JavaScript Front-End Web App Tutorial Part 1: Building a Minimal App in Seven Steps

Learn how to build a front-end web application with minimal effort, using plain JavaScript and the LocalStorage API

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

JavaScript Front-End Web App Tutorial Part 1: Building a Minimal App in Seven Steps: Learn how to build a front-end web application with minimal effort, using plain JavaScript and the LocalStorage API

by Gerd Wagner

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

This tutorial is also available in the following formats: PDF. You may run the example app from our server, or download it 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 1 of our series of six tutorials about model-based development of front-end web applications with plain JavaScript. It shows how to build such an app with minimal effort, not using any (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.

This tutorial provides theoretically underpinned and example-based learning materials and supports ***learning by doing it yourself***.

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

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

- Part 2: Handling **constraint validation**.
- 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, 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: Handling **subtype** (inheritance) relationships between object types.

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

If you are already familiar with HTML, XML and JavaScript, you may skip this chapter and immediately start developing a minimal web application by going to the next chapter.

1. The World Wide Web (WWW)

After the Internet had been established in the 1980'ies, Tim Berners-Lee developed the idea and the first implementation of the WWW in 1989 at the European research institution CERN in Geneva, Switzerland. The WWW (or, simply, "the Web") is based on the Internet technologies TCP/IP (the *Internet Protocol*) and DNS (the *Domain Name System*). Initially, the Web consisted of

1. the *Hypertext Transfer Protocol (HTTP)*,
2. the *Hypertext Markup Language (HTML)*, and
3. web server programs, acting as HTTP servers, as well as web 'user agents' (such as browsers), acting as HTTP clients.

Later, further important technology components have been added to this set of basic web technologies:

- the page/document style language *Cascading Style Sheets (CSS)* in 1995,
- the web programming language *JavaScript* in 1995,
- the *Extensible Markup Language (XML)*, as the basis of web formats like SVG and RDF/XML, in 1998,
- the XML-based *Scalable Vector Graphics (SVG)* format in 2001,
- the *Resource Description Framework (RDF)* for knowledge representation on the Web in 2004.

2. HTML and XML

HTML allows to mark up (or describe) the structure of a human-readable web document or web user interface, while XML allows to mark up the structure of all kinds of documents, data files and messages, whether they are human-readable or not. XML can also be used as the basis for defining a version of HTML that is called *XHTML*.

2.1. XML documents

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

- **document formats**, such as *XHTML5*, the *Scalable Vector Graphics* (SVG) format or the *DocBook* format,
- **data interchange file formats**, such as the *Mathematical Markup Language* (MathML) or the *Universal Business Language* (UBL),

- **message formats**, such as the web service message format SOAP

2.2. Unicode and UTF-8

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

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

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

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

2.3. XML namespaces

Generally, namespaces help to avoid name conflicts. They allow to reuse the same (local) name in different namespace contexts. Many computational languages have some form of namespace concept, for instance, Java and PHP.

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

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

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

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

The following example shows an SVG namespace declaration for an `svg` element embedded in an HTML document:

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

```
</html>
```

2.4. Correct XML documents

XML defines two syntactic correctness criteria. An XML document must be *well-formed*, and if it is based on a grammar (or schema), then it must also be *valid* with respect to that grammar, or, in other words, satisfy all rules of the grammar.

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

1. There must be exactly one root element.
2. Each element has a start tag and an end tag; however, empty elements can be closed as `<phone/>` instead of `<phone></phone>`.
3. Tags don't overlap. For instance, we cannot have

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

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

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

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

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

2.5. The evolution of HTML

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

- 1997: **HTML 4** as an SGML-based language,
- 2000: **XHTML 1** as an XML-based clean-up of HTML 4,
- 2014: ***(X)HTML 5*** in cooperation (and competition) with the WHAT working group supported by browser vendors.

As the inventor of the Web, Tim Berners-Lee developed a first version of HTML in 1990. A few years later, in 1995, Tim Berners-Lee and Dan Connolly wrote the *HTML 2* standard, which captured the common use of HTML elements at that time. In the following years, HTML has been used and gradually extended by a growing community of early WWW adopters. This evolution of HTML, which has led to a messy set of elements and attributes (called "tag soup"), has been mainly controlled by browser vendors and their competition with each other. The development of XHTML in 2000 was an attempt by the W3C to clean up this mess, but it neglected to advance HTML's functionality towards a richer user interface, which was the focus of the WHAT working group led by Ian Hickson who can be considered as the mastermind and main author of HTML 5 and many of its accompanying JavaScript APIs that made HTML fit for mobile apps.

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

- device independence,
- accessibility, and
- usability.

We adopt the symbolic equation

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

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

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

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
  <meta charset="UTF-8" />
  <title>XHTML5 Template Example</title>
  <meta name="viewport" content="width=device-width, initial-scale=1"/>
</head>
<body>
  <h1>XHTML5 Template Example</h1>
  <section><h2>First Section Title</h2>
   ...
  </section>
</body>
</html>
```

Notice that in line 1, the HTML5 document type is declared, such that browsers are instructed to use the HTML5 document object model (DOM). In the `html` start tag in line 2, using the default namespace declaration attribute `xmlns`, the XHTML namespace URI `http://www.w3.org/1999/xhtml` is declared as the default namespace for making sure that browsers, and other tools, understand that all non-qualified element names like `html`, `head`, `body`, etc. are from the XHTML namespace.

Also in the `html` start tag, we set the (default) language for the text content of all elements (here to "en" standing for English) using both the `xml:lang` attribute and the HTML `lang` attribute. This attribute duplication is a small price to pay for having a hybrid document that can be processed both by HTML and by XML tools.

Finally, in line 4, using an (empty) `meta` element with a `charset` attribute, we set the HTML document's character encoding to UTF-8, which is also the default for XML documents.

2.6. HTML forms

For user-interactive web applications, the web browser needs to render a user interface (UI). The traditional metaphor for a software application's UI is that of a ***form***. The special elements for data input, data output and user actions are called ***form controls*** or *UI* ***widgets***. In HTML, a `form` element is a section of a web page consisting of block elements that contain form controls and *labels* on those controls.

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

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

Labels are associated with a control by including the control as a child element within a `label` element (*implicit* labels), or by giving the control an `id` value and referencing this ID in the `for` attribute of the `label` element (*explicit* labels).

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

1. *single line input fields* created with an `<input name="..." />` element,
2. *single line output fields* created with an `<output name="..." />` element,
3. *push buttons* created with a `<button type="button">...</button>` element, and
4. *dropdown selection lists* created with a `select` element of the following form:

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

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

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

In an HTML-form-based data management user interface, we have a correspondence between the different kinds of properties defined in the model classes of an app and the form controls used for the input and output of their values. We have to distinguish between various kinds of **model class attributes**, which are mapped to various kinds of **form fields**. This mapping is also called **data binding**.

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

3. Styling Web Documents and User Interfaces with CSS

While HTML is used for defining the content structure of a web document or a web user interface, the ***Cascading Style Sheets (CSS)*** language is used for defining the ***presentation style*** of web pages, which means that you use it for telling the browser how you want your HTML (or XML) rendered: using which layout of content elements, which fonts and text styles, which colors, which backgrounds, and which animations. Normally, these settings are made in a separate CSS file that is associated with an HTML file via a special `link` element in the HTML's `head`.

> A first sketch of CSS was proposed in October 1994 by Håkon W. Lie who later became the CTO of the browser vendor Opera. While the official CSS1 standard dates back to December 1996, "most of it was hammered out on a whiteboard in Sophia-Antipolis" by Håkon W. Lie together with Bert Bos in July 1995 (as he explains in an interview).

CSS is based on a form of rules that consist of *selectors*, which select the document element(s) to which a rule applies, and a list of *property-value pairs* that define the styling of the selected element(s) with the help of CSS properties such as `font-size` or `color`. There are two fundamental mechanisms for computing the CSS property values for any page element as a result of applying the given set of CSS rules: *inheritance* and *the cascade*.

The basic element of a CSS layout is a rectangle, also called "box", with an inner content area, an optional border, an optional padding (between content and border) and an optional margin around the border. This structure is defined by the CSS *box model*.

We will not go deeper into CSS in this tutorial, since our focus here is on the logic and functionality of an app, and not so much on its beauty.

4. JavaScript - "the assembly language of the Web"

> JavaScript was developed in 10 days in May 1995 by Brendan Eich, then working at Netscape, as the HTML scripting language for their browser *Navigator 2* (more about history). Brendan Eich said (at the O'Reilly Fluent conference in San Francisco in April 2015): "I did JavaScript in such a hurry, I never dreamed it would become the assembly language for the Web".

JavaScript is a dynamic functional object-oriented programming language that can be used for

1. Enriching a web page by
   - generating browser-specific HTML content or CSS styling,
   - inserting dynamic HTML content,
   - producing special audio-visual effects (animations).

2. Enriching a web user interface by
   - implementing advanced user interface components,
   - validating user input on the client side,
   - automatically pre-filling certain form fields.
3. Implementing a front-end web application with local or remote data storage, as described in the book Building Front-End Web Apps with Plain JavaScript.
4. Implementing a front-end component for a distributed web application with remote data storage managed by a back-end component, which is a server-side program that is traditionally written in a server-side language such as PHP, Java or C#, but can nowadays also be written in JavaScript with NodeJS.
5. Implementing a complete distributed web application where both the front-end and the back-end components are JavaScript programs.

The version of JavaScript that is currently fully supported by modern web browsers is called "ECMAScript 2015", or simply "ES2015", but the following versions, (ES2016, ES2017, ...), are already partially supported by current browsers and back-end JS environments.

4.1. JavaScript as an object-oriented language

JavaScript is *object-oriented*, but in a different way than classical OO programming languages such as Java and C++. In JavaScript, classes, unlike objects and functions, have not been first-class citizens until ES2015 has introduced a `class` syntax. Before ES2015, classes had to be defined by following a code pattern in the form of special JS objects: either as *constructor* functions or as *factory* objects. Notice that when using (the syntactic sugar of) ES2015 `class` declarations, what is really defined internally, is still a constructor function.

However, objects can also be created without instantiating a class, in which case they are *untyped*, and properties as well as methods can be defined for specific objects independently of any class definition. At run time, properties and methods can be added to, or removed from, any object and class. This dynamism of JavaScript allows powerful forms of *meta-programming*, such as defining your own concepts of classes and enumerations (and other special datatypes).

4.2. Further reading about JavaScript

Good open access books about JavaScript are

- Speaking JavaScript, by Dr. Axel Rauschmayer.
- Eloquent JavaScript, by Marijn Haverbeke.
- Building Front-End Web Apps with Plain JavaScript, by Gerd Wagner

5. Accessibility for Web Apps

The recommended approach to providing accessibility for web apps is defined by the *Accessible Rich Internet Applications* ***(ARIA)*** standard. As summarized by Bryan Garaventa in his article on different forms of accessibility, there are 3 main aspects of accessibility for interactive web technologies: 1) keyboard accessibility, 2) screen reader accessibility, and 3) cognitive accessibility.

Further reading on ARIA:

1. How browsers interact with screen readers, and where ARIA fits in the mix by Bryan Garaventa
2. The Accessibility Tree Training Guide by whatsock.com
3. The ARIA Role Conformance Matrices by whatsock.com
4. Mozilla's ARIA overview article
5. W3C's ARIA Authoring Practices

Chapter 2. More on JavaScript

1. JavaScript Basics

In this summary we try to take all important points of the classical JavaScript summary by Douglas Crockford into consideration.

1.1. Types and data literals in JavaScript

JavaScript has three primitive datatypes: `string`, `number` and `boolean`, and we can test if a variable `v` holds a value of such a type with the help of the JS operator `typeof` as, for instance, in `typeof v === "number"`.

There are five reference types: `Object`, `Array`, `Function`, `Date` and `RegExp`. Arrays, functions, dates and regular expressions are special types of objects, but, conceptually, dates and regular expressions are primitive data values, and happen to be implemented in the form of wrapper objects.

The types of variables, array elements, function parameters and return values are not declared and are normally not checked by JavaScript engines. Type conversion (casting) is performed automatically.

The value of a variable may be

- a *data value*: either a string, a number, or a boolean;
- an *object reference*: either referencing an ordinary object, or an array, function, date, or regular expression;
- the special data value `null`, which is typically used as a default value for initializing an object variable;
- the special data value `undefined`, which is the implicit initial value of all variables that have been declared, but not initialized.

A ***string*** is a sequence of Unicode characters. String literals, like "Hello world!", 'A3F0', or the empty string "", are enclosed in single or double quotes. Two string expressions can be concatenated with the + operator, and checked for equality with the triple equality operator:

```
if (firstName + lastName === "JamesBond") ...
```

The number of characters of a string can be obtained by applying the `length` attribute to a string:

```
console.log("Hello world!".length);  // 12
```

All ***numeric*** data values are represented in 64-bit floating point format with an optional exponent (like in the numeric data literal `3.1e10`). There is no explicit type distinction between integers and floating point numbers. If a numeric expression cannot be evaluated to a number, its value is set to `NaN` ("not a number"), which can be tested with the built-in predicate `isNaN(`*expr*`)`.

The built-in function, `Number.isInteger` allows testing if a number is an ***integer***. For making sure that a numeric value is an integer, or that a string representing a number is converted to an integer, one has to apply the built-in function `parseInt`. Similarly, a string representing a decimal number can be converted to this number with `parseFloat`. For converting a number `n` to a string, the best method is using `String(n)`.

There are two predefined ***Boolean*** data literals, `true` and `false`, and the Boolean operator symbols are the exclamation mark `!` for NOT, the double ampersand `&&` for AND, and the double bar `||` for OR. When a non-Boolean value is used in a condition, or as an operand of a Boolean expression, it is implicitly converted to a Boolean value according to the following rules. The empty string, the (numerical) data literal 0, as well as `undefined` and `null`, are mapped to `false`, and all other values are mapped to `true`. This conversion can be performed explicitly with the help of the double negation operation, like in the equality test `!!undefined === false`, which evaluates to `true`.

In addition to strings, numbers and Boolean values, also ***calendar dates*** and times are important types of primitive data values, although they are not implemented as primitive values, but in the form of wrapper objects instantiating `Date`. Notice that `Date` objects do, in fact, not really represent dates, but rather date-time instants represented internally as the number of milliseconds since 1 January, 1970 UTC. For converting the internal value of a `Date` object to a human-readable string, we have several options. The two most important options are using either the standard format of ISO date/time strings of the form "2015-01-27", or localized formats of date/time strings like "27.1.2015" (for simplicity, we have omitted the time part of the date/time strings in these examples). When `x instanceof Date`, then `x.toISOString()` provides the ISO date/time string, and `x.toLocaleDateString()` provides the localized date/time string. Given any date string `ds`, ISO or localized, `new Date(ds)` creates a corresponding date object.

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

Assigning an ***empty array literal***, as in `var a = []` is the same as, but more concise than and therefore preferred to, invoking the `Array()` constructor without arguments, as in `var a = new Array()`.

Assigning an ***empty object literal***, as in `var o = {}` is the same as, but more concise than and therefore preferred to, invoking the `Object()` constructor without arguments, as in `var o = new Object()`. Notice, however, that an empty object literal `{}` is not really an empty object, as it contains property slots and method slots inherited from `Object.prototype`. So, a truly empty object (without any slots) has to be created with `null` as prototype, like in `var emptyObject = Object.create(null)`.

A summary of type testing is provided in the following table:

Type	**Example values**	**Test if x of type**
string	"Hello world!", 'A3F0'	`typeof x === "string"`
boolean	true, false	`typeof x === "boolean"`
(floating point) number	-2.75, 0, 1, 1.0, 3.1e10	`typeof x === "number"`
integer	-2, 0, 1, 250	`Number.isInteger(x)`
Object	{}, {num:3, denom:4}, {isbn:"006251587X," title:"Weaving the Web"}, {"one":1, "two":2, "three":3}	excluding `null`: `x instanceof Object` including `null`: `typeof x === "object"`
Array	[], ["one"], [1,2,3], [1,"one", {}]	`Array.isArray(x)`
Function	`function () { return "one"+1;}`	`typeof x === "function"`
Date	`new Date("2015-01-27")`	`x instanceof Date`

Type	Example values	Test if x of type
RegExp	`/(\w+)\s(\w+)/`	`x instanceof RegExp`

A summary of type conversions is provided in the following table:

Type	Convert to string	Convert string to type
boolean	`String(x)`	`Boolean(y)`
(floating point) number	`String(x)`	`parseFloat(y)`
integer	`String(x)`	`parseInt(y)`
Object	`x.toString()` or `JSON.stringify(x)`	`JSON.parse(y)`
Array	`x.toString()` or `JSON.stringify(x)`	`y.split()` or `JSON.parse(y)`
Function	`x.toString()`	`new Function(y)`
Date	`x.toISOString()`	`new Date(y)`
RegExp	`x.toString()`	`new RegExp(y)`

1.2. Variable scope

In ES5, there have only been two kinds of scope for variables declared with `var`: the global scope (with `window` as the context object) and function scope, but ***no block scope***. Consequently, declaring a variable with `var` within a code block is confusing and should be avoided. For instance, although this is a frequently used pattern, even by experienced JavaScript programmers, it is a pitfall to declare the counter variable of a `for` loop in the loop, as in

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

Instead of obtaining a variable that is scoped to the block defined by the `for` loop, JavaScript is interpreting this code (by means of "hoisting" variable declarations) as:

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

Therefore all function-scoped variable declarations (with `var`) should be placed at the beginning of a function. When a variable is to be scoped to a code block, such as to a `for` loop, it has to be declared with the keyword `let`, as in the following example:

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

1.3. Frozen, or immutable, variables

Whenever a variable is supposed to be immutable (having a frozen value), it should be declared with the keyword `const`:

```
const pi = 3.14159;
```

It is generally recommended that variables be declared with `const` whenever it is clear that their values will never be changed. This helps catching errors and it allows the JS engine to optimize code execution.

1.4. Strict Mode

Starting from ES5, we can use strict mode for getting more runtime error checking. For instance, in strict mode, all variables must be declared. An assignment to an undeclared variable throws an exception.

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

