Engineering Front-End Web Apps with Plain JavaScript

An incremental five-part in-depth tutorial about engineering front-end web applications with plain JavaScript and the Local Storage API, not using any (third-party) framework or library

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

Engineering Front-End Web Apps with Plain JavaScript: An incremental five-part in-depth tutorial about engineering front-end web applications with plain JavaScript and the Local Storage API, not using any (third-party) framework or library

by Gerd Wagner

Warning: This textbook is a draft version, so it 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 textbook is also available in the following formats: PDF [complete-tutorial.pdf]. See also the project page [http://web-engineering.info/index.html].

Publication date 2015-02-24

Acknowledgements

Thanks to Sorin Carbunaru and Felix Lehmann for proof-reading early versions of parts of this tutorial book.

Table of Contents

List of Figures

List of Tables

Foreword

This textbook shows how to build front-end web applications with plain JavaScript, not using any (third-party) framework or library. 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.

Part I. Getting Started

In this first part of the book we summarize the web's foundations and show how to build a front-end web application with minimal effort using plain JavaScript and the Local Storage API. It shows how to build such an app with minimal effort, not using any (third-party) framework or library. A front-end web app 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 app is a single-user application, which is not shared with other users.

The minimal version of a JavaScript front-end data management application discussed in this tutorial only includes a minimum of the overall functionality required for a complete app. It takes care of only one object type ("books") and supports the four standard data management operations (**C**reate/**R**ead/**U**pdate/**D**elete), but it needs to be enhanced by styling the user interface with CSS rules, and by adding further important parts of the app's overall functionality.

Table of Contents

Chapter 1. A Quick Tour of the Foundations of Web Apps

If you are already familiar with HTML, XML and JavaScript, you can skip this chapter and immediately start developing a minimal front-end web application with JavaScript in the following chapter.

The World Wide Web (WWW)

After the Internet had been established in the 1980'ies, Tim Berners-Lee [http://en.wikipedia.org/wiki/Tim_Berners-Lee] developed the idea and the first infrastructure components of the WWW in 1989 at the European research institution CERN in Geneva, Switzerland. The WWW (or, simply, "the web") is based on

- the basic Internet technologies TCP/IP and DNS,
- the *Hypertext Transfer Protocol (HTTP)*,
- the *Hypertext Markup Language (HTML)* as well as the *Extensible Markup Language (XML)*, and
- web server programs, acting as HTTP servers, as well as web 'user agents' (such as browsers), acting as HTTP clients.

HTML and XML

HTML allows to mark up (or describe) the structure of a human-readable web document or web user interface, while XML allows to mark up the structure of all kinds of documents, data files and messages, whether they are human-readable or not. HTML can be based on XML.

XML documents

XML provides a syntax for expressing structured information in the form of an *XML document* with *elements* and their *attributes*. The specific elements and attributes used in an XML document can come from any vocabulary, such as public standards or your own user-defined XML format. XML is used for specifying

- **document formats**, such as *XHTML5*, the *Scalable Vector Graphics* (SVG) format or the *DocBook* format,
- **data interchange file formats**, such as the *Mathematical Markup Language* (MathML) or the *Universal Business Language* (UBL),
- **message formats**, such as the web service message format SOAP [http://www.w3.org/TR/soap12-part0/]

Unicode and UTF-8

XML is based on Unicode, which is a platform-independent character set that includes almost all characters from most of the world's script languages including Hindi, Burmese and Gaelic. Each character is assigned a unique integer code in the range between 0 and 1,114,111. For example, the Greek letter π has the code 960, so it can be inserted in an XML document as `&#960;` (using the *XML entity* syntax).

Unicode includes legacy character sets like ASCII and ISO-8859-1 (Latin-1) as subsets.

The default encoding of an XML document is UTF-8, which uses only a single byte for ASCII characters, but three bytes for less common characters.

Almost all Unicode characters are legal in a well-formed XML document. Illegal characters are the control characters with code 0 through 31, except for the *carriage return*, *line feed* and *tab*. It is therefore dangerous to copy text from another (non-XML) text to an XML document (often, the *form feed* character creates a problem).

XML namespaces

Generally, namespaces help to avoid name conflicts. They allow to reuse the same (local) name in different namespace contexts.

XML namespaces are identified with the help of a *namespace URI* (such as the SVG namespace URI "http://www.w3.org/2000/svg"), which is associated with a *namespace prefix* (such as "svg"). Such a namespace represents a collection of names, both for elements and attributes, and allows namespace-qualified names of the form *prefix:name* (such as "svg:circle" as a namespace-qualified name for SVG circle elements).

A default namespace is declared in the start tag of an element in the following way:

```
<html xmlns="http://www.w3.org/1999/xhtml">
```

This example shows the start tag of the HTML root element, in which the XHTML namespace is declared as the default namespace.

The following example shows a namespace declaration for the SVG namespace:

```
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
   ...
  </head>
  <body>
    <figure>
      <figcaption>Figure 1: A blue circle</figcaption>
      <svg:svg xmlns:svg="http://www.w3.org/2000/svg">
        <svg:circle cx="100" cy="100" r="50" fill="blue"/>
      </svg:svg>
    </figure>
  </body>
</html>
```

Correct XML documents

XML defines two syntactic correctness criteria. An XML document must be *well-formed*, and if it is based on a grammar (or schema), then it must also be *valid* against that grammar.

An XML document is called ***well-formed***, if it satisfies the following syntactic conditions:

1. There must be exactly one root element.

2. Each element has a start tag and an end tag; however, empty elements can be closed as `<phone/>` instead of `<phone></phone>`.

3. Tags don't overlap, e.g. we cannot have

   ```
   <author><name>Lee Hong</author></name>
   ```

4. Attribute names are unique within the scope of an element, e.g. the following code is not correct:

```
<attachment file="lecture2.html" file="lecture3.html"/>
```

An XML document is called ***valid*** against a particular grammar (such as a *DTD* or an *XML Schema*), if

1. it is *well-formed*,
2. and it *respects the grammar*.

The evolution of HTML

The World-Wide Web Committee (W3C) has developed the following important versions of HTML:

- HTML4 as an SGML-based language (in 1997),
- XHTML 1 as an XML-based version of HTML4 (in 2000),
- *(X)**HTML5*** in cooperation (and competition) with the WHAT working group [http://en.wikipedia.org/wiki/WHATWG] led by Ian Hickson [http://en.wikipedia.org/wiki/Ian_Hickson] and supported by browser vendors (in 2014).

HTML was originally designed as a *structure* description language, and not as a *presentation* description language. But HTML4 has a lot of purely presentational elements such as `font`. XHTML has been taking HTML back to its roots, dropping presentational elements and defining a simple and clear syntax, in support of the goals of

- device independence,
- accessibility, and
- usability.

We adopt the symbolic equation

> ***HTML = HTML5 = XHTML5***
>
> *stating that when we say "HTM"L or "HTML"5, we actually mean XHTML5*

because we prefer the clear syntax of XML documents over the liberal and confusing HTML4-style syntax that is also allowed by HTML5.

The following simple example shows the basic code template to be used for any HTML document:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
 <head>
  <meta charset="UTF-8" />
  <title>XHTML5 Template Example</title>
 </head>
 <body>
  <h1>XHTML5 Template Example</h1>
  <section><h1>First Section Title</h1>
   ...
  </section>
 </body>
</html>
```

HTML forms

For user-interactive web applications, the web browser needs to render a user interface. The traditional metaphor for a software application's user interface is that of a *form*. The special elements for data

input, data output and form actions are called *form controls*. An HTML form is a section of a document consisting of block elements that contain controls and *labels* on those controls.

Users complete a form by entering text into *input fields* and by selecting items from *choice controls*. A completed form is submitted with the help of a *submit button*. When a user submits a form, it is sent to a web server either with the HTTP GET method or with the HTTP POST method. The standard encoding for the submission is called *URL-encoded*. It is represented by the Internet media type `application/x-www-form-urlencoded`. In this encoding, spaces become plus signs, and any other reserved characters become encoded as a percent sign and hexadecimal digits, as defined in RFC 1738.

Each control has both an initial value and a current value, both of which are strings. The initial value is specified with the control element's `value` attribute, except for the initial value of a `textarea` element, which is given by its initial contents. The control's current value is first set to the initial value. Thereafter, the control's current value may be modified through user interaction or scripts. When a form is submitted for processing, some controls have their name paired with their current value and these pairs are submitted with the form.

Labels are associated with a control by including the control as a subelement of a `label` element ("implicit labels"), or by giving the control an `id` value and referencing this id in the `for` attribute of the `label` element ("explicit labels"). Notice that implicit labels are, in 2014, still not well supported by CSS libraries and assistive technologies. Therefore, explicit labels seem preferable, despite the fact that they imply quite some overhead and repetitive code.

In the simple user interfaces of our "Getting Started" applications, we only need three types of form controls:

1. *single line input fields* created with an `<input name="..." />` element,

2. *push buttons* created with a `<button type="button">...</button>` element, and

3. *dropdown selection lists* created with a `select` element of the following form:

```
<select name="...">
  <option value="value1"> option1 </option>
  <option value="value2"> option2 </option>
  ...
</select>
```

An example of an HTML form with implicit labels for creating such a user interface is

```
<form id="Book">
  <p><label>ISBN: <input name="isbn" /></label></p>
  <p><label>Title: <input name="title" /></label></p>
  <p><label>Year: <input name="year" /></label></p>
  <p><button type="button" id="saveButton">Save</button></p>
</form>
```

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 kinds of **`input` fields**. This mapping is also called **data binding**.

In general, an attribute of a model class can always be represented in the UI by a plain `input` control (with the default setting `type="text"`), no matter which datatype has been defined as the range of the attribute in the model class. However, in special cases, other types of `input` controls (for instance, `type="date"`), or other controls, may be used. For instance, if the attribute's range is an enumeration, a `select` control or, if the number of possible choices is small enough (say, less than 8), a radio button group can be used.

JavaScript

This section provides a brief overview of JavaScript, assuming that the reader is already familiar with basic programming concepts and has some experience with programming, for instance, in PHP, Java or C#.

JavaScript is an object-oriented programming language that can be used for

1. Enriching a web page by
   - generating browser-specific HTML content or CSS styling,
   - inserting dynamic HTML content,
   - producing special audio-visual effects (animations).
2. Enriching a web user interface by
   - implementing advanced user interface components,
   - validating user input on the client side,
   - automatically pre-filling certain form fields.
3. Implementing a frontend web application with local data storage.
4. Implementing a frontend component for a distributed web application with remote data storage managed by a backend component (server-side program).

Types and data literals in JavaScript

JavaScript has three primitive data types: `string`, `number` and `boolean`. There are three reference types: `object`, `array` and `function`. Arrays and functions are just special kinds of objects. Types are not declared and not checked since a JavaScript program is not compiled. Type conversion (casting) is performed automatically.

The value of a variable may be

- a *data value*: either a string, a number, or a boolean,
- an *object reference*: either referencing an ordinary object, or an array, or a function,
- one of the following two special data values: `undefined` or `null`.

All numeric data values are represented in 64-bit floating point format with an optional exponent (like in the numeric data literal `3.1e10`). There is no explicit type distinction between integers and floating point numbers. For making sure that a numeric value is an integer, or that a string representing a number is converted to an integer, one has to apply the predefined function `parseInt`. If a numeric expression cannot be evaluated to a number, its value is set to `NaN` ("not a number").

Like in Java, there are two pre-defined Boolean data literals, `true` and `false`, and the Boolean operator symbols are the exclamation mark `!` for NOT, the double ampersand `&&` for AND, and the double bar `||` for OR. When a non-Boolean value is used in a condition, or as an operand of a Boolean expression, it is converted into a Boolean value according to the following rules. The empty string, the (numerical) data literal 0, as well as `undefined` and `null`, are mapped to *false*, and all other values are mapped to `true`.

For equality and inequality testing, always use the triple equality symbol `===` and `!==` instead of the double equality symbol `==` and `!=`. Otherwise, for instance, the number 2 would be the same as the string "2", since the condition `(2 == "2")` evaluates to true in JavaScript.

Variable scope

In the current version of JavaScript, ECMAScript 5, there are only two kinds of scope for variables: the global scope (with `window` as the context object) and function scope, but **no block scope**. Consequently, declaring a variable within a block is confusing and should be avoided. For instance, although this is a freqently used pattern, even by experienced JavaScript programmers, it is a pitfall to declare the counter variable of a `for` loop in the loop, as in

```
function foo() {
  for (var i=0; i < 10; i++) {
    ...  // do something with i
  }
}
```

Instead, and this is exactly how JavaScript is interpreting this code, we should write:

```
function foo() {
  var i=0;
  for (i=0; i < 10; i++) {
    ...  // do something with i
  }
}
```

All variables should be declared at the beginning of a function. Only in the next version of JavaScript, ECMAScript 6, block scope will be supported by means of a new form of variable declaration with the keyowrd `let`.

Strict Mode

Starting from ECMAScript 5, we can use strict mode [http://speakingjs.com/es5/ch07.html#strict_mode] for getting more runtime error checking. For instance, in strict mode, all variables must be declared. An assignment to an undeclared variable throws an exception.

We can turn strict mode on by typing the following statement as the first line in a JavaScript file or inside a `<script>` element:

```
'use strict';
```

It is generallly recommended that you use strict mode, except your code depends on libraries that are incompatible with strict mode.

Different kinds of objects

JavaScript objects are different from classical OO/UML objects. In particular, they **need not instantiate a class**. And they can have their own (instance-level) methods in the form of method slots, so they do not only have (ordinary) **property slots**, but also **method slots**. In addition they may also have **key-value slots**. So, they may have three different kinds of slots, while classical objects (called "instance specifications" in UML) only have property slots.

A JavaScript object is essentially a set of name-value-pairs, also called ***slots***, where names can be *property* names, *function* names or *keys* of a map. Objects can be created in an ad-hoc manner, using JavaScript's object literal notation (JSON), without instantiating a class:

```
var person1 = { lastName:"Smith", firstName:"Tom"};
var o1 = {};  // an empty object with no slots
```

Whenever the name in a slot is an admissible JavaScript identifier [http://mothereff.in/js-variables], the slot may be either a *property slot*, a *method slot* or a *key-value slot*. Otherwise, if the name is some

other type of string (in particular when it contains any blank space), then the slot represents a *key-value slot*, which is a map element, as explained below.

The name in a **property slot** may denote either

1. a **data-valued property**, in which case the value is a *data value* or, more generally, a *data-valued expression*;

   or

2. an **object-valued property**, in which case the value is an *object reference* or, more generally, an *object expression*.

The name in a **method slot** denotes a *JavaScript function* (better called *method*), and its value is a function definition text.

Object properties can be accessed in two ways:

1. Using the dot notation (like in C++/Java):

```
person1.lastName = "Smith"
```

2. Using a map notation:

```
person1["lastName"] = "Smith"
```

JavaScript objects can be used in many different ways for different purposes. Here are five different use cases for, or possible meanings of, JavaScript objects:

1. A ***record*** is a set of property slots like, for instance,

```
var myRecord = {firstName:"Tom", lastName:"Smith", age:26}
```

2. A ***map*** (or 'associative array') supports look-ups of *values* based on *keys* like, for instance,

```
var numeral2number = {"one":"1", "two":"2", "three":"3"}
```

   which associates the value "1" with the key "one", "2" with "two", etc. A key need not be a valid JavaScript identifier, but can be any kind of string (e.g. it may contain blank spaces).

3. An ***untyped object*** does not instantiate a class. It may have property slots and function slots like, for instance,

```
var person1 = {
  lastName: "Smith",
  firstName: "Tom",
  getInitials: function () {
    return this.firstName.charAt(0) + this.lastName.charAt(0);
  }
};
```

4. A ***namespace*** may be defined in the form of an untyped object referenced by a global object variable, the name of which represents a namespace prefix. For instance, the following object variable provides the main namespace of an application based on the Model-View-Controller (MVC) architecture paradigm where we have three subnamespaces corresponding to the three parts of an MVC application:

```
var myApp = { model:{}, view:{}, ctrl:{} };
```

5. A ***typed object*** `o` instantiates a class that is defined either by a JavaScript constructor function `C` or by a factory object `F`. See ???

Array lists

A JavaScript array represents, in fact, the logical data structure of an *array list*, which is a list where each list item can be accessed via an index number (like the elements of an array). Using the term 'array' without saying 'JavaScript array' creates a terminological ambiguity. But for simplicity, we will sometimes just say 'array' instead of 'JavaScript array'.

A variable may be initialized with a JavaScript *array literal*:

```
var a = [1,2,3];
```

Because they are array lists, JavaScript arrays can grow dynamically: it is possible to use indexes that are greater than the length of the array. For instance, after the array variable initialization above, the array held by the variable `a` has the length 3, but still we can assign a fifth array element like in

```
a[4] = 7;
```

The contents of an array `a` are processed with the help of a standard *for* loop with a counter variable counting from the first array index 0 to the last array index, which is `a.length-1`:

```
for (i=0; i < a.length; i++) { ...}
```

Since arrays are special types of objects, we sometimes need a method for finding out if a variable represents an array. We can test, if a variable `a` represents an array by applying the predefined datatype predicate `isArray` as in `Array.isArray( a)`.

For **adding** a new element to an array, we append it to the array using the `push` operation as in:

```
a.push( newElement);
```

For **deleting** an element at position `i` from an array `a`, we use the pre-defined array method `splice` as in:

```
a.splice( i, 1);
```

For **searching** a value `v` in an array `a`, we can use the pre-defined array method `indexOf`, which returns the position, if found, or -1, otherwise, as in:

```
if (a.indexOf(v) > -1)  ...
```

Maps

A map (also called 'hash map' or 'associative array') provides a mapping from keys to their associated values. The keys of a map may be string literals that include blank spaces like in:

```
var myTranslation = {
    "my house": "mein Haus",
    "my boat": "mein Boot",
    "my horse": "mein Pferd"
}
```

A map is processed with the help of a special loop where we loop over all keys of the map using the pre-defined function `Object.keys(m)`, which returns an array of all keys of a map `m`. For instance,

```
var i=0, key="", keys=[];
keys = Object.keys( myTranslation);
for (i=0; i < keys.length; i++) {
  key = keys[i];
  alert('The translation of '+ key +' is '+ myTranslation[key]);
```

```
}
```

For **adding** a new entry to a map, we associate the new value with its key as in:

```
myTranslation["my car"] = "mein Auto";
```

For **deleting** an entry from a map, we can use the pre-defined `delete` operator as in:

```
delete myTranslation["my boat"];
```

For **testing** if a map `m` has an entry for a certain key `k`, we can check the following:

```
if (m[k])  ...
```

JavaScript supports four types of basic data structures

In summary, the four types of basic data structures supported are:

1. ***array lists***, such as `["one","two","three"]`, which are special JS objects called 'arrays', but since they are dynamic, they are rather *array lists* as defined in the *Java* programming language.

2. ***maps***, which are also special JS objects, such as `{"one":1,"two":2,"three":3}`, as discussed above,

3. ***records***, which are special JS objects, such as `{firstName:"Tom",lastName:"Smith"}`, as discussed above,

4. **JSON tables**, which are special maps where the values are entity records (with a primary key slot), and the keys are the primary keys of these entity records.

Notice that our distinction between maps, records and JSON tables is a purely conceptual distinction, and not a syntactical one. For a JavaScript engine, both `{firstName:"Tom",lastName:"Smith"}` and `{"one":1,"two":2,"three":3}` are just objects. But conceptually, `{firstName:"Tom",lastName:"Smith"}` is a record because `firstName` and `lastName` are intended to denote properties or fields, while `{"one":1,"two":2,"three":3}` is a map because `"one"` and `"two"` are not intended to denote properties/fields, but are just arbitrary string values used as keys for a map.

Making such conceptual distinctions helps to better understand the options offered by JavaScript.

Defining and using classes

The concept of a **class** is fundamental in *object-oriented* programming. Objects *instantiate* (or *are classified by*) a class. A class defines the properties and methods for the objetcs that instantiate it. Having a class concept is essential for being able to implement a data model in the form of model classes. However, classes and their inheritance/extension mechanism are over-used in classical OO languages, such as in Java, where all code has to live in the context of a class. This is not the case in JavaScript where we have the freedom to use classes for implementing model classes only, while keeping method libraries in namespace objects.

There is no explicit class concept in JavaScript. However, classes can be defined in two ways:

1. In the form of a **constructor** function that allows to create new instances of the class with the help of the `new` operator. This is the classical approach recommended in the Mozilla JavaScript documents.

2. In the form of a **factory** object that uses the predefined `Object.create` method for creating new instances of the class. In this approach, the constructor-based inheritance mechanism has to be replaced by another mechanism. Eric Elliott [http://chimera.labs.oreilly.com/books/1234000000262/ch03.html#fluentstyle_javascript] has argued that factory-based classes are

a viable alternative to constructor-based classes in JavaScript (in fact, he even condemns the use of classical inheritance and constructor-based classes, throwing out the baby with the bath water).

Since we often need to define class hierarchies, and not just single classes, these two alternative approaches cannot be mixed within the same class hierarchy, and we have to make a choice whenever we build an app. Their pros and cons are summarized in Table 1.1.

Table 1.1. Constructor-based versus factory-based classes

	Pros	**Cons**
Constructor-based classes	Higher performance object creation	1. Do not allow to declare properties 2. Incompatible with object pools 3. Do not support multiple inheritance 4. Do not support multiple classification
Factory-based classes	1. Allow property declarations (with a property label, a range and many other constraints) 2. Can support object pools 3. Can support multiple inheritance 4. Can support multiple classification	Lower performance object creation

Constructor-based classes

A constructor-based class can be defined in two or three steps. First define the constructor function that defines the properties of the class and assigns them the values of the constructor parameters:

```
function Person( first, last) {
  this.firstName = first;
  this.lastName = last;
}
```

Next, define the *instance-level methods* of the class as method slots of the object property `prototype` of the constructor:

```
Person.prototype.getInitials = function () {
  return this.firstName.charAt(0) + this.lastName.charAt(0);
}
```

Finally, *class-level* ("static") *methods* can be defined as method slots of the constructor, as in

```
Person.checkName = function (n) {
  ...
}
```

An instance of such a constructor-based class is created by applying the `new` operator to the constructor function and providing suitable arguments for the constructor parameters:

```
var pers1 = new Person("Tom","Smith");
```

The method `getInitials` is invoked on the object `pers1` of type `Person` by using the 'dot notation':

```
alert("The initials of the person are: " + pers1.getInitials());
```

When a typed object `o` is created with `o = new C(...)`, where `C` references a named function with name "C", the type (or class) name of `o` can be retrieved with the introspective expression `o.constructor.name`. which returns "C" (however the function `name` property is not yet supported by Internet Explorer up to the current version 11).

In this approach, an **inheritance** mechanism is provided via the predefined `prototype` property of a constructor function. This will be explained in Part 5 of this tutorial (on *subtyping*).

Factory-based classes

In this approach we define a JavaScript object `Person` (actually representing a class) with a special `create` method that invokes the predefined `Object.create` method for creating objects of type `Person`:

```
var Person = {
  typeName: "Person",
  properties: {
    firstName: {range:"NonEmptyString", label:"First name",
        writable: true, enumerable: true},
    lastName: {range:"NonEmptyString", label:"Last name",
        writable: true, enumerable: true}
  },
  methods: {
    getInitials: function () {
      return this.firstName.charAt(0) + this.lastName.charAt(0);
    }
  },
  create: function (slots) {
    var obj=null, properties = this.properties;
    // create object
    obj = Object.create( this.methods, properties);
    // add property slot for direct type
    Object.defineProperty( obj, "type",
        {value: this, writable: false, enumerable: true});
    // initialize object
    Object.keys( slots).forEach( function (prop) {
      if (properties[prop]) obj[prop] = slots[prop];
    })
    return obj;
  }
};
```

Notice that our `Person` object actually represents a factory-based class. An instance of such a factory-based class is created by invoking its `create` method:

```
var pers1 = Person.create( {firstName:"Tom", lastName:"Smith"});
```

The method `getInitials` is invoked on the object `pers1` of type `Person` by using the 'dot notation', like in the constructor-based approach:

```
alert("The initials of the person are: " + pers1.getInitials());
```

Notice that each property declaration for an object created with `Object.create` has to include the 'descriptors' `writable: true` and `enumerable: true`, as in lines 5 and 7 of the `Person` object definition above.

In a general approach, like in the model-based development framework mODELcLASSjs presented in part 6 of this tutorial, we would not repeatedly define the `create` method in each class definition, but rather have a generic constructor function for defining factory-based classes. Such a factory class constructor, like mODELcLASS, would also provide an **inheritance** mechanism by merging the own properties and methods with the properties and methods of the superclass.

JavaScript as an object-oriented language

JavaScript is *object-oriented*, but in a different way than classical OO programming languages such as Java and C++. There is no explicit *class* concept in JavaScript. Rather, classes have to be defined in the form of special objects: either as *constructor* functions or as *factory* objects.

However, objects can also be created without instantiating a class, in which case they are *untyped*, and properties as well as methods can be defined for specific objects independently of any class definition. At run time, properties and methods can be added to, or removed from, any object and class.

Further reading about JavaScript

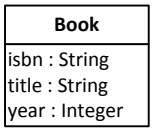
Good open access books about JavaScript are

- Speaking JavaScript [http://speakingjs.com/es5/index.html], by Dr. Axel Rauschmayer.
- Eloquent JavaScript [http://eloquentjavascript.net/], by Marijn Haverbeke.

Chapter 2. Building a Minimal JavaScript Front-End App in Seven Steps

In this chapter, we build a minimal front-end web application with plain JavaScript and Local Storage. The purpose of our example app is to manage information about books. That is, we deal with a single object type: `Book`, as depicted in Figure 2.1.

Figure 2.1. The object type `Book`.



What do we need for such an information management application? There are four standard use cases, which have to be supported by the application:

1. **Create**: Enter the data of a book that is to be added to the collection of managed books.
2. **Read**: Show a list of all books in the collection of managed books.
3. **Update** the data of a book.
4. **Delete** a book record.

For entering data with the help of the keyboard and the screen of our computer, we can use ***HTML forms***, which provide the **user interface** technology for web applications.

For maintaining a collection of data objects, we need a storage technology that allows to keep data objects in persistent records on a secondary storage device, such as a harddisk or a solid state disk. Modern web browsers provide two such technologies: the simpler one is called *Local Storage*, and the more powerful one is called *IndexDB*. For our minimal example app, we use Local Storage.

Step 1 - Set up the Folder Structure

In the first step, we set up our folder structure for the application. We pick a name for our app, such as "Public Library", and a corresponding (possibly abbreviated) name for the application folder, such as "publicLibrary". Then we create this folder on our computer's disk and a subfolder "src" for our JavaScript source code files. In this folder, we create the subfolders "model", "view" and "ctrl", following the *Model-View-Controller* paradigm for software application architectures. And finally we create an `index.html` file for the app's start page, as discussed below. Thus, we end up with the following folder structure:

```
publicLibrary
  src
    ctrl
    model
    view
  index.html
```

The start page of the app loads the `Book.js` model class file and provides a menu for choosing one of the CRUD data management operations performed by a corresponding page such as, for instance, `createBook.html`, or for creating test data with the help of the procedure `Book.createTestData()` in line 17, or clearing all data with `Book.clearData()` in line 18:

Figure 2.2. The minimal app's start page `index.html`.

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Minimal JS Front-End App Example</title>
  <script src="src/model/Book.js"></script>
</head>
<body>
  <h1>Public Library</h1>
  <h2>An Example of a Minimal JavaScript Front-End App</h2>
  <p>This app supports the following operations:</p>
  <menu>
    <li><a href="listBooks.html"><button type="button">
      List all books
    </button></a></li>
    <li><a href="createBook.html"><button type="button">
      Add a new book
    </button></a></li>
    <li><a href="updateBook.html"><button type="button">
      Update a book
    </button></a></li>
    <li><a href="deleteBook.html"><button type="button">
      Delete a book
    </button></a></li>
    <li><button type="button" onclick="Book.clearData()">
      Clear database
    </button></li>
    <li><button type="button" onclick="Book.createTestData()">
      Create test data
    </button></li>
  </menu>
</body>
</html>
```

Step 2 - Write the Model Code

In the second step, we write the code of our model class and save it in a specific model class file. In an MVC app, the model code is the most important part of the app. It's also the basis for writing the view and controller code. In fact, large parts of the view and controller code could be automatically generated from the model code. Many MVC frameworks provide this kind of code generation.

In the information design model shown in Figure 2.1 above, there is only one class, representing the object type `Book`. So, in the folder `src/model`, we create a file `Book.js` that initially contains the following code:

```
function Book( slots) {
  this.isbn = slots.isbn;
  this.title = slots.title;
  this.year = slots.year;
};
```

The model class `Book` is encoded as a JavaScript constructor function with a single `slots` parameter, which is supposed to be a record object with properties `isbn`, `title` and `year`, representing values for the *ISBN*, the *title* and the *year* attributes of the class `Book`. Therefore, in the constructor function, the values of the `slots` properties are assigned to the corresponding attributes whenever a new object is created as an instance of this class.

In addition to defining the model class in the form of a constructor function, we also define the following items in the `Book.js` file:

1. A class-level property `Book.instances` representing the collection of all Book instances managed by the application in the form of a JSON table.
2. A class-level method `Book.loadAll` for loading all managed Book instances from the persistent data store.
3. A class-level method `Book.saveAll` for saving all managed Book instances to the persistent data store.
4. A class-level method `Book.add` for creating a new Book instance.
5. A class-level method `Book.update` for updating an existing Book instance.
6. A class-level method `Book.destroy` for deleting a Book instance.
7. A class-level method `Book.createTestData` for creating a few example book records to be used as test data.
8. A class-level method `Book.clearData` for clearing the book datastore.

