MapStruct 1.1.0.Beta1 Reference Guide

Gunnar Morling, Andreas Gudian, Sjaak Derksen

2016-03-16

Table of Contents

Preface 1
1. Introduction 1
2. Set up 1
2.1. Configuration options 3
3. Defining a mapper 5
3.1. Basic mappings 5
3.2. Generating mappers from abstract classes 8
3.3. Mapping methods with several source parameters 8
3.4. Nested mappings 9
3.5. Updating existing bean instances 10
4. Retrieving a mapper 10
4.1. The Mappers factory 10
4.2. Using dependency injection 11
5. Data type conversions 12
5.1. Implicit type conversions 12
5.2. Mapping object references 14
5.3. Invoking other mappers 15
5.4. Passing the mapping target type to custom mappers 16
5.5. Mapping method resolution 17
5.6. Mapping method selection based on qualifiers 17
6. Mapping collections 21
6.1. Mapping maps 23
6.2. Collection mapping strategies 24
6.3. Implementation types used for collection mappings 25
7. Mapping Values 26
7.1. Mapping enum types 26
8. Object factories 29
9. Advanced mapping options 31
9.1. Default values and constants 31
9.2. Expressions 32
9.3. Determining the result type 33
9.4. Controlling mapping result for 'null' arguments 34
9.5. Exceptions 35
10. Reusing mapping configurations 36
10.1. Mapping configuration inheritance 36
10.2. Inverse mappings 37
10.3. Shared configurations 38
11. Customizing mappings 40

11.1. Mapping customization with decorators 41
11.2. Mapping customization with before-mapping and after-mapping methods 44

Preface

This is the reference documentation of MapStruct, an annotation processor for generating type-safe, performant and dependency-free bean mapping code. This guide covers all the functionality provided by MapStruct. In case this guide doesn't answer all your questions just join the MapStruct Google group to get help.

You found a typo or other error in this guide? Please let us know by opening an issue in the MapStruct GitHub repostory, or, better yet, help the community and send a pull request for fixing it!

1. Introduction

MapStruct is a Java annotation processor for the generation of type-safe bean mapping classes.

All you have to do is to define a mapper interface which declares any required mapping methods. During compilation, MapStruct will generate an implementation of this interface. This implementation uses plain Java method invocations for mapping between source and target objects, i.e. no reflection or similar.

Compared to writing mapping code from hand, MapStruct saves time by generating code which is tedious and error-prone to write. Following a convention over configuration approach, MapStruct uses sensible defaults but steps out of your way when it comes to configuring or implementing special behavior.

Compared to dynamic mapping frameworks, MapStruct offers the following advantages:

- Fast execution by using plain method invocations instead of reflection
- Compile-time type safety: Only objects and attributes mapping to each other can be mapped, no accidental mapping of an order entity into a customer DTO etc.
- Clear error-reports at build time, if entities or attributes can't be mapped

2. Set up

MapStruct is a Java annotation processor based on JSR 269 and as such can be used within command line builds (javac, Ant, Maven etc.) as well as from within your IDE.

It comprises the following artifacts:

- *org.mapstruct:mapstruct*: contains the required annotations such as `@Mapping`; On Java 8 or later, use *org.mapstruct:mapstruct-jdk8* instead which takes advantage of language improvements introduced in Java 8
- *org.mapstruct:mapstruct-processor*: contains the annotation processor which generates mapper implementations

For Maven based projects add the following to your POM file in order to use MapStruct:

Example 1. Maven configuration

```
...
<properties>
    <org.mapstruct.version>1.1.0.Beta1</org.mapstruct.version>
</properties>
...
<dependencies>
    <dependency>
        <groupId>org.mapstruct</groupId>
        <artifactId>mapstruct-jdk8</artifactId>
        <version>${org.mapstruct.version}</version>
    </dependency>
</dependencies>
...
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>3.5.1</version>
            <configuration>
                <source>1.8</source>
                <target>1.8</target>
                <annotationProcessorPaths>
                    <path>
                        <groupId>org.mapstruct</groupId>
                        <artifactId>mapstruct-processor</artifactId>
                        <version>${org.mapstruct.version}</version>
                    </path>
                </annotationProcessorPaths>
            </configuration>
        </plugin>
    </plugins>
</build>
...
```

If you are working with the Eclipse IDE, make sure to have a current version of the M2E plug-in. When importing a Maven project configured as shown above, it will set up the MapStruct annotation processor so it runs right in the IDE, whenever you save a mapper type. Neat, isn't it?

To double check that everything is working as expected, go to your project's properties and select "Java Compiler" → "Annotation Processing" → "Factory Path". The MapStruct processor JAR should be listed and enabled there. Any processor options configured via the compiler plug-in (see below) should be listed under "Java Compiler" → "Annotation Processing".

If the processor is not kicking in, check that the configuration of annotion processors through M2E is enabled. To do so, go to "Preferences" → "Maven" → "Annotation Processing" and select "Automatically configure JDT APT". Alternatively, specify the following in the `properties` section of your POM file: `<m2e.apt.activation>jdt_apt</m2e.apt.activation>`.

Also make sure that your project is using Java 1.6 or later (project properties → "Java Compiler" → "Compile Compliance Level"). It will not work with older versions.

2.1. Configuration options

The MapStruct code generator can be configured using *annotation processor options*.

When invoking javac directly, these options are passed to the compiler in the form *-Akey=value*. When using MapStruct via Maven, any processor options can be passed using an `options` element within the configuration of the Maven processor plug-in like this:

Example 2. Maven configuration

```
...
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <version>3.5.1</version>
    <configuration>
        <source>1.8</source>
        <target>1.8</target>
        <annotationProcessorPaths>
            <path>
                <groupId>org.mapstruct</groupId>
                <artifactId>mapstruct-processor</artifactId>
                <version>${org.mapstruct.version}</version>
            </path>
        </annotationProcessorPaths>
        <compilerArgs>
            <compilerArg>
                -Amapstruct.suppressGeneratorTimestamp=true
            </compilerArg>
            <compilerArg>
                -Amapstruct.suppressGeneratorVersionInfoComment=true
            </compilerArg>
        </compilerArgs>
    </configuration>
</plugin>
...
```

The following options exist:

Table 1. MapStruct processor options

Option	Purpose	Default
`mapstruct.suppressGeneratorTimestamp`	If set to `true`, the creation of a time stamp in the `@Generated` annotation in the generated mapper classes is suppressed.	`false`
`mapstruct.suppressGeneratorVersionInfoComment`	If set to `true`, the creation of the `comment` attribute in the `@Generated` annotation in the generated mapper classes is suppressed. The comment contains information about the version of MapStruct and about the compiler used for the annotation processing.	`false`

Option	Purpose	Default
`mapstruct.defaultComponentModel`	The name of the component model (see Retrieving a mapper) based on which mappers should be generated. Supported values are: * `default`: the mapper uses no component model, instances are typically retrieved via `Mappers#getMapper(Class)` * `cdi`: the generated mapper is an application-scoped CDI bean and can be retrieved via `@Inject` * `spring`: the generated mapper is a singleton-scoped Spring bean and can be retrieved via `@Autowired` * `jsr330`: the generated mapper is annotated with {@code @Named} and can be retrieved via `@Inject`, e.g. using Spring If a component model is given for a specific mapper via `@Mapper#componentModel()`, the value from the annotation takes precedence.	`default`
`mapstruct.unmappedTargetPolicy`	The default reporting policy to be applied in case an attribute of the target object of a mapping method is not populated with a source value. Supported values are: * `ERROR`: any unmapped target property will cause the mapping code generation to fail * `WARN`: any unmapped target property will cause a warning at build time * `IGNORE`: unmapped target properties are ignored If a policy is given for a specific mapper via `@Mapper#unmappedTargetPolicy()`, the value from the annotation takes precedence.	`WARN`

3. Defining a mapper

In this section you'll learn how to define a bean mapper with MapStruct and which options you have to do so.

