JavaScript Front-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

Gerd Wagner <G.Wagner@b-tu.de>

JavaScript Front-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

by Gerd Wagner

Warning: This tutorial manuscript 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.

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

Publication date 2015-11-12
Copyright © 2014-2015 Gerd Wagner

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 author's consent.

Table of Contents

List of Figures

List of Tables

Foreword

This tutorial is Part 4 of our series of six tutorials [http://web-engineering.info/JsFrontendApp] about model-based development of front-end web applications with plain JavaScript. 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 front-end web application can be provided by any web server, but it is executed on the user's computer device (smartphone, tablet or notebook), and not on the remote web server. Typically, but not necessarily, a front-end web application is a single-user application, which is not shared with other users.

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.

You may also want to take a look at our open access book Building Front-End Web Apps with Plain JavaScript [http://web-engineering.info/JsFrontendApp-Book], which includes all parts of the tutorial in one document, dealing with multiple object types ("books", "publishers" and "authors") and taking care of constraint validation, enumeration attributes, associations and subtypes/inheritance.

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) architecure. 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 represent unidirectional binary associations in a model without any explicit graphical rendering of the association in the model diagram.

1. References and Reference Properties

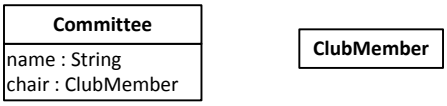
A reference can be either *human-readable* or an *internal object reference*. Human-readable references refer to identifiers that are used in human communication, such as the unique names of astronomical bodies, the ISBN of books and the employee numbers of the employees of a company. Internal object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

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

Objects 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 class `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 *object-oriented* 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 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 Figure 1.10, “The association-free Publisher-Book-Author design model” below.

Normally, a multi-valued reference property is set-valued, implying that the order of the references does not matter. In certain cases, however, it may be list-valued, such that the references are ordered.

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, or
   b. the reference in the referencing object is either dropped or replaced by a another reference.
   For every reference property in our app's model classes, we have to choose, if we allow the deletion of referenced objects, and if yes, 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.

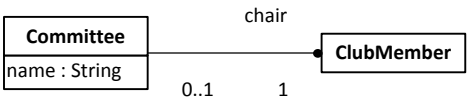
3. Modeling Reference Properties as Unidirectional Associations

A reference property (such as `chair` in the example shown in Figure 1.1 above) can be *visualized* in a UML class diagram in the form of an **association end** owned by its domain class. This requires

to connect the domain class and the range class of the reference property with an association line and annotate the end of this line at the range class side with a "dot", 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 is visually expressed with the help of a small filled circle (also called a "dot") at the end of the association line. This is illustrated in Figure 1.2 below, where the "dot" at the association end `chair` indicates that the association end represents a reference property `chair` in the class `Committee` having `ClubMember` as range.

Figure 1.2. A committee has a club member as chair expressed by 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 is modeled by an association end with a "dot", like `chair` in Figure 1.1, 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

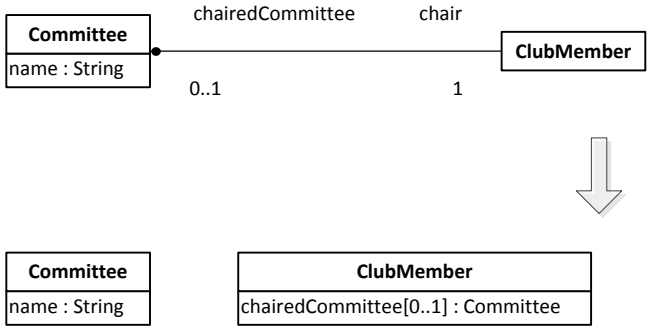
Functionality type	Meaning
one-to-one	both functional and inverse functional
many-to-one	functional
one-to-many	inverse functional
many-to-many	neither functional nor inverse functional

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 be represented as a reference property of the source class. For the case of a unidirectional one-to-one association, this is illustrated in Figure 1.3 below.

Figure 1.3. Representing the unidirectional association ClubMember has Committee as chairedCommittee as a reference property



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 the Committee-has-ClubMember-as-chair association without ownership dots

A model without association end ownership dots is acceptable as a *relational database* design model, but it is incomplete as an information design model for classical *object-oriented* (OO) programming 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 a model without association end ownership dots into a complete OO 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 turned into the association-free model shown in Figure 1.1 above.

2. Place an ownership dot at the `chairedCommittee` association end, leadig 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 5-part tutorial.

Figure 1.5. Modeling a bidirectional association between Committee and ClubMember

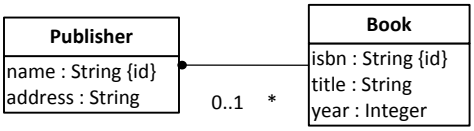
So, whenever we have modeled an association, we have to make a choice, which of its ends represents a reference property and will therefore be marked with an ownership dot. It can be either one, or both. This decision also implies a decision about the *navigability* of the association. When an association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

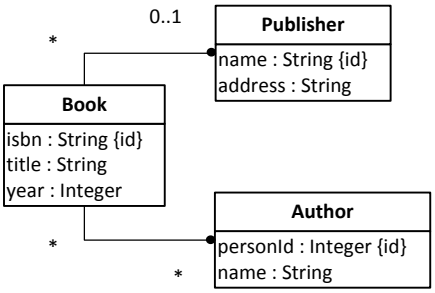
The model shown in Figure 1.6 below (about publishers and books) serves as our running example for a unidirectional functional association in this tutorial. Notice that it contains the unidirectional many-to-one association *Book*-**has**-*Publisher*.

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

How to eliminate explicit unidirectional associations from an information design model by replacing them with reference properties

Since classical OO programming languages do not support assocations as first class citizens, but only classes and reference properties, which represent unidirectional associations (but without any explicit visual rendering), we have to eliminate all explicit associations for obtaining an OO design model.

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;

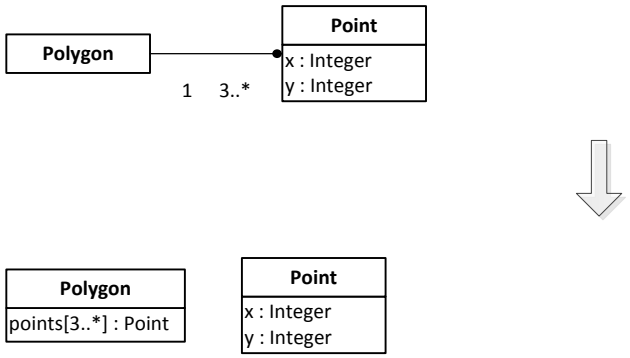
3. the same multiplicity as the target association end,

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 above.