Representing the collection of all Book instances

For representing the collection of all Book instances managed by the application, we define and initialize the class-level property `Book.instances` in the following way:

```
Book.instances = {};
```

So, initially our collection of books is empty. In fact, it's defined as an empty object literal, since we want to represent it in the form of a JSON table (a map of records) where an ISBN is a key for accessing the corresponding book record (as the value associated with the key). We can visualize the structure of a JSON table in the form of a lookup table, as shown in Table 2.1, “A JSON table representing a collection of books”.

Table 2.1. A JSON table representing a collection of books

Key	Value
006251587X	{ isbn:"006251587X," title:"Weaving the Web", year:2000 }
0465026567	{ isbn:"0465026567," title:"Gödel, Escher, Bach", year:1999 }
0465030793	{ isbn:"0465030793," title:"I Am A Strange Loop", year:2008 }

Notice that the values of such a map are records corresponding to table rows. Consequently, we could also represent them in a simple table, as shown in Table 2.2, “A collection of book objects represented as a table”.

Table 2.2. A collection of book objects represented as a table

ISBN	Title	Year
006251587X	Weaving the Web	2000
0465026567	Gödel, Escher, Bach	1999
0465030793	I Am A Strange Loop	2008

Loading all Book instances

For persistent data storage, we use the *Local Storage* API supported by modern web browsers. Loading the book records from Local Storage involves three steps:

1. Retrieving the book table that has been stored as a large string with the key "books" from Local Storage with the help of the assignment

   ```
   booksString = localStorage["books"];
   ```

   This retrieval is performed in line 5 of the program listing below.

2. Converting the book table string into a corresponding JSON table `books` with book rows as elements, with the help of the built-in procedure `JSON.parse`:

   ```
   books = JSON.parse( booksString);
   ```

   This conversion, performed in line 11 of the program listing below, is called *deserialization*.

3. Converting each row of `books`, representing a record (an untyped object), into a corresponding object of type `Book` stored as an element of the JSON table `Book.instances`, with the help of the procedure `convertRow2Obj` defined as a "static" (class-level) method in the `Book` class:

   ```
   Book.convertRow2Obj = function (bookRow) {
     var book = new Book( bookRow);
     return book;
   };
   ```

Here is the full code of the procedure:

```
Book.loadAll = function () {
  var key="", keys=[],
      booksString="", books={};
  try {
    if (localStorage["books"]) {
      booksString = localStorage["books"];
    }
  } catch (e) {
    alert("Error when reading from Local Storage\n" + e);
  }
  if (booksString) {
    books = JSON.parse( booksString);
    keys = Object.keys( books);
    console.log( keys.length +" books loaded.");
    for (i=0; i < keys.length; i++) {
      key = keys[i];
      Book.instances[key] = Book.convertRow2Obj( books[key]);
    }
  }
};
```

Notice that since an input operation like `localStorage["books"]` may fail, we perform it in a try-catch block, where we can follow up with an error message whenever the input operation fails.

Saving all Book instances

Saving all book objects from the `Book.instances` collection in main memory to Local Storage in secondary memory involves two steps:

1. Converting the JSON table `Book.instances` into a string with the help of the predefined JavaScript procedure `JSON.stringify`:

   ```
   booksString = JSON.stringify( Book.instances);
   ```

   This conversion is called *serialization*.

2. Writing the resulting string as the value of the key "books" to Local Storage:

   ```
   localStorage["books"] = booksString;
   ```

These two steps are performed in line 5 and in line 6 of the following program listing:

```
Book.saveAll = function () {
  var booksString="", error=false,
      nmrOfBooks = Object.keys( Book.instances).length;
  try {
    booksString = JSON.stringify( Book.instances);
    localStorage["books"] = booksString;
  } catch (e) {
    alert("Error when writing to Local Storage\n" + e);
    error = true;
  }
  if (!error) console.log( nmrOfBooks + " books saved.");
};
```

Creating a new Book instance

The `Book.add` procedure takes care of creating a new `Book` instance and adding it to the `Book.instances` collection :

```
Book.create = function (slots) {
  var book = new Book( slots);
  Book.instances[slots.isbn] = book;
  console.log("Book " + slots.isbn + " created!");
};
```

Updating an existing Book instance

For updating an existing `Book` instance we first retrieve it from `Book.instances`, and then re-assign those attributes the value of which has changed:

```
Book.update = function (slots) {
  var book = Book.instances[slots.isbn];
  var year = parseInt( slots.year);
  if (book.title !== slots.title) { book.title = slots.title;}
  if (book.year !== year) { book.year = year;}
  console.log("Book " + slots.isbn + " modified!");
};
```

Notice that in the case of a numeric attribute (such as `year`), we have to make sure that the value of the corresponding input parameter (`y`), which is typically obtained from user input via an HTML form, is converted from String to Number with one of the two type conversion functions `parseInt` or `parseFloat`.

Deleting an existing Book instance

A Book instance is deleted from the JSON table `Book.instances` by first testing if the table has a row with the given key (line 2), and then applying the JavaScript built-in `delete` operator:, which deletes a slot from an object, or, in our case, an element from a map:

```
Book.destroy = function (isbn) {
  if (Book.instances[isbn]) {
    console.log("Book " + isbn + " deleted");
    delete Book.instances[isbn];
  } else {
    console.log("There is no book with ISBN " +
        isbn + " in the database!");
  }
};
```

Creating test data

For being able to test our code, we may create some test data and save it in our Local Storage database. We can use the following procedure for this:

```
Book.createTestData = function () {
  Book.instances["006251587X"] = new Book(
      {isbn:"006251587X", title:"Weaving the Web", year:2000});
  Book.instances["0465026567"] = new Book(
      {isbn:"0465026567", title:"Gödel, Escher, Bach", year:1999});
  Book.instances["0465030793"] = new Book(
      {isbn:"0465030793", title:"I Am A Strange Loop", year:2008});
  Book.saveAll();
};
```

Clearing all data

The following procedure clears all data from Local Storage:

```
Book.clearData = function () {
  if (confirm("Do you really want to delete all book data?")) {
    localStorage["books"] = "{}";
  }
};
```

Step 3 - Initialize the Application

We initialize the application by defining its namespace and MVC subnamespaces. Namespaces are an important concept in software engineering and many programming languages, including Java and PHP, provide specific support for namespaces, which help grouping related pieces of code and avoiding name conflicts. Since there is no specific support for namespaces in JavaScript, we use special objects for this purpose (we may call them "namespace objects"). First we define a root namespace (object) for our app, and then we define three subnamespaces, one for each of the three parts of the application code: *model*, *view* and *controller*. In the case of our example app, we may use the following code for this:

```
var pl = { model:{}, view:{}, ctrl:{} };
```

Here, the main namespace is defined to be `pl`, standing for "Public Library", with the three subnamespaces `model`, `view` and `ctrl` being initially empty objects. We put this code in a separate file `initialize.js` in the `ctrl` folder, because such a namespace definition belongs to the controller part of the application code.

Step 4 - Implement the *List Objects* Use Case

This use case corresponds to the "Read" from the four basic data management use cases *Create-Read-Update-Delete (CRUD)*.

The user interface for this use case is provided by the following HTML page containing an HTML table for displaying the book objects. For our example app, this page would be called `listBooks.html` (in the main folder `publicLibrary`) and would contain the following HTML code:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Simple JS Front-End App Example</title>
  <script src="src/ctrl/initialize.js"></script>
  <script src="src/model/Book.js"></script>
  <script src="src/view/listBooks.js"></script>
  <script>
   window.addEventListener( "load",
       pl.view.listBooks.setupUserInterface);
  </script>
</head>
<body>
  <h1>Public Library: List all books</h1>
  <table id="books">
    <thead><tr><th>ISBN</th><th>Title</th><th>Year</th></tr></thead>
    <tbody></tbody>
  </table>
  <nav><a href="index.html">Back to main menu</a></nav>
</body>
</html>
```

Notice that this HTML file loads three JavaScript files: the controller file `src/ctrl/initialize.js`, the model file `src/model/Book.js` and the view file `src/view/listBooks.js`. The first two files contain the code for initializing the app and for the model class `Book` as explained above, and the third one, which represents the UI code of the "list books" operation, is developed now. In fact, for this operation, we just need a procedure for setting up the data management context and the UI, called `setupUserInterface`:

```
 pl.view.listBooks = {
  setupUserInterface: function () {
    var tableBodyEl = document.querySelector("table#books>tbody");
    var keys=[], key="", row={};
    // load all book objects
    Book.loadAll();
    keys = Object.keys( Book.instances);
    // for each book, create a table row with cells for the 3 attributes
    for (i=0; i < keys.length; i++) {
      key = keys[i];
      row = tableBodyEl.insertRow();
      row.insertCell(-1).textContent = Book.instances[key].isbn;
      row.insertCell(-1).textContent = Book.instances[key].title;
      row.insertCell(-1).textContent = Book.instances[key].year;
    }
  }
};
```

The simple logic of this procedure consists of two steps:

1. Read the collection of all objects from the persistent data store (in line 6).

2. Display each object as a row in a HTML table on the screen (in the loop starting in line 9).

More specifically, the procedure `setupUserInterface` first creates the book objects from the corresponding rows retrieved from Local Storage by invoking `Book.loadAll()` and then creates

the view table in a loop over all key-value slots of the JSON table `Book.instances` where each value represents a book object. In each step of this loop, a new row is created in the table body element with the help of the JavaScript DOM operation `insertRow()`, and then three cells are created in this row with the help of the DOM operation `insertCell()`: the first one for the `isbn` property value of the book object, and the second and third ones for its `title` and `year` property values. Both `insertRow` and `insertCell` have to be invoked with the argument -1 for making sure that new elements are appended to the list of rows and cells.

Step 5 - Implement the *Create Object* Use Case

For a data management operation with user input, such as the "create object" operation, an HTML page with an HTML form is required as a user interface. The form has a form field for each attribute of the `Book` class. For our example app, this page would be called `createBook.html` (in the app folder `publicLibrary`) and would contain the following HTML code:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Minimal JS Front-End App Example</title>
  <script src="src/ctrl/initialize.js"></script>
  <script src="src/model/Book.js"></script>
  <script src="src/view/createBook.js"></script>
  <script>
   window.addEventListener("load",
       pl.view.createBook.setupUserInterface);
  </script>
</head>
<body>
  <h1>Public Library: Create a new book record</h1>
  <form id="Book">
    <p><label>ISBN: <input name="isbn" /></label></p>
    <p><label>Title: <input name="title" /></label></p>
    <p><label>Year: <input name="year" /></label></p>
    <p><button type="button" name="commit">Save</button></p>
  </form>
  <nav><a href="index.html">Back to main menu</a></nav>
</body>
</html>
```

The view code file `src/view/createBook.js` contains two procedures:

1. `setupUserInterface` takes care of retrieveing the collection of all objects from the persistent data store and setting up an event handler (`handleSaveButtonClickEvent`) on the save button for handling click button events by saving the user input data;

2. `handleSaveButtonClickEvent` reads the user input data from the form fields and then saves this data by calling the `Book.saveRow` procedure.

```
pl.view.createBook = {
  setupUserInterface: function () {
    var saveButton = document.forms['Book'].commit;
    // load all book objects
    Book.loadAll();
    // Set an event handler for the save/submit button
    saveButton.addEventListener("click",
        pl.view.createBook.handleSaveButtonClickEvent);
```

```
    window.addEventListener("beforeunload", function () {
        Book.saveAll();
    });
  },
  // save user input data
  handleSaveButtonClickEvent: function () {
    var formEl = document.forms['Book'];
    var slots = { isbn: formEl.isbn.value,
        title: formEl.title.value,
        year: formEl.year.value};
    Book.add( slots);
    formEl.reset();
  }
};
```

Step 6 - Implement the *Upate Object* Use Case

Again, we have an HTML page for the user interface (`updateBook.html`) and a view code file (`src/view/updateBook.js`). The form for the UI of the "update object" operation has a selection field for choosing the book to be updated, and a form field for each attribute of the `Book` class. However, the form field for the standard identifier attribute (ISBN) is read-only because we do not allow changing the standard identifier of an existing object.

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Minimal JS Front-End App Example</title>
  <script src="src/ctrl/initialize.js"></script>
  <script src="src/model/Book.js"></script>
  <script src="src/view/updateBook.js"></script>
  <script>
   window.addEventListener("load", pl.view.updateBook.setupUserInterface);
  </script>
</head>
<body>
  <h1>Public Library: Update a book record</h1>
  <form id="Book" action="">
    <p>
      <label>Select book:
        <select name="selectBook"><option value=""> --- </option></select>
      </label>
    </p>
    <p><label>ISBN: <input name="isbn" readonly="readonly" /></label></p>
    <p><label>Title: <input name="title" /></label></p>
    <p><label>Year: <input name="year" /></label></p>
    <p><button type="button" name="commit">Save Changes</button></p>
  </form>
  <nav><a href="index.html">Back to main menu</a></nav>
</body>
</html>
```

Notice that we include a kind of empty option element, with a value of `""` and a display text of `---`, as a default choice in the `selectBook` selection list element. So, by default, the `selectBook` form control's value is undefined, requiring the user to choose one of the available options for defining a value for this element.

The `setupUserInterface` procedure now has to populate the `select` element's option list by loading the collection of all book objects from the persistent data store and creating an option element for each book object:

```
pl.view.updateBook = {
  setupUserInterface: function () {
    var formEl = document.forms['Book'],
        saveButton = formEl.commit,
        selectBookEl = formEl.selectBook;
    var key="", keys=[], book=null, optionEl=null;
    // load all book objects
    Book.loadAll();
    // populate the selection list with books
    keys = Object.keys( Book.instances);
    for (i=0; i < keys.length; i++) {
      key = keys[i];
      book = Book.instances[key];
      optionEl = document.createElement("option");
      optionEl.text = book.title;
      optionEl.value = book.isbn;
      selectBookEl.add( optionEl, null);
    }
    // when a book is selected, populate the form with the book data
    selectBookEl.addEventListener("change", function () {
        var book=null, key = selectBookEl.value;
        if (key) {
          book = Book.instances[key];
          formEl.isbn.value = book.isbn;
          formEl.title.value = book.title;
          formEl.year.value = book.year;
        } else {
          formEl.reset();
        }
    });
    saveButton.addEventListener("click",
        pl.view.updateBook.handleUpdateButtonClickEvent);
    window.addEventListener("beforeunload", function () {
        Book.saveAll();
    });
  },
  handleUpdateButtonClickEvent: function () {
    var formEl = document.forms['Book'];
    var slots = { isbn: formEl.isbn.value,
        title: formEl.title.value,
        year: formEl.year.value
    };
    Book.update( slots);
    formEl.reset();
  }
};
```

Step 7 - Implement the *Delete Object* Use Case

For the "delete object" use case, the UI form just has a selection field for choosing the book to be deleted:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Minimal JS Front-End App Example</title>
  <script src="src/ctrl/initialize.js"></script>
  <script src="src/model/Book.js"></script>
  <script src="src/view/deleteBook.js"></script>
  <script>
   window.addEventListener("load", pl.view.deleteBook.setupUserInterface);
  </script>
</head>
<body>
  <h1>Public Library: Delete a book record</h1>
  <form id="Book">
    <p>
      <label>Select book:
        <select name="selectBook"><option value=""> --- </option></select>
      </label>
    </p>
    <p><button type="button" name="commit">Delete</button></p>
  </form>
  <nav><a href="index.html">Back to main menu</a></nav>
</body>
</html>
```

The view code in `src/view/deleteBook.js` consists of the following two procedures:

```
pl.view.deleteBook = {
  setupUserInterface: function () {
    var deleteButton = document.forms['Book'].commit;
    var selectEl = document.forms['Book'].selectBook;
    var key="", keys=[], book=null, optionEl=null;
    // load all book objects
    Book.loadAll();
    keys = Object.keys( Book.instances);
    // populate the selection list with books
    for (i=0; i < keys.length; i++) {
      key = keys[i];
      book = Book.instances[key];
      optionEl = document.createElement("option");
      optionEl.text = book.title;
      optionEl.value = book.isbn;
      selectEl.add( optionEl, null);
    }
    deleteButton.addEventListener("click",
        pl.view.deleteBook.handleDeleteButtonClickEvent);
    window.addEventListener("beforeunload", function () {
        Book.saveAll();
    });
  },
  handleDeleteButtonClickEvent: function () {
    var selectEl = document.forms['Book'].selectBook;
    var isbn = selectEl.value;
    if (isbn) {
      Book.destroy( isbn);
      selectEl.remove( selectEl.selectedIndex);
    }
```

```
  }
};
```

Run the App and Get the Code

You can run the minimal app [MinimalApp/index.html] from our server or download the code [MinimalApp.zip] as a ZIP archive file.

Possible Variations and Extensions

Using IndexedDB as an Alternative to LocalStorage

Instead of using the *Local Storage* API, the *IndexedDB* [http://www.html5rocks.com/tutorials/indexeddb/todo/] API could be used for **locally storing** the application data. With *Local Storage* you only have one database (which you may have to share with other apps from the same domain) and there is no support for database tables (we have worked around this limitation in our approach). With *IndexedDB* you can set up a specific database for your app, and you can define database tables, called 'object stores', which may have indexes for accessing records with the help of an indexed attribute instead of the standard identifier attribute. Also, since *IndexedDB* supports larger databases, its access methods are asynchronous and can only be invoked in the context of a database transaction.

Alternatively, for **remotely storing** the application data with the help of a web API one can either use a back-end solution component or a cloud storage service. The remote storage approach allows managing larger databases and supports multi-user apps.

Expressing Date/Time Information with the <time> Element

Assume that our `Book` model class has an additional attribute `publicationDate`, the values of which have to be included in HTML tables and forms. While date/time information items have to be formated as strings in a human-readable form on web pages, preferably in localized form based on the settings of the user's browser, it's not a good idea to store date/time values in this form. Rather we use instances of the pre-defined JavaScript class `Date` for representing and storing date/time values. In this form, the pre-defined functions `toISOString()` and `toLocaleDateString()` can be used for turning `Date` values into ISO standard date/time strings (of the form "2015-01-27") or to localized date/time strings (like "27.1.2015"). Notice that, for simplicity, we have omitted the time part of the date/time strings.

In summary, a date/time value is expressed in three different forms:

1. Internally, for storage and computations, as a `Date` value.
2. Internally, for annotating localized date/time strings, or externally, for displaying a date/time value in a standard form, as an ISO standard date/time string, e.g., with the help of `toISOString()`.
3. Externally, for displaying a date/time value in a localized form, as a localized date/time string, e.g., with the help of `toLocaleDateString()`.

When a date/time value is to be included in a web page, we can use the `<time>` element that allows to display a human-readable representation (typically a localized date/time string) that is annotated with a standard (machine-readable) form of the date/time value.

We illustrate the use of the `<time>` element with the following example of a web page that includes two `<time>` elements: one for displaying a fixed date, and another (initially empty) element for displaying the date of today, which is computed with the help of a JavaScript function. In both cases we use the `datetime` attribute for annotating the displayed human-readable date with the corresponding machine-readable representation.

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
 <meta charset="UTF-8" />
 <title>Using the HTML5 Time Element</title>
 <script src="assignDate.js"></script>
 <script>window.addEventListener("load", assignDate);</script>
</head>
<body>
 <h1>HTML5 Time Element</h1>
 <p>HTML 2.0 was published on <time datetime="1995-11-24">November 24, 1995</ti
 <p>Today is <time id="today" datetime=""></time>.</p>
</body>
</html>
```

This web page loads and executes the following JavaScript function for computing today's date as a `Date` value and assigning its ISO standard representation and its localized representation to the `<time>` element:

```
function assignDate() {
  var dateEl = document.getElementById("today");
  var today = new Date();
  dateEl.textContent = today.toLocaleDateString();
  dateEl.setAttribute("datetime", today.toISOString());
}
```

Points of Attention

The code of this app should be extended by

- adding some **CSS styling** for the user interface pages and
- adding **constraint validation**.

We show how to do this in the follow-up tutorial JavaScript Front-End Web Apps Tutorial Part 2: Adding Constraint Validation [http://web-engineering.info/JsFrontendApp/validation-tutorial.html].

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.

Another issue with the do-it-yourself code of this example app is the **boilerplate code** needed 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 of our tutorial series, 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.

Part II. Integrity Constraints

For catching various cases of flawed data, we need to define suitable integrity constraints that can be used by the application's **data validation** mechanisms. Integrity constraints may take many different forms. The most important type of integrity constraints are **property constraints**, which define conditions on the admissible property values of an object of a certain type.

Table of Contents

Chapter 3. Integrity Constraints and Data Validation

For detecting non-admissible and inconsistent data and for preventing such data to be added to an application's database, we need to define suitable integrity constraints that can be used by the application's **data validation** mechanisms for catching these cases of flawed data. Integrity constraints are logical conditions that must be satisfied by the data in the model objects stored in the application's database. For instance, if an application is managing data about persons including their birth dates and their death dates, if they have already died, then we must make sure that for any person object with a death date, this date is not before that person object's birth date.

Integrity constraints may take many different forms. For instance, **property constraints** define conditions on the admissible property values of an object. They are defined for an object type (or class) such that they apply to all objects of that type. We concentrate on the most important kinds of property constraints:

String Length Constraints	require that the length of a string value for an attribute is less than a certain maximum number, or greater than a minimum number.
Mandatory Value Constraints	require that a property must have a value. For instance, a person must have a name, so the name attribute must not be empty.
Range Constraints	require that an attribute must have a value from the value space of the type that has been defined as its range. For instance, an integer attribute must not have the value "aaa".
Interval Constraints	require that the value of a numeric attribute must be in a specific interval.
Pattern Constraints	require that a string attribute's value must satisfy a certain pattern defined by a regular expression.
Cardinality Constraints	apply to multi-valued properties, only, and require that the cardinality of a multi-valued property's value set is not less than a given minimum cardinality or not greater than a given maximum cardinality.
Uniqueness Constraints	require that a property's value is unique among all instances of the given object type.
Referential Integrity Constraints	require that the values of a reference property refer to an existing object in the range of the reference property.
Frozen Value Constraints	require that the value of a property with this constraint must not be changed after it has been assigned initially.

The visual language of **UML class diagrams** supports defining integrity constraints either with the help of special modeling elements, such as multiplicity expressions, or with the help of *invariants* shown in a special type of rectangle attached to the model element concerned. UML invariants can be expressed in plain English or in the *Object Constraint Language (OCL)*. We use UML class diagrams for making design models with integrity constraints that are independent of a specific programming language or technology platform.

UML class diagrams provide special support for expressing *multiplicity* (or *cardinality*) constraints. This type of constraint allows to specify a *lower multiplicity* (*minimum cardinality*) or an *upper multiplicity* (*maximum cardinality*), or both, for a property or an association end. In UML, this takes the form of a multiplicity expression `l..u` where the lower multiplicity `l` is a non-negative integer and the upper multiplicity `u` is either a positive integer or the special value `*`, standing for unbounded.

For showing property multiplicity constrains in a class diagram, multiplicity expressions are enclosed in brackets and appended to the property name in class rectangles, as shown in the `Person` class rectangle in the class diagram below.

Since *integrity maintenance* is fundamental in database management, the *data definition language* part of the *relational database language SQL* supports the definition of integrity constraints in various forms. On the other hand, however, there is hardly any support for integrity constraints and data validation in common programming languages such as PHP, Java, C# or JavaScript. It is therefore important to take a systematic approach to constraint validation in web application engineering and to choose an application development framework that provides sufficient support for it. Notice that in HTML5, there is some support of data validation for user input in form fields.

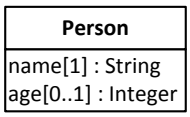
In the following sections we discuss the different types of property constraints listed above in more detail. We also show how to express them in UML class diagrams, in SQL table creation statements and, as an example of how to do it yourself in a programming language, we also show how to express them in JavaScript model class definitions, where we encode constraint validations in class-level ("static") check functions. Any systematic approach also requires to define a set of error (or 'exception') classes, including one for each of the standard property constraints listed above.

String Length Constraints

The length of a string value for a property such as the title of a book may have to be constrained, typically rather by a maximum length, but possibly also by a minimum length. In an SQL table definition, a maximum string length can be specified in parenthesis appended to the SQL datatype `CHAR` or `VARCHAR`, as in `VARCHAR(50)`.

Mandatory Value Constraints

A *mandatory value constraint* requires that a property must have a value. This can be expressed in a UML class diagram with the help of a multiplicity constraint expression where the lower multiplicity is 1. For a single-valued property, this would result in the multiplicity expression `1..1`, or the simplified expression `1`, appended to the property name in brackets. For example, the following class diagram defines a mandatory value constraint for the property `name`:



Whenever a class rectangle does not show a multiplicity expression for a property, the property is mandatory (and single-valued), that is, the multiplicity expression `1` is the default for properties.

In an SQL table creation statement, a mandatory value constraint is expressed in a table column definition by appending the key phrase `NOT NULL` to the column definition as in the following example:

```
CREATE TABLE persons(
  name  VARCHAR(30) NOT NULL,
  age   INTEGER
)
```

According to this table definition, any row of the `persons` table must have a value in the column `name`, but not necessarily in the column `age`.

In JavaScript, we can encode a mandatory value constraint by a class-level check function that tests if the provided argument evaluates to a value, as illustrated in the following example:

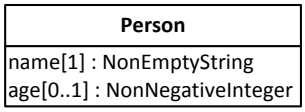
```
Person.checkName = function (n) {
  if (n === undefined) {
    return ...; // error message "A name must be provided!"
```

```
  } else {
    ...
  }
};
```

Range Constraints

A range constraint requires that a property must have a value from the value space of the type that has been defined as its range. This is implicitly expressed by defining a type for a property as its range. For instance, the attribute `age` defined for the object type `Person` in the class diagram above has the range `Integer`, so it must not have a value like "aaa", which does not denote an integer. However, it may have values like -13 or 321, which also do not make sense as the age of a person. In a similar way, since its range is `String`, the attribute `name` may have the value "" (the empty string), which is a valid string that does not make sense as a name.

We can avoid allowing negative integers like -13 as age values, and the empty string as a name, by assigning more specific datatypes as range to these attributes, such as `NonNegativeInteger` to `age`, and `NonEmptyString` to `name`. Notice that such more specific datatypes are neither predefined in SQL nor in common programming languages, so we have to implement them either in the form of user-defined types, as supported in SQL-99 database management systems such as PostgreSQL, or by using suitable additional constraints such as *interval constraints*, which are discussed in the next section. In a UML class diagram, we can simply define `NonNegativeInteger` and `NonEmptyString` as custom datatypes and then use them in the definition of a property, as illustrated in the following diagram:



In JavaScript, we can encode a range constraint by a check function, as illustrated in the following example:

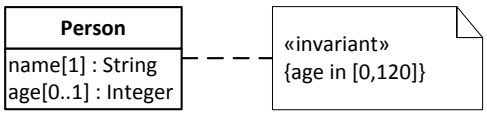
```
Person.checkName = function (n) {
  if (typeof(n) !== "string" || n.trim() === "") {
    return ...; // error message "Name must be a non-empty string!"
  } else {
    ...
  }
};
```

This check function detects and reports a constraint violation if the given value for the `name` property is not of type "string" or is an empty string.

Interval Constraints

An interval constraint requires that an attribute's value must be in a specific interval, which is specified by a minimum value or a maximum value, or both. Such a constraint can be defined for any attribute having an ordered type, but normally we define them only for numeric datatypes or calendar datatypes. For instance, we may want to define an interval constraint requiring that the `age` attribute value must be in the interval [0,120]. In a class diagram, we can define such a constraint as an "invariant" and attach it to the `Person` class rectangle, as shown in Figure 3.1 below.

Figure 3.1. The object type `Person` with an interval constraint



In an SQL table creation statement, an interval constraint is expressed in a table column definition by appending a suitable `CHECK` clause to the column definition as in the following example:

```
CREATE TABLE persons(
  name  VARCHAR(30) NOT NULL,
  age   INTEGER CHECK (age >= 0 AND age <= 120)
)
```

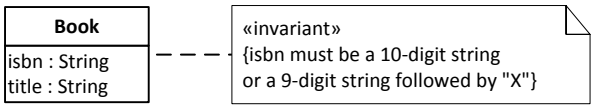
In JavaScript, we can encode an interval constraint in the following way:

```
Person.checkAge = function (a) {
  if (a < 0 || a > 120) {
    return ...; // error message "Age must be between 0 and 120!";
  } else {
    ...
  }
};
```

Pattern Constraints

A pattern constraint requires that a string attribute's value must match a certain pattern, typically defined by a *regular expression*. For instance, for the object type `Book` we define an `isbn` attribute with the datatype `String` as its range and add a pattern constraint requiring that the `isbn` attribute value must be a 10-digit string or a 9-digit string followed by "X" to the `Book` class rectangle shown in Figure 3.2 below.

Figure 3.2. The object type `Book` with a pattern constraint



In an SQL table creation statement, a pattern constraint is expressed in a table column definition by appending a suitable `CHECK` clause to the column definition as in the following example:

```
CREATE TABLE books(
  isbn   VARCHAR(10) NOT NULL CHECK (isbn ~ '^\d{9}(\d|X)$'),
  title  VARCHAR(50) NOT NULL
)
```

The ~ (tilde) symbol denotes the regular expression matching predicate and the regular expression `^\d{9}(\d|X)$` follows the syntax of the POSIX standard (see, e.g. the PostgreSQL documentation [http://www.postgresql.org/docs/9.0/static/functions-matching.html]).

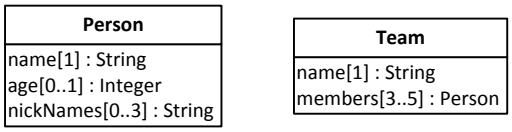
In JavaScript, we can encode a pattern constraint by using the built-in regular expression function `test`, as illustrated in the following example:

```
Person.checkIsbn = function (id) {
  if (!/\b\d{9}(\d|X)\b/.test( id)) {
    return ...; // error message like "The ISBN must be a 10-digit
                // string or a 9-digit string followed by 'X'!"
  } else {
    ...
  }
};
```

Cardinality Constraints

A cardinality constraint requires that the cardinality of a multi-valued property's value set is not less than a given **minimum cardinality** or not greater than a given **maximum cardinality**. In UML, cardinality constraints are called **multiplicity constraints**, and minimum and maximum cardinalities are expressed with the lower bound and the upper bound of the multiplicity expression, as shown in Figure 3.3 below, which contains two examples of properties with cardinality constraints.

Figure 3.3. Two object types with cardinality constraints



The attribute definition `nickNames[0..3]` in the class `Person` specifies a minimum cardinality of 0 and a maximum cardinality of 3, with the meaning that a person may have no nickname or at most 3 nicknames. The reference property definition `members[3..5]` in the class `Team` specifies a minimum cardinality of 3 and a maximum cardinality of 5, with the meaning that a team must have at least 3 and at most 5 members.

It's not obvious how cardinality constraints could be checked in an SQL database, as there is no explicit concept of cardinality constraints in SQL, and the generic form of constraint expressions in SQL, assertions, are not supported by available DBMSs. However, it seems that the best way to implement a minimum (resp. maximum) cardinality constraint is an on-delete (resp. on-insert) trigger that tests the number of rows with the same reference as the deleted (resp. inserted) row.

In JavaScript, we can encode a cardinality constraint validation for a multi-valued property by testing the size of the property's value set, as illustrated in the following example:

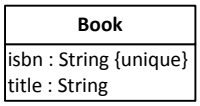
```
Person.checkNickNames = function (nickNames) {
  if (nickNames.length > 3) {
    return ...; // error message like "There must be no more than 3 nicknames!"
  } else {
    ...
  }
};
```

With Java Bean Validation annotations, we can specify `@Size( max=3) List<String> nickNames` or `@Size( min=3, max=5) List<Person> members`.

Uniqueness Constraints

A uniqueness constraint requires that a property's value is unique among all instances of the given object type. For instance, for the object type `Book` we can define the `isbn` attribute to be unique in a UML class diagram by appending the keyword `unique` in curly braces to the attribute's definition in the `Book` class rectangle shown in Figure 3.4 below.

Figure 3.4. The object type `Book` with a uniqueness constraint



In an SQL table creation statement, a uniqueness constraint is expressed by appending the keyword `UNIQUE` to the column definition as in the following example:

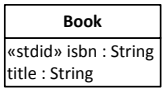
```
CREATE TABLE books(
  isbn   VARCHAR(10) NOT NULL UNIQUE,
  title  VARCHAR(50) NOT NULL
)
```

In JavaScript, we can encode this uniqueness constraint by a check function that tests if there is already a book with the given `isbn` value in the `books` table of the app's database.

Standard Identifiers (Primary Keys)

An attribute (or, more generally, a combination of attributes) can be declared to be the standard identifier for objects of a given type, if it is mandatory and unique. We can indicate this in a UML class diagram with the help of a (user-defined) stereotype «stdid» assigned to the attribute `isbn` as shown in Figure 3.5 below.

Figure 3.5. The object type `Book` with a standard identifier declaration



Notice that a standard identifier declaration implies both a mandatory value and a uniqueness constraint on the attribute concerned.

Standard identifiers are called *primary keys* in relational databases. We can declare an attribute to be the primary key in an SQL table creation statement by appending the phrase `PRIMARY KEY` to the column definition as in the following example:

```
CREATE TABLE books(
  isbn   VARCHAR(10) PRIMARY KEY,
  title  VARCHAR(50) NOT NULL
)
```

In JavaScript, we cannot easily encode a standard identifier declaration, because this would have to be part of the metadata of the class definition, and there is no standard support for such metadata in JavaScript. However, we should at least check if the given argument violates the implied mandatory value or uniqueness constraints by invoking the corresponding check functions discussed above.

Referential Integrity Constraints

A referential integrity constraint requires that the values of a reference property refer to an existing object in the range of the reference property. Since we do not deal with reference properties in this part of the tutorial, we postpone the discussion of referential integrity constraints to the next part of our tutorial.

Frozen Value Constraints

A frozen value constraint defined for a property requires that the value of this property must not be changed after it has been assigned initially. This inlcudes the special case of a **read-only value constraint** applying to mandatory properties that are initialized at object creation time. Typical examples of properties with a frozen value constraint are event properties, since the semantic principle that the past cannot be changed prohibits that the property values of past events can be changed.

In Java, a frozen value constraint can be enforced by declaring the property to be `final`. However, while in Java a `final` property must be mandatory, a frozen value constraint may also apply to an optional property.

In JavaScript, a read-only property can be defined with

```
Object.defineProperty( obj, "teamSize", {value: 5, writable: false, enumerable:
```

by making it unwritable, while an entire object `o` can be frozen by stating `Object.freeze( o)`.

We postpone the further discussion of frozen value constraints to Part 5 of our tutorial.

Constraint Validation in MVC Applications

Unfortunately, many MVC application development frameworks do not provide sufficient support for integrity constraints and data validation.

Integrity constraints should be defined in the model classes of an MVC app since they are part of the business semantics of a model class (representing a business object type). However, a more difficult question is where to perform data validation? In the database? In the model classes? In the controller? Or in the user interface (view)? Or in all of them?

A relational database management system (DBMS) performs data validation whenever there is an attempt to change data in the database, provided that all relevant integrity constraints have been defined in the database. This is essential since we want to avoid, under all circumstances, that invalid data enters the database. However, it requires that we somehow duplicate the code of each integrity constraint, because we want to have it also in the model class to which the constraint belongs.

Also, if the DBMS would be the only application component that validates the data, this would create a latency, and hence usability, problem in distributed applications because the user would not get immediate feedback on invalid input data. This problem is well-known from classical web applications where the front-end component submits the user input data via HTML form submission to a backend component running on a remote web server. Only this backend component validates the data and returns the validation results in the form of a set of error messages to the front end, Only then, typically several seconds later, and in the hard-to-digest form of a bulk message, does the user get the validation feedback. This approach is no longer considered acceptable today. Rather, in a **responsive validation** approach, the user should get immediate validation feedback on each single data input.

So, we need a data validation mechanism in the user interface (UI). Fortunately, the new HTML5 form validation API [http://www.html5rocks.com/en/tutorials/forms/constraintvalidation/] supports constraint validation in the UI. Alternatively, the jQuery Validation Plugin [http://jqueryvalidation.org/] can be used as a (non-HTML5-based) form validation API.

However, it is not sufficient to perform data validation in the user interface. We also need to do it in the model classes, and in the database, for making sure that no flawed data enters the application's persistent data store. This creates the problem of how to maintain the constraint definitions in one place (the model), but use them in two or three places (at least in the model classes and in the UI code, and possibly also in the database).We call this the **multiple validation problem**. This problem can be solved in different ways. For instance:

1. Define the constraints in a declarative language (such as *Java Persistenvy* and *Bean Validation Annotations* or *ASP.NET Data Annotations*) and generate the backend/model and front-end/UI validation code both in a backend application programming language such as Java or C#, and in JavaScript.

2. Keep your validation functions in the (PHP, Java, C# etc.) model classes on the backend, and invoke them from the JavaScript UI code via XHR. This approach can only be used for specific validations, since it implies the penalty of an additional HTTP communication latency for each validation invoked in this way.

3. Use JavaScript as your backend application programming language (such as with NodeJS), then you can encode your validation functions in your JavaScript model classes on the backend and execute them both before committing changes on the backend and on user input and form submission in the UI on the front-end side.

The simplest, and most responsive, solution is the third one, using only JavaScript both in the backend and front-end components.

The support of MVC frameworks for constraint validation can be evaluated according to the following criteria. Does the framework support

1. the declaration of **all important kinds of property constraints** as defined above (String Length Constraints, Mandatory Value Constraints, Range Constraints, etc.) in model classes?

2. validation in the model class **on assign property** and **before save object**?

3. informative **generic validation error messages** referring to the object and property concerned?

4. **app-specific validation error messages** with parameters?

5. **responsive validation** in the user interface **on input** and **on submit** based on the constraints defined in the model classes, preferably using the HTML5 constraint validation API, or at least a general front-end API like the jQuery Validation Plugin?

6. **two-fold validation** with defining constraints in the model and checking them in the model and in the view?

7. **three-fold validation** with defining constraints in the model and checking them in the model, the view and in the database system?

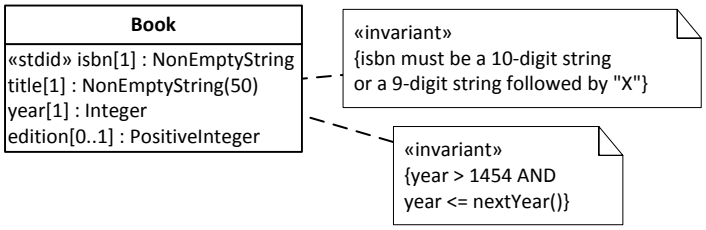
8. **reporting database validation errors** that are passed from the database system to the app?

Chapter 4. Constraint Validation in a JavaScript Front-End Web App

The *minimal* JavaScript front-end web app that we have discussed in Part 1 has been limited to support the minimum functionality of a data management app only. For instance, it did not take care of preventing the user from entering invalid data into the app's database. In this second part of the tutorial we show how to express integrity constraints in a JavaScript *model class*, and how to perform constraint validation both in the *model* part of the app and in the user interface built with HTML5.

We again consider the single-class data management problem that was considered in Part 1 of this tutorial. So, again, the purpose of our app is to manage information about books. But now we also consider the **data integrity rules** (also called 'business rules') that govern the management of book data. These integrity rules, or **constraints**, can be expressed in a UML class diagram as shown in Figure 4.1 below.

Figure 4.1. A platform-independent design model with the object type `Book` and two invariants



In this model, the following constraints have been expressed:

1. Due to the fact that the `isbn` attribute is declared to be the *standard identifier* of `Book`, it is **mandatory** and **unique**.

2. The `isbn` attribute has a **pattern constraint** requiring its values to match the ISBN-10 format that admits only 10-digit strings or 9-digit strings followed by "X".

3. The `title` attribute is **mandatory**, as indicated by its multiplicity expression [1], and has a **string length constraint** requiring its values to have at most 50 characters.

4. The `year` attribute is **mandatory** and has an **interval constraint**, however, of a special form since the maximum is not fixed, but provided by the calendaric function `nextYear()`, which we implement as a utility function.

Notice that the `edition` attribute is not mandatory, but *optional*, as indicated by its multiplicity expression [0..1]. In addition to the constraints described in this list, there are the implicit range constraints defined by assigning the datatype `NonEmptyString` as range to `isbn` and `title`, `Integer` to `year`, and `PositiveInteger` to `edition`. In our plain JavaScript approach, all these property constraints are encoded in the model class within property-specific *check* functions.

Using the HTML5 Form Validation API

We only use two methods of the HTML5 form validation API for validating constraints in the HTML-forms-based user interface of our app. The first of them, `setCustomValidity`, allows to mark a form input field as either valid or invalid by assigning either an empty string or a non-empty message string to it. The second method, `checkValidity`, is invoked on a form and tests, if all input fields have a valid value.

Notice that in our approach there is no need to use the new HTML5 attributes for validation, such as `required`, since we do all validations with the help of `setCustomValidity` and our property check functions, as we explain below.

See this Mozilla tutorial [https://developer.mozilla.org/en-US/docs/Web/Guide/HTML/HTML5/Constraint_validation] or this HTML5Rocks tutorial [http://www.html5rocks.com/en/tutorials/forms/constraintvalidation/] for more about the HTML5 form validation API.

New Issues

Compared to the minimal app [http://web-engineering.info/JsFrontendApp/MinimalApp/index.html] discussed in Part 1 (Minimal App Tutorial [http://web-engineering.info/JsFrontendApp/minimal-tutorial.html]) we have to deal with a number of new issues:

1. In the *model* code we have to take care of

   a. adding for every property of a class a ***check*** function that can be invoked for validating the constraints defined for the property, and a ***setter*** method that invokes the check function and is to be used for setting the value of the property,

   b. performing validation before any data is saved.

2. In the *user interface "(vie"w)* code we have to take care of

   a. styling the user interface with CSS rules,

   b. ***responsive validation*** on user input for providing immediate feedback to the user,

   c. validation on form submission for preventing the submission of flawed data to the model layer.

   For improving the break-down of the view code, we introduce a utility method (in `lib/util.js`) that fills a `select` form control with `option` elements the contents of which is retrieved from a JSON table such as `Book.instances`. This method is used in the `setupUserInterface` method of both the `updateBook` and the `deleteBook` use cases.

Checking the constraints in the user interface on user input is important for providing immediate feedback to the user. But it is not safe enough to perform constraint validation only in the user interface, because this could be circumvented in a distributed web application where the user interface runs in the web browser of a front-end device while the application's data is managed by a backend component on a remote web server. Consequently, we need a **two-fold validation of constraints**, first in the user interface on user input and before form submission, and subsequently in the model code before saving/sending data to the persistent datastore.

Our solution to this **mulitple validation problem** is to keep the constraint validation code in special *check* functions in the model classes and invoke these functions both in the user interface on user input and on form submission, as well as in the *create* and *update* data management methods of the model class via invoking the setters. Notice that certain relationship (such as referential integrity) constraints may also be violated through a *delete* operation, but in our single-class example we don't have to consider this.

Make a JavaScript Data Model

Using the information design model shown in Figure 4.1 above as the starting point, we make a *JavaScript* data model by performing the following steps:

1. Create a ***check*** operation for each non-derived property in order to have a central place for implementing all the constraints that have been defined for a property in the design model. For a standard identifier (or *primary key*) attribute, such as `Book::isbn`, two check operations are needed:

a. A check operation, such as `checkIsbn`, for checking all basic constraints of an identifier attribute, except the *mandatory value* and the *uniqueness* constraints.

b. A check operation, such as `checkIsbnAsId`, for checking in addition to the basic constraints the *mandatory value* and *uniqueness* constraints that are required for an identifier attribute.

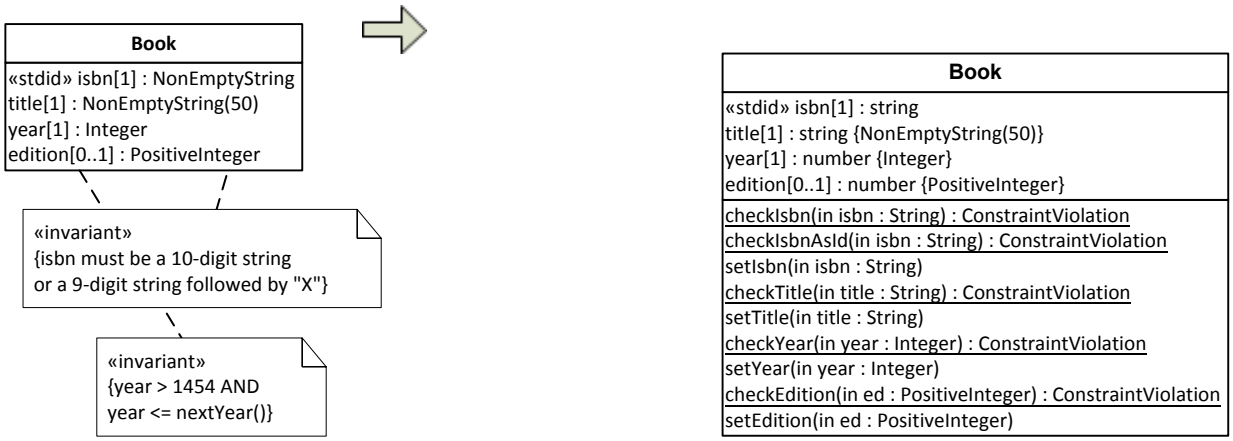
The `checkIsbnAsId` operation is invoked on user input for the `isbn` form field in the *create book* form, and also in the `setIsbn` method, while the `checkIsbn` operation can be used for testing if a value satisfies the syntactic constraints defined for an ISBN.

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.

3. Create **add** and **remove** operations for each non-derived **multi-valued** property (if there are any).

This leads to the *JavaScript data model* shown on the right hand side of the mapping arrow in the following figure.

Figure 4.2. Deriving a JavaScript data model from an information design model



Esssentially, the JavaScript data model extends the design model by adding checks and setters for each property. The attached invariants have been dropped since they are taken care of in the checks. Property ranges have been turned into JavaScript datatypes (with a reminder to their real range in curly braces). Notice that the names of check functions are underlined, since this is the convention in UML for class-level (as opposed to instance-level) operations.

Set up the folder structure and create four initial files

The MVC folder structure of our validation app extends the structure of the minimal app by adding two folders, `css` for adding the CSS file `main.css` and `lib` for adding the generic code libraries `browserShims.js` and `util.js`. Thus, we end up with the following folder structure containing four initial files:

```
publicLibrary
  css
    main.css
  lib
    browserShims.js
    errorTypes.js
```

```
  util.js
src
  ctrl
  model
  view
index.html
```

We discuss the contents of the four initial files in the following sections.

Style the user interface with CSS

We style the UI with the help of the CSS library Pure [http://purecss.io/] provided by *Yahoo*. We only use Pure's basic styles, which include the browser style normalization of normalize.css [http://necolas.github.io/normalize.css/], and its styles for forms. In addition, we define our own style rules for `table` and `menu` elements in `main.css`.

Provide general utility functions and JavaScript fixes in library files

We add two library files to the `lib` folder:

1. `util.js` contains the definitions of a few utility functions such as `isNonEmptyString(x)` for testing if `x` is a non-empty string.

2. `browserShims.js` contains a definition of the string trim function for older browsers that don't support this function (which was only added to JavaScript in ECMAScript Edition 5, defined in 2009). More browser shims for other recently defined functions, such as `querySelector` and `classList`, could also be added to `browserShims.js`.

3. `errorTypes.js` defines general classes for error (or exception) types: NoConstraintViolation, MandatoryValueConstraintViolation, RangeConstraintViolation, IntervalConstraintViolation, PatternConstraintViolation, UniquenessConstraintViolation, OtherConstraintViolation.

Create a start page

The start page of the app first takes care of the page styling by loading the *Pure CSS* base file (from the Yahoo site) and our `main.css` file with the help of the two `link` elements (in lines 6 and 7), then it loads the following JavaScript files (in lines 8-12):

1. `browserShims.js` and `util.js` from the `lib` folder, discussed in Section ,

2. `initialize.js` from the `src/ctrl` folder, defining the app's MVC namespaces, as discussed in Part 1 (the *minimal app tutorial*).

3. `errorTypes.js` from the `lib` folder, defining exception classes.

4. `Book.js` from the `src/model` folder, a model class file that provides data management and other functions discussed in Section .

Figure 4.3. The validation app's start page `index.html`.

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
  <head>
    <meta charset="UTF-8" />
    <title>JS Front-End Validation App Example</title>
    <link rel="stylesheet" type="text/css"
        href="http://yui.yahooapis.com/combo?pure/0.3.0/base-min.css" />
    <link rel="stylesheet" type="text/css" href="css/main.css" />
    <script src="lib/browserShims.js"></script>
    <script src="lib/util.js"></script>
    <script src="lib/errorTypes.js"></script>
    <script src="src/ctrl/initialize.js"></script>
    <script src="src/model/Book.js"></script>
  </head>
  <body>
    <h1>Public Library</h1> <h2>Validation Example App</h2>
    <p>This app supports the following operations:</p>
    <menu>
      <li><a href="listBooks.html"><button type="button">
          List all books</button></a></li>
      <li><a href="createBook.html"><button type="button">
          Add a new book</button></a></li>
      <li><a href="updateBook.html"><button type="button">
          Update a book</button></a></li>
      <li><a href="deleteBook.html"><button type="button">
          Delete a book</button></a></li>
      <li><button type="button" onclick="Book.clearData()">
          Clear database</button></li>
      <li><button type="button" onclick="Book.createTestData()">
          Create test data</button></li>
    </menu>
</body>
</html>
```

Write the Model Code

How to Encode a JavaScript Data Model

The JavaScript data model shown on the right hand side in Figure 4.2 can be encoded step by step for getting the code of the model layer of our JavaScript front-end app. These steps are summarized in the following section.

Summary

1. Encode the model class as a JavaScript constructor function.
2. ***Encode the check functions***, such as `checkIsbn` or `checkTitle`, in the form of class-level ('static') methods. Take care that all constraints, as specified in the JavaScript data model, are properly encoded in the check functions.
3. ***Encode the setter operations***, such as `setIsbn` or `setTitle`, 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.
4. Encode the add and remove operations, if there are any, as instance-level methods.
5. Encode any other operation.

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

Encode the model class as a constructor function

The class `Book` is encoded by means of a corresponding JavaScript *constructor function* with the same name `Book` such that all its (non-derived) properties are supplied with values from corresponding key-value slots of a `slots` parameter.

```
function Book( slots) {
  // assign default values
  this.isbn = "";   // string
  this.title = "";  // string
  this.year = 0;    // number (int)
  if (arguments.length > 0) {
    this.setIsbn( slots.isbn);
    this.setTitle( slots.title);
    this.setYear( slots.year);
    if (slots.edition) {  // optional
      this.setEdition( slots.edition);
    }
  }
};
```

In the constructor body, we first assign default values to the class properties. These values will be used when the constuctor is invoked as a default constructor (without arguments), or when it is invoked with only some arguments. It is helpful to indicate the range of a property in a comment. This requires to map the platform-independent data types of the information design model to the corresponding implicit JavaScript data types according to the following table.

Table 4.1. Datatype mapping

Platform-independent datatype	JavaScript datatype
String	string
Integer	number (int)
Decimal	number (float)
Boolean	boolean
Date	Date

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

As in the minimal app, we add a class-level property `Book.instances` representing the collection of all Book instances managed by the application in the form of a JSON table:

```
Book.instances = {};
```

Encode the property checks

Encode the property check functions in the form of class-level ('static') methods. In JavaScript, this means to define them as function slots of the constructor, as in `Book.checkIsbn`. Take care that all constraints of a property as specified in the data model are properly encoded in its check function. This concerns, in particular, the *mandatory value* and *uniqueness* constraints implied by the *standard identifier* declaration (with «`stdid`»), and the *mandatory value* constraints for all properties with multiplicity 1, which is the default when no multiplicity is shown. If any constraint is violated, an error object instantiating one of the error classes listed above in Section and defined in the file `model/errorTypes.js` is returned.

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

```
Book.checkIsbn = function (id) {
  if (!id) {
    return new NoConstraintViolation();
  } else if (typeof(id) !== "string" || id.trim() === "") {
    return new RangeConstraintViolation(
        "The ISBN must be a non-empty string!");
  } else if (!/\b\d{9}(\d|X)\b/.test( id)) {
    return new PatternConstraintViolation(
        "The ISBN must be a 10-digit string or "+
        " a 9-digit string followed by 'X'!");
  } else {
    return new NoConstraintViolation();
  }
};
```

Notice that, since `isbn` is the standard identifier attribute of `Book`, we only check the syntactic constraints in `checkIsbn`, but we check the *mandatory value* and *uniqueness* constraints in `checkIsbnAsId`, which itself first invokes `checkIsbn`:

```
Book.checkIsbnAsId = function (id) {
  var constraintViolation = Book.checkIsbn( id);
  if ((constraintViolation instanceof NoConstraintViolation)) {
    if (!id) {
      constraintViolation = new MandatoryValueConstraintViolation(
          "A value for the ISBN must be provided!");
    } else if (Book.instances[id]) {
      constraintViolation = new UniquenessConstraintViolation(
          "There is already a book record with this ISBN!");
    } else {
      constraintViolation = new NoConstraintViolation();
    }
  }
  return constraintViolation;
};
```

Encode the property setters

Encode the setter operations as (instance-level) methods. In the setter, the corresponding check function is invoked and the property is only set, if the check does not detect any constraint violation. Otherwise, the *constraint violation* error object returned by the check function is thrown. For instance, the `setIsbn` operation is encoded in the following way:

```
Book.prototype.setIsbn = function (id) {
  var validationResult = Book.checkIsbnAsId( id);
  if (validationResult instanceof NoConstraintViolation) {
    this.isbn = id;
  } else {
    throw validationResult;
  }
};
```

There are similar setters for the other properties (`title`, `year` and `edition`).

Add a serialization function

It is helpful to have a serialization function tailored to the structure of a class such that the result of serializing an object is a human-readable string representation of the object showing all relevant

information items of it. By convention, these functions are called `toString()`. In the case of the `Book` class, we use the following code:

```
Book.prototype.toString = function () {
  return "Book{ ISBN:" + this.isbn + ", title:" +
      this.title + ", year:" + this.year +"}";
};
```

Data management operations

In addition to defining the model class in the form of a constructor function with property definitions, checks and setters, as well as a `toString()` function, we also need to define the following data management operations as class-level methods of the model class:

1. `Book.convertRow2Obj` and `Book.loadAll` for loading all managed Book instances from the persistent data store.

2. `Book.saveAll` for saving all managed Book instances to the persistent data store.

3. `Book.add` for creating a new Book instance.

4. `Book.update` for updating an existing Book instance.

5. `Book.destroy` for deleting a Book instance.

6. `Book.createTestData` for creating a few example book records to be used as test data.

7. `Book.clearData` for clearing the book datastore.

All of these methods essentially have the same code as in our *minimal app* discussed in Part 1, except that now

1. we may have to catch constraint violations in suitable try-catch blocks in the procedures `Book.convertRow2Obj`, `Book.add`, `Book.update` and `Book.createTestData`; and

2. we can use the `toString()` function for serializing an object in status and error messages.

Notice that for the change operations `create` and `update`, we need to implement an all-or-nothing policy: as soon as there is a constraint violation for a property, no new object must be created and no (partial) update of the affected object must be performed.

When a constraint violation is detected in one of the setters called when `new Book(...)` is invoked in `Book.add`, the object creation attempt fails, and instead a constraint violation error message is created in line 6. Otherwise, the new book object is added to `Book.instances` and a status message is creatred in lines 10 and 11, as shown in the following program listing:

```
Book.create = function (slots) {
  var book = null;
  try {
    book = new Book( slots);
  } catch (e) {
    console.log( e.name +": "+ e.message);
    book = null;
  }
  if (book) {
    Book.instances[book.isbn] = book;
    console.log( book.toString() + " created!");
  }
```

```
};
```

Likewise, when a constraint violation is detected in one of the setters invoked in `Book.update`, a constraint violation error message is created (in line 16) and the previous state of the object is restored (in line 19). Otherwise, a status message is created (in lines 23 or 25), as shown in the following program listing:

```
Book.update = function (slots) {
  var book = Book.instances[slots.isbn],
      noConstraintViolated = true,
      updatedProperties = [],
      objectBeforeUpdate = util.cloneObject( book);
  try {
    if (book.title !== slots.title) {
      book.setTitle( slots.title);
      updatedProperties.push("title");
    }
    if (book.year !== parseInt( slots.year)) {
      book.setYear( slots.year);
      updatedProperties.push("year");
    }
    if (slots.edition && book.edition !== parseInt(slots.edition)) {
        book.setEdition( slots.edition);
        updatedProperties.push("edition");
    }
  } catch (e) {
    console.log( e.name +": "+ e.message);
    noConstraintViolated = false;
    // restore object to its state before updating
    Book.instances[slots.isbn] = objectBeforeUpdate;
  }
  if (noConstraintViolated) {
    if (updatedProperties.length > 0) {
      console.log("Properties " + updatedProperties.toString() +
          " modified for book " + slots.isbn);
    } else {
      console.log("No property value changed for book " +
          slots.isbn + " !");
    }
  }
};
```

The View and Controller Layers

The user interface (UI) consists of a start page `index.html` that allows the user choosing one of the data management operations by navigating to the corresponding UI page such as `listBooks.html` or `createBook.html` in the app folder. The start page `index.html` has been discussed in Section .

After loading the Pure [http://purecss.io/] base stylesheet and our own CSS settings in `main.css`, we first load some browser shims and utility functions. Then we initialize the app in `src/ctrl/initialize.js` and continue loading the error classes defined in `lib/errorTypes.js` and the model class `Book`.

We render the data management menu items in the form of buttons. For simplicity, we invoke the `Book.clearData()` and `Book.createTestData()` methods directly from the buttons' `onclick` event handler attribute. Notice, however, that it is generally preferable to register such event handling functions with `addEventListener(...)`, as we do in all other cases.

The data management UI pages

Each data management UI page loads the same basic CSS and JavaScript files like the start page `index.html` discussed above. In addition, it loads two use-case-specific view and controller files `src/view/`*useCase*`.js` and `src/ctrl/`*useCase*`.js` and then adds a use case initialize function (such as `pl.ctrl.listBooks.initialize`) as an event listener for the page load event, which takes care of initializing the use case when the UI page has been loaded (see Section).

