Java Back-End Web App Tutorial Part 4: Managing Unidirectional Associations

Learn how to manage unidirectional associations between object types, such as the associations assigning publishers and authors to books, using Java with Java Server Faces (JSF) as the user interface technology, the Java Persistence API (JPA) for object-to-storage mapping, and a MySQL database

Gerd Wagner <G.Wagner@b-tu.de>
Mircea Diaconescu <M.Diaconescu@b-tu.de>

Java Back-End Web App Tutorial Part 4: Managing Unidirectional Associations: Learn how to manage unidirectional associations between object types, such as the associations assigning publishers and authors to books, using Java with Java Server Faces (JSF) as the user interface technology, the Java Persistence API (JPA) for object-to-storage mapping, and a MySQL database

by Gerd Wagner and Mircea Diaconescu

Warning: This tutorial may still contain errors and may still be incomplete in certain respects. Please report any issue to Gerd Wagner at G.Wagner@b-tu.de or Mircea Diaconescu at M.Diaconescu@b-tu.de.

This tutorial is also available in the following formats: PDF [unidirectional-association-tutorial.pdf]. See also the project page [http://web-engineering.info], or run the example app [UnidirectionalAssociationApp/index.html] from our server, or download it as a ZIP archive file [UnidirectionalAssociationApp.zip].

Publication date 2018-06-22
Copyright © 2015-2018 Gerd Wagner, Mircea Diaconescu

This tutorial article, along with any associated source code, is licensed under The Code Project Open License (CPOL) [http://www.codeproject.com/info/cpol10.aspx], implying that the associated code is provided "as-is", can be modified to create derivative works, can be redistributed, and can be used in commercial applications, but the article must not be distributed or republished without the authors' consent.

Table of Contents

Foreword vi
1. Reference Properties and Unidirectional Associations 1
    1. References and Reference Properties 2
    2. Referential Integrity 3
    3. Modeling Reference Properties as Unidirectional Associations 4
    4. Representing Unidirectional Associations as Reference Properties 5
    5. Adding Directionality to a Non-Directed Association 5
    6. Our Running Example 7
    7. Eliminating Unidirectional Associations 7
        7.1. The basic elimination procedure restricted to unidirectional associations 7
        7.2. Eliminating associations from the design model 8
    8. Rendering Reference Properties in the User Interface 10
2. Unidirectional Functional Associations in Java EE 11
    1. Implementing Single-Valued Reference Properties in Java 11
    2. Make a Java Entity Class Model 12
    3. New Issues 12
    4. Write the Model Code 13
        4.1. Summary 13
        4.2. Code each class of the Entity class model 13
        4.3. Code the constraints 15
        4.4. Code getters and setters 15
        4.5. Implement a deletion policy 15
        4.6. Serialization and De-Serialization 16
    5. The View and Controller Layers 16
        5.1. Initialize the app 17
        5.2. Show information about associated objects in the *List Objects* use case 17
        5.3. Allow selecting associated objects in the *create* and *update* use cases 18
3. Implementing Unidirectional Non-Functional Associations with Java EE 22
    1. Implementing Multi-Valued Reference Properties in Java 22
    2. Make a Java Entity Class Model 22
    3. New issues 24
    4. Write the Model Code 24
        4.1. Summary 24
        4.2. Code each class of the Java Entity class model 25
        4.3. Implement a deletion policy 26
        4.4. Serialization and De-Serialization 27
    5. Write the User Interface Code 27
        5.1. Show information about associated objects in the *List Objects* use case 27
        5.2. Allow selecting associated objects in the *create* use case 28
    6. Run the App and Get the Code 30
    7. Possible Variations and Extensions 30
        7.1. Set-valued versus ordered-set-valued reference properties 30

List of Figures

1.1. A committee has a club member as chair expressed by the reference property `chair` 2
1.2. An association end with a "dot" 4
1.3. Representing unidirectional associations as reference properties 5
1.4. A model of a non-directed association between Committee and ClubMember 6
1.5. Modeling a bidirectional association between Committee and ClubMember 6
1.6. The Publisher-Book information design model with a unidirectional association 7
1.7. The Publisher-Book-Author information design model with two unidirectional associations 7
1.8. Turning a functional association end into a reference property 8
1.9. Turning a non-functional association end into a multi-valued reference property 8
1.10. An OO class model for `Publisher` and `Book` 9
1.11. An OO class model for all three classes 9
3.1. An example with a set-valued reference property `authoredBooks` and an ordered-set-valued reference property `authors` 30

List of Tables

1.1. An example of an association table 1
1.2. Different terminologies 1
1.3. Functionality types 4
1.4. Sample data for `Publisher` 9
1.5. Sample data for `Book` 9
3.1. Sample data for `Publisher` 23
3.2. Sample data for `Book` 23
3.3. Sample data for `Author` 23

Foreword

This tutorial is Part 4 of our series of six tutorials [http://web-engineering.info/JavaJpaJsfApp] about model-based development of back-end web applications with Java, using JPA and JSF. It shows how to build a web app that takes care of the three object types `Book`, `Publisher` and `Author`, as well as of the two unidirectional associations that assign a publisher and (one or more) authors to a book.

A *distributed web app* is composed of at least two parts: a front-end part, which, at least, renders the user interface (UI) pages, and a back-end part, which, at least, takes care of persistent data storage. A *back-end web app* is a distributed web app where essentially all work is performed by the back-end component, including data validation and UI page creation, while the front-end only consists of a web browser's rendering of HTML-forms-based UI pages. Normally, a distributed web app can be accessed by multiple users, possibly at the same time, over HTTP connections.

In the case of a Java/JPA/JSF back-end app, the back-end part of the app can be executed by a server machine that runs a web server supporting the Java EE specifications *Java Servlets*, *Java Expression Language (EL)*, *JPA* and *JSF*, such as the open source server Tomcat/TomEE [http://tomee.apache.org/apache-tomee.html].

The app supports the four standard data management operations (**C**reate/**R**ead/**U**pdate/**D**elete). It extends the example app of part 2 by adding code for handling the **unidirectional functional** (many-to-one) association between `Book` and `Publisher`, and the **unidirectional non-functional** (many-to-many) association between `Book` and `Author`. The other parts of the tutorial are:

- Part 1 [minimal-tutorial.html]: Building a **minimal** app.
- Part 2 [validation-tutorial.html]: Handling **constraint validation**.
- Part 3 [enumeration-tutorial.html]: Dealing with **enumerations**.
- Part 5 [bidirectional-association-tutorial.html]: Managing **bidirectional associations**, such as the associations between books and publishers and between books and authors, not only assigning authors and a publisher to a book, but also the other way around, assigning books to authors and to publishers.
- Part 6 [subtyping-tutorial.html]: Handling **subtype** (inheritance) relationships between object types.

Chapter 1. Reference Properties and Unidirectional Associations

A property defined for an object type, or class, is called a **reference property** if its values are *references* that reference an object of another, or of the same, type. For instance, the class `Committee` shown in Figure 1.1 below has a reference property `chair`, the values of which are references to objects of type `ClubMember`.

An **association** between object types classifies relationships between objects of those types. For instance, the association *Committee*-**has**-*ClubMember*-**as-chair**, which is visualized as a connection line in the class diagram shown in Figure 1.2 below, classifies the relationships FinanceCommittee-**has**-PeterMiller-**as-chair**, RecruitmentCommittee-**has**-SusanSmith-**as-chair** and AdvisoryCommittee-**has**-SarahAnderson-**as-chair**, where the objects PeterMiller, SusanSmith and SarahAnderson are of type `ClubMember`, and the objects FinanceCommittee, RecruitmentCommittee and AdvisoryCommittee are of type `Committee`.

Reference properties correspond to a special form of associations, namely to ***unidirectional*** *binary associations*. While a binary association does, in general, not need to be directional, a reference property represents a binary association that is directed from the property's domain class (where it is defined) to its range class.

In general, associations are **relationship types** with two or more **object types** participating in them. An association between two object types is called **binary**. In this tutorial we only discuss binary associations. For simplicity, we just say 'association' when we actually mean 'binary association'.

