

Introduction to HTML5



Contents

Module 1: Overview of HTML5

Lesson 1: Introducing HTML5	1-3
Lesson 2: Creating an HTML5 Document	1-7
Lesson 3: HTML5 Feature Detection	1-17

Module 2: Integrated HTML5 APIs

Lesson 1: Web Forms	2-3
Lesson 2: Playing Audio and Video	2-12
Lesson 3: The Canvas Element	2-19

Module 3: HTML5 Associated APIs

Lesson 1: Geolocation API	3-3
Lesson 2: Web Storage API	3-9
Lesson 3: Web Sockets and Web Workers	3-14

Module 1

Overview of HTML5

Contents:

Lesson 1: Introducing HTML5	1-3
Lesson 2: Creating an HTML5 Document	1-7
Lesson 3: HTML5 Feature Detection	1-17

Module Overview

- Introducing HTML5
- Creating an HTML5 Document
- HTML Feature Detection

In this module the most important characteristics and ingredients of the next major version of the HyperText Markup Language (HTML) will be covered. Its history, body of thought, industry participants, features and possibilities are discussed. The new semantic elements of HTML5 and support detection of desired features in certain user agents is part of this module, too.

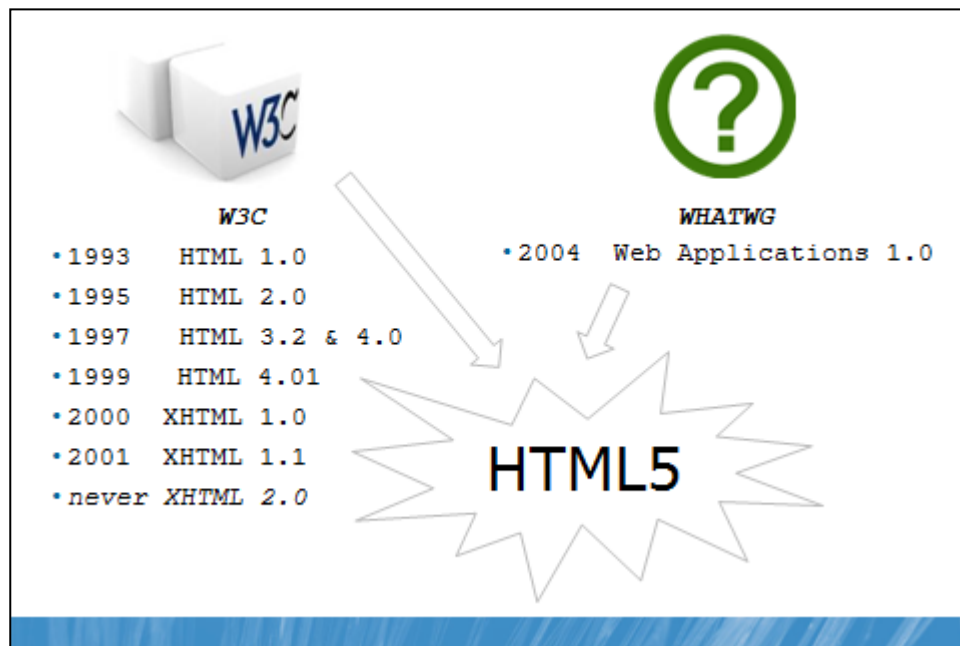
Lesson 1

Introducing HTML5

- Short History of HTML
- What is HTML5

This lesson starts with a discussion on how the HTML language has evolved to its next version. Furthermore attention is given to what HTML5 actually is and a listing of some of its features.

Short History of HTML



Key Points

HTML goes back a long time. It was first published as an Internet draft in 1993. The 1990s saw versions 2.0, 3.2, 4.0 and finally version 4.01 in 1999. The World Wide Web Consortium (W3C) assumed control of the specification.

Since 1998, the W3C believed that the future versions should follow the XML syntax, to make HTML more extensible and increase interoperability with other data formats. This idea would lead to the XHTML 1.0 specification in 2000. This XML version of HTML required for example the strict XML syntax rules like quoting attributes, closing some tags while self-closing others.

Unforgiving Rules

Starting with the 1.1 version of XHTML the W3C introduced the concept of having a page fail as a whole as soon as a syntactical error would appear. The 2.0 version of the standard was meant to continue this draconian error handling.

Do Not break The Web

In 2004, a group of people were very concerned that this strictness would break the web. They believed that browsers should be forgiving, trying to render any page, no matter which HTML it was written in, syntactically correct or not. They formed the Web Hypertext Application Working Group (WHATWG) and started developing what they believed would be the next version of the markup language, calling it Web Applications 1.0

The next few years the W3C and the WHATWG tried to ignore each other. But by the end of 2006 the W3C, still working on the draft of XHTML 2.0, realized that no major browser would implement it. The founder of the W3C announced that they would be working together with the WHATWG, and together renamed the next version HTML5.

Joined Forces

The W3C stopped working on version 2.0 of XHTML in 2009, enabling its HTML Working Group to focus completely on developing HTML5. The group includes AOL, Apple, Google, IBM, Microsoft, Mozilla, Nokia, Opera and many other vendors. An important fact is that all

browser vendors have representatives in the working groups now. Vendors have the right to veto any aspect of the specification. Because “if they don’t implement it, the spec is nothing but a work of fiction”, according to WHATWG member and editor of the HTML5 specification Ian Hickson. The specification is expected to reach *candidate recommendation* in 2012. This means when that stage is reached, HTML5 will be complete.

What is HTML5



Key Points

HTML5 is the next major revision of the HTML standard, providing new features that are necessary for modern web applications. It also standardizes many features of the web platform that web developers have been using for years.

HTML5 is designed to be cross-platform and backward compatible with existing web browsers. Some new features are built on existing features and allow for providing fallback content for older browsers. The browser's Document Object Model (DOM) is an integrated part of HTML5 now. The world of mobile devices is not ignored either. The mobile web browsers that are pre-installed on iPhones, iPads and Android phones all have very good HTML5 support.

New Features

Many new features are added by HTML5. To improve the semantic richness of a document, elements such as `<section>`, `<article>`, `<header>` and `<nav>` have been introduced.

Other elements include `<video>`, `<audio>` and `<canvas>`, as well as Scalable Vector Graphics (SVG) content integration, which improve the inclusion and handling of multimedia and graphic content on the web. All this without the need to download and install third-party plugins and having to learn their specific syntax and object model.

HTML5 gives better support for local offline storage. Web storage supports persistent data storage, similar to cookies, as well as window-local storage.

New form controls are available like calendar, date, time, email, url and search.

The required process for handling invalid documents has been defined as well, so that syntax errors will be treated in a uniform way by all user agents.

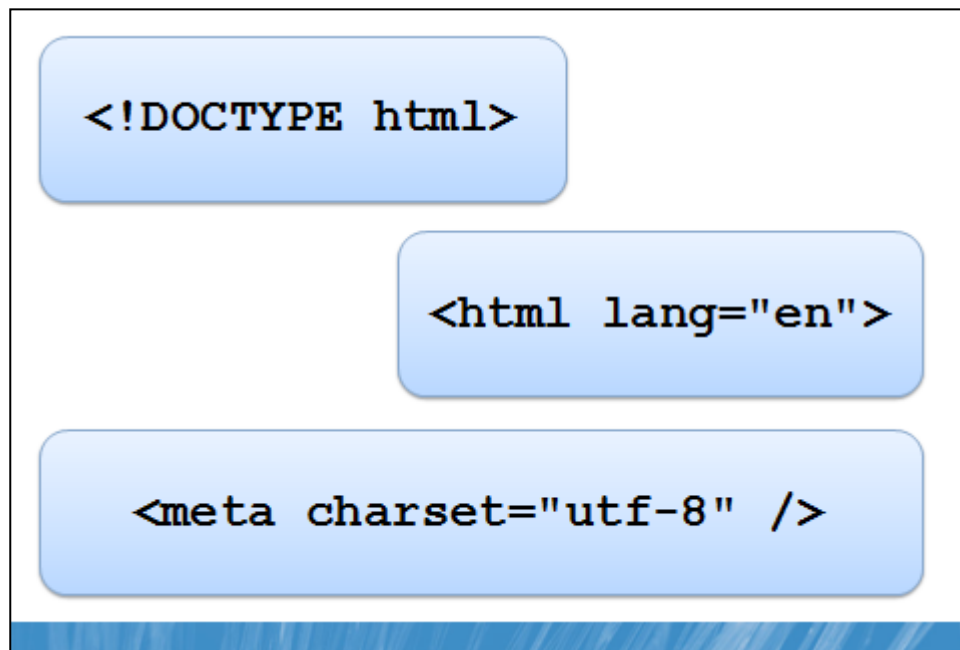
Lesson 2

Creating an HTML5 Document

- Hello HTML5
- Enter the Semantic Elements
- Putting it Together
- Show Some Style

After the brief history lesson and a quick feature teaser, it is time to get to work with the new markup standard. After introducing the new doctype and character setting some semantic elements are discussed. A basic HTML5 document will then be constructed with a belonging stylesheet to layout the page.

Hello HTML5



Key Points

The doctype and the html and meta element will be discussed here. These items make the starting point of any proper HTML5 document.

The doctype needs to be on the first line of your HTML file. This is because certain browsers might handle the page as if no doctype is presented when even a single blank line is above it.

A browser knows at least two modes: *quirks* mode and *standards* mode. Quirks mode is triggered when no doctype declaration is found and is used to render older or non-standard HTML. Standards mode is driven by the use of proper doctypes. The page is then rendered using the parsing rules belonging to the HTML specification identified by the doctype.

The new doctype declaration looks as follows:

```
<!DOCTYPE html>
```

After the doctype, the root element follows. This is our well-known HTML tag. It is advised to use the language attribute, so that screen reader software is able to use the right pronunciation.

```
<html lang="en">
```

The character set declaration is also short, and should be in the first 512 bytes of the document. The most commonly used set of characters is specified as follows:

```
<meta charset="utf-8" />
```

It is important to explicitly set the character set. Failing to do so may result in a security risk as the result of a cross-site scripting (XSS) attack. Further details are in an article entitled “UTF-7: the case of the missing charset” (<http://code.google.com/p/doctype/wiki/ArticleUtf7>).

No XML Language

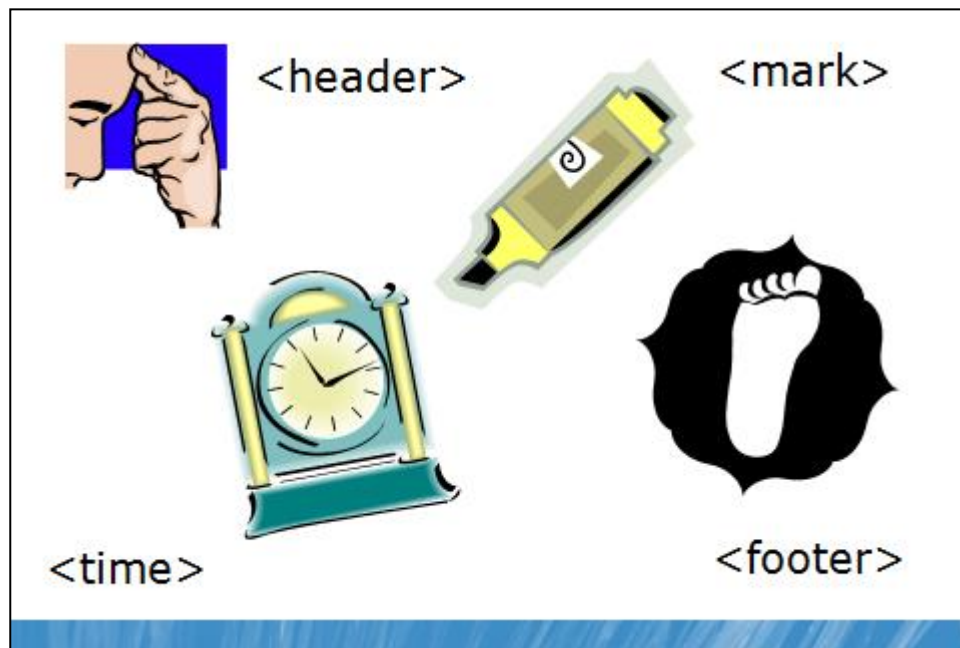
HTML5 is not an XML language. So officially, HTML5 is not case-sensitive, its elements do not need quotes around attributes and empty elements do not need to be closed. Due to the stricter

XHTML rules, the validator used cared a lot about following the XML syntax. But browsers never had any problem with ignoring the rules mentioned before. Because of maintenance purposes it is advised to pick a certain style and keep to it as a team. This course will try to keep the XML rules in mind.

Check Validity

To check if the document is a valid HTML5 document, websites like <http://html5.validator.nu> may be used.

