OpenL Tablets Reference Guide

OpenL Tablets® 5.7.4

OpenL Tablets Rules Engine

Document number: TP_OpenL_Ref_2.4_LSh
Revised: 01-18-2011

Table of Contents

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]

Audience

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

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

Related Information

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

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

Typographic Conventions

The following styles and conventions are used in this guide:

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

Typographic styles and conventions	
Convention	**Description**
Select **File > Save As**	Represents a command to perform, such as opening the **File** menu and selecting **Save As.**
Italic	• Represents any information to be entered in a field. • Represents documentation titles.
< >	Represents placeholder values to be substituted with user specific values.
Hyperlink	Represents a hyperlink. Clicking a hyperlink displays the information topic or external source.

Chapter 1: Introducing OpenL Tablets

This section introduces OpenL Tablets and describes its main concepts.

The following topics are included in this section:

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

What Is OpenL Tablets?

OpenL Tablets is a Business Rules Management System (BRMS) and Business Rules Engine (BRE) based on tables presented in Excel and Word 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 and Word documents and makes them accessible from Java programs.

OpenL Tablets is built using the OpenL technology providing a framework for development of different language configurations.

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 Java developers.

  For example, decision tables are transformed into Java methods, and data tables become accessible as Java data arrays through the familiar getter and setter JavaBeans mechanism. The transformation is performed automatically.
- 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 or Word document.
- OpenL Tablets provides calculation explanation capabilities, enabling expansion of any calculation result by pointing to source arguments in the original documents.
- OpenL Tablets provides cross-indexing and search capabilities within all project documents.

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

Basic Concepts

This section describes the following main OpenL Tablets concepts:

Rules

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

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

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

Tables

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

Projects

An **OpenL Tablets project** is a container of all resources required for processing rule related information. Usually, a project contains Excel or Word files and optionally Java code, library dependencies, Ant task for generating wrappers of table files. For detailed information on projects, see Chapter 3: 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.

Wrappers

A **wrapper** is a Java class that exposes decision tables as Java methods, data tables as Java objects and allows developers to access table information from code. To access a particular table from Java code, a wrapper Java class must be generated for the Excel or Word file where the table is defined. Wrappers are essential for solutions where compiled OpenL Tablets project code is embedded in solution applications. If tables are accessed through web services, client applications are not aware of wrappers but they may be still used on the server.

Wrappers can be dynamic or static as described in the following table:

Wrapper types	
Type	**Description**
Dynamic	For a dynamic wrapper, only a rule interface must be defined upon project creation. The rules run-time factory provides instances implementing this interface in run-time. Using dynamic wrappers, and not the static wrappers, is recommended.

Wrapper types	
Type	**Description**
Static	The wrapper Java class is generated in a rule project for a static wrapper, which contains all OpenL Tablets API usage logic to call rules.

Using dynamic wrappers, rather than static wrappers, is more advantageous as it enables OpenL Tablets users to clearly define the rules displayed in the application. Using a static wrapper can be inconvenient in that a wrapper must be regenerated each time a new version of OpenL Tablets is released.

OpenL Tablets provides a specific Ant task that can be used for static generation of a wrapper from any Excel or Word file automatically.

A static wrapper class must be regenerated in the following situations:

- A table signature, such as method name, input parameters, and return values, is modified.
- A table is added or deleted in the corresponding file.

Wrapper classes do not have to be regenerated if table data is modified or if conditions and actions are added or removed.

For more information on generating wrappers, see Generating a Wrapper.

System Overview

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

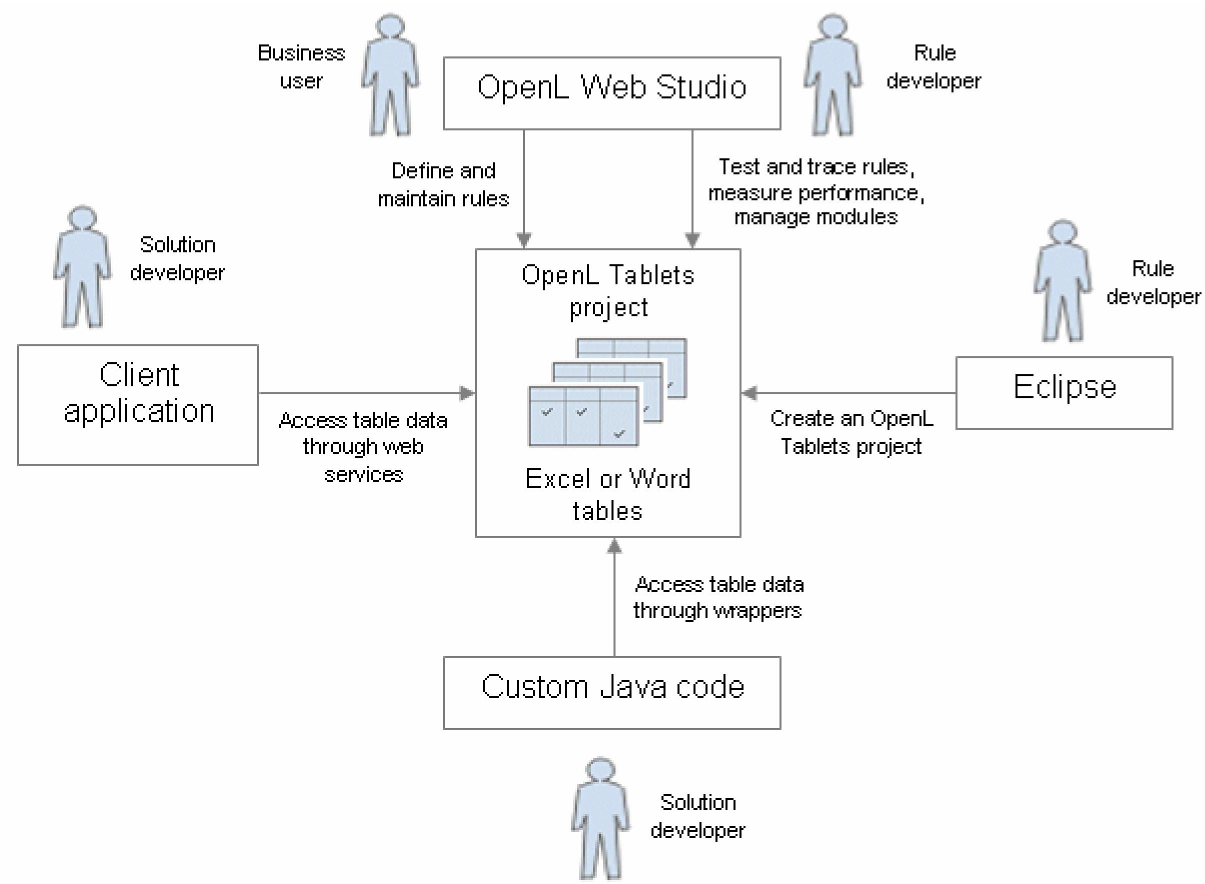


Figure 1: OpenL Tablets overview

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

A rule developer creates a new OpenL Tablets project in Eclipse.

In addition to the project itself, the rule developer also creates correctly structured tables in Excel or Word files based on requirements and includes them in the project.

A business user accesses tables in the OpenL Tablets project and defines rules.

Typically, this task is performed through OpenL Web Studio in a web browser.

A rule developer performs unit tests and performance tests on rules through advanced OpenL Web Studio features.

A developer who creates other parts of the solution employs business rules directly through the OpenL Tablets engine or remotely through web services.

Whenever required, the business user updates or adds new rules to project tables.

Installing OpenL Tablets

OpenL Tablets development environment is installed as an Eclipse update site. The installation process of the OpenL Tablets feature is the same as for any other Eclipse feature.

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

Tutorials and Examples

The OpenL Tablets Eclipse feature contains several preconfigured projects intended 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 ten tutorial projects demonstrating basic OpenL Tablets features beginning very simply and moving on to more advanced projects. Files in the tutorial projects contain detailed comments allowing new users to grasp basic concepts quickly.

