. For that, one or several rule tables with user's rules logic description must be created.

Usually rules operate with some data from user's 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 functions for business rules management in the system and introduces basic principles of using arrays.

The section includes the following topics:

- [Working with Arrays](#)
- [Working with Data Types](#)
- [Working with Functions](#)

## 4.1 Working with Arrays

An **array** is a collection of values of the same type. Separate values of an array are called **array elements**. An **array element** is a value of any data type available in the system, such as `IntValue`, `Double`, `Boolean`, and `String`. For more information on OpenL Tablets Data Types, see [Working with Data Types](#).

Square brackets in the name of the data type indicate that there is an array of values in the user's rule to be dealt with. For example, the `String[]` expression can be used to represent an array of text elements of the **String** data type, such as US state names, for example, CA, NJ, and VA. Users use arrays for different purposes, such as calculating statistics and representing multiple rates.

The following topics are included in this section:

- [Working with Arrays from Rules](#)
- [Array Index Operators](#)
- [Functions to Work with Arrays](#)
- [Rules Applied to Array](#)

### Working with Arrays from Rules

Data type arrays can be used in rules as follows:

Using data type arrays in rules	
Method	Description
By numeric index, starting from 0	In this case, by calling <code>drivers[5]</code> , a user gets the sixth element of the data type array.

Using data type arrays in rules	
Method	Description
By user defined index	This case is a little more complicated. The first field of data type is considered to be the user defined index. For example, if there is a <b>Driver</b> data type with the first String field name, a data table can be created, initializing two instances of <b>Driver</b> with the following names: John and David. Then in rules, the required instance can be called by <code>drivers["David"]</code> . All Java types, including primitives, and data types can be used for user specific indexes. When the first field of data type is of <code>int</code> type called <code>id</code> , to call the instance from array, wrap it with quotes as in <code>drivers["7"]</code> . In this case, a user does not get the eighth element in the array, but the <b>Driver</b> with ID=7. For more information on data tables, see <a href="#">Data Table</a> .
By conditional index	Another case is to use conditions that consider which elements must be selected. For this purpose, SELECT operators are used, which specify conditions for selection. For more information on how to use SELECT operators, see <a href="#">Array Index Operators</a> .
By other array index operators and functions	Any index operator listed in <a href="#">Array Index Operators</a> or a function designed to work with arrays can be applied to an array in user's rules. The full list of OpenL Tablets array functions is provided in <a href="#">Appendix B: Functions Used in OpenL Tablets</a> .

## 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.

This section provides detailed description of index operators along with examples. OpenL Tablets supports the following index operators:

- [SELECT Operators](#)
- [ORDER BY Operators](#)
- [SPLIT BY Operator](#)
- [TRANSFORM TO Operators](#)
- [Array Index Operators and Arrays of the SpreadsheetResult Type](#)
- [Advanced Usage of Array Index Operators](#)

## SELECT Operators

There are cases requiring conditions that determine the elements of the array to be selected. For example, if there is a data type **Driver** with such fields as **name** of the String type, **age** of the Integer type, and other similar data, and all drivers with the name **John** aged under **20** must be selected, use the following SELECT operator realizing conditional index:

```
arrayOfDrivers[select all having name == "John" and age < 20]
```

The following table describes the SELECT operator types:

SELECT operator types	
Type	Description
Returns the first element satisfying the condition	Returns the first matching element or null if there is no such element. <b>Syntax:</b> <code>array[!@ &lt;condition&gt;]</code> or <code>array[select first having &lt;condition&gt;]</code> <b>Example:</b> <code>arrayOfDrivers[!@ name == "John" and age &lt; 20]</code>

SELECT operator types	
Type	Description
Returns all elements satisfying the condition	Returns the array of matching elements or empty array if there are no such elements. <b>Syntax:</b> <code>array[@ &lt;condition&gt;]</code> OR <code>array [select all having &lt;condition&gt;]</code> <b>Example:</b> <code>arrayOfDrivers[@ numAccidents &gt; 3]</code>

### ORDER BY Operators

These operators are intended to sort elements of the array. Consider a data type **Claim** with such fields as **lossDate** of the Date type, **paymentAmount** of the Double type, and other similar data, and all claims must be sorted by loss date starting with the earliest one. In this case, use the ORDER BY operator, such as `claims[order by lossDate]`.

The following table describes ways of sorting:

ORDER BY sorting methods	
Method	Description
Sort elements by increasing order	<b>Syntax:</b> <code>array[^@ &lt;expression&gt;]</code> OR <code>array[order by &lt;expression&gt;]</code> OR <code>array[order increasing by &lt;expression&gt;]</code> <b>Example:</b> <code>claims[^@ lossDate]</code>
Sort elements by decreasing order	<b>Syntax:</b> <code>array[v@ &lt;expression&gt;]</code> OR <code>array[order decreasing by &lt;expression&gt;]</code> <b>Example:</b> <code>claims[v@ paymentAmount]</code>

**Note:** The operator returns the array with ordered elements. It saves element order in case of equal elements. `<expression>` by which ordering is performed must have a comparable type, such as Date, String, Number.

### SPLIT BY Operator

To split array elements into groups by definite criteria, use SPLIT BY operator, which returns a 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 three collections, `{"5000"}`, `{"2002", "2113"}` and `{"3300"}` united by codes with the equal first number.

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

**Example:** `orders[~@ orderType]`

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

The SPLIT BY operator returns a two-dimensional array containing arrays of elements split by an equal value of `<expression>`. The relative element order is preserved.

### TRANSFORM TO Operators

This operator turns source array elements into another transformed array in a quick way. Assume that a collection of claims is available, and **claim ID** and **loss date** information for each claim in the form of array of strings needs to be returned. Use the TRANSFORM TO operator, such as `claims[transform to id + " - " + dateToString(lossDate, "dd.MM.YY")]`.

The following table describes methods of transforming:



TRANSFORM TO methods	
Method	Description
Transforms elements and returns the whole transformed array	<b>Syntax:</b> array[*@ <expression>] or array[transform to <expression>] <b>Example:</b> drivers[transform to name] or drivers[*@ name]
Transforms elements and returns unique elements of the transformed array only	<b>Syntax:</b> array[*!@ <expression>] or array[transform unique to <expression>] <b>Example:</b> drivers[transform unique to vehicle] or drivers[*!@ vehicle]

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

The operator returns array of the <expression> type. The order of the elements is preserved.

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

### Array Index Operators and Arrays of the SpreadsheetResult Type

Array index operators can be used with arrays which elements are of SpreadsheetResult data type. In order to refer to a cell of SpreadsheetResult element in the operator condition, the full reference \$columnName\$rowName is used.

Consider an example with select operator. There is a rule which selects and returns spreadsheet result with value 2 in the \$Formula\$EmployeeClassId cell.

```
Method SpreadsheetResult FirstEmpl(SpreadsheetResult[] allEmployeeClassPremiums)
return allEmployeeClassPremiums[select first having $Formula$EmployeeClassId == 2];
```

Figure 127: index operator applied on array of SpreadsheetResults

where the spreadsheet result element of allEmployeeClassPremiums array is calculated from the following spreadsheet table:

Spreadsheet SpreadsheetResult EmployeeClassPremium ( EmployeeClass employeeClass )	
Step	Formula
EmployeeClassId	= id

Figure 128: Spreadsheet for allEmployeeClassPremiums array result calculation

### Advanced Usage of Array Index Operators

Consider a case when the name of the array element needs to be referred explicitly in condition or expression. For example, the policy has a collection of drivers of Driver[] data type and a user wants to select all policy drivers of the age less than 19, except for the primary driver. The following syntax with an explicit definition of the Driver d collection element can be used:

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

The expression can 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:** Array index operators can be applied to lists. Usually it requires using a named element to define a type of list components, such as `List claims = policy.getClaims(); claims[(Claim claim) order by claim.date]` or `List claims = policy.getClaims(); claims[(Claim claim) ^@ date]`.

## Functions to Work with Arrays

This section provides detailed description of the **Length** array function with examples. For more information on array functions, see [Appendix B: Functions Used in OpenL Tablets](#).

The **Length** array function returns the number of elements in the array as a result value. An example is as follows.

Rules String <b>InsuranceProcedure</b> (Policy policy)	
C1	RET1
length(vehicles)	res
	String res
<b>Car park</b>	<b>Insurance procedure</b>
2	Senior Auto Driver
1	Standard Auto Driver

Figure 129: Rule table with the length function

In this example, the **Insure** procedure depends on the number of vehicles. The policy includes vehicles field represented as array.

Test InsuranceProcedure InsuranceProcedureTest	
policy	_res_
> policyProfile2	
<b>Name of Policy</b>	<b>Insurance procedure</b>
1 Policy2	Senior Auto Driver

ID	Name of Policy	Insurance procedure
1	Policy (Policy2)	Senior Auto Driver

Figure 130: Test table for rule table with length function

Policy2 contains two vehicles as illustrated in the following data table.

Data Policy policyProfile2			
name		<b>Policy</b>	Policy2
			Sara
drivers	>driverProfiles1	<b>Drivers</b>	Spencer, Sara's Son
			2004 Honda Odyssey
vehicles	>autoProfiles2	<b>Vehicles</b>	2001 Toyota Camry
clientTier		<b>Client Tier</b>	Preferred
clientTerm		<b>Client Term</b>	

Figure 131: Data table for a test table

## Rules Applied to Array

OpenL Tablets allows applying a rule intended for work with one value to an array of values. The following example demonstrates this feature in a 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 132: Applying a rule to an array of values

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

If several input parameters for a rule are arrays where the rule expects only a single value, the rule is separately calculated for each element of these arrays, and the result is an array of the return type. In other words, OpenL Tablets executes the rule for each combination of input values from arrays and return a collection of all these combinations' results. The order in which these arrays are iterated is not specified.

**Note:** OpenL Tablets engine may run parts of one request in parallel and Dev property `Concurrent Execution` is used to enable or disable this behavior in case when the rule table is applied to an array of value instead of a single value. `Concurrent Execution` is useful for complex rule sets where parallel execution will improve performance for a single request. But note that modifying arguments of the rule are not thread safe.