Enter the Semantic Elements



Key Points

Several search engines have analyzed millions of pages in the last couple of years to find that among the most popular *div* element *class* names and *ids* are for example: *header*, *footer* and *nav*. This gives a clear idea of how web page authors try to structure their HTML documents.

This inspired the people working on the HTML5 specification to create more semantic elements, some of which are shown in the following table.

Semantic Element	Description
<article>	Defines external content. This could be a news article from an external provider or a blog or forum text. It should be independent from the rest of the document
<aside>	Defines some content aside from the content it is placed in. The aside content should be related to the surrounding content. It could be placed as a sidebar in an article.
<footer>	Defines the footer of a section or document. Typically contains the name of the author, the date the document was written or contact information.
<header>	Defines a group of introductory or navigational aids. It is intended to contain the section's heading (an <i>h1</i> to <i>h6</i> element or an <i>hgroup</i> element). It can also be used to wrap a section's table of contents, a search form or some logo.
<hgroup>	Defines the heading of a section or a document and is used to group headers <i>h1</i> to <i>h6</i> when the heading has multiple levels, such as subheadings, alternative titles or taglines.
<mark>	Defines a run of text in marked or highlighted characters for reference purposes.
<section>	Defines a generic section of a document and is a thematic grouping of content (typically with a heading), like chapters or the numbered sections of a thesis. A home page could be split into different sections for the introduction, news items and contact information.

Semantic Element	Description
<time>	Defines either a time on a 24-hour clock or a precise date in the proleptic Gregorian calendar, optionally with a time and a timezone offset.
<nav>	Defines a section of a page that contains navigation links to other (parts within) pages. Only sections that consist of major navigation blocks are appropriate for this element.


A complete list of all known, new and no longer supported HTML5 elements can be found on the “HTML Tag Reference” page: http://www.w3schools.com/html5/html5_reference.asp

Putting it Together


```

<header>
  <h1>...</h1>
</header>
<section>
  <article>...</article>
  <article>...</article>
</section>
<nav>...</nav>
<aside>...</aside>
<footer>...</footer>

```



Document Outline



Key Points

A basic HTML5 web page is shown here, using some of the most common new elements. Certain elements and attributes will be discussed along the way.

Basic HTML5 Page

We start with the elements discussed earlier, declaring the doctype, language and character set.

```

<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>A page is not valid HTML5 without a title element</title>
    <link href="first.css" rel="stylesheet" type="text/css" />
  </head>
  <body>
    <div id="container">
      <header>
        <h1>This is the Main Header</h1>
      </header>
      <div id="columnMaker">

        <section>
          <h2>This is the Articles Section Header</h2>

```

Element hgroup

The **hgroup** element is meant to group two or more related header elements, like a subtitle or tagline. By using this element, the HTML5 *document outline* does not create an unnecessary (phantom) node for each alternative title or tagline.

```

<article>
  <header>
    <hgroup>
      <h3>1st Article Header</h3>
      <h4>A related header, such as an alternative title...</h4>
      <h4>... or some tagline!</h4>

```



```

    </hgroup>
  </header>
  <p>This would be a simple paragraph.</p>
</article>

```

Element time

The **time** element may contain both a *machine*-readable and *human*-readable timestamp. The machine-readable part consists of the mandatory *datetime* attribute. The optional *pubdate* attribute may appear only once within an *article* element, indicating its publication date. The +01:00 after the number of milliseconds of the time part indicates UTC+1, the Central European Time zone (CET).

```

<article>
  <header>
    <h3>2nd Article Header</h3>
    This is written the <time datetime="2011-10-22"
      pubdate="pubdate">22nd of October, 2011</time>
  </header>
  <p>We have a meeting
    <time datetime="2011-10-23T14:00:00.000+01:00">
      tomorrow at 2 pm</time>.
    It is <mark>important</mark> that you are present.</p>
</article>

```

Other Elements

The rest of the page consists of the more intuitive **nav**(*igation*) and **aside** element. The *menu* element has no support in any browser just yet. A **footer** finally closes the body area.

```

</section>

<nav>
  <strong>Menu</strong>
  <menu type="list" label="Weekdays">
    <li><a href="sun.htm">Sunday</a></li>
    <li><a href="mon.htm">Monday</a></li>
    <!-- rest of menu -->
  </menu>
</nav>

</div>

<aside>
  <p>Advertising Space</p>
</aside>

<footer>
  <p>[footers always contain some copyright text]</p>
</footer>

</div>
</body>
</html>

```

Divisions

The addition of the semantic and other new HTML5 elements does not make the division (*div*) element useless. The semantically neutral *div* element (defined in HTML 4 as a *generic mechanism for adding structure to documents*) will still be very often used within a Cascading Style Sheet (CSS) in order to layout a web page.

Note The HTML5 specification strongly encourages authors to view the `div` element as an element of *last resort*, for when no other element is suitable. Use of the `div` element instead of more appropriate elements leads to *poor accessibility* for readers and poor maintainability for authors.

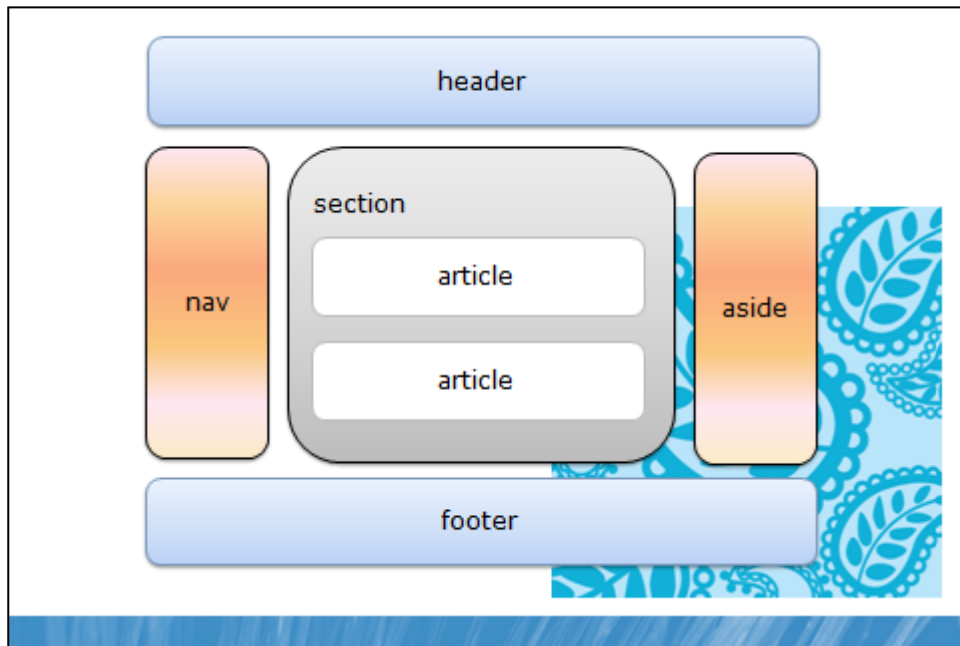
Two `div` elements have been added to our example HTML5 page to be used by the stylesheet printed on the next page. These are necessary to be able to show the page using a three-column layout.

Checking the Document Outline

By checking the HTML5 document outline (mentioned earlier in the “Element hgroup” paragraph), the page can be reviewed to determine the correct hierarchical use of all the (new semantic) html elements. Creating a logical outline keeps the document clear and maintainable. To check your document outline online, search for the words “html5 outliner” using a search engine of choice.

Browser extensions like the HTML5 Outliner for Chrome can be found at <http://code.google.com/p/h5o/>. After installing this extension, Chrome will have a little button in the upper right corner. Clicking this button will show a clickable document outline.

Show Some Style



Key Points

To create the look and feel of an HTML5 web page a Cascading Style Sheet (CSS) is used. This is done in the exact same way as when working with previous versions of HTML.

Basic CSS Document

The following stylesheet can be used to layout the HTML5 document created earlier, using a Search Engine Optimized (SEO) friendly 2-1-3 column ordering. Note that instead of mostly div styling in earlier HTML versions, the new HTML5 elements are present as CSS selectors to be styled as well.

```
nav
{
  float: left;
  width: 140px;
}

section
{
  float: right;
  width: 400px;
}

aside
{
  float: right;
  width: 140px;
}

footer
{
  display: block;
  clear: both;
}
```

```
li
{
  list-style-type: none;
}

#container
{
  width: 700px;
  position: relative;
  text-align: left;
  margin: 0 auto;
}

#columnMaker
{
  float: left;
  width: 550px;
}
```

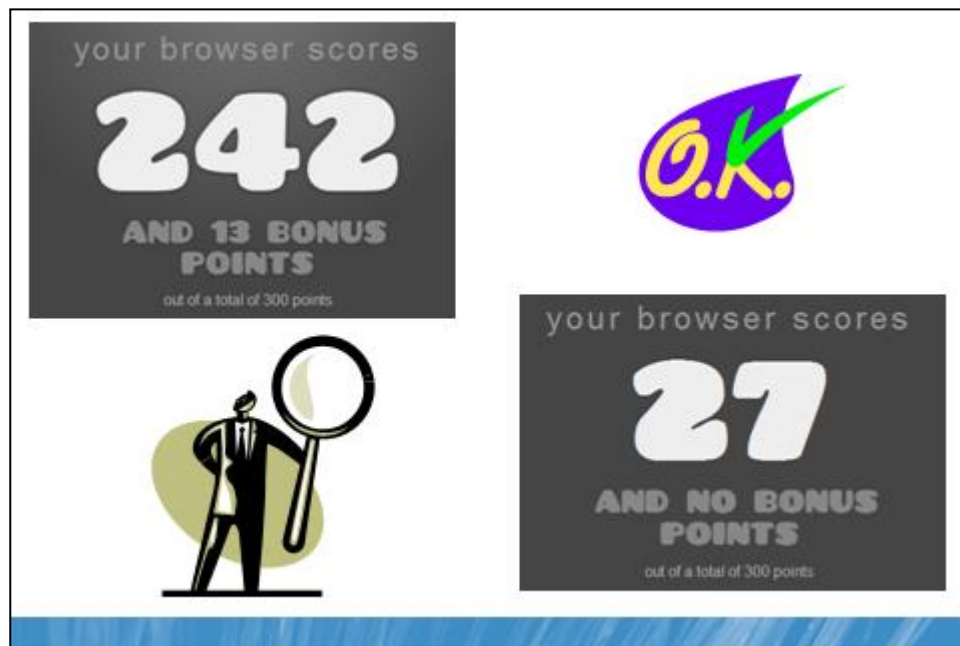
Lesson 3

HTML5 Feature Detection

- It Works On My Machine
- Another Modern Way

This lesson introduces two important methods that can be used to check whether a desired HTML5 feature is supported in the user agent at hand.

It Works On My Machine



Key Points

To make sure that a certain browser supports a desired part of HTML5, it is unwise to just check whether or not HTML5 *as a whole* is supported. That is because HTML5 is not one thing to check for, it is more a *collection* of specific *features*.

Detecting Individual Features

Common techniques to detect HTML5 features are checking if the property concerned exists on a certain global object, or creating an element and checking whether a desired method exists, calling it and making sure its return value is the one expected. Furthermore a certain element could be created, the desired property could be set and then checked for persistence of that value.

This course will show an example for the feature concerned when that particular HTML5 subject will be discussed, for example when checking for the canvas API.

```
function doesThisBrowserSupportCanvas()  
{  
    return !!document.createElement("canvas").getContext;  
}
```

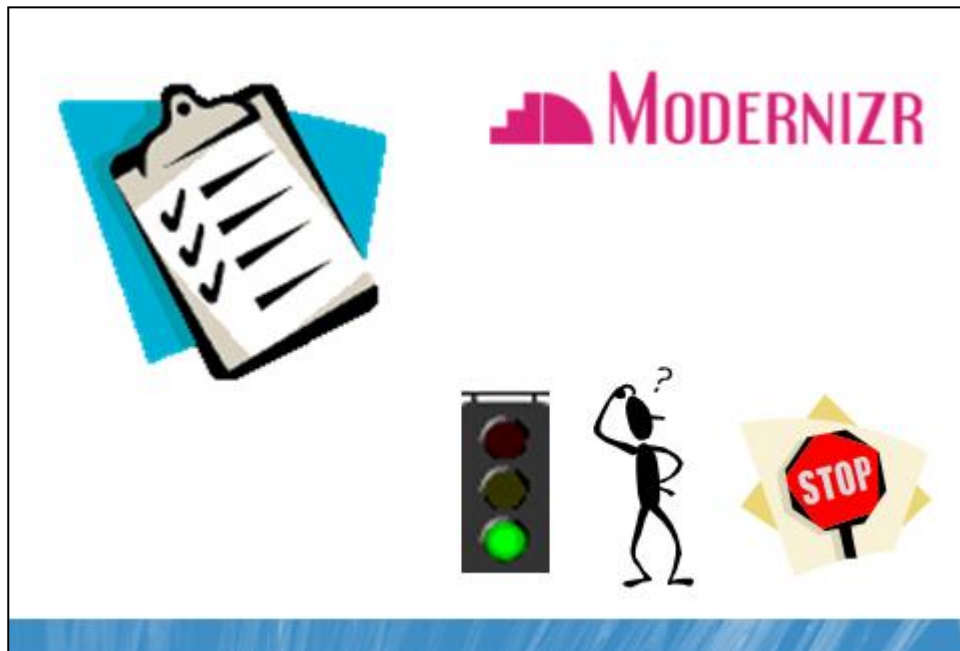
Note The *double not* (!!) operator is used to force the resulting property value to a *boolean*. It is shorter than using the ternary operator (*expression*) ? true : false.

