JavaScript and the HTML5 `<canvas>` Element

Apostolos Syropoulos
Xanthi, Greece
E-mail: `asyropoulos@aol.com`

Preface

This is a short introduction to JavaScript programming that has been designed for people who want to learn how to create simple or not so simple animations with the HTML5 `<canvas>` element. The text assumes that the reader is familiar with certain basic things (the use of a browser and a text editor), which I believe are common knowledge nowadays. This booklet is based on a pamphlet describing elements of JavaScript programming. The pamphlet was given to pupils that attended an HTML class in a vocational school. Originally, the text was in Greek. Later on, I translated it in English and I added the introduction to HTML5 and the `<canvas>` element, while the rest of the text was slightly updated.

The text starts with an introduction to HTML5, something quite necessary since JavaScript code is part of HTML documents and affect their functionality. Then, I gradually introduce JavaScript by starting with values and variables. Next, I introduce expressions and operators and the basic commands of the language. JavaScript has many predefined function (methods in object-oriented parlance) and some of them are very important for every programmer. These functions are presented next. Then, I show how one can handle events that may happen on a web page (e.g., what should happen when someone presses a button). The Document Object Model is the way to get access to the various elements that make up a web page and the 8th chapter discusses it. In chapter 9, I introduce two predefined classes that are quite useful. The last chapter shows how one can create simple and not so simple animations with the HTML5 `<canvas>` element and JavaScript.

Apostolos Syropoulos
Xanthi, Greece

December, 2018

Contents

Chapter 1

HTML5 in a Nutshell

In this chapter I will introduce the basics of the HTML5 markup language. JavaScript code is part of an HTML document and affects its rendering and functionality.

1.1 What is HTML5?

HTML5 is the very recent version of the *HyperText Markup Language*. A *markup* is a sequence of characters that you insert at certain places in a text or word processing file to indicate how the file should look when it is printed or displayed or to describe the document's logical structure. The following are some examples of textual markup that show how one can specify boldface text in various markups.

- LaTeX markup:
  ```
  normal \textbf{bold} normal again
  ```
- troff markup:
  ```
   normal
   .ft B
   bold
   .ft R
   normal again
  ```
- HTML markup:
  ```
  normal <b>bold</b> normal again
  ```
- Wiki markup:
  ```
  normal '''bold''' normal again
  ```

The prefix hyper- generally means "above" or "beyond", thus hypertext is something that goes beyond the limitations of ordinary text (e.g., it can be non-sequential). Hypertext is text which contains links to other information. The term was coined by Theodor

Holm "Ted" Nelson around 1963. Software programs such as dictionaries and encyclopedias have long used hypertext in their definitions allowing readers to quickly navigate content. HyperMedia is a term used for hypertext which is not constrained to be text: it can include graphics, video and sound, etc. Ted Nelson coined this term too.

A markup language is a set of rules specifying how to mark up content so to control its structure, formatting, or the relationship among its parts. The most widely used markup languages are SGML, HTML, and XML. SGML (i.e., the *Standard Generalized Markup Language*), is a language for defining markup languages. HTML is an "application" of SGML. XML stands for eXtensible Markup Language and it was designed to store and transport data. XML was designed to be both human- and machine-readable. HTML is the de facto standard for the presentation of information over the Web.

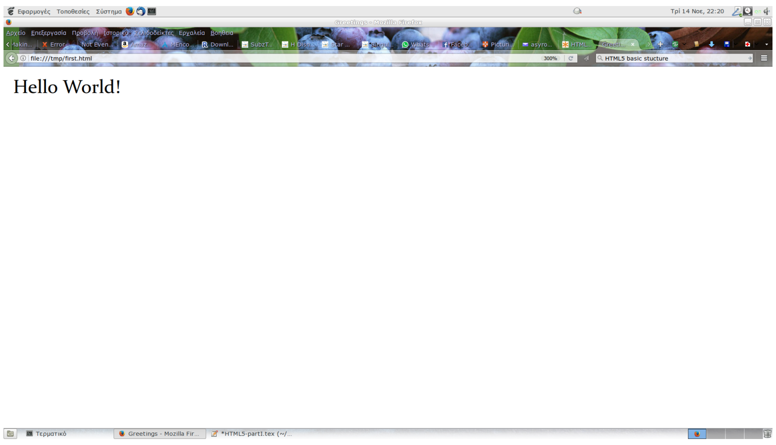
Tim Berners-Lee created what we call the Web while he was working at CERN. He created HTML and the HyperText Transfer Protocol (HTTP) and designed and implemented the first web browser. On October 1991 "HTML Tags" was made available. A description of version 2.0 of HTML was published on November 1995. HTML 3.2 was published on January 1997. HTML 4.0 was published on December 1997. HTML5 was published on October 2014 while HTML 5.1 was published on November 2016. The latest version of HTML5 is HTML 5.3 and its specification was published on 18 October 2018.

1.2 The Sturcture of an HTML5 Document

An HTML5 document has a few sections that are used to present different kinds of information. The following few lines show these sections of a simple yet complete HTML5 document.

```
<!DOCTYPE html> ⟵ document type
<html> ⟵ beginning of document
<head> ⟵ beginning of head
<meta charset="UTF-8">
<title>Greetings</title> ⟵ the title
</head> ⟵ end of head
<body> ⟵ beginning of body
Hello World! ⟵ document contents
</body> ⟵ end of body
</html> ⟵ end of document
```

The arrows and what follows them are not part of the document. Ideally, one should use an editor to type the markup, without the arrows and what follows them, and save the content to a file whose name extension is `html` (e.g., `Example.html`). Then, one can open



Hello World!

Figure 1.1: A very simple HTML5 web page.

the file with a browser and see the result, which should look like the screenshot in Fig. 1.1.

Let us examine each section of the document in some detail. Everything that appears in an HTML5 document that is enclosed in angle brackets is called an *element*. The `<head>` element is a container for metadata (data about data). This element can contain a number of other elements:

`<title>` Is used to specify the title of the document and appears on the browser's toolbar.

`<style>` An element that is used to define *style* information for a single HTML5 page, that is, *typographic* information (e.g., text justification, font selection, etc.).

`<link>` This element is used to load external files with style information.

`<meta>` An element is about meta-information that is used to specify which character set is used, what this document is about, which are the keywords, which are the authors, etc. Here are the various forms of this element:

  `<meta charset="`*`encoding`*`">` It is used to specify the character *`encoding`* of the document (usually, Unicode's `UTF-8`).

  `<meta name="description" content="`*`description`*`">` Used to provide a brief description of a document's contents (e.g., `HTML5 examples`).

  `<meta name="keywords" content="`*`keywords`*`">` Can be used to to specify the keywords of the document (e.g., `HTML5, CSS3`).

  `<meta name="author" content="`*`name`*`">` The name of the author of this document (e.g., `James T. Kirk`).

What is Text Encoding? Text in a computer system is made of characters. Each character is represented internally using an *encoding*. There are many different character encodings that have been used over the years in different locations. In one common encoding the letter "é" is represented internally by the number 233 while in another encoding this number represents the letter "щ". Thus when one prepares an email message using the first encoding and the receiver's computer system supports only the second encoding, quite probably the message will be unreadable. To overcome this problem a consortium of companies created the Unicode standard which provided support for all known scripts. Unicode is an evolving standard as it includes ancient scripts and various symbols used in mathematics, technology, etc. Initially, each character was represented by two bytes. However, this representation did not work for Unix and Unix-like systems, since most their source was written before the emergence of Unicode. Thus a *variable length* encoding was devised by Ken Thompson and Rob Pike to allow the use of Unicode in Unix systems. This encoding is called UTF-8 and it is know the most common encoding of Unicode. When preparing a web page that contains some "uncommon" characters, one should make sure that `<meta charset="UTF-8">` is specified in order to make sure that all the characters of a the document will be displayed correctly.

1.3 Text and Alignment

The element `<p>` is used to delimit paragraphs. For example, the markup

```
<p>This is a paragraph.</p>
```

shows how to delimit a paragraph. Note that the tag `</p>` is the *closing* tag while `<p>` is the opening one. Clearly, most HTML5 elements follows this "practice". To override the default paragraph formatting one should use a style parameter:

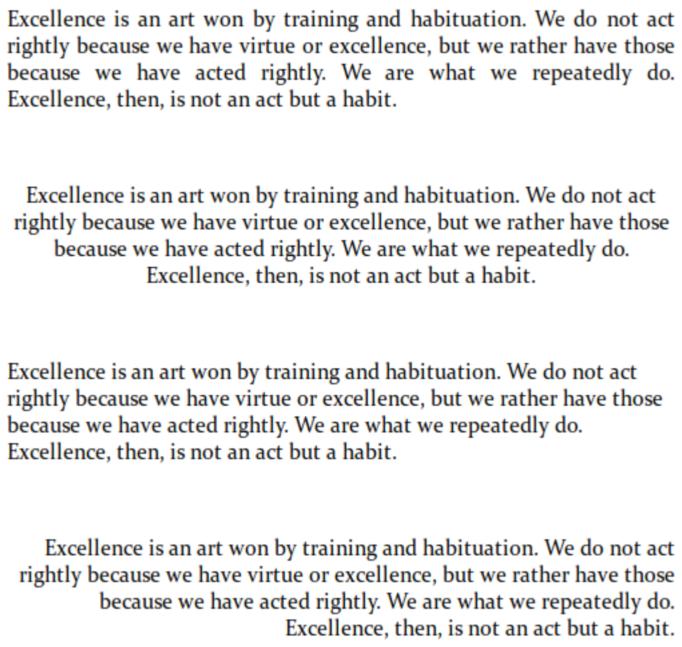
For justified text use: `<p style="text-align:justify">Justified text</p>`.

For centered text use: `<p style="text-align:center">Centered text</p>`.

For right-aligned text use: `<p style="text-align:right">Right-aligned text</p>`.

For left-aligned text use: `<p style="text-align:left">Left-aligned text</p>`. The default paragraph alignment for a left-to-right writing systems.

Figure 1.2: Paragraph alignment: justified (top), centered, right, and left-justified.



Excellence is an art won by training and habituation. We do not act rightly because we have virtue or excellence, but we rather have those because we have acted rightly. We are what we repeatedly do. Excellence, then, is not an act but a habit.

Excellence is an art won by training and habituation. We do not act rightly because we have virtue or excellence, but we rather have those because we have acted rightly. We are what we repeatedly do. Excellence, then, is not an act but a habit.

Excellence is an art won by training and habituation. We do not act rightly because we have virtue or excellence, but we rather have those because we have acted rightly. We are what we repeatedly do. Excellence, then, is not an act but a habit.

Excellence is an art won by training and habituation. We do not act rightly because we have virtue or excellence, but we rather have those because we have acted rightly. We are what we repeatedly do. Excellence, then, is not an act but a habit.

Figure 1.2 shows how Firefox renders an HTML document whose body contains the following markup:

```
<p style="text-align:justify">Excellence...</p>
<br/>
<p style="text-align:center">Excellence...</p>
<br/>
<p style="text-align:left">Excellence...</p>
<br/>
<p style="text-align:right">Excellence...</p>
```

Note that the tag `<br/>` should be used to force a line break in the visual output.

Headings are specified using the `<h1>` ... `<h6>` tags. For example, the markup that follows

```
<h1>Heading 1</h1>
<h2>Heading 2</h2>
      ⋮
<h6>Heading 6</h6>
```

is rendered by Firefox as shown in Fig. 1.3.

Heading 1

Heading 2

Heading 3

Heading 4

Heading 5

Heading 6

Figure 1.3: The six levels of headings.

Italics, Bold, ... The `<b>` tag is used to set text in bold face. For example, the markup `<b>JavaScript</b>` will make the browser to render the word JavaScript in boldface. Similarly, the `<i>` tag is used to set text in italic (slanted) face. To emphasize text we use the `<em>` tag. To get italic bold face, we use the `<i><b>` or the `<b><i>` tags and we delimit the text with `</i></b>` and `</b></i>`, respectively. The `<tt>` tag sets text in a monospaced font. We can use

```
<h3 style="font-family: MyFont">some text</h3>
```

in order to have the text of heading rendered using *`MyFont`*. The following is a rendering of some font selection tags:

Italic text **text in bold face** ***italic bold face***
emphasized text `monospaced font`

Βιέννη, Αυστρία

Vienna, Austria

Exercise 1.3.1 Write down the HTML that creates the previous output. For the calligraphic text, use any font you like. Also, in order to enter the Greek text use a character table application.

