MapStruct 1.6.0.RC1 Reference Guide

Table of Contents

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 GitHub Discussions 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 repository, or, better yet, help the community and send a pull request for fixing it!

This work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License.

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
  - mappings are incomplete (not all target properties are mapped)
  - mappings are incorrect (cannot find a proper mapping method or type conversion)

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`
- *org.mapstruct:mapstruct-processor*: contains the annotation processor which generates mapper implementations

2.1. Apache Maven

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

Example 1. Maven configuration

```xml
...
<properties>
    <org.mapstruct.version>1.6.0.RC1</org.mapstruct.version>
</properties>
...
<dependencies>
    <dependency>
        <groupId>org.mapstruct</groupId>
        <artifactId>mapstruct</artifactId>
        <version>${org.mapstruct.version}</version>
    </dependency>
</dependencies>
...
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>3.8.1</version>
            <configuration>
                <source>1.8</source>
                <target>1.8</target>
                <annotationProcessorPaths>
                    <path>
                        <groupId>org.mapstruct</groupId>
                        <artifactId>mapstruct-processor</artifactId>
                        <version>${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 annotation 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 1.8 or later (project properties → "Java Compiler" → "Compile Compliance Level"). It will not work with older versions.

2.2. Gradle

Add the following to your Gradle build file in order to enable MapStruct:

Example 2. Gradle configuration

```
...
plugins {
    ...
    id "com.diffplug.eclipse.apt" version "3.26.0" // Only for Eclipse
}

dependencies {
    ...
    implementation "org.mapstruct:mapstruct:$1.6.0.RC1"
    annotationProcessor "org.mapstruct:mapstruct-processor:$1.6.0.RC1"

    // If you are using mapstruct in test code
    testAnnotationProcessor "org.mapstruct:mapstruct-processor:$1.6.0.RC1"
}
...
```

You can find a complete example in the mapstruct-examples project on GitHub.

2.3. Apache Ant

Add the `javac` task configured as follows to your *build.xml* file in order to enable MapStruct in your Ant-based project. Adjust the paths as required for your project layout.

Example 3. Ant configuration

```
...
<javac
    srcdir="src/main/java"
    destdir="target/classes"
    classpath="path/to/mapstruct-1.6.0.RC1.jar">
    <compilerarg line="-processorpath path/to/mapstruct-processor-1.6.0.RC1.jar"/>
    <compilerarg line="-s target/generated-sources"/>
</javac>
...
```

You can find a complete example in the mapstruct-examples project on GitHub.

2.4. 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 `compilerArgs` within the configuration of the Maven processor plug-in like this:

Example 4. Maven configuration

```xml
...
<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>
        <!-- due to problem in maven-compiler-plugin, for verbose mode add
showWarnings -->
        <showWarnings>true</showWarnings>
        <compilerArgs>
            <arg>
                -Amapstruct.suppressGeneratorTimestamp=true
            </arg>
            <arg>
                -Amapstruct.suppressGeneratorVersionInfoComment=true
            </arg>
            <arg>
                -Amapstruct.verbose=true
            </arg>
        </compilerArgs>
    </configuration>
</plugin>
...
```

Example 5. Gradle configuration

```
...
compileJava {
    options.compilerArgs += [
        '-Amapstruct.suppressGeneratorTimestamp=true',
        '-Amapstruct.suppressGeneratorVersionInfoComment=true',
        '-Amapstruct.verbose=true'
    ]
}
...
```

The following options exist:

Table 1. MapStruct processor options

Option	Purpose	Default
`mapstruct.` `suppressGeneratorTimes` `tamp`	If set to `true`, the creation of a time stamp in the `@Generated` annotation in the generated mapper classes is suppressed.	`false`
`mapstruct.verbose`	If set to `true`, MapStruct in which MapStruct logs its major decisions. Note, at the moment of writing in Maven, also `showWarnings` needs to be added due to a problem in the maven-compiler-plugin configuration.	`false`
`mapstruct.` `suppressGeneratorVersi` `onInfoComment`	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 (from javax.enterprise.context or jakarta.enterprise.context, depending on which one is available with javax.inject having priority) 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` (from javax.inject or jakarta.inject, depending which one is available with javax.inject having priority), e.g. using Spring • `jakarta`: the generated mapper is annotated with {@code @Named} and can be retrieved via `@Inject` (from jakarta.inject), e.g. using Spring • `jakarta-cdi`: the generated mapper is an application-scoped (from jakarta.enterprise.context) CDI bean and can be retrieved via `@Inject` If a component model is given for a specific mapper via `@Mapper#componentModel()`, the value from the annotation takes precedence.	`default`

Option	Purpose	Default
`mapstruct.defaultInjectionStrategy`	The type of the injection in mapper via parameter `uses`. This is only used on annotated based component models such as CDI, Spring and JSR 330. Supported values are: • `field`: dependencies will be injected in fields • `constructor`: will be generated constructor. Dependencies will be injected via constructor. When CDI `componentModel` a default constructor will also be generated. If a injection strategy is given for a specific mapper via `@Mapper#injectionStrategy()`, the value from the annotation takes precedence over the option.	`field`
`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. If a policy is given for a specific bean mapping via `@BeanMapping#unmappedTargetPolicy()`, it takes precedence over both `@Mapper#unmappedTargetPolicy()` and the option.	`WARN`

Option	Purpose	Default
`mapstruct.unmappedSourcePolicy`	The default reporting policy to be applied in case an attribute of the source object of a mapping method is not populated with a target value. Supported values are: • `ERROR`: any unmapped source property will cause the mapping code generation to fail • `WARN`: any unmapped source property will cause a warning at build time • `IGNORE`: unmapped source properties are ignored If a policy is given for a specific mapper via `@Mapper#unmappedSourcePolicy()`, the value from the annotation takes precedence. If a policy is given for a specific bean mapping via `@BeanMapping#ignoreUnmappedSourceProperties()`, it takes precedence over both `@Mapper#unmappedSourcePolicy()` and the option.	`IGNORE`
`mapstruct.disableBuilders`	If set to `true`, then MapStruct will not use builder patterns when doing the mapping. This is equivalent to doing `@Mapper( builder = @Builder( disableBuilder = true ) )` for all of your mappers.	`false`

Option	Purpose	Default
`mapstruct.nullValueIterableMappingStrategy`	The strategy to be applied when `null` is passed as a source value to an iterable mapping. Supported values are: - `RETURN_NULL`: if `null` is passed as a source value, then `null` will be returned - `RETURN_DEFAULT`: if `null` is passed then a default value (empty collection) will be returned If a strategy is given for a specific mapper via `@Mapper#nullValueIterableMappingStrategy()`, the value from the annotation takes precedence. If a strategy is given for a specific iterable mapping via `@IterableMapping#nullValueMappingStrategy()`, it takes precedence over both `@Mapper#nullValueIterableMappingStrategy()` and the option.	`RETURN_NULL`
`mapstruct.nullValueMapMappingStrategy`	The strategy to be applied when `null` is passed as a source value to a map mapping. Supported values are: - `RETURN_NULL`: if `null` is passed as a source value, then `null` will be returned - `RETURN_DEFAULT`: if `null` is passed then a default value (empty map) will be returned If a strategy is given for a specific mapper via `@Mapper#nullValueMapMappingStrategy()`, the value from the annotation takes precedence. If a strategy is given for a specific map mapping via `@MapMapping#nullValueMappingStrategy()`, it takes precedence over both `@Mapper#nullValueMapMappingStrategy()` and the option.	`RETURN_NULL`

2.5. Using MapStruct with the Java Module System

MapStruct can be used with Java 9 and higher versions.

To allow usage of the `@Generated` annotation `java.annotation.processing.Generated` (part of the `java.compiler` module) can be enabled.

2.6. IDE Integration

There are optional MapStruct plugins for IntelliJ and Eclipse that allow you to have additional completion support (and more) in the annotations.

2.6.1. IntelliJ

The MapStruct IntelliJ plugin offers assistance in projects that use MapStruct.

Some features include:

- Code completion in `target`, `source`, `expression`
- Go To Declaration for properties in `target` and `source`
- Find Usages of properties in `target` and `source`
- Refactoring support
- Errors and Quick Fixes

2.6.2. Eclipse

The MapStruct Eclipse Plugin offers assistance in projects that use MapStruct.

Some features include:

- Code completion in `target` and `source`
- Quick Fixes

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 6. Java interface to define a mapper

```java
@Mapper
public interface CarMapper {

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

    @Mapping(target = "fullName", source = "name")
    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 into the corresponding property in the target type (e.g. `CarDto`):

- When a property has the same name as its target entity counterpart, it will be mapped implicitly.

- When 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 specification must be specified in the `@Mapping` annotation, e.g. *seatCount* for a property with the accessor methods `getSeatCount()` and `setSeatCount()`.

> By means of the `@BeanMapping(ignoreByDefault = true)` the default behavior will be **explicit mapping**, meaning that all mappings (including nested ones) have to be specified by means of the `@Mapping` and no warnings will be issued on missing target properties. This allows to ignore all fields, except the ones that are explicitly defined through `@Mapping`.

> Fluent setters are also supported. Fluent setters are setters that return the same type as the type being modified.
>
> E.g.
>
> ```java
> public Builder seatCount(int seatCount) {
>     this.seatCount = seatCount;
>     return this;
> }
> ```

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 7. Code generated by MapStruct

```java
// 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() );
        }
        carDto.setEngine( engineToEngineDto( car.getEngine() ) );

        return carDto;
    }

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

    private EngineDto engineToEngineDto(Engine engine) {
        if ( engine == null ) {
            return null;
        }

        EngineDto engineDto = new EngineDto();

        engineDto.setHorsePower(engine.getHorsePower());
        engineDto.setFuel(engine.getFuel());

        return engineDto;
    }
}
```

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 / create another mapping method (as e.g. for the *driver* / *engine* property, see also Mapping object references). MapStruct will only create a new mapping method if and only if the source and target property are properties of a Bean and they themselves are Beans or simple properties. i.e. they are not `Collection` or `Map` type properties.

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 be 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. Mapping Composition

MapStruct supports the use of meta annotations. The `@Mapping` annotation supports now `@Target` with `ElementType#ANNOTATION_TYPE` in addition to `ElementType#METHOD`. This allows `@Mapping` to be used on other (user defined) annotations for re-use purposes. For example:

```java
@Retention(RetentionPolicy.CLASS)
@Mapping(target = "id", ignore = true)
@Mapping(target = "creationDate", expression = "java(new java.util.Date())")
@Mapping(target = "name", source = "groupName")
public @interface ToEntity { }
```

Can be used to characterise an `Entity` without the need to have a common base type. For instance, `ShelveEntity` and `BoxEntity` do not share a common base type in the `StorageMapper` below.

```java
@Mapper
public interface StorageMapper {

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

    @ToEntity
    @Mapping( target = "weightLimit", source = "maxWeight")
    ShelveEntity map(ShelveDto source);

    @ToEntity
    @Mapping( target = "label", source = "designation")
    BoxEntity map(BoxDto source);
}
```

Still, they do have some properties in common. The `@ToEntity` assumes both target beans `ShelveEntity` and `BoxEntity` have properties: `"id"`, `"creationDate"` and `"name"`. It furthermore assumes that the source beans `ShelveDto` and `BoxDto` always have a property `"groupName"`. This concept is also known as "duck-typing". In other words, if it quacks like duck, walks like a duck its probably a duck.

Error messages are not mature yet: the method on which the problem occurs is displayed, as well as the concerned values in the `@Mapping` annotation. However, the composition aspect is not visible. The messages are "as if" the `@Mapping` would be present on the concerned method directly. Therefore, the user should use this feature with care, especially when uncertain when a property is always present.