Other Usefull Places to Check

The site <http://www.caniuse.com> shows and *compares compatibility tables for support of HTML5, CSS3, SVG and more in desktop and mobile browsers* as the site's tagline proclaims.

To check the supported features of the browser currently used, navigate to the following web site: <http://www.html5test.com>.

Another Modern Way



Key Points

Another way to check whether the current user agent supports a certain HTML5 or CSS3 feature is to use *Modernizr*, an open source library written in JavaScript. Modernizr is available under the BSD and MIT licenses and can be downloaded here: <http://www.modernizr.com>.

Easy to Use

After downloading the JavaScript file it can be included and used as shown in the following code sample. A global object named *Modernizr* will then be available, containing a *boolean* property for each detectable HTML5 and CSS3 feature. Of course, it is recommended to use the latest version of this free JavaScript library, that is about 10 kilobytes in size.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Are We Lost</title>
    <script src="modernizr.min.js" type="text/javascript"></script>
    <script type="text/javascript">

      if (Modernizr.geolocation)
      {
        // Let us navigate to our house :- )
      }
      else
      {
        // We are alone in the dark :-(
      }

    </script>
  </head>

  <body>

    ...
```


Module 2

Integrated HTML5 APIs

Contents:

Lesson 1: Web Forms	2-3
Lesson 2: Playing Audio and Video	2-12
Lesson 3: The Canvas Element	2-19

Module Overview

- Web Forms
- Playing Audio and Video
- The Canvas Element

In this module some important parts of what is sometimes considered the *integrated* HTML5 Application Programming Interfaces (APIs) are discussed. The lessons include working with the HTML5 form tags and attributes - sometimes called *Web Forms 2.0* - and the added media elements and how to control them using JavaScript. The canvas element is also an important topic to be covered by this module.

Lesson 1

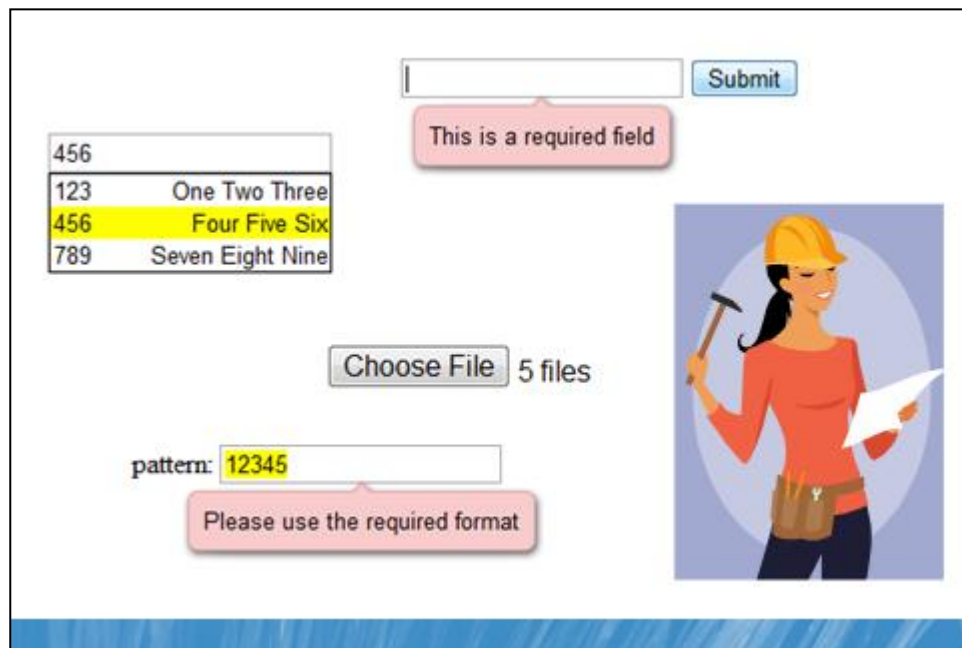
Web Forms

- Handy Attributes
- Dandy Input Types
- Fancy Elements
- Trendy Script Additions

This lesson is about web forms. According to the introductory text of the section about forms on the Web Hypertext Application Technology Working Group (WHATWG) website, it is a component of a web page that has form controls, such as text fields, buttons, check boxes, range controls or color pickers. A user can interact with such a form, providing data that can then be sent to the server for further processing. No client-side scripting is needed in many cases, though an API is available so that scripts can augment the user experience or use forms for purposes other than submitting data to a server.

Until now, the HTML form elements are rather basic in their functionality. The HTML5 specification provides some very handy general attributes to be used on elements, but also some new form elements to adorn your web forms. Furthermore some members have been added to input types to be used by JavaScript code for validation purposes.

Handy Attributes



Key Points

A couple of new attributes have been added by the HTML5 specification to improve usability and the overall user experience. A couple of them will be discussed here.

Autofocus

Using the **autofocus** attribute on an input element like text or checkbox, puts the focus on the that element when the web page loads. When more elements happen to set the autofocus attribute, the last one setting it will actually receive it. The element does not need to be inside the form to receive the focus. In the following example, the radio type input element will get focus:

```
<form action="">
  <input type="checkbox" autofocus="autofocus" />
  <input type="text" autofocus="autofocus" />
</form>

<input type="radio" autofocus="autofocus" />
```

Setting autofocus to “false” will not disable autofocus functionality for that particular element. Removing the attribute altogether does. Also be aware that browsers may scroll down to be showing the focussed element if the form is situated lower on the page.

Placeholder

To place some suggested text into an input text element, the placeholder attribute can be used as follows:

```
<input type="text" placeholder="Please enter your name" />
```

The suggested text will be visible if no content has been entered and if focus is not established. The **placeholder** text will not be posted when submitting the form.

Required

Marking a field as required and thereby cancelling the form’s submit action if no value is entered, can be done as follows:

```
<input type="text" required="required" />
```

The **required** element also gives you another hook when writing custom form validation using JavaScript.

List

The **List** attribute is used to link to a data source in the form of a *datalist* element in order to present the user with a list of options to choose from, besides the ability to enter free text. A code example is shown here:

```
<input type="text" list="myDataList" />
```

The belonging **datalist** element may live (almost) anywhere on the web page and does not render an interface.

```
<datalist id="myDataList">
  <option value="123" label="One Two Three" />
  <option value="456" label="Four Five Six" />
  <option value="789" label="Seven Eight Nine" />
</datalist>
```

Pattern

Some of the new elements that will be discussed on the next page have a certain regular expression embedded, but using the **pattern** attribute makes it possible to set and validate against a custom regular expression. The start of line (^) and end of line (\$) character can be omitted. An example of a dutch postal code follows:

```
<input type="text" pattern="[1-9][0-9]{3}\s?[a-zA-Z]{2}" />
```

Note \s means a *white space character* and ? stands for 0 or 1 repetitions of the preceding part.

Multiple

The **multiple** attribute is not new. It could already be used to enable the user to select more than one option in a *select* element. Now it can be used on an input element with its type set to file. In this way, more than one file can be uploaded at once.

```
<input type="file" multiple="multiple" />
```

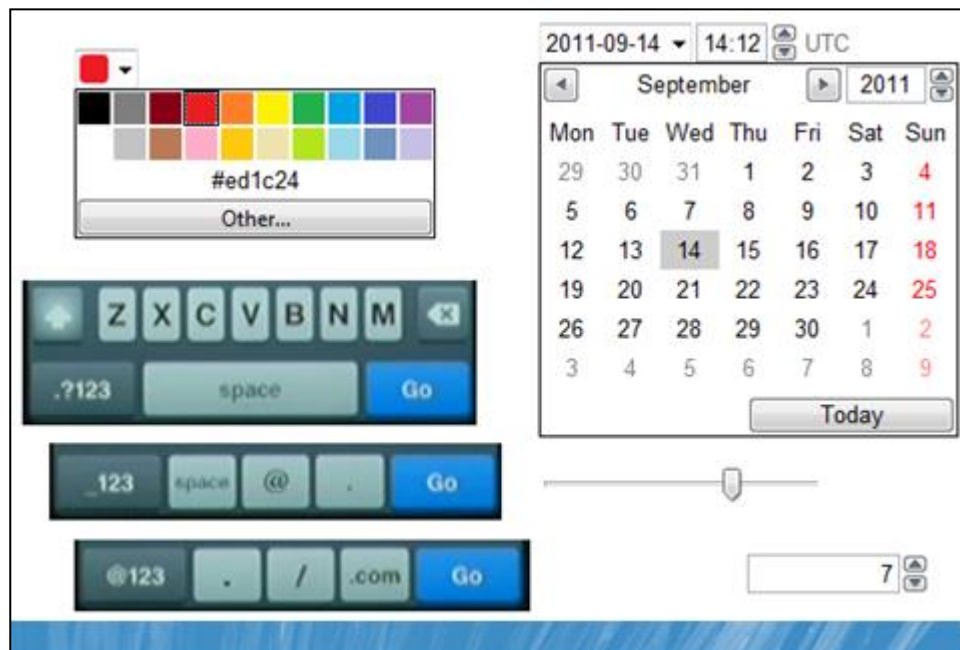
Form

To have an element that is positioned *outside* an HTML form still be posted with that same form, the **form** attribute is invented. Setting the mentioned attribute equal to the *id* of the form concerned makes sure the element is posted to the server. This could look as follows:

```
<form id="frmMain" ... >
  ...
</form>

<input type="text" name="txtOutsideForm" form="frmMain" />
```

Dandy Input Types



Key Points

A lot of new input types are available when using HTML5. The specification does not tell how the browsers should render the new types. Different browsers on different devices will present a different user experience.

For mobile devices, the onscreen keyboard display for a certain input type may also vary. To increase usability it is made easier to enter a related symbol, like @, period (.) “.com”, forward slash (/) or a complete numeric keypad (for the *tel* type), depending on the input type concerned.

Check it Out

Checking browser support using *Modernizr*, the **inputtypes** property can be checked, using the desired type, as follows:

```
var bEmailInputTypeSupport = Modernizr.inputtypes.email;
```

Another technique is to create an input element in memory, set its type attribute to the desired HTML5 type and then check for value persistence, as follows:

```
var i = document.createElement("input");
i.setAttribute("type", "email");
var bEmailInputTypeSupport = (i.type !== "text");
```

Email and Web Addresses

Using the following markup a browser can check whether an **email** address is formatted correctly. There is no definition of how errors should be reported.

```
<input type="email" />
```

A Uniform Resource Locator (URL) field should be marked using the following syntax:

```
<input type="url" />
```

Numeric Up Down

When setting input type equal to **number**, some kind of spinner, spinbox or numeric up down control might appear. Optional attributes like *min*, *max* and *step* could be used as well.

