CSSUtilities Developer’s Guide

1. Setup and Configuration

Before you can use CSSUtilities you'll need to setup and configure it, and have a working knowledge of the different ways to get data from it. This part of the documentation will take you through that:

1. **Setting it up**
2. **Getting data**
    1. Asynchronous with manual init
    2. Asynchronous with auto-init
    1. Synchronous with manual init
    2. Synchronous with auto-init
3. **Configuration options**
    - `"mode"`
    - `"async"`
    - `"page"`
    - `"base"`
    - `"attributes"`
    - `"watch"`
    - `"api"`
    - The `.supported` property

Setting it up

There are *two* script includes to put at the end of the `<head>` section — the main CSSUtilities library, plus an additional Selectors API *(Application Programming Interface)* :

```
<script type="text/javascript" src="CSSUtilities.js"></script>
<script type="text/javascript" src="Selector.js"></script>
```

> The additional Selectors API *(Application Programming Interface)* provides functionality to browsers that don't support the **querySelectorAll** <http://www.w3.org/TR/selectors-api/> method (which is Opera 9, Firefox 1.5–3.0, Konqueror, IE6 *(Internet Explorer 6)* and IE7 *(Internet Explorer 7)*), so if you don't include the additional library then CSSUtilities won't work in those browsers either. The library that's bundled by default is LlamaLab's **Selector.js** <http://llamalab.com/js/selector/>, because in my testing it gave the most consistently accurate information with a very small code size (Dean Edward's **base2** <http://code.google.com/p/base2/> and Robert Nyman's **DOMAssistant** <http://www.domassistant.com/> gave equally good results, but they're larger script files, and CSSUtilities is not exactly tiny by itself!). But you can use a different Selectors API if you want; this might make sense if you were already including the codebase for another library anyway.

The library scripts go in the `<head>` section so you can initialize as soon as possible during page load; and they go at *the end* of the `<head>` so that all the page's stylesheets come before it. But as we'll see, most of the actual scripting you do with the library will need to go at the end of the `<body>` — or be deferred until page-load — because such scripting will generally also refer to elements in the DOM *(Document Object Model)* .

Getting data

The script has two primary execution modes (asynchronous or synchronous), and two implementation modes ("browser" mode or "author" mode). These modes and their interaction have specific implications for how you use the library:

> **In "browser" mode**
>
> The script compiles all the data it needs from the `document.styleSheets` collection, therefore ***no network***

> The script compiles all the data it needs from the `document.styleSheets` collection, therefore ***no network requests are made*** and the difference in performance impact between synchronous and asynchronous execution is neglible.
>
> **In "author" mode**
>
> The script needs to re-parse every style sheet it encounters as plain text, and therefore ***multiple network requests might be made*** and the difference between synchronous and asynchronous execution is significant.
>
> If you choose synchronous execution then you should initialize the library **as soon as possible** during page load — with the script at the end of the `<head>` section and the initialization command immediately after that. This will allow the initialization time to be absorbed into the overall page loading time, and so reduce the apparent impact overall.
>
> If you choose asynchronous execution then users needn't notice the impact, because all the network requests happen in the background; but of course you still need to be aware that the initialization process will take time, and that all subsequent uses of the data methods must defer to that (via callback function, as the following design patterns will illustrate).

So with that in mind, there are **four primary design patterns** you can choose from (two for each execution mode):

Pattern A/1: Asynchronous with manual init

With this pattern you call the initialization method asynchronously, passing a callback for when it's complete. Within the callback you can then use the data methods synchronously like normal:

```
<head xmlns="http://www.w3.org/1999/xhtml" lang="en" xml:lang="en">

    ... style sheets ...

    <script type="text/javascript" src="CSSUtilities.js"></script>
    <script type="text/javascript" src="Selector.js"></script>
</head>
<body>

    ... page content ...

    <script type="text/javascript">

        //configure
        CSSUtilities.define('async', true);

        //initialize with callback
        CSSUtilities.init(function()
        {
            //synchronously call data method
            var rules = CSSUtilities.getCSSRules('#foo');

            //work with the returned data
            alert(rules);
        });

    </script>
</body>
```

Pattern A/2: Asynchronous with auto-init

With this pattern you call the data methods asynchronously without first initializing, and each of them has a completion callback of its own, which is fired once auto-init (if necessary) and the data-retrieval is complete. Within each callback the response data is passed in as an argument:

```
<head xmlns="http://www.w3.org/1999/xhtml" lang="en" xml:lang="en">

    ... style sheets ...

    <script type="text/javascript" src="CSSUtilities.js"></script>
    <script type="text/javascript" src="Selector.js"></script>
</head>
<body>

    ... page content ...

    <script type="text/javascript">

        //configure
        CSSUtilities.define('async', true);

        //asynchronously call data method with callback
        CSSUtilities.getCSSRules('#foo', function(rules)
        {
            //work with the response data
            alert(rules);
        });

    </script>
</body>
```

Every data method has its own completion callback, and it's always the last argument.

Pattern S/1: Synchronous with manual init

With this pattern you call the intialization method synchronously, and then use the data methods synchronously once that's complete; since all the calls are separate you can initialize early on, and thereby reduce the impact of synchronous network use:

```
<head xmlns="http://www.w3.org/1999/xhtml" lang="en" xml:lang="en">

    ... style sheets ...

    <script type="text/javascript" src="CSSUtilities.js"></script>
    <script type="text/javascript" src="Selector.js"></script>
    <script type="text/javascript">

        //configure
        CSSUtilities.define('async', false);

        //initialize
        CSSUtilities.init();

    </script>
</head>
<body>

    ... page content ...

    <script type="text/javascript">

        //synchronously call data method
        var rules = CSSUtilities.getCSSRules('#foo');

        //work with the returned data
        alert(rules);

    </script>
</body>
```

Pattern S/2: Synchronous with auto-init

With this pattern you call the data methods synchronously, without first initializing, and this triggers a synchronous auto-init (if necessary) before retrieving and returning the data; using this approach will probably cause an obvious pause the first time you do it:

```
<head xmlns="http://www.w3.org/1999/xhtml" lang="en" xml:lang="en">

    ... style sheets ...

    <script type="text/javascript" src="CSSUtilities.js"></script>
    <script type="text/javascript" src="Selector.js"></script>

</head>
<body>

    ... page content ...

    <script type="text/javascript">

        //configure
        CSSUtilities.define('async', false);

        //synchronously call data method
        var rules = CSSUtilities.getCSSRules('#foo');

        //work with the returned data
        alert(rules);

    </script>
</body>
```

Configuration options

CSSUtilities has a small number of global configuration options, all of which are defined *before* initialising the library (the design patterns shown above also illustrate where this happens).

Options are defined using the `define` method, which takes **two required arguments**, for the name (always a `String`) and the value (some `String` some `Boolean`):

```
CSSUtilities.define('mode', 'author');
CSSUtilities.define('async', true);
CSSUtilities.define('attributes', false);
```

(For one of the settings there's an optional third argument; **for details see :: external Selectors API settings.**)

The `define()` method includes **comprehensive validation and error reporting**, and any invalid definitions will throw a fatal error that prevents any further use of the library; **for a list of possible errors see :: Error messages.**

Here's a reference of all the available options, their meanings and possible values:

"mode"

```
CSSUtilities.define("mode", "browser");
```

implementation mode *[OPTIONAL String]*

A `String` that specifies one of two internal parser implementations — either `"browser"` or `"author"`. The default value if undefined is `"browser"`.

This variable fundamentally changes how the library gets its CSS *(Cascading Style Sheets)* data, and therefore what data it returns.

In `"browser"` mode the library uses the `document.styleSheets` collection to get its data, which means that **what we get is subject to each browser's implementation**: it only includes what the browser understands and supports, and the properties will be normalized in various ways (for example, among other things: Firefox normalizes hex *(hexadecimal (Base 16))* colors to RGB *(Red Green Blue)* , Opera normalizes colors to 6-digit hex *(hexadecimal (Base 16))* , Internet Explorer splits-up margin shorthands into their component longhand properties).

In `"author"` mode, the library loads and parses every stylesheet as plain text, which means that **each browser should return the same data**, whether or not it actually supports everything listed, and all the properties will be returned exactly as they were specified in the stylesheet — with **no normalization of units or value-types**, and everything listed in the same syntax it was defined with (*except for properties defined in* HTML *(HyperText Markup Language)* `style` *attributes — these properties cannot be returned independently of browser normalization)*.