For the "list books" use case, we get the following code in `listBooks.html`:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
  <meta charset="UTF-8" />
  <title>JS Front-End Validation App Example</title>
  <link rel="stylesheet"
        href="http://yui.yahooapis.com/pure/0.3.0/pure-min.css" />
  <link rel="stylesheet" href="css/main.css" />
  <script src="lib/browserShims.js"></script>
  <script src="lib/util.js"></script>
  <script src="lib/errorTypes.js"></script>
  <script src="src/ctrl/initialize.js"></script>
  <script src="src/model/Book.js"></script>
  <script src="src/view/listBooks.js"></script>
  <script src="src/ctrl/listBooks.js"></script>
  <script>
    window.addEventListener("load", pl.ctrl.listBooks.initialize);
  </script>
</head>
<body>
  <h1>Public Library: List all books</h1>
  <table id="books">
    <thead>
      <tr><th>ISBN</th><th>Title</th><th>Year</th><th>Edition</th></tr>
    </thead>
    <tbody></tbody>
  </table>
  <nav><a href="index.html">Back to main menu</a></nav>
</body>
</html>
```

For the "create book" use case, we get the following code in `createBook.html`:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
  <meta charset="UTF-8" />
  <title>JS Front-End Validation App Example</title>
  <link rel="stylesheet"
        href="http://yui.yahooapis.com/combo?pure/0.3.0/base-
              min.css&pure/0.3.0/forms-min.css" />
  <link rel="stylesheet" href="css/main.css" />
  <script src="lib/browserShims.js"></script>
  <script src="lib/util.js"></script>
  <script src="lib/errorTypes.js"></script>
  <script src="src/ctrl/initialize.js"></script>
  <script src="src/model/Book.js"></script>
  <script src="src/view/createBook.js"></script>
```

```
  <script src="src/ctrl/createBook.js"></script>
  <script>
    window.addEventListener("load", pl.ctrl.createBook.initialize);
  </script>
  </head>
  <body>
  <h1>Public Library: Create a new book record</h1>
  <form id="Book" class="pure-form pure-form-aligned">
    <div class="pure-control-group">
      <label for="isbn">ISBN</label>
      <input id="isbn" name="isbn" />
    </div>
    <div class="pure-control-group">
      <label for="title">Title</label>
      <input id="title" name="title" />
    </div>
    <div class="pure-control-group">
      <label for="year">Year</label>
      <input id="year" name="year" />
    </div>
    <div class="pure-control-group">
      <label for="edition">Edition</label>
      <input id="edition" name="edition" />
    </div>
    <div class="pure-controls">
      <p><button type="submit" name="commit">Save</button></p>
      <nav><a href="index.html">Back to main menu</a></nav>
    </div>
  </form>
</body>
</html>
```

Notice that for styling the form elements in `createBook.html`, and also for `updateBook.html` and `deleteBook.html`, we use the Pure [http://purecss.io/] CSS form styles. This requires to assign specific values, such as "pure-control-group", to the `class` attributes of the form's `div` elements containing the form controls. We have to use explicit labeling (with the `label` element's `for` attribute referencing the `input` element's `id`), since *Pure* does not support implicit labels where the `label` element contains the `input` element.

Initialize the app

For initializing the app, its namespace and MVC subnamespaces have to be defined. For our example app, the main namespace is defined to be `pl`, standing for "Public Library", with the three subnamespaces `model`, `view` and `ctrl` being initially empty objects:

```
var pl = { model:{}, view:{}, ctrl:{} };
```

We put this code in the file `initialize.js` in the `ctrl` folder.

Initialize the data management use cases

For initializing a data management use case, the required data has to be loaded from persistent storage and the UI has to be set up. This is performed with the help of the controller procedures `pl.ctrl.createBook.initialize` and `pl.ctrl.createBook.loadData` defined in the controller file `ctrl/createBook.js` with the following code:

```
pl.ctrl.createBook = {
  initialize: function () {
    pl.ctrl.createBook.loadData();
```

```
    pl.view.createBook.setupUserInterface();
  },
  loadData: function () {
    Book.loadAll();
  }
};
```

All other data management use cases (read/list, update, delete) are handled in the same way.

Set up the user interface

For setting up the user interfaces of the data management use cases, we have to distinguish the case of "list books" from the other ones (create, update, delete). While the latter ones require using an HTML form and attaching event handlers to form controls, in the case of "list books" we only have to render a table displaying all the books, as shown in the following program listing of `view/listBooks.js`:

```
pl.view.listBooks = {
  setupUserInterface: function () {
    var tableBodyEl = document.querySelector("table#books>tbody");
    var i=0, book=null, row={}, key="",
        keys = Object.keys( Book.instances);
    for (i=0; i < keys.length; i++) {
      key = keys[i];
      book = Book.instances[key];
      row = tableBodyEl.insertRow(-1);
      row.insertCell(-1).textContent = book.isbn;
      row.insertCell(-1).textContent = book.title;
      row.insertCell(-1).textContent = book.year;
      if (book.edition) {
        row.insertCell(-1).textContent = book.edition;
      }
    }
  }
};
```

For the *create*, *update* and *delete* use cases, we need to attach the following event handlers to form controls:

1. a function, such as `handleSubmitButtonClickEvent`, for handling the event when the user clicks the save/submit button,

2. functions for validating the data entered by the user in form fields (if there are any).

In addition, in line 28 of the following `view/createBook.js` code, we add an event handler for saving the application data in the case of a `beforeunload` event, which occurs, for instance, when the browser (or browser tab) is closed:

```
pl.view.createBook = {
  setupUserInterface: function () {
    var formEl = document.forms['Book'],
        submitButton = formEl.commit;
    submitButton.addEventListener("click",
        this.handleSubmitButtonClickEvent);
    formEl.isbn.addEventListener("input", function () {
        formEl.isbn.setCustomValidity(
            Book.checkIsbnAsId( formEl.isbn.value).message);
    });
    formEl.title.addEventListener("input", function () {
        formEl.title.setCustomValidity(
            Book.checkTitle( formEl.title.value).message);
```

```
    });
    formEl.year.addEventListener("input", function () {
        formEl.year.setCustomValidity(
            Book.checkYear( formEl.year.value).message);
    });
    formEl.edition.addEventListener("input", function () {
        formEl.edition.setCustomValidity(
            Book.checkEdition( formEl.edition.value).message);
    });
    // neutralize the submit event
    formEl.addEventListener( 'submit', function (e) {
        e.preventDefault();;
        formEl.reset();
    });
    window.addEventListener("beforeunload", function () {
        Book.saveAll();
    });
  },
  handleSubmitButtonClickEvent: function () {
    ...
  }
};
```

Notice that for each form input field we add a listener for `input` events, such that on any user input a validation check is performed because `input` events are created by user input actions such as typing. We use the predefined function `setCustomValidity` from the HTML5 form validation API for having our property check functions invoked on the current value of the form field and returning an error message in the case of a constraint violation. So, whenever the string represented by the expression `Book.checkIsbn( formEl.isbn.value).message` is empty, everything is fine. Otherwise, if it represents an error message, the browser indicates the constraint violation to the user by rendering a red outline for the form field concerned (due to our CSS rule for the `:invalid` pseudo class).

While the validation on user input enhances the usability of the UI by providing immediate feedback to the user, validation on form data submission is even more important for catching invalid data. Therefore, the event handler `handleSubmitButtonClickEvent()` performs the property checks again with the help of `setCustomValidity`, as shown in the following program listing:

```
handleSubmitButtonClickEvent: function () {
  var formEl = document.forms['Book'];
  var slots = { isbn: formEl.isbn.value,
          title: formEl.title.value,
          year: formEl.year.value,
          edition: formEl.edition.value
  };
  // set error messages in case of constraint violations
  formEl.isbn.setCustomValidity( Book.checkIsbnAsId( slots.isbn).message);
  formEl.title.setCustomValidity( Book.checkTitle( slots.title).message);
  formEl.year.setCustomValidity( Book.checkYear( slots.year).message);
  formEl.edition.setCustomValidity(
      Book.checkEdition( formEl.edition.value).message);
  // save the input data only if all of the form fields are valid
  if (formEl.checkValidity()) {
    Book.add( slots);
  }
}
```

By invoking `checkValidity()` on the form element, we make sure that the form data is only saved (by `Book.add`), if there is no constraint violation. After this

`handleSubmitButtonClickEvent` handler has been executed on an invalid form, the browser takes control and tests if the pre-defined property `validity` has an error flag for any form field. In our approach, since we use `setCustomValidity`, the `validity.customError` would be true. If this is the case, the custom constraint violation message will be displayed (in a bubble) and the `submit` event will be suppressed.

For the use case *update book*, which is handled in `view/updateBook.js`, we provide a book selection list, so the user need not enter an identifier for books (an ISBN), but has to select the book to be updated. This implies that there is no need to validate the ISBN form field, but only the title and year fields. We get the following code:

```
pl.view.updateBook = {
  setupUserInterface: function () {
    var formEl = document.forms['Book'],
        submitButton = formEl.commit,
        selectBookEl = formEl.selectBook;
    // set up the book selection list
    util.fillSelectWithOptions( Book.instances,
     selectBookEl, "isbn", "title");
    // when a book is selected, populate the form with its data
    selectBookEl.addEventListener("change", function () {
      var bookKey = selectBookEl.value;
      if (bookKey) {
        book = Book.instances[bookKey];
        formEl.isbn.value = book.isbn;
        formEl.title.value = book.title;
        formEl.year.value = book.year;
        if (book.edition) formEl.edition.value = book.edition;
      } else {
        formEl.reset();
      }
    });
    formEl.title.addEventListener("input", function () {
        formEl.title.setCustomValidity(
            Book.checkTitle( formEl.title.value).message);
    });
    formEl.year.addEventListener("input", function () {
        formEl.year.setCustomValidity(
            Book.checkYear( formEl.year.value).message);
    });
    formEl.edition.addEventListener("input", function () {
        formEl.edition.setCustomValidity(
            Book.checkEdition( formEl.edition.value).message);
    });
    submitButton.addEventListener("click",
        this.handleSubmitButtonClickEvent);
    // neutralize the submit event
    formEl.addEventListener( 'submit', function (e) {
        e.preventDefault();;
        formEl.reset();
    });
    window.addEventListener("beforeunload", function () {
        Book.saveAll();
    });
  },
```

When the save button on the *update book* form is clicked, the `title` and `year` form field values are validated by invoking `setCustomValidity`, and then the book record is updated if the form data validity can be established with `checkValidity()`:

```
  handleSubmitButtonClickEvent: function () {
    var formEl = document.forms['Book'];
    var slots = { isbn: formEl.isbn.value,
        title: formEl.title.value,
        year: formEl.year.value,
        edition: formEl.edition.value
    };
    // set error messages in case of constraint violations
    formEl.title.setCustomValidity(
        Book.checkTitle( slots.title).message);
    formEl.year.setCustomValidity(
        Book.checkYear( slots.year).message);
    formEl.edition.setCustomValidity(
            Book.checkEdition( formEl.edition.value).message);
    if (formEl.checkValidity()) {
      Book.update( slots);
    }
  }
```

The logic of the `setupUserInterface` method for the *delete* use case is similar.

Run the App and Get the Code

You can run the validation app [ValidationApp/index.html] from our server or download the code [ValidationApp.zip] as a ZIP archive file.

Evaluation

We evaluate the approach presented in this chapter of the tutorial according to the criteria defined in the previous chapter.

Table 4.2. Evaluation

Evaluation criteria	**½ ¼ ¾**
all important kinds of property constraints	1
model validation on assign and before save	1
generic validation error messages	0
responsive validation on input and on submit	1
two-fold validation	1
three-fold validation	n.a.
reporting database validation errors	n.a.
in total	80 %

Possible Variations and Extensions

Simplifying forms with implicit labels

The explicit labeling of form fields used in this tutorial requires to add an `id` value to the `input` element and a `for`-reference to its `label` element as in the following example:

```
<div class="pure-control-group">
  <label for="isbn">ISBN:</label>
  <input id="isbn" name="isbn" />
```

```
</div>
```

This technique for associating a label with a form field is getting quite inconvenient when we have many form fields on a page bceuase we have to make up a great many of unique `id` values and have to make sure that they don't conflict with any of the `id` values of other elements on the same page. It's therefore preferable to use an approach, called *implicit labeling*, that does not need all these `id` references. In this approach we make the `input` element a child element of its `label` element, as in

```
<div>
  <label>ISBN: <input name="isbn" /></label>
</div>
```

Having `input` as a child of its `label` doesn't seem very logical (rather, one would expect the `label` to be a child of an `input` element). But that's the way, it is defined in HTML5.

A small disadvantage of using implicit labels is the lack of support by popular CSS libraries, such as Pure CSS. In the following parts 3-5 of this tutorial, we will use our own CSS styling for implicitly labeled form fields.

Enumerations and enumeration attributes

In all application domains, there are enumeration datatypes that define the possible values of **enumeration attributes**. For instance, when we have to manage data about persons, we often need to include information about the gender of a person. The possible values of a `gender` attribute are restricted to one of the following: "male","female", or "undetermined". Instead of using these strings as the internal values of the enumeration attribute `gender`, it is preferable to use the positive integers 1, 2 and 3, which enumerate the possible values. However, since these integers do not reveal their meaning (as indicated by the enumeration label they stand for) in program code, for readability we rather use special constants, called *enumeration literals*, such as `GenderEL.MALE` and `GenderEL.FEMALE`, in program statements like `this.gender = GenderEL.FEMALE`. Notice that, by convention, enumeration literals are all upper case.

We can implement an enumeration in the form of a special JavaScript object definition using the `Object.defineProperties` method:

```
var BookCategoryEL = null;
Object.defineProperties( BookCategoryEL, {
  NOVEL: {value: 1, writable: false},
  BIOGRAPHY: {value: 2, writable: false},
  TEXTBOOK: {value: 3, writable: false},
  OTHER: {value: 4, writable: false},
  MAX: {value: 4, writable: false},
  labels: {value:["novel","biography","textbook","other"],
           writable: false}
});
```

Notice how this definition of an enumeration of book categories takes care of the requirement that enumeration literals like `BookCategoryEL.NOVEL` are constants, the value of which cannot be changed during program execution. This is achieved with the help of the property descriptor `writable: false` in the `Object.defineProperties` statement.

This definition allows using the enumeration literals `BookCategoryEL.NOVEL`, `BookCategoryEL.BIOGRAPHY` etc., standing for the enumeration integers 1, 2 , 3 and 4, in program statements. Notice that we use the convention to suffix the name of an enumeration with "EL" standing for "enumeration literal".

We can also use a generic approach and define an `Enumeration` class for creating enumerations:

```
function Enumeration( enumLabels) {
```

```
  var i=0, LBL="";
  Object.defineProperties( this, {
    MAX: {value: enumLabels.length, writable: false},
    labels: {value: enumLabels, writable: false}
  });
  // generate the enum literals as capitalized keys/properties
  for (i=1; i <= enumLabels.length; i++) {
    LBL = enumLabels[i-1].toUpperCase();
    this[LBL] = i;
  }
};
```

Using the `Enumeration` class allows to define a new enumeration in the following simplified way:

```
var GenderEL = new Enumeration(["novel","biography","textbook","other"])
```

Having an enumeration like `BookCategoryEL`, we can then check if an enumeration attribute like `category` has an admissible value by testing if its value is not smaller than 1 and not greater than `BookCategoryEL.MAX`.

We consider the following model class `Book` with the enumeration attribute `category`:

```
function Book( slots) {
  this.isbn = "";     // string
  this.title = "";    // string
  this.year = 0;      // number (int)
  this.category = 0;  // number (enum)
  if (arguments.length > 0) {
    this.setIsbn( slots.isbn);
    this.setTitle( slots.title);
    this.setYear( slots.year);
    this.setCategory( slots.category);
  }
};
```

For validating input values for the enumeration attribute `category`, we can use the following check function:

```
Book.checkCategory = function (c) {
  if (!c) {
    return new MandatoryValueConstraintViolation(
        "A category must be provided!");
  } else if (!util.isPositiveInteger(c) ||
             c > BookCategoryEL.MAX) {
    return new RangeConstraintViolation(
        "The category must be a positive integer " +
        "not greater than "+ BookCategoryEL.MAX +" !");
  } else {
    return new NoConstraintViolation();
  }
};
```

Notice how the range constraint defined by the enumeration `BookCategoryEL` is checked: it is tested if the input value `c` is a positive integer and if it is not greater than `BookCategoryEL.MAX`.

In the user interface, an output field for an enumeration attribute would display the enumeration label, rather than the enumeration integer. The label can be retrieved in the following way:

```
formEl.category.value = BookCategoryEL.labels[this.category];
```

For user input to a **single-valued** enumeration attribute like `Book::category`, a **radio button group** could be used if the number of enumeration literals is sufficiently small, otherwise a **single selection list** would be used. If the selection list is implemented with an HTML `select` element, the enumeration labels would be used as the text content of the option elements, while the enumeration integers would be used as their values.

For user input to a **multi-valued** enumeration attribute, a **checkbox group** could be used if the number of enumeration literals is sufficiently small, otherwise a **multiple selection list** would be used. For usability, the multiple selection list can only be implemented with an HTML `select` element, if the number of enumeration literals does not exceed a certain threshold, which depends on the number of options the user can see on the screen without scrolling.

Enumerations may have further features. For instance, we may want to be able to define a new enumeration by extending an existing enumeration. In programming languages and in other computational languages, enumerations are implemented with different features in different ways. See also the Wikipedia article on enumerations [http://en.wikipedia.org/wiki/Enumerated_type].

Points of Attention

You may have noticed the repetitive code structures (called ***boilerplate code***) needed 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 of our tutorial series, 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.

Part III. Associations

Associations are important elements of information models. 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, application development frameworks do often not provide much support for dealing with associations.

Table of Contents

Chapter 5. 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 from some class. For instance, the class `Committee` shown in Figure 5.1 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 *Club*-**has**-*ClubMember*-**as-chair**, which is visualized as a connection line in the class diagram shown in Figure 5.2 below, classifies the relationships SantaFeSoccerClub-**has**-PeterMiller-**as-chair**, CaliforniaHikerClub-**has**-SusanSmith-**as-chair** and BerlinRotaryClub-**has**-SarahAnderson-**as-chair**, where the objects PeterMiller, SusanSmith and SarahAnderson are of type `ClubMember`, and the objects SantaFeSoccerClub, CaliforniaHikerClub and BerlinRotaryClub are of type `Club`.

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 5.1. An example of an association table

***Club*-has-*ClubMember*-as-chair**	
Santa Fe Soccer Club	Peter Miller
California Hiker Club	Susan Smith
Berlin Rotary Club	Sarah Anderson

While individual relationships (such as SantaFeSoccerClub-**has**-PeterMiller-**as-chair**) are important information items in business communication and in information systems, associations (such as *Club*-**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 5.2. Different terminologies

Our preferred term(s)	**UML**	**ER Diagrams**	**Mathematics**
object	object	entity	individual
object type (class)	class	entity type	unary relation
relationship	link	relationship	tuple

Our preferred term(s)	UML	ER Diagrams	Mathematics
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.

References and Reference Properties

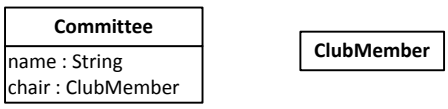
A reference can be either *human-readable* or an *internal object reference*. Human-readable references refer to identifiers that are used in human communication, such as the unique names of astronomical bodies, the ISBN of books and the employee numbers of the employees of a company. Internal object 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 also call human-readable references *ID references* and use the suffix `IdRef` for the names of human-readable reference properties. 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 `publisherIdRef`. 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 5.1 below has an attribute `name` with range `string`, and a reference property `chair` with range `ClubMember`.

Figure 5.1. 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 5.1. 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 5.1 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 5.10, “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.

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 deleted or replaced by a another reference.
   For every reference property in our app's model 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.

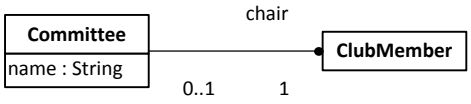
Typically, object creation dependencies are managed in the user interface by not allowing the user to enter a value of an ID reference property, but only to select one from a list of all existing target objects.

Modeling Reference Properties as Unidirectional Associations

A reference property (such as `chair` in the example shown in Figure 5.1 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 5.2 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 5.2 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 5.2. A committee has a club member as chair expressed by an association end with a "dot"



Thus, the two diagrams shown in Figure 5.1 and Figure 5.2 express essentially equivalent models. When a reference property is modeled by an association end with a "dot", like `chair` in Figure 5.1, 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 5.1, 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 5.2 assigns at most one object of type `ClubMember` as chair to an object of type `Club`. 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 5.3. Functionality types

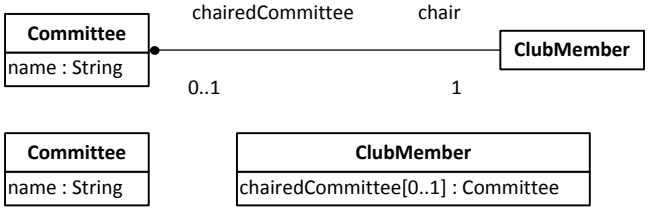
Functionality type	Meaning
one-to-one	both functional and inverse functional
many-to-one	functional
one-to-many	inverse functional
many-to-many	neither functional nor inverse functional

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).

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 5.3 below.

Figure 5.3. Representing the unidirectional association ClubMember has Committee as chairedCommittee as a reference property



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 5.4 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 5.4. 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 5.4 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.

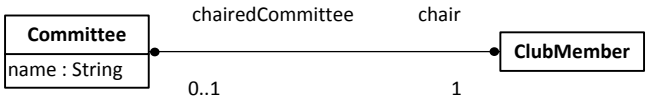
There are three options how to turn a model without association end ownership dots into a complete OO design model where all associations are either uni-directional or bi-directional: we can place an ownership dot at either end or at both ends of the association. Each of these three options defines a different way how to represent, or implement, the association with the help of reference properties. So, for the association shown in Figure 5.4 above, we have the following options:

1. Place an ownership dot at the `chair` association end, leading to the model shown in Figure 5.2 above, which can be turned into the association-free model shown in Figure 5.1 above.

2. Place an ownership dot at the `chairedCommittee` association end, leadig to the completed models shown in Figure 5.3 above.

3. Make the association bi-directional 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 5.5. Modeling a bidirectional association between Committee and ClubMember



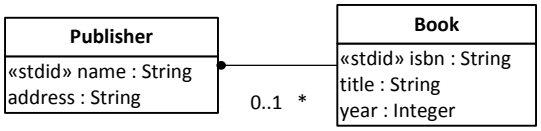
So, whenever we have modeled an association, we have to make a choice, which of its ends represents a reference property and will therefore be marked with an ownership dot. It can be either one, or both. This decision also implies a decision about the *navigability* of the association. When an association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

Our Running Example

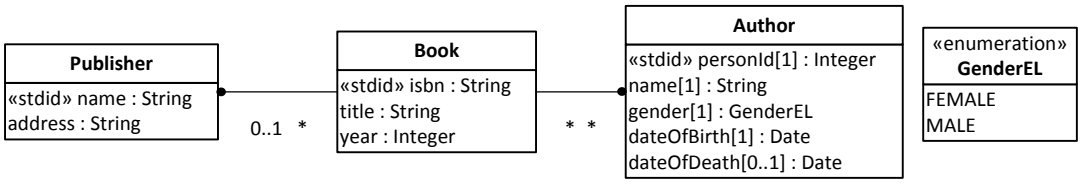
The model shown in Figure 5.6 below (about publishers and books) serves as our running example for a unidirectional functional association in this tutorial. Notice that it contains the unidirectional many-to-one association *Book*-**has**-*Publisher*.

Figure 5.6. 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 5.7 below, models a multi-valued (non-functional) reference property `authors`.

Figure 5.7. The Publisher-Book-Author information design model with two unidirectional associations



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.

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 uni-directional associations, such as the model shown in Figure 5.6 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 .

A uni-directional 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 uni-directional association is inverse functional.

This replacement procedure is illustrated for the case of a uni-directional one-to-one association in Figure 5.3 above.

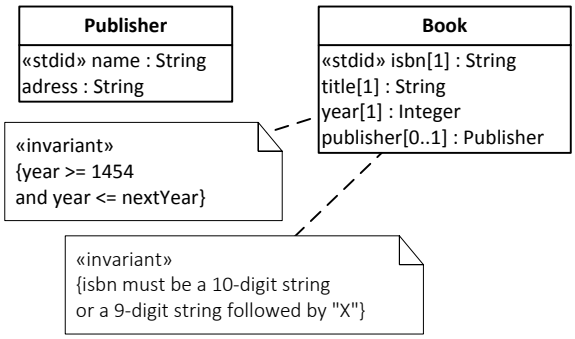
For the case of a uni-directional one-to-many association, Figure 5.8 below provides an illustration of the association elimination procedure. Here, the non-functional association end at the target class `Point` is turned into a corresponding reference property with name `points` obtained as the pluralized form of the target class name

Figure 5.8. Turn a non-functional target association end into a corresponding reference property

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 uni-directional associations with suitable reference properties. In the first step, we replace the many-to-one association *Book*-**has**-*Publisher* in the model of Figure 5.6 with a functional reference property `publisher` in the class `Book`, resulting in the following association-free model:

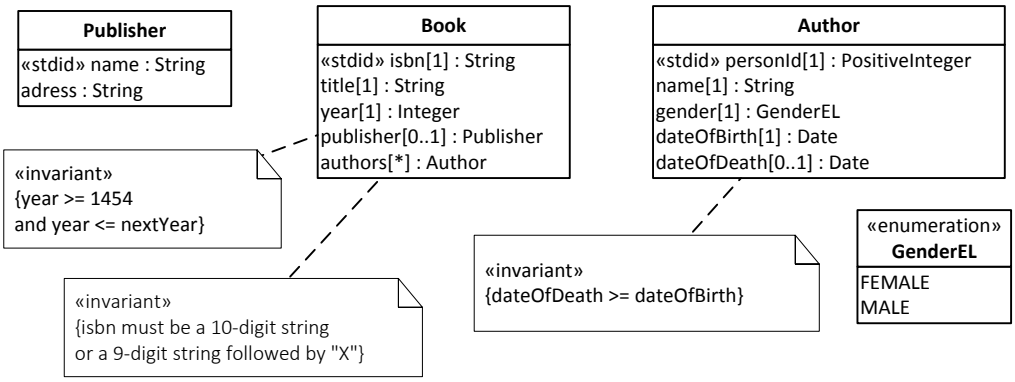
Figure 5.9. The association-free Publisher-Book design model



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 5.7 with a multi-valued reference property `authors` in the class `Book`, resulting in the following association-free model:

Figure 5.10. 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.

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 ***selection list 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.

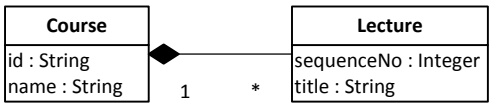
Chapter 6. Part-Whole Associations

A part-whole association is an association that represents a relationship between a part type and a whole type. Its instances are part-whole relationships between two objects where one of them is a part of the other.

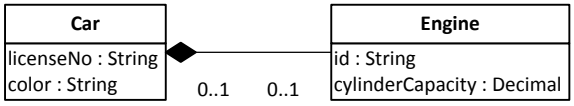
Composition

As Martin Fowler has explained [http://martinfowler.com/bliki/AggregationAndComposition.html], the main issue for characterizing composition is that "an object can only be the part of one composition relationship". This is also explained in the excellent blog post UML Composition vs Aggregation vs Association [http://bellekens.com/2010/12/20/uml-composition-vs-aggregation-vs-association/] by Geert Bellekens. In addition to this defining characteristic of a composition (to have **exclusive**, or **non-shareable**, parts), a composition may also come with a life-cycle dependency between the whole and its parts implying that when a whole is destroyed, all of its parts are destroyed with it. However, this only applies to some cases of composition, and not to others, and it is therefore not a defining characteristic.

For instance, in the `Course-Lecture` composition shown in the following diagram, we have a lifecycle dependency between courses and lectures such that when a course is dropped (from a curriculum), all its lectures are dropped/deleted as well. This is implied by the **exactly one** multiplicity at the composite side of the composition line.



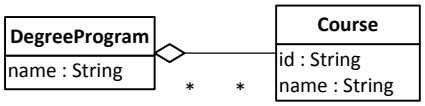
:The UML spec states: "A part may be removed from a composite instance before the composite instance is deleted, and thus not be deleted as part of the composite instance." In the example of a `Car-Engine` composition, as shown in the following diagram, it's clearly the case that the engine can be detached from the car before the car is destroyed, in which case the engine is not destroyed and can be re-used. This is implied by the **zero or one** multiplicity at the composite side of the composition line.



Tthe **multiplicity** of a composition's association end at the whole side is either 1 or 0..1, depending on the fact if parts are **separable**, that is, if they can be detached and exist on their own.

Aggregation

An aggregation is another special form of association with the intended meaning of a part-whole-relationship, where the parts of a whole can be shared with other wholes. For instance, we can model an aggregation between the classes `DegreeProgram` and `Course`, as shown in the following diagram, since a course is part of a degree program and a course can be shared among two or more degree programs (e.g. an engineering degree could share a C programmin course with a computer science degree)..



However, this characteristic of **shareable parts** doesn't mean much, really, so the UML concept of an aggregation doesn't have much semantics (the UML spec says: "Precise semantics of shared aggregation varies by application area and modeler").