```
<input type="number" min="3" max="12" step="2" />
```

Time and Again

Version 5 of HTML finally defines some kind of date and/or time picker in its standards. This could mean no more custom JavaScript or AJAX implementation to show the user a calendar or time selector. Full date and time display, including a time zone, could be marked up as follows:

```
<input type="datetime" />
```

Other possible date and/or time related type options are listed in this table:

Input Type	Description
datetime-local	Date and time display, with no setting or indication for time zones.
time	Time indicator and selector, with no time zone information.
date	Selector for calendar date.
week	Selector for a week within a given year.
month	Selector for a month within a given year.

Slider

If a number needs to be entered without typing the actual figure, setting the type to **range** could result in a slider of some form. Again the attributes *min*, *max* and *step* could be of use here.

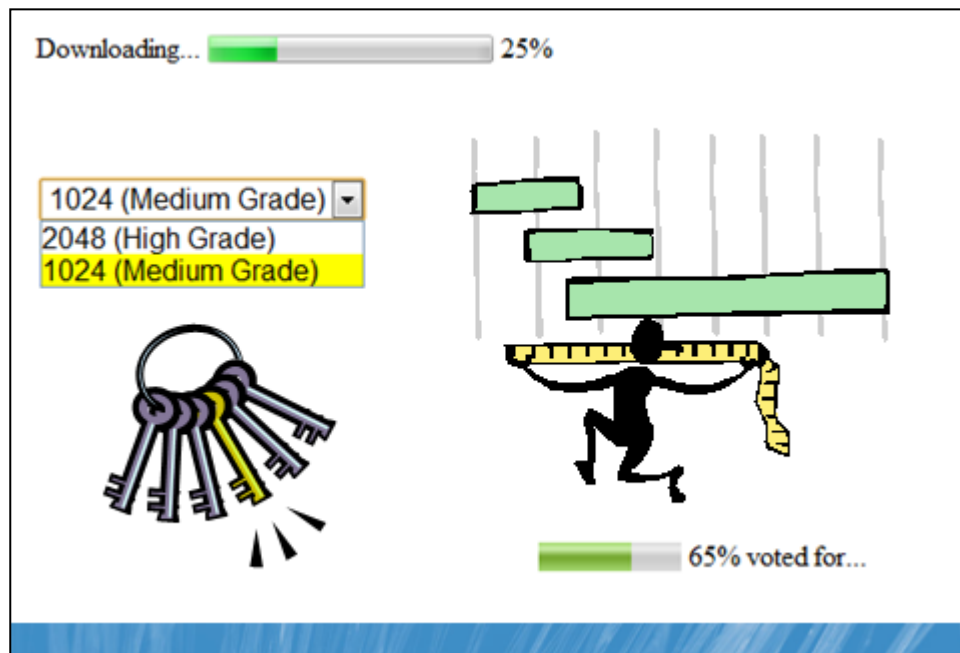
```
<input type="range" min="5" max="15" step="3" />
```

Other Types

The HTML5 specification defines some other types. They are listed here

Input Type	Description
tel	Phone number box. Numeric keypads of mobile devices may adapt to it.
search	Meant for search terms. Browsers may show rounded input box corners.
color	Shows some kind of color picker.

Fancy Elements



Key Points

Next to the attributes and input types discussed before, there are also some brand new elements documented by HTML5. We will look at them here.

Check it Out

The following example checks to see whether the *progress* element is supported by the current user agent:

```
var bProgress = (document.createElement("progress").value != undefined);
```

Progress Bar

The **progress** element could be used in conjunction with JavaScript to display the progress of a task or process as it is underway. The *max* and *value* attributes can be used to manage its progress.

```
<progress max="100" value="35"></progress>
```

Gauge

The **meter** tag is used for indicating a scalar measurement within a known range or a fractional value. Also called a *gauge*, usage could include displaying disk usage, a fraction of a voting population to have selected a certain product or political party or the relevance of a search query result.

```
<meter max="20" value="17"></meter>
```

Show some Results

For showing results coming from a piece of JavaScript like a calculation, the **output** element could be used. The *value* attribute is used to set the desired result from script.

```
<output>some result</output>
```

Asymmetric Key Generator

In case a pair of public/private cryptographic keys need to be generated, the **keygen** element can be used. When the form the control is part of is submitted, the private key is stored in the local keystore, and the public key is packaged and sent to the server.

```
<keygen name="secretKey" />
```

Additional attributes are listed in the following table:

Attribute	Description
challenge	Enables a string value to be packaged with the submitted public key.
keytype	Specifies the security algorithm of key generated. Defaults to “rsa”.

Trendy Script Additions

- <element>
- validity
 - stepMismatch
 - rangeOverflow
 - rangeUnderflow
 - tooLong
 - patternMismatch
 - typeMismatch
 - valueMissing
 - customError
 - valid
- willValidate
- checkValidity()
- setCustomValidity()
- valueAsNumber

Key Points

Besides all the new attributes, input types and elements, HTML5 also adds several JavaScript artifacts to the user agents. We will review some of them here.

Validity State Object

The **ValidityState** object is the type returned when getting the **validity** property of an *element*. This can be done as shown here:

```
var txt = document.getElementById("someInputBox");

var objValidityState = txt.validity;
```

The *ValidityState* object contains the following properties:

Property	Description
stepMismatch	True when its value does not fit the rules given by the <i>step</i> attribute.
rangeOverflow	True when its value that is too high for the <i>max</i> attribute.
rangeUnderflow	True when its value is too low for the <i>min</i> attribute
tooLong	True when its value is too long for its <i>maxlength</i> attribute.
patternMismatch	True when its value does not satisfy the <i>pattern</i> attribute.
typeMismatch	True when its value is not in the correct syntax (like <i>email</i> or <i>url</i>).
valueMissing	True when it has no value but has a required attribute.
customError	True when its custom validity error message (set by the <i>setCustomValidity</i> method) is not an empty string.
valid	True when all the constraints mentioned above have passed.

After referencing the *ValidityState* object, it will always show up-to-date information about the element concerned. That is because the validity checks it returns are updated automatically depending on any changes that may occur on the element at hand.

Other Validation Members

Before validating an element, its **willValidate** property could be used to check whether any validation rules have been defined for that element. The method **checkValidity** can be used per element and on the whole form to force all validation rules to fire and to return the outcome, as if the form was submitted normally.

Custom Validation

If it appears necessary to create your own validation functionality and show a different message to go with it, the **setCustomValidity** method could be used. The *setCustomValidity* method sets the read-only DOM attribute *validationMessage* of the element concerned and sets the *customError* property of the *ValidityState* object to false.

```
function validateForm()
{
    var txt = document.getElementById("txtRequired");

    if (txt.value == "")
    {
        txt.setCustomValidity("I'm sorry, Dave. I'm afraid I can't do that.");
    }
    else
    {
        txt.setCustomValidity("");
    }
}
```

Property **valueAsNumber**

Instead of the *value* property of an input type that always returns a *string*, the **valueAsNumber** property could be used if a numeric value is expected. It returns the value as a *floating-point* number or *NaN* if no number is present (or in case there is no value at all).

Lesson 2

Playing Audio and Video

- Containers and Codecs
- HTML5 Media Elements
- Media Scripting

The HTML5 specification contains elements that enable browsers to playback sounds and movies without an additional plugin. Media containers and codecs will be discussed before introducing the media elements and their most important attributes. Scripting these elements will be demonstrated last.

Containers and Codecs



Key Points

Before the new HTML5 media elements are presented it is important to discuss some important characteristics of media containers and codecs.

Containers

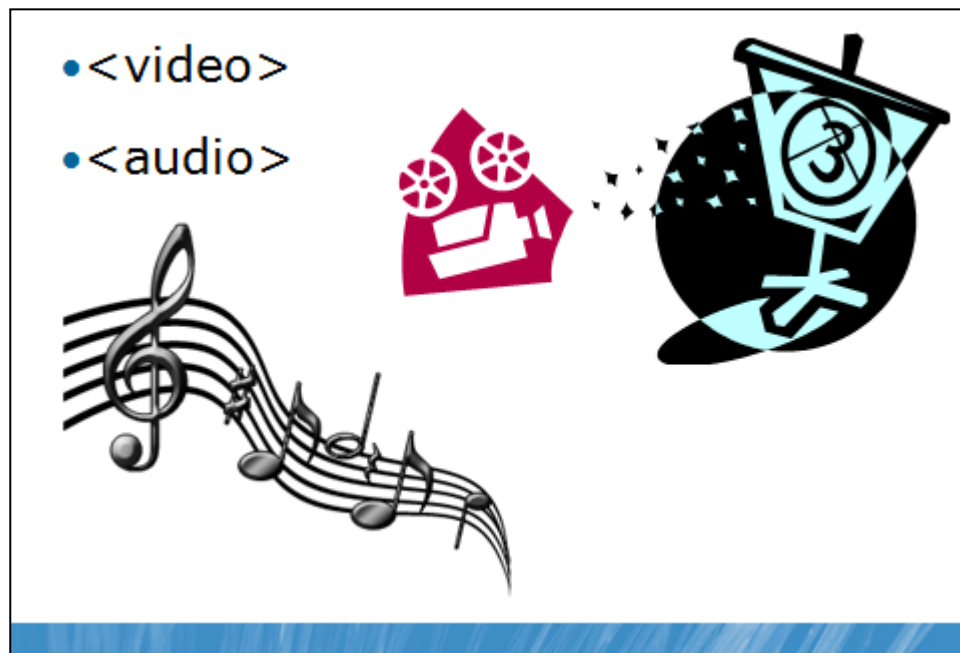
Known video and audio files like *AVI* files or *MP4* files are *container* files. They can be seen as a kind of *ZIP* or *RAR* file that contains a number of files. A Video file contains one or more video tracks and audio tracks. An audio track contains markers to synchronize the sounds with the video. At runtime the audio and video tracks are combined to play the movie. Containers can also have metadata, like the title of the video itself, cover art and sometimes episode numbers. Examples of other video containers are Flash Video (*.flv*), Matroska (*.mkv*), Ogg and WebM. Some known audio containers are WAV and AIFF.

Codecs

A Video or audio **coder** and **decoder** (*codec*) is an algorithm (software program) by which a media stream is *encoded* into a certain format. A media player *decodes* the stream again to be able to play it back. Codecs make use of compression techniques, which is necessary because raw media files would be too large to transmit and handle within an acceptable amount of time. Examples of audio codecs are MPEG-1 Audio Layer 3 (*MP3*), Advanced Audio Coding (*AAC*) and Ogg Vorbis. The most relevant codecs for video are H.264, VP8 and Ogg Theora. It is important to know that the HTML5 specification makes no recommendation about codecs at all.

For an overview of which media containers supports which media encoding format, see the following web page: http://en.wikipedia.org/wiki/Comparison_of_container_formats.

HTML5 Media Elements



Key Points

HTML5 provides two important media elements, the audio and video tag. Both the video and audio API derive from the same media API, so the events and members are very much alike. Both media tags will be discussed along with their most important attributes.

Check it Out

Using *Modernizr*, the video and audio properties can be checked to determine user agent support, like so:

```
var bCanPlayVideo = Modernizr.video;  
var bCanPlayAudio = Modernizr.audio;
```

Another technique is to create an audio or video element in memory and check its **canPlayType** property, as follows:

```
var bCanPlayVideo = !!document.createElement("video").canPlayType;  
var bCanPlayAudio = !!document.createElement("audio").canPlayType;
```

To see whether a certain media container is supported, *Modernizr* can be used as follows:

```
var strWebMSupp = Modernizr.video.webm;  
var strOggSupp = Modernizr.video.ogv;
```

The following code could be used as well:

```
var video = document.createElement("video");  
var strWebMSupp = video.canPlayType('video/webm; codecs="vp8, vorbis"');  
var strOggSupp = video.canPlayType('video/ogg; codecs="theora, vorbis"');
```

The value returned is either an *empty string* for no support, “maybe” if the only the container is asked for and supported and “probably” if both the container and the desired codec(s) are asked for and supported. No *true* is ever returned, because the user may have, for example, changed certain (mobile) device settings to disable (downloading and) playing a certain size of media files altogether, without the browser being aware of it. Furthermore, even when providing both

the container and the codecs, it still does not include information like the actual bitrate but only the maximum bitrate. An overview of *video type parameters* and their possible return values is found here: http://wiki.whatwg.org/wiki/Video_type_parameters.

Saturday Night at the Movies

The **video** tag has a *src* attribute of its own, but it is better to create more child elements of type **source**, containing different video files and optionally their Multipurpose Internet Mail Extensions (*MIME*) type and the *video* and *audio* codecs to be used. The user agent will start with the first *source* element, working its way down until a supported media file is found.

