EIS GROUP™

Reference Guide
OpenL Mapping Framework
Version 1.1.10

Document number: TP_OpenL_Mapper_RG_1.0_LSh

Revised: 08-12-2015

Table of Contents

1 Preface

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

The following topics are included in this preface:

1.1 Related Information

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

Related information	
Title	**Description**
[OpenL Tablets Reference Guide]	Document that provides overview of the OpenL Tablets technology, its basic concepts, and principles.
http://dozer.sourceforge.net/	Dozer framework home page.
http://openl-tablets.sourceforge.net/	OpenL Tablets home page.
http://openl-tablets.sourceforge.net/mapper	Mapper home page.

1.2 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.
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 the Mapping Framework

A **mapping framework** (MF) recursively copies data from one object to another. Typically, these data objects are of different complex types.

MF is built using the Dozer framework with several changes as a mapping engine, and OpenL Rules Tablets framework as a tool which provides a convenient mechanism to define conversion rules in declarative way.

The following picture demonstrates the framework lifecycle and displays how underlying frameworks are used by mapper.

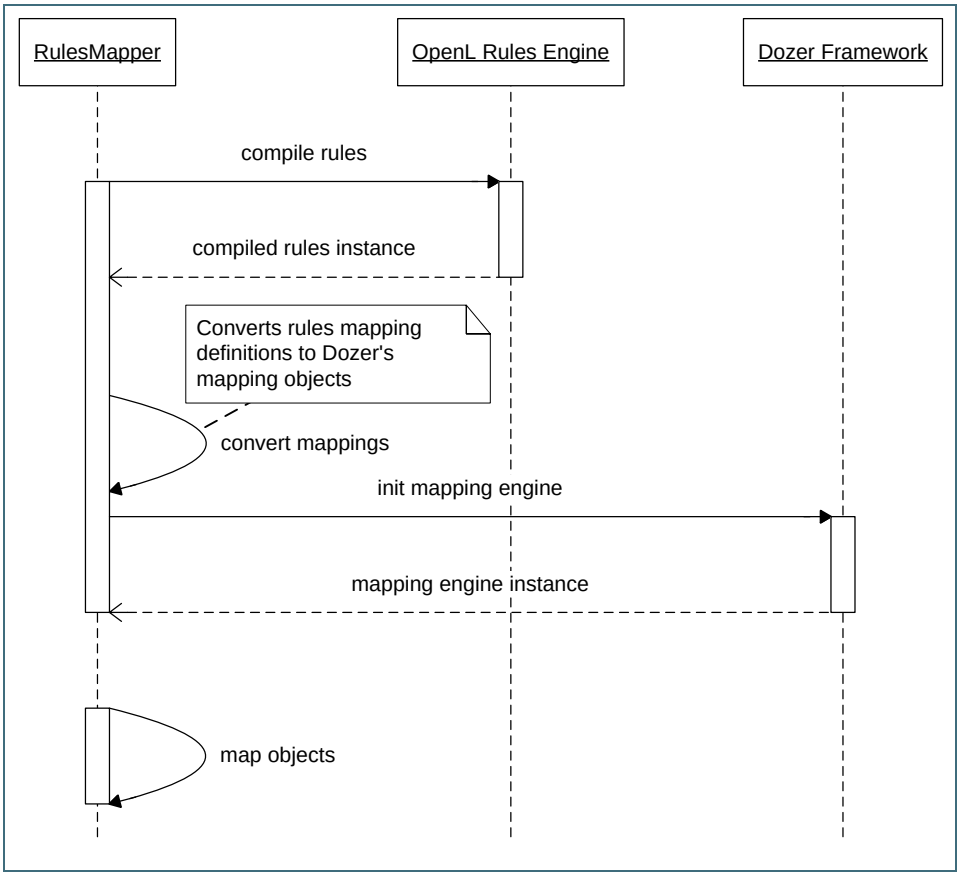

Figure 1: Mapper lifecycle sequence diagram

The mapper is used any time when one type of Java Bean must be taken and mapped to another type of Java Bean. Most field mappings can be done automatically by mapper using reflection.

MF supports simple property mapping, complex type mapping, bi-directional mapping, implicit-explicit mapping, as well as recursive mapping. This includes mapping collection attributes that also need mapping at the element level.

3 Getting Started

Proceed as follows:

1. Add the following dependency to the project's pom:

```
<dependency>
  <groupId>org.openl.rules</groupId>
  <artifactId>org.openl.rules.mapping.dev</artifactId>
  <version>1.0.0</version>
</dependency>
```

2. Create mapping rules in Excel file, for example:

Data Mapping mappings			
classA	**classB**	**fieldA**	**fieldB**
Class A	**Class B**	**Field A**	**Field B**
Source	Destination	aStringField	aStringField
Source	Destination	anIntegerField	anIntegerField

Figure 2: Mapping definition

3. Define one more table:

Environment	
import	org.openl.rules.mapping
	source.package
	destination.package

Figure 3: Import definition

`source.package and destination.package` are packages containing `Source and Destination` classes.

4. If necessary, define full Java class names.

   Examples of class names are `source.package.Source` and `destination.package.Destination`.

5. Add the following code snippet to the code to use mapper:

```
File mappingRules= new File("mapping.xlsx");
Mapper mapper = RulesBeanMapperFactory.createMapperInstance(mappingRules);

Source sourceInstance = new Source();
sourceInstance.setAStringField("string");
sourceInstance.setAnIntegerField(10);

Destination destinationInstance = mapper.map(sourceInstance, Destination.class);
```

When mapping is completed, a new instance of the `destination.package.Destination` class is returned with following fields values: `"string"` for `aStringField` field and `10` for `anIntegerField`.

