Poul Klausen

JAVA 12

www and development of the client part

Software Development

bookboon
LEARNING

POUL KLAUSEN

JAVA 12: WWW AND DEVELOPMENT OF THE CLIENT PART

SOFTWARE DEVELOPMENT

Java 12: WWW and development of the client part: Software Development
1st edition

ISBN 978-87-403-1974-3
Peer review by Ove Thomsen, EA Dania

CONTENTS

FOREWORD

This book is the twelfth in a series of books on software development. The programming language is Java, and the language and its syntax and semantics fills obviously much, but the books have also largely focus on the process and how to develop good and robust applications. This book is similar to the book Java 11 about development of web applications, but focusing on the client side. This means that key topics are style sheets and JavaScript, and in particular, the last part fills. Another topic is Ajax, like the book gives a brief introduction to PrimeFacses. The primary purpose of the book is to show that it is possible to perform complex client-side programming using JavaScript. The book requires knowledge of programming of the server side similar to what has been dealt with in the previous book, and together, the two books provide a relatively basic introduction to web application development.

As the title says this series of books deals with software development, and the goal is to teach the reader how to develop applications in Java. It can be learned by reading about the subject and by studying complete sample programs, but most importantly by yourself to do it and write your own programs from scratch. Therefore, an important part of the books is exercises and problems, where the reader has to write programs that correspond to the substance being treated in the books. All books in the series is built around the same skeleton and will consist of text and examples and exercises and problems that are placed in the text where they naturally belongs. The difference between exercises and problems is that the exercises largely deals with repetitions of the substance that is presented in the text, and furthermore it is relatively accurately described what to do. Problems are in turn more loosely described, and are typically a little bigger and there is rarely any clear best solution. These are books to be read from start to finish, but the many code examples, including exercises and problems plays a central role, and it is important that the reader predict in detail studying the code to the many examples and also solves the exercises and problems or possibly just studying the recommended solutions.

All books ends with one or two larger sample programs, which focus primarily is on process and an explanation of how the program is written. On the other hand appears the code only to a limited extent – if at all – and the reader should instead study the finished program code perhaps while testing the program. In addition to show the development of programs that are larger than the examples, which otherwise is presented, the aim of the concluding examples also is to show program examples from varying fields of application.

Most books also ends with an appendix dealing with a subject that would not be treated in the books. It may be issues on the installation of software or other topics in computer technology, which are not about software development, but where it is necessary to have an introductory knowledge. If the reader already is familiar with the subject, the current appendix can be skipped.

The programming language is, as mentioned Java, and besides the books use the following products:

1. NetBeans as IDE for application development
2. MySQL to the extent there is a need for a database server (from the book Java 6 onwards)
3. GlassFish as a web server and application server (from the book Java 11 onwards)

It is products that are free of charge and free to install, and there is even talk about products, where the installation is progressing all by itself and without major efforts and challenges. In addition, there are on the web detailed installation instructions for all the three products. The products are available on Windows and Linux, and it therefore plays no special role if you use Linux or Windows.

All sample programs are developed and tested on machines running Linux. In fact, it plays no major role, as both Java and other products work in exactly the same way whether the platform is one or the other. Some places will be in the books where you could see that the platform is Linux, and this applies primarily commands that concerning the file system. Otherwise it has no meaning to the reader that the programs are developed on a Linux machine, and they can immediately also run under Windows unless a program refers to the file system where it may be necessary to change the name of a file.

Finally a little about what the books are not. It is not "how to write" or for that matter reference manuals in Java, but it is as the title says books on software development. It is my hope that the reader when reading the books and through the many examples can find inspiration for how to write good programs, but also can be used as a source collection with a number of examples of solutions to concrete everyday programming problems that you regularly face as a software developer.

1 INTRODUCTION

In the book Java 11, I have provided a basic introduction to Web Application Development in Java. I have primarily processed the server side, where a browser sending a request to a web server, that dynamically dependent on the data sent with the request, creates the HTML document to be sent as a response to the client. This client/server principle is also the whole idea of web applications and, in fact, you can stop here as all web applications could in principle be developed according to the pattern outlined in Java 11. However, there are much more to learn and here I would like to mention:

1. The visual and the possibilities for developing web applications (web sites) with an attractive user interface, and especially user interfaces that are easy to maintain.
2. Client programming, where part of the code is executed by the browser for the purpose of preventing a request to the server every time something happens on the client page.

The first is primarily a matter of which components (items) are provided and what options are available to customize them to exactly the look and feel that you may be interested in. It's about *cascading style sheets* and HTML5. Both of them do not have anything to do with Java, but are subjects that you as a web developer must be aware of. And then the visual, that more than anything else is a question about skill and knowledge about what is a good user interface and how to design web user interfaces. Unfortunately, I only have limited knowledge about what is a good user interface, so I just have to introduce the technique.

The other is primarily about *JavaScript*, which is program code sent as script (text) along with the server's response, and which is code executed by the browser without sending requests to the server. That it's script code means that it's only text and not binary code, why it is generally perceived as harmless and can not contain malicious code, but also because it's limited what to do with JavaScript and basically you can only manipulate the HTML document's elements. JavaScript has gained interest, among other things, because the browsers interpreters to JavaScript have been optimized, why JavaScript today is effective, but also because the language has evolved so that today you can write advanced JavaScript that also can be maintained. In spite of the name, JavaScript is not Java, and it is important to make it clear that it is a simple script language, but with the same syntax as Java – and hence the name. JavaScript will fill a part in this book.

I will also mention Java applets, that are program code, which are performed by the browser, but this is common binary Java code downloaded from the server, and the browser can then use a plugin to execute the code. One should note that you have the full Java available and that an applet can, in principle, perform all that is possible with a Java application. For that reason, many people are not excited about Java applets, as applets has a major security risk, and there are also examples that the fear is justified. Java applets are currently used to a limited extent and are not dealt with further in this book.

The book will also contain topics that relate to the server, and in particular I will look at Ajax in connection with facelets, but the book also deals with other topics related to JavaFaces.

2 HTML

In the book Java 11, I have said that I do not want to treat HTML, and the following is also just an ultra short presentation that merely highlights the most important HTML concepts, but as the browser basically receives HTML, it's useful with a short introduction to the subject. This is especially true after the use of HTML5, as this standard adds some subjects and princips, and the standard is gradually supported by most browsers.

HTML is very closely linked to cascading style sheets, also known as CSS, and the goal is that HTML using markup should structure data while CSS defines how the individual elements should be displayed in the browser. Styling of HTML elements should occur in its own document, calling a *Cascading Style Sheet*, while the HTML document alone must take care of structuring the data. That way, web pages are much easier to maintain.

A HTML document has the basic form:

```
<!DOCTYPE html>
<html>
 <head>
  <title>HTML Application</title>
  <meta charset="UTF-8">
 </head>
 <body>
  …
 </body>
</html>
```

but many development tools add their own elements to the header. HTML define more structural elements where the most important are:

- *section*, that represents a section in a document, and it can together with h1–h6 elements be used to indicate the document's structure, and a section must start with a h element
- *article*, that represents an independent piece of content in a document and as the name says an article, and an article must start with a h element
- *aside*, that represents a content that is only slightly related to the rest of the page
- *header*, that represents a header section
- *footer*, that represents a footer section and will often contain information about the author, copyright information and so on
- *nav*, that represents a section in the document intended for navigation (links)
- *dialog*, that is used to mark up a conversation
- *figure*, that can be used to associate a caption together with some embedded content, such as a graphic or a video

It is not a requirement that a HTML document uses these elements (which are introduced in HTML5), but it is recommended, among other things, that they can be styled in a style sheet. Similarly, the following structure is recommended for an HTML document, where there may be more sections and where some elements can be omitted completely:

```
<!DOCTYPE html>
<html>
 <head>
  <title>Html5Document</title>
  <meta charset="UTF-8">
 </head>
 <body>
  <h1>HTML5 Document</h1>
  <header>
   <h2>Content</h2>
   <nav>
    <ul>
     <li>For example a menu</li>
    </ul>
   </nav>
  </header>
  <section>
   <h2>Section title</h2>
   <article>
    <header>
     <h2>Article title</h2>
     <p>Introdoction to this title</p>
    </header>
    <p>Here is the text.</p>
   </article>
   <article>
    <header>
     <h2>Article title</h2>
     <p>Introdoction to this title</p>
    </header>
    <p>Here is the text.</p>
   </article>
  </section>
  <aside>
   <h2>About this section</h2>
   <p>References or other.</p>
  </aside>
  <footer>
   <p>Copyright and similar</p>
  </footer>
 </body>
</html>
```

There are many other HTML elements, and there are many new elements in HTML5, but I do not want to mention the elements here. General is the form of a HTML element

```
<elem attr … >content</elem>
```

where there may be a number of attributes. If there is no content, it is allowed to write:

```
<elem attr … />
```

even though it is not actually intended to use this notation in HTML5. A very specific example could be:

```
<h1 class="header1>That is a header</h1>
```

where there is one attribute that refers to a class in a style sheet. As a starting point for this chapter, I have created a web application called *HTMLApplication*, which for the moment only has a single page *index.html*. I do not want to display the content here as the page fills a lot, and the page should show only how the above structure elements are used to structure the content of a HTML document. When you test the document, note that the structural elements have no visual effect. They are used solely to structure the text, and the visual effects are solely intended to header elements (h1–h6) as well as a list of links. In any case, the visual effect is what is default.

It should also be mentioned that it is allowed to define custom attributes and, if necessary, they should start with the word *data-* and the application is that they can be referenced from *JavaScript,* but also from a style sheet.

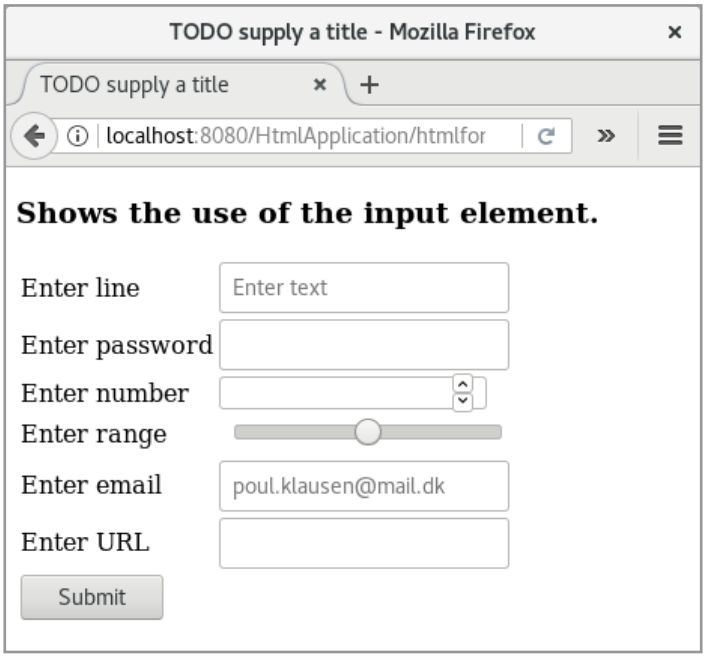
2.1 HTML FORMS

In the book Java 11, I have shown the use of HTML forms, and it is generally not associated with the big challenges, but HTML actually defines a lot more about forms. Consider the following document that is part of the *HTMLApplication* project:

```
<!DOCTYPE html>
<html>
 <head>
  <title>TODO supply a title</title>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
 </head>
 <body>
  <h1>Shows the use of the input element.</h1>
   <form method="post">
    <table>
     <tr>
      <td>Enter line</td>
      <td>
       <input type="text" id="linefield" required placeholder="Enter text" />
      </td>
     </tr>
     <tr>
      <td>Enter password</td>
      <td><input type="password" id="password" /></td>
     </tr>
     <tr>
      <td>Enter number</td>
      <td><input type="number" id="numberfield" step="5" /></td>
     </tr>
     <tr>
      <td>Enter range</td>
```

```
      <td><input type="range" id="rangefield" min="100" max="999" /></td>
     </tr>
     <tr>
      <td>Enter email</td>
      <td>
         <input type="email" id="mailfield"
         placeholder="poul.klausen@mail.dk" />
      </td>
     </tr>
     <tr>
      <td>Enter URL</td>
      <td><input type="url" id="urlfield" /></td>
     </tr>
     <tr>
      <td colspan="2"><input type="submit" value="Submit" /></td>
     </tr>
    </table>
   </form>
 </body>
</html>
```

The basic form element is *input*, and you specifies with a *type* attribut how to render the element on the screen and what else it should be able to do with the element. The default *type* is *text*, which means you can enter a single text line. Note (the first input element) that you can specify *required* and that with a placeholder you can specify a hint for the text to be entered. In particular, note the *number* and *range* types that indicate that you can only enter numbers (the last is rendered as a slider), and note the two last ones to enter an email address and a URL, respectively.



Note that the form is used solely as an example of which *input* elements exists and how to validate data before sending to the server.

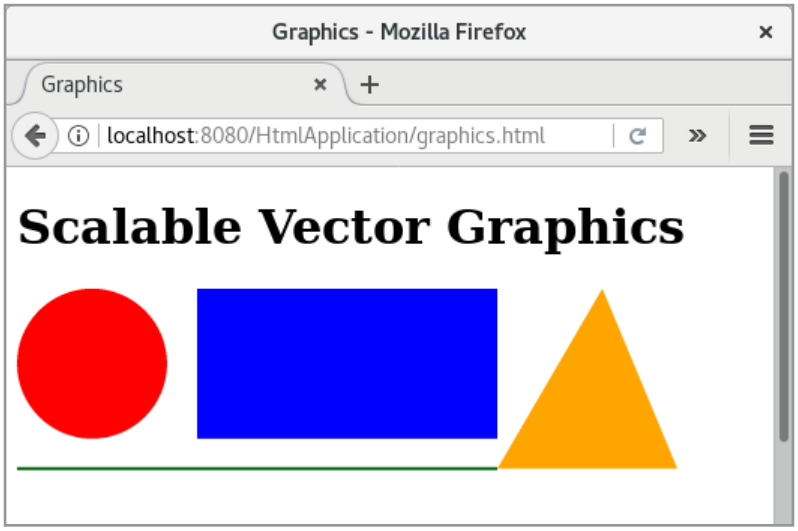
2.2 SCALABLE VECTOR GRAPHICS

In HTML5, it is also possible to work with geometric shapes like circles and rectangles. Consider the following page:

```
<!DOCTYPE html>
<html>
 <head>
  <title>Graphics</title>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
 </head>
 <body>
  <h1>Scalable Vector Graphics</h1>
  <svg height="200" width="500" xmlns="http://www.w3.org/2000/svg">
   <circle cx="50" cy="50" r="50" fill="red" />
   <rect x="120" y="0" width="200" height="100" fill="blue" />
```

```
    <line x1="0" y1="120" x2="320" y2="120" stroke-width="2"
     stroke="darkgreen" />
    <polygon points="390,0 440,120, 320,120" fill="orange" />
   </svg>
  </body>
</html>
```

Opening the page in the browser gives you the result:



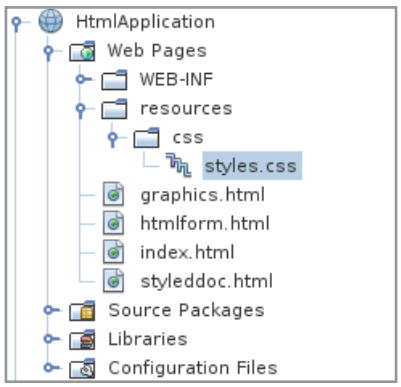
The code is easy enough to understand and you are encouraged to investigate what else is available.

PROBLEM 1

The direcctory for this book contains a document *java12*, that is a text document with the text for the two first chapters in this book, where the text is saved as plan text without formatting. The directory also has two png-files called *screen1.png* and *screen2.png*. The files are the screen dumps shown in this chapter. The task is to create a web application with a HTML document, that shows the content of the file *java12* and the two images. The page should only structure the text (display the content), you should not format the text. This is the subject of a later problem.

3 CASCADING STYLE SHEETS

After the above about HTML, I'm ready for a short introduction on style sheets, and I will take as the starting point the same program as in the previous chapter. I have added a new HTML document to the project and called it *styleddoc.html*. The content is the same as *index.html* and the goal is to show how the document's visual presentation can be changed using a style sheet. Initially I have created a directory *resources* and including a subdirectory *css* and added a style sheet named *styles.css*:



Initially, it is nothing but an empty file. The document that has to use a style sheet must have a *link* element in the header:

```
<head>
 <title>HTML Application</title>
 <meta charset="UTF-8">
 <link href="resources/css/styles.css" rel="stylesheet" type="text/css"/>
</head>
```

The syntax should be correct and you can easily drag the file into the document with the mouse, and NetBeans will automatically insert the correct link.

The idea of style sheets is as mentioned to separate content and presentation. A style sheet consists of styles identified by *selectors*, and as an example, I have added the following style for a *h1* selector to the style sheet:

```
h1 {
 font-size: 36pt;
 font-weight: bold;
 color: darkslateblue;
}
```

Such a selector is called a *type selector* or an *element selector* and means that all of the document's *h1* elements will use the style that the selector defines. You can thus define how the headers of the document should be presented, and it will apply to all documents that use that style sheet. Typically, you also wants to define a style for the body element, such as defining the font to be used as default:

```
body {
 font-family: serif;
 font-size: 10pt;
}
```

You can also define styles for *descendent selectors* that occur with a space between two selectors:

```
section p {
 margin-left: 20px;
}
```

This selector indicates that a paragraph element must have a left indentation of 20, but only for paragraphs that are nested in a *section* element.

You can also define a so-called ID selector, known as it starts with the character #

```
#content-list {
 font-size: 14pt;
 font-weight: bold;
 color: darkgray;
}
```

This selector is used by an element whose *id* is *content-list*, and as element's *id* attribute should be unique, there is only one element in a document that directly can use this style. In this case, it is the following element:

```
<p id="content-list">Content</p>
```

Finally, you can define *class selectors*, starting with a point:

```
.default-text {
 width: 800px;
 color: #222222;
}
```

An element can use a *class selector* by directly specifying it with a class attribute:

```
<p class="default-text">In the book Java 11, … </p>
```

As another example of selectors, there is defined a *class* with the name *forword* that defines the color as green:

```
.forword {
 color: darkgreen;
}

.forword h5 {
 color: darkred;
}
```

The last style means that a *h5* element under an element whose *style* is *forword* appears in red text:

```
<article class="forword">
 <h4 id="forword">Forword</h4>
 <section>
  <h5>About this book</h5>
  <p>Here is the text</p>
 </section>
</article>
```

You can also define selectors that relates to attributes. As an example, the following style means:

```
h4[id] {
 color: darkred;
}
```

that all *h4* elements that have an *id* attribute must be displayed with a dark red text. As another example, the HTML element *abbr* can be used to display a tooltip:

```
<h5>HTML <abbr title="Introduction to HTML syntax">syntax</abbr></h5>
```

Here, the value of the attribute *title* will be displayed as a tooltip if the mouse is pointing to the word *syntax*. Consider the following styles:

```
abbr[title] {
 color: gray;
}

abbr[title]:hover {
 color: red;
}
```

They define that for all *abbr* elements with a *title* attribute, the text should appear gray and holding the mouse over the text (*mouse ower*), the text should appear red.

Attributes as selectors can be especially interesting for custom attributes. Consider the following paragraph:

```
<p data-marked>Here is the text</p>
```

For example, if you want all paragraphs with this attribute to appear with a large font, you can write:

```
p[data-marked] {
 font-size: 48pt;
}
```

You are encouraged to study the final result (the document *styleddoc.html*) and especially the finished style sheet.

3.1 MORE SELECTORS

The example *StyledDocument* consists of a single HTML document as well as a style sheet. If you open the document in the browser, the result is:

I do not want to show the document here, as there is not much news of explaining. The style sheet is as follows:

```
p {
 margin-left: 30px;
 font-size: 10px;
}
```

```
.queens {
 margin-left: 10px;
}

.queens > article > h2 {
 margin-left: 20px;
 font-size: 18px;
 color: red;
}

#regent + p {
 color: blue;
}

p > span {
 font-weight: bold;
 color: darkgreen;
}

#blueking ~ p {
 font-size: 12px;
}

* {
 color: gray;
}

.king {
 margin-left: 20px;
}

.king > * {
 color: orange;
}

p[data-info="to"] {
 color: magenta;
}

p[data-info="from"] {
 color: maroon;
}

p[data-info*="th"] {
 color: pink;
 font-weight: bold;
}
```

The first two styles do not require further explanation. The third identifies elements with the class *queens* including child elements of the type *article* and again including *h2* child elements, and then *h2* elements under *article* elements under elements with the class *.queens*.

The fourth style defines a paragraph that is next sibling to an element with *id regent*. The fifth style is used by a *span* element, which is child of a paragraph element. The sixth style is used by all elements that are siblings of an element with *id blueking*.

A * is used as a selector for all elements (rarely used directly), but in combination with other selectors it can be used to style all elements in a subtree. In this case, means

```
.king > *
```

all elements under elements with the class *king*. Finally, there are the three last styles that relate to attributes. Here they are used two a paragraph with the attribute *data-info*, and in particular you can select the attributes text:

- `[attr^="text"]` attributes which starts with text
- `[attr$="text"]` attributes which ends with text
- `[attr*="text"]` attributes which contains text

and there are actually a few more options.

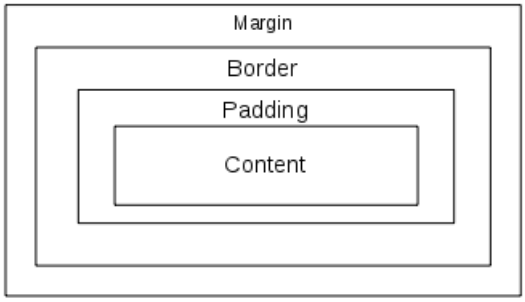
3.2 STYLES

Above I have shown examples of how to refer from a style sheet to the individual elements in the HTML tree, but there are quite a few other areas where styles are available, and where you should know what they means – and there are really many. Styling an HTML document is an extensive topic, and an adequate processing of styles is beyond the scope of this book. In this section I will outline some of the basic principles, but there is much more, and in general, in software development it is an area that requires considerable expertise before designing modern web applications. However, I would like to emphasize once again that the development tool usually supports styling and is a significant help in practice, although it is by no means always easy to understand the effects of the many possibilities.

Basics styling includes:

- how much the elements fill in the browser
- where the elements are located
- how the content of the individual elements is formatted, including fonts and colors

Basically, an HTML element appears as a box:



In the middle you have the content (if there is a content) and without having *padding*, which indicates how much space there should be outside of the content. For an element with a style, the *width* and *height* will in most cases mean the size of the content area inclusive padding, but it can be changed with *box-sizing*. Around the padding there may be a *border*, and finally there may be a *margin* about all of it. For example, if you specify a *background* color, it will relate to the entire *padding* area. You should especially note that the *margin* area is transparent and is thus only used to create spaces between the elements. You should be aware that if elements are vertically aligned and two margins (top and bottom) meet, they are automatically collapsed to one whose height is the largest of the two.

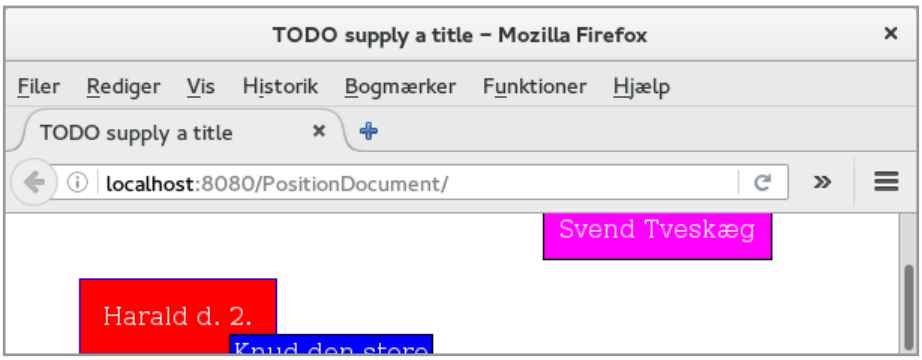
Many elements such as *p*, *h*, and *article* elements behave as described above and show the contents as a block and are therefore called *block boxes*. Others such as *strong* and *span* are called *inline boxes* as they format the content of a line. How the different elements show their content can be defined by *display* and thus override how they as default shows their content. Elements can be laid out by the browser in several ways, but the default is *static*, and for *block boxes*, it means that they are placed vertically underneath each other, whereas for *inline boxes* it means that they are laid out horizontally on the same line and wraps as required.

As mentioned, the position of elements is defined as *static*, which means that the elements float in the window, but you can define other options, as are best illustrated by an example. If you open the application *PositionDocument* in the browser, you get the window:

The elements are all *span* elements that are rendered in the following order:

1. Gorm den gamle
2. Harald Blåtand
3. Svend Tveskæg
4. Harald d. 2.
5. Knud den store
6. Hardeknud

Unless otherwise stated, they would be laid out in one row which would wrap if the line was too long. In this case, the two first and the last are laid out in default (*static*) positions, while the third has *relative* position, which means that it is laid out relative to the previous element, as is *Harald Blåtand*. You should note that the last element (*Hardeknud*) is laid out as if the element *Svend Tveskeg* was laid out in default position and that the position of the previous two is ignored as they are laid out as *absolute* and *fixed* position respectively. *Absolute* means that the element is placed in a fixed position relative to the upper left corner of the document. *Fixed* is a variant, but here the element is also placed in a fixed position, but this time relative to the part of the document that is visible. You can illustrate the difference by changing the size of the window and then scrolling it:

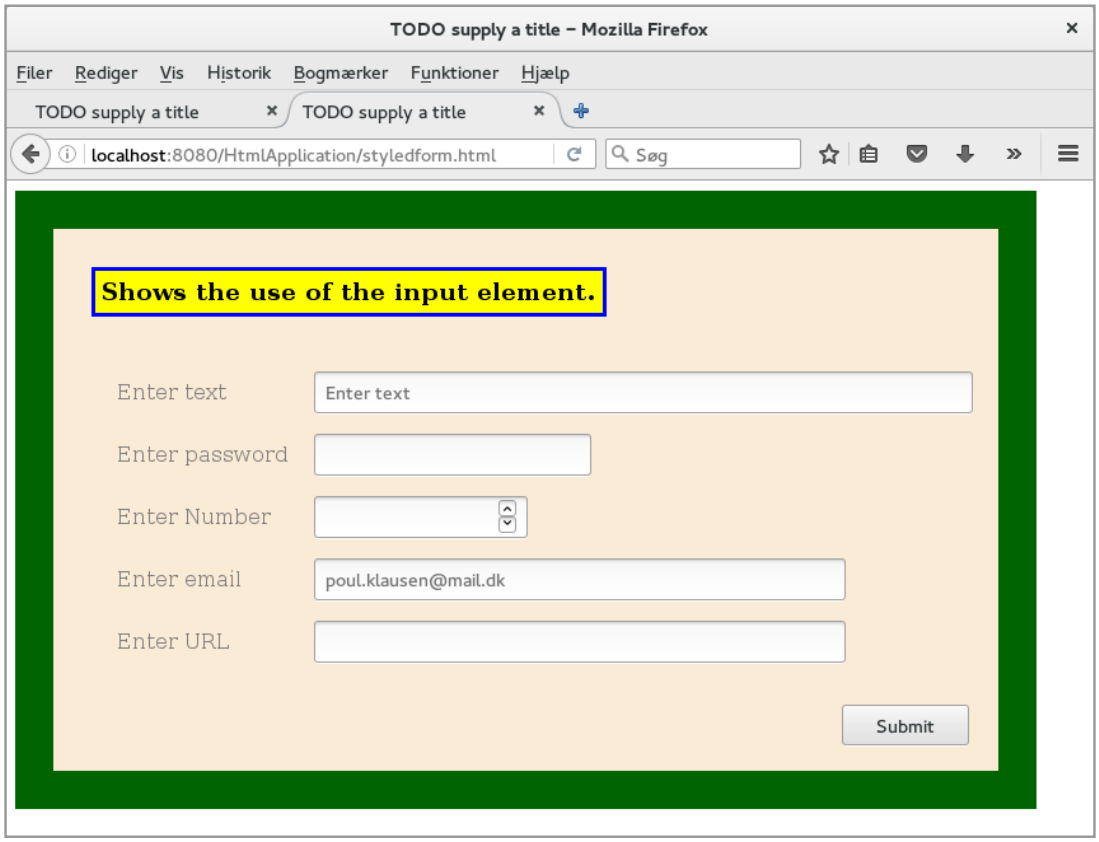


The code is shown below:

```
<!DOCTYPE html>
<html>
 <head>
  <title>TODO supply a title</title>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <style>
   .normal {
    background: yellow;
    border: 1px solid gray;
    padding: 5px;
   }
   .relative {
    background: magenta;
    color: white;
    border: 1px solid black;
    padding: 10px;
    position: relative;
    left: 50px;
    top: 30px;
   }
   .absolute {
    background: red;
    color: white;
    border: 1px solid blue;
    padding: 15px;
    position: absolute;
    left: 50px;
    top: 80px;
   }
   .fixed {
    background: blue;
    color: white;
    border: 1px solid black;
    padding: 2px;
    position: fixed;
    left: 150px;
    top: 80px;
   }
  </style>
 </head>
```

```
 <body>
  <span class="normal">Gorm den gamle</span>
  <span class="normal">Harald Blåtand</span>
  <span class="relative">Svend Tveskæg</span>
  <span class="absolute">Harald d. 2.</span>
  <span class="fixed">Knud den store</span>
  <span class="normal">Hardeknud</span>
 </body>
</html>
```

Styling includes much more than what has been said in this chapter, but the above should be enough to get an idea of what is possible. Often you are in the situation that you want to achieve a certain effect on an element, but you do not know what to write. Here is the best source actually the Internet, where there are countless examples of how to style different HTML elements. As a conclusion, I have added a document called *styledform.html* to the project *HtmlApplication*. The document shows the same form as previously mentioned, but this time no table is used and the elements are placed only by styling:



You are encouraged to study the document and especially the corresponding style sheet, and how the individual elements are styled.

