The Lunifera DTO DSL

1 Purpose

The “Lunifera DTO DSL” facilitates the creation and handling of data transfer objects (DTOs) in order to carry data between processes. Communication between processes is usually done by calls to remote interfaces and services where each call is a computationally expensive and time-consuming operation. The use of DTOs that aggregate the data that would have been transferred separately reduces the number of calls that are necessary, thus speeding up the operation. Furthermore, DTOs are decoupled from the JPA, thus eliminating burdensome dependencies.

Using the “Lunifera DTO DSL” it is easy to create DTOs and the mapper classes that link them to the respective persistence entities. Property change support is automatically included in order to have current data without direct dependencies on the persistence layer.

2 Overview

The main semantic elements of the Lunifera DTO DSL are:

- “Package” – the root element that contains all the other elements. A model can contain multiple packages.
- “Import” declarations – used to import other DTO models or the entity model files that are covered by the DTOs.
- “Datatype” declarations – a way to define datatypes that can be used (only within the package - private scope).
- “DTO” – the model of a DTO that wraps an entity. It contains further elements such as properties and references. Appropriate mapper methods can be specified.
- “Property” – a reference to an enum, a Java class or a “simple datatype” (as defined in the datatype declaration). Can be inherited from the wrapped entity and offers multiplicity.
- “Reference” – a reference to another DTO. Can be inherited from the wrapped entity and offers multiplicity.
- “Comments” can be added to all elements.

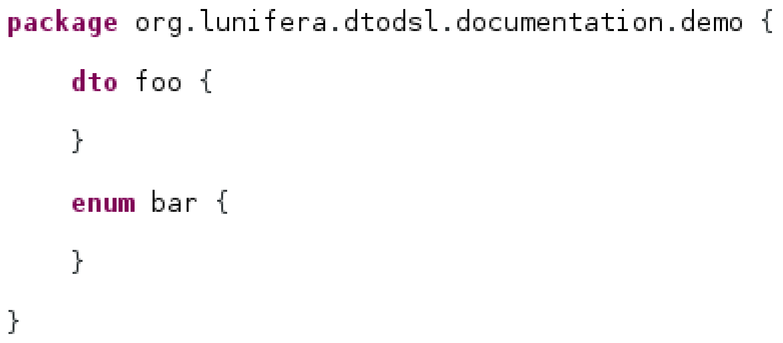
3 Package

Packages are the root element of the Lunifera DTO DSL grammar. Everything is contained in a package: Imports, datatypes, DTOs and enums have to be defined inside the Package definition. One document can contain multiple packages with unique names.

The elements a package can contain are DTOs and enums.

Additionally, a package allows import statements and the declaration of datatypes.

Package.png: A package is the topmost element and contains other items.

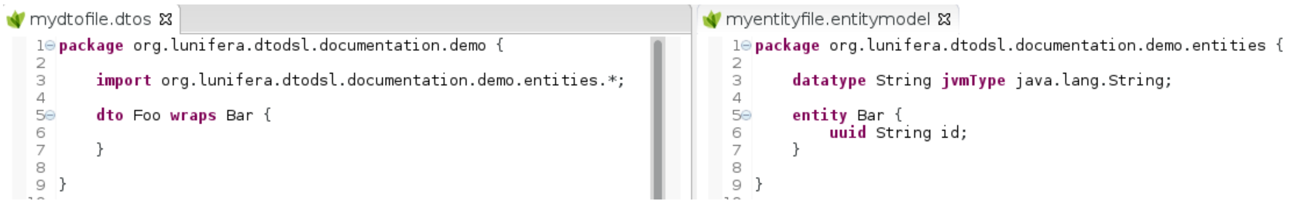


```
package org.lunifera.dtodsl.documentation.demo {

    dto foo {

    }

    enum bar {

    }

}
```

Imports

In order to wrap entities into DTOs, the “DTO DSL” has to reference entities defined in entity model files. Furthermore, it is possible to reference DTOs in other packages. The import statement is a way to address these elements by their fully qualified name.

Import statements allow the use of the *-wildcard.



```
package org.lunifera.dtodsl.documentation.demo {

    import org.lunifera.dtodsl.documentation.demo.entities.*;

    dto Foo wraps Bar {

    }

}
```

```
package org.lunifera.dtodsl.documentation.demo.entities {

    datatype String jvmType java.lang.String;

    entity Bar {
        uuid String id;
    }

}
```

Import.png: Items contained in another package can be accessed and handled if the package is imported.

Datatypes

The “Lunifera DTO DSL” allows the definition of datatypes. These are translated by the inferrer into their standard Java presentation. The behaviour of the generator can be controlled by the datatype definitions.

There are three types of datatype definitions:

- **jvmTypes**

  Datatpye definitons that map types to jvmTypes take the basic form of

  `datatype <name> jvmType <type>.`

  Specifying datatypes in this manner uses an appropriate wrapper class in the generated Java code; adding the keyword “as primitive” enforces the use of primitive datatypes where applicable:

  `datatype foo jvmType Integer` compiles to Integer whereas `datatype foo jvmType Integer as primitive` results in “int”.

- **dateTypes**

  The datatypes for handling temporal information can be defined by the following statement:

```
datatype foo dateType <date|time|timestamp>
```

Datatypes that have been defined in this manner can be used as property variables in DTOs.