1.4 Decorations

Simple Decorations To add a background color in a web page, use

```
<body style="background-color:#E6E6FA">
```
or
```
<body style="background-color:blue">
```

There are different ways to specify colors but HTML uses the so-called RGB scheme. In this scheme a color is specified as a mixture of three basic colors: red, green, and blue. Thus `#RRGGBB` is a color specification where `RR`, `GG`, and `BB` is the amount of red, green, and blue, respectively. Each amount of color is specified by a hexadecimal number in the range 00 to FF (255). Adding background image(s) is a bit involved and I will say more about it in a moment. The tag `<img src="smiley.gif" alt="Smiley face" width="42" height="42">` adds an image into a web page. The `<hr>` tag adds a horizontal line. Also, to get colored text one should use

```
<p style="color:red">This is a paragraph.</p>
```

This will render the text in red. To have a heading set in blue using the Verdana typeface, we should use something like the following:

```
<h3 style="font-family: Verdana; color:blue">Ερασμος</h3>
```

Wallpapers To add a wallpaper to a web page, one needs to use CSS, a language that describes the style of an HTML document. In particular, it describes how HTML elements are to be displayed on screen, paper, or in other media. CSS stands for *Cascading Style Sheets*. The code that follows takes care of everything!

```
<style>
   html { height: 100%; }
   body {
          background-image: url("DSC_1260.JPG");
          height: 100%;
          background-position: center;
          background-repeat: no-repeat;
          background-size: cover;
        }
   p    {
          color: blue; font: 30px Verdana;
          text-align: center;
        }
</style>
```

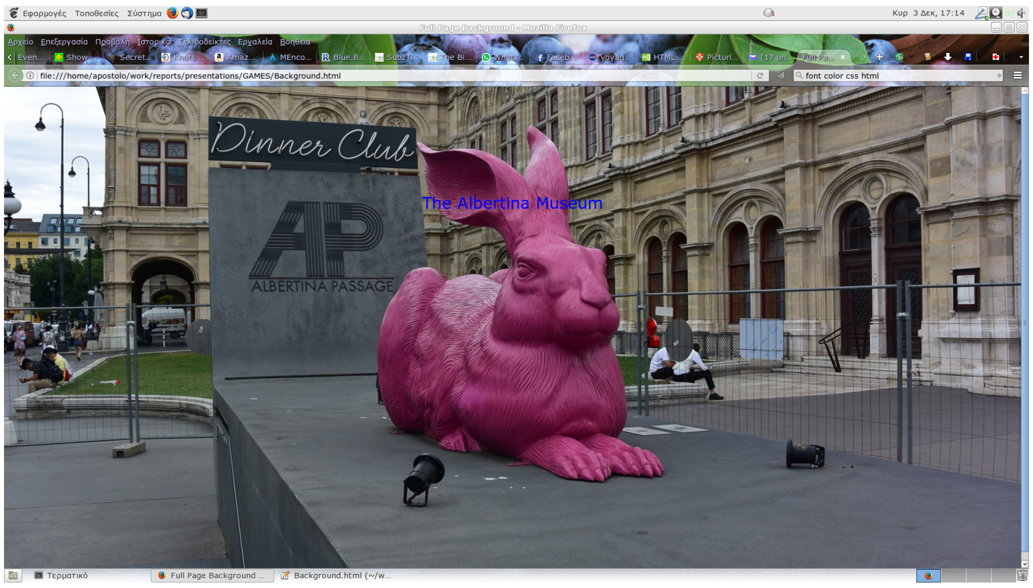


Figure 1.4: A web page with a wallpaper.

specifies that a the image file `DSC_1260.JPG` should be used as wallpaper and it should cover the whole page. Also, the image should be placed in the center of the web page and it should not form some sort of tiling. Figure 1.4 shows visual output that was created using the code above.

1.5 Links

Links allow users to click their way from page to page. Here is how to write down a link in HTML:

```
<a href="url" target="_blank">link text</a>
```

Here by clicking the *`link text`* our browser will transfer to a web page located at *`url`* in a new browser tab. URL is an acronym for Uniform Resource Locator and is a reference (an address) to a resource on the Internet. If we omit the `target` parameter, the new page will be loaded on the current tab. Links can also be images, as you already know.

```
<a href="url"><img src="img.jpg"></a>
```

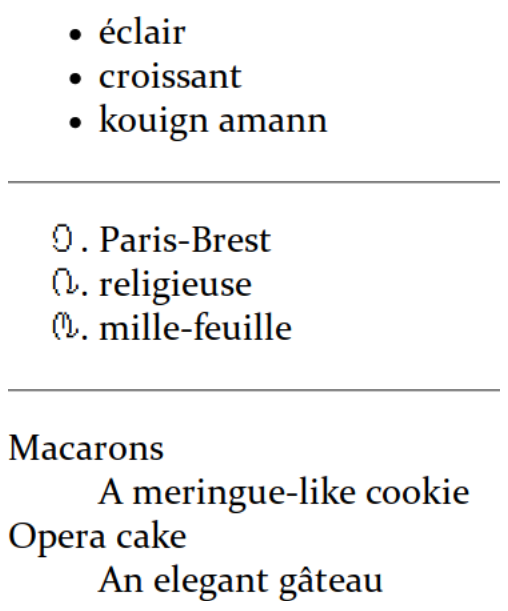
The following markup shows how to create a bookmark:

```
<h2 id="C4">Chapter 4</h2>
```

And here is a link that will transfer to the bookmark we just defined:

```
<a href="#C4">Jump to Chapter 4</a>
```

Exercise 1.5.1 Create a web page with both text and image links and bookmarks.



- éclair
- croissant
- kouign amann

᠑. Paris-Brest
᠒. religieuse
᠓. mille-feuille

Macarons
    A meringue-like cookie
Opera cake
    An elegant gâteau

Figure 1.5: The three supported types of HTML5 lists.

1.6 Lists and Tables

HTML provide authors several mechanisms for specifying lists of information. There are three types of lists: unordered, ordered, and definition or description lists. In unordered lists each item is prepented by a predefined or user-specified symbol:

```
<ul style="list-style-type: disc">
<li> éclair</li>
<li> croissant</li>
<li> kouign amann</li>
</ul>
```

Note that the possible `list-style-type` are `disc` (default), `square`, `circle`, and `none`. Of course one can omit the `style` parameter and let the browser use default values. In an ordered lists the items are prepended by consecutive numbers, letters, symbols, etc. The following is an example of such a list.

```
<ol style="list-style-type: mongolian">
<li> Paris-Brest</li>
<li> religieuse</li>
<li> mille-feuille</li>
</ol>
```

Here the `list-style-type` can assume many different values that include `lower-greek`, `lower-latin`, `katakana`, `hebrew`, `arabic-indic`, `armenian`, etc. Alternatively, one can specify a `type` as follows:

```
<ol type="type">
```

where *type* can be `1` to have items numbered with numbers (default), `A` or `a` to have items numbered with uppercase or lowercase letters, respectively, and `I` or `i` to have items numbered with uppercase or lowercase roman numerals, respectively. Finally, description or definition lists are used to describe or define a series of items. Below is a typical example of such a list.

```
<dl>
  <dt>Macarons</dt>
  <dd>A meringue-like cookie</dd>
  <dt>Opera cake</dt>
  <dd>An elegant gâteau</dd>
</dl>
```

Figure 1.5 shows how these three lists are rendered by Firefox.

Tables are used to display information in tabular form. An HTML table is defined by a `<table>` tag. Each table row is defined by a `<tr>` tag. A table header is defined by a `<th>` tag. By default, table headings are rendered using a boldface font and centered. A table data/cell is defined by a `<td>` tag. The `<caption>` tag defines a table caption. This tag must be inserted immediately after the `<table>` tag. Here is the markup of complete table:

```
<table>
 <caption>Monthly savings</caption>
 <tr>
   <th>Month</th>
   <th>Savings</th>
 </tr>
 <tr>
   <td>January</td>
   <td>$100</td>
 </tr>
</table>
```

1.7 Playing Media Files

The `<video>` tag should be used to play videos, such as a movie clip or other video streams. Currently, there are 3 supported video formats for the `<video>` element: MP4, WebM, and Ogg. Firefox supports all three formats. A simple example follows:

```
  <video width="320" height="240" controls>
     <source src="movie.mp4" type="video/mp4">
     <source src="movie.ogg" type="video/ogg">
     Your browser does not support the video tag.
 </video>
```

The `controls` parameter adds video controls, like play, pause, and volume. To play audio files, such as music or other audio streams, one should use the `<audio>` tag. Currently, there are 3 supported audio formats for the `<audio>` element: MP3, WAV, and Ogg. A simple example is shown below.

```
   <audio controls>\\
      <source src="song.ogg" type="audio/ogg">
      <source src="song.mp3" type="audio/mpeg">
      Your browser does not support the audio tag.
   </audio>
```

1.8 Forms

An HTML form is a means by which a user can interact with a web site. Forms allow users to send data to the web site. The data are processed by a *cgi script* and the result of the operation is shown to the user. For example, when someone makes purchase through a web site and enters the details of her credit card, then the user sends data to some web server. However, it is quite possible to process the data locally. In this second case, one can use JavaScript to process the data. In what follows I will explain how to specify an HTML form and what widgets can be used inside a form.

Typically an HTML forms starts with a `<form>` element like this:

```
<form action="/my-handling-form-page" method="post">
. . . . . . . . . . . . . .
</form>
```

However, here we are interested in very simple forms like the following one:

```
<form name="form's name">
. . . . . . . . . . . . . .
</form>
```

There are many and different widgets that one can use inside a form, but I am going to present a few of them.

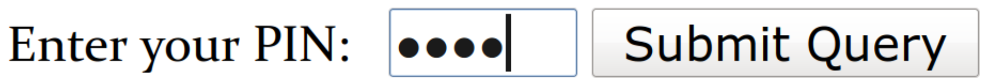


Figure 1.6: Two `<input>` elements: one of type `password` and one of type `submit`.

Single text input field This is a widget that allows a user to enter some text or anything that can fit on a single line. The general form of this widget follows:

```
<input type="text" id="some-id" name="some-name"
    value="I'm a text field">
```

Note that what makes this element a text field is the value of the `type` attribute. And if you think there might be other types, then you have guessed correctly. The attributes `id` and `name` are used to distinguish widgets. It is quite possible to use only one of these two attributes and this holds true for all widgets. The attribute `value` is used to set an initial value for a text field.

E-mail input field A variation of the `<input>` element that is used to get e-mail addresses from users:

```
<input type="email" id="email" name="email" multiple>
```

Password input field This variation of the `<input>` element is used to read passwords and the likes. However, before typing anything into such a field, make sure encryption is on (i.e., the browser is using the HTTPS protocol; not the HTTP protocol). Here is a piece of HTML markup to have the user enter a PIN:

```
<form>
 Enter your PIN:<br>
 <input type="password" id="pin" name="userPIN"
          minlength="4" maxlength="4" required>
 <input type="submit">
</form>
```

Typically, a PIN consists of exactly four digits. This is why I have set `minlength="4"` and `maxlength="4"`. If you embed this markup in a single page, you will seed that the text field is a bit long. To make it shorter we can use the `size="4"` attribute. Also, if the attribute `required` is present, then the user has to enter something. Note that here the `submit` type creates a button that should be pressed to send data to some web server or to process them locally. Figure 1.6 shows how Firefox renders the code above.

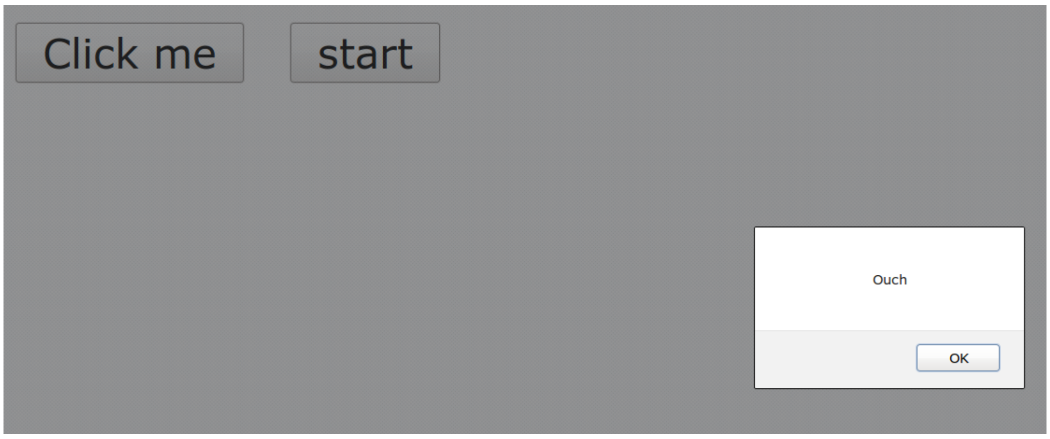


Figure 1.7: Supported HTML5 buttons. The window popped up after the first button was pressed.

Although I tried to make sure the user will enter four digits, still a user can enter four letters to fool our system. In order to solve this problem we can use the `pattern` and the `title` attributes as shown below:

```
pattern="d{4}" title="Only digist are allowed"
```

The value `d{4}` is a regular expression which is a tool to describe the most general form of families of strings. However, I will not present them here and the interested reader should consult [2] or any other book or Internet site that describes them. The `title` attribute specifies what should appear when the user enters at least one character that is not a digit.

Buttons There are two kinds of buttons: `<input>` elements of type `button` and the `<button>` element. Both represent a clickable button. The first kind can appear only in forms while the second can appear anywhere in a document that needs a button. The `submit` button is a special kind of button. The following is an example of the first kind of button:

```
<input type="button" value="Click me" onclick="alert('Ouch')">
```

When the user will click this button, then a window will pop up that will display the text *Ouch*. The markup that follows is a typical example of a button of the second kind:

```
<button id="start" name="FormsOnly">start</button>
```

Naturally, one can customize buttons but this is something that can be done with CSS.

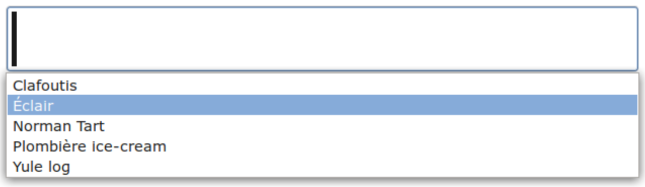


Figure 1.8: Using a `datalist>` element.

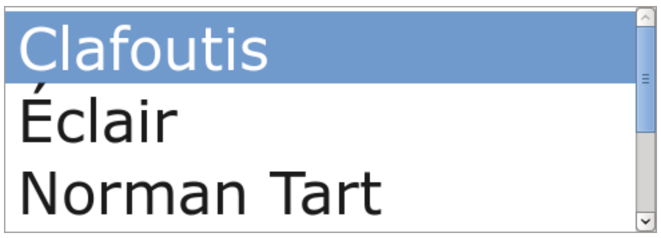


Figure 1.9: A `<select>` element with three visible options.