The **multiplicity** of an aggregation's association end at the whole side may be any number (*) because a part may belong to, or **shared** among, any number of wholes.

Chapter 7. Implementing Unidirectional Functional Associations with Plain JavaScript

The two example apps that we have discussed in previous chapters, the *minimal app* and the *validation 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 4 and 5.

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 JavaScript data model from an association-free information design model with single-valued reference properties representing *unidirectional 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.

Implementing Single-Valued Reference Properties in 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);
    ...
```

```
  }
}
```

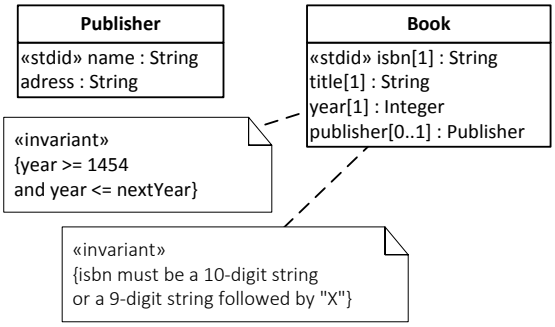
Notice that, for flexibility, the constructor parameter `slots` may contain either a `publisher` slot representing an (internal) JavaScript object reference or a `publisherIdRef` slot representing an (external) ID reference (or foreign key). We handle the resulting ambiguity in the property setter by 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 froeign key), since this is the standard identifier (or primary key) of the `Book` class.

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). We now show how to derive a JavaScript data model from this 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 operationm, 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.

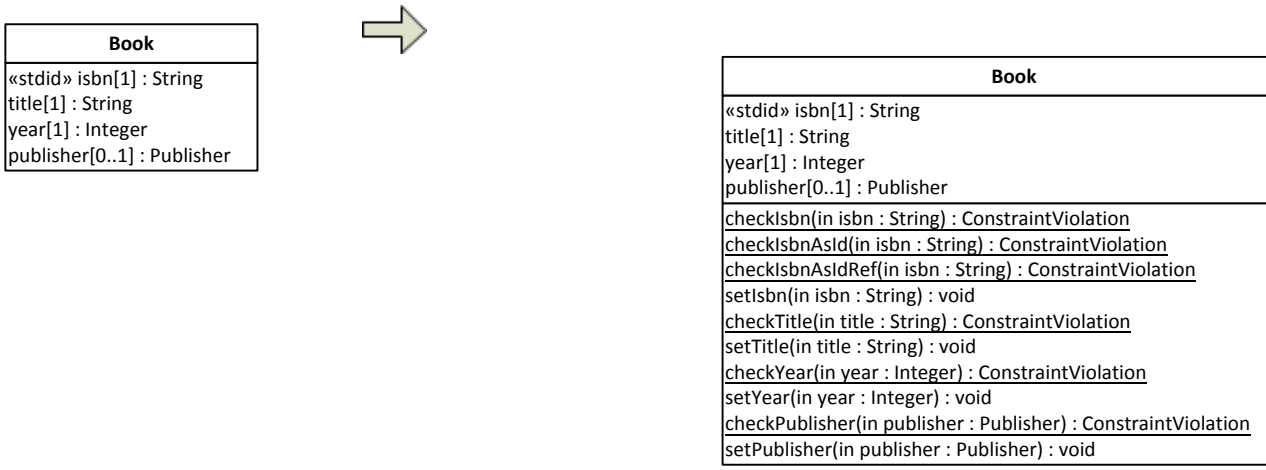
   c. An *ID reference* check operation, such as `checkIsbnAsIdRef`, for checking the *referential integrity* constraint that is required for an *ID reference (IdRef)* (or *foreign key*) attribute.

   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.

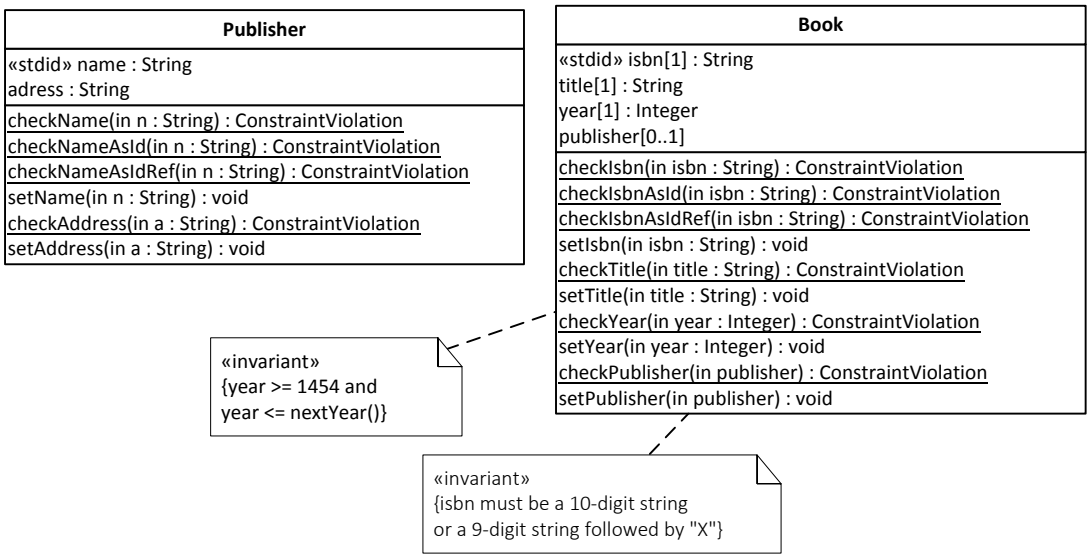
3. Create **add** and **remove** operations for each non-derived **multi-valued** property.

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 7.1. The complete JavaScript data model



New issues

Compared to the single-class app discussed in Part 2 (Validation Tutorial [validation-tutorial.html]), 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. validation of ***referential integrity constraints*** *(ID references)*

   b. 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.

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.

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 the add/remove operations as (instance-level) methods that invoke the corresponding property checks.

5. Encode any other operation.

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

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 constructor body, we first assign default values to all properties. These values will be used when the constuctor 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 a JSON table (a map of records): This property is initially set to `{}`. For instance, in the case of the model class `Publisher`, we define:

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

We encode the class `Book` 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`.

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;
};
```

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;
  }
};
```

Encode the add and remove operations

In our Piublisher-Book example we do not have any muilti-valued property, so we do not have to encode any add/remove operation. But in general, we could have a multi-valued reference property representing a unidirectional association. In that case we would have an *add* and a *remove* operation for this property and we would encode these operations as (instance-level) methods that invoke the corresponding check-as-IdRef operations.

Implement a deletion policy

For any reference property, we have to choose and implement one of the two possible deletion policies discussed in 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.");
};
```

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.name + " while deserializing a book row: " + e.message);
  }
  return book;
};
```

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).

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.

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.

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 8. 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.

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]);
    }
```

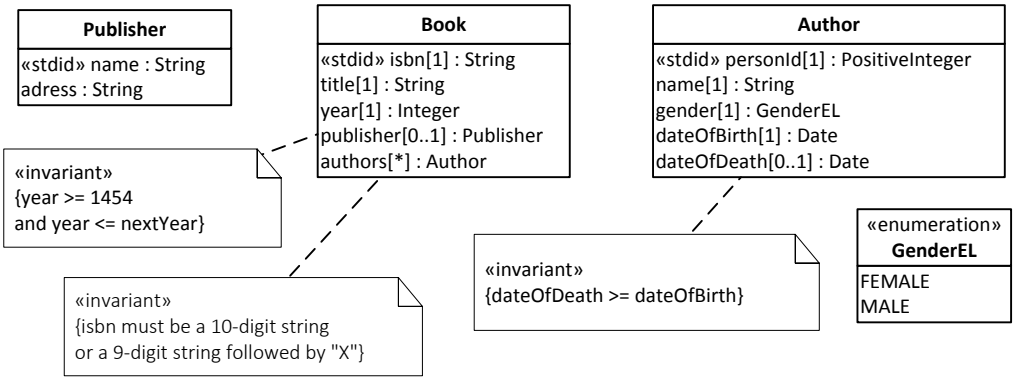
```
  } else {  // map of object references
    keys = Object.keys( a);
    for (i=0; i < keys.length; i++) {
      this.addAuthor( a[keys[i]]);
    }
  }
};
```

A set-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 set-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.

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*:



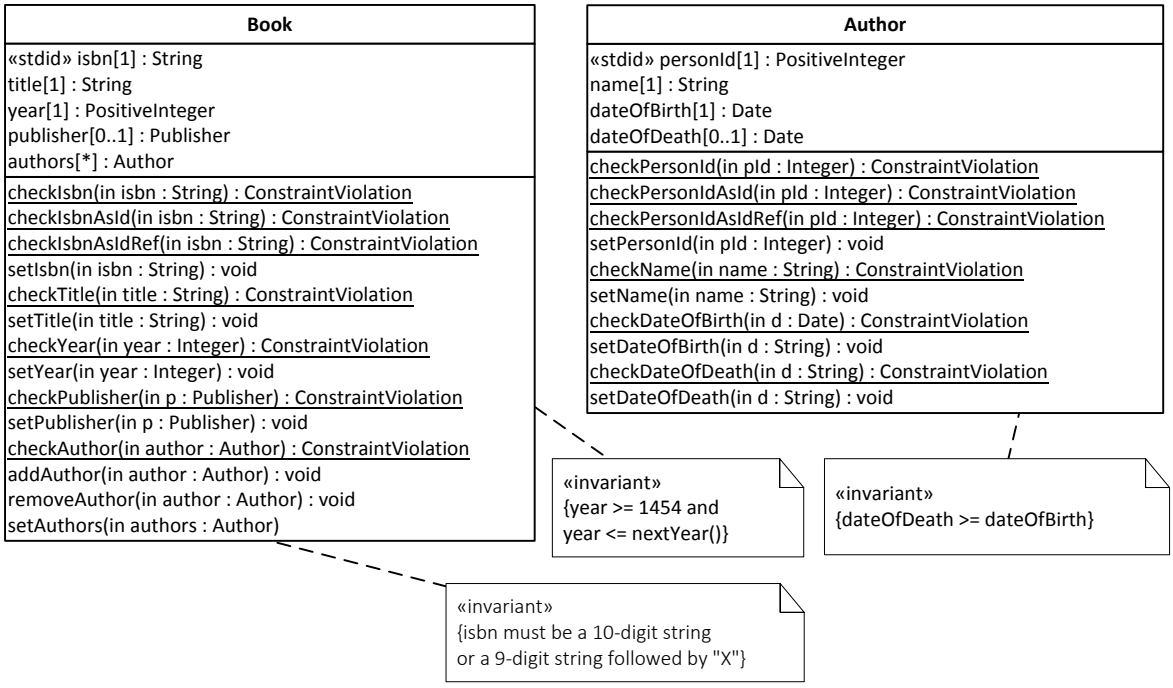
We now show how to derive a JavaScript data model from this design model in three steps.

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 7.1:



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

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-selection widget* has to be used.

Write the Model Code

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]]);
      }
    }
};
```

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.");
};
```

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;
};
```

Write the User Interface Code

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);
  }
},
```

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..

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 ***association list widget*** that allows to add (and remove) objects to (and from) a list of associated objects,

as discussed in . 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, an *association list 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 association list 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
    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 the associated authors selection widget
    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);
    }
  }
```

Run the App and Get the Code

You can run the example app [UnidirectionalAssociationApp/index.html] from our server or download it as a ZIP archive file [UnidirectionalAssociationApp.zip].

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.

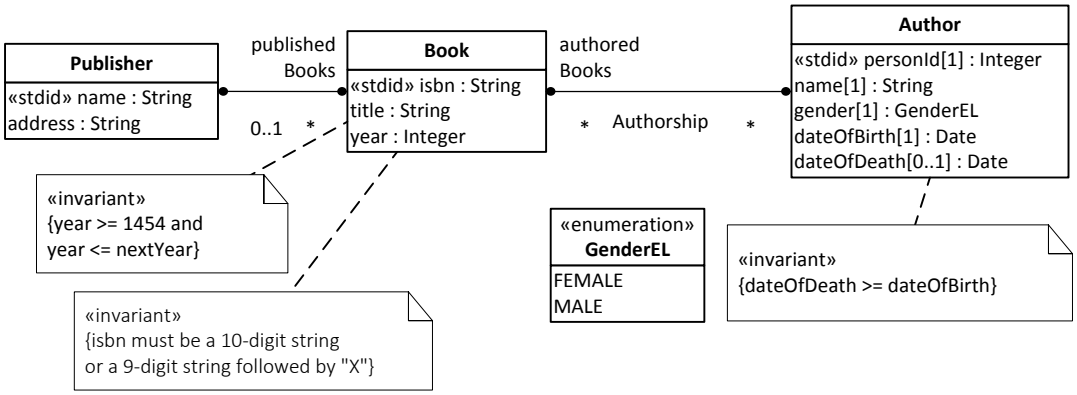
We have still included the repetitive code structures (called ***boilerplate code***) in the model layer per class and per property for constraint validation (checks and setters) and per class for the data storage management methods `add`, `update`, and `destroy`. While it is good to write this code 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 of our tutorial series, 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.

Chapter 9. Bidirectional Associations

A bidirectional association is an association that is represented as a pair of mutually inverse reference properties.

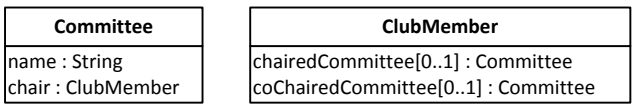
The model shown in Figure 9.1 below (about publishers, books and their authors) serves as our running example in all other parts of the tutorial. Notice that it contains two bidirectional associations, as indicated by the ownership dots at both association ends.

Figure 9.1. The Publisher-Book-Author information design model with two bidirectional associations

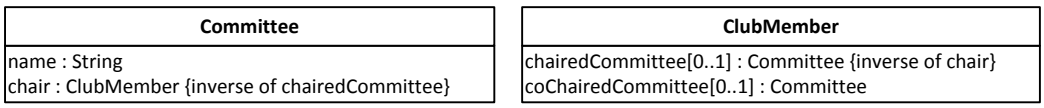


Inverse Reference Properties

For being able to easily retrieve the committees that are chaired or co-chaired by a club member, we add two reference properties to our `Committee-ClubMember` example model: the property of a club member to chair a committee (`ClubMember`::`chairedCommittee`) and the property of a club member to co-chair a committee (`ClubMember`::`coChairedCommittee`). We assume that any club member may chair or co-chair at most one committee (where the disjunction is non-exclusive). So, we get the following model:

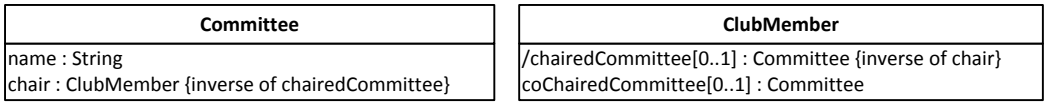


Notice that there is a close correspondence between the two reference properties `Committee`::`chair` and `ClubMember`::`chairedCommittee`. They are the **inverse** of each other: when the club member Tom is the chair of the budget committee, expressed by the tuple *"(budget committe"e, "To"m)*, then the budget committee is the committee chaired by the club member Tom, expressed by the inverse tuple *"(To"m, "budget committe"e)*. For expressing this inverse correspondence in the diagram, we append an inverse property constraint, `inverse of chair`, in curly braces to the declaration of the property `ClubMember`::`chairedCommittee`, and a similar one to the property `Committee`::`chair`, as shown in the following diagram:



We also call `Committee`::`chair` and `ClubMember`::`chairedCommittee` a **pair of mutually inverse reference properties**. Having such a pair in a model means having **redundancy** because each of the two involved reference properties can be derived from the other by inversion. This type of redundancy implies *data storage overhead* and *update overhead*, which is the price to pay for the bidirectional navigabilty that supports efficient object access in both directions.

For maintaining the duplicate information of a mutually inverse reference property pair, it is good practice to treat one of the two involved properties as the *master*, and the other one as the *slave*, and take this distinction into consideration in the code of the change methods (such as the property setters) of the affected model classes. We indicate the slave of an inverse reference property pair in a model diagram by declaring the slave property to be a **derived** property using the UML notation of a slash as a prefix of the property name as shown in the following diagram:



The property `chairedCommittee` in `ClubMember` now has a slash-prefix (`/`) indicating that it is derived.

In a UML class diagram, the derivation of a property can be specified, for instance, by an *Object Constraint Language (OCL)* expression that evaluates to the value of the derived property for the given object. In the case of a property being the inverse of another property, specified by the constraint expression `{inverse of` *anotherProperty*`}` appended to the property declaration, the derivation expression is implied. In our example, it evaluates to the committee object reference `c` such that `c.chair = this`.

There are two ways how to realize the derivation of a property: it may be *derived on read* via a read-time computation of its value, or it may be *derived on update* via an update-time computation performed whenever one of the variables in the derivation expression (typically, another property) changes its value. The latter case corresponds to a *materialzed view* in an SQL database. While a reference property that is derived on read may not guarantee efficient navigation, because the on-read computation may create unacceptable latencies, a reference property that is derived on update does provide effificient navigation.

In the case of a derived reference property, the derivation expresses **life cycle dependencies**. These dependencies require special consideration in the code of the affected model classes by providing a number of change management mechanisms based on the functionality type of the represented association (either *one-to-one*, *many-to-one* or *many-to-many*).

In our example of the derived inverse reference property `ClubMember`::`chairedCommittee`, which is single-valued and optional, this means that

1. whenever a new committee object is created (with a mandatory `chair` assignment), the corresponding `ClubMember`::`chairedCommittee` property has to be assigned accordingly;

2. whenever the `chair` property is updated (that is, a new chair is assigned to a committee), the corresponding `ClubMember`::`chairedCommittee` property has to be updated as well;

3. whenever a committee object is destroyed, the corresponding `ClubMember`::`chairedCommittee` property has to be unassigned.

In the case of a derived inverse reference property that is multi-valued while its inverse base property is single-valued (like `Publisher`::`publishedBooks` in Figure 9.4 below being derived from `Book`::`publisher`), the life cycle dependencies imply that

1. whenever a new 'base object' (such as a book) is created, the corresponding inverse property has to be updated by adding a reference to the new base object to its value set (like adding a reference to the new book object to `Publisher`::`publishedBooks`);

2. whenever the base property is updated (e.g., a new publisher is assigned to a book), the corresponding inverse property (in our example, `Publisher`::`publishedBooks`) has to be updated as well by removing the old object reference from its value set and adding the new one;

3. whenever a base object (such as a book) is destroyed, the corresponding inverse property has to be updated by removing the reference to the base object from its value set (like removing a reference to the book object to be destroyed from `Publisher`::`publishedBooks`).

We use the slash-prefix (/) notation in the example above for indicating that the property `ClubMember`::`chairedCommittee` is derived on update from the corresponding committee object.

Making an Association-Free Information Design Model

How to eliminate explicit associations from a 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 representing implicit associations, we have to eliminate all explicit associations for obtaining an OO design model.

The basic procedure

The starting point of our **association elimination** procedure is an information design model with various kinds of uni-directional and bi-directional associations, such as the model shown in Figure 9.1 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, which is typically based on navigability requirements.

Notice that both associations in the *Publisher-Book-Author* information design model, *publisher-publishedBooks* and *authoredBooks-authors* (or *Authorship*), are bi-directional as indicated by the ownership dots at both association ends. For eliminating all explicit associations from an information design model, we have to perform the following steps:

1. **Eliminate uni-directional associations**, connecting a source with a target class, by replacing them with a reference property in the source class such that the target class is its range.

2. **Eliminate bi-directional associations** by replacing them with a pair of mutually inverse reference properties.

How to eliminate uni-directional associations

A uni-directional association connecting a source with a target class is replaced with a corresponding reference property in its source class having the target class as its range. Its multiplicity is the same as the multiplicity of the target association end. Its name is the name of 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).

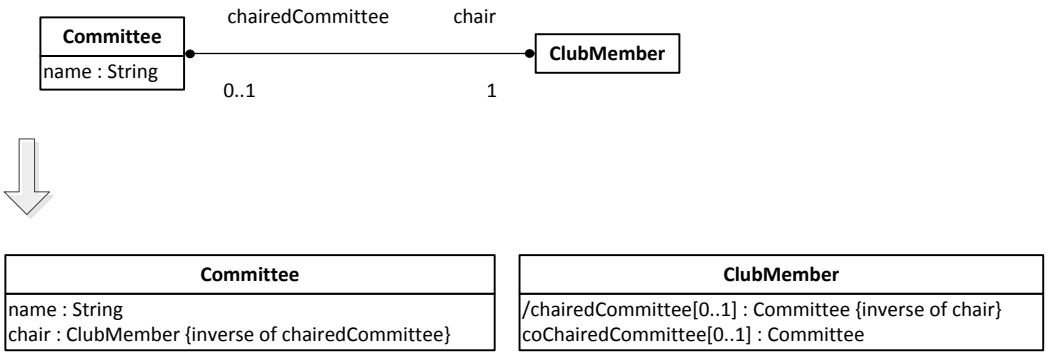
How to eliminate bi-directional associations

A bi-directional association, such as `Authorship` in the model shown in Figure 9.1 above, is replaced with a pair of mutually inverse reference properties, such as `Book`::`authors` and `Author`::`authoredBooks`. Since both reference properties represent the same information (the same set of binary relationships), it's an option to conisder one of them being the master and the other one the slave, which is derived from the master. We discuss the two cases of a one-to-one and a many-to-many association

1. In the case of a bi-directional one-to-one association, this leads to a pair of mutually inverse single-valued reference properties, one in each of the two associated classes. Since both of them represent essentially the same information (one of them is the inverse of the other), one has to choose which of them is considered the "master", and which of them is the "slave", where the "slave" property is considered to represent the inverse of the "master". In the slave class, the reference property

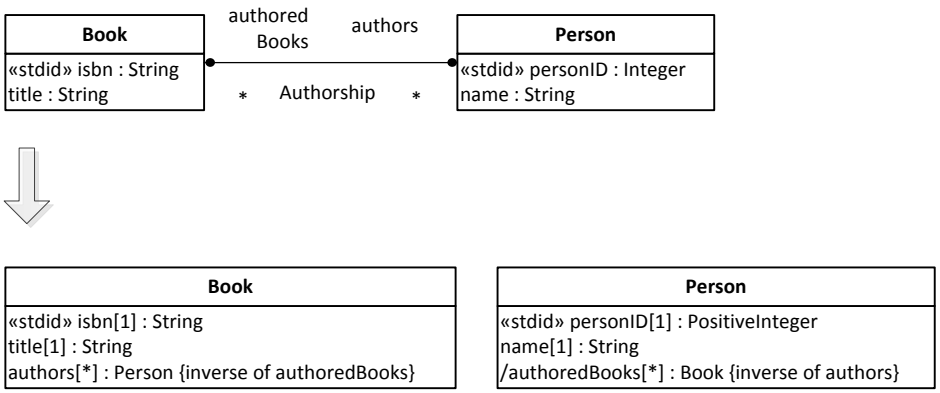
representing the inverse association is designated as a ***derived property*** that is automatically updated whenever 1) a new master object is created, 2) the master reference property is updated, or 3) a master object is destroyed.

Figure 9.2. Turn a bi-directional one-to-one association into a pair of mutually inverse single-valued reference properties



2. A bi-directional many-to-many association is mapped to a pair of mutually inverse multi-valued reference properties, one in each of the two classes participating in the association. Again, in one of the two classes, the multi-valued reference property representing the (inverse) association is designated as a ***derived property*** that is automatically updated whenever the corresponding property in the other class (where the association is maintained) is updated.

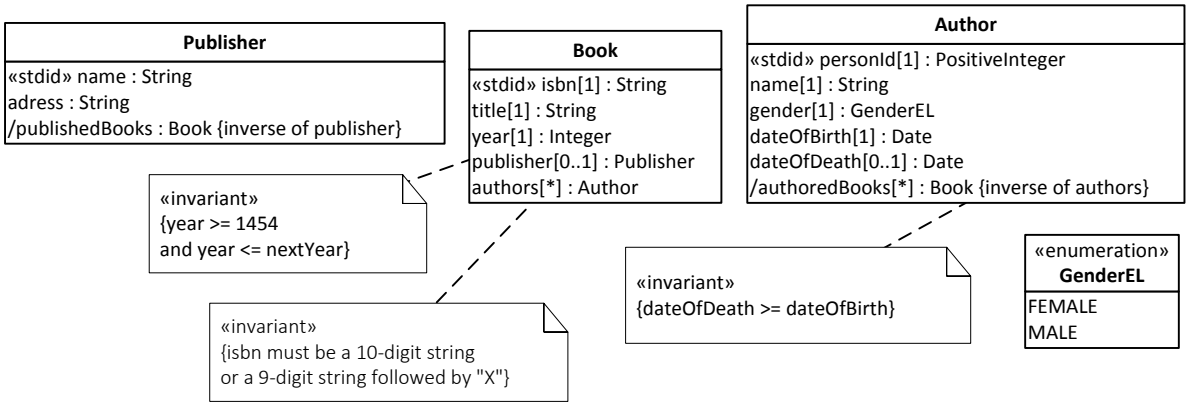
Figure 9.3. Turn a bi-directional many-to-many association into a pair of mutually inverse multi-valued reference properties



The resulting association-free design model

After replacing both bidirectional associations with reference properties, we obtain the following association-free design model:

Figure 9.4. The association-free design model



Chapter 10. Implementing Bidirectional Associations with Plain JavaScript

In this chapter of our tutorial, we show

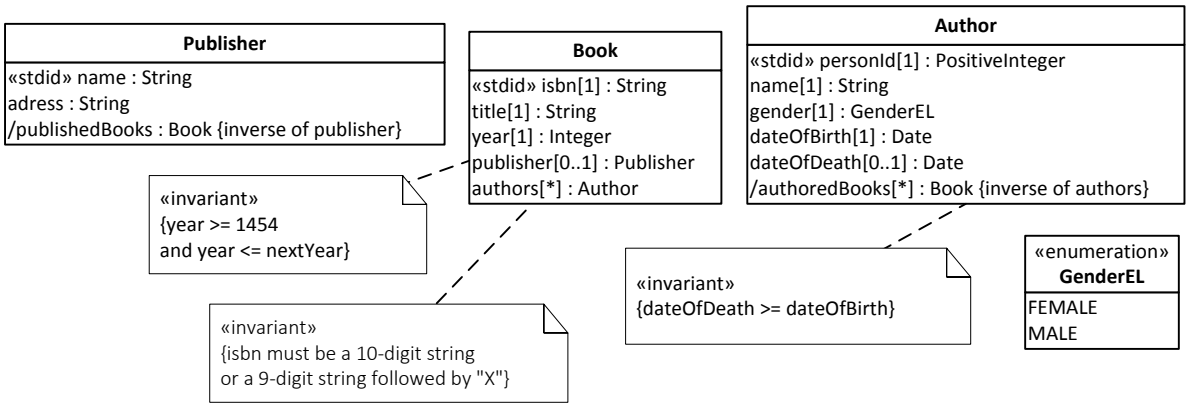
1. how to derive a JavaScript data model from an association-free information design model with **derived inverse reference properties**,

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.

Make a JavaScript Data Model

The starting point for making our JavaScript data model is an association-free information design model with derived inverse reference properties like the following one:



Notice that there are two derived inverse reference properties: `Publisher::/publishedBooks` and `Author::/authoredBooks`. We now show how to derive a JavaScript data model from this design model in three steps.

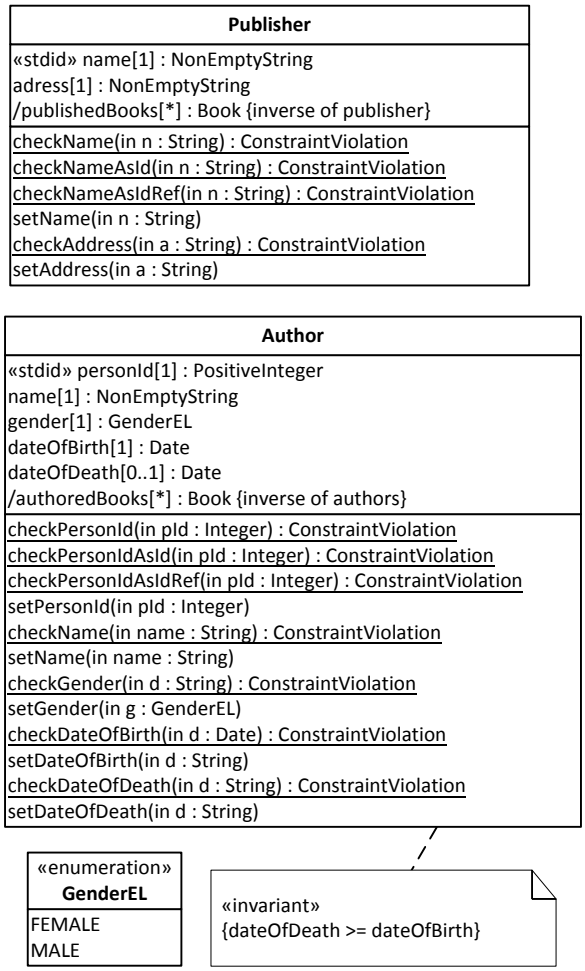
1. Create a **check** operation for each non-derived property. This step has been discussed in detail in the previous parts of the tutorial (about data validation and about unidirectional associations).

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.

This leads to the following JavaScript data model classes `Publisher` and `Author`. Notice that we don't show the model class `Book`, since it is the same as in the data model for the unidirectional association app discussed in the previous Part of our tutorial.

Figure 10.1. The JavaScript data model without `Book`



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.

New issues

Compared to the *unidirectional association app* discussed in a previous tutorial, we have to deal with a number of new technical issues:

1. In the *model code* you now have to take care of maintaining the derived inverse reference properties by maintaining the derived (sets of) inverse references that form the values of a derived inverse reference property. This requires in particular that

   a. whenever the value of a ***single-valued*** master reference property is **initialized or updated** with the help of a setter (such as assigning a reference to an `Publisher` instance `p` to `b.publisher` for a `Book` instance `b`), an inverse reference has to be assigned or added to the corresponding value of the derived inverse reference property (such as adding `b` to `p.publishedBooks`); when the value of the master reference property is *updated* and the derived inverse reference property is *multi-valued*, then the obsolete inverse reference to the previous value of the single-valued master reference property has to be deleted;

b. whenever the value of an optional ***single-valued*** master reference property is ***unset*** (e.g. by assigning `null` to `b.publisher` for a `Book` instance `b`), the inverse reference has to be removed from the corresponding value of the derived inverse reference property (such as removing `b` from `p.publishedBooks`), if the derived inverse reference property is multi-valued, otherwise the corresponding value of the derived inverse reference property has to be unset or updated;

c. whenever a reference is **added** to the value of a **multi-valued** master reference property with the help of an add method (such as adding a reference to an `Author` instance `a` to `b.authors` for a `Book` instance `b`), an inverse reference has to be assigned or added to the corresponding value of the derived inverse reference property (such as adding `b` to `a.authoredBooks`);

d. whenever a reference is **removed** from the value of a **multi-valued** master reference property with the help of a `remove` method (such as removing a reference to an `Author` instance `a` from `b.authors` for a `Book` instance `b`), the inverse reference has to be removed from the corresponding value of the derived inverse reference property (such as removing `b` from `a.authoredBooks`), if the derived inverse reference property is multi-valued, otherwise the corresponding value of the derived inverse reference property has to be unset or updated;

e. whenever an object with a single reference or with multiple references as the value of a master reference property is **destroyed** (e.g., when a `Book` instance `b` with a single reference `b.publisher` to a `Publisher` instance `p` is destroyed), the derived inverse refences have to be removed first (e.g., by removing `b` from `p.publishedBooks`).

Notice that when a new object is created with a single reference or with multiple references as the value of a master reference property (e.g., a new `Book` instance `b` with a single reference `b.publisher`), its setter or add method will be invoked and will take care of creating the derived inverse references.

2. In the *UI code* we can now exploit the inverse reference properties for more efficiently creating a list of inversely associated objects in the *list objects* use case. For instance, we can more efficiently create a list of all published books for each publisher. However, we do not allow updating the set of inversely associated objects in the *update object* use case (e.g. updating the set of published books in the *update publisher* use case). Rather, such an update has to be done via updating the master objects (in our example, the books) concerned.

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. If the property is the inverse of a derived reference property (representing a bi-directional association), make sure that the setter also assigns (or adds) corresponding references to (the value set of) the inverse property.

4. Encode the add/remove operations as (instance-level) methods that invoke the corresponding property checks. If the multi-valued reference property is the inverse of a derived reference property (representing a bi-directional association), make sure that both the add and the remove operation also assign/add/remove corresponding references to/from (the value set of) the inverse property.

5. Encode any other operation.

6. Take care of the deletion dependencies in `deleteRow` methods.

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

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

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, optional
  // derived inverse reference property
  this.publishedBooks = {};  // map of Book objects, inverse of Book::publisher

  // constructor invocation with arguments
  if (arguments.length > 0) {
    this.setName( slots.name);
    this.setAddress( slots.address);
  }
};
```

Notice that we have added the (derived) multi-valued reference property `publishedBooks`, but we do not assign it in the constructor function because it will be assigned when the inverse reference property `Book::publisher` will be assigned.