PROBLEM 2

Start with a copy of the project from problem 1. You must then add a style sheet to the project so you uses styles to format the document so it looks like the finished version of the book. All styles should be in the style sheet and you must expand the document with the necessary *ID* and *class* definitions.

4 JAVASCRIPT

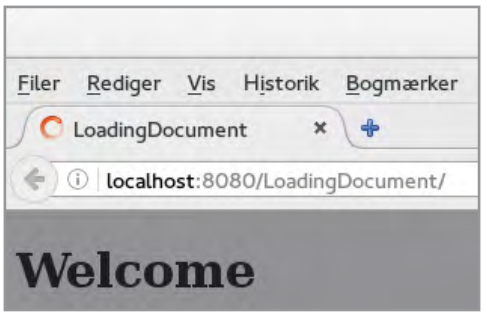
I will then give an introduction to JavaScript, which is the preferred language for code on the client side. The language has essentially the same syntax as Java, but otherwise the two languages do not have anything in common, and JavaScript is even a language that is very different from Java, but if you want to work professional as a web developer, you has to know JavaScript. Therefore, this chapter.

JavaScript is an interpreted programming language, and the source code are as Java just text, but unlike Java, the code is not translated, but is sent as text to the browser, which has a built-in interpreter that interprets the JavaScript code. It is a real interpreter who interprets the code statement for statement and execute each statement before the next is interpreted and performed. To show what JavaScript is, I want to start with a very simple example, called *LoadingDocument*. There is only one file *index.html* whose content is:

```
<!DOCTYPE html>
<html>
 <head>
  <title>LoadingDocument</title>
  <meta charset="UTF-8">
  <script>
   hello = 'Hello';

   function show() {
    alert(hello);
    hello += ' World';
   }
  </script>
  </head>
 <body>
  <h1>Welcome</h1>
  <script>show();</script>
  <h2>Go on</h2>
  <script>alert(hello);</script>
  <h3>OK</h3>
 </body>
</html>
```

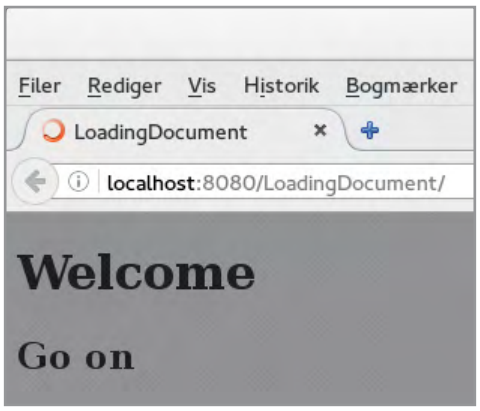
In the *head* section there is a *script* element and it is an element that can contain *JavaScript* code. In this case, a variable and a function are defined. The variable is called *hello*, and it is initialized with the text *"Hello"*. In principle, JavaScript uses variables as other programming languages, including Java, but you should primarily note that the variable has no type. The script block also has a function that basically works as a method in Java and can be executed by calling it. In this case the function is called *show()* and it has two statements. The first is an *alert()* which opens a simple popup that shows the value of the variable *hello*. The next statement changes the value of the variable, and here you should notice that the syntax is, as you know it from Java. If the application is opend in the browser, it will render the *body* of the document from start to finish, starting with the following window where the *h1* element is rendered (see below). After the *h1* element is rendered there is a new script element

```
<script>show();</script>
```



which calls the JavaScript function *show()*, that means that the browser stops and opens a popup:

The browser will continue to render the document when clicking *OK*, and the browser will then render the *h2* element:



Then immediately a script element follows, which this time just performs an alert() which in a popup shows the value of the variable *hello*:

Here you should note that the variable has been changed in the *show()* function. After the above popup is displayed, the browser resumes until clicked on *OK*, after which the browser renders the last *h3* element, and only then the entire document is parsed and displayed in the browser:

The above example shows a bit about what JavaScript is and how to add JavaScript to an HTML document, but not what JavaScript can be used for what is the goal of the following. First, however, I would like to see a little more on the syntax of the language.

4.1 NATURE OF LANGUAGES

When you start with JavaScript and come from other languages such as Java, there are primarily three things that may seem very different:

- Variables scope
- Types
- Objects and inheritance

and it's actually the three things that make programming in JavaScript much different than, for example, Java programming. As mentioned, JavaScript is an interpreted language and JavaScript code is sent to the browser along with the HTML code. The browser then has a built-in interpreter that interprets and executes the code's statements. The execution ends when all the code is executed correctly or when the interpreter comes to a statement that fails. It is not, therefore, as a Java program, where the entire program is translated to Java Byte Code before the program can be run by the Java runtime system. JavaScript is a *type-weak* language, where variables should not be declared by a particular type. It is not the same as that the language is typeless, but it means that a variable can change type and that the current type is determined by the value of the variable and how the variable is used. There are a number of rules for the language's type compatibility, rules that are quite complex and some of the things that I will explain below. JavaScript is not object-oriented in classical sense, but conversely, objects play a major role in the language. However, compared to Java, for example, there is a very big difference in how to create and inherit objects. Perhaps the biggest difference between JavaScript and other languages are functions, since functions are actually objects that may have properties and methods.

The aim of this section is to illustrate some of these many concepts without having all the details, but to such an extent that you can use JavaScript in connection with web application development. When the browser parses a HTML document, it builds a tree consisting of the document's elements. The tree is called DOM (for *Document Object Model*), and the purpose of JavaScript is accurately to modify this tree.

The DOM tree

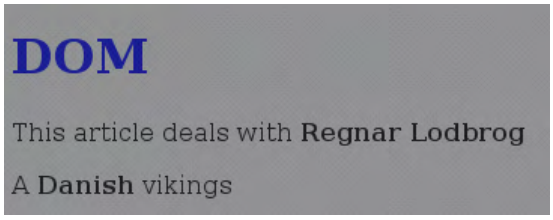
JavaScript can be placed anywhere in the document, but when the document is loaded, the browser parses the document and builds the DOM tree. If the browser meets any JavaScript, it will also perform it, and therefore it may only work if the element that the script refers to is already part of the DOM tree. Consider the following example:

```
<!DOCTYPE html>
<html>
 <head>
  <title>Simple HTML5 document</title>
  <style type="text/css">
   h1.blueStyle {
    color: blue;
   }
   span.boldStyle {
    font-weight: bold;
   }
   p.redStyle {
```

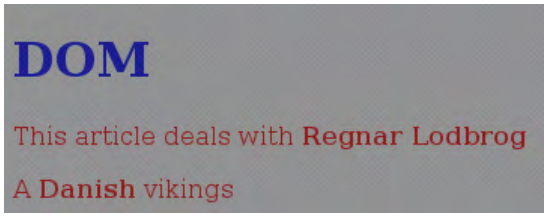
```
    color: red;
   }
  </style>

  <script>
   function print(name) {
    document.write("<p>" + name + "</p>");
   }
  </script>
</head>
<body>
  <h1 id="idHeader" class="blueStyle">DOM</h1>
  <p id="idParagraph">This article deals with <span class="boldStyle">
   Regnar Lodbrog</span></p>
  <p>A <span class="boldStyle">Danish</span> vikings</p>
  <script>
   alert(document.title);
   alert(document.getElementById("idParagraph").innerText);
   var pars = document.getElementsByTagName("p");
   for (var i = 0; i < pars.length; ++i) {
    pars[i].className = "redStyle";
   }
   alert(pars.length);
```

```
    var spans = document.getElementsByTagName("span");
    for (var i = 0; i < spans.length; ++i) {
     spans[i].style.color = "green";
    }
    var classes = document.getElementsByClassName("boldStyle");
    for (var i = 0; i < classes.length; ++i) {
     classes[i].style.fontStyle = "italic";
    }
  </script>
  <div id="names">
   <p>Svend</p>
   <p>Knud</p>
   <p>Valdemar</p>
  </div>
  <script>
   var nodes = document.getElementById("names").childNodes;
   print(nodes.length);
   for (var i = 0; i < nodes.length; ++i) {
    print(nodes[i].nodeName);
   }
   var parent = document.getElementById("names");
   for (var node = parent.firstChild.nextSibling; ; node = node.nextSibling) {
    print(node.innerText);
    node = node.nextSibling;
    if (node == parent.lastChild) {
     break;
    }
   }
  </script>
 </body>
</html>
```

The example should show what happens when the browser loads a document with JavaScript and how to refer to an item in the DOM hierarchy. Note first that a style sheet has been defined in the header. In addition, a script element has been defined that defines a single JavaScript function called *print()*. The function prints a name as a paragraph, which means that the function inserts a paragraph element in the place where the function is called. The browser will not perform the function at this location, it is only a definition of a function. The *body* part starts with a *h1* element and two paragraphs, and the browser starts displaying these items

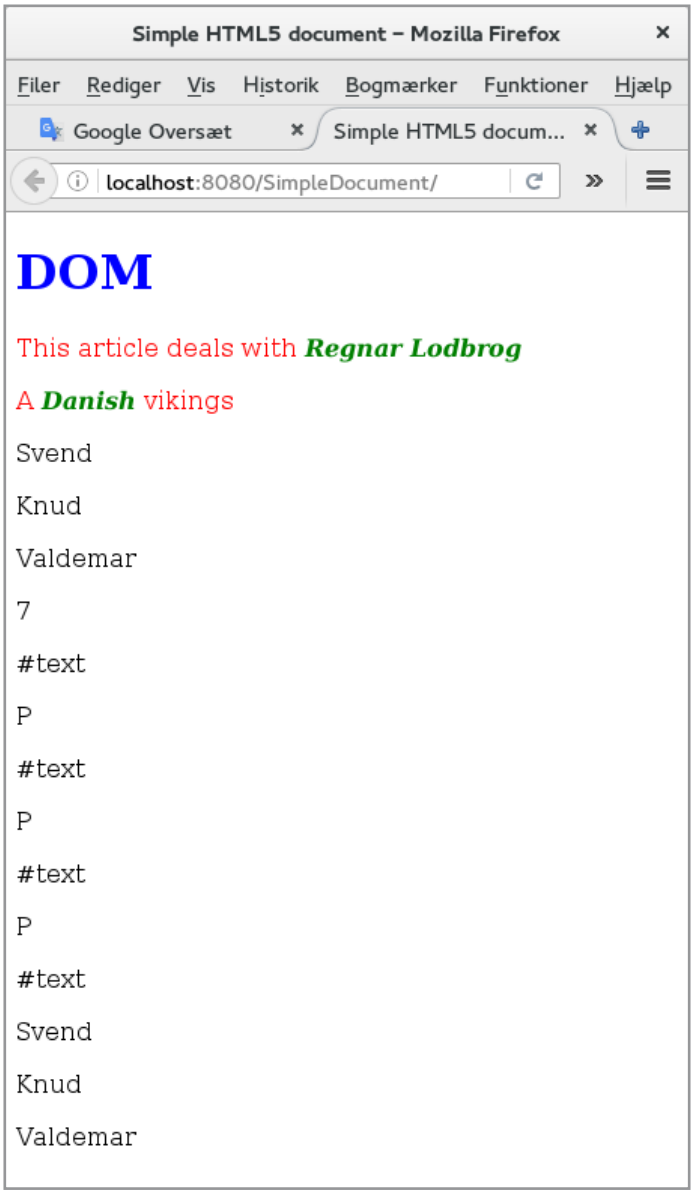


It's not that much mystery in it, but after the browser has interpreted the above elements, a *script* part comes, and here an *alert()* that shows a text. This means, as shown in the previous example, that the browser stops and waits for the user to click on *OK*. When that happens, the browser continues to process the document and the next element is another *alert()* that shows the text in the first paragraph. You should note how to refer to an element with a specific ID and how to refer to the element's text with *innerText*. The *alert()* shows the entire text, including the text in the *span* element, but the last part is not bold, as JavaScript does not interpret HTML. When you click *OK*, the browser continues to interpret and execute JavaScript statements. Here is *pars* an array of all *p* elements – at that time there are 2 – and the subsequent loop sets their *class* to *redStyle*, after which the text becomes red – also the text in the 2 *span* elements. Once that happens, a new *alert()* is performed, which shows the length of the array *pars*, and the primary purpose is to have the parser stop so that you can see that all the text in the two paragraphs is red:



When you click *OK*, the browser continues to perform JavaScript. *spans* is similarly an array with all *span* elements (there are 2) and the following loop sets the text to green.

The last part of the script starts by determining an array *classes* that contain all elements with the class *boldStyle*. Next, the font for these elements is italicized, and the result is that the browser shows the following:



The three names of Danish kings are simple HTML paragraphs, and they help to show that the browser first inserts these elements into the DOM tree after the first script block is completed. Then follow another script block. Here, *nodes* are an array with all child nodes to a *div* element. There are 7 child nodes that were not what you would expect, but the reason is that the plain text between the individual paragraphs also counts as a node. Next, a variable *parent* is created that refers to the *div* element. The subsequent loop iterates over all child nodes to *parent* starting from the first node. The loop skips the blank text elements and prints the text for paragraph elements. Please note that the loop stops with a break when you get to *lastChild*.

There are a few things that you should notice. JavaScript uses the dot notation to refer to objects' properties and methods – just as it is known from Java. In addition, please note that an *alert()* stops the browser's parsing of the document, and *alert()* may therefore be a useful tool for debugging JavaScript.

Objects in JavaScript

JavaScript is an object-oriented language, but there is nothing similar to a class, and the only method of constructing new objects is by prototyping, which is a form of copying an existing object. In this way, a new object is created, which one can best think of as a copy of another object. A bit more precisely, any object in JavaScript has a property called *prototype*, and this property refers to all properties (properties and methods) inherited from the parent object. This means that if you try to refer to a property or method of an object, the interpreter will first check if that property is defined for the object in question. If this is not the case, the object to which *prototype* refers is checked, and neither is the property found in the hierarchy until it reaches an object where the property is defined or the interpreter can conclude that the object is *undefined*. Properties and methods can be overridden, and any overrides will affect the object where the override is defined and down in the inheritance hierarchy. It is best to think of this form of inheritance as a simple list, where *prototype* refers to the previous (parent) element. An important consequence of this inheritance is that a change of property of an object is important for all objects down in the hierarchy hierarchy.

The following example should illustrate some of these concepts, and especially the syntax for how to create objects:

```
<!DOCTYPE html>
<html>
 <head>
  <title>Prototypal inheritance</title>
  <script>
   var obj0 = {
    a: 23,
    b: 29
   }

   var obj1 = Object.create(obj0);
   var obj2 = Object.create(obj1);
  </script>
 </head>
 <body>
  <h1>Prototypal arv</h1>
  <script>
   alert(obj2.a);
   obj0.a = 19;
   alert(obj1.a);
   alert(obj2.a);
   obj1.c = 41;
   alert(obj0.c);
   alert(obj1.c);
   alert(obj2.c);
   obj0.toString =
    function () { return this.a + this.b + this.c; }
   alert(obj1);
   alert(obj0);
  </script>
 </body>
</html>
```

In the document's header is a *script* block that creates three objects. The first is *obj0* and has two properties with the values 23 and 29 respectively. Next, two objects *obj1* and *obj2* are created, both of which are extensions of obj0, but at that time the three objects have the same properties.

The *body* part starts rendering an *h1* element, but otherwise the rest is a *script* block. Next, an *alert()* is used to show the value of the property *a* at obj0 and the result is a popup that displays the value 13. Next, *a* is changed to 19 and then the value of *a* for *obj1* and *obj2* are shown, which in both cases is 19, corresponding to that *obj1* and *obj2* inherit the value from *obj0*. As a next step is assigned a property *c* to *obj1* with the value 41. Here you should note that you can immediately create a property for an object and assign it a value, and this property is not known from the parent object. Therefore, the three next *alert()* statements will show *undefined*, 41 and 41, since *c* is not known from *obj0*, but from *obj1* and *obj2* (*obj2* inherit *obj1*).

obj0 inherits an empty top object called *Object* and thus particular has a *toString()* method. Therefore, both *obj1* and *obj2* have a *toString()* method and the next statement overrides *toString()* for *obj0*. The two last *alert()* statements will show 89 (the sum of the three properties *a*, *b* and *c*, known from *obj1*) and *NaN*, since *toString()* can not perform the function from *obj0*, where *c* is not known.

Scope

In JavaScript, there are slightly different rules for the scope of variables than in Java, since variables have *functional scope*. That is to say, a variable defined in a function is known only in the function, but in return throughout the function wherever it is defined. The variable is also known in any nested functions. You should also be aware that JavaScript uses *hoisting*, which means that a variable can be used anywhere in the function where it is created – even before it is defined. The reason is that the interpreter starts creating variables as that the variables all were defined at the start of the function. However, you must note that a variable at that time is not necessarily assigned a value.

A variable can also have *global scope*, which is the case if it is defined outside of a function, and such a variable can be used anywhere in the document. Normally, you write *var* in front of a variable when created, but it is actually not necessary (but can be recommended). A variable defined without *var* has automatic global scope wherever it is created.

In most programming languages, a variable is removed when the program leave the scope where the variable is defindes, and it is also the case in JavaScript. That is, variables are created when the function in which they are defined is performed. It is at least the basic rule, but it does not necessarily apply. The reason is that in JavaScript a function is itself an object, and a function can thus return a function, and for example it can return an internal function. This inner function can refer to a variable in the enclosing function, and thus there may be a reference to a local variable after a function has been completed. If so, the variable will not be removed, a fact called *closures*.

Basically, in JavaScript, there are not so many challenges regarding variables' scope (maybe just except closures), and the following document, called *ScopeDocument*, should illustrate the most important scope rules:

```
<!DOCTYPE html>
<html>
 <head>
  <title>Scope</title>
  <script>
   var c = 4;

   function test1() {
    var a = 1;
    var b = 3;
    function test2() {
     var a = 2;
     alert(a);
```

```
      alert(b);
      c = 5;
      alert(c);
     }
     test2();
     alert(a);
    }

    function power2() {
     var n = 1;
     function next() {
      n *= 2;
      return n;
     }
     return next;
    }
  </script>
 </head>
 <body>
  <h1>Test scope</h1>
  <script>
    test1();
    alert(c);
 // alert(a);
    var f = power2();
    var g = power2();
    document.write("<p>");
    for (var i = 0; i < 5; ++i) document.write(f() + " ");
    document.write("</p>");
    document.write("<p>");
    for (var i = 0; i < 5; ++i) document.write(g() + " ");
    document.write("</p>");
    document.write("<p>");
    for (var i = 0; i < 5; ++i) document.write(f() + " ");
    document.write("</p>");
    document.write("<p>");
    for (var i = 0; i < 5; ++i) document.write(g() + " ");
    document.write("</p>");
  </script>
 </body>
</html>
```

If you start in the *script* block in the header of the document, a variable *c* is defined. It is defined outside of all functions and has global scope and is known everywhere. Next, a function *test1()* is defined with two local variables *a* and *b*. They are known only in the function *test1()* and are thus also known in principle in the nested function *test2()*. However, only *b* applies, since *test2()* also defines a local variable *a* that hides the variable *a* defined in *test1()*. The function *test2()* shows the value of its local variable *a* and the local variable *b* from *test1()*. In addition, *test2()* initializes the global variable *c* and shows its value. The function *test2()* is called from the external function *test1()*, which finally shows the value of its local variable *a*.

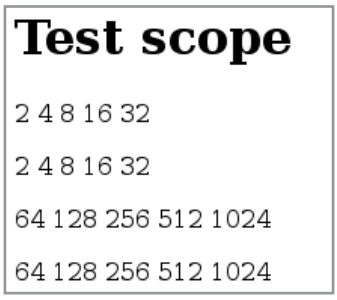
There is also defined a function *power2()*, which will illustrate the term *closure*. *power2()* has a local variable *n*, and it returns a function *next()* that uses (refers) to this variable. This means that there is a reference to the variable *n* after *prowe2()* terminates and hence *n* is not removed – that is called *closure*.

The *body* part starts with an *h1* element, after which *test1()* is called. Note that this means that the internal function *test2()* is also performed. Next, the value of the global variable *c* is displayed. If you try to show the value of the variable *a*, it is not defined in this location, and rendering of the document would be interrupted.

As the next step, two references *f* and *g* are defined for *power2()* and the result is that the function *power2()* is performed twice, and partly that *f* and *g* refer to a function with each their copy of the local variable *n*. It is illustrated by the following loops that perform the functions that *f* and *g* refers to, and thus each works on their copy of the local variable *n*. If the page is opened in the browser, the following popups will appear and shows the values:

```
2
3
5
1
5
```

and finally the page shows



Data typer

When you meet JavaScript, it may seem that the language is typeless, but it is not the case. It is only a matter of the interpreter deciding the type of variables from the context, which means, among other things, that a variable can change its type. In the next section on syntax, I will describe the type concept in JavaScript in more detail, but basically there are four types:

1. *Boolean*, which are variables or expressions whose value is *false* or *true*
2. *Number*, which is numbers and everywhere is 64 bits floating numbers
3. *String*, to strings
4. *Object*, which are collections consisting of properties and methods, which cover anything other than the above three types

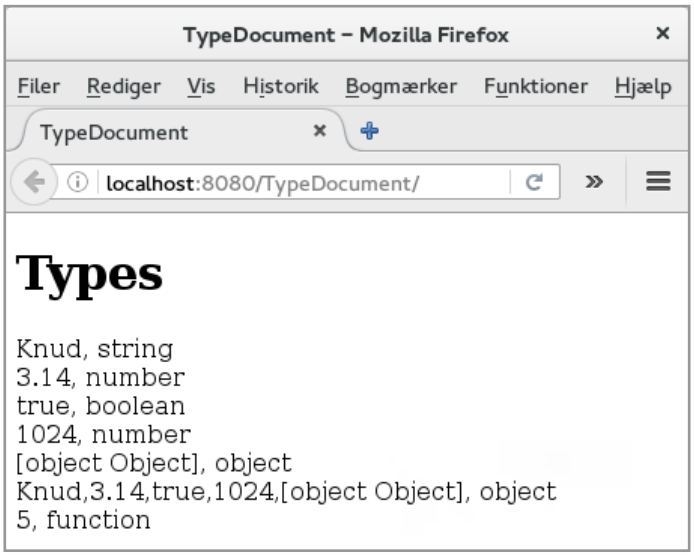
This type system involves several things, including, among other things, that there constantly must occurs type conversions. It's far from simple and something that I want to look at in the next section, but below is an example (*TypeDocument*) that can illustrate a bit about types and JavaScript:

```
<!DOCTYPE html>
<html>
 <head>
  <title>TypeDocument</title>
  <meta charset="UTF-8">
  <script>
   function add(a, b) {
    return a + b;
   }
  </script>
 </head>
 <body>
  <h1>Types</h1>
  <script>
   var arr = ["Knud", 3.14, 3 == "3", 1024, {}];
   for (var i = 0; i < arr.length; ++i)
    document.write(arr[i] + ", " + (typeof arr[i]) + "<br/>");
   document.write(arr + ", " + (typeof arr) + "<br/>");
   document.write(add(2, 3) + ", " + (typeof add) + "<br/>");
  </script>
 </body>
</html>
```

Note that in the header is defined a simple function with two parameters. The parameters have no type and all you can see is that the function performs the plus operator on the two parameters and returns the value. The script block in the body section creates an array of 5 elements of the following types:

1. the first is a *String*
2. the second is a *Number*
3. the third is a *Boolean* (that is *true* due to special conversion rules in JavaScript)
4. the fourth is a *Number*
5. the last is an *Object*

You should note that the array *arr* itself is an *Object*. The next statement, which is a *for* loop, prints the array's elements as well as their types, and the next statement again does the same for the array. Note the *typeof* operator, which determines the type of a variable. Finally, there is the last statement that performs the function *add()* and prints its type. The type of function is *Object*, but you should note that *typeof* displays it as a *function*. If the page is shown in the browser the result is:



Patterns

In this section I have shown some basic concepts regarding JavaScript, and of course there are many more details, but the above should be sufficient to illustrate that JavaScript is a much different language than Java. In the time that JavaScript has existed, the language has developed a lot, and at the same time there have been several patterns for how to write JavaScript code. In particular, it appears that a function is an object, and when a function defines a scope, it allows functions to implement a form of data encapsulation. I would like to illustrate that with the following example.

```
<!DOCTYPE html>
<html>
 <head>
  <title>Patterns</title>
  <script>
   var triple = (function () {
    var t = 0;

    function swap1() {
     t = public.a;
     public.a = public.b;
     public.b = t;
    }

    function swap2() {
     t = public.b;
     public.b = public.c;
     public.c = t;
    }

    public = {};
    public.a = 0;
    public.b = 0;
    public.c = 0;

    function build() {
     return "<p>" + public.a + ", " + public.b + ", " + public.c + "</p>";
    }

    public.print = function () {
     document.writeln(build());
    }
```

```
   public.sort = function () {
     if (public.a > public.b) swap1();
     if (public.b > public.c) swap2();
     if (public.a > public.b) swap1();
    }

    return public;
   })()
 </script>
</head>
<body>
 <h1>Patterns</h1>
 <h3>The least number</h3>
 <p>
  <script>
   document.write((function(a, b) { return a < b ? a : b })(5, 3));
  </script>
 </p>
 <h3>Is it a prime?</h3>
 <p>
  <script>
   document.write((function(n) {
    var m = Math.sqrt(n);
    return n + " is " + ((function() {
     if (n === 2 || n === 3 || n === 5 || n === 7) return true;
     if (n < 11 || n % 2 === 0) return false;
     for (var t = 3; t <= m; t += 2) if (n % t === 0) return false;
     return true;
    })() ? "a prime number" : "not a prime number");
   })(123));
  </script>
 </p>
 <p>
  <script>
   document.write((function (n) {
    var t1 = 0;
    var t2 = 1;
    while (n-- > 1) {
     var t3 = t1 + t2;
     t1 = t2;
     t2 = t3;
    }
    return t2;
   })(30));
```

```
triple.print();
triple.a = 7;
triple.b = 3;
triple.c = 5;
triple.sort();
triple.print();

triple = (function (tpl) {
 tpl.sum = function () {
  return tpl.a + tpl.b + tpl.c;
 }
 return tpl;
})(triple);
document.writeln("<p>" + triple.sum() + "</p>");

triple.sub = (function () {
 public = {};

 public.max = function () {
  var m = triple.a;
  if (triple.b > m) m = triple.b;
  if (triple.c > m) m = triple.c;
  return m;
 }
```

```
      return public;
     })();

     triple.a = 19;
     triple.b = 13;
     triple.c = 17;
     document.writeln("<p>" + triple.sub.max() + "</p>");
    </script>
   </p>
  </body>
 </html>
```

There are many details, and I will start with the script block in the body section. Generally, a function encapsulates a code that is executed when the function is executed, but puts parentheses without the entire function as

```
(function() { … })();
```

the function, as here an anonymous function, will be performed immediately. As an example, the statement

```
document.write((function(a, b) { return a < b ? a : b })(5, 3));
```

insert the least of numbers the 3 and 5 into the document by performing an anonymous function. Of course, that does not make much sense, but if the parameters instead were variables, it could make sense. Thus, it is very common to insert the value of an expression by evaluating a function. Immediately it seems a bit strange to write a method in this way, but it is quite useful, although it may result in code that is difficult to read what the next expression shows:

```
<script>
  document.write((function(n) {
  var m = Math.sqrt(n);
  return n + " is " + ((function() {
   if (n === 2 || n === 3 || n === 5 || n === 7) return true;
   if (n < 11 || n % 2 === 0) return false;
   for (var t = 3; t <= m; t += 2) if (n % t === 0) return false;
   return true;
  })() ? "a prime number" : "not a prime number");
 })(123));
</script>
```

The script determines whether 123 is a prime number or not and inserts a corresponding text in the document. The example should show that it can be done by evaluating an anonymous function, and partly that JavaScript can use loops and conditions with the same syntax as in Java, which is further elaborated in the following section. The code is not so easy to read, but the text that is inserted is the value of an anonymous method with one parameter *n*. This method has a local variable *m*, which is initialized with the square root of *n*. Also note that it is written in the same way as in Java. The anonymous function returns a value, which is a text, but the value is determined by an inner anonymous function without parameters, a function that, using the usual prime algorithm, determines whether *n* is a prime. In this case, the use of anonymous function is a bit exaggerated, but it is in some way common when writing JavaScript.