A more typesafe (but also more verbose) way would be to define base classes / interfaces on the target bean and the source bean and use `@InheritConfiguration` to achieve the same result (see Mapping configuration inheritance).

3.3. Adding custom methods to mappers

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 to handle this is to implement the custom method on another class which then is used by mappers generated by MapStruct (see Invoking other mappers).

Alternatively, when using Java 8 or later, you can implement custom methods directly in a mapper interface as default methods. The generated code will invoke the default methods if the argument and return types match.

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 8. Mapper which defines a custom mapping with a default method

```java
@Mapper
public interface CarMapper {

    @Mapping(...)
    ...
    CarDto carToCarDto(Car car);

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

The class generated by MapStruct implements the method `carToCarDto()`. The generated code in `carToCarDto()` will invoke the manually implemented `personToPersonDto()` method when mapping the `driver` attribute.

A mapper could also be defined in the form of an abstract class instead of an interface and implement the custom methods directly in the mapper class. In this case MapStruct will generate an extension of the abstract class with implementations of all abstract methods. An advantage of this approach over declaring default methods is that additional fields could be declared in the mapper class.

The previous example where the mapping from `Person` to `PersonDto` requires some special logic could then be defined like this:

Example 9. Mapper defined by an abstract class

```java
@Mapper
public abstract class CarMapper {

    @Mapping(...)
    ...
    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.

3.4. 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 10. Mapping method with several source parameters

```java
@Mapper
public interface AddressMapper {

    @Mapping(target = "description", source = "person.description")
    @Mapping(target = "houseNumber", source = "address.houseNo")
    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.

MapStruct also offers the possibility to directly refer to a source parameter.

Example 11. Mapping method directly referring to a source parameter

```java
@Mapper
public interface AddressMapper {

    @Mapping(target = "description", source = "person.description")
    @Mapping(target = "houseNumber", source = "hn")
    DeliveryAddressDto personAndAddressToDeliveryAddressDto(Person person, Integer hn);
}
```

In this case the source parameter is directly mapped into the target as the example above demonstrates. The parameter `hn`, a non bean type (in this case `java.lang.Integer`) is mapped to `houseNumber`.

3.5. Mapping nested bean properties to current target

If you don't want explicitly name all properties from nested source bean, you can use `.` as target. This will tell MapStruct to map every property from source bean to target object. The following shows an example:

Example 12. use of "target this" annotation "."

```java
@Mapper
public interface CustomerMapper {

    @Mapping( target = "name", source = "record.name" )
    @Mapping( target = ".", source = "record" )
    @Mapping( target = ".", source = "account" )
    Customer customerDtoToCustomer(CustomerDto customerDto);
}
```

The generated code will map every property from `CustomerDto.record` to `Customer` directly, without need to manually name any of them. The same goes for `Customer.account`.

When there are conflicts, these can be resolved by explicitly defining the mapping. For instance in the example above. `name` occurs in `CustomerDto.record` and in `CustomerDto.account`. The mapping `@Mapping( target = "name", source = "record.name" )` resolves this conflict.

This "target this" notation can be very useful when mapping hierarchical objects to flat objects and vice versa (`@InheritInverseConfiguration`).

3.6. 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 13. Update method

```java
@Mapper
public interface CarMapper {

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

The generated code of the `updateCarFromDto()` method will update 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.

For `CollectionMappingStrategy.ACCESSOR_ONLY` 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. Otherwise, For `CollectionMappingStrategy.ADDER_PREFERRED` or `CollectionMappingStrategy.TARGET_IMMUTABLE` the target will not be cleared and the values will be populated immediately.

3.7. Mappings with direct field access

MapStruct also supports mappings of `public` fields that have no getters/setters. MapStruct will use the fields as read/write accessor if it cannot find suitable getter/setter methods for the property.

A field is considered as a read accessor if it is `public` or `public final`. If a field is `static` it is not considered as a read accessor.

A field is considered as a write accessor only if it is `public`. If a field is `final` and/or `static` it is not considered as a write accessor.

Small example:

Example 14. Example classes for mapping

```java
public class Customer {

    private Long id;
    private String name;

    //getters and setter omitted for brevity
}

public class CustomerDto {

    public Long id;
    public String customerName;
}

@Mapper
public interface CustomerMapper {

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

    @Mapping(target = "name", source = "customerName")
    Customer toCustomer(CustomerDto customerDto);

    @InheritInverseConfiguration
    CustomerDto fromCustomer(Customer customer);
}
```

For the configuration from above, the generated mapper looks like:

Example 15. Generated mapper for example classes

```java
// GENERATED CODE
public class CustomerMapperImpl implements CustomerMapper {

    @Override
    public Customer toCustomer(CustomerDto customerDto) {
        // ...
        customer.setId( customerDto.id );
        customer.setName( customerDto.customerName );
        // ...
    }

    @Override
    public CustomerDto fromCustomer(Customer customer) {
        // ...
        customerDto.id = customer.getId();
        customerDto.customerName = customer.getName();
        // ...
    }
}
```

You can find the complete example in the mapstruct-examples-field-mapping project on GitHub.

3.8. Using builders

MapStruct also supports mapping of immutable types via builders. When performing a mapping MapStruct checks if there is a builder for the type being mapped. This is done via the `BuilderProvider` SPI. If a Builder exists for a certain type, then that builder will be used for the mappings.

The default implementation of the `BuilderProvider` assumes the following:

- The type has a parameterless public static builder creation method that returns a builder. So for example `Person` has a public static method that returns `PersonBuilder`.

- The builder type has a parameterless public method (build method) that returns the type being built. In our example `PersonBuilder` has a method returning `Person`.

- In case there are multiple build methods, MapStruct will look for a method called `build`, if such method exists then this would be used, otherwise a compilation error would be created.

- A specific build method can be defined by using `@Builder` within: `@BeanMapping`, `@Mapper` or `@MapperConfig`

- In case there are multiple builder creation methods that satisfy the above conditions then a `MoreThanOneBuilderCreationMethodException` will be thrown from the `DefaultBuilderProvider` SPI. In case of a `MoreThanOneBuilderCreationMethodException` MapStruct will write a warning in the compilation and not use any builder.

If such type is found then MapStruct will use that type to perform the mapping to (i.e. it will look for setters into that type). To finish the mapping MapStruct generates code that will invoke the build method of the builder.

> ℹ Builder detection can be switched off by means of `@Builder#disableBuilder`. MapStruct will fall back on regular getters / setters in case builders are disabled.

> ℹ The Object factories are also considered for the builder type. E.g. If an object factory exists for our `PersonBuilder` then this factory would be used instead of the builder creation method.

> ℹ Detected builders influence `@BeforeMapping` and `@AfterMapping` behavior. See Mapping customization with before-mapping and after-mapping methods for more information.

Example 16. Person with Builder example

```java
public class Person {

    private final String name;

    protected Person(Person.Builder builder) {
        this.name = builder.name;
    }

    public static Person.Builder builder() {
        return new Person.Builder();
    }

    public static class Builder {

        private String name;

        public Builder name(String name) {
            this.name = name;
            return this;
        }

        public Person create() {
            return new Person( this );
        }
    }
}
```

Example 17. Person Mapper definition

```java
public interface PersonMapper {

    Person map(PersonDto dto);
}
```

Example 18. Generated mapper with builder

```java
// GENERATED CODE
public class PersonMapperImpl implements PersonMapper {

    public Person map(PersonDto dto) {
        if (dto == null) {
            return null;
        }

        Person.Builder builder = Person.builder();

        builder.name( dto.getName() );

        return builder.create();
    }
}
```

Supported builder frameworks:

- Lombok - It is required to have the Lombok classes in a separate module. See for more information at rzwitserloot/lombok#1538 and to set up Lombok with MapStruct, refer to Lombok.

- AutoValue

- Immutables - When Immutables are present on the annotation processor path then the `ImmutablesAccessorNamingStrategy` and `ImmutablesBuilderProvider` would be used by default

- FreeBuilder - When FreeBuilder is present on the annotation processor path then the `FreeBuilderAccessorNamingStrategy` would be used by default. When using FreeBuilder then the JavaBean convention should be followed, otherwise MapStruct won't recognize the fluent getters.

- It also works for custom builders (handwritten ones) if the implementation supports the defined rules for the default `BuilderProvider`. Otherwise, you would need to write a custom `BuilderProvider`

> 💡 In case you want to disable using builders then you can pass the MapStruct processor option `mapstruct.disableBuilders` to the compiler. e.g. `-Amapstruct.disableBuilders=true`.

3.9. Using Constructors

MapStruct supports using constructors for mapping target types. When doing a mapping MapStruct checks if there is a builder for the type being mapped. If there is no builder, then MapStruct looks for a single accessible constructor. When there are multiple constructors then the following is done to pick the one which should be used:

- If a constructor is annotated with an annotation *named* `@Default` (from any package, see Non-shipped annotations) it will be used.

- If a single public constructor exists then it will be used to construct the object, and the other non public constructors will be ignored.

- If a parameterless constructor exists then it will be used to construct the object, and the other constructors will be ignored.

- If there are multiple eligible constructors then there will be a compilation error due to ambiguous constructors. In order to break the ambiguity an annotation *named* `@Default` (from any package, see Non-shipped annotations) can used.

Example 19. Deciding which constructor to use

```java
public class Vehicle {

    protected Vehicle() { }

    // MapStruct will use this constructor, because it is a single public
constructor
    public Vehicle(String color) { }
}

public class Car {

    // MapStruct will use this constructor, because it is a parameterless empty
constructor
    public Car() { }

    public Car(String make, String color) { }
}

public class Truck {

    public Truck() { }

    // MapStruct will use this constructor, because it is annotated with @Default
    @Default
    public Truck(String make, String color) { }
}

public class Van {

    // There will be a compilation error when using this class because MapStruct
cannot pick a constructor

    public Van(String make) { }

    public Van(String make, String color) { }

}
```

When using a constructor then the names of the parameters of the constructor will be used and matched to the target properties. When the constructor has an annotation *named* `@ConstructorProperties` (from any package, see Non-shipped annotations) then this annotation will be used to get the names of the parameters.

> When an object factory method or a method annotated with `@ObjectFactory` exists, it will take precedence over any constructor defined in the target. The target object constructor will not be used in that case.

Example 20. Person with constructor parameters

```java
public class Person {

    private final String name;
    private final String surname;

    public Person(String name, String surname) {
        this.name = name;
        this.surname = surname;
    }
}
```

Example 21. Person With Constructor Mapper definition

```java
public interface PersonMapper {

    Person map(PersonDto dto);
}
```

Example 22. Generated mapper with constructor

```java
// GENERATED CODE
public class PersonMapperImpl implements PersonMapper {

    public Person map(PersonDto dto) {
        if (dto == null) {
            return null;
        }

        String name;
        String surname;
        name = dto.getName();
        surname = dto.getSurname();

        Person person = new Person( name, surname );

        return person;
    }
}
```

3.10. Mapping Map to Bean

There are situations when a mapping from a `Map<String, ???>` into a specific bean is needed. MapStruct offers a transparent way of doing such a mapping by using the target bean properties (or defined through `Mapping#source`) to extract the values from the map. Such a mapping looks like:

Example 23. Example classes for mapping map to bean

```java
public class Customer {

    private Long id;
    private String name;

    //getters and setter omitted for brevity
}


@Mapper
public interface CustomerMapper {

    @Mapping(target = "name", source = "customerName")
    Customer toCustomer(Map<String, String> map);

}
```

Example 24. Generated mapper for mapping map to bean

```java
// GENERATED CODE
public class CustomerMapperImpl implements CustomerMapper {

    @Override
    public Customer toCustomer(Map<String, String> map) {
        // ...
        if ( map.containsKey( "id" ) ) {
            customer.setId( Integer.parseInt( map.get( "id" ) ) );
        }
        if ( map.containsKey( "customerName" ) ) {
            customer.setName( map.get( "customerName" ) );
        }
        // ...
    }
}
```

All existing rules about mapping between different types and using other mappers defined with `Mapper#uses` or custom methods in the mappers are applied. i.e. You can map from `Map<String, Integer>` where for each property a conversion from `Integer` into the respective property will be needed.