## 4.2 Working with Data Types

Data in OpenL Tablets must have a type of data defined. A data type indicates the meaning of the data, their possible values, and instructs OpenL Tablets how to process operations, which rules can be performed, and how these rules and operations affect data.

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

Data types in OpenL Tablets	
Type	Description
Predefined data types	Types that exist in OpenL Tablets, can be used, but cannot be modified.
Custom data types	Types created by a user as described in the <a href="#">Datatype Table</a> 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 can be used in user’s business rules in OpenL Tablets:

Simple Data Types				
Data Type	Description	Examples	Usage in OpenL Tablets	Notes
Integer	Used to work with whole numbers without fraction points.	8; 45; 12; 356; 2011	Common for representing a variety of numbers, such as driver’s age, a year, a number of points, and mileage.	Not exceeding 2,147,483,647.
Double	Used for operations with fractional numbers. Can hold very large or very small numbers.	8.4; 10.5; 12.8; 12,000.00; 44.416666666666664	Commonly used for calculating balances or discount values for representing exchange rates, a monthly income, and so on.	
BigInteger	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	Only used for operations on very big values over two billion, for example, dollar deposit in Bulgarian Leva equivalent.	
BigDecimal	Represents decimal numbers with a 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	Often used for currency calculations or in financial reports that require exact mathematical calculations, for example, a year bank deposit premium calculation.	

Simple Data Types				
Data Type	Description	Examples	Usage in OpenL Tablets	Notes
String	Represents text rather than numbers. String values are comprised of a set of characters that can contain spaces and numbers. For example, the word <b>Chrysler</b> and the phrase <b>The Chrysler factory warranty is valid for 3 years</b> are both Strings.	John Smith, London, Alaska, BMW; Driver is too young.	Represents cities, states, people names, car models, genders, marital statuses, as well as messages, such as warnings, reasons, notes, diagnosis, and other similar data.	
Boolean	Represents only two possible values: <code>true</code> and <code>false</code> . For example, if a driver is trained, the condition is <code>true</code> , and the insurance premium coefficient is 1.5. If the driver is not trained, the condition is <code>false</code> , and the coefficient is 0.25.	<code>true</code> ; <code>yes</code> ; <code>y</code> ; <code>false</code> ; <code>no</code> ; <code>n</code>	Handles conditions in OpenL Tablets.	The synonym for 'true' is 'yes', 'y'; for 'false' – 'no', 'n'.
Date	Used to operate with dates.	06/05/2010; 01/22/2014; 11/07/95; 1/1/1991.	Represents any dates, such as policy effective date, date of birth, and report date. If the date is defined as a text cell value, it is expected in the <code>&lt;month&gt;/&lt;date&gt;/&lt;year&gt;</code> format.	

Byte, Character, Short, Long, and Float data types are rarely used in OpenL Tablets, therefore, ranges of values are only provided in the following table. For more information about values, see the appropriate Java documentation.

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

## Value Data Types

In OpenL Tablets, **value data types** are exactly the same as simple data types described in [Simple Data Types](#), except for an **explanation**, a clickable field displayed in the test results table in OpenL Tablets WebStudio. Value data types provide detailed information on results of rules testing and are useful for working with calculated values to have better debugging capabilities. By clicking the linked value, users can view the source table for that value and get information on how the value is calculated.

**CarPrice2009Test** 3 test cases

Order Date	Car	Billing Region	Car Price
01/06/2009	Car model = Z4 sDrive35i brand = BMW	Address country = GreatBritain region = Scotland	53650

Rule	Country	Region	BMW		Porche			Audi		Car Price
			Z4 sDrive35i	Z4 sDrive30i	911 Carrera 4S	911 Targa 4	911 Carrera Cabriolet	2009 Audi R8 4.2 quattro Auto R Tronic	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	93200
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	121500
R7		Midland	\$52,450	\$46,550	\$93,200	\$90,400	\$87,000	\$121,500	\$112,500	

Figure 133: Usage of value data type

OpenL Tablets supports the following value data types:

- ByteValue
- ShortValue
- IntValue
- LongValue
- FloatValue
- DoubleValue
- BigIntegerValue
- BigDecimalValue

## Range Data Types

**Range Data Types** can be used when a business rule must 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 a range of ages can be defined, and one rule for all drivers from within that range can be created. The way to inform OpenL Tablets that the rule must be applied to a group of drivers is to declare driver’s age as the range data type.

OpenL Tablets supports the following range data types:

Range data types in OpenL Tablets	
Type	Description
IntRange	Intended for processing whole numbers within an interval, for example, vehicle or driver age for calculation of insurance compensations, or years of service when calculating the annual bonus.
DoubleRange	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.

The following illustration provides a very simple example of how to use a 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. An amount of cars per order is defined as IntRange data type. For a 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 134: Usage of the range data type

Supported range formats are as follows:

Range formats				
#	Format	Interval	Example	Values for IntRange
1	[<min_number>; <max_number>) Mathematic definition for ranges using square brackets for included borders and round brackets for excluded borders.	[min; max]	[1; 4]	1, 2, 3, 4
		(min; max)	(1; 4)	2, 3
		[min; max)	[1; 4)	1, 2, 3
		(min; max]	(1; 4]	2, 3, 4
2	[<min_number> .. <max_number>) Mathematic definition for ranges with two dots used instead of semicolon.	[min; max]	[1 .. 4]	1, 2, 3, 4
		(min; max)	(1 .. 4)	2, 3
		[min; max)	[1 .. 4)	1, 2, 3
		(min; max]	(1 .. 4]	2, 3, 4
3	<min_number> - <max_number>	[min; max]	1 - 4	[1; 4]
			-2 - 2	[-2; 2]
			-4 - -2	[-4; -2]
4	<min_number> .. <max_number>	[min; max]	1 .. 4	1, 2, 3, 4
5	<min_number> ... <max_number>	(min; max)	1 ... 4	2, 3
6	<<max_number>	[-∞; max)	<2	-∞ ..., -1, 0, 1
7	<=<max_number>	[-∞; max]	<=2	-∞ ..., -1, 0, 1, 2
8	><min_number>	(min; +∞]	>2	3, 4, 5, ... +∞
9	>=<min_number>	[min; +∞]	>=2	2, 3, 4, 5, ... +∞
10	><min_number> <<max_number> <<max_number> ><min_number>	(min; max)	>1 <4	2, 3
			<4 >1	2, 3
11	>=<min_number> <<max_number> <<max_number> >=<min_number>	[min; max)	>=1 <4	1, 2, 3
			<4 >=1	1, 2, 3
12	><min_number> <=<max_number> <=<max_number> ><min_number>	(min; max]	>1 <=4	2, 3, 4
			<=4 >1	2, 3, 4
13	>=<min_number> <=<max_number> <=<max_number> >=<min_number>	[min; max]	>=1 <=4	1, 2, 3, 4
			<=4 >=1	1, 2, 3, 4
14	<min_number>+	[min; +∞]	2+	2, 3, 4, 5, ... +∞

Range formats				
#	Format	Interval	Example	Values for IntRange
15	<min_number> and more	[min; +∞]	2 and more	2, 3, 4, 5, ... +∞
16	more than <min_number>	(min; +∞]	more than 2	3, 4, 5, ... +∞
17	less than <max_number>	[-∞; max)	less than 2	-∞ ..., -1, 0, 1

The following rules apply:

- Infinities in IntRange are represented as `Integer.MIN_VALUE` for  $-\infty$  and `Integer.MAX_VALUE` for  $+\infty$ .
- Using of "." and "..." requires spaces between numbers and dots.
- Numbers can be enhanced with the \$ sign as a prefix and K, M, B as a postfix, for example, \$1K = 1000.
- For negative values, use the '-' (minus) sign before the number, for example, -<number>.

## 4.3 Working with Functions

Data types are used to represent user data in the system. Business logic in rules is implemented using **functions**. Examples of functions are the **Sum** function used to calculate a sum of values and **Min/Max** functions used to find the minimum or maximum values in a set of values.

This section describes OpenL Tablets functions and provides simple usage examples. All functions can be divided into the following groups:

- math functions
- array processing functions
- date functions
- String functions
- error handling functions

The following topics are included in this section:

- [Understanding OpenL Tablets Function Syntax](#)
- [Math Functions](#)
- [Date Functions](#)
- [Special Functions and Operators](#)
- [Null Elements Usage in Calculations](#)

### Understanding OpenL Tablets Function Syntax

This section briefly describes how functions work in OpenL Tablets.

Any function is represented by the following elements:

- function name or identifier, such as **sum**, **sort**, **median**
- function parameters
- value or values that the function returns

For example, in the `max(value1, value2)` expression, **max** is the rule or function name, (**value1, value2**) are function parameters, that is, values that take part in the action. When determining **value1** and **value2** as 50 and 41, the given function looks as `max(50, 41)` and returns **50** in result as the biggest number in the couple.



If an action is performed in a rule, use the corresponding function in the rules table. For example, to calculate the best result for a gamer in the following example, use the **max** function and enter `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 to determine the gamer level.

Subsequent sections provide description for mostly often used OpenL Tablets functions. For a full list of functions, see [Appendix B: Functions Used in OpenL Tablets](#).

## Math Functions

Math functions serve for performing math operations on numeric data. These functions support all numeric data types described in [Working with Data Types](#).

The following example illustrates how to use functions in OpenL Tablets. The rule in the diagram defines a gamer level depending on the best result in three attempts.