If you look at the header section then it uses an anonymous function to embed variables and functions in a namespace, and the variables and functions will become local to this namespace. An anonymous method used in that way is usually called a *module*, and *triple* is thus an example of a module. Modules can be perceived as a pattern to implement a form of data encapsulation in JavaScript, and among other things, names are local to the module, so you are independent of which names others may have used.

If you examine the module *triple*, *t* is a local variable in the anonymous function and thus in the module, and it will thus work as a private variable. Similarly, the two functions *swap1()* and *swap2()* are private functions for the module *triple*. The module further defines a variable called *public*. Here, you should note that this variable is an *Object* (which has no properties at present), and it has a global scope and is thus known outside of the module. After the object has been created, three properties are defined, which from the outside act as public properties by the object *triple* (explanation follows below). Next, another private function *build()* is defined, and then two functions that are functions of the object *public*, which will thus act as public functions of the object *triple*. When they do, it is in the same way as for the three properties of the object *public*, because the anonymous function is returning the object *public*.

Then there is the last script block in the *body* section, which should primarily show how the module *triple* is used, but first the block shows another use of an anonymous function that is performed without being called explicit. The function determines the 30th Fibonacci number. The remaining part of the block uses the object (module) *triple* and its public properties in terms of properties and function. First, the *triple* object's *print()* method is called that just prints three 0-values (the three properties). Next, the three properties are assigned a value and the method *sort()* is executed, after which the method *print()* is executed again.

You can use the syntax with an anonymous function to expand a module with a new feature. Below is how to expand *triple* with a new function:

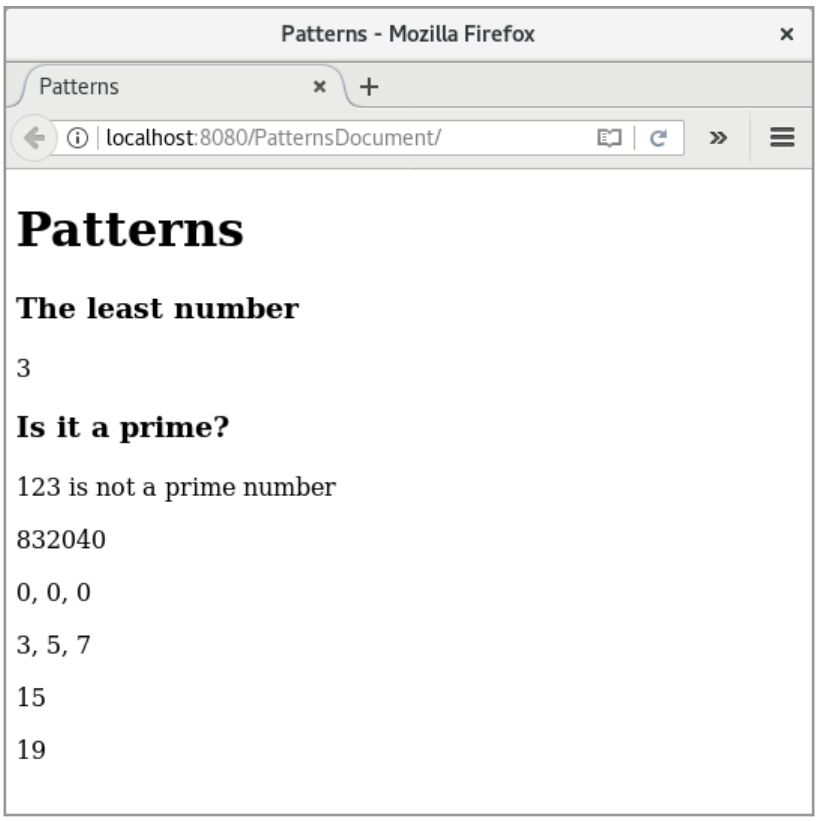
```
triple = (function (tpl) {
 tpl.sum = function () {
  return tpl.a + tpl.b + tpl.c;
 }
 return tpl;
})(triple);
```

An anonymous function has a parameter *tpl*, and the function interprets it as an object and expands it with a new property, which is a function (returning the sum of three properties). Finally, the anonymous function returns the parameter *tpl* (which has now got a new function), and is the anonymous function called with *triple* as the actual parameter and *triple* is assigned the return value, the result is that *triple* is expanded with a new function called *sum()*.

You can also by the same pattern expand *triple* with a sub-module (*triple* is an object). Here it is a sub module sub, which expands with a new function.

```
triple.sub = (function () {
 public = {};

 public.max = function () {
  var m = triple.a;
  if (triple.b > m) m = triple.b;
  if (triple.c > m) m = triple.c;
  return m;
 }
 return public;
})();
```

The result of all is that functions can be used to define a module term which is a namespace that can have private properties and provide public properties and methods available. If you perform the program, the result is as shown below:



4.2 BASIC SYNTAX

The following deals with the basic JavaScript syntax, and although the syntax is illustrated with examples, the review has to some extent character of a reference. Much of the following has already been mentioned in the previous section.

Basically, JavaScript uses the same syntax as the language Java or language inspired by C. Therefore, I would like to write JavaScript in much the same way as I have written Java code and thus apply the same rules for blocks ({ and }) and the same rules for indentation. Regarding blocks, many choose (and also many development tools) to write

```
function f() {
 // kode
}
```

instead of

```
function f()
{
 // kode
}
```

I will follow that convention – primarily because my development tool does. There is also a tradition for (inspired by Java) that let function names start with a lowercase letter, and I will also adhere to that.

In JavaScript, a statement is terminated as in Java with a semicolon, but it is actually not necessary, as most interpreters are able to set the missing semicolon itself. Of course, if you do not terminate statements with a semicolon, it requires that you follow a number of rules for how the code is written, but newer interpreters are very good to automatically determining where statements end – they simply inserts a semicolon indirectly where it is necessary for the code to makes sense. However, I want to put semicolons everywhere, partly because I am used to it from other languages and partly because I think it increases readability, but even more importantly, missing semicolons may in some cases result in the interpreter's misunderstanding of the code.

Expressions and statements

An expression is simply a code that has a value. Examples include:

```
23
"Hello World"
33 + 29
(13 + 17)/(5 + 5)
Math.sqrt(100)
(str === "Svend") && (tal > 10)
```

A statement, on the other hand, is a sequence of expressions that result in some action, for example an assignment statement. A statement can particular be composed as a block of statements, which is an important term in cases of control statements. Note in particular that a block is not necessarily a statement. For example is an object

```
{
 t1: 23,
 t2: 29
}
```

not a statement but an expression.

Operators works on expressions and JavaScript has a good deal of the way the same operators as Java. The main operators are assignment, calculating operators and comparison operators. Operators also use the same precedence rules, known from other languages, including Java. Below is a table of all operators in JavaScript and their precedence. L/R means that operators evaluate from left to right, while R/L means that operators evaluate from right to left.

Operator	Precedence	Associativity	Remark
`()`	0		Parenthesis
`. []`	1	L/R	Member and index
`new`		R/L	
`()`	2	L/R	Function call
`++ --`	3		
`! ~`	4	R/L	
`+ -`		R/L	Sign
`typeof void delete`		R/L	
`*/%`	5	L/R	
`+ -`	6	L/R	Addtion and subtraktion
`<< >> >>>`	7	L/R	
`< <= > >= in instanceof`	8	L/R	
`== != === !==`	9	L/R	
`&`	10	L/R	
`^`	11	L/R	
`|`	12	L/R	
`&&`	13	L/R	
`||`	14	L/R	
`?`	15	R/L	
`yield`	16	R/L	
`= += -= *= /= %= <<= >>= >>>= &= ^= !=`	17	R/L	
`,`	18	L/R	

Variables

You declare variables with or without the use of the word *var*:

```
var t1 = 23;
t2 = 29;
```

In both cases, a variable has been created, and the difference is important for the variable's scope. It is allowed to declare several variables in the same statement

```
var obj = {}, counter = 0, str = "Knud", flag = true;
```

JavaScript has functional scope. This means that if you create a variable with *var*, its scope is limited to the function where the variable is created or for internal functions. It can be illustrated with the following function:

```
function f()
{
 var t = "Here";  // t's scope is the function f and other scopes in f

 function g()     // g() creates a new acope in the function f()
 {
  alert(t);       // t is known here
 }
 g();
}

f();              // will alert "Here"
alert(t);         // results in an error, when t is not known
```

The scope where a variable is defined is often called for its *local scope*. When referring to a variable, the runtime system will search for the variable in the current scope. If the variable is not found here, the system will search for the variable in the containing scope, and this will continue until the system finds the variable or comes to the top scope, commonly called the *global scope*. The process is sometimes called *scope chain lookup*. Variables that are not declared in a function will always have global scope, and the same applies to variables that are declared without the use of *var*.

Variables are always created at the start of a scope – even if the declaration itself is first written later in the code:

```
function f()
{
 alert(t);
 var t = 2;
}
```

If you perform this function, it will display *undefined* with the *alert()*. The variable *t* is thus created, but it is not initialized. However, if you remove the word *var*, you will get an error as you refer to a variable that has not been created. This corresponds to the interpreter's perception of the above function as:

```
function f()
{
 var t;
 alert(t);
 t = 2;
}
```

As it can sometimes lead to hard-to-see results, it is recommended that you consistently create and initialize variables at the beginning of functions.

When a function terminates, its scope will disappear and at the same time, the runtime system will remove all local variables that the function has created. If the function is called again, its local variables will be re-created. It is at least the normal behavior, but consider the following document:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <title></title>
 <script>
  function f()
  {
   var t = 1;
   function g()
   {
    alert(t++);
   }
   return g;
  }
 </script>
</head>
<body>
 <script>
  var h = f();
  h();
  h();
 </script>
</body>
</html>
```

If it opens in the browser, the first *alert()* will show 1 and the next *alert()* show 2. If you consider the function *f()*, it has a local variable *t* that is assigned the value 1. The internal function *g()* shows the value of this variable, after which it is counted up with 1. The function *f()* returns the internal function *g()*. Note that it is legal, since a function is specifically an object. In the body section, the function *f()* is performed and its return value is stored in the variable *h*, which now refers to a function and the function *f()* terminates. Next, *h()* is performed, which means that it is the function *g()* that is performed as with an *alert()* shows the value 1, which is the value of the local variable *t* in *f()*. When *h()* is executed the second time, it will display 2. There it is still the value of *t* and you can see that *t* is not initialized again and thus not removed after *f()* is terminated. The reason is that there is an indirect reference to *t* via *g()*. When *f()* terminates, *t* is not removed since *h* refers to *g()* which refers to *t*. That a local variable in this way will not be removed after the function that creates the variable terminates is called closures.

Types

In JavaScript, you should not specify any data type when creating a variable. It is said that it is a type-weak language. However, it is not the same as JavaScript does not have types and there are basically 4 types:

- *Boolean*, which are variables or expressions that are either *false* or *true*
- *Number*, which are numbers, that are 64 bits floiting points
- *String*, which are random sequences of characters
- *Object*, which are families of properties and methods

That the language is type-weak means that the runtime system based on variables and expression values automatically determines the type. This means, in particular, that the type of a variable can be changed over its lifetime and is always determined by the value of the variable. For example the following code

```
var arr = ["Knud", 47, true, {}, f];
for (var i = 0; i < arr.length; i++) alert(typeof arr[i]);
alert(typeof arr);
```

where the function *f()* is as above, will alert the following type names (the operator *typeof*)

- *string*
- *number*
- *boolean*
- *object*
- *function*
- *object*

Here you should note that for *f*, the result is *function*, even if the type is formally *object*, and that the type of an array is an *object*.

The value of a variable can be a *primitive* where there are the following options

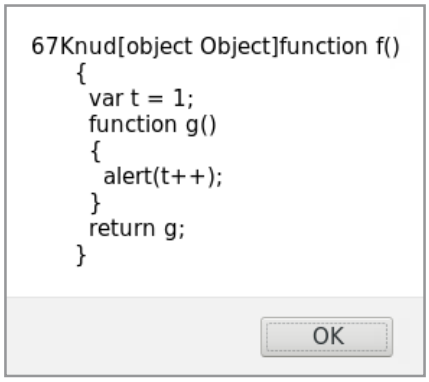
- *true* or *false* and then a *boolean*
- a *number* and then all possible numbers
- a *string*
- *null*, that means that the variable has no value
- *undefined*, that means that the variable is defined, but not initialized

Everything that is not a primitive is an *object*, and it therefore special means arrays and functions.

Consider the following code where the function *f()* again is as above:

```
var arr = [47, true, "Knud", {}, f];
var v = 19;
for (var i = 0; i < arr.length; i++) v += arr[i];
alert(v);
```

If you perform this code, you get the result:



The variable *v* has the value 19, which has the type *number*. When you loop over the array, you first determines the sum of *v* and the value 47. They are two numbers and the result is 66. Next, add *true*, which is a *boolean*, and here occurs an automatic type conversion to a number where the value *true* is converted to 1 and the result of the sum is 67. In the third iteration, the plus is performed on a *number* and a *string*, and this time occurs an automatic conversion to a string for both operands. In the fourth iteration, the operator is executed on a *string* and an *object*, and the result is a string concatenation. The same goes for the last time, since *f()* is also an *object*, but the string – the result of *toString()* – is this time the code of the function itself.

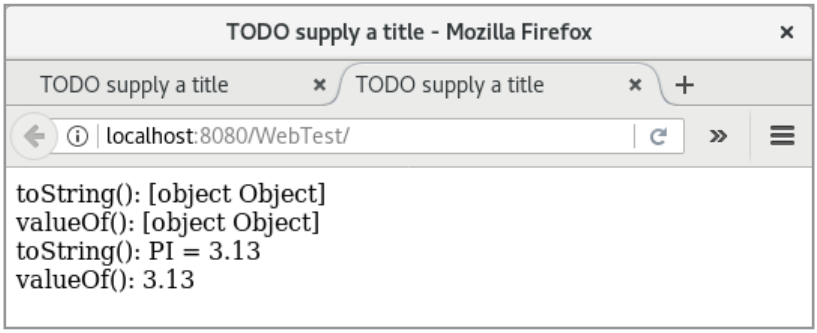
The example should show that as JavaScript is type-weak, an expression may well consist of different types of operands, and consequently implicit type conversions are always ongoing. This can naturally lead to unexpected results, and the rules are also complex. Basically, the following applies.

Note first that a variable of the type *object* has two methods *toString()* and *valueOf()* (which can be overridden), where the first returns the value of the object as a *string*, while the other returns the value as a *primitive*. For example, if you consider the following code:

```
<!DOCTYPE html>
<html>
 <head>
  <title>TODO supply a title</title>
  <meta charset="UTF-8">
  <script>
   var obj1 = {
    a: 2,
    b: 3
   };
   var obj2 = {
    x: 3.13,

    toString: function() {
     return "PI = " + this.x;
    },

    valueOf: function() {
     return this.x;
    }
   }
  </script>
 </head>
```

```
 <body>
  <script>
   document.write("toString(): " + obj1.toString() + "<br/>");
   document.write("valueOf(): " + obj1.valueOf() + "<br/>");
   document.write("toString(): " + obj2.toString() + "<br/>");
   document.write("valueOf(): " + obj2.valueOf() + "<br/>");
  </script>
 </body>
</html>
```

there are defined two objects where the latter overrides *toString()* and *valueOf()*. Opening the code in the browser is the result:



Here you can see that *toString()* and *valueOf()* as default return the string *[object object]*, but otherwise the value of the overridden methods.

For automatic type conversion, JavaScript uses three internal methods called *toPrimitive()*, *toNumber()* and *toBoolean()*. *toPrimitive()* has an argument and returns a *primitive* following the algorithm:

1. if the argument is an *object* returns *valueOf()* if it is a *primitive*, and else returns *toString()* if it returns a primitive and else reports an error
2. if the argument is a primitive return the argument

toNumber() also has an argument and returns a number according to the following algorithm:

1. if the argument has the type *Number* returns the argument
2. if the argument has the type *Boolean* returns 1, if the value is *true* and else 0
3. if the argument is *Null* returns 0
4. if the argument has the type *object* returns *toNumber(toPrimitive(argument))*
5. it the argument is *undefined* returns *NaN*
6. if the argument is a *string* returns the value of *parseInt()*, if the argument can be parsed to a *Number* and else *NaN*

Allso *toBoolean()* has an argument and returns a *boolean* according to the following algorithm:

1. it the argument has the type *boolean* returns the argument
2. if the argument is *Null* returns *false*
3. if the argument is *undefined* returns *false*
4. if the argument has the type *object* returns *true*
5. if the argument has the type *number* returns *false* if the value is 0 or *NaN* and else *true*
6. if the argument has the type *string* returns *false* if the string is empty and else *true*

As can be seen from these rules, it may be difficult to figure out the value of an expression, but in practice it rarely presents the big problems, since the value will usually be the expected, almost the value of a condition (an expression of the type *Boolean*) as well sometimes may results in a value other than expected, and here it is especially the operator == which causes problems. For this operator, the following table is used:

arg1	arg2	result
Null	Undefined	true
Undefined	Null	true
Number	String	arg1 == toNumber(arg2)
String	Number	toNumber(arg1) == arg2
Boolean	any	toNumber(arg1) == ang2
any	Boolean	arg1 == toNumber(arg2)
String or Number	Object	arg1 == toPrimitive(arg2)
Object	String or Number	toPrimitive(arg1) == arg2

A classic example of the problems that comparison may cause are the following code:

```
<script>
 if ("Hello") {
  alert("Hello" == true);
  alert("Hello" == false);
 }
</script>
```

If you try the code, you will get two *alert()*, both of which will show *false* and it is not entirely obvious. The condition for *if* has the value *true*, which immediately follows from the *toBoolean()* algorithm. The first *alert()* will apply the third last row in the above table and will thus evaluate

```
"Hello" == toNumber(true)
"Hello" == 1
toNumber("Hello") == 1
NaN == 1
```

that is *false*. The last *alert()* essentially makes the same thing:

```
"Hello" == toNumber(true)
"Hello" == 0
toNumber("Hello") == 0
NaN == 0
```

The result is that it is not always easy to find out the result of a comparison with == (or !=). Therefore, the operator === has been introduced (and !==) which simply means comparison without type conversion, and many prefer to use === rather than ==.

Another thing that can cause problems is *null* and *undefined*. *null* means that a variable has no value and you can assign a variable the value *null*. *undefined* means that a variable is not yet assigned a value, and for example, the code will

```
<script>
 var a;
 document.write(a);
 document.write("<br/>");
 a = null;
 document.write(a);
</script>
```

results in

```
undefined
null
```

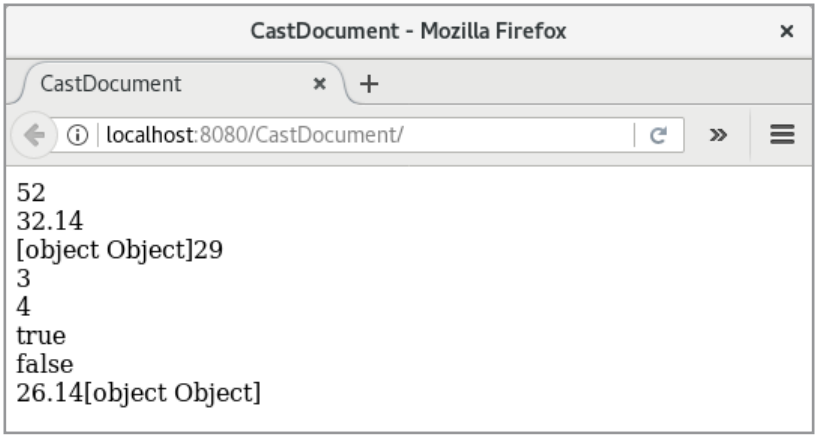
As an example of some of the above conversion rules, you can consider the document:

```
<!DOCTYPE html>
<html>
 <head>
  <meta charset="utf-8" />
  <title>CastDocument</title>
  <script>
   var obj1 = {
    a: 23,
    b: 3.14,
    toString: function () {
     return this.a;
    }
   }
   var obj2 = {
    a: 23,
    b: 3.14,
    valueOf: function () {
     return this.b;
    },
    toString: function () {
     return this.a;
    }
   }
```

```
  var obj3 = {
   a: 23,
   b: 3.14
  }
 </script>
</head>
<body>
 <script>
  document.write(obj1 + 29);
  document.write("<br/>");
  document.write(obj2 + 29);
  document.write("<br/>");
  document.write(obj3 + 29);
  document.write("<br/>");
  var t = 3;
  var u;
  document.write(3 + (t == 2));
  document.write("<br/>");
  document.write(3 + (t == 3));
  document.write("<br/>");
  document.write(3 == "3");
  document.write("<br/>");
  document.write(3 === "3");
  document.write("<br/>");
```

```
    (function () {
     document.write(obj1 + obj2 + obj3);
    })();

  </script>
 </body>
</html>
```

If the document is opened in the browser, you get the result:



Objects

An object is simply a collection of properties that may be of some value. The type of a property can be anything and especially an object, and since a function is an object, an object can also contain functions. You creates an object as follows:

```
var obj1 =
{
 a: 2,
 b: 3,
 c: 5
}
var obj2 = {}
```

where two objects have been created. The first has three properties, while the other is an empty object. In JavaScript, you do not have classes, as you know it from Java, and you can not inherit the same way you know from this language, but in JavaScript you use a form of inheritance called *prototyping* that in short means that you can create objects by expanding existing objects. For example, if you write

```
var obj3 = Object.create(obj1);
obj3.d = 7;
```

an object has been created that inherits *obj1* and expands this object with an additional property. For example, if you performs the following statements:

```
alert(obj3.a + obj3.b + obj3.c + obj3.d);
obj1.a = 11;
alert(obj3.a + obj3.b + obj3.c + obj3.d);
```

the first *alert()* will show 17 and the other 26. Here, you should note that changing the value of the property *a* on the object *obj1* also has an effect on the object *obj3*, which justifies the relationship between *obj1* and *obj3* being a form of inheritance. Technically, it is implemented by the fact that each object has a property called *prototype*. This property refers to properties inherited from the parent object (the object from which the current object is created based on). When you try to refer a property on an object, the interpreter will first search for this property in the current object and if it not find the property the interpreter will search in its parent via *prototype*. This continues until you either find the desired property or can not find it. This chain may be interrupted by overriding a property. As an example is shown another object that expands obj1 using a method:

```
var obj4 = Object.create(obj1);
obj4.f = function (t) { return t * (this.a + this.b + this.c); }
```

Note that although the method *f()* is a property in the object *obj4* which extends *obj1*, the method can not immediately refer to the object's other properties, but it can be solved with *this* as referring to the current object. The following statement

```
alert(obj4.f(2));
```

will *alert()* the value 20. As another example of how to define an object using a method, you can consider the following:

```
var obj5 =
{
 x: 3.14,
 y: 1.41,
 g: function (z) { return (this.x + this.y)/z; }
}
```

If you perform the following statement

```
alert(obj5.g(3));
```

you get the result `1.5166666666666`.

Since *obj1* is as above, you usually refer to the individual properties using the dot notation, but you can also use array notation, which uses the name of the property as an index:

```
alert(obj1.a);
alert(obj1["a"]);
```

Both of these statement will display an *alert()* with the value 2. The main application of the last notation is that it allows to loope over an object's properties without knowing their names:

```
for (var t in obj1) alert(obj1[t]);
```

This statement will alert 2, 3 and 5.

You can create objects in three ways. You can create objects directly as shown above with *obj1* and *obj5*, that is, you directly write the properties that the object should have. You can also create an object with *Object.create()* such as *obj3* and *obj4*, where an object is created as an extension of another object. Finally, you can create objects using a constructor. Consider the following function:

```
function pointConstructor(x, y)
{
 this.x = x;
 this.y = y;
 this.dist = function (p) {
  return Math.sqrt((this.x - p.x) * (this.x - p.x) +
    (this.y - p.y) * (this.y - p.y));
 }
 this.toString = function() { return "( " +
this.x + ", " + this.y + " )"; }
}
```

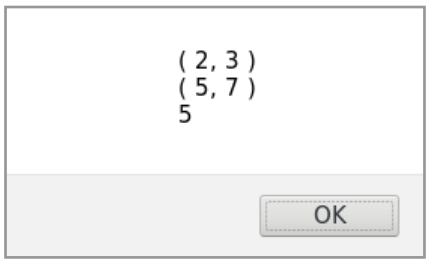
The function creates an object with 4 properties, where the last two are methods. An object is perceived as a point in a coordinate system, and the method *dist()* has a parameter that is to be interpreted as a point. The method returns the distance between the current point and the parameter *p*. The last method is an override of *toString()*. You should note that the method has parameters, but it is not necessary. Below is how to use the constructor method to create two objects:

```
var p1 = new pointConstructor(2, 3);
var p2 = new pointConstructor(5, 7);
```

The following statement

```
alert(p1 + "\n" + p2 + "\n" + p1.dist(p2));
```

will opens a popup as shown belove:



In principle, it does not matter whether an object is created in one way or another and you use the most appropriate way. However, you must note that the constructor way resembles how to create objects in Java.

I will finish this section on objects with an example of a *cup* object representing a 5-cube rafle cup where a cube should be represented by a *cube* object. The example applies in addition more control statements (loops) and also arrays that are dealt with in the next section. The example will also use the module concept described in the previous section. The code is as follows:

```
<!DOCTYPE html>
<html>
 <head>
  <title>CubesProgram</title>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <script>
   var cup = (function() {
    function cube() {
     this.eyes = 1;
     this.roll = function () {
      this.eyes = Math.floor(Math.random() * 6) + 1;
     };
     this.valueOf = function () {
      return this.eyes;
     }
    };

    public = {
     cubes: [new cube(), new cube(), new cube(), new cube(), new cube()],

     toss: function () {
      for (var i = 0; i < this.cubes.length; ++i) this.cubes[i].roll();
     },

     yatzy: function () {
      var t = this.cubes[0].eyes;
      for (var i = 1; i < this.cubes.length; ++i)
       if (this.cubes[i] != t) return false;
      return true;
     },

     toString: function () {
      var str = this.cubes[0].eyes;
      for (var i = 1; i < this.cubes.length; ++i)
       str += " " + this.cubes[i].valueOf();
      return str;
     }
```

```
    };

    return public;
   })();
  </script>
 </head>
 <body>
  <h1>Play Yatzy</h1>
  <script>
   var count = 0;
   do {
    cup.toss();
    document.write(cup + "<br/>");
    ++count;
   }
   while (!cup.yatzy());
  </script>
  <h3>You've got yatzy after <script>document.write(count)</script> attempts</h3>
 </body>
</html>
```

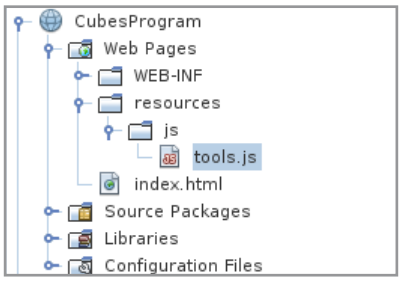
In the header section is defined a module called *cup*, which represented a cup with 5 cubes and three *public* attributes:

1. *toss()*, that is a method, which simulates the use of the cup
2. *yatzy()*, which is a method, that tests where all 5 cube values are the same
3. *toString()*, that is an override of *toString()*

Initially, a constructor method is defined that creates a *cube* object. An object *public* is defined which uses the constructor method to create an array of 5 *cube* objects. The public object has three functions that implement the above methods. The body part simulates using the cup until all cubes have the same value.

EXERCISE 1

Create a copy of the program *CupProgram*. You must then create a folder *resources* and a subfolder *js* and to that folder a JavaScript file:



You must then move all your script code from *index.xhtml* to *tools.js* – but without the script elements. *index.xhtml* must have a link in the header to the JavaScript file. You du that by draging til file name to the header. The result of *index.xhtml* should be:

```
<!DOCTYPE html>
<html>
 <head>
  <title>CubesProgram</title>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <script src="resources/js/tools.js" type="text/javascript"></script>
 </head>
 <body>
  <h1>Play Yatzy</h1>
  <script>
   var count = 0;
   do {
    cup.toss();
```

```
    document.write(cup + "<br/>");
    ++count;
   }
   while (!cup.yatzy());
  </script>
  <h3>You've got yatzy after <script>document.write(count)</script> attempts</h3>
 </body>
</html>
```

Test the program, at it should works as before.

When using JavaScript, you will usually place the code in its own file, as shown in this exercise.

Arrays

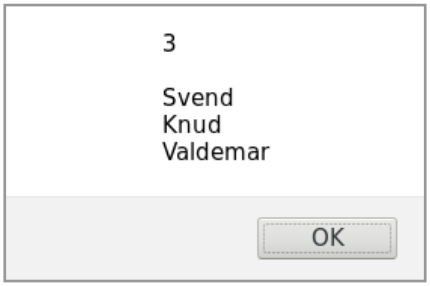
Arrays works immediately in the same way as in other programming languages and with the same syntax known from Java, but there are also important differences. In JavaScript arrays are dynamic and should not be allocated for a certain size. The following function opens a popup that shows the length of an array as well as the array's elements:

```
function show(arr)
{
 var str = arr.length + "\n";
 for (var i = 0; i < arr.length; ++i) str += "\n" + arr[i];
 alert(str);
}
```

If you consider this function, note that syntactically, an array is used in the same way as in Java. An array has a length property, and the individual elements are referenced via an 0-based index.

