JavaScript Front-End Web App Tutorial Part 3: Dealing with Enumerations

Learn how to build a front-end web application with enumeration attributes, using plain JavaScript

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

JavaScript Front-End Web App Tutorial Part 3: Dealing with Enumerations: Learn how to build a front-end web application with enumeration attributes, using plain JavaScript

by Gerd Wagner

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

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

Publication date 2015-11-12

Table of Contents

List of Figures

List of Tables

Foreword

This tutorial is Part 3 of our series of six tutorials [http://web-engineering.info/JsFrontendApp] about model-based development of front-end web applications with plain JavaScript. It shows how to build a web app where model classes have enumeration attributes.

The app supports the four standard data management operations (**C**reate/**R**ead/**U**pdate/**D**elete). The other parts of the tutorial are:

- Part 1 [minimal-tutorial.html]: Building a **minimal** app.
- Part 2 [validation-tutorial.html]: Handling **constraint validation**.
- Part 4 [unidirectional-association-tutorial.html]: Managing **unidirectional associations**, such as the associations between books and publishers, assigning a publisher to a book, and between books and authors, assigning authors to a book.
- Part 5 [bidirectional-association-tutorial.html]: Managing **bidirectional associations**, such as the associations between books and publishers and between books and authors, not only assigning authors and a publisher to a book, but also the other way around, assigning books to authors and to publishers.
- Part 6 [subtyping-tutorial.html]: Handling **subtype** (inheritance) relationships between object types.

You may also want to take a look at our open access book Building Front-End Web Apps with Plain JavaScript [http://web-engineering.info/JsFrontendApp-Book], which includes all parts of the tutorial in one document, and complements them with additional material.

Chapter 1. Enumerations and Enumeration Attributes

1. Enumerations

In all application domains, there are string-valued attributes with a fixed set of possible string values. These attributes are called *enumeration attributes*, and the fixed value sets defining their possible string values are called *enumerations*. 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 may be restricted to one of the **enumeration labels** "male","female" and "undetermined", or to one of the **enumeration codes** "M", "F" and "U". Whenever we deal with codes, we also need to have their corresponding labels, at least in a legend explaining the meaning of each code.

Instead of using the enumeration string values as the internal values of an enumeration attribute (such as `gender`), it is preferable to use a simplified internal representation for them, such as the positive integers 1, 2, 3, etc., which enumerate the possible values. However, since these integers do not reveal their meaning (which is indicated by the enumeration label) in program code, for readability we rather use special constants, called **enumeration literals**, such as `MALE` or `M`, prefixed by the name of the enumeration, like `GenderEL`, in program statements like `this.gender = GenderEL.MALE`. Notice that we follow the convention that the names of enumeration literals are written all upper case, and that we also use the convention to suffix the name of an enumeration data type with "EL" standing for "enumeration literal" (such that we can recognize from the name `GenderEL` that each instance of this datatype is a "gender enumeration literal").

There are also enumerations having records as their instances, such that one of the record fields provides the name of the enumeration literals. An example of such an enumeration is the following set of *units of measurement*:

Table 1.1. Representing an enumeration of records as a table

Units of Measurement		
Unit Symbol	**Unit Name**	**Dimension**
m	meter	length
kg	kilogram	mass
g	gram	mass
s	second	time
ms	milisecond	time

Notice that since both the "Unit Symbol" and the "Unit Name" fields are unique, either of them could be used for the name of the enumeration literals.

In summary, we can distinguish between the following three forms of enumerations:

1. **simple enumerations** define a set of self-explanatory enumeration labels;
2. **code lists** define a set of code/label pairs.
3. **record enumerations** consist of a set of records, so they are defined like classes with simple attributes defining the record fields.

These three forms of enumerations are discussed in more detail below.

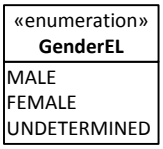
Notice that, since enumerations are used as the range of enumeration attributes, they are considered to be *data types*.

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].

1.1. Simple Enumerations

A *simple enumeration* defines a fixed set of self-explanatory enumeration labels, like in the example of a `GenderEL` enumeration shown in the following UML class diagram:

Figure 1.1. The simple enumeration `GenderEL` represented as a UML enumeration data type

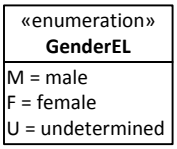


Since the labels of a simple enumeration are being used, in capitalized form, as the names of the corresponding enumeration literals (`GenderEL.MALE`, `GenderEL.FEMALE`, etc.), we may also list the (all upper case) enumeration literals in the UML enumeration data type, instead of the corresponding (lower or mixed case) enumeration labels.

1.2. Code Lists

A *code list* defines a fixed set of code/label pairs. Unfortunately, the UML concept of an enumeration data type does not support the distinction between codes as enumeration literals and their labels. For defining both codes and labels in a UML class diagram in the form of an enumeration data type, we may use the attribute compartment of the data type rectangle and use the codes as attribute names defining the enumeration literals, and set their initial values to the corresponding label. This approach results in a visual representation as in the following diagram:

Figure 1.2. The enumeration `GenderEL` defined as a code list



In the case of a code list, we can use both the codes or the labels as the names of enumeration literals, but using the codes seems preferable for brevity (`GenderEL.M`, `GenderEL.F`, etc.). For displaying the value of an enumeration attribute, it's an option to show not only the label, but also the code, like "male (M)", provided that there is sufficient space. If space is an issue, only the code can be shown.