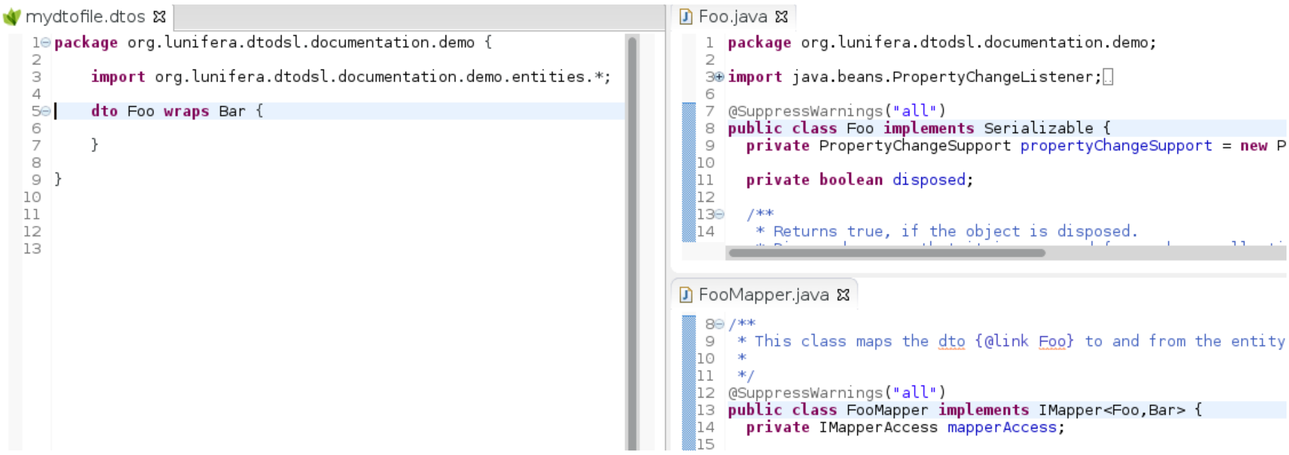
- **Blobs**

  Binary blobs can be handled by defining a datatype with the “as blob” keyword. The Java implementation of such a blob is a byte array.

4 DTOs

DTOs are the most complex elements in the “Lunifera DTO DSL”. A DTO wraps an entity defined in the “Lunifera Entity DSL” and its properties and references into a single object used for communication purposes. DTOs are defined by name, properties, references and wrapper methods. A DTO does not implement business logic.

For each DTO that is defined in a package, the “Lunifera DTO DSL” automatically generates a Java DTO class and a corresponding mapper class that handles data conversion between DTO and entity.



```
package org.lunifera.dtodsl.documentation.demo {

    import org.lunifera.dtodsl.documentation.demo.entities.*;

    dto Foo wraps Bar {

    }

}
```

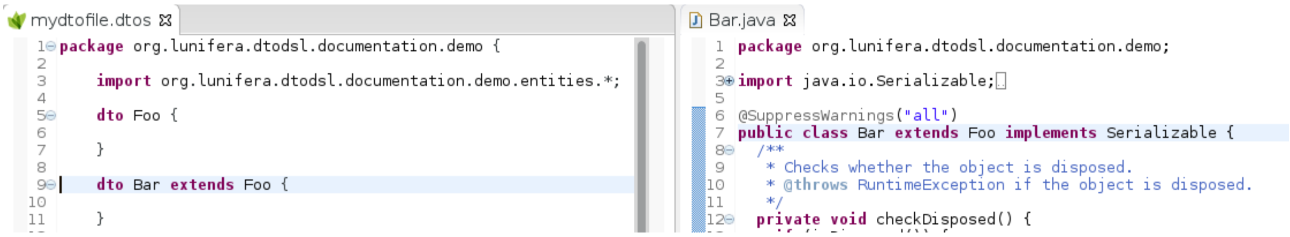
```
package org.lunifera.dtodsl.documentation.demo;

import java.beans.PropertyChangeListener;

@SuppressWarnings("all")
public class Foo implements Serializable {
  private PropertyChangeSupport propertyChangeSupport = new P

  private boolean disposed;

  /**
   * Returns true, if the object is disposed.
```

```
/**
 * This class maps the dto {@link Foo} to and from the entity
 *
 */
@SuppressWarnings("all")
public class FooMapper implements IMapper<Foo,Bar> {
  private IMapperAccess mapperAccess;
```

DTOClassMapper.png: The defined DTO is translated to a Java class and an appropriate mapper class.

The following modifiers can be placed after the “dto” keyword:

- extends – marks a dto that is derived from another dto. That means that the properties and references of the parent dto are inherited.

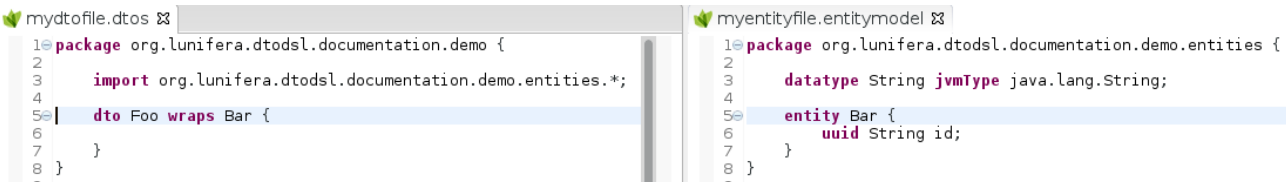


```
package org.lunifera.dtodsl.documentation.demo {

    import org.lunifera.dtodsl.documentation.demo.entities.*;

    dto Foo {

    }

    dto Bar extends Foo {

    }
```

```
package org.lunifera.dtodsl.documentation.demo;

import java.io.Serializable;

@SuppressWarnings("all")
public class Bar extends Foo implements Serializable {
  /**
   * Checks whether the object is disposed.
   * @throws RuntimeException if the object is disposed.
   */
  private void checkDisposed() {
```

Extends.png: The “extends” keyword causes a Java subclass to be created for the DTO.

- wraps – indicates the entity that is handled by the DTO. The entity has to be imported from the entity model file into the DTO package (for importing several entities, the use of the *-wildcard is advisable).



