EIS GROUP™

Developers Guide
OpenL Tablets BRMS
Release 5.16

Document number: TP_OpenL_DG_3.1_LSh

Revised: 11-18-2015

Table of Contents

1 Preface

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

The following topics are included in this preface:

1.1 Audience

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

Basic knowledge of Java, Ant, and Microsoft Excel is required to use this guide effectively.

1.2 Related Information

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

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

1.3 Typographic Conventions

The following styles and conventions are used in this guide:

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

Typographic styles and conventions	
Convention	**Description**
Hyperlink	Represents a hyperlink. Clicking a hyperlink displays the information topic or external source.
[name of guide]	Reference to another guide that contains additional information on a specific feature.

2 Introducing OpenL Tablets

This chapter introduces OpenL Tablets and describes its main concepts.

The following topics are included in this chapter:

2.1 What Is OpenL Tablets?

OpenL Tablets is a business rules management system and business rules engine based on tables presented in Excel documents. Using unique concepts, OpenL Tablets facilitates treating business documents containing business logic specifications as executable source code. Since the format of tables used by OpenL Tablets is familiar to business users, OpenL Tablets bridges a gap between business users and developers, thus reducing costly enterprise software development errors and dramatically shortening the software development cycle.

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

The major advantages of using OpenL Tablets are as follows:

- OpenL Tablets removes the gap between software implementation and business documents, rules, and policies.
- Business rules become transparent to developers.

  For example, decision tables are transformed into Java methods or directly into web service methods. 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 document.
- OpenL Tablets provides calculation explanation capabilities, enabling expansion of any calculation result by pointing to source arguments in the original documents.
- OpenL Tablets enables users to create and maintain tests to insure reliable work of all rules.
- OpenL Tablets provides cross-indexing and search capabilities within all project documents.
- OpenL Tablets provides full rules lifecycle support through its business rules management applications.
- OpenL Tablets supports the `.xls` and `.xlsx` file formats.

2.2 Basic Concepts

This section describes the basic concepts of OpenL Tablets and includes the following topics:

Rules

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

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

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

Tables

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

Projects

An OpenL Tablets **project** is a container of all resources required for processing rule related information. Usually, a project contains Excel files and Java code. For more information on projects, see the OpenL Tablets Reference Guide, chapter *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.

Wrapper

A **wrapper** is a Java object that exposes rule tables via Java methods and data tables as Java objects and allows developers to access table information from code. 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 are still used on the server.

For more information on wrappers, see Using OpenL Tablets rules from Java Code.

Execution Mode for OpenL Project

Execution mode for OpenL project is a light weight compilation mode that enables only evaluating of rules; but editing, tracing and search are not available. Since the Engine will not load test tables and keep debug information in memory in this mode, memory consumption is up to 5 times less than for debug mode.

By default, the execution mode (`exectionMode=true`) is used in OpenL Tablets Web Services.

The debug mode (`exectionMode=false`) is used by default in OpenL Tablets WebStudio.

Flag indicating required mode is introduced in runtime API and in wrappers.

To compile an OpenL Tablets project in execution mode, proceed as follows:

- If the OpenL Tablets high level API (instantiation strategies) is used, define an execution mode in a constructor of the particular instantiation strategy.
- If the low level API (Engine factories) is used, set an execution mode flag using the `setExecutionMode(boolean)` method.