So as a consequence of each approach, `"author"` mode has to make a number of Ajax requests in addition to the normal page load, where `"browser"` mode already has the data it needs. This difference may in turn affect your choice of asynchronous or synchronous execution mode; **for more about that see :: Getting data.**

Which mode you choose will obviously depend on what you need, and there are no hard or fast rules about which is better. But broadly speaking, `"author"` mode may be suitable for testing and development tools, while `"browser"` mode may be more appropriate for real-world applications.

You should also note that **only `"browser"` mode has full access to dynamically-generated styles**. In `"author"` mode, the library won't see any rules created using `addRule` or `insertRule`; but all browsers except IE *(Internet Explorer)* will see rules created as text-nodes inside a `<style>` element, and all browsers will see properties added to `.style` (subject to browser support for the individual properties). Naturally in all cases, only styles created *before* initialization will be picked-up, though you can of course re-initialize at any time.

Some of the data methods are affected by your choice of implementation, for example, where particular options are only relevant in one mode or the other; such details are documented wherever applicable.

"async"

```
CSSUtilities.define("async", false);
```

asynchronous execution mode *[OPTIONAL Boolean]*

A `Boolean` that defines whether the library's initialization should be asynchronous (`true`) or synchronous (`false`). The default value if undefined is `false` (synchronous execution).

The difference in grammar is simply that with synchronous execution you can query the library using all synchronous function calls, whereas with asynchronous execution you have to use callbacks for some.

The difference in *practical* terms though is that the execution mode primarily determines how Ajax requests are made — correspondingly synchronous or asynchronous — which in turn has an impact on usability, because synchronous requests lock-up the browser thread until they're complete (the impact depending on when you initialize and how many requests are made, factors which are affected by your choice of implementation mode).

This may sound a little convoluted at first, but once you get used to the design patterns I hope you'll appreciate the flexibility they afford; **for details please see :: Getting data.**

"page"

```
CSSUtilities.define("page", document);
```

context document *[OPTIONAL Document]*

A `Document` that defines the document object (DOM *(Document Object Model)*) you wish to inspect — containing the test elements and stylesheets you're interested in. The default value if undefined is `document`.

Generally you won't need to change this, but it's there so that you can configure the library to work with virtualized or remote documents, such as the `responseXML` of an Ajax request, or the `contentDocument` of an `<iframe>`.

If you do specify a different document, you will often need to specify a matching base URI for stylesheet paths as well.

If you're also defining `"api"` then you must define this option *first*.

"base"

```
CSSUtilities.define("base", document.location.href);
```

base URI for stylesheet paths *[OPTIONAL String]*

A `String` that defines a URI *(Uniform Resource Indicator)* base for qualifying stylesheet paths. The default value if undefined is the URL *(Uniform Resource Locator)* of the document specified in the `page` variable, which in turn defaults to `document`, and so the default value for `base` is `document.location.href`.

Most of the time you won't need to change this, but it's there so that the DOM *(Document Object Model)* you're inspecting can be at a different location than the stylesheets, or virtualized entirely such that it has no addressable location.

For example, in some versions of **CodeBurner** <http://tools.sitepoint.com/codeburner/> the DOM *(Document Object Model)* inspection works by loading a remote page using Ajax and then writing the returned HTML *(HyperText Markup Language)* into an `<iframe>`, in order to create a document object for testing. But then of course, the document's `location` no longer matches the URL *(Uniform Resource Locator)* of the original page, so any relative paths in `<link href>` stylesheet addresses could no longer be resolved. So we set the `"base"` variable to match the remote page's original URL *(Uniform Resource Locator)* , and then all the stylesheet URLs *(Uniform Resource Locators)* can be resolved against that!

The value of `base` **must be an absolute URL *(Uniform Resource Locator)*** . If you specify a folder root then you must include the trailing slash.

If you're also defining `"api"` then you must define this option *first*.

"attributes"

```
CSSUtilities.define("attributes", true);
```

parse HTML style attributes *[OPTIONAL Boolean]*

A `Boolean` that defines whether to include the data from HTML *(HyperText Markup Language)* `style` attributes. The default value if undefined is `true`.

Any such CSS *(Cascading Style Sheets)* properties defined for an element will be organised into a single rule object, with meta-data that reflects their origin — such as higher specificity, and a selector with the special value `""` (empty string).

Other things being equal, there's no reason to change this option — disabling it might mean that some data sets are incomplete, missing any properties defined this way. However if you *know* that the page you're checking *doesn't use* any `style` attributes, then you can disable this option and give the script one less thing to check.

Please also note that CSS *(Cascading Style Sheets)* properties defined in `style` attributes **cannot bypass browser normalization** in "author" mode (as opposed to properties defined in external stylehseets, which are returned independently of normalization in this mode); such inconsistency might also be a reason for disabling this option, and/or avoiding the use of HTML *(HyperText Markup Language)* `style` attributes on applicable pages (assuming you have any such choice or inclination!).

"watch"

```
CSSUtilities.define("watch", false);
```

watch for stylesheet switching *[OPTIONAL Boolean/Object]*

A `Boolean` that defines whether or not the library should monitor for stylesheet switching activity. The default value if undefined is `false`.

If this is set to `true`, the script will continually check the `disabled` state of every listed stylesheet; as soon as any of them changes, the library automatically re-initializes to refresh its data cache. Automatic re-initialization will also re-fire any stored initialization callback.

If you set this to `false` then the timer will not run, so if any state changes do occur then the library will no longer have accurate data. In that situation it would be up to you to manually re-initialize as necessary (for example, as part of a stylesheet switching script).

In "author" mode (only) you can also set this to `null`, which tells the library to *ignore* the disabled state of all stylesheets, and parse them as though they were all enabled; and no further monitoring will take place. This is a reference setting and should be used with care, because it means that the data methods may subsequently return rules and properties which are marked as active when really they're not (because they're inside a stylesheet that's disabled).

→ **Important Browser Note:**

In **Safari and other Webkit browsers**, a stylesheet element's `disabled` property will always return `false` until it's been explicitly set. This is a browser issue which affects the script in "author" mode, and means that **alternate stylesheets will appear to be normal stylesheets, enabled by default**, and the watch mechanism won't work until at least one switch has happened.

This issue is easily fixed, but the script can't do so automatically, since it may potentially change the page's default appearance or behavior.

So if you have this problem (ie. if you're running in `"author"` mode, and the `watch` setting is not `null`, and you have alternate stylesheets on the page), then you should implement the fix yourself; like this:

```
for(var links=document.getElementsByTagName('link'),
        i=0; i<links.length; i++)
{
    if(links[i].getAttribute('rel') == 'alternate stylesheet')
    {
        links[i].disabled = true;
    }
}
```

All that does is set a default `disabled` value for alternate stylesheets, which reflects the state they have anyway, but so that it can be recorded accurately. You should add the code **before intializing CSSUtilities**; and if you're implementing any kind of stylesheet switcher script, you should also add the fix **before re-creating any saved state** (such as from a cookie).

(This issue can also be fixed by adding a `disabled="disabled"` attribute to the alternate stylesheets; however that would be invalid HTML *(HyperText Markup Language)* , for some reason.)

"api"

```
CSSUtilities.define("api", false);
```

external Selectors API settings *[OPTIONAL Boolean][, OPTIONAL Function]*

A `Boolean` and a `Function` that together control any external Selectors API *(Application Programming Interface)* .

Much of the functionality available from CSSUtilities is only possible because of the existence of a **Selectors API** <http://www.w3.org/TR/selectors-api/>, that identifies elements matching a given CSS *(Cascading Style Sheets)* selector. The most recent browser versions (Opera 10, Firefox 3.5+ *(or later)* , Safari, Chrome and IE8 *(Internet Explorer 8)*) have this functionality built-in, but to make it work in the other supported browsers (Opera 9, Firefox 1.5–3.0, Konqueror, IE6 *(Internet Explorer 6)* and IE7 *(Internet Explorer 7)*) we must use an external library to provide the API *(Application Programming Interface)* ; LlamaLab's **Selector.js** <http://llamalab.com/js/selector/> is bundled by default.

So, if you leave the `"api"` option undefined, the behaviors just described will apply.

The *first* value is a `Boolean` which forces every browser to use the fallback Selectors API *(Application Programming Interface)* (`true`), or lets each browser prefer its native implementation if one is available (`false`) . The default value is `false`.

The *second* value allows you to specify a different library, other than the one bundled by default. Once you've added its `<script>` include to the page, you define a wrapper to acts as an intermediary between CSSUtilities and your library's Selectors API *(Application Programming Interface)* ; like this:

```
CSSUtilities.define("api", false, function(selector, page)
{
    return queryMyAPI(selector, page);
});
```

The wrapper will be passed a `String` selector (the `"selector"` argument) and a `Document` context (the `"page"` argument), and must return an `Array` or `NodeList` of zero or more `Element` nodes, as returned by a selector query. If your chosen API *(Application Programming Interface)* returns the data in a different structure or format, you'll have to convert it before returning.