```
'use strict';
```

It is generally recommended to use strict mode, except when code depends on libraries that are incompatible with strict mode.

1.5. Different kinds of objects

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

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

```
var person1 = { lastName:"Smith", firstName:"Tom"};
```

An empty object with no slots is created in the following way:

```
var o1 = Object.create( null);
```

Whenever the name in a slot is an admissible JavaScript identifier, the slot may be either a *property slot*, a *method slot* or a *key-value slot*. Otherwise, if the name is some other type of string (in particular when it contains any blank space), then the slot represents a *key-value slot*, which is a map element, as explained below.

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

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

   or

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

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

Object properties can be accessed in two ways:

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

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

2. Using a map notation:

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

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

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

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

2. A ***map*** (also called 'associative array', 'dictionary', 'hash map' or 'hash table' in other languages) supports look-ups of *values* based on *keys* like, for instance,

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

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

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

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

   Within the body of a method slot of an object, the special variable `this` refers to the object.

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

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

   A more advanced namespace mechanism is provided by **ES6 modules**, as explained in Part 2 of this tutorial,.

5. A ***typed object*** instantiates a class that is defined either by a JavaScript constructor function or by a factory object. See Section 1.10, “Defining and using classes”

1.6. Array lists

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

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

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

Because they are array lists, JS arrays can grow dynamically: it is possible to use indexes that are greater than the length of the array. For instance, after the array variable initialization above, the array held by the variable `a` has the length 3, but still we can assign further array elements, and may even create gaps, like in

```
a[3] = 4;
a[5] = 5;
```

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

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

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

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

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

For ***appending*** (all elements of) another array `b` to an array `a`, we `push` all the elements of `b` to `a` with the help of the spread operator `...`, like so:

```
a.push( ...b);
```

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

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

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

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

For ***looping*** over an array `a`, we have three options: either use a `for` (counter variable) loop, or the array looping method `forEach`. In any case, we can use a `for` loop:

```
for (let i=0; i < a.length; i++) {
  console.log( a[i]);
}
```

If performance doesn't matter, or if `a` is sufficiently small (say, it does not contain more than a few hundred elements), we can use the array looping method `forEach`, as in the following example, where the parameter `elem` iteratively assumes each element of the array `a` as its value:

```
a.forEach( function (elem) {
  console.log( elem);
})
```

If no counter variable is needed, however, the best option is using a `for-of` loop (introduced in ES6):

```
for (elem of a) {
  console.log( elem);
}
```

For ***cloning*** an array `a`, we can use the array function `slice` in the following way:

```
var clone = a.slice(0);
```

1.7. Maps

A map (also called 'hash map', 'associative array' or 'dictionary') provides a mapping from keys to their associated values. Traditionally, before the built-in `Map` object has been added to JS, maps have been implemented in the form of plain JS objects where the keys are string literals that may include blank spaces like in:

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

Alternatively, a proper map can be constructed with the help of the `Map` constructor:

```
var myTranslation = new Map([
    ["my house", "mein Haus"],
    ["my boat", "mein Boot"],
    ["my horse", "mein Pferd"]
])
```

A traditional map (as a plain JS object) is processed with the help of a special loop where we loop over all keys using the predefined function `Object.keys(m)`, which returns an array of all keys of a map `m`. For instance,

```
var keys = Object.keys( myTranslation);
for (let i=0; i < keys.length; i++) {
  let key = keys[i];
  console.log("The translation of " + key + " is " +
      myTranslation[key]);
}
```

Alternatively, if the `myTranslation` map is sufficiently small, we can use the `forEach` method:

```
Object.keys( myTranslation).forEach( function (key) {
  console.log("The translation of " + key + " is " +
      myTranslation[key]);
})
```

Notice that using the `forEach` method is more concise.

A proper map (i.e. a `Map` object) can be processed with the help of a for...of loop in one of the following ways:

```
// processing both keys and values
for (let [key, value] of myTranslation) {
  console.log("The translation of " + key + " is " + value);
}
// processing only keys
for (let key of myTranslation.keys()) {
  console.log("The translation of " + key + " is " +
      myTranslation.get( key));
}
// processing only values
for (let value of myTranslation.values()) {
  console.log( value)
}
```

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

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

For ***adding*** a new entry to a proper map, we use the `set` operation:

```
myTranslation.set("my car", "mein Auto");
```

For ***deleting*** an entry from a traditional map, we can use the predefined `delete` operator as in:

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

For ***deleting*** an entry from a proper map, we can use the `Map::delete` method as in:

```
myTranslation.delete("my boat");
```

For ***testing*** if a traditional map contains an entry for a certain key value, such as for testing if the translation map contains an entry for "my bike" we can check the following:

```
if ("my bike" in myTranslation)  ...
```

For ***testing*** if a proper map contains an entry for a certain key value, we can use the Boolean-valued `has` method:

```
if (myTranslation.has("my bike"))  ...
```

For ***cloning*** a traditional map `m`, we can use the composition of JSON.stringify and JSON.parse. We first serialize `m` to a string representation with JSON.stringify, and then de-serialize the string representation to a map object with JSON.parse:

```
var clone = JSON.parse( JSON.stringify( m))
```

Notice that this method works well if the map contains only simple data values or (possibly nested) arrays/maps containing simple data values. In other cases, e.g. if the map contains `Date` objects, we have to write our own clone method.

For ***cloning*** a proper map `m`, we can use the `Map` constructor in the following way:

```
var clone = new Map(m)
```

Since proper maps (defined as instances of `Map`) do not have the overhead of properties inherited from Object.prototype and operations on them, such as adding and deleting entries, are faster, they are preferable to using ordinary objects as maps. Only in cases where it is important to be compatible with older browsers that do not support `Map`, it is justified to use ordinary objects for implementing maps.

1.8. Important types of basic data structures

In summary, there are four types of important basic data structures:

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

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

3. ***maps***, which can be realized as ordinary JS objects having only key-value slots, such as `{"one":1, "two":2, "three":3}`, or as `Map` objects, as discussed above,

4. ***entity tables***, like for instance the table shown below, which are special maps where the values are entity records with a standard ID (or *primary key*) slot, such that the keys of the map are the standard IDs of these entity records.

Table 2.1. An example of an entity table representing a collection of books

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

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

Making such conceptual distinctions helps in the logical design of a program, and mapping them to syntactic distinctions, even if they are not interpreted differently, helps to better understand the intended computational meaning of the code and therefore improves its readability.

1.9. Procedures, methods and functions

Generally, a (parametrized) *procedure* is like a sub-program that can be called (with a certain number of arguments) any number of times from within a program. Whenever a procedure returns a value, it is called a *function*. In OOP, procedures are called *methods,* if they are defined in the context of a class or of an object.

In JavaScript, procedures are called "functions", no matter if they return a value or not. As shown below in Figure 2.1, JS functions are special JS objects, having an optional `name` property and a `length` property providing their number of parameters. If a variable `v` references a function can be tested with

```
if (typeof v === "function") {...}
```

Being JS objects implies that JS functions can be stored in variables, passed as arguments to functions, returned by functions, have properties and can be changed dynamically. Therefore, JS functions are first-class citizens, and JavaScript can be viewed as a *functional programming language*.

The general form of a JS ***function definition*** is an assignment of a JS *function expression* to a variable:

```
var myMethod = function theNameOfMyMethod( params) {
  ...
}
```

where `params` is a comma-separated list of parameters (or a parameter record), and `theNameOfMyMethod` is optional. When it is omitted, the method/function is ***anonymous***. In any case, JS functions are normally invoked via a variable that references the function. In the above case, this means that the JS function is invoked with `myMethod()`, and not with `theNameOfMyMethod()`. However, a named JS function can be invoked by name from within the function (when the function is recursive). Consequently, a recursive JS function must be named.

Anonymous function expressions are called *lambda expressions* (or shorter *lambdas*) in other programming languages.

As an example of an anonymous function expression being passed as an argument in the invocation of another (higher-order) function, we can take a comparison function being passed to the predefined function `sort` for sorting the elements of an array list. Such a comparison function must return a negative number if its first argument is smaller than its second argument, it must return 0 if both arguments are of the same rank, and it must return a positive number if the second argument is smaller than the first one. In the following example, we sort a list of lists of 2 numbers in lexicographic order:

```
var list = [[1,2],[1,3],[1,1],[2,1]];
list.sort( function (x,y) {
  return x[0] === y[0] ? x[1]-y[1] : x[0]-y[0]);
});
```

Alternatively, we can express the anonymous comparison function in the form of an ***arrow function*** expression:

```
list.sort( (x,y) => x[0] === y[0] ? x[1]-y[1] : x[0]-y[0]);
```

A JS ***function declaration*** has the following form:

```
function theNameOfMyMethod( params) {...}
```

It is equivalent to the following named function definition:

```
var theNameOfMyMethod = function theNameOfMyMethod( params) {...}
```

that is, it creates both a function with name `theNameOfMyMethod` and a variable `theNameOfMyMethod` referencing this function.

JS functions can have ***inner functions***. The ***closure*** mechanism allows a JS function using variables (except `this`) from its outer scope, and a function created in a closure remembers the environment in which it was created. In the following example, there is no need to pass the outer scope variable `result` to the inner function via a parameter, as it is readily available:

```
var sum = function (numbers) {
  var result = 0;
  numbers.forEach( function (n) {
      result = result + n;
  });
  return result;
};
console.log( sum([1,2,3,4]));  // 10
```

When a method/function is executed, we can access its arguments within its body by using the built-in `arguments` object, which is "array-like" in the sense that it has indexed elements and a `length` property, and we can iterate over it with a normal `for` loop, but since it's not an instance of `Array`, the JS array methods (such as the `forEach` looping method) cannot be applied to it. The `arguments` object contains an element for each argument passed to the method. This allows defining a method without parameters and invoking it with ***any number of arguments***, like so:

```
var sum = function () {
  var result = 0;
  for (let i=0; i < arguments.length; i++) {
    result = result + arguments[i];
  }
  return result;
};
console.log( sum(0,1,1,2,3,5,8));  // 20
```

A method defined on the prototype of a constructor function, which can be invoked on all objects created with that constructor, such as `Array.prototype.forEach`, where `Array` represents the constructor, has to be invoked with an instance of the class as ***context object*** referenced by the `this` variable (see also the next section on classes). In the following example, the array `numbers` is the context object in the invocation of `forEach`:

```
var numbers = [1,2,3];  // create an instance of Array
numbers.forEach( function (n) {
  console.log( n);
});
```

Whenever such a prototype method is to be invoked not with a context object, but with an object as an ordinary argument, we can do this with the help of ***the JS function `call` method*** that takes an object, on which the method is invoked, as its first parameter, followed by the parameters of the method to be invoked. For instance, we can apply the `forEach` looping method to the array-like object `arguments` in the following way:

```
var sum = function () {
  var result = 0;
  Array.prototype.forEach.call( arguments, function (n) {
    result = result + n;
  });
  return result;
};
```

A two-argument variant of the `Function.prototype.call` method, collecting all arguments of the method to be invoked in an array-like object, is `Function.prototype.apply`. The first argument to both `call` and `apply` becomes `this` inside the function, and the rest are passed through. So, `f.call( x, y, z)` is the same as `f.apply( x, [y, z])`.

Whenever a method defined for a prototype is to be invoked without a context object, or when a method defined in a method slot (in the context) of an object is to be invoked without its context object, we can bind its `this` variable to a given object with the help of ***the JS function `bind` method*** (`Function.prototype.bind`). This allows creating a shortcut for invoking a method, as in `var querySel = document.querySelector.bind( document)`, which allows to use `querySel` instead of `document.querySelector`.

The option of *immediately invoked JS function expressions* can be used for obtaining a namespace mechanism that is superior to using a plain namespace object, since it can be controlled which variables and methods are globally exposed and which are not. This mechanism is also the basis for JS *module* concepts. In the following example, we define a namespace for the model code part of an app, which exposes some variables and the model classes in the form of constructor functions:

```
myApp.model = function () {
  var appName = "My app's name";
  var someNonExposedVariable = ...;
  function ModelClass1() {...}
  function ModelClass2() {...}
  function someNonExposedMethod(...) {...}
  return {
    appName: appName,
    ModelClass1: ModelClass1,
    ModelClass2: ModelClass2
  }
}();  // immediately invoked
```

1.10. Defining and using classes

The concept of a ***class*** is fundamental in *object-oriented* programming. Objects *instantiate* (or *are classified by*) a class. A class defines the properties and methods (as a blueprint) for the objects created with it.

Having a class concept is essential for being able to implement a *data model* in the form of ***model classes*** in a Model-View-Controller (MVC) architecture. However, classes and their inheritance/extension mechanism are over-used in classical OO languages, such as in Java, where all variables and procedures have to be defined in the context of a class and, consequently, classes are not only used for implementing *object types* (or model classes), but also as containers for many other purposes in these languages. This is not the case in JavaScript where we have the freedom to use classes for implementing *object types* only, while keeping method libraries in namespace objects.

Any code pattern for defining classes in JavaScript should satisfy five requirements. First of all, (1) it should allow to define a *class name*, a set of (instance-level) ***properties***, preferably with the option to keep them 'private', a set of (instance-level) ***methods***, and a set of *class-level properties and methods*. It's desirable that properties can be defined with a range/type, and with other meta-data, such as constraints. There should also be two introspection features: (2) an ***is-instance-of predicate*** that can be used for checking if an object is a direct or indirect instance of a class, and (3) an instance-level property for retrieving the ***direct type*** of an object. In addition, it is desirable to have a third introspection feature for retrieving the *direct supertype* of a class. And finally, there should be two inheritance mechanisms: (4) ***property inheritance*** and (5) ***method inheritance***. In addition, it is desirable to have support for *multiple inheritance* and *multiple classifications*, for allowing objects to play several roles at the same time by instantiating several role classes.

There was no explicit class definition syntax in JavaScript before ES2015. Different code patterns for defining classes in JavaScript have been proposed and are being used in different frameworks. But they do often not satisfy the five requirements listed above. The two most important approaches for defining classes are:

1. In the form of a ***constructor*** function that achieves method inheritance via the prototype chain and allows to create new instances of a class with the help of the `new` operator. This is the classical approach recommended by Mozilla in their JavaScript Guide. This is also the approach implemented in ES2015 with the new `class` definition syntax.

2. In the form of a ***factory*** object that uses the predefined `Object.create` method for creating new instances of a class. In this approach, the prototype chain method inheritance mechanism is replaced by a "copy & append" mechanism. Eric Elliott has argued that factory-based classes are a viable alternative to constructor-based classes in JavaScript (in fact, he even condemns the use of classical inheritance with constructor-based classes, throwing out the baby with the bath water).

When building an app, we can use both kinds of classes, depending on the requirements of the app. Since we often need to define class hierarchies, and not just single classes, we have to make sure, however, that we don't mix these two alternative approaches within the same class hierarchy. While the factory-based approach, as exemplified by mODELcLASSjs, has many advantages, which are summarized in Table 2.2, the constructor-based approach enjoys the advantage of higher performance object creation.

Table 2.2. Required and desirable features of JS code patterns for classes

Class feature	**Constructor-based approach**	**Factory-based approach**	**mODELcLASSjs**
Define properties and methods	yes	yes	yes
***is-instance-of* predicate**	yes	yes	yes
***direct type* property**	yes	yes	yes
direct supertype property of classes	no	possibly	yes
Property inheritance	yes	yes	yes
Method inheritance	yes	yes	yes
Multiple inheritance	no	possibly	yes
Multiple classifications	no	possibly	yes
Allow object pools	no	yes	yes

1.10.1. Constructor-based classes

Only in ES2015, a user-friendly syntax for constructor-based classes has been introduced. In **Step 1.a)**, a base class `Person` is defined with two properties, `firstName` and `lastName`, as well as with an (instance-level) method `toString` and a static (class-level) method `checkLastName`:

```
class Person {
  constructor( first, last) {
    this.firstName = first;
    this.lastName = last;
  }
  toString() {
    return this.firstName + " " +
        this.lastName;
  }
  static checkLastName( ln) {
    if (typeof ln !== "string" ||
        ln.trim()==="") {
      console.log("Error: invalid last name!");
    }
  }
}
```

In **Step 1.b)**, class-level ("static") properties are defined:

```
Person.instances = {};
```

Finally, in **Step 2**, a subclass is defined with additional properties and methods that possibly override the corresponding superclass methods:

```
class Student extends Person {
  constructor( first, last, studNo) {
    super.constructor( first, last);
    this.studNo = studNo;
  }
  // method overrides superclass method
  toString() {
    return super.toString() + "(" +
        this.studNo +")";
  }
}
```

In ES5, we can define a base class with a subclass in the form of constructor functions, following a code pattern recommended by Mozilla in their JavaScript Guide, as shown in the following steps.

Step 1.a) First define the constructor function that implicitly defines the properties of the class by assigning them the values of the constructor parameters when a new object is created:

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

Notice that within a constructor, the special variable `this` refers to the new object that is created when the constructor is invoked.

Step 1.b) Next, define the *instance-level methods* of the class as method slots of the object referenced by the constructor's `prototype` property:

```
Person.prototype.toString = function () {
  return this.firstName + " " + this.lastName;
}
```

Step 1.c) *Class-level* ("static") *methods* can be defined as method slots of the constructor function itself (recall that, since JS functions are objects, they can have slots), as in

```
Person.checkLastName = function (ln) {
  if (typeof ln !== "string" || ln.trim()==="") {
    console.log("Error: invalid last name!");
  }
}
```

Step 1.d) Finally, define class-level ("static") properties as property slots of the constructor function:

```
Person.instances = {};
```

Step 2.a) Define a subclass with additional properties:

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

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

In **Step 2b)**, we set up a mechanism for **method inheritance** via the constructor's `prototype` property. We assign a new object created from the supertype's `prototype` object to the `prototype` property of the subtype constructor and adjust the prototype's constructor property:

```
// Student inherits from Person
Student.prototype = Object.create(
    Person.prototype);
// adjust the subtype's constructor property
Student.prototype.constructor = Student;
```

With `Object.create( Person.prototype)` we create a new object with `Person.prototype` as its prototype and without any own property slots. By assigning this object to the `prototype` property of the subclass constructor, we achieve that the methods defined in, and inherited from, the superclass are also available for objects instantiating the subclass. This mechanism of chaining the prototypes takes care of method inheritance. Notice that setting `Student.prototype` to `Object.create( Person.prototype)` is preferable over setting it to `new Person()`, which was the way to achieve the same in the time before ES5.

Step 2c) Define a subclass method that overrides a superclass method:

```
Student.prototype.toString = function () {
  return Person.prototype.toString.call( this) +
      "(" + this.studNo + ")";
};
```

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

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

The method `toString` is invoked on the object `pers1` by using the 'dot notation':

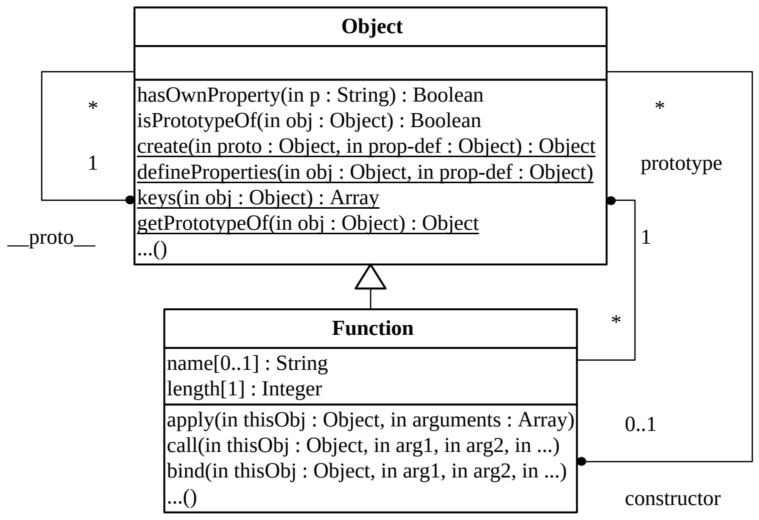
```
alert("The full name of the person is: " + pers1.toString());
```

When an object `o` is created with `o = new C(...)`, where `C` references a named function with name "C", the type (or class) name of `o` can be retrieved with the introspective expression `o.constructor.name`, which returns "C". The `Function::name` property used in this expression is supported by all browsers, except Internet Explorer versions before version 11.

In JavaScript, a ***prototype*** object is an object with method slots (and sometimes also property slots) that can be inherited by other objects via JavaScript's method/property slot look-up mechanism. This mechanism follows the ***prototype chain*** defined by the (in ES5 still unofficial) built-in reference property `__proto__` (with a double underscore prefix and suffix) for finding methods or properties. As shown below in Figure 2.1, every constructor function has a reference to a prototype object as the value of its reference property `prototype`. When a new object is created with the help of `new`, its `__proto__` property is set to the constructor's `prototype` property.

For instance, after creating a new object with `f = new Foo()`, it holds that `Object.getPrototypeOf(f)`, which is the same as `f.__proto__`, is equal to `Foo.prototype`. Consequently, changes to the slots of `Foo.prototype` affect all objects that were created with `new Foo()`. While every object has a `__proto__` property slot (except `Object`), only objects constructed with `new` have a `constructor` property slot.

Figure 2.1. The built-in JavaScript classes `Object` and `Function`.


Object
hasOwnProperty(in p : String) : Boolean
isPrototypeOf(in obj : Object) : Boolean
create(in proto : Object, in prop-def : Object) : Object
defineProperties(in obj : Object, in prop-def : Object)
keys(in obj : Object) : Array
getPrototypeOf(in obj : Object) : Object
...()
*
1
__proto__
*
prototype
1
*
0..1
constructor
Function
name[0..1] : String
length[1] : Integer
apply(in thisObj : Object, in arguments : Array)
call(in thisObj : Object, in arg1, in arg2, in ...)
bind(in thisObj : Object, in arg1, in arg2, in ...)
...()

Notice that we can retrieve the prototype of an object with `Object.getPrototypeOf(o)`, which is an official ES5 alternative to `o.__proto__`.

1.10.2. Factory-based classes

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

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

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

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

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

```
alert("The full name of the person are: " + pers1.getFullName());
```

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

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

2. Storing Database Tables with JavaScript's localStorage API

In most apps, we have some form of data management where data is represented in tables such that table rows correspond to objects, and the table schema corresponds to the objects' type. When building a front-end web app with JavaScript, the simplest approach for persistent data storage is using JavaScript's `localStorage` API, which provides a simple key-value database, but does not support database tables. So, the question is: how can we store and retrieve tables with Local Storage?

We show how to represent database tables in JavaScript in the form of (what we call) *entity tables*, and how to store these tables in Local Storage.

2.1. Entity Tables

The *JavaScript Object Notation (JSON)* defines a concise syntax for JavaScript array literals (*lists*) and JavaScript object literals (*maps*):

- ***Lists*** are expressed as comma-separated lists of data literals ***enclosed in brackets***:

  ```
  ["penguin", "arctis", "snow"]
  ```

- ***Maps*** are expressed as comma-separated lists of key-value slots ***enclosed in curly braces***:

  ```
  {"id": 2901465, "my phone number":"0049.30.227109"}
  ```

  A ***record*** is a special type of map where the keys are *admissible JavaScript identifiers* denoting properties, so they need not be enclosed in quotation marks in JavaScript code. For example, {id: 2901465, phone:"0049.30.227109"} is a record. The value of a property in a record, or the value associated with a key in a map, may be a simple data literal, or an array literal, or another object literal as in:

  ```
  {tags:["penguin","arctis"], photographer:{"last":"Wagner","first":"Gerd"}}
  ```

An entity table contains a set of records (or table rows) such that each record represents an object with a standard identifier property slot. Consequently, an entity table can be represented as a map of records such that the keys of the map are the values of the standard identifier property, and their associated values are the corresponding records, as illustrated by the following example:

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

2.2. JavaScript's LocalStorage API

For a front-end app, we need to be able to store data persistently on the front-end device. Modern web browsers provide two technologies for this purpose: the simpler one is called *Local Storage*, and the more powerful one is called *IndexedDB*. For simplicity, we use the *Local Storage* API in our example app.

A Local Storage database is created *per browser* and *per origin*, which is defined by the combination of protocol and domain name. For instance, `http://example.com` and `http://www.example.com` are different origins because they have different domain names, while `http://www.example.com` and `https://www.example.com` are different origins because of their different protocols (HTTP versus HTTPS).

The Local Storage database managed by the browser and associated with an app (via its origin) is exposed as the built-in JavaScript object `localStorage` with the methods `getItem`, `setItem`, `removeItem` and `clear`. However, instead of invoking `getItem` and `setItem`, it is more convenient to handle `localStorage` as a map, writing to it by assigning a value to a key as in `localStorage["id"] = 2901465`, and retrieving data by reading the map as in `var id = localStorage["id"]`. The following example shows how to create an entity table and save its serialization to Local Storage:

```
var persons = {};
persons["2901465"] = {id: 2901465, name:"Tom"};
persons["3305579"] = {id: 3305579, name:"Su"};
persons["6492003"] = {id: 6492003, name:"Pete"};
try {
  localStorage["personTable"] = JSON.stringify( persons);
} catch (e) {
  alert("Error when writing to Local Storage\n" + e);
}
```

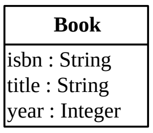
Notice that we have used the predefined method `JSON.stringify` for serializing the entity table `persons` into a string that is assigned as the value of the `localStorage` key "personTable". We can retrieve the table with the help of the predefined de-serialization method `JSON.parse` in the following way:

```
var persons = {};
try {
  persons = JSON.parse( localStorage["personTable"]);
} catch (e) {
  alert("Error when reading from Local Storage\n" + e);
}
```

Chapter 3. Building a Minimal Web App with Plain JS in Seven Steps

In this tutorial, we show how to build a minimal front-end web application with plain JavaScript and Local Storage. The purpose of our example app is to manage information about books. That is, we deal with a single object type: `Book`, as depicted in the class diagram of Figure 3.1.

Figure 3.1. The object type `Book`.



The following table shows a sample data population for the model class `Book`:

Table 3.1. A collection of book objects represented as a table

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

What do we need for a data management app? There are four standard use cases, which have to be supported by the app:

1. **Create** a new book record by allowing the user to enter the data of a book that is to be added to the collection of stored book records.
2. **Retrieve** (or *read*) all books from the data store and show them in the form of a list.
3. **Update** the data of a book record.
4. **Delete** a book record.

These four standard use cases, and the corresponding data management operations, are often summarized with the acronym *CRUD*.

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

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

1. Step 1 - Set up the Folder Structure

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

```
MinimalApp
  src
    c
    m
    v
  index.html
```

In the start page HTML file of the app, we load the file `initialize.js` and the `Book.js` model class file:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
 <meta charset="UTF-8" />
 <title>Minimal JS Front-End App Example</title>
 <script src="src/c/initialize.js"></script>
 <script src="src/m/Book.js"></script>
</head>
<body>
 ...
</body>
</html>
```

