

# Some notes on Javascript

John R Hudson<sup>\*†</sup>

11th August 2015

<b>Contents</b>		
<b>1 Introduction</b>	<b>4</b>	
<b>2 Javascript objects</b>	<b>5</b>	
2.1 The built-in environment . . . . .	5	
2.2 The host environment . . . . .	5	
<b>3 Object types</b>	<b>5</b>	
<b>1 The built-in environment</b>	<b>6</b>	
<b>4 The global object</b>	<b>6</b>	
4.1 Global functions . . . . .	6	
<b>5 Objects</b>	<b>7</b>	
5.1 Object properties . . . . .	7	
5.2 Accessing object properties . . . . .	8	
5.3 Object methods . . . . .	9	
<b>6 Functions</b>	<b>10</b>	
6.1 Built-in functions . . . . .	11	
6.2 Function methods . . . . .	11	
6.3 Constructor functions . . . . .	11	
6.4 Object prototypes . . . . .	11	
6.5 Object prototype properties . . . . .	12	
6.6 Constructor function properties . . . . .	12	
<b>7 Arrays</b>	<b>14</b>	
7.1 Array properties . . . . .	15	
7.2 Array methods . . . . .	15	
7.2.1 Stack manipulation of arrays . . . . .	15	
7.2.2 Array methods for functions . . . . .	16	
<b>8 Strings</b>	<b>16</b>	
8.1 String methods . . . . .	17	
<b>9 Boolean objects</b>	<b>18</b>	

<sup>\*</sup>With thanks to David Flanagan (2011, 2012)

<sup>†</sup>The author would welcome notification of any errors or possible misunderstandings.

<b>10</b>	<b>Number and math objects</b>	<b>19</b>	<b>19</b>	<b>The document object</b>	<b>36</b>
10.1	Number object methods . . . . .	19	19.1	Document properties . . . . .	36
10.2	Math objects . . . . .	19	19.2	Document methods . . . . .	37
10.2.1	Math object properties . . . . .	19			
10.2.2	Math object methods . . . . .	20			
<b>11</b>	<b>Date objects</b>	<b>20</b>	<b>20</b>	<b>The element node</b>	<b>38</b>
11.1	Date object methods . . . . .	22	20.1	Element node properties . . . . .	38
			20.2	Element node methods . . . . .	38
<b>12</b>	<b>Regular expressions</b>	<b>23</b>	<b>21</b>	<b>Collection objects</b>	<b>38</b>
12.1	Regular expression methods . . . . .	25	21.1	Collection object properties . . . . .	39
			21.2	Collection object methods . . . . .	39
<b>13</b>	<b>Error objects</b>	<b>25</b>	<b>22</b>	<b>Element objects</b>	<b>41</b>
			22.1	Element object properties . . . . .	42
<b>14</b>	<b>JSON</b>	<b>25</b>	22.2	Element methods . . . . .	44
			22.3	Child element methods . . . . .	46
			22.4	Element object constructors . . . . .	46
<b>15</b>	<b>Statements</b>	<b>26</b>	<b>23</b>	<b>The text node</b>	<b>46</b>
15.1	Declaring variables . . . . .	26	23.1	Text node properties . . . . .	46
15.2	Conditionals . . . . .	27			
15.3	Flow control . . . . .	27	<b>24</b>	<b>Canvas element objects</b>	<b>47</b>
15.4	Exception and error handling . . . . .	29	24.1	Contexts . . . . .	47
			24.2	2d context . . . . .	48
			24.2.1	Context, path and styles attributes . . . . .	48
			24.2.2	Hit regions . . . . .	56
			24.2.3	Pixel manipulation . . . . .	57
<b>II</b>	<b>The host environment</b>	<b>30</b>	<b>25</b>	<b>Media element objects</b>	<b>57</b>
<b>16</b>	<b>Host environment objects</b>	<b>30</b>	25.1	Media element properties . . . . .	58
			25.2	Media element methods . . . . .	61
<b>17</b>	<b>The window object</b>	<b>31</b>	25.3	audioTrack objects . . . . .	61
17.1	Window object properties . . . . .	31	25.3.1	audioTrack object properties . . . . .	61
17.2	Window object methods . . . . .	33	25.4	textTrack objects . . . . .	63
			25.4.1	textTrack object properties . . . . .	63
<b>18</b>	<b>The document node</b>	<b>35</b>			
18.1	Document node properties . . . . .	35			
18.2	Node methods . . . . .	35			

25.4.2 textTrack object methods . . . . .	64	32.4 HTTP events . . . . .	78
25.5 videoTrack objects . . . . .	64	32.5 Message events . . . . .	79
25.5.1 videoTrack object properties . . . . .	64	32.6 Message methods . . . . .	79
25.6 MimeType and Plugin objects . . . . .	64	32.7 Message constructors . . . . .	79
25.6.1 MimeType and Plugin object properties . . . . .	64	32.8 Server sent events . . . . .	80
25.6.2 MimeType and Plugin object methods . . . . .	65		
<b>26 Table element objects</b>	<b>65</b>	<b>33 HTTP Requests</b>	<b>80</b>
26.1 Table element properties . . . . .	65	33.1 HTTP methods . . . . .	80
26.2 Table element methods . . . . .	66	<b>34 ValidityState object</b>	<b>81</b>
<b>27 Form element objects</b>	<b>67</b>	34.1 ValidityState object properties . . . . .	81
27.1 Form element object properties . . . . .	67	34.2 ValidityState object methods . . . . .	81
27.2 Form element object methods . . . . .	69		
<b>28 The history object</b>	<b>69</b>	<b>III Additional Javascript features</b>	<b>82</b>
28.1 History properties . . . . .	69	<b>35 Application cache objects</b>	<b>82</b>
28.2 History methods . . . . .	70	35.1 Application cache object properties . . . . .	82
<b>29 The location object</b>	<b>70</b>	35.2 Application cache object methods . . . . .	83
29.1 Location object properties . . . . .	70	35.3 Application cache events . . . . .	83
29.2 Location object methods . . . . .	71	<b>36 Web workers</b>	<b>84</b>
<b>30 The navigator object</b>	<b>71</b>	<b>37 Web sockets</b>	<b>85</b>
30.1 Navigator object properties . . . . .	71	37.1 Web socket methods . . . . .	85
30.2 Navigator object methods . . . . .	72	<b>38 Web storage</b>	<b>86</b>
<b>31 Drop and drag</b>	<b>74</b>	38.1 Storage properties . . . . .	86
31.1 Drop and drag properties . . . . .	74	38.2 Storage methods . . . . .	86
31.2 Drop and drag methods . . . . .	75	<b>A Reserved words</b>	<b>88</b>
<b>32 Event handlers</b>	<b>76</b>	<b>B Built-in operators</b>	<b>88</b>
32.1 The DOMEvents object . . . . .	77	B.1 Arithmetic and string operators . . . . .	88
32.2 Event methods . . . . .	78	B.2 Arithmetic only operators . . . . .	89
32.3 Error event object . . . . .	78		

B.3 Relational operators . . . . .	89	B.7 Conditional operator . . . . .	90
B.4 Equality operators . . . . .	89	B.8 Assignment operators . . . . .	90
B.5 Bitwise operators . . . . .	90	B.9 Comma operator . . . . .	90
B.6 Logical operators . . . . .	90	B.10 Date operators . . . . .	91

## 1 Introduction

Javascript is an interpreted object oriented scripting language developed for browsers and reliant on the normal features of a browser. It is not therefore intended for use as a standalone language. It is case sensitive and uses UTF-16. These notes mainly cover strict mode Javascript as defined by Ecma International (2011).

Throughout these notes, the following shorthands are used in the code examples:

- a** for `<array>` meaning any array like object
- c** for any of `HTMLCollection`, `HTMLAllCollection`, `HTMLFormControlsCollection`, `HTMLOptionsCollection` or `HTMLPropertiesCollection` array objects
- d** for `<date>` or `<document>`, depending on context, meaning any date object or document
- e** for `<element>` meaning any browser element object
- f** for `<function>` meaning any function object
- h** for `<handle>` or `<HTTP Request>`, depending on context, meaning an event handle or any `HTTPRequest` object
- i** for an image
- k** for `key`
- m** for `<media resource>` meaning a media object
- n** for `<node>` or `<number>`, depending on context, meaning any node or number object
- n* meaning any number, normally an integer
- o** for `<object>` meaning any object
- p** for `<property>` or `<path>`, depending on context, meaning any property of an object or a path in a `<canvas>` object
- r** for `<RegExp>` meaning any regular expression object
- s** for `<storage>`, `<string>` or `<styles>`, depending on context, meaning any storage, any string like object or `<styles>` in a `<canvas>` object
- t** for `<text>` meaning any text in an object
- v** for a non-numeric, or undefined, value
- w** for `window` meaning the host window object

## 2 Javascript objects

A Javascript object is a collection of **properties** and their **values**; a callable object is a **function**. Properties may hold other objects, functions or primitive values, such as **undefined**, **null**, **Boolean**, **Number** or **String**, and have attributes such as *writeable*, *enumerable* and *configurable*. A **function** that is associated with a **property** of an **object** is called a **method**.

Javascript objects may be:

**host** objects provided by a web browser or a web server environment

**native** objects provided by the Javascript environment which are independent of the browser or server environment

**built-in** objects provided by the Javascript environment which are **native** objects but not independent of the browser or server environment.

### 2.1 The built-in environment

The built-in objects include the **global** object, the **Object** object, the **Function** object, the **Array** object, the **String** object, the **Boolean** object, the **Number** object, the **Math** object, the **Date** object, the **RegExp** object, the **JSON** object, and the **Error** objects **Error**, **EvalError**, **RangeError**, **ReferenceError**, **SyntaxError**, **TypeError** and **URIError**.

### 2.2 The host environment

A web browser host environment will provide objects for windows, menus, pop-ups, dialogue boxes, text areas, anchors, frames, history, cookies, and input/output along with requests, clients, files and mechanisms to lock and share data and means to attach scripting code to events such as change of focus, page and image loading, unloading, error and abort, selection, form submission, and mouse actions. This scripting code is reactive to user interaction.

---

## 3 Object types

Javascript objects are untyped; the values associated with an object's properties may be declared as one type and then used as another type. The available types are **Undefined**, **Null**, **Boolean**, **String**, **Number**, and **Object**.

You can determine which type of value an object's declared properties hold with `o.typeof()`; (see Appendix B).

Note that, when converting types, **true**, **"true"** and **1** are equivalent as are **false**, **"false"** and **0**; **undefined**, **null** and **"** (an empty string) are also **false** whereas **Infinity** and **[]** (an empty array) are **true**. **NaN** may be **true** or **false** depending on context.

You can control variable types and type conversions with `Number(" ")`, `String()`, `Boolean()` and `Object()`.

# Part I

## The built-in environment

### 4 The global object

When Javascript launches, it has a `global` object containing all the values of the properties defining:

- properties like `NaN`, `Infinity` and `undefined`
- functions like `eval()`, `parseInt()`, `parseFloat()`, `isNaN()` and `isFinite()`
- functions to decode and encode URIs from and to utf-8
- constructor functions like `Array()`, `Boolean()`, `Date()`, `Function()`, `Object()`, `Number()`, `RegExp()`, `String()` and `Error()` (section 6.3)
- properties like `Math` and `JSON`.

The global object has no `call` or `construct` property and the values of its *writeable*, *enumerable* and *configurable* attributes are `true`, `false` and `true` respectively. In the top level built-in environment, the keyword `this` refers to the global object; in the HTML document object model the keyword `window` refers to it.

Any object declared to be global becomes part of the `global` object.

#### 4.1 Global functions

**`eval("s")`** evaluates `s` as if Javascript code or returns the value of `s` if it is not.

**`parseInt("s",[b])`** returns an integer derived from `s` with base 10 if `b` is not specified or `NaN`, discarding preceding white space and succeeding non-numeric characters in the string; numeric characters preceded by `0x` or `0X` are interpreted as hexadecimal.

**`parseFloat("s")`** returns a floating point number derived from `s` or `NaN`, discarding preceding white space and succeeding non-numeric characters in the string

**`isNaN("s")`** returns `true` if `s` cannot be interpreted as a number

**`isFinite("s")`** returns `true` if `s` can be interpreted as a number which is not `Infinity` or `NaN`

## 5 Objects

All objects contain name/value pairs of properties each of which has a number of attributes; in arrays the name is always a numeric key. To create a new object which inherits the properties of the built-in object prototype use

```
Object(o);
```

or the constructor function:

```
new Object(o);
```

(section 6.3) which is equivalent to

```
var o = new.Object();
```

To create a new object which inherits the properties of an existing object use `inherit(o)`, for example,

```
var booklet = inherit(book);
```

### 5.1 Object properties

Objects may have data properties if they are not functions or accessor properties if they are. Data properties have the attributes *value*, *writable*, *enumerable* and *configurable*, the first of which

New classes can be created by creating new prototypes with

```
Object.create(o,{p1: {value: v, writable: <true/false>, enumerable: <true/false>,  
configurable: <true/false>}  
{p2: {value: v, writable: <true/false>, enumerable: <true/false>, configurable: <true/false>}  
{...}  
});
```

holds the value of the property. Accessor properties have the attributes *get*, *set*, *enumerable* and *configurable*. The default values of *value*, *get* and *set* are `undefined` and of *writable*, *configurable* and *enumerable* `false` but the *enumerable* attribute is set to `true` if the function contains a `for in` expression.

All objects have the internal properties `prototype` and `class`, `extensible` which hold a Boolean value and the internal methods `get`, `getOwnProperty`, `getProperty`, `put`, `canPut`, `hasProperty`, `delete`, `defaultValue` and `defineOwnProperty` (which has the attributes *length*, *value* (the number of parameters), *writable*, *enumerable* and *configurable* and its own property `false`) though some objects may throw a `TypeError` if the `defaultValue` internal method is called.

The `length` property of a built-in function holds the number of arguments required by the function (1 if it is a constructor function) and its *writable*, *enumerable* and *configurable* properties are all set to `false`. The `length` property of an array holds the number of key/value pairs. Changing the value of `length` has the effect of adding `undefined` key/value pairs or deleting key/value pairs at the end of the array.

All objects other than the `global` object inherit these properties from their object prototype though their precise implementation may vary between different classes of object.

In addition to these internal properties, the `Boolean`, `Date`, `Number` and `String` objects have the internal property `primitiveValue`, functions have the internal properties `call`, `hasInstance`, `scope` (the execution environment), `formalParameters` (the names of parameters) and `code` (the function's ECMAScript code), child functions have the internal properties `targetFunction`, `boundThis` and `boundArguments`, constructor functions (section 6.3) have the internal property `construct`, `RegExp` objects have the internal property `match` and arguments objects have the internal property `parameterMap`.

## 5.2 Accessing object properties

Most properties of an object may be returned or set using the `.` notation or the `[]` notation, for example,

```
o.p;
```

or

```
o["p"];
```

where the property name is a number or a string containing more than one word, for example,

```
function classof(o){
  if (o===null) return "Null";

  if (o===undefined) return "Undefined";
  return Object.prototype.toString.call(o),slice(8,-1);
}
```

In practice, for functions, it is `"Function"`, for arrays it is `"Array"`, for strings it is `"String"`, for Booleans it is `"Boolean"`, for numbers it is `"Number"`, for Math it is `"Math"`, for dates it is `"Date"`, for regular expressions it is `"RegExp"`, for error messages

```
book.topic;
book["main title"];
```

To add or modify the properties of an object, use

```
o.p="v";
```

or

```
o["p"]="v";
```

for example

```
book.author = "Flanagan";
book["main title"]= "Javascript Pocket
Reference";
```

Objects with the same `prototype` value have the same `class` value. There is no direct access to value of the `class` property but it can be returned using (from Flanagan, 2012, p. 91):

it is `"Error"` and for all other objects it is `"Object"`.

You can establish whether an object is an instance of a particular class of object by using the `instanceof` operator (appendix B.3), for example,



```
o instanceof Array;
```

returns `true` if `o` is an array.

### 5.3 Object methods

A method is a function (section 6) which has been assigned to a property of an object; it may be called with:

```
o.<name>();
```

or

```
o[name]();
```

To temporarily call a method, that is, create a new function based on the method and then discard it after its work is done, use:

```
f.call(o, v1, v2, ...);
```

or, where the number of values to be passed as parameters is uncertain, by specifying the parameters as an array,

```
f.apply(o, [<array>]);
```

Note that Javascript does not check that the correct number of parameters have been passed; unless zero arguments have been passed, it simply gives missing parameters the value `undefined`. If more parameters have been passed than are specified, you can use

```
arguments.length
```

to find the number of arguments passed and the extra parameters can then be accessed using `arguments[n]` where  $n$  is the position of the argument in the list starting from 1.

**o.hasOwnProperty(p);** returns `true` if the object has the declared property `p` (inherited properties are ignored).

**o.isPrototypeOf(o);** returns `true` if `o` shares the prototype of the object.

**o.propertyIsEnumerable(p);** returns `true` if the declared property `p` of the object has the attribute *enumerable* (inherited properties are ignored).

**o.toString();** by default returns a string of the form `[<object>,<class>]` but most objects return more specific values.

**o.toLocaleString();** returns the string in a locale sensitive form; this is currently only implemented for `Array`, `Date` and `Number`.

**o.valueOf();** returns an object as defined by a specific implementation, in the case of functions the `Object.prototype` object of that implementation.

## 6 Functions

A function is a callable object; functions may be declared with:

```
function <name>(<parameters>) {  
    <statements>  
};
```

or

```
Function (v1, v2, ... vn, <body>);
```

or

```
new Function (v1, v2, ... vn, <body>);
```

all of which create a new function and a prototype for the function so that it can be used as a constructor (section 6.3).

Function names may commence with a letter, an underscore or a dollar sign.

Functions containing statements may be declared within top levels statements, functions or variable declarations but not within other statements (section 15).

A function may be assigned to

- a variable
- a property of an object, in which case it is known as a *method*.

If the first line of a function is "use strict"; Javascript strict mode is used meaning, among other things, that all variables must be declared, certain expressions which might otherwise be ignored generate a **TypeError** and functions invoked as functions rather than as *methods* have the value of **this undefined**.

<name> may be omitted if the function is assigned to a variable, e.g.

```
var v = function (<parameters>) {  
    return <expression>  
};
```

Such functions may also be declared within a statement.

Variables declared within a function have local scope and take precedence over global variables; all other variables have global scope. This means you can create a library of functions which can be used in any program because their variables will not conflict with any of the global variables in the program. To completely hide the local variables within a function defining an expression, enclose the function in parentheses and call it immediately with:

```
(function (<parameters>) {  
    return <expression>  
})();
```

Combined with a variable declaration

```
var v = (function (<parameters>) {  
    return <expression>  
})();
```

this gives *v* the value returned by the function rather than assigning the function to the variable.

Functions are called with:

```
<name>();
```

and *methods*, that is, functions assigned to an object property, with:

```
o.<name>();
```

or

```
o[name]();
```

## 6.1 Built-in functions

**Boolean(o);** returns the Boolean value of an object

**Number(o);** returns the number value of an object

**Object(o);**

**String(o);** returns the string value of an object

## 6.2 Function methods

Functions have no equivalents to `o.toLocaleString()`; `o.hasOwnProperty(p)`; `o.isPrototypeOf(o)`; or `o.propertyIsEnumerable(p)`; instead, they have

**f.apply(<environment>,a);** which temporarily calls a method which returns the results of applying a particular function to all the values in an array and then discards it after its work is done

**f.bind(<environment>[,v1[,v2,...]]);** which returns a child function which does not have a `prototype` property or the `scope`, `formalParameters` or `code` internal properties. Instead, child functions have the internal properties `targetFunction`, `boundThis` (defining the environment) and `boundArgs` (the values passed). The behaviour of the `call()`, `construct()` and `hasInstance()` internal methods will depend on the properties the child function has inherited.

**f.call(<environment>[,v1[,v2,...]]);** temporarily calls a method which returns the results of applying a particular function to all the values provided for it and then discards it after its work is done; the value of its `length` property is 1 because `v1[,v2,...]` are optional.

**f.toString()** returns the source code of the function

## 6.3 Constructor functions

All functions automatically have a `prototype` property so that they can be used as constructor functions. These conventionally begin with an uppercase character and use the `new` operator to call the `construct` internal method of the function, for example,

```
new Array();
new Boolean();
...
```

in the case of built-in constructor functions. But new constructor functions can be created with

```
function <Name>(<parameters>) {
  <statements>

  return <expression>
};
```

which can be used with the `new` operator.

Both create a new empty object with properties inherited from its prototype or a new object with the values of one or more properties set from the parameters passed to the constructor function.

## 6.4 Object prototypes

The different object prototypes called by a constructor function when asked to create a new object can be accessed through the `prototype` property of the constructor function, for example,

```
Object.prototype;
Array.prototype;
...
```

These normally contain the default inherited properties of an object of a particular type but, as prototypes are objects which can be modified or newly created, with, for example,

```
Object.create(Object.prototype);
```

an object's inherited properties may be different from those of the built-in Javascript prototypes. To access the inherited properties of an object, use:

```
Object.getPrototypeOf(o);
```

To determine whether an object is inherited from another object use:

## 6.6 Constructor function properties

The value of the `prototype` property of a constructor function is the prototype object, for example, `Object.prototype`, `Array.prototype` and the value of the `length` property is 1. You can obtain the value of a property of an object with:

```
Object.getOwnPropertyDescriptor(o,p);
```

```
Object.getOwnPropertyNames(o);
```

returns an array of the names of the declared properties of an object.