The `<datalist>` element The `<datalist>` element can be used to specify a list of predefined options for an `<input>` element. The code below shows how to write down this element and Fig. 1.8 shows how Firefox renders this code.

```
   <form>
      Which dessert do you like the most?<br>
      <input list="desserts" name="desserts">
      <datalist id="desserts">
         <option value="Clafoutis">
         <option value="Éclair">
         <option value="Norman Tart">
         <option value="Plombière ice-cream">
         <option value="Yule log">
      </datalist>
 </form>
```

The `<select>` element This element defines a drop-down list and the HTML code of a typical example follows:

```
   <select name="desserts">
      <option value="clafoutis">Clafoutis</option>
      <option value="eclair">Éclair</option>
      <option value="normantart">Norman Tart</option>
```

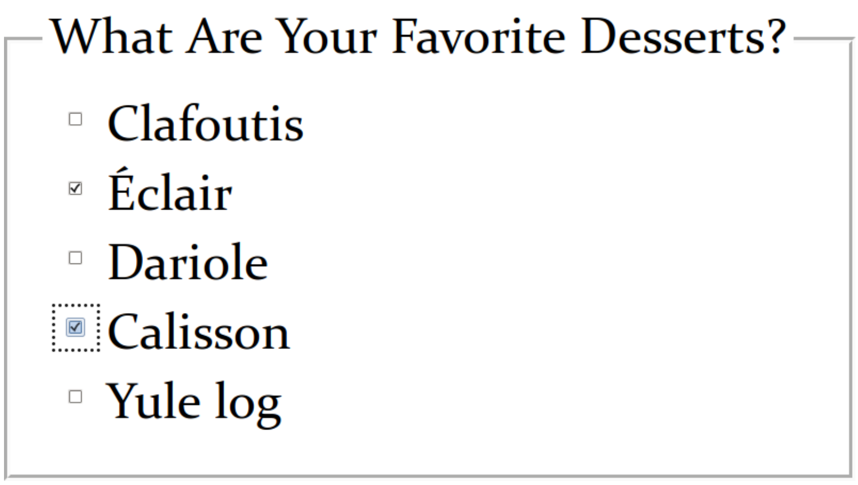


Figure 1.10: A demonstration of checkboxes.

```
      <option value="plombiere">Plombière ice-cream</option>
      <option value="yule">Yule log</option>
   </select>
```

One can have an option *preselected* by adding the attribute `selected` as shown below:

```
<option value="eclair" selected>Éclair</option>
```

Also, by changing the declaration of the list as follows

```
<select name="desserts" size="3">
```

we specify that three list elements will appear instead of one (see Fig. 1.9).

Checkboxes A checkbox is rendered as a square box that can be ticked. They allow one to select one or more values for submission in a form. The following markup creates a form that allows a user to select his/her favorite French desserts.

```
<form>
  <fieldset>
  <legend>What Are Your Favorite Desserts?</legend>
   <input type="checkbox" name="_dessert" value="clafoutis">Clafoutis<br>
   <input type="checkbox" name="_dessert" value="eclair">Éclair<br>
   <input type="checkbox" name="_dessert" value="dariole">Dariole<br>
   <input type="checkbox" name="_dessert" value="calisson">Calisson<br>
   <input type="checkbox" name="_dessert" value="yule">Yule log<br>
  </fieldset>
</form>
```

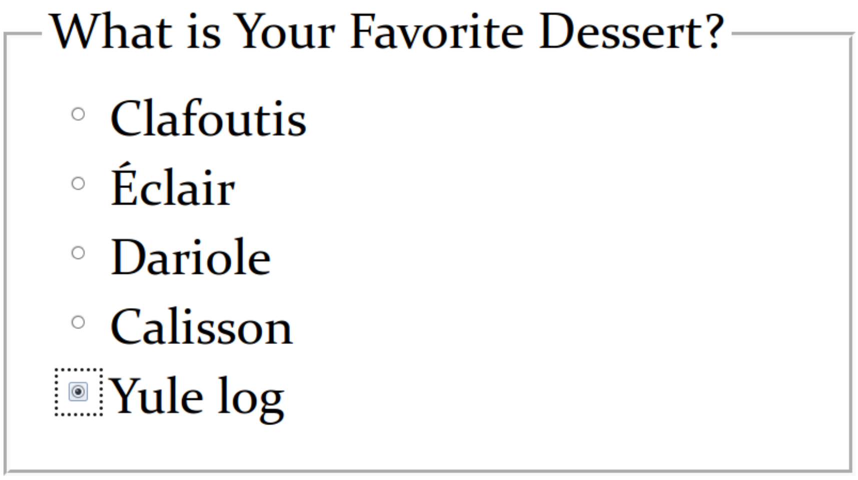


Figure 1.11: A demonstration of radio buttons.

The `<fieldset>` element makes it possible to group elements in a form. When one includes the attribute `checked` in one of the checkboxes, then this one is preselected. The visual output of the markup above is shown in Fig. 1.10.

Radio Buttons Radio buttons can be created with `<input>` elements of type `radio`. They are used in radio groups—collections of radio buttons describing a set of related options. A user can select only one button. Radio buttons are usually rendered as small circles that are filled or highlighted when selected. The following markup creates a form that allows a user to select his/her favorite French dessert.

```
<form>
  <fieldset>
  <legend>What Is Your Favorite Dessert?</legend>
   <input type="radio" name="_dessert" value="clafoutis">Clafoutis<br>
   <input type="radio" name="_dessert" value="eclair">Éclair<br>
   <input type="radio" name="_dessert" value="dariole">Dariole<br>
   <input type="radio" name="_dessert" value="calisson">Calisson<br>
   <input type="radio" name="_dessert" value="yule">Yule log<br>
  </fieldset>
</form>
```

The visual output of the markup above is shown in Fig. 1.11.

Text-areas An `<textarea>` element is rendered as a multi-line plain-text editing control. This should be used when one wants his/her users to enter a multi-line text (e.g., a

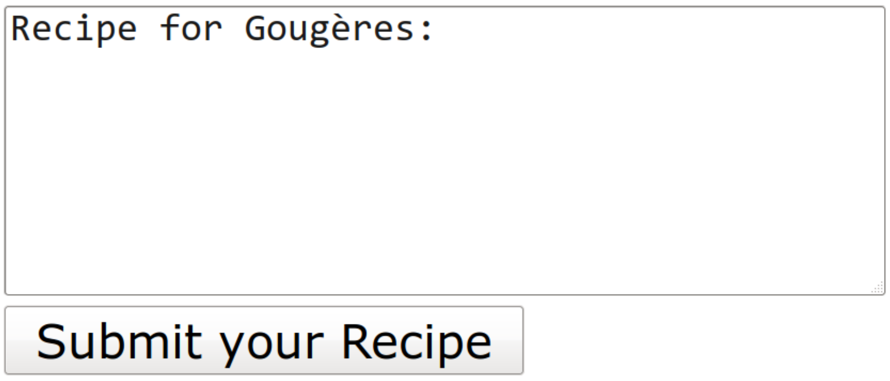

Recipe for Gougères:

Figure 1.12: A form with a text-area.

comment, a review, etc.). The following markup describes a form with a text-area and a button that is supposed to submit the data to a web server.

```
  <form>
    <textarea id="recipe" name="recipe"
              cols="40" rows="6">Recipe for Gougères:
    </textarea><br>
    <input type="submit" value="Submit your Recipe">
</form>
```

The visual output of the markup above is shown in Fig. 1.12.

Chapter 2

What is JavaScript?

JavaScript is an object-oriented programming language that was initially designed and implemented by Netscape. The language is the most widely used language for client-side scripting of web pages. HTML5 and all relatively older versions of HTML support the `<script>` tag that allows the inclusion of JavaScript code into HTML code. JavaScript can be used to make *dynamic* web pages.

The `<script>` tag contains a number of JavaScript commands and its general form follows:

```
<script>
  JavaScript commands…
</script>
```

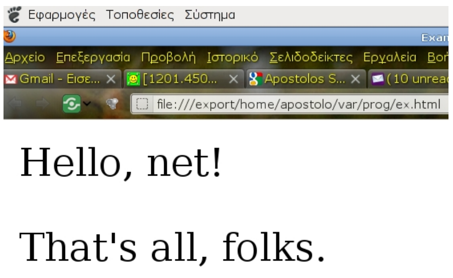
There is no limit to the number of `<script>` tags an HTML file may contain and, naturally, there is no limit to the number of commands each tag may contain.

All modern browsers support client-side scripting but just in case someone uses a browser that does not "understand" JavaScript, then we should use the `<noscript>` tag as follows:

```
<script>
  JavaScript commands…
</script>
<noscript>
  Your browser does not support JavaScript!
</noscript>
```

If the JavaScript code we are using is stored in an external file, then we have to use the `src` attribute of the `<script>` tag:

```
  …
```


Εφαρμογές Τοποθεσίες Σύστημα
Αρχείο Επεξεργασία Προβολή Ιστορικό Σελιδοδείκτες Εργαλεία
Gmail - Εισε...
[1201.450...
Apostolos S...
(10 unrea
file:///export/home/apostolo/var/prog/ex.html

Hello, net!

That's all, folks.

Figure 2.1: How Firefox renders the complete HTML5 file of section 2.

```
<head>
<meta charset="UTF-8">
<title>My Page</title>
<script src="common.js">
</script>
</head>
<body>
  …
```

Note that when using the `src` attribute, nothing should appear between the `<script>` and `</script>` tags. Also, the file that contains the JavaScript code should have the `.js` filename extension. Clearly, the file can be located on the server or to some remote location. In what follows I give the complete contents of an HTML5 file and in figure 2.1 one can see how the file is rendered by Firefox.

```
<!DOCTYPE html>
<html> <head>
<meta charset="UTF-8">
<title>Example</title>
</head>
<body>
<script>
document.write("Hello, net!")
</script>
<noscript>No JavaScript support. Sorry</noscript>
<p> That's all, folks.</p>
</body> </html>
```

The command `document.write("Hello,  net!")` creates content that is added in the body of the HTML page. The argument can be some text, as in this case, or HTML markup. The sister command `document.writeln("`*`content`*`")` writes the *`content`* on a new line so to make the line of the resulting HTML file shorter. To force a new line in the HTML file, one should use a command like the following one:

```
document.write("<br/>")
```

Chapter 3

Values and Variables

For any concrete problem we have to perform calculations and enumerations in order to deliver a solution to it. Thus programming languages provide various *data types* that allow users to represent data. A data type is a collection of all possible values of a certain kind (e.g., numbers, fractions, vectors, etc.) plus various operators that map values of this kind to other values of the same kind (e.g., the rational numbers and the operators $+$, $-$, $\cdot$, and $\div$ form a data type). Usually, a programming language supports a fraction of the values of a certain data type since, for example, not all integer numbers can be represented in the finite memory of a computer.

JavaScript supports the following (primitive) data types:

- Numbers, that is, quantities like `41` or `3.14159`.[1]
- Boolean (logical) values, that is, the values `true` and `false`.
- Sequences of character like `"Hello!"` that are known as *strings*. Note that a string must be enclosed in quotation marks. One can use either single or double quotation marks, for instance, `"John's girlfriend."` or `'Mary came!'`.
- The value `null`, which denotes a *no value*.
- The value `undefined`, which denotes an *undefined* value.

The values `null` and `undefined` are used mainly in *comparisons*. Data types in JavaScript are *dynamic*, which means that, for example, a number can be taken as string or vice versa.

In a sense, a variable is a name that is used to designate a value. For example, in mathematics we write π to designate the number 3.1415926535897.... In fact, a variable is the name of some storage location (i.e., a little part of our computer's memory) where a value

[1]In continental Europe people use a comma to separate the whole part from the fractional part of a number. In the US they use a period. Most programming languages follow the US convention.

is stored. However, variables are like the x in the mathematical definition $f(x) = 3 + x$. In different words, a variable may designate different but not unrelated values during the course of execution of a single program. For example, if we count objects and we use a variable to hold the number of objects counted so far, the variable will hold different values at different moments.

The first character of the name of a variable must be a letter. For names that have more than one letter, the remaining characters can be letters, digits, or the symbol _ (underscore). For example, consider the list of "variable" names that follows:

```
interest   myvar123   1var   _notAvar   A_Valid_Var
```

The 3rd and the 4th words cannot be used as variable names. In addition, the variable names

```
brazuca     Brazuca
```

are different because the language is case-sensitive, that is, the case of letter does matter.

There are three ways to introduce a variable into a program:

- Using an *assignment* statement that is written as `variable = value`. For example, `x=3`.
- Using a declaration, that is, writing the word `var` and then the name of the variable. For example, `var x`. In this case the variable automatically is assigned the value `undefined`.
- Using a combination of the previous two methods, that is, `var x=4;`.

The command `x=3` specifies that the variable `x` will hold the number three. However, we can assign more "involved" things to variables:

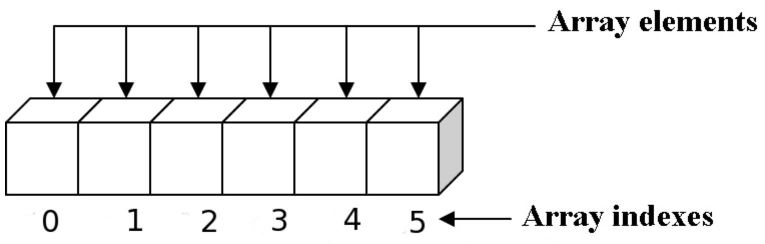
```
x=4; y=6
z=2*x; x=x+1
```

Note that the symbol "`;`" (semicolon) is used to *terminate* a command, nevertheless, it is not necessary to add it when a command is the last command on a line. The first two commands assign to variables `x` and `y` the values `4` and `6`, respectively. In the third command, variable `z` is assigned two times the value of the variable `x`. And in the fourth command, the value of variable `x` is increased by one. Note that the symbol "`=`" does not denote equality.

A comparison yields a Boolean value. For example, the comparison `3>4` yields the value `false` while the comparison `4==5` yields the value `true`. The following table summarizes the available comparison operators and it is assumed that `x = 5` and `var y` were just executed.

Operator	Description	Comparing	Returns
`==`	equal to	`x == 8`	`false`
		`x == 5`	`true`
		`x == "5"`	`true`
		`y == undefined`	`true`
`===`	equal value and equal type	`x === 5`	`true`
		`x === "5"`	`false`
`!=`	not equal	`x != 8`	`true`
		`y != undefined`	`false`
`!==`	not equal value or not equal type	`x !== 5`	`false`
		`x !== "5"`	`true`
		`x !== 8`	`true`
`>`	greater than	`x > 8`	`false`
`<`	less than	`x < 8`	`true`
`>=`	greater than or equal to	`x >= 8`	`false`
`<=`	less than or equal to	`x <= 8`	`true`

An array is a special variable that can hold more than one value at a time. Each value is assigned to an element of an array and one can refer to an array by its index. The picture that follows depicts an array and its indices. Note that indices are consecutive integers starting from zero.



