JavaScript Front-End Web App Tutorial Part 2: Adding Constraint Validation

Learn how to build a front-end web application with responsive constraint validation using plain JavaScript

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

JavaScript Front-End Web App Tutorial Part 2: Adding Constraint Validation: Learn how to build a front-end web application with responsive constraint validation using plain JavaScript

by Gerd Wagner

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

This tutorial is also available in the following formats: PDF. You may run the example app from our server, or download the code as a ZIP archive file. See also our Web Engineering project page.

Publication date 2021-04-12

Table of Contents

List of Figures

List of Tables

Foreword

This tutorial is Part 2 of our series of six tutorials about model-based development of front-end web applications with plain JavaScript. It shows how to build a single-class front-end web application with constraint validation using plain JavaScript, and no third-party framework or library. While libraries and frameworks may help to increase productivity, they also create black-box dependencies and overhead, and they are not good for learning how to do it yourself.

A front-end web application can be provided by any web server, but it is executed on the user's computer device (smartphone, tablet or notebook), and not on the remote web server. Typically, but not necessarily, a front-end web application is a single-user application, which is not shared with other users.

The *minimal* JavaScript app that we have discussed in the first part of this 6-part tutorial has been limited to support the minimum functionality of a data management app only. However, it did not take care of preventing users 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/storage code of the app and in the user interface built with HTML5.

The simple form of a JavaScript data management application presented in this tutorial takes care of only one object type ("books") for which it supports the four standard data management operations (**C**reate/**R**etrieve/**U**pdate/**D**elete). It extends the minimal app discussed in the Minimal App Tutorial by adding ***constraint validation*** (and some CSS styling), but it needs to be enhanced by adding further important parts of the app's overall functionality. The other parts of the tutorial are:

- Part 1: Building a **minimal** app.
- Part 3: Dealing with **enumerations**.
- Part 4: 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: Managing **bidirectional associations**, such as the associations between books and publishers and between books and authors, also assigning books to authors and to publishers.
- Part 6: Handling **subtype** (inheritance) relationships between object types.

Chapter 1. Integrity Constraints and Data Validation

1. Introduction

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 entered by a user and stored in the application's database.

For instance, if an application is managing data about persons including their birth dates and their death dates, then we must make sure that for any person record with a death date, this date is not before that person's birth date.

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, like choosing an application development framework that provides sufficient support for it.

Unfortunately, many web application development frameworks do not provide sufficient support for defining integrity constraints and performing data validation. Integrity constraints should be defined in one (central) place in an app, and then be used for configuring the user interface and for validating data in different parts of the app, such as in the user interface and in the database. In terms of usability, the goals should be:

1. To prevent the user from entering invalid data in the user interface (UI) by limiting the input options, if possible.
2. To detect and reject invalid user input as early as possible by performing constraint validation in the UI for those UI widgets where invalid user input cannot be prevented by limiting the input options.
3. To prevent that invalid data pollutes the app's main memory state and persistent database state by performing constraint validation also in the model layer and in the database.

HTML5 provides support for validating user input in an HTML-forms-based user interface (UI). Here, the goal is to provide immediate feedback to the user whenever invalid data has been entered into a form field. This UI mechanism of ***responsive validation*** is an important feature of modern web applications. In traditional web applications, the back-end component validates the data and returns the validation results in the form of a set of error messages to the front-end. Only then, often several seconds later, and in the hard-to-digest form of a bulk message, does the user get the validation feedback.

2. Integrity Constraints

Integrity constraints (or simply *constraints*) are logical conditions on the data of an app. They may take many different forms. The most important type of constraints, ***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 cases 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 match 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 (also called 'Key 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 must not be changed after it has been assigned initially.

The visual language of UML class diagrams supports defining integrity constraints either in a special way for special cases (like with predefined keywords), or, in the general case, with the help of *invariants*, which are conditions expressed either in plain English or in the *Object Constraint Language (OCL)* and shown in a special type of rectangle attached to the model element concerned. We use UML class diagrams for modeling constraints in *design models* 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 not smaller than `l` or the special value `*` standing for *unbounded*. For showing property multiplicity (or cardinality) constrains in a class diagram, multiplicity expressions are enclosed in brackets and appended to the property name, as shown in the `Person` class rectangle below.

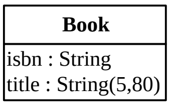
In the following sections, we discuss the different types of property constraints listed above in more detail. We also show how to express some of them in computational languages such as *UML* class diagrams, *SQL* table creation statements, *JavaScript* model class definitions, or the annotation-based languages *Java Bean Validation* annotations and *ASP.NET Data Annotations*.

Any systematic approach to constraint validation also requires to define a set of error (or 'exception') classes, including one for each of the standard property constraints listed above.

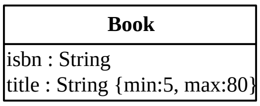
2.1. String Length Constraints

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)`.

UML does not define any special way of expressing string length constraints in class diagrams. Of course, we always have the option to use an *invariant* for expressing any kind of constraint, but it seems preferable to use a simpler form of expressing these property constraints. One option is to append a maximum length, or both a minimum and a maximum length, in parenthesis to the datatype name, like so

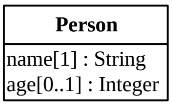


Another option is to use min/max constraint keywords in the property modifier list:



2.2. 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 code 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 "A name must be provided!"; // constraint violation error message
  } else return "";  // no constraint violation
};
```

With Java Bean Validation, a mandatory property like `name` is annotated with `NotNull` in the following way:

```
@Entity
public class Person {
  @NotNull
  private String name;
  private int age;
}
```

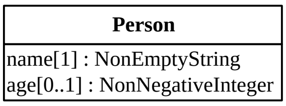
The equivalent ASP.NET Data Annotation is `Required` as shown in