1.3. Record Enumerations

A *record enumeration* defines a record type with a unique field designated to provide the enumeration literals, and a fixed set of records of that type. In general, a record type is defined by a set of field

definitions (in the form of primitive data type attributes), such that one of the unique fields is defined to be the enumeration literal field, and a set of operation definitions.

Unfortunately, record enumerations, as the most general form of an enumeration data type, are not supported by the current version of UML (2.5) where the general form of an enumeration is defined as a special kind of data type (with optional field and operation definitions) having an additional list of unique strings as enumeration literals (shown in a fourth compartment). The UML definition does neither allow designating one of the unique fields as the enumeration literal field, nor does it allow populating an enumeration with records.

Consequently, for showing a record enumeration in a UML class diagram, we need to find a workaround. For instance, if our modeling tools allows adding a drawing, we could draw a rectangle with four compartments, such that the first three of them correspond to the name, properties and operations compartments of a data type rectangle, and the fourth one is a table with the names of properties/fields defined in the second compartment as column headers, as shown in the table below.

Table 1.2. Representing a record enumeration as a table

UnitEL		
«el» unitSymbol: String unitName: String dimension: String		
Unit Symbol	Unit Name	Dimension
m	meter	length
kg	kilogram	mass
g	gram	mass
s	second	time
ms	milisecond	time

2. Enumeration Attributes

An *enumeration attribute* is an attribute that has an enumeration as its range.

In the user interface, an output field for an enumeration attribute would display the enumeration label, rather than its internal value, the corresponding enumeration index.

For allowing user input to an enumeration attribute, we can use the UI concept of a (drop-down) *selection list*, which may be implemented with an HTML `select` element, such that the enumeration labels would be used as the text content of its `option` elements, while the enumeration indexes would be used as their values. We have to distinguish between *single-valued* and *multi-valued* enumeration attributes. In the case of a **single-valued** enumeration attribute, we use a standard `select` element. In the case of a **multi-valued** enumeration attribute, we use a `select` element with the HTML attribute setting `multiple="multiple"`. For both cases, an example is shown in Figure 1.3. While the single `select` element for "Original language" shows the initially selected option "---" denoting "nothing selected", the multiple `select` element "Other available languages" shows a small window displaying four of the options that can be selected.

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 (like 20), which depends on the number of options the user can see on the screen without scrolling.

For user input to a **single-valued** enumeration attribute, a **radio button group** can be used instead of a single selection list, if the number of enumeration literals is sufficiently small (say, not larger than 7). A radio button group is implemented with a HTML `fieldset` element acting as a container of labeled `input` elements of type "radio", all having the same name, which is normally equal to the name of the represented enumeration attribute.

For user input to a **multi-valued** enumeration attribute, a **checkbox group** can be used instead of a multiple selection list, if the number of enumeration literals is sufficiently small (say, not larger than 7). A checkbox group is implemented with a HTML `fieldset` element acting as a container of labeled `input` elements of type "checkbox", all having the same name, which is normally equal to the name of the represented enumeration attribute.

3. Enumerations in Computational Languages

Defining enumerations is directly supported in information modeling languages (such as in *UML Class Diagrams*), in data schema languages (such as in *XML Schema*, but not in *SQL*), and in many programming languages (such as in *C++* and *Java*, but not in *JavaScript*).

3.1. Enumerations in SQL

Unfortunately, standard SQL does not support enumerations. Some DBMS, such as *MySQL* and *Postgres*, provide their own extensions of SQL column definitions in the CREATE TABLE statement allowing to define enumeration-valued columns.

A MySQL enumeration [https://dev.mysql.com/doc/refman/5.7/en/enum.html] is specified as a list of enumeration labels with the keyword `ENUM` within a column definition, like so:

```
CREATE TABLE people (
    name VARCHAR(40),
    gender ENUM('MALE', 'FEMALE', 'UNDETERMINED')
);
```

A Postgres enumeration [http://postgresguide.com/sexy/enums.html] is specified as a special user-defined type that can be used in columns definitions:

```
CREATE TYPE GenderEL AS ENUM ('MALE', 'FEMALE', 'UNDETERMINED');
CREATE TABLE people (
   name text,
   gender GenderEL
)
```

3.2. Enumerations in XML Schema

In XML Schema, an enumeration datatype can be defined as a simple type restricting the primitive type `xs:string` in the following way:

```
<xs:simpleType name="BookCategoryEL">
  <xs:restriction base="xs:string">
    <xs:enumeration value="NOVEL"/>
    <xs:enumeration value="BIOGRAPHY"/>
    <xs:enumeration value="TEXTBOOK"/>
    <xs:enumeration value="OTHER"/>
  </xs:restriction>
</xs:simpleType>
```