2.3 System Overview

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

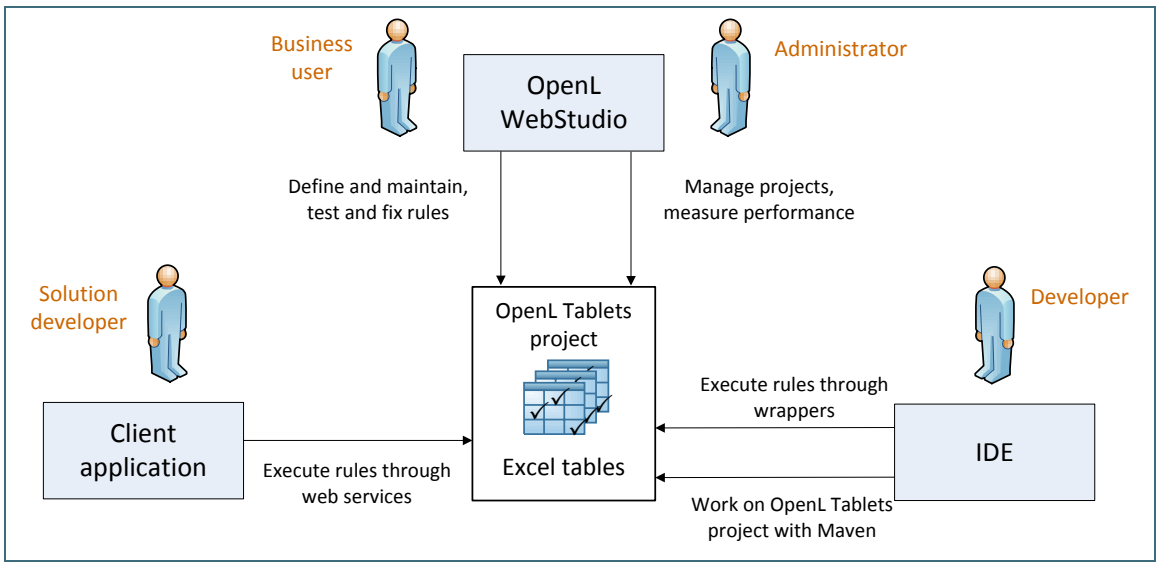

Figure 1: OpenL Tablets overview

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

1. A business analyst creates a new OpenL Tablets project in OpenL Tablets WebStudio.
   Optionally, development team may provide the analyst with a project in case of complex configuration.
   The business analyst also creates correctly structured tables in Excel files based on requirements and includes them in the project. Typically, this task is performed through Excel or OpenL Tablets WebStudio in a web browser.
2. The business analyst performs unit and integration tests by creating test tables and performance tests on rules through OpenL Tablets WebStudio.
   As a result, fully working rules are created and ready to be used.
3. A developer adds configuration to the project according to application needs.
   Alternatively, they can create a new OpenL Tablets project in their IDE via OpenL Maven Archetype and adjust it to use business user input.
4. A developer employs business rules directly through the OpenL Tablets engine or remotely through web services.
5. Whenever required, the business user updates or adds new rules to project tables.
   OpenL Tablets business rules management applications, such as OpenL Tablets WebStudio, Rules Repository, and Rule Service, can be set up to provide self-service environment for business user changes.

2.4 Quick Start with OpenL Tablets

OpenL Tablets provide a few ways to create a project. We recommend using Simple Project Maven Archetype approach for creating a project for the first time or create it via OpenL Tablets WebStudio. For more information on approaches for creating a project with detailed descriptions, see How to Start With OpenL Rules Project.

After a project is created, a zip or Excel file for importing the project to OpenL Tablets WebStudio can be used. For more information on importing an existing project into OpenL Tablets WebStudio, see OpenL Tablets WebStudio User Guide.

OpenL Tablets WebStudio provides convenient UI to work with rules. However, its usage can be avoided by working with rules from IDE only using the OpenL Tablets Maven plugin. The plugin provides compilation and testing of rules and wrapper generation support.

Also, OpenL Tablets has OpenL Tablet Demo Package available at OpenL Tablets website. A demo is a zip file that contains a Tomcat with configured OpenL Tablets WebStudio and OpenL Tablets Web Services projects. It can be used to effectively start using OpenL Tablets products.

3 OpenL Tablets Rules Projects

This chapter describes how to create and use OpenL Tablets Rules projects.

The following topics are included in this chapter:

3.1 OpenL Rules Project

OpenL Rules project is a project that contains Excel files with OpenL Tablets rules and may contain a rules project descriptor. The rules project descriptor is a XML file that defines project configuration and allows setting project dependencies.

OpenL Rules Project can easily use rules from other projects via dependency functionality.

3.2 Rules Project Descriptor

A rules project descriptor is a XML file that contains information about the project and configuration details used by OpenL to load and compile the rules project. The predefined name that is used for a rules project descriptor is *rules.xml.*

This section includes the following topics:

Quick Overview

The following code fragment is an example of the rules project descriptor:

```
<project>
	<!-- Project name. -->
	<name>Project name</name>
	<!-- Optional. Comment string to project. -->
	<comment>comment</comment>

	<!-- OpenL project includes one or more rules modules.  -->
	<modules>

		<module>
			<name>MyModule1</name>
			<type>API</type>

<!--
				Rules document which is usually an excel file in the project.
			-->
```

```
                        <rules-root path="MyModule1.xls"/>

                </module>

                <module>
                        <name>MyModule2</name>
                        <type>API</type>

<!--
                                Rules document which is usually an excel file in the project.
                        -->
                        <rules-root path="MyModule2.xls"/>
                        <method-filter>
                                <includes>
                                        <value> * </value>
                                </includes>
                        </method-filter>
                </module>
        </modules>

<dependencies>
                <dependency>
                        <name>projectName</name>
                        <autoIncluded>false</autoIncluded>
                </dependency>
        </dependencies>
        <properties-file-name-pattern>{lob}</properties-file-name-pattern>
        <properties-file-name-
processor>default.DefaultPropertiesFileNameProcessor</properties-file-name-processor>
        <!-- Project's classpath (list of all source dependencies). -->
        <classpath>
                <entry path="path1"/>
                <entry path="path2"/>
        </classpath>

</project>
```

Descriptor Elements

The descriptor file contains several sections that describe project configuration:

- Project Configurations
- Module Configurations
- Dependency Configurations
- Classpath Configurations

Project Configurations

The project configurations are as follows:

Project section		
Tag	**Required**	**Description**
name	yes	Project name. It is a string value which defines a user-friendly project name.
comment	no	Comment for project.
dependency	no	Dependencies to projects.
modules	yes	Project modules. A project can have one or several modules.
classpath	no	Project relative classpath.

Project section		
Tag	**Required**	**Description**
properties-file-name-pattern	no	File name pattern to be used by the file name processor. The file name processor adds extracted module properties from a module file name.
properties-file-name-processor	no	Custom implementation of `org.openl.rules.project.PropertiesFileNameProcessor` used instead of default implementation.