CSSUtilities will test the wrapper as soon as you define it, and will throw a fatal error if it returns invalid data, that prevents any further use of the library.

You can further test it by examining the data it returns yourself — just change the first value to `true` and then all browsers will be using it.

If you are defining this option, you must define both `"page"` and `"base"` *first*, as they cannot be re-defined afterwards.

Here are some sample wrappers for a selection of popular libraries:

base2 <http://code.google.com/p/base2/>

base2 returns a static node list where square-bracket notation is not supported, so it must be converted to an `Array`:

```
CSSUtilities.define('api', false, function(selector, page)
{
    var nodes = [],
        list = base2.DOM.Document.querySelectorAll(page, selector);
    for(var len=list.length, i=0; i<len; i++)
    {
        nodes.push(list.item(i));
    }
    return nodes;
});
```

DOMAssistant <http://www.domassistant.com/>

DOMAssistant occasionally returns `undefined` for a selector it's unable to match, so we test for that and return an empty array instead:

```
CSSUtilities.define('api', false, function(selector, page)
{
    var nodes = $(page).cssSelect(selector);
    return typeof nodes != 'undefined' ? nodes : [];
});
```

Sizzle / jQuery <http://sizzlejs.com/>

Sizzle is missing support for a number of CSS3 *(Cascading Style Sheets Level 3)* selectors.

```
CSSUtilities.define('api', false, function(selector, page)
{
    return Sizzle(selector, page);
});
```

prototype <http://www.prototypejs.org/>

```
CSSUtilities.define('api', false, function(selector, page)
{
    return new Selector(selector).findElements(page);
});
```

dojo <http://www.dojotoolkit.org/>

```
CSSUtilities.define('api', false, function(selector, page)
{
    return dojo.query(selector, page);
});
```

YUI <http://developer.yahoo.com/yui/>

```
CSSUtilities.define('api', false, function(selector, page)
{
    return YAHOO.util.Selector.query(selector, page);
});
```

The `.supported` property

CSSUtilities includes a global property for filtering-out known unsupported browsers. It's set as soon as the script exists, and can be used as a filter to minimize or eliminate scripting errors in older browsers.

The value is formed by testing for three properties — `document.getElementById`, `document.styleSheets` and `document.nodeType` — and if *any one* of those properties is missing, the browser will be marked as unsupported. This is known to completely exclude Internet Explorer 5.5 and Opera 8, but can be reasonably assumed to exclude most legacy browsers from that generation or earlier.

If you wish to use this then, simply wrap any or all uses of the library's methods with an “if-supported” condition:

```
if(CSSUtilities.supported)
{
    CSSUtilities.init();
    //etc.
}
```

2. Functions Reference

This part of the documentation has a detailed technical breakdown of all the public methods the CSSUtilities library provides. There's also a reference list of errors you may encounter in data sets or thrown to the console, and a general list of specifications:

Configuration methods

If you've read through the Setup and Configuration guide then you should already be quite familiar with these configuration methods, which are used for setting-up and initializing the library. This provides a single point of reference you can come back to later and remind yourself of details.

`CSSUtilities.define(name, value1 [, value2]);`

The `define` method is used for modifying the library's configuration options, before initialising; it has no return value.

All the options are private variables and can only be modified using this method; **for a guide to the available options see :: Configuration options.**

```
CSSUtilities.define(
    "mode",                // option name                [REQUIRED String]
    "author",              // option value1              [REQUIRED String|Boolean]
    function() { ... },    // option value2              [OPTIONAL Function]
    );
```

Arguments

> **option name *[REQUIRED String]***
>
> A `String` that specified which option you're defining.
>
> **option value1 *[REQUIRED String|Boolean]***
>
> This parameter is the value you're defining; some options take a `String` value while others take a `Boolean`.
>
> Values are comprehensively validated, and any invalid definitions will throw a fatal error that prevents any further use of the library; **for a list of possible errors see :: Error messages.**
>
> **option value2 *[OPTIONAL Function]***
>
> For one of the config options a second parameter is available, which must be a `Function`; **for details see :: `"api"`.**

CSSUtilities.init(oncomplete);

The `init` method initializes the library, populating its data cache and making it available for use; it has no return value.

This method is either synchronous or asynchronous, depending on your choice of execution mode; **for details and design patterns please see :: Getting data.**

```
CSSUtilities.init(
    function() { ... }     // callback when complete       [OPTIONAL Function]
    );
```

Arguments

> **callback on completion** ***[OPTIONAL Function]***
>
> A `Function` that will be stored and called once initialization has finished. If you're using an asynchronous execution pattern then the use of this callback may be critical to the way you use the library overall. But regardless of execution mode, the callback will be fired when initialization is complete.
>
> Any callback you pass to this method will be stored for later use, **and automatically fired again if automatic re-initialization occurs**. This can happen if you use the watch setting to monitor stylesheet switching, which triggers auto-init whenever it detects any disabled state changes. If this happens, whatever was stored *the last time you called* `init` *manually*, will be fired again.

Data methods

The data methods are the meat-and-drink of CSSUtilities, each providing different information about the CSS *(Cascading Style Sheets) you're using. This section provides a complete reference of every available method, its arguments, and the data it returns.*

CSSUtilities.getCSSRules(element [, media] [, properties] [, altstates] [, oncomplete])

The `getCSSRules` method finds and returns all the CSS *(Cascading Style Sheets)* rules that apply to a specified element within specified media, including any that are inherited.

```
var rules = CSSUtilities.getCSSRules(
    "#heading"             // element reference or ID      [REQUIRED Object|String]
    "screen",              // media context                [OPTIONAL String]
    "selector,css",        // data to accept               [OPTIONAL String]
    false,                 // include alternate states     [OPTIONAL Boolean]
    function() { ... }     // callback when complete       [OPTIONAL Function]
    );
```

Arguments

> **element reference or ID** ***[REQUIRED Object|String]***
>
> A reference to the element you wish to check, which can either be an `Object` reference to the element itself, or a `String` which is the element's `ID` plus a leading `"#"` (like an ID-selector, eg. `"#content"`).
>
> This argument is required will throw an error if undefined.
>
> **media context** ***[OPTIONAL String]***
>
> A comma-delimited `String` that specifies which media-types the returned rules should apply to. You can specify `"*"` for all media, or any valid CSS *(Cascading Style Sheets)* media types (including `"all"`); there are also two special values: `"none"` and `"current"`.

The value `"none"` means, rules which apply don't apply to any media. The media context evaluation is implemented using each rule's **computed** media, which may be different from the media-types that were actually defined for it. For example, an `"@media all"` wrapper nominally applies to all media; but if that wrapper is in a stylesheet included with the `media` attribute `"screen,projection"`, then the rules inside the `@media` wrapper actually only apply to `"screen"` and `"projection"`. This means that some rules may ultimately compute to no media at all, and in that case their media will be listed as `"none"`.

The value `"current"` refers to whatever media type applies to the current page view. In most situations it will be `"screen"`, but this value dynamically changes to match whatever it is. You can test this out easily in Opera, because it applies `"projection"` media when it's in Full Screen mode. *(**Please note:** detecting the current view media is not supported in Safari 3 or Konqueror 4; you can still use this feature but you'll always get rules for* `"screen"`*. Detection may also fail in* XML *(eXtensible Markup Language) environments, and if so it will also fallback to* `"screen"`*.)*

A value of `"all"` for this argument **does not include** `"none"` — if you want those rules to be included you can specify `"all,none"` or `"*"`.

CSSUtilities is currently **not sensitive** to **CSS media queries** <http://www.w3.org/TR/css3-mediaqueries/>, but neither does it remove them — you cannot specify a media query for this argument, and any queries that form part of a stylesheet's own `media` definition will be ignored in media-context evaluations; but any such queries are still listed as part of the rule's applicable media types.

So for example, if a rule had the media `"screen and (color)"`, it would be returned for any search where the media-context included `"screen"`, but its media type would still be listed as `"screen and (color)"`. *(**Please note:** there are also some caveats to the library's awareness of media query syntax: comma-delimited queries, those beginning with* `"not"`*, and those using shorthand syntax like* `"@media (color)"`*, are all misinterpreted as unique types in their own right, and will therefore only be returned when the media-context is* `"all"` *or* `"*"`*.)*

If undefined or empty this argument defaults to `"screen"`.

data to accept *[OPTIONAL String]*

A comma-delimited `String` that lists what data you want to returned `rules` collection to include. Each rule in the returned collection is an `Object` that includes only the members you specify here — values such as `"selector"` to include the selector-text of each rule, or `"media"` to include the media each rule applies to.