Encode the property checks

The property checks from the *unidirectional association app* do not need to be changed in any way.

Encode the setter operations

Any setter for a reference property that is coupled to a derived inverse reference property (implementing a bi-directional association), now also needs to assign/add/remove inverse references to/from the corresponding value (set) of the inverse property. An example of such a setter is the following `setPublisher` method:

```
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) {
    if (this.publisher) {  // update existing book record
      // delete the obsolete inverse reference in Publisher::publishedBooks
      delete this.publisher.publishedBooks[ this.isbn];
    }
    // assign the new publisher reference
    this.publisher = Publisher.instances[ publisherIdRef];
    // add the inverse reference to publisher.publishedBooks
    this.publisher.publishedBooks[ this.isbn] = this;
  } else {
    throw constraintViolation;
  }
};
```

Encode the add and remove operations

For any multi-valued reference property that is coupled to a derived inverse reference propertiy, both the *add* and the *remove* method also have to assign/add/remove corresponding references to/from (the value set of) the inverse property.

For instance, for the multi-valued reference property `Book::authors` that is coupled to the derived inverse reference propertiy `Author:authoredBooks` for implementing the bidirectional authorship association between `Book` and `Author`, the `addAuthor` method is encoded in the following way:

```
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) {
    authorIdRefStr = String( authorIdRef);
    // add the new author reference
    this.authors[ authorIdRefStr] = Author.instances[ authorIdRefStr];
    // automatically add the derived inverse reference
    this.authors[ authorIdRefStr].authoredBooks[ this.isbn] = this;
  }
};
```

For the remove operation `removeAuthor` we obtain the following code:

```
Book.prototype.removeAuthor = function( a) {
  var constraintViolation=null, authorIdRef=0, authorIdRefStr="";
  // an author can be given as an ID reference or an object reference
  if (typeof(a) !== "object") {
    authorIdRef = parseInt( a);
  } else {
    authorIdRef = a.authorId;
  }
  constraintViolation = Book.checkAuthor( authorIdRef);
  if (authorIdRef && constraintViolation instanceof NoConstraintViolation) {
    authorIdRefStr = String( authorIdRef);
    // automatically delete the derived inverse reference
    delete this.authors[ authorIdRefStr].authoredBooks[ this.isbn];
    // delete the author reference
    delete this.authors[ authorIdRefStr];
  }
};
```

Take care of deletion dependencies

When a `Book` instance `b`, with a single reference `b.publisher` to a `Publisher` instance `p` and multiple references `b.authors` to `Author` instances, is destroyed, the derived inverse references have to be removed first (e.g., by removing `b` from `p.publishedBooks`).

```
Book.deleteRow = function (isbn) {
  var book = Book.instances[isbn], keys=[], i=0;
  if (book) {
```

```
    console.log( book.toString() + " deleted!");
    if (book.publisher) {
      // remove inverse reference from book.publisher
      delete book.publisher.publishedBooks[isbn];
    }
    // remove inverse references from all book.authors
    keys = Object.keys( book.authors);
    for (i=0; i < keys.length; i++) {
      delete book.authors[keys[i]].authoredBooks[isbn];
    }
    // finally, delete book from Book.instances
    delete Book.instances[isbn];
  } else {
    console.log("There is no book with ISBN " + isbn + " in the database!");
  }
};
```

Exploiting Derived Inverse Reference Properties in the User Interface

We can now exploit the derived inverse reference properties `Publisher::publishedBooks` and `Author::authoredBooks` for more efficiently creating a list of associated books in the *list publishers* and *list authors* use cases.

Show information about published books in the *List Publishers* use case

For showing information about published books in the *list publishers* use case, we can now exploit the derived inverse reference property `publishedBooks`:

```
pl.view.publishers.list = {
  setupUserInterface: function () {
    var tableBodyEl = document.querySelector("div#listPublishers>table>tbody");
    var pKeys = Object.keys( Publisher.instances);
    var row=null, publisher=null, listEl=null;
    tableBodyEl.innerHTML = "";
    for (var i=0; i < pKeys.length; i++) {
      publisher = Publisher.instances[pKeys[i]];
      row = tableBodyEl.insertRow(-1);
      row.insertCell(-1).textContent = publisher.name;
      row.insertCell(-1).textContent = publisher.address;
      // create list of books published by this publisher
      listEl = util.createListFromAssocArray( publisher.publishedBooks, "title"
      row.insertCell(-1).appendChild( listEl);
    }
    document.getElementById("managePublishers").style.display = "none";
    document.getElementById("listPublishers").style.display = "block";
  }
};
```

Part IV. Inheritance in Class Hierarchies

Subtypes and inheritance are important elements of information models. 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, application development frameworks do often not provide much support for dealing with subtypes and inheritance.

Table of Contents

Chapter 11. Subtyping and Inheritance

The concept of a *category*, or *subclass*, is a fundamental concept in natural language, mathematics, and informatics. For instance, in English, we say that *a bird is an animal*, or the class of all birds is a **subclass** of the class of all animals. In linguistics, the noun *bird* is a **hyponym** of the noun *animal*.

An object type may be specialized by subtypes (for instance, *Bird* is specialized by *Parrot*) or generalized by supertypes (for instance, *Bird* and *Mammal* are generalized by *Animal*). Specialization and generalization are two sides of the same coin.

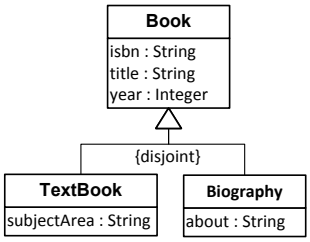
A category **inherits** all features from its supertypes. When a category inherits attributes, associations and constraints from a supertype, this means that these features need not be explicitly rendered for the category in the class diagram, but the reader of the diagram has to know that all features of a supertype also apply to its subtypes.

When an object type has more than one supertype, we have a case of **multiple inheritance**, which is common in conceptual modeling, but prohibited in many object-oriented programming languages, such as Java and C#, where subtyping leads to **class hierarchies** with a unique direct supertype for each object type.

Introducing Subtypes by Specialization

A new category may be introduced by specialization whenever new features of more specific types of objects have to be captured. We illustrate this for our example model where we want to capture text books and biographies as special cases of books. This means that text books and biographies also have an ISBN, a title and a publishing year, but in addition they have further features such as the attributes `subjectArea` of text books and `about` of biographies. Consequerntly, we introduce the object types `TextBook` and `Biography` by specializing the object type `Book`, that is, as subtypes of `Book`.

Figure 11.1. The object type `Book` is specialized by two subtypes: `TextBook` and `Biography`



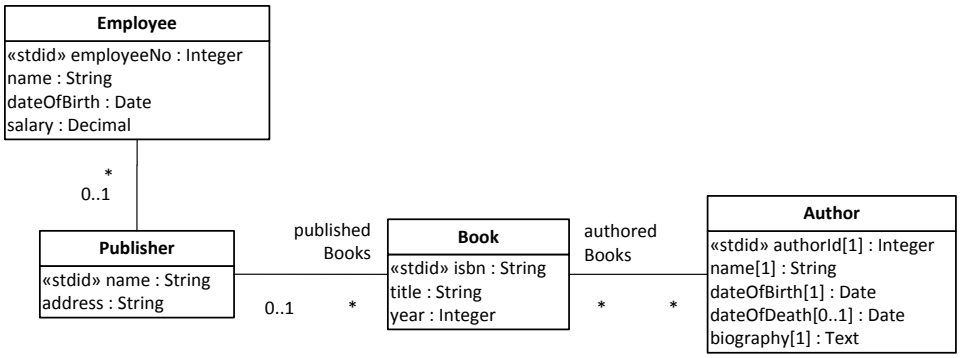
When specializing an object type, we define additional features for the newly added category. In many cases, these additional features are more specific properties. For instance, in the case of `TextBook` specializing `Book`, we define the additional attribute `subjectArea`. In some programming languages, such as in Java, it is therefore said that the category **extends** the supertype.

However, we can also specialize an object type without defining additional properties (or operations/methods), but by defining aditional constraints.

Introducing Supertypes by Generalization

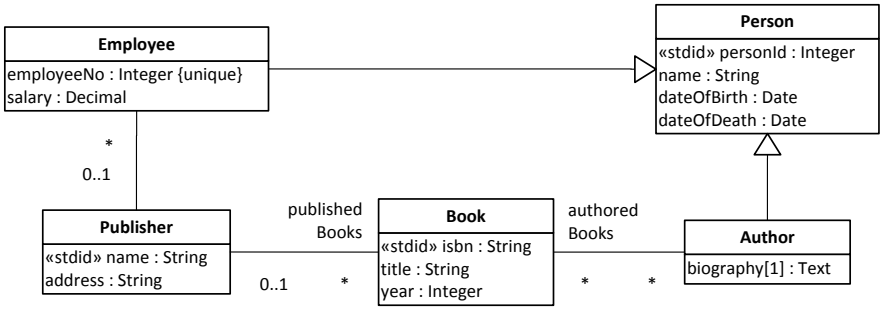
We illustrate generalization with the following example, which extends the information model of Part 4 by adding the object type `Employee` and associating employees with publishers.

Figure 11.2. The object types `Employee` and `Author` share several attributes



After adding the object type `Employee` we notice that `Employee` and `Author` share a number of attributes due to the fact that both employees and authors are people, and *being an employee* as well as *being an author* are **roles** played by people. So, we may generalize these two object types by adding a joint supertype `Person`, as shown in the following diagram.

Figure 11.3. The object types `Employee` and `Author` have been generalized by adding the common supertype `Person`



When generalizing two or more object types, we move (and centralize) a set of features shared by them in the newly added supertype. In the case of `Employee` and `Author`, this set of shared features consists of the attributes `name`, `dateOfBirth` and `dateOfDeath`. In general, shared features may include attributes, associations and constraints.

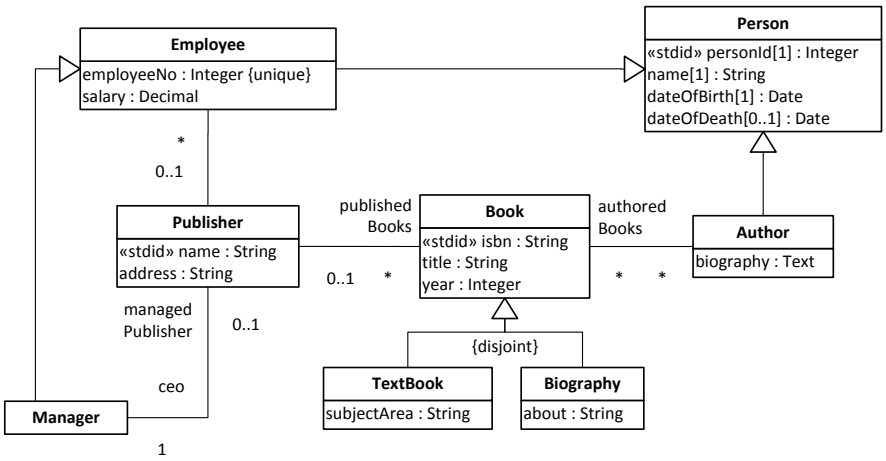
Notice that since in an information design model, each top-level class needs to have a standard identifier, in the new class `Person` we have declared the standard identifier attribute `personId`, which is inherited by all subclasses. Therefore, we have to reconsider the attributes that had been declared to be standard identifiers in the subclasses before the generalization. In the case of `Employee`, we had declared the attribute `employeeNo` as a standard identifier. Since the employee number is an important business information item, we have to keep this attribute, even if it is no longer the standard idenifier. Because it is still an alternative idenifier (a "key"), we declare it to be *unique*. In the case of `Author`, we had declared the attribute `authorId` as a standard identifier. Assuming that this attribute represents a purely technical, rather than business, information item, we dropped it, since it's no longer needed as an identifier for authors. Consequently, we end up with a model which allows to identify employees either by their employee number or by their `personId` value, and to identify authors by their `personId` value.

We consider the following extension of our original example model, shown in Figure 11.4, where we have added two class hierarchies:

1. the disjoint (but incomplete) segmentation of `Book` into `TextBook` and `Biography`,

2. the overlapping and incomplete segmentation of `Person` into `Author` and `Employee`, which is further specialized by `Manager`.

Figure 11.4. The complete class model containing two inheritance hierarchies



Intension versus Extension

The **intension** of an object type is given by the set of its features, including attributes, associations and operations.

The **extension** of an object type is the set of all objects instantiating the object type. The extension of an object type is also called its *population*.

We have the following duality: while all features of a supertype are included in the features of its subtypes (intensional inclusion), all instances of a category are included in the instances of its supertypes (extensional inclusion). This formal structure has been investigated in Figure formal concept analysis [http://en.wikipedia.org/wiki/Formal_concept_analysis].

Due to the intension/extension duality we can specialize a given type in two different ways:

1. By **extending the type's intension** through adding features in the new category (such as adding the attribute `subjectArea` in the category `TextBook`).

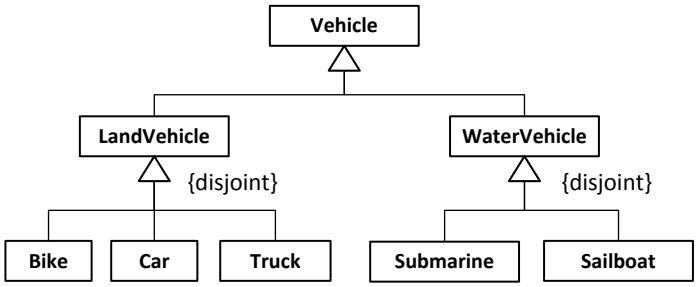
2. By **restricting the type's extension** through adding a constraint (such as defining a category `MathTextBook` as a `TextBook` where the attribute `subjectArea` has the specific value "Mathematics").

Typical OO programming languages, such as Java and C#, only support the first possibility (specializing a given type by extending its intension), while XML Schema and SQL99 also support the second possibility (specializing a given type by restricting its extension).

Type Hierarchies

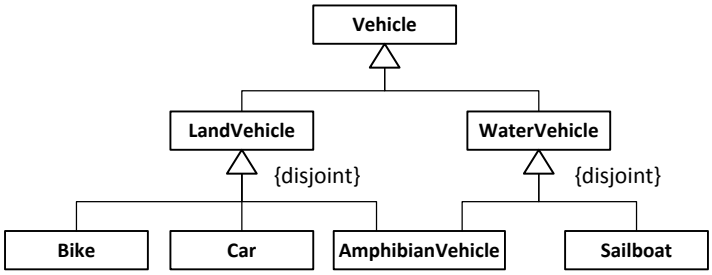
A *type hierarchy* (or *class hierarchy*) consists of two or more types, one of them being the root (or top-level) type, and all others having at least one direct supertype. When all non-root types have a unique direct supertype, the type hierarchy is a **single-inheritance hierarchy**, otherwise it's a **multiple-inheritance hierarchy**. For instance, in 11.5 below, the class `Vehicle` is the root of a single-inheritance hierarchy, while Figure 11.6 shows an example of a multiple-inheritance hierarchy, due to the fact that `AmphibianVehicle` has two direct superclasses: `LandVehicle` and `WaterVehicle`.

Figure 11.5. A class hierarchy having the root class `Vehicle`



The simplest case of a class hierarchy, which has only one level of subtyping, is called a *generalization set* in UML, but may be more naturally called **segmentation**. A segmentation is **complete**, if the union of all subclass extensions is equal to the extension of the superclass (or, in other words, if all instances of the superclass instantiate some subclass). A segmentation is **disjoint**, if all subclasses are pairwise disjoint (or, in other words, if no instance of the superclass instantiates more than one subclass). Otherwise, it is called *overlapping*. A complete and disjoint segmentation is a **partition**.

Figure 11.6. A multiple inheritance hierarchy



In a class diagram, we can express these constraints by annotating the shared generalization arrow with the keywords *complete* and *disjoint* enclosed in braces. For instance, the annotation of a segmentation with {complete, disjoint} indicates that it is a partition. By default, whenever a segmentation does not have any annotation, like the segmentation of `Vehicle` into `LandVehicle` and `WaterVehicle` in Figure 11.6 above, it is {incomplete, overlapping}.

An information model may contain any number of class hierarchies.

The *Class Hierarchy Merge* Design Pattern

Consider the simple class hierarchy of the design model in Figure 11.1 above, showing a disjoint segmentation of the class `Book`. In such a case, whenever there is only one level (or there are only a few levels) of subtyping and each category has only one (or a few) additional properties, it's an option to refactor the class model by merging all the additional properties of all subclasses into an expanded version of the root class such that these subclasses can be dropped from the model, leading to a simplified model.

This *Class Hierarchy Merge* design pattern comes in two forms. In its simplest form, the segmentations of the original class hierarchy are **disjoint**, which allows to use a single-valued `category` attribute for representing the specific category of each instance of the root class corresponding to the unique subclass instantiated by it. When the segmentations of the original class hierarchy are not disjoint, that is, when at least one of them is **overlapping**, we need to use a multi-valued `category` attribute for representing the set of types instantiated by an object. In this tutorial, we only discuss the simpler case of *Class Hierarchy Merge* refactoring for disjoint segmentations, where we take the following refactoring steps:

1. Add an **enumeration type** that contains a corresponding enumeration literal for each segment subclass. In our example, we add the enumeration type `BookCategoryEL`.

2. Add a `category` attribute to the root class with this enumeration as its range. The `category` attribute is mandatory [1], if the segmentation is complete, and optional [0..1], otherwise. In our example, we add a `category` attribute with range `BookCategoryEL` to the class `Book`. The `category` attribute is optional because the segmentation of `Book` into `TextBook` and `Biography` is incomplete.

3. Whenever the segmentation is **rigid** (does not allow *dynamic classification*), we designate the `category` attribute as **frozen**, which means that it can only be assigned once by setting its value when creating a new object, but it cannot be changed later.

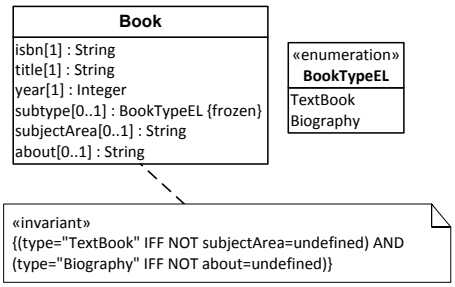
4. Move the properties of the segment subclasses to the root class, and make them **optional**. We call these properties, which are typically listed below the `category` attribute, **segment properties**. In our example, we move the attribute `subjectArea` from `TextBook` and `about` from `Biography` to `Book`, making them *optional*, that is [0..1].

5. Add a constraint (in an invariant box attached to the expanded root class rectangle) enforcing that the optional subclass properties have a value if and only if the instance of the root class instantiates the corresponding category. In our example, this means that an instance of `Book` is of category "TextBook" if and only if its attribute `subjectArea` has a value, and it is of category "Biography" if and only if its attribute `about` has a value.

6. Drop the segment subclasses from the model.

In the case of our example, the result of this design refactoring is shown in Figure 11.7 below. Notice that the constraint (or "invariant") represents a logical sentence where the logical operator keyword "IFF" stands for the logical equivalence operator "if and only if" and the property condition `prop=undefined` tests if the property `prop` does not have a value.

Figure 11.7. The design model resulting from applying the Class Hierarchy Merge design pattern



Subtyping and Inheritance in Computational Languages

Subtyping and inheritance have been supported in *Object-Oriented Programming (OOP)*, in database languages (such as *SQL99*), in the XML schema definition language *XML Schema*, and in other computational languages, in various ways and to different degrees. At its core, subtyping in computational languages is about defning type hierarchies and the inheritance of features: mainly properties and methods in OOP; table columns and constraints in SQL99; elements, attributes and constraints in XML Schema.

In general, it is desirable to have support for **multiple classification** and **multiple inheritance** in type hierarchies. Both language features are closely related and are considered to be advanced features, which may not be needed in many applications or can be dealt with by using workarounds.

Multiple classification means that an object has more than one direct type. This is mainly the case when an object plays multiple roles at the same time, and therefore directly instantiates multiple

classes defining these roles. Multiple inheritance is typically also related to role classes. For instance, a student assistant is a person playing both the role of a student and the role of an academic staff member, so a corresponding OOP class `StudentAssistant` inherits from both role classes `Student` and `AcademicStaffMember`. In a similar way, in our example model above, an `AmphibianVehicle` inherits from both role classes `LandVehicle` and `WaterVehicle`.

Subtyping and Inheritance with OOP Classes

The minimum level of support for subtyping in OOP, as provided, for instance, by Java and C#, allows defining inheritance of propertes and methods in single-inheritance hierarchies, which can be inspected with the help of an **is-instance-of** predicate that allows testing if a class is the direct or an indirect type of an object. In addition, it is desirable to be able to inspect inheritance hierarchies with the help of

1. a pre-defined instance-level property for retrieving **the direct type of an object** (or its *direct types*, if multiple classification is allowed);

2. a pre-defined type-level property for retrieving **the direct supertype of a type** (or its *direct supertypes*, if multiple inheritance is allowed).

A special case of an OOP language is JavaScript, which does not (yet) have an explicit language element for classes, but only for constructors. Due to its dynamic programming features, JavaScript allows using various code patterns for implementing classes, subtyping and inheritance (as we discuss in the next section on JavaScript).

Subtyping and Inheritance with Database Tables

A standard DBMS stores information (objects) in the rows of tables, which have been conceived as set-theoretic relations in classical *relational* database systems. The relational database language SQL is used for defining, populating, updating and querying such databases. But there are also simpler data storage techniques that allow to store data in the form of table rows, but do not support SQL. In particular, key-value storage systems, such as JavaScript's Local Storage API, allow storing a serialization of a **JSON table** (a map of records) as the string value associated with the table name as a key.

While in the classical, and still dominating, version of SQL (SQL92) there is no support for subtyping and inheritance, this has been changed in SQL99. However, the subtyping-related language elements of SQL99 have only been implemented in some DBMS, for instance in the open source DBMS *PostgreSQL*. As a consequence, for making a design model that can be implemented with various frameworks using various SQL DBMSs (including weaker technologies such as *MySQL* and *SQLite*), we cannot use the SQL99 features for subtyping, but have to model inheritance hierarchies in database design models by means of plain tables and foreign key dependencies. This mapping from class hierarchies to relational tables (and back) is the business of ***Object-Relational-Mapping*** frameworks such as Hibernate [http://en.wikipedia.org/wiki/Hibernate_%28Java%29] (or any other JPA [http://en.wikibooks.org/wiki/Java_Persistence/What_is_JPA%3F] Provider) or the Active Record [http://guides.rubyonrails.org/association_basics.html] approach of the Rails [http://rubyonrails.org/] framework.

There are essentially two alternative approaches how to represent a class hierarchy with relational tables:

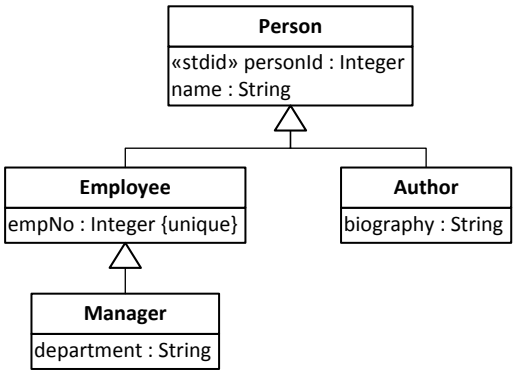
1. **Single Table Inheritance** [http://www.martinfowler.com/eaaCatalog/singleTableInheritance.html] is the simplest approach, where the entire class hierarchy is represented with a single table, containing columns for all attributes of the root class and of all its subclasses, and named after the name of the root class.

2. **Joined Tables Inheritance** [http://en.wikibooks.org/wiki/Java_Persistence/Inheritance#Joined.2C_Multiple_Table_Inheritance] is a more logical approach, where each subclass is represented by a corresponding subtable connected to the supertable via its primary key referencing the primary key of the supertable.

Notice that the *Single Table Inheritance* approach is closely related to the *Class Hierarchy Merge* design pattern discussed in above. Whenever this design pattern has already been applied in the design model, or the design model has already been refactored according to this design pattern, the class hierarchies concerned (their subclasses) have been eliminated in the design, and consequently also in the data model to be encoded in the form of class definitions in the app's model layer, so there is no need anymore to map class hierarchies to database tables. Otherwise, when the *Class Hierarchy Merge* design pattern does not get applied, we would get a corresponding class hierarchy in the app's model layer, and we would have to map it to database tables with the help of the *Single Table Inheritance* approach.

We illustrate both the *Single Table Inheritance* approach and the *Joined Tables Inheritance* with the help of two simple examples. The first example is the `Book` class hierarchy, which is shown in Figure 11.1 above. The second example is the class hierarchy of the `Person` roles `Employee`, `Manager` and `Author`, shown in the class diagram in Figure 11.8 below.

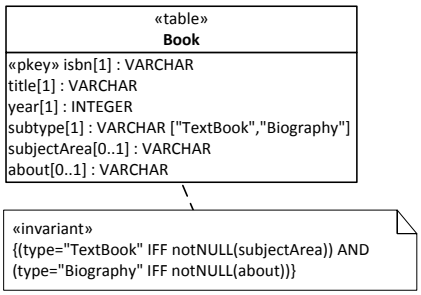
Figure 11.8. A class model with a `Person` roles hierarchy



Single Table Inheritance

Consider the single-level class hierarchy shown in Figure 11.1 above, which is an incomplete disjoint segmentation of the class `Book`, as the design for the model classes of an MVC app. In such a case, whenever we have a model class hierarchy with only one level (or only a few levels) of subtyping and each category has only one (or a few) additional properties, it's preferable to use *Single Table Inheritance*, so we model a single table containing columns for all attributes such that the columns representing additional attributes of subclasses are optional, as shown in the SQL table model in Figure Figure 11.9, “An SQL table model with a single table representing the `Book` class hierarchy” below.