To create a tutorial project, proceed as follows:

1. In Eclipse, select **File > New > Project.**

In the new project wizard, expand the **OpenL Tablets > Tutorials** folder.

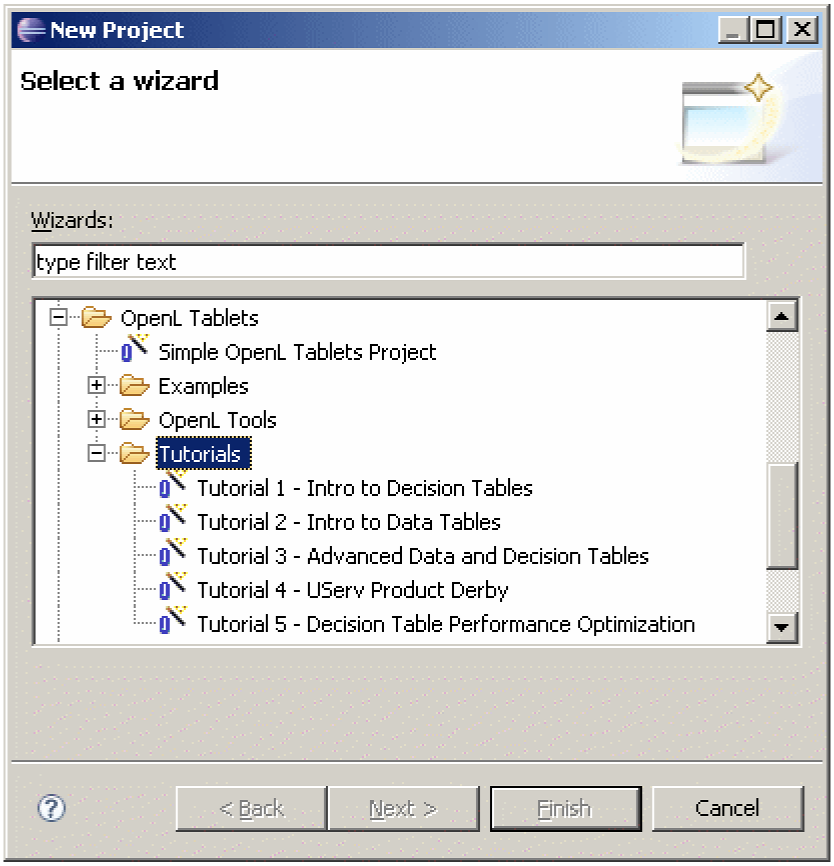


Figure 2: Creating tutorial projects

Select an appropriate tutorial project, and click **Next.**

In the next page, click **Finish.**

Examples

OpenL Tablets provides five example projects that demonstrate how OpenL Tablets can be used in various business domains.

To create an example project, proceed as follows:

1. In Eclipse, select **File > New > Project.**

In the new project wizard, expand the **OpenL Tablets > Examples** folder.

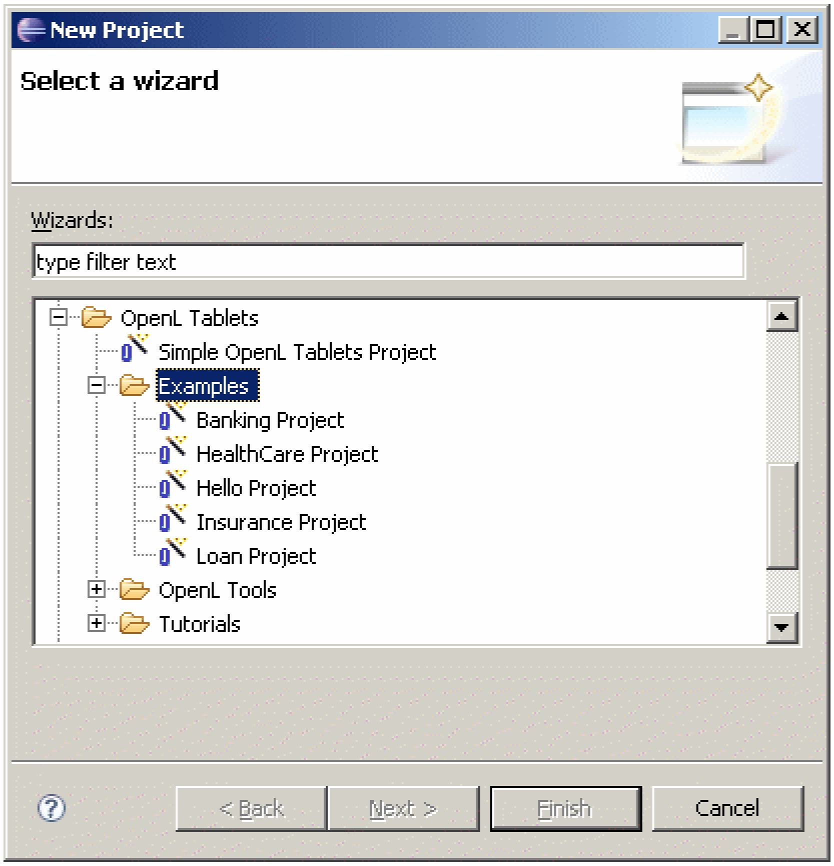


Figure 3: Creating example projects

Select an appropriate example project and click **Next.**

In the next page, click **Finish.**

Chapter 2: Creating Tables for OpenL Tablets

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

The following topics are included in this section:

- Table Recognition Algorithm
- Table Properties
- Table Types

Table Recognition Algorithm

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

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

The following is the general table recognition algorithm:

1. The engine looks into each spreadsheet and tries to identify logical tables.

   Logical tables must be separated by at least one empty row or column or start at the very first row or column. Table parsing is performed from left to right and from top to bottom. The first populated cell that does not belong to a previously parsed table becomes the top-left corner of a new logical table.

The engine reads text in the top left cell of a recognized logical table to determine its type.

   If the top left cell of a table starts with a predefined keyword then such table is recognized as an OpenL Tablets table.

   The following are the supported keywords:

Table type keywords	
Keyword	**Table type**
`Rules` or `DT`	Decision Table
`Data`	Data Table
`Datatype`	Data Type Table
`Testmethod`	Test Table
`Runmethod`	Run Method Table
`Method` or `Code`	Method Table
`Environment`	Configuration Table
`Properties`	Properties Table
`Spreadsheet`	Spreadsheet Table

Table type keywords	
Keyword	**Table type**
`ColumnMatch`	Column Match Table
`TBasic`	TBasic Table
`SimpleRules`	SimleRules Table
`SimpleLookup`	SimleLookup Table

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 or Word files.

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

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

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

Table Properties

For all table types, except for Properties Table, Configuration Table and **other** tables (that are not OpenL tables), properties can be defined containing information about the table. The properties list 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.

Properties are defined in the section of the table which goes in the next row (usually the second one) after the table **header** and before other table contents. The properties section is optional and can miss in the table. The first cell in the properties row contains keyword "**properties**" and is merged across all cells in column if more than 1 property is defined. The number of rows in properties section is equal to number of properties defined for the table. Each row in properties section contains pair property name and its value in consecutive cells (2nd and 3rd columns).

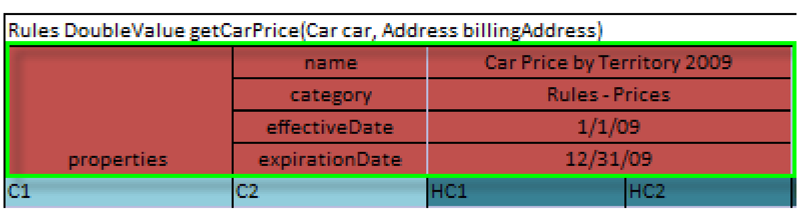


Figure 4: Table properties example

Category and Module Level Properties

Table properties can be defined not only for each table separately, but for all tables in some category or whole module. Separate Properties Table is created to define such properties. Only properties allowed to be inherited from category and/or module level can be defined in this table. Some properties can be defined only inside table, e.g. "name."

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

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

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

Default Value

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

There is no property value definition inside rules file for the default value.