Table 1.1. An example of an association table

***Committee*-has-*ClubMember*-as-chair**	
Finance Committee	Peter Miller
Recruitment Committee	Susan Smith
Advisory Committee	Sarah Anderson

While individual relationships (such as FinanceCommittee-**has**-PeterMiller-**as-chair**) are important information items in business communication and in information systems, associations (such as *Committee*-**has**-*ClubMember*-**as-chair**) are important elements of *information models*. Consequently, software applications have to implement them in a proper way, typically as part of their *model* layer within a *model-view-controller* (MVC) architecture. Unfortunately, many application development frameworks lack the required support for dealing with associations.

In mathematics, associations have been formalized in an abstract way as sets of uniform tuples, called *relations*. In *Entity-Relationship (ER)* modeling, which is the classical information modeling approach in information systems and software engineering, objects are called *entities*, and associations are called *relationship types*. The *Unified Modeling Language (UML)* includes the *UML Class Diagram* language for information modeling. In UML, object types are called *classes*, relationship types are called *associations*, and individual relationships are called "links". These three terminologies are summarized in the following table:

Table 1.2. Different terminologies

Our preferred term(s)	UML	ER Diagrams	Mathematics
object	object	entity	individual

Our preferred term(s)	UML	ER Diagrams	Mathematics
object type (class)	class	entity type	unary relation
relationship	link	relationship	tuple
association (relationship type)	association	relationship type	relation
functional association		one-to-one, many-to-one or one-to-many relationship type	function

We first discuss reference properties, which implicitly represent unidirectional binary associations in an "association-free" class model (a model without any explicit association element).

1. References and Reference Properties

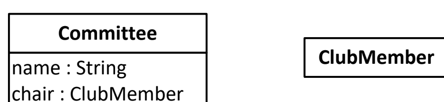
A reference can be either *human-readable* or an *internal object reference*. Human-readable references refer to identifiers that are used in human communication, such as the unique names of astronomical bodies, the ISBN of books and the employee numbers of the employees of a company. Internal object references refer to the computer memory addresses of OOP objects, thus providing an efficient mechanism for accessing objects in the main memory of a computer.

Some languages, like SQL and XML, only support human-readable, but not internal references. In SQL, human-readable references are called *foreign keys*, and the identifiers they refer to are called *primary keys*. In XML, human-readable references are called *ID references* and the corresponding attribute type is `IDREF`.

Objects in an OO program can be referenced either with the help of human-readable references (such as integer codes) or with internal object references, which are preferable for accessing objects efficiently in main memory. Following the XML terminology, we call human-readable references *ID references*. We follow the standard naming convention for ID reference properties where an ID reference property defined in a class `A` and referencing objects of class `B` has the name `b_id` using the suffix `_id`. When we store persistent objects in the form of records or table rows, we need to convert internal object references, stored in properties like `publisher`, to ID references, stored in properties like `publisher_id`. This conversion is performed as part of the serialization of the object by assigning the standard identifier value of the referenced object to the ID reference property of the referencing object.

In OO languages, a property is defined for an object type, or class, which is its *domain*. The values of a property are either *data values* from some datatype, in which case the property is called an **attribute**, or they are *object references* referencing an object from some class, in which case the property is called a **reference property**. For instance, the class `Committee` shown in Figure Figure 1.1 below has an attribute `name` with range `String`, and a reference property `chair` with range `ClubMember`.

Figure 1.1. A committee has a club member as chair expressed by the reference property `chair`



Object-oriented programming languages, such as JavaScript, PHP, Java and C#, directly support the concept of *reference properties*, which are properties whose range is not a *datatype* but a *reference type*, or *class*, and whose values are object references to instances of that class.

By default, the multiplicity of a property is `1`, which means that the property is **mandatory** and **functional** (or, in other words, *single-valued*), having **exactly one** value, like the property `chair` in class `Committee` shown in Figure 1.1. When a functional property is **optional** (not mandatory), it has the multiplicity `0..1`, which means that the property's minimum cardinality is 0 and its maximum cardinality is 1.

A reference property can be either **single-valued** (*functional*) or **multi-valued** (*non-functional*). For instance, the reference property `Committee::chair` shown in Figure 1.1 is single-valued, since it assigns a unique club member as chair to a club. An example of a *multi-valued* reference property is provided by the property `Book::authors` shown in Figure 1.11 below.

Normally, the collection value of a multi-valued reference property is a set of references, implying that the order of the references does not matter. In certain cases, however, the order matters and, consequently, the collection value of such a multi-valued reference property is an ordered set of references, typically implemented as a list.

2. Referential Integrity

References are important information items in our application's database. However, they are only meaningful, when their *referential integrity* is maintained by the app. This requires that for any reference, there is a referenced object in the database. Consequently, any reference property `p` with domain class `C` and range class `D` comes with a *referential integrity constraint* that has to be checked whenever

1. a new object of type `C` is created,
2. the value of `p` is changed for some object of type `C`,
3. an object of type `D` is destroyed.

A referential integrity constraint also implies two *change dependencies*:

1. An **object creation dependency**: an object with a reference to another object can only be created after the referenced object has been created.
2. An **object destruction dependency**: an object that is referenced by another object can only be destroyed after
   a. the referencing object is destroyed first (the CASCADE deletion policy), or
   b. the reference in the referencing object is either dropped (the DROP-REFERENCE deletion policy) or replaced by another reference.

   For every reference property in our app's model classes, we have to choose, which of these two possible *deletion policies* applies.

In certain cases, we may want to relax this strict regime and allow creating objects that have non-referencing values for an ID reference property, but we do not consider such cases.

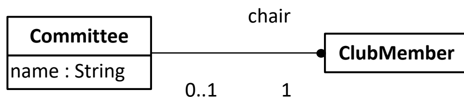
Typically, object creation dependencies are managed in the user interface by not allowing the user to enter a value of an ID reference property, but only to select one from a list of all existing target objects.

3. Modeling Reference Properties as Unidirectional Associations

A reference property (such as `chair` in the example shown in Figure 1.1 above) can be modeled in a UML class diagram in the form of an **association end** owned by its domain class, which is visualized with the help of a small filled circle (also called a "dot"). This requires to connect the domain class and the range class of the reference property with an association line, place an *ownership dot* at the end of this line at the range class side, and annotate this association end with the property name and with a multiplicity symbol, as shown in Figure 1.2 below for the case of our example. In this way we get a **unidirectional association**, the **source** class of which is the property's **domain** and the **target** class of which is the property's **range**.

The fact that an association end is *owned* by the class at the other end, as visually expressed by the *association end ownership dot* at the association end `chair` in the example shown in Figure 1.2 below, implies that the association end represents a reference property. In the example of Figure 1.2, the represented reference property is `Committee::chair` having `ClubMember` as range. Such an association, with only one association end ownership dot, is *unidirectional* in the sense that it allows \`navigation´ (object access) in one direction only: from the class at the opposite side of the dot (the *source* class) to the class where the dot is placed (the *target* class).

Figure 1.2. An association end with a "dot"



Thus, the two diagrams shown in Figure 1.1 and Figure 1.2 express essentially equivalent models. When a reference property, like `chair` in Figure 1.1, is modeled by an association end with a "dot", then the property's multiplicity is attached to the association end. Since in a design model, all association ends need to have a multiplicity, we also have to define a multiplicity for the other end at the side of the `Committee` class, which represents the inverse of the property. This multiplicity (of the inverse property) is not available in the original property description in the model shown in Figure 1.1, so it has to be added according to the intended semantics of the association. It can be obtained by answering the question "is it mandatory that any `ClubMember` is the `chair` of a `Committee`?" for finding the minimum cardinality and the question "can a `ClubMember` be the `chair` of more than one `Committee`?" for finding the maximum cardinality.

When the value of a property is a set of values from its range, the property is **non-functional** and its multiplicity is either `0..*` or `n..*` where n > 0. Instead of `0..*`, which means "neither mandatory nor functional", we can simply write the asterisk symbol `*`. The association shown in Figure 1.2 assigns at most one object of type `ClubMember` as chair to an object of type `Committee`. Consequently, it's an example of a **functional association**.

