Mapping Framework Reference Guide

Version 1.1.0

OpenL Tablets BRMS

Table of Contents

Introduction

A Mapping framework (MF) recursively copies data from one object to another. Typically, these data objects will be of different complex types.

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

The picture below demonstrates lifecycle of the framework and shows how underlying frameworks are used by mapper.

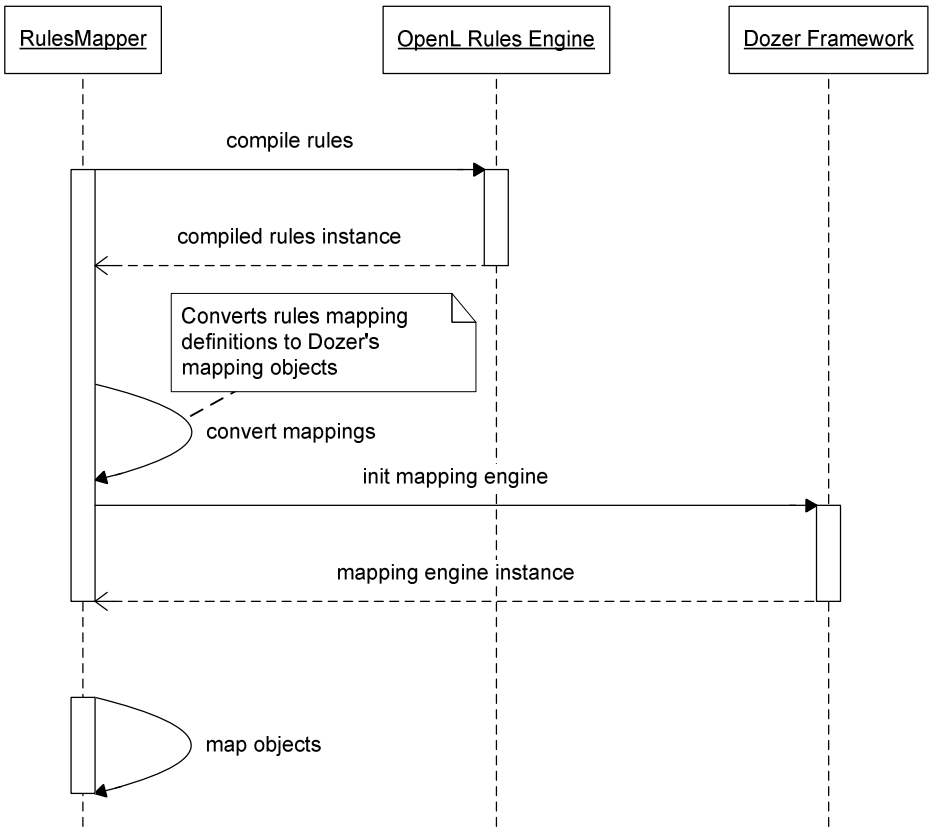

Figure 1. Mapper lifecycle sequence diagram

The mapper is used any time you need to take one type of Java Bean and map it 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.

Links

Dozer framework home page - http://dozer.sourceforge.net/

OpenL Tablets home page - http://openl-tablets.sourceforge.net/

Mapper home page - http://openl-tablets.sourceforge.net/mapper

Getting started

Add the following dependency to your project's pom.

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

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

Table 1. Mappingdefinition

Define one more table:

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

Table 2. Import definition

where *source.package* and *destination.package* are packages which contain *Source* and *Destination* classes. Also you can define full java class names. For the sample class names will be *source.package.Source* and *destination.package.Destination*.