System Properties

Some properties can have system value which is set by OpenL tool or engine. Currently there are no properties set by Engine. There are properties updated by OpenL Web Studio (refer to *OpenL Web Studio User's Guide*).

Properties for Particular Table Type

There are properties defined to be used just for particular types of tables. It means that they make sense just for tables with special type and can be defined only for them.

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

Table Versioning

Rules can be versioned in 2 different ways using Table properties. The most useful versioning is using Business Dimension properties.

Business Dimension Properties

OpenL rules can be versioned using Table properties. For all rules table types (Rules, Spreadsheet, TBasic, Method, ColumnMatch, SimpleRules and SimpleLookup) properties in category “Business Dimension” are used to version rules by property values. If there is any rules which should differ depending on the date or state (for example), it can be duplicated and for each table different property values should be set. OpenL Tablets Engine will choose appropriate rule during execution by runtime context values.

Rules versioned by Business Dimension properties can be of different table types.

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

Active Table

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

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

An example of an inactive table version follows:

Rules DoubleValue driverRiskScoreOverloadTest(String driverRisk)		
	name	Driver Risk Score Table
	category	Driver-Scoring
	version	0.0.1
properties	active	FALSE
C1		RET1
risk == driverRisk		score
String risk		DoubleValue score
Driver Risk		Score
High Risk Driver		100
		0

Figure 5: An inactive table version

Table Types

OpenL Tablets employs the following table types:

- Decision Table
- Data Type Table
- Data Table
- Test Table
- Run Method Table
- Method Table
- Configuration Table
- Properties Table
- Spreadsheet Table
- Column Match Table
- Tbasic Table

Decision Table

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

The following topics are included in this section:

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

Decision Table Structure

The following is an example of a decision table:

1	Rules void hello1(int hour)			
2	properties	name		Day Hour Classification
3		category		Day and Time
4	**Rule**	**C1**		**A1**
5		min <= hour && hour <= max		System.out.println(greeting + ", World!")
6		int min	int max	String greeting
7	**Rule**	**From**	**To**	**Greeting**
8	R10	0	11	Good Morning
9	R20	12	17	Good Afternoon
10	R30	18	21	Good Evening
11	R40	22	23	Good Night

Figure 6: Decision table

The following table describes its structure:

<table>
<tr><th colspan="3">Decision table structure</th></tr>
<tr><th>Row number</th><th>Mandatory</th><th>Description</th></tr>
<tr><td>1</td><td>Yes</td><td>Table header, which has the following pattern:
<code><keyword> <method header></code>
where <code><keyword></code> is either 'Rules' or 'DT' and <code><method header></code> is a signature of a method used to access the decision table and provide input parameters.
For example, the table in the preceding diagram can be invoked as follows:
<code>HelloWrapper tableWrapper = new HelloWrapper();</code>
<code>tableWrapper.hello1(15);</code>
where <code>HelloWrapper</code> is the wrapper class of the Excel or Word file. For general information on wrappers, see Wrappers.
Normally, this row is hidden to business users.</td></tr>
<tr><td>2 and 3</td><td>No</td><td>Rows containing table properties. Each application using OpenL Tablets rules can utilize properties for different purposes.
For example, a predefined table property 'name' is used to specify business user oriented name of the table displayed in OpenL Web Studio.
Although the provided decision table example contains two property rows, there can be any number of property rows in a table.
Normally, property rows are hidden to business users.</td></tr>
<tr><td>4</td><td>Yes</td><td>Row consisting of the following cell types:
<table>
<tr><th>Type</th><th>Description</th><th>Examples</th></tr>
<tr><td>Condition column header</td><td>Identifies that the column contains rule condition and its parameters. It must start with character 'C' followed by a number.</td><td><code>C1, C5, C8</code></td></tr>
<tr><td>Horizontal condition column header</td><td>Identifies that the column contains horizontal rule condition and its parameters. It must start with character 'HC' followed by a number.
Horizontal conditions are used in lookup tables only.</td><td><code>HC1, HC5, HC8</code></td></tr>
<tr><td>Action column header</td><td>Identifies that the column contains rule actions. It must start with character 'A' followed by a number.</td><td><code>A1, A2, A5</code></td></tr>
<tr><td>Return value column header</td><td>Identifies that the column contains values to be returned to the calling program. A table can have multiple return columns, however, only the first fired not blank value is returned.</td><td><code>RET1</code></td></tr>
</table>
All other cells in this row are ignored and can be used as comments.
If a table contains action columns, the engine executes actions for all rules whose conditions are true. If a table has a return column, the engine stops processing rules after the first executed rule. If a return column has a blank cell and the rule is executed, the engine does not stop but continues checking rules in the table.
Normally, this row is hidden to business users.</td></tr>
</table>

<table>
<tr><th colspan="3">Decision table structure</th></tr>
<tr><th>Row number</th><th>Mandatory</th><th>Description</th></tr>
<tr><td>5</td><td>Yes</td><td>Row containing cells with code statements for condition, action, and return value column headers. OpenL Tablets supports Java grammar enhanced with OpenL Business Expression (BEX) grammar features. For information on the BEX language, see Appendix A: BEX Language Overview.

Code in these cells can use any Java objects and methods visible to the OpenL Tablets engine. For information on enabling the OpenL Tablets engine to use custom Java packages, see Configuration Table.

Purpose of each cell in this row depends on the cell above it as follows:
<table>
<tr><th>Cell above</th><th>Purpose</th></tr>
<tr><td>Condition column header</td><td>Specifies the logical expression of the condition. It can reference parameters in the method header and parameters in cells below.

The cell can contain several expressions, but the last expression must return a Boolean value. All condition expressions must be true to execute a rule.</td></tr>
<tr><td>Horizontal condition</td><td>The same to Condition column header.</td></tr>
<tr><td>Action column header</td><td>Specifies code to be executed if all conditions of the rule are true. The code can reference parameters in the method header and parameters in cells below.</td></tr>
<tr><td>Return value column header</td><td>Specifies expression used for calculating the return value. The type of the last expression must match the return value specified in the method header. The explicit return statement is also supported.

This cell can reference parameters in the method header and parameters in cells below.</td></tr>
</table>
Normally, this row is hidden to business users.</td></tr>
<tr><td>6</td><td>Yes</td><td>Row containing parameter definition cells. Each cell in this row specifies the type and name of parameters in cells below it.

Parameter name must be one word corresponding to Java identification rules.

Parameter type must be one of the following:
• primitive Java types
• Java classes visible to the engine
• one-dimensional arrays of the above types as described in Working with Arrays
• data tables or their attributes as described in Using Advanced Data Tables
Normally, this row is hidden to business users.</td></tr>
<tr><td>7</td><td>Yes</td><td>Descriptive column titles. The rule engine does not use them in calculations but they are intended for business users working with the table. Cells in this row can contain any arbitrary text and be of any layout that does not correspond to other table parts. The height of the row is determined by the first cell in the row.</td></tr>
<tr><td>8 and below</td><td>Yes</td><td>Concrete parameter values.</td></tr>
</table>

Lookup Tables

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

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

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

Lookup table must have:

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

Lookup table can have:

- Rule column

Lookup table cannot have comment column!

Advanced usage:

- Lookup table (in theory) might have vertical Actions which will be processed the same way as vertical conditions.