The following table provides an overview about the different cases of functionality of an association:

Table 1.3. Functionality types

Functionality type	Meaning
one-to-one	both functional and inverse functional
many-to-one	functional

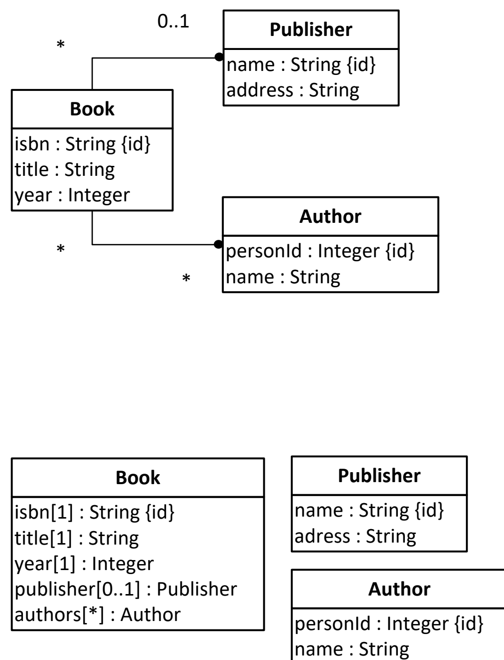
Functionality type	Meaning
one-to-many	inverse functional
many-to-many	neither functional nor inverse functional

Notice that the directionality and the functionality type of an association are independent of each other. So, a unidirectional association can be either functional (one-to-one or many-to-one), or non-functional (one-to-many or many-to-many).

4. Representing Unidirectional Associations as Reference Properties

A unidirectional association between a source and a target class can be represented as a reference property of the source class. This is illustrated in Figure 1.3 below for two unidirectional associations: a many-to-one and a many-to-many association.

Figure 1.3. Representing unidirectional associations as reference properties



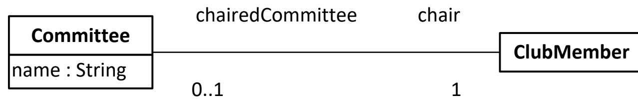
Notice that, in a way, we have eliminated the two explicit associations and replaced them with corresponding reference properties resulting in a class model that can be coded with a classical OOP language in a straightforward way. OOP languages do not support associations as first class citizens. They do not have a language element for defining associations. Consequently, an OOP class design model, which we call **OO class model**, must not contain any explicit association.

5. Adding Directionality to a Non-Directed Association

When we make an information model in the form of a UML class diagram, we typically end up with a model containing one or more associations that do not have any ownership defined for their ends, as, for instance, in Figure 1.4 below. When there is no ownership dot at either end of an association, such

as in this example, this means that the model does not specify how the association is to be represented (or realized) with the help of reference properties. Such an association does not have any direction. According to the UML 2.5 specification, the ends of such an association are "owned" by itself, and not by any of the classes participating in it.

Figure 1.4. A model of a non-directed association between Committee and ClubMember



An information design model that contains an association without association end ownership dots is acceptable as a *relational database* design model, but it is incomplete as a design model for OOP languages.

For instance, the model of Figure 1.4 provides a relational database design with two entity tables, `committees` and `clubmembers`, and a separate one-to-one relationship table `committee_has_clubmember_as_chair`. But it does not provide a design for Java classes, since it does not specify how the association is to be implemented with the help of reference properties.

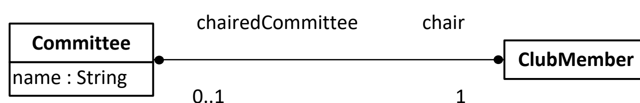
There are three options how to turn an information design model of a non-directed association (without any association end ownership dots) into an information design model where all associations are either unidirectional or bidirectional: we can place an ownership dot at either end or at both ends of the association. Each of these three options defines a different way how to represent, or implement, the association with the help of reference properties. So, for the association shown in Figure 1.4 above, we have the following options:

1. Place an ownership dot at the `chair` association end, leading to the model shown in Figure 1.2 above, which can be transformed into the OO class model shown in Figure 1.1 above.

2. Place an ownership dot at the `chairedCommittee` association end, leading to the completed models shown in Figure 1.3 above.

3. Make the association bidirectional by placing ownership dots at both association ends with the meaning that the association is implemented in a redundant manner by a pair of mutually inverse reference properties `Committee::chair` and `ClubMember::chairedCommittee`, as discussed in the next part of our tutorial.

Figure 1.5. Modeling a bidirectional association between Committee and ClubMember



So, whenever we have modeled an association, we have to make a choice, which of its ends represents a reference property and will therefore be marked with an ownership dot. It can be either one, or both. This decision also implies a decision about the *navigability* of the association. When an association end represents a reference property, this implies that it is navigable (via this property).

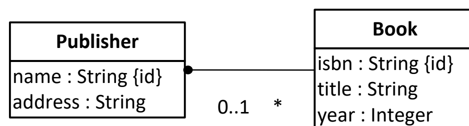
In the case of a functional association that is not one-to-one, the simplest design is obtained by defining the direction of the association according to its functionality, placing the association end ownership dot

at the association end with the multiplicity `0..1` or `1`. For a non-directed one-to-one or many-to-many association, we can choose the direction as we like, that is, we can place the ownership dot at either association end.

6. Our Running Example

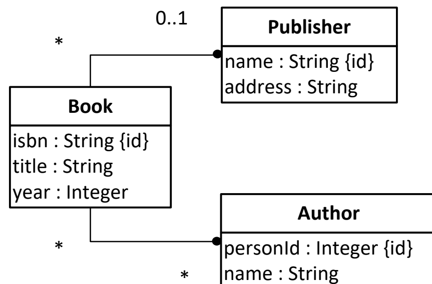
The model shown in Figure 1.6 below (about publishers and books) serves as our running example for a unidirectional functional association in this tutorial. Notice that it contains the unidirectional many-to-one association *Book*-**has**-*Publisher*.

Figure 1.6. The Publisher-Book information design model with a unidirectional association



We may also have to deal with a non-functional (multi-valued) reference property representing a unidirectional non-functional association. For instance, the unidirectional many-to-many association between `Book` and `Author` shown in Figure 1.7 below, models a multi-valued (non-functional) reference property `authors`.

Figure 1.7. The Publisher-Book-Author information design model with two unidirectional associations



7. Eliminating Unidirectional Associations

Since classical OO programming languages do not support associations as first class citizens, but only classes and reference properties representing unidirectional associations, we have to eliminate all explicit associations from general information design models for obtaining OO class models.

7.1. The basic elimination procedure restricted to unidirectional associations

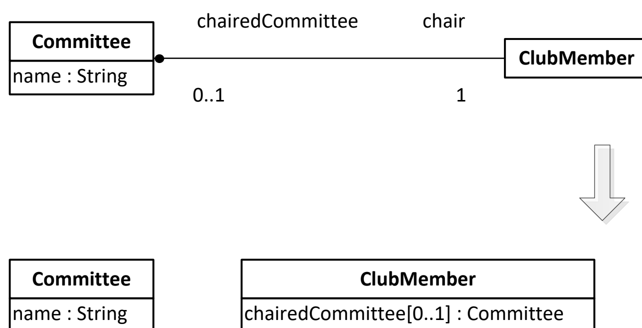
The starting point of our restricted **association elimination** procedure is an information design model with various kinds of unidirectional associations, such as the model shown in Figure 1.6 above. If the model still contains any non-directional associations, we first have to turn them into directional ones by making a decision on the ownership of their ends, as discussed in Section 5.

A unidirectional association connecting a source with a target class is replaced with a corresponding reference property in its source class having

1. the same name as the association end, if there is any, otherwise it is set to the name of the target class (possibly pluralized, if the reference property is multi-valued);
2. the target class as its range;
3. the same multiplicity as the target association end,
4. a uniqueness constraint if the unidirectional association is inverse functional.

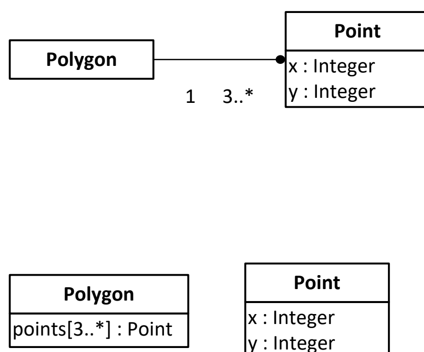
This replacement procedure is illustrated for the case of a unidirectional one-to-one association in Figure 1.3 below.

Figure 1.8. Turning a functional association end into a reference property



For the case of a unidirectional one-to-many association, Figure 1.9 below provides an illustration of the association elimination procedure. Here, the non-functional association end at the target class `Point` is turned into a corresponding reference property with name `points` obtained as the pluralized form of the target class name.