Add the following code snippet into your 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);
```

After mapper completes mapping new instance of *destination.package.Destination* class will be returned with following fields' values: *"string"* for *aStringField* field and *10* for *anIntegerField*.

Overview of Field Mapping Algorithm

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

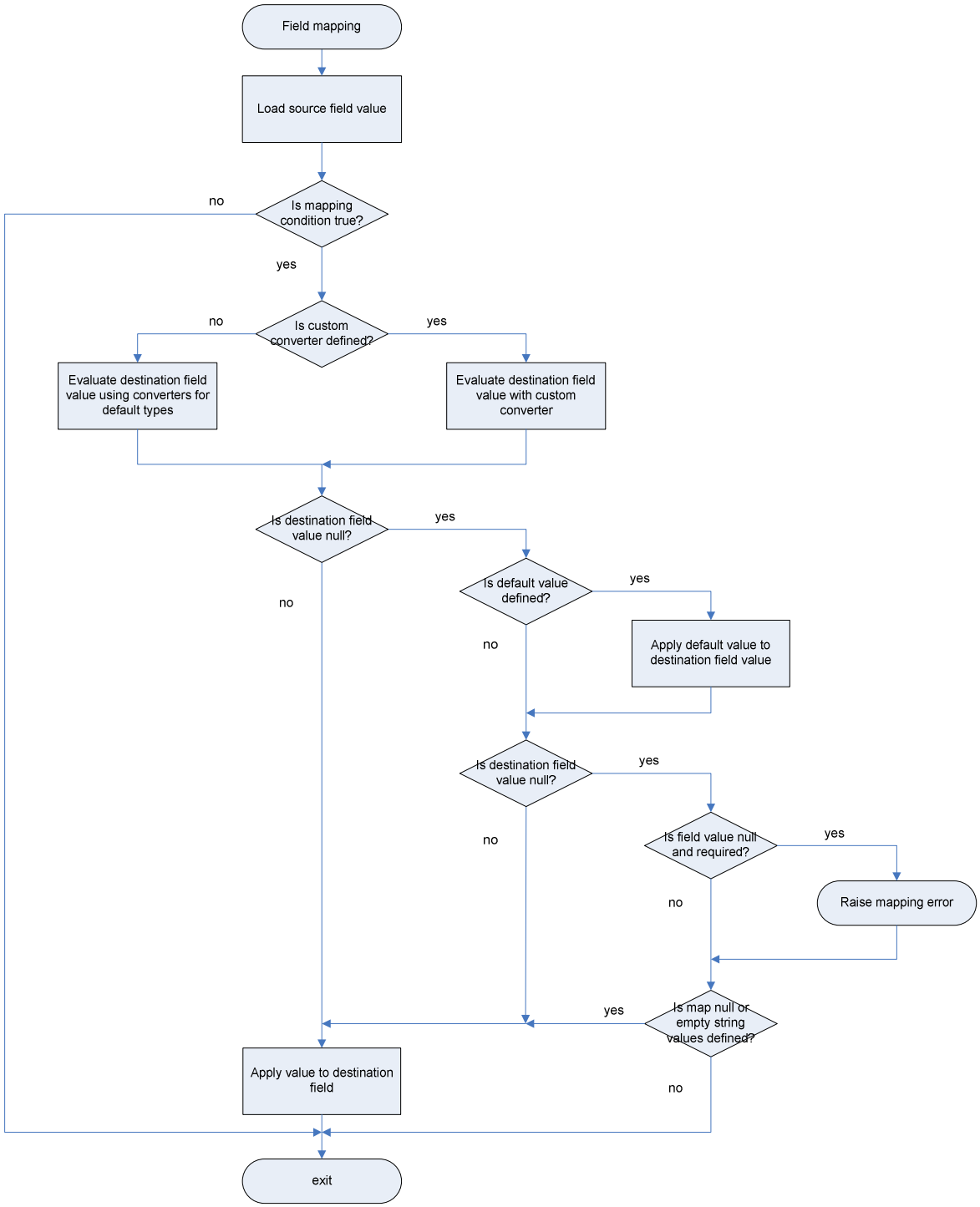

Figure 2. Single field mapping algorithm

Mappings

Rules file format

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

Basic property mapping

Simple property mapping

In case of need to map *srcField* of *source.package.Source* class into *destField* of *destination.package.Destination* class you should provide 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

Table 3. Simple property mapping

If you need to map *destField* of *destination.package.Destination* class into *srcField* of *source.package.Source* class you shouldn't define a new one mapping rule because all mappings are bi-directional by default.

Implicit property mapping

By default mapping framework maps properties with matching names. Look at the following case: source object has field with name *myField* and destination object has matching field *myField*. In this case mapping framework performs mapping for this fields automatically. You can change behavior of this feature using **wildcard** configuration parameter (see Configuration section for more information).

Recursive mapping

Mapping framework supports full class level mapping recursion. If you have any complex types defined at field level mappings in your object, mapping processor will search the mappings for a class level mapping between the two classes that you have mapped. If you do not have any mappings, it will only map fields that are of the same name between the complex types.

Data type conversion

Data type conversion is performed automatically by the mapping engine. Currently, supported the following types of conversions (all of them are bi-directional):

- 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
- Each of these can be mapped to one another: java.util.Date, java.sql.Date, java.sql.Time, java.sql.Timestamp, java.util.Calendar, java.util.GregorianCalendar
- String to any of the supported Date/Calendar Objects.
- Objects containing a toString() method that produces a long representing time in (ms) 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

Table 4. Date format configuration at field level

You can define also date format at class level and global level using **dateFormat** configuration parameter (see Configuration section for more information).

One way mapping

In case of need to map a field pair only in one way you can set it at 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

Table 5. One way mapping definition

When one-way mapping is specified mapping from *destField* into *srcField* will be ignored.

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, mapper will invoke the custom converter to perform the data mapping instead of the standard mapping logic.
Custom converter can be defined in different ways. The following definition shows the first one.

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

Table 6. Convert method definition using OpenL Tablets' method

Parameter **convertMethodAB** tells mapper that should be used *typeCdLookup* convert method for current field pair. The following table is our convert method which is defined using OpenL Tablets rules table component (see OpenL Tablets documentation for more information).

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); ""

Table 7. Custom converter definition using OpenL Tablets component

The following conventions are used for convert methods defition:

- method has to have same name as defined by **convertMethodAB**parameter;
- method has to provide 2 parameters: first has to be assignable from source field type, second has to be assignable from destination field type;
- the destination field should be assignable from method return type.

Note, that convert method cannot be used during reverse mapping. In case of need to map from field B into field A you should provide appropriate convert method using **convertMethodBA** parameter.

Static Java Method as Custom Converter

In the same way you can use static java method as a convert method. In this case you should provide class name and method name. For example, *utils.ConverterUtils.typeCdLookup* or *ConverterUtils.typeCdLookup* (in this case you should define *utils* package in import section of Enviroment 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

Table 8. Convert method definition using java static method

Custom Converter Class

For more advanced functionality you can define your own custom converter implementation class. In this case you have to implement the *org.dozer.CustomConverter* interface in order for mapper to accept it. Otherwise an exception will be thrown.