Rules DoubleValue getCarPrice(Car car, Address billingAddress)				
properties	name	Car Price by Territory		
C1	C2	HC1	HC2	RET1
country	region	brand	model	price
Country countryName	String regionName	CarBrand carBrand	String carModel	DoubleValue price
Country	Region	BMW		
		Z4 sDrive35i	Z4 sDrive30i	
USA	Pacific West	$50 650	$44 750	
	West	$51 000	$43 050	
	Mid Atlantic	$51 450	$45 550	
GreatBritain	England	$52 650	$46 750	
	Wales	$52 650	$46 750	
	Scotland	$52 650	$46 750	

Figure 7: A lookup table example

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

Rules DoubleValue getCarPrice(Car car, Address billingAddress)				
properties	name	Car Price by Territory		
C1	C2	C3	C4	RET1
country	region	brand	model	price
Country countryName	String regionName	CarBrand carBrand	String carModel	DoubleValue price
Country	Region	Brand	Model	Price
USA	Pacific West	BMW	Z4 sDrive35i	$50 650
USA	West	BMW	Z4 sDrive35i	$51 000
USA	Mid Atlantic	BMW	Z4 sDrive35i	$51 450
GreatBritain	England	BMW	Z4 sDrive35i	$52 650
GreatBritain	Wales	BMW	Z4 sDrive35i	$52 650
GreatBritain	Scotland	BMW	Z4 sDrive35i	$52 650
USA	Pacific West	BMW	Z4 sDrive30i	$44 750
USA	West	BMW	Z4 sDrive30i	$43 050
USA	Mid Atlantic	BMW	Z4 sDrive30i	$45 550
GreatBritain	England	BMW	Z4 sDrive30i	$46 750
GreatBritain	Wales	BMW	Z4 sDrive30i	$46 750
GreatBritain	Scotland	BMW	Z4 sDrive30i	$46 750

Figure 8: Lookup table representation as a simple decision table

Implementation Details

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

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

To work with this kind of table `TransformedGridTable` object is created as constructor parameters it has an original grid table of lookup table (without header) and the `CoordinatesTransformer` that converts table coordinates to work with both vertical and horizontal conditions.

As the result we get a `GridTable` and works with it as a simple decision table structure, all coordinates transformations with lookup structure goes inside. Work with columns/rows is based on physical (not logical) structure of the table.

Simple Decision Tables

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

SimleRules Table

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

Header format:

SimpleRules <Return type> methodName(<Parameter type 1> parameterName1, (<Parameter type 1> parameterName1,….)

SimpleRules Table Example

SimpleRules String vehicleInjuryRating(String bodyType, String airbags, 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 9: SimpleRules table example

SimleLookup Table

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

Header format:

SimpleLookup <Return type> methodName(<Parameter type 1> parameterName1, (<Parameter type 1> parameterName1,….)

SimpleLookup Table Example

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

Figure 10: SimpleLookup table example

Decision Table Interpretation

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

Local Parameters in Decision Table

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

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

Simplified declarations

Case#1

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

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

Figure 11: Decision Table where user should provide expression and simple equal operation for condition declaration

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

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

Figure 12: Simplified Decision Table

How it works?

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

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

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

In the next step OpenL will create appropriate condition evaluator.

Case#2

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

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

Figure 13: Decision Table where user can omit name in declaration

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

Transposed Decision Tables

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

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

Figure 14: Transposed decision table

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

Working with Arrays

Arrays can be defined as follows for all tables that have properties of the `enum[]` type and for data tables with fields of the array type:

- horizontally
- vertically
- comma separated arrays

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

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

Figure 15: 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 16: Arranging array values vertically

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

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

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

Data Policy policyProfile4			
properties	modifiedOn	5/5/10	
	modifiedBy	LOCAL	
	name	Policies Set 4	
	category	Policy-Data	
name		**Policy**	Policy4
drivers	>driverProfiles3	**Drivers**	test1,test3\,4,test2
vehicles	>autoProfiles3	**Vehicles**	1965 VW Bug
clientTier		**Client Tier**	Elite
clientTerm		**Client Term**	Long Term

Figure 17: Array values separated by comma

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

- test 1
- test 3, 4
- test 2

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

Figure 17: Array values separated by comma. The second example

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

- value1
- value2
- value3

OpenL methods helpers to work with arrays

To facilitate work with arrays, OpenL Tablets provides two methods, one of them determines whether a particular element is included in an array:

```
contains(Object[] ary, Object ojb)
```

The other one checks if one array contains all elements from the other:

```
contains(Object[] ary1, Object[] ary2)
```

These methods work with Java objects, also there are methods with same functionality for all primitive types. The following example displays a table using the `contains` method:

Rules String getType1(String num)					
C1					RET1
contains(set, num)					result
String[] set					String result
Number Set					**Result**
1	3	5	7	9	Odd
2	4	6	8		Even

Figure 18: Checking arrays in conditions

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.

Representing Date Values

To represent date values in table cells, the following format must be used:

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

The value must always be preceded with an apostrophe. 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
```

OpenL Tablets also recognizes the native Excel date format.

Representing Boolean Values

OpenL Tablets supports the following formats of Boolean values:

- True, TRUE, Yes, YES
- False, FALSE, No, NO

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

Using Calculations in Table Cells

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

The following decision table demonstrates calculations in table cells:

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

Figure 19: Decision table with calculations

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

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

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

Data Type Table

Description

A **data type table** defines an OpenL Tablets data carrier. Using data types inside the OpenL Tablets rules is recommended, since using data types created in OpenL Tablets table from Java code is limited in the current implementation. For Java code, the preferable method is to use Java or Vocabulary type. For information on how this is done, see Data Table.

The following is an example of a data type table defining a data type called Person:

Datatype Person	
String	name
String	ssn
Date	dob
String	gender
String	maritalStatus

Figure 20: Data type table

The first row is the header containing the keyword 'Datatype' followed by the name of the data type. Every row, beginning with the second one, represents one property of the data type. The first column contains property types; the second column contains corresponding property names.

How OpenL handles Datatypes inside

Datatypes tables are being processed second after Properties Table it is done to build a domain model that is used in rules.

Datatype header format:

[Datatype <typename>] or [Datatype <typename> extends <parentTypeName>] or [Datatype <typename> <aliastype>]

Inheritance in Datatypes

There is possibility to inherit one datatype from another in OpenL Tablets. New datatype that inherits another will have access to all fields defined in parent datatype. If child datatype contains fields that defined in parent you will get warnings or errors (if field declared with different types in child and parent).

Also constructor with all fields of child datatype will contains all fields from parent fields and *toString*, *equals* and *hashCode* methods will use all fields form parent datatype.

Byte code generation at runtime

At runtime, when Openl engine instance is being built, for each datatype component java byte code is being generated, it represents simple java bean for this datatype. This byte code is being loaded to classloader so the object of type `Class<?>` can be accessible. Using this object through reflections new instances are being created and fields of datatypes are being initialized (see `DatatypeOpenClass` and `DatatypeOpenField` classes).

Java files generation

As generation of datatypes is performed on runtime and users can`t access this classes in their code, so to the `JavaWrapperAntTask` was added a goal "generate datatypes". It adds the possibility of generation java files and putting it to the file system. So users can use this types in their code. But there is a limitation: users shouldn`t create instances of datatype classes before creating an instance of OpenL engine. This is because the static class that was on the file system will be loaded first to the classloader. That is not good as the engine consider to work with datatypes generated at runtime.

Examples of right and wrong usage of datatypes

Look at the next 2 examples, represented right and wrong usage of datatype classes in your code. Examples will be shown with usage of dynamic OpenL wrapper.

At first call the ant task "*make wrapper generate datatypes*" and you will get java files of datatypes at your file system.

```
public interface RulesInterface {
       // define the interface of the rules engine. it will be succesfully
       // compiled as we have all appropriate classes at file system
       DoubleValue clientDiscount(Policy policy);

}
```

This example demonstrates the right usage of datatype beans. To the classloader will be loaded class generated at runtime.

```
EngineFactory<RulesInterface> engineFactory = new
       EngineFactory<RulesInterface>(RuleEngineFactory.RULE_OPENL_NAME,
"rules/main/Tutorial_4.xls", RulesInterface.class);

// create new instance of OpenL engine
RulesInterface test = engineFactory.makeInstance();
Policy policy = null;

// create new instance of datatype class after building OpenL instance
try {
     policy = (Policy) Class.forName("org.openl.generated.beans.Policy").newInstance();
     policy.setClientTier("Preferred");
} catch (Exception e) {
     e.printStackTrace();
}
System.out.println(test.clientDiscount(policy));
```

And this one demonstrates the wrong usage of datatype beans. To the classloader will be loaded class that was compiled from the file system.