4 Field Mapping Algorithm Overview

The following picture illustrates algorithm which is used during field mapping:

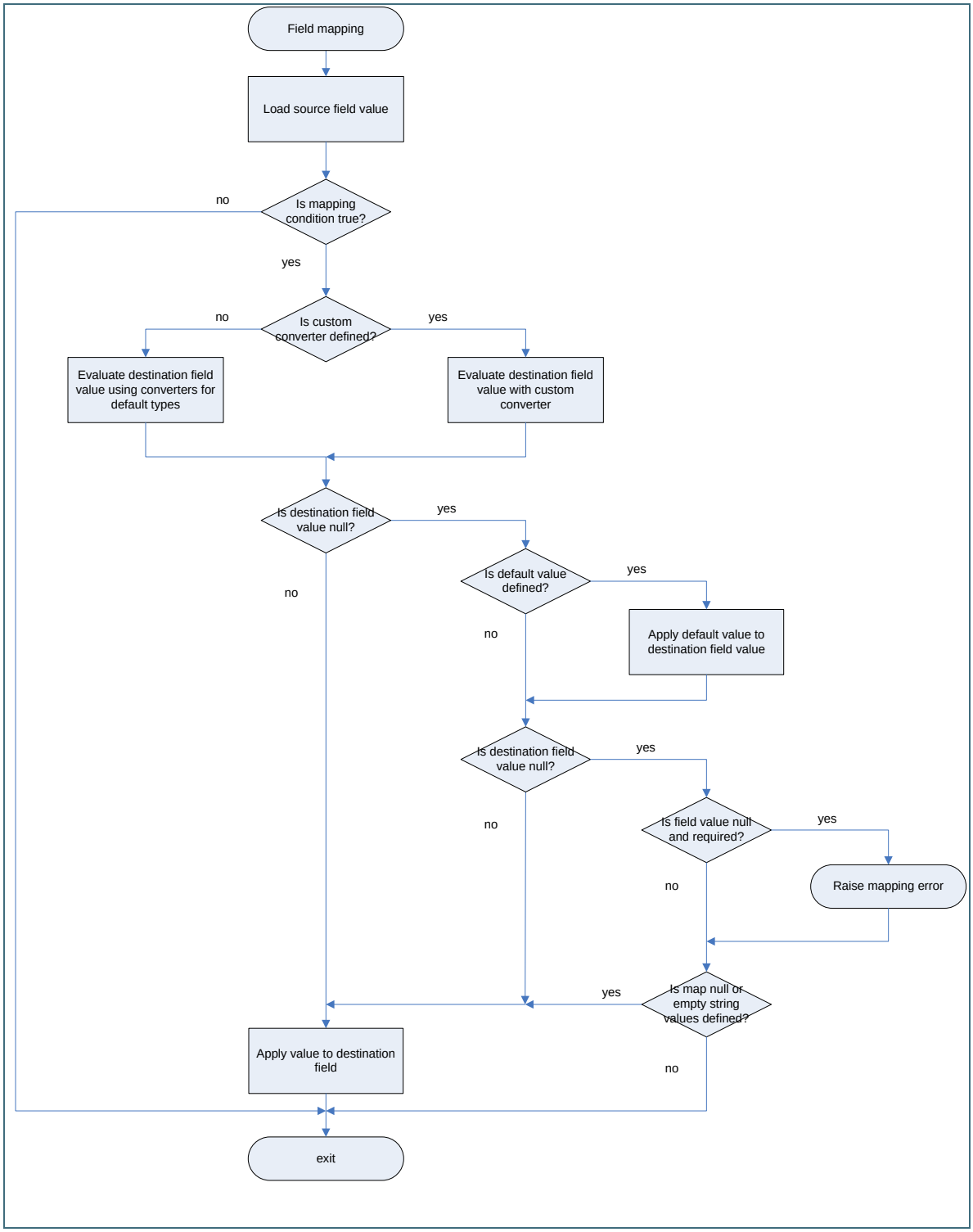

Figure 4: Single field mapping algorithm

5 Mappings

This chapter describes mappings and includes the following topics:

5.1 Rules File Format

Mapping file is a valid OpenL Tablets rules file. It must contain tables with all required mapping configuration and custom converter implementation.

5.2 Basic Property Mapping

This section describes basic property mapping and includes the following topics:

Simple Property Mapping

To map `srcField` of the `source.package.Source` class into `destField` of the `destination.package.Destination` class, provide a mapping rule such as the following one:

Data Mapping mappings			
classA	**classB**	**fieldA**	**fieldB**
Class A	**Class B**	**Field A**	**Field B**
source.package.Source	destination.package.Destination	srcField	destField

Figure 5: Simple property mapping

To map `destField` of the `destination.package.Destination` class into `srcField` of `source.package.Source` class, do not define a new mapping rule because all mappings are bi-directional by default.

Implicit Property Mapping

By default, mapping framework maps properties with matching names. For example, a source object has a field with a name `myField` and a destination object has a matching field `myField`. In this case, the mapping framework performs mapping for these fields automatically. To change the behavior of this feature, use the **wildcard** configuration parameter as described in Configuration.

Recursive Mapping

The mapping framework supports full class level mapping recursion. For any complex types defined at the field level mappings in the object, the mapping processor will search the mappings for a class level mapping between the two mapped classes. If no mappings are found, the processor only maps fields of the same name between the complex types.

Data Type Conversion

Data type conversion is performed automatically by the mapping engine. The following types of bi-directional conversions are supported:

- PrimitivetoPrimitiveWrapper
- PrimitivetoCustomWrapper
- PrimitiveWrappertoPrimitiveWrapper
- PrimitivetoPrimitive
- ComplexTypetoComplexType
- StringtoPrimitive
- StringtoPrimitiveWrapper
- String to Complex Type if the Complex Type contains a String constructor
- StringtoMap
- CollectiontoCollection
- CollectiontoArray
- MaptoComplexType
- MaptoCustomMapType
- EnumtoEnum

The following data types can be mapped to each other:

- `java.util.Date`
- `java.sql.Date`
- `java.sql.Time`
- `java.sql.Timestamp`

- `java.util.Calendar`
- `java.util.GregorianCalendar`

String can be mapped to any of the supported `Date`/`Calendar` objects.