In order to create an array one has to write the name of the array, the symbol "=", the values, which will be stored to the array, separated by commas in square brackets, and an optional semicolon:

```
coffees = ["French Roast", "Columbian", "Kona"];
```

If we want to print the first element of this array, we have to use the following command:

```
document.writeln(coffees[0])
```

If we add an *empty* element in the declaration of an array, then this element has the `undefined` value. For example, in the declaration that follows

```
lang = ["Pascal", "Perl", ,"C", "Java"]
```

the first element, that is, `lang[0]`, has the value `Pascal`, the second element (i.e., `lang[1]`) has the value `Perl`, the third element (i.e., `lang[2]`) has the `undefined` value, etc.

In natural sciences we use the scientific notation to write numbers that are either too big or too small. In this notation we write numbers as 3.16×10^5, where 3.16 is called the *significand* and 5 is just an exponent. It is straightforward to write numbers in the scientific notation in JavaScript. First we write the significand and then either the letter `e` or the letter `E` followed by the exponent. Thus the number 3.16×10^5 should be written as `3.16e5` or as `3.16E5`. In some cases we need to enter numbers in hexadecimal or octal notation. Hexadecimal numbers have the number 16 as their basis and so they have 16 digits: the decimal digits 0–9, plus the letters A, B, C, D, E, and F. In order to write a number in hexadecimal notation, we need to first type `0x` or `0X` and then the number, For example, `0xABCD`. Octal numbers are written with eight digits, that is, the digits 0–7. In order to write an octal number we first write a zero and then digits of the number. For example, `0777` is an octal number.

A string consists of simple ASCII characters, escape sequences, or Unicode characters. An escape sequence corresponds to a special character that cannot be entered conventionally. Typically, an escape sequence is the character that marks the end of line, the escape button, etc. Escape sequences consist of a backslash followed by a single letter (e.g., `\b`). The escape sequences supported by JavaScript are:

Character	Meaning
`\b`	Backspace
`\f`	Form feed
`\n`	New line
`\r`	Carriage return
`\t`	Tab
`\'`	Apostrophe
`\"`	Double quotation marks
`\\`	Backslash

If we want to enter a character that cannot be accessed from our keyboard, we can use the escape sequence `\XXX` where `XXX` is an octal number denoting this character. For example, the symbol © can be accessed with the character `\251`. All Unicode characters that cannot be accessed from our keyboard, can be accessed with an escape sequence of the form `\uXXXX`, where `XXXX` is a hexadecimal number that corresponds to a specific character. For example, the escape sequence `\u00a9` corresponds to the symbol ©.

Chapter 4

Expressions and Operators

An *expression* consists of an number of variables, values, and operators. The operators denote some operation (e.g., addition, multiplication, string concatenation, etc.) and the result of an expression is a value. Assignment commands first evaluate the expression on the right of the = sign and then stores the computed result to the memory location designated by the variable on the left of = sign. For example, the command

```
var x = 7+10
```

stores the number 17 to the storage location designated by variable x.

JavaScript supports a number of different operators:

- Assignment operators
- Comparison operators
- Arithmetic operators
- Bitwise operators
- Boolean operators
- String operators
- Special operators

We have already seen comparison operators but in what follows I am going to present all other categories of operators.

4.1 Arithmetic Operators

An arithmetic operator takes one or two numerical values (either literals or variables) as their operands and yields a single numerical value. The arithmetic operators `+`, `-`, `*`, `/` are used to specify an addition, a subtraction, a multiplication, or a division, respectively. Thus the expression `(3+4-3*19)+4` will evaluate to `-46`. In addition, JavaScript supports the following operators:

Operator	Explanation
`x % y`	Yields the reminder of the integer division of `x` by `y`.
`x++`	Yields the value of `x` and then increases the value of `x` by one.
`x--`	Yields the value of `x` and then decreases the value of `x` by one.
`++x`	Increases the value of `x` by one and yields this new value.
`--x`	Decreases the value of `x` by one and yields this new value.
`-x`	Yields the opposite of `x`.

Thus the values of the the variables defined as follows

```
var y = 25 % 5
var z = 27 % 5
```

are `0` and `2`, respectively. Also, the following example explains the functionality of the operators `++` and `--`:

```
var x=4,y=4
document.write(x-(y++))
//The result is 0, because the operator first yields the value
//and then increases the value of the variable.
document.writeln("<br>")
y=4
document.writeln(x-(++y))
//The number -1 will appear on the web page, because the operator
//increases the value of the variable and then yields this value.
```

4.2 Assignment Operators

An assignment operator stores the value that is on the right of the operator to the storage location designated by the variable on the left of the operator. The basic assignment operators are:

Operator	Is shorthand for
`x += y`	`x = x + y`
`x -= y`	`x = x - y`
`x *= y`	`x = x * y`
`x /= y`	`x = x / y`
`x %= y`	`x = x % y`

4.3 Boolean Operators

A Boolean operator is used with Boolean values or expressions. Recall that the result of a comparison is a Boolean value. The Boolean operators supported by JavaScript are:

Operator	Yields
`x && y`	`true` only if both `x` and `y` evaluate to `true`
`x || y`	`true` if either `x` or `y` evaluates to `true`
`! x`	`true` if `x` evaluates to `false`, `true` otherwise

4.4 String Operators

The concatenation operator `+` joins two strings and yields a new string. For example, the operation `"snow" + "mobile"` yields the string `"snowmobile"`. In addition, the operator `+=` behaves similar to the one that is used to assign numerical values to variables. For example, the following code

```
var x="la"; var y="lo";
x += y;
document.writeln(x);
```

prints the string `"lalo"`.

Chapter 5

Basic JavaScript Commands

In general, a command is an instruction given by a user telling a computer to do something. When writing a program, commands are put together by a programmer to achieve a specific task. JavaScript provides a number of commands that can be used to achieve an impressive number of things. The basic commands of the language are:

- Conditional commands: `if...else` and `switch`.
- Repetition commands: `for`, `while`, `do...while`, a labeled command (used in repetition commands without being such a command), the `break` command and the `continue` command.
- Comments.

Note that there are more commands (e.g., exception handling commands and object manipulation commands) but we cannot cover the whole language in a short introduction. Recall that the semicolon terminates commands and that the symbols "`{`" and "`}`" define a block of commands that are considered as a single command.

5.1 The `if` Command

The command `if` has two forms:

```
if (condition){
   commands
}
```

Form (a)

```
if (condition){
  commands A
}
else{
  commands B
}
```

Form (b)

In the case of the first form of the command, the `condition` is checked and if it evaluates to `true`, then the `commands` are executed. Otherwise, the whole command has not effect. In the case of the second form, `commands A` are executed if `condition` evaluates to `true`. If it evaluates to `false`, then `commands B` are executed. Of course we can omit the curly brackets if we have a single command on either form and/or branch.

Example 5.1.1 The following command prints the string `"okay!"` simply because `3>2` evaluates to `true`.

```
if (3>2)
  document.write("okay!")
else
  document.write("problem!")
```

5.2 The `switch` Command

In a sense, the `switch` command is like a generalized `if` command. The general form of this command is shown below:

```
switch (expression){
  case label1:
    command1;
    break;
  case label2:
    command2;
    break;
  ...
  default: commandN;
}
```

The language processor first finds which label matches with the value of the *`expression`* and then executes the corresponding command. If no label matches the *`expression`*, then `commandN` is executed, provided this branch is specified. If the `default` branch is not specified and no label matches the *`expression`*, then the command has no effect. By omitting the `break` commands we allow the language processor to execute all commands that follow the first command whose label matches the *`expression`*.

Example 5.2.1 If the value of the variable `expr` is `"Cherries"`, the command

```
switch(expr){
   case "Oranges":
      document.writeln("Oranges are $0.59 a kilo<br>");
      break;
   case "Apples":
      document.writeln("Apples are $0.32 a kilo<br>");
      break;
   case "Cherries":
      document.writeln("Cherries are $3.00 a kilo<br>");
      break;
   case "Bananas":
      document.writeln("Bananas are $0.48 a kilo<br>");
      break;
  default:
     document.writeln("Sorry, we are out of " + expr + "<br>");
}
```

will output the markup

```
Cherries are $3.00 a kilo <br>
```

In case there were no `break` commands, then we would see the following in our web browser:

Cherries are $3.00 a kilo
Bananas are $0.48 a kilo
Sorry, we are out of Cherries

5.3 The `for` Command

The `for` command is used when we a priori know the number of repetitions. We implicitly use a counter whose value determines when the command terminates. This command has the following general form:

```
for (initialization; condition; increment-decrement){
  commands
}
```

It is possible to omit either the *`initialization`* part, the `condition`, or the *`increment-decrement`* parts. However, even if we omit something we cannot omit the semicolons. For example, the empty `for` command is written as follows:

```
for(;;){}
```

WARNING! This command never terminates! Let us see exactly what happens when a `for` command is executed.

1. First the language processor executes the *`initialization`* part, provided it exists. This command usually gives an initial value to a counter (i.e., a variable used to count or enumerate things).

2. The `condition` is evaluated. If it is `false`, then the command terminates. If it is `true`, then it proceeds with the *`commands`* are executed.

3. Actual execution of the *`commands`*.

4. The *`increment-decrement`* part is evaluated and execution continues with step 2.

Example 5.3.1 The code that follows computes the sum of the numbers from 1 to 100:

```
var sum = 0;
for(i=1; i<=100; i++){
   sum += i;
}
document.write("<p> The sum 1+2+...+100 is equal to ")
document.write(sum); document.write("</p>")
```

Exercise 5.3.1 Use a for-command to compute the product of the numbers from 1 to 200.

5.4 The `do-while` Command

The `do-while` executes a number of commands and its general form is as follows:

```
do {
  commands
} while (condition)
```

The *`commands`* are executed at least one time and then, the language processor evaluates the *`condition`*. If it is `false`, execution stops. Otherwise, the *`commands`* are executed again, the *`condition`* is re-evaluated, etc. Note that here we cannot omit the curly brackets.

Example 5.4.1 The code that follows prints the integer numbers from one to fine:

```
i=0;
do{
   i++;
   document.writeln(i);
} while (i<=5);
```

If the value of variable `i` was equal to six, then the command would print this number and terminate.

5.5 The `while` Command

The `while` command is a repetition command similar to the `do-while` command and its general form follows:

```
while (condition){
  commands
}
```

Initially, if the *`condition`* evaluates to `false`, then the *`commands`* are not executed and the command aborts execution.

Example 5.5.1 The code that follows prints the numbers from one to five:

```
i=1;
while (i<=5){
  document.write(i); document.write(" ");
  i++;
};
```

Example 5.5.2 The command that follows prints continuously the word Hello:

```
while(true){
  document.write("Hello ");
}
```

5.6 Labels

A label is a mechanism to give a name to a command. This way we can refer to this command later on. For example, we use a label to name a repetition command and then using the `break` and `continue` commands we can specify if the repetition command should stop or continue. Here is how we add a label to a command:

```
Label: command
```

Clearly, the name of a *Label* cannot be the same as any word that has a predefined meaning like the word var.

Example 5.6.1 In this example, we name a while command:

```
Loop:
while(themark == true){
   doSomething();
}
```

5.7 The break Command

The break command is used to prematurely terminate a repetition command or a switch command, as we have already seen. Repetition commands maybe *nested* (i.e., one command may contain another command) and the break makes it possible to terminate prematurely any repetition command. The most general form of this command follows:

break or break *Label*

The second form of the command is used to terminate the command that has this specific *Label* label.

Example 5.7.1 The code that follows scans the array a to find the element that is equal to the variable theValue and exists the repetition command if it finds it.

```
for (i=0; i < a.length; i++){
  if (a[i] == theValue) {
    break;
  }
}
```

5.8 The continue Command

The continue command is used to *restart* a repetition command or a labeled command. In particular,

- When used in a repetition command, the continue command stops the execution a command sequence in the current iteration and continues execution of the repetition command with a new repetition. Unlike the break command, the continue command does not abort execution of a repetition command. In a while command, the language processor is forced to re-evaluate the `condition` while in a `for` command it performs again the `increment-decrement` part.

- If the `continue` command is followed by some label, the command with this label is affected.

The command can be entered as follows:

```
continue   or   continue label
```

Example 5.8.1 In the code that follows there is a `while` command that contains a `continue` command. The former is executed when variable `i` will become equal to three.

```
i = 0
n = 0
while (i < 5) {
  i++
  if (i == 3){
    continue
  }
  document.write(n); document.write(" ")
  n += i
}
```

This code will output the numbers 0, 1, 3, and 7.

Exercise 5.8.1 What is the output of the following commands?

```
var str = "";
loop1: for (var i = 0; i < 5; i++) {
  if (i === 1)
    continue loop1;
  str = str + i;
}
document.write(str);
```

5.9 Comments

The symbols `//` start a comment and everything that follows these symbols until the end of line is ignored by the language processor.

Chapter 6

Predefined Functions

A set is a collection of things (e.g., numbers, objects, people, etc.). A function is a relation between the elements of two or more sets. In particular, a function from a set A to a set B maps every element of A to one and only one element of B. As an example, consider abs that maps any integer number to its absolute value. Thus abs maps 3 to 3 and −3 to 3 or in symbols: $\text{abs}(3) = -3$ and $\text{abs}(-3) = 3$, respectively. It is easy to see that abs is actually a function. If we want to define this function mathematically, we have to write something like $\text{abs}(x) = |x|$. Here the entity x is called a *parameter*, while the number 5 in abs(5) is an argument. In computer jargon we say a function *returns* a value, which is the value that the function maps to its argument. One can say that a function is a method that transforms elements of one set to elements of another set. In this sense, JavaScript provides a number of predefined functions (strictly speaking they are *methods*, that is, functions attached to *objects* but we have said nothing about objects so far). In what follows, I will present a few basic functions.