For the case of a unidirectional one-to-many association, Figure 1.8 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

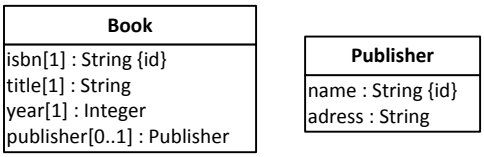
Figure 1.8. Turn a non-functional target association end into a corresponding reference property



7.2. Eliminating associations from the Publisher-Book-Author 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 association-free model:

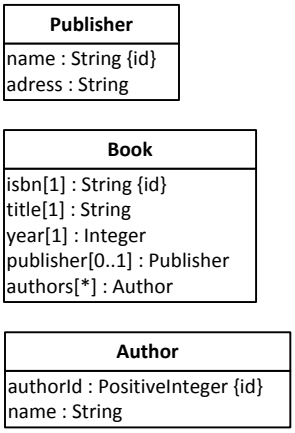
Figure 1.9. The association-free Publisher-Book design model



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 appending the multiplicity `0..1` to its name.

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 association-free model:

Figure 1.10. The association-free Publisher-Book-Author design model



After the platform-independent association-free information design 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 encoded in the chosen platform. Thus, for any platform considered, the tutorial contains two sections: 1) how to make the platform-specific data model, and 2) how to encode this model.

8. Rendering Reference Properties in the User Interface

In an HTML-form-based user interface (UI), we have a correspondence between the different kinds of properties defined in the model classes of an app and the form controls used for the input and output of their values. We have to distinguish between various kinds of model class **attributes**, which are typically mapped to various types of HTML **`input` fields**, and reference properties, which are typically mapped to HTML **`select` controls** (selection lists). 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 always be represented by a single-`select` control 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 **multi-valued reference property** can be represented by a multiple-`select` control in the UI. However, this control 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 an *association list* containing all associated objects choosen 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 associated objects.

Such a ***multi-select widget*** consists of

1. a HTML list element containing the associated objects, where each list item contains a push button for removing the object from the association list;

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 Plain JavaScript

The three example apps that we have discussed in previous chapters, the *minimal app*,the *validation app*, and the *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 plain JavaScript data model from an association-free information design model with single-valued reference properties representing *unidirectional functional associations*,

2. how to encode the data model in the form of plain JavaScript model classes,

3. how to write the view and controller code based on the model code.

1. Implementing Single-Valued Reference Properties with Plain JavaScript

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 would have to take a decision if this value is expected to be an internal object reference or an ID reference. In JavaScript, however, we can take a more flexible approach and allow using either of them, as shown in the following exmple:

```
function Book( slots) {
  // set the default values for the parameter-free default constructor
  ...
  this.publisher = null;  // optional reference property
  ...
  // constructor invocation with a slots argument
```

```
  if (arguments.length > 0) {
    ...
    if (slots.publisher) this.setPublisher( slots.publisher);
    else if (slots.publisherIdRef) this.setPublisher( slots.publisherIdRef);
    ...
  }
}
```

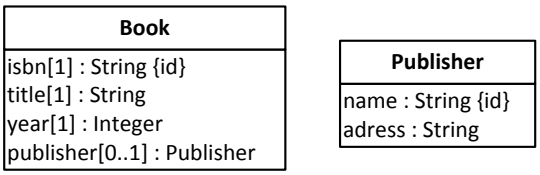
Notice that, for flexibility, the constructor parameter `slots` may contain either a `publisher` slot representing an (internal) JavaScript object reference or a `publisherIdRef` slot representing an (external) ID reference (or foreign key). We handle the resulting ambiguity in the property setter by checking the type of the argument as shown in the following code fragment:

```
Book.prototype.setPublisher = function (p) {
  ...
  var publisherIdRef = "";
  if (typeof(p) !== "object") {  // ID reference
    publisherIdRef = p;
  } else {                       // object reference
    publisherIdRef = p.name;
  }
}
```

Notice that the `name` of a publisher is used as an ID reference (or foreign key), since this is the standard identifier (or primary key) of the `Publisher` class.

2. Make a JavaScript Data Model

The starting point for making a JavaScript data model is an association-free information design model like the following one:



How to make this design model has been discussed in the previous chapter (about unidirectional associations).

The meaning of the design model and its reference property `publisher` can be illustrated by a sample data population for the two model classes `Book` and `Publisher`:

Table 2.1. Sample data for `Publisher`

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

Table 2.2. Sample data for `Book`

ISBN	Title	Year	Publisher
0553345842	The Mind's I	1982	Bantam Books
1463794762	The Critique of Pure Reason	2011	

ISBN	Title	Year	Publisher
1928565379	The Critique of Practical Reason	2009	
0465030793	I Am A Strange Loop	2000	Basic Books

We now show how to derive a JavaScript data model from the design model in three steps.

1. Create a **check** operation for each non-derived property in order to have a central place for implementing ***property constraints***. For a standard identifier property (such as `Book::isbn`), three check operations are needed:

   a. A basic check operation, such as `checkIsbn`, for checking all syntactic constraints, but not the *mandatory value* and the *uniqueness* constraints.

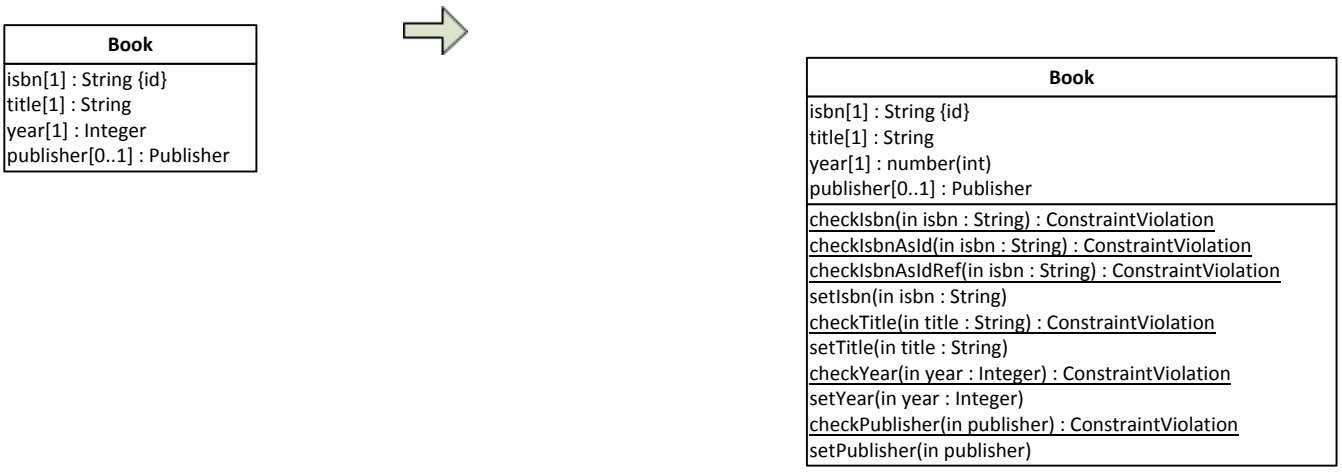
   b. A standard ID check operation, such as `checkIsbnAsId`, for checking the *mandatory value* and *uniqueness* constraints that are required for an identifier (or *primary key*) attribute.

   c. An *ID reference* check operation, such as `checkIsbnAsIdRef`, for checking the *referential integrity* constraint that is required for an *ID reference (IdRef)* (or *foreign key*) attribute.