3.3. Enumerations in JavaScript

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}
});
```

This definition allows using the enumeration literals `BookCategoryEL.NOVEL`, `BookCategoryEL.BIOGRAPHY` etc., standing for the enumeration indexes 1, 2 , 3 and 4, in program statements. Notice how this definition 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.

We can also use a generic approach and define a meta-class `Enumeration` for creating enumerations in the form of special JS objects:

```
function Enumeration( enumLabels) {
  var i=0, LBL="";
  this.MAX = enumLabels.length;
  this.labels = enumLabels;
  // generate the enum literals as capitalized keys/properties
  for (i=1; i <= enumLabels.length; i++) {
    LBL = enumLabels[i-1].toUpperCase();
    this[LBL] = i;
  }
  // prevent any runtime change to the enumeration
  Object.freeze( this);
};
```

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

```
var BookCategoryEL = 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`. Also, the label can be retrieved in the following way:

```
formEl.category.value = BookCategoryEL.labels[this.category - 1];
```

As an example, we consider the following model class `Book` with the enumeration attribute `category`:

```
function Book( slots) {
  this.isbn = "";     // string
  this.title = "";    // string
  this.category = 0;  // number (BookCategoryEL)
  if (arguments.length > 0) {
    this.setIsbn( slots.isbn);
    this.setTitle( slots.title);
    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 (!Number.isInteger(c) || c < 1 ||
            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`.

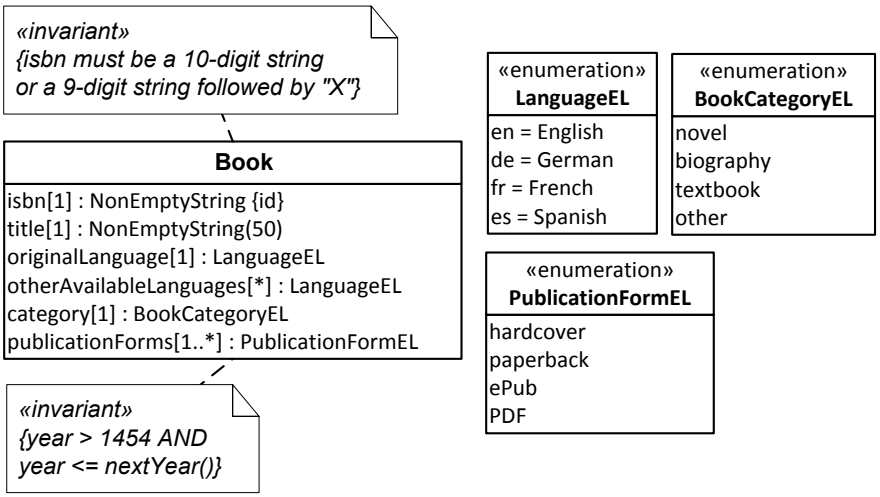
Chapter 2. Implementing Enumeration Attributes in a Plain JS App

In this chapter, we show how to build a front-end web application with **enumeration attributes**, using plain JavaScript. In addition to the topic of enumeration attributes, we also show how to deal with ***multi-valued attributes*** because in many cases, enumeration attributes are multi-valued.

We again consider the simple data management problem that was considered in Part 2 of this tutorial. So, again, the purpose of our app is to manage information about books. But now we have four additional *enumeration attributes*, as shown in the UML class diagram in Figure 2.1 below:

1. the single-valued mandatory attribute `originalLanguage` with the enumeration datatype `LanguageEL` as its range,
2. the multi-valued optional attribute `otherAvailableLanguages` with range `LanguageEL`,
3. the single-valued mandatory attribute `category` with range `BookCategoryEL`
4. the multi-valued mandatory attribute `publicationForms` with range `PublicationFormEL`

Figure 2.1. An information design model for the object type `Book`



Notice that the attributes `otherAvailableLanguages` and `publicationForms` are *multivalued*, as indicated by their multiplicity expressions [*] and [1..*]. This means that the possible values of these attributes are sets of enumeration literals, such as the set {ePub, PDF}, which can be represented in JavaScript as a corresponding array list of enumeration literals, `[PublicationFormEL.EPUB, PublicationFormEL.PDF]`.

The meaning of the design model and its enumeration properties can be illustrated by a sample data population for the model class `Book`:

Table 2.1. Sample data for `Book`