Figure 1.9. Turning a non-functional association end into a multi-valued reference property

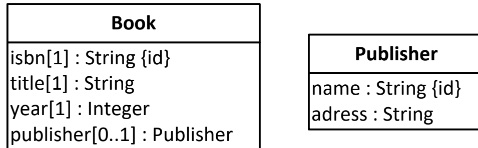


7.2. Eliminating associations from the design model

In the case of our running example, the Publisher-Book-Author information design model, we have to replace both unidirectional associations with suitable reference properties. In the first step, we replace

the many-to-one association *Book*-**has**-*Publisher* in the model of Figure 1.6 with a functional reference property `publisher` in the class `Book`, resulting in the following OO class model:

Figure 1.10. An OO class model for `Publisher` and `Book`



Notice that since the target association end of the *Book*-**has**-*Publisher* association has the multiplicity `0..1`, we have to declare the new property `publisher` as optional by defining its multiplicity to be `0..1`.

The meaning of this OO class model and its reference property `publisher` can be illustrated by a sample data population for the two model classes `Book` and `Publisher` as presented in the following tables:

Table 1.4. Sample data for `Publisher`

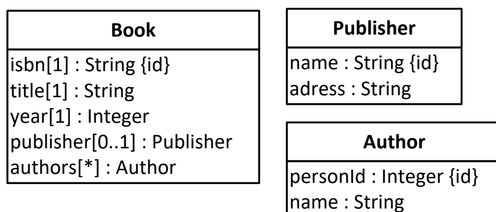
Name	Address
Bantam Books	New York, USA
Basic Books	New York, USA

Table 1.5. Sample data for `Book`

ISBN	Title	Year	Publisher
0553345842	The Mind's I	1982	Bantam Books
1463794762	The Critique of Pure Reason	2011	
1928565379	The Critique of Practical Reason	2009	
0465030793	I Am A Strange Loop	2000	Basic Books

In the second step, we replace the many-to-many association *Book*-**has**-*Author* in the model of Figure 1.7 with a multi-valued reference property `authors` in the class `Book`, resulting in the following OO class model:

Figure 1.11. An OO class model for all three classes



After the platform-independent OO class model has been completed, one or more platform-specific data models, for a choice of specific implementation platforms, can be derived from it. Such a platform-specific data model can still be expressed in the form of a UML class diagram, but it contains only

modeling elements that can be directly coded in the chosen platform. Thus, for any platform considered, two guidelines are needed: 1) how to make the platform-specific data model, and 2) how to code this model.

8. Rendering Reference Properties in the User Interface

The widgets used for data input and output in a (CRUD) data management user interface (UI) normally correspond to properties defined in a model class of an app. We have to distinguish between (various types of) *input fields* corresponding to (various kinds of) *attributes*, and *choice widgets* (such as *selection lists*) corresponding to *enumeration attributes* or to *reference properties*. Representing reference properties in the UI with `select` controls, instead of `input` fields, prevents the user from entering invalid ID references, so it takes care of *referential integrity*.

In general, a **single-valued reference property** can be rendered as a single-selection list in the UI, no matter how many objects populate the reference property's range, from which one specific choice is to be made. If the cardinality of the reference property's range is sufficiently small (say, not greater than 7), then we can also use a *radio button group* instead of a selection list.

A **multivalued reference property** can be rendered as a multiple-selection list in the UI. However, the corresponding multiple-`select` control of HTML is not really usable as soon as there are many (say, more than 20) different options to choose from because the way it renders the choice is visually too scattered. In the special case of having only a few (say, no more than 7) options, we can also use a checkbox group instead of a multiple-selection list. But for the general case of having in the UI a list containing all associated objects chosen from the reference property's range class, we need to develop a special UI widget that allows to add (and remove) objects to (and from) a list of chosen objects.

Such a ***multiple-choice widget*** consists of

1. an HTML list element containing the chosen (associated) objects, where each list item contains a push button for removing the object from the choice;
2. a single-`select` control that, in combination with a push button, allows to add a new associated object from the range of the multi-valued reference property.

Chapter 2. Implementing Unidirectional Functional Associations with Java EE

The three example apps that we have discussed in previous chapters, the *minimal app*, *validation app* and *enumeration app*, have been limited to managing the data of one object type only. A real app, however, has to manage the data of several object types, which are typically related to each other in various ways. In particular, there may be **associations** and **subtype** (inheritance) relationships between object types. Handling associations and subtype relationships are advanced issues in software application engineering. They are often not sufficiently discussed in software development text books and not well supported by application development frameworks. In this part of the tutorial, we show how to deal with unidirectional associations, while bidirectional associations and subtype relationships are covered in parts 5 and 6.

We adopt the approach of **model-based development**, which provides a general methodology for engineering all kinds of artifacts, including data management apps. For being able to understand this tutorial, you need to understand the underlying concepts and theory. Either you first read the theory chapter on reference properties and associations, before you continue to read this tutorial chapter, or you start reading this tutorial chapter and consult the theory chapter only on demand, e.g., when you stumble upon a term that you don't know.

A unidirectional *functional* association is either one-to-one or many-to-one. In both cases such an association is represented, or implemented, with the help of a *single-valued* reference property.

In this chapter of our tutorial, we show

1. how to derive a *data model* in the form of a Java **Entity class model** from an information design model with single-valued reference properties representing *unidirectional functional associations*,
2. how to code the Java Entity class model in the form of entity classes (representing *model classes*),
3. how to write the view and controller code based on the entity classes.

1. Implementing Single-Valued Reference Properties in Java

A single-valued reference property, such as the property `publisher` of the object type `Book`, allows storing internal references to objects of another type, such as `Publisher`. When creating a new object, the constructor function needs to have a parameter for allowing to assign a suitable value to the reference property. In a typed programming language, such as Java, we may have to take a decision if this value is expected to be an internal object reference or an ID reference. Using the JPA object-to-storage mapping technology, we can work with object references and leave the mapping to corresponding ID references to JPA. The `Book` class is extended as follows:

```
@Entity @Table( name="books")
@ViewScoped @ManagedBean( name="book")
public class Book {
  ...
```

```
  private Publisher publisher;

  public Book() {}
  public Book( String isbn, String title, Integer year,
      Publisher publisher) {...}

  public Publisher getPublisher() {...}
  public void setPublisher( Publisher publisher) {...}
  ...
}
```

2. Make a Java Entity Class Model

The starting point for making a Java Entity class model is an OO class model like the one shown in Figure 1.10.

As explained in the previous parts of this tutorial, we obtain the Java Entity class `Publisher` from the corresponding information design class by (1) making all properties private, (2) using Java datatype classes (String, Integer, etc.), (3) adding the stereotype «get/set» to properties (specifying public getters and setters), (4) adding a `toString` function, (5) adding the (class-level) data storage methods `create`, `retrieve`, `update` and `delete`:

«Entity» **Publisher**
«get/set» -name : String {id}
«get/set» -adress : String
+checkNameAsId(in n : String, in em : EntityManager) : ConstraintViolation
+toString() : String
+retrieveAll(in em : EntityManager) : List<Publisher>
+retrieve(in em : EntityManager, in name : String) : Publisher
+create(in em : EntityManager, in name : String, in address : String)
+update(in em : EntityManager, in name : String, in address : String)
+delete(in em : EntityManager, in name : String)

In a Java Entity class model, for any association end dot in the OO class model, we show the reference property representing the association end and annotate it with the functionality type of the unidirectional association represented by it. In our example, we add a reference property `publisher` in the entity class `Book` and annotate it with the association's functionality type `manyToOne`:

«Entity» **Book**
«get/set» -isbn[1] : String {id}
«get/set» -title[1] : String
«get/set» -year[1] : Integer
«get/set» -publisher[0..1] : Publisher {manyToOne}
+checkIsbnAsId(in isbn : String, in em : EntityManager) : ConstraintViolation
+toString() : string
+...()

3. New Issues

Compared to the single-class apps discussed in the previous parts of this tutorial, we have to deal with a number of new technical issues:

1. In the *model* code we now have to take care of **reference properties** that require

   a. defining ***referential integrity constraints*** with the help of suitable JPA annotations;

   b. extending the code of the `create`, `update` and `delete` methods by taking the reference properties into consideration;

   c. a method for converting between (internal) object references and corresponding (external) ID references in serialization and de-serialization operations.