Objects containing the `toString()` method that produces a long representing time in milliseconds can be mapped to any supported `Date`/`Calendar` object.

String to Date Conversion

A date format for the String can be specified at the field level so that the necessary data type conversion can be performed.

Data Mapping mappings				
classA	**classB**	**fieldA**	**fieldB**	**fieldADateFormat**
ClassA	**ClassB**	**FieldA**	**FieldB**	**Field A Date Format**
Source	Destination	stringField	dateField	yyyy-MM-dd

Figure 6: Date format configuration at the field level

To define the date format at the class level and global level, use the **dateFormat** configuration parameter as described in Configuration.

5.3 One Way Mapping

To map a field pair in one way only, set it at the field level using **oneWay** parameter.

Data Mapping mappings				
classA	**classB**	**fieldA**	**fieldB**	**oneWay**
Class A	**Class B**	**Field A**	**Field B**	**Is one way?**
Source	Destination	srcField	destField	true

Figure 7: One way mapping definition

If a one-way mapping is specified, mapping from `destField` into `srcField` is ignored.

5.4 Custom Converters

Custom converters are used to perform custom mapping between two objects. When a custom converter is specified for a class A and class B combination, the mapper invokes the custom converter to perform the data mapping instead of the standard mapping logic. A custom converter can be defined in different ways.

The following topics are included in this section:

Using the OpenL Tablets Method for Convert Method Definition

The following table describes how to define a convert method using the OpenL Tablets method.

Data Mapping mappings				
classA	**classB**	**fieldA**	**fieldB**	**convertMethodAB**
Class A	**Class B**	**Field A**	**Field B**	**Convert method (a into b)**
Source	Destination	srcField	destField	typeCdLookup

Figure 8: Convert method definition using the OpenL Tablets method

The **convertMethodAB** parameter tells the mapper that the `typeCdLookup` convert method must be used for the current field pair. The following table describes the convert method defined using the OpenL Tablets rules table component:

Rules String typeCdLookup(String key, String dest)	
C1	**RET**
key	
String	
Codes	Values
HOME	03
CO	06
TE	04
	=error ("No PolicyTypeCd lookup for key " + key); ""

Figure 9: Custom converter definition using the OpenL Tablets component

For more information on defining a convert method using the OpenL Tables rule table component, see ***[OpenL Tablets Reference Guide].***

Rules for Convert Method Definition

The following rules apply to the convert methods definition:

- A method has the same name as defined by **convertMethodAB** parameter.
- A method provides two parameters; the first one is assignable from the source field type, and the second one is assignable from the destination field type.
- The destination field must be assignable from the method return type.

The convert method cannot be used during reverse mapping. To map from a field B into a field A, provide the appropriate convert method using the **convertMethodBA** parameter.

Using a Static Java Method as a Custom Converter

To use a static Java method as a convert method, provide a class name and a method name.

For example, `utils.ConverterUtils.typeCdLookup` or `ConverterUtils.typeCdLookup` (if the second example is used, define the `utils` package in the import section of the Environment table).

Data Mapping mappings				
classA	**classB**	**fieldA**	**fieldB**	**convertMethodAB**
Class A	**Class B**	**Field A**	**Field B**	**Convert method (a into b)**
Source	Destination	srcField	destField	ConverterUtils.typeCdLookup

Figure 10: Convert method definition using the Java static method

Defining a Custom Converter Class

For more advanced functionality, define a custom converter implementation class. In this case, implement the `org.dozer.CustomConverter` interface in order for the mapper to accept it; otherwise, an exception is thrown.

```
public interface CustomConverter {

Object convert(Object existingDestinationFieldValue,
Object sourceFieldValue,
Class<?> destinationClass,
Class<?> sourceClass);

}
```

Note: Custom converters get invoked even when the source value is null, so the null values in the custom converter implementation must be handled explicitly.

When the custom converter implementation is created, register it for usage in mapper. An example is as follows:

```
Map<String, CustomConverter> converters =
                                    new HashMap<String, CustomConverter>();

converters.put("isExists", new CustomConverter(){
      public Object convert(Object existingDestinationFieldValue,
                            Object sourceFieldValue,
                            Class<?> destinationClass,
                            Class<?> sourceClass) {
            return sourceFieldValue != null;
      }
});

File source = new File("CustomConvertersWithIdTest.xlsx");
Mapper mapper = RulesBeanMapperFactory.createMapperInstance(source, converters, null);
…
```

The following mapping demonstrates how a custom converter can be referenced by ID from the registry of custom converters:

Data Mapping mappings				
classA	**classB**	**fieldA**	**fieldB**	**convertMethodABId**
Class A	**Class B**	**Field A**	**Field B**	**Convert method (a into b)**
Source	Destination	srcField	destField	isExists

Figure 11: Convert method definition by converter ID

Using a Class Level Custom Converter

Custom converters can be defined at the class level.

Data Converter defaultConverters		
classA	**classB**	**convertMethod**

Class A	Class B	Convert method
Source	Destination	ConverterUtils.typeCdLookup

Figure 12: Default custom converters definition

Defined default converter is used by a mapping processor when an appropriate class pair must be mapped but the custom converter at the field level is not defined.

Custom Converter Search Algorithm

Mapper uses the following order to find appropriate converter:

- Get a custom converter by its ID.
- Get a custom converter by name, using either a rule or a Java method.
- Get a default custom converter.

5.5 Field Mapping Conditions

Field mapping conditions are used at runtime to decide whether field mapping must be executed. When a field mapping condition is specified for a field A and field B combination, a mapper invokes it at the start of field mapping flow as described in Field Mapping Algorithm Overview.