ISBN	Title	Original language	Other languages	Category	Publication forms
0553345842	The Mind's I	English (en)	de, es, fr	novel	paperback, ePub, PDF

ISBN	Title	Original language	Other languages	Category	Publication forms
1463794762	The Critique of Pure Reason	German (de)	de, es, fr, pt, ru	other	paperback, PDF
1928565379	The Critique of Practical Reason	German (de)	de, es, fr, pt, ru	other	paperback
0465030793	I Am A Strange Loop	English (en)	es	textbook	hardcover, ePub

1. New Issues

Compared to the Validation App [http://oxygen.informatik.tu-cottbus.de/webeng/JsFrontendApp/ValidationApp/index.html] discussed in Part 2 (Validation App Tutorial [http://oxygen.informatik.tu-cottbus.de/webeng/JsFrontendApp/validation-tutorial.html]) we have to deal with the following new issues:

1. Enumeration datatypes have to be defined in a suitable way as part of the model code.

2. Enumeration attributes have to be defined in model classes and handled in the user interface.

In terms of coding, the new issues are:

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

   a. defining the **enumeration datatypes** with the help of a utility (meta-)class `Enumeration`, which is discussed below;

   b. defining the **single-valued enumeration attributes** (like `Book::originalLanguage`) together with their checks and setters (like `Book.checkOriginalLanguage`, `Book.prototype.setOriginalLanguage`);

   c. defining the **multi-valued enumeration attributes** (like `Book::publicationForms`) together with their checks and setters (like `Book.checkPublicationForms`, `Book.prototype.setPublicationForms` and `Book.prototype.addPublicationForm`);

   d. extending the methods `Book.update`, and `Book.prototype.toString` such that they take care of the added enumeration attributes.

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

   a. adding new table columns in `listBooks.html` and suitable form controls (such as **selection lists**, **radio button groups** or **checkbox groups**) in `createBook.html` and `upateBook.html`;

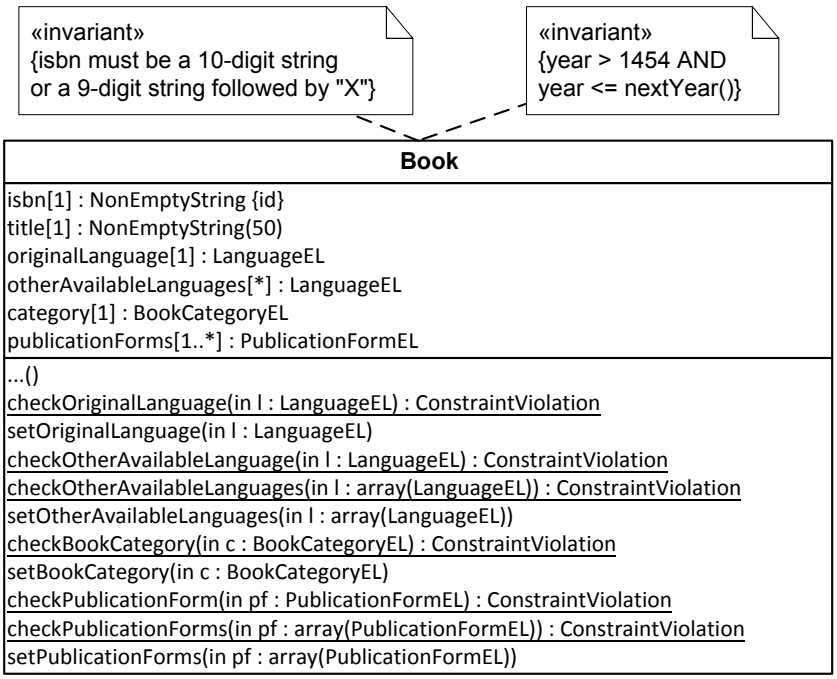
   b. creating output for the new attributes in the method `pl.view.listBooks.setupUserInterface()`;

   c. allowing input for the new attributes in the methods `pl.view.createBook.setupUserInterface()` and `pl.view.updateBook.setupUserInterface()`.

2. Make a JavaScript Data Model

Using the information design model shown in Figure 2.1 above as the starting point, we make a *JavaScript* data model, essentially by adding checks and setters, as explained in Part 2 (Validation App Tutorial [http://oxygen.informatik.tu-cottbus.de/webeng/JsFrontendApp/validation-tutorial.html]):

Figure 2.2. A JavaScript data model, for the object type `Book`



Notice that, for any multi-valued enumeration attribute (like `someThings`) we add a class-level check function for single values (like `checkSomeThing`) and another one for value sets (like `checkSomeThings`) returning an object of type `ConstraintViolation`, and a setter (like `setSomeThings`) for assigning a set of enumeration literals to the multi-valued enumeration attribute.

3. Set up the Folder Structure with Library Files

The MVC folder structure of our enumeration app extends the structure of the validation app by adding the file `Enumeration.js` in the `lib` folder. Thus, we get the following folder structure containing six initial files:

```
publicLibrary
  css
    main.css
  lib
    browserShims.js
    errorTypes.js
    util.js
    Enumeration.js
  src
    ctrl
    model
    view
  index.html
```

In the `Enumeration.js` file, discussed in the next section, we define a meta-class `Enumeration` for creating enumerations as instances of this meta-class with the help of statements like `GenderEL = new Enumeration(["male", "female", "undetermined"])`.

4. The Meta-Class `Enumeration`

We define an `Enumeration` meta-class, which supports both simple enumerations and code lists, but not record enumerations. While a simple enumeration is defined by a list of labels in the form of a JS array as the constructor arguiment such that the labels are used for the names of the enumeration literals, a code list is defined as a special kind of key-value map in the form of a JS object as the constructor arguiment such that the codes are used for the names of the enumeration literals. Consequently, the constructor needs to test if the invocation argument is an array or an object. The following first part of the code shows how simple enumerations are created:

```
function Enumeration( enumArg) {
  var i=0, lbl="", LBL="";
  if (Array.isArray( enumArg)) {
    // a simple enum defined by a list of labels
    if (!enumArg.every( function (n) {
            return (typeof(n) === "string"); })) {
      throw new OtherConstraintViolation(
        "A list of enumeration labels must be an array of strings!");
    }
    this.labels = enumArg;
    this.enumLitNames = this.labels;
    this.codeList = null;
  } else if (typeof(enumArg) === "object" &&
             Object.keys( enumArg).length > 0) {
    ... // a code list defined by a map
  } else  {
    throw new OtherConstraintViolation(
        "Invalid Enumeration constructor argument: "+ enumArg);
  }
  this.MAX = this.enumLitNames.length;
  // generate the enumeration literals by capitalizing/normalizing
  for (i=1; i <= this.enumLitNames.length; i++) {
    // replace " " and "-" with "_"
    lbl = this.enumLitNames[i-1].replace(/( |-)/g, "_");
    // convert to array of words, capitalize them, and re-convert
    LBL = lbl.split("_").map( function (lblPart) {
        return lblPart.toUpperCase();
    }).join("_");
    // assign enumeration index
    this[LBL] = i;
  }
  Object.freeze( this);
};
```

Notice that a label like "text book" or "text-book" is converted to the enumeration literal name "TEXT_BOOK". The following second part of the code shows how code list enumerations are created:

```
function Enumeration( enumArg) {
  var i=0, lbl="", LBL="";
  if (Array.isArray( enumArg)) {
    ...
  } else if (typeof(enumArg) === "object" &&
             Object.keys( enumArg).length > 0) {
    // a code list defined by a map
    if (!Object.keys( enumArg).every( function (code) {
            return (typeof( enumArg[code]) === "string"); })) {
      throw new OtherConstraintViolation(
          "All values of a code list map must be strings!");
    }
    this.codeList = enumArg;
    // use the codes as the names of enumeration literals
    this.enumLitNames = Object.keys( this.codeList);
```

```
    this.labels = this.enumLitNames.map( function (c) {
          return enumArg[c] +" ("+ c +")";
    });
  } else  {
    throw new OtherConstraintViolation(
        "Invalid Enumeration constructor argument: "+ enumArg);
  }
  ...  // as above
};
```

5. Write the Model Code

How to Encode a JavaScript Data Model

5.1. Encode the enumerations

Enumerations are encoded in the following way with the help of the meta-class `Enumeration`:

```
var  PublicationFormEL = new Enumeration(["hardcover","paperback","ePub","PDF"])
var  LanguageEL = new Enumeration({"en":"English", "de":"German",
                  "fr":"French", "es":"Spanish"})
```

Notice that `LanguageEL` defines a code list, while `PublicationFormEL` defines a simple enumeration.

5.2. Encode the model class as a constructor function

The class `Book` is encoded by means of a corresponding JavaScript *constructor function* 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.originalLanguage = 0;  // number (from LanguageEL)
  this.otherAvailableLanguages = [];  // list of numbers (from LanguageEL)
  this.category = 0;  // number (from BookCategoryEL)
  this.publicationForms = [];  // list of numbers (from PublicationFormEL)
  // assign properties only if the constructor is invoked with an argument
  if (arguments.length > 0) {
    this.setIsbn( slots.isbn);
    this.setTitle( slots.title);
    this.setOriginalLanguage( slots.originalLanguage);
    this.setOtherAvailableLanguages( slots.otherAvailableLanguages);
    this.setCategory( slots.category);
    this.setPublicationForms( slots.publicationForms);
  }
};
```

5.3. Encode the enumeration attribute checks

Encode the enumeration attribute checks in the form of class-level ('static') functions that check if the argument is a valid enumeration index not smaller than 1 and not greater than the enumeration's MAX value. For instance, for the `checkOriginalLanguage` function we obtain the following code:

```
Book.checkOriginalLanguage = function (l) {
  if (l === undefined) {
    return new MandatoryValueConstraintViolation(
        "An original language must be provided!");
  } else if (!Number.isInteger( l) || l < 1 || l > LanguageEL.MAX) {
```

```
    return new RangeConstraintViolation("Invalid value for original language: "+ l);
  } else {
    return new NoConstraintViolation();
  }
};
```

For a multi-valued enmueration attribute, such as `publicationForms`, we break down the check code into two functions, one for checking if a value is a valid enumeration index (`checkPublicationForm`), and another one for checking if all members of a set of values are valid enumeration indexes (`checkPublicationForms`):

```
Book.checkPublicationForm = function (p) {
  if (!p && p !== 0) {
    return new MandatoryValueConstraintViolation(
        "No publication form provided!");
  } else if (!Number.isInteger( p) || p < 1 ||
        p > PublicationFormEL.MAX) {
    return new RangeConstraintViolation(
        "Invalid value for publication form: "+ p);
  } else {
    return new NoConstraintViolation();
  }
};
Book.checkPublicationForms = function (pubForms) {
  var i=0, constrVio=null;
  if (pubForms.length === 0) {
    return new MandatoryValueConstraintViolation(
        "No publication form provided!");
  } else {
    for (i=0; i < pubForms.length; i++) {
      constrVio = Book.checkPublicationForm( pubForms[i]);
      if (!(constrVio instanceof NoConstraintViolation)) {
        return constrVio;
      }
    }
    return new NoConstraintViolation();
  }
};
```

5.4. Encode the enumeration attribute setters

Both for single-valued and for multi-valued enumeration attributes an ordinary setter is defined. In the case of a multi-valued enumeration attribute, this setter assigns an entire set of values (in the form of a JS array) to the attribute after checking its validity.