2. In the *user interface* (view) code we now have to take care of

   a. showing information about associated objects in the *retrieve/list all* use case;

   b. allowing to select an associated object from a list of all existing instances of the target class in the *create object* and *update object* use cases.

4. Write the Model Code

The Java Entity class model can be directly coded for getting the model layer code of our Java back-end app.

4.1. Summary

1. Code each class from the Java Entity class model as a corresponding entity class.

2. Code an {id} property modifier with the JPA annotation `@Id`

3. Code any property modifier denoting the functionality type of a reference property, such as {manyToOne}, with the corresponding JPA annotation, such as `@ManyToOne`.

4. Code the integrity constraints specified in the model with the help of Java Bean Validation annotations (or custom validation annotations).

5. Code the getters and setters.

6. Code the `create`, `retrieve`, `update` and `delete` storage management operations as class-level methods.

These steps are discussed in more detail in the following sections.

4.2. Code each class of the Entity class model

Each class `C` of the Java Entity class model is coded as an annotated bean class with name `C` having a default constructor (with no parameters) and a constructor with entity creation parameters.

For instance, the `Book` class from the Entity class model is coded in the following way:

```
@Entity @Table( name="books")
@ViewScoped @ManagedBean( name="book")
public class Book {
  @Id @NotNull( message="An ISBN is required!")
  private String isbn;
```

```
  @Column( nullable=false)
  @NotNull( message="A title is required!")
  private String title;
  @Column( nullable=false)
  @NotNull( message="A year is required!")
  private Integer year;
  @ManyToOne( fetch=FetchType.EAGER)
  private Publisher publisher;

  public Book() {}
  public Book( String isbn, String title,
      Integer year, Publisher publisher) {
    this.setIsbn( isbn);
    this.setTitle( title);
    this.setYear( year);
    this.setPublisher( publisher);
  }
  ...  // getters, setters, etc.
}
```

The `@ManyToOne` annotation on the property `publisher` is used for specifying the functionality type of the association *Book has Publisher* represented by the reference property `publisher` since it holds that *a book has one publisher* and *a publisher has many books*. This annotation also allows to specify a *fetch type* with the parameter `fetch` taking one of the following two possible values:

- `FetchType.EAGER`, implying that a retrieval of an entity includes a retrieval of the associated entity referenced by the entity with this property. In our example, this means that when a `Book` entity is retrieved, the `Publisher` entity referenced by the book's `publisher` property is also retrieved. This behavior is very useful and it should be used whenever the data to be retrieved can be handled in main memory.

- `FetchType.LAZY`, implying that referenced entities are not automatically retrieved when a referencing entity is retrieved. In our example, this means that the referenced `Publisher` entity is not retrieved together with a referencing `Book` entity, leaving the value of the `publisher` property set to `null`. With this fetching behavior, a referenced entity has to be retrieved separately by invoking the reference property's getter in the context of a transaction (in our example by invoking `getPublisher`).

In the case of a *single-valued* reference property (representing a *functional* association) annotated with either `@OneToOne` or `@ManyToOne`, the default value is `FetchType.EAGER`, so referenced entities are fetched together with referencing entities, while in the case of a *non-functional* association (with either `@OneToMany` or `@ManyToMany`), the default value is `FetchType.LAZY`, so referenced entities are not fetched together with referencing entities, but have to be retrieved separately, if needed.

As a result of these JPA annotations, the following SQL table creation statement for creating the `books` table is generated:

```
CREATE TABLE IF NOT EXISTS `books` (
  `ISBN` varchar(10) NOT NULL,
  `TITLE` varchar(255) NOT NULL,
  `YEAR` int(11) NOT NULL,
  `PUBLISHER_NAME` varchar(255) DEFAULT NULL
      FOREIGN  KEY (`PUBLISHER_NAME`) REFERENCES `publishers` (`NAME`)
);
```

4.3. Code the constraints

Take care that all property constraints specified in the entity class model are properly coded by using suitable Bean Validation annotations, as explained in Part 2 (Validation Tutorial [validation-tutorial.html]). For instance, for the `name` attribute, we have to use the JPA annotation `@Id` for specifying that the attribute corresponds to the primary key column of the database table to which the entity class is mapped, and the Bean Validation annotation `@NotNull` for defining a mandatory value constraint that is checked before an entity is saved to the database.

In the case of the `address` attribute, we have to define a mandatory value constraint in two forms: with the JPA annotation `@Column( nullable=false)` for the corresponding table column, and with the Bean Validation annotation `@NotNull` for the attribute.

```
@Id @NotNull( message="A name is required!")
private String name;

@Column( nullable=false)
@NotNull( message="An address is required!")
private String address;
```

Notice that, unfortunately, the current Java EE technology requires defining the same constraint twice, once for the database in the form of a JPA annotation, and once for the Java app in the form of a Bean Validation annotation.

4.4. Code getters and setters

Code the setter operations as (instance-level) methods. The setters only assign values and do not perform any validation, since the property constraints are only checked before save by the Java EE/JPA execution environment. The getters simply return the actual values of properties.

4.5. Implement a deletion policy

For any reference property, we have to choose and implement a deletion policy for taking care of the corresponding object destruction dependency in the `delete` method of the reference property's range class. In our case, we have to choose between

1. deleting all books published by the deleted publisher;

2. dropping from all books published by the deleted publisher the reference to the deleted publisher.

We choose the second policy, which can only be used inf the case of an optional reference property such as `book.publisher`. This is shown in the following code of the `Publisher.delete` method where for all `book` entities concerned the property `book.publisher` is cleared:

```
public static void delete( EntityManager em, UserTransaction ut,
    String name) throw Exception {
  ut.begin();
  Publisher publisher = em.find( Publisher.class, name);
  // find all Books which have this publisher
  Query query = em.createQuery(
      "SELECT b FROM Book b WHERE b.publisher.name = :name");
  query.setParameter( "name", name);
  List<Book> books = query.getResultList();
```

```
  // clear these books' publisher reference
  for ( Book b: books) { b.setPublisher( null);}
  em.remove( publisher);
  ut.commit();
}
```

The method loops through all `Book` entities referencing the publisher to be destroyed and sets their `publisher` property to `null`.

4.6. Serialization and De-Serialization

Based on JPA annotations, together with suitable converter classes when needed, serialization (from Java objects to table rows) as well as the corresponding de-serialization (from table columns to Java objects) are performed automatically.

5. The View and Controller Layers

The user interface (UI) consists of a start page for navigating to the data management UI pages and one UI page for each object type and data management use case. All these UI pages are defined in the form of JSF view files in subfolders of the `WebContent/views/` folder. We create the *Main Menu* page `index.xhtml` in the subfolder `WebContent/views/app` with the following content:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="..." xmlns:h="..." xmlns:f="...">
  <ui:composition template="/WEB-INF/templates/page.xhtml">
    <ui:define name="content">
      <h2>Public Library</h2>
      <h:button value="Manage publishers"
          outcome="/views/publishers/index" />
      <h:button value="Manage books" outcome="/views/books/index" />
      <h:form>
        <h:commandButton value="Clear database"
            action="#{appCtrl.clearData()}" />
        <h:commandButton value="Create test data"
            action="#{appCtrl.createTestData()}" />
      </h:form>
    </ui:define>
  </ui:composition>
</html>
```

It creates the menu buttons which provide the redirects to corresponding views for each of the management pages. Additionally, we need to create a corresponding AppController class which is responsible for the creation and deletion of the test data. The controller is used to combine code of the `Publisher.createTestData` and `Publisher.clearData` methods with `Book.createTestData` and `Book.clearData` methods as follows:

```
@ManagedBean( name="appCtrl")
@SessionScoped
public class AppController {
  @PersistenceContext( unitName="UnidirAssocApp")
  private EntityManager em;
```