When a raw map or a map that does not have a String as a key is used, then a warning will be generated. The warning is not generated if the map itself is mapped into some other target property directly as is.

3.11. Adding annotations

Other frameworks sometimes requires you to add annotations to certain classes so that they can easily detect the mappers. Using the `@AnnotateWith` annotation you can generate an annotation at the specified location.

For example Apache Camel has a `@Converter` annotation which you can apply to generated mappers using the `@AnnotateWith` annotation.

Example 25. AnnotateWith source example

```java
@Mapper
@AnnotateWith(
  value = Converter.class,
  elements = @AnnotateWith.Element( name = "generateBulkLoader", booleans = true )
)
public interface MyConverter {
    @AnnotateWith( Converter.class )
    DomainObject map( DtoObject dto );
}
```

Example 26. AnnotateWith generated mapper

```java
@Converter( generateBulkLoader = true )
public class MyConverterImpl implements MyConverter {
    @Converter
    public DomainObject map( DtoObject dto ) {
        // default mapping behaviour
    }
}
```

3.12. Adding Javadoc comments

MapStruct provides support for defining Javadoc comments in the generated mapper implementation using the `org.mapstruct.Javadoc` annotation.

This functionality could be relevant especially in situations where certain Javadoc standards need to be met or to deal with Javadoc validation constraints.

The `@Javadoc` annotation defines attributes for the different Javadoc elements.

Consider the following example:

Example 27. Javadoc annotation example

```java
@Mapper
@Javadoc(
    value = "This is the description",
    authors = { "author1", "author2" },
    deprecated = "Use {@link OtherMapper} instead",
    since = "0.1"
)
public interface MyAnnotatedWithJavadocMapper {
    //...
}
```

Example 28. Javadoc annotated generated mapper

```java
/**
 * This is the description
 *
 * @author author1
 * @author author2
 *
 * @deprecated Use {@link OtherMapper} instead
 * @since 0.1
 */
public class MyAnnotatedWithJavadocMapperImpl implements
MyAnnotatedWithJavadocMapper {
    //...
}
```

The entire Javadoc comment block can be provided directly as well.

Example 29. Javadoc annotation example with the entire Javadoc comment block provided directly

```java
@Mapper
@Javadoc(
    "This is the description\n"
  + "\n"
  + "@author author1\n"
  + "@author author2\n"
  + "\n"
  + "@deprecated Use {@link OtherMapper} instead\n"
  + "@since 0.1\n"
)
public interface MyAnnotatedWithJavadocMapper {
    //...
}
```

Or using Text blocks:

Example 30. Javadoc annotation example with the entire Javadoc comment block provided directly using Text blocks

```java
@Mapper
@Javadoc(
    """
    This is the description

    @author author1
    @author author2

    @deprecated Use {@link OtherMapper} instead
    @since 0.1
    """
)
public interface MyAnnotatedWithJavadocMapper {
    //...
}
```

4. Retrieving a mapper

4.1. The Mappers factory (no dependency injection)

When not using a DI framework, 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 31. 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 32. Declaring an instance of a mapper (interface)

```java
@Mapper
public interface CarMapper {

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

    CarDto carToCarDto(Car car);
}
```

Example 33. Declaring an instance of a mapper (abstract class)

```java
@Mapper
public abstract class CarMapper {

    public static final 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 34. Accessing a mapper

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

Note that mappers generated by MapStruct are stateless and 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 and **not** via the `Mappers` class as described above. 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 latter 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 and constants are defined in a class `MappingConstants.ComponentModel`. 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 35. A mapper using the CDI component model

```java
@Mapper(componentModel = MappingConstants.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 36. Obtaining a mapper via dependency injection

```java
@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.

4.3. Injection strategy

When using dependency injection, you can choose between constructor, field, or setter injection. This can be done by either providing the injection strategy via `@Mapper` or `@MapperConfig` annotation.

Example 37. Using constructor injection

```java
@Mapper(componentModel = MappingConstants.ComponentModel.CDI, uses = EngineMapper
.class, injectionStrategy = InjectionStrategy.CONSTRUCTOR)
public interface CarMapper {
    CarDto carToCarDto(Car car);
}
```

The generated mapper will inject classes defined in the **uses** attribute if MapStruct has detected that it needs to use an instance of it for a mapping. When `InjectionStrategy#CONSTRUCTOR` is used, the constructor will have the appropriate annotation and the fields won't. When `InjectionStrategy#FIELD` is used, the annotation is on the field itself. When `InjectionStrategy#SETTER` is used the annotation is on a generated setter method. For now, the default injection strategy is field injection, but it can be configured with Configuration options. It is recommended to use constructor injection to simplify testing.

When you define mappers in Spring with circular dependencies compilation may fail. In that case utilize the `InjectionStrategy#SETTER` strategy.

> For abstract classes or decorators setter injection should be used.

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 be 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. The `Mapper` and `MapperConfig` annotations have a method `typeConversionPolicy` to control warnings / errors. Due to backward compatibility reasons the default value is `ReportingPolicy.IGNORE`.

- Between all Java primitive types (including their wrappers) and `String`, e.g. between `int` and `String` or `Boolean` and `String`. A format string as understood by `java.text.DecimalFormat` can be specified.

Example 38. Conversion from int to String

```java
@Mapper
public interface CarMapper {

    @Mapping(source = "price", numberFormat = "$#.00")
    CarDto carToCarDto(Car car);

    @IterableMapping(numberFormat = "$#.00")
    List<String> prices(List<Integer> prices);
}
```

- Between `enum` types and `String`.
- Between `enum` types and `Integer`, according to `enum.ordinal()`.
  - When converting from an `Integer`, the value needs to be less than the number of values of the enum, otherwise an `ArrayOutOfBoundsException` is thrown.
- Between big number types (`java.math.BigInteger`, `java.math.BigDecimal`) and Java primitive types (including their wrappers) as well as String. A format string as understood by `java.text.DecimalFormat` can be specified.

Example 39. Conversion from BigDecimal to String

```java
@Mapper
public interface CarMapper {

    @Mapping(source = "power", numberFormat = "#.##E0")
    CarDto carToCarDto(Car car);

}
```

- 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 40. Conversion from Date to String

```java
@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 `javax.xml.datatype.XMLGregorianCalendar`, `java.util.Calendar`.

- Between Jodas `org.joda.time.LocalDateTime`, `org.joda.time.LocalDate` and `javax.xml.datatype.XMLGregorianCalendar`, `java.util.Date`.

- Between `java.time.LocalDate`, `java.time.LocalDateTime` and `javax.xml.datatype.XMLGregorianCalendar`.

- 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.Instant`, `java.time.Duration`, `java.time.Period` from Java 8 Date-Time package and `String` using the `parse` method in each class to map from `String` and using `toString` to map into `String`.

- Between `java.time.ZonedDateTime` from Java 8 Date-Time package and `java.util.Date` where, when mapping a `ZonedDateTime` from a given `Date`, the system default timezone is used.

- Between `java.time.LocalDateTime` from Java 8 Date-Time package and `java.util.Date` where timezone UTC is used as the timezone.

- Between `java.time.LocalDate` from Java 8 Date-Time package and `java.util.Date` / `java.sql.Date` where timezone UTC is used as the timezone.

- Between `java.time.Instant` from Java 8 Date-Time package and `java.util.Date`.

- Between `java.time.LocalDateTime` from Java 8 Date-Time package and `java.time.LocalDate` from the same package.

- Between `java.time.ZonedDateTime` from Java 8 Date-Time package and `java.util.Calendar`.

- Between `java.sql.Date` and `java.util.Date`

- Between `java.sql.Time` and `java.util.Date`

- Between `java.sql.Timestamp` and `java.util.Date`

- When converting from a `String`, omitting `Mapping#dateFormat`, it leads to usage of 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](#).

- Between `java.util.Currency` and `String`.

  - When converting from a `String`, the value needs to be a valid [ISO-4217](#) alphabetic code otherwise an `IllegalArgumentException` is thrown.

- Between `java.util.UUID` and `String`.

  - When converting from a `String`, the value needs to be a valid [UUID](#) otherwise an `IllegalArgumentException` is thrown.

- Between `String` and `StringBuilder`

- Between `java.net.URL` and `String`.

  - When converting from a `String`, the value needs to be a valid [URL](#) otherwise a `MalformedURLException` is thrown.

- Between `java.util.Locale` and `String`.

  - When converting from a `Locale`, the resulting `String` will be a well-formed IETF BCP 47 language tag representing the locale. When converting from a `String`, the locale that best represents the language tag will be returned. See [Locale.forLanguageTag()](#) and [Locale.toLanguageTag()](#) for more information.

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 41. 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:

1. If source and target attribute have the same type, the value will be simply copied **direct** 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.

2. If source and target attribute type differ, check whether there is 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.

3. 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.

4. If no such method exists MapStruct will apply **complex** conversions:

   a. mapping method, the result mapped by mapping method, like this: `target = method1( method2( source ) )`

   b. built-in conversion, the result mapped by mapping method, like this: `target = method( conversion( source ) )`

   c. mapping method, the result mapped by build-in conversion, like this: `target = conversion( method( source ) )`

5. If no such method was found MapStruct will try to generate an automatic sub-mapping method that will do the mapping between the source and target attributes.

6. If MapStruct could not create a name based mapping method an error will be raised at build time, indicating the non-mappable attribute and its path.

A mapping control (`MappingControl`) can be defined on all levels (`@MapperConfig`, `@Mapper`, `@BeanMapping`, `@Mapping`), the latter taking precedence over the former. For example: `@Mapper( mappingControl = NoComplexMapping.class )` takes precedence over `@MapperConfig( mappingControl = DeepClone.class )`. `@IterableMapping` and `@MapMapping` work similar as `@Mapping`. MappingControl is experimental from MapStruct 1.4. `MappingControl` has an enum that corresponds to the first 4 options above: `MappingControl.Use#DIRECT`, `MappingControl.Use#MAPPING_METHOD`, `MappingControl.Use#BUILT_IN_CONVERSION` and `MappingControl.Use#COMPLEX_MAPPING` the presence of which allows the user to switch **on** a option. The absence of an enum switches **off** a mapping option. Default they are all present enabling all mapping options.

> In order to stop MapStruct from generating automatic sub-mapping methods as in 5. above, one can use `@Mapper( disableSubMappingMethodsGeneration = true )`.

> The user has full control over the mapping by means of meta annotations. Some handy ones have been defined such as `@DeepClone` which only allows direct mappings. The result: if source and target type are the same, MapStruct will make a deep clone of the source. Sub-mappings-methods have to be allowed (default option).

During the generation of automatic sub-mapping methods Shared configurations will not be taken into consideration, yet. Follow issue #1086 for more information.

Constructor properties of the target object are also considered as target properties. You can read more about that in Using Constructors

5.3. Controlling nested bean mappings

As explained above, MapStruct will generate a method based on the name of the source and target property. Unfortunately, in many occasions these names do not match.

The '.' notation in an `@Mapping` source or target type can be used to control how properties should be mapped when names do not match. There is an elaborate example in our examples repository to explain how this problem can be overcome.