5.5. Write a serialization function

The object serialization function now needs to include the values of enumeration attributes:

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

Notice that for multi-valued enumeration attributes we call the `toString()` function that is predefined for JS arrays.

5.6. Data management operations

There are only two new issues in the data management operations compared to the validation app:

1. We have to make sure that the `util.cloneObject` method, which is used in `Book.update`, takes care of copying array-valued attributes, which we didn't have before (in the validation app).

2. In the `Book.update` method we now have to check if the values of array-valued attributes have changed, which requires to test if two arrays are equal or not. For code readability, we add an array equality test method to `Array.prototype` in `browserShims.js`, like so:

   ```
   Array.prototype.isEqualTo = function (a2) {
     return (this.length === a2.length) && this.every( function( el, i) {
       return el === a2[i]; });
   };
   ```

   This allows us to express these tests in the following way:

   ```
   if (!book.publicationForms.isEqualTo( slots.publicationForms)) {
     book.setPublicationForms( slots.publicationForms);
     updatedProperties.push("publicationForms");
   }
   ```

5.7. Creating test data

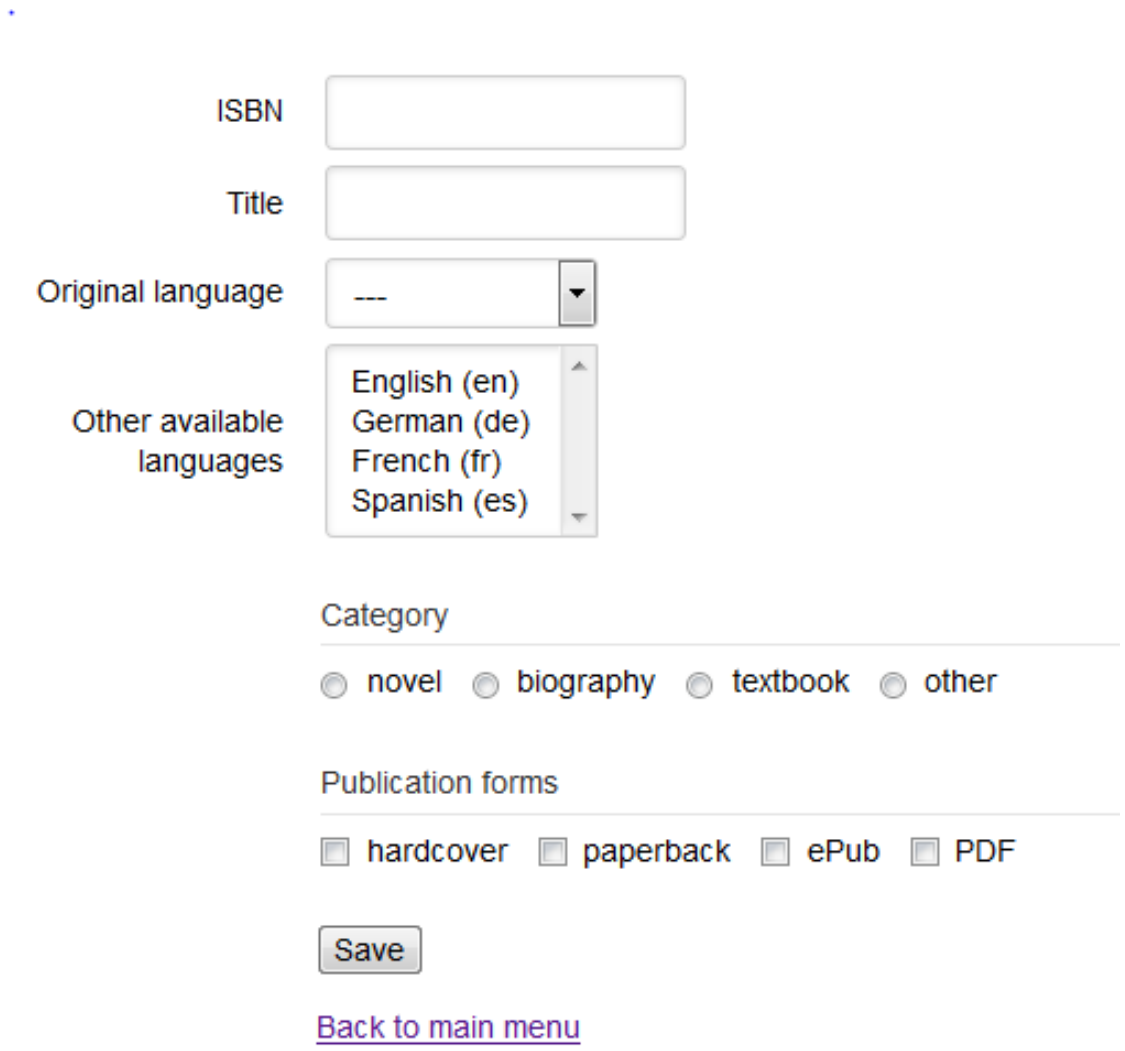
In the test data records that are created by Book.createTestData, we now have to provide values for single- and multi-valued enumeration attributes. For readability, we use enumeration literals instead of enumeration indexes:

```
Book.createTestData = function () {
  try {
    Book.instances["006251587X"] = new Book({isbn:"006251587X",
        title:"Weaving the Web", originalLanguage:LanguageEL.EN,
        otherAvailableLanguages:[LanguageEL.DE,LanguageEL.FR],
        category:BookCategoryEL.NOVEL,
        publicationForms:[PublicationFormEL.EPUB,PublicationFormEL.PDF]});
    ...
    Book.saveAll();
  } catch (e) {
    console.log( e.constructor.name + ": " + e.message);
  }
};
```