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

Figure 135: An example of using the 'max' function

The following topics are included in this section:

- [Math Functions Used in OpenL Tablets](#)
- [Round Function](#)

### Math Functions Used in OpenL Tablets

The following table lists math functions used in OpenL Tablets:

Math functions used in OpenL Tablets	
Function	Description
<b>min/max</b>	Returns the smallest or biggest number in a set of numbers for array or multiple values. The function result is a number. <code>min/max(number1, number2, ...)</code> <code>min/max(array[])</code> is the array that must be previously defined in the given rule table or in a different table. In the previous example, the <code>max(score1, score2, score3)</code> expression is used to define the highest result for a player. For example, <code>max(1, 5, 3)</code> gives <b>5</b> as the result so the player level is <b>medium</b> as defined in the RET1 column.
<b>sum</b>	Adds all numbers in the provided array and returns the result as a number. <code>sum(number1, number2, ...)</code> <code>sum(array[])</code>
<b>avg</b>	Returns the arithmetic average of array elements. The function result is a number. <code>avg(number1, number2, ...)</code> <code>avg(array[])</code>

Math functions used in OpenL Tablets	
Function	Description
<b>product</b>	Multiplies numbers from the provided array and returns the product as a number. <code>product(number1, number2, ...)</code> <code>product(array[])</code>
<b>mod</b>	Returns the remainder after a number is divided by a divisor. The result is a numeric value and has the same sign as the divisor. <code>mod(number, divisor)</code> <code>number</code> is a numeric value which's remainder must be found. <code>divisor</code> is the number used to divide the <code>number</code> . If the divisor is <b>0</b> , the <b>mod</b> function returns an error.
<b>sort</b>	Returns values from the provided array in ascending sort. The result is an array. <code>sort(array[])</code>
<b>round</b>	Rounds a value to a specified number of digits. For more information on the ROUND function, see <a href="#">Round Function</a> .

### Round Function

The **ROUND** function is used to round a value to a specified number of digits. For example, in financial operations, users may want to calculate insurance premium with accuracy up to two decimals. Usually a number of digits in long data types, such as DoubleValue or BigDecimal, must be limited. The ROUND function allows rounding a value to a whole number or to a fractional number with limited number of signs after decimal point.

The ROUND function syntax is as follows:

ROUND function syntax	
Syntax	Description
<code>round(DoubleValue)</code>	Rounds to the whole number.
<code>round(DoubleValue, int)</code>	Rounds to the fractional number. <code>int</code> is a number of digits after decimal point.
<code>round(DoubleValue, int, int)</code>	Rounds to the fractional number and enables to get results different from usual mathematical rules: <ul style="list-style-type: none"> <li>• The first <code>int</code> stands for a number of digits after decimal point.</li> <li>• The second <code>int</code> stands for a rounding mode represented by a constant, for example, 1- ROUND DOWN, 4- ROUND HALF UP.</li> </ul>

The following topics are included in this section:

- [round\(DoubleValue\)](#)
- [round\(DoubleValue,int\)](#)
- [round\(DoubleValue,int,int\)](#)

#### **round(DoubleValue)**

This syntax is used to round to a whole number. The following example demonstrates function usage:

Rules DoubleValue <b>roundToInteger</b> (DoubleValue value2)	
C1	RET1
true	
Boolean condition	
<b>Condition</b>	<b>Rate</b>
	=round(value)

Figure 136: Rounding to integer

Testmethod <b>roundToInteger</b> 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 55: Test table for rounding to integer

**round(DoubleValue,int)**

This function is used to round to a fractional number. The second parameter defines a number of digits after decimal point.

Rules DoubleValue <b>round</b> (DoubleValue value)	
C1	RET1
true	
Boolean condition	
<b>Condition</b>	<b>Rate</b>
	=round(value,2)

Figure 138: Rounding to a fractional number

Testmethod <b>round</b> roundTest		
_description_	value	_res_
TestID	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 139: Test table for rounding to a fractional number

**round(DoubleValue,int,int)**

This function allows rounding to a fractional number and get results by applying different mathematical rules. The following parameters are expected:

- Number to round
- The first *int* stands for a number of digits after decimal point
- The second *int* stands for a rounding mode represented by a constant, for example, 1- ROUND\_DOWN, 4- ROUND\_HALF\_UP.

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

Constants list		
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 the nearest neighbor unless both neighbors are equidistant, in which case round up.
5	ROUND_HALF_DOWN	Rounding mode to round towards the 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 information on the constants representing rounding modes, see [http://docs.oracle.com/javase/6/docs/api/constant-values.html#java.math.BigDecimal.ROUND\\_HALF\\_DOWN](http://docs.oracle.com/javase/6/docs/api/constant-values.html#java.math.BigDecimal.ROUND_HALF_DOWN).

For more information on the constants with examples, see <http://docs.oracle.com/javase/6/docs/api/java/math/RoundingMode.html>, *Enum Constant Details* section.

The following example demonstrates how the rounding works with the `ROUND_DOWN` constant.

Rules DoubleValue <b>round</b> (DoubleValue value)	
C1	RET1
true	
Boolean condition	
<b>Condition</b>	<b>Rate</b>
	=round(value,2,1)

Figure 140: Usage of the `ROUND_DOWN` constant

Testmethod <b>round</b> roundTest		
_description_	value	_res_
<b>Test ID</b>	<b>TestType</b>	<b>Test Result</b>
Test1	32.285	32.28
Test2	42.287	42.28
Test3	52.283	52.28
Test4	62.289	62.28

Figure 141: Test table for rounding to fractional number using the `ROUND_DOWN` constant

## Date Functions

OpenL Tablets supports a wide range of date functions that can be applied in the rule tables. The following date functions return an Integer data type:

Date functions used in OpenL Tablets that return an Integer data type	
Function	Description
<b>absMonth</b>	Returns the number of months since AD. <b>absMonth (Date)</b>
<b>absQuarter</b>	Returns the number of quarters since AD as an integer value. <b>absQuarter (Date)</b>
<b>dayOfWeek</b>	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, and so on. <b>dayOfWeek (Date d)</b>
<b>dayOfMonth</b>	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. <b>dayOfMonth (Date d)</b>
<b>dayOfYear</b>	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. <b>dayOfYear (Date d)</b>
<b>weekOfMonth</b>	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. <b>weekOfMonth (Date d)</b>

Date functions used in OpenL Tablets that return an Integer data type	
Function	Description
<b>weekOfYear</b>	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. <code>weekOfYear(Date d)</code>
<b>second</b>	Returns a second (0 to 59) for an input date. <code>second(Date d)</code>
<b>minute</b>	Returns a minute (0 to 59) for an input date. <code>minute(Date d)</code>
<b>hour</b>	Returns the hour of the day in 12 hour format for an input date. <code>hour(Date d)</code>
<b>hourOfDay</b>	Returns the hour of the day in 24 hour format for an input date. <code>hourOfDay(Date d)</code>

The following date function returns a String data type:

Date function used in OpenL Tablets that returns a String data type	
Function	Description
<b>amPm(Date d)</b>	Returns Am or Pm value for an input date. <code>amPm(Date d)</code>

The following figure displays values returned by date functions for a particular input date specified in the **MyDate** field.

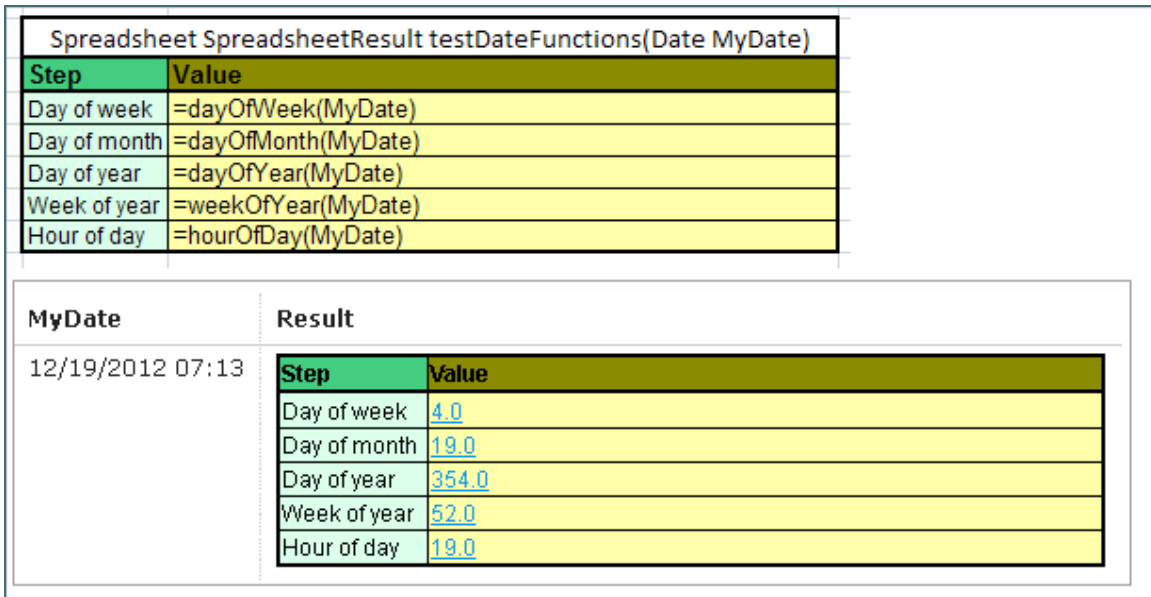


Figure 142: Date functions in OpenL Tablets

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

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

RiskFactor3Test <span>3 test cases</span>		
Date	Expected	Result
12/21/2012	<u>0.85</u>	✓ <u>0.85</u>
12/22/2012	<u>1</u>	✓ <u>1</u>
12/19/2012	<u>0.75</u>	✓ <u>0.75</u>

Figure 143: A risk factor depending on a day of the week

## Special Functions and Operators

OpenL Tablets supports a variety of different special functions and syntax to make rules creation easier and more convenient for business users.

The following topics are included in this section:

- [Error Function](#)
- [Performing Operations via Formula](#)

### Error Function

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

In the following example, if the value for a coverage limit of an insurance policy exceeds 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\$")

Figure 144: Usage of the ERROR function

### Performing Operations via Formula

A user can write several operations in a cell’s formula or in expression statement of the Decision table by separating each operation with the ‘;’ sign. The result of the last operation is defined as a returned value of the cell as follows:

```
`= Expression1; Expression2; ...; ResultedExpression
```

In practice, it is widely used when a user needs to store calculated values in the input object fields by using the following syntax:

```
`= field = value
```

or

```
`= field1 = value1; field2 = value2 ...; ResultedExpression
```

In the following example, the **Age** step calculates the age and stores the result in the **vehicleAge** field of the input object **vehicle**, the **Scoring** step calculates several scoring parameters, stores them in the **scoring** object, and returns the object with updated fields as a result of the step:

Spreadsheet SpreadsheetResult <b>DetermineVehiclePremium</b> (Vehicle vehicle)	
Step	Value
Age	= vehicleAge = CurrentYear() - year
TheftRating	= VehicleTheftRating (bodyType, price)
Scoring	= scoring.eligibility = "Eligible"; scoring.theftRating = \$TheftRating; scoring

### Null Elements Usage in Calculations

This section describes how null elements (an element with an empty value) are processed in calculations.