Module Configurations

The module configurations are as follows:

Module section		
Tag	**Required**	**Description**
name	yes/no	Module name. It is a string value which defines a user-friendly module name. **Note:** It is used by OpenL Tablets WebStudio application as a module display name. It is not required for modules defined via wildcard.
type	yes	Module instantiation type. Possible values are case-insensitive and can be **dynamic**, **api**, or **static** (deprecated). It defines the way of OpenL project instantiation.
classname	yes/no	Name of rules interface. It is used together with *type*. It is not required for the **api** type.
method-filer	no	Filter that defines tables to be used for interface generation. Java regular expression can be used to define a filter for multiple methods.
rules-root	yes/no	Path to the main file of a rules module. It is used together with **type**. Ant pattern can be used to define multiple modules via wildcard. For more information on Ant patterns, see Ant patterns.

Dependency Configurations

The dependency configurations are as follows:

Dependency section		
Tag	**Required**	**Description**
name	yes	Dependency project name.
autoIncluded	yes	Identifier, which, if set to **true**, that all modules from the dependency project will be used in this project module. If it is set to **false**, modules from the dependency project can be used in this project as dependencies, and each module will define its own list of used dependency modules.

Classpath Configurations

The classpath configurations are as follows:

Classpath section		
Tag	**Required**	**Description**
entry	no	Path for the classpath entry, that is, classes or jar file.

3.3 Project Resolving

The `RulesProjectResolver` Java class resolves all OpenL Tablets projects inside the workspace. The Resolver lists all folders in the workspace and tries to detect the OpenL Tablets project by the predefined strategy. The easiest way to initialize `RulesProjectResolver` is to use the `loadProjectResolverFromClassPath()` static method that uses `project-resolver-beans.xml` from classpath, that is, Spring beans configuration that defines all resolving strategies and their order.

Make sure that the resolving strategies are in the correct order, as some projects may be matched by several resolving strategies. By default, the resolving strategies are in the following order:

Resolving strategies		
Number	**Strategy**	**Description**
1.	Project descriptor resolving strategy	The strictest resolving strategy. It is based on the descriptor file as described previously in this section.
2.	Excel file resolving strategy	A resolving strategy for the simplest OpenL project which contains only Excel files in root folder without wrappers and descriptor. Each Excel file represents a module.

3.4 How to Start with OpenL Rules Project

Firstly, an OpenL Rules project must be created. It can be done in the following ways:

- using Maven archetype
- using OpenL Tablets WebStudio
- manually

See the following sections for detailed information:

- Creating a Project Using the Maven Archetype
- Creating a Project in OpenL Tablets WebStudio
- Creating a Project Manually
- Editing Rules
- Using OpenL Tablets Rules from Java Code
- Handling Data and Data Types in OpenL Tablets

Creating a Project Using the Maven Archetype

OpenL Tablets provides the Maven archetype which can be used to create a simple OpenL Rules project.

To create a project using the Maven archetype, proceed as follows:

1. Execute the following command in command line:
   ```
   mvn archetype:generate
   ```
   Maven runs the archetype console wizard.
2. Select the **openl-simple-project-archetype** menu item.
   As an alternative way is using the following command:
   ```
   mvn archetype:generate
   -DarchetypeGroupId=org.openl.rules
   -DarchetypeArtifactId=openl-simple-project-archetype
   -DarchetypeVersion=5.X.X
   ```
3. Follow with the Maven creation wizard.

After all steps are completed, a new Maven based project appears in the file system. It is an OpenL Rules project which has one module with simple rules in it.

4. Execute the following command in the command line from the root of the project folder to compile the project:
   ```
   mvn install
   ```

After executing this command, the following files can be found in the target folder:

1. zip file with "-deployable" suffix for importing the project to OpenL Tablets WebStudio.
   For more information, see ***[OpenL Tablets WebStudio User Guide].***
2. zip file (with "-runnable" suffix) that can be executed after extracting it.
   It demonstrates how OpenL Tablets rules can be invoked from Java code.
3. jar file that contains only compiled Java classes.
   This jar can be put in classpath of the project and used as a depended library.

Creating a Project in OpenL Tablets WebStudio

OpenL Tablets WebStudio allows users to create new rule projects in the Repository in one of the following ways:

- create a rule project from template
- create a rule project from Excel files
- create a rule project from zip archive
- import a rule project from workspace

The following diagram explains how projects are stored in OpenL Tablets WebStudio and then deployed and used by OpenL Tablets Web Services:

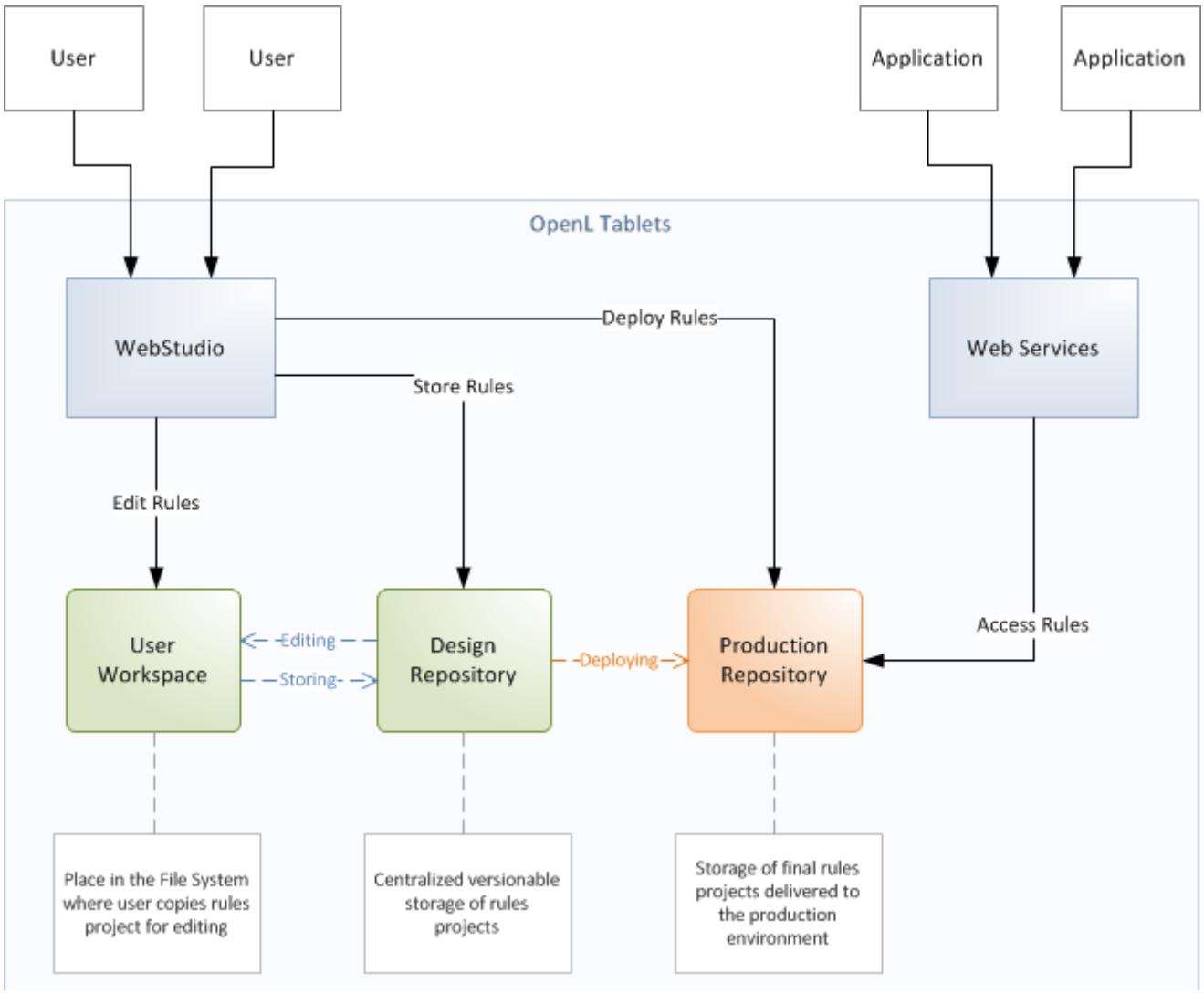

Figure 2: OpenL Tablets WebStudio and OpenL Tablets Web Services Integration

When a user starts editing a project, it is extracted from Design Repository and placed in the file system, in a user workspace. The project becomes locked in Design Repository for editing by other users. After editing is finished, the user saves the project. An updated version of the project is saved to Design Repository and becomes available for editing by other users.

OpenL Tablets Web Services use separate repository instance, Production Repository. OpenL Tablets WebStudio can be configured to deploy complete and tested rules projects to that repository.

For more information, see the OpenL Tablets WebStudio User Guide.

Creating a Project Manually

OpenL does not oblige a user to use predefined ways of project creation and enables using the user's own project structure. The Project Resolving mechanism can be used as a base for the project structure definition. Depending on the resolving strategy, more or less files and folders are to be created, but several project elements definition is mandatory. For more information on manually creating a project, see OpenL Rules Project.

Editing Rules

When a project is created, business rules have to be defined. It can be done using OpenL Tablets WebStudio or manually using MS Excel. If the simple rules project is used, there are several simple predefined rules that can be used as an example.

Using OpenL Tablets Rules from Java Code

For access to rules and data in Excel tables, OpenL Tablets API is used. OpenL Tablets provides a wrapper to facilitate easier usage.

This section illustrates the creation of a wrapper for a **Simple** project in IDE. There is only one rule **hello1** 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 3: 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 wrapper object as follows:

```
import static java.lang.System.out;
import org.openl.rules.runtime.RulesEngineFactory;

public class Example {

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

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

   }
}
```

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

The interface can be generated by OpenL Tablets in runtime if the developer does not define it when initializing the rule engine factory. In this case, rules can be executed via reflection.

The following example illustrates using a wrapper with a generated interface in runtime:

```
public static void callRulesWithGeneratedInterface(){
```

```
        // Creates new instance of OpenL Rules Factory
        RulesEngineFactory<?> rulesFactory =
new RulesEngineFactory<Object>("TemplateRules.xls");
                            //Creates new instance of dynamic Java Wrapper for our lesson
        Object rules = rulesFactory.newInstance();

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

        Class<?> clazz = rulesFactory.getInterfaceClass();

        try{
                Method method = clazz.getMethod("hello1", int.class);
                out.println("* Executing OpenL rules...\n");
                method.invoke(rules, hour);
        }catch(NoSuchMethodException e){
        }catch (InvocationTargetException e) {
        }catch (IllegalAccessException e) {
        }
}
```

This section includes the following topics:

- Using OpenL Tablets Rules with the Runtime Context
- Using OpenL Tablets Projects from Java Code
- Accessing a Test Table from Java Code
- Generating Java Classes from Datatype Tables

Using OpenL Tablets Rules with the Runtime Context

This section describes using runtime context for dispatching versioned rules by dimension properties values.