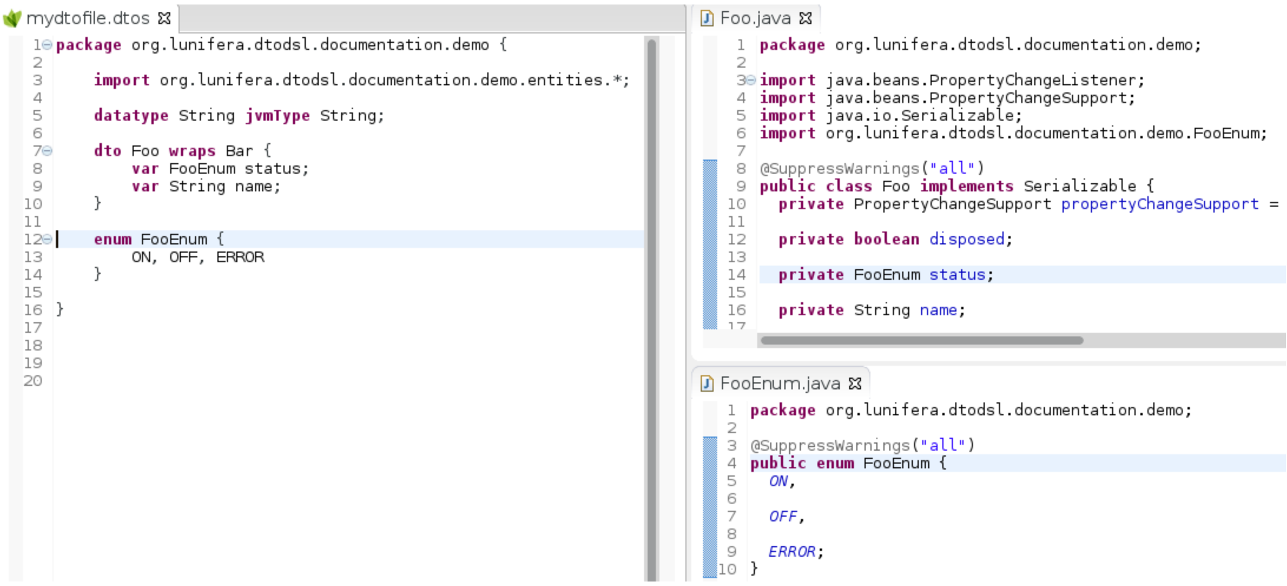
```
package org.lunifera.dtodsl.documentation.demo {

    import org.lunifera.dtodsl.documentation.demo.entities.*;

    dto Foo wraps Bar {

    }
}
```

```
package org.lunifera.dtodsl.documentation.demo.entities {

    datatype String jvmType java.lang.String;

    entity Bar {
        uuid String id;
    }
}
```

Wraps.png: The “wraps” keyword links the DTO to its respective entity. Thus, properties and references of the entity are made available in the DTO (and can be accessed via content assist).

5 Enums

Enums are an abstraction above the Java enum. They compile to enum classes and can be used as properties in DTOs.



```
package org.lunifera.dtodsl.documentation.demo {

    import org.lunifera.dtodsl.documentation.demo.entities.*;

    datatype String jvmType String;

    dto Foo wraps Bar {
        var FooEnum status;
        var String name;
    }

    enum FooEnum {
        ON, OFF, ERROR
    }

}
```

```
package org.lunifera.dtodsl.documentation.demo;

import java.beans.PropertyChangeListener;
import java.beans.PropertyChangeSupport;
import java.io.Serializable;
import org.lunifera.dtodsl.documentation.demo.FooEnum;

@SuppressWarnings("all")
public class Foo implements Serializable {
  private PropertyChangeSupport propertyChangeSupport =

  private boolean disposed;

  private FooEnum status;

  private String name;
```

```
package org.lunifera.dtodsl.documentation.demo;

@SuppressWarnings("all")
public enum FooEnum {
  ON,

  OFF,

  ERROR;
}
```

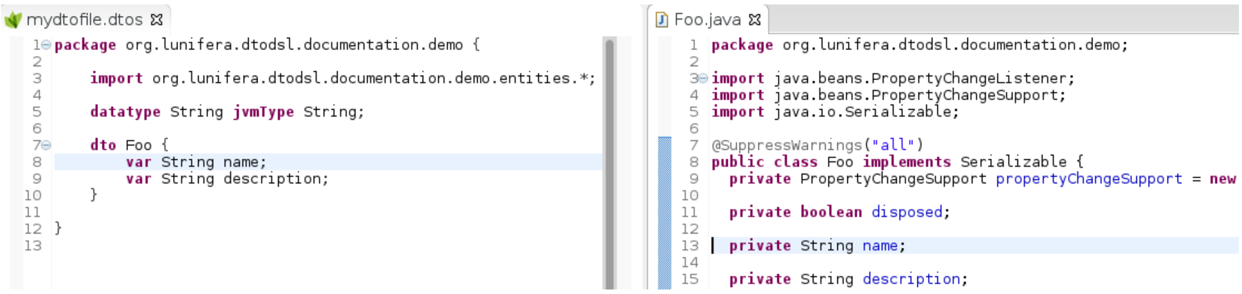
Enum.png: Defining enums allows using them as variables in DTOs.

6 Properties

Properties of DTOs are references to datatypes or enums. They can be regarded as variables and are defined by a keyword followed by a datatype (Java type or datatype defined in the datatype section) and a name.

Property Keywords

- var – The basic property; defines a variable that can be used within the DTO. Appropriate getters and setters are created.

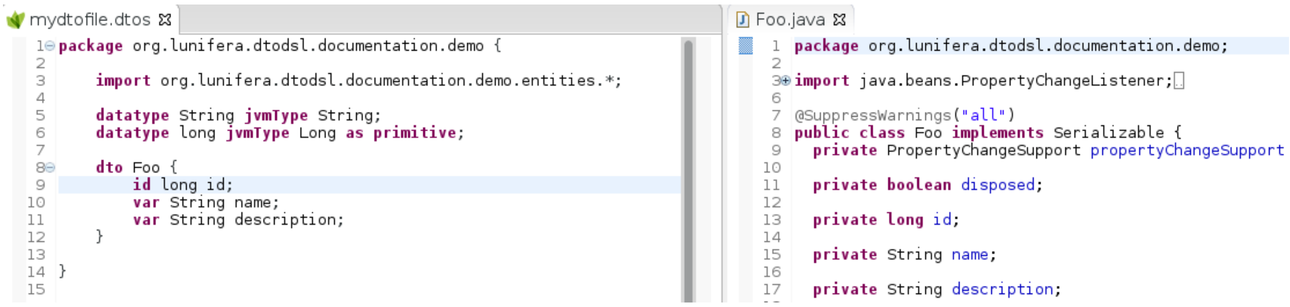


```
package org.lunifera.dtodsl.documentation.demo {

    import org.lunifera.dtodsl.documentation.demo.entities.*;

    datatype String jvmType String;

    dto Foo {
        var String name;
        var String description;
    }

}
```

```
package org.lunifera.dtodsl.documentation.demo;

import java.beans.PropertyChangeListener;
import java.beans.PropertyChangeSupport;
import java.io.Serializable;

@SuppressWarnings("all")
public class Foo implements Serializable {
  private PropertyChangeSupport propertyChangeSupport = new

  private boolean disposed;

  private String name;

  private String description;
```

Var.png: Variables can be defined using the “var” keyword.