6.1 Function `toString()`

This is a "function" that is used to get a string representation of an object. If one wants a string representation of a number, then the best thing to do is to create a `Number` object and then to generate a string from it. Here is how this can be done:

```
var x = new Number(19)
document.writeln(x.toString()+"<br/>");
```

Function `toString` get an optional argument which is an integer in the range 2 through 36 specifying the base to use for representing the numeric value. For example, the following code

```
var x = new Number(19)
```

```
document.writeln(19+"<br/>");
document.writeln(x.toString("2")+"<br/>");
document.writeln(x.toString("8")+"<br/>");
document.writeln(x.toString("16")+"<br/>");
```

will print the following:

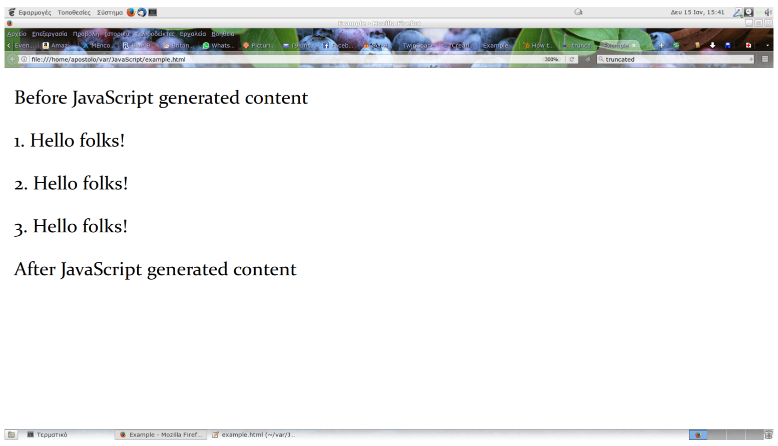
```
     19
     10011
     23
     13
```

6.2 The `write` and `writeln` Functions

We have already demonstrated the use of these functions in a few examples so far. Thus it should be clear that they produce strings. However, the strings can be either text or HTML markup, which will be inserted there where the JavaScript code is executed. The following code demonstrates the use of these functions to generate both content and text.

```
<!DOCTYPE html>
<html> <head>
<meta charset="UTF-8">
<title>Example</title>
</head>
<body>
<p> Before JavaScript generated content</p>
<script>
for (i=0; i<10; i++) {
  document.write("<p>"+ ((i+1).toString()) + ". Hello folks!</p>")
}
</script>
<noscript>No JavaScript support. Sorry</noscript>
<p> After JavaScript generated content</p>
</body> </html>
```

The (truncated) output of this code is shown below:


Before JavaScript generated content

1. Hello folks!

2. Hello folks!

3. Hello folks!

After JavaScript generated content

6.3 Function `eval`

The `eval` function evaluates its string argument, which is supposed to contain JavaScript commands or expressions, and returns the result of the computation. For example, the following code

```
var a = eval("3 * 2") + "<br>";
document.write(a)
```

will print the number six. In general, one should use this function with caution. A "real world" usage example of function `eval` is presented in section 7.2.

6.4 Function `isFinite`

Function `ifFinite` examines its argument and if it is finite number, that is, if its argument is not equal to `Infinity` (i.e., a global property that is a numeric value representing infinity) or it is not `NaN` (i.e., a global property that is a value representing Not-A-Number), then it returns `true`. Otherwise, it returns `false`. In the code that follows, we assign to a variable a very large number through a mechanism that we will explain in section 9.2 and compare it against `Infinity`.

```
var maxNumber = Math.pow(10, 1000); // max positive number

if (maxNumber === Infinity)
  document.write("Let's call it Infinity!");
```

The code will print *`Let's call it Infinity!`*.

6.5 Function `isNaN`

This function examines its argument and if it is not `NaN`, it returns `true`. Otherwise, it returns `false`. For example, the code that follows

```
var x = '100F';
if (isNaN(x))
  document.write('Not a Number!');
else
  document.write(x * 1000);
```

will print *`Not a Number!`* because a string is not a number!

6.6 Functions `parseInt` & `parseFloat`

These functions are used to transform their string input into numbers. Function `parseInt` has a second optional argument which is the base in which the number is written. For example, for binary numbers, the base should be 2. In the example that follows, the browser will show the 420 (the number 120120 in the ternary numeral system is the number 420 in the decimal numeral system).

```
var x = "120120";
var parsed = parseInt(x, 3);
if (isNaN(parsed))
  document.write("Not a Number")
else
  document.write(parsed)
```

Similarly, function `parseFloat` takes as argument a string. Then it examines its argument and if it is a float number (i.e., a number with a fractional part), it returns that number. Otherwise, it returns `NaN`.

6.7 Functions `setInterval` & `setTimeout`

Functions `setInterval` and `setTimeout` are used to execute another function in specific time intervals or once after a specific number of seconds has passed, respectively. These functions have been used to create animations with HTML5 `<canvas>` element (see Chapt. 10) but not anymore as now the suggested method uses a different function. Typically, both functions return a value that has no use and which can assigned to some useless variable. For instance, the command

```
var dummy = setInterval("function()", time)
```

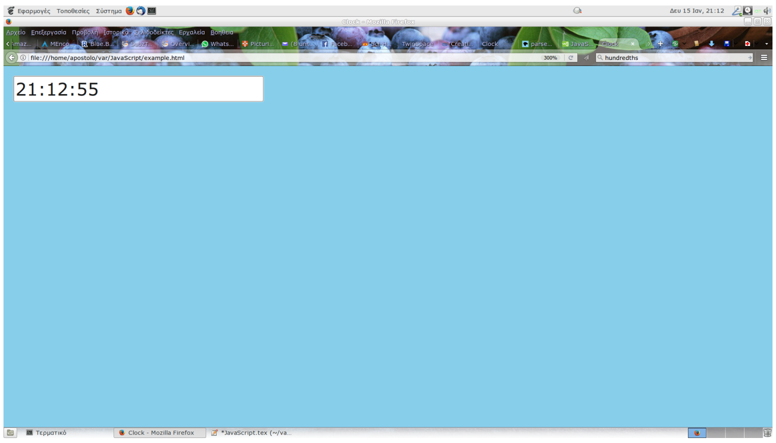
will execute *function* every *time* centiseconds (i.e., *time* hundredths of a second). The command

```
var dummy = setTimeout("function()", time)
```

executes *function* only once after *time* centiseconds. A typical use of function `setInterval` is the creation of a simple digital clock. The complete code is presented below but since it uses some things we have not presented so far, the reader is advised to return here after she has grasped the material in sections 6.8 and 9.1.

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Clock</title>
<script>
function printTime(){
   now = new Date();
   nowHours = now.getHours();
   nowMinutes = now.getMinutes();
   nowSeconds = now.getSeconds();
   return ((nowHours<10)? "0" : "") + nowHours+":"+
          ((nowMinutes<10)? "0" : "") + nowMinutes+":" +
          ((nowSeconds<10)? "0" : "") + nowSeconds;
}
</script>
</head>
<body style="background-color: #87CEEB;">
<form name="clockForm">
<input type="text" name="clockText" value="">
</form>
<script>
function setTime(){
   document.clockForm.clockText.value = printTime();
}
var dummy = setInterval("setTime()",100);
</script>
</body>
</html>
```

The screenshot that follows shows the output generated by the code just presented.


21:12:55

6.8 Defining New Functions

JavaScript, as all modern programming languages, allows its users to define their own functions. In order to define a function we first type the word `function` and then

- the name of the function, which should be no different than a variable name;
- a sequence of parameters separated by comma surrounded by parentheses;
- the body of the function that is used to map the arguments to a value or values. Typically, the body contains many commands and is surrounded by curly brackets.

The following code snippet defines a new function that doubles its argument. More precisely, it returns the square of its argument.

```
function square(number){
  return number*number
}
```

Command `return` stops the execution of the code of the body and forces the function to map its argument(s) to the value that follows this word. Now we can use the function we have just defined:

```
x = square(3)
```

Clearly, this command assigns the number 9 to variable x.

Functions in JavaScript can be *recursive*, that is, they can be defined in terms of themselves. This practically means that the body of the functions includes calls to the functions that is being defined! At first this seems quite strange to say the least: How can a function that has not been defined call itself? Recursive functions are perfectly valid mathematical objects and they are very useful. Every recursive function includes a *termination condition* and a *recursion condition*. For example, the factorial of a positive integer n (denoted $n!$) is defined to be the number $1 \times 2 \times 3 \times \cdots \times (n-1) \times n$ if $n \geq 1$ or 1 if $n = 0$ or more compactly as the recursive function that follows.

$$n! = \begin{cases} 0, & \text{if } n = 0 \\ n \times (n-1)!, & \text{otherwise} \end{cases} \tag{6.1}$$

From this definition it should be clear that the termination condition is $n = 0$. Let us see how we can realize definition 6.1 as a recursive JavaScript function:

```
function factorial(n){
   if ((n == 0) || (n == 1))
      return 1
   else{
      result = (n * factorial(n-1))
      return result
   }
}
```

Now it is possible to compute the factorial of 5 with the following command:

```
document.writeln(factorial(5))
```

Exercise 6.8.1 Try to define the Fibonacci function

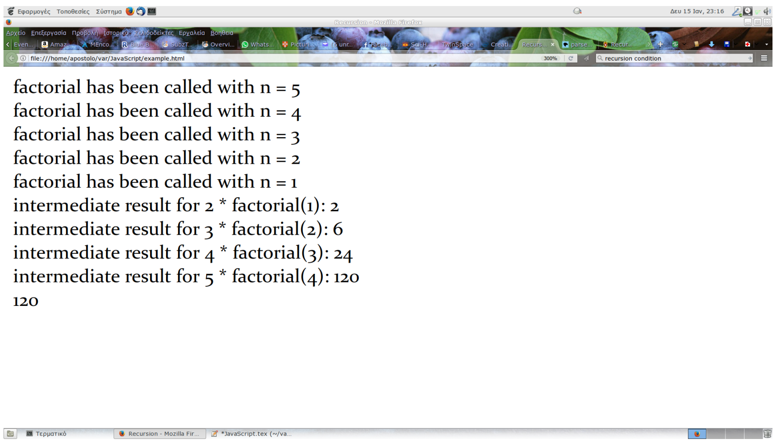
$$\text{Fib}(n) = \begin{cases} 1, & \text{if } n = 0 \text{ or } n = 1, \\ \text{Fib}(n-1) + \text{Fib}(n-2), & \text{otherwise.} \end{cases}$$

as a JavaScript function.

Although I have explained how recursive functions are defined I have not really explained how they are actually computed. Instead of going into the full details, I have augmented the definition above with a few output commands that make things easier to understand:

```
function factorial(n){
    document.write("factorial has been called with n = ")
    document.writeln((n.toString())+"<br>")
    if (n == 1) {
      return 1
    }
    else {
      res = n * factorial(n-1)
      document.write("intermediate result for "+ (n.toString()))
      document.write(" * factorial(" + ((n-1).toString())+ "): ")
      document.write((res.toString())+"<br>")
      return res
    }
}
```

Suppose we want to compute the factorial of 5, then this what this function will print.



```
factorial has been called with n = 5
factorial has been called with n = 4
factorial has been called with n = 3
factorial has been called with n = 2
factorial has been called with n = 1
intermediate result for 2 * factorial(1): 2
intermediate result for 3 * factorial(2): 6
intermediate result for 4 * factorial(3): 24
intermediate result for 5 * factorial(4): 120
120
```

It is worth noting that it is recommended to put function definitions in the `<head>` of a web page. This way we will be absolutely sure they will be visible in the rest of the HTML file.

The arguments we pass to a function call correspond to the parameters that have been used in the definition of a function. In simple words, if we have the following function definition

`function s(x1,x2,x3){…}`

then in the function call `s(1,3,2)`, parameters `x1`, `x2`, and `x3` will be substituted by the numbers `1`, `3`, and `2`, respectively. However, we can use the predefined array `arguments` to process function arguments. As expected, the first argument of a function is stored in position 0 of the array, the second in position 1, etc. Thus in the case of our example, the following expressions are all `true`:

```
arguments[0] == 1
arguments[1] == 3
arguments[2] == 2
```

The total number of arguments is equal to

```
arguments.length
```

This flexibility allows the definition of functions with variable number of arguments. Typically, we specify the *required* parameters and name collectively the optional. In the example that follows all parameters are optional.

```
function mySum(args){
  result = 0
  for (var i = 0; i<arguments.length; i++){
    result += arguments[i]
  }
  return result
}
```

Clearly, this function computes the sum of its arguments. Thus the call

```
document.write(mySum(1,2,3,4,5))
```

will print the number 15.

Exercise 6.8.2 Assume that `Math.pow(a,b)` computes the expression $\mathtt{a}^{\mathtt{b}}$. Define a function that will compute the expression $x_1^n + x_2^n + \cdots + x_k^n$, where n is the first argument and $x_1, \ldots, x_k$ the second argument, the third argument, etc.

6.9 Anonymous Functions

In JavaScript it is possible to define *anonymous* functions, that is, functions with no name. To define such a function just omit the name of the function. A typical example of such a function declaration follows:

```
 var  F = function (a,b){return a+b;};
document.write(F(3,5))
```

Anonymous functions can be used to define methods in new class definitions. However, I do not plan to talk about class definitions.

Chapter 7

JavaScript Event Handling

Most JavaScript applications handle events. These are things that can happen on the screen of a browser. Once an event happens, scripts can be notified and respond accordingly. For instance, if a user clicks a button on a web page, a script might respond to that action by displaying an information box. Table 7.1 shows the various events supported by JavaScript and what triggers them.

7.1 Defining an Event Handler

In order to define an event handler we need something that will trigger an event. As we showed previously there are many things that can trigger events. And these events happen when the user tries to interact with certain HTML components. The most natural way to handle an event is to specify in the tag that specifies a component what will happen if a certain event happens. But let us be more practical.

In order to define an event handler we add its name inside a tag followed by an equal sign (`=`), followed by JavaScript code that will be invoked when the event associated with the event handler happens. The JavaScript code should be enclosed in double quotation marks. If, for some reason, we need to have double quotation marks inside the code, then we simply replace them with single quotation marks.