In the simplest scenario there's a property `fish` which has an identical name in `FishTankDto` and `FishTank`. For this property MapStruct automatically generates a mapping: `FishDto fishToFishDto(Fish fish)`. MapStruct cannot possibly be aware of the deviating properties `kind` and `type`. Therefore this can be addressed in a mapping rule: `@Mapping(target="fish.kind", source="fish.type")`. This tells MapStruct to deviate from looking for a name `kind` at this level and map it to `type`.

Example 42. Mapper controlling nested beans mappings I

```java
@Mapper
public interface FishTankMapper {

    @Mapping(target = "fish.kind", source = "fish.type")
    @Mapping(target = "fish.name", ignore = true)
    @Mapping(target = "ornament", source = "interior.ornament")
    @Mapping(target = "material.materialType", source = "material")
    @Mapping(target = "quality.report.organisation.name", source =
"quality.report.organisationName")
    FishTankDto map( FishTank source );
}
```

The same constructs can be used to ignore certain properties at a nesting level, as is demonstrated in the second `@Mapping` rule.

MapStruct can even be used to "cherry pick" properties when source and target do not share the same nesting level (the same number of properties). This can be done in the source – and in the target type. This is demonstrated in the next 2 rules: `@Mapping(target="ornament", source="interior.ornament")` and `@Mapping(target="material.materialType", source="material")`.

The latter can even be done when mappings first share a common base. For example: all properties that share the same name of `Quality` are mapped to `QualityDto`. Likewise, all properties of `Report` are mapped to `ReportDto`, with one exception: `organisation` in `OrganisationDto` is left empty (since

there is no organization at the source level). Only the `name` is populated with the `organisationName` from `Report`. This is demonstrated in `@Mapping(target="quality.report.organisation.name", source="quality.report.organisationName")`

Coming back to the original example: what if `kind` and `type` would be beans themselves? In that case MapStruct would again generate a method continuing to map. Such is demonstrated in the next example:

Example 43. Mapper controlling nested beans mappings II

```java
@Mapper
public interface FishTankMapperWithDocument {

    @Mapping(target = "fish.kind", source = "fish.type")
    @Mapping(target = "fish.name", expression = "java(\"Jaws\")")
    @Mapping(target = "plant", ignore = true )
    @Mapping(target = "ornament", ignore = true )
    @Mapping(target = "material", ignore = true)
    @Mapping(target = "quality.document", source = "quality.report")
    @Mapping(target = "quality.document.organisation.name", constant = "NoIdeaInc"
    )
    FishTankWithNestedDocumentDto map( FishTank source );

}
```

Note what happens in `@Mapping(target="quality.document", source="quality.report")`. `DocumentDto` does not exist as such on the target side. It is mapped from `Report`. MapStruct continues to generate mapping code here. That mapping itself can be guided towards another name. This even works for constants and expression. Which is shown in the final example: `@Mapping(target="quality.document.organisation.name", constant="NoIdeaInc")`.

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

> 💡 Instead of configuring everything via the parent method we encourage users to explicitly write their own nested methods. This puts the configuration of the nested mapping into one place (method) where it can be reused from several methods in the upper level, instead of re-configuring the same things on all of those upper methods.

> ℹ️ In some cases the `ReportingPolicy` that is going to be used for the generated nested method would be `IGNORE`. This means that it is possible for MapStruct not to report unmapped target properties in nested mappings.

5.4. Invoking custom mapping method

Sometimes mappings are not straightforward and some fields require custom logic.

The example below demonstrates how the properties `length`, `width` and `height` in `FishTank` can be mapped to the `VolumeDto` bean, which is a member of `FishTankWithVolumeDto`. `VolumeDto` contains the properties `volume` and `description`. Custom logic is achieved by defining a method which takes `FishTank` instance as a parameter and returns a `VolumeDto`. MapStruct will take the entire parameter `source` and generate code to call the custom method `mapVolume` in order to map the `FishTank` object to the target property `volume`.

The remainder of the fields could be mapped the regular way: using mappings defined defined by means of `@Mapping` annotations.

Example 44. Manually implemented mapping method

```java
public class FishTank {
    Fish fish;
    String material;
    Quality quality;
    int length;
    int width;
    int height;
}

public class FishTankWithVolumeDto {
    FishDto fish;
    MaterialDto material;
    QualityDto quality;
    VolumeDto volume;
}

public class VolumeDto {
    int volume;
    String description;
}

@Mapper
public abstract class FishTankMapperWithVolume {

    @Mapping(target = "fish.kind", source = "source.fish.type")
    @Mapping(target = "material.materialType", source = "source.material")
    @Mapping(target = "quality.document", source = "source.quality.report")
    @Mapping(target = "volume", source = "source")
    abstract FishTankWithVolumeDto map(FishTank source);

    VolumeDto mapVolume(FishTank source) {
        int volume = source.length * source.width * source.height;
        String desc = volume < 100 ? "Small" : "Large";
        return new VolumeDto(volume, desc);
    }
}
```

Note the `@Mapping` annotation where `source` field is equal to `"source"`, indicating the parameter name `source` itself in the method `map(FishTank source)` instead of a (target) property in `FishTank`.

5.5. 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 one mapper type per application module) or if 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 45. Manually implemented mapper class

```java
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 46. Referencing another mapper class

```java
@Mapper(uses=DateMapper.class)
public interface 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.

5.6. 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.

e.g.

Example 47. Example classes for the passing target type example

```java
public class Car {

    private Person owner;
    // ...
}

public class Person extends BaseEntity {

    // ...
}

public class Reference {

    private String pk;
    // ...
}

public class CarDto {

    private Reference owner;
    // ...
}
```

Example 48. Mapping method expecting mapping target type as parameter

```java
@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 = MappingConstants.ComponentModel.CDI, uses =
ReferenceMapper.class )
public interface CarMapper {

    Car carDtoToCar(CarDto carDto);
}
```

MapStruct will then generate something like this:

Example 49. Generated code

```java
//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;
    }
}
```

5.7. Passing context or state objects to custom methods

Additional *context* or *state* information can be passed through generated mapping methods to custom methods with @Context parameters. Such parameters are passed to other mapping methods, @ObjectFactory methods (see Object factories) or @BeforeMapping / @AfterMapping methods (see Mapping customization with before-mapping and after-mapping methods) when applicable and can thus be used in custom code.

@Context parameters are searched for @ObjectFactory methods, which are called on the provided context parameter value if applicable.

@Context parameters are also searched for @BeforeMapping / @AfterMapping methods, which are called on the provided context parameter value if applicable.

Note: no null checks are performed before calling before/after mapping methods on context parameters. The caller needs to make sure that null is not passed in that case.

For generated code to call a method that is declared with @Context parameters, the declaration of the mapping method being generated needs to contain at least those (or assignable) @Context parameters as well. The generated code will not create new instances of missing @Context parameters nor will it pass a literal null instead.

Example 50. Using @Context parameters for passing data down to hand-written property mapping methods

```java
public abstract CarDto toCar(Car car, @Context Locale translationLocale);

protected OwnerManualDto translateOwnerManual(OwnerManual ownerManual, @Context
Locale locale) {
    // manually implemented logic to translate the OwnerManual with the given
Locale
}
```

MapStruct will then generate something like this:

Example 51. Generated code

```java
//GENERATED CODE
public CarDto toCar(Car car, Locale translationLocale) {
    if ( car == null ) {
        return null;
    }

    CarDto carDto = new CarDto();

    carDto.setOwnerManual( translateOwnerManual( car.getOwnerManual(),
translationLocale );
    // more generated mapping code

    return carDto;
}
```

5.8. 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.

5.9. Mapping method selection based on qualifiers

In many occasions one requires mapping methods with the same method signature (apart from the name) that have different behavior. MapStruct has a handy mechanism to deal with such situations: `@Qualifier` (`org.mapstruct.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, let's 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 52. Several mapping methods with identical source and target types

```java
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 53. Mapper causing an ambiguous mapping method error

```java
@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 54. Declaring a qualifier type

```java
import org.mapstruct.Qualifier;

@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 55. Declaring qualifier types for mapping methods

```java
import org.mapstruct.Qualifier;

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

```java
import org.mapstruct.Qualifier;

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

Please take note of the target `TitleTranslator` on type level, `EnglishToGerman`, `GermanToEnglish` on method level!

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

Example 56. Mapper using qualifiers

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

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

}
```

Example 57. Custom mapper qualifying the methods it provides

```java
@TitleTranslator
public class Titles {

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

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

> ⚠ Please make sure the used retention policy equals retention policy `CLASS` (`@Retention(RetentionPolicy.CLASS)`).

> ⚠ 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 58. Custom mapper, annotating the methods to qualify by means of `@Named`

```java
@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 59. Mapper using named

```java
@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.

5.10. Combining qualifiers with defaults

Please note that the `Mapping#defaultValue` is in essence a `String`, which needs to be converted to the `Mapping#target`. Providing a `Mapping#qualifiedByName` or `Mapping#qualifiedBy` will force MapStruct to use that method. If you want different behavior for the `Mapping#defaultValue`, then please provide an appropriate mapping method. This mapping method needs to transforms a `String` into the desired type of `Mapping#target` and also be annotated so that it can be found by the `Mapping#qualifiedByName` or `Mapping#qualifiedBy`.

Example 60. Mapper using defaultValue

```java
@Mapper
public interface MovieMapper {

    @Mapping( target = "category", qualifiedByName = "CategoryToString",
defaultValue = "DEFAULT" )
    GermanRelease toGerman( OriginalRelease movies );

    @Named("CategoryToString")
    default String defaultValueForQualifier(Category cat) {
        // some mapping logic
    }
}
```

In the above example in case that category is null, the method `CategoryToString( Enum.valueOf( Category.class, "DEFAULT" ) )` will be called and the result will be set to the category field.

Example 61. Mapper using defaultValue and default method.

```java
@Mapper
public interface MovieMapper {

    @Mapping( target = "category", qualifiedByName = "CategoryToString",
defaultValue = "Unknown" )
    GermanRelease toGerman( OriginalRelease movies );

    @Named("CategoryToString")
    default String defaultValueForQualifier(Category cat) {
        // some mapping logic
    }

    @Named("CategoryToString")
    default String defaultValueForQualifier(String value) {
        return value;
    }
}
```

In the above example in case that category is null, the method `defaultValueForQualifier( "Unknown" )` will be called and the result will be set to the category field.

If the above mentioned methods do not work there is the option to use `defaultExpression` to set the default value.

Example 62. Mapper using defaultExpression

```java
@Mapper
public interface MovieMapper {

    @Mapping( target = "category", qualifiedByName = "CategoryToString",
defaultExpression = "java(\"Unknown\")" )
    GermanRelease toGerman( OriginalRelease movies );

    @Named("CategoryToString")
    default String defaultValueForQualifier(Category cat) {
        // some mapping logic
    }
}
```

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 63. Mapper with collection mapping methods

```java
@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:

Example 64. Generated collection mapping methods

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

    Set<String> set = new LinkedHashSet<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 65. Usage of collection mapping method to map a bean property

```java
//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 66. 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 (from a java package) 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 67. 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 68. Generated implementation of map mapping method

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

    Map<Long, Date> map = new LinkedHashMap<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`, `ADDER_PREFERRED` and `TARGET_IMMUTABLE`.

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 applied to the presence/absence 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(`@Target Type`)
`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
`TARGET_IMMUTABLE`	set-s	exception	set-s	exception	set-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 explicitly. 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 instead 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`	`LinkedHashSet`
`SortedSet`	`TreeSet`
`NavigableSet`	`TreeSet`
`Map`	`LinkedHashMap`
`SortedMap`	`TreeMap`
`NavigableMap`	`TreeMap`

Interface type	Implementation type
ConcurrentMap	ConcurrentHashMap
ConcurrentNavigableMap	ConcurrentSkipListMap

7. Mapping Streams

The mapping of `java.util.Stream` is done in a similar way as the mapping of collection types, i.e. by defining mapping methods with the required source and target types in a mapper interface.

The generated code will contain the creation of a `Stream` from the provided `Iterable`/array or will collect the provided `Stream` into an `Iterable`/array. If a mapping method or an implicit conversion for the source and target element types exists, then this conversion will be done in `Stream#map()`. The following shows an example:

Example 69. Mapper with stream mapping methods

```java
@Mapper
public interface CarMapper {

    Set<String> integerStreamToStringSet(Stream<Integer> integers);

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

    CarDto carToCarDto(Car car);
}
```

The generated implementation of the `integerStreamToStringSet()` 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:

Example 70. Generated stream mapping methods

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

    return integers.map( integer -> String.valueOf( integer ) )
        .collect( Collectors.toCollection( LinkedHashSet<String>::new ) );
}

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

    return cars.map( car -> carToCarDto( car ) )
        .collect( Collectors.toCollection( ArrayList<CarDto>::new ) );
}
```

> ⚠️ If a mapping from a `Stream` to an `Iterable` or an array is performed, then the passed `Stream` will be consumed and it will no longer be possible to consume it.