3.1. Basic mappings

To create a mapper simply define a Java interface with the required mapping method(s) and annotate it with the `org.mapstruct.Mapper` annotation:

Example 3. Maven configuration

```
@Mapper
public interface CarMapper {

    @Mappings({
        @Mapping(source = "make", target = "manufacturer"),
        @Mapping(source = "numberOfSeats", target = "seatCount")
    })
    CarDto carToCarDto(Car car);

    @Mapping(source = "name", target = "fullName")
    PersonDto personToPersonDto(Person person);
}
```

The `@Mapper` annotation causes the MapStruct code generator to create an implementation of the `CarMapper` interface during build-time.

In the generated method implementations all readable properties from the source type (e.g. `Car`) will be copied ino the corresponding property in the target type (e.g. `CarDto`). If a property has a different name in the target entity, its name can be specified via the `@Mapping` annotation.

The property name as defined in the JavaBeans spefication must be specified in the `@Mapping` annotation, e.g. *seatCount* for a property with the accessor methods `getSeatCount()` and `setSeatCount()`.

When using Java 8 or later, you can omit the `@Mappings` wrapper annotation and directly specify several `@Mapping` annotations on one method.

To get a better understanding of what MapStruct does have a look at the following implementation of the `carToCarDto()` method as generated by MapStruct:

Example 4. Code generated by MapStruct

```
// GENERATED CODE
public class CarMapperImpl implements CarMapper {

    @Override
    public CarDto carToCarDto(Car car) {
        if ( car == null ) {
            return null;
        }

        CarDto carDto = new CarDto();

        if ( car.getFeatures() != null ) {
            carDto.setFeatures( new ArrayList<String>( car.getFeatures() ) );
        }
        carDto.setManufacturer( car.getMake() );
        carDto.setSeatCount( car.getNumberOfSeats() );
        carDto.setDriver( personToPersonDto( car.getDriver() ) );
        carDto.setPrice( String.valueOf( car.getPrice() ) );
        if ( car.getCategory() != null ) {
            carDto.setCategory( car.getCategory().toString() );
        }

        return carDto;
    }

    @Override
    public PersonDto personToPersonDto(Person person) {
        //...
    }
}
```

The general philosophy of MapStruct is to generate code which looks as much as possible as if you had written it yourself from hand. In particular this means that the values are copied from source to target by plain getter/setter invocations instead of reflection or similar.

As the example shows the generated code takes into account any name mappings specified via `@Mapping`. If the type of a mapped attribute is different in source and target entity, MapStruct will either apply an automatic conversion (as e.g. for the *price* property, see also Implicit type conversions) or optionally invoke another mapping method (as e.g. for the *driver* property, see also Mapping object references).

Collection-typed attributes with the same element type will be copied by creating a new instance of the target collection type containing the elements from the source property. For collection-typed attributes with different element types each element will mapped individually and added to the target collection (see Mapping collections).

MapStruct takes all public properties of the source and target types into account. This includes properties declared on super-types.

3.2. Generating mappers from abstract classes

In some cases it can be required to manually implement a specific mapping from one type to another which can't be generated by MapStruct. One way for this is to implement such method on another class which then is used by mappers generated by MapStruct (see Invoking other mappers).

Alternatively you can define a mapper in form of an abstract class instead of an interface and implement custom methods directly in this mapper class. In this case MapStruct will generate an extension of the abstract class with implementations of all abstract methods.

As an example let's assume the mapping from `Person` to `PersonDto` requires some special logic which can't be generated by MapStruct. You could then define the mapper from the previous example like this:

Example 5. Mapper defined by an abstract class

```
@Mapper
public abstract class CarMapper {

    @Mappings(...)
    public abstract CarDto carToCarDto(Car car);

    public PersonDto personToPersonDto(Person person) {
        //hand-written mapping logic
    }
}
```

MapStruct will generate a sub-class of `CarMapper` with an implementation of the `carToCarDto()` method as it is declared abstract. The generated code in `carToCarDto()` will invoke the manually implemented `personToPersonDto()` method when mapping the `driver` attribute.

3.3. Mapping methods with several source parameters

MapStruct also supports mapping methods with several source parameters. This is useful e.g. in order to combine several entities into one data transfer object. The following shows an example:

Example 6. Mapping method with several source parameters

```
@Mapper
public interface AddressMapper {

    @Mappings({
        @Mapping(source = "person.description", target = "description"),
        @Mapping(source = "address.houseNo", target = "houseNumber")
    })
    DeliveryAddressDto personAndAddressToDeliveryAddressDto(Person person, Address
address);
}
```

The shown mapping method takes two source parameters and returns a combined target object. As with single-parameter mapping methods properties are mapped by name.

In case several source objects define a property with the same name, the source parameter from which to retrieve the property must be specified using the `@Mapping` annotation as shown for the `description` property in the example. An error will be raised when such an ambiguity is not resolved. For properties which only exist once in the given source objects it is optional to specify the source parameter's name as it can be determined automatically.

Specifying the parameter in which the property resides is mandatory when using the `@Mapping` annotation.

Mapping methods with several source parameters will return `null` in case all the source parameters are `null`. Otherwise the target object will be instantiated and all properties from the provided parameters will be propagated.

3.4. Nested mappings

MapStruct will handle nested mappings, by means of the `.` notation:

Example 7. Mapping method with several source parameters

```
@Mappings({
    @Mapping(target = "chartName", source = "chart.name"),
    @Mapping(target = "title", source = "song.title"),
    @Mapping(target = "artistName", source = "song.artist.name"),
    @Mapping(target = "recordedAt", source = "song.artist.label.studio.name"),
    @Mapping(target = "city", source = "song.artist.label.studio.city"),
    @Mapping(target = "position", source = "position")
})
ChartEntry map(Chart chart, Song song, Integer position);
```

Note: the parameter name (`chart`, `song`, `position`) is required, since there are several source parameters in the mapping. If there's only one source parameter, the parameter name can be ommited.

MapStruct will perform a null check on each nested property in the source.

Also non java bean source parameters (like the `java.lang.Integer`) can be mapped in this fashion.

3.5. Updating existing bean instances

In some cases you need mappings which don't create a new instance of the target type but instead update an existing instance of that type. This sort of mapping can be realized by adding a parameter for the target object and marking this parameter with `@MappingTarget`. The following shows an example:

Example 8. Update method

```
@Mapper
public interface CarMapper {

    void updateCarFromDto(CarDto carDto, @MappingTarget Car car);
}
```

The generated code of the `updateCarFromDto()` method will upate the passed `Car` instance with the properties from the given `CarDto` object. There may be only one parameter marked as mapping target. Instead of `void` you may also set the method's return type to the type of the target parameter, which will cause the generated implementation to update the passed mapping target and return it as well. This allows for fluent invocations of mapping methods.

Collection- or map-typed properties of the target bean to be updated will be cleared and then populated with the values from the corresponding source collection or map.

4. Retrieving a mapper

4.1. The Mappers factory

Mapper instances can be retrieved via the `org.mapstruct.factory.Mappers` class. Just invoke the `getMapper()` method, passing the interface type of the mapper to return:

Example 9. Using the Mappers factory

```
CarMapper mapper = Mappers.getMapper( CarMapper.class );
```

By convention, a mapper interface should define a member called `INSTANCE` which holds a single instance of the mapper type:

Example 10. Declaring an instance of a mapper

```
@Mapper
public interface CarMapper {

    CarMapper INSTANCE = Mappers.getMapper( CarMapper.class );

    CarDto carToCarDto(Car car);
}
```

This pattern makes it very easy for clients to use mapper objects without repeatedly instantiating new instances:

Example 11. Accessing a mapper

```
Car car = ...;
CarDto dto = CarMapper.INSTANCE.carToCarDto( car );
```

Note that mappers generated by MapStruct are thread-safe and thus can safely be accessed from several threads at the same time.

4.2. Using dependency injection

If you're working with a dependency injection framework such as CDI (Contexts and Dependency Injection for Java™ EE) or the Spring Framework, it is recommended to obtain mapper objects via dependency injection as well. For that purpose you can specify the component model which generated mapper classes should be based on either via `@Mapper#componentModel` or using a processor option as described in Configuration options.