   For a reference property, such as `Book::publisher`, the check operation, `Book.checkPublisher`, has to check the corresponding *referential integrity constraint*, and possibly also a *mandatory value constraint*, if the property is mandatory.

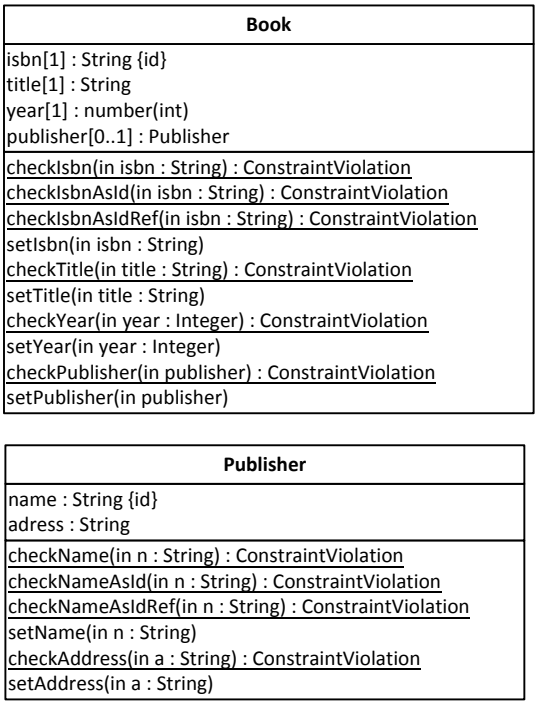
2. Create a **setter** operation for each non-derived **single-valued** property. In the setter, the corresponding check operation is invoked and the property is only set, if the check does not detect any constraint violation.

This leads to the following JavaScript data model class `Book`, where the class-level ('static') methods are shown underlined:



We have to perform a similar transformation also for the class `Publisher`. This gives us the complete JavaScript data model derived from the above association-free model, as depicted in the following class diagram.

Figure 2.1. The complete JavaScript data model



3. New issues

Compared to the single-class app discussed in Part 2 [validation-tutorial.html] (and Part 3), 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. maintenance of ***referential integrity***
   b. choosing and implementing one of the two possible deletion policies discussed in 2 for managing the corresponding object destruction dependency in the `destroy` method of the property's range class,
   c. conversion between (internal) object references and (external) ID references in the serialization and de-serialization procedures.
2. In the *user interface "(vie"w)* code we now have to take care of
   a. showing information about associated objects in the *list objects* 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.

The last issue, allowing to select an associated object from a list of all existing instances of some class, can be solved with the help of an HTML `select` form element.

4. Write the Model Code

How to Encode a JavaScript Data Model

The JavaScript data model can be directly encoded for getting the code of the model layer of our JavaScript frontend app.

4.1. Summary

1. Encode each model class as a JavaScript constructor function.
2. Encode the property checks in the form of class-level ('static') methods. Take care that all constraints of a property as specified in the JavaScript data model are properly encoded in the property checks.
3. Encode the property setters as (instance-level) methods. In each setter, the corresponding property check is invoked and the property is only set, if the check does not detect any constraint violation.
4. Encode any other operation.

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

4.2. Encode each class of the JavaScript data model as a constructor function

Each class `C` of the data model is encoded by means of a corresponding JavaScript *constructor function* with the same name `C` having a single parameter `slots`, which has a key-value slot providing a value for each non-derived property of the class. The range of these properties should be indicated in a comment. In the case of a reference property the range is another model class.

In the constructor body, we first assign default values to all properties. The default value of a reference property is `null`. These default values will be used when the constructor is invoked as a default constructor, that is, without any argument. If the constructor is invoked with arguments, the default values may be overwritten by calling the *setter* methods for all properties.

For instance, the `Publisher` class from the JavaScript data model is encoded in the following way:

```
function Publisher( slots) {
  // set the default values for the parameter-free default constructor
  this.name = "";     // String
  this.address = "";  // String
  // constructor invocation with arguments
  if (arguments.length > 0) {
    this.setName( slots.name);
    this.setAddress( slots.address);
  }
};
```

Since the setters may throw constraint violation exceptions, the constructor function, and any setter, should be called in a try-catch block where the catch clause takes care of logging suitable error messages.

For each model class `C`, we define a class-level property `C.instances` representing the collection of all `C` instances managed by the application in the form of an entity table (a map of records): This property is initially set to `{}`. For instance, in the case of the model class `Book`, we define:

```
Publisher.instances = {};
```

The `Book` class from the JavaScript data model is encoded in a similar way:

```
function Book( slots) {
  // set the default values for the parameter-free default constructor
  this.isbn = "";         // string
  this.title = "";        // string
  this.year = 0;          // number(int)
```

```
  this.publisher = null;  // Publisher
  // constructor invocation with a slots argument
  if (arguments.length > 0) {
    this.setIsbn( slots.isbn);
    this.setTitle( slots.title);
    this.setYear( slots.year);
    if (slots.publisher) this.setPublisher( slots.publisher);
    else if (slots.publisherIdRef) this.setPublisher( slots.publisherIdRef);  }
}
```

Notice that the `Book` constructor can be invoked either with an object reference `slots.publisher` or with an ID reference `slots.publisherIdRef`. This liberal approach makes using the `Book` constructor more flexible and more robust.

4.3. Encode the property checks

Take care that all constraints of a property as specified in the JavaScript data model are properly encoded in its check function, as explained in Part 2 (Validation Tutorial [validation-tutorial.html]). Error classes are defined in the file `lib/errorTypes.js`.