Mapping conditions can be defined in different ways.

The following topics are included in this section:

- Using the OpenL Tablets Method to Define a Field Mapping Condition
- Rules for Field Mapping Condition Definition
- Using a Static Java Method as a Condition
- Defining a Custom Condition Class
- Conditions Search Algorithm

Using the OpenL Tablets Method to Define a Field Mapping Condition

The following table describes how to define a field mapping condition using the OpenL Tablets method.

Data Mapping mappings				
classA	**classB**	**fieldA**	**fieldB**	**conditionAB**
Class A	**Class B**	**Field A**	**Field B**	**Map field (a into b)**
Source	Destination	srcField	destField	mapField

Figure 13: Field mapping condition method definition using the OpenL Tablets method

The **conditionAB** parameter tells to mapper that for the current field pair mapping, the `mapField` method must be invoked to make a decision. If the method result is `true`, the mapper executes mapping rule; otherwise, it skips the rule.

An example of condition method definition using OpenL Tablets method table component is as follows:

Method boolean mapField(Object src, Object dest)
return src != null;

Figure 14: Field mapping condition example

For more information on defining a convert method using the OpenL Tables rule table component, see ***[OpenL Tablets Reference Guide].***

Rules for Field Mapping Condition Definition

The following rules apply to the convert methods definition:

- A method has the same name as defined by the **conditionAB** parameter.
- A method provides two parameters; the first one assignable from the source field type, and the second one assignable from the destination field type.
- A method return type must be of the Boolean type.

The condition method is not used during reverse mapping. To use the condition during mapping from field B into field A, provide an appropriate method using the **conditionBA** parameter.

Using a Static Java Method as a Condition

To use a static Java method as a condition method, provide a class name and a method name.

For example, `utils.ConditionUtils.mapField` or `ConditionUtils.mapField` (if the second example is used, define the `utils` package in the import section of the Environment table).

Data Mapping mappings				
classA	**classB**	**fieldA**	**fieldB**	**conditionAB**
Class A	**Class B**	**Field A**	**Field B**	**Condition method (a into b)**
Source	Destination	srcField	destField	ConditionUtils.mapUtils

Figure 15: Condition method definition using a Java static method

Defining a Custom Condition Class

To define a custom field mapping condition implementation class, implement the `org.dozer.FieldMappingCondition` interface for the mapper to accept it. Otherwise, an exception will be thrown.

```
public interface FieldMappingCondition {

      boolean mapField(Object sourceFieldValue,
            Object destFieldValue,
            Class<?> sourceType,
            Class<?> destType);
}
```

Note: A field mapping condition gets invoked even when the source value is null, so the null values must be explicitly handled in the implementation.

When the implementation is created, register it for usage in mapper. An example is as follows:

```
Map<String, FieldMappingCondition> conditions =
                                    new HashMap<String, FieldMappingCondition>();
conditions.put("mapField", new FieldMappingCondition() {
```

```
        public boolean mapField(Object sourceFieldValue,
                                Object destFieldValue,
                                Class<?> sourceType,
                                Class<?> destType) {
            return sourceFieldValue !=null;
        }
    });

    File source = new File("FieldMappingConditionsWithIdTest.xlsx");
    Mapper mapper = RulesBeanMapperFactory.createMapperInstance(source, null, conditions);
    …
```

The following mapping illustrates how a condition can be referenced by ID from the conditions registry:

Data Mapping mappings				
classA	**classB**	**fieldA**	**fieldB**	**conditionABId**
Class A	**Class B**	**Field A**	**Field B**	**Condition method (a into b)**
Source	Destination	srcField	destField	mapField

Figure 16: Condition method definition by ID

Conditions Search Algorithm

The mapper uses the following order to find the appropriate field condition:

- Get a condition by its ID.
- Get a condition by name, using either a rule or a Java method.

5.6 Default Values

At the field mapping level, a default value for the destination field can be defined. It will be set into destination property if the source value is `null` as described in Field Mapping Algorithm Overview.

Data Mapping mappings					
classA	**classB**	**fieldA**	**fieldB**	**fieldADefaultValue**	**fieldBDefaultValue**
Class A	**Class B**	**Field A**	**Field B**	**Field A Default Value**	**Field B Default Value**
Source	Destination	srcField	destField	Some value	Another value

Figure 17: Field default value definition

A default value is a string value which will be converted into field's type. Currently, the default value string can be converted into the following types:

- primitive types, that is, int, double, short, char, long, Boolean, byte, and float
- wrapper types, that is, `java.lang.Integer`, `java.lang.Double`, `java.lang.Short`, `java.lang.Character`, `java.lang.Long`, `java.lang.Boolean`, `java.lang.Byte`, and `java.lang.Float`
- `java.math.BigDecimal`
- `java.math.BigInteger`
- `java.lang.String`
- `java.util.Date`
- `java.util.Calendar`
- complex types with a constructor with the `java.lang.String` parameter provided

5.7 Empty Source Mapping

Empty source mapping is a special case of field initialization. Typically empty source mapping is used for constant definition.

Data Mapping mappings					
classA	**classB**	**fieldA**	**fieldB**	**oneWay**	**fieldBDefaultValue**
Class A	**Class B**	**Field A**	**Field B**	**Is one way?**	**Field B Default Value**
Source	Destination		destField	true	My constant

Figure 18: Empty source mapping definition

Empty source mapping is always the one-way mapping. It must be ignored during reverse mapping.

5.8 Multi Source Mapping

Using multi source mapping is beneficial for data calculation using several sources of input data.

A custom converter must be provided for this type of field mapping.