Currently there is support for CDI and Spring (the later either via its custom annotations or using the JSR 330 annotations). See Configuration options for the allowed values of the `componentModel` attribute which are the same as for the `mapstruct.defaultComponentModel` processor option. In both cases the required annotations will be added to the generated mapper implementations classes in order to make the same subject to dependency injection. The following shows an example using CDI:

Example 12. A mapper using the CDI component model

```
@Mapper(componentModel = "cdi")
public interface CarMapper {

    CarDto carToCarDto(Car car);
}
```

The generated mapper implementation will be marked with the `@ApplicationScoped` annotation and thus can be injected into fields, constructor arguments etc. using the `@Inject` annotation:

Example 13. Obtaining a mapper via dependency injection

```
@Inject
private CarMapper mapper;
```

A mapper which uses other mapper classes (see Invoking other mappers) will obtain these mappers using the configured component model. So if `CarMapper` from the previous example was using another mapper, this other mapper would have to be an injectable CDI bean as well.

5. Data type conversions

Not always a mapped attribute has the same type in the source and target objects. For instance an attribute may be of type `int` in the source bean but of type `Long` in the target bean.

Another example are references to other objects which should be mapped to the corresponding types in the target model. E.g. the class `Car` might have a property `driver` of the type `Person` which needs to be converted into a `PersonDto` object when mapping a `Car` object.

In this section you'll learn how MapStruct deals with such data type conversions.

5.1. Implicit type conversions

MapStruct takes care of type conversions automatically in many cases. If for instance an attribute is of type `int` in the source bean but of type `String` in the target bean, the generated code will transparently perform a conversion by calling `String#valueOf(int)` and `Integer#parseInt(String)`, respectively.

Currently the following conversions are applied automatically:

- Between all Java primitive data types and their corresponding wrapper types, e.g. between `int` and `Integer`, `boolean` and `Boolean` etc. The generated code is `null` aware, i.e. when converting a wrapper type into the corresponding primitive type a `null` check will performed.
- Between all Java primitive number types and the wrapper types, e.g. between `int` and `long` or

`byte` and `Integer`.

Converting from larger data types to smaller ones (e.g. from `long` to `int`) can cause a value or precision loss. There will be an option for raising a warning in such cases in a future MapStruct version.

- Between all Java primitive types (including their wrappers) and `String`, e.g. between `int` and `String` or `Boolean` and `String`.
- Between `enum` types and `String`.
- Between big number types (`java.math.BigInteger`, `java.math.BigDecimal`) and Java primitive types (including their wrappers) as well as String
- Between `JAXBElement<T>` and `T`, `List<JAXBElement<T>>` and `List<T>`
- Between `java.util.Calendar`/`java.util.Date` and JAXB's `XMLGregorianCalendar`
- Between `java.util.Date`/`XMLGregorianCalendar` and `String`. A format string as understood by `java.text.SimpleDateFormat` can be specified via the `dateFormat` option as this:

Example 14. Conversion from Date to String

```
@Mapper
public interface CarMapper {

    @Mapping(source = "manufacturingDate", dateFormat = "dd.MM.yyyy")
    CarDto carToCarDto(Car car);

    @IterableMapping(dateFormat = "dd.MM.yyyy")
    List<String> stringListToDateList(List<Date> dates);
}
```

- Between Jodas `org.joda.time.DateTime`, `org.joda.time.LocalDateTime`, `org.joda.time.LocalDate`, `org.joda.time.LocalTime` and `String`. A format string as understood by `java.text.SimpleDateFormat` can be specified via the `dateFormat` option (see above).
- Between Jodas `org.joda.time.DateTime` and `java.util.Calendar`.
- Between Jodas `org.joda.time.LocalDateTime`, `org.joda.time.LocalDate` and `java.util.Date`.
- Between `java.time.ZonedDateTime`, `java.time.LocalDateTime`, `java.time.LocalDate`, `java.time.LocalTime` from Java 8 Date-Time package and `String`. A format string as understood by `java.text.SimpleDateFormat` can be specified via the `dateFormat` option (see above).
- Between `java.time.ZonedDateTime` from Java 8 Date-Time package and `java.util.Date` where, when mapping a `ZonedDateTime` from a given `Date`, systems default timezone is used.
- Between `java.time.LocalDateTime` from Java 8 Date-Time package and `java.util.Date` where. When converting a `LocalDateTime` from a given `Date`, systems default timezone is used. When mapping a `Date` to a `LocalDateTime` UTC is used as the timzone.
- Between `java.time.ZonedDateTime` from Java 8 Date-Time package and `java.util.Calendar`.

- When converting from a `String`, omitting `Mapping#dateFormat` results in using the default pattern and date format symbols for the default locale. An exception to this rule is `XmlGregorianCalendar` which results in parsing the `String` according to XML Schema 1.0 Part 2, Section 3.2.7-14.1, Lexical Representation.

5.2. Mapping object references

Typically an object has not only primitive attributes but also references other objects. E.g. the `Car` class could contain a reference to a `Person` object (representing the car's driver) which should be mapped to a `PersonDto` object referenced by the `CarDto` class.

In this case just define a mapping method for the referenced object type as well:

Example 15. Mapper with one mapping method using another

```
@Mapper
public interface CarMapper {

    CarDto carToCarDto(Car car);

    PersonDto personToPersonDto(Person person);
}
```

The generated code for the `carToCarDto()` method will invoke the `personToPersonDto()` method for mapping the `driver` attribute, while the generated implementation for `personToPersonDto()` performs the mapping of person objects.

That way it is possible to map arbitrary deep object graphs. When mapping from entities into data transfer objects it is often useful to cut references to other entities at a certain point. To do so, implement a custom mapping method (see the next section) which e.g. maps a referenced entity to its id in the target object.

When generating the implementation of a mapping method, MapStruct will apply the following routine for each attribute pair in the source and target object:

- If source and target attribute have the same type, the value will be simply copied from source to target. If the attribute is a collection (e.g. a `List`) a copy of the collection will be set into the target attribute.
- If source and target attribute type differ, check whether there is a another mapping method which has the type of the source attribute as parameter type and the type of the target attribute as return type. If such a method exists it will be invoked in the generated mapping implementation.
- If no such method exists MapStruct will look whether a built-in conversion for the source and target type of the attribute exists. If this is the case, the generated mapping code will apply this conversion.
- Otherwise an error will be raised at build time, indicating the non-mappable attribute.

5.3. Invoking other mappers

In addition to methods defined on the same mapper type MapStruct can also invoke mapping methods defined in other classes, be it mappers generated by MapStruct or hand-written mapping methods. This can be useful to structure your mapping code in several classes (e.g. with on mapper type per application module) or you want to provide custom mapping logic which can't be generated by MapStruct.

For instance the `Car` class might contain an attribute `manufacturingDate` while the corresponding DTO attribute is of type String. In order to map this attribute, you could implement a mapper class like this:

Example 16. Manually implemented mapper class

```
public class DateMapper {

    public String asString(Date date) {
        return date != null ? new SimpleDateFormat( "yyyy-MM-dd" )
            .format( date ) : null;
    }

    public Date asDate(String date) {
        try {
            return date != null ? new SimpleDateFormat( "yyyy-MM-dd" )
                .parse( date ) : null;
        }
        catch ( ParseException e ) {
            throw new RuntimeException( e );
        }
    }
}
```

In the `@Mapper` annotation at the `CarMapper` interface reference the `DateMapper` class like this:

Example 17. Referencing another mapper class

```
@Mapper(uses=DateMapper.class)
public class CarMapper {

    CarDto carToCarDto(Car car);
}
```

When generating code for the implementation of the `carToCarDto()` method, MapStruct will look for a method which maps a `Date` object into a String, find it on the `DateMapper` class and generate an invocation of `asString()` for mapping the `manufacturingDate` attribute.

Generated mappers retrieve referenced mappers using the component model configured for them. If e.g. CDI was used as component model for `CarMapper`, `DateMapper` would have to be a CDI bean as well. When using the default component model, any hand-written mapper classes to be referenced by MapStruct generated mappers must declare a public no-args constructor in order to be instantiable.

5.4. Passing the mapping target type to custom mappers

When having a custom mapper hooked into the generated mapper with `@Mapper#uses()`, an additional parameter of type `Class` (or a super-type of it) can be defined in the custom mapping method in order to perform general mapping tasks for specific target object types. That attribute must be annotated with `@TargetType` for MapStruct to generate calls that pass the `Class` instance representing the corresponding property type of the target bean.