For instance, for the `checkName` operation we obtain the following code:

```
Publisher.checkName = function (n) {
  if (!n) {
    return new NoConstraintViolation();
  } else if (typeof(n) !== "string" || n.trim() === "") {
    return new TypeConstraintViolation(
        "The name must be a non-empty string!");
  } else {
    return new NoConstraintViolation();
  }
};
```

Notice that, since the `name` attribute is the standard ID attribute of `Publisher`, we only check syntactic constraints in `checkName`, and check the mandatory value and uniqueness constraints in `checkNameAsId`, which invokes `checkName`:

```
Publisher.checkNameAsId = function (n) {
  var constraintViolation = Publisher.checkName( n);
  if ((constraintViolation instanceof NoConstraintViolation)) {
    if (!n) {
      return new MandatoryValueConstraintViolation(
          "A value for the name must be provided!");
    } else if (Publisher.instances[n]) {
      constraintViolation = new UniquenessConstraintViolation(
          "There is already a publisher record with this name!");
    } else {
      constraintViolation = new NoConstraintViolation();
    }
  }
  return constraintViolation;
};
```

Since for any standard ID attribute, we may have to deal with ID references (foreign keys) in other classes, we need to provide a further check function, called `checkNameAsIdRef`, for checking the referential integrity constraint, as illustrated in the following example:

```
Publisher.checkNameAsIdRef = function (n) {
  var constraintViolation = Publisher.checkName( n);
  if ((constraintViolation instanceof NoConstraintViolation)) {
    if (!Publisher.instances[n]) {
      constraintViolation = new ReferentialIntegrityConstraintViolation(
          "There is no publisher record with this name!");
    }
  }
  return constraintViolation;
};
```

The condition `(!Publisher.instances[n])` checks if there is no publisher object with the given name `n`, and then creates a `constraintViolation` object. This referential integrity constraint check is used by the following `Book.checkPublisher` function:

```
Book.checkPublisher = function (publisherIdRef) {
  var constraintViolation = null;
  if (!publisherIdRef) {
    constraintViolation = new NoConstraintViolation();  // optional
  } else {
    // invoke foreign key constraint check
    constraintViolation = Publisher.checkNameAsIdRef( publisherIdRef);
  }
  return constraintViolation;
};
```

4.4. Encode the property setters

Encode the setter operations as (instance-level) methods. In the setter, the corresponding check operation is invoked and the property is only set, if the check does not detect any constraint violation. In the case of a reference property, we allow invoking the setter either with an internal object reference or with an ID reference. The resulting ambiguity is resolved by testing if the argument provided in the invocation of the setter is an object or not. For instance, the setter operation `setPublisher` is encoded in the following way:

```
Book.prototype.setPublisher = function (p) {
  var constraintViolation = null;
  var publisherIdRef = "";
  // a publisher can be given as ...
  if (typeof(p) !== "object") {  // an ID reference or
    publisherIdRef = p;
  } else {                       // an object reference
    publisherIdRef = p.name;
  }
  constraintViolation = Book.checkPublisher( publisherIdRef);
  if (constraintViolation instanceof NoConstraintViolation) {
    // create the new publisher reference
    this.publisher = Publisher.instances[ publisherIdRef];
  } else {
    throw constraintViolation;
  }
};
```

4.5. Implement a deletion policy

For any reference property, we have to choose and implement one of the two possible deletion policies discussed in 2 for managing the corresponding object destruction dependency in the `destroy` method of the 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 go for the second option. This is shown in the following code of the `Publisher.destroy` method where for all concerned book objects `book` the property `book.publisher` is cleared:

```
Publisher.destroy = function (name) {
  var publisher = Publisher.instances[name];
  var book=null, keys=[];
  // delete all references to this publisher in book objects
  keys = Object.keys( Book.instances);
  for (var i=0; i < keys.length; i++) {
    book = Book.instances[keys[i]];
    if (book.publisher === publisher) delete book.publisher;
```

```
  }
  // delete the publisher record
  delete Publisher.instances[name];
  console.log("Publisher " + name + " deleted.");
};
```

4.6. Serialization and De-Serialization

The serialization method `convertObj2Row` converts typed objects with internal object references to corresponding (untyped) record objects with ID references:

```
Book.prototype.convertObj2Row = function () {
  var bookRow = util.cloneObject(this), keys=[];
  if (this.publisher) {
    // create publisher ID reference
    bookRow.publisherIdRef = this.publisher.name;
  }
  return bookRow;
};
```

The de-serialization method `convertRow2Obj` converts (untyped) record objects with ID references to corresponding typed objects with internal object references:

```
Book.convertRow2Obj = function (bookRow) {
  var book={}, persKey="";
  var publisher = Publisher.instances[bookRow.publisherIdRef];
  // replace the publisher ID reference with object reference
  delete bookRow.publisherIdRef;
  bookRow.publisher = publisher;
  try {
    book = new Book( bookRow);
  } catch (e) {
    console.log( e.constructor.name + " while deserializing a book row: " + e.message);
  }
  return book;
};
```

5. The View and Controller Layers

The user interface (UI) consists of a start page for navigating to the data management UI pages, one for each object type (in our example, `books.html` and `publishers.html`). Each of these data management UI pages contains 5 sections, such as *manage books*, *list books*, *create book*, *update book* and *delete book*, such that only one of them is displayed at any time (by setting the CSS property `display:none` for all others).

5.1. Initialize the app

For initializing the data management use cases, the required data (all publisher and book records) are loaded from persistent storage. This is performed in a controller procedure such as `pl.ctrl.books.manage.initialize` in `ctrl/books.js` with the following code:

```
pl.ctrl.books.manage = {
  initialize: function () {
    Publisher.loadAll();
    Book.loadAll();
    pl.view.books.manage.setUpUserInterface();
  }
};
```

The `initialize` method for managing book data loads the publisher table and the book table since the book data management UI needs to provide selection list for both object types. Then the menu for book data managemetn options is set up by the `setUpUserInterface` method.

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:

```
pl.view.books.list = {
  setupUserInterface: function () {
    var tableBodyEl = document.querySelector(
                      "section#Book-R>table>tbody");
    var keys = Object.keys( Book.instances);
    var row=null, listEl=null, book=null;
    tableBodyEl.innerHTML = "";
    for (var i=0; i < keys.length; i++) {
      book = Book.instances[keys[i]];
      row = tableBodyEl.insertRow(-1);
      row.insertCell(-1).textContent = book.isbn;
      row.insertCell(-1).textContent = book.title;
      row.insertCell(-1).textContent = book.year;
      row.insertCell(-1).textContent =
          book.publisher ? book.publisher.name : "";
    }
    document.getElementById("Book-M").style.display = "none";
    document.getElementById("Book-R").style.display = "block";
  }
};
```

For a multi-valued reference property, the table cell would have to be filled with a list of all associated objects referenced by the property.