Data Mapping mappings					
classA	**classB**	**fieldA**	**fieldB**	**oneWay**	**convertMethodAB**
Class A	**Class B**	**Field A**	**Field B**	**Is one way?**	**Convert method (a into b)**
Source1	Dest	field1	destField	true	multiSourceFieldConverter
		field2			
		field3			

Figure 19: Multi source mapping definition

An array of source field values is used as a value for the convert method first parameter. The second one is the existing destination value.

Method Object multiSourceFieldConverter(Object[] src, Object dest)

```
String[] result = new String[src.length];

for (inti = 0; i< src.length; i++) {
if (src[i] != null) {
    result[i] = src[i].toString();
}
}

return result;
```

Figure 20: Custom converter for multi source mapping

Multi source mapping is **always** one-way mapping. It must be ignored during reverse mapping.

5.9 Index Mapping

This section describes index mapping. Fields that need to be looked up or written to by indexed property are supported.

The following topics are included in this section:

Simple Index

Mapping using simple index uses the index operator to locate the required element in an array or collection object.

Data Mapping mappings			
classA	**classB**	**fieldA**	**fieldB**
Class A	**Class B**	**Field A**	**Field B**
Source	Destination	anArrayField[1]	firstField
Source	Destination	aListField[1]	secondField

Figure 21: Simple index mapping definition

Simple index value is 1-based integer value, that is, index starts with the value 1 and the first element is at 1 index at the array or collection.

Simple index can be used for destination field definition as usually.

Mapping to the End of Collection

Mapping to the end of collection is a special case of simple index and it is defined as a **zero** index. It can be used to map source data to the destination element and append destination element to the collection. Note that zero is not the index of the last element. Mapping processor creates a new element, maps data to the element, and appends it to the end of collection if this type of index value is used.

Data Mapping mappings			
classA	**classB**	**fieldA**	**fieldB**
Class A	**Class B**	**Field A**	**Field B**
Source	Destination	anArrayField[1]	destField[0]
Source	Destination	anArrayField[2]	destField[0]

Figure 22: Mapping to the end of collection

Index of the real element is calculated by mapping processor every time when the zero index appears. This type of index cannot be used for deep mapping definition because the index value is changed at runtime. If it is used in deep mapping, an exception is thrown. It cannot be used for the source field path definition because in this case, a mapper always returns the `null` value.

Expression Index

The mapper allows defining an expression to look up for an array or collection element. The defined expression must be the JXpath compliant filter expression.

Data Mapping mappings			
classA	**classB**	**fieldA**	**fieldB**

Class A	Class B	Field A	Field B
Source	Destination	anArrayField[@name='John']	firstField

Figure 23: Expression index mapping definition

The expression index cannot be used for destination field definition. Field mapping, which uses expression index, must be marked as one-way mapping.

5.10 Collection Item Discriminator

A collection item discriminator allows defining a destination collection element at runtime when the source object is mapped to the collection or array. For example, the target collection contains list of policies and the collection element representing the required policy is not known. In this case, use the following field mapping definition:

Data Mapping mappings				
classA	**classB**	**fieldA**	**fieldB**	**fieldBDiscriminator**
Class A	**Class B**	**Field A**	**Field B**	**Field B Collection Item Discriminator**
Source	Destination	array	list	discriminator

Figure 24: Collection item discriminator usage example

The mapping processor maps the collection into collection through the following steps:

1. Get the source object or the next element from the source collection.
2. Get the target element from the destination collection.
   If the collection item discriminator is provided, it is used to get the target element.
3. If the target element is not found, for example, discriminator is not defined or discriminator returned a `null` value, create a new one and add it to the end of the destination collection.
4. Map the source element to the target element.

A discriminator can be defined as an OpenL method or as a static Java method. An example is as follows:

```
Method Object discriminator(Object src, List dest)

// target element resolving goes here …

return null;
```

Figure 25: Collection item discriminator method example

If the destination field type is an array of primitives, the mapping processor always adds the mapped element as a new one to the end of the array.

5.11 Deep Mapping

The mapper can map deep properties. An example is mapping an object with a String property to the other object with a String property that is several levels deep within the object graph.

Data Mapping mappings			
classA	**classB**	**fieldA**	**fieldB**
Class A	**Class B**	**Field A**	**Field B**

Source	Destination	srcField.stringField	destField.nestedField.stringField

Figure 26: Deep mapping usage example

The following topics on deep mapping are included in this section:

- Deep Mapping Using the Field Type
- Using Type Hints for Deep Mapping

Deep Mapping Using the Field Type

The field type parameter enables defining the type to be used for the current field.

For example, consider `srcField` defined as a `BaseType` type and `destField` defined as a `DestType` type. The `DestType` type is a super type of the `CustomType` type. At runtime, `destField` is `CustomType` value, and the mapper must be told to map `srcField` into `destField` as the `BaseType` value into `CustomType` value instead of `BaseType` value into `DestType` value, because, for example, `CustomType` value has extra data that must be mapped. In this case, use the **fieldBType** parameter to define the required type of the field.

Data Mapping mappings				
classA	**classB**	**fieldA**	**fieldB**	**fieldBType**
Class A	**Class B**	**Field A**	**Field B**	**Field B Type**
Source	Destination	srcField	destField[0]	CustomType

Figure 27: Field type casting example

The field type can be used for destination field and collection elements only.

Using Type Hints for Deep Mapping

The type hints concept is the same as the field type concept but provided for a deep mapping use case.

The field type hints value is an array of class names that will be used as a type of the appropriate field in the defined field path. For more information on array definition, see ***[OpenL Tablets Reference Guide]***, section **Arrays in OpenL Tablets.**

Data Mapping mappings				
classA	**classB**	**fieldA**	**fieldB**	**fieldBHint**
Class A	**Class B**	**Field A**	**Field B**	**Field B Hint**
Source	Destination	srcField.stringField	destField.nestedField.stringField	DestType, NestedType

Figure 28: Type hints usage example

The field type hints can be simplified to skip the class type definition for the appropriate field in the defined field path.