For instance, the `CarDto` could have a property `owner` of type `Reference` that contains the primary key of a `Person` entity. You could now create a generic custom mapper that resolves any `Reference` objects to their corresponding managed JPA entity instances.

Example 18. Mapping method expecting mapping target type as parameter

```
@ApplicationScoped // CDI component model
public class ReferenceMapper {

    @PersistenceContext
    private EntityManager entityManager;

    public <T extends BaseEntity> T resolve(Reference reference, @TargetType
Class<T> entityClass) {
        return reference != null ? entityManager.find( entityClass, reference
.getPk() ) : null;
    }

    public Reference toReference(BaseEntity entity) {
        return entity != null ? new Reference( entity.getPk() ) : null;
    }
}

@Mapper(componentModel = "cdi", uses = ReferenceMapper.class )
public interface CarMapper {

    Car carDtoToCar(CarDto carDto);
}
```

MapStruct will then generate something like this:

Example 19. Generated code

```
//GENERATED CODE
@ApplicationScoped
public class CarMapperImpl implements CarMapper {

    @Inject
    private ReferenceMapper referenceMapper;

    @Override
    public Car carDtoToCar(CarDto carDto) {
        if ( carDto == null ) {
            return null;
        }

        Car car = new Car();

        car.setOwner( referenceMapper.resolve( carDto.getOwner(), Owner.class ) );
        // ...

        return car;
    }
}
```

5.5. Mapping method resolution

When mapping a property from one type to another, MapStruct looks for the most specific method which maps the source type into the target type. The method may either be declared on the same mapper interface or on another mapper which is registered via `@Mapper#uses()`. The same applies for factory methods (see Object factories).

The algorithm for finding a mapping or factory method resembles Java's method resolution algorithm as much as possible. In particular, methods with a more specific source type will take precedence (e.g. if there are two methods, one which maps the searched source type, and another one which maps a super-type of the same). In case more than one most-specific method is found, an error will be raised.

When working with JAXB, e.g. when converting a `String` to a corresponding `JAXBElement<String>`, MapStruct will take the `scope` and `name` attributes of `@XmlElementDecl` annotations into account when looking for a mapping method. This makes sure that the created `JAXBElement` instances will have the right QNAME value. You can find a test which maps JAXB objects here.

5.6. Mapping method selection based on qualifiers

In many occasions one requires mapping methods with the same method signature (appart from

the name) that have different behavior. MapStruct has a handy mechanism to deal with such situations: `@Qualifier`. A 'qualifier' is a custom annotation that the user can write, 'stick onto' a mapping method which is included as used mapper, and can be referred to in a bean property mapping, iterable mapping or map mapping. Multiple qualifiers can be 'stuck onto' a method and mapping.

So, lets say there is a hand-written method to map titles with a `String` return type and `String` argument amongst many other referenced mappers with the same `String` return type - `String` argument signature:

Example 20. Several mapping methods with identical source and target types

```
public class Titles {

    public String translateTitleEG(String title) {
        // some mapping logic
    }

    public String translateTitleGE(String title) {
        // some mapping logic
    }
}
```

And a mapper using this handwritten mapper, in which source and target have a property 'title' that should be mapped:

Example 21. Mapper causing an ambiguous mapping method error

```
@Mapper( uses = Titles.class )
public interface MovieMapper {

     GermanRelease toGerman( OriginalRelease movies );

}
```

Without the use of qualifiers, this would result in an ambiguous mapping method error, because 2 qualifying methods are found (`translateTitleEG`, `translateTitleGE`) and MapStruct would not have a hint which one to choose.

Enter the qualifier approach:

Example 22. Declaring a qualifier type

```
@Qualifier
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.CLASS)
public @interface TitleTranslator {
}
```

And, some qualifiers to indicate which translator to use to map from source language to target language:

Example 23. Declaring qualifier types for mapping methods

```
@Qualifier
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.CLASS)
public @interface EnglishToGerman {
}
```

```
@Qualifier
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.CLASS)
public @interface GermanToEnglish {
}
```

Please take note of the retention `TitleTranslator` on class level, `EnglishToGerman`, `GermanToEnglish` on method level!

Then, using the qualifiers, the mapping could look like this:

Example 24. Mapper using qualifiers

```
@Mapper( uses = Titles.class )
public interface MovieMapper {

     @Mapping( target = "title", qualifiedBy = { TitleTranslator.class,
EnglishToGerman.class } )
     GermanRelease toGerman( OriginalRelease movies );

}
```

Example 25. Custom mapper qualifying the methods it provides

```
@TitleTranslator
public class Titles {

    @EnglishToGerman
    public String translateTitleEG(String title) {
        // some mapping logic
    }

    @GermanToEnglish
    public String translateTitleGE(String title) {
        // some mapping logic
    }
}
```

A class / method annotated with a qualifier will not qualify anymore for mappings that do not have the `qualifiedBy` element.

The same mechanism is also present on bean mappings: `@BeanMapping#qualifiedBy`: it selects the factory method marked with the indicated qualifier.

In many occasions, declaring a new annotation to aid the selection process can be too much for what you try to achieve. For those situations, MapStruct has the `@Named` annotation. This annotation is a pre-defined qualifier (annotated with `@Qualifier` itself) and can be used to name a Mapper or, more directly a mapping method by means of its value. The same example above would look like:

Example 26. Custom mapper, annotating the methods to qualify by means of `@Named`

```
@Named("TitleTranslator")
public class Titles {

    @Named("EnglishToGerman")
    public String translateTitleEG(String title) {
        // some mapping logic
    }

    @Named("GermanToEnglish")
    public String translateTitleGE(String title) {
        // some mapping logic
    }
}
```

Example 27. Mapper using named

```
@Mapper( uses = Titles.class )
public interface MovieMapper {

     @Mapping( target = "title", qualifiedByName = { "TitleTranslator",
"EnglishToGerman" } )
     GermanRelease toGerman( OriginalRelease movies );

}
```

Although the used mechanism is the same, the user has to be a bit more careful. Refactoring the name of a defined qualifier in an IDE will neatly refactor all other occurrences as well. This is obviously not the case for changing a name.

6. Mapping collections

The mapping of collection types (`List`, `Set` etc.) is done in the same way as mapping bean types, i.e. by defining mapping methods with the required source and target types in a mapper interface. MapStruct supports a wide range of iterable types from the Java Collection Framework.

The generated code will contain a loop which iterates over the source collection, converts each element and puts it into the target collection. If a mapping method for the collection element types is found in the given mapper or the mapper it uses, this method is invoked to perform the element conversion. Alternatively, if an implicit conversion for the source and target element types exists, this conversion routine will be invoked. The following shows an example:

Example 28. Mapper with collection mapping methods

```
@Mapper
public interface CarMapper {

    Set<String> integerSetToStringSet(Set<Integer> integers);

    List<CarDto> carsToCarDtos(List<Car> cars);

    CarDto carToCarDto(Car car);
}
```

The generated implementation of the `integerSetToStringSet` performs the conversion from `Integer` to `String` for each element, while the generated `carsToCarDtos()` method invokes the `carToCarDto()` method for each contained element as shown in the following:

1. Generated collection mapping methods

```
//GENERATED CODE
@Override
public Set<String> integerSetToStringSet(Set<Integer> integers) {
    if ( integers == null ) {
        return null;
    }

    Set<String> set = new HashSet<String>();

    for ( Integer integer : integers ) {
        set.add( String.valueOf( integer ) );
    }

    return set;
}

@Override
public List<CarDto> carsToCarDtos(List<Car> cars) {
    if ( cars == null ) {
        return null;
    }

    List<CarDto> list = new ArrayList<CarDto>();

    for ( Car car : cars ) {
        list.add( carToCarDto( car ) );
    }

    return list;
}
```

Note that MapStruct will look for a collection mapping method with matching parameter and return type, when mapping a collection-typed attribute of a bean, e.g. from `Car#passengers` (of type `List<Person>`) to `CarDto#passengers` (of type `List<PersonDto>`).