```
// create new instance of datatype class before building OpenL instance
Policy policy = null;
try {
     policy = (Policy) Class.forName("org.openl.generated.beans.Policy").newInstance();
```

```
     policy.setClientTier("Preferred");
} catch (Exception e) {
     e.printStackTrace();
}
EngineFactory<RulesInterface> engineFactory = new
      EngineFactory<RulesInterface>(RuleEngineFactory.RULE_OPENL_NAME,
      "rules/main/Tutorial_4.xls", RulesInterface.class);
// create new instance of OpenL engine
RulesInterface test = engineFactory.makeInstance();
System.out.println(test.clientDiscount(policy));
```

Access datatype at runtime and after building OpenL wrapper

After parsing, each datatype is put to compilation context, so it will be accessible for rules during binding. Also all datatypes are placed to `IOpenClass` of whole module and will be accessible from `CompiledOpenClass#getTypes` when Openl wrapper wiill be generated.
Each `TableSyntaxNode` that is of type *xls.datatype* contains an object of datatype as its member.

Working with Datatype arrays from rules

There are 2 possibilities to work with Datatype arrays from rules:

- by numeric index, starting from 0
- by user defined index

In the first case by call drivers[5] you will get the 6 element of the Datatype array.
Second case is a little more complicated. First field of datatype is considered to be the user defined index. For example if we have a Datatype Driver with first String field name, we can create a Data Table, initializing two instances of Driver with names: John and David. Than in rules we can call the instance we need by drivers["David"]. You can use all Java types (including primitives) and Datatypes for your indexes. When the first field of Datatype is of int type called id, to call the instance from array, wrap it with quotes: e.g. drivers["7"], in this case you won`t get the 8 element in the array, but the Driver with id equals to 7.

Data Table

A **data table** contains relational data that can be referenced as follows:

- from other tables within OpenL Tablets
- from Java code through wrappers as Java arrays
- through OpenL Tablets run-time API as a field of the `Rules` class instance

Data tables can contain Java classes, OpenL Tablets data types, or types loaded in OpenL Tablets from other sources, for example, using the Vocabulary mechanism. For information on data types, see Data Type Table.

The following topics are included in this section:

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

Using Simple Data Tables

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

Data int numbers
this
Numbers
10
20
30
40
50

Figure 21: Simple data table

The first row is the header containing text in the following format:

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

In simple data tables, the keyword 'this' must be used for the following types:

- all primitive Java types
- class `java.lang.String`
- class `java.util.Date`
- all Java classes with a public constructor with a single String parameter

In the example above, information in the data table can be accessed from Java code as shown in the following code example:

```
int[] num = tableWrapper.getNumbers();

for (int i = 0; i < num.length; i++) {
       System.out.println(num[i]);
}
```

where `tableWrapper` is an instance of the wrapper class of the Excel or Word file. For information on wrappers, see Wrappers.

Using Advanced Data Tables

Advanced data tables are used for storing information for complex constructions, such as Java beans and data types. For information on data types, see Data Type Table.

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

```
Data <Java bean or data type> <data table name>
```

Each cell in the second row contains an attribute name of the data type or Java bean. Normally, the second row is hidden to business users.

The third row contains attribute display names. Each row starting with the fourth one contains values for specific data type instances.

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

Datatype Person	
String	name
String	ssn

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

Figure 22: Data type table and a corresponding data table

Data tables can use Java beans instead of data types. For example, instead of using a data type table Person, a developer can use the following Java bean:

```
public class Person {

	String name;
	String ssn;

	public String getName() {
		return name;
	}
	public void setName(String name) {
		this.name = name;
	}

	public String getSsn() {
		return ssn;
	}
	public void setSsn(String ssn) {
		this.ssn = ssn;
	}

}
```

If a Java bean is used, to avoid compilation error, the package where the Java bean is located must be imported using a configuration table as described in Configuration Table.

In Java code, the data table `p1` can be accessed as follows:

```
Person[] persArr = tableWrapper.getP1();

for (int i = 0; i < persArr.length; i++) {
	System.out.println(persArr[i].getName() + ' ' + persArr[i].getSsn());
}
```

where `tableWrapper` is an instance of the Excel or Word file wrapper. For information on wrappers, see Wrappers.

Ensuring Data Integrity

If a data table contains values defined in another data table, it is important to specify this relationship so that OpenL Tablets can check data integrity during compilation. 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 diagram, the data table **cities** contains values from the table **states.** To ensure users enter correct values, a reference to the **code** column in the **states** table is defined.

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

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

Figure 23: Defining a reference to another data table

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

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

```
>states
```

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

Specifying Data for Aggregated Objects

Data tables can be used to specify attributes of referenced objects. An object referenced by a data table is called an **aggregated object.** To specify an attribute of an aggregated object, the following format must be used in the row containing data table attribute names:

```
<name of reference to the aggregated object>.<object attribute>
```

To illustrate this approach, assume there are two Java classes `ZipCode` and `Address` defined:

```
public class ZipCode {

       String zip1; // 5-digit part - mandatory
       String zip2; // 4-digit part - optional

       public String getZip1() {
              return zip1;
       }
       public void setZip1(String zip1) {
              this.zip1 = zip1;
       }
```

```
        public String getZip2() {
                return zip2;
        }
        public void setZip2(String zip2) {
                this.zip2 = zip2;
        }
}

public class Address {
        String street;
        String city;
        ZipCode zip;

        public String getCity() {
                return city;
        }
        public void setCity(String city) {
                this.city = city;
        }
        public String getStreet() {
                return street;
        }
        public void setStreet(String street) {
                this.street = street;
        }
        public ZipCode getZip() {
                return zip;
        }
        public void setZip(ZipCode zip) {
                this.zip = zip;
        }
}
```

As can be seen from the code, the `Address` class contains a reference to the `ZipCode` class. A data table can be created that specifies values for both classes at the same time, for example:

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

Figure 24: Specifying values for aggregated objects

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

All Java classes referenced in a data table must be imported using a configuration table as described in Configuration Table.

Note: The reference path can be of any arbitrary depth, for example `account.person.address.street`.

Test Table

A **test table** is used to perform unit tests on executable tables, such as decision tables and method tables. It calls a particular method, provides test input values, and checks whether the returned value matches the expected value. Test tables are mostly used for testing decision tables.

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

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

Testmethod ampmTo24 ampmTo24Test		
ampmHr	ampm	_res_
Hour	**AM/PM**	**24 Hr**
3	AM	3
12	AM	0
12	PM	12
3	PM	15

Figure 25: 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:

```
Testmethod <method name> <test table name>
```

'Testmethod' is a keyword that identifies a test table. The second parameter is the name of the decision table method or any other Java method to be tested. The third parameter is the name of the test table, which is also the name of the method by which the test table can be executed from Java code.

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

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

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

Testmethod driverAgeType driverAgeTypeTest		
properties	name	Driver Age Type Test
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 26: An example of a test table with a context value

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

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

Figure 27: An example of a test table with a expected error column

Test results can be accessed through the test table API. For example, the following code fragment executes all test runs in a test table called **insuranceTest** and displays the number of failed test runs:

```
TableWrapper tableWrapper = new TableWrapper();

TestResult tr = tableWrapper.insuranceTestTestAll();

System.out.println("Number of failed test runs: "+tr.getNumberOfFailures());
```

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

Run Method Table

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

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

The following is an example of a run method table:

Runmethod append appendRun	
firstWord	**secondWord**
Hi,	John!
Hello,	Mary!
Good morning,	Bob!

Figure 28: Run method table

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

A run method table has the following structure:

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

```
Runmethod <method to call> <run method table name>
```

- The second row contains cells with method input parameter names.
- Starting with the third row, each row is a set of input parameters to be passed to the called method.

Method Table

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

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

Figure 29: Method table

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

```
<keyword> <return type> <method name and parameters>
```

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

The second row, and the following rows, is the actual code to be executed. It can reference parameters passed to the method and all Java objects and tables visible to the OpenL Tablets engine.

Method in the preceding example table can be called from Java code as follows:

```
ApprovalRulesWrapper tableWrapper = new ApprovalRulesWrapper();