Figure 11.9. An SQL table model with a single table representing the `Book` class hierarchy

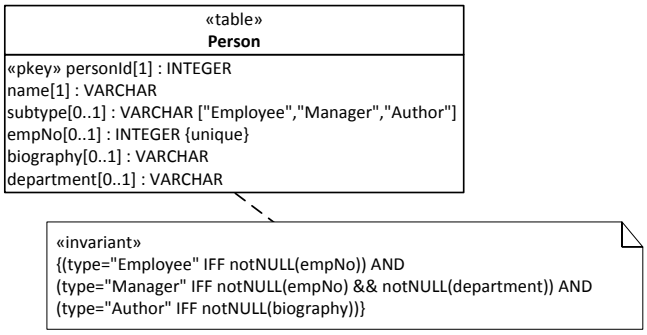


Notice that it is good practice to add a special column, which is called *discriminator column* in JPA and often has the name `category`, for representing the category of each row corresponding to the subclass instantiated by the represented object. Such a `category` column would normally be string-valued, but constrained to one of the names of the subclasses. If the DBMS supports enumerations, it could also be enumeration-valued.

Based on the `category` of a book, we have to enforce that if and only if it is "TextBook", its attribute `subjectArea` has a value, and if and only if it is "Biography", its attribute `about` has a value. This implied constraint is expressed in the invariant box attached to the `Book` table class in the class diagram above, where the logical operator keyword "IFF" represents the logical equivalence operator "if and only if". It needs to be implemented in the database, e.g., with an SQL table CHECK clause or with SQL triggers.

Consider the class hierarchy shown in Figure 11.8 above. With only three additional attributes defined in the subclasses `Employee`, `Manager` and `Author`, this class hierarchy could again be implemented with the *Single Table Inheritance* approach. In the SQL table model, we can express this as shown in Figure 11.10 below.

Figure 11.10. An SQL table model with a single table representing the `Person` roles hierarchy

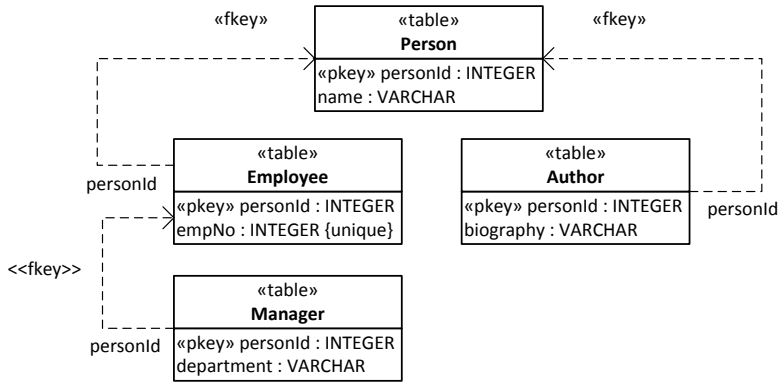


In the case of a multi-level class hierarchy where the subclasses have little in common, the *Single Table Inheritance* approach does not lead to a good representation.

Joined Tables Inheritance

In a more realistic model, the subclasses of `Person` shown in Figure 11.8 above would have many more attributes, so the *Single Table Inheritance* approach would be no longer feasible, and the *Joined Tables Inheritance* approach would be needed. In this approach we get the SQL data model shown in Figure 11.11 below. This SQL table model connects subtables to their supertables by defining their primary key attribute(s) to be at the same time a foreign key referencing their supertable. Notice that foreign keys are visuallized in the form of UML dependency arrows stereotyped with «fkey» and annotated at their source table side with the name of the foreign key column.

Figure 11.11. An SQL table model with the table `Person` as the root of a table hierarchy

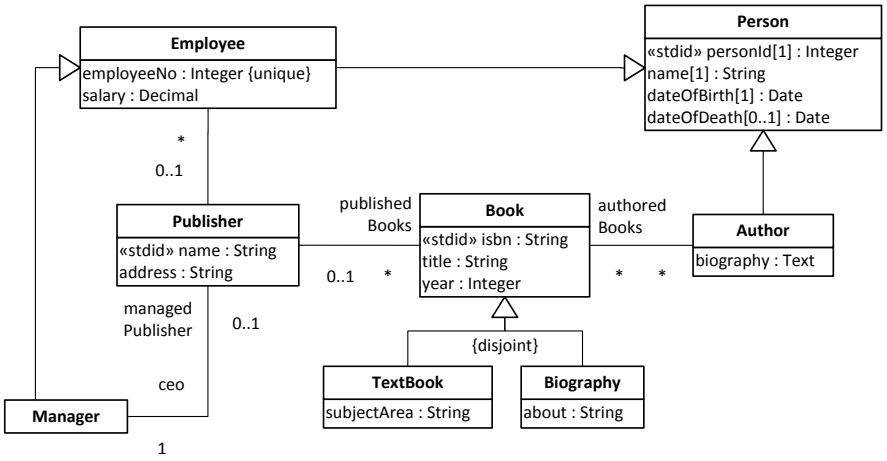


The main disadvantage of the *Joined Tables Inheritance* approach is that for querying any subclass *join queries* are required, which may create a performance problem.

Chapter 12. Subtyping in a Plain JavaScript Frontend App

Whenever an app has to manage the data of a larger number of object types, there may be various **category** (inheritance) relationships between some of the object types. Handling category relationships is an advanced issue in software application engineering. It is often not well supported by application development frameworks.

Figure 12.1. A class model containing two inheritance hierarchies



In this chapter of our tutorial, we first explain the general approach to constructor-based subtyping in JavaScript before presenting two case studies based on fragments of the information model of our running example, the *Public Library* app, shown above:

1. In the first case study, we consider the single-level class hierarchy with root `Book` shown in Figure 12.2 below (which is an incomplete disjoint segmentation) . We use the *Class Hierarchy Merge* design pattern (discussed in Section) for refactoring this class hierarchy to a single class that is mapped to a persistent database table stored with JavaScript's *Local Storage*.

2. In the second case study, we consider the multi-level class hierarchy consisting of the `Person` roles `Employee`, `Manager` and `Author`, shown in Figure 12.2 below. We use the *Joined Tables Inheritance* approach for mapping this class hierarchy to a set of database tables that are related with each other via foreign key dependencies.

In both cases we show

1. how to derive a JavaScript data model, and a corresponding JSON table model, from the class hierarchiy (representing an information design model),

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.

Figure 12.2. The object type `Book` as the root of a disjoint segmentation

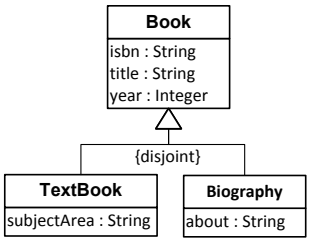
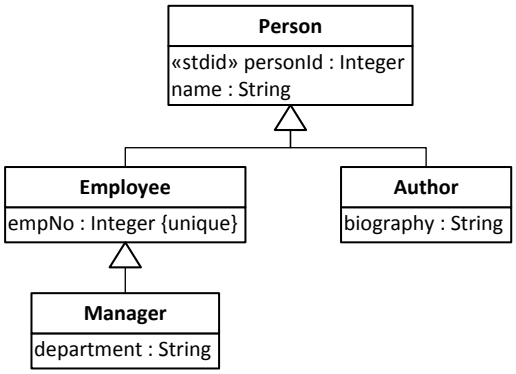


Figure 12.3. The `Person` roles hierarchy



Constructor-Based Subtyping in JavaScript

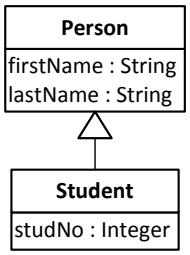
Since JavaScript does not have an explicit class concept, subtyping is not directly supported, but certain forms of subtyping can be implemented with the help of certain code patterns. Any subtyping code pattern should satisfy four requirements, as explained in the previous section. It should provide two inheritance mechanisms: (1) inheritance of properties and (2) inheritance of methods. And it should provide two introspection features: (3) an *is-instance-of* predicate that can be used for checking if an object is a direct or non-direct instance of a class, and (4) an instance-level property for retrieving the direct type of an object. In addition, it is desirable to have a third introspection feature: a *supertype* property of classes that can be used for retrieving the direct supertype of a class.

As we have explained in Part 1 [minimal-tutorial.html] of this tutorial, classes can be defined in two alternative ways: **constructor-based** and **factory-based**. Both approaches have their own way of implementing inheritance. In this part of our tutorial we only discuss subtyping and inheritance for constructor-based classes, while in our JavaScript Frontend Apps [complete-tutorial.html] book we also discuss subtyping and inheritance for factory-based classes

For defining a category in a constructor-based class hierarchy, we use a 3-part code pattern recommended by Mozilla in their JavaScript Guide [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Details_of_the_Object_Model]. We use the example shown in 12.4 below. We first define (the constructor function of) a base class `Person`:

```
function Person( first, last) {
  this.firstName = first;
  this.lastName = last;
}
```

Figure 12.4. `Student` is a category of `Person`



The category `Student` is defined in the following way:

```
function Student( first, last, studNo) {
  Person.call( this, first, last);  // invoke superclass constructor
  this.studNo = studNo;  // define and assign additional properties
}
```

By invoking the supertype constructor with `Person.call( this, ...)` for the new object created (referenced by `this`) as an instance of the category `Student`, we achieve that the property slots created in the supertype constructor (`firstName` and `lastName`) are also created for the category instance, along the entire chain of supertypes within a given class hierarchy. In this way we set up a ***property inheritance*** mechanism that makes sure that the *own* properties defined for an object on creation include the *own* properties defined by the superclass contructors.

In addition to this property inheritance mechanism, we set up a mechanism for ***method inheritance*** via the constructor's `prototype` property. We assign a new object created from the supertype's `prototype` object to the `prototype` property of the category constructor and reset the prototype's constructor property:

```
Student.prototype = Object.create( Person.prototype);  // inherit from Person
Student.prototype.constructor = Student;  // reset the category's constructor
```

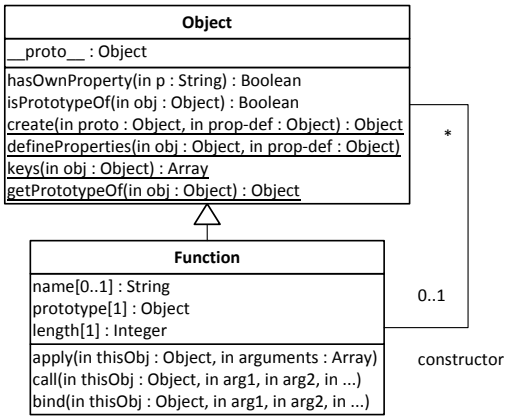
By assigning an empty supertype instance to the prototype property of the category constructor, we achieve that the methods defined in, and inherited by, the supertype are also available for objects instantiating the category. This mechanism of chaining the prototypes takes care of method inheritance. Notice that setting `Student.prototype` to `Object.create( Person.prototype)`, which creates a new object with its `prototype` set to `Person.prototype` and without any own properties, is preferable over setting it to `new Person()`, which was the way to achieve the same in the time before ECMAScript 5.

Finally, we define the additional methods of the subclass as method slots of its `prototype` object:

```
Student.prototype.setMatrNo = function (studNo) {
  this.studNo = studNo;
}
```

When an object is constructed, the built-in reference property `__proto__` (with a double underscore prefix and suffix) is set to the value of the constructor's `prototype` property. For instance, after creating a new object with `f = new Foo()`, it holds that `f.__proto__` is equal to `Foo.prototype`. Consequently, changes to the slots of `Foo.prototype` affect all objects that were created with `new Foo()`. While every object has a `__proto__` reference property (except `Object`), every function has a `prototype` reference property.

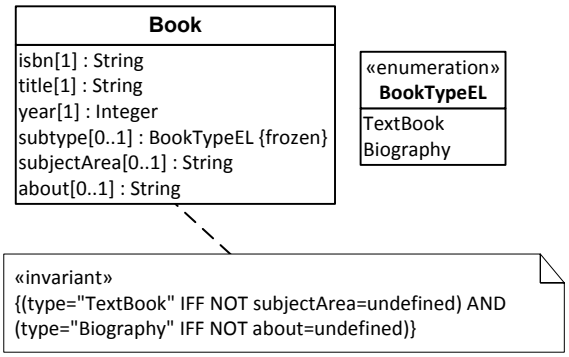
Figure 12.5. The built-in JavaScript classes `Object` and `Function`.



Case Study 1: Eliminating a Class Hierarchy

Simple class hierarchies can be eliminated by applying the *Class Hierarchy Merge* design pattern. The starting point for our case study is the simple class hierarchy shown in the information design model of Figure 12.2 above, representing a disjoint (but incomplete) segmentation of `Book` into `TextBook` and `Biography`. This model is first simplified by applying the *Class Hierarchy Merge* design pattern, resultng in the model shown in Figure 12.6.

Figure 12.6. The simplified information design model obtained by applying the Class Hierarchy Merge design pattern



We can now derive a *JavaScript data model* from this design model.

Make the JavaScript data model

We make the *JavaScript data model* in 3 steps:

1. Turn the design model's **enumeration type**, which contains an enumeration literal for each segment subclass, into a corresponding JavaScript map where the enumeration literals are (by convention uppercase) keys associated with an integer value that enumerates the literal. For instance, for the first enumeration literal "TextBook" we get the key-value pair TEXTBOOK=1.

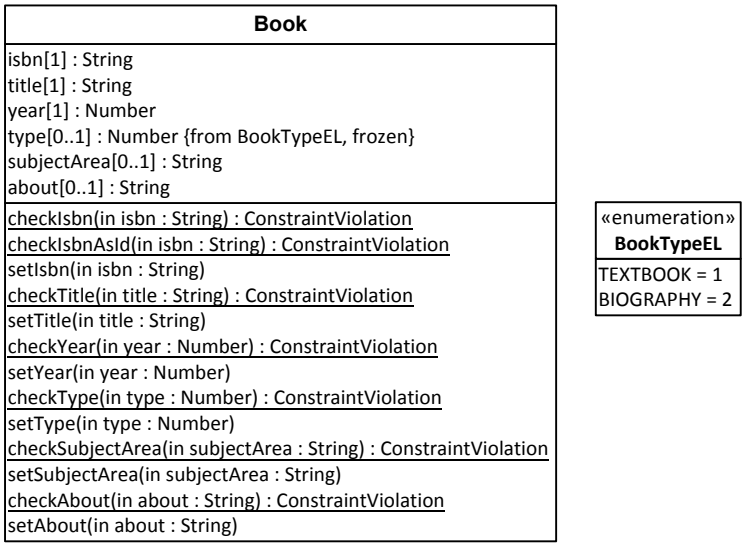
2. Turn the plaform-independent datatypes (defined as the ranges of attributes) into JavaScript datatypes. This includes the case of enumeration-valued attributes, such as `category`, which are turned into numeric attributes restricted to the enumeration integers of the underlying enumeration type.

3. Add property **checks** and **setters**, as described in Part 2 of this tutorial. The `checkSubtype` and `setCategory` methods, as well as the checks and setters of the segment properties need special

consideration according to their implied sematnics. In particular, a segment property's check and setter methods must ensure that the property can only be assigned if the `category` attribute has a value representing the corresponding segment. We explain this implied validation semantics in more detail below when we discuss how the JavaScript data model is encoded.

This leads to the JavaScript data model shown in Figure 12.7, where the class-level ('static') methods are underlined:

Figure 12.7. The JavaScript data model



New issues

Compared to the validation app [ValidationApp/index.html] discussed in Part 2 of this tutorial, we have to deal with a number of new issues:

1. In the *model code* we have to take care of

   a. Adding the constraint violation class *FrozenValueConstraintViolation* to `errorTypes.js`.

   b. Encoding the enumeration type to be used as the range of the `category` attribute (`BookCategoryEL` in our example).

   c. Encoding the `checkSubtype` and `setCategory` methods for the `category` attribute. In our example this attribute is *optional*, due to the fact that the book types segmentation is *incomplete*. If the segmentation, to which the *Class Hierarchy Merge* pattern is applied, is complete, then the `category` attribute is *mandatory*.

   d. Encoding the *checks* and *setters* for all segment properties such that the check methods take the category as a second parameter for being able to test if the segment property concerned applies to a given instance.

   e. Refining the serialization method `toString()` by addding a `category` case distinction (`switch`) statement for serializing only the segment properties that apply to the given category.

   f. Implementing the *Frozen Value Constraint* for the `category` attribute in `Book.update` by updating the `category` of a book only if it has not yet been defined. This means it cannot be updated anymore as soon as it has been defined.

2. In the *UI code* we have to take care of

a. Adding a "Special type" column to the display table of the "List all books" use case in `books.html`. A book without a special category will have an empty table cell, while for all other books their category will be shown in this cell, along with other segment-specific attribute values. This requires a corresponding switch statement in `pl.view.books.list.setupUserInterface` in the `books.js` view code file.

b. Adding a "Special type" select control, and corresponding form fields for all segment properties, in the forms of the "Create book" and "Update book" use cases in `books.html`. Segment property form fields are only displayed, and their validation event handlers set, when a corresponding book category has been selected. Such an approach of rendering specific form fields only on certain conditions is sometimes called "dynamic forms".

Encode the model classes of the JavaScript data model

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

Summary

1. Encode the enumeration type (to be used as the range of the `category` attribute) as a special JavaScript object mapping upper-case keys, representing enumeration literals, to corresponding enumeration integers.

2. Encode the model class (obtained by applying the *Class Hierarchy Merge* pattern) in the form of a JavaScript constructor function with class-level check methods attached to it, and with instance-level setter methods attached to its `prototype`.

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

Encode the enumeration type `BookCategoryEL`

The enumeration type `BookCategoryEL` is encoded as a special JavaScript object, mapping upper-case keys representing enumeration literals to enumeration integers, at the beginning of the `Book.js` model class file in the following way:

```
BookCategoryEL = Object.defineProperties( {}, {
    TEXTBOOK: { value: 1, writable: false},
    BIOGRAPHY: { value: 2, writable: false},
    MAX: { value: 2, writable: false},
    names: {value:["Textbook","Biography"], writable: false}
});
```

Notice that the names of the enumeration literals are stored in the `names` property of `BookCategoryEL` such that they can be easily retrieved, for instance, for showing the category of a book on a form.

Encode the model class `Book`

We encode the model class `Book` in the form of a constructor function where the `category` attribute as well as the segment attributes `subjectArea` and `about` are optional:

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

```
  this.year = 0;          // Number (PositiveInteger)
/* optional properties
  this.category            // Number {from BookCategoryEL}
  this.subjectArea        // String
  this.about              // String
*/
  if (arguments.length > 0) {
    this.setIsbn( slots.isbn);
    this.setTitle( slots.title);
    this.setYear( slots.year);
    if (slots.category) this.setCategory( slots.category);
    if (slots.subjectArea) this.setSubjectArea( slots.subjectArea);
    if (slots.about) this.setAbout( slots.about);
  }
}
```

We encode the `checkSubtype` and `setCategory` methods for the `category` attribute in the following way:

```
Book.checkSubtype = function (t) {
  if (!t) {
    return new NoConstraintViolation();
  } else {
    if (!isInteger( t) || t < 1 || t > BookCategoryEL.MAX) {
      return new RangeConstraintViolation(
          "The value of category must represent a book type!");
    } else {
      return new NoConstraintViolation();
    }
  }
};

Book.prototype.setCategory = function (t) {
  var constraintViolation = null;
  if (this.category) {  // already set/assigned
    constraintViolation = new FrozenValueConstraintViolation(
        "The category cannot be changed!");
  } else {
    t = parseInt( t);
    constraintViolation = Book.checkSubtype( t);
  }
  if (constraintViolation instanceof NoConstraintViolation) {
    this.category = t;
  } else {
    throw constraintViolation;
  }
};
```

While the setters for segment properties follow the standard pattern, their checks have to make sure that the attribute applies to the category of the instance being checked. This is achieved by checking a combination of a property value and a category, as in the following example:

```
Book.checkSubjectArea = function (sa,t) {
  if (t === undefined) t = BookCategoryEL.TEXTBOOK;
  if (t === BookCategoryEL.TEXTBOOK && !sa) {
    return new MandatoryValueConstraintViolation(
        "A subject area must be provided for a textbook!");
  } else if (t !== BookCategoryEL.TEXTBOOK && sa) {
    return new OtherConstraintViolation("A subject area must not
```

```
        be provided if the book is not a textbook!");
  } else if (sa && (typeof(sa) !== "string" || sa.trim() === "")) {
    return new RangeConstraintViolation(
        "The subject area must be a non-empty string!");
  } else {
    return new NoConstraintViolation();
  }
};
```

In the serialization function `toString`, we serialize the category attribute and the segment properties in a `switch` statement:

```
Book.prototype.toString = function () {
  var bookStr = "Book{ ISBN:"+ this.isbn +", title:"+ this.title +
      ", year:"+ this.year;
  switch (this.category) {
  case BookCategoryEL.TEXTBOOK:
    bookStr += ", textbook subject area:"+ this.subjectArea;
    break;
  case BookCategoryEL.BIOGRAPHY:
    bookStr += ", biography about: "+ this.about;
    break;
  }
  return bookStr +" }";
};
```

In the update method of a model class, we only set a property if it is to be updated (that is, if there is a corresponding slot in the `slots` parameter) and if the new value is different from the old value. In the special case of a `category` attribute with a *Frozen Value Constraint*, we need to make sure that it can only be updated, along with an accompanying set of segement properties, if it has not yet been assigned. Thus, in the `Book.update` method, we perform the special test if `book.category === undefined` for handling the special case of an initial assignment, while we handle updates of the optional segment properties `subjectArea` and `about` in a more standard way:

```
Book.update = function (slots) {
  var book = Book.instances[slots.isbn],
      updatedProperties = [],
      ...;
  try {
    ...
    if ("category" in slots && book.category === undefined) {
      book.setCategory( slots.category);
      updatedProperties.push("category");
      switch (slots.category) {
      case BookCategoryEL.TEXTBOOK:
        book.setSubjectArea( slots.subjectArea);
        updatedProperties.push("subjectArea");
        break;
      case BookCategoryEL.BIOGRAPHY:
        book.setBiography( slots.biography);
        updatedProperties.push("biography");
        break;
      }
    }
    if ("subjectArea" in slots && "subjectArea" in book &&
          book.subjectArea !== slots.subjectArea) {
      book.setSubjectArea( slots.subjectArea);
      updatedProperties.push("subjectArea");
    }
```

```
    if ("about" in slots && "about" in book &&
            book.about !== slots.about) {
      book.setAbout( slots.about);
      updatedProperties.push("about");
    }
  } catch (e) {
    ...
  }
  ...
};
```

Write the View and Controller Code

The user interface (UI) consists of a start page that allows navigating to the data management pages (in our example, to `books.html`). Such a data management page contains 5 sections: *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).

Summary

We have to take care of handling the `category` attribute and the `subjectArea` and `about` segment properties both in the "List all books" use case as well as in the "Create book" and "Update book" use cases by

1. Adding a segment information column ("Special type") to the display table of the "List all books" use case in `books.html`.

2. Adding a "Special type" select control, and corresponding form fields for all segment properties, in the forms of the "Create book" and "Update book" use cases in `books.html`. Segment property form fields are only displayed, and their validation event handlers set, when a corresponding book category has been selected. Such an approach of rendering specific form fields only on certain conditions is sometimes called "dynamic forms".

Adding a segment information column in "List all books"

We add a "Special type" column to the display table of the "List all books" use case in `books.html`:

```
<table id="books">
  <thead><tr><th>ISBN</th><th>Title</th><th>Year</th><th>Special type</th></tr>
  <tbody></tbody>
</table>
```

A book without a special category will have an empty table cell in this column, while for all other books their category will be shown in this column, along with other segment-specific information. This requires a corresponding switch statement in `pl.view.books.list.setupUserInterface` in the `view/books.js` file:

```
if (book.category) {
  switch (book.category) {
  case BookCategoryEL.TEXTBOOK:
    row.insertCell(-1).textContent = book.subjectArea + " textbook";
    break;
  case BookCategoryEL.BIOGRAPHY:
    row.insertCell(-1).textContent = "Biography about "+ book.about;
    break;
  }
}
```

Adding a "Special type" select control in "Create book" and "Update book"

In both use cases, we need to allow selecting a special category of book ('textbook' or 'biography') with the help of a select control, as shown in the following HTML fragment:

```
<p class="pure-control-group">
  <label for="creBookType">Special type: </label>
  <select id="creBookType" name="category"></select>
</p>
<p class="pure-control-group Textbook">
  <label for="creSubjectArea">Subject area: </label><input id="creSubjectArea" n
</p>
<p class="pure-control-group Biography">
  <label for="creAbout">About: </label><input id="creAbout" name="about" />
</p>
```

Notice that we have added "Textbook" and "Biography" as additional class values to the HTML `class` attributes of the `p` elements containing the corresponding form controls. This allows what is called *dynamic forms*: easy rendering and un-rendering of "Textbook" and "Biography" form controls, depending on the value of the `category` attribute.

In the `handleTypeSelectChangeEvent` handler, segment property form fields are only displayed, with `pl.view.app.displaySegmentFields`, and their validation event handlers set, when a corresponding book category has been selected:

```
pl.view.books.handleTypeSelectChangeEvent = function (e) {
  var formEl = e.currentTarget.form,
      typeIndexStr = formEl.category.value,  // the array index of BookCategoryE
      category=0;
  if (typeIndexStr) {
    category = parseInt( typeIndexStr) + 1;
    switch (category) {
    case BookCategoryEL.TEXTBOOK:
      formEl.subjectArea.addEventListener("input", function () {
        formEl.subjectArea.setCustomValidity(
            Book.checkSubjectArea( formEl.subjectArea.value).message);
      });
      break;
    case BookCategoryEL.BIOGRAPHY:
      formEl.about.addEventListener("input", function () {
        formEl.about.setCustomValidity(
            Book.checkAbout( formEl.about.value).message);
      });
      break;
    }
    pl.view.app.displaySegmentFields( formEl, BookCategoryEL.names, category);
  } else {
    pl.view.app.undisplayAllSegmentFields( formEl, BookCategoryEL.names);
  }
};
```

Case Study 2: Implementing a Class Hierarchy

Whenever a class hierarchy is more complex, we cannot simply eliminate it, but have to implement it (1) in the app's model code, (2) in its user interface and (3) in the underlying database. The starting

point for our case study is the design model shown in Figure 12.3 above. In the following sections, we derive a *JavaScript data model* and a *JSON table model* from the design model. The JSON table model is used as a design for the object-to-JSON mapping that we need for storing the objects of our app in Local Storage.

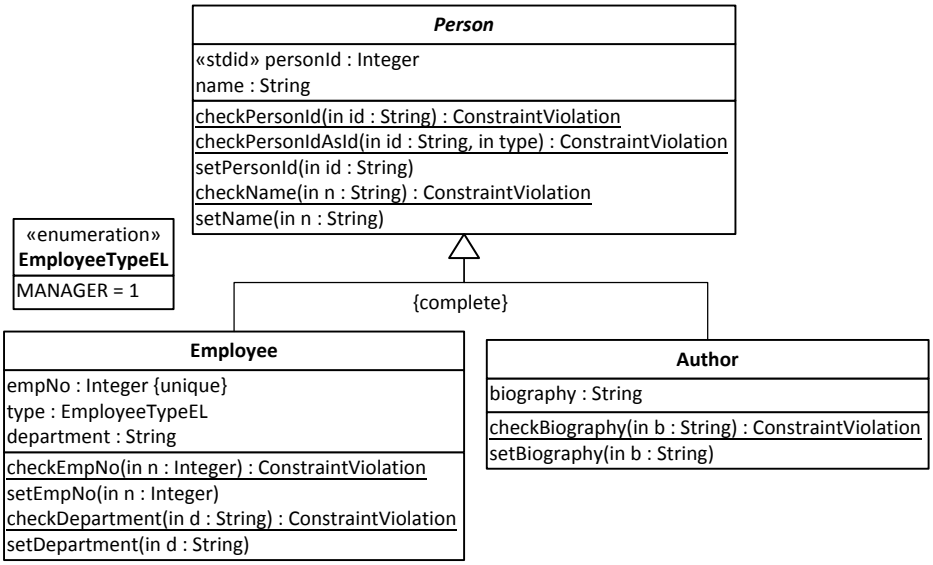
Make the JavaScript data model

We design the *model classes* of our example app with the help of a *JavaScript data model* that we derive from the *design model* by essentially leaving the generalizatiion arrows as they are and just adding *checks* and *setters* to each class, as described in Part 2 of this tutorial. However, in the case of our example app, it is natural to apply the *Class Hierarchy Merge* design pattern (discussed in Section) to the segmentation of `Employee` for simplifying the data model by eliminating the `Manager` subclass. This leads to the model shown in Figure 12.8 below. Notice that we have also made two technical design decisions:

1. We have declared the segmentation of `Person` into `Employee` and `Author` to be **complete**, that is, any person is an employee or an author (or both).

2. We have turned `Person` into an **abstract class** (indicated by its name written in italics in the class rectangle), which means that it cannot have direct instances, but only indirect ones via its subclasses `Employee` and `Author`, implying that we do not need to maintain its extension (in a map like `Person.instances`), as we do for all other non-abstract classes. This technical design decision is compatible with the fact that any `Person` is an `Employee` or an `Author` (or both), and consequently there is no need for any instance to instantiate `Person` diretly.