```
public class Person{
  [Required]
  public string name { get; set; }
  public int age { get; set; }
}
```

2.3. 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 code a range constraint by a check function, as illustrated in the following example:

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

```
};
```

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.

In a Java EE web app, for declaring empty strings as non-admissible user input we must set the context parameter

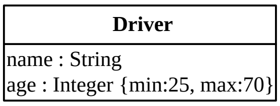
```
javax.faces.INTERPRET_EMPTY_STRING_SUBMITTED_VALUES_AS_NULL
```

to `true` in the web deployment descriptor file `web.xml`.

In ASP.NET, empty strings are non-admissible by default.

2.4. 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 [25,70]. In a class diagram, we can define such a constraint by using the property modifiers `min` and `max`, as shown for the `age` attribute of the `Driver` class in the following diagram.



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 drivers(
  name  VARCHAR NOT NULL,
  age   INTEGER CHECK (age >= 25 AND age <= 70)
)
```

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

```
Driver.checkAge = function (a) {
  if (a < 25 || a > 70) {
    return "Age must be between 25 and 70!";
  } else return "";
};
```

In Java Bean Validation, we express this interval constraint by adding the annotations `Min(0)` and `Max(120)` to the property `age` in the following way:

```
@Entity
public class Driver {
  @NotNull
  private String name;
  @Min(25) @Max(70)
  private int age;
```

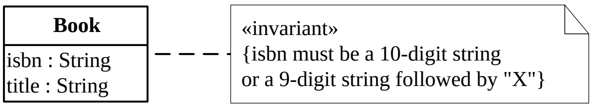
```
}
```

The equivalent ASP.NET Data Annotation is `Range(25,70)` as shown in

```
public class Driver{
  [Required]
  public string name { get; set; }
  [Range(25,70)]
  public int age { get; set; }
}
```

2.5. 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 the following diagram.



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).

In JavaScript, we can code 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 "The ISBN must be a 10-digit string or a 9-digit string followed by 'X'!";
  } else return "";
};
```

In Java EE Bean Validation, this pattern constraint for `isbn` is expressed with the annotation `Pattern` in the following way:

```
@Entity
public class Book {
  @NotNull
  @Pattern(regexp="^\\(\d{9}(\d|X))$")
  private String isbn;
  @NotNull
```

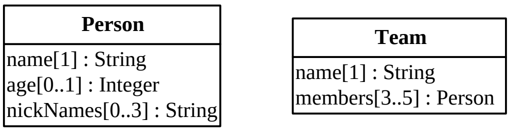
```
  private String title;
}
```

The equivalent ASP.NET Data Annotation is `RegularExpression` as shown in

```
public class Book{
  [Required]
  [RegularExpression(@"^(\d{9}(\d|X))$")]
  public string isbn { get; set; }
  public string title { get; set; }
}
```

2.6. 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 the following diagram, which contains two examples of properties 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 (or maximum) cardinality constraint is an on-delete (or on-insert) trigger that tests the number of rows with the same reference as the deleted (or inserted) row.

In JavaScript, we can code 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:

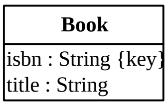
```
Person.checkNickNames = function (nickNames) {
  if (nickNames.length > 3) {
    return "There must be no more than 3 nicknames!";
  } else return "";
};
```

With Java Bean Validation annotations, we can specify

```
@Size( max=3)
List<String> nickNames
@Size( min=3, max=5)
List<Person> members
```

2.7. Uniqueness Constraints

A *uniqueness constraint* (or *key constraint*) requires that a property's value (or the value list of a list of properties in the case of a composite key constraint) is unique among all instances of the given object type. For instance, in a UML class diagram with the object type `Book` we can define the `isbn` attribute to be *unique*, or, in other words, a *key*, by appending the (user-defined) property modifier keyword `key` in curly braces to the attribute's definition in the `Book` class rectangle shown in the following diagram.



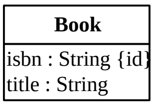
In an SQL table creation statement, a uniqueness constraint is expressed by appending the keyword `UNIQUE` to the column definition as in the following example:

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

In JavaScript, we can code 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.

2.8. Standard Identifiers (Primary Keys)

A unique attribute (or a composite key) can be declared to be the standard identifier for objects of a given type, if it is mandatory (or if all attributes of the composite key are mandatory). We can indicate this in a UML class diagram with the help of the property modifier `id` appended to the declaration of the attribute `isbn` as shown in the following diagram.



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

Often, practitioners do not recommended using a composite key as a standard ID, since composite identifiers are more difficult to handle and not always supported by tools. Whenever an object type does not have a key attribute, but only a composite key, it may therefore be preferable to add an artificial standard ID attribute (also called *surrogate ID*) to the object type. However, each additional surrogate ID has a price: it creates some cognitive and computational overhead. Consequently, in the case of a simple composite key, it may be preferable not to add a surrogate ID, but use the composite key as the standard ID.

There is also an argument against using any real attribute, such as the `isbn` attribute, for a standard ID. The argument points to the risk that the values even of natural ID attributes like `isbn` may have to be changed during the life time of a business object, and any such change would require an unmanageable effort to change also all corresponding ID references. However, the business semantics of natural ID attributes implies that they are frozen. Thus, the need of a value change can only occur in the case of a data input error. But such a case is normally detected early in the life time of the object concerned, and at this stage the change of all corresponding ID references is still manageable.

Standard IDs 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 object-oriented programming languages, like JavaScript and Java, we cannot code a standard ID declaration, because this would have to be part of the metadata of a class definition, and there is no support for such metadata. However, we should still check the implied mandatory value and uniqueness constraints.

2.9. Referential Integrity Constraints

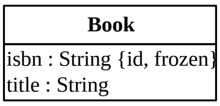
A referential integrity constraint requires that the values of a reference property refer to an object that exists in the population of the property's range class. Since we do not deal with reference properties in this chapter, we postpone the discussion of referential integrity constraints to Part 4 of our tutorial.

2.10. Frozen and Read-Only 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. This includes the special case of ***read-only value constraints*** on mandatory properties that are initialized at object creation time.

Typical examples of properties with a frozen value constraint are standard identifier attributes and event properties. In the case of events, the semantic principle that the past cannot be changed prohibits that the property values of events can be changed. In the case of a standard identifier attribute we may want to prevent users from changing the ID of an object since this requires that all references to this object using the old ID value are changed as well, which may be difficult to achieve (even though SQL provides special support for such ID changes by means of its `ON UPDATE CASCADE` clause for the change management of foreign keys).

The following diagram shows how to define a frozen value constraint for the `isbn` attribute:



In Java, a *read-only* value constraint can be enforced by declaring the property to be `final`. In JavaScript, a *read-only* property slot can be implemented as in the following example:

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

where the property slot `obj.teamSize` is made unwritable. An entire object `obj` can be frozen with `Object.freeze( obj)`.

We can implement a frozen value constraint for a property in the property's setter method like so:

```
Book.prototype.setIsbn = function (i) {
  if (this.isbn === undefined) this.isbn = i;
```

```
  else console.log("Attempt to re-assign a frozen property!");
}
```

2.11. Beyond property constraints

So far, we have only discussed how to define and check *property constraints*. However, in certain cases there may be also integrity constraints that do not just depend on the value of a particular property, but rather on

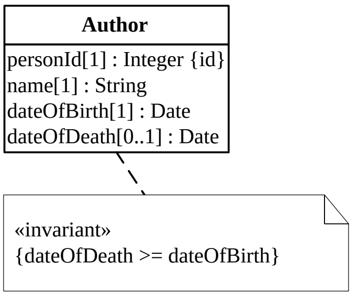
1. the values of several properties of a particular object (object-level constraints),
2. the value of a property before and its value after a change attempt (dynamic constraints),
3. the set of all instances of a particular object type (type-level constraints),
4. the set of all instances of several object types.

OCL

The *Object Constraint Language* (OCL) was defined in 1997 as a formal logic language for expressing integrity constraints in UML version 1.1. Later, it was extended for allowing to define also (1) derivation expressions for defining derived properties, and (2) preconditions and postconditions for operations, in a class model.

In a class model, property constraints can be expressed within the property declaration line in a class rectangle (typically with keywords, such as `id`, `max`, etc.). For expressing more complex constraints, such as object-level or type-level constraints, we can attach an *invariant* declaration box to the class rectangle(s) concerned and express the constraint in unambiguous plain English or in pseudo-code . A simple example of an object-level constraint expressed as an invariant is shown in Figure 1.1.

Figure 1.1. An example of an object-level constraint



A general approach for implementing *object-level constraint validation* consists of taking the following steps:

1. Choose a fixed name for an object-level constraint validation function, such as `validate`.
2. For any class that needs object-level constraint validation, define a `validate` function returning either a `ConstraintViolation` or a `NoConstraintViolation` object.
3. Call this function, if it exists, for the given model class,

a. in the UI/view, on form submission;

b. in the model class, before save, both in the `create` and in the `update` method.

Constraints affecting two or more model classes could be defined in the form of static methods (in a model layer method library) that are invoked from the `validate` methods of the affected model classes.

3. Responsive Validation

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 back-end component running on a remote web server. Only this back-end component validates the data and returns the validation results in the form of a set of error messages to the front-end. Only then, often 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. Technically, this can be achieved with the help of event handlers for the user interface events `input` or `change`.

Responsive validation requires a data validation mechanism in the user interface (UI), such as the HTML5 form validation API. Alternatively, the jQuery Validation Plugin can be used as a (non-HTML5-based) form validation API.

The HTML5 form validation API essentially provides new types of `input` fields (such as `number` or `date`) and a set of new attributes for form control elements for the purpose of supporting responsive validation performed by the browser. Since using the new validation attributes (like `required`, `min`, `max` and `pattern`) implies defining constraints in the UI, they are not really useful in a general approach where constraints are only checked, but not defined, in the UI.

Consequently, 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 field as either valid or invalid by assigning either an empty string or a non-empty (constraint violation) message string.

The second method, `checkValidity`, is invoked on a form before user input data is committed or saved (for instance with a form submission). It tests, if all fields have a valid value. For having the browser automatically displaying any constraint violation messages, we need to have a `submit` event, even if we don't really submit the form, but just use a `save` button.

See this Mozilla tutorial or this HTML5Rocks tutorial for more about the HTML5 form validation API.

4. Constraint Validation in MVC Applications

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. Consequently, data validation needs to start in the user interface (UI).

However, it is not sufficient to perform data validation in the UI. 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 other 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 Bean Validation Annotations* or *ASP.NET Data Annotations*) and generate the back-end/model and front-end/UI validation code both in a back-end 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 back-end, 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.

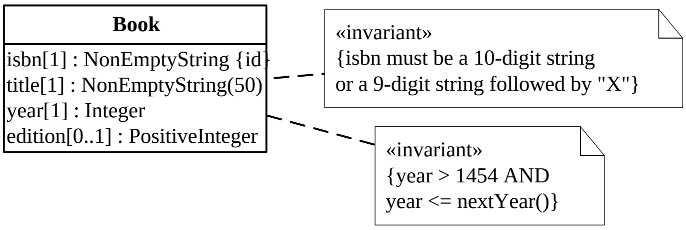
3. Use JavaScript as your back-end application programming language (such as with NodeJS), then you can code your validation functions in your JavaScript model classes on the back-end and execute them both before committing changes on the back-end and on user input and form submission in the UI on the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

The simplest, and most responsive, solution is the third one, using only JavaScript both for the back-end and front-end components of a web app.

5. Adding Constraints to a Design Model

We again consider the book data management problem that was considered in Part 1 of this tutorial. But now we also consider the ***data integrity rules*** (or '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 1.2 below.

Figure 1.2. A design model defining the object type `Book` with 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 calendar 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 coded in the model class within property-specific *check* functions.

The meaning of the design model can be illustrated by a sample data population respecting all constraints:

Table 1.1. Sample data for `Book`

ISBN	Title	Year	Edition
006251587X	Weaving the Web	2000	3
0465026567	Gödel, Escher, Bach	1999	2
0465030793	I Am A Strange Loop	2008	

6. Summary

1. Constraints are logical conditions on the data of an app. The simplest, and most important, types of constraints are property constraints and object-level constraints.

2. Constraints should be defined in the model classes of an MVC app, since they are part of their business semantics.

3. Constraints should be checked in various places of an MVC app: in the UI/view code, in model classes, and possibly in the database.

4. Software applications that include CRUD data management need to perform two kinds of bi-directional object-to-string type conversions:

    a. Between the model and the UI: converting model object property values to UI widget values, and, the other way around, converting input widget values to property values. Typically, widgets are form fields that have string values.

    b. Between the model and the datastore: converting model objects to storage data sets (called serialization), and, the other way around, converting storage data sets to model objects (called de-serialization). This involves converting property values to storage data values, and, the other way around, converting storage data values to property values. Typically, datastores are either JavaScript's local storage or IndexedDB, or SQL databases, and objects have to be mapped to some form of table rows. In the case of an SQL database, this is called "Object-Relational Mapping" (ORM).

5. Do not perform any string-to-property-value conversion in the UI code. Rather, this is the business of the model code.

6. For being able to observe how an app works, or, if it does not work, where it fails, it is essential to log all critical application events, such as data retrieval, save and delete events, at least in the JavaScript console.

7. Responsive validation means that the user, while typing, gets immediate validation feedback on each input (keystroke), and when requesting to save the new data.

Chapter 2. Implementing Constraint Validation in a Plain JS Web App

1. Introduction

The minimal JavaScript front-end web application that we have discussed in the Minimal App Tutorial 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 chapter, 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 show how to perform responsive validation with the HTML5 form validation API. Since using the new HTML5 `input` field types and validation attributes (like `required`, `min`, `max` and `pattern`) implies defining constraints in the UI, they are not really useful in a best-practice approach where constraints are only checked, but not defined, in the UI.

Consequently, we will not use the new HTML5 features for defining constraints in the UI, but only use two methods of the HTML5 form validation API:

1. `setCustomValidity`, which allows to mark a form field as either valid or invalid by assigning either an empty string or a non-empty (constraint violation) message string;
2. `checkValidity`, which is invoked on a form before user input data is committed or saved (for instance with a form submission); it tests, if all fields have a valid value.

In the case of two special kinds of attributes, having *calendar dates* or *colors* as values, it is desirable to use the new UI widgets defined by HTML5 for picking a date or picking a color (with corresponding `input` field types).

2. New Issues

Compared to the Minimal App discussed in the Minimal App Tutorial we have to deal with a number of new issues:

1. In the *model* code we have to add for every property of a class
   a. a ***check*** function that can be invoked for validating the constraints defined for the property, and
   b. a ***setter*** method that invokes the check function and is to be used for setting the value of the property.
2. In the *user interface ("view")* code we have to take care of
   a. ***responsive validation*** on user input for providing immediate feedback to the user,
   b. 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 an entity table such as `Book.instances`. This method is used in the `setupUserInterface` method of both the `updateBook` and the `deleteBook` use cases.

3. As a namespace approach (for avoiding name conflicts), we will now use ES6 **modules**, instead of a global namespace object with subnamespace objects, like `pl = {m:{}, v:{}, c:{}}`.

Checking the constraints in the user interface (UI) 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 UI, because this could be circumvented in a distributed web application where the UI runs in the web browser of a front-end device while the application's data is managed by a back-end component on a remote web server. Consequently, we need multiple constraint validation, first in the UI *on input* (or *on change*) and *on form submission*, and subsequently in the model layer before saving/sending data to the persistent data store. And in an application based on a DBMS we may also use a third round of validation before persistent storage by using the validation mechanisms of the DBMS. This is a must, when the application's database is shared with other apps.

Our proposed solution to this *multiple validation problem* is to keep the constraint validation code in special *check* functions in the model classes and invoke these functions both in the UI 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 *referential integrity* constraints (and other relationship constraints) may also be violated through a *delete* operation, but in our single-class example we don't have to consider this.

3. Using ES6 Modules

Normal modules are library code files that explicitly **export** those (variable, function and class) names that other modules can use (as implicitly frozen like `const` declarations). A module that is to use items from another module needs to explicitly **import** them from that other module using import statements. It is recommended that all JS module files use the file extension "mjs" for indicating that they are different from classical script files.

Web pages can load module files, possibly along with classical script files, with the help of a special type of `script` element.

The meaning of ES6 modules is based on the following principles:

1. A JS library file can be turned into a module by using "export" for all library items. Other modules can "import" its items.

2. Any ordinary script file that is to use one or more items from a module has itself to be turned into a module ("only modules can use modules"). Since it doesn't export anything, such a module could also be called an "import module".

3. All modules, no matter if they export anything or are just "import modules", are separated from the global scope in the following sense: they have read access to items from the global scope such as DOM objects (like `document`) or other global objects (like `Array`), but they cannot create any names (including objects and functions) in the global scope. This implies, for instance, that a JS function defined in a module cannot be assigned to an `onclick` event handler attribute in an HTML file..

Using modules implies that we can no longer use the global scope for the names of functions/classes, which is a restriction that is considered a good practice in software engineering.

An example of a normal (library) module file is `util.mjs` with the following code:

```
function isNonEmptyString(x) {
  return typeof(x) === "string" && x.trim() !== "";
}
```

```
...
export { isNonEmptyString, ... };
```

An example of a module that imports certain items from other modules and then uses them in its own code, and also exports some of its own items is the model class file `Book.mjs` with the following import/export statements:

```
import { isNonEmptyString, ... } from "../../lib/util.mjs";
import { NoConstraintViolation, MandatoryValueConstraintViolation, ... }
    from "../../lib/errorTypes.mjs";
export default function Book( slots) {...}
```

Since this module only exports one class (*Book*), a default export us used, allowing simplified imports.

An example of a module that does not export anything, but only imports certain items, is the view code file `createBook.mjs` with the following import statements:

```
import Book from "../../src/m/Book.mjs";
import { fillSelectWithOptions } from "../../lib/util.mjs";
...
```

An HTML page (here: `createBook.html`) can load such a module with a special type of `script` element:

```
<script src="src/v/createBook.mjs" type="module"></script>
```

Notice that this `script` element's `type` attribute is set to "module".

Alternatively, the code of such a module can be embedded in the HTML page like so:

```
<script type="module">
 import Book from "./src/m/Book.mjs";
 const clearButton = document.getElementById("clearData");
 // Set event handler for the button "clearData"
 clearButton.addEventListener("click", function () {Book.clearData();});
</script>.
```

4. Make a JavaScript Class Model

Using the information design model shown in Figure 1.2 above as the starting point, we make a *JavaScript* class 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 attribute, such as `Book::isbn`, two check operations are needed:

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

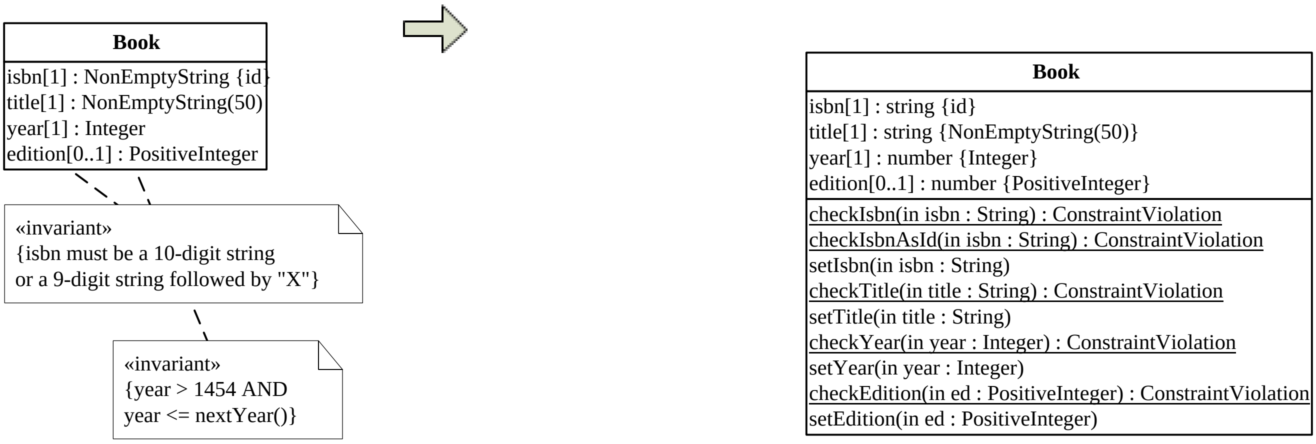
    b. An extended check operation, such as `checkIsbnAsId`, for checking, in addition to the basic constraints, the *mandatory value* and *uniqueness* constraints that are required for a standard identifier attribute.

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.

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

Figure 2.1. From an information design model to a JS class model



Essentially, the JS class 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.

5. Set up the Folder Structure

The folder structure of the validation app extends the folder structure of the minimal app by adding two subfolders:

1. a `css` folder containing CSS style files for styling the user interface pages of the app;

2. a `lib` folder containing the code library files `errorTypes.mjs` and `util.mjs`.

Thus, we get the following folder structure:

```
publicLibrary
  css
    main.css
    normalize.css
  lib
    errorTypes.mjs
    util.mjs
  src
```

```
    m
    v
  index.html
```

Notice that the `src` folder only contains two subfolders `m` and `v` (for *model* and *view*), as we have dropped the `c` folder since the minimal app's *controller* code for defining namespace objects is no longer needed due to the use of ES6 modules.

We discuss the contents of the added library files in the following subsections.

5.1. Provide utility functions and error classes in library files

We add two library files (in the form of ES6 *modules*) 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. `errorTypes.js` defines classes for error (or exception) types corresponding to the basic types of property constraints discussed above: StringLengthConstraintViolation, MandatoryValueConstraintViolation, RangeConstraintViolation, IntervalConstraintViolation, PatternConstraintViolation, UniquenessConstraintViolation. In addition, a class *NoConstraintViolation* is defined for being able to return a validation result object in the case of no constraint violation.

5.2. Create a start page

The start page `index.html` takes care of loading CSS style files with the help of the following two `link` elements:

```
<link rel="stylesheet" href="css/normalize.css">
<link rel="stylesheet" href="css/main.css">
```

The first CSS file (normalize.css) is a widely used collection of style normalization rules making browsers render all HTML elements more consistently. The second file (main.css) contains the specific styles of the app's user interface (UI) pages.

Since the app's start page does not provide much UI functionality, but only a few navigation links and two buttons, only a few lines of code are needed for setting up the buttons' event listeners. This is taken care of in an embedded `script` element of type `module`:

```
<script type="module">
 import Book from "./src/m/Book.mjs";
 const clearButton = document.getElementById("clearData"),
       generateTestDataButtons = document.querySelectorAll("button.generateTestData");
 // Set event handler for the button "clearData"
 clearButton.addEventListener("click", function () {Book.clearData();});
 generateTestDataButtons.forEach( function (el) {
     el.addEventListener("click", function () {Book.generateTestData();})
 });
</script>.
```

Notice how the Book class is loaded by importing the module `Book.js` from the `src/m` folder.

6. Write the Model Code

The JS class model shown on the right hand side in Figure 2.1 can be coded 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.

6.1. Summary

1. Code the model class as a JavaScript constructor function.
2. ***Code 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 JS class model, are properly coded in the check functions.
3. ***Code 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. Code the add and remove operations, if there are any, as instance-level methods.

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

6.2. Code the model class as a constructor function

The class `Book` is coded as a corresponding *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)
  // assign properties only if the constructor is invoked with an argument
  if (arguments.length > 0) {
    this.setIsbn( slots.isbn);
    this.setTitle( slots.title);
    this.setYear( slots.year);
    // optional property
    if (slots.edition) this.setEdition( slots.edition);
  }
};
```

In the constructor body, we first assign default values to the class properties. These values will be used when the constructor 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 datatypes of the information design model to the corresponding implicit JavaScript datatypes according to the following table.

Table 2.1. Datatype mapping

Platform-independent datatype	JavaScript datatype	SQL
String	string	CHAR(*n*) or VARCHAR(*n*)

Platform-independent datatype	JavaScript datatype	SQL
Integer	number (int)	INTEGER
Decimal	number (float)	REAL, DOUBLE PRECISION or DECIMAL(*p*,*s*)
Boolean	boolean	BOOLEAN
Date	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 app in the form of an entity table:

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

6.3. Code the property checks

Code the property check functions in the form of class-level ('static') methods. In JavaScript, this means to define them as method slots of the constructor, as in `Book.checkIsbn` (recall that a constructor is a JS object, since in JavaScript, functions are objects, and as an object, it can have slots).

Take care that all constraints of a property as specified in the class model are properly coded in its check function. This concerns, in particular, the *mandatory value* and *uniqueness* constraints implied by the *standard identifier* declaration (with `{id}`), 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 5.1 and defined in the file `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;
};
```

We assume that all check functions and setters can deal both with proper data values (that are of the attribute's range type) and also with string values that are supposed to represent proper data values, but have not yet been converted to the attribute's range type. We take this approach for avoiding datatype conversions in the user interface ("view") code. Notice that all data entered by a user in an HTML form field is of type String and must be converted (or de-serialized) before its validity can be checked and it can be assigned to the corresponding property. It is preferable to perform these type conversions in the model code, and not in the user interface code..

For instance, in our example app, we have the integer-valued attribute `year`. When the user has entered a value for this attribute in a corresponding form field, in the *Create* or *Update* user interface, the form field holds a string value. This value is passed to the `Book.add` or `Book.update` method, which invokes the `setYear` and `checkYear` methods. Only after being validated, this string value is converted to an integer and assigned to the `year` attribute.

6.4. Code the property setters

Code 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 coded in the following way:

```
Book.prototype.setIsbn = function (id) {
  const 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`).

6.5. Add a serialization function

It is helpful to have an object serialization function tailored to the structure of an object (as defined by its 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 () {
  var bookStr = `Book{ ISBN: ${this.isbn}, title: ${this.title}, year: ${this.year}`;
  if (this.edition) bookStr += `, edition: ${this.edition}`;
  return bookStr;
};
```

6.6. 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()` serialization function, we also need to define the following data management operations as class-level methods of the model class:

1. `Book.convertRec2Obj` and `Book.retrieveAll` 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 and adding it to the collection of all Book instances.
4. `Book.update` for updating an existing Book instance.
5. `Book.destroy` for deleting a Book instance.
6. `Book.createTestData` for creating a few sample book records to be used as test data.
7. `Book.clearData` for clearing the book data store.

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 `Book` procedures `convertRec2Obj`, `add`, `update` and `createTestData`;
2. we create more informative status and error log messages for better observing what's going on; and
3. we can use the `toString()` function for serializing an object in status and error messages.

Notice that for the change operations `add` (create) and `update`, we need to implement an all-or-nothing policy: whenever 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. Otherwise, the new book object is added to `Book.instances` and a status message is created, as shown in the following program listing:

```
Book.add = function (slots) {
  var book = null;
  try {
    book = new Book( slots);
  } catch (e) {
```

```
    console.log( `${e.constructor.name}: ${e.message}`);
    book = null;
  }
  if (book) {
    Book.instances[book.isbn] = book;
    console.log( `${book.toString()} created!`);
  }
};
```

When an object of a model class is to be updated, we first create a clone of it for being able to restore it if the update attempt fails. In the object update attempt, we only assign those properties of the object the value of which has changed, and we report this in a status log.

Normally, all properties defined by a model class, except the standard identifier attribute, can be updated. It is, however, possible to also allow updating the standard identifier attribute. This requires special care for making sure that all references to the given object via its old standard identifier are updated as well.

When a constraint violation is detected in one of the setters invoked in `Book.update`, the object update attempt fails, and instead the error message of the constraint violation object thrown by the setter and caught in the `update` method is shown, and the previous state of the object is restored. Otherwise, a status message is created, as shown in the following program listing:

```
Book.update = function (slots) {
  var noConstraintViolated = true,
      updatedProperties = [];
  const book = Book.instances[slots.isbn],
        objectBeforeUpdate = 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 && slots.edition !== book.edition) {
      // slots.edition has a non-empty value that is new
      book.setEdition( slots.edition);
      updatedProperties.push("edition");
    } else if (!slots.edition && book.edition !== undefined) {
      // slots.edition has an empty value that is new
      delete book.edition;  // unset the property "edition"
      updatedProperties.push("edition");
    }
  } catch (e) {...}
  ...
};
```

Notice that optional properties, like `edition`, need to be treated in a special way. If the user doesn't enter any value for them in a *Create* or *Update* user interface, the form field's value is the empty string `""`. In the case of an optional property, this means that the property is not assigned a value in the *add* use case, or that it is *unset* if it has had a value in the *update* use case. This is different from the case of a mandatory property, where the empty string value obtained from an empty form field may or may not be an admissible value.

If there is a constraint violation exception, an error message is written to the log and the object concerned is reset to its previous state:

```
Book.update = function (slots) {
  ...
  try {
    ...
  } catch (e) {
    console.log( e.constructor.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}!`);
    }
  }
};
```

7. Write the View Code

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 `retrieveAndListAllBooks.html` or `createBook.html` in the app folder.

The start page `index.html` has been discussed above in Section 5.2. It sets up two buttons for clearing the app's database by invoking the procedure `Book.clearData()` and for creating sample data by invoking the procedure `Book.createTestData()` from the buttons' `click` event listeners.

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 a use-case-specific view code file `src/v/`*useCase*`.js`.

For setting up the user interfaces of the data management use cases, we have to distinguish the case of "Retrieve/List All" 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 "Retrieve/List All" we only have to render a table displaying all books, as in the case of the Minimal App discussed in Part 1 of this tutorial.

For the *Create*, *Update* and *Delete* use cases, we need to (1) import the `Book` class, (2) define variables for accessing the UI form element and the save/delete button, and (3) load all book records, like so:

```
import Book from "../m/Book.mjs";
const formEl = document.forms["Book"],
      saveButton = formEl["commit"];
Book.retrieveAll();  // load all book records
```

...and then add event listeners for:

1. responsive validation on form field `input` events,
2. handling the event when the user clicks (or pushes) the *save* (or *delete*) button,
3. making sure the main memory data is saved when a `beforeunload` event occurs, that is, when the browser window/tab is closed.

7.1. Set up the user interface for *Create Book*

For the use case *Create*, we obtain the following code (in `v/createBook.mjs`) for adding event listeners for *responsive validation*:

```
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);
});
```

Notice that for each 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).

In addition to the event handlers for responsive constraint validation, we need two more event handlers: one for validation on form data submission and one for the event when the browser window (or tab) is closed.

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. In the form data submission event handler, the property checks are performed again (with the help of `setCustomValidity`), as shown in the following program listing:

```
saveButton.addEventListener("click", function () {
  const slots = { isbn: formEl.isbn.value,
          title: formEl.title.value,
          year: formEl.year.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);
  if (formEl.edition.value) {
    slots.edition = formEl.edition.value;
    formEl.edition.setCustomValidity(
        Book.checkEdition( slots.edition).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 event handler has been executed on an invalid form, the browser takes control and tests if the predefined 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).

Since the *Save* button has the type "submit", clicking it creates a *submit* event. For suppressing the browser's built-in *submit* event processing, we invoke the DOM operation `preventDefault` in a *submit* event handler like so:

```
formEl.addEventListener("submit", function (e) {
  e.preventDefault();
  formEl.reset();
});
```

Finally, still in the module `v/createBook.mjs`, we set a handler for the event when the browser window (or tab) is closed, taking care to save all data to persistent storage:

```
window.addEventListener("beforeunload", Book.saveAll);
```

7.2. Set up the user interface for *Update Book*

In the UI of the use case *Update*, which is handled in `v/updateBook.mjs`, we do not have an `input`, but rather an `output` field for the standard identifier attribute `isbn`, since it is not supposed to be modifiable. Consequently, we don't need to validate any user input for it. However, we need to set up a selection list (in the form of an HTML `select` element) allowing the user to select a book in the first step, before its data can be modified. This requires to add a `change` event listener on the `select` element such that the fields of the HTML form can be filled with the data of the selected object, as taken care of by the following code:

```
const formEl = document.forms["Book"],
      saveButton = formEl["commit"],
      selectBookEl = formEl["selectBook"];
// set up the book selection list
fillSelectWithOptions( Book.instances, selectBookEl, "isbn", "title");
// when a book is selected, populate the form with its data
selectBookEl.addEventListener("change", function () {
  const bookKey = selectBookEl.value;
  if (bookKey) {  // set form fields
    const book = Book.instances[bookKey];
```

```
    ["isbn","title","year","edition"].forEach( function (p) {
      formEl[p].value = book[p] ? book[p] : "";
      // delete previous custom validation error message
      formEl[p].setCustomValidity("");
    });
  } else {
    formEl.reset();
  }
});
```

There is no need to set up responsive validation for the standard identifier attribute `isbn`, but for all other form fields, as shown above for the *Create* use case.

The logic of `v/deleteBook.mjs` for the *Delete* use case is similar. We only need to take care that the object to be deleted can be selected by providing a selection list, like in the *Update* use case. No validation is needed for the *Delete* use case.

You can run the validation app from our server or download the code as a ZIP archive file.