Example 29. Usage of collection mapping method to map a bean property

```
//GENERATED CODE
carDto.setPassengers( personsToPersonDtos( car.getPassengers() ) );
...
```

Some frameworks and libraries only expose JavaBeans getters but no setters for collection-typed properties. Types generated from an XML schema using JAXB adhere to this pattern by default. In this case the generated code for mapping such a property invokes its getter and adds all the mapped elements:

Example 30. Usage of an adding method for collection mapping

```
//GENERATED CODE
carDto.getPassengers().addAll( personsToPersonDtos( car.getPassengers() ) );
...
```

It is not allowed to declare mapping methods with an iterable source and a non-iterable target or the other way around. An error will be raised when detecting this situation.

6.1. Mapping maps

Also map-based mapping methods are supported. The following shows an example:

Example 31. Map mapping method

```
public interface SourceTargetMapper {

    @MapMapping(valueDateFormat = "dd.MM.yyyy")
    Map<String, String> longDateMapToStringStringMap(Map<Long, Date> source);
}
```

Similar to iterable mappings, the generated code will iterate through the source map, convert each value and key (either by means of an implicit conversion or by invoking another mapping method) and put them into the target map:

Example 32. Generated implementation of map mapping method

```
//GENERATED CODE
@Override
public Map<Long, Date> stringStringMapToLongDateMap(Map<String, String> source) {
    if ( source == null ) {
        return null;
    }

    Map<Long, Date> map = new HashMap<Long, Date>();

    for ( Map.Entry<String, String> entry : source.entrySet() ) {

        Long key = Long.parseLong( entry.getKey() );
        Date value;
        try {
            value = new SimpleDateFormat( "dd.MM.yyyy" ).parse( entry.getValue()
);
        }
        catch( ParseException e ) {
            throw new RuntimeException( e );
        }

        map.put( key, value );
    }

    return map;
}
```

6.2. Collection mapping strategies

MapStruct has a `CollectionMappingStrategy`, with the possible values: `ACCESSOR_ONLY`, `SETTER_PREFERRED` and `ADDER_PREFERRED`.

In the table below, the dash `-` indicates a property name. Next, the trailing `s` indicates the plural form. The table explains the options and how they are apply to the presence/absense of a `set-s`, `add-` and / or `get-s` method on the target object:

Table 2. Collection mapping strategy options

Option	Only target set-s Available	Only target add- Available	Both set-s / add- Available	No set-s / add- Available	Existing Target(`@TargetType`)
`ACCESSOR_ONLY`	set-s	get-s	set-s	get-s	get-s
`SETTER_PREFERRED`	set-s	add-	set-s	get-s	get-s

Option	Only target set-s Available	Only target add- Available	Both set-s / add- Available	No set-s / add- Available	Existing Target(`@TargetType`)
`ADDER_PREFERRED`	set-s	add-	add-	get-s	get-s

Some background: An `adder` method is typically used in case of generated (JPA) entities, to add a single element (entity) to an underlying collection. Invoking the adder establishes a parent-child relation between parent - the bean (entity) on which the adder is invoked - and its child(ren), the elements (entities) in the collection. To find the appropriate `adder`, MapStruct will try to make a match between the generic parameter type of the underlying collection and the single argument of a candidate `adder`. When there are more candidates, the plural `setter` / `getter` name is converted to singular and will be used in addition to make a match.

The option `DEFAULT` should not be used explicitely. It is used to distinguish between an explicit user desire to override the default in a `@MapperConfig` from the implicit Mapstruct choice in a `@Mapper`. The option `DEFAULT` is synonymous to `ACCESSOR_ONLY`.

When working with an `adder` method and JPA entities, Mapstruct assumes that the target collections are initialized with a collection implementation (e.g. an `ArrayList`). You can use factories to create a new target entity with intialized collections in stead of Mapstruct creating the target entity by its constructor.

6.3. Implementation types used for collection mappings

When an iterable or map mapping method declares an interface type as return type, one of its implementation types will be instantiated in the generated code. The following table shows the supported interface types and their corresponding implementation types as instantiated in the generated code:

Table 3. Collection mapping implementation types

Interface type	Implementation type
`Iterable`	`ArrayList`
`Collection`	`ArrayList`
`List`	`ArrayList`
`Set`	`HashSet`
`SortedSet`	`TreeSet`
`NavigableSet`	`TreeSet`
`Map`	`HashMap`
`SortedMap`	`TreeMap`
`NavigableMap`	`TreeMap`
`ConcurrentMap`	`ConcurrentHashMap`

Interface type	Implementation type
`ConcurrentNavigableMap`	`ConcurrentSkipListMap`

7. Mapping Values

7.1. Mapping enum types

MapStruct supports the generation of methods which map one Java enum type into another.

By default, each constant from the source enum is mapped to a constant with the same name in the target enum type. If required, a constant from the source enum may be mapped to a constant with another name with help of the `@ValueMapping` annotation. Several constants from the source enum can be mapped to the same constant in the target type.

The following shows an example:

Example 33. Enum mapping method

```
@Mapper
public interface OrderMapper {

    OrderMapper INSTANCE = Mappers.getMapper( OrderMapper.class );

    @ValueMappings({
        @ValueMapping(source = "EXTRA", target = "SPECIAL"),
        @ValueMapping(source = "STANDARD", target = "DEFAULT"),
        @ValueMapping(source = "NORMAL", target = "DEFAULT")
    })
    ExternalOrderType orderTypeToExternalOrderType(OrderType orderType);
}
```

Example 34. Enum mapping method result

```
// GENERATED CODE
public class OrderMapperImpl implements OrderMapper {

    @Override
    public ExternalOrderType orderTypeToExternalOrderType(OrderType orderType) {
        if ( orderType == null ) {
            return null;
        }

        ExternalOrderType externalOrderType_;

        switch ( orderType ) {
            case EXTRA: externalOrderType_ = ExternalOrderType.SPECIAL;
            break;
            case STANDARD: externalOrderType_ = ExternalOrderType.DEFAULT;
            break;
            case NORMAL: externalOrderType_ = ExternalOrderType.DEFAULT;
            break;
            case RETAIL: externalOrderType_ = ExternalOrderType.RETAIL;
            break;
            case B2B: externalOrderType_ = ExternalOrderType.B2B;
            break;
            default: throw new IllegalArgumentException( "Unexpected enum
constant: " + orderType );
        }

        return externalOrderType_;
    }
}
```

By default an error will be raised by MapStruct in case a constant of the source enum type does not have a corresponding constant with the same name in the target type and also is not mapped to another constant via `@ValueMapping`. This ensures that all constants are mapped in a safe and predictable manner. The generated mapping method will throw an IllegalStateException if for some reason an unrecognized source value occurs.

MapStruct also has a mechanism for mapping any remaining (unspecified) mappings to a default. This can be used only once in a set of value mappings. It comes in two flavors: `<ANY_REMAINING>` and `<ANY_UNMATCHED>`.

In case of source `<ANY_REMAINING>` MapStruct will continue to map a source enum constant to a target enum constant with the same name. The remainder of the source enum constants will be mapped to the target specified in the `@ValueMapping` with `<ANY_REMAINING>` source.

MapStruct will **not** attempt such name based mapping for `<ANY_UNMATCHED>` and directly apply the target specified in the `@ValueMapping` with `<ANY_UNMATCHED>` source to the remainder.

MapStruct is able to handle `null` sources and `null` targets by means of the `<NULL>` keyword.

Constants for `<ANY_REMAINING>`, `<ANY_UNMAPPED>` and `<NULL>` are available in the `MappingConstants` class.

Finally `@InheritInverseConfiguration` and `@InheritConfiguration` can be used in combination with `@ValueMappings`.