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

The first rule, covering an Auto line of business, is as follows:

Rules void hello1(int hour)			
	createdOn	4/7/10	
	createdBy	**LOCAL**	
properties	**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 4: The Auto rule

Pay attention to the rule line with the LOB property.

The second rule, covering a Home line of business, is as 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 5: The Home rule

A wrapper enables the user to define which of these rules must be executed:

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

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

As a result, the code of the 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.RulesEngineFactory;
import org.openl.runtime.IEngineWrapper;
import org.openl.vm.IRuntimeEnv;
public class ExampleOfUsingRuntimeContext {

              public static void main(String[] args) {
                     //define the interface
                     RulesEngineFactory<simple> rulesFactory = new
RulesEngineFactory<Simple>("TemplateRules.xls", Simple.class);
                     Simple rules = (Simple) rulesFactory.newInstance();
                     // Getting runtime environment which contains context
                     IRuntimeEnv env = ((IEngineWrapper) rules).getRuntimeEnv();
                     // Creating context (most probably in future, the code will be different)
                     IRulesRuntimeContext context = RulesRuntimeContextFactory.
buildRulesRuntimeContext();                    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.

Using OpenL Tablets Projects from Java Code

OpenL Tablets projects can be instantiated via `SimpleProjectEngineFactory`. This factory is designed to be created via `SimpleProjectEngineFactoryBuilder`. A builder has to be configured. The main builder method is `setProject(String location)`. The project location folder has to be specified via this method.

The following example instantiates the OpenL Tablets project:

```
ProjectEngineFactory<Object> projectEngineFactory = new
SimpleProjectEngineFactory.SimpleProjectEngineFactoryBuilder<Object>().setProject(<project
location>) .build();
Object instance = projectEngineFactory.newInstance();
```

The above example instantiates the OpenL Tablets project generated in runtime interface. A method from instantiated project can be invoked via reflection mechanism. `ProjectEngineFactory` returns generated interface via the `getInterfaceClass()` method.

If a static interface must be used, the interface must be specified in `SimpleProjectEngineFactoryBuilder`. The following example illustrates how to instantiate a project with a static interface.

```
SimpleProjectEngineFactory<SayHello> simpleProjectEngineFactory = new
SimpleProjectEngineFactoryBuilder<SayHello>().setProject<project location>)
                .setInterfaceClass(SayHello.class)
                .build();
SayHello instance = simpleProjectEngineFactory.newInstance();
```

`SimpleProjectEngineFactoryBuilder` has additional methods to configure an engine factory. For example, the `setWorkspace()` method defines a project workspace for dependent projects resolving. The execution mode can be changed via the `setExecutionMode()` method. By default, the runtime execution mode is enabled. If the instance class needs to provide runtime context, it must be specified via `setProvideRuntimeContext(true)`.

OpenL Tablets WebStudio supports compilation of a module from project in single mode. A module can be compiled from a project in a single module via `setModule(String moduleName)`. If this method is used with the **single mode** module name, compilation for this module is used from the project.

Accessing a Test Table from Java Code

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:

```
RulesEngineFactory<?> rulesFactory = new RulesEngineFactory<?>("Tutorial_1.xls");
IOpenClass openClass = rulesFactory.getCompiledOpenClass();
IRuntimeEnv env = SimpleVMFactory.buildSimpleVM().getRuntimeEnv();
Object target = openClass.newInstance(env);
IOpenMethod method = openClass.getMatchingMethod("testMethodName", testMethodParams);
TestUnitsResults res = (TestUnitsResults) testMethod.invoke(engine, new Object[0], env);
```

Generating Java Classes from Datatype Tables

Some rules require complex data models as input parameters. Developers have to generate classes for each datatype defined in an Excel file for using them in a static interface as method arguments. The static interface can be used in engine factory. For more information on how to create and use a wrapper, see Using OpenL Tablets rules from Java Code.

Note: Datatype is an OpenL table of the Datatype type created by a business user. It defines a custom data type. Using these data types inside the OpenL Tablets rules is recommended as the best practice. For more information on datatypes, see ***[OpenL Tablets Reference Guide]***, the **Datatype Table** section.

To generate datatype classes, proceed as follows:

1. For Maven, configure the OpenL Maven plugin as described in Configuring the OpenL Maven Plugin and run the Maven script.
2. For Ant, configure the Ant task file as described in Configuring the Ant Task File and execute the Ant task file.

Configuring the OpenL Maven Plugin

To generate an interface for rules and datatype classes defined in the MS Excel file, add the following Maven configuration to the `pom.xml` file:

```
<build>
  [...]
  <plugins>
      [...]
      <plugin>
          <groupId>org.openl.rules</groupId>
          <artifactId>openl-maven-plugin</artifactId>
          <version>${openl.rules.version}</version>
          <configuration>
              <generateInterfaces>
                  <generateInterface>
                      <srcFile>src/main/openl/rules/TemplateRules.xls</srcFile>
                      <targetClass>
                        org.company.gen.TemplateRulesInterface
                      </targetClass>
                  </generateInterface>
              </generateInterfaces>
          </configuration>
          <executions>
              <execution>
                  <goals>
                      <goal>generate</goal>
                  </goals>
              </execution>
          </executions>
      </plugin>

  </plugins>
  [...]
</build>
```

In this case, classes and rules project descriptor, `rules.xml`, is generated on each Maven run on generate-sources phase.