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

For allowing to select objects 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 a utility method `fillSelectWithOptions`. In the case of the *create book* use case, the UI is set up by the following procedure:

```
pl.view.books.create = {
  setupUserInterface: function () {
    var formEl = document.querySelector("section#Book-C > form"),
        publisherSelectEl = formEl.selectPublisher,
        submitButton = formEl.commit;
    // define event handlers for responsive validation
    formEl.isbn.addEventListener("input", function () {
      formEl.isbn.setCustomValidity(
          Book.checkIsbnAsId( formEl.isbn.value).message);
    });
    // set up the publisher selection list
    util.fillSelectWithOptions( publisherSelectEl, Publisher.instances, "name");
    // define event handler for submitButton click events
    submitButton.addEventListener("click", this.handleSubmitButtonClickEvent);
    // define event handler for neutralizing the submit event
    formEl.addEventListener( 'submit', function (e) {
      e.preventDefault();
      formEl.reset();
    });
    // replace the manageBooks form with the Book-C form
    document.getElementById("Book-M").style.display = "none";
    document.getElementById("Book-C").style.display = "block";
    formEl.reset();
  },
  handleSubmitButtonClickEvent: function () {
```

```
    ...
  }
};
```

When the user clicks the submit button, all form control values, including the value of the `select` control, are copied to a `slots` list, which is used as the argument for invoking the `add` method after all form fields have been checked for validity, as shown in the following program listing:

```
  handleSubmitButtonClickEvent: function () {
    var formEl = document.querySelector("section#Book-C > form");
    var slots = {
        isbn: formEl.isbn.value,
        title: formEl.title.value,
        year: formEl.year.value,
        publisherIdRef: formEl.selectPublisher.value
    };
    // check input fields and show constraint violation error messages
    formEl.isbn.setCustomValidity( Book.checkIsbnAsId( slots.isbn).message);
    /* ... (do the same with title and year) */
    // save the input data only if all of the form fields are valid
    if (formEl.checkValidity()) {
      Book.add( slots);
    }
  }
```

The `setupUserInterface` code for the *update book* use case is similar.

Chapter 3. Implementing Unidirectional Non-Functional Associations with Plain JavaScript

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

In this chapter, we show

1. how to derive a JavaScript data model from an association-free information design model with *multi-valued reference properties* representing *unidirectional non-functional associations*,
2. how to encode the JavaScript data model in the form of JavaScript model classes,
3. how to write the view and controller code based on the model code.

1. Implementing Multi-Valued Reference Properties in JavaScript

A mulit-valued reference property, such as the property `authors` of the object type `Book`, allows storing a set of references to objects of some type, such as `Author`. When creating a new object of type `Book`, the constructor function needs to have a parameter for providing a suitable value for this reference property. In JavaScript we can allow this value to be a set (or list) of internal object references or of ID references, as shown in the following example:

```
function Book( slots) {
  // set the default values for the parameter-free default constructor
  ...
  this.authors = {};   // map of Author object referencess
  ...
  // constructor invocation with a slots argument
  if (arguments.length > 0) {
    ...
    this.setAuthors( slots.authors || slots.authorsIdRef);
    ...
  }
}
```

Notice that the constructor parameter `slots` is expected to contain either an `authors` or an `authorsIdRef` slot. The JavaScript expression `slots.authors || slots.authorsIdRef`, using the disjunction operator `||`, evaluates to `authors`, if `slots` contains an object reference slot `authors`, or to `authorsIdRef`, otherwise. We handle the resulting ambiguity in the property setter by checking the type of the argument as shown in the following code fragment:

```
Book.prototype.setAuthors = function (a) {
  var keys=[], i=0;
  this.authors = {};
  if (Array.isArray(a)) {  // array of ID references
    for (i= 0; i < a.length; i++) {
      this.addAuthor( a[i]);
    }
  } else {  // map of object references
    keys = Object.keys( a);
    for (i=0; i < keys.length; i++) {
      this.addAuthor( a[keys[i]]);
    }
```

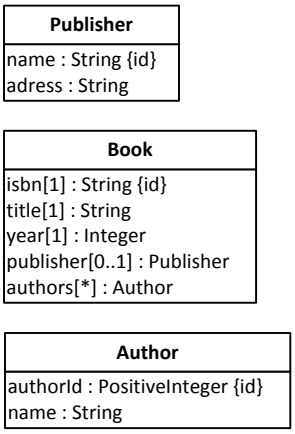
```
  }
};
```

A multi-valued reference property can be implemented as a property with either an array, or a map, of object references, as its value. We prefer using maps for implementing multi-valued reference properties since they guarantee that each element is unique, while with an array we would have to prevent duplicate elements. Also, an element of a map can be easily deleted (with the help of the `delete` operator), while this requires more effort in the case of an array. But for implementing list-valued reference properties, we need to use arrays.

We use the standard identifiers of the referenced objects as keys. If the standard identifier is an integer, we must take special care in converting ID values to strings for using them as keys.

2. Make a JavaScript Data Model

The starting point for making a JavaScript data model is an association-free information design 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*:



The meaning of the design 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

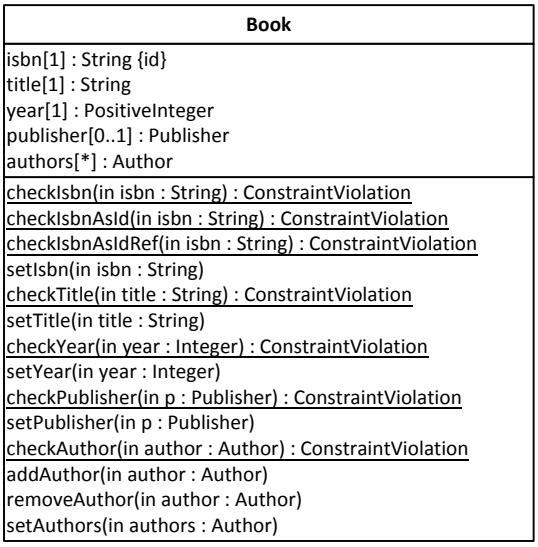
We now show how to derive a JavaScript data model from the design model in three steps.

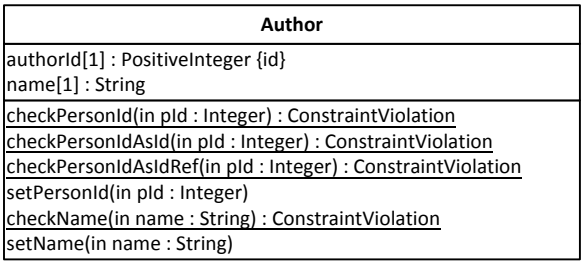
1. Create a **check** operation for each non-derived property in order to have a central place for implementing ***property constraints***. For any reference property, no matter if single-valued (like `Book::publisher`) or multi-valued (like `Book::authors`), the check operation (`checkPublisher` or `checkAuthor`) has to check the corresponding *referential integrity constraint*, which requires that all references reference an existing object, and possibly also a *mandatory value constraint*, if the property is mandatory.