```
<tag event_handler="JavaScript Code">
```

For instance, if we have defined a function called `compute`, we can use it every time the user presses a button and we can activate it as shown below:

```
<input type="button" value="Calculate"
          onClick="compute(this.form)">
```

Of course we could write a series of commands but obviously it is easier to call a function that will process the event. However, the function must know which component

Event	**Objects Affected By**	**Triggered …**	**Event handler**
Change	Text fields, text areas, and lists	The user just changed its value	`onChange`
Click	All kinds of buttons	The user pressed the right button of the mouse while the cursor is on a button	`onClick`
KeyDown	Documents, images, hyperlinks, and text-areas	The user pressed a key and keeps it pressed	`onKeyDown`
KeyPress	Same as the previous case	The user has just pressed a key	`onKeyPress`
KeyUp	Same as the previous case	The user released a key	`onKeyUp`
MouseDown	Documents, buttons, and hyperlinks	The user pressed the left mouse button	`onMouseDown`
MouseMove	None	The user has moved the mouse	`onMouseMove`
MouseOut	Text areas, hyperlinks, etc.	The user moves the mouse outside an area	`onMouseOut`
MouseOver	Hyperlinks	The user moves the mouse over a hyperlink	`onMouseOver`
MouseUp	Documents and hyperlinks	The user has released the mouse button	`onMouseUp`

Table 7.1: Events supported by JavaScript.

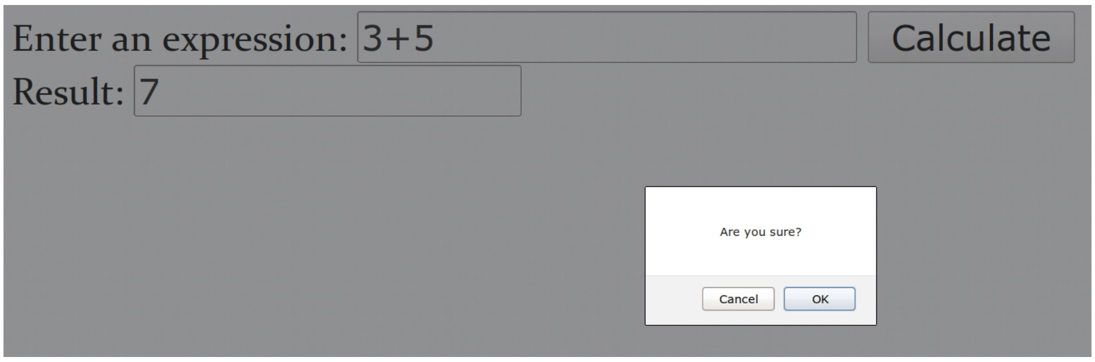


Figure 7.1: Output generated by the form described in section 7.2

triggered the event (better: which component is responsible for this event). And this is exactly the reason why we pass the expression `this.form` as argument to the function. The symbol `this` is the button and the symbol `form` is the name of the form to which the button belongs.

7.2 Using Event Handlers

In the form that is shown in Fig. 7.1 one can enter in the first text field an arithmetic expression (e.g., `4+7*8`), and then, after pressing the button and making the right choice, you can see the result in the second text field. The code that produces the form is shown below:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>On-line calculator</title>
<script>
function compute(f){
   if (confirm("Are you sure?"))
      f.result.value = eval(f.expr.value)
   else
      alert("Please come back again!")
}
</script></head>
<body>
<form>
Enter an expression:
```

```
<input type="text" name="expr">
<input type="button" value="Calculate"
 onClick="compute(this.form)">
<br>
Result:
<input type="text" name="result" size=15>
</form>
</body></html>
```

The function that is defined in the head of the web page takes only on argument that happens to be a form. The function uses the method `confirm` of the `window` class. By pressing the button a dialog box pops up and the user has to chose between `OK` and `Cancel`. If the user presses `OK`, the function `confirm` returns `true` and the expression in the first text field is evaluated and the result is printed in the second text field. Otherwise, a window pops up containing a message that informs the user to come back again (try it!). In fact, the function call

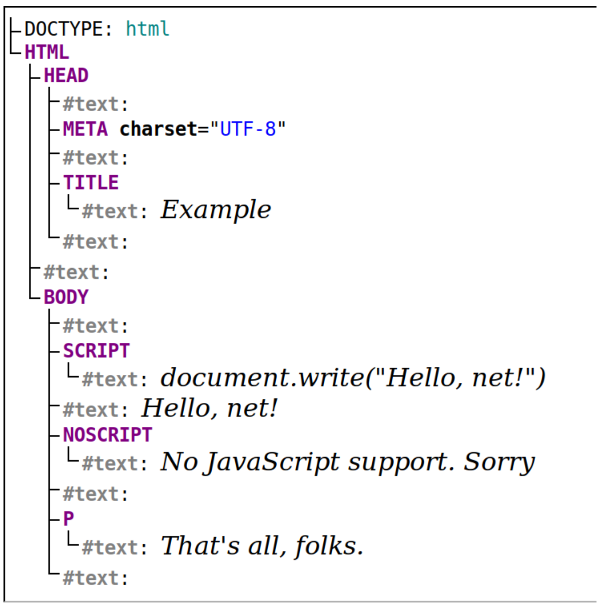
`alert("`*`message`*`");`

creates a (little) window that contains the *`message`*.

Chapter 8

The Document Object Model

When a browser loads a web page, then it creates internally a *Document Object Model* (DOM) of the page. The DOM reflects the structure of the web page and includes all relevant information. In different words, it hierarchically represents the page so that programs can change the document structure, style, and content. In order to understand what HTML hierarchy means consider the complete source file on page 22. Then, the screenshot that follows shows the structure of this file:



```
├DOCTYPE: html
└HTML
  ├HEAD
  │ ├#text:
  │ ├META charset="UTF-8"
  │ ├#text:
  │ ├TITLE
  │ │ └#text: Example
  │ └#text:
  ├#text:
  └BODY
    ├#text:
    ├SCRIPT
    │ └#text: document.write("Hello, net!")
    ├#text: Hello, net!
    ├NOSCRIPT
    │ └#text: No JavaScript support. Sorry
    ├#text:
    ├P
    │ └#text: That's all, folks.
    └#text:
```

All DOM related information is stored to object `document`. This is something predefined that takes its value once a web page has been read by the browser.

When a web page is loaded and `document` has been initialized, then the next logical question is: How do we get access to the various components of a web page? There are five methods but we are going to discuss only the first three methods.

Select Element by ID All HTML elements can have an `id` attribute that has as value some string. However, this value must be unique within a HTML document. For example, here is an element with an `id` attribute:

```
<h1 id="myHeader">Hello World!</h1>
```

In order to get access to this element we can use method `getElementById` and here is how:

```
var a = document.getElementById("myHeader");
```

Select Element by NAME A few HTML elements (e.g., `<img>` elements, forms, and elements of a form) can have a `name` attribute. The value of this attribute is a string and there is no need to be unique in any way. The following shows a form element with a `name` attribute:

```
Name: <input type="text" name="fullname"><br>
```

To refer to this element one should use the `getElementsByName` method:

```
var b = document.getElementsByName("fullname");
```

Select Element by TYPE Sometimes we want to select all elements of a specific type in order to process them collectively. For example, consider the following form:

```
<form id="myform" action="" method="get"
style="text-align:center;">
.......................
<button type="button" id="b1" name="myButton">5</button>
&nbsp;&nbsp;
.......................
<button type="button" id="b5" name="myButton">-5</button>
&nbsp;&nbsp;
</form>
```

Then, if we want to have an array that will hold all buttons, we should use the `getElementsByTagName` method as shown below:

```
var buttons = document.getElementsByTagName('button');
```

Example 8.0.1 Suppose we have an element and we want to change its value after ten seconds. The code that follows shows exactly how this can be done.

```
<!DOCTYPE html>
<html> <head>
<meta charset="UTF-8">
<title>Example</title>
</head>
<body>
<h1 id="xx">Hello User!</h1>
<script>
setTimeout(function(){
            document.getElementById("xx").innerHTML =
                                             "Bye User!";},
           10000);
</script>
</body> </html>
```

Here we see how one can use anonymous functions. In this case there was absolutely no reason to define a function that will be used only once. Also, note that in order to set the new value of the element we use the `innerHTML` property. In newer versions of the DOM specification the `innerHTML` property sets or returns the HTML content (inner HTML) of an element.

Exercise 8.0.1 Modify the script so that prints two times the original text of the header. Hint. To get the value of an HTML element use

```
var x = document.getElementById("xx").innerHTML;
```

Chapter 9

Predefined JavaScript Classes

JavaScript is an object oriented language. Objects are instances of classes and a class is entity with attributes and methods. The attributes are values that characterize each object while the methods allow the modification of attributes and/or access to them. Naturally, we need a mechanism to create new instances of a class. We are not going to discuss how these ideas are implemented in JavaScript but we are going to discuss how to use instances of class `Date` and `Math`.

9.1 Class `Date`

JavaScript predefines the `Date` class that should be used to compute dates, find properties about dates, etc. In addition, class `Date` can be used to perform time related computations. A date is represented by an integer that is equal to the number of milliseconds since January 1, 1970, 00:00:00 UTC (UTC stands for Universal Time Coordinated also known as *"Zulu" Military Time*), with leap seconds ignored. Currently, JavaScript can express any date and time, to millisecond precision, within 100 million days before or after 01/01/1970. This means that the JavaScript clock will not cause us any problem until the year 275,755 A.D.

In order to create a new object one should use any of the following commands:

```
var x = new Date();
var x = new Date(milliseconds);
var x = new Date(datestring);
var x = new Date(year, month, day, h, min, sec, ms);
```

The first command assigns to `x` a new `Date` object corresponding to the current date and time. The second command computes the date from the number of milliseconds that are passed as argument. A *`datestring`* has the following general form:

YYYY-*MM*-*DD*T*HH*:*mm*:*ss*.*sss*Z

The parts of the date strings are the year, the month (01 for January and 12 for December), the day of the month with values from 01 to 31, the letter `T` separates the date from the time, the number of complete hours that have passed since midnight as two decimal digits from 00 to 24, the number of complete minutes since the start of the hour as two decimal digits from 00 to 59, the number of complete seconds since the start of the minute as two decimal digits from 00 to 59, a dot, the number of complete milliseconds since the start of the second as three decimal digits, and `Z` is the time zone offset specified as `Z` (for UTC) or either + or - followed by a time expression `HH:mm`. For example, the current time when these pages were written was:

```
2018-01-18T21:48:43+2
```

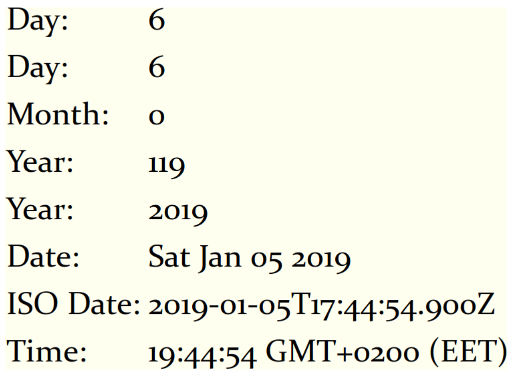
In the fourth case, one should be careful and have on mind that the *`month`* is 0-based which means that January is assumed to be the 0th month and December the 11th. Obviously, we can choose not to specify some of the arguments.

There are four kinds of methods that can be used to process dates.

- Methods like `setHours`, `setMonth`, etc., that should be used to alter the hours, month, etc., of a date.
- Methods like `getHours`, `getMonth`, etc., that should be used to get the hours, the month, etc., of a date as integers.
- Methods like`toDateString`, `toTimeString` that return strings representing a `Date` structure, the date part of a `Date`, and the time part of a `Date`, respectively.
- The methods `now()`, `parse()`, and `UTC()` that return the current time in milliseconds since 01/01/1970, parse a string representing a date and return its representation in milliseconds, and return the millisecond representation of the specified UTC date and time, respectively.

Example 9.1.1 The code that follows

```
var x = new Date();
document.writeln("<table>")
document.writeln("<tr><td>Day:</td><td>"+x.getDay()+"</td></tr>")
document.writeln("<tr><td>Day:</td><td>"+x.getDay()+"</td></tr>")
document.writeln("<tr><td>Month:</td><td>"+x.getMonth()+"</td></tr>")
document.writeln("<tr><td>Year:</td><td>"+x.getYear()+"</td></tr>")
document.writeln("<tr><td>Year:</td><td>"+x.getFullYear()+"</td></tr>")
```



```
Day:      6
Day:      6
Month:    0
Year:     119
Year:     2019
Date:     Sat Jan 05 2019
ISO Date: 2019-01-05T17:44:54.900Z
Time:     19:44:54 GMT+0200 (EET)
```

Figure 9.1: Output generated by the code presented in example 9.1.1.

```
document.writeln("<tr><td>Date:</td><td>"+x.toDateString()+"</td></tr>")
document.write("<tr><td>ISO Date:</td><td>")
document.writeln(x.toISOString()+"</td></tr>")
document.writeln("<tr><td>Time:</td><td>"+x.toTimeString()+"</td></tr>")
document.writeln("</table>")
```

will create a table that is shown in Fig. 9.1.

9.2 The `Math` Class

The predefined class `Math` provides methods and attributes that are useful in mathematical calculations. For example, the attribute `PI` is the constant $\pi = 3{,}1415\ldots$. So if one wants to compute the area of a disk, she might use the expression `Math.PI*r*r`. Similarly, one can use the mathematical functions presented below. For example, the expression

```
document.write(Math.sin(1.5707963267949))
```

will print the sine of 90°. Note that all angles are expressed radians.[1] Also, one does not need to create an object of `Math` to use the functions and attributes described here. The following table describes some of the functions that the class `Math` defines.

[1] If μ is the measure of an angle in degrees, then the measure in radians is: `(Math.PI/180)*μ`.