Since arrays have property keys rather than property names, the equivalent for an array is

```
Array.isArray(o);
```

which returns `true` if the object is an array.

```
Object.create(o[,p {<attribute>: v, <attribute>: v, ...}]);
```

creates a new object with the specified properties. In the absence of a property list, it uses the properties of the prototype.

To add or update a single declared property of an existing object, use

```
Object.defineProperty(o,p {value: v, writable: <true/false>, enumerable:  
<true/false>, configurable: <true/false>});
```

```
o.isPrototypeOf(x);
```

## 6.5 Object prototype properties

The value of the `prototype` property of every prototype other than the object prototype is `Object.prototype`; the value of the `prototype` property of the object prototype is `null`. The values of their attributes *writable*, *enumerable* and *configurable* are `true`, `false` and `false` respectively and the value of their `extensible` property is `true`. The `constructor` property holds the value of their `constructor`.

To add or update a number of properties in an existing object, use

```
Object.defineProperties (o,{p1: {value: v, writable: <true/false>, enumerable:
<true/false>, configurable: <true/false>}
{p2: {value: v, writable: <true/false>, enumerable: <true/false>, configurable:
<true/false>}
{...}
});
```

Note that, if you assign a value to an inherited property, it simply changes the value of the property; if you declare a new property with the same name as an inherited property, it hides the inherited property.

String constructors have the properties

```
String.fromCharCode (ch1,ch2 ,...) ;
```

which returns the string value of a sequence of characters.

Number constructors have the properties

```
Number.MAX_VALUE;
```

which returns the largest positive finite value of the Number type, which is approximately  $1.7976931348623157 * 10^{308}$ .

```
Number.MIN_VALUE;
```

returns the smallest positive value of the Number type, which is approximately  $5 * 10^{-324}$ .

```
Number.NaN;
```

returns NaN.

```
Number.NEGATIVE_INFINITY;
```

returns  $-\infty$ .

```
Number.POSITIVE_INFINITY;
```

returns  $+\infty$ .

Date constructors have the properties

```
Date.parse(s);
```

which returns a number value representing the time in milliseconds.

```
Date.UTC(year,month[,date[,hours[,minutes[,
seconds[,ms]]]]]);
```

returns a number value representing the time specified in the arguments in UTC milliseconds.

```
Date.now();
```

returns a number value representing the current time in UTC milliseconds.

The **extensible** property of an object is accessible using the constructor function property:

```
Object.isExtensible(o);
```

which returns a Boolean value. To prevent further properties and their values being added to an object, use:

```
Object.preventExtensions(o);
```

To prevent any changes to the existing properties, use:

```
Object.seal(o);
```

and, to prevent any changes at all to the object, use:

```
Object.freeze(o);
```

You can determine whether an object is sealed or frozen with

```
Object.isSealed(o);
```

or

```
Object.isFrozen(o);
```

which return Boolean values but you cannot extend, unseal or unfreeze an object once any have been set.

```
Object.keys(o);
```

returns an array of the enumerable declared properties of an array like object.

---

## 7 Arrays

An array is a particular instance of an object in which the name of each value property is expressed as a number, starting with 0, known as the object key; arrays are untyped and may hold elements of different types — so objects may be contained in arrays as well as containing arrays.

Arrays inherit from the `Array.prototype` and may be declared with a comma separated list of values; if there is no value between two commas, the element is `undefined`; if no values are defined, the array is said to be sparse.

```
Array(v1,v2,...);
```

and

```
new Array(v1,v2,...);
```

both create a new array.

```
new Array(n);
```

creates a new array with  $n$  object keys but their values empty; *ergo*, trying to create an array with a single value will fail — it must either be empty or have two or more values supplied.

To add a further element to or replace an existing element in the array use

```
a[n] = v;
```

where  $n$  is the position of the element counting from 0. To return the value associated with an object key, use

```
a[n];
```

where  $n$  is the position of the element.

Arrays may be contained in objects and objects in arrays. Elements in arrays contained in arrays, thereby forming a matrix, can be declared with:

```
a[y][x] = v1,v2,...;
```

## 7.1 Array properties

The `extensible` property of an array to set to `true` as are its attributes `writable`, `enumerable` and `configurable` and these attributes of the `length` property. Note that arrays use a variant of the `defineOwnProperty` internal method used by other objects.

**a.length** returns, or sets, the number of elements in an array; so

```
a.length = 7;
```

deletes any elements after number 6 or adds `undefined` elements if the length is less than 7.

## 7.2 Array methods

**a.concat()** concatenates arrays, or other objects holding arrays, into a new array.

**a.indexOf(v[,n])** returns the number of the first element in an array or array like object which matches `v`, or `-1` if not found, starting from the first element or element `n` if specified.

**a.join()** returns the elements of an array, or any object holding array like elements, in CSV<sup>1</sup> (the default); an alternative separator, for example, `a.join(" ")` or `a.join("-")` may be specified or no separator specified, for example, `a.join("")`, which returns a concatenated string.

---

<sup>1</sup>The opposite of `s.split(",")`.

**a.lastIndexOf(v[, -n])** returns the number of the first element in an array or array like object which matches `v`, or `-1` if not found, starting from the last element or element `-n` if specified.

**a.reverse()** reverses the order of the elements an array or an array like object.

**a.slice(n[,e])** returns a new array starting from element `n` of an array or array like object and going up to but not including element `e` or the end if `e` is unspecified, counting from the last element and going up to the start if `n` or `e` are negative.

**a.sort([x,y])** returns an array created from an array or an array like object in which the elements have been sorted with respect to whether `x` exceeds `y` or not; where `x` or `y` are `undefined`, returns an array by ASCII values; in this case, change alphabetic values to upper or lower case before sorting.

**a.splice(s[,n,v1,v2,...])** returns an array of *the n elements starting from s deleted from an array, or an array like object* where `v1,v2,...` have been inserted at `s` in the source array or array like object.

**a.toString()** returns an array in CSV removing all other delimiters within the array.

### 7.2.1 Stack manipulation of arrays

**a.pop()** returns the value of the last element in an array or array like object.

**a.push(v1,v2,...)** adds successive elements to the end of an array or array like object.

**a.shift()** returns the value of the first element in an array or array like object.

**a.unshift(v1,v2,...)** adds successive elements to the start of an array or array like object.

### 7.2.2 Array methods for functions

**a.every(f[,<environment>])** iterates through an array executing the same criterion specifying function and returns **true** if all elements satisfy the criterion or if the array is empty.

**a.filter(f[,<environment>])** iterates through an array or an array like object executing the same criterion specifying function and returns an array of those elements which satisfy the criterion.

**a.forEach(f[,<environment>])** iterates through an array executing the same function on each element.

**a.map(f[,<environment>])** iterates through an array or an array like object executing the same function on each element and returning a new array.

**a.reduce(f[,n])** iterates through an array or array like object (starting with a key *n* or the first key in the array) executing a combining function on each element and returning the result of the combining function.

**a.reduceRight(f[,n])** iterates through an array in reverse (starting with a key *n* or the last key in the array) executing a combining function on each element and returning the result of the combining function.

**a.some(f[,<environment>])** iterates through an array or an array like object executing the same criterion specifying function and returns **false** if no elements satisfy the criterion or if the array is empty.

---

## 8 Strings

Strings only contain unsigned 16-bit units of UTF-16 text. Strings must be quoted within a pair of single or double quotes; inner strings must have the opposite quote style to the outer string.

```
String("<string>");
```

returns the value of the string, rather than creating a string object. A **String** object may be created with

```
new String("<string>");
```

whose **primitiveValue** internal property holds the value of the string.

The value of the **length** property of a string prototype is 0 and the other properties of a string prototype can be accessed in the same ways as an object (section 6.5).

**s.length** returns the length of **s**

and strings may be treated as character arrays so that

**s[n]** returns the character at position *n* counting from 0.

Strings may include the following escaped characters:



<code>\0</code> null	<code>\r</code> CR
<code>\b</code> backspace	<code>\"</code> double quote
<code>\t</code> tab	<code>\'</code> single quote
<code>\n</code> newline	<code>\\</code> backslash
<code>\v</code> vertical tab	<code>\x xx</code> Latin-1 character
<code>\f</code> FF	<code>\u xxxx</code> Unicode character

Strings may be concatenated with the `+` operator and the value of any variable other than `null` and `undefined` may be converted to a string with `toString()`.

## 8.1 String methods

**s.charAt(*n*)** returns the *n*th character in a string or a string like object counting from 0 or an empty string if the *n*th character does not exist

**s.charCodeAt(*n*)** returns the ASCII value of the *n*th character in a string or a string like object counting from 0 or NaN if the *n*th character does not exist

**s.concat(*s1,s2[,s3,...]*)** returns a string consisting of *s1* followed by *s2* [followed by *s3 ...*]

**s.indexOf(*s[,n]*)** returns the character position of the first character of the first occurrence of the substring *s*, starting at character *n* or the start of the string, within a string or string like object or `-1` if not found

**s.lastIndexOf(*s[,n]*)** returns the character position of the first character of the last occurrence of the substring *s*, ending at character *n* or the end of the string, within a string or string like object or `-1` if not found

**s.localeCompare(*s2*)** returns an integer expressed as a string value representing the difference in Unicode values between two strings in a string or string like object; in other words, 0 means they are identical

**s.match(*r*)** returns an array containing the match (or matches if the `global` flag is set) of a regular expression in a string or string like object

**s.replace("<substring>",*s/f*)** if the second parameter is a string, returns a string with all instances of `<substring>` in a string or string like object replaced with *s* (use `s.match(r)` with regular expressions); if the second parameter is a function, returns the string created as a result of evaluating the function with at least three parameters: the substring to be matched, any offset and the substring to replace it; this method can be refined by using

**\$&** for the replacement text

**\$'** for whatever precedes the matched substring

**\$'** for whatever follows the matched substring

**\$n** for the *n*th occurrence up to 9

**\$nn** for the 10th to 99th occurrence

**\$\$** for `$`

**s.search(*r*)** returns the character position in a string or string like object of a regular expression at the first match or `-1` if no match is returned

**s:slice(*s[,e]*)** returns a string value starting from character *s* of a string or string like object and going up to but not including element *e* or the end if *e* is unspecified, counting from the last element and going up to the start if *s* and *e* are negative.

**s.split([<separator>[,n]])** returns an array of up to *n* elements containing the string or string like object split wherever the separator appeared in it; if there is no separator, the array consists of the individual characters in the string; **s.split(",")** reverses the result of **a.join()** returning an array containing the string split wherever a comma appeared in it; it may be used with regular expressions

**s.substring(*n*[,*e*])** returns a substring starting at character *n* and going up to but not including character *e* from a string or string like object

**s.toLowerCase()** returns a new UTF-16 string using only lower case characters, not necessarily the reverse of **s.toUpperCase()**

**s.toLocaleLowerCase()** returns a new UTF-16 string using only lower case characters (for languages where conversions to lower case are not straightforward)

**s.toUpperCase()** returns a new UTF-16 string using only upper case characters, not necessarily the reverse of **s.toLowerCase()**

**s.toLocaleUpperCase()** returns a new UTF-16 string using only upper case characters (for languages where conversions to lower case are not straightforward)

**s.trim()** returns a string value with leading and trailing white space removed

## 9 Boolean objects

```
Boolean(v);
```

returns the value of the Boolean, rather than creating a `Boolean` object. A `Boolean` object may be created with

```
new Boolean(v);
```

whose `primitiveValue` internal property holds the Boolean value. The following take the Boolean value `false`:

```
undefined // the value of an empty variable  
null // an object with no values  
0  
-0  
NaN  
"" // an empty string
```

## 10 Number and math objects

`Number(n)`;

returns the value of the number, rather than creating a `Number` object. A `Number` object may be created with

`new Number(n)`;

whose `primitiveValue` internal property holds the number.

Numbers are stored as 64-bit floating point numbers; they may be the integers from  $-2^{31}$  to  $2^{31}-1$  or 0 to  $2^{32}-1$ , `NaN`, `+Infinity`, `-Infinity`, `+0` and `-0`.<sup>2</sup>

`Infinity` is a valid result of a maths operation. Dividing 0 by 0 or `Infinity` by `Infinity` produces `NaN`. To test for `NaN`, use either `x!=x`; or `isNaN(x)`; which return `true` if `x=NaN`.

`isFinite(n)`;

returns `true` if `n` is not `NaN`, `Infinity` or `-Infinity`.

Do not precede base-10 numbers with 0 and note that, because they are represented in binary, there are no exact matches for many decimal numbers making comparisons of raw numbers difficult. Numbers preceded by `0x` or `0X` are interpreted as hexadecimal. `n.n...en` or `n.n...En` is interpreted as the exponential  $n.n... \times 10^n$ .

### 10.1 Number object methods

**`n.toExponential(n)`** returns a string in exponential format with one integer and `n` decimal places (up to 20) derived from a number

**`n.toFixed(n)`** returns a string of `n` decimal places derived from a number

---

<sup>2</sup>However, `NaN` and `Infinity` cannot be represented in a floating point number and therefore cannot be used in HTML.

**`n.toLocaleString()`** converts a number to a string expressed in a locale dependent format determined by the host environment

**`n.toPrecision(n)`** returns a string of `n` significant figures (up to 21) derived from a number; it may be returned in fixed or exponential format in which case it is equivalent to `n.toExponential(n - 1)`; if `n` is undefined, `n.toString()` is substituted

**`n.toString(b)`** converts a number to a string expressing its base `b` value; the default is 10 and the numbers from 10 to 35 are expressed as `a` to `z`

### 10.2 Math objects

`Math` objects do not have the `call` or `construct` internal properties; so they do not have a constructor function, the value of their `prototype` property being the object prototype.

#### 10.2.1 Math object properties

**`Math.E`**  $\varepsilon$

**`Math.LN2`** natural log of 2

**`Math.LN10`** natural log of 10

**`Math.LOG2E`** log to base 2 of  $\varepsilon$

**`Math.LOG10E`** log to base 10 of  $\varepsilon$

**`Math.PI`**  $\pi$

**Math.SQRT1\_2**  $\sqrt{\frac{1}{2}}$

**Math.SQRT2**  $\sqrt{2}$

### 10.2.2 Math object methods

**Math.abs(*n*)** returns the absolute value of *n*

**Math.acos(*n*)** returns the arc cosine of *n* expressed in radians

**Math.asin(*n*)** returns the arc sine of *n* expressed in radians

**Math.atan(*n*)** returns the arc tangent of *n* expressed in radians

**Math.atan2(*y,x*)** returns the arc tangent between the *x* axis and *y*, counter-clockwise expressed in radians

**Math.ceil(*n*)** returns the smallest integer greater than or equal to *n*

**Math.cos(*n*)** returns the cosine of *n* expressed in radians

**Math.exp(*n*)** returns  $\varepsilon^n$

**Math.floor(*n*)** returns the greatest integer less than or equal to *n*

**Math.log(*n*)** returns the natural log of  $n^3$

**Math.max(*v1,v2[,v3...]*)** returns the highest value among *v1,v2[,v3...]*

**Math.min(*v1,v2[,v3...]*)** returns the lowest value among *v1,v2[,v3...]*

**Math.pow(*x,y*)** returns  $x^y$

**Math.random()** generates a random number between 0 and 1

**Math.round(*n*)** rounds to the nearest integer (upwards if *n* is equidistant from two integers, so  $-3.5$  returns  $-3$ )

**Math.sin(*n*)** returns the sine of *n* expressed in radians

**Math.sqrt(*n*)** returns  $\sqrt[n]{n}$

**Math.tan(*n*)** returns the tangent of *n* expressed in radians

---

## 11 Date objects

**Date** objects contain a number expressing milliseconds backwards or forwards from midnight (00.00) on 1 January 1970 UTC (= +0). Years, months and days are calculated from this using the Gregorian system and days are assumed to have 86,400,000 milliseconds, that is, leap seconds are ignored. Months are expressed by the integers 0 to 11 and days by the integers 1 to 31.

A host environment is expected to implement a Local Time

Zone Adjustment expressed in milliseconds and a Daylight Saving Adjustment where appropriate.

Date strings take the form:

YYYY-MM-DDTHH:mm:ss.sssZ

Where

---

<sup>3</sup>To calculate logs to base 10, use  $\frac{\text{Math.log}(x)}{\text{Math.LN10}}$

**YYYY** is the decimal digits of the year 0000 to 9999 in the Gregorian calendar; note that year 0000 in the Gregorian calendar is 1 BC as there was no year 0; years outside this range may be specified as six digit years prefixed by + or - but may only be used between environments which recognise six digit years; note that, as there was no year 0 in the Gregorian calendar, the year -000001 is 2 BC and 2000 BC is the year -001999.

**MM** is the month of the year from 01 (January) to 12 (December).

**DD** is the day of the month from 01 to 31.

**T** indicates the beginning of the time element.

**HH** is the number of complete hours that have passed since midnight as two decimal digits from 00 to 24.

A `Date` object may be created with

```
new Date();
```

or

```
new Date(n);
```

or

```
new Date(year[,month[,date[,hours[,minutes[,seconds[,ms ]]]]]]);
```

The first creates an object whose `primitiveValue` internal property holds the UTC value current time, the second an object whose `primitiveValue` internal property holds the value *n* milliseconds, the third an object whose `primitiveValue` internal property holds the UTC value of the time specified in the argu-

**mm** is the number of complete minutes since the start of the hour as two decimal digits from 00 to 59.

**ss** is the number of complete seconds since the start of the minute as two decimal digits from 00 to 59.

**sss** is the number of complete milliseconds since the start of the second as three decimal digits.

**Z** is the time zone offset specified as "Z" (for UTC) or either "+" or "-" followed by a time expression `HH:mm`

Note that `T24.00` and `T00.00` on successive days refer to the same time.

```
Date();
```

returns a string representing the current UTC time.

ments.

The value of the `length` property of a date prototype is 0 and the other properties can be accessed in the same ways as an object (section 6.5).

## 11.1 Date object methods

**d.getDate();** returns the local format date of a date

**d.getUTCDate();** returns the UTC date of a date

**d.getDay();** returns the local format day of the week of a date

**d.getUTCDay();** returns the UTC day of the week of a date

**d.getFullYear();** returns the local format year of a date

**d.getUTCFullYear();** returns the UTC year of a date

**d.getHours();** returns the local format hours of a date

**d.getUTCHours();** returns the UTC hours of a date

**d.getMinutes();** returns the local format minutes of a date

**d.getUTCMinutes();** returns the UTC minutes of a date

**d.getMonth();** returns the local format month of a date

**d.getUTCMonth();** returns the UTC month of a date

**d.getSeconds();** returns the local format seconds of a date

**d.getUTCSeconds();** returns the UTC seconds of a date

**d.getMilliseconds();** returns the local format milliseconds of a date

**d.getUTCMilliseconds();** returns the UTC milliseconds of a date

**d.getTime();** returns the time value of a date

**d.getTimezoneOffset();** returns the timezone offset in minutes of a date

**d.setDate(<date>);** sets the value of the `primitiveValue` property of a `Date` object to `<date>` after adjusting for local time

**d.setUTCDate(<date>);** sets the value of the `primitiveValue` property of a `Date` object to `<date>`

**d.setFullYear(year[,month[,date]]);** sets the value of the `primitiveValue` property of a `Date` object to that represented by the arguments adjusting for local time

**d.setUTCFullYear(year[,month[,date]]);** sets the value of the `primitiveValue` property of a `Date` object to that represented by the arguments

**d.setHours(hour[,min[,sec[,ms]]]);** sets the value of the `primitiveValue` property of a `Date` object to that represented by the arguments after adjusting for local time

**d.setUTCHours(hour[,min[,sec[,ms]]]);** sets the value of the `primitiveValue` property of a `Date` object to that represented by the arguments

**d.setMonth(month[,date]);** sets the value of the `primitiveValue` property of a `Date` object to that represented by the arguments adjusting for local time

**d.setUTCMonth(month[,date]);** sets the value of the `primitiveValue` property of a `Date` object to that represented by the arguments

**d.setMinutes(min[,sec[,ms]]);** sets the value of the `primitiveValue` property of a `Date` object to that represented by the arguments after adjusting for local time

**d.setUTCMinutes(min[,sec[,ms]]);** sets the value of the `primitiveValue` property of a `Date` object to that represented by the arguments

**d.setSeconds(sec[,ms]);** sets the value of the `primitiveValue` property of a `Date` object to that represented by the arguments after adjusting for local time

**d.setUTCSeconds(sec[,ms]);** sets the value of the `primitiveValue` property of a `Date` object to that represented by the arguments

**d.setMilliseconds(n);** sets the value of the `primitiveValue` property of a `Date` object to `n` milliseconds after adjusting for local time

**d.setUTCMilliseconds(n);** sets the value of the `primitiveValue` property of a `Date` object to `n` milliseconds

**d.setTime(<time>);** sets the value of the `primitiveValue` property of a `Date` object to `<time>`

**d.toString();** returns the date in a human readable string form

**d.toLocaleDateString();** returns the local date in a human readable string form

**d.toISOString();** returns the UTC date in the human readable string form `YYYY-MM-DDTHH:mm:ss.sssZ`

**d.toUTCString();** returns the UTC date in a human readable string form, normally omitting the `T`

**d.toJSON(k);** returns a string representation of the date for use with `JSON.stringify` (section 14); note that this method can also be used with objects other than `Date` objects

**d.toString();** returns the date and time in a human readable string form

**d.toLocaleString();** returns the local date and time in a human readable string form

**d.getTimeString();** returns the time in a human readable string form

**d.toLocaleTimeString();** returns the local time in a human readable string form

**d.valueOf();** returns a number representing the value of a date

## 12 Regular expressions

A regular expression object may be created using

```
RegExp(<pattern>,<flags>);
```

or

```
new RegExp(<pattern>,<flags>);
```

Both use the function prototype object as their constructor and so the `RegExp` prototype object does not have a `valueOf` property (section 5.3).