Each `<generateInterface>` section has a number of parameters described in the following table.

<generateInterface> section parameters			
Name	**Type**	**Required**	**Description**
`srcFile`	String	true	Reference to the Excel file for which an interface class must be generated.
`targetClass`	String	true	Full name of the interface class to be generated. OpenL Tablets WebStudio recognizes modules in projects by interface classes and uses their names in UI. If there are multiple wrappers with identical names, only one of them is recognized as a module in OpenL Tablets WebStudio.
`displayName`	String	false	End user-oriented title of the file that appears in OpenL Tablets WebStudio. A default value is Excel file name without extension.
`targetSrcDir`	String	false	Folder where the generated interface class must be placed. An example is `src/main/java`. The default value is `${project.build.sourceDirectory}`.
`openlName`	String	false	OpenL Tablets configuration to be used. For OpenL Tablets, the `org.openl.xls` value must always be used. The default value is `org.openl.xls`.
`userHome`	String	false	Location of user-defined resources relative to the current OpenL Tablets project. The default value is `..`

<generateInterface> section parameters			
Name	**Type**	**Required**	**Description**
`userClassPath`	String	false	Reference to the folder with additional compiled classes that is imported by the module when the interface is generated. The default value is `null`.
`ignoreTestMethods`	boolean	false	If true, test methods are not added to interface class. It is used only in `JavaInterfaceAntTask`. The default value is `true`.
`generateUnitTests`	boolean	false	Parameter that overwrites the base generateUnitTests value.
`unitTestTemplatePath`	String	false	Parameter that overwrites the base unitTestTemplatePath value.
`overwriteUnitTests`	boolean	false	Parameter that overwrites the base overwriteUnitTests value.

For more configuration options, see the OpenL Tablets Maven Plugin Guide.

Configuring the Ant Task File

An example of the build file is as follows:

```
<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 datatypes"
		targetSrcDir="gen"
	>
	</openlgen>

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

</target>
</project>
```

When the file is executed, it automatically creates Java classes for datatypes 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 as described in the following table:

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

<openlgen> section parameters	
Parameter	**Description**
`displayName`	End user-oriented title of the file that appears in OpenL Tablets WebStudio.
`targetSrcDir`	Folder where the generated wrapper class must be placed.

Handling Data and Data Types in OpenL Tablets

This section includes the following topics about data and data types handling in OpenL Tablets:

- Datatype Lifecycle
- Inheritance in Datatypes
- Byte Code Generation at Runtime
- Java Files Generation
- OpenL Internals: Accessing a Datatype at Runtime and After Building an OpenL Wrapper
- Data Table

Datatype Lifecycle

Datatype lifecycle is as follows:

1. A Datatype table is created in the rules file.
   At runtime, Java class is generated for each datatype as described in Byte Code Generation at Runtime.
2. If Java classes are generated from a Datatype table as described in Generating Java Classes from Datatype Tables, the appropriate generated Java classes must be included in classpath as described in Java Files Generation.

Inheritance in Datatypes

In OpenL Tablets, one datatype can be inherited from another one. The new data type inherited from another one has access to all fields defined in the parent data type. If a child datatype contains fields defined in the parent datatype, and the field is declared with different types in the child and the parent datatype, warnings or errors appear.

The constructor with all fields of the child datatype contains all fields from the parent datatype, and the `toString`, `equals` and `hashCode` methods use all fields form the parent datatype.

Byte Code Generation at Runtime

At runtime, when OpenL Tablets engine instance is being built, for each datatype component, Java byte code is generated as described in Java Files Generation in case there are no previously generated Java files on classpath. It represents a simple Java bean for this datatype. This byte code is loaded to classloader so the object of type `Class<?>` can be accessed. When using this object through reflections, new instances are created and fields of datatypes are initialized. For more information, see the `DatatypeOpenClass` and `DatatypeOpenField` classes.

Attention! If Java class files for the datatypes on classpath are previously generated, they are used at runtime, regardless of changes made in Excel. To apply these changes, remove Java files and generate Java classes from the Datatype tables as described in Generating Java Classes from Datatype Tables.

Java Files Generation

As generation of datatypes is performed at runtime and developers cannot access these classes in their code, the mechanism described at Generating Java Classes from Datatype Tables is introduced. It allows generating Java files and putting them on the file system so users can use these data types in their code.

OpenL Internals: Accessing a Datatype at Runtime and After Building an OpenL Wrapper

After parsing, each data type is put to compilation context and becomes accessible for rules during binding. All data types are placed to `IOpenClass` of the whole module and are accessible from `CompiledOpenClass#getTypes` when the OpenL Tablets wrapper is generated.

Each `TableSyntaxNode` of the `xls.datatype` type contains an object of data type as its member.

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 the OpenL Tablets runtime API as a field of the `Rules` class instance

Data int numbers
this
Numbers
10
20
30
40
50

Figure 6: Simple data table

In this example, information in the data table can be accessed from the Java code as illustrated 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 file.

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 7: Datatype table and a corresponding data 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 file wrapper.