2. Create a **set** operation for each non-derived **single-valued** property. In the setter, the corresponding check operation is invoked and the property is only set, if the check does not detect any constraint violation.

3. Create an **add**, a **remove** and a **set** operation for each non-derived **multi-valued** property. In the case of the `Book::authors` property, we would create the operations `addAuthor`, `removeAuthor` and `setAuthors` in the `Book` class rectangle.

This leads to the following JavaScript data model, where we only show the classes `Book` and `Author`, while the missing class `Publisher` is the same as in Figure 2.1:





Notice that, for simplicity, we do not include the code for all validation checks shown in the data model in the code of the example app.

3. New issues

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

1. In the *model* code we now have to take care of **multi-valued reference properties** that require
   a. implementing an ***add*** and a ***remove*** operation, as well as a ***setter*** for assigning a set of references with the help of the add operation.
   b. converting a map of internal object references to an array of ID references in the serialization function `convertObj2Row` and converting such an array back to a map of internal object references in the de-serialization function `convertRow2Obj`.
2. In the *user interface "(vie"w)* code we now have to take care of
   a. showing information about **a set** of associated objects in the *list objects* use case;
   b. allowing to select **a set** 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 set of associated objects from a list of all existing instances of some class, can, in general, not 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-select widget* has to be used.

4. Write the Model Code

4.1. Encode the add and remove operations

For the multi-valued reference property `Book::authors`, we need to encode the operations `addAuthor` and `removeAuthor`. Both operations accept one parameter denoting an author either by ID reference (the author ID as integer or string) or by an internal object reference. The code of `addAuthor` is as follows:

```
Book.prototype.addAuthor = function (a) {
  var constraintViolation=null,
      authorIdRef=0, authorIdRefStr="";
  // an author can be given as ...
  if (typeof( a) !== "object") {  // an ID reference or
    authorIdRef = parseInt( a);
  } else {                        // an object reference
    authorIdRef = a.authorId;
  }
  constraintViolation = Book.checkAuthor( authorIdRef);
  if (authorIdRef &&
      constraintViolation instanceof NoConstraintViolation) {
    // add the new author reference
    authorIdRefStr = String( authorIdRef);
    this.authors[ authorIdRefStr] =
        Author.instances[ authorIdRefStr];
  }
};
```

The code of `removeAuthor` is similar to `addAuthor`:

```
Book.prototype.removeAuthor = function (a) {
  var constraintViolation = null;
  var authorIdRef = "";
  // an author can be given as ID reference or object reference
  if (typeof(a) !== "object") authorIdRef = parseInt( a);
  else authorIdRef = a.authorId;
  constraintViolation = Book.checkAuthor( authorIdRef);
  if (constraintViolation instanceof NoConstraintViolation) {
    // delete the author reference
    delete this.authors[ authorIdRef];
  }
};
```

For assigning an array of ID references, or a map of object references, to the property `Book::authors`, the method `setAuthors` adds them one by one with the help of `addAuthor`:

```
Book.prototype.setAuthors = function (a) {
  var keys=[];
  this.authors = {};
  if (Array.isArray(a)) {  // array of IdRefs
    for (i= 0; i < a.length; i++) {
      this.addAuthor( a[i]);
    }
  } else {  // map of object references
    keys = Object.keys( a);
    for (i=0; i < keys.length; i++) {
      this.addAuthor( a[keys[i]]);
    }
  }
};
```

4.2. Implement a deletion policy

For the reference property `Book::authors`, we have to implement a deletion policy in the `destroy` method of the `Author` class. We have to choose between

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

2. dropping 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.destroy` method where for all concerned book objects `book` the author reference `book.authors[authorKey]` is dropped:

```
Author.destroy = function (id) {
  var authorKey = id.toString(),
      author = Author.instances[authorKey],
      key="", keys=[], book=null;
  // delete all dependent book records
  keys = Object.keys( Book.instances);
  for (i=0; i < keys.length; i++) {
    key = keys[i];
    book = Book.instances[key];
    if (book.authors[authorKey]) delete book.authors[authorKey];
  }
  // delete the author record
  delete Author.instances[authorKey];
  console.log("Author " + author.name + " deleted.");
};
```

4.3. Serialization and De-Serialization

The serialization method `convertObj2Row` converts typed objects with internal object references to corresponding (untyped) record objects with ID references:

```
Book.prototype.convertObj2Row = function () {
```

```
  var bookRow = util.cloneObject(this), keys=[];
  // create authors ID references
  bookRow.authorsIdRef = [];
  keys = Object.keys( this.authors);
  for (i=0; i < keys.length; i++) {
    bookRow.authorsIdRef.push( parseInt( keys[i]));
  }
  if (this.publisher) {
    // create publisher ID reference
    bookRow.publisherIdRef = this.publisher.name;
  }
  return bookRow;
};
```

The de-serialization method `convertRow2Obj` converts (untyped) record objects with ID references to corresponding typed objects with internal object references:

```
Book.convertRow2Obj = function (bookRow) {
  var book=null, authorKey="",
      publisher = Publisher.instances[bookRow.publisherIdRef];
  // replace the "authorsIdRef" array of ID references
  // with a map "authors" of object references
  bookRow.authors = {};
  for (i=0; i < bookRow.authorsIdRef.length; i++) {
    authorKey = bookRow.authorsIdRef[i].toString();
    bookRow.authors[authorKey] = Author.instances[authorKey];
  }
  delete bookRow.authorsIdRef;
  // replace publisher ID reference with object reference
  delete bookRow.publisherIdRef;
  bookRow.publisher = publisher;

  try {
    book = new Book( bookRow);
  } catch (e) {
    console.log( e.constructor.name +
        " while deserializing a book row: " + e.message);
  }
  return book;
};
```

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 is filled with a list of the names of all authors with the help of the utility function `util.createListFromMap`:

```
pl.view.books.list = {
  setupUserInterface: function () {
    var tableBodyEl = document.querySelector(
            "section#Book-R>table>tbody");
    var row=null, book=null, listEl=null,
        keys = Object.keys( Book.instances);
    tableBodyEl.innerHTML = "";  // drop old contents
    for (i=0; i < keys.length; i++) {
      book = Book.instances[keys[i]];
      row = tableBodyEl.insertRow(-1);
      row.insertCell(-1).textContent = book.isbn;
      row.insertCell(-1).textContent = book.title;
      row.insertCell(-1).textContent = book.year;
      // create list of authors
      listEl = util.createListFromMap(
          book.authors, "name");
      row.insertCell(-1).appendChild( listEl);
```

```
      row.insertCell(-1).textContent =
          book.publisher ? book.publisher.name : "";
    }
    document.getElementById("Book-M").style.display = "none";
    document.getElementById("Book-R").style.display = "block";
  }
};
```