```
<video width="400" controls="controls">
  <source src="vid.webm" type='video/webm; codecs="vp8, vorbis"' />
  <source src="vid.mp4" type='video/mp4; codecs="avc1.42E01E, mp4a.40.2"' />
  <source src="vid.ogv" type='video/ogg; codecs="theora, vorbis"' />
  Sorry, your browser does not support the video element.
</video>
```

It is recommended to specify as much (correct and verified) detail as possible for the **type** attribute. The browser will check this attribute first and if it exists, it is used to determine whether to download that particular file. Not specifying the *type* attribute will result in the browser downloading (part of the) file before concluding whether or not it is supported, wasting bandwidth, memory, processor time and user patience.

Play it Again, Sam

As said, the audio element is based on the same media API as the video tag. So to play some sound or music from inside an HTML5 compatible browser, the following syntax could be used:

```
<audio controls="controls" >
  <source src="mus.ogg" type='audio/ogg; codecs="vorbis"' />
  <source src="mus.mp3" type='audio/mpeg; codecs="mp3"' />
  <source src="mus.aac" type='audio/mp4; codecs="mp4a.40.5"' />
  Sorry, your browser does not support the audio element.
</audio>
```

Common Attributes

The next table lists some of the attributes that can be used on both media elements, with the exception of the *poster* and *height* and *width* attributes, which only work for the video element.

Attribute	Description
controls	Shows controls like play, volume and media progress.
autoplay	Starts the media concerned automatically.
loop	Starts the desired media from the beginning after it finishes.
src	Points to the media file that is to be played.
poster	Indicates the image to be shown while the video is being downloaded or until the video is started (<i>video element only</i>).
height, width	Size of the video in pixels (<i>video element only</i>).

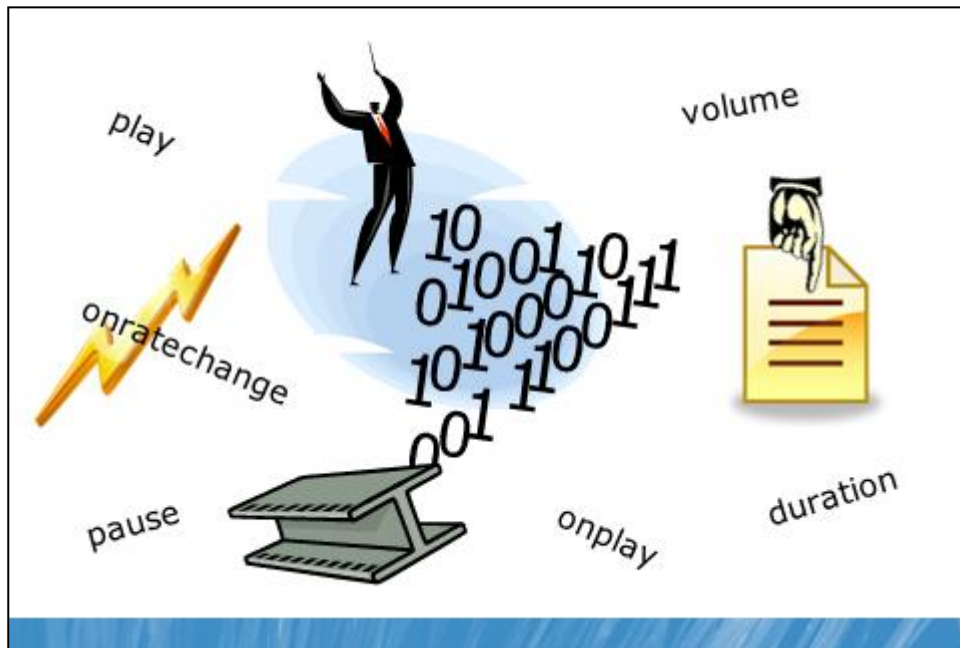
Visit http://www.quackit.com/html_5/tags/html_video_tag.cfm for more attributes and other information on the video element (a link to the audio element is present on that page as well).

Web Server MIME Types

If a media file will not play in the browser, make sure the web server does not block the MIME type concerned. Ways for setting MIME types differ for every web server brand. When using Microsoft *ASP.NET*, MIME types can be set by declaring the following elements in the so-called **web.config** file.

```
<configuration>
  <system.webServer>
    <staticContent>
      <mimeTypeMap fileExtension=".mp4" mimeType="video/mp4" />
      <mimeTypeMap fileExtension=".m4v" mimeType="video/m4v" />
    </staticContent>
  </system.webServer>
</configuration>
```


Media Scripting



Key Points

The media elements can be controlled by JavaScript as well. Creating custom playback control possibilities, responding to events or setting properties at runtime, it is all possible. We will discuss and show some possibilities here.

Properties

The next table shows some media properties and their purpose.

Property	Description
currentTime	The current playback time in seconds. Setting this attribute moves the playback position to the specified time index.
duration	Amount of seconds that the full media clip. If unknown, <i>NaN</i> is returned.
paused	Boolean indicating whether the media clip is being paused (<i>read-only</i>). If the clip is not started yet, its value is <i>true</i> .
volume	Gets or sets the volume of the playing media clip, taking a float value ranging from 0.0 (silent) to 1.0 (loudest).
playbackRate	The current playback speed, for example 1.0 is normal, 2.0 is two times faster forward, 0.5 is half speed.

The following code example shows how to use the value of an input type range to set the *volume* of the media element.

```
var videoElm = document.getElementById("videoElement");
var volRange = document.getElementById("volumeRange");

videoElm.volume = volRange.value;
```

For more information on these and other properties of media elements, see the following web page: <http://www.chipwreck.de/blog/2010/03/01/html-5-video-dom-attributes-and-events/>.

Methods

The following table shows some media methods and their purpose.

Method	Description
play	Will always start the media clip from the current playback position. When a clip is first loaded, this will be the beginning of the clip.
pause	Pauses the media clip playback when currently playing.

Here is an example of how to start playing a video only when the element is in *pause* mode.

```
if (videoElm.paused)
{
    videoElm.play();
}
```

More methods are found here: <http://www.kaizou.org/2009/04/html-5-audio-video-elements/>.

Events

The media elements also publish events that client-side scripts can subscribe and respond to. Some examples are listed in the next table.

Event	Description
onplay	Happens when playback has begun. Fired after the play() method has returned, or when the autoplay attribute has caused playback to begin.
onpause	Occurs when playback has been paused. Fired after the pause() method has returned.
onratechange	Fires if either the <i>defaultPlaybackRate</i> or the <i>playbackRate</i> attribute has just been changed.

In the media element, the desired event can be coupled declaratively to an event handler method:

```
<video onplay="togglePlayPauseButtonCaption('Pause');" ... >
```

The *addEventListener* method of the element concerned can also be used to programmatically hook up the method concerned, as follows:

```
videoElm.addEventListener(
    "pause",
    function ()
    {
        togglePlayPauseButtonCaption('Play');
    },
    false // The event handler is to be executed in the bubbling phase,
);      // instead of the capturing phase
```

Note When using the *addEventListener* method, the first argument (*EventType*) needs to be written *without* the “on” prefix.

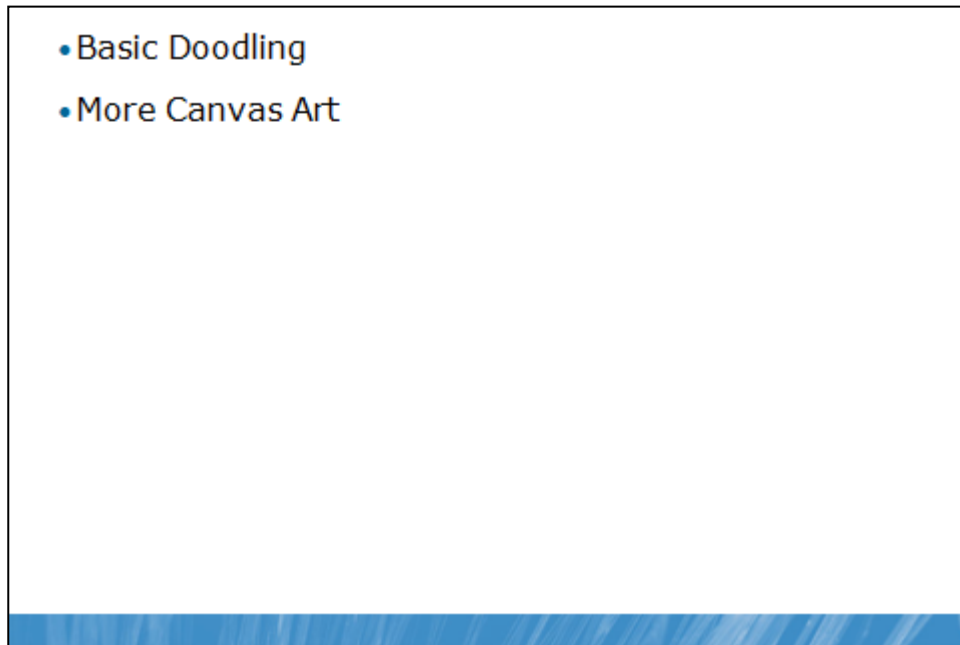
The belonging method could look like this:

```
function togglePlayPauseButtonCaption(desiredCaption)
{
    document.getElementById("playOrPauseButton").value = desiredCaption;
}
```

For information on media events, see: <http://www.w3.org/TR/html5/video.html#mediaevents>.

Lesson 3

The Canvas Element

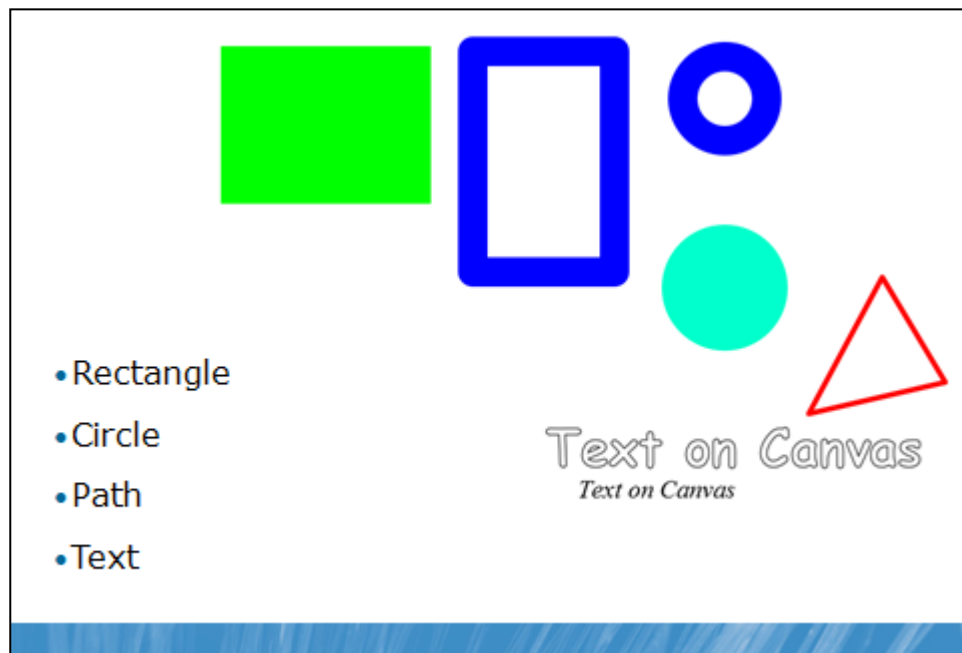


Canvas is the element in HTML5 that enables web developers to create a drawing environment within the browser. JavaScript code is able to access the canvas element through a full set of drawing functions similar to other common graphics APIs. Examples of canvas usage include building graphs, animations, games, and image composition.

The *Canvas 2D* API is a low level, procedural model that updates a resolution-dependent bitmap. Therefore it is *not* a *vector-based* environment like Scalable Vector Graphics (SVG), which the HTML5 specification allows to be embedded directly in web pages as well.

The specifications warn web authors not to use the canvas element in a document when a more suitable element is available. It is said, for example, not to be appropriate to use a canvas element to render a page heading. If the presentation of the heading is graphically intense, it should be authored using the appropriate element (like *h1*) and then styled using a Cascading Style Sheet (CSS).

Basic Doodling



Key Points

The HTML5 *canvas* element is meant to draw graphics on a web page using JavaScript. It is a rectangular area of which every pixel can be controlled. The canvas element has several methods for drawing paths, boxes, circles, characters and adding images. This topic will discuss this element and some of the key features of the belonging *Canvas 2D* API.

Check it Out

To make sure the user agent at hand supports canvas, use the following *Modernizr* property:

```
var bCanvasSupport = Modernizr.canvas;
```

Checking for canvas browser support without an external tool is done as follows:

```
var bCanvasSupport = !!document.createElement('canvas').getContext;
```

Stretching the Canvas

A web page may contain one or more canvas elements. They can be styled like any other element using Cascading Style Sheets (CSS). The next example shows an inline style to create a border around the canvas. If *width* and *height* are not present, the canvas measures will *default* to 300 pixels wide by 150 pixels high.

```
<canvas id="myCanvas" width="800" height="500" style="border:1px solid;">
  We are sorry to inform you that this browser does NOT support canvas.
</canvas>
```

Stick to the Context

To be able to control the canvas element just marked up, a reference to its *2D* context needs to be created (which is the only context type *available* at this time). This could be done using a function in JavaScript, as shown here:

```
function get2dContext()
{
  var myCanvas = document.getElementById("myCanvas");
  var myContext = myCanvas.getContext("2d");
}
```

```
    return myContext;
}
```

Rectangles and Circles

After getting a reference to the 2D context of the canvas concerned, several basic drawing figures can be created. For example, a *solid* green rectangle could be made like this:

```
// Getting hold of the context first.
var ctx = get2dContext();

ctx.fillStyle = "#00FF00";
ctx.fillRect(20, 30, 200, 150);
```

A solid blue colored *open* rectangle, having a line thickness of 28 *canvas coordinate space units* and rounded corners is created as follows:

```
ctx.lineWidth = 28;
ctx.lineJoin = "round";

ctx.strokeStyle = "#0000FF";
ctx.strokeRect(260, 35, 135, 210);
```

Circles can be drawn using the *arc* method of the context. The next code draws an open circle.