Regular expressions have a `source` property containing the text of the regular expression, a `lastIndex` property containing the position of the last match which initially has the value 0 and three Boolean properties each of whose value is set to `true` if the cor-

responding flag is set:

**g** all matches (the `global` property)

**i** case insensitive (the `ignoreCase` property)

**m** include matches that cross line boundaries (the `multiline` property).

Regular expression patterns can use all the escape characters (section 8) except backspace plus any control character, for example

`\cJ` control-J

The regular expression operators are:

`.` any character other than a line ending

`-` range, for example `a-z`

`^` any characters after

`$` any characters before

`[ ]` any character within the brackets

`[^ ]` any character other than those within the brackets

`\b` a word boundary (i.e. between an ASCII word character and a non-ASCII word character)

`\B` a boundary that is not a word boundary

`[\b]` backspace

`\d` any ASCII digit

`\D` any character other than an ASCII digit

`\s` any Unicode whitespace

`\S` any character other than Unicode whitespace

`\w` any ASCII word character (`=[a-zA-Z0-9]`)

`\W` any character other than an ASCII word character

`(?=<p>)` a pattern with pattern `<p>` following

`(?!<p>)` a pattern where pattern `<p>` does not follow

Searches may be elaborated with

`|` OR

`( )` group elements or enclose an optional element in the pattern

`(?: )` group elements but ignore them in the search

`\n` the *n*th group in the pattern

Searches may be limited with

`{n}` the first *n* occurrences, for example `{\d{4}}` four digits in a row

`{n,}` at least *n* occurrences

`{n,m}` at least *n* but not more than *m* occurrences

`*` = `{0,}`

`+` = `{1,}`

`?` = `{0,1}` i.e. does not match each repeated occurrence of a pattern



## 12.1 Regular expression methods

**r.exec(s)** returns an array containing the results of matching `s` against the regular expression plus matches of any parenthesised patterns and sets the properties `index` and `lastIndex` to the position of the match and the position of the last match respectively, enabling it to undertake re-

peated searches starting from `lastIndex` or returns `null` if there were no matches and sets `lastIndex` to 0

**r.test(s)** returns `true` if there is a match between `s` and the regular expression

**r.toString()** returns a string value containing the regular expression and the flags concatenated

---

## 13 Error objects

```
Error("<message>");
```

and

```
new Error("<message>");
```

both create an `Error` object whose `message` internal property holds the value of the message and whose `name` internal property is initialised as `"Error"`.

**e.toString()** returns the value of the `message` property or, if that is `undefined`, the value of the `name` property.

Whenever a runtime error occurs, one of the following native `Error` objects is called: `EvalError`, `RangeError`, `ReferenceError`, `SyntaxError`, `TypeError`, `URIError`. Their prototypes are the error object prototype and they do not have an `e.toString()` method.

---

## 14 JSON

A `JSON` object has two functions:

**JSON.parse(t[,k,v])** which converts a `JSON` string into a JavaScript object substituting `v` wherever it encounters `k` and `v` as a key/value pair in the `JSON` string

**JSON.stringify(<object>[,<replacer>[,<space>]])** which converts an object's properties into a string, optionally modified by a function or selected according the properties of a

number or string array specified as the `<replacer>` and/or with additional space to improve human readability

It does not have a `construct` or a `call` property and so cannot be used as a constructor or as a function.

Note that

- a cyclic structure will throw a `TypeError`

- a JSON string is a comma separated list of properties enclosed in braces
- properties consist of property names wrapped in double quotes followed by a colon followed by their value
- strings are wrapped in double quotes
- numbers are represented as strings
- NaN and Infinity are represented as null in number strings
- arrays consist of a list of comma separated values enclosed in brackets
- functions and undefined values are represented as null in arrays but are not otherwise represented in JSON strings.

## 15 Statements

Statements are executed and return a completion value. Statements conclude with a semi-colon; *ergo* more than one statement may be on a line. Where a semi-colon has been omitted, Javascript will interpolate one wherever it allows successive statements to be parsed but this can lead to unexpected results.

Comments within a statement are not executed if they are preceded by // and appended to a line; otherwise, they must be enclosed in /\* ... \*/ in which case new lines must begin with \*.

A statement may have a label which can be used to call it from elsewhere in the Javascript code, for example

```
<label>: {<(list of )statement(s)>};
```

Labels may commence with a letter, an underscore or a dollar sign.

If the first line of a statement is `use strict`, Javascript strict mode is used meaning, among other things, that all variables must be declared and certain expressions which might otherwise be ignored generate a `TypeError`.

### 15.1 Declaring variables

Variables may be declared without an assigned value but it is good practice always to declare variables as this aids debugging, for example

```
var v = v;
```

declares a variable `v` and assigns the value `v` to it (use "" to enclose string values).

```
var v = prompt(" ");
```

opens a prompt box and assigns any value entered in it to `v`.

Variables may commence with a letter, an underscore or a dollar sign.

A declaration using an existing variable name is treated as an assignment to the existing variable.

Numbers, strings, Boolean values, property/value pairs, arrays or functions may be assigned to variables. For example, an object may be declared with:

```
var <name> = {
```

```

    <property>: v,
    <property>: v,
    ...
};

```

as in

```

var book = {
    topic: "JavaScript",
    fat: true
};

```

You can also use constructor functions (section 6.3), for example

```

var book = new Object {
    topic: "JavaScript",
    fat: true
};

```

A variable can take a function as its value, for example

```

var <name> = <function> (<parameters>) {
    <statement(s)>
};

```

Variables declared within a function or a `for` statement have local scope; all others have global scope.

### 15.3 Flow control

```

do{<statement>
} while(condition expression);

```

### 15.2 Conditionals

```

if(condition expression){
    return <result of statement>;
}
if(condition expression){
    return <result of statement>;
else {
    return <result of statement>;
}
}

```

if statements may be nested in `else` statements producing "elseif" switches or `switch` may be used, for example

```

switch(<expression>) {
    case <expression>: <statement(s)>; break;
    case <expression>: <statement(s)>; break;
    case <expression>: <statement(s)>; break;
    ...
    default: <statement(s)>; break;
}

```

`<expression>` may be a number or a string.

```

while(condition expression){
<statement>
};

for(<initialisation>;<test>;<increment>){
<statement>
}:

for(var <declaration>;<expression>;<expression>){
<statement>
};

for(<expression> in <expression>){
<statement>
};

for(var <declaration> in <expression>){
<statement>
};

```

Note that

- variables declared within a for statement have local scope
- **for in** enumerates variables according to the order of creation of the objects unless the object is an array in which case it uses the row/column order of the array.

With all statements, you can use

```
return;
```

to end the execution of the statement or

```
return <expression> ;
```

to end the execution of the statement after returning the value of the expression.

With labelled statements, you can also use:

```
continue <label>;
```

or

```
break <label>;
```

to jump to the labelled statement even from within a loop. Each can be used without a label if there is an implicit label within the loop.

## 15.4 Exception and error handling

```
throw <expression>;
```

tells the user that an error has occurred.

```
try{ <statement(s)>
};
[catch(<error code>){ <statement(s)>
};]
```

executes if `try` throws an exception identifiable by an error code (which may not be `eval` or `arguments` in strict mode).

```
[finally{
<statement(s)>
};]
```

if present, always executes after a `try` statement where there are no `catch` statements or if the `catch` statements fail.

```
debugger;
```

calls any debugging software which may be available to monitor the code.

# Part II

## The host environment

### 16 Host environment objects

The host environment must provide at least the following objects (Hickson, 2013, 2014):

- `ApplicationCache` (section [35 on page 82](#))
- `BarProp` containing a Boolean attribute `visible` which is `true` if a `locationbar`/`menubar`/`personalbar`/`scrollbar`/`statusbar`/`toolbar` is visible<sup>4</sup>
- canvas objects (section [24 on page 47](#))
- collection objects (section [21 on page 38](#))
- data objects such as `File`, `Blob`, `FileList` and `ImageData`
- data transfer objects such as `DataTransfer`, `DataTransferItem` and `DataTransferItemList`
- `Date` (section [11 on page 20](#))
- `Document` (section [18 on page 35](#)) along with
  - `DocumentFragment` to hold the contents of `<template>` elements
  - `DOMElementMap` representing the current CSS element reference identifiers; has methods for getting, setting and deleting name-value pairs
- `DragEvent` (section [31 on page 74](#))

---

<sup>4</sup>Being considered for removal (July 2013)

- `HTMLElement` objects for each HTML element (sections [22 on page 41](#), [25 on page 57](#), [24 on page 47](#), [26 on page 65](#) and [27 on page 67](#)); other element related objects include
  - `DOMString` object whose `primitiveValue` property holds a string of 16 bit unsigned integers
  - `DOMStringMap` object for the element's `data-*` attributes; has methods for getting, setting and deleting name-value pairs
  - `DOMSettableTokenList`
  - `DOMTokenList`
- event objects, such as `MouseEvent`, (section [32 on page 76](#))
- form element objects (section [27 on page 67](#)) including
  - `FileList` object listing the selected files of the form control
  - `HTMLOptionElement`
  - `HTMLSelectElement`
  - `RadioNodeList`
- `Function` (section [6 on page 10](#))
- `Geolocation`
- `History` (section [28 on page 69](#))

- image objects such as
    - `CanvasImageSource`
    - `HTMLCanvasElement`
    - `ImageBitmap` which may have been obtained from an `<img>` , `<video>` or `<canvas>` element or a `Blob`, `ImageData`, `CanvasRenderingContext2D` or `ImageBitmap` object
  - `Location` (section [29 on page 70](#))
  - media element objects (sections [25 on page 57](#), [25.3 on page 61](#), [25.4 on page 63](#), [25.5 on page 64](#) and [25.6 on page 64](#)) along with objects such as
    - `MediaController`
    - `MediaError`
    - `TimeRanges`
  - `Navigator` (section [30 on page 71](#))
  - `NodeList`
    - `PropertyNodeList` containing a list of nodes which match a specific property in an `HTMLPropertiesCollection` (section [21 on page 38](#)) array object
  - `Screen` the CSS screen object
  - `Storage` (section [38 on page 86](#))
  - `SVGMatrix`
  - transferable objects such as:
    - `ArrayBuffer` and `ArrayBufferView`
    - `CanvasProxy`
    - `MessagePort`
  - `ValidityState` (section [34 on page 81](#))
  - `WebSocket` (section [37 on page 85](#))
  - `Window`
    - `WindowProxy`
    - `WindowTimers`
  - webworker objects (section [36 on page 84](#)).
- 

## 17 The window object

Each open window, tab or `<iframe>` element in all the browsers open on a device has its own `Window` object; this is the global object in client side Javascript. While those open in windows or tabs are independent of each other, those opened in `<iframe>` elements are not.

### 17.1 Window object properties

`w[n]` returns the *n*th child browsing context

`w[name]` returns the named window; prefer `d.getElementById()`; see page [37](#)

**w.applicationCache**

**w.closed** returns `true` if the window has been closed, `false` otherwise.

**w.contentWindow** returns the value of the name of the `<iframe>`'s `Window` object — the same as the value of the `name=" "` attribute, if any, of the `<iframe>` element

**w.dialogArguments** returns the argument that was passed to the `showModalDialog()` method (see Page 34); NB: the `showModalDialog()` method is being obsoleted

**w.document** returns the `Document` object associated with the `Window` object

**w.frameElement** returns the value of the `<iframe>`'s element object or `null` if there isn't one

**w.frames** returns the current `Window` object of a child frame

**w.id** returns the value of the first `id=""` value in an HTML element (use of this property is deprecated)

**w.innerHeight** returns the height in pixels of the viewport

**w.innerWidth** returns the width in pixels of the viewport

**w.length** returns the number of child browsing contexts.

**w.localStorage** returns the `Storage` object associated with the window's local storage

**w.locationbar.visible** returns `true` if the location bar is visible; otherwise, returns `false`

**w.menubar.visible** returns `true` if the menu bar is visible; otherwise, returns `false`

**w.name[=*v*]** returns or sets the name of the `Window` object

**w.pageXOffset** returns the amount in pixels by which the document has scrolled horizontally

**w.pageYOffset** returns the amount in pixels by which the document has scrolled vertically

**w.parent** returns the `WindowProxy` for the parent browsing context; the same as `self` in a top-level window, the parent `Window` object of an `<iframe>` element

**w.personalbar.visible** returns `true` if the personal bar is visible; otherwise, returns `false`

**w.returnValue[=*v*]** returns or sets the current return value for the `Window` object that will be returned by the `showModalDialog()` method (see Page 34); NB: the `showModalDialog()` method is being obsoleted

**w.screen** refers to the `Screen` CSS object which has the properties:

**availHeight** the height of the screen excluding panels

**availWidth** the width of the screen excluding panels

**height** the total height of the screen

**width** the total width of the screen

**w.scrollbars.visible** returns `true` if the scroll bars are visible; otherwise, returns `false`

**w.self** the same as the `window` property

**w.sessionStorage** returns the `Storage` object associated with the window's session storage

**w.statusbar.visible** returns `true` if the status bar is visible; otherwise, returns `false`



**w.toolbar.visible** returns `true` if the toolbar is visible; otherwise, returns `false`

**w.top** returns the `WindowProxy` for the top-level browsing context; the same as `self` in a top-level window, the top-level parent `Window` object of an `<iframe>` element

**w.window** the current `Window` object

Each `<iframe>` element has the properties

## 17.2 Window object methods

**w.alert(<message>);** displays the given message and waits for the user to dismiss it; calls the `navigator.yieldForStorageUpdates()` method (see page 74)

**<result>=w.atob(data);** takes a Unicode string containing base64-encoded binary data, decodes it, and returns a string consisting of characters in the range `U+0000` to `U+00FF`, each representing a binary byte with values `0x00` to `0xFF` respectively, corresponding to that binary data

**w.blur();** unfocuses the window; use of this method is discouraged; allow the user to control window focus instead

**<result>=w.btoa(data);** takes a Unicode string containing only characters in the range `U+0000` to `U+00FF`, each representing a binary byte with values `0x00` to `0xFF` respectively, and converts it to its base64 representation, which it returns

**w.clearInterval(h);** cancels the `timeout` set with `setInterval()` identified by `h` (see page 34)

**w.clearTimeout(h);** cancels the `timeout` set with `setTimeout()` identified by `h` (see page 34)

**w.close();** closes the window.

**<result>=w.confirm(<message>);** displays an OK/Cancel prompt with the given message, waits for the user to dismiss it, and returns `true` if the user clicks OK and `false` if the user clicks Cancel; calls the `navigator.yieldForStorageUpdates()` method (see page 74)

**<promise>=w.createImageBitmap(image[,sx,sy,sw,sh]);** takes an image, which can be an `<img>` element, `<video>`, or `<canvas>` element, a `Blob` object, an `ImageData` object, a `CanvasRenderingContext2D` object or another `ImageBitmap` object, and returns a `Promise` that is resolved when a new `ImageBitmap` is created; if no `ImageBitmap` object can be constructed, the promise is rejected; if `sx`, `sy`, `sw`, and `sh` arguments are provided, the source image is cropped to the given pixels, with any pixels missing in the original replaced by transparent black; using the source image's pixel coordinate space, not CSS pixels

**w.external.AddSearchProvider(url);** adds the search engine described by the OpenSearch description document at `url`; the OpenSearch description document has to be on the same server as the script that calls this method

**<installed>=w.external.IsSearchProviderInstalled(url);** returns a value based on comparing `url` to the URLs of the results pages of the installed search engines using a prefix match; only results pages on the same domain as the script that calls this method are checked:

- 0** None of the installed search engines match `url`
- 1** One or more installed search engines match `url`, but none are the user's default search engine

2 The user's default search engine matches `url`

**w.focus()**; focuses the window; use of this method is discouraged; allow the user to control window focus instead

**<window>=w.open([url[,target[,features[,replace]]])**; opens a window to show `url` (defaults to `about:blank`), and returns it:

- `target` argument gives the name of the new window; if a window exists with that name already, it is reused
- `replace`, if `true`, means that whatever page is currently open in that window will be removed from the window's session history
- `features` argument can be used to influence the rendering of the new window

**w.postMessage(<message>,targetOrigin[,transfer])**; posts a message to the given window; messages can be structured objects, for example, nested objects and arrays, can contain JavaScript values (strings, numbers, Dates, etc), and can contain certain data objects such as `File`, `Blob`, `FileList`, and `ArrayBuffer` objects; objects listed in `transfer` are transferred, not just cloned, meaning that they are no longer usable on the sending side; if the origin of the target window doesn't match the given origin, the message is discarded, to avoid information leakage; to send the message to the target regardless of origin, set the target origin to `*`; to restrict the message to same-origin targets only, without needing explicitly to state the origin, set the target origin to `/`.

**w.print()**; prompts the user to print the page; calls the `navigator.yieldForStorageUpdates()` method (see page 74)

**<result>=w.prompt(<message>[,default])**; displays a prompt with the given `message`, waits for the user to dismiss it, and returns the value that the user entered; if

the user cancels the prompt, then returns `null` instead; if `default` is present, this is used as a default; calls the `navigator.yieldForStorageUpdates()` method (see page 74)

**w.scrollBy(n,m)**; adds the values `n` and `m` to the current scrollbar offsets

**w.scrollTo(x,y)**; scrolls the current window so that the value of `left` is `x` and the value of `top` is `y`

**w.setInterval(f,n)**; runs a function every `n` milliseconds

**h=w.setInterval(code[,timeout])**; schedules a `timeout` to compile and run code every `timeout` milliseconds.

**h=w.setInterval(handler[,timeout[,arguments...]])**; schedules a `timeout` to run `handler` every `timeout` milliseconds; arguments are passed straight through to `handler`

**w.setTimeout(f,n)**; runs a function after `n` milliseconds, or asap if `n = 0`

**h=w.setTimeout(code[,timeout])**; schedules a `timeout` to compile and run code after `timeout` milliseconds.

**h=w.setTimeout(handler[,timeout[,arguments...]])**; schedules a `timeout` to run `handler` after `timeout` milliseconds; arguments are passed straight through to `handler`

**<result>=w.showModalDialog(url[,argument])**; prompts the user with the given page, waits for that page to close, and returns the return value; calls the `navigator.yieldForStorageUpdates()` method (see page 74); NB: the `showModalDialog()` method is being obsolete

**w.stop()**; cancels the document load

## 18 The document node

The `Document` object is central to the Document Object Model API, a tree structure for representing the elements in HTML; it can be accessed using the `Document` object.

Each HTML document has a `documentType` node which is the parent of all the nodes in the document each of which has an associated object. A `NodeList` is a read-only array like object. The following properties of the `documentType` node are inherited by all nodes in a document:

### 18.1 Document node properties

**d.childNodes** a `NodeList` of child nodes

**d.firstChild** the first node or `null` if the node has no children

**d.lastChild** the last node or `null` if the node has no children

**d.localName**

**d.namespaceURI**

**d.nextSibling** the next sibling node, in the order in which they appear in the document

**d.nodeName** the tag value of an HTML element in uppercase

**d.nodeType** holds the values

1 for an `element` node

3 for a `text` node

8 for a `comment` node

9 for a `document` node

**d.nodeValue** the textual content of a `text` (section 23) or `comment` node

**d.ownerDocument**

**d.parentNode** holds the parent node or `null` if it is the `Document` object

**d.previousSibling** the previous sibling node, in the order in which they appear in the document

*Unnecessary whitespace in an HTML document can create additional child nodes in the DOM, leading to unintended consequences when using these properties.*

### 18.2 Node methods

**n.appendChild(*n*);** appends or moves node *n* to the end of the `NodeList` of an element

**n.cloneNode(*n*);** creates a copy of an existing node

**n.insertBefore(*n*[,*n*]);** inserts node *n*, which may be an existing or a new child node, before child node *n* of a `NodeList` or at the end if [*n*] is not specified

**n.parentNode.removeChild(*n*);** removes child node *n* in the `NodeList` from a node

**n.parentNode.replaceChild(<node>,*n*);** replaces node *n* in the `NodeList` with <node>

## 19 The document object

### 19.1 Document properties

To find a document's position, see section 22.

**d.activeElement** returns the currently focused element

**d.body[=v]** returns or sets the first child of an `<html>` element, provided it is a `<body>` or `<frameset>` element

**d.commands** returns an `HTMLCollection` array object of the elements in the document that define commands and have IDs.