```
var arr1 = ["Svend", "Knud", "Valdemar"];
```

The statement creates an array of three elements and sends this array to the function *show()*, you receives an alert as shown below:



Since an array has no type, an array can contain elements of different types:

```
var arr2 = ["Gorm", 3.13, { x: 2, y: 3 }, obj1];
```

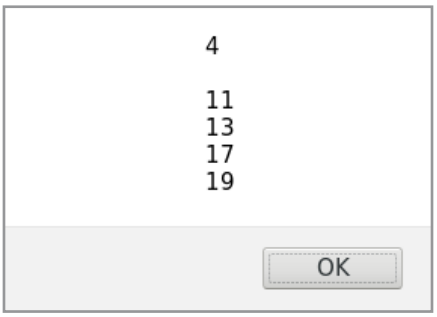
where the array contains a string, a number and two objects. You can also create an array with a constructor function:

```
var arr3 = new Array();
show(arr3);
```

and the result is an empty array. If you have an array, you can immediately add items, and the array will dynamically expand:

```
arr3[0] = 11;
arr3[1] = 13;
arr3[2] = 17;
arr3[3] = 19;
show(arr3);
```

and the array now have 4 elements:



An array may also have "holes", and thus elements that are not defined:

```
arr1[5] = "Erik";
show(arr1);
```

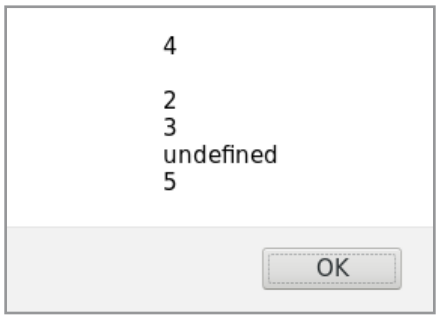
Constructor functions can have parameters:

```
var arr4 = new Array(4);
var arr5 = new Array(23, 29, 31, 37);
```

and here the first creates an array of length 4 with 4 undefined elements, while the other creates an array with 4 elements. Finally, you should note that you can change the value of the property *length* and thus delete items in an array. As mentioned, you reference the elements in an array using a numeric index, but indexing is actually more flexible than you would expect. Consider the following code:

```
var arr = new Array();
arr[0] = 2;
arr[1] = 3;
arr["3"] = 5;
arr["abc"] = 7;
show(arr);
```

First, note that the code does not fail and it will display the following alert:



If the index is not a number, the interpreter will try to convert it to an integer and the result of

```
arr["3"] = 5;
```

is therefore the element with index 3. On the other hand, it is not immediately clear what

```
arr["abc"] = 7;
```

means when "abc" can not be converted to a number. An array is an object, and therefore you can specifically define its own properties. This is exactly what happens here as *arr["abc"]* is interpreted as a property named *abc*, which then gets the value 7. There is reason to be aware of this interpretation as it is easy to accidentally add properties to an array. In this case the statement

```
alert(arr.abc);
```

will show 7. The elements in an array can be anything and hence especially other arrays. It allows to simulate multidimensional arrays. For example will the following code display an alert with the value 130:

```
var v1 = [2, 3, 6, 7];
var v2 = [11, 13, 17, 19];
var v3 = [23, 29];
var v = [v1, v2, v3];
var s = 0;
for (var i = 0; i < v.length; ++i) for (var j = 0; j < v[i].length; ++j) s += v[i][j];
alert(s);
```

In particular, note how to iterates the array *v* as a 2-dimensional array, but it is the programmer's responsibility that the elements in *v* are actually arrays. Otherwise, you will get an error. Also note that *v* illustrates a heterogeneous array and thus an array where the rows do not have the same length.

In fact, JavaScript is quite flexible as to how to define an array. Below is defined an array consisting of two arrays, and these arrays have elements of different types. The one even has an element that is not defined.

```
var konger = [["Gorm", , 958], ["Harald Blåtand", 958, 986]];
for (var i = 0; i < konger.length; ++i) show(konger[i]);
```

You should note that there are a number of standard functions for arrays that I mentions in a later section.

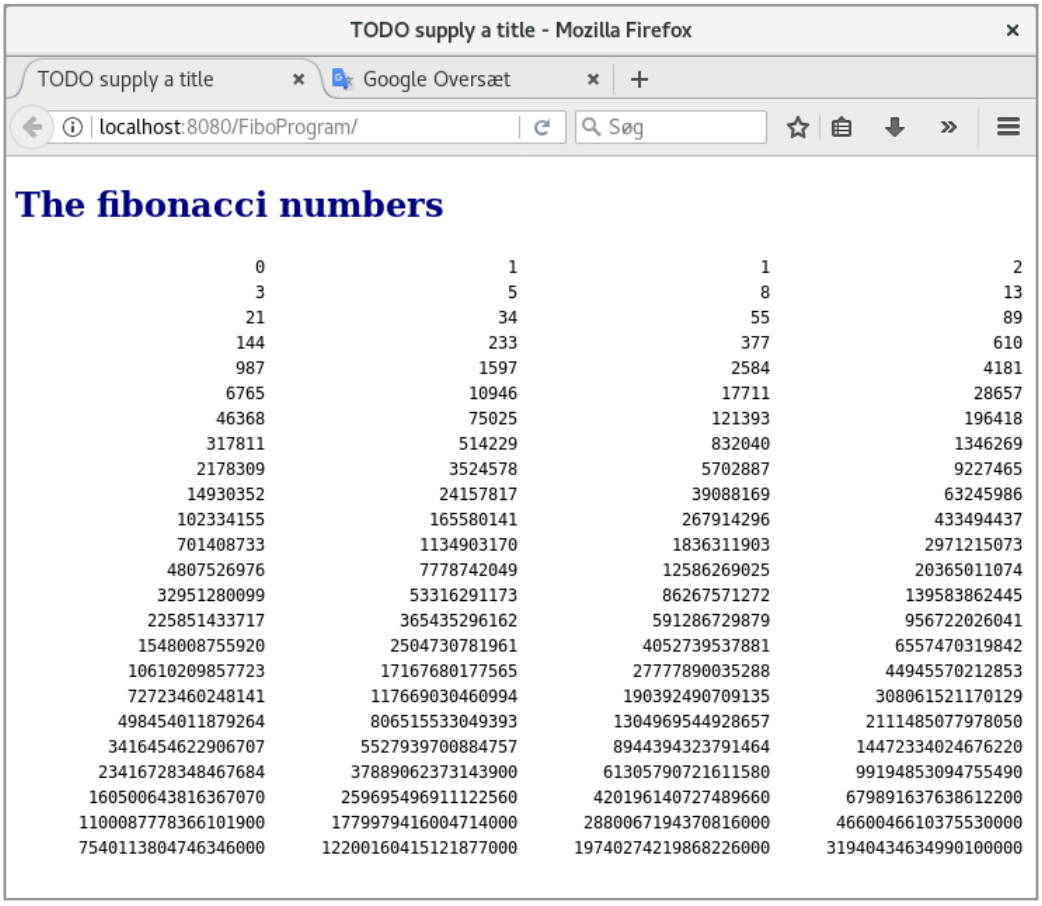
EXERCISE 2

The Fibonacci numbers are, as you know, the following sequence:

```
0, 1, 1, 2, 3, 5, 8, 13, 21, ….
```

Create a new project, that you can call *FiboProgram*. The project must have a start page *index.html* that shows the first 95 fibonacci numbers in a table (se the window below). The program should be written as follows:

1. The project must have a JavaScript file, that contains a module called *fibonacci*, which automatically creates a private array, that contains the fibonacci numbers. The module must have two public properties, where the first is called *length*, and is a simple property whose value is the length of the array, while the other property is a function *get(n)*, that returns the n'th fibonacci number.
2. The table and the header text should be styled by a simple style sheet.
3. The start page *index.html* must dynamic generate the table in JavaScript.



Functions

In JavaScript is a function as mentioned an object. That means more things, for example that a function can be return value from another function and that a function can be a parameter to another function. It also means that a function may have properties and methods like any other object. These relationships means that functions in many ways are different from functions in other languages like Java.

A function is usually created with the reserved word function:

```
function factorial(n)
{
}
```

and the function may have an arbitrary number of parameters and especially none. Since a function can refer to all variables in its own scope, the function can specifically refer to itself and you can therefore write recursive functions:

```
function factorial(n)
{
 if (n < 1) return 1;
 return n * factorial(n – 1);
}
```

It is allowed to declare a function multiple times in the same scope, which simply means that the function name is given a different value. It can also give unexpected results. For example, if you consider the following code:

```
function f()
{
 return 23;
}
alert(f());
function f()
{
 return 29;
}
```

it will alert 29. The reason is that JavaScript collects all definitions at the start of a scope and the above is thus equivalent to:

```
function f()
{
 return 23;
}
function f()
{
 return 29;
}
alert(f());
```

In JavaScript, you can also define a function as an expression:

```
var show = function (arr)
{
 var str = arr.length + "\n";
 for (var i = 0; i < arr.length; ++i) str += "\n" + arr[i];
 alert(str);
}
show([2, 3, 5, 7]);
```

The Varablen *show* refers to an anonymous function, and the use of anonymous functions is widely used in JavaScript. Consider the following example:

```
function validate(x, ok)
{
 if (ok(x)) return true;
 alert(x + " er en ulovlig værdi");
 return false;
}
```

The function has two parameters, the latter being interpreted as a function that will validate the first parameter. The function could, for example, be used as follows:

```
validate(1234, function (t) { return t >= 100 && t <= 999; });
```

where the function should validate if the first argument is a 3-digit number. The validation function is sent as an anonymous function.

When a function is performed, it occurs in a given context, and the function has access to variables, objects, and other functions in this context through its scope. Internally, this context is referenced by the *this* pointer, which is indirectly transferred to the function when it is called. In this context, it is necessary to distinguish between a function and a method where a method is a function defined as a property in an object. If you perform a usual function, its context will be the window object, while the context of a method is the parent object. It can be illustrated by the following code, where *m* is a method in the object *obj*, while *f()* is a global function with an inner function *g()*.

```
var obj =
{
 m : function() { alert(this === obj); }
}
function f()
{
 alert(this === window);
 function g()
 {
  alert(this === window);
 };
 g();
}
obj.m();
f();
```

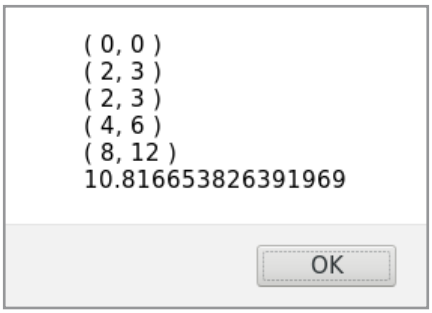
If you execute the code, it will display *true* in an *alert()* three times. First, the method *m()* is executed and its context is the object *obj*. Next, *f()* is executed and it starts performing the internal function *g()* whose context is the window object in the same way as the external function *f()*.

The most frequent use of *this* is in connection with methods and constructor functions. If you have a function and type *new* in front of the function, it means creating an empty object, and this will then refer to the context for this object. As an example, below is shown a constructor function that creates a point:

```
function createPoint(x, y)
{
 this.x = x;
 this.y = y;

 this.move = function (p) { this.x = p.x; this.y = p.y; };
 this.add = function (p) { this.x += p.x; this.y += p.y; };
 this.scale = function (t) { this.x *= t, this.y *= t; };
 this.dist = function (p) {
  return Math.sqrt((this.x - p.x) * (this.x - p.x) +
  (this.y - p.y) * (this.y - p.y)); };
 this.toString = function() { return "( " +
this.x + ", " + this.y + " )"; }
}
```

The following code uses the function to create two objects, and you should note in particular that the objects do not inherit each other and each have there own variables on which they work on:

```
var p1 = new createPoint(0, 0);
var str = p1.toString();
var p2 = new createPoint(2, 3);
str += "\n" + p2;
p1.move(p2);
str += "\n" + p1;
p1.add(p2);
str += "\n" + p1;
p1.scale(2);
str += "\n" + p1;
str += "\n" + p1.dist(p2);
alert(str);
```



Also note that the above recalls the class concept from an object-oriented programming language, but it has nothing to do with classes – JavaScript has only objects.

Usually, a function is performed by specifying parentheses after the function name, if any, the actual parameters are written between parentheses. The parentheses () are in this context called for the *function invokes operator*. However, there is an alternative to calling a function. You can use *apply()* or *call()*, where you can specify the context in which the function should work within. Consider the following code:

```
var scale = 10;
var obj1 =
{
 scale: 100
}
var obj2 =
{
 scale: 1000
}
function sum(x, y, z)
{
 return this.scale * (x + y + z);
}
alert(sum(2, 3, 5));
alert(sum.apply(obj1, [ 2, 3, 5]));
alert(sum.call(obj2, 2, 3, 5));
```

It will show with an alert 100, 1000 and 10000 respectively. At the top is defined a variable called scale that has the value 10. Next, two objects are defined which each have a property called scale and their values are 100 and 1000 respectively. These objects each define their context (scope). The function *sum()* has three parameters and returns the sum of these parameters after multiplying the sum with the value of a variable *scale* from the current context. The first alert shows the value of this function by calling it in the normal way, and since it is a global function, its context will be the window object, and that is why the global *scale* variable is used. The next *alert()* performs the function again, but this time using *apply()*. The first parameter is the context in which the function should work, and here it is *obj1*. That is that *scale* this time is the property *scale* from *obj1* that has the value 100. Note that the parameters are transmitted as an array. The last alert works in principle in the same way, but this time the function *call()* is used and *obj2* is used as a context. The difference between *apply()* and *call()* is that in the first one you specify parameters for the function as an array, while the other indicates the parameters as a list.

Programkontrol

As the last section regarding the basic syntax and semantics I want to show the language's control statements, and here is not much to explain. Secondly, I have already used most of the control statements several times, and basically, there are the same statements as in Java.

Regarding conditions, JavaScript has an *if* statement, which has the same syntax as in Java, and the semantics are also the same. The only difference is the condition, which should be an expression that evaluates a *boolean*, and here corresponding to the conversion rules described above, there is somewhat greater flexibility (and possibilities of errors) than the case is in Java.

JavaScript also has a *switch* statement that also works in the same way as in Java. The same goes for *while* and *do* loops. The most commonly used loop is as in Java the *for* loop, and the classic *for* loop works exactly the same as in Java:

```
for (var i = 0; i < arr.length; ++i) s += arr[i];
```

In Java, you also have a *foreach* loop, which is actually an iterator. There is a corresponding variant of a *for* loop in JavaScript, sometimes called *for-in*. It can be used to iterate all properties in an object. Below is an object with 4 properties and the following loop determines the sums of these four properties with an *for-in* loop:

```
var obj =
{
 a: 2,
 b: 3,
 c: 5,
 d: 7
}
var s = 0;
for (var t in obj) s += obj[t];
alert(s);
```

4.3 GLOBAL OBJECTS AND FUNCTIONS

JavaScript is born with several global objects and some global functions that are immediately available and in terms of methods offers a variety of services. The global objects are

1. Array
2. Boolean
3. Date
4. Math
5. Number
6. RegExp
7. String

I do not want to review these objects and their methods here, but the names tell you a bit about what you can do with them and you are encouraged to investigate the properties of the objects (there are many reefs on the internet describing these objects and methods). Finally, there are the following global functions, whose behavior you are also encouraged to investigate:

1. eval()
2. isFinite()
3. isNaN()
4. parseFloat()
5. parseInt()

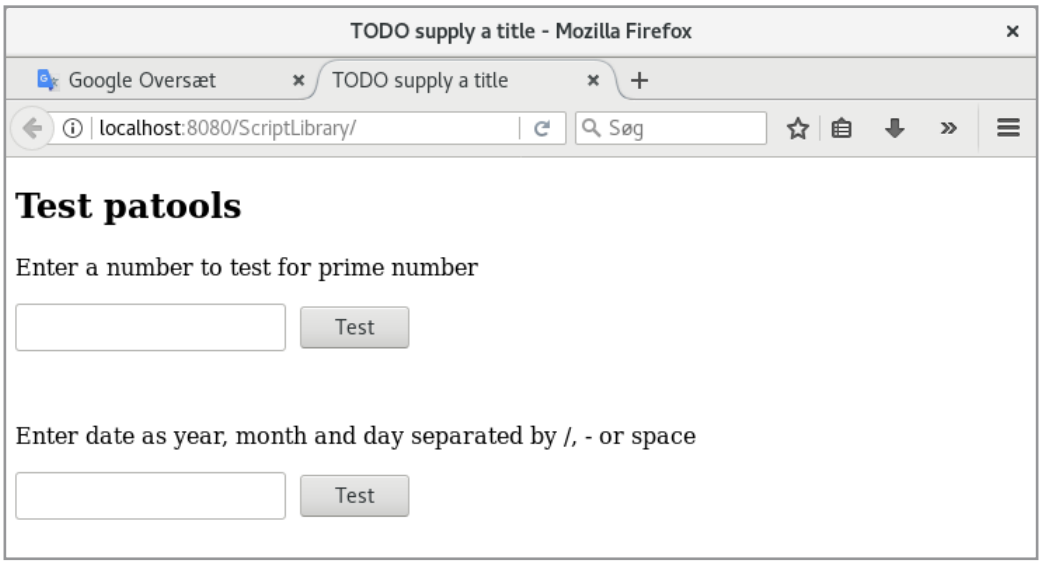
EXERCISE 3

Create a web application, that you can call *ScriptLibrary*. Create a folder *resources* and a subfolder *js* and add a JavaScript file with the name *pascript.js*. The file should implements a module called *patools* (or whatever name you want), and you should think of the module as a JavaScript library. The library should have four properties:

1. *isInt*, that is a function with one parameter, and the function should returns *true*, if the parameter is a legal integer.
2. *isPrime*, that is a function with one parameter, and the function should returns *true*, if the parameter is a prime number.
3. *isLeapyear*, that is a function with one parameter, and the function should returns *true*, if the parameter represents a leap year as a year between 1700 and 9999.
4. *isDate*, that is a function with one parameter, and the function should returns *true*, if the parameter represents a legal date between 1700 and 9999. The parameter must be a string of the form year, month and day (that is YYYY_MM_DD), where the separation character must be -, /, space or nothing.

When you have written the library methods, you should test the module from *index.html* (see below). The html code could be:

```
<body>
 <h2>Test patools</h2>
 <form>
  <p>Enter a number to test for prime number</p>
  <input type="text" id="txtnumber"/>  
  <input type="button" value="Test" onclick="validate1();" />
  <p><span id="result1"></span></p>
  <br/>
  <p>Enter date as year, month and day separated by /, - or space</p>
  <input type="text" id="txtdate"/>  
  <input type="button" value="Test" onclick="validate2();" />
  <p><span id="result2"></span></p>
 </form>
</body>
```



You must write event handlers for the two buttons. The first must test where the value in the first input field is a prime number and update til first *span* element with a message. The second must do the same, but for the other elements and test where the value is a legal date.

The event handlers should be written in the script block in the header. Note that it is also necessary to add a link element for your JavaScript library.

4.4 DOM

DOM stands for *Document Object Model* and is a W3C standard for how browsers must build an HTML document as a hierarchy of objects. DOM is not part of JavaScript, but the DOM objects allow JavaScript to manipulate the objects and thus refer to the contents of the current document from script code. In combination with JavaScript, DOM is the key to developing websites where something is happening on the client side. DOM is a standard, but it is up to the browser vendors to implement this standard and the standard is no better than to the extent that the browser vendors live up to it. This means that there may be (and are) differences between different browsers and different versions of the same browser, but the most basic objects are, however, implemented roughly standard.

The root of the DOM tree is called *window* and is an object consisting of several properties and collections. The main property is *document* that represents the document that the browser shows. Consider, for example, the following document (*DOMTree* project):

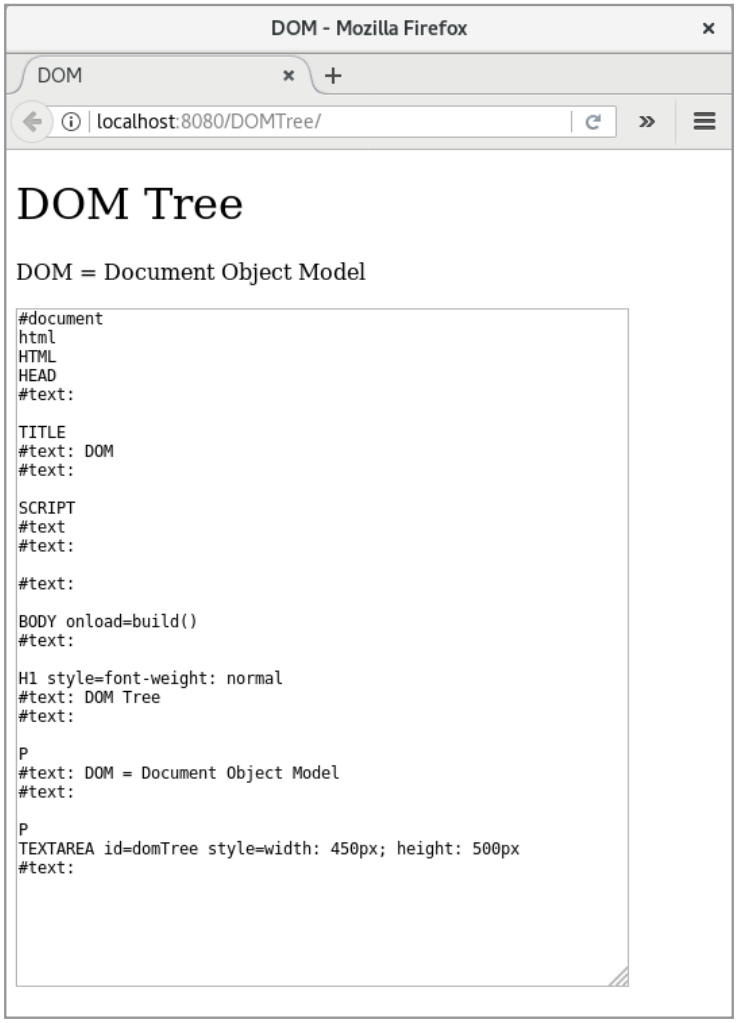
```
<!DOCTYPE html>
<html>
 <head>
  <title>DOMTree</title>
  <script>
   tree = "";
   function parse(node)
   {
    tree += node.nodeName;
    if (node.nodeName === "#text" && node.parentNode.nodeName !== "SCRIPT")
     tree += ": " + node.textContent;
    if (node.attributes != null)
     for (var i = 0; i < node.attributes.length; ++i)
      tree += " " + node.attributes[i].name + "=" + node.attributes[i].value;
    tree += "\n";
    if (node.childNodes != null)
     for (var j = 0; j < node.childNodes.length; ++j)
      parse(node.childNodes[j]);
   }
   function build()
   {
    parse(document);
    document.getElementById('domTree').value = tree;
   } </script>
   </head>
 <body onload="build()">
  <h1 style="font-weight: normal">DOM Tree</h1>
  <p>DOM = Document Object Model</p>
  <p><textarea id="domTree" style="width: 450px; height: 500px"></textarea></p>
 </body>
</html>
```

In the DOM tree, each element in the document is represented by a node, and it also applies to elements that are not visible, such as a *script* element. The typical use of JavaScript is to modify the DOM tree, including specifically changing the individual nodes. In this case, the script element begins by defining a global variable. The function *parse()* has a node as a parameter, and it starts by adding this node's name to the global variable *tree*. If attributes are attached to the node, they are also added to the variable as key/value pair. Finally, if there are child nodes, the function *parse()* is called recursively for each child node.

You must note the syntax for the function *parse()*. Of course, it is not so easy to guess what the individual properties are called, but when you see the result, it's easy to follow what happens.

There is another function called *build()* which calls the above recursive function with the *document* object as parameter. When the function *parse()* is performed, the function *build()* will with a reference to the element with id *domTree* (which is a text area) insert the value of the global variable *tree*. The *build()* function is performed as an event handler for *onload* that occurs after the document is loaded and displayed in the browser, and after that the DOM tree is built.

If you opens the document in a browser, you get the window belove, and here you can see how each element in the document is represented by a node in the DOM tree.



Refer elements in DOM

With regard to the use of DOM, the first step is to learn how to refer to the individual elements of the DOM tree and then do something with these elements. This is basically done by referring to the elements *id* attributes or the elements *class* attribute, which is illustrated in the introduction to this chapter with the example *SimpleDocument*. The example shows a little about what options are available to find a particular element or array of elements in the DOM tree. The basic methods are:

- *getElementById()*
- *getElementsByTagName()*
- *getElementsByClassName()*
- *querySelector()*
- *querySelectorAll()*

Here, the last two are the most advanced, as they use CSS selectors as explained in chapter 3 of this book, and the difference is that the first returns the first element that matches the selector, while the other returns all the elements that match. In addition, a node has the following properties:

- *firstChild*
- *lastChild*
- *nextSibling*
- *previousSibling*

that can be used to traverse the DOM tree.

Modify the DOM tree

It is also possible to modify the DOM tree using JavaScript, and although already done in the above examples there are many more options. You can modify the elements properties and change their content, and you can even move an element in the tree. You can also delete an element, and you can insert new elements. Again, I will illustrate some of the possibilities with an example:

```
<!DOCTYPE html>
<html>
 <head>
  <title>Modify document elements</title>
  <style>
   .blueClass {
    background-color: lightsteelblue;
   }
   .plainClass {
    font-weight: normal;
   }
  </style>
 </head>
 <body>
  <h1>Header</h1>
  <h3 id="txt">Modifies DOM elements using Javascript</h3>
  <p id="adr"><a>Torus data</a></p>
  <p id="par">More text about <span style="font-weight:bold">
   DOM</span> objects</p>
  <script>
   alert(par.innerText);
   alert(par.innerHTML);
   var link = document.querySelector("#adr a");
   link.href = "http://bookboon.com";
```

```
    link.style.backgroundColor = "#ff0000";
    link.style.color = "#ffffff";
    link.target = "_blank";
    var txt = document.getElementById("txt");
    txt.classList.add("blueClass");
    txt.classList.add("plainClass");
    document.getElementsByTagName("h1")[0].innerText = "Hello World";
    par.innerHTML = "<ul><li>Svend</li><li>Knud</li><li>Valdemar</li></ul>";
    var head = document.head;
    var elem = document.createElement("script");
    elem.innerText =
"function factorial(n) { if (n < 2) return 1; return n * factorial(n - 1); }";
    head.appendChild(elem);
    var body = document.body;
    var text = document.createElement("p");
    text.id = "fact";
    body.appendChild(text);
    document.getElementById(
     "fact").innerHTML = "10! = <b>" + factorial(10) + "</b";
  </script>
 </body>
</html>
```

The page defines in the header two classes, called respectively *blueClass* and *plainClass*. The document has 4 elements as a *h1* element, a *h3* element and two paragraphs. The first paragraph contains a *link* and the other a *span* element. Then follow a script block, starting with an alert:

```
alert(par.innerText);
```

which shows the text in the last paragraph. The paragraph is referred to by its ID, and you should note that you can do it directly. You should also note that the HTML element *span* is not displayed as part of *innerText*. The result is that after the page is opened, the browser displays the following:

and next the following alert:

The next alert:

```
alert(par.innerHTML);
```

shows the same element, but this time it's *innerHTML*. Unlike *innerText, innerHTML* shows the entire text including HTML elements:

The first paragraph has an *id* named *adr*. This paragraph has a *link* (an element) as a child element, and the variable *link* is set to refer to this element. Here you should especially note the method *querySelector()* and the syntax to refer to a child element for the element with *id adr*:

```
var link = document.querySelector("#adr a");
```

The following statements are used to modify properties of this element.

The variable *txt* refers to the *h3* element. An HTML element has (in HTML5) a list for *class* objects. The reference *txt* is used to add *class* objects to the element *h3*. The result is that *h3* gets a blue background and the text is displayed as normal. The next statement:

```
document.getElementsByTagName("h1")[0].innerText = "Hello World";
```

changes the text in the element *h1* by changing the element's *innerText*. You must primarily note the reference (the *h1* element has no *id*), and *getElementsByTagName()* is an array with all *h1* elements, and you get the first (and in this case only) by referring the item with index 0.

par are *id* for a paragraph containing some HTML and the following statement replaces this HTML with a list of 3 elements:

```
par.innerHTML = "<ul><li>Svend</li><li>Knud</li><li>Valdemar</li></ul>";
```

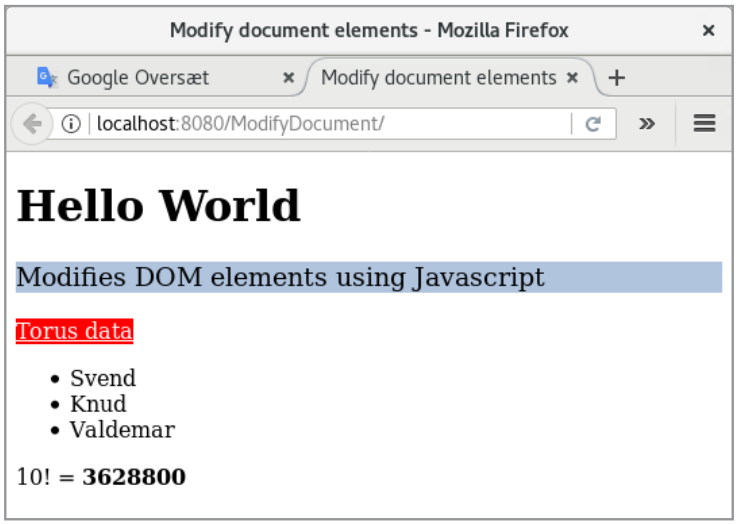
You can therefore specifically change a DOM object to something completely different. Next, two variables *head* and *elem* are defined, which refer respectively to the document's header as well as a new element that is a *script* element. You can in that way create new HTML elements:

```
var elem = document.createElement("script");
```

Next, this element's *innerText* is defined as a JavaScript function. The new script element is added to the *head* variable as a child node. This means that in the header part of the document there dynamically is added a script element with a Javascript function. Next, two more variables are defined. *body* is a reference to the body of the document, and *txt* is a reference to a paragraph element. The new (and empty) paragraph is assigned an *id* and is added to the body of the page, meaning it is inserted as an empty node in the DOM tree. Finally, the content of the new paragraph is changed to some HTML. Note that it includes the dynamically added Javascript function being executed and the value is inserted in the new paragraph:

```
document.getElementById("fact").innerHTML = "10! = <b>" + factorial(10) + "</b";
```

After clicking OK for the last alert, the page content will be as follows:



Events

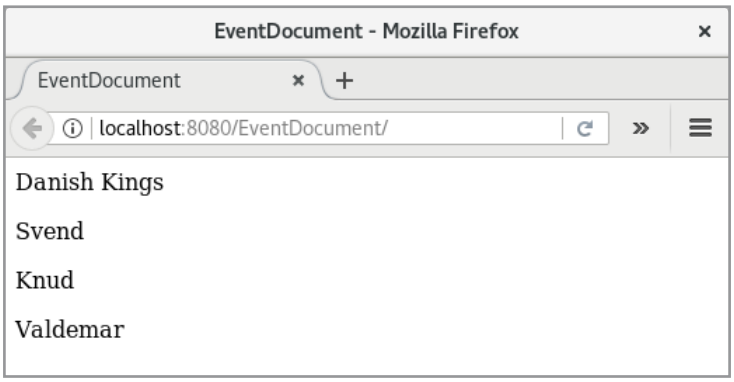
DOM defines exactly as events how to interact with the individual elements. Most events relates to the mouse, and for a particular event you can associate an event handler with an element in the document. When a user interacts with an element, the browser checks whether an event handler has been registered for that element, and if so, the handler is executed. The elements are in a document part of a hierarchy and an event will bubble up through the tree of all parent elements, and for all parent elements where an event handler has been registered for that event, this will be excuted. When the event reaches the top – and that is, the body element – an event will follow the same path back until it returns to the element that originally raised the event. The two phases are called the *bubbling phase* and the *capturing phase* respectively, and you can specify the phase in which you wish to process the event.

When an event handler is performed, it is done in the same way as any other JavaScript function within a context, and DOM defines that it is the context of the element to which that particular handler is related. You can therefore refer to the element that the event concerns with the *this* reference.

DOM defines the following events:

- events for the mouse: *click*, *mousedown*, *mouseup*, *mousemove*, *mouseover*, *mouseout* and more
- events for the keyboard: *keypress*, *keydown*, *keyup*
- events for objects: *load*, *error*, *scroll*
- events for forms: *select*, *change*, *submit*, *reset*, *focus*
- events for user interaction: *focusin*, *focusout*

Consider, for example, a page with the following content:



Here the upper text is a *div* element, while the three bottom are paragraphs. The code is as follows:

```
<!DOCTYPE html>
<html>
 <head>
  <title>EventDocument</title>
  <script>
   var text;

   function kingCapture()
   {
    text += "\n" + this.innerText + " capture";
   }

   function kingBubble()
   {
    text += "\n" + this.innerText + " bubble";
   }

   function kingsCapture()
   {
    text = "Kings capture";
   }

   function kingsBubble()
   {
    text += "\nKings bubble";
    alert(text);
   }
  </script>
 </head>
```

```
<body>
 <div>Danish Kings</div>
 <div>
  <p id="paragraf1">Svend</p>
  <p id="paragraf2">Knud</p>
  <p id="paragraf3">Valdemar</p>
 </div>
 <script>
  var p1 = document.getElementById("paragraf1");
  var p2 = document.getElementById("paragraf2");
  var p3 = document.getElementById("paragraf3");
  p1.addEventListener("click", kingBubble);
  p2.addEventListener("click", kingBubble);
  p3.addEventListener("click", kingBubble);
  p1.addEventListener("click", kingCapture, true);
  p2.addEventListener("click", kingCapture, true);
  p3.addEventListener("click", kingCapture, true);
  document.getElementsByTagName("div")[1].
   addEventListener("click", kingsCapture, true);
  document.getElementsByTagName("div")[1].
   addEventListener("click", kingsBubble);
 </script>
</body>
/html>
```

In the header are defined two simple event handlers. You should note the use of the *this* reference, which will refer to the element that has raised the event. The HTML code requires no explanations, but it is the subsequent script block that assign event handlers to the HTML elements. First, there are references to the three paragraphs, and then two event handlers are assigned for a click event for each paragraph. The one is a handler for the bubbling phase (*false*), and the other is a handler for capturing the phase (*true*). You should note how to associate an event handler with the method *addEventListener()*. As a result, clicking on one of the three paragraphs will cause an alert to show which events are fired. The last statement in the script block assigns an event handler to the first *div* element. Here you should notice how to find the item and how to set the event handler as an anonymous function.

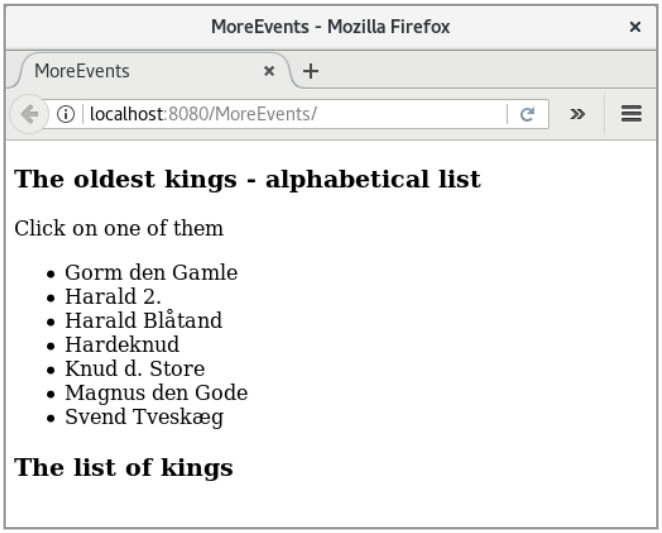
There is also a method called *removeEventListerner()* and that has the same parameters as *addEventListener()*. This function is used to remove an event handler from an element.

When an event handler is performed, an event object is transferred to the handler. Consider the following code as example:

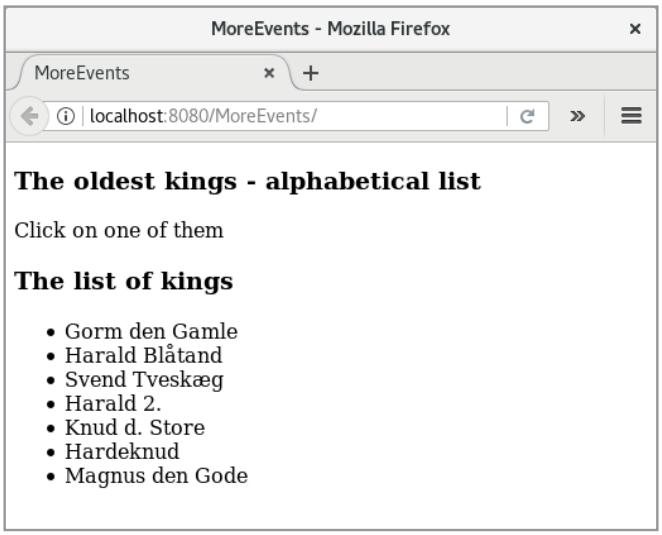
```
<!DOCTYPE html>
<html>
 <head>
  <title>MoreEvents</title>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
 </head>
 <body>
  <h3>The oldest kings – alphabetical list</h3>
  <p>Click on one of them</p>
  <ul id="list1">
   <li>Gorm den Gamle</li>
   <li>Harald 2.</li>
   <li>Harald Blåtand</li>
   <li>Hardeknud</li>
   <li>Knud d. Store</li>
   <li>Magnus den Gode</li>
   <li>Svend Tveskæg</li>
  </ul>
  <h3>The list of kings</h3>
  <ul class="list2"></ul>
  <script>
   var list = document.querySelector(".list2");
   var kings = document.querySelectorAll("li");
   for (var i = 0; i < kings.length; i++)
   {
```

```
      kings[i].addEventListener("click",
        function (e) { list.appendChild(e.target); }, false);
     }
    </script>
   </body>
  </html>
```

If the code appears in the browser, the result is:



The list with *id list1* has 7 names arranged alphabetically. Next, an empty list is identified by a *class* attribute. The following script block sets a reference to the empty list using the *class* attribute and then a reference to an array of all *li* elements. The last loop associates an anonymous event handler to these *li* elements. This handler uses the event object – here called *e* – to move the element that is clicked to the empty list. Below is the result after clicking on all names (in the correct order) in the top list:



The most important thing about the example is how to transfer a parameter to the event handlers. It is an object called *event* and, as illustrated below, it is an advanced object with many properties and methods.

Properties of the *event* object

- *event.clientX* and *event.clientY* indicates in the case of a mouse event the coordinates for the mouse relative to the browser window.
- *event.offsetX* and *event.offsetY* indicates in the case of a mouse event the coordinates for the mouse relative to the element that has raised the event.
- *event.keyCode* indicates in case of a key event code for key pressed.
- *event.target* is a pointer to the node in the DOM tree, which has raised the event.
- *event.currentTarget* is a pointer to the node in the DOM tree, which is bubbled or captured
- *event.eventPhase* indicates the event phase as 1 for capture, 2 for target, 3 for bubbling.
- *event.type* indicates event type such as click, key press, etc.
- *event.relatedTarget* indicates for some events (for example *mouseout*) the element that originally raised the event.

- *event.stopPropagation()* is a method that can be called to stop an event propagation up or down in the DOM tree, but other registered event handlers are still performing.
- *event.stopImmediatePropagation()* is a method that as *event.stopPropagation()*, stops an event propagation up or down in the DOM tree, but it does not performs other registered event handlers.
- *event.preventDefault()* is a method that stops a default event handling if possible.

Looking at the above code nothing happens when you point to the individual elements in the top list. This can be solved by assigning an event handler for *mouseover* and *mouseout*. I have added the following event handlers:

```
function blue()
{
 this.setAttribute("style", "color:blue;");
}
function black()
{
 this.setAttribute("style", "color:black;");
}
function moveKing(e)
{
 e.target.removeEventListener("mouseover", blue, false);
 e.target.removeEventListener("mouseout", black, false);
 e.target.setAttribute("style", "color:black;");
 list.appendChild(e.target);
}
```

The first two handlers must be used for respectively *mouseover* and *mouseout* for the elements in the top list. You should note that the latter has a parameter that will be a reference to an event object. The event handler must be used to move an element. When an element is moved, it will no longer be highlighted when the mouse points to it. Therefore, the two event handlers for *mouseover* and *mouseout* are removed, and you should note that *e.target* refers to the element that has raised the event. In addition, the color must be set to black, and then the element is moved to the bottom list in the same way as above.

The HTML code is the same as above, including the script that associates event handlers:

```
<script>
 var list = document.querySelector(".list2");
 document.getElementById("list1").addEventListener("click", moveKing, false);
 var kings = document.querySelectorAll("li");
 for (var i = 0; i < kings.length; i++)
 {
```

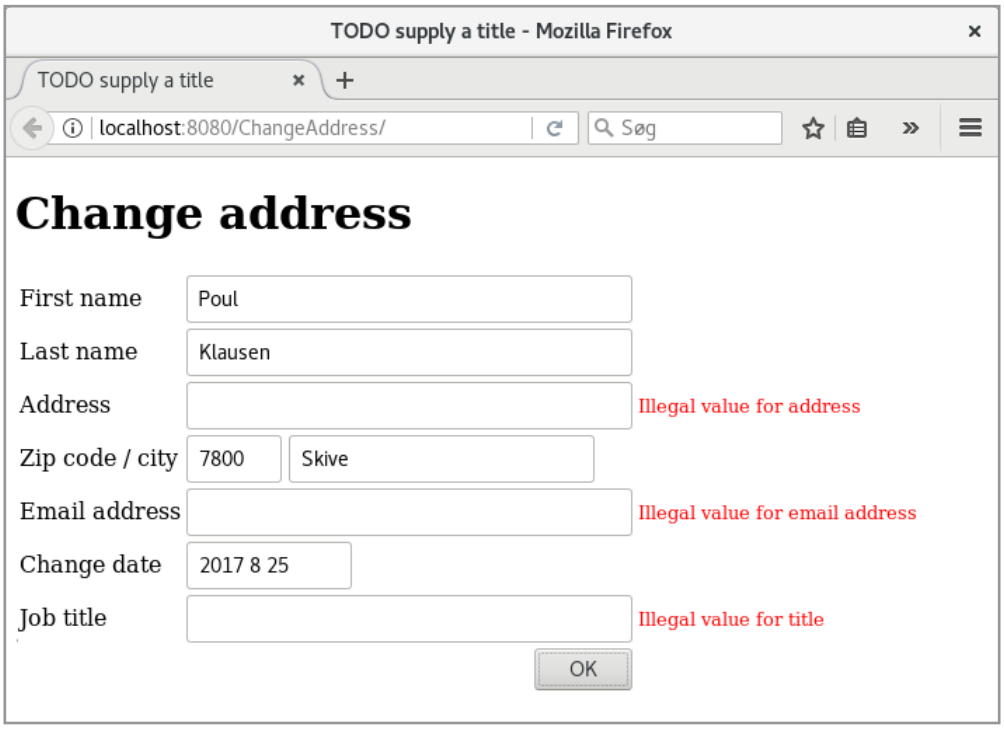
```
    kings[i].addEventListener("mouseover", blue, false);
    kings[i].addEventListener("mouseout", black, false);
  }
</script>
```

Here you should first note that the event handler *moveKing()* for click events is assigned to the list itself and not to the individual list elements. Here are used that events bubbles up and clicking on a list element, the event then bubbles up to the element's parent, which is the *ul* element, that has attached an event handler, and in the handler, *e.target* will refer to the list element that have fired the event.

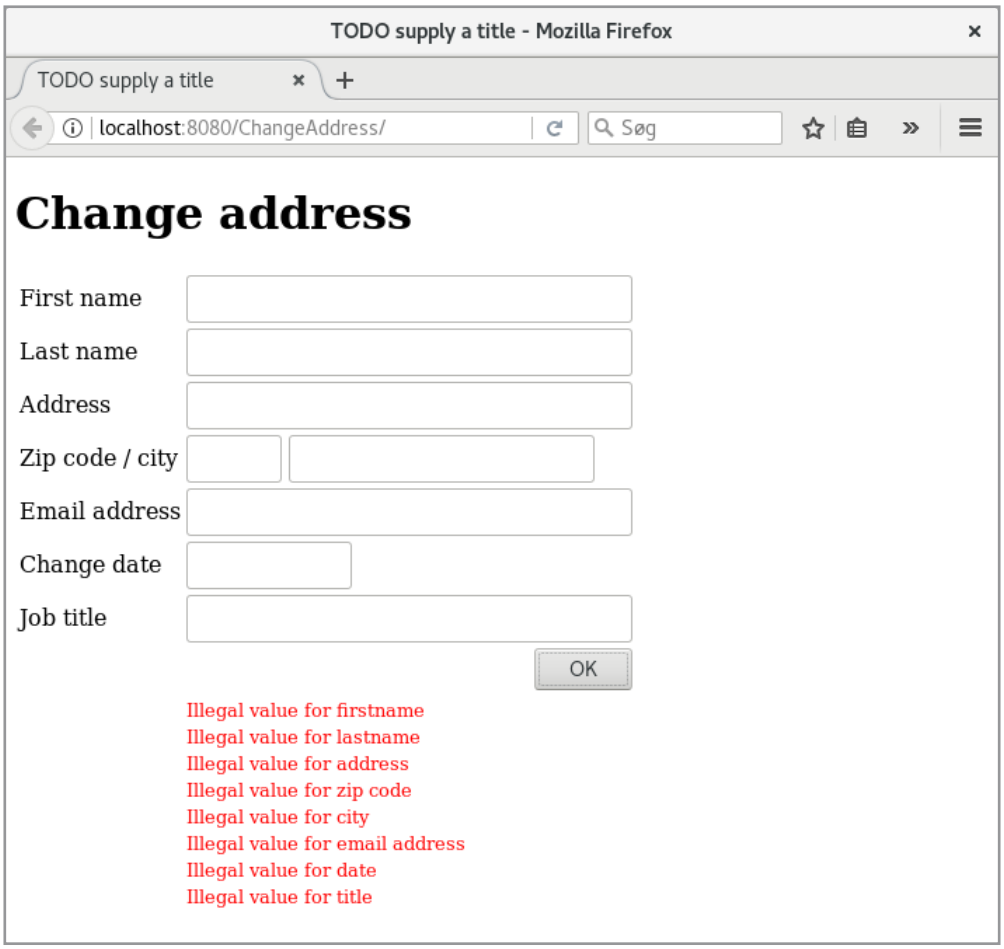
PROBLEM 3

Create a new project that you can call *ChangeAddress*. The project should only have one page *index.html*, and it should basic be a form with component arranged in a table:

When the user enters data, they should be validated, and the validation must take place every time a field losts focus. Al fields must be validated and for first name, last name, address, city and title the only requirement is that the fields must not be empty. Zip code must be 4 digits while email address and date must be a legal email address and a legal date, respectively. All validation must happen i JavaScript on the client side, and below is a window where has entered values for first name, last name, zip code, city name and date:


TODO supply a title - Mozilla Firefox
TODO supply a title
localhost:8080/ChangeAddress/
Søg
Change address
First name
Poul
Last name
Klausen
Address
Illegal value for address
Zip code / city
7800
Skive
Email address
Illegal value for email address
Change date
2017 8 25
Job title
Illegal value for title
OK

And below a window where the OK button is clicked without data entered:



To validate an email address you should use a regular expression. It is part of the task to find out how and how the syntax is. To validate a date, you should use the JavaScript library from exersice 3, and it must be possible to enter a date in the same format as described in this exercise. The button should be a submit button, and you can with an *onsubmit* event for the *form* element test if a click on the button must result in a submit (it should not be if not all fields are filled out correctly).

If all fields are correctly filled and clicking on the button, nothing else than an submit must be done with the result that all fields are blank.

5 JAVASERVER FACES AND AJAX

In this book, I have alone looked at the client side, including how to define a layout of a page using styles, and how to use JavaScript to modify the individual elements of the DOM tree. It takes place in the browser and without the server's involvement. In this chapter, I will again look at the server side where the code is executed on the basis of a request from the browser and where the server then answers back with a response, which means that the browser should render the document again. With the help of JavaScript, it is possible to execute that form of a request/response in such a way that the user does not observes the page being updated and that the application essentially behave in the same way as a standard desktop application. This technique is called *Ajax*.

Ajax is a family of technologies that enable JavaScript to perform tasks asynchronously when the browser sends requests to the server in the background, which then sends responces back to the client that can be used to update the DOM tree. The technology is widely used in all modern web applications.

You can basically associate ajax facilities with a JSF input component with the element *f:ajax*, which means an interaction between client and server, based on a particular event, an interaction performed asynchronously and parallel with the user using the application. A number of attributes are attached to a *f:ajax* element, where all if not specified has a default value:

- *delay*, which is a value specified in milliseconds and indicates the maximum size of a delay between request and response (*none* disables this feature).
- *disabled*, which is a *boolean* indicating that the function is disabled (*false* is default).
- *event*, which specifies the event for the component that triggers the ajax request.
- *execute*, that specifies a list of components to be performed on the server.
- *immediate*, which is a *boolean* that indicates that the input value should be processed immediately.
- *listener*, which specifies the name of a listener method to be performed.
- *onevent*, that specifies the name of a JavaScript function to be performed.
- *onerror*, that specifies the name of a JavaScript function for error handling.
- *render*, which is a list of components to be rendered after the ajax function.

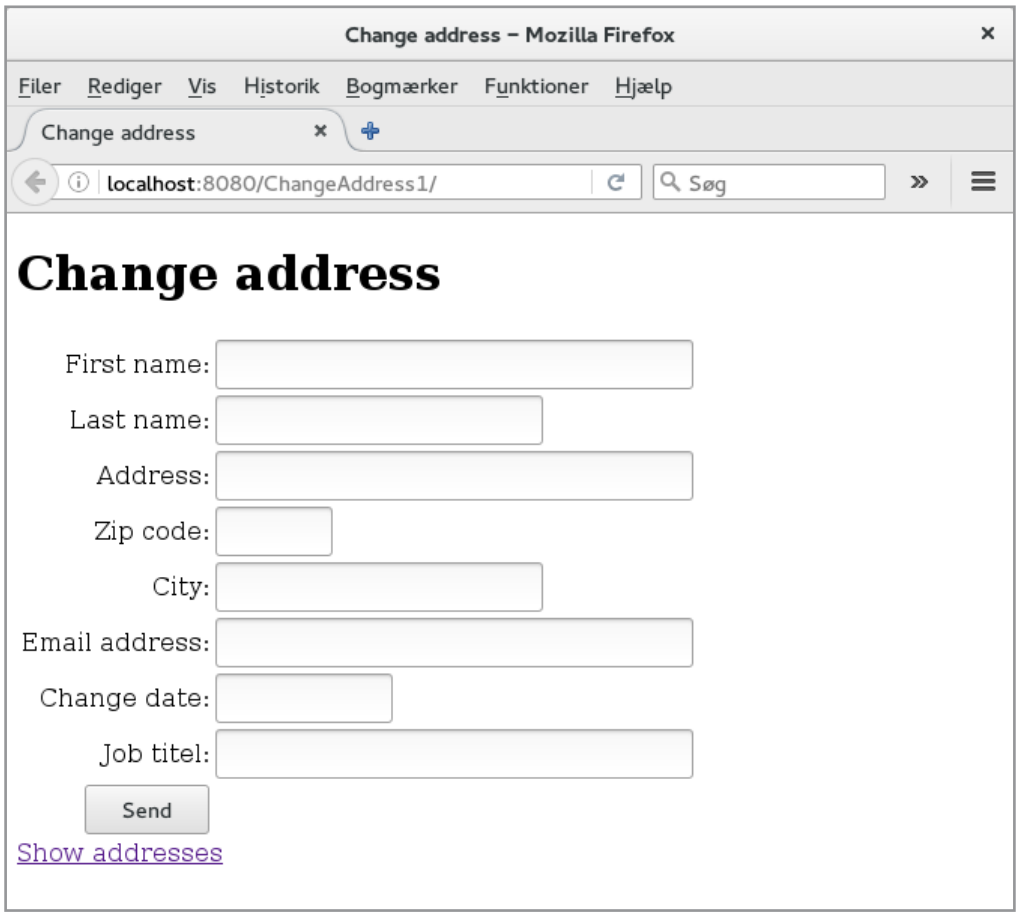
For the *execute* and *render* attributes, the following terms are available for specifying components:

- *@all*, that means all components
- *@form*, that means all components in the form, that contains the component for the ajax function
- *@none*, that means none components and is default for *render*
- *@this*, that means the ajax functions parent component
- The components ID separated by spaces
- A JSF expression

To illustrate how ajax works, I want to show a number of examples that extend an existing application with ajax functions, and it will be based on the application *ChangeAddress3* from the book Java 11.

5.1 VALDATION OF FIELDS

I have started with a copy of the project *ChangeAddress3* from Java 11, and I have called the copy for *ChangeAddress1*. If you open the application in the browser, you get the following window:



where a user can enter an address and other information. Clicking the *Send* button sends this information to the server where they are stored in a list. If you want to see the content of the list, you can click on the bottom link. When data is sent to the server, they are first validated and if there are errors, data is not saved, but the server responds with an error messages.

It works fine, but it would be more appropriate to validate each field immediately after the text is entered. As it is the server that validates the fields, a request must be made to the server for each field, and ajax can help, such that it happens completely transparent to the user. In fact, it is extremely simple and consists of changing the *index.xhtml* document as shown below:

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC … >
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html"
      xmlns:f="http://xmlns.jcp.org/jsf/core">
 <h:head>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>
  <title>Change address</title>
  <h:outputStylesheet library="css" name="styles.css"/>
 </h:head>
 <h:body>
  <h1>Change address</h1>
  <h:form>
   <h:panelGrid columns="3" columnClasses="rightalign,leftalign,leftalign">
    <h:outputLabel value="First name:" for="firstname"/>
    <h:inputText id="firstname" label="First name" style="width: 300px"
     value="#{indexController.firstname}" >
     <f:validateRequired/>
     <f:ajax event="blur" render="firstnameError"/>
    </h:inputText>
    <h:message for="firstname" class="error-message" id="firstnameError" />
    <h:outputLabel value="Last name:" for="lastname"/>
    <h:inputText id="lastname" label="Lastname" style="width: 200px"
     value="#{indexController.lastname}" >
     <f:validateRequired/>
     <f:ajax event="blur" render="lastnameError"/>
    </h:inputText>
    <h:message for="lastname" class="error-message" id="lastnameError"/>
    <h:outputLabel value="Address:" for="address"/>
    <h:inputText id="address" label="Address" style="width: 300px"
     value="#{indexController.address}" >
     <f:validateRequired/>
     <f:ajax event="blur" render="addressError"/>
    </h:inputText>
    <h:message for="address" class="error-message" id="addressError"/>
    <h:outputLabel value="Zip code:" for="code" />
    <h:inputText id="code" label="Zipcode" style="width: 60px"
     value="#{indexController.code}">
     <f:validateLength minimum="4" maximum="4"/>
     <f:ajax event="blur" render="codeError"/>
    </h:inputText>
    <h:message for="code" class="error-message" id="codeError"/>
    <h:outputLabel value="City:" for="city"/>
    <h:inputText id="city" label="City" style="width: 200px"
     value="#{indexController.city}" >
     <f:validateRequired/>
     <f:ajax event="blur" render="cityError"/>
    </h:inputText>
    <h:message for="city" class="error-message" id="cityError"/>
```

```
    <h:outputLabel value="Email address:" for="email"/>
    <h:inputText id="email" label="Email address" style="width: 300px"
     value="#{indexController.email}">
     <f:validator validatorId="emailValidator"/>
     <f:ajax event="blur" render="emailError"/>
    </h:inputText>
    <h:message for="email" class="error-message" id="emailError"/>
    <h:outputLabel value="Change date:" for="date"/>
    <h:inputText id="date" label="Change date" required="true"
     style="width: 100px" value="#{indexController.date}">
     <f:validator validatorId="dateValidator"/>
     <f:ajax event="blur" render="dateError"/>
    </h:inputText>
    <h:message for="date" class="error-message" id="dateError"/>
    <h:outputLabel value="Job titel: " for="title"/>
    <h:inputText id="title" required="false" style="width: 300px"
     value="#{indexController.title}" />
    <h:panelGroup/>
    <h:commandButton value="Send" action="#{indexController.add()}" />
   </h:panelGrid>
   <a href="list.xhtml">Show addresses</a>
  </h:form>
 </h:body>
</html>
```