```
var endAngle = Math.PI * 2; // 360 degrees

ctx.beginPath();

ctx.arc(500, // x
      80, // y
      40, // radius
      0, // start angle
      endAngle, // end angle
      true // clockwise (false = counter-clockwise)
);

ctx.closePath();

ctx.stroke(); // use ctx.fill() to draw a solid circle
```

For information on how to create an oval shape, see the following page on the HTML5 Canvas Tutorials site via <http://www.html5canvastutorials.com/tutorials/html5-canvas-ovals>.

Stay on the Path

The *Path* object enables the drawing of custom shapes. Using paths, all kinds of complicated shapes can be created, with multiple line and curve segments and even subpaths.

First the *outline* of the figure can be drawn. This can be seen as drafting with a *pencil*. Then the shape can be drawn using a stroke or solidly filled. This can be seen as the *inking* part, after which the path becomes visible.

The following piece of source code draws a triangle using the path object. Using the *ctx.fill()* method instead of *ctx.stroke()* on the last line would draw a solid triangle in this case.

```
// First "pencil" the desired outline.
ctx.beginPath();

ctx.moveTo(650, 250);
```

```
ctx.lineTo(710, 350);
ctx.lineTo(580, 380);

// Close the Path to its start point.
ctx.closePath();

ctx.strokeStyle = "#FF0000";
ctx.lineWidth = 5;

// Now draw the line for real using "ink".
ctx.stroke();
```

For more information on *paths*, see <http://diveintohtml5.org/canvas.html#paths>.

Signing a Painting

The *2D Canvas* API makes it possible to draw both *solid* and *outlined* characters. An example of how to do this, is shown here:

```
ctx.font = "bold 48px Comic Sans MS";
ctx.fillText("Hello HTML5 Canvas", 30, 40); // use ctx.strokeText() for
// outlined characters
```

See https://developer.mozilla.org/en/drawing_text_using_a_canvas for more information.

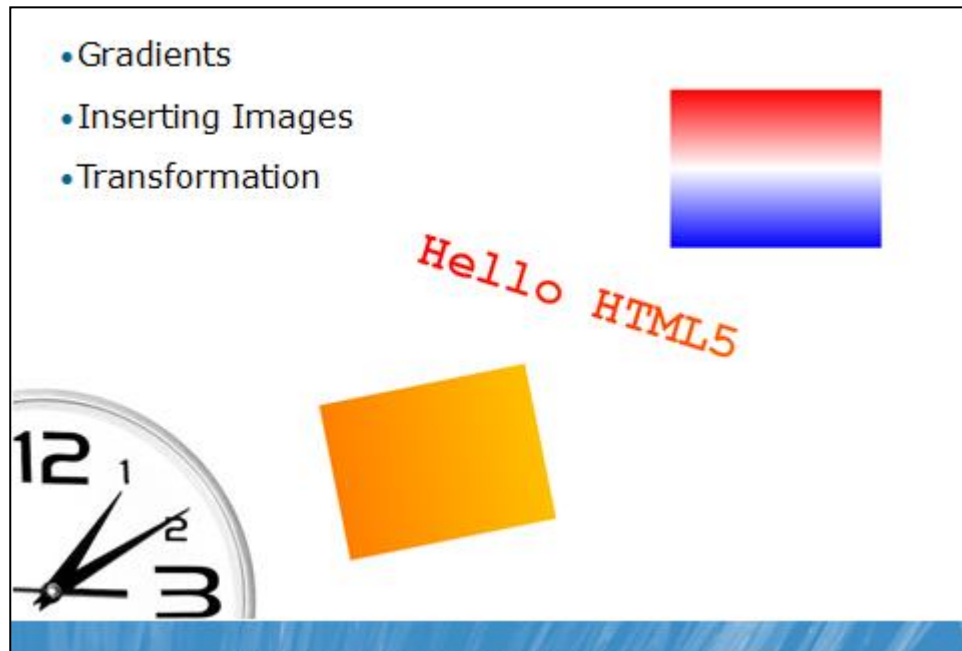
Starting with a Clean Slate

The trick used to *erase* the drawing context of the canvas is setting its *width* or *height* property to another value. The following method shows an example of how this could be done.

```
function resetCanvas()
{
    var myCanvas = document.getElementById("myCanvas");

    myCanvas.height = myCanvas.height++;
    myCanvas.height = myCanvas.height--;
}
```

More Canvas Art



Key Points

In this topic some less basic *Canvas 2D* API functionality will be discussed.

Color Gradient

When working with the canvas element, it is also possible to work with both *linear* and *radial* color gradients. The `createLinearGradient()` method shown below takes four arguments. The first two define the *x* and *y* coordinates of the *starting* point of the gradient, the last two represent the gradient's *ending* point.

In the code example the gradient is positioned vertically (from top to bottom), because both *x* coordinates are equal and only the *y* coordinates differ. Using two or more color stops (using a floating point value between 0.0 and 1.0 representing the position between the start and end points), the gradient can be completed. The gradient object is then set as the **fillStyle** of the rectangle to be drawn.

```
// Paints along a line from (x0, y0) to (x1, y1)
var myGradient = ctx.createLinearGradient(
    580, // x0
    60, // y0
    580, // x1
    150 + 60 // y1
);

myGradient.addColorStop(0.0, "Red");
myGradient.addColorStop(0.5, "White");
myGradient.addColorStop(1.0, "Blue");

ctx.fillStyle = myGradient;

ctx.fillRect(580, 60, 200, 150);
```

For another example of *linear* gradients and how to create a *radial* gradient, see the following page: http://www.tutorialspoint.com/html5/canvas_create_gradients.htm.

Paint me a Picture

It is also possible to load an image and place in anywhere inside the canvas using the method **drawImage()**. The first two parameters of this method are the coordinates of the spot the picture should be drawn. The optional third and fourth parameter are used to scale the image.

If an image is attempted to be drawn onto the canvas before it has been loaded completely, the image will not be rendered at all. That is the reason for the *callback* method to be executed after the image finished loading. The *callback* method can then safely call the *drawImage()* method.

```
function loadImage()
{
    imgClock = new Image();

    imgClock.src = "image/clock.jpg";

    // Call the addImage() method as soon as the image is loaded.
    imgClock.onload = addImage;
}

function addImage()
{
    var ctx = get2dContext();

    ctx.drawImage(imgClock, 0, 185, 500, 315);
}
```

Transformation

Using the *Canvas 2D* API performing *transformations* like *scaling*, *translating* and *rotating* on the canvas are made possible. It is advised to *save* the state of the context just before any transformation is performed. Then the desired transformations are to be performed at the origin of the canvas (coordinate 0,0). Right after that, the saved context state should be *restored* again.

This is necessary because (unless you manually reset all transformation settings) all transformation (and other context) settings will remain active on future shapes, as well.

So by calling the **save()** method on the context, its state is stored on a stack. When calling **restore()**, the last saved state is retrieved from that stack and all context settings are restored.

The following example first sets the rotation of the canvas to 30 degrees and the translation to 100, 100. Then a text is put on it. The canvas is rotated *back* 50 degrees and a rectangle is drawn.

```
var ctx = document.getElementById("myCanvas").getContext("2d");

ctx.save();

ctx.rotate(0.30);
ctx.translate(100, 100);

ctx.font = "bold 48px Courier New";
ctx.textBaseline = "top";
ctx.fillText("Hello HTML5", 0, 0);

ctx.rotate(-0.50);
```



```
ctx.fillRect(400, 80, 200, 150);  
  
ctx.restore();
```

More information and working examples about several transformation methods can be found at the following page: https://developer.mozilla.org/en/Canvas_tutorial/Transformations.

Module 3

HTML5 Associated APIs

Contents:

Lesson 1: Geolocation API	3-3
Lesson 2: Web Storage API	3-9
Lesson 3: Web Sockets and Web Workers	3-14

Module Overview

- Geolocation API
- Web Storage API
- Web Sockets and Web Workers

A lot of segments of the HTML5 initiative were originally part of the HTML5 specification and were moved to separate standards documents later on, in order to keep the specification focused. It was considered smart to discuss and edit some of these features on a separate track before making them into official specifications. Some small markup issue would not hold up the process of the entire specification in this way.

This module discusses some of these Application Programming Interfaces (APIs) that are not part of the official HTML5 specifications, but have their own Web Hypertext Application Technology Working Group (WHATWG) and/or World Wide Web Consortium (W3C) specifications.

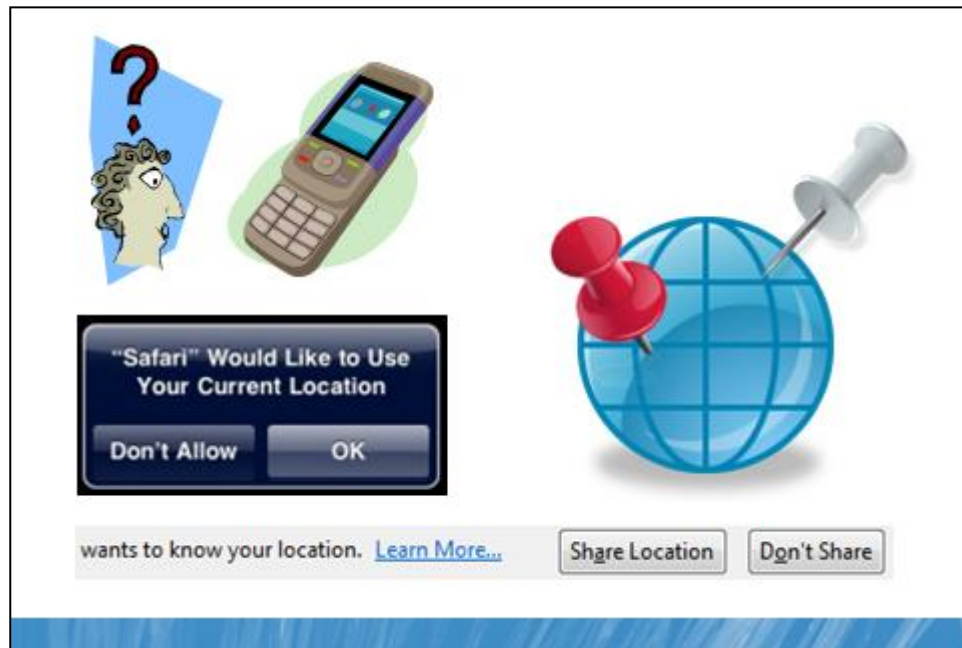
Lesson 1

Geolocation API

- What is the Geolocation API
- API Objects and Members

The Geolocation API allows web developers to build applications that can determine where the user is and then share or act on that information. The method of position discovery is left up to the user agent. This API gives developers a set of objects and methods to handle and process location data.

What is the Geolocation API



Key Points

The Geolocation API is an effort by the World Wide Web Consortium (W3C) to standardize an interface to retrieve the geographical location information for a client-side device. It defines a set of ECMAScript standard compliant objects that executes in the client application. It tries to locate the client device by using location information servers, which are transparent for the API.

Examples of methods that are used for position determination are IP address, Wi-Fi and Bluetooth MAC address, RFID, Wi-Fi connection location, or device GPS and GSM/CDMA cell IDs. The location is returned with a given accuracy depending on the best location information source available. No guarantee is given that the API returns the device's actual location.

An opt-in Service

Geolocation support is opt-in, meaning the browser will never force the user to reveal its current physical location to a remote server. The user has to give permission to the application in order for the location data to be received. This addresses privacy concerns and needs to be accounted for in the planning and design of any location-aware application. The user experience differs from browser to browser.

API Objects and Members

- Global Navigator Object
 - `navigator.geolocation`
- One-shot position request
 - `getCurrentPosition()`
- Requesting repeated position updates
 - `watchPosition()`, `clearWatch()`



Key Points

The geolocation API can be reached through the global navigator object via a property called `geolocation` and has the following methods:

- `getCurrentPosition`
- `watchPosition`
- `clearWatch`

All these methods will be discussed next, together with error handling techniques.

Getting the Position Once

The method `getCurrentPosition` makes a so-called one-shot position request, taking two callback functions as arguments, the first one to be called for a successful find, the other (which is optional) in case of any error. The next JavaScript code example illustrates this.