6. Write the View and Controller Code

The examle app's user interface for creating a new book record looks as in Figure 2.3 below.

Figure 2.3. The user interface for creating a new book record with ISBN, title and four enumeration attributes



6.1. Selection lists

We use HTML selection lists for rendering the enumeration attributes `originalLanguage` and `otherAvailableLanguages` in the HTML forms in `createBook.html` and `upateBook.html`. Since the attribute `otherAvailableLanguages` is multi-valued, we need a **multiple selection** list for it, as shown in the following HTML code:

```
<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="ol">Original language</label>
      <select id="ol" name="originalLanguage"></select>
    </div>
```

```
    <div class="pure-control-group">
      <label for="oal">Other available languages</label>
      <select id="oal" name="otherAvailableLanguages"
              multiple="multiple"></select>
    </div>
    ...
  </form>
</body>
```

While we define the `select` container elements for these selection lists in `createBook.html` and `upateBook.html`, we fill in their `option` child elements dynamically in the `setupUserInterface` methods in `view/createBook.js` and `view/updateBook.js` with the help of the utility method `util.fillSelectWithOptions`. In the case of a single `select` element, the user's single-valued selection can be retrieved from the `value` attribute of the `select` element, while in the case of a multiple `select` element, the user's multi-valued selection can be retrieved from the `selectedOptions` attribute of the `select` element.

6.2. Choice widgets

Since the enumeration attributes `category` and `publicationForms` have not more than seven possible values, we can use a *radio button group* and a *checkbox group* for rendering them in an HTML-form-based UI. Such **choice widgets** are formed with the help of the container element `fieldset` as shown in the following HTML fragment:

```
<body>
  <h1>Public Library: Create a new book record</h1>
  <form id="Book" class="pure-form pure-form-aligned">
    ...
    <fieldset class="pure-controls" data-bind="category">
      <legend>Category</legend>
    </fieldset>
    <fieldset class="pure-controls" data-bind="publicationForms">
      <legend>Publication forms</legend>
    </fieldset>
    <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>
```

Notice that we use a custom attribute `data-bind` for indicating to which attribute of the underlying model class the choice widget is bound. In the same way as the `option` child elements of a selection list, also the `labeled` `input` child elements of a choice widget are created dynamically with the help of the utility method `util.createChoiceWidget` in the `setupUserInterface` methods in `view/createBook.js` and `view/updateBook.js`. Like a selection list implemented with the HTML `select` element that povides the user's selection in the `value` or `selectedOptions` attribute, our choice widgets also need a DOM attribute that holds the user's single- or multi-valued choice. We dynamically add a custom attribute `data-value` to the choice widget's `fieldset` element for this purpose in `util.createChoiceWidget`.

```
setupUserInterface: function () {
  var formEl = document.forms['Book'],
      origLangSelEl = formEl.originalLanguage,
      otherAvailLangSelEl = formEl.otherAvailableLanguages,
      categoryFieldsetEl = formEl.querySelector(
          "fieldset[data-bind='category']"),
      pubFormsFieldsetEl = formEl.querySelector(
          "fieldset[data-bind='publicationForms']"),
      submitButton = formEl.commit;
  // set up the originalLanguage selection list
  util.fillSelectWithOptions( origLangSelEl, LanguageEL.labels);
  // set up the otherAvailableLanguages selection list
```

```
  util.fillSelectWithOptions( otherAvailLangSelEl, LanguageEL.labels);
  // set up the category radio button group
  util.createChoiceWidget( categoryFieldsetEl, "category", [],
      "radio", BookCategoryEL.labels);
  // set up the publicationForms checkbox group
  util.createChoiceWidget( pubFormsFieldsetEl, "publicationForms", [],
      "checkbox", PublicationFormEL.labels);
  ...
},
```

6.3. Responsive validation for selection lists and choice widgets

Since selection lists and choice widgets do not allow arbitrary user input, we do not have to check constraints such as range constraints or pattern constraints on user input, but only mandatory value constraints. In our example app the enumeration attributes `originalLanguage`, `category` and `publicationForms` are mandatory, while `otherAvailableLanguages` is optional.