```
public interface CustomConverter {

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

}
```

Custom Converters get invoked even when the source value is null, so you need to explicitly handle null values in your custom converter implementation.

After you created your own implementation of custom converter you should register it for usage in mapper.For example:

```
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 custom converter can be referenced by ID from 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

Table 9. Convert method definition by converter ID

Class Level Custom Converter

Custom converters can be defined at class level.

Data Converter defaultConverters		
classA	**classB**	**convertMethod**
Class A	**Class B**	**Convert method**
Source	Destination	ConverterUtils.typeCdLookup

Table 10. Default custom converters definition

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

Custom Converter Search Algorithm

Mapper uses the following order to find appropriate converter:

- get custom converter by its ID;
- get custom converter by name (either rule or java method);
- get default custom converter.

Field mapping conditions

Field mapping conditions are used to make a decision at runtime when field mapping should be executed and when shouldn't. When a field mapping condition is specified for a field A and field B combination, mapper will invoke it at start of field mapping flow (see Overview of Field Mapping Algorithm section for more information).

Mapping conditions can be defined in different ways. The following definition shows one of them.

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

Table 11. Field mapping condition method definition using OpenL Tablets' method

Parameter **conditionAB** tells to mapper that for current field pair mapping should be invoked *mapField* method to make a decision. If method result is *true* mapper will execute mapping rule; otherwise – skip

it. The following table is example of condition method definition using OpenL Tablets method table component (see OpenL Tablets documentation for more information).

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

Table 12. Field mapping conditionexample

The following conventions are used for convert methods definition:

- method has to have same name as defined by **conditionAB** parameter;
- method has to provide 2 parameters: first has to be assignable from source field type, second has to be assignable form destination field type;
- method return type must be of boolean type.

Note, that condition method will not be used during reverse mapping. In case of need to use condition during mapping from field B into field A you should provide appropriate method using **conditionBA** parameter.

Static Java Method as Condition

In the same way you can use static java method as a condition method. In this case you should provide class name and method name. For example, *utils.ConditionUtils.mapField* or *ConditionUtils.mapField* (in this case you should define *utils* package in import section of 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

Table 13. Condition method definition using java static method

Condition Class

You can define your own field mapping condition implementation class. In this case you have to implement the *org.dozer.FieldMappingCondition* interface in order for 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 you need to explicitly handle null values in your implementation.