Example 35. Enum mapping method, <NULL> and <ANY_REMAINING>

```
@Mapper
public interface SpecialOrderMapper {

    SpecialOrderMapper INSTANCE = Mappers.getMapper( SpecialOrderMapper.class );

    @ValueMappings({
        @ValueMapping( source = MappingConstants.NULL, target = "DEFAULT" ),
        @ValueMapping( source = "STANDARD", target = MappingConstants.NULL ),
        @ValueMapping( source = MappingConstants.ANY_REMAINING, target = "SPECIAL"
)
    })
    ExternalOrderType orderTypeToExternalOrderType(OrderType orderType);
}
```

Example 36. Enum mapping method result, <NULL> and <ANY_REMAINING>

```
// GENERATED CODE
public class SpecialOrderMapperImpl implements SpecialOrderMapper {

    @Override
    public ExternalOrderType orderTypeToExternalOrderType(OrderType orderType) {
        if ( orderType == null ) {
            return ExternalOrderType.DEFAULT;
        }

        ExternalOrderType externalOrderType_;

        switch ( orderType ) {
            case STANDARD: externalOrderType_ = null;
            break;
            case RETAIL: externalOrderType_ = ExternalOrderType.RETAIL;
            break;
            case B2B: externalOrderType_ = ExternalOrderType.B2B;
            break;
            default: externalOrderType_ = ExternalOrderType.SPECIAL;
        }

        return externalOrderType_;
    }
}
```

Note: MapStruct would have refrained from mapping the `RETAIL` and `B2B` when `<ANY_UNMAPPED>` was used instead of `<ANY_REMAINING>`.

The mapping of enum to enum via the `@Mapping` annotation is **DEPRECATED**. It will be removed from future versions of MapStruct. Please adapt existing enum mapping methods to make use of `@ValueMapping` instead.

8. Object factories

By default, the generated code for mapping one bean type into another will call the default constructor to instantiate the target type.

Alternatively you can plug in custom object factories which will be invoked to obtain instances of the target type. One use case for this is JAXB which creates `ObjectFactory` classes for obtaining new instances of schema types.

To do make use of custom factories register them via `@Mapper#uses()` as described in Invoking other mappers. When creating the target object of a bean mapping, MapStruct will look for a parameterless method, or a method with only one `@TargetType` parameter that returns the required target type and invoke this method instead of calling the default constructor:

Example 37. Custom object factories

```
public class DtoFactory {

     public CarDto createCarDto() {
         return // ... custom factory logic
     }
}
```

```
public class EntityFactory {

     public <T extends BaseEntity> T createEntity(@TargetType Class<T>
entityClass) {
         return // ... custom factory logic
     }
}
```

```
@Mapper(uses= { DtoFactory.class, EntityFactory.class } )
public interface CarMapper {

    OrderMapper INSTANCE = Mappers.getMapper( CarMapper.class );

    CarDto carToCarDto(Car car);

    Car carDtoToCar(CarDto carDto);
}
```

```
//GENERATED CODE
public class CarMapperImpl implements CarMapper {

    private final DtoFactory dtoFactory = new DtoFactory();

    private final EntityFactory entityFactory = new EntityFactory();

    @Override
    public CarDto carToCarDto(Car car) {
        if ( car == null ) {
            return null;
        }

        CarDto carDto = dtoFactory.createCarDto();

        //map properties...

        return carDto;
    }

    @Override
    public Car carDtoToCar(CarDto carDto) {
        if ( carDto == null ) {
            return null;
        }

        Car car = entityFactory.createEntity( Car.class );

        //map properties...

        return car;
    }
}
```

9. Advanced mapping options

This chapter describes several advanced options which allow to fine-tune the behavior of the generated mapping code as needed.

9.1. Default values and constants

Default values can be specified to set a predefined value to a target property if the corresponding source property is `null`. Constants can be specified to set such a predefined value in any case. Default values and constants are specified as String values and are subject to type conversion either via built-in conversions or the invocation of other mapping methods in order to match the type required by the target property.

A mapping with a constant must not include a reference to a source property. The following examples shows some mappings using default values and constants:

Example 38. Mapping method with default values and constants

```
@Mapper(uses = StringListMapper.class)
public interface SourceTargetMapper {

    SourceTargetMapper INSTANCE = Mappers.getMapper( SourceTargetMapper.class );

    @Mappings( {
        @Mapping(target = "stringProperty", source = "stringProp", defaultValue =
"undefined"),
        @Mapping(target = "longProperty", defaultValue = "-1"),
        @Mapping(target = "stringConstant", constant = "Constant Value"),
        @Mapping(target = "integerConstant", constant = "14"),
        @Mapping(target = "longWrapperConstant", constant = "3001"),
        @Mapping(target = "dateConstant", dateFormat = "dd-MM-yyyy", constant =
"09-01-2014"),
        @Mapping(target = "stringListConstants", constant = "jack-jill-tom")
    } )
    Target sourceToTarget(Source s);
}
```

If `s.getStringProp() == null`, then the target property `stringProperty` will be set to `"undefined"` instead of applying the value from `s.getStringProp()`. If `s.getLongProp() == null`, then the target property `longProperty` will be set to `-1`. The String `"Constant Value"` is set as is to the target property `stringConstant`. The value `"3001"` is type-converted to the `Long` (wrapper) class of target property `longWrapperConstant`. Date properties also require a date format. The constant `"jack-jill-tom"` demonstrates how the hand-written class `StringListMapper` is invoked to map the dash-separated list into a `List<String>`.

9.2. Expressions

By means of Expressions it will be possible to include constructs from a number of languages.

Currently only Java is supported as language. This feature is e.g. useful to invoke constructors. The entire source object is available for usage in the expression. Care should be taken to insert only valid Java code: MapStruct will not validate the expression at generation-time, but errors will show up in the generated classes during compilation.

The example below demonstrates how two source properties can be mapped to one target:

Example 39. Mapping method using an expression

```
@Mapper
public interface SourceTargetMapper {

    SourceTargetMapper INSTANCE = Mappers.getMapper( SourceTargetMapper.class );

    @Mapping(target = "timeAndFormat",
         expression = "java( new org.sample.TimeAndFormat( s.getTime(),
s.getFormat() ) )")
    Target sourceToTarget(Source s);
}
```

The example demonstrates how the source properties `time` and `format` are composed into one target property `TimeAndFormat`. Please note that the fully qualified package name is specified because MapStruct does not take care of the import of the `TimeAndFormat` class (unless its used otherwise explicitly in the `SourceTargetMapper`). This can be resolved by defining `imports` on the `@Mapper` annotation.

Example 40. Declaring an import

```
imports org.sample.TimeAndFormat;

@Mapper( imports = TimeAndFormat.class )
public interface SourceTargetMapper {

    SourceTargetMapper INSTANCE = Mappers.getMapper( SourceTargetMapper.class );

    @Mapping(target = "timeAndFormat",
         expression = "java( new TimeAndFormat( s.getTime(), s.getFormat() ) )")
    Target sourceToTarget(Source s);
}
```

9.3. Determining the result type

When result types have an inheritance relation, selecting either mapping method (`@Mapping`) or a factory method (`@BeanMapping`) can becomes ambigious. Suppose an Apple and a Banana, which is are both specializations of Fruit.

Example 41. Specifying the result type of a bean mapping method

```
@Mapper( uses = FruitFactory.class )
public interface FruitMapper {

    @BeanMapping( resultType = Apple.class )
    Fruit map( FruitDto source );

}
```

```
public class FruitFactory {

    public Apple createApple() {
        return new Apple( "Apple" );
    }

    public Banana createBanana() {
        return new Banana( "Banana" );
    }
}
```

So, which `Fruit` must be factorized in the mapping method `Fruit map(FruitDto source);`? A `Banana` or an `Apple`? Here's were the `@BeanMapping#resultType` comes in handy. It controls the factory method to select, or in absence of a factory method, the return type to create.

The same mechanism is present on mapping: `@Mapping#resultType` and works like you expect it would: it selects the mapping method with the desired result type when present.

The mechanism is also present on iterable mapping and map mapping. `@IterableMapping#elementTargetType` is used to select the mapping method with the desired element in the resulting `Iterable`. For the `@MapMapping` a similar purpose is served by means of `#MapMapping#keyTargetType` and `MapMapping#valueTargetType`.

9.4. Controlling mapping result for 'null' arguments

MapStruct offers control over the object to create when the source argument of the mapping method equals `null`. By default `null` will be returned.

However, by specifying `nullValueMappingStrategy = NullValueMappingStrategy.RETURN_DEFAULT` on `@BeanMapping`, `@IterableMapping`, `@MapMapping`, or globally on `@Mapper` or `@MappingConfig`, the mapping result can be altered to return empty **default** values. This means for:

- **Bean mappings**: an 'empty' target bean will be returned, with the exception of constants and

expressions, they will be populated when present.