3.5 Customizing Table Properties

The OpenL Tablets design allows customizing available table properties. OpenL Tablets Engine employs itself to provide support of properties customization. The `TablePropertiesDefinitions.xlsx` file contains all declaration required to handle and process table properties.

Updating table properties requires recompiling the OpenL Tablets product. The developer has to contact the OpenL Tablets provider to retrieve the table properties file. When the changes are made, the developer has to send the file back to the provider, and a new OpenL Tablets package is delivered to the developer.

Alternatively, the developer can recompile OpenL Tablets from sources of their own.

Dispatching Table Properties

Previously selecting tables that correspond to the current runtime context are processed by the Java code. Now the rules dispatching is the responsibility of the generated Dispatcher decision table. Such table is generated for each group of methods overloaded by dimension properties. The Dispatcher table works like all decision tables so the first rule matched by properties is executed even if there are several tables matched by properties. Previously, in Java code dispatching, `AmbiguousMethodException` would be thrown in such case.

To support both functionalities, the `dispatching.mode` system property is introduces. It has the following possible values:

`dispatching.mode` property values	
Value	**Description**
java	Dispatching is processed by Java code. The benefit of such approach is stricter dispatching: if several tables are matched by properties, `AmbiguousMethodException` is thrown.
dt	Dispatching is processed by the Dispatcher decision table. The main benefit of this approach is performance: decision table is invoked much faster than Java code dispatching is performed.

If the system property is not specified or if the `dispatching.mode` property has an incorrect value, the Java approach is used by default.

Tables Priority Rules

To make tables dispatching more flexible, **tablesPriorityRules** DataTable in `TablePropertiesDefinitions.xlsx` is used. Each element of this table defines one rule of how to compare two tables using their properties to find more suitable table if several tables are matched by properties. Priority rules are used sequentially in comparison of two tables: if one priority rule gives result of the same priority of tables, the next priority rule is used.

Priority rules are used differently in the Dispatcher table approach and Java code dispatching but have the same sense: select suitable table if there are several tables matched by dimension Properties.

In case of the Dispatching table, priority rules are used to sort methods of an overloaded group. Each row of the Dispatcher table represents a rule, so after sorting, high priority rules are at the top of decision tables, and if several rows of the decision table are fired, only the first one, of the highest priority, is executed.

In case of Java code, dispatching priority rules is used after selecting tables that correspond to the current runtime context: all matched tables are sorted in order to select one with the highest priority. If it is impossible

to find the priority with the highest rule when several tables have the same priority and are of a higher priority than all other tables, `AmbiguousMethodException` is thrown.

There are two predefined priority rules and possibility to implement Java class that compares two tables using their properties:

- **min(<property name>)**
  A table that has lower value of property specified will have a higher priority. The property specified by name must be `instanceof Comparable<class of property value>`.
- **max(<property name>)**
  A table that has a higher value of property specified will have a higher priority. The property specified by name must be `instanceof Comparable<class of property value>`.

To specify the Java comparator of tables, the `javaclass:<java class name>` expression must be used. Java class must implement `Comparator<ITableProperties>`.

3.6 Tables Validation

The validation phase follows the binding phase and allows checking all tables for errors and accumulating all errors.

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

- status, which can be fail or success
- all error and warning messages that occurred

This section includes the following topics:

Table Properties Validators

The table properties that are described in `TablePropertyDefinition.xlsx` can have constraints. Some constraints have predefined validators associated with them.

To add a property validator, proceed as follows:

1. Add constraint as follows:
   1. Define constraint in `TablePropertyDefinition.xlsx`, in the constraints field.
   2. Create constraint class and add it to `ConstraintFactory`.
2. Create a validator as follows:
   1. Create a class of the validator and define it in the method `org.openl.codegen.tools.type.TablePropertyValidatorsWrapper.init()` constraint associated with the validator.
   2. If necessary, modify the velocity script `RulesCompileContext-validators.vm` in project `org.openl.rules.gen` that generates `org.openl.xls.RulesCompileContext`.
   3. To generate new `org.openl.xls.RulesCompileContext` with the validator, run `org.openl.codegen.tools.GenRulesCode.main(String[])`.
3. Write unit tests.

Existing Validators

The existing validators are as follows:

- **Unique in module validator** verifies uniqueness in a module of a property.
- **Active table validator** verifies correctness of an "active" property.
  There can be only one active table validator per active table.
- **Regular expression validator** verifies string properties matching against the predefined regex pattern.
- **Gap/overlap validator** makes gap and overlap analysis for decision tables with the **validateDT** property set to **on**.
- **Dimension properties validator**

3.7 Module Dependencies: Classloaders

The dependency class resolution mechanism is implemented using specialized classloading.

Each dependency has its own Java classloader so all classes used in compiling a specified module, including generated datatype Java classes, are stored in the dependency classloader.