8. Possible Variations and Extensions

8.1. Adding an object-level validation function

When object-level validation (across two or more properties) is required for a model class, we can add a custom validation function `validate` to it, such that object-level validation can be performed before save by invoking `validate` on the object concerned. For instance, for expressing the constraint defined in the class model shown in Figure 1.1, we define the following validation function:

```
Author.prototype.validate = function () {
  if (this.dateOfDeath && this.dateOfDeath < this.dateOfBirth) {
    throw new ConstraintViolation(
          "The dateOfDeath must be after the dateOfBirth!");
  }
};
```

When a `validate` function has been defined for a model class, it can be invoked in the create and update methods. For instance,

```
Author.add = function (slots) {
  var author = null;
  try {
    author = new Author( slots);
    author.validate();
  } catch (e) {
    console.log( e.constructor.name +": "+ e.message);
  }
};
```

8.2. Using implicit JS setters

Since ES5, JavaScript has its own form of setters, which are implicit and allow having the same semantics as explicit setter methods, but with the simple syntax of direct access. In addition to having the advantage of a simpler syntax, implicit JS setters are also safer than explicit setters because they decrease the likelihood of a programmer circumventing a setter by using a direct property assignment when instead a setter should be used. In other OOP languages, like Java, this is prevented by declaring properties to be 'private'. But JavaScript does not have this option.

The following code defines implicit setter and getter methods for the property `title`:

```
Object.defineProperty( Book.prototype, "title", {
  set: function(t) {
    var validationResult = Book.checkTitle( t);
    if (validationResult instanceof NoConstraintViolation) {
      this._title = t;
    } else {
      throw validationResult;
    }
  },
  get: function() {
    return this._title;
  }
});
```

Notice that, also in the constructor definition, the internal property `_title`, used for storing the property value, is not used for setting/getting it, but rather the virtual property `title`:

```
Book = function (slots) {
  this.learnUnitNo = 0;
  this.title = "";
  if (arguments.length > 0) {
    this.learnUnitNo = slots.learnUnitNo;
    this.title = slots.title;
    // optional property
    if (slots.subjectArea) this.subjectArea = slots.subjectArea;
  }
});
```

We will start using implicit setter and getter methods, along with ES2015 class definitions, in our 3rd tutorial on enumeration attributes.

9. Points of Attention

9.1. Boilerplate code

An issue with the do-it-yourself code of this example app is the *boilerplate code* needed

1. per model class for the storage management methods `add`, `update`, `destroy`, etc.;
2. per model class and property for getters, setters and validation checks.

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 our mODELcLASSjs tutorial, we present an approach how to put these methods in a generic form in a meta-class, such that they can be reused in all model classes of an app.

9.2. Configuring the UI for preventing invalid user input

Many of the new HTML5 input field types (like `number`, `tel`, `email`, `url`, `date` (together with `datetime-local`, `time` and `month`) or `color`) are intended to allow web browsers rendering corresponding `input` elements in the form of UI widgets (like a *date* picker or a *color* picker) that limit the user's input options such that only valid input is possible. In terms of usability, it's preferable to prevent users from entering invalid data instead of allowing to enter it and only then checking its validity and reporting errors.

Input fields for decimal number input should not be defined like

```
<input type="number" name="..." />
```

but rather like

```
<input type="text" inputmode="decimal" name="..." />
```

because this provides for a better user experience on mobile phones.

9.3. Improving the user experience by showing helpful auto-complete suggestions

While browsers have heuristics for showing auto-complete suggestions, you cannot rely on them, and should better add the `autocomplete` attribute with a suitable value. For instance, in iOS Safari, setting the input type to "tel" does only show auto-complete suggestions if `autocomplete="tel"` is added.

HTML5 defines more than 50 possible values for the `autocomplete` attribute. So, you have to make an effort looking up the one that best suits your purposes.

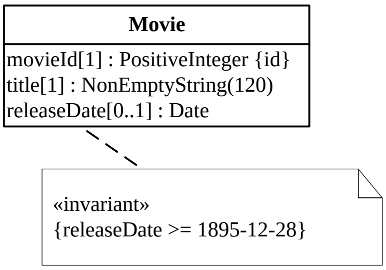
You can also create your own custom auto-complete functionality with `datalist`.

10. Practice Project

The purpose of the app to be built is managing information about movies. Like in the book data management app discussed in the tutorial, you can make the simplifying assumption that all the data can be kept in main memory. Persistent data storage is implemented with JavaScript's Local Storage API.

The app deals with just one object type: `Movie`, as depicted in Figure 2.2 below. In the subsequent parts of the tutorial, you will extend this simple app by adding enumeration-valued attributes, as well as actors and directors as further model classes, and the associations between them.

Figure 2.2. The object type `Movie` defined with several constraints



In this model, the following constraints have been expressed:

1. Due to the fact that the `movieId` attribute is declared to be the *standard identifier* of `Movie`, as expressed by the property annotation `{id}` shown after the property range, it is ***mandatory*** and ***unique***.
2. 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 120 characters.
3. The `releaseDate` attribute has an ***interval constraint***: it must be greater than or equal to 1895-12-28.

Notice that the `releaseDate` 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 `PositiveInteger` to `movieId`, `NonEmptyString` to `title`, and `Date` to `releaseDate`. In our plain JavaScript approach, all these property constraints are coded in the model class within property-specific *check* functions.

Following the tutorial, you have to take care of

1. adding for every property a ***check*** function that validates 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,
2. ***performing validation*** before any data is saved in the `Movie.add` and `Movie.update` methods.

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

1. styling the user interface with CSS rules (by integrating a CSS library such as Yahoo's Pure CSS),
2. ***validation on user input*** for providing immediate feedback to the user,
3. ***validation on form submission*** for preventing the submission of invalid data.

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

Table 2.2. Sample data

Movie ID	Title	Release date
1	Pulp Fiction	1994-05-12
2	Star Wars	1977-05-25
3	Casablanca	1943-01-23
4	The Godfather	1972-03-15

In this project, and in all further projects, 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 and is checked with JSHint (http://www.jshint.com).

If you have any questions about how to carry out this project, you can ask them on our discussion forum.