For selection lists, whenever a new selection is made by the user, a `change` event is raised by the browser, so we declare the following mandatory value check as an event listener for `change` events on the `select` element:

```
setupUserInterface: function () {
  ...
  // add event listeners for responsive validation
  formEl.title.addEventListener("input", function () {
      formEl.title.setCustomValidity(
          Book.checkTitle( formEl.title.value).message);
  });
  // simplified validation: check only mandatory value
  origLangSelEl.addEventListener("change", function () {
    origLangSelEl.setCustomValidity(
        (!origLangSelEl.value) ? "A value must be selected!":"" );
  });
  // simplified validation: check only mandatory value
  categoryFieldsetEl.addEventListener("click", function () {
    formEl.category[0].setCustomValidity(
        (!categoryFieldsetEl.getAttribute("data-value")) ?
            "A category must be selected!":"" );
  });
  // simplified validation: check only mandatory value
  pubFormsFieldsetEl.addEventListener("click", function () {
    var val = pubFormsFieldsetEl.getAttribute("data-value");
    formEl.publicationForms[0].setCustomValidity(
       (!val || Array.isArray(val) && val.length === 0) ?
            "At least one publication form must be selected!":"" );
  });
  ...
},
```

7. Run the App and Get the Code

You can run the enumeration app [http://oxygen.informatik.tu-cottbus.de/webeng/JsFrontendApp/EnumerationApp/index.html] from our server or download the code [http://oxygen.informatik.tu-cottbus.de/webeng/JsFrontendApp/EnumerationApp.zip] as a ZIP archive file.

8. Possible Variations and Extensions

The meta-class `Enumeration` could be extended by adding support for *record enumerations*.

9. Points of Attention

9.1. Database size and memory management

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, roughly 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.

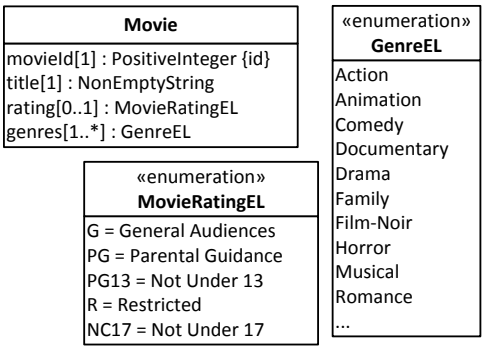
9.2. Boilerplate code

Another issue with the do-it-yourself code of this example app is the *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 another tutorial [mODELcLASS-validation-tutorial.html], we 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.

10. Practice Project

The purpose of the app to be built is managing information about movies. The app deals with just one object type, `Movie`, and with two enumerations, as depicted in the following class diagram. In the subsequent parts of the tutorial, you will extend this simple app by adding actors and directors as further model classes, and the associations between them.

Figure 2.4. The object type `Movie` defined together with two enumerations.



First make a list of all the constraints that have been expressed in this model. Then code the app by following the guidance of this tutorial and the Validation Tutorial [http://oxygen.informatik.tu-cottbus.de/webeng/JsFrontendApp/validation-tutorial.html].

Compared to the practice project of our validation tutorial, two attributes have been added: the optional single-valued enumeration attribute `rating`, and the multi-valued enumeration attribute `genres`.

Following the tutorial, you have to take care of

1. defining the *enumeration datatypes* `MovieRatingEL` and `GenreEL` with the help of the meta-class `Enumeration`;

2. defining the *single-valued enumeration attribute* `Movie::rating` together with a check and a setter;

3. defining the *multi-valued enumeration attributes* `Movie::genres` together with a check and a setter;

4. extending the methods `Movie.update`, and `Movie.prototype.toString` such that they take care of the added enumeration attributes.

in the *model* code of your app, while In the *user interface* ("view") code you have to take care of

1. adding new table columns in `listMovies.html` and suitable form controls (such as *selection lists*, *radio button groups* or *checkbox groups*) in `createMovie.html` and `upateMovie.html`;

2. creating output for the new attributes in the method `...view.listMovies.setupUserInterface();`

3. allowing input for the new attributes in the methods `...view.createMovie.setupUserInterface()` and `...view.upateMovie.setupUserInterface()`.

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

Table 2.2. Sample data

Movie ID	Title	Rating	Genres
1	Pulp Fiction	R	Crime, Drama
2	Star Wars	PG	Action, Adventure, Fantasy, Sci-Fi
3	Casablanca	PG	Drama, Film-Noir, Romance, War
4	The Godfather	R	Crime, Drama

In this assignment, and in all further assignment, you have to make sure that your pages comply with the XML syntax of HTML5 (by means of XHTML5 validation), and that your JavaScript code complies with our Coding Guidelines [http://oxygen.informatik.tu-cottbus.de/webeng/Coding-Guidelines.html] and is checked with JSLint (http://www.jslint.com [http://www.jslint.com/]).

If you have any questions about how to carry out this project, you can ask them on our discussion forum [http://web-engineering.info/forum/13].