The same implementation types as in Implementation types used for collection mappings are used for the creation of the collection when doing `Stream` to `Iterable` mapping.

8. Mapping Values

8.1. Mapping enum to 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 71. Enum mapping method

```java
@Mapper
public interface OrderMapper {

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

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

Example 72. Enum mapping method result

```java
// 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 and only applies to the source. It comes in two flavors: `<ANY_REMAINING>` and `<ANY_UNMAPPED>`. They cannot be used at the same time.

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_UNMAPPED>` and directly apply the target specified in the `@ValueMapping` with `<ANY_UNMAPPED>` source to the remainder.

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

In addition, the constant value `<THROW_EXCEPTION>` can be used for throwing an exception for particular value mappings. This value is only applicable to `ValueMapping#target()` and not `ValueMapping#source()` since MapStruct can't map from exceptions.

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

Finally `@InheritInverseConfiguration` and `@InheritConfiguration` can be used in combination with `@ValueMappings`. `<ANY_REMAINING>` and `<ANY_UNMAPPED>` will be ignored in that case.

The following code snippets exemplify the use of the aforementioned constants.

Example 73. Enum mapping method, `<NULL>` and `<ANY_REMAINING>`

```java
@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 74. 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>`.

Example 75. Enum mapping method with `<THROW_EXCEPTION>`

```
@Mapper
public interface SpecialOrderMapper {

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

    @ValueMappings({
        @ValueMapping( source = "STANDARD", target = "DEFAULT" ),
        @ValueMapping( source = "C2C", target = MappingConstants.THROW_EXCEPTION )
    })
    ExternalOrderType orderTypeToExternalOrderType(OrderType orderType);
}
```

Example 76. Enum mapping method with `<THROW_EXCEPTION>` *result*

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

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

        ExternalOrderType externalOrderType;

        switch ( orderType ) {
            case STANDARD: externalOrderType = ExternalOrderType.DEFAULT;
            break;
            case C2C: throw new IllegalArgumentException( "Unexpected enum
constant: " + orderType );
            default: throw new IllegalArgumentException( "Unexpected enum
constant: " + orderType );
        }

        return externalOrderType;
    }
}
```

⚠️ 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.

8.2. Mapping enum-to-String or String-to-enum

MapStruct supports enum to a String mapping along the same lines as is described in enum-to-enum types. There are similarities and differences:

enum to `String`

1. Similarity: All not explicit defined mappings will result in each source enum constant value being mapped a `String` value with the same constant value.

2. Similarity: `<ANY_UNMAPPED>` stops after handling defined mapping and proceeds to the switch/default clause value.

3. Difference: `<ANY_REMAINING>` will result in an error. It acts on the premise that there is name similarity between enum constants in source and target which does not make sense for a String type.

4. Difference: Given 1. and 3. there will never be unmapped values.

5. Similarity: `<THROW_EXCEPTION>` can be used for throwing an exception for particular enum values.

`String` to enum

1. Similarity: All not explicit defined mappings will result in the target enum constant mapped from the `String` value when that matches the target enum constant name.

2. Similarity: `<ANY_UNMAPPED>` stops after handling defined mapping and proceeds to the switch/default clause value.

3. Similarity: `<ANY_REMAINING>` will create a mapping for each target enum constant and proceed to the switch/default clause value.

4. Difference: A switch/default value needs to be provided to have a determined outcome (enum has a limited set of values, `String` has unlimited options). Failing to specify `<ANY_REMAINING>` or `<ANY_UNMAPPED>` will result in a warning.

5. Similarity: `<THROW_EXCEPTION>` can be used for throwing an exception for any arbitrary `String` value.

8.3. Custom name transformation

When no `@ValueMapping`(s) are defined then each constant from the source enum is mapped to a constant with the same name in the target enum type. However, there are cases where the source enum needs to be transformed before doing the mapping. E.g. a suffix needs to be applied to map from the source into the target enum.

Example 77. Enum types

```java
public enum CheeseType {

    BRIE,
    ROQUEFORT
}

public enum CheeseTypeSuffixed {

    BRIE_TYPE,
    ROQUEFORT_TYPE
}
```

Example 78. Enum mapping method with custom name transformation strategy

```java
@Mapper
public interface CheeseMapper {

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

    @EnumMapping(nameTransformationStrategy = "suffix", configuration = "_TYPE")
    CheeseTypeSuffixed map(CheeseType cheese);

    @InheritInverseConfiguration
    CheeseType map(CheeseTypeSuffix cheese);
}
```

Example 79. Enum mapping method with custom name transformation strategy result

```java
// GENERATED CODE
public class CheeseSuffixMapperImpl implements CheeseSuffixMapper {

    @Override
    public CheeseTypeSuffixed map(CheeseType cheese) {
        if ( cheese == null ) {
            return null;
        }

        CheeseTypeSuffixed cheeseTypeSuffixed;

        switch ( cheese ) {
            case BRIE: cheeseTypeSuffixed = CheeseTypeSuffixed.BRIE_TYPE;
            break;
            case ROQUEFORT: cheeseTypeSuffixed = CheeseTypeSuffixed.
ROQUEFORT_TYPE;
            break;
            default: throw new IllegalArgumentException( "Unexpected enum
constant: " + cheese );
        }

        return cheeseTypeSuffixed;
    }

    @Override
    public CheeseType map(CheeseTypeSuffixed cheese) {
        if ( cheese == null ) {
            return null;
        }

        CheeseType cheeseType;

        switch ( cheese ) {
            case BRIE_TYPE: cheeseType = CheeseType.BRIE;
            break;
            case ROQUEFORT_TYPE: cheeseType = CheeseType.ROQUEFORT;
            break;
            default: throw new IllegalArgumentException( "Unexpected enum
constant: " + cheese );
        }

        return cheeseType;
    }
}
```

MapStruct provides the following out of the box enum name transformation strategies:

- *suffix* - Applies a suffix on the source enum

- *stripSuffix* - Strips a suffix from the source enum

- *prefix* - Applies a prefix on the source enum

- *stripPrefix* - Strips a prefix from the source enum

- *case* - Applies case transformation to the source enum. Supported *case* transformations are:

  - *upper* - Performs upper case transformation to the source enum

  - *lower* - Performs lower case transformation to the source enum

  - *capital* - Performs capitalisation of the first character of every word in the source enum and everything else to lowercase. A word is split by "_"

It is also possible to register custom strategies. For more information on how to do that have a look at Custom Enum Transformation Strategy

8.4. ValueMapping Composition

The `@ValueMapping` annotation supports now `@Target` with `ElementType#ANNOTATION_TYPE` in addition to `ElementType#METHOD`. This allows `@ValueMapping` to be used on other (user defined) annotations for re-use purposes. For example:

Example 80. Custom value mapping annotations

```
@Retention( RetentionPolicy.CLASS )
@ValueMapping(source = "EXTRA", target = "SPECIAL")
@ValueMapping(source = MappingConstants.ANY_REMAINING, target = "DEFAULT")
public @interface CustomValueAnnotation {
}
```

It can be used to describe some common value mapping relationships to avoid duplicate declarations, as in the following example:

Example 81. Using custom combination annotations

```
@Mapper
public interface ValueMappingCompositionMapper {

    @CustomValueAnnotation
    ExternalOrderType orderTypeToExternalOrderType(OrderType orderType);

    @CustomValueAnnotation
    @ValueMapping(source = "STANDARD", target = "SPECIAL")
    ExternalOrderType duplicateAnnotation(OrderType orderType);
}
```

9. Object factories

By default, the generated code for mapping one bean type into another or updating a bean 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 make use of custom factories register them via `@Mapper#uses()` as described in Invoking other mappers, or implement them directly in your mapper. When creating the target object of a bean mapping, MapStruct will look for a parameterless method, a method annotated with `@ObjectFactory`, 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 82. Custom object factories

```java
public class DtoFactory {

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

```java
public class EntityFactory {

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

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

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

    CarDto carToCarDto(Car car);

    Car carDtoToCar(CarDto carDto);
}
```

```java
//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;
    }
}
```

Example 83. Custom object factories with update methods

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

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

    void updateOwnerDto(Owner owner, @MappingTarget OwnerDto ownerDto);

    void updateOwner(OwnerDto ownerDto, @MappingTarget Owner owner);
}
```

```java
//GENERATED CODE
public class OwnerMapperImpl implements OwnerMapper {

    private final DtoFactory dtoFactory = new DtoFactory();

    private final EntityFactory entityFactory = new EntityFactory();

    private final OwnerMapper ownerMapper = Mappers.getMapper( OwnerMapper.class
);

    @Override
    public void updateOwnerDto(Owner owner, @MappingTarget OwnerDto ownerDto) {
        if ( owner == null ) {
            return;
        }

        if ( owner.getCar() != null ) {
            if ( ownerDto.getCar() == null ) {
                ownerDto.setCar( dtoFactory.createCarDto() );
            }
            // update car within ownerDto
        }
        else {
            ownerDto.setCar( null );
        }

        // updating other properties
    }

    @Override
    public void updateOwner(OwnerDto ownerDto, @MappingTarget Owner owner) {
        if ( ownerDto == null ) {
            return;
        }

        if ( ownerDto.getCar() != null ) {
            if ( owner.getCar() == null ) {
                owner.setCar( entityFactory.createEntity( Car.class ) );
            }
            // update car within owner
        }
        else {
            owner.setCar( null );
        }

        // updating other properties
    }
}
```

In addition, annotating a factory method with `@ObjectFactory` lets you gain access to the mapping sources. Source objects can be added as parameters in the same way as for mapping method. The `@ObjectFactory` annotation is necessary to let MapStruct know that the given method is only a factory method.

Example 84. Custom object factories with `@ObjectFactory`

```java
public class DtoFactory {

    @ObjectFactory
    public CarDto createCarDto(Car car) {
        return // ... custom factory logic
    }
}
```

10. Advanced mapping options

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

10.1. 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. When the target type is a primitive or a boxed type, the String value is taken literal. Bit / octal / decimal / hex patterns are allowed in such a case as long as they are a valid literal. In all other cases, constant or default values 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 example shows some mappings using default values and constants:

Example 85. Mapping method with default values and constants

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

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

    @Mapping(target = "stringProperty", source = "stringProp", defaultValue =
"undefined")
    @Mapping(target = "longProperty", source = "longProp", 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.getLongProperty() == 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>`.