After you create your own implementation you should register it for usage in mapper. For example:

```
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 demonstrates how condition can be referenced by ID from 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

Table 14. Condition method definition by ID

Conditions Search Algorithm

Mapper uses the following order to find appropriate field condition:

- get condition by its ID;
- get condition by name (either rule or java method).

Default values

At the field mapping level you can define default value for destination field. It will be set into destination property if source value is *null* (see High level field mapping algorithm section for more information).

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

Table 15. Field default value definition

Default value is a string value which will be converted into field's type. Currently, default value string can be converted into the following types: primitive types (int, double, short, char, long, boolean, byte, float), wrapper types (java.lang.Integer, java.lang.Double, java.lang.Short, java.lang.Character, java.lang.Long, java.lang.Boolean, java.lang.Byte, java.lang.Float), java.math.BigDecimal, java.math.BigInteger, java.lang.String, java.util.Date, java.util.Calendar, complex types (constructor with java.lang.String parameter should beprovided).

Empty source mapping

Empty source mapping is a special case of field initialization. The common case for which can be used current type of mapping is 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

Table 16. Empty source mapping definition

Empty source mapping is **always** one-way mapping. It should be ignored during reverse mapping.

Multi source mapping

Multi source mapping is useful when data calculation is required using several sources of input data. Custom converter should 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			

Table 17. Multi source mapping definition

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

Table 18. Custom converter for multi source mapping

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

Index mapping

Fields that need to be looked up or written to by indexed property are supported.

Simple index

Mapping using simple index is a usage of index operator to locate required element in 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

Table 19. A simple index mapping definition

Simple index value is 1-based integer value. It means that index starts from 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 usual.

Mapping to the end of collection

Mapping to the end of collection is a special case of simple index and it's defined as "0" index. It can be used when you need to map source data into destination element and append destination element to collection. Note that "0" is not index of last element. Mapping processor will create new element, map data into element and append 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]

Table 20. Mapping to the end of collection

Index of real element is calculated by mapping processor every time when "0" index appears. This type of index cannot be used for deep mapping definition because index value is changed at runtime. If it's used in deep mapping an exception will be thrown. Also you should not use it for source field path definition because in this case mapper always returns *null* value.

Expression index

Mapper provides ability to define expression to look up array or collection element. Defined expression must be 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

Table 21. An expression index mapping definition

Expression index cannot be used for destination field definition. Field mapping which uses expression index should be marked as one-way mapping.

Collection item discriminator

A collection item discriminator gives you ability to define a destination collection element at runtime when you map source object into collection or array. For example, your target collection contains list of policies but you don't know which collection element represents required policy. In this case you can 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

Table 22. Collection item discriminator usage example

When mapping processor starts to map collection into collection it will use the following steps:

- get source object or next element from source collection;

- get target element from destination collection (if collection item discriminator is provided it will be used to get target element);
- if target element is not found (for example discriminator is not defined or discriminator returned *null* value) create a new one and add it to the end of destination collection;
- map source element into target element.

A discriminator can be defined as an OpenL method or as a static java method. For example:

Method Object discriminator(Object src, List dest)
// target element resolving goes here … **return** null;

Table 23. Collection item discriminator method example

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

Deep mapping

Mapper provides ability to map deep properties. An example would be when you have an object with a String property. Your other object has a String property but it 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

Table 24. Deep mapping usage example

Field type

Field type parameter can be used to define type which should be used for current field.

For example, *srcField* defined as a *BaseType* type and *destField* defined as a *DestType* type. The *DestType* type is super type of *CustomType* type. At runtime *destField* is *CustomType* value and you need to tell mapper to map *srcField* into *destField* as *BaseType* value into *CustomType* value instead of *BaseType* value into *DestType* value because, for example, *CustomType* value has extra data that should be mapped. In this case you can use **fieldBType** parameter to define required type of 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

Table 25. Field type casting example

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

Type hints

Type hints concept the same as a field type concept but it's provided for deep mapping use case.

Field type hints value is array of class names (see OpenL Tablets documentation for more information about array definitions). Each of them will be used as a type of appropriate field in 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	DestType, NestedType

Table 26. Type hints usage example

Field type hints can be simplified if you would like to skip class type definition for appropriate field in 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

Table 27. Simplified type hints definition example

Note that leading comma is not omitted because we should keep correspondence between fields and hints. Trailing commas can be omitted, for example, *NestedType, , ,* is equal *NestedType* but *NestedType,,,OtherType* is not equal *NestedType,OtherType*.

Configuration

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

Name	Value	Description
MapNulls	true	Defines how null values will be processed: skipped or mapped as usual.
MapEmptyStrings	true	Defines how empty strings will be processed: skipped or mapped as usual.
TrimStrings	false	Defines how strings will be processed: trim operation will be executed or not.
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.

Table 28. Mapping default policies

In accordance with business needs policies could 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

Table 29. Mapping process configuration definition at global level

Global configuration must be only one. If there is several definitions, an exception will be 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

Table 30. Mapping process configuration definition at class level

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

Configuration Priority

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

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

Custom bean factories

You can configure mapper to use custom bean factories to create new instances of destination data objects during the mapping process. By default mapper just creates a new instance of any destination objects using a default constructor. You can specify your own bean factories to instantiate the data objects.

Your custom bean factory must implement the *org.dozer.BeanFactory* interface.