You can specify `"*"` to return *all* available data, or any of the following members:

- `"selector"` (the rule's selector text)
- `"css"` (the complete CSS *(Cascading Style Sheets)* text of the rule)
- `"index"` (the rule's overall source index)
- `"specificity"` (the specificity of this rule)
- `"inheritance"` (the rule's inheritance context)
- `"altstate"` (whether this rule applies to an alternate state of the element)
- `"media"` (the computed media-types that the rule applies to)
- `"xmedia"` (the media-types that the author defined for the rule)
- `"owner"` (a representation of the rule's grammatical context)
- `"ssid"` (the internal ID of the rule's owning stylesheet in the library's data cache)
- `"href"` (the qualified URL *(Uniform Resource Locator)* of the rule's owning stylesheet)
- `"properties"` (the individual CSS *(Cascading Style Sheets)* properties for this rule, with further information about their status)

You'll find more details about the structure and content of each member in the return value documentation (or follow the individual links).

If undefined or empty this argument defaults to `"*"` (all available data).

include alternate states *[OPTIONAL Boolean]*

A `Boolean` that specifies whether you want the returned data to include rules that only apply to alternate states of the element, such as `:hover` and `:active`.

Accordingly, any such rules in the returned data will have their properties marked as status `"inactive"`.

If undefined this argument defaults to `false`.

callback when complete ***[OPTIONAL Function]***

A `Function` that will be called once the method has completed. The function will be passed a single argument: the `rules` `Array` that would otherwise be its return value.

Optional arguments between the required arguments and the callback can be omitted entirely, and their default values will be inserted automatically. As long as the *required* arguments are there, it doesn't matter how many intermediate optional arguments are missing: the callback can always be the last one. So both syntaxes in the example below are valid and interchangeable:

```
var rules = CSSUtilities.getCSSRules("#foo", "screen", "*", false, myCallback);

var rules = CSSUtilities.getCSSRules("#foo", myCallback);
```

This is designed to be used with asynchronous execution, and is sensitive to the initialization state of the library — if init has not been called, or has not finished, the method will wait until it has, and only then fire the specified callback. You can see an example of its use in the Asynchronous with auto-init design pattern.

This argument has no default value if undefined.

Return value ***[Array]***

Return value

The method returns an `Array` of zero or more rules, in order of specificity, each of which is an `Object` containing some or all of the following members, as specified in the `accept` argument:

"selector" ***[String]***

A `String` containing the complete selector for this rule (equivalent to the `selectorText` property).

If the rule has multiple comma-delimited selectors, then the entire selector will normally be returned. But the one exception to this is Internet Explorer when the library is in "browser" mode, in which case the browser's internal CSS *(Cascading Style Sheets)* parsing splits such rules into multiple individual rules, each with a single selector (eg. a rule with the selector `"html,body"` would become two rules with `"html"` and `"body"`).

For a rule object that represents the properties defined in a `style` attribute, its selector will have the special value `""` (empty string).

"css" ***[String]***

A `String` containing the complete CSS text of the rule (equivalent to the `style.cssText` property).

"index" ***[Number]***

A `Number` which is the overall source index of this rule, representing its position in the cascade.

This value is used as part of the library's specificity sorting algorithms, and is also the index of the rule in the library's data cache (which stores rules in cascade order).

"specificity" ***[Array]***

An `Array` of four numbers that expresses the relative specificity of this rule, with respect to its inheritance context.

The array is in order of specificity categories, from highest (values in the style attribute) to lowest (simple element selectors), as defined in the **CSS3 specification** <http://www.w3.org/TR/css3-selectors/#specificity>.

Since this value is contextual, any inherited rule will therefore have zero specificity and return the value `[0,0,0,0]`.

There's more about specificity in the `getCSSSelectorSpecificity` method documentation.

"inheritance" ***[Array]***

An `Array` of `Element` references that represents this instance of this rule's inheritance path, in order of DOM *(Document Object Model)* traversal.

So for **inherited rules**, the last member of the array will always be the `parentNode` of the element specified in the original query. For example, if the specified element is an `<h1>` which is a direct child of `<body>`, then an inherited rule with the selector `"html"` would have the inheritance array `[[HTMLHtmlElement], [HTMLBodyElement]];`

Inherited rules may in fact appear *multiple times* in a returned data set, each time with a progressively deeper inheritance path, denoting that the rule is inherited through, and applies to, multiple intermediate elements.

For **non-inherited rules** rules then, this array will always be empty.

"altstate" ***[Boolean]***

A `Boolean` that indicates whether this rule applies only to an alternate state of the element, such as `:hover` or `:active`.

Such rules are only included in the returned data at all if the altstates argument is set to `true`, and will furthermore have their properties marked as status `"inactive"`.

"media" ***[String]***

A `String` containing the computed media-types that this rule applies to, comma-delimited if more than one type applies. If a rule applies to all media this will have the value `"all"`; if it applies to no media it will have the value `"none"`.

This is a *computed* value — it returns only those types that the rule actually applies to, which may be different from the media that were originally specified for it. There's more info about that in the media context argument documentation.

"xmedia" ***[String]***

A `String` containing the media-types that were originally specified for the rule, which may be different from its computed media. Depending on how the rule was included, this might be the media specified in an `@media` statement, or added to the end of an `@import`, or specified in the media attribute of a `<link>` element.

This data is **only available in author mode**, and does not form part of any media context evaluation. However it may be useful for comparison, or for re-creating the original CSS *(Cascading Style Sheets)* source-code for a set of rules.

"owner" ***[String]***

A `String` that represents the immediate grammatical context of this rule, according to what surrounds it and how the stylesheet that contains it was included. It will be one of the following fixed values:

- `"style"` (a rule inside a `<style>` block)
- `"link"` (a rule inside a stylesheet that was included using a `<link>` element)
- `"xml-stylesheet"` (a rule inside a stylesheet that was included using an `<?xml-stylesheet?>` processing instruction)
- `"@import"` (a rule inside a stylesheet that was included using an `@import` statement)
- `"@media"` (a rule inside an `@media` block)
- `"@style"` (a rule defined using a `style` attribute in markup)

"ssid" ***[Number]***

A `Number` which is the internal ID of this rule's owning stylesheet within the library's data cache.

The value represents the order in which the stylesheet includes occur — that is, the source-order of the elements or `@import` statements that declare them — which may differ from the source-order of the rules inside them, because of the way the cascade works (rules inside imported stylesheets come *before* the rules in their parent).

For a rule that represents the element's `style` attribute, this will be `Infinity`.

"href" ***[String/Object]***

A `String` which is the qualified URL *(Uniform Resource Locator)* of this rule's owning stylesheet. This will always be a fully-qualified address, irrespective of the syntax that was originally used.

If the stylesheet has no URL *(Uniform Resource Locator)* (like a rule inside a `<style>` block) then this will be `null`.

"properties" ***[Object]***

An `Object` containing the individual CSS *(Cascading Style Sheets)* properties for this rule, extracted from its css text, and indexed by property name. Each member of the object is a further object, which has two members listing the `"value"` and `"status"` of the property, for example:

```
{
    "color":         { "value": "#333",  "status": "active"     },
    "font-weight":   { "value": "bold",  "status": "cancelled"  },
    "border":        { "value": "none",  "status": "active"     }
}
```

The `status` property will be one of the following fixed values:

- `"active"` (the property applies to the rule)
- `"cancelled"` (the property has been cancelled-out by another definition with higher specificity)
- `"inactive"` (the property only applies to an alternate state of the element, such as `:hover`)

If you're working in "browser" mode, this object may contain many more properties than you're expecting, because of the action of browser normalization — browsers converting CSS *(Cascading Style Sheets)* property definitions to a different syntax, for their own convenience — Internet Explorer, for example, splits-out `margin` and `padding` shorthands into their component longhand definitions; Opera does the same thing with `border`. You may also notice changes in units and values, such as hex *(hexadecimal)* colors converted to RGB *(Red Green Blue)* in Firefox.

If the rule has no properties which apply to the specified element (ie. an inherited rule has no inheritable properties, or a direct rule has no properties at all), then this object will be `null`.

If the attributes option is set to `true`, and the specified element has a non-empty `style` attribute, then the returned array will also include a special rule object containing the properties defined in that attribute.

The array is sorted in order of specificity, from lowest to highest; where multiple rules have the same specificity value then they're sorted in order of source index (thereby maintaining an overall specificity order). If multiple rules with the same specificity also have the same source index (such as a rule with multiple, comma-delimited selectors), then they're sorted in traversal order of inheritance depth (ie. `"html"` before `"body"`).