In some calculations, for example, **a+b** or **a\*b**, values **a** and **b** can be **null** elements. If one of the calculated values is **null**, it is recognized as **0** for sum operations or as **1** for multiply operations.

The following diagrams demonstrate this rule.

SmartRules Double <b>Operations</b> (String operationType, Double a, Double b)	
Operation	Result
SUBSTRACT	= a - b
ADD	= a + b
DIVIDE	= a / b
MULTIPY	= a * b
POW	= a ** b

Figure 145: Rules for null elements usage in calculations

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



Test Operations <b>OperationsTest</b>			
operationType	a	b	_res_
<b>Operation</b>	<b>A</b>	<b>B</b>	<b>Result</b>
SUBSTRUCT	5.0	3.0	2.0
SUBSTRUCT	5.0		5.0
SUBSTRUCT		3.0	-3.0
SUBSTRUCT			
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			
MULTIPY	8.0	4.0	32.0
MULTIPY	8.0		8.0
MULTIPY		4.0	4.0
MULTIPY			
POW	2.0	3.0	8.0
POW	2.0		1.0
POW		3.0	
POW			

Figure 146: Test table for null elements usage in calculations

If all values are **null**, the result is also **null**.

# 5 OpenL Tablets Business Expression Language

---

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

The following topics are included in this chapter:

- [Java Business Object Model as a Basis for OpenL Tablets Business Vocabulary](#)
- [New Keywords and Avoiding Possible Naming Conflicts](#)
- [Simplifying Expressions with Explanatory Variables](#)
- [Simplifying Expressions by the Using Unique in Scope Concept](#)
- [OpenL Tablets Programming Language Framework](#)

## 5.1 Java Business Object Model as a Basis for OpenL Tablets Business Vocabulary

OpenL Tablets minimizes the effort required to build a business vocabulary. Using BEX does not require any special mapping, and the existing Java BOM automatically becomes the basis for OpenL Tablets 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`

## 5.2 New Keywords and Avoiding Possible Naming Conflicts

In the previous chapter, a new **of the** keyword was introduced. 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

When adding new keywords to OpenL Tablets BEX language, 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 new keywords appear in the lower case. For example,

there is an attribute called `isLessThanCoverageLimit`. When referring to it as `is less than coverage limit`, there is going to be a name clash with the keyword, but if `Is Less Than Coverage Limit` is written, no clash appears. Possible direction in extending keywords is to add numeric, measurement units, measure sensitive comparisons, such as `is longer than` or `is colder than`, or use any other similar approach.

## 5.3 Simplifying Expressions with Explanatory Variables

Consider a rather simple expression in Java:

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

In BEX language, the same expression can be rewritten 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 Tablets BEX offers 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 resembles the one used in scientific publications and is easy to understand for anybody. It is believed that the syntax provides the best mix of mathematical clarity and business readability.

## 5.4 Simplifying Expressions by Using the Unique in Scope Concept

Humans differ from computers, in particular, by their ability to understand the scope of a language expression. For example, when discussing an insurance policy and the effective date is mentioned, there is no need to say the fully qualifying expression **the effective date of the policy** every time, because the context of the effective date is clearly understood. On the other hand, when discussing two policies, for example, the old and the new ones, one needs to say **the effective date of the new policy**, or **the effective date of the old policy**, to differentiate between two policies.

Similarly, when humans write so-called business documents, that is, files that serve as a reference point to a rule developer, they also often use an implied context in mind. Therefore in documentation, they often use business terms, such as **effective date**, **driver**, and **account**, with the implied scope in mind. Scope resolution is left to a so-called rules engineer, who has to do it by manually analyzing BOM and setting appropriate paths from root objects.

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

## 5.5 OpenL Tablets Programming Language Framework

Business rules consist of rules, where each rule has a condition and action. A condition is a Boolean expression, the one that returns `true` or `false`. An action can be any sequence, usually simple, of programming statements.

Consider an expression `driver.age < 25`.

From semantic perspective, the expression defines the relationship between a value defined by the `driver.age` expression and literal `25`. This can be interpreted as **age of the driver is less than 25 years** or **select drivers who are younger than 25 years old**, or any other similar phrase.

From the programming language perspective, the semantic part is irrelevant due to the following reasons:

- A statement must be valid in the language grammar.
- A statement must be correct from the type-checking point of view.
- If the language is compiled, the results of compiling, such as valid binary code, or bytecode, or code in another target language, can be considered as possible results of compiling and must be produced from the statement.
- A runtime system, interpreter, or virtual machine must be able to execute, or interpret, this statement's compiled code and produce a resulting object.

The following topics are included in this section:

- [OpenL Tablets Grammars](#)
- [Context, Variables and Types](#)
- [OpenL Tablets Type System](#)
- [OpenL Tablets as OpenL Tablets Type Extension](#)
- [Operators](#)
- [Binary Operators Semantic Map](#)
- [Unary Operators](#)
- [Cast Operators](#)
- [Strict Equality and Relation Operators](#)
- [List of org.openl.j Operators](#)
- [List of org.openl.j Operator Properties](#)

### OpenL Tablets Grammars

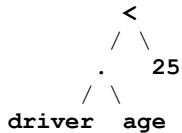
When the OpenL Tablets parser parses an OpenL Tablets expression, it produces a syntax tree. Each tree node has a node type, a literal value, a reference to the source code for displaying errors and debugging, and also may contain child nodes. This is similar to what other parsers do, with one notable exception – the OpenL Tablets Grammar is not hard-coded, it can be configured, and a different one can be used. For all practical purposes, as of today, only the following grammars implemented in OpenL Tablets are distributed:

OpenL Tablets grammar	
Grammar	Description
<code>org.openl.j</code>	Based on the classic Java 1.3 grammar. No templates and exception handling are supported.
<code>org.openl.bex</code>	<code>org.openl.j</code> grammar with business natural language extensions.

By default, `org.openl.bex` is used in the OpenL Tablets business rules product.

An experimental **org.openl.n3 grammar** is available, and **org.openl.sql** grammar is targeted to be added in the future.

The syntax tree produced by the **org.openl.j** grammar for the expression mentioned previously in this chapter is as follows:



Types of nodes are as follows:

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

Node type names are significant. More information on the type names is available further in this chapter.

The grammar used in `org.openl.j` is similar not only to Java but to any other language in the C/C++/Java/C# family. This makes OpenL Tablets easily to learn and apply by the huge pool of available Java/Cxx programmers and adds to its strength. Proliferation of new languages like Ruby and Groovy, multiple proprietary languages used in different business rules engines, CEP engines and so on, introduce new semantics to the programming community and make usage of new technologies much harder.

OpenL Tablets team makes their best to stay as close to the Java syntax as possible to make sure that the "entities would not be multiplied beyond necessity".

## Context, Variables and Types

After the syntax tree is created, syntax nodes must be bound to their semantic definitions. At this stage, OpenL Tablets uses specific binders for each node type. The modular structure of OpenL Tablets allows definition of custom binders for each node type. Once a syntax node is bound into the bound node, it is assigned a type, thus making the process type-safe.

Most of the time, the standard Java approach is used to assign type to the variable, so it must be defined in the context of the OpenL Tablets framework. Typical examples include the following components:

- method parameter
- local variable
- member of surrounding class
  - For OpenL Tablets, it is usually the implementation of `IOpenClass` called **module**.
- external types accessed as static, which are mostly Java classes imported into OpenL Tablets

Fields and methods used in binding context do not exist in Java; OpenL Tablets allows programmatically adding custom types, fields, and methods into binding context. For different examples of how it can be done, see the source code of the `OpenLBuilder` classes in different packages. For example, `org.openl.j` automatically imports all classes from the `java.util` in addition to the standard `java.lang` package. Since version 5.1.1, `java.math` is imported automatically.

## OpenL Tablets Type System

Java is a type-safe language. However, its type-safety ends when Java has to deal with types that lie outside of the Java type system, such as database tables, http requests, or XML files.

There are two approaches to deal with those external types:

Approaches to deal with types outside the Java type system	
Approach	Specifics
using API	<ul style="list-style-type: none"> <li>API approach is inherently not type-safe, it treats attribute as literal strings; therefore, even spelling errors become visible only in runtime.</li> <li>Unless the standard API exists, user's program becomes dependent on the particular API.</li> </ul>
using code generation	<ul style="list-style-type: none"> <li>Code generation requires 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.</li> <li>Often, generators introduce dependencies with runtime libraries that affect portability of the code.</li> <li>Generators usually require full conversion from external data into Java objects that may incur an unnecessary performance penalty when only a few attributes must be accessed.</li> </ul>

The **OpenL Tablets open type** system provides a simple way of adding new types into the OpenL Tablets language. It only requires defining a class object that implements the `OpenClass` interface and adding it to the OpenL Tablets type system. Implementations can vary, but access to object attributes and methods has the same syntax and provides the same type-checking in all OpenL Tablets code throughout the user application.

## OpenL Tablets as OpenL Tablets Type Extension

OpenL Tablets is built on top of the OpenL Tablets type system, thus enabling natural integration into any Java or OpenL Tablets environment. Using the OpenL Tablets methodology, **decision tables** become **methods**, and **data tables** become **fields**. Similar conversion happens to all project artifacts. As a result, any project component can be easily modularly accessed through Java or OpenL Tablets 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 methods with priorities defined by grammar. OpenL Tablets has two major types of operators, unary and binary. In addition, there are operator types used in special cases. A complete list of OpenL Tablets operators used in `org.openl.j` grammar is available at [List of org.openl.j Operators](#).

OpenL Tablets has a modular structure, so OpenL Tablets has configurable, high-level separate components like **parser** and **binder**, and each node type can have its own `NodeBinder`. At the same time, the single `NodeBinder` can be assigned to a group of operators, as in the case of the **op.binary** prefix.