System.out.println(tableWrapper.getGreeting("John"));
```

Configuration Table

OpenL Tablets allows externalizing business logic into Excel or Word files. However, these files can still use objects and methods defined in the Java environment. To enable use of Java objects and methods in Excel or Word tables, the file must have a configuration table. A **configuration table** provides information to the OpenL Tablets engine about available Java packages. Another purpose of a configuration file is to point to other Excel and Word files that can be referenced in tables.

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**
`import`	Imports the specified Java package so that its objects and methods can be used in tables.
`include`	Includes another Excel or Word file so that its tables and data can be referenced in tables of the current file.
`language`	Language import functionality.
`extension`	External set of rules for expanding OpenL Tablets capabilities. After adding, external rules are complied with OpenL Tablets rules and work jointly.
`vocabulary`	Ability to use user created dynamic classes in OpenL Tablets.

The following is an example of a configuration table:

Environment	
import	com.exigen.claims.data
	org.openl.meta
include	../include/Approval_TestData.xls

Figure 30: Configuration table

Properties Table

Description

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

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

<table>
<tr><th colspan="2">Properties table elements</th></tr>
<tr><th>Element</th><th>Description</th></tr>
<tr><td>scope</td><td>Identifies levels on which the property inheritance is defined. Available values are as follows:
<table>
<tr><th>Scope level</th><th>Description</th></tr>
<tr><td>Module</td><td>Identifies properties defined for the whole module and inherited by all tables in it. There can be only one table with the Module scope in one module.
<table>
<tr><th colspan="2">Properties property_test1</th></tr>
<tr><td>scope</td><td>Module</td></tr>
<tr><td>effectiveDate</td><td>4/7/10</td></tr>
<tr><td>expirationDate</td><td>4/28/11</td></tr>
<tr><td>lang</td><td>EN</td></tr>
<tr><td>currency</td><td>USD</td></tr>
<tr><td>state</td><td>CA</td></tr>
</table>
Figure 31: A properties table with the Module level scope</td></tr>
<tr><td>Category</td><td>Identifies properties applied to all tables where the category name equals the name specified in the category element.
<table>
<tr><th colspan="2">Properties property_test2</th></tr>
<tr><td>scope</td><td>Category</td></tr>
<tr><td>category</td><td>Testing</td></tr>
<tr><td>country</td><td>CA,CH,DE,FR</td></tr>
<tr><td>lob</td><td>Home</td></tr>
<tr><td>lang</td><td>GER</td></tr>
<tr><td>currency</td><td>CAD</td></tr>
</table>
Figure 32: A properties table with the Category level scope</td></tr>
</table></td></tr>
<tr><td>category</td><td>Defines the category if the scope element is set to Category. If no value is specified, the category name is retrieved from the sheet name.</td></tr>
<tr><td>Module</td><td>Identifies that properties can be overridden and inherited on the Module level.</td></tr>
</table>

Spreadsheet Table

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

The format of the spreadsheet table header is as follows:

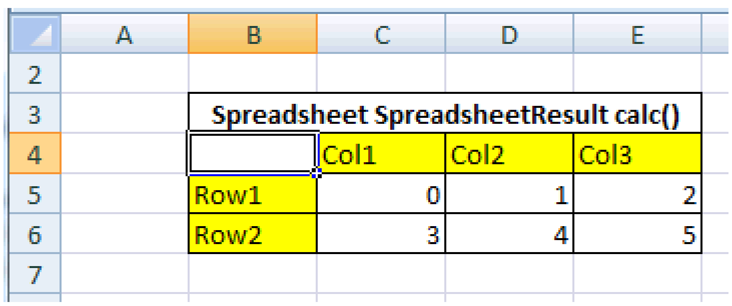
```
Spreadsheet SpreadsheetResult nameOfTableOrMethod(ARGUMENTS)
```

The following table describes the spreadsheet table header syntax:

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

Spreadsheet table header syntax	
Element	**Description**
SpreadsheetResult	Type of the return value. SpreadsheetResult returns the calculated content of the whole table. If only a single value is required, its type must be defined as a return type and calculated in the row or column named **RETURN.**
nameOfTableOrMethod	Java valid name of the table as for any executable table.
ARGUMENTS	Input arguments as for any executable table.

The first column and row of a spreadsheet table make the table column and row names. Values in other cells are the table values. An example 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 33: Spreadsheet table organization

A spreadsheet table can contain simple values, such as a string, numeric, range value, or advance value referencing another cell value. The following table describes how another cell value can be referenced in a spreadsheet table:

Methods of referencing another cell		
Notation	**Method**	**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.

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> ReturnType nameOfTableOrMethod(ARGUMENTS)
```

The following table describes the spreadsheet 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.

Column match table header syntax	
Element	**Description**
ReturnType	Type of the return value.
nameOfTableOrMethod	Java valid name of the table or method as for any executable table, exposing this table in an OpenL Tablets wrapper.
ARGUMENTS	Input arguments as for any executable table.

The following predefined algorithms are available:

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

Each algorithm has the following mandatory columns:

<table>
<tr><th colspan="2">Algorithm mandatory columns</th></tr>
<tr><th>Column</th><th>Description</th></tr>
<tr><td>Names</td><td>Names refer to the table or method arguments and bind an argument to a particular row. The same argument can be referred in multiple rows.

Arguments are referenced by their short names. For example, if an argument in a table is a Java Bean with the some property, it is enough to specify some in the names column.</td></tr>
<tr><td>Operations</td><td>The operations column defines how to match or check arguments to values in a table. The following operations are available:
<table>
<tr><th>Operation</th><th>Checks for</th><th>Description</th></tr>
<tr><td>match</td><td>equality or belonging to a range</td><td>The argument value must be equal to or within a range of check values.</td></tr>
<tr><td>min</td><td>minimally required value</td><td>The argument must not be less than the check value.</td></tr>
<tr><td>max</td><td>maximally allowed value</td><td>The argument must not be greater than the check value.</td></tr>
</table>
The min and max operations work with numeric and date types only.

The min and max operations can be replaced with the match operation and ranges. This approach adds more flexibility because it enables the checking of all cases within one row.</td></tr>
<tr><td>Values</td><td>The values column typically has multiple sub columns containing table values.</td></tr>
</table>

MATCH Algorithm

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

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

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

Figure 34: An example of the MATCH algorithm table

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

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

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

SCORE Algorithm

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

- names
- operations
- weight
- values

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

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

Figure 35: An example of the SCORE algorithm table

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

The following limitations apply:

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

WEIGHTED Algorithm

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

The WEIGHTED algorithm requires the same columns as the SCORE algorithm. Yet it expects that first three rows are **Return Values, Total Score,** and **Score. Return Values** and **Total Score** represent the MATCH algorithm, and the **Score** row is the beginning of the SCORE part.

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

Figure 36: An example of the WEIGHTED algorithm table

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

TBasic Table

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

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

The format of the TBasic table header is as follows:

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

The following table describes the spreadsheet table header syntax:

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

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

Algorithm parts	
Element	**Description**
Algorithm precondition or preprocessing	Executed when the component starts execution.
Algorithm steps	Represents the main logic of the component.
Postprocess	Identifies a part executed when the algorithm part is executed.
User functions and subroutines	Contains user functions definition and subroutines.

Chapter 3: OpenL Business Expression Language (BEX)

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

Java Business Object Model as Basis for OpenL Business Vocabulary(OBV)

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

```
driver.age
```

and

```
Age of the Driver
```

Another example:

```
policy.effectiveDate
```

and

```
Effective Date of the Policy
```

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

New Keywords, how to avoid possible naming conflicts

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

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

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

Simplifying Expressions with explanatory variables

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

In Java:

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

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

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

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

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

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

Simplifying Expressions by Using "Unique in Scope" concept

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

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

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

OpenL Vocabulary and OpenL BEX

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

Future developments, compatibility etc.