Data Mapping mappings				
classA	**classB**	**fieldA**	**fieldB**	**fieldBHint**
Class A	**Class B**	**Field A**	**Field B**	**Field B Hint**
Source	Destination	srcField.stringField	destField.nestedField.stringField	,NestedType

Figure 29: Simplified type hints definition example

Note that the leading comma is not omitted to keep correspondence between fields and hints. Trailing commas can be omitted, for example, `NestedType, , ,` is equal to `NestedType`, but `NestedType,,,OtherType` is not equal to `NestedType,OtherType`.

5.12 Configuration

Configurations are used to set default mapping parameters. By default, the mapper uses the following policies during mapping process:

Mapping default policies		
Name	**Value**	**Description**
MapNulls	true	Defines whether null values will be skipped or mapped as usually.
MapEmptyStrings	true	Defines whether empty strings will be skipped or mapped as usually.
TrimStrings	false	Defines whether trim operation will be executed on strings.
Fields are required	false	Defines that fields are required and they cannot be null.
Apply wildcard	true	Defines that mapping processor will map matching fields implicitly.

In accordance with business needs, policies can be changed with configuration components. Configuration can be applied at global level and at class level. Configurations are optional.

Data GlobalConfiguration globalConfiguration					
mapNulls	**trimStrings**	**mapEmptyStrings**	**requiredFields**	**dateFormat**	**wildcard**
Map Nulls	**Trim Strings**	**Map Empty Strings**	**Fields are required**	**Date format**	**Wildcard**
FALSE	TRUE	TRUE	FALSE	MM-dd-yyyy	FALSE

Figure 30: Mapping process configuration definition at global level

Global configuration must be the only one defined. If there are several definitions, an exception is thrown.

Data ClassMappingConfiguration classConfiguration							
classA	**classB**	**mapNulls**	**trimStrings**	**mapEmptyStrings**	**requiredFields**	**dateFormat**	**wildcard**
Class A	**Class B**	**Map Nulls**	**Trim Strings**	**Map Empty Strings**	**Fields are required**	**Date format**	**Wildcard**
A	B	FALSE	TRUE	FALSE	FALSE	MM-dd-yyyy	FALSE

Figure 31: Mapping process configuration definition at class level

Class level configuration will be used by mapper for the specified class pair mapping independent of mapping direction.

5.13 Configuration Priority

Mapping processor uses the following order to get the appropriate mapping configuration:

- field level configuration
- class level configuration
- global level configuration
- default mapping policy

5.14 Custom Bean Factories

The mapper can be configured to use custom bean factories to create new instances of destination data objects during the mapping process. By default, the mapper only creates a new instance of any destination objects using a default constructor. Specify your own bean factories to instantiate the data objects.

The custom bean factory must implement the `org.dozer.BeanFactory` interface.

```
public interface BeanFactory {
      Object createBean(Object source,
                        Class<?> sourceClass,
                        String targetBeanId);

}
```

Then update configuration for the target class.

Data ClassMappingConfiguration classConfiguration			
classA	**classB**	**classABeanFactory**	**classBBeanFactory**
Class A	**Class B**	**Class A Bean Factory**	**Class B Bean Factory**
A	B	ClassABeanFactory	ClassBBeanFactory

Figure 32: Bean factory usage example

The Dozer framework provides `org.dozer.factory.JAXBBeanFactory` and `org.dozer.factory.XMLBeanFactory` classes to support JAXB and XMLBeans objects.

5.15 Create Method

The mapper can be configured to use custom static create methods to create new instances of destination data objects during the mapping process.

Data Mapping mappings				
classA	**classB**	**fieldA**	**fieldB**	**fieldACreateMethod**
Class A	**Class B**	**Field A**	**Field B**	**Field A Create Method**
Source	Destination	srcField	destField	CreateMethodClass.createMethod

Figure 33: Field type casting example

`Create` method is a java static method with no parameters.

```
public class CreateMethodClass {

      public static CustomType createMethod() {
            return new CustomType();
    }
}
```

5.16 Mapping Inheritance

This section describes mapping inheritance principles and includes the following topics:

- Introducing Super Class Mapping

Introducing Super Class Mapping

When mapping subclasses with base class attributes, individual mappings for these attributes are not required if they are already defined for super class mapping. For example, consider the following classes:

```
public class Source {
	private String firstField;
	private String secondField;

…
}

public class ParentDest {
	private String firstField;

…
}

public class ChildDest extends ParentDest {
	private String secondField;

	…
}
```

For these classed, define the following mapping:

Data Mapping mappings			
classA	**classB**	**fieldA**	**fieldB**
Class A	**Class B**	**Field A**	**Field B**
Source	ParentDest	firstField	firstField
Source	ChildDest	secondField	secondField

Figure 34: Mapping attributes of a parent class

In case of mapping the `Source` object to `ChildDest`, the mapping processor uses mappings of the `Source-ParentDest` class pair to map attributes of the super class, that is, for the previously described example, `Source.firstField` into `ParentDest.firstField`.

Overriding Super Mapping Definition

Super mapping definition can be overridden by defining a new field mapping which uses the same field paths and appropriate class pair.

Data Mapping mappings				
classA	**classB**	**fieldA**	**fieldB**	**convertMethodAB**
Class A	**Class B**	**Field A**	**Field B**	**Convert Method (a into b)**
Source	ParentDest	firstField	firstField	
Source	ChildDest	firstField	firstField	customConvertMethod

Figure 35: Overriding mapping attributes of a parent class

In this example, the mapping processor will use the first field mapping definition for the `Source` and `ParentDest` classes, and the second one for the `Source and ChildDest` classes.