- **Primitives**: the default values for primitives will be returned, e.g. `false` for `boolean` or `0` for `int`.
- **Iterables / Arrays**: an empty iterable will be returned.
- **Maps**: an empty map will be returned.

The strategy works in a hierarchical fashion. Setting `nullValueMappingStrategy` on mapping method level will override `@Mapper#nullValueMappingStrategy`, and `@Mapper#nullValueMappingStrategy` will override `@MappingConfig#nullValueMappingStrategy`.

9.5. Exceptions

Calling applications may require handling of exceptions when calling a mapping method. These exceptions could be thrown by hand-written logic and by the generated built-in mapping methods or type-conversions of MapStruct. When the calling application requires handling of exceptions, a throws clause can be defined in the mapping method:

Example 42. Mapper using custom method declaring checked exception

```
@Mapper(uses = HandWritten.class)
public interface CarMapper {

    CarDto carToCarDto(Car car) throws GearException;
}
```

The hand written logic might look like this:

Example 43. Custom mapping method declaring checked exception

```
public class HandWritten {

    private static final String[] GEAR = {"ONE", "TWO", "THREE", "OVERDRIVE",
"REVERSE"};

    public String toGear(Integer gear) throws GearException, FatalException {
        if ( gear == null ) {
            throw new FatalException("null is not a valid gear");
        }

        if ( gear < 0 && gear > GEAR.length ) {
            throw new GearException("invalid gear");
        }
        return GEAR[gear];
    }
}
```

MapStruct now, wraps the `FatalException` in a `try-catch` block and rethrows an unchecked `RuntimeException`. MapStruct delegates handling of the `GearException` to the application logic because it is defined as throws clause in the `carToCarDto` method:

Example 44. try-catch block in generated implementation

```
// GENERATED CODE
@Override
public CarDto carToCarDto(Car car) throws GearException {
    if ( car == null ) {
        return null;
    }

    CarDto carDto = new CarDto();
    try {
        carDto.setGear( handWritten.toGear( car.getGear() ) );
    }
    catch ( FatalException e ) {
        throw new RuntimeException( e );
    }

    return carDto;
}
```

Some **notes** on null checks. MapStruct does provide null checking only when required: when applying type-conversions or constructing a new type by invoking its constructor. This means that the user is responsible in hand-written code for returning valid non-null objects. Also null objects can be handed to hand-written code, since MapStruct does not want to make assumptions on the meaning assigned by the user to a null object. Hand-written code has to deal with this.

10. Reusing mapping configurations

This chapter discusses different means of reusing mapping configurations for several mapping methods: "inheritance" of configuration from other methods and sharing central configuration between multiple mapper types.

10.1. Mapping configuration inheritance

Method-level configuration annotations such as `@Mapping`, `@BeanMapping`, `@IterableMapping`, etc., can be **inherited** from one mapping method to a **similar** method using the annotation `@InheritConfiguration`:

Example 45. Update method inheriting its configuration

```
@Mapper
public interface CarMapper {

    @Mapping(target = "numberOfSeats", source = "seatCount")
    Car carDtoToCar(CarDto car);

    @InheritConfiguration
    void carDtoIntoCar(CarDto carDto, @MappingTarget Car car);
}
```

The example above declares a mapping method `carToDto()` with a configuration to define how the property `numberOfSeats` in the type `Car` shall be mapped. The update method that performs the mapping on an existing instance of `Car` needs the same configuration to successfully map all properties. Declaring `@InheritConfiguration` on the method lets MapStruct search for inheritance candidates to apply the annotations of the method that is inherited from.

One method **A** can inherit the configuration from another method **B** if all types of **A** (source types and result type) are assignable to the corresponding types of **B**.

Methods that are considered for inheritance need to be defined in the current mapper, a super class/interface, or in the shared configuration interface (as described in Shared configurations).

In case more than one method is applicable as source for the inheritance, the method name must be specified within the annotation: `@InheritConfiguration( name = "carDtoToCar" )`.

A method can use `@InheritConfiguration` and override or amend the configuration by additionally applying `@Mapping`, `@BeanMapping`, etc.

10.2. Inverse mappings

In case of bi-directional mappings, e.g. from entity to DTO and from DTO to entity, the mapping rules for the forward method and the reverse method are often similar and can simply be inversed by switching `source` and `target`.

Use the annotation `@InheritInverseConfiguration` to indicate that a method shall inherit the inverse configuration of the corresponding reverse method.

Example 46. Inverse mapping method inheriting its configuration

```
@Mapper
public interface CarMapper {

    @Mapping(source = "numberOfSeats", target = "seatCount")
    CarDto carToDto(Car car);

    @InheritInverseConfiguration
    Car carDtoToCar(CarDto carDto);
}
```

Here the `carDtoToCar()` method is the reverse mapping method for `carToDto()`. Note that any attribute mappings from `carToDto()` will be applied to the corresponding reverse mapping method as well. They are automatically reversed and copied to the method with the `@InheritInverseConfiguration` annotation.

Specific mappings from the inversed method can (optionally) be overridden by `ignore`, `expression` or `constant` in the mapping, e.g. like this: `@Mapping(target = "numberOfSeats", ignore=true)`.

A method **A** is considered a **reverse** method of a method **B**, if the result type of **A** is the **same** as the single source type of **B** and if the single source type of **A** is the **same** as the result type of **B**.

Methods that are considered for inverse inheritance need to be defined in the current mapper, a super class/interface.

If multiple methods qualify, the method from which to inherit the configuration from needs to be specified using the `name` property like this: `@InheritInverseConfiguration(name = "carToDto")`.

Nested properties are excluded (silently ignored) from reverse mapping. The same holds true for expressions and constants. Reverse mapping will take place automatically when the source property name and target property name are identical. Otherwise, `@Mapping` should specify both the target name and source name. In all cases, a suitable mapping method needs to be in place for the reverse mapping.

10.3. Shared configurations

MapStruct offers the possibility to define a shared configuration by pointing to a central interface annotated with `@MapperConfig`. For a mapper to use the shared configuration, the configuration interface needs to be defined in the `@Mapper#config` property.

The `@MapperConfig` annotation has the same attributes as the `@Mapper` annotation. Any attributes not given via `@Mapper` will be inherited from the shared configuration. Attributes specified in `@Mapper` take precedence over the attributes specified via the referenced configuration class. List properties such as `uses` are simply combined:

Example 47. Mapper configuration class and mapper using it

```
@MapperConfig(
    uses = CustomMapperViaMapperConfig.class,
    unmappedTargetPolicy = ReportingPolicy.ERROR
)
public interface CentralConfig {
}
```

```
@Mapper(config = CentralConfig.class, uses = { CustomMapperViaMapper.class } )
// Effective configuration:
// @Mapper(
//     uses = { CustomMapperViaMapper.class, CustomMapperViaMapperConfig.class },
//     unmappedTargetPolicy = ReportingPolicy.ERROR
// )
public interface SourceTargetMapper {
  ...
}
```

The interface holding the `@MapperConfig` annotation may also declare **prototypes** of mapping methods that can be used to inherit method-level mapping annotations from. Such prototype methods are not meant to be implemented or used as part of the mapper API.