For example, if you consider the first field for entering a first name, you must specify which validator to perform and then the element:

```
<f:ajax event="blur" render="firstnameError"/>
```

Here you indicate that the *ajax* function must be executed after the input field has lost focus, as well as which element may be rendered in case of an error that is the element of the error message. When that event occurs, a request is made to the server, but asynchronously and only for the current input component. As a result, the fields are validated as the form is completed and the user will experience the form as filling out a form in a conventional desktop application.

You should note that as an alternative, all validation functions could be written as JavaScript functions in this case, but in other cases validation functions require a request to the server.

EXERCISE 4

In this exercise, you should make a change to the above example. Start by creating a copy of the project *ChangeAddress1*. You must now change the program so the field for entering the city name is *readonly*. The database *padata* has a table *zipcode*, which is a list of Danish zip codes. You must change the validation of the field to zip code, so a zip code is only legal if there is an existing zip code in the table *zipcode*. If this is the case, the city name must be updated with the corresponding city name, and otherwise the city name must be blank. The validation of the zip code field must still be done using ajax.

5.2 SUBMIT FIELDS WITHOUT RELOAD

As next, I want to show how you with an submit can send all fields to the server, but without reloading the page. The benefits are not so big, but it gives a quieter window, as it all happens asynchronously, and the user thus, to a lesser extent, notes that data being sent to the server, and since the entire document do not has be rendered again. I will use the same example as above, and I have started with a copy that I have called *ChangeAddress2*. In *index.xhtml*, I've changed in two places. The *Submit* button now has an ajax function:

```
<h:commandButton value="Send" action="#{indexController.add()}" >
 <f:ajax event="action" execute="@form" render="@all"/>
</h:commandButton>
```

and you should note that the function is triggered by an action event that the function should relate to all form fields and that all fields must be rendered. In addition, the last link must be changed to a *h:commandLink*:

```
<h:commandLink value="Show addresses" action="list.
xhtml" immediate="true" />
```

and here you should especially note the last attribute that is required because the command should not validate the form fields.

Finally, the link on the page list.xhtml is changed to a *commandLink* (what it should have been all the time).

In this case, all form fields are sent to the server by a submit, and it is all form fields that are updated after the ajax function is completed. It is not always you are interested in that, and in fact there could be only a few fields. In this case, I assume the date field should always be today, and in *index.xhtml* I have defined the component *readonly*:

```
<h:outputLabel value="Current date:" for="date"/>
<h:inputText id="date" label="Change date" readonly="true" tabindex="-1"
 style="width: 100px" value="#{indexController.date}" />
```

You should note that there is no longer any ajax function associated with the component, and I have also changed the text in the corresponding label.

In order to initialize the field *date* I have in the class *Person* changed the constructor so it initializes the *date* field to the current date:

```
public Person()
{
 Calendar cal = Calendar.getInstance();
 date = String.format("%02d-%02d-%04d", cal.get(Calendar.DATE),
  cal.get(Calendar.MONTH) + 1, cal.get(Calendar.YEAR));
}
```

Finally, the ajax function of the submit button in *index.xhtml* is changed:

```
<h:commandButton value="Send" action="#{indexController.add()}" >
 <f:ajax event="action" execute="firstname lastname address code city email title"
  render="firstname lastname address code city email title"/>
</h:commandButton>
```

That is, I have now explicitly specified which form fields are to be sent to the server, and which form fields must be rendered after the function has been completed.

In this case, the advantage is of course limited, but in other contexts it may matter, and remember that the whole idea of ajax is only to update that part of a page that is necessary.

I will make another change of the application, and I have started creating a copy that I have called *ChangeAddress3*. In *index.xhtml* the fields are validated with each their ajax function. You can actually achieve the same by encapsulating the fields to be validated in a f:ajax element:

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC … >
<html … >
 <h:head>
  …
 </h:head>
 <h:body>
  <h1>Change address</h1>
  <h:form id="mainForm">
   <h:panelGrid columns="3" columnClasses="rightalign,leftalign,leftalign">
    <f:ajax event="blur" render="firstnameError lastnameError addressError
     codeError cityError emailError">
     <h:outputLabel value="First name:" for="firstname"/>
     <h:inputText id="firstname" label="First name" style="width: 300px"
```

```
    value="#{indexController.firstname}" >
    <f:validateRequired/>
   </h:inputText>
   <h:message for="firstname" class="error-message" id="firstnameError" />
   <h:outputLabel value="Last name:" for="lastname"/>
   <h:inputText id="lastname" label="Lastname" style="width: 200px"
    value="#{indexController.lastname}" >
    <f:validateRequired/>
   </h:inputText>
   <h:message for="lastname" class="error-message" id="lastnameError"/>
   <h:outputLabel value="Address:" for="address"/>
   <h:inputText id="address" label="Address" style="width: 300px"
    value="#{indexController.address}" >
    <f:validateRequired/>
   </h:inputText>
   <h:message for="address" class="error-message" id="addressError"/>
   <h:outputLabel value="Zip code:" for="code" />
   <h:inputText id="code" label="Zipcode" style="width: 60px"
    value="#{indexController.code}">
    <f:validateLength minimum="4" maximum="4"/>
   </h:inputText>
   <h:message for="code" class="error-message" id="codeError"/>
   <h:outputLabel value="City:" for="city"/>
   <h:inputText id="city" label="City" style="width: 200px"
    value="#{indexController.city}" >
    <f:validateRequired/>
   </h:inputText>
   <h:message for="city" class="error-message" id="cityError"/>
   <h:inputText id="email" label="Email address" style="width: 300px"
    value="#{indexController.email}">
    <f:validator validatorId="emailValidator"/>
   </h:inputText>
   <h:message for="email" class="error-message" id="emailError"/>
  </f:ajax>
  <h:outputLabel value="Current date:" for="date"/>
  <h:inputText id="date" label="Change date" readonly="true" tabindex="-1"
   style="width: 100px" value="#{indexController.date}" />
  <h:message for="date" class="error-message" id="dateError"/>
  <h:outputLabel value="Job titel: " for="title"/>
  <h:inputText id="title" required="false" style="width: 300px"
   value="#{indexController.title}" />
  <h:panelGroup/>
  <h:commandButton value="Send" action="#{indexController.add()}" >
   <f:ajax event="action"
    execute="firstname lastname address code city email title"
    render="firstname lastname address code city email title"/>
  </h:commandButton>
```

```
    </h:panelGrid>
    <h:commandLink value="Show addresses" action="list.xhtml" immediate="true" />
  </h:form>
  <h:form id="form2">
  </h:form>
 </h:body>
</html>
```

The validation takes place in the same way, but the writing method is all the more straightforward. Note that it is necessary to specify the fields (for error messages) to be rendered. However, there is one difference since validation occurs every time a lost focus event occurs and hence each time an input field is left. This means that it is only the error message regarding the relevant input component that appears – which may sometimes also be the most appropriate.

5.3 CONVERTERS

In the following example I will show the use of converters where you can attach a converter to an input component. It has not directly to do with the ajax, but is often used in conjunction with ajax. The example is the same as above, and I have started with a copy of *ChangeAddress3*, which I have called *ChangeAddress4*. As an example of using converters, it must be such that the city name will automatically be converted to uppercase letters, while the email address will automatically be converted to lowercase letters. Such a converter can be written as follows:

```
package changeaddress.validators;

import javax.faces.component.UIComponent;
import javax.faces.context.FacesContext;
import javax.faces.convert.Converter;
import javax.faces.convert.FacesConverter;

@FacesConverter("changeaddress.validators.UpperConverter")
public class UpperConverter implements Converter
{
 @Override
 public Object getAsObject
  (FacesContext context, UIComponent component, String value)
 {
  return value.toUpperCase();
 }
```

```
@Override
public String getAsString
 (FacesContext context, UIComponent component, Object value)
{
 return value.toString().toUpperCase();
}
```

and is relatively trivial. There is a corresponding converter class named *LowerConverter*. As an example of using a slightly more interesting converter, the application has been changed so that you can edit the date field again, but I have preserved that the field is initialized with today. On the other hand, I want greater flexibility with regard to entering a date, so it should be allowed to use spaces instead of hyphens. To solve it, I have added the following converter to the program:

```
package changeaddress.validators;

import javax.faces.component.UIComponent;
import javax.faces.context.FacesContext;
import javax.faces.convert.Converter;
import javax.faces.convert.FacesConverter;
```

```
@FacesConverter("changeaddress.validators.DateConverter")
public class DateConverter implements Converter
{
 @Override
 public Object getAsObject
  (FacesContext context, UIComponent component, String value)
 {
  String[] elems = value.split("-");
  if (elems.length == 3) return value;
  elems = value.split("/");
  if (elems.length == 3) return value.replace('/', '-');
  elems = value.split(" ");
  if (elems.length == 3) return value.replace(' ', '-');
  return value;
 }

 @Override
 public String getAsString
  (FacesContext context, UIComponent component, Object value)
 {
  String text = value.toString();
  String[] elems = text.split("-");
  if (elems.length == 3) return text;
  elems = text.split("/");
  if (elems.length == 3) return text.replace('/', '-');
  elems = text.split(" ");
  if (elems.length == 3) return text.replace(' ', '-');
  return value.toString();
 }
}
```

Back there is the application, for example, shown belov is the use of *DateConverter*:

```
<h:outputLabel value="Enter date:" for="date"/>
<h:inputText id="date" label="Change date" style="width: 100px"
 value="#{indexController.date}" >
 <f:validator validatorId="dateValidator"/>
 <f:converter converterId="changeaddress.validators.DateConverter"/>
</h:inputText>
<h:message for="date" class="error-message" id="dateError"/>
```

The other two convertes are used in the same way. Since the value entered in the field must be updated immediately after the entry, the *f:ajax* element must also be updated:

```
<f:ajax event="blur" render="city email date firstnameError lastnameError
 addressError codeError cityError emailError dateError">
```

Looking at the input component *date*, it has both a validator and a converter, and here you should be aware of the order of when the methods are performed:

1. *getAsObject()* // the class *DateConverter*
2. *validate()* // the class *DateValidator*
3. *getAsString()* // the class *DateConverter*

5.4 JSF LISTENERS

JSF components can raise different events when rendered (where the names tells when the events occurs):

- *preRenderComponent,*
- *postAddToView*
- *preValidate*
- *postValidate*

You can define listeners for these events, which are Java code, which are performed on the server. As an example, I have created a copy of the project *ChangeAddress4* and called the copy *ChangeAddress5*. First of all, I have changed the code for *index.xhtml* so that, like in *ChangeAddress1*, it has an ajax function for each input component. As the next step, the *IndexController* is expanded with two methods:

```
public String isWeekend()
{
 String[] elems = person.getDate().split("-");
 Calendar date = new GregorianCalendar(Integer.parseInt(elems[2]),
  Integer.parseInt(elems[1]) - 1, Integer.parseInt(elems[0]));
 return date.get(Calendar.DAY_OF_WEEK) == Calendar.SUNDAY ||
  date.get(Calendar.DAY_OF_WEEK) == Calendar.SATURDAY ? "red" : "darkgreen";
}

public void checkForWeekend(ComponentSystemEvent event)
{
 UIOutput output = (UIOutput) event.getComponent();
 if (isWeekend().equals("red")) output.setValue("Weekend");
 else output.setValue("Everyday");
}
```

The class *Person* has a property *date* initialized with today, and the first method test where this date is for a weekend (Saturday or Sunday). If that is the case, the method returns the text *red* and otherwise the text *darkgreen*. The next method is a listener method. Its parameter is an event and the method determines a reference to the component to which the event relates. If the Person object's *date* property is a date of a weekend, the component's value is set to *Weekend* and otherwise to *Everyday*.

The two methods are basically simple Java methods, and I will now show how they can be used as event handlers in *index.xhtml*. The form is expanded with a new element of the type *outputText* (the only requirement is that it is a component with a value property):

```
<h:outputLabel value="Enter date:" for="date"/>
<h:inputText id="date" label="Change date" style="width: 100px"
 value="#{indexController.date}" >
 <f:validator validatorId="dateValidator"/>
 <f:converter converterId="changeaddress.validators.DateConverter"/>
 <f:ajax event="blur" render="dateError weekend"/>
</h:inputText>
<h:message for="date" class="error-message" id="dateError"/>
```

```
<h:panelGroup/>
<h:outputText id="weekend"
 style="font-style: italic; color: #{indexController.isWeekend()};">
 <f:event type="preRenderComponent"
  listener="#{indexController.checkForWeekend}"/>
</h:outputText>
<h:panelGroup/>
```

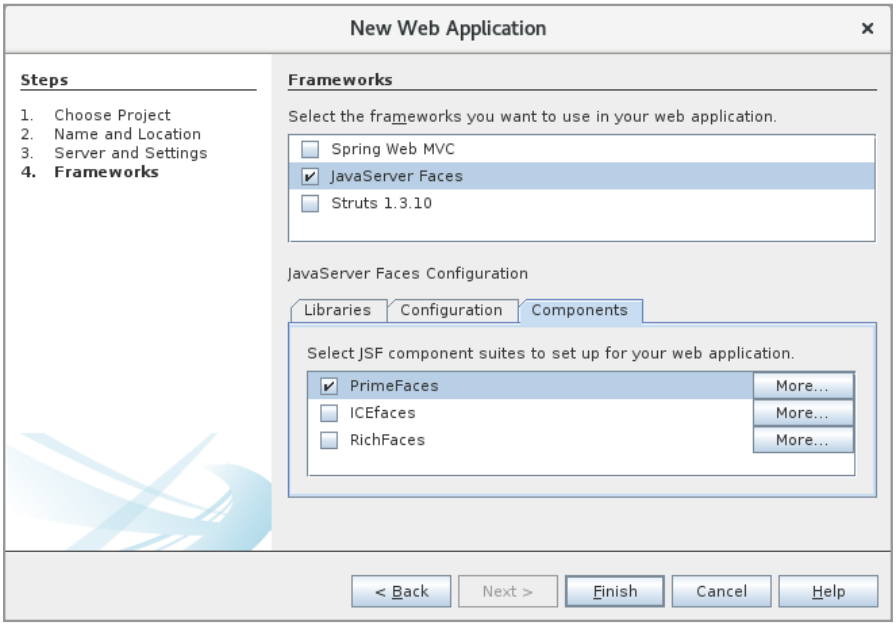
The new elements have an *id* with the value *weekend* as well as a *style* attribute, where you should primarily note that the value of color is determined by the return value from the method *isWeekend()*, which is either *red* or *darkgreen*. As a result, the text appears either as red or green. Finally, the element defines an event of the type *preRenderComonent*. That is, the event handler (the element's listener) is performed before the element is rendered and the handler is the method *checkForWeekend()* in the controller class. It is executed each time the element is rendered, which naturally occurs when the page is rendered, but it also happens when the input component to date is changed, as the ajax function indicates that the element with *id weekend* must be rendered.

6 COMPONENT LIBRARIES

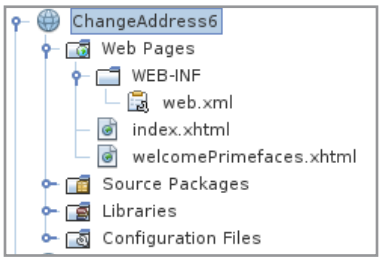
As shown in the previous chapter, JSF and especially in combination with Ajax provide a range of facilities that makes it easy to develop attractive and dynamic web applications, but there is a lot more to do with, and among other things there are several thirds parts JSF component libraries. In this chapter I will give an introductory note to one of these libraries, called *PrimeFaces*. Basically, it's a library that essentially defines its own component for each of the standard JSF components as well as expanding with a skeleton for a web applcatation that makes it easy to develop a stable application from scratch. The motivation for using *PrimeFaces* is primarily to make it easy to develop user-friendly web applications, but also that the library offers other services that JSF does not immediately provide, and in fact, the idea is that the user to the least extent should work with style sheets and JavaScript – two things that in practice can be quite time consuming.

As mentioned, there are other JSF component libraries, but I would like to introduce PrimeFaces, as the library is directly supported by NetBeans, without the need to install the libraries in question (which is also quite easy). One can achieve significant benefits with PrimeFaces, but the price is that you must know the library (a bit like jQuery) and you should be aware that the developers of PrimeFaces have an idea of how web applications should look and work, and if you want to deviate from it, it is not always that easy.

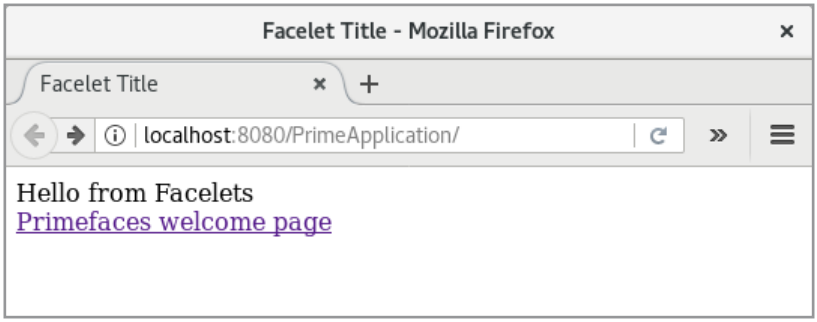
I want to start with a usual web application, which I have called *ChangeAddress6*. When I come to Frameworks, I've chosen *JavaFaces* as usual, and here I clicked on the *Components* tab and chose *PrimeFaces*:



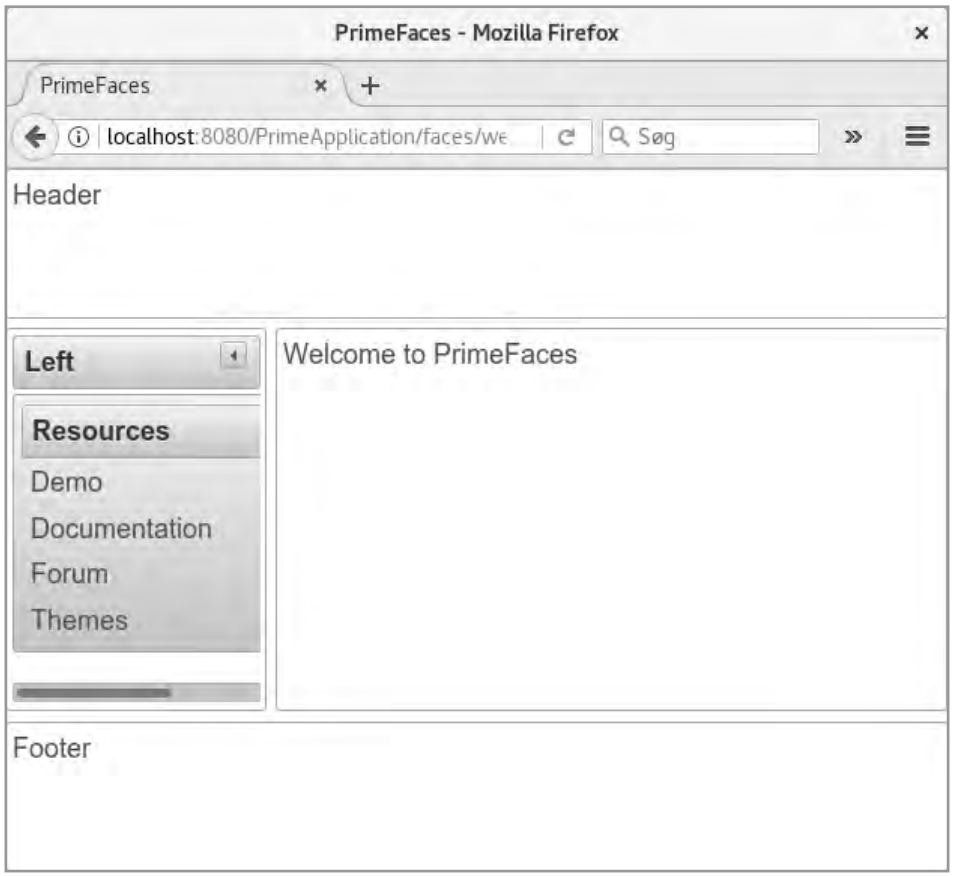
When I then click Finish, NetBeans as usually creates a web application with three files added:



In addition to *index.xhtml* and *web.xml*, another *welcomePrimefaces.xhtml* page has been created, which is the application's start page. The result is a fully completed web application and opens it in the browser, you get the following window:



It is *index.xhtml*, but clicking on the link opens the window:



which is *welcomePrimefaces.xhtml*. That is, NetBeans has generated a skeleton for a page, and it is then the task of the programmer to fill in with something sensible. You should note that the page's design is made without the use of style sheets or JavaScript, but all built into the library.

index.xhtml does not contain anything new or anything about *PrimeFaces*, and of course, the goal is to change it for the specific task. The new things are found in *welcomePrimes.xhtml*, where NetBeans has created the following page:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
 "http://www.w3.org/TR/html4/loose.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:ui="http://java.sun.com/jsf/facelets"
      xmlns:p="http://primefaces.org/ui">
 <f:view contentType="text/html">
  <h:head>
```

```
    <f:facet name="first">
     <meta content='text/html; charset=UTF-8' http-equiv="Content-Type"/>
     <title>PrimeFaces</title>
    </f:facet>
   </h:head>
   <h:body>
    <p:layout fullPage="true">
     <p:layoutUnit position="north" size="100" resizable="true" closable="true"
      collapsible="true">
      Header
     </p:layoutUnit>
     <p:layoutUnit position="south" size="100" closable="true"
      collapsible="true">
      Footer
     </p:layoutUnit>
     <p:layoutUnit position="west" size="175" header="Left" collapsible="true">
      <p:menu>
       <p:submenu label="Resources">
        <p:menuitem value="Demo"
         url="http://www.primefaces.org/showcase-labs/ui/home.jsf" />
        <p:menuitem value="Documentation"
         url="http://www.primefaces.org/documentation.html" />
        <p:menuitem value="Forum" url="http://forum.primefaces.org/" />
        <p:menuitem value="Themes"
         url="http://www.primefaces.org/themes.html" />
       </p:submenu>
      </p:menu>
     </p:layoutUnit>
     <p:layoutUnit position="center">
      Welcome to PrimeFaces
     </p:layoutUnit>
    </p:layout>
   </h:body>
  </f:view>
</html>
```

If you let your eyes run through the code, you'll see that it's all about a template, whatever it's all are about. Starting at the top, you should note that a new namespace has been added:

```
xmlns:p="http://primefaces.org/ui
```

which is the namespace that defines all PrimeFaces elements and must be part of any page that uses PrimeFaces. Then there is the body section where all elements this time are PrimeFaces (*p:*) elements. The first is a *p:layout* element that defines the overall design. This is done by using nested *p:layoutUnit* elements that divide the window into five areas:

1. *north*, which defines an area at the top of the page. The width will automatically be the entire browser window, while the *size* attribute indicates the height.
2. *west*, which defines the left-hand area where the height will automatically be the part of the browser window, which is not used by *north* and *south*, while the width is defined by *size*.
3. *south*, which is the bottom area where the width is automatically the entire browser window, while the height is determined by the *size* attribute.
4. *east*, which is the area to the right, but NetBeans does not insert this area (it is not used so often). The height of the area is the same as *west* while the width is defined by the *size* attribute.
5. *center*, which is simply the browser area not used by the four other areas.

That is, a *p:layout* element divides the window in the same way as it is known from a *BorderLayout* in Java.

Between the individual areas there are lines that you can drag with the mouse and thus change the size of the area – a bit like a *GridSplitter* in a Swing window.

Other attributes are also used, for example, *north* and *south* are defined *closeable* and *collapsible*, which means that the area can be *collapsed* by double clicking on the dividing line. Similarly, the left area is defined as *collapsible*, which occurs by clicking the arrow in the line with the text *Left*.

NetBeans inserts content into the four areas. For the three, it's nothing but a text, while for the left-hand side it's a meun. Here you should especially note that a menu can be nested.