**d.cookie[=v]** returns or sets the cookies associated with the document

**d.currentScript** returns the `<script>` element that has most recently started executing or `null`

**d.defaultView** returns the `Window` object of the active document

**d.designMode[=v]** returns or sets `on` if the document is editable, and `off` if it isn't; focuses the document and resets the selection in that document

**d.dir[=v]** returns or sets the value of the `<html>` element's `dir=" "` attribute, if any, to `ltr`, `rtl` or `auto`, replacing the existing value; does nothing if the `<html>` element does not exist

**d.domain[=v]** returns the current domain used for security checks; can be set to a value that removes subdomains, to change the effective script origin to allow pages on other subdomains of the same domain (if they do the same thing) to access each other

**d.embeds** returns an `HTMLCollection` array object of the `<embed>` elements in the document (`=document.plugins`)

**d.forms** returns an `HTMLCollection` array object of the `<form>` elements in the document

**d.forms.v** returns the `<form>` element whose `id` or `name` contains `v`

**d.head** returns the `<head>` element of the document

**d.images** returns an `HTMLCollection` array object of the `<img>` elements in the document

**d.lastModified** returns when the document was last modified in `MM/DD/YYYY hh:mm:ss` format or the current time if it has not been modified

**d.links** returns an `HTMLCollection` array object of the `<a>` and `<area>` elements in the document that have `href=" "` attributes

**d.localStorage** returns the `Storage` object assigned for local storage

**d.plugins** returns an `HTMLCollection` array object of the `<embed>` elements in the document (`=document.embeds`)

**d.readyState** returns `loading`, `interactive` (when it has loaded but is still loading sub-resources) or `complete`

**d.referrer** returns the address of the document from which the user navigated to this one; in the case of an HTML document, it matches the `Referer` (*sic*) header that was sent when fetching the page (the HTML `noreferrer` link type may be used to block this behaviour)

**d.scripts** returns an `HTMLCollection` array object of the `<script>` elements in the document

**d.sessionStorage** returns the `Storage` object assigned for session storage; each `Document` object must have a separate object for its Window's `sessionStorage` attribute

**d.title[=*v*]** returns, or sets if the root element is not an `SVGDocument`, the value of the `<title>` element of the document

## 19.2 Document methods

**createDocument();**

**createHTMLDocument();**

**d.close();** closes the input stream that was opened by the `d.open()` method.

**d.createElement("tag");** creates a new `<tag>` element

**d.createElementNS("URI","tag");** creates a new `<tag>` element in the namespace specified by URI

**d.createTextNode("v");** creates a new text node whose value is "v"

**d.getElementById("v");** returns the element whose `id="v"`

**d.getElementsByName("v");** returns a `NodeList` of the elements whose `name="v"`

**d.getElementsByTagName("tag");** returns an array containing the elements whose name is `<tag>` in the document; `tag` is case insensitive because HTML tags are case insensitive; the array is dynamically updated and individual elements can be accessed in the same manner as elements in any array

**d.getElementsByTagName(v1,v2,...);** returns an array containing the elements whose `class="v1", "v2" ...` which is dynamically updated

**d.getItems([types]);** returns a `NodeList` of the elements in the `Document` that create Microdata `items`, that are not part of other `items`, and that are of the types given in the space-separated list of types, if any

**d.hasFocus();** returns `true` if the document has focus; otherwise, returns `false`

**d=d.open([type[,replace]]);** replaces the existing document as if it was a new `Document` object, but reusing the previous object, which is then returned; `type` may be `text/html` (the default) in which case the HTML parser is used to parse `document.write()`; otherwise, content is parsed as plain text; if `replace` is present, the `Window`, `Location`, `History`, `ApplicationCache`, and `Navigator`, objects, the various `BarProp` objects, the two `Storage` objects and the various `HTMLCollection` array objects are replaced

**w=d.open(url,name,features[,replace]);** = `window.open()` method (see page 34)

**d.querySelector(v);** returns the first element whose CSS selector="v" or null

**d.querySelectorAll(v);** returns an array, which may be empty, containing the elements whose CSS selector = "v" which is not dynamically updated

**d.write();** adds the given string(s) to the document's input stream; however, this method is unreliable and use of it is discouraged

**d.writeln();** adds the given string(s) terminated by a line feed to the document's input stream

## 20 The element node

Each HTML element has an `element` node which is the parent of all the nodes in the element each of which has an associated object. A `NodeList` is a read-only array like object. The following properties of the `element` node are inherited by all nodes in an element:

### 20.1 Element node properties

**e.childElementCount** returns the number of child elements in the element

**e.children** returns a `NodeList` of element child nodes in the element

**e.children.length** returns the number of child elements in the element

**e.firstElementChild** returns the first element node or `null` if the node has no children

**e.lastElementChild** returns the last element node or `null` if the node has no children

**e.nextElementSibling** returns the next element sibling node, in the order in which they appear in the document

**e.previousElementSibling** returns the next element sibling node, in the order in which they appear in the document

**e.tagName**

### 20.2 Element node methods

**select.add(e[,before]);** inserts an element before the node given by `before` which can be a number, in which case element is inserted before the item with that number, or an element from the list of options, in which case element is inserted before that element; if `before` is omitted, `null`, or a number out of range, then element will be added at the end of the list

---

## 21 Collection objects

Collection objects include:

- `HTMLCollection`<sup>5</sup> which defines the object's supported property indices
- `HTMLFormControlsCollection` defining the elements in a `<form>` or `<fieldset>` element

- `HTMLOptionsCollection` defining the `<option>` elements in a `<select>` element
- `HTMLPropertiesCollection` array object which contains all the properties of an element

<sup>5</sup>`HTMLAllCollection` is retained for legacy reasons (Hickson, 2013, 2.7.2.1)

## 21.1 Collection object properties

**c[n]** returns the element or item with index *n* from the collection; the items are sorted in tree order

**c[name]** returns

- an `HTMLCollection` array object containing all those `<a>`, `<area>`, `<embed>`, `<form>`, `<frameset>`, `<iframe>`, `<img>`, `<input>`, `<map>`, `<meta>`, `<object>`, `<select>` or `<textarea>` elements,<sup>6</sup>
- a `RadioNodeList` object containing all those elements in an `HTMLFormsCollection` object
- the first matching object from an `HTMLOptionsCollection` array object or
- a `PropertyNodeList` object containing any elements that add a property named `name`; the `name` index has to be one of the values listed in `name=" "`

whose `id="name"` or whose `name="name"` from the collection

**c.length** returns the number of elements in the collection

**c.length[=v]** returns or sets the number of elements in an `HTMLOptionsCollection` array object, truncating the number of `<option>` elements if *v* is smaller, or adding new blank `<option>` elements if *v* is larger, than the existing number of `<option>` elements but without adding or removing any `<optgroup>` elements though it may remove children from them

**c.names** returns an array of the property names of the elements in an `HTMLPropertiesCollection` array object

<sup>6</sup>also the deprecated elements `<applet>` and `<frame>`.

<sup>7</sup>also the deprecated elements `<applet>` and `<frame>`.

**c.selectedIndex[=n]** returns or sets the index of the first selected item, if any, or `-1` if there is no selected item, in an `HTMLOptionsCollection` array object

**radioNodeList.v[=v]** returns or sets the value of the first checked radio button represented by the object

**select.options** returns an `HTMLOptionsCollection` array object of the list of options.

**select.selectedOptions** returns an `HTMLCollection` array object of the list of options that are selected.

## 21.2 Collection object methods

**c(n)**; returns the item with index *n* from the collection; the items are sorted in tree order

**c(name)**; returns

- an `HTMLCollection` array object containing all those `<a>`, `<area>`, `<embed>`, `<form>`, `<frameset>`, `<iframe>`, `<img>`, `<input>`, `<map>`, `<meta>`, `<object>`, `<select>` or `<textarea>` elements,<sup>7</sup>
- a `RadioNodeList` object containing all those elements in an `HTMLFormsCollection` object or
- the first matching object from an `HTMLOptionsCollection` array object

whose `id="name"` or whose `name="name"` from the collection

**c.add(e[,before])**; inserts an element into an `HTMLOptionsCollection` array object before the node given by `before` which can be



- a number, in which case element is inserted before the item with that number, or
- an element from the collection, in which case element is inserted before that element

if `before` is omitted, `null`, or a number out of range, then element will be added at the end of the list

**c=c.item(*n*);** returns the element or item with index *n* from the collection; the items are sorted in tree order

**c=c.item(name);** or

**c=c.namedItem(name);** returns an HTMLCollection array object containing all those `<a>`, `<area>`, `<embed>`, `<form>`, `<frameset>`, `<iframe>`, `<img>`, `<input>`, `<map>`, `<meta>`, `<object>`, `<select>` or `<textarea>` elements<sup>8</sup> whose `id="name"` or whose `name="name"` from the collection

**c=c.tags(tagName);** returns a collection that is a filtered view of the current collection, containing only elements with the given tag name

**c=d.getElementsByName(v);** returns a NodeList object of the elements whose `name="v"` which is dynamically updated

Note that the `name=" "` value of a `<form>` or `<img>` element becomes a property of the Document object and the `name=" "` value of an `<iframe>` a property of the Window object and so are returned by `document.v` or `window.v`

**e=c.item(*n*);** returns the element or item with index *n* from the collection; the items are sorted in tree order

**e=c.item(name);**

**e=c.namedItem(name);** returns

- the item from a collection containing `<a>`, `<area>`, `<embed>`, `<form>`, `<frameset>`, `<iframe>`, `<img>` or `<object>` elements<sup>9</sup> or
- the first matching object from an HTMLOptions Collection array object

whose `id="name"` or whose `name="name"`

**nodeList=c.namedItem(name);** returns the first matching object from an HTMLOptionsCollection array object whose `id="name"` or whose `name="name"`

**propertyNodeList.getValues();** returns an array of the values of an HTMLPropertiesCollection array object

**propertyNodeList=c.namedItem(name);** returns a Property NodeList object containing any elements that add a property named name

**radioNodeList=c.namedItem(name);** returns the item whose `id="name"` or whose `name="name"` from an HTMLFormsCollection array object

---

<sup>8</sup>also the deprecated elements `<applet>` and `<frame>`.

<sup>9</sup>also the deprecated elements `<applet>` and `<frame>`.



## 22 Element objects

HTML`Element` objects represent HTML elements in a document; each has a `CSSStyleDeclaration` CSS object for the element's `style=""` attribute; some elements can have alternatives to the HTML`Element` object:

- HTML`HtmlElement` for `<html>` elements (=documentElement)
- HTML`HeadElement` for `<head>` elements
- HTML`TitleElement` for `<title>` elements
- HTML`BaseElement` for `<base>` elements
- HTML`LinkElement` for `<link>` elements
- HTML`MetaElement` for `<meta>` elements
- HTML`StyleElement` for `<style>` elements
- HTML`ScriptElement` for `<script>` elements
- HTML`TemplateElement` for `<template>` elements
- HTML`BodyElement` for `<body>` elements
- HTML`HeadingElement` for `<h1>`--`<h2>` elements
- HTML`ParagraphElement` for `<p>` elements
- HTML`HRElement` for `<hr>` elements
- HTML`PreElement` for `<pre>` elements
- HTML`QuoteElement` for `<blockquote>` and `<q>` elements
- HTML`OLListElement` for `<ol>` elements
- HTML`ULListElement` for `<ul>` elements
- HTML`LIElement` for `<li>` elements
- HTML`DLListElement` for `<dl>` elements
- HTML`DivElement` for `<div>` elements
- HTML`AnchorElement` for `<a>` elements
- HTML`DataElement` for `<data>` elements
- HTML`TimeElement` for `<time>` elements
- HTML`SpanElement` for `<span>` elements
- HTML`BRElement` for `<br>` elements
- HTML`ModElement` for `<ins>` and `<del>` elements
- HTML`ImageElement` for `<img>` elements
- HTML`IFrameElement` for `<iframe>` elements
- HTML`EmbedElement` for `<embed>` elements
- HTML`ObjectElement` for `<object>` elements
- HTML`ParamElement` for `<param>` elements
- HTML`MapElement` for `<map>` elements
- HTML`AreaElement` for `<area>` elements
- HTML`KeygenElement` for `<keygen>` elements
- HTML`OutputElement` for `<output>` elements
- HTML`ProgressElement` for `<progress>` elements
- HTML`MeterElement` for `<meter>` elements
- HTML`DetailsElement` for `<details>` elements

- `HTMLMenuElement` for `<menu>` elements
- `HTMLMenuItemElement` for `<menuitem>` elements
- `HTMLDialogElement` for `<dialog>` elements
- `HTMLCanvasElement` for `<canvas>` elements

## 22.1 Element object properties

HTML attributes and CSS values, other than those which are also reserved words in Javascript, become properties of elements and can be queried or modified using Javascript. Wherever an HTML attribute contains a space or a CSS value contains a hyphen, this is removed and the succeeding letter capitalised.

Other than for `class`, HTML attributes which are reserved words in Javascript are preceded by `html`, for example, `htmlFor` corresponds to the `for` attribute of a `<label>` element.

**e.accessKeyLabel** exposes the `Access Key` facet of the command

**e.classList** returns the `DOMTokenList` for the element

**e.className** returns all the values of an element's `class=" "` attribute — a substitute for the HTML `class=" "` attribute (see above)

**e.commandChecked** exposes the `Checked State` facet of the command

**e.commandDisabled** exposes the `Disabled State` facet of the command

**e.commandHidden** exposes the `Hidden State` facet of the command

**e.commandIcon** exposes the `Icon` facet of the command

**e.commandLabel** exposes the `Label` facet of the command.

**e.commandType** exposes the `Type` facet of the command

**e.contentEditable[=v]** returns or sets `true`, `false` or `inherit` based on the state of the `contenteditable` attribute

**e.cssElementMap** returns a `DOMElementMap` object for the document representing the current CSS element reference identifiers

**e.dataset** returns a `DOMStringMap` object for the element's `data-*` attributes, converting hyphenated names to camel-cased; for example, `data-foo-bar=" "` becomes `element.dataset.fooBar`.

**e.draggable[=v]** returns or sets `true` if the element is `draggable`; otherwise, returns `false`

**e.id** exposes the `ID` facet of the command.

**e.innerHTML** returns the HTML content of an element

**e.isContentEditable** returns `true` if the element is `editable`; otherwise, returns `false`

**e.itemValue[=v]** returns or sets the value of an element with an `itemprop` attribute

**e.outerHTML**

**e.properties** returns an `HTMLPropertiesCollection` array object with all the properties on an element with an `itemscope` attribute; otherwise, an empty `HTMLPropertiesCollection` array object

**e.selectionDirection[=v]** returns or sets the current direction of the selection using one of the values `forward`, `backward` or `none`

**e.selectionEnd[=v]** returns the offset to the end of the selection or sets the offset to change the end of the selection

**e.selectionStart[=v]** returns the offset to the start of the selection or sets the offset to change the start of the selection

**e.spellcheck[=v]** returns or sets **true** if the element is to have its spelling and grammar checked; otherwise, returns **false**

**e.style** returns a `CSSStyleDeclaration` CSS object for the element's `style=" "` attribute

**e.style.<cssText>** sets or modifies the value property of the `CSSStyleDeclaration` CSS object, for example:

```
e.style.fontSize="24pt";
```

**e.text[=v]** returns, if the `<a>` element, and/or sets (to replace the element's children with the given value), the contents of `<script>` or `<title>` elements, ignoring child nodes that aren't `Text` nodes

**e.title** exposes the Hint facet of the command

**e.validationMessage** returns the error message that would be shown to the user if the element was to be checked for validity

**e.validity.badInput** returns **true** if the user has provided input in the user interface that the user agent is unable to convert to a value; **false** otherwise

**e.validity.customError** returns **true** if the element has a custom error; **false** otherwise

**e.validity.patternMismatch** returns **true** if the element's value doesn't match the provided pattern; **false** otherwise

**e.validity.rangeOverflow** returns **true** if the element's value is higher than the provided maximum; **false** otherwise

**e.validity.rangeUnderflow** returns **true** if the element's value is lower than the provided minimum; **false** otherwise

**e.validity.stepMismatch** returns **true** if the element's value doesn't fit the rules given by the `step` attribute; **false** otherwise

**e.validity.tooLong** returns **true** if the element's value is longer than the provided maximum length; **false** otherwise

**e.validity.tooShort** returns **true** if the element's value, if not an empty string, is shorter than the provided minimum length; **false** otherwise

**e.validity.typeMismatch** returns **true** if the element's value is not in the correct syntax; **false** otherwise

**e.validity.valid** Returns **true** if the element's value has no validity problems; **false** otherwise

**e.validity.valueMissing** returns **true** if the element has no value but is a required field; **false** otherwise

**e.willValidate** returns **true** if the element will be validated when the form is submitted; **false** otherwise

**dialog.returnValue[=result]** returns or sets the `<dialog>` element's return value

**i.complete** returns **true** if the image has been completely downloaded or if no image is specified or available; otherwise, returns **false**

**i.currentSrc** returns the URL of the image

**i.naturalWidth**

**i.naturalHeight** returns the intrinsic dimensions of the image, or zero if the dimensions are not known

**i.newImage([width[,height]])** returns a new element, optionally with width and height set

**i.width[=*n*]**

**i.height[=*n*]** returns or sets the actual rendered dimensions of the image, or zero if the dimensions are not known

**<imageBitmap>.height** returns the intrinsic height of the image, in CSS pixels

**<imageBitmap>.resolution** returns the intrinsic linear pixel density of the image, in image pixels per CSS pixels.

**<imageBitmap>.width** returns the intrinsic width of the image, in CSS pixels

**keygen.type** returns the string keygen

**map.areas** returns an HTMLCollection array object of the <area> elements in the map.

**map.images** returns an HTMLCollection array object of the <img> and <object> elements that use the map

**template.content** returns the contents of a <template> element which are stored in a DocumentFragment object associated with a different document so as to avoid the template contents (form controls from being submitted, scripts from executing, and so forth) interfering with the main document

## 22.2 Element methods

**e.add(v);** adds a value to the `classList` property

**e.blur();** unfocuses the element; use of this method is discouraged; focus another element instead

**e.checkValidity();** returns `true` if the element's value has no validity problems; `false` otherwise in which case it fires an invalid event at the element

**e.click();** triggers the Action of the command

**e.contains(v);** returns `true` if the `classList` property contains the value

**e.focus();** focuses the element

**e.forceSpellCheck();** forces the user agent to report spelling and grammar errors on the element (if checking is enabled), even if the user has never focused the element. (If the method is not invoked, user agents can hide an error in the text that wasn't entered by the user.)

**e.getAttribute(attribute);** returns the value of an HTML attribute as a string

**e.getBoundingClientRect();** returns an element's position as an object with the properties `left`, `right`, `top` and `bottom`; `left` and `top` are the *x* and *y* coordinates of the upper left corner and `right` and `bottom` the *x* and *y* coordinates of the bottom right corner. These values include the padding and border but not the margin of the element. To find the document position, add the values of the page offsets, if any, `window.pageXOffset` and `window.pageYOffset` (section 17).

**e.getValue();** returns a non-standard HTML value or a CSS value

**e.hasAttribute();** returns true if an HTML element has an HTML attribute

**e.insertAdjacentHTML(v,"HTML");** inserts a string of HTML at position v which may be:

**beforebegin** before the opening tag

**afterbegin** after the opening tag

**beforeend** before the closing tag

**afterend** after the closing tag

**e.remove(v);** removes a value from the `classList` property

**e.removeAttribute();** removes an HTML attribute

**e.reportValidity();** returns `true` if the element's value has no validity problems and, if the event is not cancelled, reports this to the user; `false` otherwise in which case it fires an invalid event at the element

**e.scrollToIntoView();** resets the values of `window.pageXOffset` and `window.pageYOffset` to 0. However, if the argument `false` is passed, it will reset the values of `window.pageXOffset` and `window.pageYOffset` to ensure that the bottom of the element appears in the viewport.

**e.select();** selects everything in the text field

**e.setAttribute(attribute,v);** sets the value of an HTML attribute where v is a string

**e.setCustomValidity(<message>);** sets a custom error, so that the element would fail to validate; `message` is the message to be shown to the user when reporting the problem to the user; if empty, it clears the custom error

**e.setRangeText(replacement[,start,end[,selectionMode]]);** replaces a range of text with the new text; if the start and end arguments are not provided, the range is assumed

to be the selection; `selectionMode` determines how the selection should be set after the text has been replaced:

- `select` selects the newly inserted text
- `start` moves the selection to just before the inserted text
- `end` moves the selection to just after the selected text
- `preserve` (the default) attempts to preserve the selection

**e.setSelectionRange(start,end[,direction]);** changes the selection to cover the given substring in the given direction; if the direction is omitted, it will be reset to be the platform default (none or forward)

**e.setValue();** sets a non-standard HTML value or a CSS value

**e.toggle(v);** adds a value to the `classList` property if it is absent or removes it if it is present

**dialog.close([result]);** closes the `<dialog>` element; `result`, if provided, provides a return value

**dialog.requestFullscreen();**

**dialog.show([anchor]);** displays the `<dialog>` element; `anchor`, if provided, provides an anchor point to which the element will be fixed

**dialog.showModal([anchor]);** displays the `<dialog>` element and makes it the top-most modal dialog; `anchor`, if provided, provides an anchor point to which the element will be fixed; this method honours the `autofocus` attribute

**embed.getSVGDocument()** returns the Document object being used for SVG images in an `<embed>` element

## 22.3 Child element methods

**e.getElementsByTagName(tag);** returns an array containing the child elements whose name is `<tag>` in the element; `tag` is case insensitive because HTML tags are case insensitive; the array is dynamically updated and individual elements can be accessed in the same manner as elements in any array

**e.getElementsByClassName(v1,v2,...);** returns an array containing the child elements whose `class="v1"` or `"v2" ...`

**e.querySelector(v);** returns the first child element whose CSS selector = `v` or `null`

**e.querySelectorAll(v);** returns an array, which may be empty,

**<option>=new Option([text[,v[,defaultSelected[,selected]]]]);** returns a new `<option>` element where

- `text` sets the contents of the element
- `v` sets the `value` attribute
- `defaultSelected` sets the `selected` attribute
- `selected` sets whether or not the element is selected; if it is omitted, even if the `defaultSelected` argument is `true`, the element is not selected

containing the child elements whose CSS selector = `v` which is not dynamically updated

## 22.4 Element object constructors

**<audio>=new Audio([url]);** returns a new `<audio>` element, with the `src=" "` attribute set to the value passed by `url`, if applicable

**<controller>=new MediaController();** returns a new `MediaController` object.

**i=new Image([width[,height]]);** returns a new `<img>` element, with the `width` and `height` attributes set to the values passed in the relevant arguments, if applicable

---

## 23 The text node

### 23.1 Text node properties

**n.textContent** returns the plain text of an element

## 24 Canvas element objects

Canvas element objects include `CanvasRenderingContext2D`, `CanvasDrawingStyles`, `CanvasGradient`, `CanvasPathMethods`, `CanvasPattern`, `DrawingStyle`, `Path`, `TextMetrics` and `WebGLRenderingContext`.