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

OpenL Programming Language Framework

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

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

```
driver.age < 25
```

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

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

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

OpenL Grammars

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

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

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

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

The node types of the nodes are

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

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

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

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

Context, Variables and Types

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

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

- Method parameter
- Local Variable
- Member of surrounding class (in case of OpenL it is usually the implementation of IOpenClass called Module)
- External types accessed as static, mostly Java classes that are imported into OpenL
- Fields and Methods in binding context - this is a feature that does not exist in Java; OpenL allows programmatically add custom types, fields and methods into Binding Context; for different examples of how it could be done you need to take a good look at the source code of OpenLBuilder classes in different packages. For example, **org.openl.j** automatically imports all the classes from the **java.util** in addition to the standard **java.lang** package. Since version 5.1.1 **java.math** is also being imported automatically

OpenL Type System

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

OpenL Tablets as OpenL Type extension

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

Operators

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

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

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

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

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

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

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

Binary Operators' Semantic Map

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

Unary Operators

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

Cast Operators

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

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

Strict equality and relation operators

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

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

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

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

The list of org.openl.j Operators

In the order of priority:

Assignment operators	
Operator	**org.openl.j operator**
=	op.assign
+=	op.assign.add
-=	op.assign.subtract
*=	op.assign.multiply
/=	op.assign.divide
%=	op.assign.rem
&=	op.assign.bitand
|=	op.assign.bitor
^=	op.assign.bitxor

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

Implication	
Operator	**org.openl.j operator**
->	op.binary.impl (*)

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

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

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

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

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

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

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

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

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

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

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

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

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

Chapter 4: Working With Projects

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

The following topics are included in this section:

- Project Structure
- Creating a Project
- Generating a Wrapper

Project Structure

The best way to use the OpenL Tablets rule technology in a solution is to create an OpenL Tablets project in Eclipse. An OpenL Tablets project is a Java project with an OpenL Tablets facet. A typical OpenL Tablets project contains the following general elements:

OpenL Tablets project contents	
Element	**Description**
Main content	
Excel files	Physical storage of rules and data in the form of tables.
Additional Java code	Optional classes for domain models and for processing or testing rules in the project. Solution developers can decide whether to include additional code in OpenL Tablets projects.
Additional content	
Ant task for generating static wrappers	Ant configuration file used for creating wrapper classes for Excel files so that they can be accessed from code. For general information on wrappers, see Wrappers.
Wrappers	Java classes providing access to OpenL Tablets rules in Excel files. Wrappers are generated as described in Generating a Wrapper.
Third party dependencies	Java libraries files used in rules and Java code.

The following table describes common OpenL Tablets folders in the physical project structure. However, the structure can be adjusted according to the developer's preferences, for example, to comply with the Maven structure.

Rule projects must contain the default folders **rules** and **gen** to be recognized.

OpenL Tablets project structure	
Folder	**Contents**
`src`	Contains all project Java classes apart from wrappers.
`rules`	Contains Excel files with rules.
`gen`	Contains generated wrapper classes.
`bin`	Contains compiled Java code.
`build`	Contains Ant configuration file for generating wrapper classes. For information on generating wrappers, see Generating a Wrapper.

OpenL Tablets project structure	
Folder	**Contents**
`libs`	Contains rules project dependencies JAR files.

Creating a Project

The simplest way to create an OpenL Tablets project is to add a simple OpenL Tablets in Eclipse to the installed OpenL Tablets Eclipse Update Site.

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

Generating a Wrapper

Access to rules and data in Excel tables is realized through OpenL Tablets API. OpenL Tablets provides wrappers to developers to facilitate easier usage.

In OpenL Web Studio, a rule project must have a static wrapper created in the `gen` folder to be recognized. For general information on wrappers, see Wrappers.

The following topics are included in this section:

- Generating a Dynamic Wrapper
- Using a Dynamic Wrapper in the Run-time Context
- Generating a Static Wrapper
- Example of Using a Static and Dynamic Wrapper

Generating a Dynamic Wrapper

Only an interface must be defined when creating a project for a dynamic wrapper. OpenL Tablets users can clearly define rules displayed in the application by using this interface. Using dynamic wrappers is a recommended practice.

This section illustrates the creation of a dynamic wrapper for a **Simple** project in Eclipse with the OpenL Tablets Eclipse Update Site installed. Only one rule **hello1** is created in the **Simple** project by default.

Rules void hello1(int hour)			
Rule	C1	C2	A1
	min <= hour	hour <= max	System.out.println(greeting + ", World!")
	int min	int 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

Figure 37: The hello1 rule table

Proceed as follows:

1. In the project `src` folder, create an interface as follows:
```
public interface simple {
void hello1(int i);
}
```
2. Create a class for a wrapper as follows:
```
package template;

import static java.lang.System.out;
import org.openl.rules.runtime.RuleEngineFactory;
public class Dynamic_wrapper {

    public static void main(String[] args) {
            //define the interface
            RuleEngineFactory<simple> rulesFactory = new
RuleEngineFactory<simple>("rules/TemplateRules.xls", simple.class);

            simple rules = rulesFactory.newInstance();
            rules.hello1(12);

}

}
```

When the class is run, it executes and displays **Good Afternoon, World!**

Using a Dynamic Wrapper in the Run-time Context

This section describes the use of the run-time context for dispatching rules by dimension properties values, when rules are overloaded by properties.

For example, consider two rules overloaded by dimension properties. Both rules have the same name.

The first rule, covering an auto policy, follows:

Rules void hello1(int hour)			
properties	createdOn	4/7/10	
	createdBy	LOCAL	
	lob	Auto	
Rule	C1	C2	A1
	min <= hour	hour <= max	System.out.println(greeting + ", World!")
	int min	int 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

Figure 38: The auto policy rule

The second rule, covering a homeowner policy, follows:

Rules void hello1(int hour)			
properties	modifyOn	4/7/10	
	modifiedBy	LOCAL	
	lob	Home	
Rule	C1	C2	A1
	min <= hour	hour <= max	System.out.println(greeting + ",Guys!")
	int min	int max	String greeting
Rule	From	To	Greeting
R10	0	11	It is Mornig
R20	12	17	It is Afternoon
R30	18	21	It is Evening
R40	22	23	It is Night

Figure 39: The homeowner policy rule

A dynamic wrapper enables the user to identify which of these rules must be executed.

```
// Getting runtime environment which contains context
IRuntimeEnv env = ((IEngineWrapper<simple>) rules).getRuntimeEnv();

        // Creating context
       IRulesRuntimeContext context = new DefaultRulesRuntimeContext();
env.setContext(context);
// define context
context.setLob("Home);
```

As a result, the code of the dynamic wrapper with the run-time context resembles the following:

```
import static java.lang.System.out;

import org.openl.rules.context.DefaultRulesRuntimeContext;
import org.openl.rules.context.IRulesRuntimeContext;
import org.openl.rules.runtime.RuleEngineFactory;
import org.openl.runtime.IEngineWrapper;
import org.openl.vm.IRuntimeEnv;
public class Dynamic_wrapper {

             public static void main(String[] args) {
                    //define the interface
```

```
                    RuleEngineFactory<simple> rulesFactory = new
RuleEngineFactory<simple>("rules/TemplateRules.xls", simple.class);
                    simple rules = rulesFactory.newInstance();
                    // Getting runtime environment which contains context
               IRuntimeEnv env = ((IEngineWrapper<simple>) rules).getRuntimeEnv();
               // Creating context (most probably in future, the code will be different)
               IRulesRuntimeContext context = new DefaultRulesRuntimeContext();

               env.setContext(context);
               context.setLob("Home");
                    rules.hello1(12);

       }


}
```

Run this class. In the console, ensure that the rule with **lob = Home** was executed. With the input parameter **int = 12,** the **It is Afternoon, Guys** phrase is displayed.

Generating a Static Wrapper

To generate a static wrapper class, proceed as follows:

1. Configure the Ant task file as described in Configuring the Ant Task File.
2. Execute the Ant task file as described in Executing the Ant Task File.

For an example of how to use a static and dynamic wrapper, see Example of Using Static and Dynamic Wrappers.

Configuring the Ant Task File

When a new OpenL Tablets project is created, it already contains an Ant task file `GenerateJavaWrapper.build.xml` located in the `build` folder. When the file is executed, it automatically creates wrapper Java classes for specified Excel files. The Ant task file must be adjusted to match contents of the specific project.