**op.binary.or ||** and **op.binary.and &&** have separate `NodeBinders` to provide short-circuiting for boolean operands. For all other binary operators, OpenL Tablets 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 **add()** method named in the following order:

- **Tx add(T1 p1, T2 p2)** in the `org.openl.operators` namespace in `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 executed in runtime. So, to override binary operator **t1 OP t2**, where **t1**, **t2** are objects of classes **T1**, **T2**, perform the following steps:

1. Check operators and find the operator's type name.  
The last part of the type name is the name of the method to be implemented.
2. Use one of the following options available for implementing operators:
  - Put it into the `YourCustomOperators` class as a static method and register the class as a library in the `org.openl.operators` namespace  
For more information on how to do that, see `OpenLBuilder` code.
  - Implement **public Tx name(T2 p2)** as method in T1.
  - Implement **Tx name(T1 p1,T2 p2)** as method in T1.
  - Implement **static public Tx name(T1 p1,T2 p2)** as method in T2.
3. If **T1** and **T2** are different, define both **OP(T1, T2)** and **OP(T2, T1)**, unless **autocast()** operator can be relied on or binary operators semantic **map**. **Autocast** can help skipping implementation when there is already an operator implemented for the autocasted type.  
For example, when having **OP(T1, double)**, there is no need to implement **OP(T1, int)** because **int** is autocasted to **double**. Some performance penalty can be incurred by doing this though. For more information on binary operators semantic map, see [Binary Operators Semantic Map](#).

## Binary Operators Semantic Map

There is a convenient feature called *operator semantic map*. It makes implementing some of the operators easier by describing symmetrical and inverse properties for some operators as described in [List of opg.openl.j Operator Properties](#).

## Unary Operators

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

## Cast Operators

**Cast operators** in general correspond to Java guidelines and come in two types, **cast** and **autocast**. **T2 autocast (T1 from, T2 to)** methods are used to overload implicit cast operators, as 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 two parameters, only T1 from parameter is actually used. The second parameter is used to avoid ambiguity in Java method resolution.

## Strict Equality and Relation Operators

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

```
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)`.

## List of org.openl.j Operators

The `org.openl.j` operators in order of priority are as follows:

org.openl.j operators	
Operator	org.openl.j operator
<b>Assignment</b>	
=	op.assign
+=	op.assign.add
-=	op.assign.subtract
*=	op.assign.multiply
/=	op.assign.divide
%=	op.assign.rem
&=	op.assign.bitand
=	op.assign.bitor
^=	op.assign.bitxor
<b>Conditional Ternary</b>	
? :	op.ternary.qmark
<b>Implication</b>	
->	op.binary.impl <sup>(*)</sup>
<b>Boolean OR</b>	
or "or"	op.binary.or
<b>Boolean AND</b>	
&& or "and"	op.binary.and
<b>Bitwise OR</b>	
	op.binary.bitor
<b>Bitwise XOR</b>	
^	op.binary.bitxor
<b>Bitwise AND</b>	
&	op.binary.bitand
<b>Equality</b>	
==	op.binary.eq
!=	op.binary.ne
====	op.binary.strict_eq <sup>(*)</sup>
!==	op.binary.strict_ne <sup>(*)</sup>
<b>Relational</b>	
<	op.binary.lt
>	op.binary.gt
<=	op.binary.le
>=	op.binary.ge



org.openl.j operators	
Operator	org.openl.j operator
<==	op.binary.strict_lt <sup>(*)</sup>
>==	op.binary.strict_gt <sup>(*)</sup>
<===	op.binary.strict_le <sup>(*)</sup>
>===	op.binary.strict_ge <sup>(*)</sup>
Bitwise Shift	
<<	op.binary.lshift
>>	op.binary.rshift
>>>	op.binary.rshiftu
Additive	
+	op.binary.add
-	op.binary.subtract
Multiplicative	
*	op.binary.multiply
/	op.binary.divide
%	op.binary.rem
Power	
**	op.binary.pow <sup>(*)</sup>
Unary	
+	op.unary.positive
-	op.unary.negative
++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 <sup>(*)</sup>

**Note:** <sup>(\*)</sup> Operators do not exist in Java standard and exist only in org.openl.j. They can be used and overloaded if necessary.

## List of opg.openl.j Operator Properties

opg.openl.j operator properties	
Operator group	Operators
<b>Symmetrical</b>	eq(T1, T2) <=> eq(T2, T1) add(T1, T2) <=> add(T2, T1)
<b>Inverse</b>	le(T1, T2) <=> gt(T2, T1) lt(T1, T2) <=> ge(T2, T1)

---

$$\text{ge}(T1, T2) \Leftrightarrow \text{lt}(T2, T1)$$
$$\text{gt}(T1, T2) \Leftrightarrow \text{le}(T2, T1)$$

---

# 6 Working with Projects

---

This chapter describes creating an OpenL Tablets project. For more information on projects, see [Projects](#).

The following topics are included in this chapter:

- [Project Structure](#)
- [Rules Runtime Context](#)
- [Project and Module Dependencies](#)

## 6.1 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 Excel files which are physical storage of rules and data in the form of tables. On the logical structure level, Excel files represent modules of the project. Additionally, a project can contain `rules.xml`, Java classes, JAR files, according to developer's needs, and other related documents, such as guides and instructions.

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

**Note for experienced users:** The `rules.xml` project file is a rules project descriptor that contains project and configuration details. For instance, a user may redefine a module name there that is the same as a name of the corresponding Excel file by default. When updating project details via OpenL Tablets WebStudio, the `rules.xml` file is automatically created or updated accordingly. For more information on configuring `rules.xml`, see [\[OpenL Tablets Developer's Guide\]](#), *Rules Project Descriptor* section.

The following topics are included in this section:

- [Multi Module Project](#)
- [Creating a Project](#)
- [Project Sources](#)

### Multi Module Project

All modules inside one project have mutual access to each other's tables. It means that a rule or table of a module of a project is accessible and can be referenced and used from any rule of any module of the same project. Projects with several rule modules are called **multi module projects**.

To define compilation order of modules in a project, **module dependencies** are used. When a rule table must be run from another project, project dependencies must be used. For more information on using dependencies, see [Project and Module Dependencies](#).

### 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.

## Project Sources

Project sources can be added from developer created artifacts, such as jars and Java classes, which contain a reference to the folder with additional compiled classes to be imported by the module. For that, a rules project must contain the `rules.xml` file created in the project root folder.

Saved classpath is automatically added to the `rules.xml` file. After that, classpath can be used in rules. Classpath can indicate both specific jar and folder with libraries. The asterisk `*` symbol can be used for the varying part in the classpath.

```
<classpath>
  <entry path="."/>
  <entry path="auto-rating-model.jar"/>
  <entry path="lib/*.jar"/>
</classpath>
```

Figure 147: Classpath description in the `rules.xml`

To use a classpath in dependent projects, place a common classpath inside the main dependency project and then reuse it in all dependent projects.

## 6.2 Rules Runtime Context

OpenL Tablets supports rules overloading by metadata, or business dimension properties.

Sometimes a user needs business rules that work differently but have the same input.

Consider provided vehicle insurance and a premium calculation rule defined for it as follows:

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

For different US states, there are different bonus calculation policies. In a simple way, for all states there must be 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 a 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 a user has one common premium calculation rule and several different rules for bonus calculation. When running premium calculation rule, provide the current state as an additional input for OpenL Tablets to choose the appropriate rule. Using this information OpenL Tablets makes decision which bonus method must be invoked. This kind of information is called **runtime data** and must be set into runtime context before running the calculations.

The following OpenL Tablets table snippets illustrate this 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 148: The group of Decision Tables overloaded by properties

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

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

## Managing Rules Runtime Context from Rules

The following additional internal methods for modification, retrieving, and restoring runtime context support work with runtime context from OpenL Tablets rules:

Internal methods for work with runtime context	
Method	Description
getContext()	Returns a copy of the current runtime context.
<div style="border: 1px solid black; padding: 5px; margin: 10px 0;"> <pre>Method DoubleValue calcRateForDate (Policy policy, Date date) IRulesRuntimeContext context = getContext(); context.currentDate = date; setContext(context); return calcRate(policy);</pre> </div>	
emptyContext()	Returns new empty runtime context.
setContext(IRulesRuntimeContext context)	Replaces the current runtime context with the specified one.

Figure 56: Using the getContext function in a method

**Internal methods for work with runtime context**

Method	Description
--------	-------------

modifyContext(String propertyName, Object propertyValue)      Modifies the current context by one property: adds a new one or replaces by specified if property with such a name already exists in the current context.

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

Figure 149: Using modifyContext in a rules table

**Note:** All properties from the current context remain available after modification, so it is only one property update.

restoreContext()	Discharges the last changes in runtime context. The context is rolled back to the state before the last <b>setContext</b> or <b>modifyContext</b> .
------------------	---

```

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

Figure 150: Using restoreContext in a method table

**ATTENTION:** All changes and rollbacks must be controlled manually: all changes applied to runtime context will remain after rule execution. Make sure that the changed context is restored after the rule is executed to prevent unexpected behavior of rules caused by unrestored context.

**Note:** The `org.openl.rules.context` package must be imported as illustrated in the following figure so that a user can work with runtime context from rules:

```

Environment
import org.openl.rules.context
    
```

## 6.3 Project and Module Dependencies

**Dependencies** provide 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, if a user has several projects with different modules, all user projects share the same domain model or use similar helpers rules, and to avoid rules duplication, put the common rules and data to a separate module and add this module as dependency for all required modules.

Dependencies glossary	
Term	Description
Dependency module	Module that is used as a dependency.
Dependency project	Project that is used as a dependency.
Root module	Module that has dependency declaration, explicit via environment or implicit via project dependency, to other module.
Root project	Project that has dependency declaration to other project.

The following topics are included in this section:

- [Dependencies Description](#)
- [Dependencies Configuration](#)
- [Import Configuration](#)
- [Components Behavior](#)

### Dependencies Description

The **module dependency** feature allows making a hierarchy of modules when rules of one module depend on rules of another module. As 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. An exception is as follows.