Figure 12.8. The JavaScript data model of the Person class hierarchy



Make the JSON table model

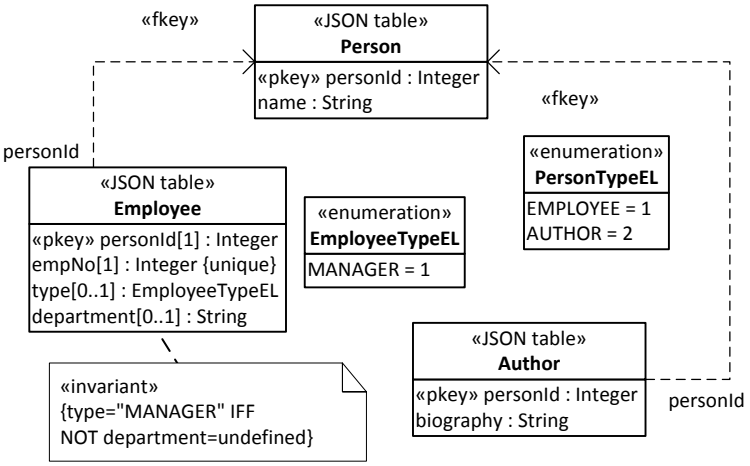
Since we use *Local Storage* as the persistent storage technology for our example app, we have to deal with simple key-value storage. For each model class with a signular name `Entity`, we use its pluralized name `entities` as a key such that its associated value is the JSON table serialization of the main memory object collection `Entity.instances`.

We design a set of suitable JSON tables and the structure of their records in the form of a *JSON table model* that we derive from the design model by following certain rules. We basically have two choices how to organize our JSON data store and how to derive a corresponding JSON table model: either according to the *Single Table Inheritance* approach, where a segmentation or an entire class hierarchy

is represented with a single table, or according to the *Joined Tables Inheritance* approach, where we have a separate table for each model class of the class hierarchy. Both approaches, which are discussed in section , can be combined for the same design model.

In our example it seems natural to apply the Single Table Inheritance approach to the incomplete segmentation of `Employee` with just one segment subclass `Manager`, while we apply the Joined Tables Inheritance approach to the complete segmentation of `Person` into `Employee` and `Author`. This results in the model shown in Figure 12.9 below.

Figure 12.9. The JSON table model of the Person class hierarchy



Notice that we have replaced the «stdid» stereotype with «pkey» for indicating that the attributes concerned act as primary keys in combination with foreign keys expressed by the dashed dependency arrows stereotyped with «fkey» and annotated with the foreign key attribute (`personId`) at their source end. An example of an admissible population for this table model is the following:

Person	**Employee**	**Author**
{personId: 1001, name:"Gerd Wagner"}	{personId: 1001, empNo: 21035}	{personId: 1001, biography:"Born in ..."}
{personId: 1002, name:"Tom Boss"}	{personId: 1002, empNo:23107, category: EmployeeTypeEL.MANAGER, department:"Faculty 1"}	{personId: 1077, biography:"Kant was ..."}
{personId: 1077, name:"Immanuel Kant"}		

Notice the mismatch between the JavaScript data model shown in Figure 12.7 above, which is the basis for the model classes `Person`, `Employee` and `Author` as well as for the main memory database consisting of `Employee.instances` and `Author.instances` on one hand side, and the JSON table model, shown in Figure 12.9 above on the other hand side. While we do not have any `Person` records in the main memory database, we do have them in the persistent datastore based on the JSON table model. This mismatch results from the complete structure of JavaScript subclass instances, which include all property slots, as opposed to the fragmented structure of database tables based on the *Joined Tables Inheritance* approach.

New issues

Compared to the model of our first case study, shown in Figure 12.7 above, we have to deal with a number of new issues in the *model code*:

1. Defining the category relationships between `Employee` and `Person`, as well as between `Author` and `Person`, using the JavaScript code pattern for constructor-based inheritance discussed in Section .

2. When loading the instances of a category from persistent storage (as in `Employee.loadAll` and `Author.loadAll`), their slots for inherited supertype properties, except for the standard identifier attribute, have to be reconstructed from corresponding rows of the supertable (`persons`).

3. When saving the instances of `Employee` and `Author` as records of the JSON tables `employees` and `authors` to persistent storage (as in `pl.view.employees.manage.exit` and `pl.view.authors.manage.exit`), we also need to save the records of the supertable `persons` by extracting their data from corresponding `Employee` or `Author` instances.

Encode the model classes of the JavaScript data model

The JavaScript data model shown in Figure 12.8 above can be directly encoded for getting the code of the model classes `Person`, `Employee` and `Author` as well as for the enmueration type `EmployeeTypeEL`.

Define the category relationships

We define the category relationships between `Employee` and `Person`, as well as between `Author` and `Person`, using the JavaScript code pattern for constructor-based inheritance discussed in Section . For instance, in `model/Employee.js` we define:

```
function Employee( slots) {
  // set the default values for the parameter-free default constructor
  Person.call( this);  // invoke the default constructor of the supertype
  this.empNo = 0;      // Number (PositiveInteger)
  // constructor invocation with arguments
  if (arguments.length > 0) {
    Person.call( this, slots);  // invoke the constructor of the supertype
    this.setEmpNo( slots.empNo);
    if (slots.category) this.setCategory( slots.category);  // optional
    if (slots.department) this.setDepartment( slots.department);  // optional
  }
};
Employee.prototype = Object.create( Person.prototype);  // inherit from Person
Employee.prototype.constructor = Employee;  // adjust the constructor property
```

Reconstruct inherited supertype properties when loading the instances of a category

When loading the instances of a category from persistent storage (as in `Employee.loadAll` and `Author.loadAll`), their slots for inherited supertype properties, except for the standard identifier attribute, have to be reconstructed from corresponding rows of the supertable (`persons`). For instance, in `model/Employee.js` we define:

```
Employee.loadAll = function () {
  var key="", keys=[], persons={}, employees={}, employeeRow={}, i=0;
  if (!localStorage["employees"]) {
    localStorage.setItem("employees", JSON.stringify({}));
  }
  try {
    persons = JSON.parse( localStorage["persons"]);
    employees = JSON.parse( localStorage["employees"]);
```

```
  } catch (e) {
    console.log("Error when reading from Local Storage\n" + e);
  }
  keys = Object.keys( employees);
  console.log( keys.length +" employees loaded.");
  for (i=0; i < keys.length; i++) {
    key = keys[i];
    employeeRow = employees[key];
    // complete record by adding slots ("name") from supertable
    employeeRow.name = persons[key].name;
    Employee.instances[key] = Employee.convertRow2Obj( employeeRow);
  }
};
```

Reconstruct and save the supertable `Person` when saving the main memory data

When saving the instances of `Employee` and `Author` as records of the JSON tables `employees` and `authors` to persistent storage (as in `pl.view.employees.manage.exit` and `pl.view.authors.manage.exit`), we also need to save the records of the supertable `persons` by extracting their data from corresponding `Employee` or `Author` instances. For this purpose, we define in `model/Person.js`:

```
Person.saveAll = function () {
  var key="", keys=[], persons={}, i=0, n=0;
  keys = Object.keys( Employee.instances);
  for (i=0; i < keys.length; i++) {
    key = keys[i];
    emp = Employee.instances[key];
    persons[key] = {personId: emp.personId, name:emp.name};
  }
  keys = Object.keys( Author.instances);
  for (i=0; i < keys.length; i++) {
    key = keys[i];
    if (!persons[key]) {
      author = Author.instances[key];
      persons[key] = {personId: author.personId, name: author.name};
    }
  }
  try {
    localStorage["persons"] = JSON.stringify( persons);
    n = Object.keys( persons).length;
    console.log( n +" persons saved.");
  } catch (e) {
    alert("Error when writing to Local Storage\n" + e);
  }
};
```

Part V. Using the Model-Based Development Framework mODELcLASS

In this last part we show how to avoid repetitive code structures ("boilerplate code") by using the model-based development framework mODELcLASS, which has been developed as a result of writing this book.

Table of Contents

Chapter 13. The Model-Based Development Framework mODELcLASSjs

In the Model-View-Controller (MVC) paradigm, the UI is called 'view', and the term 'controller' denotes the glue code needed for integrating the UI code with the model classes, or, in MVC jargon, for integrating the 'view' with the 'model'. Using a model-based development approach, **the model classes of an app are obtained by encoding the app's data model**, which is typically expressed in the form of a UML class diagram. Since it is the task of the model layer to define and validate constraints and to manage data storage, we need reusable model code taking care of this in a generic manner for avoiding per-class and per-property boilerplate code for constraint validation, and per-class boilerplate code for data storage management. This is where mODELcLASSjs comes in handy. It provides

1. a generic **check** method for validating property constraints, and

2. the generic storage management methods **add**, **update** and **destroy** for creating new (persistent) objects/rows, for updating existing objects/rows and for deleting them.

For using the functionality of mODELcLASSjs in your app, you have to include its code, either by downloading mODELcLASS.js [https://bitbucket.org/gwagner57/entitytypejs/downloads] to the lib folder and use a local script loading element like the following:

```
<script src="lib/mODELcLASS.js"></script>
```

or with the help of a remote script loading element like the following:

```
<script src="http://web-engineering.info/JsFrontendApp/mODELcLASS.js"></script>
```

Then you can create your app's model classes (with property and method declarations) as instances of the meta-class mODELcLASS:

```
Book = new mODELcLASS({
  typeName: "Book",
  properties: {
    isbn: {range:"NonEmptyString", isStandardId: true, label:"ISBN", pattern:/\
        patternMessage:'The ISBN must be a 10-digit string or a 9-digit string
    title: {range:"NonEmptyString", min: 2, max: 50, label:"Title"},
    year: {range:"Integer", min: 1459, max: util.nextYear(), label:"Year"}
  },
  methods: {
    ...
  }
});
```

Notice that the declaration of a property includes the constraints that apply to it. For instance, the declaration of the property `isbn` includes a pattern constraint reqiuring that the ISBN must be a 10-digit string or a 9-digit string followed by "X".

After defining a model class, you can create new 'model objects' instantiating it by invoking the `create` method provided by mODELcLASS:

```
var book1 = Book.create({isbn:"006251587X", title:"Weaving the Web", year: 2000
```

You can then apply the following properties and methods, all pre-defined by mODELcLASS

1. the property `type` for retrieving the object's direct type,

2. the method `toString()` for serializing an object,

3. the method `set(` prop, val`)` for setting an object property after checking all property constraints,

4. the method `isInstanceOf(` Class`)` for testing if an object is an instance of a class.

The use of these mODELcLASS features is illustrated by the following examples:

```
console.log( book1.type.typeName);  // "Book"
console.log( book1.toString());  // "Book{ isbn:"006251587X", ...}"
book1.set("year", 1001);  // "IntervalConstraintViolation: Year must not be sma
book1.set("year", 2001);  // change the year to 2001
console.log( book1.isInstanceOf( Book));  // true
```

You can also invoke the generic check method provided by mODELcLASS in the user interface code. For instance, for responsive validation with the HTML5 constraint validation API, you may do the following:

```
var formEl = document.forms["Book"];
formEl.isbn.addEventListener("input", function () {
  formEl.isbn.setCustomValidity(
      Book.check( "isbn", formEl.isbn.value).message);
});
```

Here we define an event handler for `input` events on the ISBN input field. It invokes the `setCustomValidity` method of the HTML5 constraint validation API for setting a validation error message that results from invoking `Book.check` for validating the constraints defined for the `isbn` property for the user input value from the form field `formEl.isbn`. The `check` method returns a constraint violation object with a message property. If no constraint is violated, the message is an empty string, so nothing happens. Notice that you don't have to write the code of the `check` method, as it is provided by mODELcLASSjs.

You can also use the methods provided by any mODELcLASS for managing data storage, like, for instance,

```
// populate Book.instances from database (by default, Local Storage)
Book.loadAll();
// create a new 'model object' and store it in the database
Book.add( {isbn:"0465030793", title:"I Am A Strange Loop", year: 2008});
```

By default, data is stored locally with the help of JavaScript's *Local Storage* API in the form of 'stringified' JSON tables. However, mODELcLASSjs will also provide the option to store data locally with *IndexedDB*, instead of *Local Storage*, or to store data remotely with the help of XHR messaging, for more demanding data management apps.

Model-Based Development

We must not confuse the term 'model' as used in the MVC paradigm, and adopted by many web development frameworks, and the term 'model' as used in UML and other modeling languages. While the former refers to the model classes of an app, the latter refers to the concept of a model either as a simplified description of some part of the real world, or as a design blueprint for construction.

In model-based engineering, models are the basis for designing and implementing a system, no matter if the system to be built is a software system or another kind of complex system such as a manufacturing machine, a car, or an organisation.

In model-based software development, we distinguish between three kinds of models:

1. solution-independent **domain models** describing a specific part of the real-world and resulting from requirements and domain engineering in the system analysis, or inception, phase of a development project;

2. platform-independent **design models** specifying a logical system design resulting from the design activities in the elaboration phase;

3. platform-specific **implementation models** (including **data models**) as the result of technical system design in the implementation phase.

Domain models are the basis for designing a software system by making a platform-independent *design model*, which is in turn the basis for implementing a system by making an *implementation model* and encoding it in the language of the chosen platform. Concerning information modeling, we first make a domain information model, then we derive an information design model from it, and finally map the information design model to a **data model** for the chosen platform. With an object-oriented programming (OOP) approach, we encode this data model in the form of **model classes**, which are the basis for designing and implementing a data management user interface (UI).

mODELcLASSjs facilitates model-based app development by allowing a direct encoding of an information design model, including subtype/inheritance relationships.

The Philosophy and Features of mODELcLASSjs

The concept of a **class** is fundamental in *object-oriented* programming. Objects *instantiate* (or *are classified by*) a class. A class defines the properties and methods for the objetcs that instantiate it.

There is no explicit class concept in JavaScript. However, classes can be defined in two ways:

1. In the form of a **constructor** function that allows to create new instances of the class with the help of the `new` operator. This is the classical approach recommended in the Mozilla JavaScript documents.

2. In the form of a **factory** object that uses the predefined `Object.create` method for creating new instances of the class.

Since we normally need to define class hierarchies, and not just single classes, these two alternative approaches cannot be mixed within a class hierarchy, and we have to make a choice whenever we build an app. With mODELcLASSjs, you choose the second approach with the following benefits:

1. Properties are declared (with a property label, a range and many other constraints)

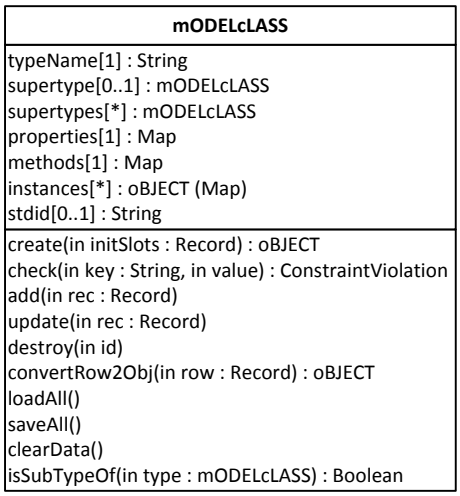
2. Supports object pools

3. Supports multiple inheritance and multiple classification

These benefits come with a price: objects are created with lower performance, mainly due to the fact that `Object.create` is slower than `new`. On the other hand, for apps with lots of object creation and destruction, such as games and simulations, mODELcLASSjs provides object pools for avoiding performance problems due to garbage collection.

The properties and methods of the meta-class mODELcLASS are listed in the following class diagram:

Figure 13.1. The meta-class mODELcLASS



Notice that in the class diagram, we use the type `oBJECT` for values representing those special JavaScript objects that instantiate a model class created with mODELcLASS. These objects have a pre-defined property `type` referencing the model class they instantiate as their direct type. The values of the pre-defined property `instances` are maps of `oBJECT`s representing the extension (or population) of a model class.

The Check Method

Any mODELcLASS has a class-level `check` method for validating all kinds of property constratints. Since this method can validate all properties of a model class, it takes as its first parameter the name of the property, and as its second parameter the value to be validated:

```
mODELcLASS.prototype.check = function (prop, val) {
  var propDeclParams = this.properties[prop],
      range = propDeclParams.range,
      min = propDeclParams.min,
      max = propDeclParams.max,
      minCard = propDeclParams.minCard,
      maxCard = propDeclParams.maxCard,
      pattern = propDeclParams.pattern,
      msg = propDeclParams.patternMessage,
      label = propDeclParams.label || prop;
```

Notice that the definition of a model class comes with a set of poperty declarations in `properties`. An example of such a poperty declaration is the declaration of the attribute `title`:

```
title: {range:"NonEmptyString", min: 2, max: 50, label:"Title"},
```

In this example we have the property declaration parameters `range`, `min`, `max` and `label`. In lines 2-10 above, all these property declaration parameters are copied to local variables for having convenient shortcuts.

In the `check` method, the first check is concerned with **mandatory value constraints**:

```
  if (!propDeclParams.optional && val === undefined) {
      return new MandatoryValueConstraintViolation("A value for "+
          label +" is required!");
  }
```

The next check is concerned with **range constraints**:

```
switch (range) {
case "String":
  if (typeof( val) !== "string") {
    return new RangeConstraintViolation("The "+ label +
        " must be a string!");
  }
  break;
case "NonEmptyString":
  if (typeof(val) !== "string" || val.trim() === "") {
    return new RangeConstraintViolation("The "+ label +
        " must be a non-empty string!");
  }
  break;
...  // other cases
case "Boolean":
  if (typeof( val) !== "boolean") {
    return new RangeConstraintViolation("The value of "+ label +
        " must be either 'true' or 'false'!");
  }
  break;
}
```

Then there are several range-specific checks concerning (1) **string length constraints** and **pattern constraints**:

```
if (range === "String" || range === "NonEmptyString") {
  if (min !== undefined && val.length < min) {
    return new StringLengthConstraintViolation("The length of "+
        label + " must be greater than "+ min);
  } else if (max !== undefined && val.length > max) {
    return new StringLengthConstraintViolation("The length of "+
        label + " must be smaller than "+ max);
  } else if (pattern !== undefined && !pattern.test( val)) {
      return new PatternConstraintViolation( msg || val +
          "does not comply with the pattern defined for "+ label);
  }
}
```

and (2) **interval constraints**:

```
if (range === "Integer" || range === "NonNegativeInteger" ||
    range === "PositiveInteger") {
  if (min !== undefined && val < min) {
    return new IntervalConstraintViolation( label +
        " must be greater than "+ min);
  } else if (max !== undefined && val > max) {
    return new IntervalConstraintViolation( label +
        " must be smaller than "+ max);
  }
}
```

Then the next check is concerned with **cardinality constraints**, which may apply to list-valued or map-valued properties.

```
if (minCard !== undefined &&
    (Array.isArray(val) && val.length < minCard ||
     typeof(val)==="object" && Object.keys(val).length < minCard)) {
```

```
    return new CardinalityConstraintViolation(
        "A set of at least "+ minCard +" values is required for "+ label);
  }
  if (maxCard !== undefined &&
      (Array.isArray(val) && val.length > maxCard ||
       typeof(val)==="object" && Object.keys(val).length > maxCard)) {
    return new CardinalityConstraintViolation("A value set for "+ label +
        " must not have more than "+ maxCard +" members!");
  }
```

Then the next check is concerned with **uniqueness constraints**, which can only be checked by inspecting the entire population of the model class. Assuming that this population has been loaded into the main memory collection *modelclass*`.instances`, the following code is used:

```
  if (propDeclParams.unique && this.instances) {
    keys = Object.keys( this.instances);
    for (i=0; i < keys.length; i++) {
      if ( this.instances[keys[i]][prop] === val) {
        return new UniquenessConstraintViolation("There is already a "+
            this.typeName +" with a(n) "+ label +" value "+ val +"!");
      }
    }
  }
```

Finally, the *mandatory value constraints* and the *uniqueness constraints* implied by a **standard identifier declaration** are checked:

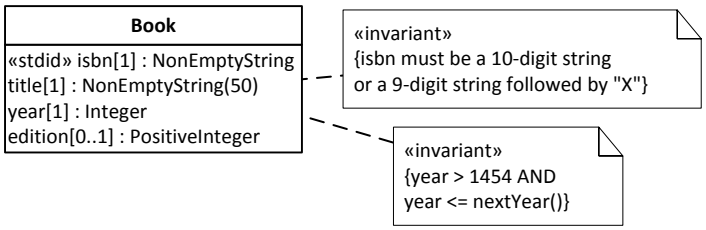
```
  if (propDeclParams.isStandardId) {
    if (val === undefined) {
      return new MandatoryValueConstraintViolation("A value for the " +
          "standard identifier attribute "+ label +" is required!");
    } else if (this.instances && this.instances[val]) {
      return new UniquenessConstraintViolation("There is already a "+
          this.typeName +" with a(n) "+ label +" value "+ val +"!");
    }
  }
```

Chapter 14. Constraint Validation with mODELcLASSjs

In this part of the tutorial, we show how to build a single-class app with constraint validation using the model-based development framework mODELcLASSjs for avoiding boilerplate model code. Compared to the app discussed in Part 2, we deal with the same issues: showing 1) how to **define constraints** in a model class, 2) how to **perform responsive validation** in the user interface based on the constraints defined in the model classes. The main difference when using mODELcLASSjs is that defining constraints becomes much simpler. The check methods used in Part 2 are no longer needed. Since constraints are defined in a purely declarative manner, their textual encoding corresponds directly to their expression in the information design model. This implies that we can directly encode the information design model without first creating a data model from it.

As in Part 2, the purpose of our app is to manage information about books. The information items and constraints are described in the information design model shown in Figure 14.1 below.

Figure 14.1. A platform-independent design model with the class `Book` and two invariants



Encoding the Design Model

We now show how to encode the ten integrity constraints defined by the design model shown in Figure 14.1 above.

1. For the first three of the four properties defined in the `Book` class, we have a *mandatory value constraint*, indicated by the multiplicity expression [1]. However, since properties are mandatory by default in mODELcLASSjs, we don't have to encode anything for them. Only for the property `edition`, we need to encode that it is optional with the key-value pair `optional: true`, as shown in the `edition` property declaration in the class definition below.

2. The `isbn` attribute is declared to be the *standard identifier* of `Book`. We encode this (and the implied uniqueness constraint) in the `isbn` property declaration with the key-value pair `isStandardId: true`, as shown in the class definition below.

3. The `isbn` attribute has a *pattern constraint* requiring its values to match the ISBN-10 format that admits only 10-digit strings or 9-digit strings followed by "X". We encode this with the key-value pair `pattern:/\b\d{9}(\d|X)\b/` and the special constraint violation message defined by `patternMessage:"The ISBN must be a 10-digit string or a 9-digit string followed by 'X'!"`.

4. The `title` attribute has an *string length constraint* with a maximum of 50 characters. This is encoded with `max: 50`.

5. The `year` attribute has an *interval constraint* with a minimum of 1459 and a maximum that is not fixed, but provided by the utility function `nextYear()`. We can encode this constraint with the key-value pairs `min: 1459` and `max: util.nextYear()`.

6. Finally, there are four *range constraints*, one for each property. We encode them with corresponding key-value pairs, like `range:"NonEmptyString"`.

This leads to the following definition of the model class `Book` :

```
Book = new mODELcLASS({
  typeName: "Book",
  properties: {
    isbn: {range:"NonEmptyString", isStandardId: true, label:"ISBN", pattern:/\
        patternMessage:"The ISBN must be a 10-digit string or a 9-digit string
    title: {range:"NonEmptyString", max: 50},
    year: {range:"Integer", min: 1459, max: util.nextYear()},
    edition: {range:"PositiveInteger", optional: true}
  }
});
```

For such a model class definition, mODELcLASSjs provides generic data management operations (`Book.add`, `Book.update`, `Book.destroy`, etc.) as well as property checks and setters (`Book.check` and `bookObject.set`).

Project Set-Up

The MVC folder structure of this project is the same as in the plain JavaScript validation app project discussed in Part 2 of the tutorial [http://web-engineering.info/JsFrontendApp/validation-tutorial.html] (or in Chapter 3 of the book). Also, the same library files are used.

The start page of the app first takes care of the page styling by loading the *Pure CSS* base file (from the Yahoo site) and our `main.css` file with the help of the two `link` elements (in lines 6 and 7), then it loads several JavaScript library files (in lines 8-12), including the mODELcLASS file, the app initialization script `initialize.js` from the `src/ctrl` folder and the model class `Book.js` from the `src/model` folder.

Figure 14.2. The mODELcLASS validation app's start page `index.html`.

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
  <head>
    <meta charset="UTF-8" />
    <title>JS Frontend Validation App Example</title>
    <link rel="stylesheet" type="text/css" href="http://yui.yahooapis.com/combo
    <link rel="stylesheet" type="text/css" href="css/main.css" />
    <script src="lib/browserShims.js"></script>
    <script src="lib/util.js"></script>
    <script src="lib/errorTypes.js"></script>
    <script src="http://web-engineering.info/JsFrontendApp/mODELcLASS.js"></scri
    <script src="src/ctrl/initialize.js"></script>
    <script src="src/model/Book.js"></script>
  </head>
  <body>
    <h1>Example: Public Library</h1>
    <h2>mODELcLASS Validation App</h2>
    <p>This app supports the following operations:</p>
    <menu>
      <li><a href="listBooks.html"><button type="button">List all books</button>
      <li><a href="createBook.html"><button type="button">Add a new book</butto
      <li><a href="updateBook.html"><button type="button">Update a book</button>
      <li><a href="deleteBook.html"><button type="button">Delete a book</button>
      <li><button type="button" onclick="Book.clearData()">Clear database</butt
      <li><button type="button" onclick="Book.createTestData()">Create test data
    </menu>
</body>
</html>
```

The View and Controller Layers

The user interface (UI) is the same as in the plain JavaScript validation app project discussed in Part 2 of the tutorial [http://web-engineering.info/JsFrontendApp/validation-tutorial.html] (or in Chapter 3 of the book). There is only one difference. For responsive constraint validation, where input event handlers are used to check constraints on user input, the generic check functon `Book.check` is used:

```
pl.view.createBook = {
  setupUserInterface: function () {
    var formEl = document.forms['Book'],
        submitButton = formEl.commit;
    submitButton.addEventListener("click",
        this.handleSubmitButtonClickEvent);
    formEl.isbn.addEventListener("input", function () {
        formEl.isbn.setCustomValidity(
            Book.check("isbn", formEl.isbn.value).message);
    });
    formEl.title.addEventListener("input", function () {
        formEl.title.setCustomValidity(
            Book.check("title", formEl.title.value).message);
    });
   ...
  },
};
```

While the validation on user input enhances the usability of the UI by providing immediate feedback to the user, validation on form submission is even more important for catching invalid data. Therefore,

the event handler `handleSubmitButtonClickEvent()` performs the property checks again, as shown in the following program listing:

```
handleSubmitButtonClickEvent: function () {
  var formEl = document.forms['Book'];
  var slots = { isbn: formEl.isbn.value,
        title: formEl.title.value,
        year: formEl.year.value,
        edition: formEl.edition.value
  };
  // set error messages in case of constraint violations
  formEl.isbn.setCustomValidity( Book.check( "isbn", slots.isbn).message);
  formEl.title.setCustomValidity( Book.check( "title", slots.title).message);
  formEl.year.setCustomValidity( Book.check( "year", slots.year).message);
  formEl.edition.setCustomValidity( Book.check( "edition", slots.edition).messa
  // save the input data only if all of the form fields are valid
  if (formEl.checkValidity()) {
    Book.add( slots);
  }
}
```

Run the App and Get the Code

You can run the mODELcLASS validation app [mODELcLASS-ValidationApp/index.html] from our server or download the code [mODELcLASS-ValidationApp.zip] as a ZIP archive file.

Evaluation

We evaluate the approach presented in this chapter of the tutorial according to the criteria defined in Part 2.

Table 14.1. Evaluation

Evaluation criteria	**½ ¼ ¾**
all important kinds of property constraints	1
model validation on assign and before save	1
generic validation error messages	1
responsive validation on input and on submit	1
two-fold validation	1
three-fold validation	n.a.
reporting database validation errors	n.a.
in total	**100 %**

Concluding Remarks

After eliminating the repetitive code structures (called *boilerplate code*) needed in the model layer for constraint validation and for the data storage management methods, there is still a lot of boilerplate code needed in the UI. In a follow-up article of our tutorial series, we will present an approach how to avoid this UI boilerplate code.

Chapter 15. Associations and Subtyping with mODELcLASS

Coming Soon!

Glossary

Hypertext Transfer Protocol	HTTP is a stateless request/response protocol using human-readable text messages for the communication between web clients and web servers. The main purpose of HTTP has been to allow fetching web documents identified by URLs from a web browser, and invoking the operations of a backend web application program from a HTML form executed by a web browser. More recently, HTTP is increasingly used for providing web APIs and web services.
Hypertext Markup Language	HTML allows marking up (or describing) the structure of a human-readable web document or web user interface. The XML-based version of HTML, which is called "XHTML5", provides a simpler and cleaner syntax compared to traditional HTML.
Extensible Markup Language	XML allows to mark up the structure of all kinds of documents, data files and messages in a machine-readable way. XML may also be human-readable, if the tag names used are self-explaining. XML is based on Unicode. SVG and MathML are based on XML, and there is an XML-based version of HTML. XML provides a syntax for expressing structured information in the form of an *XML document* with *elements* and their *attributes*. The specific elements and attributes used in an XML document can come from any vocabulary, such as public standards or user-defined XML formats.