For each Excel file, an individual `<openlgen>` section must be added between the `<target>` and `</target>` tags.

Each `<openlgen>` section has a number of parameters that must be adjusted. The following table describes `<openlgen>` section parameters:

Parameters in the <openlgen> section	
Parameter	**Description**
`openlName`	OpenL configuration to be used. For OpenL Tablets, the following value must always be used: `org.openl.xls`
`userHome`	Location of user defined resources relative to the current OpenL Tablets project.
`srcFile`	Reference to the Excel or Word file for which a wrapper class must be generated.

Parameters in the <openlgen> section	
Parameter	**Description**
`targetClass`	Full name of the wrapper class to be generated. OpenL Web Studio recognizes modules in projects by wrapper classes and uses their names in the user interface. If there are multiple wrappers with identical names, only one of them is recognized as a module in OpenL Web Studio.
`displayName`	End user oriented title of the file that appears in OpenL Web Studio.
`targetSrcDir`	Folder where the generated wrapper class must be placed.

The following is an example of the `GenerateJavaWrapper.build.xml` file:

```
<project name="GenJavaWrapper" default="generate" basedir="../">
	<taskdef name="openlgen" classname="org.openl.conf.ant.JavaWrapperAntTask"/>

<target name="generate">
	<echo message="Generating wrapper classes..."/>

	<openlgen openlName="org.openl.xls" userHome="."
		srcFile="rules/Rules.xls"
		targetClass="com.exigen.claims.RulesWrapper"
		displayName="Rule table wrapper"
		targetSrcDir="gen"
	>
	</openlgen>

	<openlgen openlName="org.openl.xls" userHome="."
		srcFile="rules/Data.xls"
		targetClass=" com.exigen.claims.DataWrapper"
		displayName="Data table wrapper"
		targetSrcDir="gen"
	>
	</openlgen>

</target>
</project>
```

Executing the Ant Task File

To execute the Ant task file and generate wrappers, proceed as follows:

3. In Eclipse, refresh the project.
4. Execute the Ant task XML file as an Ant build.
5. Refresh the project again so that wrapper classes are displayed in Eclipse.

Once wrappers are generated, the corresponding Excel or Word files can be used in the solution.

Example of Using a Static and Dynamic Wrapper

The following example illustrates the use of static and dynamic wrappers:

```
public class Tutorial1Main {

	public interface Tutorial1Rules {
void hello1(int i);
	}

	public static void main(String[] args)
```

```
    {
     out.println("\n* OpenL Tutorial 1\n");

     out.println("Working using static wrapper...\n");

           callRulesWithStaticWrapper();

     out.println("\nWorking using dynamic wrapper...\n");

           callRulesWithDynamicWrapper();

    }

    private static void callRulesWithStaticWrapper() {
           //Get current hour
           Calendar calendar = Calendar.getInstance();
           int hour = calendar.get(Calendar.HOUR_OF_DAY);

           //Creates new instance of Java Wrapper for our lesson
           Tutorial_1Wrapper tut1 = new Tutorial_1Wrapper();

           //Step 1
     out.println("* Executing OpenL rules...\n");
     // Call the method wrapping Decision Table "hello1"
           tut1.hello1(hour);
    }

    private static void callRulesWithDynamicWrapper(){
           // Creates new instance of OpenL Rules Factory
           RuleEngineFactory<Tutorial1Rules> rulesFactory = new
RuleEngineFactory<Tutorial1Rules>("rules/Tutorial_1.xls", Tutorial1Rules.class);

           //Creates new instance of dynamic Java Wrapper for our lesson
     Tutorial1Rules rules = rulesFactory.newInstance();

     //Get current hour
           Calendar calendar = Calendar.getInstance();
           int hour = calendar.get(Calendar.HOUR_OF_DAY);

     out.println("* Executing OpenL rules...\n");
     rules.hello1(hour);
    }
}
```

Rules Runtime Context

Description

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

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

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

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

OpenL engine provides more elegant solution for this case.

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

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

The following OpenL tables snippet shows our sample in action:

Rules DoubleValue bonus()		
properties	name	Bonus Premium
	state	STATE #1
RET1		
bonusPremium		
DoubleValue bonusPremium		
Bonus Premium		
$100		

Rules DoubleValue bonus()		
properties	name	Bonus Premium
	state	STATE #2
RET1		
bonusPremium		
DoubleValue bonusPremium		
Bonus Premium		
$150		

Rules DoubleValue bonus()		
properties	name	Bonus Premium
	state	STATE #N
RET1		
bonusPremium		
DoubleValue bonusPremium		
Bonus Premium		
$200		

Figure 41: Overloaded by properties group of Decision Tables

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

OpenL has predefined runtime context which already has several properties.
Refer See Business Dimension properties (dimensional column table) in `TablePropertiesDefinitions.xlsx` document for more details.

Using Rules Runtime Context in java code

OpenL static java wrapper

The following code spippet demonstrates how can be used rules runtime context and his variables in application.

```
...
TestWrapper wrapper = new TestWrapper();
IRulesRuntimeContext context = wrapper.getRuntimeContext();

Calendar calendar = Calendar.getInstance();
calendar.set(2003, 5, 15);
context.setCurrentDate(calendar.getTime());

DoubleValue res1 = wrapper.driverRiskScoreOverloadTest("High Risk Driver");
...
```

Rules engine factory

Rules engine factory provides method that returns runtime context only when your interface extends *IRulesRuntimeContextProvider* interface.

```
public interface ITestI extends IRulesRuntimeContextProvider {
    DoubleValue driverRiskScoreOverloadTest(String driverRisk);
}
```

The following code spippet demonstrates how can be used rules runtime context and his variables in application.

```
...
File xlsFile = new File("RulesContextTest.xls");
EngineFactory engineFactory = new RuleEngineFactory(xlsFile, ITestI.class);
ITestI instance = engineFactory.newInstance();

IRulesRuntimeContext context = instance.getRuntimeContext();

Calendar calendar = Calendar.getInstance();
calendar.set(2003, 5, 15);
context.setCurrentDate(calendar.getTime());

DoubleValue res1 = instance.driverRiskScoreOverloadTest("High Risk Driver");
...
```

Validation for Tables

Description

Validation phase is after binding phase that is serves to checks all tables for errors and accumulate all errors.

Validators

All possible validators are stored in `ICompileContext` of OpenL class. (The default compile context is `org.openl.xls.RulesCompileContext` that is generated automatically).
Validators get the OpenL and array of `TableSyntaxNodes` that represent tables for check and must return `ValidationResult`.
`ValidationResult`:

- status(fail or success) and
- all error/warn messages that occured

Table properties validators

Table properties that is described in `TablePropertyDefinition.xlsx` can have constraints. Some constraints have predifined validators associated with them.

Adding own property validator:

1. Add constraint:

   1.1. Define constraint in TablePropertyDefinition.xlsx(constraints field)

   1.2. Create constraint class and add it into the ConstraintFactory

2. Create own validator

   2.1. Create class of your validator and define in method org.openl.codegen.tools.type.TablePropertyValidatorsWrapper.init() constraint associated with validator.

   2.2. If it needed you can modify velosity script RulesCompileContext-validators.vm in project org.openl.rules.gen that will generate org.openl.xls.RulesCompileContext

   2.3. Run org.openl.codegen.tools.GenRulesCode.main(String[]) to generate new org.openl.xls.RulesCompileContext with your valiadator

3. Write unit tests!

Existing validators

- **Unique in module** validator checks uniqueness in module of some property
- **Active table validator** checks correctness of "active" property. Only one active table

Appendix A: BEX Language Overview

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

The following topics are included in this section:

Introduction to BEX

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

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

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

Keywords

The following table represents BEX keyword equivalents to Java expressions:

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

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

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

Simplifying Expressions

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

- Notation of Explanatory Variables
- Uniqueness of Scope

Notation of Explanatory Variables

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

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

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

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

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

Uniqueness of Scope

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

Index