- id (deprecated) – defines a field in the DTO for mapping an id property (used as primary key in the persistence layer).

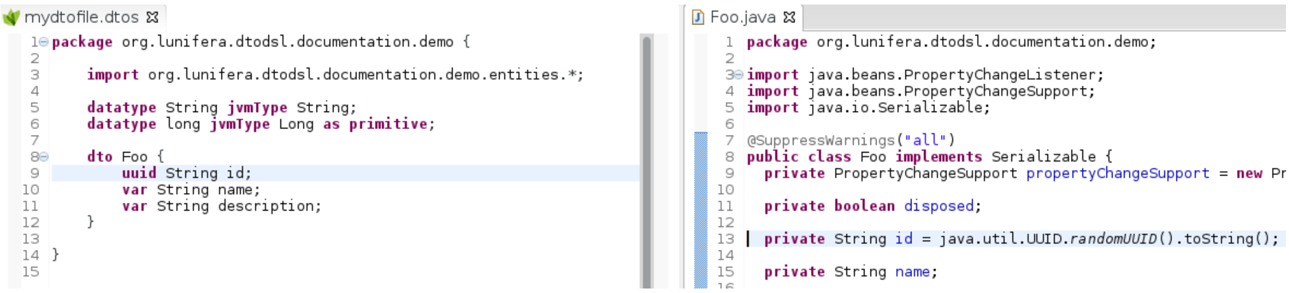


```
package org.lunifera.dtodsl.documentation.demo {

    import org.lunifera.dtodsl.documentation.demo.entities.*;

    datatype String jvmType String;
    datatype long jvmType Long as primitive;

    dto Foo {
        id long id;
        var String name;
        var String description;
    }

}
```

```
package org.lunifera.dtodsl.documentation.demo;

import java.beans.PropertyChangeListener;□

@SuppressWarnings("all")
public class Foo implements Serializable {
  private PropertyChangeSupport propertyChangeSupport

  private boolean disposed;

  private long id;

  private String name;

  private String description;
```

Id.png: The “id” keyword allows mapping an id property (JPA primary key) – deprecated.

- uuid – creates a field for the mapping of Universally Unique IDs as primary keys. UUIDs have to be strings.

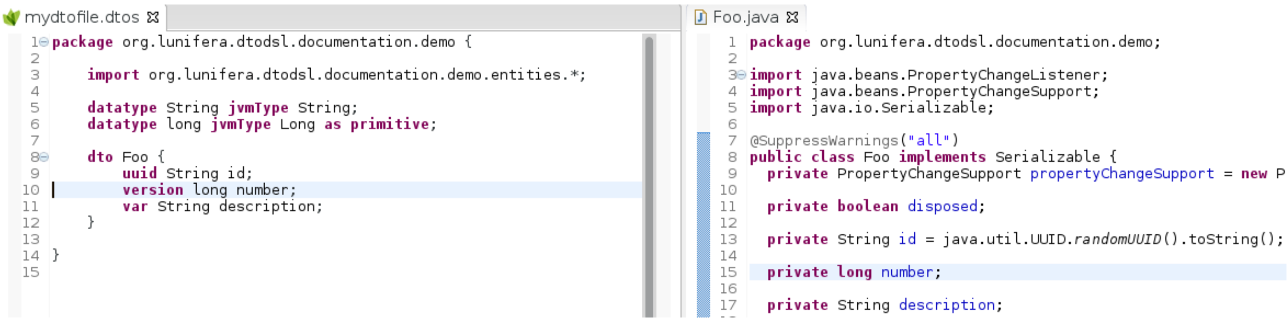


```
package org.lunifera.dtodsl.documentation.demo {

    import org.lunifera.dtodsl.documentation.demo.entities.*;

    datatype String jvmType String;
    datatype long jvmType Long as primitive;

    dto Foo {
        uuid String id;
        var String name;
        var String description;
    }

}
```

```
package org.lunifera.dtodsl.documentation.demo;

import java.beans.PropertyChangeListener;
import java.beans.PropertyChangeSupport;
import java.io.Serializable;

@SuppressWarnings("all")
public class Foo implements Serializable {
  private PropertyChangeSupport propertyChangeSupport = new Pr

  private boolean disposed;

  private String id = java.util.UUID.randomUUID().toString();

  private String name;
```

Uuid.png: By using the “uuid” keyword, reliably unique IDs can be used as primary keys and mapped to a DTO.

- version – defines a field for mapping a version property used by the JPA-Compiler.

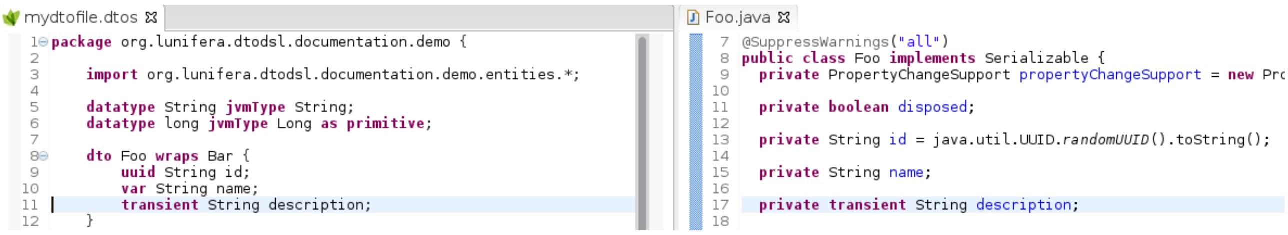


```
package org.lunifera.dtodsl.documentation.demo {

    import org.lunifera.dtodsl.documentation.demo.entities.*;

    datatype String jvmType String;
    datatype long jvmType Long as primitive;

    dto Foo {
        uuid String id;
        version long number;
        var String description;
    }

}
```

```
package org.lunifera.dtodsl.documentation.demo;

import java.beans.PropertyChangeListener;
import java.beans.PropertyChangeSupport;
import java.io.Serializable;

@SuppressWarnings("all")
public class Foo implements Serializable {
  private PropertyChangeSupport propertyChangeSupport = new P

  private boolean disposed;

  private String id = java.util.UUID.randomUUID().toString();

  private long number;

  private String description;
```

version.png: A version property can be mapped into a DTO by the “version” keyword.