If there are no rules for the specified element and media, this method will return an empty `Array`.

`CSSUtilities.getCSSProperties(element [, media] [, oncomplete])`

The `getCSSProperties` method lists all the CSS *(Cascading Style Sheets)* properties that apply to a specified element within specified media.

```
var properties = CSSUtilities.getCSSProperties(
    "#heading"              // element reference or ID      [REQUIRED Object|String]
    "screen",               // media context                [OPTIONAL String]
    function() { ... }      // callback when complete       [OPTIONAL Function]
    );
```

Arguments

element reference or ID *[REQUIRED Object|String]*

A reference to the element you wish to check, which can either be an `Object` reference to the element itself, or a `String` which is the element's `ID` plus a leading `"#"` (like an ID-selector, eg. `"#content"`).

This argument is required will throw an error if undefined.

media context *[OPTIONAL String]*

A comma-delimited `String` that specifies which media-types the returned properties should apply to. You can specify `"*"` for all media, or any valid CSS *(Cascading Style Sheets)* media types (including `"all"`); there are also two special values: `"none"` and `"current"`.

The value `"none"` means, rules which apply don't apply to any media. The media context evaluation is implemented using each rule's **computed** media, which may be different from the media-types that were actually defined for it. For example, an `"@media all"` wrapper nominally applies to all media; but if that wrapper is in a stylesheet included with the `media` attribute `"screen,projection"`, then the rules inside the `@media` wrapper actually only apply to `"screen"` and `"projection"`. This means that some rules may ultimately compute to no media at all, and in that case their media will be listed as `"none"`.

The value `"current"` refers to whatever media type applies to the current page view. In most situations it will be `"screen"`, but this value dynamically changes to match whatever it is. You can test this out easily in Opera, because it applies `"projection"` media when it's in Full Screen mode. *(**Please note:** detecting the current view media is not supported in Safari 3 or Konqueror 4; you can still use this feature but you'll always get rules for* `"screen"`*. Detection may also fail in* XML *(eXtensible Markup Language) environments, and if so it will also fallback to* `"screen"`*.)*

A value of `"all"` for this argument **does not include** `"none"` — if you want those rules to be included you can specify `"all,none"` or `"*"`.

CSSUtilities is currently **not sensitive** to **CSS media queries** <http://www.w3.org/TR/css3-mediaqueries/>, but neither does it remove them — you cannot specify a media query for this argument, and any queries that form part of a stylesheet's own `media` definition will be ignored in media-context evaluations; but any such queries are still listed as part of the rule's applicable media types.

So for example, if a rule had the media `"screen and (color)"`, it would be returned for any search where the media-context included `"screen"`, but its media type would still be listed as `"screen and (color)"`. *(**Please note:** there are also some caveats to the library's awareness of media query syntax: comma-delimited queries, those beginning with* `"not"`*, and those using shorthand syntax like* `"@media (color)"`*, are all misinterpreted as unique types in their own right, and will therefore only be returned when the media-context is* `"all"` *or* `"*"`*.)*

If undefined or empty this argument defaults to `"screen"`.

callback when complete *[OPTIONAL Function]*

A `Function` that will be called once the method has completed. The function will be passed a single argument: the `properties` `Object` that would otherwise be its return value.

Optional arguments between the required arguments and the callback can be omitted entirely, and their default values will be inserted automatically. As long as the *required* arguments are there, it doesn't matter how many intermediate optional arguments are missing: the callback can always be the last one. So both syntaxes in the example below are valid and interchangeable:

```
var properties = CSSUtilities.getCSSProperties("#foo", "screen", myCallback);

var properties = CSSUtilities.getCSSProperties("#foo", myCallback);
```

This is designed to be used with asynchronous execution, and is sensitive to the initialization state of the library — if init has not been called, or has not finished, the method will wait until it has, and only then fire the specified callback. You can see an example of its use in the Asynchronous with auto-init design pattern.

This argument has no default value if undefined.

Return value *[Array]*

Return value

The method returns an `Object` of one or more CSS *(Cascading Style Sheets)* properties, each of which is a `String` value indexed by its `String` property name, for example:

```
{
    "color": "#333 !important",
    "font-weight": "bold",
    "border": "none"
}
```

Only properties which *actually apply to the element now* are included in the data returned by this method; properties which have been cancelled out by others with higher specificity, or which only apply to alternate states of the element, will not be included (if you want that data you should use the `getCSSRules` method instead).

The data will include any active properties from the element's `style` attribute, if the "attributes" option is set to `true`.

If there are no properties for the specified element and media, this method will return `null`.

`CSSUtilities.getCSSSelectors(element [, media] [, directonly] [, oncomplete])`

The `getCSSSelectors` method lists all the individual CSS *(Cascading Style Sheets)* selectors that apply to a specified element within specified media; either only those which select the element explicitly, or also those which select an ancestor it can inherit from.

```
var selectors = CSSUtilities.getCSSSelectors(
    "#heading"              // element reference or ID      [REQUIRED Object|String]
    "screen",               // media context                [OPTIONAL String]
    true,                   // direct selectors only        [OPTIONAL Boolean]
    function() { ... }      // callback when complete       [OPTIONAL Function]
    );
```

Arguments

element reference or ID *[REQUIRED Object|String]*

A reference to the element you wish to check, which can either be an `Object` reference to the element itself, or a `String` which is the element's `ID` plus a leading `"#"` (like an ID-selector, eg. `"#content"`).

This argument is required will throw an error if undefined.

media context *[OPTIONAL String]*

A comma-delimited `String` that specifies which media-types the returned selectors should apply to. You can specify `"*"` for all media, or any valid CSS *(Cascading Style Sheets)* media types (including `"all"`); there are also two special values: `"none"` and `"current"`.

The value `"none"` means, rules which apply don't apply to any media. The media context evaluation is implemented using each rule's **computed** media, which may be different from the media-types that were actually defined for it. For example, an `"@media all"` wrapper nominally applies to all media; but if that wrapper is in a stylesheet included with the `media` attribute `"screen,projection"`, then the rules inside the `@media` wrapper actually only apply to `"screen"` and `"projection"`. This means that some rules may ultimately compute to no media at all, and in that case their media will be listed as `"none"`.

The value `"current"` refers to whatever media type applies to the current page view. In most situations it will be `"screen"`, but this value dynamically changes to match whatever it is. You can test this out easily in Opera, because it applies `"projection"` media when it's in Full Screen mode. *(**Please note:** detecting the current view media is not supported in Safari 3 or Konqueror 4; you can still use this feature but you'll always get rules for* `"screen"`. *Detection may also fail in* XML *(eXtensible Markup Language) environments, and if so it will also fallback to* `"screen"`*.)*

A value of `"all"` for this argument **does not include** `"none"` — if you want those rules to be included you can specify `"all,none"` or `"*"`.

CSSUtilities is currently **not sensitive** to **CSS media queries** <http://www.w3.org/TR/css3-mediaqueries/>, but neither does it remove them — you cannot specify a media query for this argument, and any queries that form part of a stylesheet's own `media` definition will be ignored in media-context evaluations; but any such queries are still listed as part of the rule's applicable media types.

So for example, if a rule had the media `"screen and (color)"`, it would be returned for any search where the media-context included `"screen"`, but its media type would still be listed as `"screen and (color)"`. *(**Please note:** there are also some caveats to the library's awareness of media query syntax: comma-delimited queries, those beginning with* `"not"`*, and those using shorthand syntax like* `"@media (color)"`*, are all misinterpreted as unique types in their own right, and will therefore only be returned when the media-context is* `"all"` *or* `"*"`*.)*

If undefined or empty this argument defaults to `"screen"`.

direct selectors only ***[OPTIONAL Boolean]***

A `Boolean` that specifies whether you only want selectors which explicitly select the element you're referring to (`true`), or you also want those which select an ancestor it can inherit from (`false`).

So for example, given this list of selectors:

- `html`
- `html > body`
- `body h1`
- `h1:not([id="xxx"])`

A selectors search for the `<h1>` element with a value of `true` for this argument would only return the **last two** selectors; but with a value of `false` it would return **all four**.

If undefined this argument defaults to `true`.

callback when complete ***[OPTIONAL Function]***

A `Function` that will be called once the method has completed. The function will be passed a single argument: the `selectors` `Array` that would otherwise be its return value.