The start page provides a menu for choosing one of the CRUD data management use cases. Each use case is performed by a corresponding page such as, for instance, `createBook.html`. The menu also contains options for creating test data with the help of the procedure `Book.createTestData()` and for clearing all data with `Book.clearData()`:

```
<body>
 <h1>Public Library</h1>
 <h2>An Example of a Minimal JavaScript Front-End App</h2>
 <p>This app supports the following operations:</p>
 <menu>
  <li><a href="retrieveAndListAllBooks.html">
       List all books</a></li>
  <li><a href="createBook.html">Add a new book</a></li>
  <li><a href="updateBook.html">Update a book</a></li>
  <li><a href="deleteBook.html">Delete a book</a></li>
  <li><button type="button" onclick="Book.clearData()">
    Clear database
  </button></li>
  <li><button type="button" onclick="Book.createTestData()">
    Create test data
  </button></li>
 </menu>
</body>
```

2. Step 2 - Write the Model Code

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

In the information design model shown in Figure 3.1 above, there is only one class, representing the object type `Book`. So, in the folder `src/m`, we create a file `Book.js` that initially contains the following code:

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

The model class `Book` is coded as a JavaScript constructor function with a single `slots` parameter, which is a record object with fields `isbn`, `title` and `year`, representing the constructor parameters to be assigned to the *ISBN*, the *title* and the *year* attributes of the class `Book`. Notice that, for getting a simple name, we have put the class name `Book` in the global scope, which is okay for a small app with only a few classes. In general, however, we should use the model namespace for model classes, which requires class/constructor definitions like

```
pl.m.Book = function (slots) {...}
```

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

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

2.1. Representing the collection of all `Book` instances

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

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

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

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

Notice that the values of such a map are records corresponding to table rows. Consequently, we could also represent them in a simple table, as shown in Table 3.1.

2.2. Creating a new `Book` instance

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

```
Book.add = function (slots) {
  const book = new Book( slots);
  // add book to the collection of Book.instances
  Book.instances[slots.isbn] = book;
  console.log(`Book ${slots.isbn} created!`);
};
```

2.3. Loading all `Book` instances

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

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

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

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

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

   This conversion is called *de-serialization*.

3. Converting each row of `books`, representing a record (an untyped object), into a corresponding object of type `Book` stored as an element of the entity table `Book.instances`, with the help of the procedure `convertRec2Obj` defined as a "static" (class-level) method in the `Book` class:

   ```
   Book.convertRec2Obj = function (bookRow) {
     const book = new Book( bookRow);
   ```

```
  return book;
};
```

Here is the full code of the procedure:

```
Book.retrieveAll = function () {
  var booksString="";
  try {
    if (localStorage["books"]) {
      booksString = localStorage["books"];
    }
  } catch (e) {
    alert("Error when reading from Local Storage\n" + e);
  }
  if (booksString) {
    const books = JSON.parse( booksString);
    const keys = Object.keys( books);
    console.log(`${keys.length} books loaded.`);
    for (let i=0; i < keys.length; i++) {
      let key = keys[i];
      Book.instances[key] = Book.convertRec2Obj( books[key]);
    }
  }
};
```

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

2.4. Updating a `Book` instance

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

```
Book.update = function (slots) {
  const book = Book.instances[slots.isbn],
        year = parseInt( slots.year);  // convert string to integer
  if (book.title !== slots.title) book.title = slots.title;
  if (book.year !== year) book.year = year;
  console.log(`Book ${slots.isbn} modified!`);
};
```

2.5. Deleting a `Book` instance

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

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

2.6. Saving all `Book` instances

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

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

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

   This conversion is called *serialization*.

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

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

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

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

2.7. Creating test data

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

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

```
};
```

2.8. Clearing all data

The following procedure clears all data from Local Storage:

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

3. Step 3 - Initialize the Application

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

```
var pl = { m:{}, v:{}, c:{} };
```

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

4. Step 4 - Implement the *Create* Use Case

For our example app, the user interface page for the CRUD use case *Create* is called `createBook.html` located in the `MinimalApp` folder. In its `head` element, it loads the app initialization file `initialize.js`, the model class file `Book.js` and the view code file `createBook.js`, and adds a `load` event listener for setting up the *Create* user interface:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Minimal JS Front-End App Example</title>
  <script src="src/c/initialize.js"></script>
  <script src="src/m/Book.js"></script>
  <script src="src/v/createBook.js"></script>
  <script>
   window.addEventListener("load",
       pl.v.createBook.setupUserInterface);
  </script>
</head>
<body>...</body>
</html>
```

For a data management use case with user input, such as "Create", an HTML form is required as a user interface. The form typically has a labelled `input` or `select` field for each attribute of the model class:

```
<body>
 <header>
  <h1>Create a new book record</h1>
 </header>
 <main>
  <form id="Book">
   <div><label>ISBN: <input name="isbn" /></label></div>
   <div><label>Title: <input name="title" /></label></div>
   <div><label>Year: <input name="year" /></label></div>
   <div><button type="button" name="commit">Save</button></div>
  </form>
 </main>
 <footer>
  <a href="index.html">Back to main menu</a>
 </footer>
</body>
```

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

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

```
pl.v.createBook = {
  setupUserInterface: function () {
    const saveButton = document.forms["Book"].commit;
    // load all book objects
    Book.retrieveAll();
    // set an event handler for the save/submit button
    saveButton.addEventListener("click",
        pl.v.createBook.handleSaveButtonClickEvent);
    // handle the event when the browser window/tab is closed
    window.addEventListener("beforeunload", function () {
        Book.saveAll();
    });
  },
  ...
};
```

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

```
pl.v.createBook = {
  ...
  handleSaveButtonClickEvent: function () {
    const formEl = document.forms["Book"];
    const slots = { isbn: formEl.isbn.value,
        title: formEl.title.value,
        year: formEl.year.value };
    Book.add( slots);
    formEl.reset();
  }
};
```

5. Step 5 - Implement the *Retrieve/List All* Use Case

The user interface for the CRUD use case *Retrieve* consists of an HTML table for displaying the data of all model objects. For our example app, this page is called `retrieveAndListAllBooks.html`, located in the main folder `MinimalApp`, and it contains the following code in its `head` element:

```
<head>
  <meta charset="UTF-8" />
  <title>Simple JS Front-End App Example</title>
  <script src="src/c/initialize.js"></script>
  <script src="src/m/Book.js"></script>
  <script src="src/v/retrieveAndListAllBooks.js"></script>
  <script>
   window.addEventListener( "load",
       pl.v.retrieveAndListAllBooks.setupUserInterface);
  </script>
</head>
```

Notice that, in addition to loading the app initialization JS file and the model class JS file, we load the view code file (here: `retrieveAndListAllBooks.js`) and invoke its `setupUserInterface` procedure via a `load` event listener. This is the pattern we use for all four CRUD use cases.

```
<body>
 <header>
  <h1>Retrieve and list all book records</h1>
 </header>
 <main>
  <table id="books">
   <thead><tr><th>ISBN</th><th>Title</th><th>Year</th></tr></thead>
   <tbody></tbody>
  </table>
 </main>
 <footer>
  <a href="index.html">Back to main menu</a>
 </footer>
</body>
```

In the `setupUserInterface` procedure, we first set up the data management context by retrieving all book data from the database and then fill the table by creating a table row for each book object from `Book.instances`:

```
 pl.v.retrieveAndListAllBooks = {
  setupUserInterface: function () {
    const tableBodyEl = document.querySelector("table#books>tbody");
    // load all book objects
    Book.retrieveAll();
    // for each book, create a table row with a cell for each attribute
    for (let key of Object.keys( Book.instances)) {
      const row = tableBodyEl.insertRow();
      row.insertCell().textContent = Book.instances[key].isbn;
      row.insertCell().textContent = Book.instances[key].title;
      row.insertCell().textContent = Book.instances[key].year;
    }
};
```

More specifically, the procedure `setupUserInterface` creates the view table in a loop over all objects of `Book.instances`. In each step of this loop, a new row is created in the table body element with the help of the JavaScript DOM operation `insertRow()`, and then three cells are created in this row with the help of the DOM operation `insertCell()`: the first one for the `isbn` property value of the book object, and the second and third ones for its `title` and `year` property values.

6. Step 6 - Implement the *Update* Use Case

Also for the *Update* use case, we have an HTML page for the user interface (`updateBook.html`) and a view code file (`src/v/updateBook.js`). The HTML form for the UI of the "update book" operation has a selection field for choosing the book to be updated, an `output` field for the standard identifier attribute `isbn`, and an `input` field for each attribute of the `Book` class that can be updated. Notice that by using an `output` field for the standard identifier attribute, we do not allow changing the standard identifier of an existing object.