5.17 Context Based Mapping

This section describes the context based mapping. **Mapping ID** defines the field mapping definitions to be used by the mapping processor, while **mapping parameters** define how the appropriate field mappings will be processed.

The following topics are included:

Mapping Parameters

Mapping flow can be changed by a user at runtime using mapping parameters. Mapping parameters can be used by custom converters and field mapping conditions.

Data Mapping mappings				
classA	**classB**	**fieldA**	**fieldB**	**convertMethodABId**
Class A	**Class B**	**Field A**	**Field B**	**Convert method (a into b)**
Source	Destination	srcField	destField	convertMethodId

Figure 36: Convert method definition by converter ID

When using mapping parameters, provide implementation of the `org.dozer.BaseMappingParamsAwareCustomConverter` class. `BaseMappingParamsAwareCustomConverter` defines the extended **convert** method which must be implemented by user. The following example illustrates how the mapping parameters can be used:

```
Map<String, CustomConverter> converters =
                                    new HashMap<String, CustomConverter>();
converters.put("convertMethodId", new BaseMappingParamsAwareCustomConverter(){

      public Object convert(MappingParameters params,
                            Object existingDestinationFieldValue,
                            Object sourceFieldValue,
                            Class<?> destinationClass,
                            Class<?> sourceClass) {
            return params.get("value");
      }
  });

File source = new File("mapping.xlsx");
Mapper mapper = RulesBeanMapperFactory.createMapperInstance(source, converters, null);

MappingContext context = new MappingContext();
MappingParameters params = new MappingParameters();
params.put("value", "value1");
context.setParams(params);

Destinationdest = mapper.map(source, Destination.class, context);
…
```

When using an OpenL method as a custom converter or Java static class, extend the method signature with the `org.dozer.MappingParameters formal` parameter.

return params.get(“value”);

Figure 37: Custom converter which uses mapping parameters

The mapping processor uses the following rules to find the appropriate OpenL method or Java static method when using it as a convert method:

- If mapping parameters are defined by user, mapping processor searches for a convert method with extended signature.

  If the method is found, the mapper invokes it; otherwise, the mapper searches for a method with usual signature.
- If mapping parameters are not defined by a user, the mapping processor searches for a convert method with usual signature.

The mapping processor uses the same approach for field mapping conditions. When using mapping parameters for field mapping conditions, provide implementation of the `org.dozer.BaseMappingParamsAwareFieldMappingCondition` class as a condition method.

The mapping framework also provides the `org.dozer.factory.BaseMappingParamsAwareBeanFactory` class which uses mapping parameters to create bean instances.

Mapping ID

The mapping flow can be changed by using the mapping ID value.

classA	**classB**	**fieldA**	**fieldB**	**convertMethodAB**	**mapId**
Class A	**Class B**	**Field A**	**Field B**	**Convert method (a into b)**	**Mapping ID**
Source	Destination	srcField	destField	convertMethod	mappingID
Source	Destination	srcField	destField		

Figure 38: Field mapping definitions with mapID parameter

A user can specify which field mapping must be used by the mapping processor. In general case, the mapping processor will use the second definition to map. If the first mapping definition must be used, set the `mappingID` value to the **mapId** property of mapping context as follows:

```
…
MappingContext context = new MappingContext();
context.setMapId("mappingID");
Destination dest = mapper.map(source, Destination.class, context);
…
```

The mapping processor uses the following rules to define the field mappings to be used:

- If the mapping ID is not specified for the context, get field mapping definitions without `mapId` value, that is, a general set of field mappings.
- If the mapping ID is specified for context, proceed as follows:
  1. Get field mapping definitions with the same `mapId` value as the provided mapping ID.
  2. Get field mapping definitions without the `mapId` value which do not define the same field mapping as already selected with the defined `mapId` value.

Field mapping definitions that are not selected will not be processed at runtime.

6 Appendix A: Mapping Bean Fields

Mapping definition element fields are described in the following table:

Mapping bean fields				
Name	**Type**	**Required**	**Default value**	**Description**
classA	Class<?>	✓yes		Class name of the first object to map.
classB	Class<?>	✓yes		Class name of the second object to map.
fieldA	String[]	✓yes		Field name of the first object to map.
fieldB	String	✓yes		Field name of the second object to map.
convertMethodAB	String	no		OpenL rule used as a convert method or Java class static method.
convertMethodBA	String	no		OpenL rule used as a convert method or Java class static method.
convertMethodABId	String	no		ID of the convert method.
convertMethodBAId	String	no		ID of the convert method.
oneWay	Boolean	no	false	One- or bi-directional mapping.
mapNulls	Boolean	no	true	Null values processing method, which is skipping values or mapping as usually.
mapEmptyStrings	Boolean	no	true	Empty strings processing method, which is skipping values or mapping as usually.
trimStrings	Boolean	no	false	Strings processing method, which identifies whether trim operation will be executed.
fieldACreateMethod	String	no		Name of create method to be used to create field A value object instance or Java class static method.
fieldBCreateMethod	String	no		Name of create method to be used to create field B value object instance or Java class static method.
fieldADefaultValue	String	no		Default value of the field.
fieldBDefaultValue	String	no		Default value of the field.
fieldADateFormat	String[]	no		Date format string for the field A value.
fieldBDateFormat	String	no		Date format string for the field B value.
fieldAHint	Class<?>[][]	no		Type hints for the field A used for deep mapping only.
fieldBHint	Class<?>[]	no		Type hints for field B used for deep mapping only.
fieldAType	Class<?>[]	no		Type of the field A value.
fieldBType	Class<?>	no		Type of the field B value.
fieldARequired	Boolean	no	false	Identifier that the field is required and cannot be null.
fieldBRequired	Boolean	no	false	Identifier that the field is required and cannot be null.
conditionAB	String	no		OpenL Boolean condition expression identifying whether the current mapping will be used in the mapping process.
conditionBA	String	no		OpenL Boolean condition expression identifying whether the current mapping will be used in the mapping process.