```
  @Resource()
  UserTransaction ut;

  public String clearData() {
    try {
      Book.clearData( em, ut);
      Publisher.clearData( em, ut);
    } catch ( Exception e) {
      e.printStackTrace();
    }
    return "index";
  }
  public String createTestData() {
    try {
      Publisher.createTestData( em, ut);
      Book.createTestData( em, ut);
    } catch ( Exception e) {
      e.printStackTrace();
    }
    return "index";
  }
}
```

The deletion of `Book` and `Publisher` data must be done in a particular order for avoiding referential integrity violations (books have to be deleted first, before their publishers are deleted).

5.1. Initialize the app

Since the code runs in a Tomcat container, the initialization is made internally by the container.

5.2. Show information about associated objects in the *List Objects* use case

In our example we have only one reference property, `Book::publisher`, which is functional. For showing information about the optional publisher of a book in the *list books* use case, the corresponding cell in the HTML table is filled with the name of the publisher, if there is any:

```
<ui:composition template="/WEB-INF/templates/page.xhtml">
  <ui:define name="content">
    <h:dataTable value="#{bookCtrl.books}" var="b">
      ...
      <h:column>
        <f:facet name="header">Publisher</f:facet>
        #{b.publisher.name}
      </h:column>
    </h:dataTable>
    <h:button value="Main menu" outcome="index" />
  </ui:define>
</ui:composition>
```

Notice the cascade call used in the `#{b.publisher.name}` JSF expression: accessing the property `name` of the `publisher` which is a property of the book instance `b`.

5.3. Allow selecting associated objects in the *create* and *update* use cases

In this section, we discuss how to create and update an object which has reference properties, such as a `Book` object with the reference property `publisher`.

5.3.1. Create the Object-to-String converter

HTML require string or number values to populate forms. In our case, the property `publisher` of the `Book` class is an object reference. JSF allows the mapping from object and string and back by using *face converters*, which are classes annotated with `@FacesConverter` and implementing the `javax.faces.convert.Converter` interface:

```
@FacesConverter( value="pl.m.converter.PublisherConverter")
public class PublisherConverter implements Converter {
  @Override
  public Object getAsObject( FacesContext context,
      UIComponent component, String value) {
    PublisherController ac = FacesContext
        .getCurrentInstance()
        .getApplication()
        .evaluateExpressionGet( context, "#{publisherCtrl}",
            PublisherController.class);
    EntityManager em = ac.getEntityManager();
    if (value == null) return null;
    else return em.find( Publisher.class, value);
  }
  @Override
  public String getAsString( FacesContext context,
      UIComponent component, Object value) {
    if (value == null) return null;
    else if (value instanceof Publisher) {
      return ((Publisher) value).getName();
    }
    else return null;
  }
}
```

It defines two methods, `getAsObject` and `getAsString`, responsible for the two mappings. It allows to define custom representations which can be used not only in connection with HTML forms but also allows serialization which can be used with display views. For being able to extract an object from database we need an `EntityManager` which can be obtained from our controller class instances (e.g., the managed bean `publisherCtrl`). The instance of the controller can be obtained by using the `FacesContext` singletone as shown above:

```
PublisherController ac = FacesContext.getCurrentInstance()
     .getApplication()
     .evaluateExpressionGet( context, "#{publisherCtrl}",
         PublisherController.class);
```

In addition, we need to add a `getEntityManager` method in the controller class as follows:

```
@ManagedBean( name="publisherCtrl")
```

```
@SessionScoped
public class PublisherController {
  @PersistenceContext( unitName="UnidirAssocApp")
  private EntityManager em;
  @Resource()
  UserTransaction ut;

  public EntityManager getEntityManager() {
    return this.em;
  }
  ...
}
```

JSF needs to compare two publisher instances, when the publisher list has to auto-select the current publisher, in the *update book* use case. For this reason, the `Publisher` model class needs to implement the `equals` method. In our case, two publishers are "one and the same", if the values of their `name` property are equal:

```
@Override
public boolean equals( Object obj) {
  if (obj instanceof Publisher) {
    Publisher publisher = (Publisher) obj;
    return ( this.name.equals( publisher.name));
  } else return false;
}
```

5.3.2. Write the view code

To allow selection of objects which have to be associated with the currently edited object from a list in the *create* and *update* use cases, an HTML selection list (a `select` element) is populated with the instances of the associated object type with the help of JSF `h:selectOneMenu` element. The `WebContent/views/books/create.xhtml` file is updated as follows:

```
<ui:composition template="/WEB-INF/templates/page.xhtml">
 <ui:define name="content">
  <h:form id="createBookForm" styleClass="pure-form pure-form-aligned">
   <h:panelGrid columns="3">
    ...
    <h:outputLabel for="publisher" value="Publisher:" />
    <h:selectOneMenu id="publisher" value="#{book.publisher}">
     <f:selectItem itemValue="" itemLabel="---" />
     <f:selectItems value="#{publisherCtrl.publishers}" var="p"
                    itemLabel="#{p.name}" itemValue="#{p}" />
     <f:converter converterId="pl.m.converter.PublisherConverter" />
    </h:selectOneMenu>
    <h:message for="publisher" errorClass="error" />
    </h:panelGrid>
     <h:commandButton value="Create"
         action="#{bookCtrl.create( book.isbn, book.title,
         book.year, book.publisher.name)}" />
  </h:form>
 </ui:define>
</ui:composition>
```

The object-to-string and back converter is specified by using the `f:converter` JSF element and its associated `@converterId attribute`. The value of this attribute must be the same with the one specified by the `value` property of the annotation `@FacesConverter` used when the converter class is defined, e.g., `@FacesConverter( value="pl.m.converter.PublisherConverter")`. In general, the converter simple class name should be used (e.g., `PublisherConverter`), or the complete fully qualified name, if there is a risk of naming conflicts (e.g., `pl.m.converter.PublisherConverter`). We recommend to use the fully qualified name in all cases.

The `#{publisherCtrl.publishers}` JSF expression results in calling the `getPublishers` method of the `PublisherController` class, which is responsible to return the list of available publishers:

```
public List<Publisher> getPublishers() {
  return Publisher.getAllObjectsAll( em);
}
```

The creation of the book is obtained by using the `h:commandButton` JSF element. It results in the invocation of the `create` method of the `BookController` class:

```
public String create( String isbn, String title, Integer year,
    Publisher publisher) {
  try {
    Book.create( em, ut, isbn, title, year, publisher);
    // Enforce clearing the form after creating the Book row.
    // Without this, the form will show the latest completed data
    FacesContext facesContext = FacesContext.getCurrentInstance();
    facesContext.getExternalContext().getRequestMap().remove( "book");
  } catch ( EntityExistsException e) {
    try {
      ut.rollback();
    } catch ( Exception e1) {
      e1.printStackTrace();
    }
    e.printStackTrace();
  } catch ( Exception e) {
    e.printStackTrace();
  }
  return "create";
}
```

It simply calls the `Book.create` method, responsible with the creation of a new row in the books table.

The code for the *update book* use case is very similar, the only important modification being the addition of the `select` element (i.e., using the `h:selectOneMenu` JSF element), which allows to select which book to update:

```
<ui:composition template="/WEB-INF/templates/page.xhtml">
 <ui:define name="content">
  <h:form id="createBookForm" styleClass="pure-form pure-form-aligned">
   <h:panelGrid columns="3">
    ...
    <h:outputLabel for="selectBook" value="Select book: " />
```

```
    <h:selectOneMenu id="selectBook" value="#{book.isbn}">
     <f:selectItem itemValue="" itemLabel="---" />
     <f:selectItems value="#{bookCtrl.books}" var="b"
         itemValue="#{b.isbn}" itemLabel="#{b.title}" />
     <f:ajax listener="#{bookCtrl.refreshObject( book)}"
         render="isbn title year publisher"/>
    </h:selectOneMenu>
    <h:message for="selectedBook"/>
     ...
   </h:panelGrid>
   <h:commandButton value="Create" action="#{bookCtrl.create(
       book.isbn, book.title, book.year, book.publisher.name)}" />
  </h:form>
 </ui:define>
</ui:composition>
```

Chapter 3. Implementing Unidirectional Non-Functional Associations with Java EE

A unidirectional non-functional association is either *one-to-many* or *many-to-many*. In both cases such an association is represented with the help of a *multi-valued* reference property.

In this chapter, we show

1. how to derive a *data model* in the form of a **Java Entity class model** from an information design model with *multi-valued reference properties* representing *unidirectional non-functional associations*,
2. how to code the Java Entity class model in the form of entity classes (representing *model classes*),
3. how to write the view and controller code based on the model code.

1. Implementing Multi-Valued Reference Properties in Java

A multi-valued reference property, such as `Book::authors`, allows storing a collection of references to objects of some type, such as references to `Author` objects. When creating a new object of type `Book`, it must be possible to assign one or more authors to the book. to assign the constructor needs an extra parameter for the multi-valued reference property. In general, we can allow this value to be a set (or list) of internal object references or of ID references. However, using JPA there is no need for ID references, being very easy to use object references:

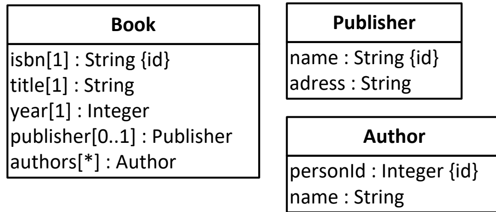
```
@Entity @Table( name="books")
@ManagedBean( name="book")
@ViewScoped
public class Book {
  ...
  private Set<Author> authors;