- transient – marks the property to be transient. Transient properties are not mapped from the DTO to the entity and thus not persisted. Getter and setter methods are created.

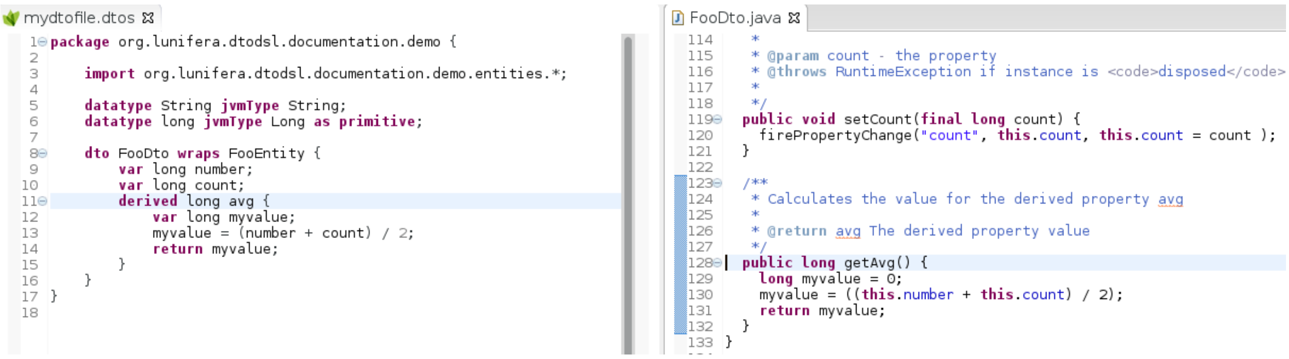


```
package org.lunifera.dtodsl.documentation.demo {

    import org.lunifera.dtodsl.documentation.demo.entities.*;

    datatype String jvmType String;
    datatype long jvmType Long as primitive;

    dto Foo wraps Bar {
        uuid String id;
        var String name;
        transient String description;
    }
```

```
@SuppressWarnings("all")
public class Foo implements Serializable {
  private PropertyChangeSupport propertyChangeSupport = new Prc

  private boolean disposed;

  private String id = java.util.UUID.randomUUID().toString();

  private String name;

  private transient String description;

```

Transient.png: The “transient” keyword allows properties to be excluded from the mapping.

- derived – marks the property to be derived from other values. Derived properties are not persisted; and only a getter is created.



```
package org.lunifera.dtodsl.documentation.demo {

    import org.lunifera.dtodsl.documentation.demo.entities.*;

    datatype String jvmType String;
    datatype long jvmType Long as primitive;

    dto FooDto wraps FooEntity {
        var long number;
        var long count;
        derived long avg {
            var long myvalue;
            myvalue = (number + count) / 2;
            return myvalue;
        }
    }
}

```

```
   *
   * @param count - the property
   * @throws RuntimeException if instance is <code>disposed</code>
   *
   */
  public void setCount(final long count) {
    firePropertyChange("count", this.count, this.count = count );
  }

  /**
   * Calculates the value for the derived property avg
   *
   * @return avg The derived property value
   */
  public long getAvg() {
    long myvalue = 0;
    myvalue = ((this.number + this.count) / 2);
    return myvalue;
  }
}
```

Derived.png: The “derived” keyword marks properties that are based on other values and computed at runtime. The calculation logic is translated to Java. Derived properties are excluded from the mapping.

7 References

The Lunifera DTO DSL tracks relationships between DTOs by using the concept of references. Where appropriate, back references (“opposite”) can be added.

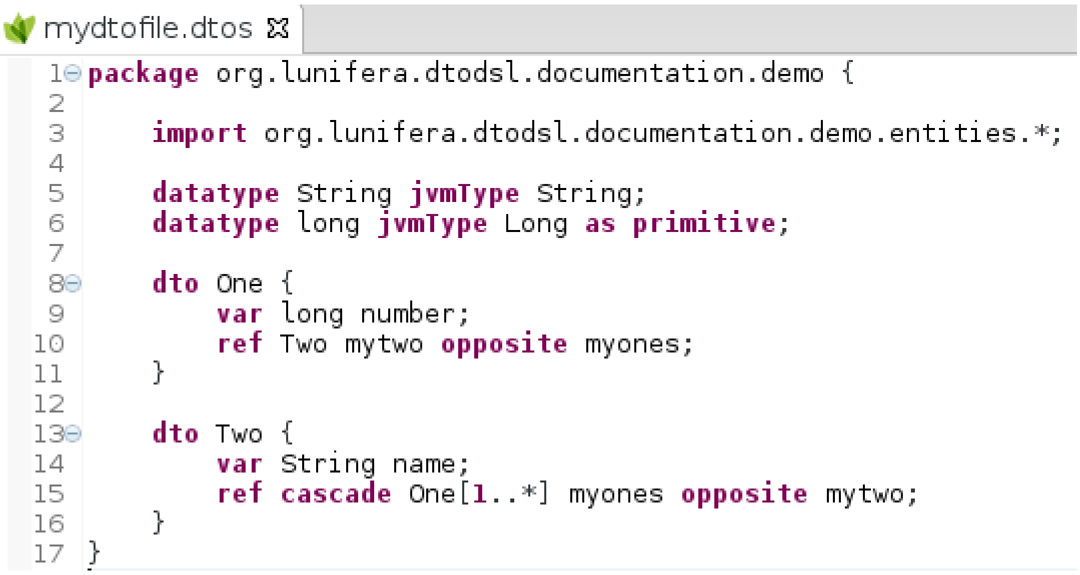
References are defined by the “ref” keyword, a type and a name. The exact type of reference can be specified as follows:

Reference Modifiers

- Multiplicity and nullability: The Lunifera DTO DSL supports the specification of multiplicities in square brackets appended to the type:
  [1] defines a non-nullable one-to-one relationship.
  [0..1] defines a nullable one-to-one relationship (Default behavior).

[1..*] defines a non-nullable one-to-many relationship. Needs an opposite reference.
[0..*] or simply [*] defines a nullable one-to-many relationship. Needs an opposite reference.