Optional arguments between the required arguments and the callback can be omitted entirely, and their default values will be inserted automatically. As long as the *required* arguments are there, it doesn't matter how many intermediate optional arguments are missing: the callback can always be the last one. So both syntaxes in the example below are valid and interchangeable:

```
var selectors = CSSUtilities.getCSSSelectors("#foo", "screen", true, myCallback);

var selectors = CSSUtilities.getCSSSelectors("#foo", myCallback);
```

This is designed to be used with asynchronous execution, and is sensitive to the initialization state of the library — if init has not been called, or has not finished, the method will wait until it has, and only then fire the specified callback. You can see an example of its use in the Asynchronous with auto-init design pattern.

This argument has no default value if undefined.

Return value *[Array]*

Return value

The method returns an `Array` of zero or more selectors, each of which is a `String` CSS *(Cascading Style Sheets)* selector, trimmed of leading or trailing whitespace.

The method will return every selector that applies to the element, including those that only apply to alternate states, such as `:hover` and `:focus`. But it won't return pseudo-element selectors, because they don't actually apply to the element.

If a single matching rule has multiple comma-delimited selectors, such as `"html,body"` then these will be split and **returned separately**; so in that example you'd get two selectors, `"html"` and `"body"`. But if only some of the individual selectors apply to the specified element or its ancestors, then only those which apply will be returned.

If there are no selectors for the specified element and media, this method will return an empty `Array`.

`CSSUtilities.getCSSSelectorSpecificity(selector [, element] [, oncomplete])`

The `getCSSSelectorSpecificity` method tells you the specificity of any CSS *(Cascading Style Sheets)* selector, either as an absolute value, or relative to an element reference.

```
var specificity = CSSUtilities.getCSSSelectorSpecificity(
    "body > div",          // selector text              [REQUIRED String]
    "#heading"             // element reference or ID    [OPTIONAL Object|String]
    function() { ... }     // callback when complete     [OPTIONAL Function]
    );
```

Arguments

selector text *[REQUIRED String]*

A `String` that is the CSS *(Cascading Style Sheets)* selector you want to test.

You can only pass **one selector at a time** to this method, it can't accept multiple comma-delimited selectors and will throw the multiple selectors error if you try.

element reference or ID *[OPTIONAL Object|String]*

An optional reference to a context element against which the selector should be evaluated. This can either be an `Object` reference to the element itself, or a `String` which is the element's `ID` plus a leading `"#"` (like an ID-selector, eg. `"#content"`).

If you omit this argument or it's `null`, then the value you get back from this method will be that selector's **absolute specificity**, ie. its specificity calculated purely from the syntax of the selector. For example, the selector `"div#content > p"` has a specificity of `[0,1,0,2]` , since it contains one ID selector and two element-type selectors.

But if do you pass an element reference here, the value you get back will be **relative to that element**: so if the selector in question *explicitly selects* the element, you'll still get an absolute specificity; if it doesn't select the element but does select one of its ancestors, you'll get an **inherited specificity**, which is always `[0,0,0,0]`; if it doesn't select the element *or* any if its ancestors, you'll get `null`.

If undefined this argument defaults to `null`.

callback when complete ***[OPTIONAL Function]***

A `Function` that will be called once the method has completed. The function will be passed a single argument: the `specificity Array` that would otherwise be its return value.

Optional arguments between the required arguments and the callback can be omitted entirely, and their default values will be inserted automatically. As long as the *required* arguments are there, it doesn't matter how many intermediate optional arguments are missing: the callback can always be the last one. So both syntaxes in the example below are valid and interchangeable:

```
var selectors = CSSUtilities.getCSSSelectorSpecificity("body > div", null, myCallback);

var selectors = CSSUtilities.getCSSSelectorSpecificity("body > div", myCallback);
```

This is designed to be used with asynchronous execution, and is sensitive to the initialization state of the library — if init has not been called, or has not finished, the method will wait until it has, and only then fire the specified callback. You can see an example of its use in the Asynchronous with auto-init design pattern.

This argument has no default value if undefined.

Return value ***[Array|Object]***

Return value

The method returns an `Array` with exactly four members, each of which is a `Number` representing the value of a specificity score group (as defined in **CSS3** <http://www.w3.org/TR/css3-selectors/#specificity> from a root specification in **CSS2.1** <http://www.w3.org/TR/2009/CR-CSS2-20090908/cascade.html#specificity>). From highest to lowest, these are:

1. The style attribute
2. ID selectors
3. Class selectors, attribute selectors and pseudo-classes
4. Element-type selectors and pseudo-elements

Specificity values work in an infinite number base: if you start with a value of `1119` and add `1` to the final digit, you don't get `1120`, you get `111A` (ie. `1-1-1-10`).

The point is that **a value in one specificity group can never equal or overtake a value in a higher group**. Or to put that in CSS *(Cascading Style Sheets)* terms, no amount of element-types selectors can ever have more specificity than just one ID selector.

It's impractical then to document specificity values as single numbers at all, because of the number base they operate in — it makes much more sense to document them as arrays. To cite the earlier example again, if you start with `[1,1,1,9]` and add `1` to the final digit, then obviously you'll get `[1,1,1,10]`.

So, this method returns an `Array` of four digits. If you specified an element for the second argument, and the specified selector selects one of its ancestors but not the element itself, this method will return `[0,0,0,0]`. If the specified selector doesn't select the element nor any of its ancestors, this method will return `null`.

`CSSUtilities.getCSSStyleSheetRules([media] [, accept] [, ssid] [, oncomplete])`

The `getCSSStyleSheetRules` method returns all the rules in the library's data cache (ie. all the rules that apply to the current page), within specified media, and/or from a specified stylesheet. If all media and all stylesheets are specified, this method will return the entire data cache.

```
var rules = CSSUtilities.getCSSStyleSheetRules(
    "screen",              // media context              [OPTIONAL String]
    "selector,css",        // properties to accept       [OPTIONAL String]
    -1,                    // single stylesheet ID       [OPTIONAL Number]
    function() { ... }     // callback when complete     [OPTIONAL Function]
    );
```

Arguments

media context *[OPTIONAL String]*

A comma-delimited `String` that specifies which media-types the returned rules should apply to. You can specify `"*"` for all media, or any valid CSS *(Cascading Style Sheets)* media types (including `"all"`); there are also two special values: `"none"` and `"current"`.

The value `"none"` means, rules which apply don't apply to any media. The media context evaluation is implemented using each rule's **computed** media, which may be different from the media-types that were actually defined for it. For example, an `"@media all"` wrapper nominally applies to all media; but if that wrapper is in a stylesheet included with the `media` attribute `"screen,projection"`, then the rules inside the `@media` wrapper actually only apply to `"screen"` and `"projection"`. This means that some rules may ultimately compute to no media at all, and in that case their media will be listed as `"none"`.

The value `"current"` refers to whatever media type applies to the current page view. In most situations it will be `"screen"`, but this value dynamically changes to match whatever it is. You can test this out easily in Opera, because it applies `"projection"` media when it's in Full Screen mode. *(**Please note:** detecting the current view media is not supported in Safari 3 or Konqueror 4; you can still use this feature but you'll always get rules for* `"screen"`. *Detection may also fail in* XML *(eXtensible Markup Language) environments, and if so it will also fallback to* `"screen"`*.)*

A value of `"all"` for this argument **does not include** `"none"` — if you want those rules to be included you can specify `"all,none"` or `"*"`.

CSSUtilities is currently **not sensitive** to **CSS media queries** <http://www.w3.org/TR/css3-mediaqueries/>, but neither does it remove them — you cannot specify a media query for this argument, and any queries that form part of a stylesheet's own `media` definition will be ignored in media-context evaluations; but any such queries are still listed as part of the rule's applicable media types.

So for example, if a rule had the media `"screen and (color)"`, it would be returned for any search where the media-context included `"screen"`, but its media type would still be listed as `"screen and (color)"`. *(**Please note:** there are also some caveats to the library's awareness of media query syntax: comma-delimited queries, those beginning with* `"not"`*, and those using shorthand syntax like* `"@media (color)"`*, are all misinterpreted as unique types in their own right, and will therefore only be returned when the media-context is* `"all"` *or* `"*"`*.)*

If undefined or empty this argument defaults to `"screen"`.

data to accept *[OPTIONAL String]*

A comma-delimited `String` that lists what data you want to returned `rules` collection to include. Each rule in the returned collection is an `Object` that includes only the members you specify here — values such as `"selector"` to include the selector-text of each rule, or `"media"` to include the media each rule applies to.