To enable a script to access the `<canvas>` element, it must have an `id=" "` attribute; for example,

```
<canvas id="a">
```

creates the variable `a_canvas` which can be located in the DOM or with

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

and can be manipulated with `<id>_context.<expression>`.

### 24.1 Contexts

Every canvas can have one associated primary context called using, for example:

```
var a_context=a_canvas.getContext("<value>"[,<arguments>]);
```

where `value` may be `2d` (section [24.2 on the next page](#)) or `webgl` (which supports a 3D context using OpenGL 2.0).

```
a_canvas.supportsContext("<value>"[,<arguments>]);  
a_canvas.probablySupportsContext("<value>"[,<arguments>]);
```

returns `true` if the canvas supports the context specified by `value`.

You can give a context a new rendering context with

```
a_context=new CanvasRenderingContext2D([<width>,<height>]);
```

**canvas.height** returns height of the current canvas element

**canvas.width** returns width of the current canvas element

To reset a `<canvas>` element, simply declare its width or height, for example,

```
a_canvas.width=" ";
```

```
a_canvas.width=a_canvas.width;
```

but note that this does not reset the *origin-clean* flag.

Control of a canvas element can be handed over to a proxy, for example, another HTML element such as an `<iframe>` element or a web worker (section 36 on page 84), which is not able to access a canvas directly, with:

```
proxy=a_canvas.transferControlToProxy();
```

while

```
a_canvas.setContext("<context>");
proxy.setContext("<context>");
```

binds the canvas or the proxy to that context.

## 24.2 2d context

You can save an existing canvas image as an `image/png` at 96dpi or the native density of the image (using the second option), unless the optional `<type>` argument is `image/jpeg`,<sup>10</sup> to a URL with:

```
<url>=a_canvas.toDataURL([<type>]);
```

or to a Javascript object which can be accessed using the `<callback>` with:

```
a_canvas.toBlob(<callback>[,<type>]);
```

You can find the resolution at which a bitmap will be drawn with

```
w.screen.canvasResolution;
```

<sup>10</sup>If saving to a JPEG, a number between 0.0 and 1.0 may be added as an argument to specify the compression.

### 24.2.1 Context, path and styles attributes

The `CanvasRenderingContext2D` object properties and methods include `context`, `path` and `styles` properties and methods; these are given as `context` properties and methods with a note at the end indicating where the same construct may be used with `path` and/or `styles`. In each of these expressions ‘`context`’ should be replaced by the primary context, for example, ‘`a_context`’,

#### General context properties

**`context.canvas`** returns the current canvas element

**`context.currentTransform[=v]`** returns, or sets, the current transform in SVG matrix format

**`context.fillStyle[=v]`** returns, or sets, the current fill style, the value may be a CSS colour (the default fill style is `#000` = black), a `CanvasGradient` or `CanvasPattern` object

**`context.globalAlpha[=v]`** returns, or sets, the current alpha value

**`context.globalCompositeOperation[=v]`** returns, or sets, the current composition operation; the possible values are:

- `clear`
- `copy A` (B is ignored)
- `destination B` (A is ignored)
- `destination-atop B atop A`.
- `destination-in B in A`.
- `destination-out B out A`.
- `destination-over B over A`.
- `lighter A plus B`; show the sum of the RGB values with a limit of 255



- `source-atop` A over B; B only shows where it is opaque and A is transparent
  - `source-in` A in B; A only where B is opaque
  - `source-out` A out B; A only where B is transparent
  - `source-over` A over B; B only shows where A is transparent; the default
  - `xor` A minus B plus B minus A; the non-overlapping regions are combined
  - `<vendorName>-<operationName>` a vendor specific extension
- context.height[=*v*]** returns or sets the canvas height in CSS pixels, which will also update the canvas height

**context.strokeStyle[=*v*]** returns, or sets, the current stroke style; the value may be a CSS colour (the default stroke style is #000 = black), a `CanvasGradient` or `CanvasPattern` object, which may be

**gradient.addColorStop(*offset*,*colour*)**; adds a colour stop with the given colour to the gradient at the given offset where the offset is in the range 0--1; there must be two colour stops for each gradient

**gradient=context.createLinearGradient(*x0*,*y0*,*x1*,*y1*)**; returns a `CanvasGradient` object that represents a linear gradient that paints along the line given by *x0*,*y0* to *x1*,*y1*; where *x0*,*x1* or *y0*,*y1* are the same, the gradient will be one dimensional

**pattern=context.createPattern(*image*,*repetition*)**; returns a `CanvasPattern` object that uses the given image and repeats in the direction(s) given by the repetition argument, which may be

- `repeat` moves the image in both directions; the default
- `repeat-x` moves the image horizontally only
- `repeat-y` moves the image vertically only
- `no-repeat` does not move the image

**gradient=context.createRadialGradient(*x0*,*y0*,*r0*,*x1*,*y1*,*r1*)**; returns a `CanvasGradient` object that represents a radial gradient that paints along the cone from the circle at origin *x0*,*y0* with radius *r0* to the circle at *x1*,*y1* with radius *r1* (NOT SUPPORTED IN IE>8)

**pattern.setTransform(*<transform>*)**; sets the transform that will be applied when rendering the pattern

Gradient methods are always associated with a particular canvas context, for example,

```
var my_gradient=context.createLinearGradient(0, 0, 300, 0);
```

**context.width[=*v*]** returns or sets the canvas width in CSS pixels, which will also update the canvas width

### General context methods

**context.commit();** displays the current frame

**context.resetTransform();** resets the current transform

**context.restore();** pops the top state of the drawing state stack to the canvas

**context.rotate(*angle*);** rotates the drawing by the angle in radians

**context.save();** pushes a copy of the drawing state to the top of the drawing state stack

**context.scale(*x,y*);** scales the drawing by the horizontal and vertical factors, *x* and *y* respectively

**context.setTransform(*a,b,c,d,e,f*);** resets the current transform to the identity matrix, and then invokes the `transform(a,b,c,d,e,f)` method with the same arguments

**context.translate(*x,y*);** moves the drawing by *x* units horizontally and *y* units vertically

**context.transform(*a,b,c,d,e,f*);** replaces the current transformation matrix with the result of multiplying the current transformation matrix with the matrix described by:

```
a c e
b d f
0 0 1
```

Note that different browsers handle this transformation differently.

### Drawing style object constructor

New `DrawingStyle` objects can be created with:

```
s=new DrawingStyle([e]);
```

### Line context properties

**context.lineCap[=*v*]** returns. or sets, the current line cap style; possible line cap styles are `butt`, `round` and `square`; the default is `butt`; also `styles`

**context.lineDashOffset** returns. or sets, the current line dash style phase offset; the default is 0; also `styles`

**context.lineJoin[=*v*]** returns. or sets, the current line join style; possible line join styles are `bevel`, `round` and `miter`; the default is `miter`; also `styles`

**context.miterLimit[=*v*]** returns, or sets, the current miter limit ratio; the default is 10.0; also `styles`

**context.lineWidth[=*v*]** returns, or sets, the current line width; the default is 1.0; also `styles`

### Line context methods

**context.arcTo(*x1,y1,x2,y2,radiusX[,radiusY,rotation]*);** adds an arc with the given control points and radius to the current subpath, connected to the previous point by a straight line; the optional arguments create an elliptic arc and control its clockwise rotation; also `path`

**context.arc(*x,y,radius,startAngle,endAngle[,anticlockwise]*);** adds points to the subpath such that the arc described by the circumference of the circle described by the arguments, starting at the given start angle and ending at the

given end angle, going in the given direction (defaulting to clockwise), is added to the path, connected to the previous point by a straight line; also `path`

**context.beginPath();** resets the current path

**context.bezierCurveTo(cp1x,cp1y,cp2x,cp2y,x,y);** adds the given point to the current subpath, connected to the previous one by a cubic Bézier curve with the given control points; also `path`

**context.clearRect(x,y,w,h);** clears all pixels on the canvas in the rectangle where the origin of `x,y` is the top left corner of the canvas to transparent black

**context.clip([fillRule]);** further constrains the clipping region to

**context.ellipse(x,y,radiusX,radiusY,rotation,startAngle,endAngle[,anticlockwise]);** adds points to the subpath such that the arc described by the circumference of the ellipse described by the arguments, starting at the given start angle and ending at the given end angle, going in the given direction (defaulting to clockwise), is added to the path, connected to the previous point by a straight line.; also `path`

**context.fill([p]);** fills the subpaths of the default or the specified path with the current fill style

**context.fill(p,[fillRule]);** fills the subpaths of the specified path with the given fill style

**context.fillRect(x,y,w,h);** paints the rectangle where the origin of `x,y` is the top left corner of the canvas onto the canvas, using the current fill style

**context.isPointInPath([p],x,y[,fillRule]);** returns true if the given point is in the default or specified path with the specified fill style

**context.isPointInStroke([p],x,y);** returns true if the given point

the the default path using the specified fill style

**context.clip(p[,fillRule]);** further constrains the clipping region to the specified path using the specified fill style

**context.closePath();** marks the current subpath as closed, and starts a new subpath with a point the same as the start and end of the newly closed subpath; also `path`

**context.drawSystemFocusRing([p],e);** draws a focus ring around the default or the specified path of the element; to test whether there is a custom focus ring around the default or the specified path of the element and draw it if necessary use

**shouldDraw=context.drawCustomFocusRing([p],e);**

in the stroke region would be in the default or the specified path

**context.lineTo(x,y);** adds the given point to the current subpath, connected to the previous one by a straight (invisible) line; to get a line 1 pixel wide, always specify `x` as `n.5`; also `path`

**context.moveTo(x,y);** creates a new subpath with the given point; to start a line 1 pixel wide, always specify `x` as `n.5`; also `path`

**context.quadraticCurveTo(cpx,cpy,x,y);** adds the given point to the current subpath, connected to the previous one by a quadratic Bézier curve with the given control point; also

path

**context.rect(x,y,w,h);** adds a new closed subpath to the path, representing the given rectangle; also `path`

**context.resetClip();** removes the constraints on the path

**context.scrollPathIntoView([p]);** scroll the default or the given path into view

**context.setLineDash(<segments>);** sets the current line dash style; also `styles`; to get the current line dash style use the `<segments>` argument is a list of the distances to use to turn the line off and on

`<segments>=context.getLineDash();`

**context.stroke([p]);** strokes (makes visible) the subpaths of the default or the specified path with the current stroke style

**context.strokeRect(x,y,w,h);** paints the box that outlines the rectangle where the origin of `x,y` is the top left corner of the canvas onto the canvas, using the current stroke style

### Text context properties

**context.direction[=v]** returns, or sets, the current directionality; also `styles`; values may be `ltr`, `rtl` or `inherit`, the default

**context.font[=v]** returns, or sets, the current font settings; also `styles`; the value may be anything you can put in a CSS font rule including:

- `bold`
- `npx`
- `sans-serif`
- `serif`

**context.textAlign[=v]** returns, or sets, the current text alignment settings; also `styles`; values may be:

- `start` (the default) depends on directionality
- `end` depends on directionality
- `left`
- `right`
- `center`

N.B. these are not quite the same as those in the CSS `text-align` rule.

**context.textBaseline[=v]** returns, or sets, the current baseline alignment settings (figure 1); also `styles`; values may be:

- `top` the top of the em square
- `hanging` the hanging baseline
- `middle` the middle of the em square
- `alphabetic` the alphabetic baseline; the default
- `ideographic` the ideographic baseline
- `bottom` the bottom of the em square

N.B. `top` and `bottom` may create unexpected effects in some languages.

### Text context methods

**context.fillText(text,x,y[,maxWidth]);** fill the given text at the given position; if a maximum width is provided, the text will be scaled to fit that width if necessary

**context.strokeText(text,x,y[,maxWidth]);** stroke the given text at the given position; if a maximum width is provided, the text will be scaled to fit that width if necessary; the following return

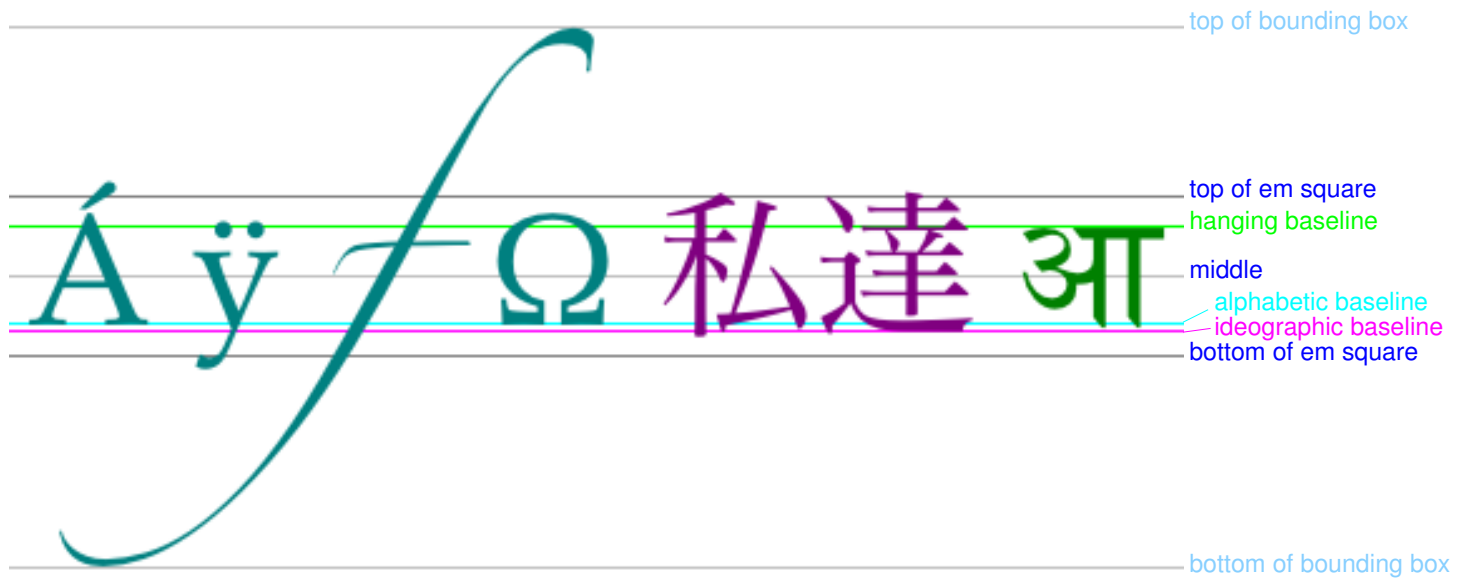


Figure 1: Text baselines

**metrics.measureText(text)**; returns a `TextMetrics` object with the metrics of the given text in the current font; the `TextMetrics` object properties can be returned with:

**metrics.actualBoundingBoxAscent** the distance from the horizontal line indicated by the `textBaseline` attribute to the top of the bounding rectangle of the given text, in CSS pixels; positive numbers indicate a distance going up from the given baseline; this number can vary greatly based on the input text, even if the first font specified covers all the characters in the input; for example, the `actualBoundingBoxAscent` of a lowercase 'o' from an alphabetic baseline would be less than that of an uppercase 'F'; the value can easily be negative; for example, the distance from the top of the `em` box (`textBaseline` value "top") to the top of the bounding rectangle when the given text is just a single comma "," would likely (unless the font is quite unusual) be negative

**metrics.actualBoundingBoxDescent** the distance from the horizontal line indicated by the `textBaseline` attribute to the bottom of the bounding rectangle of the given text, in CSS pixels; positive numbers indicate a distance going down from the given baseline

The `actualBoundingBoxAscent` and `actualBoundingBoxDescent` attributes are useful when drawing a bounding box around specific text.

**metrics.actualBoundingBoxLeft** the distance parallel to the baseline from the alignment point given by the `textAlign` attribute to the left side of the

bounding rectangle of the given text, in CSS pixels; positive numbers indicate a distance going left from the given alignment point; the sum of this value and `actualBoundingBoxRight` can be wider than the width of the inline box (`width`), in particular with slanted fonts where characters overhang their advance width

**metrics.actualBoundingBoxRight** the distance parallel to the baseline from the alignment point given by the `textAlign` attribute to the right side of the bounding rectangle of the given text, in CSS pixels; positive numbers indicate a distance going right from the given alignment point

**metrics.fontBoundingBoxAscent** the distance from the horizontal line indicated by the `textBaseline` attribute to the top of the highest bounding rectangle of all the fonts used to render the text, in CSS pixels; positive numbers indicate a distance going up from the given baseline; this value and the next are useful when rendering a background that must have a consistent height even if the exact text being rendered changes

**metrics.fontBoundingBoxDescent** the distance from the horizontal line indicated by the `textBaseline` attribute to the bottom of the lowest bounding rectangle of all the fonts used to render the text, in CSS pixels; positive numbers indicating a distance going down from the given baseline.

**metrics.alphabeticBaseline** the distance from the horizontal line indicated by the `textBaseline` attribute to the alphabetic baseline of the line box, in CSS pixels; positive numbers indicate

that the given baseline is below the alphabetic baseline (zero if the given baseline is the alphabetic baseline)

**metrics.hangingBaseline** the distance from the horizontal line indicated by the `textBaseline` attribute to the hanging baseline of the line box, in CSS pixels; positive numbers indicate that the given baseline is below the hanging baseline (zero if the given baseline is the hanging baseline)

**metrics.ideographicBaseline** the distance from the horizontal line indicated by the `textBaseline` attribute to the ideographic baseline of the line box, in CSS pixels; positive numbers indicate that the given baseline is below the ideographic baseline (zero if the given baseline is the ideographic baseline)

**metrics.emHeightAscent** the distance from the horizontal line indicated by the `textBaseline` attribute to the top of the em square in the line box, in CSS pixels; positive numbers indicate that the given baseline is below the top of the em square (so this value will usually be positive); Zero if the given baseline is the top of the em square; half the font size if the given baseline is the middle of the

em square

**metrics.emHeightDescent** the distance from the horizontal line indicated by the `textBaseline` attribute to the bottom of the em square in the line box, in CSS pixels; positive numbers indicate that the given baseline is below the bottom of the em square (so this value will usually be negative or Zero if the given baseline is the top of the em square)

**metrics.width** the width of that inline box, in CSS pixels.

#### Path object constructors

**p=new Path2D();** creates an empty 2D Path object,

**p=new Path2D(p);** creates a 2D Path object that is a copy of p

**p=new Path2D(paths[,fillRule]);** creates a 2D Path object that outlines the given paths with the given fillRule

**p=new Path2D(d);** creates an SVG object

#### Path object methods

**p.addPath(p,<transform>);**

**p.addPathByStrokingPath(p,s,<transform>);**

Horizontal text may be added with

```
p.addPathByStrokingText(t,s,<transform>,x,y[,<maxWidth>]);
```

```
p.addText(t,s,<transform>,x,y[,<maxWidth>]);
```

and non-horizontal with

```
p.addPathByStrokingText(t,s,<transform>,p[,<maxWidth>]);
```

**p.addText(t,s,<transform>,p[,<maxWidth>]);** where  
<maxWidth> causes the text to be scaled to that width.

### Image properties

**context.imageSmoothingEnabled[=v]** sets or resets whether  
smoothing will take place during any rescaling

**context.shadowBlur[=v]** returns, or sets, the current level of blur  
applied to shadows

**context.shadowColor[=v]** returns, or sets, the current shadow  
colour

**context.shadowOffsetX[=v]**

**context.shadowOffsetY[=v]** returns, or sets, the current shadow  
offset

### Image methods

**context.drawImage(image,dx,dy);** draws an image with its top  
left hand corner at dx,dy

**context.drawImage(image,dx,dy,dw,dh);** draws an image with  
its top left hand corner at dx,dy and width dw and height  
dh (ONLY UNIFORM SCALING SUPPORTED IN EARLIER VER-  
SIONS OF IE)

**context.drawImage(image,sx,sy,sw,sh,dx,dy,dw,dh);** clips an  
image to make sx,sy the top left hand corner, sw its width  
and sh its height and draws it with its top left hand corner  
at dx,dy and width dw and height dh (NOT SUPPORTED IN  
EARLIER VERSIONS OF IE)

## 24.2.2 Hit regions

A context may have a hit region which is added with

**context.addHitRegion(<options>);** where <options> may be:

**path()** where the value of this path is empty or null, the  
current default path is used

**fillRule()** the fill rule used to determine what pixels are in-  
side a path

**id()** used to reference events and to remove a hit region; the  
default is an empty string

**parentID()** the id() of any parent region; intended to assist  
accessibility navigation tools

**cursor()** the cursor to use; the default is `inherit`

**control()** an element in the canvas through which events  
are routed; the default value is null; intended to assist  
accessibility navigation tools

**label()** a label for an element; intended to assist accessibil-  
ity navigation tools

**role()** an ARIA role intended to assist accessibility naviga-  
tion tools; only to be used in the absence of a label()   
attribute; the default value is null

**context.removeHitRegion(id);** removes a hit region

**event.region;** returns the id of a hit region when the mouse cursor  
is over it or null when it is not.

**touch.region;** returns the id of a hit region when the touch point  
is placed on a surface or null if there is none at that touch  
point



### 24.2.3 Pixel manipulation

Pixel manipulation may be undertaken using `ImageData` objects.

#### **ImageData object constructors**

**`imagedata=new ImageData(sw,sh)`**; creates an `ImageData` object using the screen pixels

**`imagedata=new ImageData(data,sw[,sh])`**; creates a new `ImageData` object using data provided by a `Uint8ClampedArray` argument.