The following diagram illustrates a project in which the content of **Module\_1** and **Module\_2** depends on the content of **Module\_3**, where thin black arrows are module dependencies:

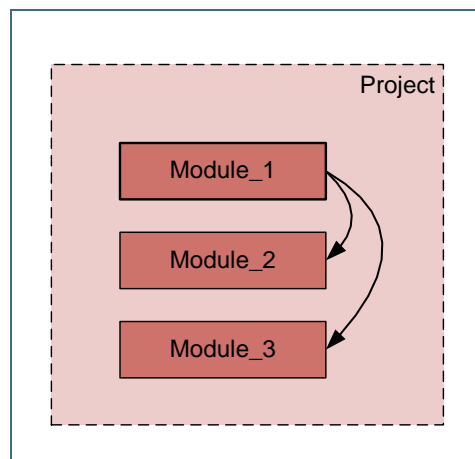


Figure 151: Example of a project with modules hierarchy

In addition, **project dependency** enables accessing modules of other projects from the current one:

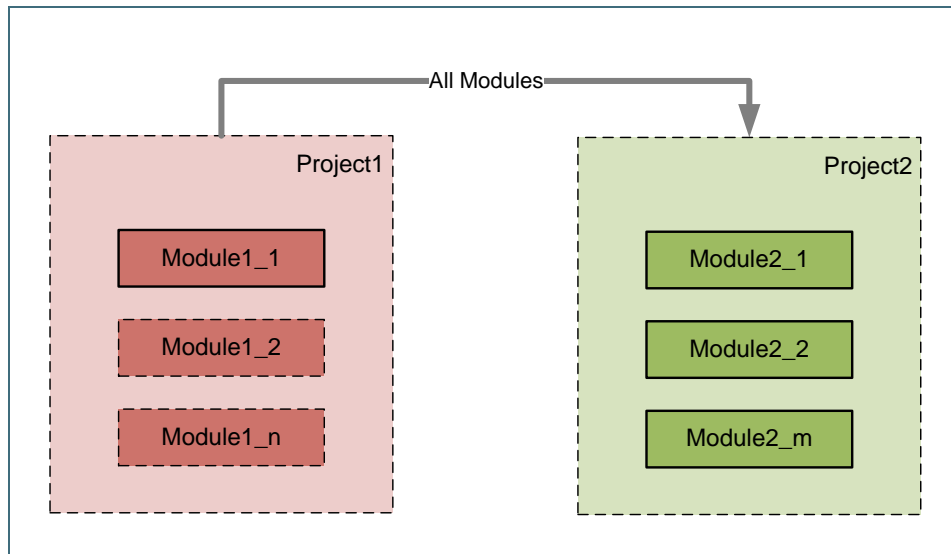


Figure 152: Example of a project dependency with all modules

The previous diagram displays that any module of **Project1** can execute any table of any module of **Project2**: thick gray arrow with the **All Modules** label 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**:

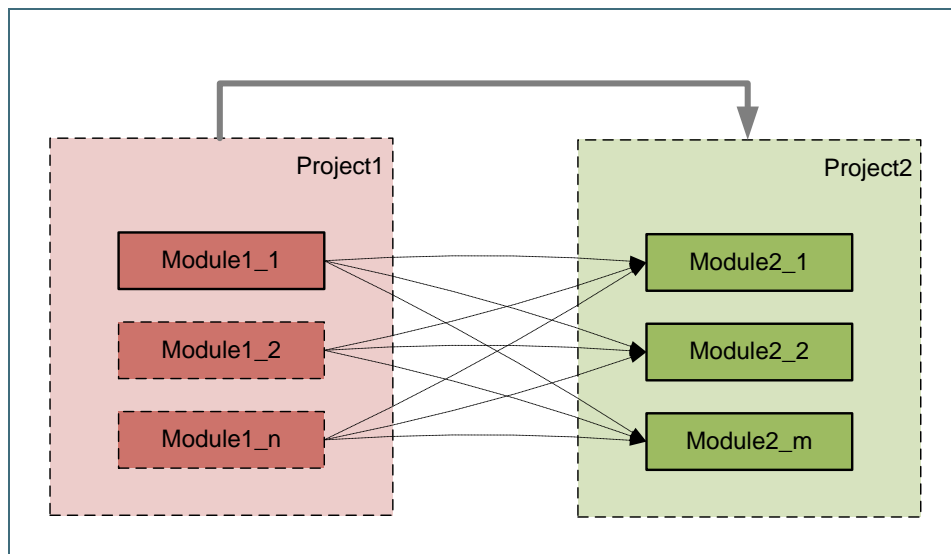


Figure 153: Interpretation of a project dependency (with all modules)

The project dependency with the **All Modules** setting switched on provides access to any module of a dependency project from the current root project.

Users may combine module and project dependencies if only a particular module of another project must be used. An example is as follows:



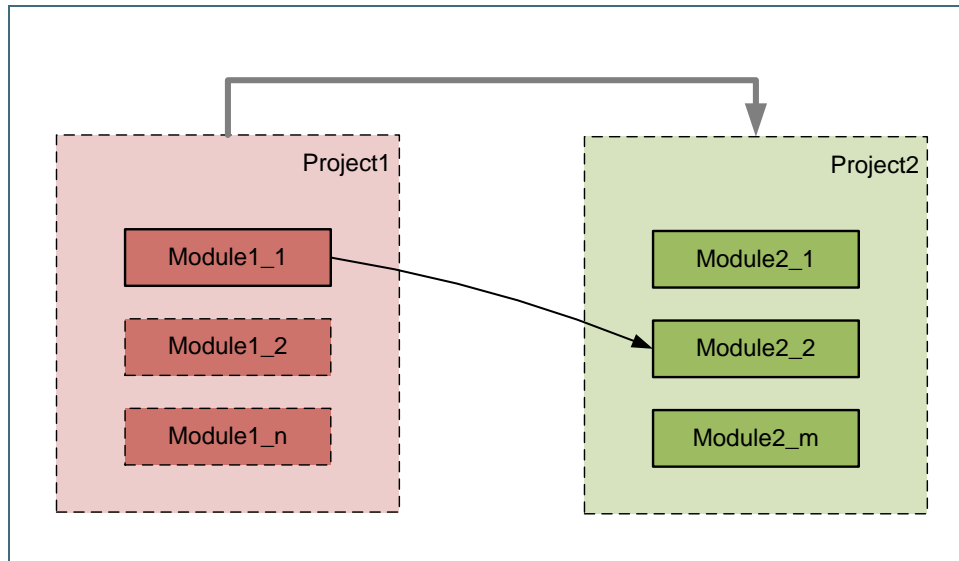


Figure 154: Example of a project and module dependencies combined

In the example, for defined external **Project2**, only the content of **Module2\_2** is accessible from **Project1**: thick gray arrow without label is a project dependency which defines other projects where dependency module can be located.

If the project dependency does not have the **All Modules** setting enabled, dependencies are determined on the module level, and such project dependencies serve the isolation purpose thus enabling getting a dependency module from particular external projects.

After adding a dependency, all its rules, data fields, and data types are accessible from the root module. The root module can call dependency rules.

## Dependencies Configuration

This section describes dependencies configuration.

1. To add a dependency to a module, add the instruction to a configuration table as described in [Configuration Table](#) using the **dependency** command and the name of the module to be added. A module can contain any number of dependencies. Dependency modules can also have dependencies. Avoid using cyclic dependencies.

Environment	
dependency	Module2_2

Figure 155: Example of configuring module dependencies

2. To configure a project dependency, in a rules project descriptor, in the `rules.xml` file created in the project root folder, in the **Dependency** section, for the **name** tag used for defining the dependency project name, set the **autoIncluded** tag to **true** or **false**.

```

<dependencies>
  <dependency>
    <name>Project Name</name>
    <autoIncluded>>false</autoIncluded>
  </dependency>
</dependencies>
<properties-file-name>processor</properties-file-name>

```

Figure 156: Example of configuring project dependencies – fragment of rules.xml

For more information on configuring `rules.xml`, see [\[OpenL Tablets Developer's Guide\]](#), *Rules Project Descriptor* section.

By a business user, project dependencies are easily set and updated in OpenL Tablets WebStudio as described in [\[OpenL Tablets WebStudio User Guide\]](#), *Defining Project Dependencies* section.

A project can contain any number of dependencies. Dependency projects may also have dependencies. Avoid cyclic dependencies. Module names of the root and dependency projects must be unique.

When OpenL Tablets is processing a module, if there is any dependency declaration, it is loaded and compiled before the root module. When all required dependencies are successfully compiled, OpenL Tablets compiles the root module with awareness about rules and data from dependencies.

## Import Configuration

Using import instructions allows adding external rules and data types from developer created artifacts, such as jars and Java classes, located outside the Excel based rule tables. In the import instruction, list all Java packages, Java classes, and libraries that must become accessible in the module.

Import configuration is defined using the **Environment** table as described in [Configuration Table](#). Configuration can be made for any user mode, single-user mode or multi-user mode. For proper import configuration, classpath must be registered in project sources as described in [Project Sources](#).

In the following example, the **Environment** table contains an import section with reference to the corresponding Java package:

Environment	
import	com.generated.rating.auto

Figure 157: Example of configuring module import

**Note:** For importing packages or classes, the same syntax is used. Firstly, OpenL Tablets tries to import the specified class. If it is not found, the system identifies it as a package and imports all classes from the specified package.

To import the library to the module, the following syntax is used:

```
org.packagename.ClassName.*
```

It adds all static methods from the corresponding class. A user can call these methods inside OpenL rules directly without indicating the class name. An example is using `rotate(str, shift)` instead of `StringUtils.rotate(str, shift)`.

Common Java imports can be placed only into the main, or dependency, project or module. When working with a dependent project, there is no need to specify **Import** in this project. Import data is retrieved directly from the dependency project. Dependency instruction makes all import instructions applied to the dependent module.

## Components Behavior

All OpenL Tablets components can be divided into three types:

- Rules in rule tables as described in [Decision Table](#), [Spreadsheet Table](#), [Method Table](#), [TBasic Table](#).
- Data in data tables as described in [Data table](#).
- Data types in data type tables as described in [Datatype Table](#).