```
public interface BeanFactory {

      Object createBean(Object source,
                        Class<?> sourceClass,
                        String targetBeanId);

}
```

Finally you should update configuration for 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

Table 31.Bean factory using example

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

Create method

You can configure mapper 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

Table 32. Field type casting example

Create method is a java static method with no parameters.

```
public class CreateMethodClass {

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

Mapping inheritance

In case of mapping subclasses that also have base class attributes you don't need to define for this attributes individual mappings if they are already defined for super class mapping. For example, we have 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;

	…
}
```

Next, we defined 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

Table 33. Mapping attributes of parent class

In case of mapping *Source* object into *ChildDest* mapping processor will use mappings of *Source-ParentDest* class pair to map attributes of super class (*Source.firstField* into *ParentDest.firstField* in our example).

Overriding

Super mapping definition can be overridden. You should define a new one field mapping which uses 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

Table 34. Overriding mapping attributes of parent class

For our example mapping processor will use the first field mapping definition for classes *Source* and *ParentDest* and the second one for classes *Source* and *ChildDest*.

Context based mapping

Mapping parameters

Mapping flow can be changed by 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

Table 35. Convert method definition by converter ID

In case of using mapping parameters you should to provide implementation of *org.dozer.BaseMappingParamsAwareCustomConverter* class. The *BaseMappingParamsAwareCustomConverter* defines extended **convert** method which should be implemented by user. The following example shows how can be used mapping parameters:

```
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);