#### **ImageData methods**

**`context.putImageData(imagedata,dx,dy[,dirtyX,dirtyY,dirtyWidth,dirtyHeight])`**; paints data from an `ImageData` object on the canvas

**`imagedata=context.createImageData(<imagedata>)`**; creates a new `ImageData` object with the dimensions of an existing `ImageData` object

**`imagedata=context.createImageData(sw,sh)`**; creates an `ImageData` object using the screen pixels

**`imagedata=context.getImageData(sx,sy,sw,sh)`**; returns an `ImageData` object containing the image data for the given rectangle

#### **ImageData properties**

**`imagedata.data`** returns an array in RGBA order of the data in an `ImageData` object

**`imagedata.width`** returns the width of an `ImageData` object

**`imagedata.height`** returns the height of an `ImageData` object

**`imagedata.resolution`**; returns the resolution of an `ImageData` object

---

## 25 Media element objects

As well as the `HTMLElement` object, some elements can have alternatives to the `HTMLElement` object:

- `HTMLEmbedElement` for `<embed>` elements
- `HTMLObjectElement` for `<object>` elements
- `HTMLParamElement` for `<param>` elements
- `HTMLMediaElement` or `HTMLVideoElement` for `<video>` elements

ments

- `HTMLMediaElement` or `HTMLAudioElement` for `<audio>` elements
- `HTMLSourceElement` for `<source>` elements
- `HTMLTrackElement` for `<track>` elements

## 25.1 Media element properties

**controller.buffered** returns a `TimeRanges` object that represents the intersection of the time ranges for which the user agent has all relevant media data for all the slaved media elements

**controller.currentTime[=*n*]** returns or sets the current playback position, in seconds, as a position between zero time and the current duration.

**controller.defaultPlaybackRate[=*n*]** returns or sets the default playback rate; this rate has no effect on playback until the user switches to fast-forward mode and then returns to normal playback mode

**controller.duration** returns the difference between the earliest playable moment and the latest playable moment; returns zero if there is no media.

**controller.muted[=*v*]** returns `true` if all audio is muted (regardless of other attributes either on the `MediaController` or on any media elements slaved to this `MediaController`), and `false` otherwise; can be set, to change whether the audio is muted or not

**controller.paused** returns `true` if playback is paused; `false` otherwise; when `true`, any media element slaved to this `MediaController` will be stopped.

**controller.playbackState** returns the state that the `MediaController` was in the last time it fired events as a result of reporting the `MediaController` state; values may be:

**playing** the media is actively playing

**ended** the media is not playing because playback has reached the end

**waiting** the media is not playing for some other reason (e.g. it is paused)

**controller.playbackRate[=*n*]** returns or sets the current playback rate

**controller.played** returns a `TimeRanges` object that represents the union of the time ranges in all the slaved media elements that have been played.

**controller.readyState** returns the state that the `Media controller` object was in the last time it fired events as a result of reporting the `MediaController` state. The values of this attribute are the same as for the `readyState` attribute of media elements (page 60)

**controller.seekable** returns a `TimeRanges` object that represents the intersection of the time ranges into which the user agent can seek for all the slaved media elements

**controller.volume[=*n*]** returns or sets the current playback volume multiplier, as a number in the range 0.0 to 1.0, where 0.0 is the quietest and 1.0 the loudest

**event.track** returns the track object to which the event relates

**m.audioTracks** returns an `AudioTrackList` object representing the audio tracks available in the media resource

**m.audioTracks[*n*]** returns the specified `AudioTrack` object

**m.audioTracks.length** returns the number of tracks in the list

**m.buffered** returns a `TimeRanges` object that represents the ranges of the media resource that the user agent has buffered

**m.canPlayType(type)** returns an empty string (a negative response), **maybe** or **probably** based on how confident the user agent is that it can play media resources of the given **type**

**m.controller[=controller]** returns or sets the current **MediaController** object, if any, for the media element to an explicit **MediaController**; returns **null** otherwise; setting an explicit **MediaController** removes the **mediagroup** attribute, if any.

**m.currentSrc** returns the address of the current media resource or an empty string when there is no media resource.

**m.currentTime[=n]** returns the playback position in seconds or seeks the time given by **value**

**m.defaultPlaybackRate[=n]** returns or sets the default playback rate, when the user is not fast-forwarding or reversing; a new rate has only affects playback if the user switches to a fast-forward mode and then returns to normal playback; if the element has a current **MediaController**, the **defaultPlaybackRate** attribute is ignored and the current **MediaController**'s **defaultPlaybackRate** is used instead

**m.duration** returns the length of the media resource in seconds, **NaN** if the duration isn't available or **Infinity** for unbounded streams<sup>11</sup>

**m.ended** returns **true** if playback has reached the end of the media resource

**m.error** returns a **MediaError** object representing the current error state of the element or **null** if there is no error

**m.error.code** returns the current error's error code attribute of a **MediaError** object:

---

<sup>11</sup>**NaN** and **Infinity** cannot be represented in a floating point number and therefore cannot be used in HTML.

**MEDIA\_ERR\_ABORTED(1)** the fetching process for the media resource was aborted by the user agent at the user's request

**MEDIA\_ERR\_NETWORK(2)** a network error of some description caused the user agent to stop fetching the media resource, after the resource was established to be usable

**MEDIA\_ERR\_DECODE(3)** an error of some description occurred while decoding the media resource, after the resource was established to be usable

**MEDIA\_ERR\_SRC\_NOT\_SUPPORTED(4)** The media resource indicated by the **src=" "** attribute was not suitable

**m.length** returns the number of ranges in the object

**m.muted[=v]** returns **true** if audio is muted, overriding the volume attribute, and **false** otherwise; can be set

**m.networkState** returns the current state of network activity for the element represented by the **networkState** attribute:

**NETWORK\_EMPTY(0)** the element has not yet been initialized

**NETWORK\_IDLE(1)** the element's resource selection algorithm is active and has selected a resource, but it is not actually using the network at this time

**NETWORK\_LOADING(2)** the user agent is actively trying to download data

**NETWORK\_NO\_SOURCE(3)** the element's resource selection algorithm is active, but it has not yet found a resource to use

**m.paused** returns `true` if playback is paused; `false` otherwise

**m.playbackRate**`[=n]` returns or sets the current rate playback, where 1.0 is normal speed; when the element has a current `MediaController`, the `playbackRate` attribute is ignored and the current `MediaController`'s `playbackRate` is used instead.

**m.played** returns a `TimeRanges` object that represents the ranges of the media resource that the user agent has played

**m.readyState** returns the current state of the element with respect to rendering the current playback position, the greatest value describing the state of the element:

**HAVE\_NOTHING(0)** no information available; media elements whose `networkState` attribute are set to `NETWORK_EMPTY` are always in the `HAVE_NOTHING` state

**HAVE\_METADATA(1)** Enough of the resource has been obtained that the duration of the resource is available; the dimensions of a `<video>` element are also available but there is no data for the current playback position

**HAVE\_CURRENT\_DATA(2)** data for the current playback position is available, but not enough successfully to advance or there is no more data to obtain in the direction of playback; note that the difference between this and previous state normally only matters when using the `<canvas>` element

**HAVE\_FUTURE\_DATA(3)** data for the current playback position is available, as well as enough to advance and the text tracks are ready: note that:

- once playback has ended, the current playback position can never advance and

- the difference between this and the previous state only really matters if advancing frame by frame

**HAVE\_ENOUGH\_DATA(4)** the conditions for the `HAVE_FUTURE_DATA` state are met, as well as either:

- data is being fetched at a rate where the current playback position would not overtake the available data before playback reaches the end of the media resource, or
- waiting longer will not result in further data being obtained, for example, if the buffer is full

**m.seekable** returns a `TimeRanges` object that represents the ranges of the media resource to which it is possible for the user agent to seek

**m.seeking** returns `true` if the user agent is currently seeking

**m.textTracks**`[n]` returns the `TextTrack` object representing the *n*th text track in the media element's list of text tracks

**m.textTracks.length** returns the number of all the text tracks associated with the media element

**m.videoTracks** returns a `VideoTrackList` object representing the video tracks available in the media resource

**m.videoTracks**`[n]` returns the specified `VideoTrack` object

**m.videoTracks.length** returns the number of tracks in the list

**m.videoTracks.selectedIndex** returns the index of the currently selected track, if any, or `-1` otherwise

**m.volume**`[=n]` returns or sets the current playback volume, as a number in the range 0.0 to 1.0, where 0.0 is the quietest and 1.0 the loudest.

**video.videoWidth**

**video.videoHeight** returns the intrinsic dimensions of the `<video>` element, or zero if the dimensions are not known

**m.addTextTrack(<kind>[,<label>[,<language>]]);** creates and returns a new `TextTrack` object, which is also added to the media element's list of text tracks

**m.audioTracks.getTrackById(id)** returns the `AudioTrack` object with the given `id` or `null` if no track has that identifier

**m.end(*n*);** returns the time for the end of the range with the given index

**m.fastSeek(<time>);** seeks to near the given time as fast as possible, trading precision for speed; to seek to a precise time, use the `currentTime` attribute instead; does nothing if the media resource has not been loaded

**m.load();** causes the element to reset and start selecting and loading a new media resource from scratch

**m.pause();** sets the `paused` attribute to `true`, loading the media resource if necessary

**m.play();** sets the `paused` attribute to `false`, loading the media resource and beginning playback if necessary; if playback had ended, will restart it from the beginning

**m.start(*n*);** returns the time for the start of the range with the given index

**m.textTracks.getTrackById(id)** returns the `TextTrack` object with the given `id` or `null` if no track has that identifier

## 25.2 Media element methods

**controller.pause();** sets the `paused` attribute to `true`

**controller.play();** sets the `paused` attribute to `false` and invokes the `play()` method of each slaved media element

**controller.unpause();** sets the `paused` attribute to `false`

**m.videoTracks.getTrackById(id)** returns the `VideoTrack` object with the given `id` or `null` if no track has that identifier

**newAudio([url])** returns a new `<audio>` element with the `src=""` attribute set to the URL

## 25.3 audioTrack objects

`AudioTrack` objects include `AudioTrack` and `AudioTrackList`.

### 25.3.1 audioTrack object properties

**<audioTrack>.enabled[=*v*]** returns `true` if the given track is active, and `false` otherwise; can be set, to change whether the track is enabled or not; if multiple audio tracks are enabled simultaneously, they are mixed

**<audioTrack>.id** returns the ID of the given track; can be used with a fragment identifier if the format supports the Media Fragments URI syntax, and that can be used with the `getTrackById()` method

**<audioTrack>.kind** returns the category the given track falls into (see Table 1)

Table 1: Track categories

<b>alternative</b>	alternative to the main track, e.g. a different take of a song (audio), or a different angle (video)	Audio and video	Ogg: "audio/alternate" or "video/alternate"
<b>captions</b>	version of the main video track with captions burnt in (for legacy content)	Video only	
<b>description</b>	audio description of a video track	Audio only	Ogg: "audio/audiodesc"
<b>main</b>	primary audio or video track	Audio and video	Ogg: "audio/main" or "video/main"; WebM: the "FlagDefault" element is set
<b>main-desc</b>	primary audio track, mixed with audio descriptions	Audio only	AC3 audio in MPEG-2
<b>sign</b>	sign-language interpretation of an audio track	Video only	Ogg: "video/sign"
<b>subtitles</b>	main video track with subtitles burnt in (for legacy content)	Video only	
<b>translation</b>	translated version of the main audio track	Audio only	Ogg: "audio/dub"
<b>commentary</b>	commentary on the primary audio or video track	Audio and video	
(empty string)	no explicit kind	Audio and video	Any other track type, track role, or combination of track roles not described above

`<audioTrack>.label` returns the label of the given track, if known, or the empty string otherwise.

`<audioTrack>.language` returns the language of the given track, if known, or the empty string

## 25.4 textTrack objects

TextTrack objects include `TextTrack`, `TextTrackList`, `TextTrackCue` and `TextTrackCueList`.

### 25.4.1 textTrack object properties

`cue.endTime[=n]` returns or sets the `TextTrack` cue end time in seconds

`cue.id[=v]` returns or sets the `TextTrack` cue identifier

`cue.pauseOnExit[=v]` returns `true` if the `TextTrack` cue pause-on-exit flag is set, `false` otherwise or sets

`cue.startTime[=n]` returns or sets the `TextTrack` cue start time in seconds

`cue.track` returns the `TextTrack` object to which this text track cue belongs, if any, or `null` otherwise.

`cuelist.length` returns the number of cues in the `TextTrackCueList`

`cuelist[n]` returns the text track cue with index *n* in the `TextTrackCueList`; the cues are sorted in text track cue order.

`cuelist.getCueById(id)` returns the text track cue with with the given *id* or `null` if no track has that identifier

`textTrack.activeCues` returns the `TextTrack` cues that are currently active (i.e. that start before the current playback position and end after it) as a `TextTrackCueList` object.

`textTrack.cues` returns the `TextTrack` cues, as a `TextTrackCueList` object

`textTrack.id` returns the ID of the given track which can be used with a fragment identifier if the format supports the Media Fragments URI syntax, and with the `getTrackById()` method; for `TextTrack` objects corresponding to `<track>` elements, this is the ID of the `<track>` element.

`textTrack.inBandMetadataTrackDispatchType` returns the text track in-band metadata track dispatch type string

`textTrack.kind` returns the text track kind string.

`textTrack.label` returns the text track label, if there is one, or the empty string otherwise (cue to create one!)

`textTrack.language` returns the text track language string

`textTrack.mode[=v]` returns or sets the text track mode, represented by one of the following:

**disabled** The text track disabled mode.

**hidden** The text track hidden mode.

**showing** The text track showing mode

`track.readyState` returns the text track readiness state, represented by a number from the following list:

**track.NONE(0)** The text track not loaded state.

**track.LOADING(1)** The text track loading state.

**track.LOADED(2)** The text track loaded state.

**track.ERROR(3)** The text track failed to load state.

**track.track** returns the `TextTrack` object corresponding to the text track of the `<track>` element

#### 25.4.2 textTrack object methods

**cueList.getCueById(id)**; returns the first `TextTrack` cue (in `TextTrack` cue order) with text track cue identifier `id` or `null` if none of the cues have the given identifier or if the argument is the empty string

**textTrack.addCue(cue)**; adds the given cue to `TextTrack` object's list of cues

**textTrack.removeCue(cue)**; removes the given cue from `TextTrack` object's cues

#### 25.5 videoTrack objects

`VideoTrack` objects include `VideoTrack` and `VideoTrackList`.

##### 25.5.1 videoTrack object properties

**videoTrack.id** returns the ID of the given track; can be used with a fragment identifier if the format supports the Media Fragments URI syntax, and that can be used with the `getTrackById()` method

**videoTrack.kind** returns the category the given track falls into (see Table 1)

**videoTrack.label** returns the label of the given track, if known, or the empty string otherwise

**videoTrack.language** returns the language of the given track, if known, or the empty string

**videoTrack.selected[=v]** returns `true` if the given track is active, and `false` otherwise; can be set, to change whether the track is selected or not; either zero or one video track is selected; selecting a new track while a previous one is selected will unselect the previous one

#### 25.6 MIMEType and Plugin objects

`MimeType` and `Plugin` objects include `MimeType` and `Plugin`, `MimeTypeArray` and `PluginArray`.

##### 25.6.1 MIMEType and Plugin object properties

`<mimeType>.description` returns the MIME type's description

`<mimeType>.enabledPlugin` returns the `Plugin` object that implements this MIME type.

`<mimeType>.suffixes` returns the MIME type's typical file extensions, in a comma-separated list

`<mimeType>.type` returns the MIME type

`plugin[n]` returns the specified `MimeType` object

`<plugin>[name]` returns the `MimeType` object for the given MIME type

`<plugin>.description` returns the plugin's description

`<plugin>.filename` returns the plugin library's filename, if applicable on the current platform

`<plugin>.length` returns the number of MIME types, represented by `MimeType` objects, supported by the plugin

`<plugin>.name` returns the plugin's name



### 25.6.2 MimeType and Plugin object methods

**<MimeType>=plugin.item(*n*);** returns the specified MimeType object

**<MimeType>=plugin.item(name);** returns the MimeType object for the given MIME type

**<MimeType>=w.navigator.mimeTypes.item(*n*);** returns the specified MimeType object

**<MimeType>=w.navigator.mimeTypes.item(name);** returns the MimeType object for the given MIME type

**<plugin>=w.navigator.plugins.item(*n*);** returns the specified Plugin object

**<plugin>=w.navigator.plugins.item(name);** returns the Plugin object for the plugin with the given name

---

## 26 Table element objects

As well as the `HTMLElement` object some elements can have alternatives to the `HTMLElement` object:

- `HTMLTableElement` for `<table>` elements
- `HTMLTableCaptionElement` for `<caption>` elements
- `HTMLTableColElement` for `<colgroup>` and `<col>` elements
- `HTMLTableSectionElement` for `<tbody>`, `<thead>` and `<tfoot>` elements
- `HTMLTableRowElement` for `<tr>` elements
- `HTMLTableDataCellElement` or `HTMLTableCellElement` for `<td>` elements
- `HTMLTableHeaderCellElement` or `HTMLTableCellElement` for `<th>` elements

### 26.1 Table element properties

**<cell>.cellIndex** returns the position of the cell in the row's cells list or `-1` if the element isn't in a row; this does not neces-

sarily correspond to the x-position of the cell in the table, since earlier cells might cover multiple rows or columns

**table.caption[=*v*]** returns or sets the table's `<caption>` element

**table.rows** returns an `HTMLCollection` array object of the `<tr>` elements of the table

**table.tBodies** returns an `HTMLCollection` array object of the `<tbody>` elements of the table

**table.tFoot[=*v*]** returns or sets the table's `<tfoot>` element

**table.tHead[=*v*]** returns or sets the table's `<thead>` element

**tbody.rows** returns an `HTMLCollection` array object of the `<tr>` elements of the table section

**tr.cells** returns an `HTMLCollection` array object of the `<td>` and `<th>` elements of the row.

**tr.rowIndex** returns the position of the row in the table's rows list or `-1` if the element isn't in a table

**tr.sectionRowIndex** returns the position of the row in the table section's rows list or `-1` if the element isn't in a table section

## 26.2 Table element methods

**table.createCaption();** ensures the table has a `<caption>` element, and returns it

**table.createTBody();** ensures the table has a `<tbody>` element, and returns it

**table.createTFoot();** ensures the table has a `<tfoot>` element, and returns it

**table.createTHead();** ensures the table has a `<thead>` element, and returns it

**table.deleteCaption();** ensures the table does not have a `<caption>` element

**table.deleteRow(*n*);** removes the `<tr>` element with the given position in the table; the index `-1` is equivalent to deleting the last row of the table

**table.deleteTFoot();** Ensures the table does not have a `<tfoot>` element

**table.deleteTHead();** ensures the table does not have a `<thead>` element

**table.insertRow([*n*]);** creates a `<tr>` element, along with a `<tbody>` element if required, inserts them into the table at

the position given by *n*, and returns the `<tr>`; index `-1`, which is the default if the argument is omitted, is equivalent to inserting at the end of the table

**table.stopSorting();** removes all the sorting attributes which are causing a table to sort

**tbody.deleteRow(*n*);** removes the `<tr>` element with the given position in the table section; the index `-1` is equivalent to deleting the last row of the table section

**tbody.insertRow([*n*]);** creates a `<tr>` element, inserts it into the table section at the position given by the argument, and returns the `<tr>`; the index `-1`, which is the default if the argument is omitted, is equivalent to inserting at the end of the table section

**th.sort();** act as if the user had indicated that this was to be the new primary sort column; the table won't actually be sorted until the script terminates

**tr.deleteCell(*n*);** removes the `<td>` or `<th>` element with the given position in the row; the index `-1` is equivalent to deleting the last cell of the row

**tr.insertCell([*n*]);** creates a `<td>` element, inserts it into the table row at the position given by the argument, and returns the `<td>`; the index `-1`, which is the default if the argument is omitted, is equivalent to inserting at the end of the table row

## 27 Form element objects

Apart from `HTMLFormElement`, there are specific media element objects, such as:

- `HTMLFormElement` for `<form>` elements
- `HTMLFieldsetElement` for `<fieldset>` elements
- `HTMLLegendElement` for `<legend>` elements
- `HTMLLabelElement` for `<label>` elements
- `HTMLInputElement` for `<input>` elements
- `HTMLButtonElement` for `<button>` elements
- `HTMLSelectElement` for `<select>` elements
- `HTMLDataListElement` for `<datalist>` elements
- `HTMLOptGroupElement` for `<optgroup>` elements
- `HTMLOptionElement` for `<option>` elements
- `HTMLTextAreaElement` for `<textarea>` elements

Where a `<form>` element has an `id=" "` attribute referenced by the `form=" "` attribute of an `<input>` element, its value becomes the name of the `element` object.

### 27.1 Form element object properties

`<control>.labels` Returns a `NodeList` object of all the `<label>` elements that the form control is associated with.

`datalist.options` returns an `HTMLCollection` array object of the `<option>` elements in the `<datalist>` element

`e.form` returns the element's form owner or `null` if there isn't one

`e.willValidate` returns `true` if the element will be validated when the form is submitted; `false` otherwise

`fieldset.type` returns the string "fieldset".

`fieldset.elements` returns an `HTMLFormControlsCollection` array object of the form controls in the element

`form.elements` returns an `HTMLCollection` array object of the form controls in the form (excluding image buttons for historical reasons)

`form[n]` returns the *n*th element in the form (excluding image buttons for historical reasons)

`form.length` returns the number of form controls in the form (excluding image buttons for historical reasons).