The following table describes behavior of different OpenL Tablets components in dependency infrastructure:

OpenL Tablets components behavior in dependency infrastructure			
Operations or components	Rules	Datatypes	Data
Can access components in a root module from dependency.	Yes.	Yes.	Yes.
Both root and dependency modules contain a similar component.	<ol style="list-style-type: none"> <li>1. Rules with the same signature and without dimension properties: duplicate exception.</li> <li>2. Methods with the same signature and with a number of dimension properties: they are wrapped by Method Dispatcher. At runtime, a method that matches the runtime context properties is executed.</li> <li>3. Methods with the same signature and with property active: only one table can be set to true. Appropriate validation checks this case at compilation time.</li> </ol>	Duplicate exception.	Duplicate exception.
None of root and dependency modules contain the component.	<b>There is no such method</b> exception during compilation.	There is no such data type exception during compilation.	<b>There is no such field</b> exception during compilation.

# 7 Appendix A: BEX Language Overview

---

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

The following topics are included in this chapter:

- [Introduction to BEX](#)
- [Keywords](#)
- [Simplifying Expressions](#)
- [Operators Used in OpenL Tablets](#)

## 7.1 Introduction to BEX

BEX language provides 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, the `policy.effectiveDate` Java expression is equivalent to the **Effective Date of the Policy** BEX expression.

If the Java model correctly reflects business vocabulary, no further action is required. Otherwise, custom type-safe mapping or renaming can be applied.

## 7.2 Keywords

The following table represents BEX keyword equivalents to Java expressions:

BEX keywords equivalent to Java expressions	
Java expression	BEX equivalents
<code>==</code>	<ul style="list-style-type: none"> <li>• equals to</li> <li>• same as</li> </ul>
<code>!=</code>	<ul style="list-style-type: none"> <li>• does not equal to</li> <li>• different from</li> </ul>
<code>a.b</code>	b of the a
<code>&lt;</code>	is less than
<code>&gt;</code>	is more than
<code>&lt;=</code>	<ul style="list-style-type: none"> <li>• is less or equal</li> <li>• is in</li> </ul>
<code>!&gt;</code>	is no more than
<code>&gt;=</code>	is more or equal
<code>!&lt;</code>	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 to 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**.