Note that this argument has a different range of values than the corresponding argument in the `getCSSRules` *method; it does work in the same way though —*

You can specify `"*"` to return *all* available data, or any of the following members:

- `"selector"` (the rule's selector text)
- `"css"` (the complete CSS *(Cascading Style Sheets)* text of the rule)

- `"media"` (the computed media-types that the rule applies to)
- `"xmedia"` (the media-types that the author defined for the rule)
- `"owner"` (a representation of the rule's grammatical context)
- `"href"` (the qualified URL *(Uniform Resource Locator)* of the rule's owning stylesheet)
- `"ssid"` (the internal ID of the rule's owning stylesheet in the library's data cache)
- `"index"` (the index of this rule in the library's data cache)
- `"properties"` (the individual CSS *(Cascading Style Sheets)* properties for this rule)

You'll find more details about the structure and content of each member in the return value documentation (or follow the individual links).

If undefined or empty this argument defaults to `"*"` (all available data).

single stylesheet ID ***[OPTIONAL Number]***

A `Number` which with you can specify that you're only interested in rules from one specific stylesheet.

The number itself refers to the internal `ssid` (StyleSheet ID) of the stylesheet, which can be found as the `ssid` property of each stylesheet returned by getCSSStyleSheets(), or each rule returned by getCSSRules() (for its owning stylesheet). If you refer to an `ssid` that doesn't exist, you'll get an invalid ssid error.

If undefined, this argument defaults to `-1`, which means "all stylesheets".

callback when complete ***[OPTIONAL Function]***

A `Function` that will be called once the method has completed. The function will be passed a single argument: the `rules Array` that would otherwise be its return value.

Optional arguments before the callback can be omitted entirely, and their default values will be inserted automatically. It doesn't matter how many intermediate optional arguments are missing: the callback can always be the last one. So both syntaxes in the example below are valid and interchangeable:

```
var rules = CSSUtilities.getCSSStyleSheetRules("screen", "*", -1, myCallback);

var rules = CSSUtilities.getCSSStyleSheetRules(myCallback);
```

This is designed to be used with asynchronous execution, and is sensitive to the initialization state of the library — if init has not been called, or has not finished, the method will wait until it has, and only then fire the specified callback. You can see an example of its use in the Asynchronous with auto-init design pattern.

This argument has no default value if undefined.

Return value ***[Array]***

Return value

The method returns an `Array` of zero or more rules, in cascade order, each member of which is an `Object` that contains some or all of the following members (as specified in the `accept` argument):

"selector" ***[String]***

A `String` containing the complete selector for this rule (equivalent to the `selectorText` property).

If the rule has multiple comma-delimited selectors, then the entire selector will normally be returned. But the one exception to this is Internet Explorer when the library is in "browser" mode, in which case the browser's internal CSS *(Cascading Style Sheets)* parsing splits such rules into multiple individual rules, each with a single selector (eg. a rule with the selector `"html,body"` would become two rules with `"html"` and `"body"`).

"css" ***[String]***

A `String` containing the complete CSS text of the rule (equivalent to the `style.cssText` property).

"media" ***[String]***

A `String` containing the computed media-types that this rule applies to, comma-delimited if more than one type applies. If a rule applies to all media this will have the value `"all"`; if it applies to no media it will have the value `"none"`.

This is a *computed* value — it returns only those types that the rule actually applies to, which may be different from the media that were originally specified for it. There's more info about that in the media context argument documentation.

"xmedia" ***[String]***

A `String` containing the media-types that were originally specified for the rule, which may be different from its computed media. Depending on how the rule was included, this might be the media specified in an `@media` statement, or added to the end of an `@import`, or specified in the media attribute of a `<link>` element.

This data is **only available in author mode**, and does not form part of any media context evaluation. However it may be useful for comparison, or for re-creating the original CSS *(Cascading Style Sheets)* source-code for a set of rules.

"owner" ***[String]***

A `String` that represents the immediate grammatical context of this rule, according to what surrounds it and how the stylesheet that contains it was included. It will be one of the following fixed values:

- `"style"` (a rule inside a `<style>` block)
- `"link"` (a rule inside a stylesheet that was included using a `<link>` element)
- `"xml-stylesheet"` (a rule inside a stylesheet that was included using an `<?xml-stylesheet?>` processing instruction)
- `"@import"` (a rule inside a stylesheet that was included using an `@import` statement)
- `"@media"` (a rule inside an `@media` block)

"href" ***[String|Object]***

A `String` which is the qualified URL *(Uniform Resource Locator)* of this rule's owning stylesheet. This will always be a fully-qualified address, irrespective of the syntax that was originally used.

If the stylesheet has no URL *(Uniform Resource Locator)* (like a rule inside a `<style>` block) then this will be `null`.

"ssid" ***[Number]***

A `Number` which is the internal ID of this rule's owning stylesheet within the library's data cache.

The value represents the order in which the stylesheet includes occur — that is, the source-order of the elements or `@import` statements that declare them — which may differ from the source-order of the rules inside them, because of the way the cascade works (rules inside imported stylesheets come *before* the rules in their parent).

For a rule that represents the element's `style` attribute, this will be `Infinity`.

"index" ***[Number]***

A `Number` which is the index of this rule in the library's data cache.

This value is used as part of the library's specificity sorting algorithms, and is also the rule's overall source index, and therefore its position in the cascade.

"properties" ***[Object]***

An `Object` containing the individual CSS *(Cascading Style Sheets)* properties for this rule, extracted from its css text, and indexed in name/value pairs (the same format returned by the `getCSSProperties` method), for example:

```
{
    "color": "#333 !important",
    "font-weight": "bold",
    "border": "none"
}
```

If you're working in "browser" mode, this object may contain many more properties than you're expecting, because of the action of browser normalization — browsers converting CSS *(Cascading Style Sheets)* property definitions to a different syntax, for their own convenience — Internet Explorer, for example, splits-out `margin` and `padding` shorthands into their component longhand definitions; Opera does the same thing with `border`. You may also notice changes in units and values, such as hex *(hexadecimal)* colors converted to RGB *(Red Green Blue)* in Firefox.

This object will contain *all* the properties defined for this rule, with no further information about which are active, inactive or cancelled (if you want that information you should use the `getCSSRules` method for a specific element).

If there are no properties defined for this rule, this object will be `null`.

Properties defined in `style` attributes **are not included** in the rules returned by this method (if you want that information you should use the `getCSSRules` method for a specific element).

If no rules have been defined for the page, this method will return an empty `Array`.

`CSSUtilities.getCSSStyleSheets([oncomplete])`

The `getCSSStyleSheets` method returns all the stylesheets in the library's data cache (ie. all the stylesheets that apply to the page). This method is primarily intended to provide control data, and lists every stylesheet reference found during initialization , whetheror not any useful data was ultimately retrieved from it.

```
var stylesheets = CSSUtilities.getCSSStyleSheets(
    function() { ... }     // callback when complete       [OPTIONAL Function]
    );
```

Arguments

callback when complete ***[OPTIONAL Function]***

A `Function` that will be called once the method has completed. The function will be passed a single argument: the `stylesheets Array` that would otherwise be its return value.

This is designed to be used with asynchronous execution, and is sensitive to the initialization state of the library — if init has not been called, or has not finished, the method will wait until it has, and only then fire the specified callback. You can see an example of its use in the Asynchronous with auto-init design pattern.

This argument has no default value if undefined.

Return value *[Array]*

Return value

The method returns an `Array` of zero or more stylesheets, in order of occurence, each member of which is an `Object` with the following members:

"ssid" ***[Number]***

A `Number` which is the internal ID of this stylesheet.

The value represents the order in which the stylesheet includes occur — that is, the source-order of the elements or `@import` statements that declare them — which may differ from the source-order of the rules inside them, because of the way the cascade works (rules inside imported stylesheets come *before* the rules in their parent).

"href" ***[String|Object]***

A `String` which is the stylesheet's qualified URL *(Uniform Resource Locator)* . This will always be a fully-qualified address, irrespective of the syntax that was originally used.

If the stylesheet has no URL *(Uniform Resource Locator)* (like with a `<style>` block) then this will be `null`.

"owner" ***[String]***

A `String` that represents this stylesheet's immediate grammatical context, according to how it was created or included. It will be one of the following fixed values:

- `"style"` (a stylesheet created as a `<style>` block)
- `"link"` (a stylesheet included with a `<link>` element)
- `"xml-stylesheet"` (a stylesheet included with an `<?xml-stylesheet?>` processing instruction)
- `"@import"` (a stylesheet included with an `@import` statement)

"media" ***[String]***

A `String` containing the computed media-types that this stylesheet applies to, comma-delimited if more than one type applies. If a stylesheet applies to all media this will have the value `"all"`; if it applies to no media it will have the value `"none"`.