Example 48. Mapper configuration class with prototype methods

```
@MapperConfig(
    uses = CustomMapperViaMapperConfig.class,
    unmappedTargetPolicy = ReportingPolicy.ERROR,
    mappingInheritanceStrategy = MappingInheritanceStrategy
.AUTO_INHERIT_FROM_CONFIG
)
public interface CentralConfig {

    // Not intended to be generated, but to carry inheritable mapping annotations:
    @Mapping(target = "primaryKey", source = "technicalKey")
    BaseEntity anyDtoToEntity(BaseDto dto);
}
```

```
@Mapper(config = CentralConfig.class, uses = { CustomMapperViaMapper.class } )
public interface SourceTargetMapper {

    @Mapping(target = "numberOfSeats", source = "seatCount")
    // additionally inherited from CentralConfig, because Car extends BaseEntity
and CarDto extends BaseDto:
    // @Mapping(target = "primaryKey", source = "technicalKey")
    Car toCar(CarDto car)
}
```

The attributes `@Mapper#mappingInheritanceStrategy()` / `@MapperConfig#mappingInheritanceStrategy()` configure when the method-level mapping configuration annotations are inherited from prototype methods in the interface to methods in the mapper:

- `EXPLICIT` (default): the configuration will only be inherited, if the target mapping method is annotated with `@InheritConfiguration` and the source and target types are assignable to the corresponding types of the prototype method, all as described in Mapping configuration inheritance.
- `AUTO_INHERIT_FROM_CONFIG`: the configuration will be inherited automatically, if the source and target types of the target mapping method are assignable to the corresponding types of the prototype method. If multiple prototype methods match, the ambiguity must be resolved using `@InheritConfiguration(name = …)`.

11. Customizing mappings

Sometimes it's needed to apply custom logic before or after certain mapping methods. MapStruct provides two ways for doing so: decorators which allow for a type-safe customization of specific mapping methods and the before-mapping and after-mapping lifecycle methods which allow for a generic customization of mapping methods with given source or target types.

11.1. Mapping customization with decorators

In certain cases it may be required to customize a generated mapping method, e.g. to set an additional property in the target object which can't be set by a generated method implementation. MapStruct supports this requirement using decorators.

When working with the component model `cdi`, use CDI decorators with MapStruct mappers instead of the `@DecoratedWith` annotation described here.

To apply a decorator to a mapper class, specify it using the `@DecoratedWith` annotation.

Example 49. Applying a decorator

```
@Mapper
@DecoratedWith(PersonMapperDecorator.class)
public interface PersonMapper {

    PersonMapper INSTANCE = Mappers.getMapper( PersonMapper.class );

    PersonDto personToPersonDto(Person person);

    AddressDto addressToAddressDto(Address address);
}
```

The decorator must be a sub-type of the decorated mapper type. You can make it an abstract class which allows to only implement those methods of the mapper interface which you want to customize. For all non-implemented methods, a simple delegation to the original mapper will be generated using the default generation routine.

The `PersonMapperDecorator` shown below customizes the `personToPersonDto()`. It sets an additional attribute which is not present in the source type of the mapping. The `addressToAddressDto()` method is not customized.

Example 50. Implementing a decorator

```
public abstract class PersonMapperDecorator implements PersonMapper {

    private final PersonMapper delegate;

    public PersonMapperDecorator(PersonMapper delegate) {
        this.delegate = delegate;
    }

    @Override
    public PersonDto personToPersonDto(Person person) {
        PersonDto dto = delegate.personToPersonDto( person );
        dto.setFullName( person.getFirstName() + " " + person.getLastName() );
        return dto;
    }
}
```

The example shows how you can optionally inject a delegate with the generated default implementation and use this delegate in your customized decorator methods.

For a mapper with `componentModel = "default"`, define a constructor with a single parameter which accepts the type of the decorated mapper.

When working with the component models `spring` or `jsr330`, this needs to be handled differently.

11.1.1. Decorators with the Spring component model

When using `@DecoratedWith` on a mapper with component model `spring`, the generated implementation of the original mapper is annotated with the Spring annotation `@Qualifier("delegate")`. To autowire that bean in your decorator, add that qualifier annotation as well:

Example 51. Spring-based decorator

```
public abstract class PersonMapperDecorator implements PersonMapper {

    @Autowired
    @Qualifier("delegate")
    private PersonMapper delegate;

    @Override
    public PersonDto personToPersonDto(Person person) {
        PersonDto dto = delegate.personToPersonDto( person );
        dto.setName( person.getFirstName() + " " + person.getLastName() );

        return dto;
    }
}
```

The generated class that extends the decorator is annotated with Spring's `@Primary` annotation. To autowire the decorated mapper in the application, nothing special needs to be done:

Example 52. Using a decorated mapper

```
@Autowired
private PersonMapper personMapper; // injects the decorator, with the injected
original mapper
```

11.1.2. Decorators with the JSR 330 component model

JSR 330 doesn't specify qualifiers and only allows to specifically name the beans. Hence, the generated implementation of the original mapper is annotated with `@Named("fully-qualified-name-of-generated-implementation")` (please note that when using a decorator, the class name of the mapper implementation ends with an underscore). To inject that bean in your decorator, add the same annotation to the delegate field (e.g. by copy/pasting it from the generated class):

Example 53. JSR 330 based decorator

```
public abstract class PersonMapperDecorator implements PersonMapper {

    @Inject
    @Named("org.examples.PersonMapperImpl_")
    private PersonMapper delegate;

    @Override
    public PersonDto personToPersonDto(Person person) {
        PersonDto dto = delegate.personToPersonDto( person );
        dto.setName( person.getFirstName() + " " + person.getLastName() );

        return dto;
    }
}
```

Unlike with the other component models, the usage site must be aware if a mapper is decorated or not, as for decorated mappers, the parameterless `@Named` annotation must be added to select the decorator to be injected:

Example 54. Using a decorated mapper with JSR 330

```
@Inject
@Named
private PersonMapper personMapper; // injects the decorator, with the injected
original mapper
```

`@DecoratedWith` in combination with component model `jsr330` is considered experimental as of the 1.0.0.CR2 release. The way the original mapper is referenced in the decorator or the way the decorated mapper is injected in the application code might still change.

11.2. Mapping customization with before-mapping and after-mapping methods

Decorators may not always fit the needs when it comes to customizing mappers. For example, if you need to perform the customization not only for a few selected methods, but for all methods that map specific super-types: in that case, you can use **callback methods** that are invoked before the mapping starts or after the mapping finished.

Callback methods can be implemented in the abstract mapper itself or in a type reference in `Mapper#uses`.

Example 55. Mapper with @BeforeMapping and @AfterMapping hooks

```
@Mapper
public abstract class VehicleMapper {

    @BeforeMapping
    protected void flushEntity(AbstractVehicle vehicle) {
        // I would call my entity manager's flush() method here to make sure my
entity
        // is populated with the right @Version before I let it map into the DTO
    }

    @AfterMapping
    protected void fillTank(AbstractVehicle vehicle, @MappingTarget
AbstractVehicleDto result) {
        result.fuelUp( new Fuel( vehicle.getTankCapacity(), vehicle.getFuelType()
) );
    }

    public abstract CarDto toCarDto(Car car);
}

// Generates something like this:
public class VehicleMapperImpl extends VehicleMapper {

    public CarDto toCarDto(Car car) {
        flushEntity( car );

        if ( car == null ) {
            return null;
        }
        // ...

        fillTank( car, carDto );

        return carDto;
    }
}
```

Only methods with return type `void` may be annotated with `@BeforeMapping` or `@AfterMapping`. The methods may or may not have parameters.

If the `@BeforeMapping` / `@AfterMapping` method has parameters, the method invocation is only generated if all parameters can be **assigned** by the source or target parameters of the mapping method:

- A parameter annotated with `@MappingTarget` is populated with the target instance of the mapping.

- A parameter annotated with `@TargetType` is populated with the target type of the mapping.
- Any other parameter is populated with a source parameter of the mapping, whereas each source parameter is used once at most.

All before/after-mapping methods that **can** be applied to a mapping method **will** be used. Mapping method selection based on qualifiers can be used to further control which methods may be chosen and which not. For that, the qualifier annotation needs to be applied to the before/after-method and referenced in `BeanMapping#qualifiedBy` or `IterableMapping#qualifiedBy`.

The order in which the selected methods are applied is roughly determined by their location of definition (although you should consider it a **code smell** if you need to rely on their order):

- The order of methods within one type can not be guaranteed, as it depends on the compiler and the processing environment implementation.
- Methods declared in one type are used after methods declared in their super-type.
- Methods implemented in the mapper itself are used before methods from types referenced in `Mapper#uses`.
- Types referenced in `Mapper#uses` are searched for before/after-mapping methods in the order specified in the annotation.

`@BeforeMapping` and `@AfterMapping` are considered experimental as of the 1.0.0.CR1 release. Details in the selection of before/after mapping methods that are applicable for a mapping method or the order in which they are called might still be changed.