- opposite reference: Lifecycle references between entities need the specification of an opposite reference in order to navigate back to the original object after following the reference. The Lunifera DTO DSL includes a mechanism to map such references.
- cascade: In the Lunifera Entity DSL, the “cascade” specifier controls the behaviour of the database on delete operations. In the DTO DSL, these cascading references can be mapped to DTOs by the same keyword.



```
package org.lunifera.dtodsl.documentation.demo {

    import org.lunifera.dtodsl.documentation.demo.entities.*;

    datatype String jvmType String;
    datatype long jvmType Long as primitive;

    dto One {
        var long number;
        ref Two mytwo opposite myones;
    }

    dto Two {
        var String name;
        ref cascade One[1..*] myones opposite mytwo;
    }
}
```

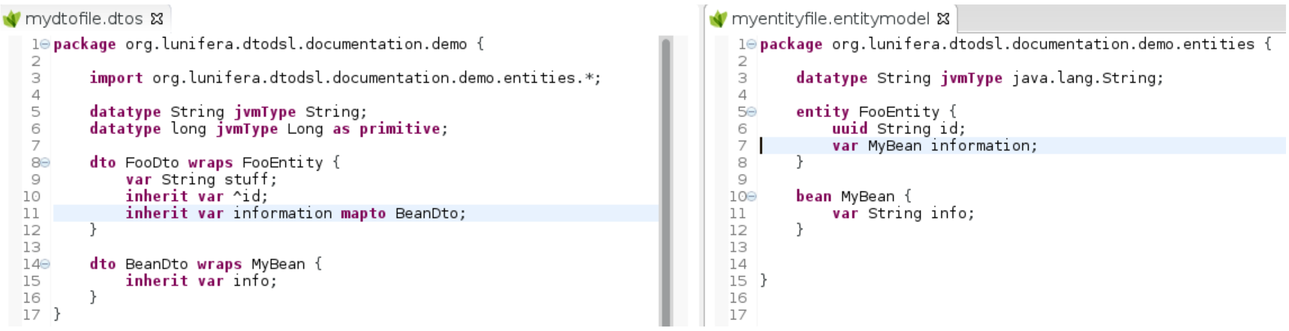
References.png: An example of two DTOs referencing each other. The second DTO maps a cascading non-nullable to-many reference.

8 Inherited features

Inherited properties and references are the core of the “Lunifera DTO DSL”. They provide the mechanism for mapping entities and their features to the respective DTOs and their features. There are two types of inherited features: inherited properties and inherited references.

Inherited features are defined by the “inherited” keyword followed by the feature type and the name of the feature in the entity. After that, the feature in the DTO can be specified using the “mapto” keyword.

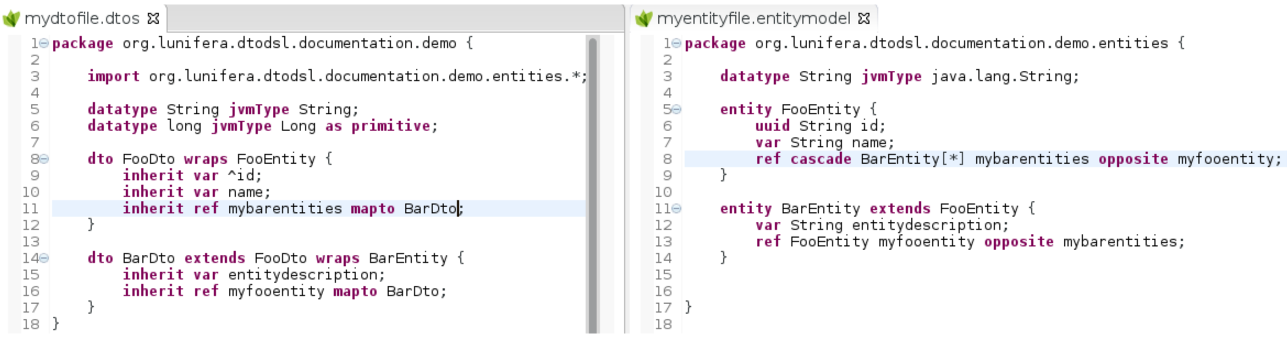
- inherit var: used for mapping entity properties to DTO properties
- inherit ref: used for mapping entity references to DTO references
- mapto: defines the DTO the feature should be mapped to. In the case of “inherit var”, this keyword is used to point to a DTO that maps a bean from the entity model file. In the case of “inherit ref”, a DTO must be specified.



```
package org.lunifera.dtodsl.documentation.demo {

    import org.lunifera.dtodsl.documentation.demo.entities.*;

    datatype String jvmType String;
    datatype long jvmType Long as primitive;

    dto FooDto wraps FooEntity {
        var String stuff;
        inherit var ^id;
        inherit var information mapto BeanDto;
    }

    dto BeanDto wraps MyBean {
        inherit var info;
    }
}
```

```
package org.lunifera.dtodsl.documentation.demo.entities {

    datatype String jvmType java.lang.String;

    entity FooEntity {
        uuid String id;
        var MyBean information;
    }

    bean MyBean {
        var String info;
    }


}
```

Inheritvar.png: The “inherit var” keyword is used to map entity properties to DTO properties. Content assist is available. The “mapto” keyword specifies which DTO an embedded bean is mapped to.



```
package org.lunifera.dtodsl.documentation.demo {

    import org.lunifera.dtodsl.documentation.demo.entities.*;

    datatype String jvmType String;
    datatype long jvmType Long as primitive;

    dto FooDto wraps FooEntity {
        inherit var ^id;
        inherit var name;
        inherit ref mybarentities mapto BarDto;
    }

    dto BarDto extends FooDto wraps BarEntity {
        inherit var entitydescription;
        inherit ref myfooentity mapto BarDto;
    }
}
```