The utility function `util.createListFromMap` has the following code:

```
createListFromMap: function (aa, displayProp) {
  var listEl = document.createElement("ul");
  util.fillListFromMap( listEl, aa, displayProp);
  return listEl;
},
fillListFromMap: function (listEl, aa, displayProp) {
  var keys=[], listItemEl=null;
  // drop old contents
  listEl.innerHTML = "";
  // create list items from object property values
  keys = Object.keys( aa);
  for (var j=0; j < keys.length; j++) {
    listItemEl = document.createElement("li");
    listItemEl.textContent = aa[keys[j]][displayProp];
    listEl.appendChild( listItemEl);
  }
},
```

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.

```
<section id="Book-C" class="UI-Page">
  <h1>Public Library: Create a new book record</h1>
  <form>
    <div class="field">
      <label>ISBN: <input type="text" name="isbn" /></label>
    </div>
    <div class="field">
      <label>Title: <input type="text" name="title" /></label>
    </div>
    <div class="field">
      <label>Year: <input type="text" name="year" /></label>
    </div>
    <div class="select-one">
      <label>Publisher: <select name="selectPublisher"></select></label>
    </div>
    <div class="select-many">
      <label>Authors:
        <select name="selectAuthors" multiple="multiple"></select>
      </label>
    </div>
    <div class="button-group">
      <button type="submit" name="commit">Save</button>
      <button type="button" onclick="pl.view.books.manage.refreshUI()">
       Back to menu</button>
    </div>
  </form>
</section>
```

The *create book* UI is set up by populating selection lists for selecting the authors and the publisher with the help of a utility method `fillSelectWithOptions` as shown in the following program listing:

```
pl.view.books.create = {
  setupUserInterface: function () {
    var formEl = document.querySelector("section#Book-C > form"),
```

```
        publisherSelectEl = formEl.selectPublisher,
        submitButton = formEl.commit;
    // define event handlers for form field input events
      ...
    // set up the (multiple) authors selection list
    util.fillSelectWithOptions( authorsSelectEl,
        Author.instances, "authorId", {displayProp:"name"});
    // set up the publisher selection list
    util.fillSelectWithOptions( publisherSelectEl,
        Publisher.instances, "name");
    ...
  },
  handleSubmitButtonClickEvent: function () {
    ...
  }
};
```

When the user clicks the submit button, all form control values, including the value of any single-`select` control, are copied to a corresponding `slots` record variable, which is used as the argument for invoking the `add` method after all form fields have been checked for validity. Before invoking `add`, we first have to create (in the `authorsIdRef` slot) a list of author ID references from the selected options of the multiple author selection list, as shown in the following program listing:

```
  handleSubmitButtonClickEvent: function () {
    var i=0,
        formEl = document.querySelector("section#Book-C > form"),
        selectedAuthorsOptions = formEl.selectAuthors.selectedOptions;
    var slots = {
        isbn: formEl.isbn.value,
        title: formEl.title.value,
        year: formEl.year.value,
        authorsIdRef: [],
        publisherIdRef: formEl.selectPublisher.value
    };
    // check all input fields
    ...
    // save the input data only if all of the form fields are valid
    if (formEl.checkValidity()) {
      // construct the list of author ID references
      for (i=0; i < selectedAuthorsOptions.length; i++) {
        slots.authorsIdRef.push( selectedAuthorsOptions[i].value);
      }
      Book.add( slots);
    }
  }
```

The *update book* use case is discussed in the next section.

5.3. Allow selecting associated objects in the *update* use case

Unfortunatley, the multiple-`select` control is not really usable for displaying and allowig to maintain the set of associated authors in realistic use cases where we have several hundreds or thousands of authors, because the way it renders the choice is visually too scattered. So we better use a special ***multi-select widget*** that allows to add (and remove) objects to (and from) a list of associated objects, as discussed in 8. In order to show how this widget can replace the multiple-selection list discussed in the previous section, we use it now in the *update book* use case.

For allowing to maintain the set of authors associated with the currently edited book in the *update book* use case, a *multi-select widget* as shown in the HTML code below, is populated with the instances of the `Author` class.

```
<section id="Book-U" class="UI-Page">
  <h1>Public Library: Update a book record</h1>
  <form>
```

```
    <div class="select-one">
      <label>Select book: <select name="selectBook"></select></label>
    </div>
    <div class="field">
      <label>ISBN: <output name="isbn"></output></label>
    </div>
    <div class="field">
      <label>Title: <input type="text" name="title" /></label>
    </div>
    <div class="field">
      <label>Year: <input type="text" name="year" /></label>
    </div>
    <div class="select-one">
      <label>Publisher: <select name="selectPublisher"></select></label>
    </div>
    <div class="widget">
      <label for="updBookSelectAuthors">Authors: </label>
      <div class="MultiSelectionWidget" id="updBookSelectAuthors"></div>
    </div>
    <div class="button-group">
      <button type="submit" name="commit">Save</button>
      <button type="button"
       onclick="pl.view.books.manage.refreshUI()">Back to menu</button>
    </div>
  </form>
</section>
```

The *update book* UI is set up (in the `setupUserInterface` procedure shown below) by populating

1. the selection list for selecting the book to be updated with the help of the utility method `fillSelectWithOptions`, and
2. the selection list for updating the publisher with the help of the utility method `fillSelectWithOptions`,

while the multi-select widget for updating the associated authors of the book is only populated (in `handleSubmitButtonClickEvent`) when a book to be updated has been chosen.

```
pl.view.books.update = {
  setupUserInterface: function () {
    var formEl = document.querySelector("section#Book-U > form"),
        bookSelectEl = formEl.selectBook,
        publisherSelectEl = formEl.selectPublisher,
        submitButton = formEl.commit;
    // set up the book selection list
    util.fillSelectWithOptions( bookSelectEl, Book.instances,
        "isbn", {displayProp:"title"});
    bookSelectEl.addEventListener("change", this.handleBookSelectChangeEvent);
    ... // define event handlers for title and year input events
    // set up the associated publisher selection list
    util.fillSelectWithOptions( publisherSelectEl, Publisher.instances, "name");
    // define event handler for submitButton click events
    submitButton.addEventListener("click", this.handleSubmitButtonClickEvent);
    // define event handler for neutralizing the submit event and reseting the form
    formEl.addEventListener( 'submit', function (e) {
      var authorsSelWidget = document.querySelector(
              "section#Book-U > form .MultiSelectionWidget");
      e.preventDefault();
      authorsSelWidget.innerHTML = "";
      formEl.reset();
    });
    document.getElementById("Book-M").style.display = "none";
    document.getElementById("Book-U").style.display = "block";
    formEl.reset();
  },
```