Method	Description
`abs`	Absolute value
`sin`, `cos`, and `tan`	Sine, cosine, and tangent
`acos`, `asin`, and `atan`	Inverse trigonometric functions
`exp` and `log`	Raise to the power of *e* and natural logarithm
`ceil`	Returns the minimum integer that is greater than or equal to its argument
`floor`	Returns the minimum integer that is less than or equal to its argument
`min` and `max`	Returns the minimum or maximum of its arguments, respectively
`pow`	Raises the first argument to the a power equal to the second argument
`round`	Rounds its arguments to the closest integer
`sqrt`	Returns the square root of its argument

Chapter 10

The HTML5 <canvas> Element

The `<canvas>` element together with the audio and video elements are the things that make HTML5 shine! This element has many applications: interactive movies, games, diagrams, charts, etc. One can create whole sites with the `<canvas>` element but this is something most people do not recommend. There is no need to use Flash and Action Script anymore! The `<canvas>` element is open source technology and it is natively supported by all major browsers. In what follows I will present how canvas can be used for 2D graphics. For 3D graphics we need WebGL, which will not be covered here.

10.1 A Simple Example

The `<canvas>` element provides a surface where one can draw, paint, etc., using the JavaScript programming language. Therefore, playing with the `<canvas>` element is very useful for someone who wants to master this language. Let us start with a simple example that draws a triangle on the computer screen (see Fig. 10.1).

```
<!DOCTYPE html>
<html>
  <head>
  <meta charset="UTF-8">
  <title>Canvas Example</title>
  </head>
  <body>
  <canvas id="canvas1" width="200" height="200">
   Your browser doesn't support "canvas".
  </canvas>
  <script>
  var c = document.getElementById('canvas1'); //  1
```

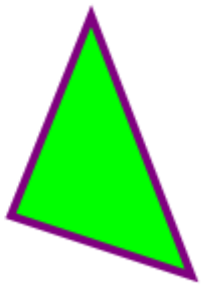


Figure 10.1: A triangle drawn with HTML5's `<canvas>` element.

```
  var ctx = c.getContext('2d');               //  2
  ctx.beginPath();                            //  3
  ctx.moveTo(100,10);                         //  4
  ctx.lineTo(150,140);                        //  5
  ctx.lineTo(60,110);                         //  6
  ctx.closePath();                            //  7
  ctx.fillStyle = 'lime';                     //  8
  ctx.strokeStyle = 'purple';                 //  9
  ctx.lineWidth = 4;                          // 10
  ctx.fill();                                 // 11
  ctx.stroke();                               // 12
  </script>
  </body>
</html>
```

Here we set up a canvas, that is a drawing area, whose width and height is 200 pixels. What goes between `<canvas>` and `</canvas>` appears on your browsers tab if the canvas element is not supported. The canvas gets a unique identification (i.e., `canvas1`) so to refer to it from the JavaScript code. Using the DOM, JavaScript gets access to the canvas in line 1 of the code. Thus variable `c` is an internal representation of the canvas. The call `c.getContext('2d')` returns a drawing context. This simply means we can use variable `ctx` to draw on the canvas. In general, the inclusion of these two lines is necessary in order to do anything useful on a canvas. When drawing on a canvas we specify shapes by paths that consists of line segments. When specifying these line segments we only write down their ebd points, since the beginning is always the end of the previous segment. Each point is determined by its coordinates. The canvas elements uses a peculiar coordinate system where the point $(0, 0) = (x_{\min}, y_{\min})$ is located on the upper left corner of a canvas. The y coordinate grows while going downwards wheareas the x coordinate grows while going rightwards. Thus the point $(x_{\max}, y_{\max})$ is located at lower-right corner of the canvas (see Fig .10.2). It should be clear that there are no negative coordinates.

The call `ctx.beginPath()` is used to specify the start of the *path* that will be used to

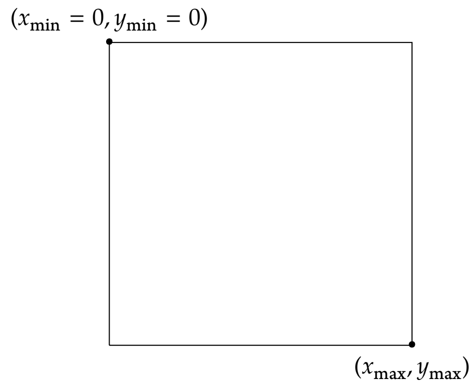


Figure 10.2: The coordinate system of the canvas element.

draw the triangle. The call `ctx.moveTo(x,y)` makes point (x, y) part of the path, nevertheless, nothing will be drawn between the previous point and the new point. The command `ctx.lineTo(w,z)` draws a line segment from the previously specified point or the end of previously specified line segment to (w, z). The call `ctx.closePath()` draws a line segment from the current point back to the starting point of the path. The attibutes `fillStyle`, `strokeStyle`, and `lineWidth` are use to set the color of the area surrounded by a path, the color of the path itself, and the width of the lines that make up the path. Colors can by entered by name, as in this case, or by specifying the amount of red, green, and blue as in the following example:

```
    ctx.fillStyle = "#FF0000";
```

The commands `ctx.fill()` and `ctx.stroke()` are used to fill the area surrounded by a path with the color specified and to draw the path with line segments of specified width and color, respectively. Of course, the commands presented so far are not the only drawing commands. In what follows I will briefly describe the rest of these commands.

There are three methods related to rectangles:

`fillRect(`*`x, y, width, height`*`)` Fills a rectangular area with the default color (black) or a color that is specified with `fillStyle`.

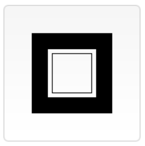
`strokeRect(`*`x, y, width, height`*`)` Draws a rectangle with the default color (black) or the color specified with `strokeStyle` and width specified with `lineWidth`.

`clearRect(`*`x, y, width, height`*`)` Clears the specified rectangular area, making it fully transparent.

`rect(`*`x, y, width, height`*`)` Adds a rectangular path to a currently open path. The top-left corner of the rectangular path is at (x, y) and has *`width`* and *`height`*.

As an example consider the commands on the left that draw the shape on the right:

```
ctx.fillRect(25, 25, 100, 100);
ctx.clearRect(45, 45, 60, 60);
ctx.strokeRect(50, 50, 50, 50);
```



To draw arcs or circles, we use the `arc` or `arcTo` methods.

`arc(`*`x, y, r, ϑ, φ, anticlockwise`*`)` Draws an arc which is centered at (x, y) with radius equal to r starting at ϑ and ending at φ going in the given direction indicated by the Boolean value *`anticlockwise`* (defaulting to clockwise).

`arcTo(`*`x1, y1, x2, y2, radius`*`)` Draws an arc with the given control points and radius, connected to the previous point by a straight line.

Bézier curves are used in the design of computer fonts and animation. Typically, a Bézier curve consists of a starting point $(\mathbf{P}_1)$, an ending point $(\mathbf{P}_n)$, and one, two, three, etc., control points $(\mathbf{P}_2, \mathbf{P}_3, \dots)$. A quadratic Bézier curve, that is a curve that has one control point, is described by the following formula:

$$\mathbf{B}(t) = (1-t)^2\mathbf{P}_1 + 2(1-t)t\mathbf{P}_2 + t^2\mathbf{P}_3, \ t \in [0,1].$$

A qubic Bézier curve, that is a curve that has two control points, is described by the following formula:

$$\mathbf{B}(t) = (1-t)^3\mathbf{P}_1 + 3(1-t)^2t\mathbf{P}_2 + 3(1-t)t^2\mathbf{P}_3 + t^3\mathbf{P}_4, \ t \in [0,1].$$

In order to draw a quadratic Bézier curve on a canvas on should use the method

```
quadraticCurveTo(cp1x, cp1y, x, y)
```

Here (*`cp1x, cp1y`*) are the coordinates of the control point while the starting point is the point where the drawing pen is located and (x, y) is the ending point. Cubic Bézier curves can be drawn with the following method:

```
bezierCurveTo(cp1x, cp1y, cp2x, cp2y, x, y)
```

Here (*`cp1x, cp1y`*) and (*`cp2x, cp2y`*) are the coordinates of the first and the second control point, respectively, while (x, y) are coordinates of the ending point and the starting point is the point where the drawing pen is located before the execution of this method.

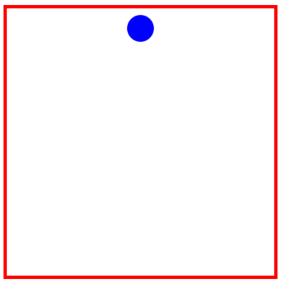


Figure 10.3: A bouncing ball animation. The red square encloses the canvas area.

10.2 A Simple Animation

A canvas can be also used to create animations and, more generally, computer games. Since creating computer games is quite involved, we will focus on the creation of simple animations. The examples will demonstrate the capabilities the canvas offers to anyone wishing to create simple or complex animations. Basically, animation is impelmented by a call to function

```
requestAnimationFrame(animationFunction);
```

where *`animationFunction`* is a user defined function that described what happens at each animation step (i.e., what goes in each frame of the animation) and includes this function call at the end of the function definition. I will start with the code that produces a bouncing ball, that is, a ball that moves back and forth on a straight line inside a frame (see Fig. 10.3).

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Bouncing Ball</title>
<style>
#canvas { border: 5px solid red}
</style>
</head>
<body>
<canvas id="canvas" width="400" height="400"</canvas>
<script>
var canvas = document.getElementById('canvas');
var ctx = canvas.getContext('2d');
var x = 200, y = 20, value = 5, sign = 1;
```

```
requestAnimationFrame(animationFunction);

function animationFunction() {
  ctx.clearRect(0, 0, canvas.width, canvas.height);
  ctx.beginPath();
  ctx.arc(x,y,20,0,2*Math.PI);
  ctx.fillStyle = "blue";
  ctx.fill();

  y += value*sign;
  if (y == 380 )
    sign = -1
  else if ( y == 20 )
    sign = 1
  requestAnimationFrame(animationFunction);
}
</script>
</body>
</html>
```

The canvas is enclosed in a red square because of the following style declaration:

```
#canvas { border: 5px solid red}
```

In order to realize this animation, I have implemented a very simple "algorithm": First draw a circle whose center is at (200, 20) and whose radius is 20, then wait for a bit, clear the drawing area and redraw an identical circle at (200, 25), etc. When the y-coordinate becomes equal to 380, this means that the circle touched the lower edge of the canvas and it is time to go up. When y-coordinate becomes equal to 20, this means that the circle touched the upper edge of the canvas and it is time to go down. I have used the variable `sign` to control the direction to which the ball should move. As was noted above, animation starts when JavaScript starts the execution of the following function call:

```
requestAnimationFrame(animationFunction);
```

First of all, function `animationFunction` clears the drawing area. Then, it draws a circle centered at (`x`, `y`). Next, it changes the value of variable y by adding to it the result of the multiplication `value*sign`. After this, it checks whether there is a reason to change the value of variable `sign` from 1 to −1 or vice versa. In the end of the code there is the following recursive call

```
requestAnimationFrame(animationFunction);
```

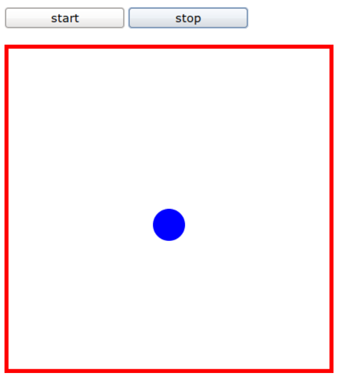


Figure 10.4: An interactive bouncing ball animation. The buttons control the animation.

This is necessary to let the animation process repeat itself. Naturally, this animation never stops. However, it would be interesting to be able to stop animation.

Exercise 10.2.1 Modify the code above so that the ball moves horizontally and starts from $(20, 200)$.

10.3 Simple Interactive Animation

In order to make the animation code of the previous section more interesting, we can add some kind of interaction. The easiest thing to do is to add two buttons that will start and stop the animation. Figure 10.4 shows the new version of the animation web page.

Since the main difference between the two web pages lies in the JavaScript code, I will present only the new parts of the code.

```
<button id="start">start</button>
<button id="stop">stop</button>
. . . . . . . . . . . . .
<script>
. . . . . . . . . . . . .
var requestId;
//
function start() {
  if (!requestId)
    requestId=window.requestAnimationFrame(animateFunction);
}
//
function stop() {
  if (requestId) {
    window.cancelAnimationFrame(requestId);
    requestId = undefined;
  }
}
//
function animateFunction() {
  requestId = undefined;
  . . . . . . . . . . . .
  start();
}
//
document.querySelector("#start").addEventListener('click',
function() { start(); });
//
document.querySelector("#stop").addEventListener('click',
function() { stop(); });
</script></body></html>
```

Variable `requestId` is used to internally control the animation. When its value is `undefined`, then animation commences; otherwise, it stops. Functions `start` and `stop` are used to start and stop the animation, respectively. More specifically, the command

```
requestId=window.requestAnimationFrame(animateFunction)
```

assigns to variable `requestId` a value that is returned by the callback. This value is used to stop the animation with the command

```
window.cancelAnimationFrame(requestId);
```

And since the animation associated with the value of `requestId` has been canceled, this variable is assigned the value `undefined`.

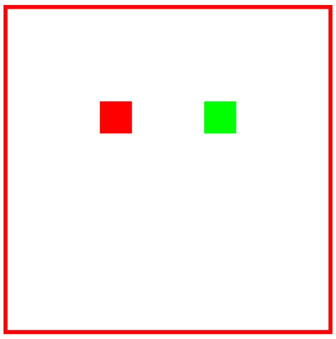
In the simplest case the call `document.querySelector(`*`selector`*`)` returns the first element within the document that matches the *`selector`*, which is a CSS-style element id (i.e., an element id prefixed with the symbol `#`). And this is exactly why we use the `"#start"` and `"#stop"` selectors. Once the element is selected, we need to associate an event-handler with it. In our case, buttons can be *clicked* and so we call method `addEventListener` as follows:

```
addEventListener('click', function() { start(); });
```

There are many *events*: the `abord` event is fired when the loading of a resource has been aborted, the `blur` event is fired when an element has lost focus, the `input` event is fired when the value of an `<input>`, `<select>`, or `<textarea>` element is changed, the `keydown` event is fired when a key is pressed down), the `mousedown` event is fired when a pointing device button is pressed on an element, etc. Note that for each *`element`*`down` event there is an *`element`*`up` event.

10.4 Stopping An Animation

Suppose we want to create an animation that stops automatically. For example, we can have two objects on the top of a canvas and let them move diagonal until they reach the bottom of the canvas. The screenshot that follows shows exactly how this simple idea would look like:



The JavaScript code that follows implements the intended functionality:

```
var x = 10, y = 10, value = 5;

requestAnimationFrame(animateFunction);

function animateFunction() {
  ctx.clearRect(0, 0, canvas.width, canvas.height);
  ctx.fillStyle = "#FF0000";
  ctx.fillRect(x, y, 40, 40);
  ctx.fillStyle = "#00FF00";
  ctx.fillRect(360-x, y, 40, 40);

  x += value
  y += value
  if (y < 360 )
    requestAnimationFrame(animateFunction);
}
```

Note that here the animation repeats only if a certain condition is met. Otherwise, it simply stops. The following is the non-recommended way to stop animations:

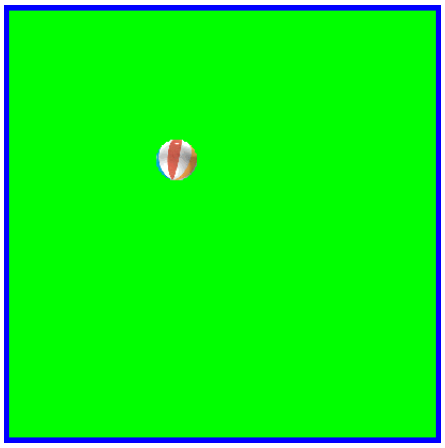


Figure 10.5: Using images in animation.