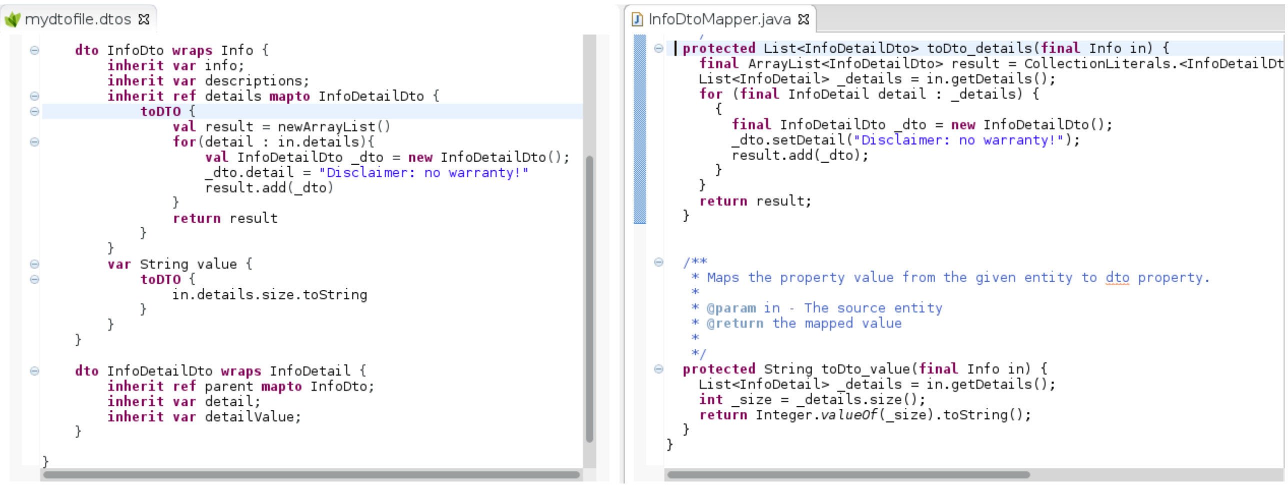
```
package org.lunifera.dtodsl.documentation.demo.entities {

    datatype String jvmType java.lang.String;

    entity FooEntity {
        uuid String id;
        var String name;
        ref cascade BarEntity[*] mybarentities opposite myfooentity;
    }

    entity BarEntity extends FooEntity {
        var String entitydescription;
        ref FooEntity myfooentity opposite mybarentities;
    }


}
```

Inheritref.png: The “inherit ref” keyword allows for mapping entity references to DTO references. Content assist is available, and multiplicities and opposite references are automatically carried over from the entity model file. The “mapto” keyword indicates the DTO that serves as the other end of the reference.

9 Custom mappers

The Lunifera DTO DSL provides a mechanism for creating custom mappers between DTOs and entities. These can be defined after the declaration of a property or reference and control the information exchange for the respective feature.

In order to create a custom mapper, the “toDTO” and “fromDTO” keywords are used, followed by the mapper definition in curly braces. “toDTO” controls the flow of information from entity to DTO, “fromDTO” vice versa. Within the curly braces, the special words “in” and “out” may be used to reference source and destination of the mapping.



```
    dto InfoDto wraps Info {
        inherit var info;
        inherit var descriptions;
        inherit ref details mapto InfoDetailDto {
            toDTO {
                val result = newArrayList()
                for(detail : in.details){
                    val InfoDetailDto _dto = new InfoDetailDto();
                    _dto.detail = "Disclaimer: no warranty!"
                    result.add(_dto)
                }
                return result
            }
        }
        var String value {
            toDTO {
                in.details.size.toString
            }
        }
    }

    dto InfoDetailDto wraps InfoDetail {
        inherit ref parent mapto InfoDto;
        inherit var detail;
        inherit var detailValue;
    }

}
```

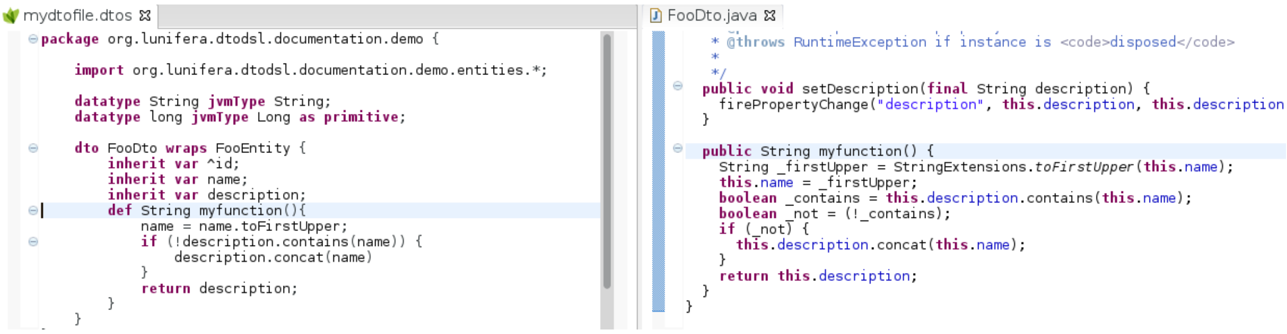
```
  protected List<InfoDetailDto> toDto_details(final Info in) {
    final ArrayList<InfoDetailDto> result = CollectionLiterals.<InfoDetailDt
    List<InfoDetail> _details = in.getDetails();
    for (final InfoDetail detail : _details) {
      {
        final InfoDetailDto _dto = new InfoDetailDto();
        _dto.setDetail("Disclaimer: no warranty!");
        result.add(_dto);
      }
    }
    return result;
  }


  /**
   * Maps the property value from the given entity to dto property.
   *
   * @param in - The source entity
   * @return the mapped value
   *
   */
  protected String toDto_value(final Info in) {
    List<InfoDetail> _details = in.getDetails();
    int _size = _details.size();
    return Integer.valueOf(_size).toString();
  }
}
```

Mapping.png: The “toDTO” section creates a custom mapping from entity to DTO: First a local variable (ArrayList) is created, then an InfoDetail with a disclaimer is added to each entry. The “in” keyword is used to reference the entity for the iteration.

10 Operations

The Lunifera DTO DSL supports the declaration of operations that can be performed within the DTOs. These operations are based on Xbase and offer a huge set of semantic features, extending the featureset of Java. Xbase additionally offers features like closures. When using operations, bear in mind that DTOs are not intended to contain business logic.

Operations can be declared by the “def” keyword.