When a book to be updated has been chosen, the form input fields `isbn`, `title` and `year`, and the `select` control for updating the publisher, are assigned corresponding values from the chosen book, and the associated authors selection widget is set up:

```
handleBookSelectChangeEvent: function () {
  var formEl = document.querySelector("section#Book-U > form"),
      authorsSelWidget = formEl.querySelector(
          ".MultiSelectionWidget"),
      key = formEl.selectBook.value,
      book=null;
  if (key !== "") {
    book = Book.instances[key];
    formEl.isbn.value = book.isbn;
    formEl.title.value = book.title;
    formEl.year.value = book.year;
    // set up a multi-select widget for associated authors
    util.createMultiSelectionWidget( authorsSelWidget,
        book.authors, Author.instances, "authorId", "name");
    // assign associated publisher to index of select element
    formEl.selectPublisher.selectedIndex =
        (book.publisher) ? book.publisher.index : 0;
  } else {
    formEl.reset();
    formEl.selectPublisher.selectedIndex = 0;
  }
},
```

When the user, after updating some values, finally clicks the submit button, all form control values, including the value of the single-`select` control for assigning a publisher, are copied to corresponding slots in a `slots` record variable, which is used as the argument for invoking the `update` method after all values have been checked for validity. Before invoking `update`, a list of ID references to authors to be added, and another list of ID references to authors to be removed, is created (in the `authorsIdRefToAdd` and `authorsIdRefToRemove` slots) from the updates that have been recorded in the associated authors selection widget with the help of `classList` values, as shown in the following program listing:

```
handleSubmitButtonClickEvent: function () {
  var i=0, assocAuthorListItemEl=null,
      authorsIdRefToAdd=[], authorsIdRefToRemove=[],
      formEl = document.querySelector("section#Book-U > form"),
      authorsSelWidget =
          formEl.querySelector(".MultiSelectionWidget"),
      authorsAssocListEl = authorsSelWidget.firstElementChild;
  var slots = { isbn: formEl.isbn.value,
        title: formEl.title.value,
        year: formEl.year.value,
        publisherIdRef: formEl.selectPublisher.value
      };
  // commit the update only if all of the form fields values are valid
  if (formEl.checkValidity()) {
    // construct authorsIdRefToAdd and authorsIdRefToRemove
    for (i=0; i < authorsAssocListEl.children.length; i++) {
      assocAuthorListItemEl = authorsAssocListEl.children[i];
      if (assocAuthorListItemEl.classList.contains("removed")) {
        authorsIdRefToRemove.push(
            assocAuthorListItemEl.getAttribute("data-value"));
      }
      if (assocAuthorListItemEl.classList.contains("added")) {
        authorsIdRefToAdd.push(
            assocAuthorListItemEl.getAttribute("data-value"));
      }
    }
    // if the add/remove list is non-empty create a corresponding slot
    if (authorsIdRefToRemove.length > 0) {
      slots.authorsIdRefToRemove = authorsIdRefToRemove;
    }
    if (authorsIdRefToAdd.length > 0) {
      slots.authorsIdRefToAdd = authorsIdRefToAdd;
    }
    Book.update( slots);
  }
}
```

6. How to Run the App, Get the Code and Ask Questions

You can run the example app [UnidirectionalAssociationApp/index.html] from our server, download it as a ZIP archive file [UnidirectionalAssociationApp.zip] and ask questions about the concept of unidirectional associations and how to implement them on our discussion forum [http://web-engineering.info/forum/14].

7. Points of Attention

Notice that in this tutorial, we have made the assumption that all application data can be loaded into main memory (like all book data is loaded into the map `Book.instances`). This approach only works in the case of local data storage of smaller databases, say, with not more than 2 MB of data, roughgly corresponding to 10 tables with an average population of 1000 rows, each having an average size of 200 Bytes. When larger databases are to be managed, or when data is stored remotely, it's no longer possible to load the entire population of all tables into main memory, but we have to use a technique where only parts of the table contents are loaded.

We have still included the repetitive code structures (called ***boilerplate code***) in the model layer per class and per property for constraint validation (checks and setters) and per class for the data storage management methods `add`, `update`, and `destroy`. While it is good to write this code a few times for learning app development, you don't want to write it again and again later when you work on real projects. In Part 6, we will present an approach how to put these methods in a generic form in a meta-class called `mODELcLASS`, such that they can be reused in all model classes of an app.

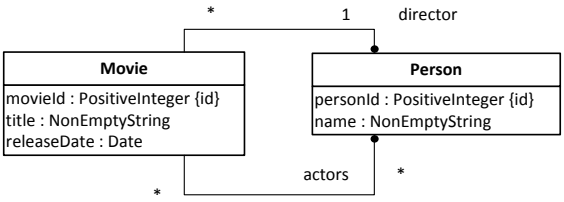
8. Practice Project

This project is based on the information design model shown below. The app from the previous assignments is to be extended by adding the possibility to manage data about the actors and the director of a movie. This is achieved by adding a model class `Person` and two unidirectional associations between `Movie` and `Person`:

1. a many-to-one association assigning exactly one person as the **director** of a movie, and

2. a many-to-many association assigning zero or more persons as the **actors** of a movie.

Figure 3.1. Two unidirectional associations between `Movie` and `Person`.



This project includes the following tasks:

1. Make an ***association-free design model*** derived from the given design model.

2. Make a ***JavaScript data model*** derived from the association-fee design model.

3. Encode your JavaScript data model, following the guidelines of the tutorial.

You can use the following sample data for testing your app:

Table 3.4. Movies

Movie ID	Title	Release date	Director	Actors
1	Pulp Fiction	1994-05-12	3	5, 6
2	Star Wars	1977-05-25	2	7, 8
3	Dangerous Liaisons	1988-12-16	1	9, 5

Table 3.5. People

Person ID	Name
1	Stephen Frears
2	George Lucas
3	Quentin Tarantino
5	Uma Thurman
6	John Travolta
7	Ewan McGregor
8	Natalie Portman
9	Keanu Reeves

Make sure that your pages comply with the XML syntax of HTML5, and that your JavaScript code complies with our Coding Guidelines [http://oxygen.informatik.tu-cottbus.de/webeng/Coding-Guidelines.html] and is checked with JSLint (http://www.jslint.com [http://www.jslint.com/]).

If you have any questions about this project, you can ask them on our discussion forum [http://web-engineering.info/forum/14].