…
```

In case of using OpenL mehod as a custom converter or java static class you should extend method signature with *org.dozer.MappingParameters* formal parameter, for example:

Method Object convertMethod(MappingParameters params, Object src, Object dest)
return params.get("value");

Table 36. Custom converter which uses mapping parameters

Mapping processor uses the following rules to find appropriate OpenL method or java static method if you are using it as a convert method:

- If mapping parameters are defined by user mapping processor tries to find convert method with extended signature. If method is found mapper will invoke it else mapper tries to find method with usual signature;
- If mapping parameters are not defined by user mapping processor tries to find convert method with usual signature.

Mapping processor uses the same approach for field mapping conditions. In case of using mapping parameters for field mapping conditions you should provide implementation of *org.dozer.BaseMappingParamsAwareFieldMappingCondition* class as a condition method.

Also mapping framework provides *org.dozer.factory.BaseMappingParamsAwareBeanFactory* class which uses mapping parameters to create bean instances.

Mapping ID

Also mapping flow can be changed by user using mapping ID value.

Data Mapping mappings					
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		

Table 37. Field mapping definitions with mapId parameter

Now user can direct which of field mapping will be used by mapping processor. In general case mapping processor will use the second definition to map. If required to use the first mapping definition user should set *"mappingID"* value to **mapId** property of mapping context:

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

Mapping processor uses the following rules to define which of field mappings will be used:

- if mapping id is not specified for context get field mapping definitions which don't have **mapId** value (general set of field mappings);
- if mapping id is specified for context:
    a) get field mapping definitions which have the same **mapId** value as provided mapping id;
    b) get field mapping definitions which don't have **mapId** value and which don't define the same field mapping as already selected with defined **mapId** value.

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

Generally speaking, mapping id defines which field mapping definitions will be used by mapping processor and mapping parameters define how will be processed appropriate field mappings.

Appendix A. Mapping bean fields

Mapping definition element fields are described in the following table:

Name	Type	Required	Defaultvalue	Description
classA	Class<?>	✓yes		Class name of first object to mapping.
classB	Class<?>	✓yes		Class name of second object to mapping.
fieldA	String[]	✓yes		Field name of first object to mapping.
fieldB	String	✓yes		Field name of second object to mapping.
convertMethodAB	String	no		OpenL rule which used as a convert method or java class static method.
convertMethodBA	String	no		OpenL rule which used as a convert method or java class static method.
convertMethodABId	String	no		Id of convert method
convertMethodBAId	String	no		Id of convert method
oneWay	Boolean	no	false	Defines one- or bi-directional mapping.
mapNulls	Boolean	no	true	Defines how null values will be processed: skipped or mapped as usual.
mapEmptyStrings	Boolean	no	true	Defines how empty strings will be processed: skipped or mapped as usual.
trimStrings	Boolean	no	false	Defines how strings will be processed: trim operation will be executed or not.
fieldACreateMethod	String	no		Name of create method which will be used to create field A value object instance or java class static method.
fieldBCreateMethod	String	no		Name of create method which will be used to create field B value object instance or java class static method.
fieldADefaultValue	String	no		Default value of field.
fieldBDefaultValue	String	no		Default value of field.
fieldADateFormat	String[]	no		Date format string for field A value.
fieldBDateFormat	String	no		Date format string for field B value.
fieldAHint	Class<?>[][]	no		Type hints for 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 field A value.
fieldBType	Class<?>	no		Type of field B value.
fieldARequired	Boolean	no	false	Defines that field is required and it cannot be null.
fieldBRequired	Boolean	no	false	Defines that field is required and it cannot be null.
conditionAB	String	no		OpenL boolean condition

				expression which defines that current mapping will be used in mapping process or not.
conditionBA	String	no		OpenL boolean condition expression which defines that current mapping will be used in mapping process or not.
conditionABId	String	no		Id of condition method.
conditionBAId	String	no		Id of 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

Table 38. Mappingbeanfields

Appendix B. ClassMappingConfiguration bean fields

ClassMappingConfiguration definition element fields are described in the following table:

Name	Type	Required	Defaultvalue	Description
classA	Class<?>	✓yes		Class name of first object to mapping.
classB	Class<?>	✓yes		Class name of second object to mapping.
classABeanFactory	Class<?>	no		Bean factory class.
classBBeanFactory	Class<?>	no		Bean factory class.
mapNulls	Boolean	no	true	Defines how null values will be processed: skipped or mapped as usual.
mapEmptyStrings	Boolean	no	true	Defines how empty strings will be processed: skipped or mapped as usual.
trimStrings	Boolean	no	false	Defines how strings will be processed: trim operation will be executed or not.
requiredFields	Boolean	no	false	Defines that fields are required and they cannot be null.
wildcard	Boolean	no	true	Defines that mapping processor will map matching fields implicitly.
dateFormat	String	no		Default date format which will be used for string to date conversion and vise versa.

Table 39.ClassMappingConfiguration bean fields

Appendix C. GlobalConfiguration bean fields

GlobalConfiguration definition element fields are described in the following table:

Name	Type	Required	Defaultvalue	Description
mapNulls	Boolean	no	true	Defines how null values will be processed: skipped or mapped as usual.
mapEmptyStrings	Boolean	no	true	Defines how empty strings will be processed: skipped or mapped as usual.
trimStrings	Boolean	no	false	Defines how strings will be processed: trim operation will be executed or not.
requiredFields	Boolean	no	false	Defines that fields are required and they cannot be null.
wildcard	Boolean	no	true	Defines that mapping processor will map matching fields implicitly.
dateFormat	String	no		Default date format which will be used for string to date conversion and vise versa.

Table 40.GlobalConfiguration bean fields

Appendix D. Converter bean fields

Converter definition element fields are described in the following table:

Name	Type	Required	Description
classA	Class<?>	✓yes	Class name of first object to mapping.
classB	Class<?>	✓yes	Class name of second object to mapping.
convertMethod	String	✓yes	Convert method name

Table 41.Converter bean fields

Appendix E. Date and Time format

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

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. ''represents a single quote. All other characters are not interpreted; they're simply copied into the output string during formatting or matched against the input string during parsing.

The following pattern letters are defined (all other characters from 'A' to 'Z' and from 'a' to 'z' are reserved):

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

Table 42.Letter patterns

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's needed 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 show 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 time zone.

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

Table 43.Patterns usage examples