I will now show how I have modified the application to look like the other *ChangeAddress* applications. First, I renamed (the menu item *Refcator*) the page *welcomePrimefaces.xhtml* to *start.xhtml*. Note that this means that it is manually necessary to change the name in the page *index.xhtml*. Then I changed the body part to the following:

```
<h:body>
 <f:facet name="last">
  <h:outputStylesheet name="css/styles.css" />
 </f:facet>
 <p:layout fullPage="true">
  <p:layoutUnit position="north" size="100" resizable="true" closable="true"
   collapsible="true">
   <h1>Change address</h1>
  </p:layoutUnit>
  <p:layoutUnit position="south" size="100" closable="true" collapsible="true">
   <h3 style="text-align: center">Enter your address, your job position and from
 when this information applies</h3>
  </p:layoutUnit>
  <p:layoutUnit position="west" size="175" header="Commands" collapsible="true">
   <h:form>
    <p:menu>
     <p:submenu label="Functions">
      <p:menuitem value="Show addresses" ajax="false" action="list.xhtml" />
     </p:submenu>
    </p:menu>
   </h:form>
  </p:layoutUnit>
  <p:layoutUnit position="center">
   <h:form>
    <p:panelGrid columns="3" columnClasses="rightalign,leftalign,leftalign"
     id="panel">
     <p:outputLabel value="First name:" for="firstname"/>
     <p:inputText id="firstname" label="First name" style="width: 300px"
```

```
 value="#{indexController.firstname}" >
 <f:validateRequired/>
 <f:ajax event="blur" render="firstnameError" />
</p:inputText>
<p:message for="firstname" id="firstnameError" />
<p:outputLabel value="Last name:" for="lastname"/>
<p:inputText id="lastname" label="Lastname" style="width: 200px"
 value="#{indexController.lastname}" >
 <f:validateRequired/>
 <f:ajax event="blur" render="lastnameError"/>
</p:inputText>
<p:message for="lastname" id="lastnameError"/>
<p:outputLabel value="Address:" for="address"/>
<p:inputText id="address" label="Address" style="width: 300px"
 value="#{indexController.address}" >
 <f:validateRequired/>
 <f:ajax event="blur" render="addressError"/>
</p:inputText>
<p:message for="address" id="addressError"/>
<p:outputLabel value="Zip code:" for="code" />
<p:inputText id="code" label="Zipcode" style="width: 60px"
 value="#{indexController.code}">
 <f:validateLength minimum="4" maximum="4"/>
 <f:ajax event="blur" render="codeError"/>
</p:inputText>
<p:message for="code" id="codeError"/>
<p:outputLabel value="City:" for="city"/>
<p:inputText id="city" label="City" style="width: 200px"
 value="#{indexController.city}" >
 <f:validateRequired/>
 <f:converter converterId="changeaddress.validators.UpperConverter"/>
 <f:ajax event="blur" render="cityError"/>
</p:inputText>
<p:message for="city" id="cityError"/>
<p:outputLabel value="Email address:" for="email"/>
<p:inputText id="email" label="Email address" style="width: 300px"
 value="#{indexController.email}">
 <f:validator validatorId="emailValidator"/>
 <f:converter converterId="changeaddress.validators.LowerConverter"/>
 <f:ajax event="blur" render="emailError"/>
</p:inputText>
<p:message for="email" id="emailError"/>
<p:outputLabel value="Enter date:" for="date"/>
<p:inputText id="date" label="Change date" style="width: 100px"
 value="#{indexController.date}" >
 <f:validator validatorId="dateValidator"/>
 <f:converter converterId="changeaddress.validators.DateConverter"/>
```

```
      <f:ajax event="blur" render="dateError weekend"/>
     </p:inputText>
     <p:message for="date" id="dateError"/>
     <p:outputLabel/>
     <p:outputLabel id="weekend"
      style="font-style: italic; color: #{indexController.isWeekend()};">
      <f:event type="preRenderComponent"
       listener="#{indexController.checkForWeekend}"/>
     </p:outputLabel>
     <p:outputLabel/>
     <p:outputLabel value="Job titel: " for="title"/>
     <p:inputText id="title" required="false" style="width: 300px"
      value="#{indexController.title}" />
     <p:outputLabel/>
     <p:commandButton value="Send" actionListener="#{indexController.add}"
      id="commandId" update="panel" />
    </p:panelGrid>
   </h:form>
  </p:layoutUnit>
 </p:layout>
</h:body>
```

Of course there is a lot to note. At the beginning, the loading of a style sheet is defined as I return to below. Next, you can notice how *north* and *south* are changed and that both areas contains only plain HTML. Also, the *west* area has been changed so the menu now has only one menu item with a link to the page that shows an overview of the entered addresses. With only one menu item, it is probably a little overdone to use a *p:menu* element, but I have retained it as an example. The relevant menu item should no longer be translated into a common link, but instead a reference to another document in the same application (like a *commandLink*). This happens by using the attribute *action* instead of *url*, but in return, the menu must be placed in a form, as it should be translated to a submit of this form.

Then there is the *center* area, which contains most of the code copied from the *ChangeAddress5* project. In addition, all Java classes from this project are copied and, moreover, they are completely unchanged. If you consider the *center* area in *start.xhtml*, you can see that essentially nothing has happened except that all *h:* elements are changed to *p:* elements and thus to PrimeFaces elements. However, there are some other changes. *outputText* does not exist as a PrimeFaces element, and here the element is changed to *p:outputLabel*. Earlier, all message elements had a class, but it has now been removed since it no longer has effect. The element *groupPanel* also does not exist as a PrimeFaces element and is replaced by a *p:outputLabel* element. The most important change, however, relates to the element *p:commandButton*. It uses as default ajax, but does not know an *action* event. Instead, an *actionListener* attribute must specify the action to be performed as well as update which sections of the user interface should be updated and here it is the panel containing the input components.

If you see the code you can see that PrimeFaces is quite simple, and the result in web pages that look nice, but if you want to differ from what PrimeFaces have defined, it's not so simple. PrimeFaces is based on style sheets and a number of internal styles, and if you change a page's look and feel, the method is of changing these styles. You can also define styles in the usual way (see the *width* of the above *p:inputText* elements), but it's not always easy to predict the effect as you can not immediately see what the individual PrimeFaces elements are translated into. Here it may be helpful to study the source code in the browser. A *p:panelGrid* by default sets a thin frame without the individual cells and without the entire component, and in this case I wants to remove the frame. In addition, I would like to use a smaller font for error messages, and I have therefore added the following style sheet:

```
.ui-panelgrid > * > tr, .ui-panelgrid > * > tr > td.ui-panelgrid-cell {
  border: none;
}

.ui-message.ui-widget {
 font-size: 10pt;
}
```

and here it's not so easy to know what classes to override. In addition, be aware that the style sheet must be loaded after PrimeFaces' own style sheets are used and one way is to do it at the start the body section as shown in the code above.

Then there is the page *list.xhtml*:

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC … >
<html xmlns="http://www.w3.org/1999/xhtml" … >
 <f:view contentType="text/html">
  <h:head>
   <f:facet name="first">
    <meta content='text/html; charset=UTF-8' http-equiv="Content-Type"/>
    <title>Adresses</title>
   </f:facet>
  </h:head>
  <h:body>
   <p:layout fullPage="true">
    <p:layoutUnit position="north" size="100" resizable="true" closable="true"
     collapsible="true">
     <h1>Addresses</h1>
    </p:layoutUnit>
    <p:layoutUnit position="south" size="100" closable="true"
     collapsible="true">
     <h3 style="text-align: center">Enter your address, your job
     position and from when this information applies</h3>
    </p:layoutUnit>
    <p:layoutUnit position="east" size="275" resizable="false" header="Bird"
     closable="false" collapsible="true">
     <h:form>
      <img src="resources/images/bird.jpg" width="265px" alt=""/><br/><br/>
      <p:commandLink value="Back to start" action="start.xhtml"/>
     </h:form>
    </p:layoutUnit>
    <p:layoutUnit position="center">
     <h:form id="mainForm">
      <p:dataTable id="addrTable" var="addr"
       value="#{indexController.persons}"
       rendered="#{indexController.persons.size() > 0}">
       <f:facet name="header">
        Entered addresses in this session
       </f:facet>
       <p:column id="nameCol">
        <f:facet name="header">Name</f:facet>
        <p:outputLabel id="name" value="#{addr.firstname}
         #{addr.lastname}"/>
```

```
    </p:column>
    <p:column id="addrCol">
     <f:facet name="header">Address</f:facet>
     <p:outputLabel id="addr" value="#{addr.address}"/>
    </p:column>
    <p:column id="cityCol">
     <f:facet name="header">City</f:facet>
     <p:outputLabel id="city" value="#{addr.code} #{addr.city}"/>
    </p:column>
    <p:column id="jobCol">
     <f:facet name="header">Title</f:facet>
     <p:outputLabel id="job" value="#{addr.title}"/>
    </p:column>
    <p:column id="mailCol">
     <f:facet name="header">Mail</f:facet>
     <p:outputLabel id="mail" value="#{addr.email}"/>
    </p:column>
    <p:column id="dateCol">
     <f:facet name="header">Date</f:facet>
     <p:outputLabel id="date" value="#{addr.date}"/>
    </p:column>
   </p:dataTable>
  </h:form>
</p:layoutUnit>
```

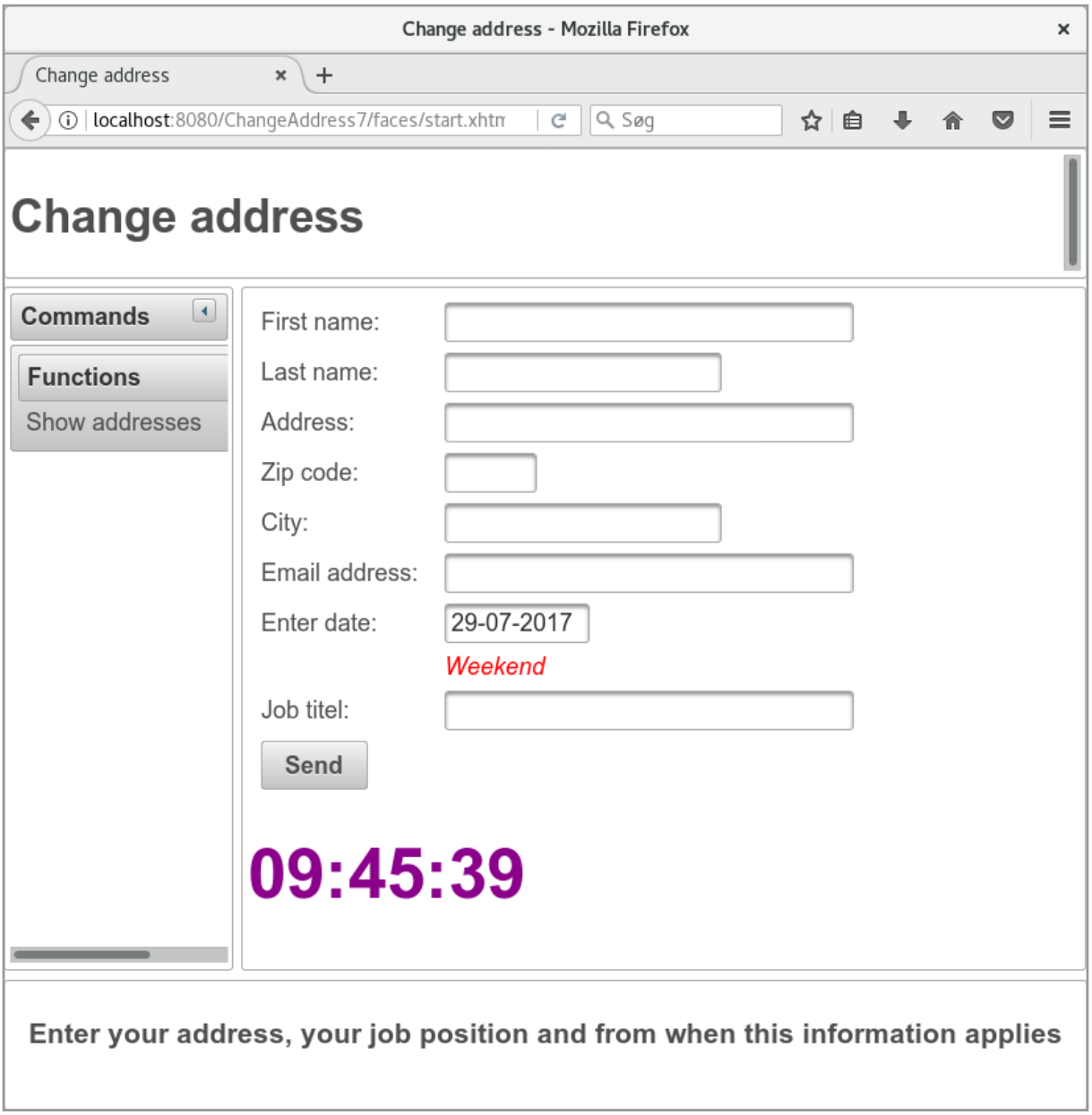
```
    </p:layout>
   </h:body>
  </f:view>
 </html>
```

Again, the code *list.xhtml* is similar to *ChangeAddress5*, and the biggest difference is in addition to *start.xhtml* the *p:layout* elelemtn and most of the *h:* elements are replaced by *p:* elements. The layout is this time expanded with an *east* area – just to show the syntax. In the area there is an image (which is added to the application as a resource) and it is only for there to be something, but also to emphasize that you can perfectly combine PrimeFaces and usual HTML. When you try out the application, note how the table is formatted, and it is one of the most important reasons for using PrimeFaces that you generally get a nice result without having to struggle with style sheets and the like.

6.1 HOW TO POLL THE SERVER

It is always the client (the browser) that addresses the server with a request when something is going to happen, and it is never the server that takes the initiative to contact the client. For example, if you want the user interface updated, the client must send a request to the server. You can automate this behavior using PrimeFaces what the following example will illustrate. The example is an extension of *ChangeAddress6* and is called *ChangeAddress7*, and the form *start.xhtml* has been changed so it displays the clock (see below). In principle, it is not the big challenges and is primarily a matter of expanding *IndexController* with the following property:

```
public String getTime()
{
 Calendar time = Calendar.getInstance();
 return String.format("%02d:%02d:%02d", time.get(Calendar.HOUR_OF_DAY),
  time.get(Calendar.MINUTE), time.get(Calendar.SECOND));
}
```



However, the clock is updated every second, which means that the client must send a request to the server every second to update the clock. It is said that the client should pole the server. This can be implemented using PrimeFaces as follows:

```
<p:poll id="poll" interval="1" update="timelabel"/>
<div class="time-text">
 <p:outputLabel id="timelabel" value="#{indexController.time}"/>
</div>
```

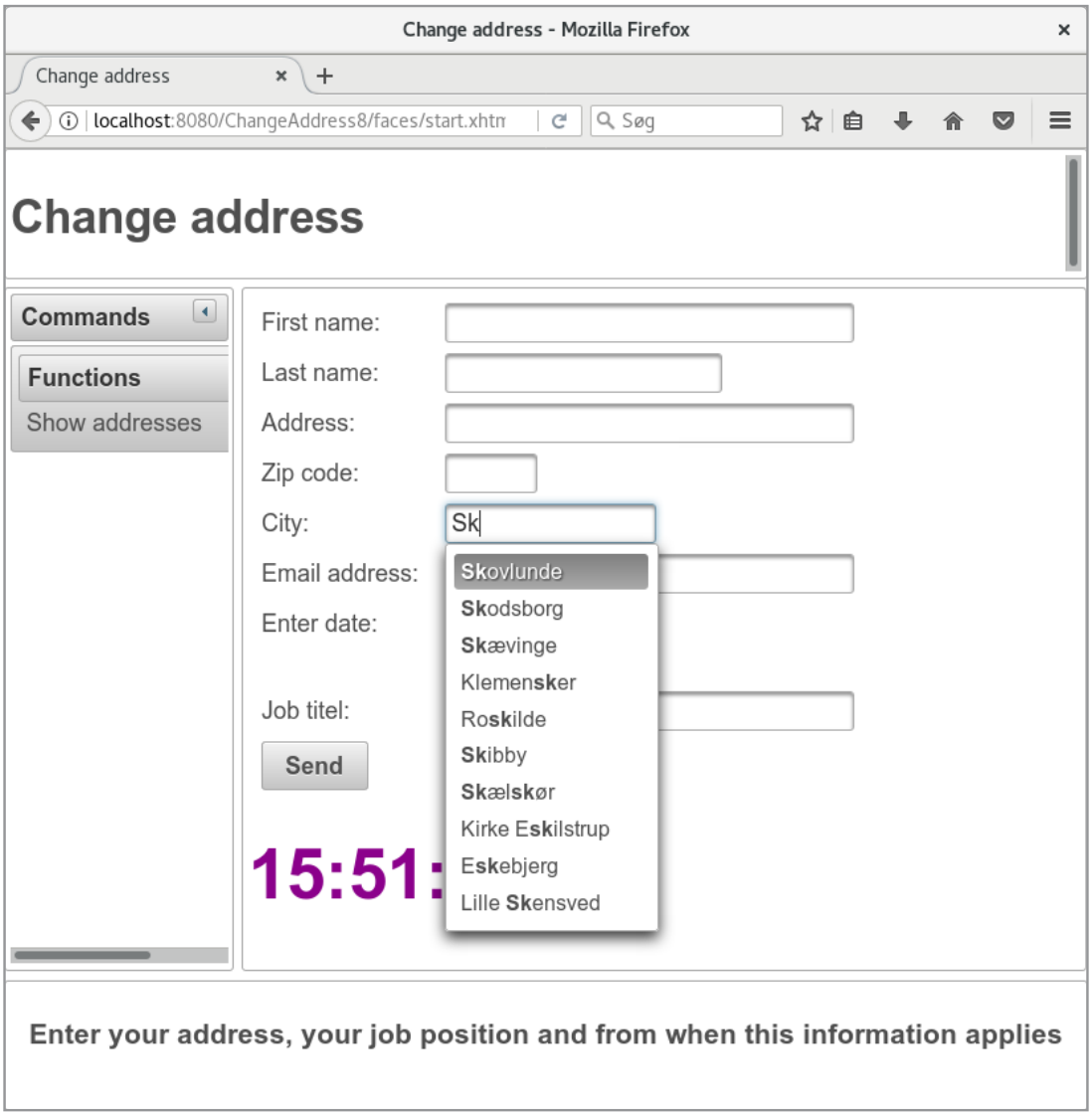
You should note that the element *p:poll* automatically uses ajax, so it is only the current label that is being updated. To display the clock well, the style sheet is expanded with the following class:

```
.time-text {
 margin-top: 30px;
 color: darkmagenta;
 font-weight: bold;
 font-size: 40pt;
}
```

and again, you should note that conventional web technologies can be combined with PrimeFaces without difficulty. The opposite also applies to the use of a function such as *p:poll* without else using PrimeFaces.

6.2 A P:AUTOCOMPLETE ELEMENT

As another application of PrimeFaces, I want to show a so-called autocomplete. The application *ChangeAddress8* is another extension, and if you in the field for the city name begin to enter the name, you get a dropdown list with the name that matches:



Specifically, the program has been modified so that if you are in the field for zip code and enter an existing zip code (only existing zip codes are legal), then the city name will be filled in automatically and if you enter in the city name field and enter text, you will see above a list of city names that match (contains) the entered text and select a city name, the field for zip code is automatically updated.

To solve the task I have first added the following class, which is a table of Danish zip codes:

```
package changeaddress.validators;

import java.util.ArrayList;
import java.util.List;
```

```
public class Zipcodes
{
 public static String getCity(String code)
 {
  for (String[] elem : codeArray) if (elem[0].equals(code)) return elem[1];
  return null;
 }

 public static String getCode(String city)
 {
  city = city.toLowerCase();
  for (String[] elem : codeArray)
   if (elem[1].toLowerCase().equals(city)) return elem[0];
  return null;
 }

 public static List<String> getCities(String text)
 {
  text = text.toLowerCase();
  List<String> res = new ArrayList();
  for (String[] elem : codeArray)
   if (elem[1].toLowerCase().contains(text)) res.add(elem[1]);
  return res;
 }

 private static String codeArray [][] = {
  { "0800", "Høje Taastrup" },
  { "0900", "København C" },
  …
 }
}
```

The methods do not require any particular explanation, but a more dynamic solution would of course be to load the zip codes from a database. A validator class has also been added to zip codes, which do nothing but test if a zip code is found in the table above. As the class does not add anything new, I will not display the code here. Next, the controller class is modified, where I have only shown the code that has changed:

```
public class IndexController implements Serializable
{
 …

 public void setCode(String code)
 {
  person.setCode(code);
```

```
  person.setCity(Zipcodes.getCity(code));
 }
 ...

 public List<String> complete(String text)
 {
  return Zipcodes.getCities(text);
 }

 public void select(SelectEvent e)
 {
  String city = e.getObject().toString();
  setCode(Zipcodes.getCode(city));
 }
}
```

Note that *setCode()* has been changed so it now also initializes the city name from the zip code table. The *complete()* method is the method that returns the list of city names that match the text entered in the city name field. Finally, there is the *select()* method, which is an event handler that is performed when a city name is selected in the list.

Back there is *start.xhtml* where the fields for zip code and city name are changed:

```
<p:outputLabel value="Zip code:" for="code" />
<p:inputText id="code" label="Zipcode" style="width: 60px"
 value="#{indexController.code}">
 <f:validator validatorId="zipcodeValidator"/>
 <f:ajax event="blur" render="codeError city"/>
</p:inputText>
<p:message for="code" id="codeError"/>

<p:outputLabel value="City:" for="city"/>
<p:autoComplete id="city" value="#{indexController.city}" maxResults="10"
 completeMethod="#{indexController.complete}">
 <p:ajax event="itemSelect" listener="#{indexController.select}"
  resetValues="true" update="code" />
</p:autoComplete>
<p:message for="city" id="cityError"/>
```

Regarding entering the zip code, there is not much new, but note that there is another validator attached. Note that it is also defined that the field for the city name must be updated. The most important is by entering city name, where the element has now been changed to a *p:autoComplete* element. When you see the syntax, it's easy enough to figure out what's happening, but note the special way to bind the *complete()* method that returns data to the list, and also note how to reference the event handler. Finally, note that the field for entering the zip code must be updated.

Note that the style sheet is also expanded.

The examples in this chapter show a little bit about what is possible with PrimeFaces, but there is much more, and if you are developing web applications, it's worth working to investigate what PrimeFaces offers. Here you should, among other things, be aware that you can download a documentation from the Internet as pdf:

```
https://www.primefaces.org/docs/guide/primefaces_user_guide_6_0.pdf
```

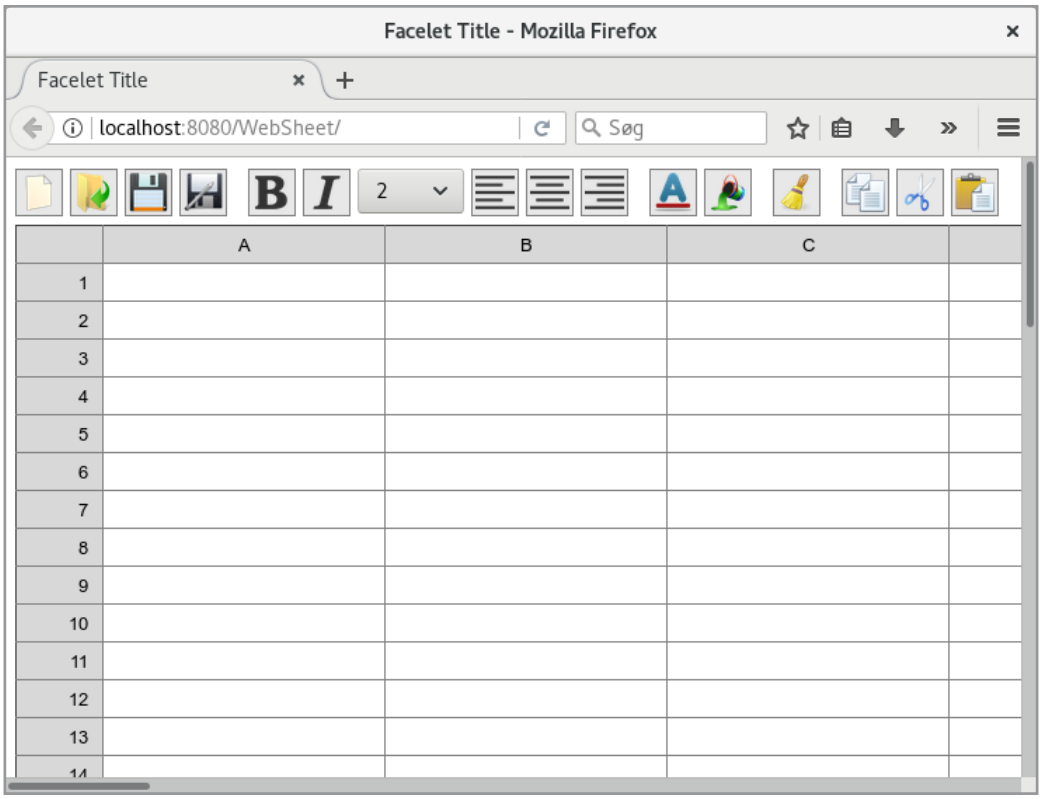
Also, be aware that there is a very good online documentation with examples:

```
https://www.primefaces.org/showcase/
```

Finally, I would like to mention that there are other component libraries for JSF, which it is also worth to look on.

8 WEBSHEET

As the final example of this book, I will write a web application, which is a simple spreadsheet. The application is called *WebSheet*, and if you open it in the browser, you get the following window:



A spreadsheet requires a high user interaction, where the user must be able to enter numbers and formulas in the individual cells, and in the case of a formula, the expression must be evaluated. The client has to do a lot, and the goal of the project is to show how it can be implemented in JavaScript, thus showing something about what is possible in JavaScript. To facilitate the work, I have used jQuery, which is a JavaScript library that provides a wide range of features available, including specially finished elements for the user interface such as dialog boxes, context menus, and more. If you do not know jQuery, you can read the book's appendix A, which gives a brief introduction to jQuery.

As mentioned above, the goal is to show how to use JavaScript to program the client side of a web application, but it is not a fully functional spreadsheet. Not that the program can not be used, but there is a lot that is missing or things that could be better.

8.1 THE PROGRAM'S FUNCTIONS

To enter in a cell, first the cell must be opened that occurs as *CTRL* + *SHIFT* and click with the mouse. Then you can enter a number, a formula or a text. The entry is accepted either by pressing *Tab* or *Enter*.

You can select one or more cells using the mouse and proceed in the usual way by holding down the left button and drag the mouse, and the CTRL and SHIFT keys have the usual functionality.

The program has a toolbar with 16 functions, as mentioned from the left has the following functions:

1. Create a new spreadsheet and enter the number of rows (default 50) and the number of columns (default 20).
2. Open an existing spreadsheet where you can search among the spreadsheets that have been created and saved.
3. Save the spreadsheet that you work – if it has been saved. Otherwise, the *Saveas* function is called.
4. Save the spreadsheet as a new spreadsheet, which means entering a name.

5. Sets the text in the selected cells to bold.
6. Sets the text in the selected cells to italic.
7. Sets the number of decimals for the selected cells.
8. Sets the text in the selected cells to left adjusted.
9. Sets the text in the selected cells to centered.
10. Sets the text in the selected cells to right adjusted.
11. Sets the text color for the selected cells.
12. Sets the background color for the selected cells.
13. Deletes all formatting for the selected cells or for the entire spreadsheet, if no cells are selected.
14. Copies the selected cells.
15. Deletes the contents of the selected cells and stores the content.
16. Inserts cells that are saved or deleted.

In addition, right-clicking on a row header you get a context menu with four functions:

1. Select all cells in the row.
2. Insert an empty row above the current row.
3. Insert an empty row under the current row.
4. Remove the row.

The same goes for columns:

1. Select all cells in the column.
2. Insert an empty column before the current column.
3. Insert an empty column after the current column.
4. Remove the column.

8.2 DESIGN

The content of a spreadsheet must be saved and that have to be done on the server side. The content must thus be sent to and from the server. To facilitate this transport, a spreadsheet is represented as a JSON object. If you do not know about JSON, read this book's appendix B, which gives a brief presentation of JSON. A spreadsheet is represented as an object in the form:

```
var datadefs = (function() {
 public = {};
 public.values = [];
 public.options = {
```

```
 bold: [],
 italic: [],
 align: [],
 background: [],
 foreground: [],
 decimals: []
};
```

values is an array widt the spreadsheets data elements (one element for each data cell). That is, the array does not contain values for row and column headers. The array has one element for each row, where each element is an array with data elements of the form

```
{
 value:
 type:
}
```

Here *value* is the cell's value, and *type* can be *n* for number, *s* for string and *e* for expression. As an example, the data presentation for a blank spreadsheet with 3 rows and 2 columns is shown below.