10.2. Expressions

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

Currently only Java is supported as a 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 86. Mapping method using an expression

```java
@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 it's used otherwise explicitly in the `SourceTargetMapper`). This can be resolved by defining `imports` on the `@Mapper` annotation.

Example 87. Declaring an import

```java
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);
}
```

10.3. Default Expressions

Default expressions are a combination of default values and expressions. They will only be used when the source attribute is `null`.

The same warnings and restrictions apply to default expressions that apply to expressions. Only Java is supported, and MapStruct will not validate the expression at generation-time.

The example below demonstrates how a default expression can be used to set a value when the source attribute is not present (e.g. is `null`):

Example 88. Mapping method using a default expression

```java
imports java.util.UUID;

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

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

    @Mapping(target="id", source="sourceId", defaultExpression = "java(
UUID.randomUUID().toString() )")
    Target sourceToTarget(Source s);
}
```

The example demonstrates how to use defaultExpression to set an `ID` field if the source field is null, this could be used to take the existing `sourceId` from the source object if it is set, or create a new `Id` if it isn't. Please note that the fully qualified package name is specified because MapStruct does not take care of the import of the `UUID` class (unless it's used otherwise explicitly in the `SourceTargetMapper`). This can be resolved by defining imports on the @Mapper annotation (see Expressions).

10.4. Subclass Mapping

When both input and result types have an inheritance relation, you would want the correct specialization be mapped to the matching specialization. Suppose an `Apple` and a `Banana`, which are both specializations of `Fruit`.

Example 89. Specifying the sub class mappings of a fruit mapping

```java
@Mapper
public interface FruitMapper {

    @SubclassMapping( source = AppleDto.class, target = Apple.class )
    @SubclassMapping( source = BananaDto.class, target = Banana.class )
    Fruit map( FruitDto source );

}
```

If you would just use a normal mapping both the `AppleDto` and the `BananaDto` would be made into a `Fruit` object, instead of an `Apple` and a `Banana` object. By using the subclass mapping an `AppleDtoToApple` mapping will be used for `AppleDto` objects, and an `BananaDtoToBanana` mapping will be used for `BananaDto` objects. If you try to map a `GrapeDto` it would still turn it into a `Fruit`.

In the case that the `Fruit` is an abstract class or an interface, you would get a compile error.

To allow mappings for abstract classes or interfaces you need to set the `subclassExhaustiveStrategy` to `RUNTIME_EXCEPTION`, you can do this at the `@MapperConfig`, `@Mapper` or `@BeanMapping` annotations. If you then pass a `GrapeDto` an `IllegalArgumentException` will be thrown because it is unknown how to map a `GrapeDto`. Adding the missing (`@SubclassMapping`) for it will fix that.

Mapping method selection based on qualifiers can be used to further control which methods may be chosen to map a specific subclass. For that, you will need to use one of `SubclassMapping#qualifiedByName` or `SubclassMapping#qualifiedBy`.

> 💡 If the mapping method for the subclasses does not exist it will be created and any other annotations on the fruit mapping method will be inherited by the newly generated mappings.

> ℹ️ Combining `@SubclassMapping` with update methods is not supported. If you try to use subclass mappings there will be a compile error. The same issue exists for the `@Context` and `@TargetType` parameters.

10.5. Determining the result type

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

Example 90. Specifying the result type of a bean mapping method

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

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

}
```

```java
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 where 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`.

10.6. 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 `@MapperConfig`, 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.
- **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 `@MapperConfig#nullValueMappingStrategy`.

10.7. Controlling mapping result for 'null' collection or map arguments

With Controlling mapping result for 'null' arguments it is possible to control how the return type should be constructed when the source argument of the mapping method is `null`. That is applied for all mapping methods (bean, iterable or map mapping methods).

However, MapStruct also offers a more dedicated way to control how collections / maps should be mapped. e.g. return default (empty) collections / maps, but return `null` for beans.

For collections (iterables) this can be controlled through:

- `MapperConfig#nullValueIterableMappingStrategy`
- `Mapper#nullValueIterableMappingStrategy`
- `IterableMapping#nullValueMappingStrategy`

For maps this can be controlled through:

- `MapperConfig#nullValueMapMappingStrategy`

- `Mapper#nullValueMapMappingStrategy`

- `MapMapping#nullValueMappingStrategy`

How the value of the `NullValueMappingStrategy` is applied is the same as in Controlling mapping result for 'null' arguments

10.8. Controlling mapping result for 'null' properties in bean mappings (update mapping methods only).

MapStruct offers control over the property to set in an `@MappingTarget` annotated target bean when the source property equals `null` or the presence check method results in 'absent'.

By default the target property will be set to null.

However:

1. By specifying `nullValuePropertyMappingStrategy` = `NullValuePropertyMappingStrategy.SET_TO_DEFAULT` on `@Mapping`, `@BeanMapping`, `@Mapper` or `@MapperConfig`, the mapping result can be altered to return **default** values. For `List` MapStruct generates an `ArrayList`, for `Map` a `LinkedHashMap`, for arrays an empty array, for `String` `""` and for primitive / boxed types a representation of `false` or `0`. For all other objects an new instance is created. Please note that a default constructor is required. If not available, use the `@Mapping#defaultValue`.

2. By specifying `nullValuePropertyMappingStrategy` = `NullValuePropertyMappingStrategy.IGNORE` on `@Mapping`, `@BeanMapping`, `@Mapper` or `@MapperConfig`, the mapping result will be equal to the original value of the `@MappingTarget` annotated target.

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

> 🛈 Some types of mappings (collections, maps), in which MapStruct is instructed to use a getter or adder as target accessor (see `CollectionMappingStrategy`), MapStruct will always generate a source property null check, regardless of the value of the `NullValuePropertyMappingStrategy`, to avoid addition of `null` to the target collection or map. Since the target is assumed to be initialised this strategy will not be applied.

> 💡 `NullValuePropertyMappingStrategy` also applies when the presence checker returns `not present`.

10.9. Controlling checking result for 'null' properties in bean mapping

MapStruct offers control over when to generate a `null` check. By default (`nullValueCheckStrategy = NullValueCheckStrategy.ON_IMPLICIT_CONVERSION`) a `null` check will be generated for:

- direct setting of source value to target value when target is primitive and source is not.

- applying type conversion and then:

    a. calling the setter on the target.

    b. calling another type conversion and subsequently calling the setter on the target.

    c. calling a mapping method and subsequently calling the setter on the target.

First calling a mapping method on the source property is not protected by a null check. Therefore generated mapping methods will do a null check prior to carrying out mapping on a source property. Handwritten mapping methods must take care of null value checking. They have the possibility to add 'meaning' to `null`. For instance: mapping `null` to a default value.

The option `nullValueCheckStrategy = NullValueCheckStrategy.ALWAYS` will always include a null check when source is non primitive, unless a source presence checker is defined on the source bean.

The strategy works in a hierarchical fashion. `@Mapping#nullValueCheckStrategy` will override `@BeanMapping#nullValueCheckStrategy`, `@BeanMapping#nullValueCheckStrategy` will override `@Mapper#nullValueCheckStrategy` and `@Mapper#nullValueCheckStrategy` will override `@MapperConfig#nullValueCheckStrategy`.

10.10. Source presence checking

Some frameworks generate bean properties that have a source presence checker. Often this is in the form of a method `hasXYZ`, `XYZ` being a property on the source bean in a bean mapping method. MapStruct will call this `hasXYZ` instead of performing a `null` check when it finds such `hasXYZ` method.

> The source presence checker name can be changed in the MapStruct service provider interface (SPI). It can also be deactivated in this way.

> Some types of mappings (collections, maps), in which MapStruct is instructed to use a getter or adder as target accessor (see `CollectionMappingStrategy`), MapStruct will always generate a source property null check, regardless the value of the `NullValueCheckStrategy` to avoid addition of `null` to the target collection or map.

10.11. Conditional Mapping

Conditional Mapping is a type of Source presence checking. The difference is that it allows users to write custom condition methods that will be invoked to check if a property needs to be mapped or not. Conditional mapping can also be used to check if a source parameter should be mapped or not.

A custom condition method for properties is a method that is annotated with `org.mapstruct.Condition` and returns `boolean`. A custom condition method for source parameters is annotated with `org.mapstruct.SourceParameterCondition`, `org.mapstruct.Condition(appliesTo = org.mapstruct.ConditionStrategy#SOURCE_PARAMETERS)` or meta-annotated with `Condition(appliesTo = ConditionStrategy#SOURCE_PARAMETERS)`

e.g. if you only want to map a String property when it is not `null`, and it is not empty then you can do something like:

Example 91. Mapper using custom condition check method

```java
@Mapper
public interface CarMapper {

    CarDto carToCarDto(Car car);

    @Condition
    default boolean isNotEmpty(String value) {
        return value != null && !value.isEmpty();
    }
}
```

The generated mapper will look like:

Example 92. Custom condition check in generated implementation

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

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

        CarDto carDto = new CarDto();

        if ( isNotEmpty( car.getOwner() ) ) {
            carDto.setOwner( car.getOwner() );
        }

        // Mapping of other properties

        return carDto;
    }
}
```

When using this in combination with an update mapping method it will replace the `null-check` there, for example:

Example 93. Update mapper using custom condition check method

```java
@Mapper
public interface CarMapper {

    CarDto carToCarDto(Car car, @MappingTarget CarDto carDto);

    @Condition
    default boolean isNotEmpty(String value) {
        return value != null && !value.isEmpty();
    }
}
```

The generated update mapper will look like:

Example 94. Custom condition check in generated implementation

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

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

        if ( isNotEmpty( car.getOwner() ) ) {
            carDto.setOwner( car.getOwner() );
        } else {
            carDto.setOwner( null );
        }

        // Mapping of other properties

        return carDto;
    }
}
```

Additionally `@TargetPropertyName` or `@SourcePropertyName` of type `java.lang.String` can be used in custom condition check method:

Example 95. Mapper using custom condition check method with @TargetPropertyName and @SourcePropertyName

```java
@Mapper
public interface CarMapper {

    @Mapping(target = "owner", source = "ownerName")
    CarDto carToCarDto(Car car, @MappingTarget CarDto carDto);

    @Condition
    default boolean isNotEmpty(
        String value,
        @TargetPropertyName String targetPropertyName,
        @SourcePropertyName String sourcePropertyName
    ) {

        if ( targetPropertyName.equals( "owner" )
            && sourcePropertyName.equals( "ownerName" ) ) {

            return value != null
                    && !value.isEmpty()
                    && !value.equals( value.toLowerCase() );
        }
        return value != null && !value.isEmpty();
    }
}
```

The generated mapper with @TargetPropertyName and @SourcePropertyName will look like:

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

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

        if ( isNotEmpty( car.getOwner(), "owner", "ownerName" ) ) {
            carDto.setOwner( car.getOwner() );
        } else {
            carDto.setOwner( null );
        }

        // Mapping of other properties

        return carDto;
    }
}
```

> ❗ If there is a custom `@Condition` method applicable for the property it will have a precedence over a presence check method in the bean itself.

> ℹ️ Methods annotated with `@Condition` in addition to the value of the source property can also have the source parameter as an input.
>
> `@TargetPropertyName` and `@SourcePropertyName` parameters can only be used in `@Condition` methods.

Mapping method selection based on qualifiers is also valid for `@Condition` methods. In order to use a more specific condition method you will need to use one of `Mapping#conditionQualifiedByName` or `Mapping#conditionQualifiedBy`.