This is a *computed* value — it returns only those types that the rule actually applies to, which may be different from the media that were originally specified for it. For example, an ordinary `"@import"` statement nominally applies to all media; but if that import is in a stylesheet included with the `media` attribute `"screen,projection"`, then the rules inside the imported stylesheet actually only apply to `"screen"` and `"projection"`. This of course means that some stylesheets may ultimately compute to no media at all, and in that case their media will be listed as `"none"`.

"xmedia" ***[String]***

A `String` containing the media-types that were originally specified for the stylesheet, which may be different from its computed media. Depending on how the stylesheet was included, this might be the media added to the end of an `@import` statement, or specified in the media attribute of a `<link>` element.

This data is **only available in author mode**. It may be useful for comparison, or for re-creating the original source-code declaring the stylesheet.

"stylesheet" or "stylenode" ***[Object]***

In browser mode this property will be called `"stylesheet"`, and is a reference to the actual `CSSStyleSheet` object. This reference can be used to probe into the stylesheet's CSS DOM *(Cascading Style Sheets Document Object Model)* as far as you wish, but of course what you find there will vary significantly by browser and implementation.

In author mode this property will be called `"stylenode"`, and is a reference to the top-level node that included this stylesheet, such as a `<link>` element or an `<?xml-stylesheet?>` processing instruction. In most cases this will be the owning element of the stylesheet itself, but for stylesheets included using an `@import` statement, it will be the owning element of the import's parent stylesheet (or keep checking upwards until we find a node).

"rules" ***[Number]***

A `Number` which is the total number of valid rules extracted from this stylesheet and saved to the data cache.

This may not be exactly the same as the actual number of rules defined in the stylesheet, for a few possible reasons. One reason is that Internet Explorer in "browser" mode splits-up rules that have multiple selectors into multiple rules, each with a single selector. Another is that disabled stylesheets are not usually parsed (unless the "watch" setting is `null`), and will therefore return zero rules.

"message" ***[String]***

A `String` which is a short message describing the result of the parsing operation on this stylesheet. If everything went okay this will simply say `"OK"`, otherwise it will be one of the data-retrieval and parsing error messages listed below.

If no stylesheets have been defined for the page, this method will return an empty `Array`.

Error messages

CSSUtilities provides comprehensive error-trapping and reporting; you may encounter custom errors that are thrown while using its methods, or as messages in the returned data itself.

Execution errors

Execution errors are those caused by incorrect usage or missing resources, and are thrown to the browser's error console as soon as they occur; in most cases these errors are fatal and will prevent any further use.

`CSSUtilities (Fatal Error): The specified mode is not valid`

Thrown by the `define` method for an invalid definition of "mode".

`CSSUtilities (Fatal Error): The specified async setting is not valid`

Thrown by the `define` method for an invalid definition of "async".

`CSSUtilities (Fatal Error): The specified document is not a Document`

Thrown by the `define` method for an invalid definition of "page".

`CSSUtilities (Fatal Error): The specified base is not an absolute URL`

Thrown by the `define` method for an invalid definition of "base".

`CSSUtilities (Fatal Error): The specified attributes setting is not valid`

Thrown by the `define` method for an invalid definition of "attributes".

`CSSUtilities (Fatal Error): The specified watch setting is not valid`

Thrown by the `define` method for an invalid definition of "watch".

`CSSUtilities (Fatal Error): The specified api settings are not valid`

Thrown by the `define` method for an invalid definition of "api".

`CSSUtilities (Fatal Error): Your Selectors API is not returning the right data`

Thrown by the `define` method after testing the wrapper defined for an external Selectors API if it fails to return either an `Array` or a `NodeList`.

`CSSUtilities (Fatal Error): The Selectors API is missing`

> Thrown by the internal `getElementsBySelector` method if the bundled Selectors API (`Selector.js`) is called and found to be missing.
>
> `CSSUtilities (Fatal Error): You cannot define "`*`[option]`*`" after "api", it must be defined first`
>
> Thrown by the `define` method if you attempt to define "base" or "page" *after* having defined "api", because they have to be defined first. The error message will include the name of the option in question (eg. "You cannot define "base" after "api" ...").
>
> `CSSUtilities (Fatal Error): Unable to communicate with the network`
>
> Thrown by the internal `ajaxload` method if it's unable to instantiate any network requests (most likely because no connection is available).
>
> `CSSUtilities.`*`[method name]`*` has an invalid element reference`
>
> Thrown by the `getCSSRules`, `getCSSProperties`, `getCSSSelectors` or `getCSSSelectorSpecificity` method if the element argument is neither a valid element nor the ID of a valid element. The error message will include the name of the calling method (eg. "CSSUtilities.getCSSRules has ...").
>
> `CSSUtilities.getCSSSelectorSpecificityrequires  a valid Selector reference`
>
> Thrown by the `getCSSSelectorSpecificity` method if the selector argument is invalid (eg. empty or undefined).
>
> `CSSUtilities.getCSSSelectorSpecificitycan  only process one selector at a time`
>
> Thrown by the `getCSSSelectorSpecificity` method if the selector argument contains multiple comma-delimited selectors, instead of just one.
>
> `CSSUtilities.getCSSStyleSheetRules has an invalid Stylesheet ID`
>
> Thrown by the `getCSSStyleSheetRules` method if the `ssid` argument is specified but invalid.

Data-retrieval and parsing errors

These refer to problems the script encountered while loading and parsing stylesheets during initialization , and are silently handled and recorded; any such messages are subsequently included in the message data returned by the `getCSSStyleSheets` method.

> `Network Failure or Security Violation`
>
> The library attempted to request a stylesheet but the request failed, either because of network problems, or because it violated the same-origin policy, but it isn't known which.
>
> `Network Failure`
>
> The library attempted to request a stylesheet, but the request failed because of network problems.
>
> `Security Violation`
>
> The library attempted to request a stylesheet, but the request failed because it violated the same-origin policy.
>
> `Data is not CSS`
>
> The library requested a stylesheet but the returned data had a MIME-type that was not `"text/css"`.
>
> *`[HTTP Errors]`*

The library requested a stylesheet but the request returned an identifiable HTTP *(HyperText Transfer Protocol)* error, such as "404 Not Found".

`Unspecified Error`

The library attempted to parse a stylesheet but failed for an unknown reason.

`Discarded Duplicate`

The library identified a stylesheet with exactly the same URL *(Uniform Resource Locator)* as another in its existing data set, and therefore disabled that stylesheet and ignored its rules. (Duplicates have to be disabled to prevent the possibility of infinite recursion.)

`Stylesheet is disabled`

The library identified a stylesheet that was already disabled, and determined that its rules should not be included because they don't apply to the current view. (In most cases this will be because it's an alternate stylesheet that's current inactive; monitoring changes caused by stylesheet switching is controlled by the "watch" setting.)

`Unsupported node type`

The library was unable to parse a stylesheet because it used an include method that the browser doesn't fully support. (This will only ever happen in Safari 3 with `<?xml-stylesheet?>` processing instructions, because it doesn't support the necessary pseudo-attributes.)

General specifications

The following is a technical breakdown of the language constructs and specifications that CSSUtilities supports, and those that it notably doesn't support:

CSSUtilities can load and extract data from the following sources:

- by reading stylesheets in the `document.styleSheets` collection
- by loading stylesheets directly from their source elements, for which all forms of stylesheet include are supported — `<link>` elements, `<style>` elements, `<?xml-stylesheet?>` processing isntructions, `@import` statements in other stylesheets, and `style` attributes in markup
- from documents which are HTML *(HyperText Markup Language)* , XHTML *(eXtensible HyperText Markup Language)* , or any other form of XML *(eXtensible Markup Language)*
- from physical source documents, or from virtualized DOMs *(Document Object Models)* such as the `responseXML` of an Ajax request

The library can parse and understand the following language constructs, both in standard implementations up to CSS3 *(Cascading Style Sheets Level 3)* , and the proprietary implementations in Internet Explorer 6, 7 and 8:

- all selectors, pseudo-classes and pseudo-elements (except namespaced selectors)
- any properties, with or without host-browser normalization
- all shorthands and the longhands they represent
- all static media types
- all inheritable properties
- specificity calculation
- source index
- `!important` rules
- `@media` statements
- `@import` statements, including validation of their source position

It **does not** understand the following constructs, and will either ignore them, or remove them from its data cache at initialization :

- `@font-face` (they don't affect the library's functionality and are ignored)
- `@charset` (likewise ignored)

> - `@namespace` (declarations are ignored; namespaced selectors are not supported and will be treated according to their literal syntax)
> - media queries (they're preserved in recorded media-types, but do not form part of any media-context evaluation)
> - vendor-specific pseudo-classes or pseudo-elements (eg. `::-moz-focus-inner`, which are therefore not included in specificity calculations)