  public Book() { // required by JPA @Entity annotation!}
  public Book( String isbn, String title, Integer year, Publisher publisher, Se

  public Set<Author> getAuthors() { return this.authors;}
  public void setAuthors( Set<Author> authors) { this.authors = authors;}
  ...
}
```

2. Make a Java Entity Class Model

The starting point for making a Java Entity class model is an OO class model where reference properties represent associations. The following model for our example app contains the multi-valued reference property `Book::authors`, which represents the unidirectional many-to-many association *Book*-**has**-*Author*:



This model contains, in addition to the unidirectional many-to-one association *Book*-**has**-*Publisher*, the unidirectional many-to-many association *Book*-**has**-*Author*, which corresponds to the multi-valued reference property `Book::authors`.

The meaning of the OO class model and its reference properties `publisher` and `authors` can be illustrated by a sample data population for the three model classes:

Table 3.1. Sample data for `Publisher`

Name	Address
Bantam Books	New York, USA
Basic Books	New York, USA

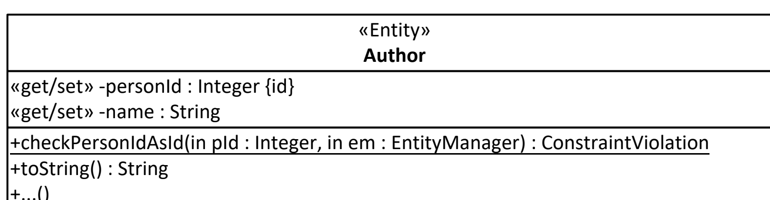
Table 3.2. Sample data for `Book`

ISBN	Title	Year	Authors	Publisher
0553345842	The Mind's I	1982	1, 2	Bantam Books
1463794762	The Critique of Pure Reason	2011	3	
1928565379	The Critique of Practical Reason	2009	3	
0465030793	I Am A Strange Loop	2000	2	Basic Books

Table 3.3. Sample data for `Author`

Author ID	Name
1	Daniel Dennett
2	Douglas Hofstadter
3	Immanuel Kant

We obtain the Java Entity class `Author` from the corresponding class in the OO class model, as explained in the previous parts of this tutorial, by (1) making all properties private, (2) using Java datatype classes, (3) adding public getters and setters, (4) adding a `toString` function, (5) adding the data storage methods `create`, `retrieve`, `update` and `delete`:



In the Java Entity class model, any reference property specified in the OO class model is annotated with the functionality type of the association represented by it. In our example, we annotate the reference property `authors` in the Entity class `Book` with `manyToMany`:

«Entity» **Book**
«get/set» -isbn[1] : String {id} «get/set» -title[1] : String «get/set» -year[1] : Integer «get/set» -publisher[0..1] : Publisher {manyToOne} «get/set» -authors[*] : Author {manyToMany}
+checkIsbnAsId(in isbn : String, in em : EntityManager) : ConstraintViolation +addAuthor(in author : Author) +removeAuthor(in author : Author) +toString() : string +...()

3. New issues

Compared to dealing with a unidirectional functional association, as discussed in the previous section, we have to deal with the following new technical issues:

1. In *model classes* we now have to take care of **multi-valued reference properties** that have to be annotated with **`@OneToMany`** or **`@ManyToMany`**.
2. In the *user interface* code we now have to take care of
   a. showing information about a collection of associated objects in the view table columns that render multi-valued reference properties in the *retrieve/list all objects* use case;
   b. allowing to select a collection of associated objects from a list of all existing instances of the target class in the *create object* and *update object* use cases.

The last issue, allowing to select a collection of associated objects from a list of all existing instances of some class, in general, cannot be solved with the help of an HTML `select multiple` form element because of usability problems. Whenever the set of selectable options is greater than a certain threshold (defined by the number of options that can be seen on the screen without scrolling), the HTML `select multiple` element is no longer usable, and an alternative *multi-selection widget* has to be used.

4. Write the Model Code

The Java Entity class model can be directly coded for getting the code of the model layer of our Java back-end app.

4.1. Summary

1. Code each class of the Java Entity class model as a corresponding entity class.
2. Code an {id} property modifier with the JPA annotation `@Id`
3. Code any property modifier denoting the functionality type of a reference property, such as {manyToOne}, with the corresponding JPA annotation, such as `@ManyToOne`.
4. Code the integrity constraints specified in the model with the help of Java Bean Validation annotations (or custom validation annotations).

5. Code the getters and setters as well as the *add* and *remove* methods for multi-valued properties.
6. Code the `create`, `retrieve`, `update` and `delete` storage management operations as class-level methods.

These steps are discussed in more detail in the following sections.

4.2. Code each class of the Java Entity class model

For the multi-valued reference property `Book::authors`, we use a parametrized `Set` type:

```
@Entity @Table( name="books")
@ViewScoped @ManagedBean( name="book")
public class Book {
  ...
  @ManyToMany( fetch=FetchType.EAGER)
  private Set<Author> authors;

  public Book() {}
  public Book( String isbn, String title, Integer year,
      Publisher publisher, Set<Author> authors) {...}
  ...
  public Set<Author> getAuthors() { return this.authors;}
  public void setAuthors( Set<Author> authors) { this.authors=authors;}
  ...
}
```

The JPA annotation `@ManyToMany` allows to specify the **Many-To-Many** relation between the `Book` and `Author`. The annotation parameter `FetchType.EAGER` is used, so when a `Book` instance is created, the list of authors is populated with the corresponding `Author` instances. As a result of this annotation, a relation table between `Book` and `Author` is created, and the resulting SQL code is shown below:

```
CREATE TABLE IF NOT EXISTS `books_author` (
  `Book_ISBN` varchar(10) NOT NULL,
  `authors_PERSONID` int(11) NOT NULL
);
```

The resulting class name is the concatenation, underscore separated, of the corresponding table names (e.g., `books_author`). The primary key columns from each of the two tables are used to implement the relation. The corresponding column names are created as follows:

- for the table (e.g., `books`) which correspond to the class with the `@ManyToMany` annotation (e.g., `Book`), the class name is used as well as the primary key column name, (e.g., `Book_ISBN`).
- for the other table (e.g., `authors`), the table name is concatenated with the primary key column name, (e.g., authors_PERSONID).

It is possible to control these parameters, i.e., table name and relation column names, by using the `@JoinTable` annotation. To obtain a custom named relation table, e.g., `books_authors` and the corresponding custom named columns, e.g., `book_isbn` and `author_personid`, one can use:

```
@JoinTable( name="books_authors",
```

```
joinColumns = {@JoinColumn( name="book_isbn", referencedColumnName=
inverseJoinColumns = {@JoinColumn( name="author_personid", referenc
```

In our application, we keep the default, so a `@JoinTable` annotation is not used.

The corresponding Author class is coded as a simple Java class with the corresponding JPA and Java Validation API annotations:

```
@Entity
@Table( name="author")
@ManagedBean( name="author")
@ViewScoped
public class Author {
  @Id @PositiveInteger
  private Integer personId;
  @Column( nullable=false)
  @NotNull( message="A name is required!")
  private String name;
  @Column( nullable=false)
  @NotNull( message="A date of birth is required!")
  @Past private Date dateOfBirth;
  @Past private Date dateOfDeath;

  public Author() {}
  public Author( Integer personId, String name, Date dateOfBirth,
      Date dateOfDeath) {...}

  public Integer getPersonId() {...}
  public void setPersonId( Integer personId) {...}
  public String getName() {...}
  public void setName( String name) {...}
  public Date getDateOfBirth() {...}
  public void setDateOfBirth( Date dateOfBirth) {...}
  public Date getDateOfDeath() {...}
  public void setDateOfDeath( Date dateOfDeath) {...}

  public static void create( EntityManager em, UserTransaction ut, Integer pers
      String name, Date dateOfBirth, Date dateOfDeath) {...}
  public static void update( EntityManager em, UserTransaction ut, Integer pers
      String name, Date dateOfBirth, Date dateOfDeath) {...}
  public static void delete( EntityManager em, UserTransaction ut,
      Integer personId) {...}
}
```

Custom validation annotations are defined and implemented (i.e., `@PositiveInteger`) as shown in Part 2 (Validation Tutorial [validation-tutorial.html]).

4.3. Implement a deletion policy

For the reference property `Book::authors`, we have to implement a deletion policy in the `delete` method of the `Author` class. If we just try to delete an author, and the author is referenced by any of the book records, then an integrity constraint violation fires, and the author cannot be deleted. We have two possiblitities for this situation:

1. delete all books (co-)authored by the deleted author;

2. drop from all books (co-)authored by the deleted author the reference to the deleted author.

We go for the second option. This is shown in the following code of the `Author.delete` method:

```
public static void delete( EntityManager em, UserTransaction ut,
    Integer personId) throws Exception {
  ut.begin();
  Author author = em.find( Author.class, personId);
  // find all books with this author
  Query query = em.createQuery( "SELECT DISTINCT b FROM Book b "+
      "INNER JOIN b.authors a WHERE a.personId = :personId");
  query.setParameter( "personId", personId);
  List<Book> books = query.getResultList();
  // update the corresponding book-to-author relations from the association
  // table (otherwise the author can't be deleted)
  for ( Book b : books) {
    b.getAuthors().remove( author);
  }
  // remove the author entry (table row)
  em.remove( author);
  ut.commit();
}
```

Essentially, there are three steps for this operation:

- create a JPQL query which allows to select all book instances for this author - remember, this is an unidirectional association, there is no direct method available for this case.
- for every found book which reference this author, we have to remove the author from its authors list.
- remove tha author - now is safe and no relation specific error should occur.

4.4. Serialization and De-Serialization

In Java, by using the JPA built-in annotations (i.e., `@Entity` for the class and the corresponding ones for the properties) together with converter classes where is the case as shown in Part 3 (Eumeration Tutorial [enumeration-tutorial.html]), the serialization is made internally. This means, the serialization from Java object to the corresponding database table row as well as the de-serilaization from database table column to Java object are made automatically.

5. Write the User Interface Code

5.1. Show information about associated objects in the *List Objects* use case

For showing information about the authors of a book in the *list books* use case, the corresponding cell in the HTML table has to be filled with a list of the names of all authors. For this, we implement a method in the Book class which returns the serialized (as we like to have it displayed) list of author names (in the form, *authorName_1, authorName_2, ..., authorName_n*):

```
public String getAuthorNames() {
  String result = "";
  int i = 0, n = 0;
  if ( this.authors != null) {
    n = this.authors.size();
    for ( Author author : this.authors) {
      result += author.getName();
      if ( i< n - 1) {
        result +=", ";
      }
      i++;
    }
  }
  return result;
 }
```

In the view file, `WebContent/views/books/listAll.xhtml` we use this method in the JSF expression corresponding to the authors cell:

```
<ui:composition template="/WEB-INF/templates/page.xhtml">
  <ui:define name="content">
    <h:dataTable value="#{bookController.books}" var="b">
      ...
      <h:column>
        <f:facet name="header">Publisher</f:facet>
        #{b.publisher.name}
      </h:column>
      <h:column>
        <f:facet name="header">Authors</f:facet>
        #{b.authorNames}
      </h:column>
    </h:dataTable>
    <h:button value="Back" outcome="index" />
  </ui:define>
</ui:composition>
```

Remember, using `b.authorNames` results in calling a method named `getAuthorNames` of the specific `b` object.

5.2. Allow selecting associated objects in the *create* use case

For allowing to select multiple authors to be associated with the currently edited book in the *create book* use case, a multiple selection list (a `select` element with `multiple="multiple"`), as shown in the HTML code below, is populated with the instances of the associated object type. The following code is part of `WebContent/views/books/create.xhtml` view file:

```
<ui:composition template="/WEB-INF/templates/page.xhtml">
  <ui:define name="content">
    <h:form id="createBookForm">
      <h:panelGrid columns="3">
        ...
```

```
        <h:outputLabel for="authors" value="Authors:" />
        <h:selectManyListbox id="authors" value="#{book.authors}">
          <f:selectItems value="#{authorController.authors}" var="a"
                         itemLabel="#{a.name}" itemValue="#{a}" />
          <f:converter converterId="AuthorConverter" />
        </h:selectManyListbox>
        <h:message for="authors" errorClass="error" />
      </h:panelGrid>
      <h:commandButton value="Create" action="#{bookController.create(
       book.isbn, book.title, book.year, book.publisher, book.authors)}"/>
    </h:form>
    <h:button value="Back" outcome="index" />
  </ui:define>
</ui:composition>
```

Remember that we have to add the `equals` method for the `Author` model class. In our case, two authors are "one and the same" if the values of their `personId` property are equal:

```
@Override
public boolean equals( Object obj) {
  if (obj instanceof Author) {
    Author author = (Author) obj;
    return ( this.personId.equals( author.personId));
  } else return false;
}
```

The same like in the case of the `Publisher`, a JSF converter class is used to serialize authors from object instance to view strings, when the select list has to be populated, and back to object instance, when the `create` method of the `BookController` is called (the "Create" button was pushed by the user). The `BookController.create` method is updated with a new parameter, i.e., the corresponding list of selected authors:

```
public String create( String isbn, String title, Integer year, Publisher publish
  try {
    Book.create( em, ut, isbn, title, year, publisher, authors);
    // Enforce clearing the form after creating the Book row.
    // Without this, the form will show the latest completed data
    FacesContext facesContext = FacesContext.getCurrentInstance();
    facesContext.getExternalContext().getRequestMap().remove( "book");
  } catch ( EntityExistsException e) {
    try {
      ut.rollback();
    } catch ( Exception e1) {
      e1.printStackTrace();
    }
    e.printStackTrace();
  } catch ( Exception e) {
    e.printStackTrace();
  }
  return "create";
}
```

The *update book* use case is very similar with the *create book* use case.

6. Run the App and Get the Code

Running your application is simple. First stop (if already started, otherwise skip this part) your Tomcat/TomEE server by using `bin/shutdown.bat` for Windows OS or `bin/shutdown.sh` for Linux. Next, download and unzip the ZIP archive file [UnidirectionalAssociationApp.zip] containing all the source code of the application and also the ANT script file which you have to edit and modify the `server.folder` property value. Now, execute the following command in your console or terminal: `ant deploy -Dappname=unidirectionalassociationapp`. Last, start your Tomcat web server (by using `bin/startup.bat` for Windows OS or `bin/startup.sh` for Linux). Please be patient, this can take some time depending on the speed of your computer. It will be ready when the console display the following info: `INFO: Initializing Mojarra [some library versions and paths are shonw here] for context '/unidirectionalassociationapp'`. Finally, open a web browser and type: `http://localhost:8080/unidirectionalassociationapp/faces/views/app/index.xhtml`

You may want to download the ZIP archive [lib.jar] containing all the dependency libaries, or run the unidirectional association app [http://yew.informatik.tu-cottbus.de/tutorials/java/unidirectionalassociationapp/] directly from our server.

7. Possible Variations and Extensions

7.1. Set-valued versus ordered-set-valued reference properties

In Java,both set-valued and ordered-set-valued reference properties, such as `Author::authoredBooks` and `Book::authors` in the example model below, can be implemented with the predefined generic interface type `List<T>` as their range. However, it seems that set-valued reference properties should be rather implemented with `Set<T>`, such as declaring `Set<Book>` as the range of `Author::authoredBooks`, while ordered-set-valued reference properties should be implemented with `SortedSet<T>`, such as declaring `SortedSet<Author>` as the range of `Book::authors`. This issue still has to be investigated for future editions of this tutorial.

Figure 3.1. An example with a set-valued reference property `authoredBooks` and an ordered-set-valued reference property `authors`