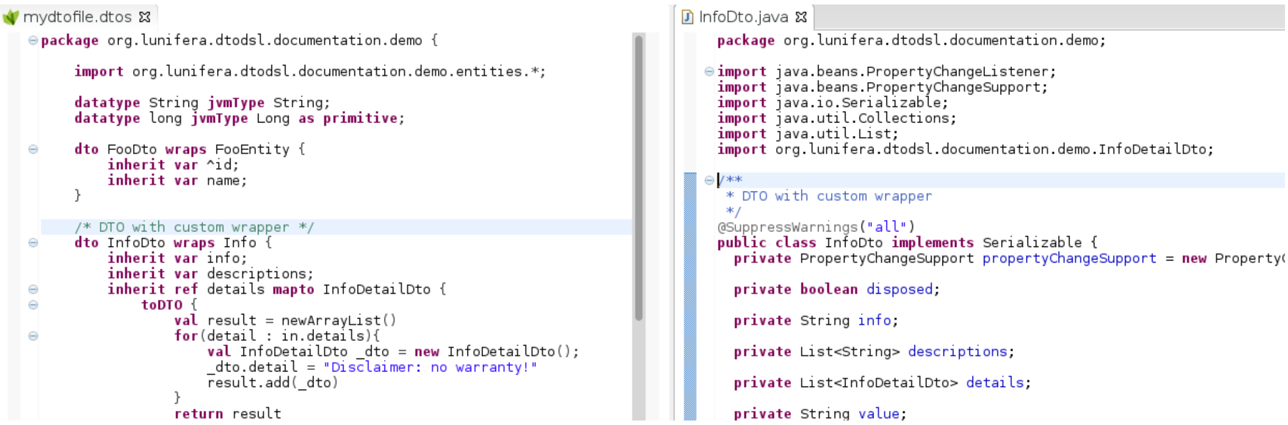
```
package org.lunifera.dtodsl.documentation.demo {

    import org.lunifera.dtodsl.documentation.demo.entities.*;

    datatype String jvmType String;
    datatype long jvmType Long as primitive;

    dto FooDto wraps FooEntity {
        inherit var ^id;
        inherit var name;
        inherit var description;
        def String myfunction(){
            name = name.toFirstUpper;
            if (!description.contains(name)) {
                description.concat(name)
            }
            return description;
        }
    }
```

```
   * @throws RuntimeException if instance is <code>disposed</code>
   *
   */
  public void setDescription(final String description) {
    firePropertyChange("description", this.description, this.description
  }

  public String myfunction() {
    String _firstUpper = StringExtensions.toFirstUpper(this.name);
    this.name = _firstUpper;
    boolean _contains = this.description.contains(this.name);
    boolean _not = (!_contains);
    if (_not) {
      this.description.concat(this.name);
    }
    return this.description;
  }
}
```

Def.png: The “def” keyword allows the inlining of custom operations in the Lunifera DTO DSL code. These methods are translated to Java methods.

11 Comments

Comments can be added anywhere in a DTO model file and are copied over into the generated Java code. Comments are enclosed in /* ... */.

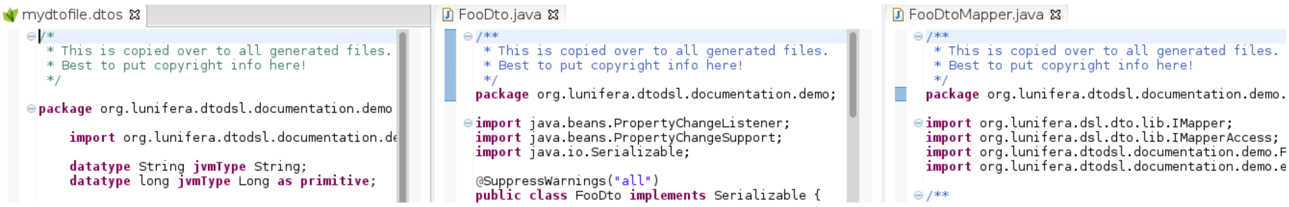


```
package org.lunifera.dtodsl.documentation.demo {

    import org.lunifera.dtodsl.documentation.demo.entities.*;

    datatype String jvmType String;
    datatype long jvmType Long as primitive;

    dto FooDto wraps FooEntity {
        inherit var ^id;
        inherit var name;
    }

    /* DTO with custom wrapper */
    dto InfoDto wraps Info {
        inherit var info;
        inherit var descriptions;
        inherit ref details mapto InfoDetailDto {
            toDTO {
                val result = newArrayList()
                for(detail : in.details){
                    val InfoDetailDto _dto = new InfoDetailDto();
                    _dto.detail = "Disclaimer: no warranty!"
                    result.add(_dto)
                }
                return result
```

```
package org.lunifera.dtodsl.documentation.demo;

import java.beans.PropertyChangeListener;
import java.beans.PropertyChangeSupport;
import java.io.Serializable;
import java.util.Collections;
import java.util.List;
import org.lunifera.dtodsl.documentation.demo.InfoDetailDto;

/**
 * DTO with custom wrapper
 */
@SuppressWarnings("all")
public class InfoDto implements Serializable {
  private PropertyChangeSupport propertyChangeSupport = new Property(

  private boolean disposed;

  private String info;

  private List<String> descriptions;

  private List<InfoDetailDto> details;

  private String value;
```

Comment.png: The comments added in the Entity DSL are transformed to Java-style comments in the generated code.

Comments before the “package” keyword are copied over to all generated Java classes – this is the place for copyright notices.



```
/*
 * This is copied over to all generated files.
 * Best to put copyright info here!
 */

package org.lunifera.dtodsl.documentation.demo

    import org.lunifera.dtodsl.documentation.de

    datatype String jvmType String;
    datatype long jvmType Long as primitive;
```

```
/**
 * This is copied over to all generated files.
 * Best to put copyright info here!
 */
package org.lunifera.dtodsl.documentation.demo;

import java.beans.PropertyChangeListener;
import java.beans.PropertyChangeSupport;
import java.io.Serializable;

@SuppressWarnings("all")
public class FooDto implements Serializable {
```

```
/**
 * This is copied over to all generated files.
 * Best to put copyright info here!
 */
package org.lunifera.dtodsl.documentation.demo.

import org.lunifera.dsl.dto.lib.IMapper;
import org.lunifera.dsl.dto.lib.IMapperAccess;
import org.lunifera.dtodsl.documentation.demo.F
import org.lunifera.dtodsl.documentation.demo.e

/**
```

*Commentcopyright.png: A comment before the “package” keyword ends up in **all** generated Java classes (DTO and mapper classes).*