`form[name]` returns the form control (or, if there are several, a `RadioNodeList` of the form controls) in the form with the given ID or name (excluding image buttons for historical reasons); or, if there are none, returns the `<img>` element with the given ID; once an element has been referenced using a particular name, that name will continue being available as a way to reference that element in this method, even if the element's actual ID or name changes, for as long as the element remains in the document; if there are multiple matching items, then a `RadioNodeList` object containing all those elements is returned

`input.checked[=v]` returns or sets the current checkedness of the form control

**input.files** returns a `FileList` object listing the selected files of the form control or `null` if the control isn't a file control.

**input.indeterminate[=*v*]** when set, overrides the rendering of `checkbox` controls so that the current value is not visible

**input.list** returns the `<datalist>` element indicated by the `list` attribute

**input.value[=*v*]** returns or sets the current value of the form control

**input.valueAsDate[=*v*]** returns or sets a `Date` object representing the form control's value, if applicable; otherwise, returns `null`

**input.valueAsNumber[=*v*]** returns or sets a number representing the form control's value, if applicable; otherwise, returns `NaN`<sup>12</sup> which sets the underlying value to the empty string

**input.valueHigh[=*v*]** returns or sets the high component of a form control's value, if applicable; otherwise, returns `NaN` which sets the high component to the empty string

**input.valueLow[=*v*]** returns or sets the low component of a form control's value, if applicable; otherwise, returns `NaN` which sets the low component to the empty string

**label.control** returns the form control that is associated with this element

**legend.form** returns the element's `<form>` element, if any, or `null` otherwise.

**option.form** returns the element's `<form>` element, if any, or `null` otherwise.

**option.index** returns the index of the element in its `<select>` element's options list

**option.selected** returns `true` if the element is selected, and `false` otherwise; can be set, to override the current state of the element

**option.text** same as `textContent`, except that spaces are collapsed and script elements are skipped (see section 23.1 on page 46)

**output.defaultValue[=*v*]** returns or sets the element's current default value

**output.type** returns the string `output`

**output.value[=*v*]** returns or sets the element's current value

**progress.position** returns the result of dividing the current value by the maximum value unless it is an indeterminate progress bar for which it returns `-1`

**select[*n*]** returns the item with index *n* from the list of options; the items are sorted in tree order

**select.length[=*n*]** returns or sets the number of elements in the list of options, if necessary, truncating the number of `<option>` elements in the `<select>` element or adding new blank `<option>` elements to the `<select>` element

**select.options** returns an `HTMLOptionsCollection` of the list of options

**select.selectedIndex[=*n*]** returns or sets the index of the first selected item, if any, or `-1` if there is no selected item

**select.selectedOptions** returns an `HTMLCollection` array object of the options selected

---

<sup>12</sup>(NaN cannot be represented in a floating point number)

**select.type** returns `select-multiple` if the element has a `multiple` attribute, and `select-one` otherwise.

**select.value[=*v*]** returns or sets the value of the first selected item, if any, or the empty string if there is no selected item

**textarea.type** returns the string `textarea`

**textarea.v** returns or sets the current value of the element

## 27.2 Form element object methods

**cell=tr.insertCell([*n*]);** creates a `<td>` element, inserts it into the table row at the position given by the argument, and returns the `<td>`; the index `-1`, which is the default if the argument is omitted, is equivalent to inserting at the end of the row

**form.submit();** submits the form

**form.reset();** resets the form

**form.checkValidity();** returns `true` if the form's controls are all valid; otherwise, returns `false`

**form.reportValidity();** returns `true` if the form's controls are all valid and informs the user; otherwise, returns `false` and informs the user

**form.requestAutocomplete();** triggers `autofill` where the `autocomplete=""` attribute is not set to `on` or `off`

**input.stepUp([*n*]);**

**input.stepDown([*n*]);** Changes the form control's value by the value given in the `step` attribute, multiplied by *n* unless the `step` attribute's value is `any`; the default is `1`

**new Option([*text*[,*value*[,*defaultSelected*[,*selected*]]]]);**

returns an `<option>` element, optionally with `text` as its content, and the `value` and `selected` attributes set; however, the `<option>` element will only be selected if `selected` is `true`

**select.add(*element*[,*before*]);** adds an element to the end of the list, or before the node specified by `before`

**select.item(*n*);**

**select.namedItem(*name*);** returns the first item with ID or name from the list of options or `null` if no element with that ID could be found

---

## 28 The history object

### 28.1 History properties

**e.newURL** returns the URL of the session history entry that is now current

**e.oldURL** returns the URL of the session history entry that was

previously current

**e.persisted** returns `false` if the page is newly being loaded (and the load event will fire); otherwise, returns `true`

**e.returnValue[=*v*]** returns or sets the return value of the event,

that is, the message to show the user

**e.state** returns a copy of the information that was provided to `pushState()` or `replaceState()`

**w.history** returns the joint session history

**w.history.length** returns the number of entries in the joint session history

**w.history.state** returns the current `State` object

## 28.2 History methods

**w.history.back()**; steps back a page (= back button) in the joint session history, or a frame if the user is in a child frame of the page, or does nothing if there is no previous page

**w.history.forward()**; steps forward a page (=forward button) in the joint session history, or a frame if the user is in a child frame of the page, or does nothing if there is no next page

**w.history.go([n])**; steps forwards, or backwards if *n* is negative, through the pages, or frames, in the joint session history; 0 will reload the current page

**w.history.pushState(data,title[,url])**; pushes the given data onto the session history, with the given title, and, if provided and not `null`, the given URL.

**w.history.replaceState(data,title[,url])**; updates the current entry in the session history to have the given data, title, and, if provided and not `null`, URL

---

## 29 The location object

### 29.1 Location object properties

**d.location[=v]** or

**w.location[=v]** returns or sets the `Location` object of the `Window` which has the properties:

- **href** the URL of the window, which is decomposed into
  - **protocol** for example, `https`
  - **host**
  - **hostname**
  - **port**
  - **pathname**

hence:

**w.location.href[=v]** returns, or sets, the current page's location

- **search** enables navigation within a window
- **hash** holds the application's state; setting this property allows the contents of the page to be changed without changing the page

Note: `location.toString()` returns only the **href** property; so `location.toString()` and `location.href.toString()` are equivalent.

Changing the value of location with:

```
location = "<URL>";
```

or

```
location = "#top";
```

enables document navigation.<sup>13</sup>

Note: Javascript scripts are not allowed to access most of the properties of `Window` objects, in particular the `Document` object, with a different `location.protocol`, `location.host` or `location.port` property (aka *same origin policy*); so

changing the values in the `location` properties may prevent a script from running.

## 29.2 Location object methods

**location.assign(url)**; navigates to the given page.

**location.reload()**; reloads the current page

**location.replace(url)**; removes the current page from the session history and navigates to the given page

---

## 30 The navigator object

### 30.1 Navigator object properties

**w.navigator** refers to the `Navigator` object of the `Window` which has the properties:

- **appName** the full name of the browser
- **appVersion** the version of the browser (there is no standard for this property)
- **geolocation** refers to the `Geolocation` of the window
- **onLine** returns `true` if the browser is online
- **platform** the operating system
- **userAgent** the contents of the `USER-AGENT` HTTP header (there is no standard for this property)

**w.navigator.appCodeName** returns the string `Mozilla`

**w.navigator.appName** returns the full name of the browser

**w.navigator.appVersion** returns the version of the browser

**w.navigator.cookieEnabled** returns `false` if setting a cookie will be ignored, and `true` otherwise

**w.navigator.javaEnabled** returns `true` if there's a plugin that supports the MIME type `application/x-java-vm`

**w.navigator.language** returns a language tag representing the user's preferred language

**w.navigator.languages** returns an array of language tags where the user's preferred language comes first

**w.navigator.mimeTypes[n]** returns the specified `MimeType` object

**w.navigator.mimeTypes[name]** returns the `MimeType` object for the given MIME type

<sup>13</sup>`#top` does not work if an HTML element has `id="top"`.

**w.navigator.mimeTypes.length** returns the number of MIME types, represented by `MimeType` objects, supported by the plugins that the user agent reports

**w.navigator.onLine** returns `false` if the user agent is definitely offline (disconnected from the network) or `true` if the user agent might be online; the events `online` and `offline` are fired when the value of this attribute changes

**w.navigator.platform** returns the name of the operating system

**w.navigator.plugins[n]** returns the specified `Plugin` object

**w.navigator.plugins[name]** returns the `Plugin` object for the plugin with the given name

**w.navigator.plugins.length** returns the number of plugins, represented by `Plugin` objects, that the user agent reports

**w.navigator.product** returns the string `Gecko`

**w.navigator.userAgent** returns the complete `USER-AGENT` HTTP header

## 30.2 Navigator object methods

`<state>=w.navigator.isContentHandlerRegistered(mimeType,url)`; returns one of the following strings describing the state of the handler given by the arguments:

**new** indicates that no attempt has been made to register the given `handler` (or that the `handler` has been unregistered); it would be appropriate to promote the availability of the `handler` or to just automatically register the `handler`

**registered** indicates that the given `handler` has been registered or that the site is blocked from registering the `handler`; trying to register the `handler` again would have no effect

**declined** indicates that the given `handler` has been offered but was rejected; trying to register the `handler` again may prompt the user again

see page [73](#) for the blacklisted values of `MimeType`

`<state>=w.navigator.isProtocolHandlerRegistered(scheme,url)`; see page [72](#) for the permitted values of `scheme`

**w.navigator.mimeTypes.item(n)** returns the  $n$ th `MimeType` object

**w.navigator.mimeTypes.item(name)** returns the `MimeType` object with the specified name

**w.navigator.plugins.item(n)** returns the  $n$ th `Plugin` object

**w.navigator.plugins.item(name)** returns the `Plugin` object for the plugin with the given name

**w.navigator.plugins.refresh([refresh])**; updates the lists of supported plugins and MIME types for this page, and reloads the page if the lists have changed

**w.navigator.registerProtocolHandler(scheme,url,title)** registers



a handler for the given scheme or content type, at the given URL, with the given title; the mandatory string %s in the URL is used as a placeholder for where to put the URL of the content to be handled; scheme may be a string starting web+ or one of:

- bitcoin
- geo
- im
- irc
- ircs
- magnet
- mailto
- mms
- news
- nntp
- sip
- sms
- smsto
- ssh
- tel
- urn
- webcal
- wtai
- xmpp

#### **w.navigator.registerContentHandler(mimeType,url,title)**

registers a handler for the given scheme or content type, at the given URL, with the given title; the mandatory string %s in the URL is used as a placeholder for where to put the URL of the content to be handled; mimeType may be

- application/rss+xml
- application/atom+xml

it may NOT be one of:

**<state>=w.navigator.unregisterProtocolHandler(scheme,url)**

**<state>=w.navigator.unregisterContentHandler(mimeType,url)** unregisters the handler given by the arguments

- application/x-www-form-urlencoded
- application/xhtml+xml
- application/xml
- image/gif
- image/jpeg
- image/png
- image/svg+xml
- multipart/x-mixed-replace
- text/cache-manifest
- text/css
- text/html
- text/ping
- text/plain
- text/xml

REPEAT: apart from the first two the remaining mimeTypees are **blacklisted**.

**w.navigator.taintEnabled()** returns false

**w.navigator.yieldForStorageUpdates()** if a script uses the `document.cookie` API, or the `localStorage` API, the browser will block other scripts from accessing cookies or storage until the first script finishes; calling the `navigator.yieldForStorageUpdates()` method tells the

user agent to unblock any other scripts that may be blocked, even though the script hasn't returned; values of cookies and items in the `Storage` objects of `localStorage` attributes can change after calling this method, whence its name.

## 31 Drop and drag

### 31.1 Drop and drag properties

`<dataTransfer>.dropEffect[=v]` returns or sets the kind of operation that is currently selected; the possible values are:

- `none` no operation allowed; dropping here will cancel the drag-and-drop operation
- `copy` data will be copied if dropped here
- `link` data will be linked if dropped here
- `move` data will be moved if dropped here

`<dataTransfer>.effectAllowed[=v]` returns or sets the kinds of operations that are to be allowed; the possible values are:

EFFECTALLOWED	DROPEFFECT
<code>none</code>	<code>none</code>
<code>copy</code>	<code>copy</code>
<code>copyLink</code>	<code>copy</code> , or, if appropriate, <code>link</code>
<code>copyMove</code>	<code>copy</code> , or, if appropriate, <code>move</code>
<code>all</code>	<code>copy</code> , or, if appropriate, either <code>link</code> or <code>move</code>
<code>link</code>	<code>link</code>
<code>linkMove</code>	<code>link</code> , or, if appropriate, <code>move</code>
<code>move</code>	<code>move</code>
<code>uninitialized</code>	if a selection from a text field, <code>move</code> , or, if appropriate, either <code>copy</code> or <code>link</code>
	if a selection, <code>copy</code> , or, if appropriate, either <code>link</code> or <code>move</code>
	if an <code>&lt;a&gt;</code> element with an <code>href=" "</code> attribute, <code>link</code> , or, if appropriate, either <code>copy</code> or <code>move</code>
<code>any other case</code>	<code>copy</code> , or, if appropriate, either <code>link</code> or <code>move</code>

`<dataTransfer>.files` returns a `FileList` object of the files being

dragged, if any

**<dataTransfer>.items** returns a `DataTransferItemList` object, with the drag data

**<dataTransfer>.types** returns an array listing the formats that were set in the `dragstart` event; in addition, if any files are being dragged, then one of the types will be the string `Files`

**delete <items[n]** removes the *n*th entry in the drag data store

**e.dataTransfer** returns the `DataTransfer` object for the event

**<item>.kind** returns the drag data item kind, one of:

**string** a plain Unicode string

**file** a file

**<item>.type** returns the drag data item type string

**<items>[n]** returns the `DataTransferItem` object representing the *n*th entry in the drag data store

**<items>.length** returns the number of items in the drag data store

## 31.2 Drop and drag methods

**<data>=dataTransfer.getData(format)**; returns the specified data or the empty string; `format` may be `text` or `url`

**<dataTransfer>.clearData([format])**; removes the data of the specified formats or all if `format` is omitted; `format` may be `text` or `url`

**<dataTransfer>.setData(format,data)**; adds the specified data; `format` may be `text` or `url`

**<dataTransfer>.setDragImage(element,x,y)**; uses the given element to update the drag feedback, replacing any previously specified feedback

**<file>=<item>.getAsFile()**; returns a `File` object, if the drag data item kind is `File`

**<items>.add(data[,type])**; adds a new entry for the given data to the drag data store object; if the data is plain text then a `type` string has to be provided also

**<items>.clear()**; removes all the entries in the drag data store object

**<items>.remove(n)** removes the item which is *n*th in the drag data store object

**<item>.getString(callback)**; invokes the `callback` with the string data as the argument, if the drag data item kind is `Plain Unicode string`

## 32 Event handlers

Event handler objects include `Event`, `EventHandler`, `EventSource`, `EventTarget`, `ErrorEvent`; they respond to an event and always have a function object or `null` as their value; their function objects always have the properties:

- `type` the type of event on which they operate
- `target` the object associated with the event.

and may have

- `isTrusted` a Boolean attribute which may be `true` or `false`

and other properties relevant to their `type` holding, for example, the coordinates of a mouse movement or the value of the key pressed.

Some function objects generate a single default action, for example loading a page when a URL is clicked.

While variables declared within most function objects have local scope, those declared within function objects whose event handler is registered as an HTML attribute have global scope (and thus no access to local variables).

Each event has a `type` and a `name`, which is a string describing the event, for example

- `application cache` *cached*, *checking*, *downloading*, *error*, *noupdate*, *obsolete* (onobsoletechecking), *progress*, *updateready*
- `click` *click*, *contextmenu*, *dblclick*; the property `detail` returns whether it was a single, double or triple click; a click event following immediately by another generates a *dblclick* event

- `document` (and `window`) *readystatechange* (document only), *reset*, *submit*
- `drag and drop` *drag*, *dragend*, *dragenter*, *dragexit*, *dragleave*, *dragover*, *dragstart*, *drop*

Drag and drop events are normally associated with event objects generated by `mousemove` events; these have the property `dataTransfer` which refers to the object holding the information about the data to be transferred.

- `form` *blur*,\* *focus*,\* *changes*, *reset*, *submit*<sup>14</sup>
- `global` *abort*, *blur*, *cancel*, *change*, *close*, *error*,\* *focus*, *input*, *invalid*, *load*,\* *scroll*,\* *select*, *show*, *sort*<sup>15</sup>
- `keyboard` *keydown*, *keypress*, *keyup*; *keydown* events involving a printable character generate a *keypress* event which sets the value of the `keyCode` property holding the value of the key pressed along with the key modifier properties:
  - `altKey` returns `true` if the Alt key was pressed
  - `ctrlKey` returns `true` if the Ctrl key was pressed
  - `metaKey` returns `true` if the Meta key was pressed
  - `shiftKey` returns `true` if the Shift key was pressed
- `media` *addtrack*, *canplay*, *canplaythrough*, *cuechange*, *change*, *durationchange*, *emptied*, *ended*, *enter*, *exit*, *loadeddata*, *loadedmetadata*, *loadstart*, *pause*, *play*, *playing*, *progress*, *ratechange*, *removetrack*, *resize*, *seeked*, *seeking*, *stalled*, *suspend*, *timeupdate*, *volumechange*, *waiting*

---

<sup>14</sup>\* has a different meaning when specified on `<body>` and `<frameset>` elements as those elements expose event handlers of the Window object with the same names.

<sup>15</sup>\* see note above.

- **mousemove** *mousedown*, *mouseup* (a *mousedown* followed by a *mouseup* event generates a **click** event), *mouseover* (deprecated), *mouseenter*, *mouseout* (deprecated), *mouseleave*, *mousewheel*; all have the key modifier properties of the keyboard type plus:
  - **clientX** returns the *x* position of the mouse in window coordinates
  - **clientY** returns the *y* position of the mouse in window coordinates

*mouseout/mouseleave* events have a **relatedTarget** property which returns the element over which the mouse is now hovering and *mousewheel* events have properties for the amount and direction of the wheel movement
- **touchscreen** *touchend*, *touchmove*, *touchstart*; these have the properties:
  - **changedTouches** an array describing the positions of a touch
  - **rotation** the clockwise angle in degrees of the rotation of the finger involved between the initial and current position
  - **scale** the ratio  $\frac{\text{current distance}}{\text{initial distance}}$  of the two fingers involved in a gesture; a value of <1 signifies a *pinch close* event; otherwise, it is a *pinch open* event
- **web workers and web sockets** *close*, *connect*, *message*, *offline*, *online*, *open*
- **window** *afterprint*, *autocomplete*, *autocompleteerror*, *beforeprint*, *beforeunload*, *hashchange*, *languagechange*, *message*, *offline*, *online*, *pagehide*, *pageshow*, *popstate*, *resize*, *scroll*, *storage*, *toggle*, *unload*

To associate an event handler with an event, preface the name of the event with *on*, for example,

**e.onblur** initiate action when the user leaves a field

**e.onchange** initiate action when the user changes field

**e.onclick** initiate action when the user clicks on an input

**e.onfocus** initiate action when the user goes to a form field

**d.onload** initiate action when a document has loaded

Apart from *blur*, *focus*, *mouseenter*, *mouseleave* and *scroll*, most event handlers associated with a document or document element become associated with its ancestor elements, for example,

**e.region**; returns the **id** of a hit region when the mouse cursor is over it or **null** when it is not.

**e.relatedTarget** returns the other event target involved in this event; for example, when a **show** event fires on a `<menu>` element, the other event target involved in the event would be the element for which the menu is being shown

**e.returnValue[=*v*]** returns or sets the current return value of the event (the message to show the user)

**e.track** returns the track object (**TextTrack**, **AudioTrack**, or **VideoTrack**) to which the event relates.

### 32.1 The DOMEvents object

The **DOMEvents** object holds the properties of **click** and **mousemove** events:

- **click** the event
- **UIEvent** the value of the **detail** attribute

- `MouseEvent` the interface
- `MouseEventInit` the dictionary type

## 32.2 Event methods

**`e.addEventListener(e,f[,<true/false>])`**; executes `f` on `<event>` or captures it if `true` (capturing involves intercepting the event before its function is executed)

**`e.preventDefault()`**; prevents the default function associated with an event from executing

**`e.removeEventListener(e,f[,<true/false>])`**; removes the specified event listener

**`e.stopPropagation()`**; prevents an event handler from propagating at any point in the propagation

**`e.stopImmediatePropagation()`**; prevents all event handlers from propagating at any point in the propagation

## 32.3 Error event object

The `DOMException` object is the string object which may take one of the values:

- `IndexSizeError`
- `HierarchyRequestError`
- `WrongDocumentError`
- `InvalidCharacterError`
- `NoModificationAllowedError`
- `NotFoundError`

- `NotSupportedError`
- `InvalidStateError`
- `SyntaxError`
- `InvalidModificationError`
- `NamespaceError`
- `InvalidAccessError`
- `SecurityError`
- `NetworkError`
- `AbortError`
- `URLMismatchError`
- `QuotaExceededError`
- `TimeoutError` (whose code is the number 23, for legacy reasons)
- `InvalidNodeTypeError`
- `DataCloneError`

## 32.4 HTTP events

**`readystatechange`** reports when the `readyState` property of an `XMLHttpRequest` function object changes; its function object properties include:

- `lengthComputable` returns `true` if `total != 0`
- `loaded` the number of bytes transferred so far
- `total` the total number of bytes to be transferred; a value of 0 means that the total length is not known

## 32.5 Message events

Message events are used in communications by WebWorkers and WebSockets. Their event objects have the properties:

**broadcastChannel.name** returns the name of the BroadcastChannel object

**channel.port1** returns the first MessagePort object

**channel.port2** returns the second MessagePort object

**e.data** returns the data of the message

**e.origin** returns the origin of the message, for server-sent events and cross-document messaging

**e.lastEventId** returns the last event ID string, for server-sent events

**e.ports** returns the MessagePort array sent with the message, for cross-document messaging and channel messaging

**e.reason** returns the general reason for the failure of the `requestAutocomplete()` which may be:

- "" the reason is unknown
- cancel the user cancelled it
- disabled autofill is disabled
- invalid the entry does not satisfy the form's constraints

**e.source** returns the WindowProxy of the source Window object, for cross-document messaging, and the MessagePort being attached, in the connect event fired at SharedWorkerGlobalScope objects.