```
{
 "values":[
  [{"value":"","type":"s"},{"value":"","type":"s"}],
  [{"value":"","type":"s"},{"value":"","type":"s"}],
  [{"value":"","type":"s"},{"value":"","type":"s"}]
 ],
 "options":
  {
   "bold": [],
   "italic": [],
   "align": [],
   "background": [],
   "foreground": [],
   "decimals": []
  }
}
```

The object *options* is used for formatting, and *bold* and *italic* contains objects of the form

```
{
 row:
 col:
}
```

that defines a cell ind the table. *align* contains objects of the form:

```
{
 row:
 col:
 align:
}
```

where the value of the last attribute is *left*, *center* or *right*. The two arrays *background* and *foreground* contains objects of the form

```
{
 row:
 col:
 val:
}
```

where the last attribute is a color value. Finally, the last array *decimals* contains objects of the form:

```
{
 row:
 col:
 dec:
}
```

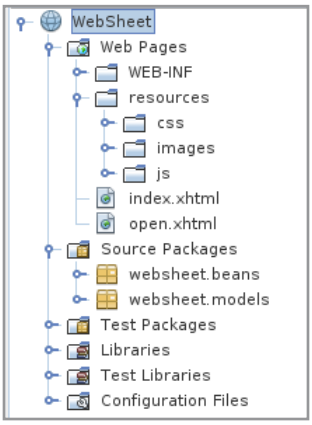
where the last attribute is the number of decimals.

A spreadsheet must be saved somewhere, and a database has been selected for this:

```
use sys;
drop database if exists websheet;
create database websheet;
use websheet;

create table sheets (
 id int auto_increment primary key,
 name varchar(256) not null,
 created date not null,
 modified date not null,
 rows int not null,
 cols int not null,
 content longtext not null
);
```

It is a simple database with only one table. The names tells you what the individual columns are to be used for, but you should especially note the two columns *rows* and *cols*, which contains the number of rows and the number of columns respectively. Finally, note the last column *content* that is for the spreadsheet's content.

Below is an overview of the project's files, and thus the overall architecture. There are two packages for Java classes, and there are two *xhtml* pages, so it is a very simple application. Most of the code however, is found under the *js* folder that contains the JavaScript files.



8.3 PROGRAMMING

I want to start with the server side and here the package *websheet.models*, which contains 7 classes. The user name and password for the database is as previously stored in *web.xml*, and three of the classes are the same as shown in other examples and are used to retrieve the user information and create a connection to the database. The most important class is the class *Repository*, which is a singleton that provides the necessary database operations available, and since the database is simple, this class is similar simple. The package also contains a class *Tools* with some static helper methods. Finally there are two model classes. The one is called *SheetName* and is a class that represents 4 columns from the database table (*id*, *name*, *created* and *modified*). The important thing is that it does not represent the content of a spreadsheet as it can fill a part. The class is used when searching for spreadsheets. The last model class is called *SheetMode* and is derived from *SheetName* and expands with the last three columns. The class is used to represent the worksheet you are working on.

Then there is the package *websheet.beans* containing 6 classes, and here is the class *MainController*, the most important, which is a named bean used by the two *xhtml* pages. Two of the classes are *DateValidator* and *DateConverter*, which are used in *open.xhtml* to validate a date and convert a date (a *Calendar* object) to a string, respectively, and does not contain anything new to what is mentioned in the previous book. Then there are three classes *TableHeader*, *TableRow* and *TableCell*, which are used to construct the HTML table. The classes are really redundant (at least the task could be solved without), but they are included for any extensions of the program, where these classes could be expanded with new properties.

Then there is the class *MainController*, which is a relatively comprehensive class. In principle, it is simple and the most important thing is that it has actions for the first four functions in the toolbar, which are the only functions that call the server side. The class is used as controller for both *index.xhtml* and *open.xhtml*, and you should consider whether the class should be divided into two, so each *xhtml* document had its own controller.

The rest of the code is client code, primarily as JavaScript and style sheets. Regarding the latter, there are 4 style sheets, the two being part of jQuery and are not modified. The other two are relatively simple and are used by *index.xhtml* (*styles.css*) and *open.xhtml* (*dialogs.css*) respectively. There are 8 JavaScript files, but only the 3 are written for this project. The four belongs to jQuery, while the fifth is a standard parser for JSON.

The application largely uses jQuery and in addition to the ability to easily select elements in the DOM tree, jQuery are used for the following:

- to select cells in the table using the mouse, for example, by dragging the mouse
- to open a dialog, either as a simple message box or where the user can enter data
- to a context menu that opens if the user right-clicking on either a row or column header
- to open a simple color selector where the user can choose a color for either text or background

It requires that several files relating to jQuery are downloaded and added to the project. It should be noted that jQuery offers far more than what is used in this example.

For the sake of the current project, 3 JavaScript files have been added:

1. Expressions.js
2. Websheet.js
3. ColorSelector.js

The first contains a single module *expression*, which includes all the functions necessary for formulas. In the book Java 3, in the final example, I have shown a class that has methods that can parse and evaluate a mathematical expression. The module *expression* is primarily a converting of this class to JavaScript, but with two changes:

1. *expression* supports only two mathematical functions, namely the square root function and a sum function. However, also the power function is supported by implementing the operator ^.
2. Instead of common variables, an expression works on cell references, which means that the parser must be changed a bit.

For the *sum* function, it may sum up a number of values (constants or cell references), which are indicated as a comma separated list. You can also specify two cell references in the same row or column and separated by semicolon, and the function summarizes all values between these references.

The implementation of expressions (formulas) is thus relatively simple as one can directly apply the algorithms from Java 3. However, the challenges relate to three situations:

1. When inserting rows or columns, expression's references to cells may change. For example, if you insert a row, all references in expressions that are larger than the index for the new row are counted by 1. The same applies to columns.
2. By copying/pasting formulas, the formula's cell references must be changed in relation to how far a formula is moved.
3. Formulas can refer to other formulas via cell references, and thus there is the possibility of circular references. This problem is partially solved by the fact that after each entry of a formula, the program checks whether the formula results in a circular reference.

Some of the module's methods are used to handle these situations.

Several of the methods may raise an exception and, if it happens, the exception action is an *alert()*. For a finished application, of course, it makes no sense, but I have maintained them, as they may be useful if others want to work on the project. In fact, it is relatively difficult to troubleshoot JavaScript code, and one of the difficulties with JavaScript is that there are very many posibilities for errors, and it is difficult to test the code for errors. In an operational situation, you could therefore replace the relevant alert's with a request to the server and, using ajax, send the appropriate error messages to the server that could save them in a log file or a database table. This allows you to check on a regular basis whether the code results in errors, what has caused the error, that can make the maintenance of the program considerably easier.

Then there is the JavaScript file *websheet.js*, which contains all the script that will be used to handle the user interaction. The file is extensive and consists of three modules:

1. *datadefs*, which is the data presentation of the current spreadsheet
2. *sheet,* containing all methods to manipulate the spreadsheet and then all functions used from the toolbar
3. *handlers,* that contain event handlers to edit the content of cells

The first I mentioned during the design and the last one is relatively simple. In order to edit the content of a cell, a cell in the table contains two components, one being a *div* element with a *span* element, which generally shows the text. It is the content of this element that is formatted according to the selected formats. The other element is an input component that is generally invisible. Holding down the CTRL and SHIFT keys and clicking the mouse will change the visibility of these two components, and you can edit the content. When you click again or enter *Tab* or *Enter* switches on visibility again and the user interface is updated. The latter is not quite simple. First, it may mean that formulas elsewhere in the spreadsheet must be updated if they depend on the cell that has been changed. Secondly, in case it is a formula that is entered, it must be validated that it not results in a circular reference. These are things that are handled by functions in the *handlers* module.

There is also a function *initCells()* that is not part of the above modules. The task of the function is to associate the event handlers in the module *Handlers* to the table's cells. This function is used when the window is initialized by the browser.

An important part of the module *sheet* relates to formatting, in which you must be able to format cells, ensure that the formatting are saved and used again when a saved spreadsheet is loaded. In addition, formatting may be modified if you insert/delete rows and columns. You should be aware that formatting is not copied in connection with copy/paste.

Another part concerns the toolbar's buttons to open and save spreadsheets, where requests are sent to the server. In particular, be aware that a spreadsheet is saved, asynchronously, using ajax.

Finally, there are the buttons to copy/paste, and especially the event handler for paste is not simple. The functions do not use the machine's clipboard, but instead an internal array. The reason is to make it simple, but you should note that there is actually an API so that you from JavaScript can use the system clipboard.

Then there is the file *ColorSelector.js*, which makes nothing but defining a simple object for color selection. Only a few colors can be chosen, but jQuery offers a more advanced color selection dialog that would be a good option. When I have chosen to write my own, it is alone to make the program simple.

Back there are the *xhtml* pages where there are two. *open.xhtml* is a simple form that is used to open a spreadsheet. You can search the database by name and date, and the page shows a list of spreadsheets that exist and matches the search criteria. Then there is *index.xhtml*, which is the home page and the actual spreadsheet. The page does not fill much, but in addition to the spreadsheet's table, there is the toolbar and dialog boxes that jQuery implements using hidden *div* elements. You should also note that the communication with the server is primarily done by using hidden fields at the end of the page.

8.4 CONCLUSION

As mentioned, the purpose of this project is to show a little about what is possible with JavaScript, and in fact, it is possible to program complex logic that is performed on the client side alone. JavaScript is in fact a fairly complex programming language, and there is a lot more to learn than what this book is about. However, it is not more difficult to learn JavaScript than another programming language, and if you need to use JavaScript to a large extent, there is nothing more than going systematically, for example, reading a book exclusively related to JavaScript. However, there is one thing that should be noted that since JavaScript is not translated by a compiler, and is not type-strong, there is a high risk that the code contains errors and, worse, it is both difficult and time consuming to find the mistakes. The conclusion is, therefore, that JavaScript is a fully-fledged programming language, but it is far from an effective language for developing major programs.

The current project is not a complete spreadsheet, which can be used as an alternative to, for example, *LibriOffice Calc* – nor does it make sense to try to develop such an alternative. However, the project may have applications, and if you think of a web application that needs a spreadsheet-like page, the project or parts of it may be used. Should the project be expanded, so it looks like a real spreadsheet, there are some missing and I will mention some features that could significantly improve the program. In fact, each feature could be a task or project in itself.

1. Data entry should be improved as you do not always have to click the mouse to open a cell. It is primarily a question about to treat events related to the keyboard, such as arrow keys.
2. The program should apply the system clipboard so that you can copy data to other programs. It is primarily a matter of getting into the corresponding JavaScript API.
3. The expression module should be expanded with more mathematical functions. It's actually a relatively simple task, and the algorithms can be found in the book Java 3.
4. However, the biggest lack is performance where the application becomes ineffective on large spreadsheets. An option to solve the problem could be to show only a part of a spreadsheet on the client side and then place some navigation buttons in the toolbar. This improvement is not quite simple.

APPENDIX A: JQUERY

In chapter 4 on JavaScript, I mentioned jQuery, which is a large library of JavaScript functions, and if you work as a web developer, you can not avoid jQuery and, if nothing else, jQuery provides so many functions that it is worth the effort to investigate what it is. So, therefore, a brief introduction, but also because I have actually used jQuery in the final example in the last chapter.

First, you have to grab the product and that is the latest version, as new versions are constantly being added. You can go to the page

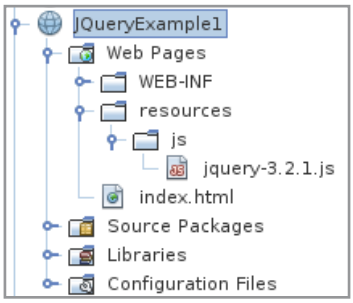
```
https://jquery.com/download/
```

from where you can download the product as a JavaScript file, and thus as a plain text file with the code (right click on the link and select *Save link as*). In fact, you must download two files:

1. *jquery-3.2.1.js*
2. *jquery-3.2.1.min.js*

where the version number can of course differ. Both files contains the same JavaScript code, and the difference is, that the latter is compressed, so it is significantly smaller (about one third). The goal is that while developing, you can use the first one where the code is written in readable form and with comments, and when the program is finished, you can replace it with the last one. Here you must remember that JavaScript is text that is sent along with the HTML document, and you are therefore for the sake of bandwidth interested in sending as few data as possible.

After downloading the files, nothing else must happen, and you are ready to use jQuery in your projects (and that is what of it you want to apply). As an example, I have created a project named *JQueryExample1*:



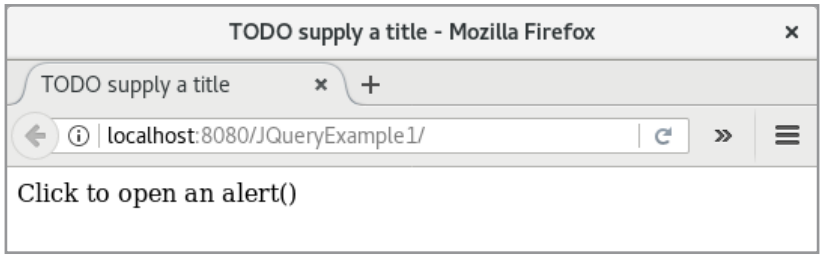
As you can see, under the directory *resources*, I have created a subdirectory named *js* and placed *jquery-3.2.1.js* there. The content of *index.html* is as follows:

```
<!DOCTYPE html>
<html>
 <head>
  <title>TODO supply a title</title>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <script src="resources/js/jquery-3.2.1.js" type="text/javascript"></script>
  <script>
   $(document).ready(function() {
    $("div").click(function() {
     alert("Hello world!");
    });
   });
  </script>
 </head>
 <body>
  <div>Click to open an alert()</div>
 </body>
</html>
```

First, note that in the header there is a script element that refers to the jQuery file. In addition, there is a script element with some JavaScript code that uses jQuery. Here means

```
$(document).ready
```

that the following JavaScript code is executed – after the DOM tree is constructed, but before the tree's elements are rendered by the browser. In this case, it means performing an anonymous function that has one statement that associates a *click* event handler with an *alert()* to all *div* elements. Specifically, note the syntax *$()*, which is a jQuery function, which refers to a collection of elements in the DOM tree, and you can then perform a JavaScript function on these items. In this case, the *body* part has only a single *div* element and if you opens the document in the browser, the result is:



What is not very mysterious, but if you click on the text, you get a popup:

The example shows two things. First of all, what is technically necessary to use jQuery, and second, what is the idea of jQuery, that is, in an easy way to select items in the DOM tree and do something with them. The first goes quite easily and you simply copy the library to the project (in fact there is an option since multiple browser vendors make the library available and you can just link to the current library). As for the other, it all builds on the *$()* function, which is just an alias for *jQuery()*, and the above script block could be written as follows:

```
<script>
 jQuery(document).ready(function() {
  jQuery("div").click(function() {
   alert("Hello World!");
  });
 });
</script>
```

(the project *JQueryExample2*). Viewed from the program, it does not matter what you write, but it is standard to write *$()*, what can be recommended. The use of jQuery is so widespread that most developers perceives *$()* as jQuery code.

A whole different thing is what you can use as an argument for the function *$()* and which functions you can specify should work on the elements (*click()* is an example) and here there are simply many options, a lot more than can be accommodated in this book so it is necessary to read the documentation:

```
http://www.tutorialspoint.com/jquery/jquery_tutorial.pdf
```

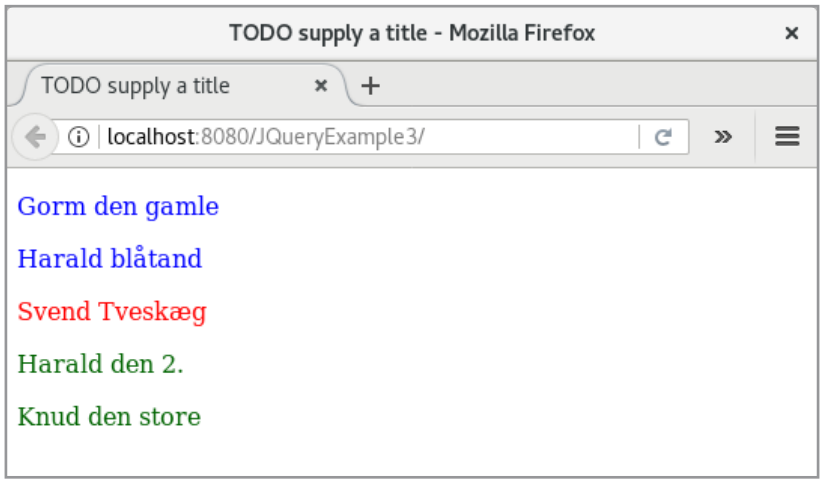
which provides an adequate and easily readable documentation. I would like to justify, with a few examples of what you can with jQuery.

Generally, the parameter of *$()* may be any selector corresponding to what is discussed in chapter 3 on style sheets. Consider, for example, the following HTML document (*JQueryExample3*):

```
<!DOCTYPE html>
<html>
 <head>
  <title>TODO supply a title</title>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <script src="resources/js/jquery-3.2.1.js" type="text/javascript"></script>
  <script>
   $(document).ready(function() {
    var e = $("p");
    for (var i = 0; i < e.length; ++i) e[i].style.color = 'darkgreen';
    var c = $(".first");
    for (var i = 0; i < c.length; ++i) c[i].style.color = 'blue';
    var v = $("#viking");
    for (var i = 0; i < v.length; ++i) v[i].style.color = 'red';
   });
  </script>
```

```
 </head>
 <body>
  <div>
   <p class="first">Gorm den gamle</p>
   <p class="first">Harald blåtand</p>
   <p id="viking" class="first">Svend Tveskæg</p>
   <p>Harald den 2.</p>
   <p>Knud den store</p>
  </div>
 </body>
</html>
```

The *body* part is simple and consists of 5 paragraph elements in a *div* element. You should note that the first three defines a class attribute (although no equivalent css class is defined in any place), and the third element has an ID. The JavaScript code (the jQuery function *$().ready()*) performs a function that starts by selecting all paragraph elements, and the result is a collection of 5 elements. The next *for* loop iterates over these elements and turns the color to green. Next, the same is done where the color is set to blue, but only for the elements whose class is *first*. Finally, the color is set to red, but this time only for those elements that have a specific *id* (and there is only one). If the document is opened in the browser, you get the result:



In particular, note that the elements in the DOM tree are processed by the JavaScript code before the document appears in the browser.

The above is easy enough to understand, but it can be written easier using a jQuery function:

```
<!DOCTYPE html>
<html>
 <head>
  <title>TODO supply a title</title>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <script src="resources/js/jquery-3.2.1.js" type="text/javascript"></script>
  <script>
   $(document).ready(function() {
    $("*").css("font-size", "24pt");
    $("p").css("color", "darkgreen");
    $(".first").css("color", "blue");
    $("#viking").css("color", "red");
    $(".last, #viking").css("font-size", "36pt");
   });
  </script>
 </head>
 <body>
  <div>
   <p class="first">Gorm den gamle</p>
   <p class="first">Harald blåtand</p>
   <p id="viking" class="first">Svend Tveskæg</p>
   <p class="last">Harald den 2.</p>
   <p class="last">Knud den store</p>
  </div>
 </body>
</html>
```

Note that the last two paragraph elements this time has a class *last*. If you consider the first select in the JavaScript function, it selects all elements and sets the font size to 24 points, but it is done using the function *css()*, which is a jQuery function that works on all elements that are selected. The same applies to the next three statements, which also use the function *css()* instead of writing the necessary loops. This is where you meet the strength of jQuery, and there are simply a large number of functions that you can read all about in the documentation mentioned above. In particular, note the last statement that sets the font to 36 points for all elements that either have id *viking* or class *last*.

As another example (*JQueryExample5*), a document similar to the above is shown below, but where the example shows the use of multiple functions:

```
<!DOCTYPE html>
<html>
 <head>
  <title>TODO supply a title</title>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <script src="resources/js/jquery-3.2.1.js" type="text/javascript"></script>
  <style>
   .red-class {
    color: red;
   }
   .blue-class {
    color: blue;
   }
   .green-class {
    color: darkgreen;
   }
  </style>
  <script>
   $(document).ready(function() {
    var text = $("p").attr("title");
    $("h1").text(text + "s");
    $("#viking").attr("title", "Last Danish viking King")
    $(".first").addClass("red-class");
    $(".last").addClass("blue-class");
    $("#viking").click(function () {
     $(this).toggleClass("green-class");
    });
    alert($("h1").html());
    $("h2").html("but there are more Kings and Queens")
   });
  </script>
 </head>
```

```
 <body>
  <div>
   <h1></h1>
   <p class="first" title="The first Danish king">Gorm den gamle</p>
   <p class="first">Harald blåtand</p>
   <p id="viking" class="first">Svend Tveskæg</p>
   <p class="last">Harald den 2.</p>
   <p class="last">Knud den store</p>
   <h2></h2>
  </div>
 </body>
</html>
```

Note first that the *body* part is essentially the same as in the previous example but it is expanded with a *h1* element and a *h2* element, both of which are empty. Also note that a *title* attribute has been added to the first paragraph element. Then note that three simple styles are defined with each their class name.

Then the jQuery block (the function). First, the value of the *title* attribute is saved in a variable. This is done by selecting all paragraphs and then using the *attr()* function. It returns the value of *title* for the first element, that has a *title* attribute, and in this case there is only one element (which is the first). The value of the variable is then used to associate a text with the *h1* element, which occurs with the function *text()*. The third statement defines a *title* attribute of the element with id *viking*, which again occurs with the *attr()* function in an override, which specifies the attribute as well as the value. Then is attached a class *red-class* to all elements whose class is *first*. Please note that an element may have more values for the class – an element can be of more classes. The next statment performs the same for all elements whose class is *last*, but this time the class *blue-class* is added.

As a next step is added an event handler for click events to the element with id *viking*. The result is that if you click on the element, the color will change from red to green and click again, the color will change back to red. The next last statement shows an *alert()* with the value of *innerHtml* for the element *h1* while the last initializes the *h2* element. If you open the document, you first get the following popup and then the following window:

Finally, I will show an example that modifies the DOM tree:

```
<!DOCTYPE html>
<html>
 <head>
  <title>TODO supply a title</title>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <script src="resources/js/jquery-3.2.1.js" type="text/javascript"></script>
  <script>
   jQuery(document).ready(function() {
    $("div").append("<p id='norwegian'>Magnus</p>");
    $("#norwegian").before("<p>Hardeknud</p>");
    var text = $("#norwegian").text();
    $("#norwegian").replaceWith("<h2><span>" + text + "</span> den gode</h2>");
    $("span").css("color", "blue");
   });
  </script>
 </head>
 <body>
  <div>
   <h1></h1>
   <p class="first" title="The first Danish king">Gorm den gamle</p>
   <p class="first">Harald blåtand</p>
   <p id="viking" class="first">Svend Tveskæg</p>
   <p class="last">Harald den 2.</p>
   <p class="last">Knud den store</p>
  </div>
 </body>
</html>
```


TODO supply a title - Mozilla Firefox
TODO supply a title
localhost:8080/JQueryExample6/
Gorm den gamle
Harald blåtand
Svend Tveskæg
Harald den 2.
Knud den store
Hardeknud
Magnus den gode

As for the *body* part, there is nothing new compared to the previous element, only that the *h2* element is removed. The jQuery function starts by adding a *p* element to the *div* element. This happens with the function *append()*, which adds the element as the last child to the *div* element. The next statement uses the function *before()* to add another *p* element, but this time an element that is also a child to the *div* element, but in front of the element just inserted (the element with the value *Magnus*). The third statement stores the value of the element with id *norwegian* in a variable, and the next statement replaces the element with id *norwegian* with another element (an *h2* element containing a *span* element). As the last changes the color of the element is changed to blue.

The above is only a hint of what is possible with jQuery and you are encouraged to investigate what else is. In particular, I would like to mention that the library also contains functions that use ajax. This means that you using JavaScript can communicate asynchronously with the server, which gives a whole new dimension regarding the use of JavaScript.

In addition to what is mentioned above, there are also more extensions of jQuery, and in particular I want to mention *jQuery UI*, which can be downloaded from:

```
https://jqueryui.com/
```

It is an extension that defines a number of elements for the user interface such as dialogs and the ability to select elements using the mouse – and more. When you download the product, you will get a JavaScript file (*jquery-ui.js*) and an associated style sheet (*jquery-ui.css*), and a project that illustrates the application.

APPENDIX B: JSON

JSON is a standard for data exchange primarily between client and server in web applications, but in principle, JSON can be used in many other contexts, and in relation to data exchange, JSON can to some extent be seen as an alternative to XML. JSON is characterized by being easy to write and read (there are very few and simple rules), being text based and being platform independent.

A JSON document has in principle the same syntax as an object in JavaScript, and an example could be:

```
danish = {
 "kings": [
 {
  "name":"Gorn den Gamle",
  "to": "958"
 },
 {
  "name":"Harald Blåtand",
  "from":"958",
  "to": "987"
 },
 {
  "name":"Svend Tverskæg",
  "from":"987",
  "to": "1014"
 }
 ]
};
```

The basic syntax is that data are represented as key/value pair separated by colon. Several data elements (key/value pairs) can be gathered as objects in a collection where the elements are separated by commas. Finally, the value of a data element may be an array. In general, JSON supports the following data types:

1. *Number* as in JavaScript
2. *String* that is double-quoted Unicode with backslash as escaping
3. *Boolean* that is *true* or *false*
4. *Array* which is an ordered sequence of values separated by comma
5. *Object* that is an unordered collection of key:value pairs

The above can therefore also be written as follows, where there are no quotation marks around the years:

```
danish = {
 "kings": [
 {
  "name": "Gorn den Gamle",
  "to": 958
 },
 {
  "name": "Harald Blåtand",
  "from": 958,
  "to": 987
 },
 {
  "name": "Svend Tverskæg",
  "from": 987,
  "to": 1014
 }
 ]
};
```

With regard to escaping of characters in strings, the same symbols are used as in Java and JavaScript. Put a little differently, a JSON data structure has the same syntax as a JavaScript object consisting solely of properties.

As an example, is below shown some JavaScript that prints the above JSON data structure:

```
<script>
 for (var i = 0; i < danish.kings.length; ++i)
 {
  var king = danish.kings[i];
  document.write(king.name);
  document.write("<br/>");
  if ("" + king.from !== 'undefined' && "" + king.to !== 'undefined')
   document.write("From: " + king.from + " to: " + king.to);
  else if ("" + king.from !== 'undefined') document.write("From: " + king.from);
  else if ("" + king.to !== 'undefined') document.write("To: " + king.to);
  document.write("<br/>");
  document.write("<br/>");
 }
</script>
```

As another example, is shown a JSON structure that defines a 2-dimensional array:

```
numbers = {
 "primes" : [
  [2, 3, 5, 7],
  [11, 13, 17, 19],
  [23, 29],
  [31, 37],
  [41, 43, 47]
 ]
};
```

and the following JavaScript block prints the numbers:

```
<script>
 for (var i = 0; i < numbers.primes.length; ++i)
 {
  for (var j = 0; j < numbers.primes[i].length; ++j)
   document.write(numbers.primes[i][j] + " ");
 }
 document.write("<br/>");
</script>
```

As long as you only have to manipulate JSON data on the client page, there is actually not much else to tell than what appears from the above, but if data has to be sent to or received from a server, they must be parsed to be used. For example, if you want to send JSON data from the client to a Java bean on the server side, and afterwards it should be able to use that data, they must be parsed or decoded to a Java object, using standard parsers that can be downloaded and used by the server code. Similarly, on the server side, you can encode beans as JSON data before they are sent to the client, and on the client side, correspondingly, you must download JavaScript parsers that can decode the data sent from the server. I do not want to review in this place, but I can mention that it is quite simple and requires only some browsing on the Internet.