```
if (y == 360 )
  cancelAnimationFrame();

requestAnimationFrame(animateFunction);
```

A natural questions is this: Assume we want to use an image instead of a drawing, so how can replace the circle with the image of a ball? The first thing that has to be done is to have the image loaded into some variable. For this we create an object of class `Image`, that should appear at the default initial position. Thus the first two lines that follow are standard.

```
var img = new Image();
img.addEventListener('load', function() {
  ctx.drawImage(img, x, y);}, false);
img.src = "./ball.png";
```

The third line of the code is used to actually load the external image into the variable. Note that now we can use the method

```
ctx.drawImage(img, x, y);
```

to draw the image. This places the upper-left corner of the image at (x, y). Figure 10.5 shows th result of replacing a drawing with an image.

10.5 Moving Objects

The first step in the construction of a simple game is the ability to move objects using certain keys (e.g., the arrow keys). For example, a good exercise would be to create an page where the user can move an object (i.e., an image) on a canvas using the arrow keys. Let us start with the definition of some useful variables:

```
var x = 10, y = 10; //initial position
var activeKey = ""; // the key currently pressed
var img = new Image(); // loading an image
. . . . . . . . . . . .
var dx = 0, dy = 0; // horizontal and vertical direction
var speed = 100; // pixels per second
```

All these variables are related with the following "equations"

```
    x +=  dx / 60 * speed;
    y += dy / 60 * speed;
```

that compute the new position of the moving object. Let us see how we can handle the use of the arrow keys.

First we need to specify what should happen when the user presses an arrow key and keeps it down. If and when the key is released, the script must act accordingly. Let us see what should happen in the first case.

```
document.addEventListener('keydown', function(e) {
    if (activeKey == e.key) return;
    activeKey = e.key;

    if (activeKey == "ArrowLeft")
        dx = -1;
    else if (activeKey == "ArrowUp")
        dy = -1;
    else if (activeKey == "ArrowRight")
        dx = 1;
    else if (activeKey == "ArrowDown")
        dy = 1;
});
```

Here we register an event handler for the `keydown` event. The argument to the anonymous function is an event and `e.key` returns the name of the key just pressed. In each case we specify the value of variable that specifies the direction on the horizontal or the vertical axis. In JavaScript we need to specify what should happen when a key is released or else the object will continue moving on the same direction.

```
document.addEventListener('keyup', function(e) {
    switch (e.key) {
        case "ArrowLeft":
        case "ArrowRight":
            dx = 0;
            break;
        case "ArrowUp":
        case "ArrowDown":
            dy = 0;
            break;
    }
    activeKey = "";
});
```

Literally this means that if ones releases the left or right arrow key, then the image should stop moving and this is why variable `dx` becomes equal to zero. Similarly, if the user releases the down or the up arrow key, then variable `dy` should become equal to zero. What is left is to explain how to implement the function that realizes the animation. First we need to clean the canvas:

```
function fun(){
    ctx.fillStyle = "#00FF00";
    ctx.fillRect(0, 0, 400, 400);
```

Next, we need to compute the new position of the object:

```
    var xnew = x + dx / 60 * speed;
    var ynew = y + dy / 60 * speed;
```

It is quite possible that the new position may fall outside the area of the canvas and, naturally, this is something one needs to prevent. First let us see how compute the new position when the left arrow key is pressed:

```
    if (activeKey == 'ArrowLeft') {
      if (xnew >= 0)
        x = xnew;
      else if (xnew  < 0)
        x = 0;
    }
```

If the new position is on the left of the point $(0, y)$, then we should make the new position the point $(0, y)$. Otherwise, the new position is the one just calculated. Obviously, one should handle the use of the up arrow key in a similar way:

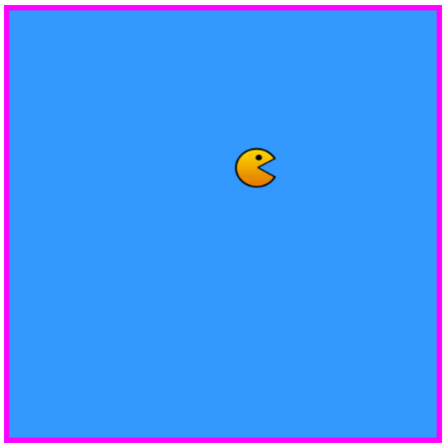


Figure 10.6: A moving Pacman.

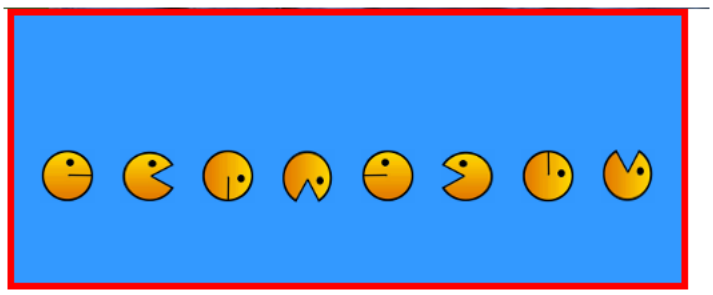
```
    if (activeKey == 'ArrowUp') {
      if (ynew >= 0)
        y = ynew;
      else if (ynew  < 0)
        y = 0;
    }
    . . . . . . . . . . . . . . .
    ctx.drawImage(img, x, y);
    requestAnimationFrame(fun);
}
```

The last thing to do is to draw the object in the new position and move to the next step.

Let us make things a bit more complicated and assume that we want also to animate the moving object. For example, we could replace the ball with something like a pacman (see Fig. 10.6). In this case the first thing we need to do is to load eight images—two for each direction. This can be done with the following code:

```
var images = new Array();
for(i=0; i<8; i++) {
  images[i] = new Image();
  images[i].addEventListener('load',
            function() { ctx.drawImage(img, x, y); },
            false);
  images[i].src = './smallpacman'+i.toString()+'.png';
}
```

Here we use an array to hold the images and for each array element we do exactly what we did when we were loading one image. The images that this code loads are shown below.



Exercise 10.5.1 Create a web page that will show eight icons as in the previous screenshot.

The next step is to write the code that will display on a specific direction (e.g., up) the two "faces" one after the other continuously. This way we animate the moving object. Here is what should happen when we press the right arrow key:

```
if (activeKey == 'ArrowRight') {
  if (xnew <= 360)
    x = xnew;
  else if (xnew  > 360)
    x = 360;
  image_index++;
  if (image_index % 2 == 0)
    image_index = 0;
  else
    image_index = 1;
}
```

Variable `image_index` is used to choose the correct image. First we increase its value by one and then we check whether it is an even or an odd number and we assign it the values zero and one, respectively. This way we make sure only the first images are chosen when the right arrow is pressed. When the user presses the up arrow key, we do something similar as the following code reveals.

```
  image_index++;
  if (image_index % 8 == 0)
    image_index = 6;
  else
    image_index = 7;
```

Exercise 10.5.2 Write the code that will handle the cases where the left and down arrows are pressed. (Hint. Check out the screenshot with the eight faces).

Finally, the selection process should not happen when no key is pressed. This means that one of variables `dx` and `dy` is equal to zero:

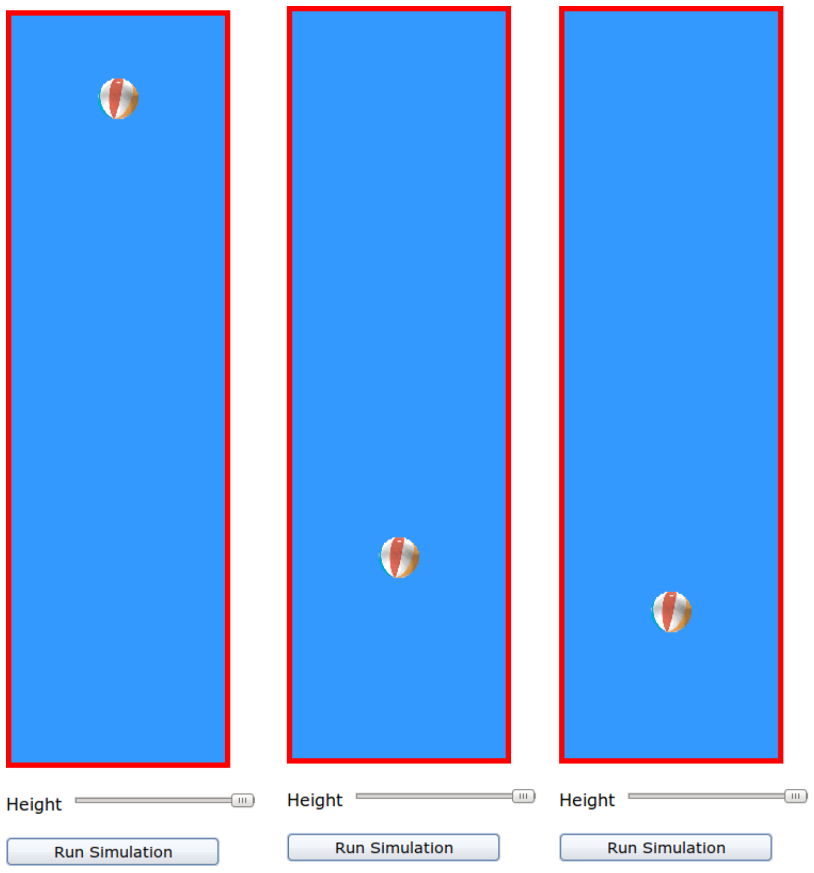


Figure 10.7: A simple simulation application.

```
if (dx !=0 || dy != 0) {
  . . . . . . . . . . . .
}
```

Note that the expression `dx !=0 || dy != 0` is equivalent to the expression

```
!(dx == 0 && dy == 0)
```

10.6 A Simple Simulation

Suppose we want to create a simple simulation application of a free falling ball as shown in Fig. 10.7. Clearly, the first thing we need to do is to design the user interface.

```
<canvas id="mycanvas" width="200" height="700">
</canvas><br/><br/>
<label>Choose Height</label>
<input type="range" min="20" max="94" value="20" id="BallHeight">
```

```
<br/><br/>
<button id="runapp">Run Simulation</button>
```

In this case, I am using an slidebar because it is easier to choose a height with it than writing the desired height. The slidebar is described by the `<input type="range"...>` markup. The attribute `value` is used to set the initial value of the slidebar. The tag `<label>` is used to *label* controls. Let me now present the JavaScript code. The first thing one types are the variables that will be used:

```
var height = 700; //canvas height
var h, a = 0.1, v, ballAbsorption = 0.8;
var Hslider = document.getElementById("BallHeight");
```

There is no reason to repeat how the image of the ball is loaded so let us skip it. Variable `Hslider` will be used to read the value the user has selected:

```
function init() {
   h = 7*Hslider.value;
   v = 0;
}
```

When the user pushes the button, the following code is executed:

```
document.querySelector("#runapp").addEventListener('click',
  function() { init(); requestAnimationFrame(animateFunction); });
```

An finally here is the code that impements the animation:

```
function animateFunction() {
  if (h <= 0 && v > 0) {
    v *= -1 * ballAbsorption; // bounding with less velocity

    if (v > -0.1 && v < 0.1)
      cancelAnimationFrame();
  }
  // Move the ball
  v += a; // accelerating
  h -= v; // falling (if v < 0)
  ctx.fillRect(0, 0, 200, 700);
  ctx.drawImage(img, 80, height - h - 40);
  requestAnimationFrame(animateFunction);
}
```

Exercise 10.6.1 Explain what does function `animateFunction` do?

Acknowledgement

I would like to thank Kyriaki Athanassiadou for proof-reading a draft of this text and thus helping me improve its linguistic quality.

Bibliography

[1] Flanagan, D. *JavaScript: The Definitive Guide*, 6th ed. O'Reilly Media, Sebastopol, CA, USA, 2011.

[2] Friedl, J. *Mastering Regular Expressions.* O'Reilly Media, Sebastopol, CA, USA, 2006.

[3] Fulton, S., and Fulton, J. *HTML5 Canvas*, 2nd ed. O'Reilly Media, Sebastopol, CA, USA, 2013.

[4] Geary, D. *Core HTML5 2D Game Programming.* Prentice Hall, Upper Saddle River, NJ, USA, 2015.

[5] McCabe, P. *Create Computer Games: Design and Build Your Own Game.* John Wiley & Sons, Inc., Hoboken, NJ, USA, 2018.

[6] Sarris, S. *HTML5 Unleashed.* SAMS, Indianapolis, IN, USA, 2014.