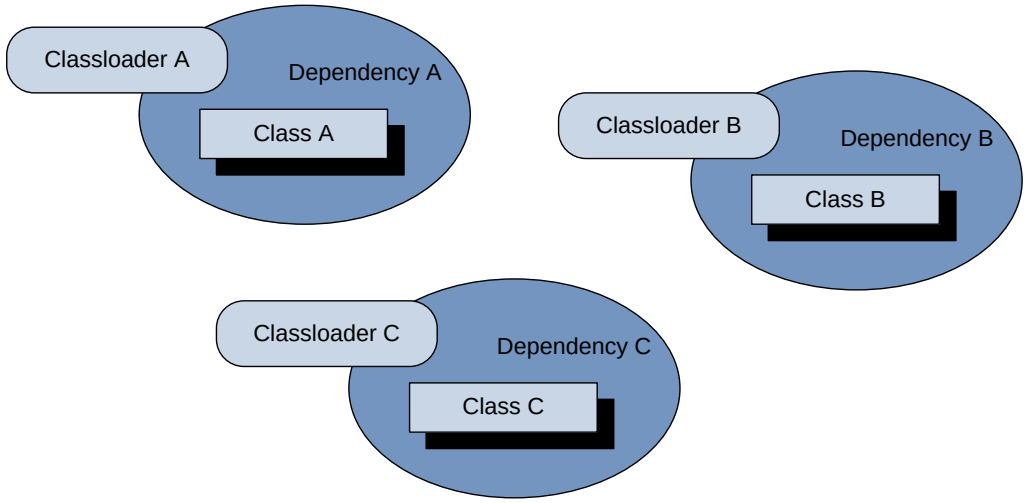

Figure 8: Dependency classloaders

The root module contains references to all its dependencies classloaders. When loading any class, the following algorithm is executed:

1. Get all dependencies classloaders.
2. Search for the required class in each dependency classloader, one by one.
3. If a class is found, return it.
4. If a class does not exist, search for the class by its classloader.

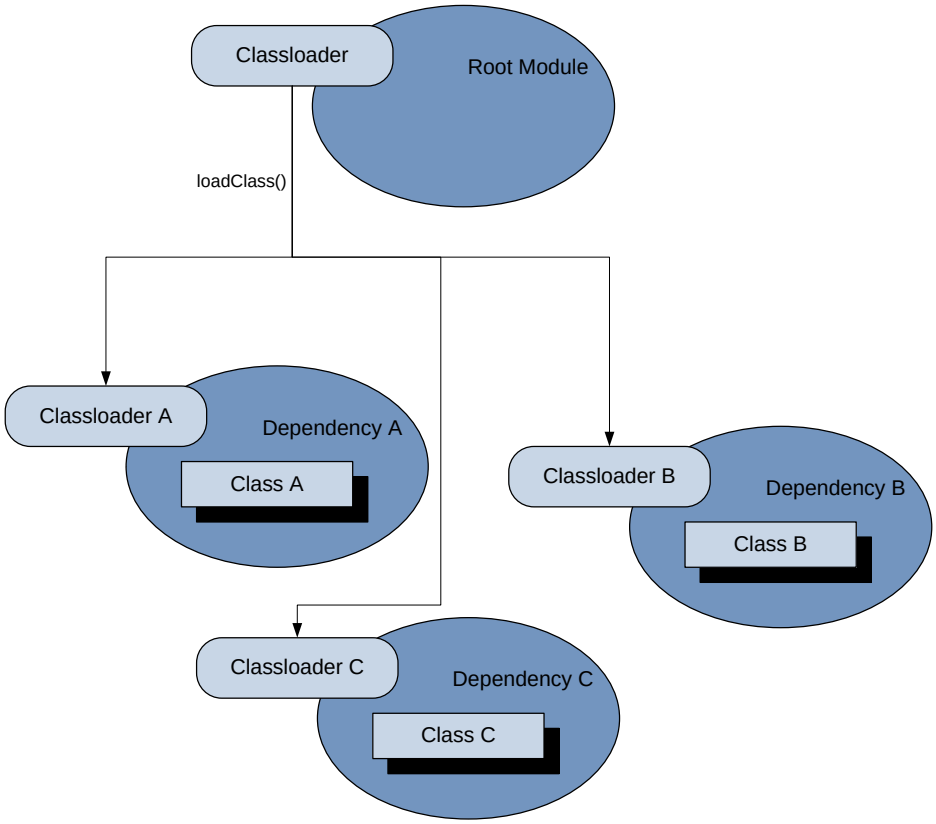

Figure 9: Load class from root module

For the dependency management feature, provide an appropriate `DependencyManager` object to the entry point for the OpenL Tablets compilation.

Note: Using the same class in two classloaders can cause an error because the class will be loaded by two different classloaders.

3.8 Peculiarities of OpenL Tablets Implementation

This section describes OpenL Tablets implementation specifics and includes the following topics:

- Lookup Tables Implementation Details
- Range Types Instantiation

Lookup Tables Implementation Details

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

To work with this kind of a table, the `TransformedGridTable` object is created with the constructor parameters it had in the original grid table of the lookup table, without a header, and `CoordinatesTransformer` that converts table coordinates to work with both vertical and horizontal conditions.

As a result, a `GridTable` is received. It works as a decision table structure. All coordinate transformations with lookup structure go inside. The work with columns and rows is based on the physical, not logical, structure of the table.

Range Types Instantiation

`IntRange` can be created in one of the following ways:

`IntRange` creation methods	
Format	**Description**
`new IntRange(int min_number, int max_number)`	Covers all numbers between `min_number` and `max_number`, including borders.
`new IntRange(Integer value)`	Covers only a given value as the beginning and the end of the range.
`new IntRange(String rangeExpression)`	Borders are parsed by formats of `rangeExpression`.

The same formats and restrictions are used in `DoubleRange`.