## 32.6 Message methods

**broadcastChannel.postMessage(<message>);** sends the message to other BroadcastChannel objects

**broadcastChannel.close();** closes the BroadcastChannel object

**port.close();** disconnects the port, so that it is no longer active

**port.postMessage(message[,transfer]);** posts a message through the channel; objects listed in transfer are transferred, not just cloned, meaning that they are no longer usable on the sending side

**port.start();** begins dispatching messages received on the port

**portCollection.add(port);** adds port to the collection, if it isn't already present

**portCollection.clear();** removes all ports from the collection.

**portCollection.iterate(callback);** Calls callback for each port in the collection

**portCollection.remove(port);** removes port from the collection, if it is present

## 32.7 Message constructors

**<broadcastChannel>=new BroadcastChannel** Returns a new BroadcastChannel object

**<channel>=new MessageChannel();** returns a new MessageChannel object with two new MessagePort objects

**<portCollection>=new PortCollection();** returns a new empty PortCollection object

## 32.8 Server sent events

To enable servers to push data to Web pages over HTTP or using dedicated server-push protocols, create an `EventSource` object with:

```
var source = new EventSource(v);
```

where `v` is the URL of the script providing the event messages; it has the attributes:

- `withCredentials` a Boolean attribute which, when initialised, is set to `false`

- `readyState` which can have the following values:

**CONNECTING(0)** the connection has not yet been established, or is reconnecting (the initial state)

**OPEN(1)** the connection is dispatching events as it receives them.

**CLOSED(2)** the connection is not open, and the user agent is not trying to reconnect; either there was a fatal error or the `close()` method was invoked

---

## 33 HTTP Requests

HTTP Requests are made using `XMLHttpRequest` function object (which may, but in a Javascript context rarely does, use XML); new requests should always be made using the constructor:

```
new XMLHttpRequest();
```

to avoid interfering with anything currently being handled by the host object or any other similar object; it has the attributes:

- `readyState` the value of the HTTP Request's state which may be

**0** `open()` has not been called

**1** `open()` has been called

**2** headers have been received

**3** response body has been received

**4** response has been received

- `responseText` the body of the response text
- `status` the value of an HTTP status response
- `statusText` the text of an HTTP status response
- `upload` the value of an object that defines an `addEventListener()` method and its progress event properties for an upload event

### 33.1 HTTP methods

**`h.getResponseHeader()`**; returns the contents of the response header

**`h.getAllResponseHeaders()`**; returns the contents of all the response headers

**`h.open("<method>","<url>")`**; initiates an HTTP request where `<method>` may be:



GET  
POST  
DELETE  
HEAD  
OPTIONS

PUT

`h.setRequestHeader("Content type","text/plain");` sets the request header for the POST method

`h.send("<message>");` sends the body of the message, in the case of the GET method, this is `null`.

---

## 34 ValidityState object

### 34.1 ValidityState object properties

The ValidityState object properties are all Boolean attributes:

- `valueMissing`
- `typeMismatch`
- `patternMismatch`
- `tooLong`
- `rangeUnderflow`

- `rangeOverflow`
- `stepMismatch`
- `badInput`
- `customError`
- `valid` true if none of the other conditions are true

### 34.2 ValidityState object methods

`o.checkValidity();`

# Part III

## Additional Javascript features

### 35 Application cache objects

An application cache enables a browser to cache files on a user's device so that they are available even when the device is offline. The cache host is a `Document` or a `SharedWorkerGlobalScope` object with an associated `ApplicationCache` object.

The cache manifest always starts with

```
CACHE MANIFEST
```

and may contain up to four sections, `FALLBACK`, `NETWORK`, `CACHE` and `SETTINGS`, for example,

```
CACHE MANIFEST
FALLBACK:
/
/offline.html
NETWORK:
*
CACHE:
/example.css
/example.js
/example.jpg
SETTINGS:
*
```

In this example, the first `/` in the `FALLBACK` section is a wildcard for any page the user had already cached; the second entry is an alternative page if that fails. The `NETWORK` section may contain an online whitelist but `*` says that anything may be downloaded. The `CACHE` section contains the files that must be downloaded.

#### 35.1 Application cache object properties

`<cache>=<self>.applicationCache` (In a shared worker.) returns the `ApplicationCache` object that applies to the current shared worker

`cache.status` returns the current status of the application cache, as given by the constants defined below:

**UNCACHED(0)** the `ApplicationCache` object's cache host is not associated with an application cache at this time

**IDLE(1)** the `ApplicationCache` object's cache host is associated with an application cache whose application cache group's update status is `idle`, and that application cache is the newest cache in its application cache group, and the application cache group is not marked as `obsolete`

**CHECKING(2)** the `ApplicationCache` object's cache host is associated with an application cache whose application cache group's update status is `checking`

**DOWNLOADING(3)** the `ApplicationCache` object's cache host is associated with an application cache whose application cache group's update status is `downloading`.

**UPDATEREADY(4)** the `ApplicationCache` object's cache host is associated with an application cache whose application cache group's update status is `idle`, and whose application cache group is not marked as `obsolete`, but that application cache is not the newest cache in its group.

**OBSOLETE()** the `ApplicationCache` object's cache host is associated with an application cache whose application cache group is marked as `obsolete`

`<cache>=w.applicationCache` (In a window.) returns the `ApplicationCache` object that applies to the active document of that window

### 35.2 Application cache object methods

**cache.abort();** cancels the application cache download process; intended for Web applications showing their own caching progress UI, in case the user wants to stop the update (e.g. because bandwidth is limited)

**cache.swapCache();** Switches to the most recent application cache, if there is a newer one, but this does not cause previously-loaded resources to be reloaded; the `updateready` event will fire before this method can be called; once it fires, the Web application can, at its leisure, call this method to switch the underlying cache to the one with the more recent

updates; an easier alternative to `swapCache()` is just to reload the entire page at a time suitable for the user, using `location.reload()`

**cache.update();** invokes the application cache download process; usually unnecessary, as user agents will generally take care of updating application caches automatically, but can be useful for long-lived applications which might stay open in a browser tab for weeks at a time and need to test for updates each day.

### 35.3 Application cache events

**checking** loading the cache manifest for the first time

**noupdate** the cache manifest has not changed

**downloading** loading updates to the cache manifest

**progress** returns the `total` number of items to be downloaded and the number `loaded` so far

**cached** resources are ready for use

**updateready** updates ready for use

**obsolete** page to which cache manifest relates has been deleted; cache manifest will be delete

## 36 Web workers

Web worker objects include: `Worker`, `WorkerGlobalScope`, `WorkerLocation`, `WorkerNavigator`, `DedicatedWorkerGlobalScope`, `SharedWorker`, `SharedWorkerGlobalScope`, `MessageChannel`, `MessagePort` and `PortCollection`.

Web workers are long running scripts; they are expected to be heavy weight and not to be around in large numbers.

### WebWorker constructors

**`var <worker>=new Worker(v)`**; initiates a new worker object where `v` is a Javascript file containing the script.

Creating a `Worker` object creates

- a `WorkerGlobalScope` object with two attributes:
  - `self` which returns itself
  - `location` which returns the `WorkerLocation` object representing the URL of the script used to initialize the worker, and
- a `DedicatedWorkerGlobalScope` object with an implicit `MessagePort` object

Each `WorkerGlobalScope` object has

- a list of the all the `MessagePort` objects that are entangled with another port and that have one (but only one) port owned by worker global scope; this list includes the implicit `MessagePort` in the case of dedicated workers
- a list of the worker's workers; initially this list is empty; it is populated when the worker creates or obtains further workers
- a list of the worker's documents; initially this list is empty; it is populated when the worker is created and dynamically updated

- a `navigator` attribute which points to a `WorkerNavigator` interface identifying the browser being used.

A `Worker` is

- a `permissible` worker if the list of the worker's documents is not empty
- a `protected` worker if it is a `permissible` worker and either
  - it has outstanding timers, database transactions, or network connections, or its list of the worker's ports is not empty, or
  - it is a shared worker
- an `active needed` worker if any of the `Document` objects in the worker's documents are fully active
- a `suspendable` worker if it is not an `active needed` worker but it is a `permissible` worker

**`var <worker>=new SharedWorker(v)`**; initiates a new shared worker object

A `SharedWorker` object is associated with a `MessagePort` object and the `SharedWorkerGlobalScope` object has the attribute `name` which is used to associate shared workers.

To receive messages from a worker use

```
<worker>.onmessage=function(event){...};
```

To send messages to a worker use

```
<worker>.postMessage('...');
```

To receive from and post messages to a shared worker, you need to add the port, for example:

```
<worker>.<port>.onmessage=function(event){...};
```

```
<worker>.<port>.addEventListener(e,f,<true/false>)];
```

Dedicated workers use `MessagePort` objects each of which has a port message queue.

---

## 37 Web sockets

The `WebSocket` constructor function enables a page to establish a connection with an external page:

```
new WebSocket("ws://ws.<file>"[,<protocols>]);
```

in order to execute a script; for a secure `WebSocket`, use:

```
new WebSocket("wss://ws.<file>"[,<protocols>]);
```

It has the properties:

- `extensions`
- `protocol`
- `readyState` which can have the following values:

**CONNECTING(0)** the connection has not yet been established (the initial value)

**OPEN(1)** the connection is established and communication is possible

**CLOSING(2)** the connection is going through the closing handshake, or the `close()` method has been invoked

**CLOSED(3)** the connection has been closed or could not be opened

### 37.1 Web socket methods

`<ws>.close();` closes the connection

`<ws>.send(v);` sends `v` over a web socket; `v` may be a string, a `Blob` object, an `ArrayBuffer` object or an `ArrayBufferView` object

## 38 Web storage

A `Storage` object provides access to a list of key/value pairs, sometimes called items, both of which are strings; multiple separate objects can all be associated with the same list of key/value pairs simultaneously.

Local storage is possible where a web browser's `Window` object has the properties `localStorage` and `sessionStorage`. These properties hold the values of the `Storage` objects created on the user's own computer either permanently in the case of `localStorage` or as long as a tab is opened in the case of `sessionStorage`.

Note that

- different browsers have different `Window` objects and therefore local storage objects accessible to one `Window` object will not be accessible to another
- local `Storage` objects associated with a URL will be accessible to all authors able to access that URL
- session `Storage` objects accessible to one top level tab of a browser will not be accessible to another
- no encryption or security is available and so sensitive data should never be stored in storage objects.

### 38.1 Storage properties

**s.key** returns the item that was set or removed; its value is `null` after `s.clear()`;

**s.length** returns the number of key/value pairs currently present in the list associated with the object

**s.newValue** returns the most recently set value; its value is `null` after `s.removeItem()`;

**s.oldValue** returns the most recently changed value; its value is `null` after `s.setItem()`; has created a new item

**s.storageArea** returns the value of the `Window` object property `localStorage` or `sessionStorage`

**s.url** the URL in string format of the document responsible for the change

### 38.2 Storage methods

**s.clear()**; deletes all the values held in a `Storage` object

**s.getItem(k)**; returns the value of item `k`

**s.key(n)** returns the key of the `n`th pair in the list or `null`

**s.removeItem(k)**; removes item `k` or does nothing if no item with that key exists

**s.setItem(k,v)**; stores `v` to item `k`; if `k` already exists, updates `v`; otherwise, adds another key/value pair to the object

## References

- Ecma International (2011, June). *ECMAScript language specification* (5.1 ed.). Geneva: Ecma International. <http://www.ecma-international.org>.
- Flanagan, D. (2011). *JavaScript: the definitive guide* (Sixth ed.). Sebastapol, CA: O'Reilly Media.
- Flanagan, D. (2012). *JavaScript pocket reference* (Third ed.). Sebastapol, CA: O'Reilly Media.
- Hickson, I. (editor). (2013, 12 July). HTML: living standard. <http://www.whatwg.org/specs/web-apps/current-work/>.
- Hickson, I. (editor). (2014, 6 August). HTML: living standard. <http://www.whatwg.org/specs/web-apps/current-work/>.

## A Reserved words

Words have been reserved in various versions of Javascript; some are reserved in strict mode and some for certain use cases. This list simply lists them all whether or not they are reserved in a particular version of Javascript.

abstract	continue	false	instanceof	public	true
arguments	debugger	final	int	return	try
boolean	default	finally	interface	short	typeof
break	delete	float	let	static	var
byte	do	for	long	super	void
case	double	function	native	switch	volatile
catch	else	goto	new	synchronized	while
char	enum	if	null	this	with
class	eval	implements	package	throw	yield
const	export	import	private	throws	
	extends	in	protected	transient	

## B Built-in operators

The built-in operators, from high to low precedence, are ++ -- - + [unary] ~ ! delete typeof void \* / % + - << >> >>> < <= > >= instanceof in == != === !== & ^ | && || ? : = += and the remaining compound operators. Note that = ++ -- and delete always have side effects on a program.

**new** calls a built-in constructor to create a new object based its prototype (section 6.3).

**delete()** deletes the value of a declared property, not that property; where the *configurable* attribute of that property is **false**, it throws a **TypeError**; it does not delete the values of inherited properties, only those declared for the object. *Deleting the value of a property in the prototype object affects all objects inherited from that prototype.*

**typeof()** returns one of "undefined", "boolean", "number", "string", "object" or "function"; null is regarded as an object and an object with the call property is regarded as a function.

**void()** discards the value of its operand, thereby making it **undefined**.

### B.1 Arithmetic and string operators

**++** increment;  $n++$  is the same as  $n = n + 1$  or

converts a value to a number (**Number()**) and increments it

**--** decrement;  $n--$  is the same as  $n = n - 1$  or



converts a value to a number (`=Number()`) and decrements it

+ concatenation (where at least one value is a string; `=String()`) or

addition, in which case any operation involving `+0` has a positive sign unless the other is a nonzero negative value and `(+Infinity) + (-Infinity) = NaN` as does any operation involving `NaN` or

if unary, converts a value to a number (`=Number()`)

- negation or

if unary and not `NaN`, converts a value to a negative number (`=Number()`)

## B.2 Arithmetic only operators

\* multiply; note that `Infinity * 0 = NaN` as does any operation involving `NaN`

- divide; note that `Infinity/Infinity = NaN` and `0/0 = NaN` as does any operation involving `NaN`

% modulus; note that dividing `Infinity`, dividing by zero or any operation involving `NaN` gives the result `NaN`

## B.3 Relational operators

These return Boolean values unless one side is `NaN` in which case they return `undefined`.

< less than

<= less than or equal to

<sup>16</sup> `!== undefined` returns `true` only for declared properties.

> more than

>= more than or equal to

**in** returns `true` if a string matches the name of any property, inherited or declared,<sup>16</sup> in an object or `TypeError` if it is not an object; otherwise, it returns `false`.

**instanceof** returns `true` if a relational expression matches the value of the `hasInstance` property of a function or `TypeError` if the object is not a function; otherwise, it returns `false`.

## B.4 Equality operators

== equality

!= inequality

The equality operator returns `false` for any comparison involving `NaN` but `true` when

- `null` and `undefined` are compared
- a `toNumber()` operation on a string or a Boolean yields an equality or
- the value of an object's `primitiveValue` internal property is the same as a number or a string.

If you wish to force a particular type comparison, precede each value with

`" "+` for a string comparison

`+` for a number comparison

`!` for a Boolean comparison.

`===` strict equality

`!==` strict inequality

The strict equality operator returns `false` if

- the values being compared are of different types
- the values are different, except where one is `+0` and the other `-0` or
- one is `NaN`

but `true` if both values are the same or one value is `undefined` or `null`.

## B.5 Bitwise operators

Bitwise operators return a signed 32-bit integer.

`&` AND

`^` XOR

`|` OR

`~` NOT

`<<` shift left

`>>` shift right with sign extension

`>>>` shift right with zero extension

`&= n&y` is the same as `n=n AND y`  
and so on for `^` `|` `<<` `>>` and `>>>`

## B.6 Logical operators

`&&` AND

`||` OR

`!` NOT (i.e. invert Boolean value)

`!!` converts to Boolean (`=Boolean()`)

## B.7 Conditional operator

`?` : if the expression before `?` returns `true`, return the value of the expression before the `::`; otherwise return the value of the expression after the `::`; for example, the absolute value of `x` can be returned with:

```
x>0?x:-x;
```

## B.8 Assignment operators

`=` assignment

`+=` `n+=y` is the same as `n=n+y`

and so on for `-` `*` `/` and `%`

## B.9 Comma operator

`,` discard the result of the expression before the comma and use the result of the expression after it (most often used in `for` loops)

## B.10 Date operators

**MakeTime(hour,min,sec,ms)** returns the number of milliseconds represented by the arguments

**MakeDay(year,month,date)** returns the number of days represented by the arguments

**MakeDate(day,time)** returns the number of milliseconds represented by the arguments

**TimeClip(time)** returns the number of milliseconds represented by the argument

## Object index

### A

ApplicationCache, [37](#), [82](#)  
ArrayBuffer, [34](#), [85](#)  
ArrayBufferView, [85](#)  
AudioTrack, [58](#), [61](#), [77](#)  
AudioTrackList, [58](#), [61](#)

### B

BarProp, [37](#)  
Blob, [33](#), [34](#), [85](#)  
Boolean, [18](#)  
BroadcastChannel, [79](#)

### C

CanvasDrawingStyles, [47](#)  
CanvasGradient, [47–49](#)  
CanvasPathMethods, [47](#)  
CanvasPattern, [47–49](#)  
CanvasRenderingContext2D, [33](#), [47](#), [48](#)  
CSSStyleDeclaration, [41](#), [43](#)

### D

DataTransfer, [75](#)  
DataTransferItem, [75](#)  
DataTransferItemList, [75](#)  
Date, [20](#), [68](#)  
DedicatedWorkerGlobalScope, [84](#)  
Document, [32](#), [35–37](#), [40](#), [71](#), [82](#), [84](#)  
DocumentFragment, [44](#)  
DOMElementMap, [42](#)  
DOMEvents, [77](#)  
DOMException, [78](#)

DOMStringMap, [42](#)  
DrawingStyle, [47](#), [50](#)

### E

Error, [25](#)  
ErrorEvent, [76](#)  
EvalError, [25](#)  
Event, [76](#)  
EventHandler, [76](#)  
EventSource, [76](#), [80](#)  
EventTarget, [76](#)

### F

File, [34](#), [75](#)  
FileList, [34](#), [68](#), [74](#)

### G

Geolocation, [71](#)

### H

History, [37](#), [69](#)  
HTMLCollection, [36–40](#), [44](#), [67](#), [68](#)  
HTMLElement, [41](#), [57](#), [65](#)  
HTMLFormControlsCollection, [38](#), [67](#)  
HTMLFormsCollection, [39](#), [40](#)  
HTMLOptionsCollection, [38–40](#), [68](#)  
HTMLPropertiesCollection, [38](#), [40](#), [42](#)

### I

ImageBitmap, [33](#)  
ImageData, [33](#), [57](#)

## **J**

JSON, [25](#)

## **L**

Location, [37](#), [70](#)

## **M**

Math, [19](#)

MediaController, [46](#), [58](#), [59](#)

MediaError, [59](#)

MessageChannel, [79](#), [84](#)

MessagePort, [79](#), [84](#), [85](#)

Microdata, [37](#), [42](#)

MimeType, [64](#), [71](#), [72](#)

MimeTypeArray, [64](#)

## **N**

Navigator, [37](#), [71](#)

NodeList, [35](#), [38](#), [40](#), [67](#)

Number, [19](#)

## **P**

Path, [47](#), [55](#)

Plugin, [64](#), [72](#)

PluginArray, [64](#)

PortCollection, [79](#), [84](#)

PropertyNodeList, [39](#), [40](#)

## **R**

RadioNodeList, [39](#), [67](#)

RangeError, [25](#)

ReferenceError, [25](#)

RegExp, [23](#)

## **S**

Screen, [32](#)

SharedWorker, [84](#)

SharedWorkerGlobalScope, [79](#), [82](#), [84](#)

State, [70](#)

Storage, [32](#), [36](#), [37](#), [74](#), [86](#)

String, [16](#)

SyntaxError, [25](#)

## **T**

TextMetrics, [47](#), [54](#)

TextTrack, [60](#), [63](#), [77](#)

TextTrackCue, [63](#)

TextTrackCueList, [63](#)

TextTrackList, [63](#)

TimeRanges, [58](#), [60](#)

TypeError, [25](#)

## **U**

URIError, [25](#)

## **V**

ValidityState, [81](#)

VideoTrack, [60](#), [64](#), [77](#)

VideoTrackList, [60](#), [64](#)

## **W**

WebGLRenderingContext, [47](#)

Window, [31](#), [33](#), [36](#), [37](#), [40](#), [70](#), [71](#), [79](#), [86](#)

WindowProxy, [32](#), [33](#), [79](#)

Worker, [84](#)

WorkerGlobalScope, [84](#)

WorkerLocation, [84](#)

WorkerNavigator, [84](#)

## **X**

XMLHttpRequest, [78](#), [80](#)

The document is licensed under the [Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported License](https://creativecommons.org/licenses/by-nc-sa/3.0/)