```
<main>
 <form id="Book">
  <div><label>Select book:
    <select name="selectBook">
     <option value=""> --- </option></select>
  </label></div>
  <div><label>ISBN: <output name="isbn" /></label></div>
  <div><label>Title: <input name="title" /></label></div>
  <div><label>Year: <input name="year" /></label></div>
  <div><button type="button" name="commit">
   Save Changes</button></div>
 </form>
</main>
```

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

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

```
pl.v.updateBook = {
  setupUserInterface: function () {
    const formEl = document.forms["Book"],
        saveButton = formEl.commit,
        selectBookEl = formEl.selectBook;
    Book.retrieveAll();  // load all book objects
    // populate the selection list with books
    for (let key of Object.keys( Book.instances)) {
      const book = Book.instances[key];
      const optionEl = document.createElement("option");
      optionEl.text = book.title;
      optionEl.value = book.isbn;
      selectBookEl.add( optionEl, null);
    }
    // when a book is selected, fill the form with its data
    selectBookEl.addEventListener("change",
        pl.v.updateBook.handleBookSelectionEvent);
    // set an event handler for the submit/save button
    saveButton.addEventListener("click",
        pl.v.updateBook.handleSaveButtonClickEvent);
    // handle the event when the browser window/tab is closed
    window.addEventListener("beforeunload", Book.saveAll);
  },
  ...
}
```

A book selection event is caught via a listener for `change` events on the `select` element. When a book is selected, the form is filled with its data:

```
  handleBookSelectionEvent: function () {
    const formEl = document.forms["Book"],
          selectBookEl = formEl.selectBook,
          key = selectBookEl.value;
    if (key) {
      const book = Book.instances[key];
      formEl.isbn.value = book.isbn;
      formEl.title.value = book.title;
      formEl.year.value = book.year;
    } else {
      formEl.reset();
    }
  },
```

When the save button is activated, a `slots` record is created from the form field values and used as the argument for calling `Book.update`:

```
  handleSaveButtonClickEvent: function () {
    const formEl = document.forms["Book"],
        selectBookEl = formEl.selectBook;
    const slots = { isbn: formEl.isbn.value,
        title: formEl.title.value,
        year: formEl.year.value
    };
    Book.update( slots);
    // update the selection list option element
    selectBookEl.options[selectBookEl.selectedIndex].text = slots.title;
    formEl.reset();
  }
};
```

7. Step 7 - Implement the *Delete* Use Case

The user interface for the *Delete* use case just has a `select` field for choosing the book to be deleted:

```
<main>
 <form id="Book">
  <div>
   <label>Select book: <select name="selectBook">
     <option value="">---</option></select>
   </label>
  </div>
  <div><button type="button" name="commit">Delete</button></div>
 </form>
</main>
```

Like in the *Update* case, the `setupUserInterface` procedure in the view code in `src/v/delete-Book.js` loads the book data into main memory, populates the book selection list and adds some event listeners. The event handler for *Delete* button click events has the following code:

```
handleDeleteButtonClickEvent: function () {
  const selectEl = document.forms["Book"].selectBook,
        isbn = selectEl.value;
  if (isbn) {
    Book.destroy( isbn);
    // remove deleted book from select options
    selectEl.remove( selectEl.selectedIndex);
  }
}
```

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

8. Possible Variations and Extensions

8.1. Using IndexedDB as an Alternative to LocalStorage

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

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

8.2. Styling the User Interface

For simplicity, we have used raw HTML without any CSS styling. But a user interface should be appealing. So, the code of this app should be extended by adding suitable CSS style rules.

Today, the UI pages of a web app have to be *adaptive* (frequently called "responsive") for being rendered on different devices with different screen sizes and resolutions, which can be detected with CSS media queries. The main issue of an adaptive UI is to have a *fluid layout*, in addition to proper viewport settings.

Whenever images are used in a UI, we also need an approach for adaptive bitmap images: serving images in smaller sizes for smaller screens and in higher resolutions for high resolution screens, while preferring scalable SVG images for diagrams and artwork. In addition, we may decrease the font-size of headings and suppress unimportant content items on smaller screens.

For our purposes, and for keeping things simple, we customize the adaptive web page design defined by the HTML5 Boilerplate project (more precisely, the minimal "responsive" configuration available on www.initializr.com). It just consists of an HTML template file and two CSS files: the browser style normalization file `normalize.css` (in its minified form) and a `main.css`, which contains the HTML5 Boilerplate style and our customizations. Consequently, we use a new `css` subfolder containing these two CSS files:

```
MinimalApp-with-CSS
  css
    main.css
    normalize.min.css
  src
    c
    m
    v
  index.html
```

One customization change we have made in `index.html` is to replace the `<div class="main">` container element with the new HTML 5.1 element `<main>` such that we obtain a simple and clear UI page structure provided by the sequence of the three container elements `<header>`, `<main>` and `<footer>`. This change in the HTML file requires corresponding changes in `main.css`. In addition, we define our own styles for `<table>`, `<menu>` and `<form>` elements. Concerning the styling of HTML forms, we define a simple style for implicitly labeled form control elements.

The start page `index.html` now must take care of loading the CSS page styling files with the help of the following two `link` elements:

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

Since the styling of user interfaces is not our primary concern, we do not discuss the details of it and leave it to our readers to take a closer look. You can run the CSS-styled minimal app from our server or download its code as a ZIP archive file.

9. Points of Attention

9.1. Catching invalid data

The app discussed in this chapter is limited to support the minimum functionality of a data management app only. It does not take care of preventing users from entering invalid data into the app's database. In Part 2 of this tutorial, we show how to express integrity constraints in a model class, and how to perform data validation both in the model/storage code of the app and in the HTML5-based user interface.

9.2. 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.

9.3. Boilerplate code

Another issue with the do-it-yourself code of this example app is the *boilerplate code* needed per model class for the data storage management methods `add`, `retrieve`, `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 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.

9.4. Serializing and de-serializing attribute values

Serializing an attribute value means to convert it to a suitable string value. For standard datatypes, such as numbers, a standard serialization is provided by the predefined conversion function `String`. When a string value, like "13" or "yes", represents the value of a non-string-valued attribute, it has to be *de-serialized*, that is, converted to the range type of the attribute, before it is assigned to the attribute. This is the situation, for instance, when a user has entered a value in a form input field for an integer-valued attribute. The value of the form field is of type string, so it has to be converted (de-serialized) to an integer using the predefined conversion function `parseInt`.

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, which has to be converted to an integer in an assignment like the following:

```
this.year = parseInt( formEl.year.value);
```

One important question is: where should we take care of de-serialization: in the "view" (before the value is passed to the "model" layer), or in the "model"? Since attribute range types are a business concern, and the business logic of an app is supposed to be encapsulated in the "model", de-serialization should be performed in the "model" layer, and not in the "view".

9.5. Implicit versus explicit form field labels

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

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

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

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

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

A small disadvantage of using implicit labels may be the lack of support by certain CSS libraries. In the following parts of this tutorial, we will use our own CSS styling for implicitly labeled form fields.

9.6. Synchronizing views with the model

When an app is used by more than one user at the same time, we have to take care of somehow synchronizing the possibly concurrent read/write actions of users such that users always have current data in their "views" and are prevented from interfering with each other. This is a very difficult problem, which is attacked in different ways by different approaches. It has been mainly investigated for multi-user database management systems and large enterprise applications built on top of them.

The original MVC proposal included a data binding mechanism for automated one-way model-to-view synchronization (updating the model's views whenever a change in the model data occurs). We didn't take care of this in our minimal app because a front-end app with local storage doesn't really have multiple concurrent users. However, we can create a (rather artificial) situation that illustrates the issue:

1. Open the *Update* UI page of the minimal app twice (for instance, by opening `updateLearningUnit.html` twice), such that you get two browser tabs rendering the same page.

2. Select the same learning unit on both tabs, such that you see its data in the *Update* view.

3. Change one data item of this learning unit on one of the tabs and save your change.

4. When you now go to the other tab, you still see the old data value, while you may have expected that it would have been automatically updated.

A mechanism for automatically updating all views of a model object whenever a change in its property values occurs is provided by the *observer pattern* that treats any view as an observer of its model object. Applying the observer pattern requires that (1) model objects can have a multi-valued reference property like *observers*, which holds a set of references to view objects; (2) a *notify* method can be invoked on view objects by the model object whenever one of its property values is changed; and (3) the *notify* method defined for view objects takes care of refreshing the user interface.

Notice, however, that the general model-view synchronization problem is not really solved by automatically updating all (other users') views of a model object whenever a change in its data occurs. Because this would only help, if the users of these views didn't make themselves any change of the data item concerned, meanwhile. Otherwise, their changed data value would be overwritten by the automated refresh, and they may not even notice this, which is not acceptable in terms of usability.

9.7. Architectural separation of concerns

From an architectural point of view, it is important to keep the app's model classes independent of

1. the user interface (UI) code because it should be possible to re-use the same model classes with different UI technologies;

2. the storage management code because it should be possible to re-use the same model classes with different storage technologies.

In this tutorial, we have kept the model class `Book` independent of the UI code, since it does not contain any references to UI elements, nor does it invoke any view method. However, for simplicity, we didn't keep it independent of storage management code, since we have included the method definitions for add, update, destroy, etc., which invoke the storage management methods of JavaScrpt's `localStorage` API. Therefore, the separation of concerns is incomplete in our minimal example app.

We show in our mODELcLASSjs tutorial how to achieve a more complete separation of concerns by defining abstract storage management methods in a special storage manager class, which is complemented by libraries of concrete storage management methods for specific storage technologies, called *storage adapters*.

10. Practice Projects

If you have any questions about how to carry out the following projects, you can ask them on our discussion forum.

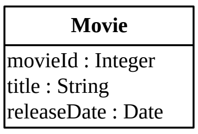
10.1. Managing information about movies

The purpose of the app to be developed 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. So, on application start up, the movie data is read from a persistent data store. When the user quits the application, the data has to be saved to the persistent data store, which should be implemented with JavaScript's Local Storage API, as in the tutorial.

The app deals with just one object type: `Movie`, as depicted in Figure 3.2 below. In the subsequent parts of the tutorial, you will extend this simple app by adding integrity constraints, enumeration attributes, further model classes for actors and directors, and the associations between them.

Notice that `releaseDate` is an attribute with range `Date`, so you need to find out how to display, and support user input of, calendar dates.

Figure 3.2. The object type `Movie`.



For developing the app, simply follow the sequence of seven steps described in the tutorial:

1. Step 1 - Set up the Folder Structure
2. Step 2 - Write the Model Code
3. Step 3 - Initialize the Application
4. Step 4 - Implement the *List Objects* Use Case
5. Step 5 - Implement the *Create Object* Use Case
6. Step 6 - Implement the *Update Object* Use Case

7. Step 7 - Implement the *Delete Object* Use Case

You can use the following sample data for testing:

Table 3.2. 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

Make sure that

1. your HTML pages comply with the XML syntax of HTML5,
2. international characters are supported by using UTF-8 encoding for all HTML files,
3. your JavaScript code complies with our Coding Guidelines and is checked with JSHint (for instance, instead of the unsafe equality test with "==", always the strict equality test with "===" has to be used).

10.2. Managing persistent data with IndexedDB

Improve your app developed in project 1 by replacing the use of the `localStorage` API for persistent data storage with using the more powerful IndexedDB API.

Glossary

C

CRUD	CRUD is an acronym for ***C**reate*, ***R**ead/Retrieve*, ***U**pdate*, ***D**elete*, which denote the four basic data management operations to be performed by any software application.
Cascading Style Sheets	CSS is used for defining the presentation style of web pages by telling the browser how to render their HTML (or XML) contents: using which layout of content elements, which fonts and text styles, which colors, which backgrounds, and which animations. Normally, these settings are made in a separate CSS file that is associated with an HTML file via a special link element in the HTML's head element.

D

Document Object Model	The DOM is an abstract API for retrieving and modifying nodes and elements of HTML or XML documents. All web programming languages have DOM bindings that realize the DOM.
Domain Name System	The DNS translates user-friendly domain names to IP addresses that allow to locate a host computer on the Internet.

E

ECMAScript	A standard for JavaScript defined by the industry organization "Ecma International".
Extensible Markup Language	XML allows to mark up the structure of all kinds of documents, data files and messages in a machine-readable way. XML may also be human-readable, if the tag names used are self-explaining. XML is based on Unicode. SVG and MathML are based on XML, and there is an XML-based version of HTML. XML provides a syntax for expressing structured information in the form of an *XML document* with *elements* and their *attributes*. The specific elements and attributes used in an XML document can come from any vocabulary, such as public standards or user-defined XML formats.

H

Hypertext Markup Language	HTML allows marking up (or describing) the structure of a human-readable web document or web user interface. The XML-based version of HTML, which is called "XHTML5", provides a simpler and cleaner syntax compared to traditional HTML.
Hypertext Transfer Protocol	HTTP is a stateless request/response protocol based on the Internet technologies TCP/IP and DNS, using human-readable text messages for the communication between web clients and web servers. The main purpose of HTTP has been to allow fetching web documents identified

	by URLs from a web browser, and invoking the operations of a back-end web application program from an HTML form executed by a web browser. More recently, HTTP is increasingly used for providing web APIs and web services.

I

IANA	IANA stands for *Internet Assigned Numbers Authority*, which is a subsidiary of ICANN responsible for names and numbers used by Internet protocols.
ICANN	ICANN stands for *Internet Corporation of Assigned Names and Numbers*, which is an international nonprofit organization that maintains the domain name system.
IndexedDB	A JavaScript API for indexed data storage managed by browsers. Indexing allows high-performance searching. Like many SQL DBMS, IndexedDB supports database transactions.
I18N	A set of best practices that help to adapt products to any target language and culture. It deals with multiple character sets, units of measure, keyboard layouts, time and date formats, and text directions.

J

JSON	JSON stands for *JavaScript Object Notation*, which is a data-interchange format following the JavaScript syntax for object literals. Many programming languages support JSON as a light-weight alternative to XML.

M

MathML	An open standard for representing mathematical expressions, either in data interchange or for rendering them within webpages.
MIME	A MIME type (also called "media type" or "content type") is a keyword string sent along with a file for indicating its content type. For example, a sound file might be labeled `audio/ogg`, or an image file `image/png`.
***M**odel-**V**iew-**C**ontroller*	MVC is a general architecture metaphor emphasizing the principle of separation of concerns, mainly between the model and the view, and considering the model as the most fundamental part of an app. In MVC frameworks, "M", "V" and "C" are defined in different ways. Often the term “model” refers to the app’s data sources, while the “view” denotes the app’s code for the user interface, which is based on CSS-styled HTML forms and DOM events, and the “controller” typically denotes the (glue) code that is in charge of mediating between the *view* and the *model*.

O

Object Constraint Language	The OCL is a formal logic language for expressing integrity constraints, mainly in UML class models. It also allows defining deriva-

	tion expressions for defining derived properties, and defining preconditions and postconditions for operations, in a class model.
Object-Oriented Programming	OOP is a programming paradigm based on the concepts of *objects* and *classes* instantiated by objects. Classes are like blueprints for objects: they define their *properties* and the *methods/functions* that can be applied to them. A higher-level characteristic of OOP is *inheritance* in class hierarchies: a subclass inherits the features (properties, methods and constraints) of its superclass.
Web Ontology Language	OWL is formal logic language for knowledge representation on the Web. It allows defining vocabularies (mainly classes with properties) and supports expressing many types of integrity constraints on them. OWL is the basis for performing automated inferences, such as checking the consistency of an OWL vocabulary. Vocabularies, or data models, defined in the form of UML class models can be converted to OWL vocabularies and then checked for consistency.

P

Portable Network Graphics	PNG is an open (non-proprietary) graphics file format that supports lossless data compression.
Polyfill	A polyfill is a piece of JavaScript code for emulating a standard JavaScript method in a browser, which does not support the method.

R

Resource Description Framework	RDF is a W3C language for representing machine-readable propositional information on the Web.

S

Standard Generalized Markup Language	SGML is an ISO specification for defining markup languages. HTML4 has been defined with SGML. XML is a simplified successor of SGML. HTML5 is no longer SGML-based and has its own parsing rules.
Scalable Vector Graphics	SVG is a 2D vector image format based on XML. SVG can be styled with CSS and made interactive using JavaScript. HTML5 allows direct embedding of SVG content in an HTML document.
Slot	A slot is a name-value pair. In an object of an object-oriented program (for instance, in a Java object), a slot normally is a property-value pair. But in a JavaScript object, a slot may also consist of a method name and a method body or it may be a key-value pair of a map.

U

Unicode	A platform-independent character set that includes almost all characters from most of the world's script languages including Hindi, Burmese and Gaelic. Each character is assigned a unique integer code in the range between 0 and 1,114,111. For example, the Greek letter

	π has the code 960. Unicode includes legacy character sets like ASCII and ISO-8859-1 (Latin-1) as subsets. XML is based on Unicode. Consequently, the Greek letter π (with code 960) can be inserted in an XML document as &#960; using the XML entity syntax. The default encoding of Unicode characters in an XML document is UTF-8, which uses only a single byte for ASCII characters, but three bytes for less common characters.
Uniform Resource Identifier	A URI is either a *Uniform Resource Locator (URL)* or a *Uniform Resource Name (URN)*.
Uniform Resource Locator	A URL is a resource name that contains a web address for locating the resource on the Web.
Unified Modeling Language	The UML is an industry standard that defines a set of modeling languages for making various kinds of models and diagrams in support of object-oriented problem analysis and software design. Its core languages are *Class Diagrams* for information/data modeling, and *Sequence Diagrams*, *Activity Diagrams* and *State Diagrams* (or *State Charts*) for process/behavior modeling.
Uniform Resource Name	A URN refers to a resource without specifying its location.
User Agent	A user agent is a front-end web client program such as a web browser.

W

WebM	WebM is an open (royalty-free) web video format supported by Google Chrome and Mozilla Firefox, but not by Microsoft Internet Explorer and Apple Safari.
Web Hypertext Application Technology Working Group	The *WHATWG* was established in 2004 by former employees of Apple, Mozilla, and Opera who have been unhappy with the slow progress of web technology standardization due to W3C's choice to focus on the standardization of XHTML2. Led by Ian Hickson, they developed HTML5 and related JavaScript APIs in competition and collaboration with the W3C.
World Wide Web	The WWW (or, simply, "the Web") is a huge client-server network based on HTTP, HTML and XML, where web browsers (and other 'user agents'), acting as HTTP clients, access web server programs, acting as HTTP servers.
World Wide Web Consortium	The *W3C* is an international organization in charge of developing and maintaining web standards.

X

XML HTTP Request	The XML HTTP Request (XHR) API allows a JavaScript program to exchange HTTP messages with back-end programs. It can be used for retrieveing/submitting information from/to a back-end program without submitting HTML forms. XHR-based approaches have been subsumed under the acronym "AJAX" in the past.