## 7.3 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 that 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
```

The expression is hard to read. However, it becomes 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, a user can write **effective date** instead of **effective date of the policy**. BEX automatically determines 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.

## 7.4 Operators Used in OpenL Tablets

The full list of OpenL Tablets operators in order of priority is as follows:

Operator	Description
<b>Assignment and Compound assignment operators</b>	
=	Simple assignment operator. <code>C = A + B</code> will assign value of <code>A + B</code> into <code>C</code> . Chaining the assignment operator is possible in order to assign a single value to multiple variables: <code>C = A = B</code> .
+=	Add and assignment operator. It adds right operand to the left operand and assigns the result to left operand. <code>C += A</code> is equivalent to <code>C = C + A</code> .

Operator	Description
-=	Subtract and assignment operator. It subtracts right operand from the left operand and assigns the result to left operand. $C -= A$ is equivalent to $C = C - A$ .
*=	Multiply and assignment operator. It multiplies right operand with the left operand and assigns the result to left operand. $C *= A$ is equivalent to $C = C * A$ .
/=	Divide and assignment operator. It divides left operand with the right operand and assigns the result to left operand. $C /= A$ is equivalent to $C = C / A$ .
%=	Remainder and assignment operator. It divides and takes the remainder using two operands and assigns the result to left operand. $C %= A$ is equivalent to $C = C \% A$ .
&=	Bitwise AND and assignment operator. $C \&= 2$ is same as $C = C \& 2$ .
=	Bitwise OR and assignment operator. $C ^= 2$ is same as $C = C \^ 2$ .
^=	Bitwise XOR and assignment operator. $C  = 2$ is same as $C = C   2$ .
Conditional Operator	
? :	Conditional (ternary) operator that takes three operands and is used as shortcut for if-then-else statement). If <code>condition</code> is true, the operator returns the value of <code>valueIfTrue</code> expression; otherwise, it returns the value of <code>valueIfFalse</code> .
<code>condition ? valueIfTrue : valueIfFalse</code>	
Boolean OR	
or "or"	Logical OR operator. If any of the two operands are true, then the condition becomes true.
Boolean AND	
&& or "and"	Logical AND operator. If both the operands are true, then the condition becomes true.
Bitwise OR	
	Bitwise OR operator copies a bit if it exists in either operand.
Bitwise XOR	
^	Bitwise XOR operator copies the bit if it is set in one operand but not both.
Bitwise AND	
&	Bitwise AND operator copies a bit to the result if it exists in both operands.
Equality	
==	Equality operator checks if the values of two operands are equal or not, if yes then condition becomes true.
!=	Inequality operator checks if the values of two operands are equal or not, if values are not equal then condition becomes true.
====	Strict equality operator checks if the values of two operands are equal or not without considering inaccuracy of float point values. If values are equal strictly then condition becomes true.
!==	Strict Inequality operator checks if the values of two operands are equal or not without considering inaccuracy of float point values. If values are not equal strictly then condition becomes true.
Relational	
<	Less than operator checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.

Operator	Description
>	Greater than operator checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.
<=	Less than or equal to operator checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.
>=	Greater than or equal to operator checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.
<==	Strict less than operator checks if the value of left operand is less than the value of right operand without considering inaccuracy of float point values.
>==	Strict greater than operator checks if the value of left operand is greater than the value of right operand without considering inaccuracy of float point values.
<===	Strict less than or equal to operator checks if the value of left operand is less than or equal to the value of right operand without considering inaccuracy of float point values.
>===	Strict greater than or equal to operator checks if the value of left operand is greater than or equal to the value of right operand without considering inaccuracy of float point values.
<b>Additive</b>	
+	Addition operator adds values on either side of the operator.
-	Subtraction operator subtracts right-hand operand from left-hand operand.
<b>Multiplicative</b>	
*	Multiplication operator multiplies values on either side of the operator.
/	Division operator divides left-hand operand by right-hand operand.
%	Remainder operator divides left-hand operand by right-hand operand and returns remainder.
<b>Power</b>	
**	Exponentiation operator returns the result of raising left-hand operand to the power right-hand operand.
<b>Unary</b>	
+	Unary plus operator precedes its operand and indicates positive value.
-	Unary negation operator precedes its operand and negates it.
++	Increment operator increments its operand (increases the value of operand by 1) and returns a value. If used postfix, with operator after operand (for example, x++), then it returns the value before incrementing. For instance, x = 3; y = x++; gives y = 3, x = 4; If used prefix with operator before operand (for example, ++x), then it returns the value after incrementing. For instance, x = 3; y = ++x; gives y = 4, x = 4.
--	Decrement operator decrements its operand (decreases the value of operand by 1) and returns a value. If used postfix, with operator after operand (for example, x--), then it returns the value before decrementing. For instance, x = 3; y = x--; gives y = 3, x = 2; If used prefix with operator before operand (for example, --x), then it returns the value after decrementing. For instance, x = 3; y = --x; gives y = 2, x = 2.
!	Logical NOT operator reverses the logical state of its operand. If a condition is true then Logical NOT operator will make it false.
~	Bitwise NOT operator inverts the bits of its operand (performs the NOT operator on each bit).

Operator	Description
(Datatype) x	Cast operator converts the operand value $x$ to the specified <code>Datatype</code> type.
x	Absolute operator returns absolute value of $x$ without regard to its sign.



# 8 Appendix B: Functions Used in OpenL Tablets

This chapter provides a complete list of functions available in OpenL Tablets and includes the following sections:

- [Math Functions](#)
- [Array Functions](#)
- [Date Functions](#)
- [String Functions](#)
- [Special Functions](#)

## 8.1 Math Functions

Math functions	
Function	Description
<b>abs</b> (double a)	Returns the absolute value of a number.
<b>acos</b> (double a)	Returns the arc cosine of a value. The returned angle is in the range 0.0 through pi.
<b>asin</b> (double a)	Returns the arc sine of a value. The returned angle is in the range -pi/2 through pi/2.
<b>atan</b> (double a)	Returns the arc tangent of a value; the returned angle is in the range -pi/2 through pi/2.
<b>atan2</b> (double y, double x)	Returns the angle theta from the conversion of rectangular coordinates (x, y) to polar coordinates (r, theta).
<b>cbrt</b> (double a)	Returns the cube root of a double value.
<b>ceil</b> (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.
<b>copySign</b> (double magnitude, double sign) / (float magnitude, float sign)	Returns the first floating-point argument with the sign of the second floating-point argument.
<b>cos</b> (double a)	Returns the trigonometric cosine of an angle.
<b>cosh</b> (double x)	Returns the hyperbolic cosine of a double value.
<b>exp</b> (double a)	Returns Euler's number e raised to the power of a double value.
<b>expm1</b> (double x)	Returns $e^x - 1$ .
<b>floor</b> (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.
<b>format</b> (double d)	Formats double value.
<b>format</b> (double d, String fmt)	Formats double value according to Format fmt.
<b>getExponent</b> (double a)	Returns the unbiased exponent used in the representation of a.
<b>getExponent</b> (double x, double y)	Returns $\sqrt{x^2 + y^2}$ without intermediate overflow or underflow.
<b>IEEEremainder</b> (double f1, double f2)	Computes the remainder operation on two arguments as prescribed by the IEEE 754 standard.
<b>log</b> (double a)	Returns the natural logarithm (base e) of a double value.
<b>log10</b> (double a)	Returns the base 10 logarithm of a double value.

Math functions	
Function	Description
<b>log1p</b> (double x)	Returns the natural logarithm of the sum of the argument and 1.
<b>mod</b> (double number, double divisor)	Returns the remainder after a number is divided by a divisor.
<b>nextAfter</b> (double start, double direction) / (float start, float direction)	Returns the floating-point number adjacent to the first argument in the direction of the second argument.
<b>pow</b> (double a, double b)	Returns the value of the first argument raised to the power of the second argument.
<b>quotient</b> (double number, double divisor)	Returns the quotient from division number by divisor.
<b>random</b> ()	Returns a double value with a positive sign, greater than or equal to 0.0 and less than 1.0.
<b>rint</b> (double a)	Returns the double value that is closest in value to the argument and is equal to a mathematical integer.
<b>round</b> (double value)	Returns the closest value to the argument, with ties rounding up.
<b>round</b> (double value, int scale, int roundingMethod)	Returns a BigDecimal which scale is the specified value, and which unscaled value is determined by multiplying or dividing this BigDecimal's unscaled value by the appropriate power of ten to maintain its overall value.
<b>roundStrict</b> (double value)	Returns the closest value to the argument without adding ulp.
<b>scalb</b> (double a, int scaleFactor)	Return $a \times 2^{\text{scaleFactor}}$ rounded as if performed by a single correctly rounded floating-point multiply to a member of the double value set.
<b>signum</b> (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.
<b>sin</b> (double a)	Returns the trigonometric sine of an angle.
<b>sinh</b> (double x)	Returns the hyperbolic sine of a double value.
<b>sqrt</b> (double a)	Returns the correctly rounded positive square root of a double value.
<b>tan</b> (double a)	Returns the trigonometric tangent of an angle.
<b>tanh</b> (double x)	Returns the hyperbolic tangent of a double value.
<b>toDegrees</b> (double angrad)	Converts an angle measured in radians to an approximately equivalent angle measured in degrees.
<b>toRadians</b> (double angdeg)	Converts an angle measured in degrees to an approximately equivalent angle measured in radians.
<b>ulp</b> (double value)	Returns the size of an ulp of the argument.

## 8.2 Array Functions

Array functions	
Function	Description
<b>add</b> (array[],element)	Copies the given array and adds the given element at the end of the new array.
<b>add</b> (array[],index, element)	Inserts the specified element at the specified position in the array.

Array functions	
Function	Description
<b>addAll</b> (array1[], array2[])	Adds all elements of the given arrays into a new array.
<b>addIgnoreNull</b> (array[], element)	Copies the given array and adds the given element at the end of the new array.
<b>addIgnoreNull</b> (array[], int index, element)	Inserts the specified element at the specified position in the array.
<b>allFalse</b> (Boolean[] values)	Returns true if all array elements are false.
<b>anyFalse</b> (Boolean[] values)	Returns true if any array element is false.
<b>allTrue</b> (Boolean[] values)	Returns true if all array elements are true.
<b>anyTrue</b> (Boolean[] values)	Returns true if any array element is true.
<b>avg</b> (array[])	Returns the arithmetic average of the array of number elements.
<b>big</b> (array[], int position)	Removes null values from array, sorts an array in descending order and returns the value at position ' <i>position</i> '.
<b>concatenate</b> (array)	Joins several values in one text string.
<b>contains</b> (array, elem)	Checks if the value is in the given array.
<b>indexOf</b> (array[], elem)	Finds the index of the given value in the array.
<b>intersection</b> (String[] array1, String[] array2)	Returns a new array containing elements common to the two arrays.
<b>isEmpty</b> (array[])	Checks if an array is empty or null.
<b>flatten</b> (arrayN)	Returns a flatten array with values from arrayN. Returns a single dimension array of elements.
<b>length</b> (array)	Returns the number of elements in the array.
<b>max</b> (array[])	Returns the maximal value in the array of numbers.
<b>min</b> (array[])	Returns the minimal value in the array of numbers.
<b>noNulls</b> (array[])	Checks if the array is non-empty and has only non-empty elements.
<b>product</b> (array [] values)	Multiplies the numbers from the provided array and returns the product as a number.
<b>remove</b> (array [], int index)	Removes the element at the specified position from the specified array.
<b>removeElement</b> (array [], element)	Removes the first occurrence of the specified element from the specified array.
<b>removeNulls</b> (T[] array)	Returns a new array without null elements.
<b>slice</b> (array, int startIndexInclusive, int endIndexExclusive)	Returns a part of the array from startIndexInclusive to endIndexExclusive.
<b>small</b> (Array[], int position)	Removes null values from array, sorts an array in ascending order and returns the value at position ' <i>position</i> '.
<b>sort</b> (Array[])	Sorts the specified array of values into ascending order, according to the natural ordering of its elements.
<b>sum</b> (array[])	Returns the sum of numbers in the array.

## 8.3 Date Functions

Date functions	
Function	Description
<b>absMonth</b> (Date dt)	Returns the number of months since AD.
<b>absQuarter</b> (Date dt)	Returns the number of quarters since AD as an integer value.
<b>amPm</b> (Date dt)	Returns <i>Am</i> or <i>Pm</i> value for an input Date as a String.
<b>dateToString</b> (Date dt)	Converts a date to the String.
<b>dateToString</b> (Date dt, String dateFormat)	Converts a date to the String according dateFormat.
<b>dayDiff</b> (Date dt1, Date dt2)	Returns the difference in days between endDate and startDate.
<b>dayOfMonth</b> (Date dt)	Returns the day of month.
<b>dayOfWeek</b> (Date dt)	Returns the day of week.
<b>dayOfYear</b> (Date dt)	Returns the day of year.
<b>firstDateOfQuarter</b> (int absQuarter)	Returns the first date of quarter.
<b>hour</b> (Date dt)	Returns the hour.
<b>hourOfDay</b> (Date dt)	Returns the hour of day.
<b>lastDateOfQuarter</b> (int absQuarter)	Returns the last date of the quarter.
<b>lastDayOfMonth</b> (Date dt)	Returns the last date of the month.
<b>minute</b> (Date dt)	Returns the minute.
<b>month</b> (Date dt)	Returns the month (0 to 11) of an input date.
<b>monthDiff</b> (Date dt1, Date dt2)	Return the difference in months before d1 and d2.
<b>quarter</b> (Date dt)	Returns the quarter (0 to 3) of an input date.
<b>second</b> (Date dt)	Returns the second of an input date.
<b>toString</b> (Date dt)	Converts a date to the string in MM/dd/yyyy format.
<b>weekDiff</b> (Date dt1, Date dt2)	Returns the difference in weeks between endDate and startDate.
<b>weekOfMonth</b> (Date dt)	Returns the week of the month within which that date is.
<b>weekOfYear</b> (Date dt)	Returns the week of the year on which that date falls.
<b>yearDiff</b> (Date dt1, Date dt2)	Returns the difference in years between endDate and startDate.
<b>year</b> (Date dt)	Returns the year (0 to 59) for an input Date.

## 8.4 String Functions

String functions		
Function	Description	Comment
<b>concatenate</b> (str1, str2,...)	Joins several values in one text string.	
<b>contains</b> (String str, char searchChar)	Checks if String contains a search character, handling null.	

String functions		
Function	Description	Comment
<b>contains</b> (String str, String searchStr)	Checks if String contains a search String, handling null.	
<b>containsAny</b> (String str, char[] chars)	Checks if the String contains any character in the given set of characters.	
<b>containsAny</b> (String str, String searchChars)	Checks if the String contains any character in the given set of characters.	
<b>endsWith</b> (String str, String suffix)	Check if a String ends with a specified suffix.	
<b>isEmpty</b> (String str)	Checks if a String is empty ("") or null.	
<b>length</b> (String str)	Returns the number of characters in a string.	
<b>lowerCase</b> (String str)	Converts a String to lower case.	
<b>removeEnd</b> (String str, String remove)	Removes a substring only if it is at the end of a source string, otherwise returns the source string.	
<b>removeStart</b> (String str, String remove)	Removes a substring only if it is at the beginning of a source string, otherwise returns the source string.	
<b>replace</b> (String str, String searchString, String replacement)	Replaces all occurrences of a String within another String.	
<b>replace</b> (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.	
<b>startsWith</b> (String str, String prefix)	Check if a String starts with a specified prefix.	
<b>substring</b> (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.
<b>substring</b> (String str, int beginIndex, int endIndex)	Gets a substring from the specified String.	A negative start position can be used to start or end n characters from the end of the String.
<b>trim</b> (String str)	Removes whitespace characters from both ends of a string.	
<b>upperCase</b> (String str)	Converts a String to upper case.	

## 8.5 Special Functions

Special functions		
Function	Description	Comment
<b>error</b> (String "msg") <b>error</b> (String "msg");null	Shows the error message.	
<b>fieldName</b> (object)	Returns a value from the specified field of the object	<code>age(driver)</code> is the same as <code>driver.age</code> ; <code>age(drivers)</code> returns the ages list of all drivers.
<b>getValues</b> (MyAliasDatatype)	Returns array of values from the MyAliasDatatype alias data type.	Returns MyAliasDatatype[] .
<b>instanceOf</b> (Object, className.class)	Returns Boolean value defining if the Object is of the specified class. This function is deprecated.	
<b>toString</b> (value)	Converts value to the string.	

# 9 Index

---

## A

aggregated object  
definition, 63  
specifying data, 63

## array

definition, 87  
elements, 87  
index operators, 88  
working from rules, 87

## B

BEX language, 125  
explanatory variables, 126  
introduction, 125  
keywords, 125  
simplifying expressions, 126  
unique scope, 126

## Boolean values

representing, 55

## C

## calculations

using in table cells, 56

## column match table

definition, 81

## configuration table

definition, 71

## D

data integrity, 65

## data table

advanced, 62  
definition, 61  
simple, 61

## data type table

definition, 58

data types, 92

## date values

representing, 55

## decision table

definition, 40  
interpretation, 44, 48  
structure, 41  
transposed, 53

## E

examples, 10, 12

## F

functions in rules, 97

## G

## guide

audience, 6  
related information, 6  
typographic conventions, 6

## M

## method table

definition, 71

## O

## OpenL Tablets

advantages, 8  
basic concepts, 8  
creating a project, 116  
definition, 8  
introduction, 8  
project, 9  
rules, 9  
tables, 9

OpenL Tablets, 13

## OpenL Tablets project

definition, 9

## P

## project

creating, 116  
definition, 9  
modifying contents, 71  
structure, 116

## properties table

definition, 74

## R

## rule

definition, 9

## run table

- definition, 70
- structure, 71

**S**

- spreadsheet table
  - definition, 74
- system overview, 9

**T**

- table cells

- using calculations, 56
- Table Part functionality, 84
- TBasic table
  - definition, 84
- test table
  - definition, 66
  - structure, 66
- tutorials, 10