Mapping bean fields				
Name	**Type**	**Required**	**Default value**	**Description**
conditionABId	String	no		ID of the condition method.
conditionBAId	String	no		ID of the condition method.
fieldADiscriminator	String	no		Discriminator as OpenL rule or Java class static method.
fieldBDiscriminator	String	no		Discriminator as OpenL rule or Java class static method.

7 Appendix B: ClassMappingConfiguration Bean Fields

ClassMappingConfiguration definition element fields are described in the following table:

ClassMappingConfiguration bean fields				
Name	**Type**	**Required**	**Default value**	**Description**
classA	Class<?>	✓yes		Class name of the first object to map.
classB	Class<?>	✓yes		Class name of the second object to map.
classABeanFactory	Class<?>	no		Bean factory class.
classBBeanFactory	Class<?>	no		Bean factory class.
mapNulls	Boolean	no	true	Null values processing method, which is skipping values or mapping as usually.
mapEmptyStrings	Boolean	no	true	Empty strings processing method, which is skipping values or mapping as usually.
trimStrings	Boolean	no	false	Strings processing method, which identifies whether trim operation will be executed.
requiredFields	Boolean	no	false	Identifier that fields are required and cannot be null.
wildcard	Boolean	no	true	Identifier that the mapping processor will map matching fields implicitly.
dateFormat	String	no		Default date format to be used for converting string to date and the opposite way.

8 Appendix C: GlobalConfiguration Bean Fields

GlobalConfiguration definition element fields are described in the following table:

GlobalConfiguration bean fields

Name	Type	Required	Default value	Description
mapNulls	Boolean	no	true	Null values processing method, which is skipping values or mapping as usually.
mapEmptyStrings	Boolean	no	true	Empty strings processing method, which is skipping values or mapping as usually.
trimStrings	Boolean	no	false	Strings processing method, which identifies whether trim operation will be executed.
requiredFields	Boolean	no	false	Identifier that fields are required and cannot be null.
wildcard	Boolean	no	true	Identifier that the mapping processor will map matching fields implicitly.
dateFormat	String	no		Default date format to be used for converting string to date and the opposite way.

9 Appendix D: Converter Bean Fields

Converter definition element fields are described in the following table:

Converter bean fields			
Name	**Type**	**Required**	**Description**
classA	Class<?>	✓yes	Class name of the first object to map.
classB	Class<?>	✓yes	Class name of the second object to map.
convertMethod	String	✓yes	Convert method name.

10 Appendix E: Date and Time Format

Mapping framework uses the `java.text.SimpleDateFormat` class to work with dates.

For more information on this class, see http://docs.oracle.com/javase/1.5.0/docs/api/java/text/SimpleDateFormat.html.

Date and time formats are specified by **date and time pattern** strings. Within date and time pattern strings, unquoted letters from 'A' to 'Z' and from 'a' to 'z' are interpreted as pattern letters representing the components of a date or time string. Text can be quoted using single quotes (') to avoid interpretation. The double quote symbol '' represents a single quote. All other characters are not interpreted and are simply copied into the output string during formatting or matched against the input string during parsing.

The following table describes defined pattern letters, while all other characters from 'A' to 'Z' and from 'a' to 'z' are reserved:

Letter patterns

Letter	Date or Time Component	Presentation	Examples
G	Era designator	Text	AD
y	Year	Year	1996; 96
M	Month in year	Month	July; Jul; 07
w	Week in year	Number	27
W	Week in month	Number	2
D	Day in year	Number	189
d	Day in month	Number	10
F	Day of week in month	Number	2
E	Day in week	Text	Tuesday; Tue
a	Am/pm marker	Text	PM
H	Hour in day (0-23)	Number	0
k	Hour in day (1-24)	Number	24
K	Hour in am/pm (0-11)	Number	0
h	Hour in am/pm (1-12)	Number	12
m	Minute in hour	Number	30
s	Second in minute	Number	55
S	Millisecond	Number	978
z	Timezone	General timezone	Pacific Standard Time; PST; GMT-08:00
Z	Timezone	RFC 822 timezone	-0800

Pattern letters are usually repeated, as their number determines the exact presentation:

- **Text:** For formatting, if the number of pattern letters is 4 or more, the full form is used; otherwise a short or abbreviated form is used if available. For parsing, both forms are accepted, independent of the number of pattern letters.

- **Number:** For formatting, the number of pattern letters is the minimum number of digits, and shorter numbers are zero-padded to this amount. For parsing, the number of pattern letters is ignored unless it is required to separate two adjacent fields.
- **Year:** For formatting, if the number of pattern letters is 2, the year is truncated to 2 digits; otherwise it is interpreted as number.

For parsing, if the number of pattern letters is more than 2, the year is interpreted literally, regardless of the number of digits. So using the pattern "MM/dd/yyyy", "01/11/12" parses to Jan 11, 12 A.D.

The following examples display how date and time patterns are interpreted in the U.S. locale. The given date and time are 2001-07-04 12:08:56 local time in the U.S. Pacific time zone.

Patterns usage examples	
Date and Time Pattern	**Result**
"yyyy.MM.dd G 'at' HH:mm:ss z"	2001.07.04 AD at 12:08:56 PDT
"EEE, MMM d, ''yy"	Wed, Jul 4, '01
"h:mm a"	12:08 PM
"hh 'o''clock' a, zzzz"	12 o'clock PM, Pacific Daylight Time
"K:mm a, z"	0:08 PM, PDT
"yyyyy.MMMMM.dd GGG hh:mmaaa"	02001.July.04 AD 12:08 PM
"EEE, dMMMyyyyHH:mm:ssZ"	Wed, 4 Jul 2001 12:08:56 -0700
"yyMMddHHmmssZ"	010704120856-0700