```
if (navigator.geolocation) // or use if (Modernizr.geolocation)
{
    navigator.geolocation.getCurrentPosition(successCallback, errorCallback);
}
else
{
    alert("Geolocation API not available");
}
```

Location was Found

The callback function that is called in case of success, takes a *position* object as parameter. The possible properties of this position object are listed here.

	Property	Data Type	Remark
Has a value	coords.latitude	double	Decimal degrees
	coords.longitude	double	Decimal degrees
	coords.accuracy	double	Meters
	timestamp	DOMTimeStamp	Like a Date() object
Can be null	coords.altitude	double <i>or null</i>	Meters above the reference ellipsoid
	coords.altitudeAccuracy	double <i>or null</i>	Meters
	coords.heading	double <i>or null</i>	Degrees clockwise from true north
	coords.speed	double <i>or null</i>	Meters per second (m/s)

The method called when a position was found, could look as follows:

```
function successCallback(position)
{
    var latitude = position.coords.latitude;
    var longitude = position.coords.longitude;
    var datetime = new Date(position.timestamp);
    var accuracy = position.coords.accuracy;

    alert("On " + datetime.toLocaleString() + ", you could be here: "
        + latitude + ", " + longitude
        + "\n(give or take " + accuracy + " meters).");
}
```

Something is Wrong

If something is not working out, for example the user does not give permission to share his or her position, the errorCallback method is called. This method takes an error object parameter containing the error *code* and sometimes the *message* property is filled as well. The following table lists the possible error codes and their meaning.

Constant	Value	Description
PERMISSION_DENIED	1	User selects “Don’t Share” button or otherwise denies access to his location.
POSITION_UNAVAILABLE	2	The network is down or the positioning satellites cannot be contacted.
TIMEOUT	3	The Network is up but it takes too long to calculate the user’s position.

Here is an example of an error handling method:

```
function errorCallback(err)
{
    switch (err.code)
    {
        case err.PERMISSION_DENIED:
            //
            break;
        case err.POSITION_UNAVAILABLE:
            //
            break;
        case err.TIMEOUT:
            //
            break;
        default: // UNKNOWN_ERROR
            //
            break;
    }
}
```



```
}
}
```

Follow Me Please

The **watchPosition()** method has the same signature as the **getCurrentPosition()** method shown before. The main difference is that the callback function will be called every time the location of the device changes. Actively polling the current position is therefore not necessary. The device determines the optimal polling interval, and will call the desired method when a change in the user's position is determined.

The return value of the **watchPosition()** method is a *unique watch identifier*. In order to stop the watching mechanism, this identifier should be passed to the **clearWatch()** method. The JavaScript methods **setInterval()** and **clearInterval()** work the same way.

The following code snippets show how to use both methods including the use of the unique watch identifier.

```
var numWatchID;

numWatchID = navigator.geolocation.watchPosition(
    successRepeatingCallback, errorCallback, { enableHighAccuracy: true } );

// ...

function stopFollowingMe()
{
    navigator.geolocation.clearWatch(numWatchID);
}
```

Final Options

There is a third argument that can be passed to both the **getCurrentPosition()** and **watchPosition()** methods, an object of type *PositionOptions*. This object has the following properties:

Property	Data Type	Default	Remark
enableHighAccuracy	boolean	false	true is usually slower
maximumAge	long	0	milliseconds (ms)
timeout	long	0	milliseconds (ms)

Each of these properties will be explained in short.

- **enableHighAccuracy** makes sure that the device tries to get a more detailed reading of the current position. If the phone includes a Global Positioning System (GPS) receiver, that service will be used to get a more detailed latitude and longitude. Unfortunately this will demand more power from the battery.
- **maximumAge** can be set to allow the device to answer more quickly to a position request by using cached location data. It can be set to *Infinity* to always use data from cache memory. A default value of 0 (zero) makes sure the browser will look up a new position on each request.
- **timeout** specifies the time to wait for a position, before the error callback is called with the **TIMEOUT** code. If not set, no time limit is applied.

The next code example shows how to set the properties on the *PositionOptions* object argument using the JSON notation.

```
navigator.geolocation.getCurrentPosition(successCallback, errorCallback,  
  {  
    enableHighAccuracy: true,  
    maximumAge: 60000,  
    timeout: 10000  
  }  
);
```

Lesson 2

Web Storage API

- Storage Overview
- Scripting the Storage

Web Storage is a separate HTML5 specification and is sometimes referred to as Local Storage, DOM Storage, HTML5 Storage or Offline Storage. It is an Application Programming Interface (API) that makes persisting data in the client browser possible and available for repeated access across requests. It is also possible to store data beyond the lifetime of the user agent.

Storage Overview



Key Points

Until now the only way to store some data on the client machine was by using cookies. Their size is limited to 4 kilobytes and they are being transported to and from the web server on every request. Two kinds of cookies exist. *Session* or *temporary* cookies are available during the browser session and are stored in browser memory. *Persistent* or *permanent* cookies are stored in small text files on the hard drive of the client computer and are available depending on their expiration date.

Session and Local Storage

The Web Storage API gives the possibility to store data in the form of name/value pairs within the client browser. The amount of storage space is about 5 MB (depending on the user agent). This data is never transmitted to or from the web server. Two ways of local storage are present: *Session Storage* and *Local Storage*. Both implement the same *Web Storage* interface.

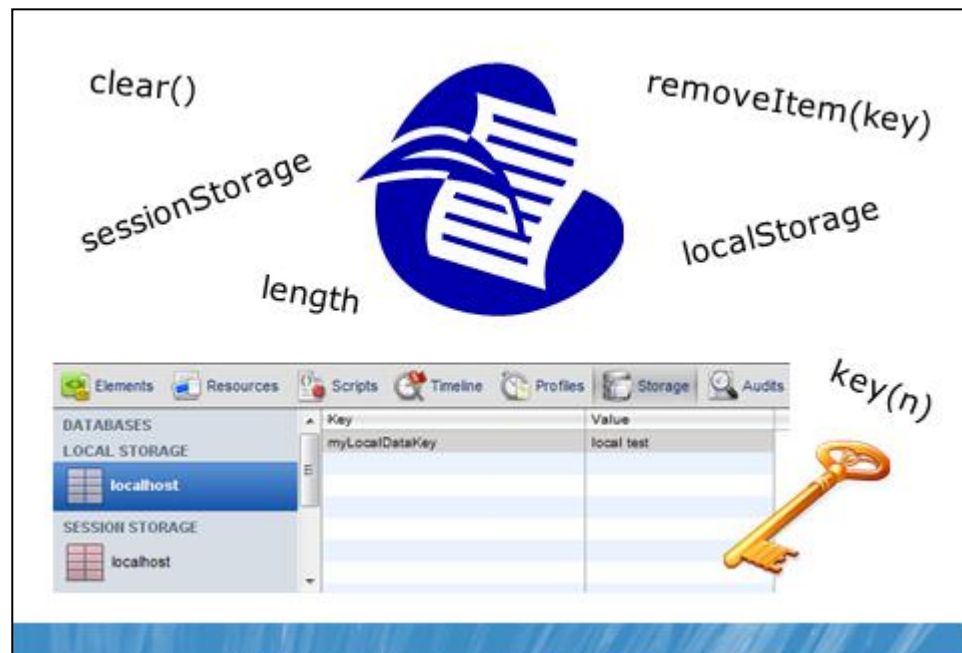
Using *Session Storage*, values persist as long as the browser (tab) window exists and are only visible from the browser (tab) windows in which they were created. This prevents “leaking” from one (tab) window to another, something that can happen when using cookies.

Local Storage values persist beyond (tab) window and even user agent lifetimes and are shared across every browser (tab) at the same origin. Local Storage data will remain available until explicitly removed by client side code or the user.

Another Local Storage Option

Besides Web Storage there is also the *Web SQL Database* API, another separate part of the HTML5 specification. This API can be used for storing data in databases that can be queried using a variant of SQL. For more information on these specifications and the uncertain future of it, see <http://html5doctor.com/introducing-web-sql-databases>.

Scripting the Storage



Key Points

This topic will discuss and show how to use the members of the *Storage* interface, implemented by both the *sessionStorage* and *localStorage* objects.

Check it Out

When using *Modernizr*, support for both local and session storage can be checked like so:

```
var bLocalStorage = Modernizr.localstorage;
var bSessionStorage = Modernizr.sessionstorage;
```

Another way to determine whether storage is supported by the current browser is as follows:

```
var bLocalStorage = ("localStorage" in window)
    && window["localStorage"] !== null;

var bSessionStorage = ("sessionStorage" in window)
    && window["sessionStorage"] !== null;
```

Put it Away

To store a name/value pair in either the *localStorage* or the *sessionStorage* is shown here:

```
localStorage.setItem("myLocalDataKey", "some local data value");
```

Another syntax is possible as well by using a so-called *expando* property:

```
localStorage.myLocalDataKey = "some local data value";
```

Where it says **localStorage**, **sessionStorage** can be used. This goes for all code examples.

Get it Back

Retrieving data works as follows. *Expando* property syntax can be used here as well.

```
var someResult = sessionStorage.getItem("mySessionDataKey");

var sameResult = sessionStorage.mySessionDataKey;
```

Other properties and methods of the `localStorage` and `sessionStorage` objects are listed here.

Property or Method	Description
<code>removeItem(key)</code>	This method removes the item from the list.
<code>length</code>	Property to get the length of the storage object array.
<code>key(n)</code>	Method used to return the name of the <i>n-th</i> object in the array.
<code>clear()</code>	Calling this method empties all key/value pairs from the array.

Catch the Storage Event

An event is fired when any change is made to the data. An event listener can be registered to the window object, which allows tracking of the *Storage* object concerned, the key name, the old and new value and the url of the web page from which the event fired.

Registering an event handler for the storage event looks as follows:

```
window.addEventListener("storage", showMeTheEventValues, false);
```

Note The last *boolean* argument of the *addEventListener* method indicates whether the event handler is to be executed in the *capturing* phase (*true*) or in the *bubbling* phase (*false*).

The belonging method is shown here.

```
function showMeTheEventValues(e)
{
    var out = "key: " + e.key
        + ", newValue: " + e.newValue
        + ", oldValue: " + e.oldValue
        + ", url: " + e.url
    + ", storageArea: " + e.storageArea;