If we want to only map cars that have an id provided then we can do something like:

Example 97. Mapper using custom condition source parameter check method

```java
@Mapper
public interface CarMapper {

    CarDto carToCarDto(Car car);

    @SourceParameterCondition
    default boolean hasCar(Car car) {
        return car != null && car.getId() != null;
    }
}
```

The generated mapper will look like:

Example 98. Custom condition source parameter check generated implementation

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

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

        CarDto carDto = new CarDto();

        carDto.setOwner( car.getOwner() );

        // Mapping of other properties

        return carDto;
    }
}
```

10.12. 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 99. Mapper using custom method declaring checked exception

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

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

The hand written logic might look like this:

Example 100. Custom mapping method declaring checked exception

```java
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 101. try-catch block in generated implementation

```java
// 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.

11. 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.

11.1. 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 102. Update method inheriting its configuration

```java
@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 `carDtoToCar()` 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.

> 🛈 `@InheritConfiguration` cannot refer to methods in a used mapper.

11.2. 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.

In the example below, there is no need to write the inverse mapping manually. Think of a case where there are several mappings, so writing the inverse ones can be cumbersome and error prone.

Example 103. Inverse mapping method inheriting its configuration and ignoring some of them

```java
@Mapper
public interface CarMapper {

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

    @InheritInverseConfiguration
    @Mapping(target = "numberOfSeats", ignore = true)
    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 needs to be specified using the `name` property like this: `@InheritInverseConfiguration(name = "carToDto")`.

`@InheritConfiguration` takes, in case of conflict precedence over `@InheritInverseConfiguration`.

Configurations are inherited transitively. So if method `C` defines a mapping `@Mapping( target = "x", ignore = true)`, `B` defines a mapping `@Mapping( target = "y", ignore = true)`, then if `A` inherits from `B` inherits from `C`, `A` will inherit mappings for both property `x` and `y`.

`@Mapping#expression`, `@Mapping#defaultExpression`, `@Mapping#defaultValue` and `@Mapping#constant` are excluded (silently ignored) in `@InheritInverseConfiguration`.

`@Mapping#ignore` is only applied when `@Mapping#source` is also present in `@InheritInverseConfiguration`.

Reverse mapping of nested source properties is experimental as of the 1.1.0.Beta2 release. 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.

> **ℹ** `@InheritInverseConfiguration` cannot refer to methods in a used mapper.

11.3. 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 104. Mapper configuration class and mapper using it

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

```java
@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 105. Mapper configuration class with prototype methods

```java
@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);
}
```

```java
@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 = ···)` which will cause `AUTO_INHERIT_FROM_CONFIG` to be ignored.

- `AUTO_INHERIT_REVERSE_FROM_CONFIG`: the inverse 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 `@InheritInverseConfiguration(name = ···)` which will cause `AUTO_INHERIT_REVERSE_FROM_CONFIG` to be ignored.

- `AUTO_INHERIT_ALL_FROM_CONFIG`: both the configuration and the inverse configuration will be inherited automatically. The same rules apply as for `AUTO_INHERIT_FROM_CONFIG` or `AUTO_INHERIT_REVERSE_FROM_CONFIG`.

12. 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.

12.1. 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 106. 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 107. Implementing a decorator

```java
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.

12.1.1. 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 108. Spring-based decorator

```java
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 109. Using a decorated mapper

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

12.1.2. 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 110. JSR 330 based decorator

```java
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 111. Using a decorated mapper with JSR 330

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

12.2. 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, in a type reference in `Mapper#uses`, or in a type used as `@Context` parameter.

Example 112. Mapper with @BeforeMapping and @AfterMapping hooks

```java
@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;
        }

        CarDto carDto = new CarDto();
        // attributes mapping ...

        fillTank( car, carDto );

        return carDto;
    }
}
```

If the `@BeforeMapping` / `@AfterMapping` method has parameters, the method invocation is only generated if the return type of the method (if non-`void`) is assignable to the return type of the mapping method and 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.

- Parameters annotated with `@Context` are populated with the context parameters of the mapping method.

- Any other parameter is populated with a source parameter of the mapping.

For non-`void` methods, the return value of the method invocation is returned as the result of the mapping method if it is not `null`.

As with mapping methods, it is possible to specify type parameters for before/after-mapping methods.

Example 113. Mapper with @AfterMapping hook that returns a non-null value

```java
@Mapper
public abstract class VehicleMapper {

    @PersistenceContext
    private EntityManager entityManager;

    @AfterMapping
    protected <T> T attachEntity(@MappingTarget T entity) {
        return entityManager.merge(entity);
    }

    public abstract CarDto toCarDto(Car car);
}

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

    public CarDto toCarDto(Car car) {
        if ( car == null ) {
            return null;
        }

        CarDto carDto = new CarDto();
        // attributes mapping ...

        CarDto target = attachEntity( carDto );
        if ( target != null ) {
            return target;
        }

        return carDto;
    }
}
```

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 of the method invocation is determined primarily by their variant:

1. `@BeforeMapping` methods without parameters, a `@MappingTarget` parameter or a `@TargetType` parameter are called before any null-checks on source parameters and constructing a new target bean.

2. `@BeforeMapping` methods with a `@MappingTarget` parameter are called after constructing a new target bean.

3. `@AfterMapping` methods are called at the end of the mapping method before the last `return` statement.

Within those groups, the method invocations are ordered by their location of definition:

1. Methods declared on `@Context` parameters, ordered by the parameter order.

2. Methods implemented in the mapper itself.

3. Methods from types referenced in `Mapper#uses()`, in the order of the type declaration in the annotation.

4. Methods declared in one type are used after methods declared in their super-type.

Important: the order of methods declared within one type can not be guaranteed, as it depends on the compiler and the processing environment implementation.

> Before/After-mapping methods can also be used with builders:
>
> - `@BeforeMapping` methods with a `@MappingTarget` parameter of the real target will not be invoked because it is only available after the mapping was already performed.
>
> - To be able to modify the object that is going to be built, the `@AfterMapping` annotated method must have the builder as `@MappingTarget` annotated parameter. The `build` method is called when the `@AfterMapping` annotated method scope finishes.
>
> - The `@AfterMapping` annotated method can also have the real target as `@TargetType` or `@MappingTarget`. It will be invoked after the real target was built (first the methods annotated with `@TargetType`, then the methods annotated with `@MappingTarget`)

13. Using the MapStruct SPI

To use a custom SPI implementation, it must be located in a separate JAR file together with a file named after the SPI (e.g. `org.mapstruct.ap.spi.AccessorNamingStrategy`) in `META-INF/services/` with the fully qualified name of your custom implementation as content (e.g. `org.mapstruct.example.CustomAccessorNamingStrategy`). This JAR file needs to be added to the

annotation processor classpath (i.e. add it next to the place where you added the mapstruct-processor jar).

> ℹ️ It might also be necessary to add the jar to your IDE's annotation processor factory path. Otherwise you might get an error stating that it cannot be found, while a run using your build tool does succeed.

13.1. Custom Accessor Naming Strategy

SPI name: `org.mapstruct.ap.spi.AccessorNamingStrategy`

MapStruct offers the possibility to override the `AccessorNamingStrategy` via the Service Provider Interface (SPI). A nice example is the use of the fluent API on the source object `GolfPlayer` and `GolfPlayerDto` below.

Example 114. Source object GolfPlayer with fluent API.

```java
public class GolfPlayer {

    private double handicap;
    private String name;

    public double handicap() {
        return handicap;
    }

    public GolfPlayer withHandicap(double handicap) {
        this.handicap = handicap;
        return this;
    }

    public String name() {
        return name;
    }

    public GolfPlayer withName(String name) {
        this.name = name;
        return this;
    }
}
```

Example 115. Source object GolfPlayerDto with fluent API.

```java
public class GolfPlayerDto {

    private double handicap;
    private String name;

    public double handicap() {
        return handicap;
    }

    public GolfPlayerDto withHandicap(double handicap) {
        this.handicap = handicap;
        return this;
    }

    public String name() {
        return name;
    }

    public GolfPlayerDto withName(String name) {
        this.name = name;
        return this;
    }
}
```

We want `GolfPlayer` to be mapped to a target object `GolfPlayerDto` similar like we 'always' do this:

Example 116. Source object with fluent API.

```java
@Mapper
public interface GolfPlayerMapper {

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

    GolfPlayerDto toDto(GolfPlayer player);

    GolfPlayer toPlayer(GolfPlayerDto player);

}
```

This can be achieved with implementing the SPI `org.mapstruct.ap.spi.AccessorNamingStrategy` as in the following example. Here's an implemented `org.mapstruct.ap.spi.AccessorNamingStrategy`:

Example 117. CustomAccessorNamingStrategy

```java
/**
 * A custom {@link AccessorNamingStrategy} recognizing getters in the form of
{@code property()} and setters in the
 * form of {@code withProperty(value)}.
 */
public class CustomAccessorNamingStrategy extends DefaultAccessorNamingStrategy {

    @Override
    public boolean isGetterMethod(ExecutableElement method) {
        String methodName = method.getSimpleName().toString();
        return !methodName.startsWith( "with" ) && method.getReturnType().getKind
() != TypeKind.VOID;
    }

    @Override
    public boolean isSetterMethod(ExecutableElement method) {
        String methodName = method.getSimpleName().toString();
        return methodName.startsWith( "with" ) && methodName.length() > 4;
    }

    @Override
    public String getPropertyName(ExecutableElement getterOrSetterMethod) {
        String methodName = getterOrSetterMethod.getSimpleName().toString();
        return IntrospectorUtils.decapitalize( methodName.startsWith( "with" ) ?
methodName.substring(  4 ) : methodName );
    }
}
```

The `CustomAccessorNamingStrategy` makes use of the `DefaultAccessorNamingStrategy` (also available in mapstruct-processor) and relies on that class to leave most of the default behaviour unchanged.

> 💡 Fore more details: The example above is present in our examples repository (https://github.com/mapstruct/mapstruct-examples).

13.2. Mapping Exclusion Provider

SPI name: `org.mapstruct.ap.spi.MappingExclusionProvider`

MapStruct offers the possibility to override the `MappingExclusionProvider` via the Service Provider Interface (SPI). A nice example is to not allow MapStruct to create an automatic sub-mapping for a certain type, i.e. MapStruct will not try to generate an automatic sub-mapping method for an excluded type.

The `DefaultMappingExclusionProvider` will exclude all types under the `java` or `javax` packages. This means that MapStruct will not try to generate an automatic sub-mapping method between some custom type and some type declared in the Java class library.

Example 118. Source object

```java
public class Source {

    static class NestedSource {
        private String property;
        // getters and setters
    }

    private NestedSource nested;
    // getters and setters
}
```

Example 119. Target object

```java
public class Target {

    static class NestedTarget {
        private String property;
        // getters and setters
    }

    private NestedTarget nested;
    // getters and setters
}
```

Example 120. Mapper definition

```java
@Mapper
public interface ErroneousCustomExclusionMapper {

    Target map(Source source);
}
```

We want to exclude the `NestedTarget` from the automatic sub-mapping method generation.

Example 121. CustomMappingExclusionProvider

```java
import java.util.regex.Pattern;
import javax.lang.model.element.Name;
import javax.lang.model.element.TypeElement;

import org.mapstruct.ap.spi.MappingExclusionProvider;

public class CustomMappingExclusionProvider implements MappingExclusionProvider {
    private static final Pattern JAVA_JAVAX_PACKAGE = Pattern.compile( "^javax?
\\..*" );

    @Override
    public boolean isExcluded(TypeElement typeElement) {
        Name name = typeElement.getQualifiedName();
        return name.length() != 0 && ( JAVA_JAVAX_PACKAGE.matcher( name ).matches
() ||
            name.toString().equals(
"org.mapstruct.ap.test.nestedbeans.exclusions.custom.Target.NestedTarget" ) );
    }
}
```

13.3. Custom Builder Provider

SPI name: org.mapstruct.ap.spi.BuilderProvider

MapStruct offers the possibility to override the `DefaultProvider` via the Service Provider Interface (SPI). A nice example is to provide support for a custom builder strategy.