    alert(out);
}
```

The event argument of the *Storage Event* has the following properties:

Property	Meaning
<code>key</code>	The key value that was removed or updated.
<code>newValue</code>	Contains the value after the change.
<code>oldValue</code>	Contains the previous value before it was updated.
<code>url</code>	Page of origin where the event occurred.
<code>storageArea</code>	Provides a reference to the <i>sessionStorage</i> or <i>localStorage</i> concerned.

Exploring Web Storage

In some browsers the values stored in local and session storage can be browsed, edited and deleted. Here are some browsers and the steps to take to view these storage panels.

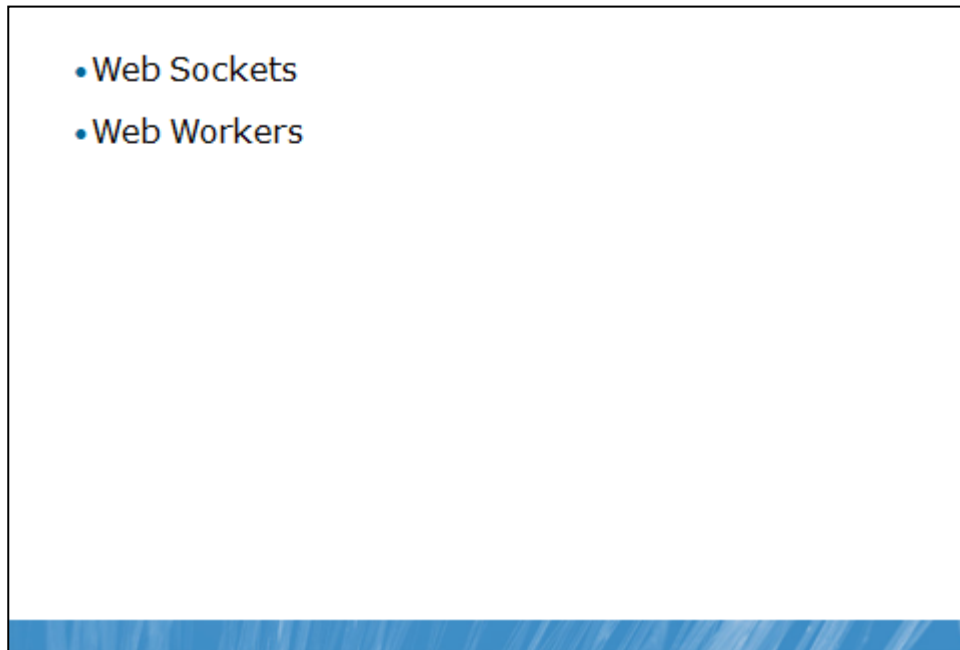
Browser	Steps
Opera	Press [Menu] in upper left corner, then select <i>Show Menu Bar</i> . In the now appearing menu, select Tools > Advanced > Opera Dragon Fly. In the appearing Dragon Fly pane, select the <i>Storage</i> tab above and then the <i>Local Storage</i> and <i>Session Storage</i> tabs at the bottom.
Chrome	Click the <i>Wrench</i> menu button (top right) and select Tools > Developer Tools. In the appearing pane select the <i>Storage</i> tab. Different <i>storage</i> options can found on the left.

Safari	Press [ALT] to show the menu. Select menu Edit > Preferences. Click the Advanced button. Check the “Show Develop menu in menu bar” checkbox. Now press [ALT] again. Select menu <i>Develop</i> > Show Web Inspector. In the appearing pane, select the <i>Storage</i> tab above. Different storage options can found on the left.
Firefox	Using the Firebug panel, select the Console tab. In the bottom left corner, after the >>> characters either type <i>sessionStorage</i> or <i>localStorage</i> followed by [ENTER]

Note All the latest browsers support native JSON encoding using the `JSON.stringify()` and `JSON.parse()` methods. These methods could be used to store and retrieve (custom) *objects* as strings in and out of Web Storage.

Lesson 3

Web Sockets and Web Workers



Although they are different Application Programming Interfaces (APIs), both *Web Sockets* and *Web Workers* share the same communication API. That is why they are discussed together in this lesson.

Web Sockets

Since the beginning of its existence, the web has been unidirectional. Web pages were only able to send a request to a web server. The web server could never take the initiative to send a message to the client.

A few tricks have been used to mimic the behavior of bi-directional communication. One of them is called *short polling*, where the client makes a request every few seconds to check whether server data has changed. So either new data or an empty message are returned, resulting in a huge amount of server requests and high CPU usage due to constant data change tracking.

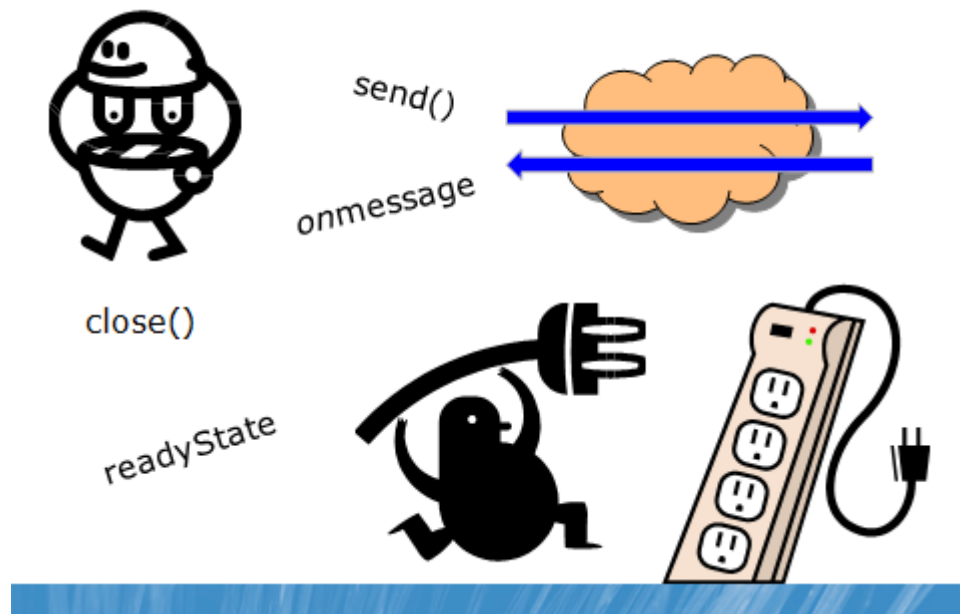
Another way to imitate some sort of two-way communication is known as *long polling*. In this case the client makes a single request and the server keeps the connection open until the desired data has changed, after which the server sends its response. This results in a overhead in CPU usage as well because of constant data change tracking. Furthermore (several) thousands of open connections must be handled by the web server simultaneously.

The HTML5 *Web Sockets* API defines a full-duplex communication technology that operates over a single socket and is exposed via a JavaScript interface in user agents.

Web Workers

Heavy computing has historically been difficult to achieve using JavaScript on the browser. This was due to the single threaded nature of the scripting language. The *Web Workers* specification defines an API that allows developers to spawn background workers running JavaScript code in parallel to their main page. This allows for thread-like operations with the passing of string messages as coordination mechanism.

Web Sockets



Key Points

The *Web Sockets* API makes it possible to create a bi-directional connection between a server and a client. It involves a real-time connection that stays open until it is explicitly closed. So *Web Sockets* enable the web server to send a request to the client. So instead of polling, the server can *push* data to the client browser.

When a new connection needs to be set up, the client browser opens an HTTP connection to the server first and during the initial handshake they both upgrade to use the *Web Socket* protocol. Once the connection had been established, the *Web Socket* provides a full-duplex channel between client and server. This enabled sending text-based messages in either direction at the same time.

Check it Out

Should *Modernizr* be used, the following line of code determines support for *Web Sockets*:

```
var bSocketSupport = Modernizr.websockets;
```

This is another way to check whether the *Web Sockets* API can be used in the current browser:

```
var bSocketSupport = !!window.WebSocket;
```

First Contact

The first step is creating a new Web Socket connection to an *endpoint* at a certain url (using the prefix *wss://* would indicate a secure connection).

```
var ws = new WebSocket("ws://localhost:8181/test");
```

Handling Events

Using the new *Web Socket* object that was returned, events can be handled using the common JavaScript syntax. The next example shows different ways to receive a message from the server.

```
ws.onmessage = receiveMsg;
// or
ws.addEventListener("message", receiveMsg, false);
```

```
// and the belonging event handler method
function receiveMsg(e)
{
    // message is in e.data
}

// or by using an anonymous function
ws.onmessage = function(e)
{
    // message is in e.data
}
```

The following table shows other events that can be listened to.

Event	Description
<i>onopen</i>	Occurs when the connection has been established.
<i>onclose</i>	Occurs when the connection is closed.
<i>onerror</i>	Occurs when an exception has been raised during communication.

Properties

The property members of the Web Socket object are shown in the next table.

Property	Description
<i>readyState</i>	This property of data type <i>unsigned short</i> that holds the current state of the connection. The following values are possible (<i>readonly</i>): 0 (<i>CONNECTING</i>) : connection has not yet been established 1 (<i>OPEN</i>) : connection has been established and communication is possible 2 (<i>CLOSING</i>) : connection is going through the closing handshake 3 (<i>CLOSED</i>) : connection has been closed or could not be opened
<i>bufferedAmount</i>	The number of <i>UTF-8</i> text bytes that were queued using the <i>send()</i> method.

These members could be used as follows:

```
function updateStatus()
{
    var readyState      = document.getElementById("readyState");
    var bufferedAmount = document.getElementById("bufferedAmount");

    readyState.innerHTML = ws.readyState;
    bufferedAmount.innerHTML = ws.bufferedAmount;
}
```

Methods

The functions that are supported by the *Web Sockets* object are listed in this table.

Method	Description
<i>send(string)</i>	Transmits string data using the connection it is called on.
<i>close()</i>	Terminates the connection it is called on.

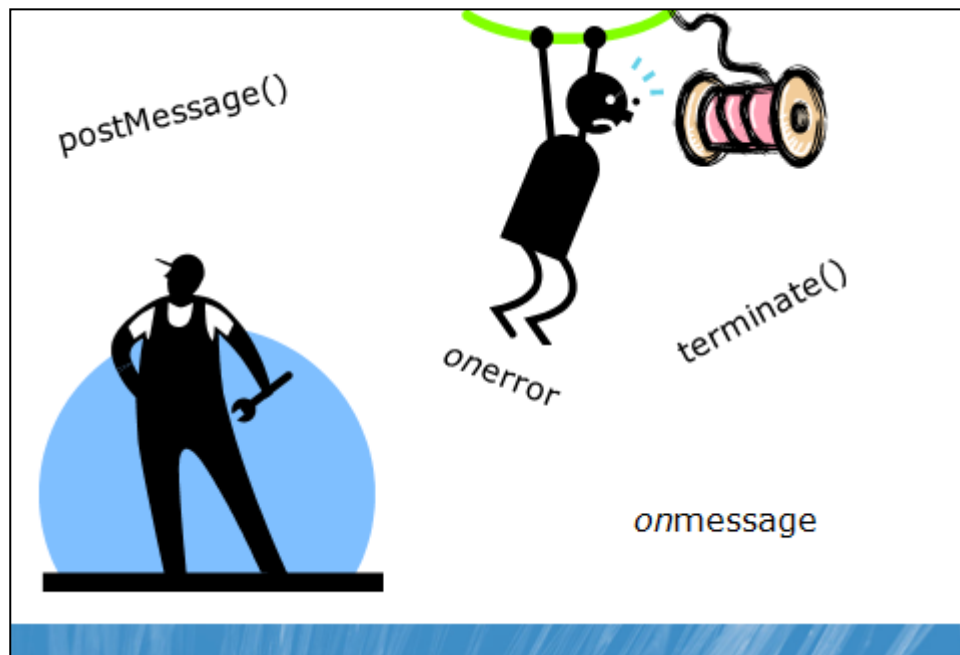
The following code example shows how a message could be sent to the server:

```
function sendToServer()
{
    var msg = document.getElementById("txtMessage").value;

    ws.send(msg);
}
```

Note All the latest browsers support native JSON encoding using the `JSON.stringify()` and `JSON.parse()` methods. These methods could be used to send and receive (custom) *objects* as strings to and from the web server.

Web Workers



Key Points

The *Web Workers* API makes it possible to run client-side script code in the background, separated from and independent of any user interface action or code. This allows long tasks (like complex calculations, making network requests or accessing local storage) to be executed without yielding to keep the page responsive. *Web Workers* have no access to the browser Document Object Model (DOM), meaning they cannot read or modify the HTML document directly.

Check it Out

To make sure the *Web Workers* API is supported by the user agent, *Modernizr* is helping out.

```
var bWebWorkerSupport = Modernizr.webworkers;
```

Checking the *Worker* property of the browser's *window* object can also be used to determine Web Workers support, like so:

```
var bWebWorkerSupport = !!window.Worker;
```

Separate Worker File

All communication between the JavaScript code on the web page and the Web Worker is performed by using asynchronous *callback* methods sending text strings. The Web Worker methods need to be in a separate *.js* file. The content of such a file could be as follows:

```
onmessage = function (e)
{
    postMessage("Worker starts counting to " + e.data);

    startWorking(1, parseInt(e.data));

    postMessage("Worker finished counting to " + e.data);
}
```

In this case, an *anonymous* method is used to handle the *message* event. Every common JavaScript event handling syntax could have been used here as well, like a separate event handler function or the *addEventListener()* method.

The **postMessage()** method is used to send string messages back to the calling client. The actual so-called *worker method* that is present in the same *.js* file is shown next:

```
function startWorking(begin, end)
{
  for (var i = begin; i <= end; i++)
  {
    if (i % 1000 == 0)
    {
      postMessage(i);
    }
  }
}
```

Make it Work

The *.js* file containing worker methods needs to be called from an HTML5 web page, another *.js* file (or even from within another worker method). The following line of code instantiates a new Web Worker object using an *url* to the *.js* file discussed before.

```
var webWorker = new Worker("myFirstWorker.js");
```

To receive messages sent from the worker methods, common JavaScript event handler syntax can be used, for example:

```
webWorker.onmessage = handleMsgFromWorker;

function handleMsgFromWorker(e)
{
  // message is in e.data
}
```

To finally set the *Web Worker* example mentioned earlier in motion, this code could be used:

```
webWorker.postMessage(numTo);
```

Exception Handling

When a *runtime error* occurs anywhere inside the running *Web Worker* code, the *onerror* event is raised. The belonging *event* parameter has some properties that are of interest, listed in the next table.

Property	Description
message	The error message in human-readable form.
filename	The full <i>url</i> of the script file in which the exception occurred.
lineno	The line number of the script file concerned that caused the exception.

Handling a *Web Worker* exception could look as follows:

```
webWorker.onerror = function (e)
{
  alert("Error : " + e.message + "\n"
    + "File : " + e.filename + "\n"
    + "Line : " + e.lineno);
}
```

Termination

If the *Web Worker* needs to be terminated immediately from the *outside*, the method `webWorker.terminate()` can be used. From one of the methods in the *.js* file, the `close()` method could be called to *terminate* the worker from the *inside*.

Importing Scripts

If some external (*.js*) script file needs to be imported by a Web Worker *.js* file, the following syntax is used (this alternative mechanism is necessary because Web Workers do not have access to the *document* object). A *comma-separated* list of files is also possible.

```
importScripts("someScript.js", "anotherOne.js");
```

Shared Web Workers

Since *Web Worker* threads need some time to start initially and use a lot of memory per dedicated *Web Worker*, being able to share the Web Worker instead of creating one for each open window (from the same domain) can improve performance.

Shared Worker objects need a bit more work to set up than dedicated Web Workers do. For more information on this, see <http://dev.w3.org/html5/workers/#shared-workers-introduction>.