Example 122. Custom Builder Provider which disables Builder support

```java
import javax.lang.model.type.TypeMirror;

public class NoOpBuilderProvider implements BuilderProvider {

    @Override
    public BuilderInfo findBuilderInfo(TypeMirror type) {
        return null;
    }

}
```

13.4. Custom Enum Naming Strategy

SPI name: `org.mapstruct.ap.spi.EnumMappingStrategy`

MapStruct offers the possibility to override the `EnumMappingStrategy` via the Service Provider Interface (SPI). This can be used when you have certain enums that follow some conventions within your organization. For example all enums which implement an interface named `CustomEnumMarker` are prefixed with `CUSTOM_` and the default value for them when mapping from `null` is `UNSPECIFIED`

Example 123. Normal Enum

```
public enum CheeseType {
    BRIE,
    ROQUEFORT;
}
```

Example 124. Custom marker enum

```
public enum CustomCheeseType implements CustomEnumMarker {

    UNSPECIFIED,
    CUSTOM_BRIE,
    CUSTOM_ROQUEFORT;
}
```

We want `CheeseType` and `CustomCheeseType` to be mapped without the need to manually define the value mappings:

Example 125. Custom enum mapping

```
@Mapper
public interface CheeseTypeMapper {

    CheeseType map(CustomCheeseType cheese);

    CustomCheeseType map(CheeseType cheese);
}
```

This can be achieved with implementing the SPI `org.mapstruct.ap.spi.EnumMappingStrategy` as in the following example. Here's an implemented `org.mapstruct.ap.spi.EnumMappingStrategy`:

Example 126. Custom enum naming strategy

```java
public class CustomEnumMappingStrategy extends DefaultEnumMappingStrategy {

    @Override
    public String getDefaultNullEnumConstant(TypeElement enumType) {
        if ( isCustomEnum( enumType ) ) {
            return "UNSPECIFIED";
        }

        return super.getDefaultNullEnumConstant( enumType );
    }

    @Override
    public String getEnumConstant(TypeElement enumType, String enumConstant) {
        if ( isCustomEnum( enumType ) ) {
            return getCustomEnumConstant( enumConstant );
        }
        return super.getEnumConstant( enumType, enumConstant );
    }
     protected String getCustomEnumConstant(String enumConstant) {
        if ( "UNSPECIFIED".equals( enumConstant ) ) {
            return MappingConstantsGem.NULL;
        }
         return enumConstant.replace( "CUSTOM_", "" );
    }
     protected boolean isCustomEnum(TypeElement enumType) {
        for ( TypeMirror enumTypeInterface : enumType.getInterfaces() ) {
            if ( typeUtils.asElement( enumTypeInterface ).getSimpleName()
.contentEquals( "CustomEnumMarker" ) ) {
                return true;
            }
        }
         return false;
    }
}
```

The generated code then for the `CheeseMapper` looks like:

Example 127. Generated CheeseTypeMapper

```java
public class CheeseTypeMapperImpl implements CheeseTypeMapper {

    @Override
    public CheeseType map(CustomCheeseType cheese) {
        if ( cheese == null ) {
            return null;
        }

        CheeseType cheeseType;

        switch ( cheese ) {
            case UNRECOGNIZED: cheeseType = null;
            break;
            case CUSTOM_BRIE: cheeseType = CheeseType.BRIE;
            break;
            case CUSTOM_ROQUEFORT: cheeseType = CheeseType.ROQUEFORT;
            break;
            default: throw new IllegalArgumentException( "Unexpected enum
constant: " + cheese );
        }

        return cheeseType;
    }

    @Override
    public CustomCheeseType map(CheeseType cheese) {
        if ( cheese == null ) {
            return CustomCheeseType.UNSPECIFIED;
        }

        CustomCheeseType customCheeseType;

        switch ( cheese ) {
            case BRIE: customCheeseType = CustomCheeseType.CUSTOM_BRIE;
            break;
            case ROQUEFORT: customCheeseType = CustomCheeseType.CUSTOM_ROQUEFORT;
            break;
            default: throw new IllegalArgumentException( "Unexpected enum
constant: " + cheese );
        }

        return customCheeseType;
    }
}
```

13.5. Custom Enum Transformation Strategy

SPI name: `org.mapstruct.ap.spi.EnumTransformationStrategy`

MapStruct offers the possibility to other transformations strategies by implementing `EnumTransformationStrategy` via the Service Provider Interface (SPI). A nice example is to provide support for a custom transformation strategy.

Example 128. Custom Enum Transformation Strategy which lower-cases the value and applies a suffix

```java
import org.mapstruct.ap.spi.EnumTransformationStrategy;

public class CustomEnumTransformationStrategy implements
EnumTransformationStrategy {

    @Override
    public String getStrategyName() {
        return "custom";
    }

    @Override
    public String transform(String value, String configuration) {
        return value.toLowerCase() + configuration;
    }
}
```

13.6. Additional Supported Options Provider

SPI name: `org.mapstruct.ap.spi.AdditionalSupportedOptionsProvider`

MapStruct offers the ability to pass through declared compiler args (or "options") provided to the MappingProcessor to the individual SPIs, by implementing `AdditionalSupportedOptionsProvider` via the Service Provider Interface (SPI).

Example 129. Custom Additional Supported Options Provider that declares
`myorg.custom.defaultNullEnumConstant` *as an option to pass through*

```java
import java.util.Collections;
import java.util.Set;

import org.mapstruct.ap.spi.AdditionalSupportedOptionsProvider;

public class CustomAdditionalSupportedOptionsProvider implements
AdditionalSupportedOptionsProvider {

    @Override
    public Set<String> getAdditionalSupportedOptions() {
        return Collections.singleton( "myorg.custom.defaultNullEnumConstant" );
    }

}
```

The value of this option is provided by including an `arg` to the `compilerArgs` tag when defining your custom SPI implementation.

Example 130. Example maven configuration with additional options

```xml
<configuration>
    <annotationProcessorPaths>
        <path>
            <groupId>org.myorg</groupId>
            <artifactId>custom-spi-impl</artifactId>
            <version>${project.version}</version>
        </path>
    </annotationProcessorPaths>
    <compilerArgs>
        <arg>-Amyorg.custom.defaultNullEnumConstant=MISSING</arg>
    </compilerArgs>
</configuration>
```

Your custom SPI implementations can then access this configured value via `MapStructProcessingEnvironment#getOptions()`.

Example 131. Accessing your custom options

```java
import javax.lang.model.element.TypeElement;

import org.mapstruct.ap.spi.DefaultEnumMappingStrategy;
import org.mapstruct.ap.spi.MapStructProcessingEnvironment;

public class UnknownEnumMappingStrategy extends DefaultEnumMappingStrategy {

    private String defaultNullEnumConstant;

    @Override
    public void init(MapStructProcessingEnvironment processingEnvironment) {
        super.init( processingEnvironment );
        defaultNullEnumConstant = processingEnvironment.getOptions().get(
"myorg.custom.defaultNullEnumConstant" );
    }

    @Override
    public String getDefaultNullEnumConstant(TypeElement enumType) {
        return defaultNullEnumConstant;
    }
}
```

14. Third-party API integration

14.1. Non-shipped annotations

There are various use-cases you must resolve ambiguity for MapStruct to use a correct piece of code. However, the primary goal of MapStruct is to focus on bean mapping without polluting the entity code. For that reason, MapStruct is flexible enough to interact with already defined annotations from third-party libraries. The requirement to enable this behavior is to match the *name* of such annotation. Hence, we say that annotation can be *from any package*.

The annotations *named* `@ConstructorProperties` and `@Default` are currently examples of this kind of annotation.

> ⚠️ If such named third-party annotation exists, it does not guarantee its `@Target` matches with the intended placement. Be aware of placing a third-party annotation just for sake of mapping is not recommended as long as it might lead to unwanted side effects caused by that library.

A very common case is that no third-party dependency imported to your project provides such annotation or is inappropriate for use as already described. In such cases create your own annotation, for example:

```java
package foo.support.mapstruct;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Target(ElementType.CONSTRUCTOR)
@Retention(RetentionPolicy.CLASS)
public @interface Default {

}
```

14.2. Lombok

MapStruct works together with Project Lombok as of MapStruct 1.2.0.Beta1 and Lombok 1.16.14.

MapStruct takes advantage of generated getters, setters, and constructors and uses them to generate the mapper implementations. Be reminded that the generated code by Lombok might not always be compatible with the expectations from the individual mappings. In such a case, either Mapstruct mapping must be changed or Lombok must be configured accordingly using `lombok.config` for mutual synergy.

> ⚠️ Lombok 1.18.16 introduces a breaking change (changelog). The additional annotation processor `lombok-mapstruct-binding` (Maven) must be added otherwise MapStruct stops working with Lombok. This resolves the compilation issues of Lombok and MapStruct modules.
>
> ```xml
> <path>
>     <groupId>org.projectlombok</groupId>
>     <artifactId>lombok-mapstruct-binding</artifactId>
>     <version>0.2.0</version>
> </path>
> ```

14.2.1. Set up

The set up using Maven or Gradle does not differ from what is described in Set up. Additionally, you need to provide Lombok dependencies.

Example 132. Maven configuration

```xml
<properties>
    <org.mapstruct.version>1.6.0.RC1</org.mapstruct.version>
    <org.projectlombok.version>1.18.16</org.projectlombok.version>
    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
</properties>

<dependencies>
    <dependency>
        <groupId>org.mapstruct</groupId>
        <artifactId>mapstruct</artifactId>
        <version>${org.mapstruct.version}</version>
    </dependency>

    <!-- lombok dependency should not end up on classpath -->
    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <version>${org.projectlombok.version}</version>
        <scope>provided</scope>
    </dependency>
</dependencies>

<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>3.8.1</version>
            <configuration>
                <source>1.8</source>
                <target>1.8</target>
                <annotationProcessorPaths>
                    <path>
                        <groupId>org.mapstruct</groupId>
                        <artifactId>mapstruct-processor</artifactId>
                        <version>${org.mapstruct.version}</version>
                    </path>
                    <path>
                        <groupId>org.projectlombok</groupId>
                        <artifactId>lombok</artifactId>
                        <version>${org.projectlombok.version}</version>
                    </path>

                    <!-- additional annotation processor required as of Lombok
1.18.16 -->
                    <path>
                        <groupId>org.projectlombok</groupId>
```

```
                    <artifactId>lombok-mapstruct-binding</artifactId>
                    <version>0.2.0</version>
                </path>
            </annotationProcessorPaths>
        </configuration>
    </plugin>
</plugins>
</build>
```

Example 133. Gradle configuration (3.4 and later)

```
dependencies {

    implementation "org.mapstruct:mapstruct:$1.6.0.RC1"
    compileOnly "org.projectlombok:lombok:1.18.16"
    annotationProcessor "org.projectlombok:lombok-mapstruct-binding:0.2.0"
    annotationProcessor "org.mapstruct:mapstruct-processor:$1.6.0.RC1"
    annotationProcessor "org.projectlombok:lombok:1.18.16"
}
```

The usage combines what you already know from Defining a mapper and Lombok.

Example 134. Usage of MapStruct with Lombok

```java
@Data
public class Source {

    private String test;
}

public class Target {

    private Long testing;

    public Long getTesting() {
        return testing;
    }

    public void setTesting( Long testing ) {
        this.testing = testing;
    }
}

@Mapper
public interface SourceTargetMapper {

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

    @Mapping( source = "test", target = "testing" )
    Target toTarget( Source s );
}
```

A working example can be found on the GitHub project mapstruct-lombok.