API Management

Change the API symbol in the global variable namespace under which ComponentJS is exposed. By default ComponentJS is exposed under the symbol name `ComponentJS`. It is a common convention to change the symbol to `cs` (for "component system/service") to have a convenient short-hand.

```
ComponentJS.symbol([name: String]): ComponentJS
```

Change symbol of ComponentJS API to global variable *name* and return it. If *name* is not given, ComponentJS does not occupy any global namespace slot at all — then it is required to store the return value and use ComponentJS directly through it.

```
ComponentJS.symbol("cs")        /* standard    */
var cs = ComponentJS.symbol()   /* alternative */
```

```
ComponentJS.version = { major: Number, minor: Number, micro: Number, date: Number }
```

Access the ComponentJS implementation version "`major.minor.micro`" and the corresponding release `date` (in format YYYYMMDD).

```
if (ComponentJS.version.date < 20120101)
     throw new Error("need at least ComponentJS as of 20120101")
```

Library Management

ComponentJS can be extended through plugins, so it can initialize some of its internals only once all plugins were loaded and executed.

```
ComponentJS.bootstrap(): Void
```

Bootstrap the ComponentJS library by initializing its internals. This has to be called at least before any calls to **`create`**(), but can be called after any calls to **`symbol`**(), **`debug`**() or **`ns`**().

```
cs.bootstrap()
```

```
ComponentJS.shutdown(): Void
```

Shutdown the ComponentJS library by destroying its internals. This implicitly destroy the existing component tree, too.

```
cs.shutdown()
```

```
ComponentJS.plugin(): String[]
```

```
ComponentJS.plugin(name: String): Boolean
```

```
ComponentJS.plugin(name: String, callback: Function): Void
```

Return the names of all registered plugins, check for the registration of a particular plugin with name *name* or register a new plugin under name *name* with callback function *callback*. The callback function *callback* should have the signature "*callback*(*_cs*: *ComponentJS_API_internal*, *$cs*: *ComponentJS_API_external*, *GLOBAL*: *Environment*): *Void*" where *_cs* is the internal ComponentJS API (you have to check the source code of ComponentJS to know what you can do with it), *$cs* is the external ComponentJS API (the one described in this document) and *GLOBAL* is the global environment object (usually `window` in a browser, `global` in Node.js, etc).

```
/*  add a "foo()" method to all components  */
ComponentJS.plugin("foo", function (_cs, $cs, GLOBAL) {
    var trait = $cs.trait({
        protos: {
            foo: function () {
                ...
            }
        }
    });
    _cs.latch("ComponentJS:bootstrap:comp:mixin", function (mixins) {
        mixins.push(trait);
    });
});
```

Debugging

ComponentJS has special support for debugging its run-time processing, especially for visualizing the current component tree in real-time.

```
ComponentJS.debug(): Number
```

```
ComponentJS.debug(level: Number): Void
```

```
ComponentJS.debug(level: Number, message: String): Void
```

Get current debug level, or configure the debugging through maximum debug-level *level* (`0` disables all debug messages, 9 enables all debug messages) or log a particular *message* under debug-level *level*.

```
cs.debug(0)
if (cs.plugin("debugger")) {
    if (cs.debug_instrumented()) {
         cs.debug(9)
         cs.debug_window({ ... })
    }
}
```

```
ComponentJS.debug_instrumented(): Boolean
```

[AVAILABLE THROUGH THE COMPONENTJS PLUGIN "debugger" ONLY] Determine whether the current browser is "instrumented" for debugging, i.e., whether the browser's built-in debugger is activated (in detached mode only). Currently ComponentJS is able to determine Firefox's Firebug and Chrome's Inspector only.

```
if (cs.debug_instrumented()) ...
```

```
ComponentJS.debug_window({ enable: Boolean, autoclose: Boolean, name: String, width: Number = 800, height: Number = 600, natural: Boolean = false })
```

```
ComponentJS.debug_window(enable: Boolean, autoclose: Boolean, name: String)
```

[AVAILABLE THROUGH THE COMPONENTJS PLUGIN "debugger" ONLY] On `enable` `true`/`false` open/close the extra browser window containing the ComponentJS debugger view for the ComponentJS

application identified by `name`. If `autoclose` is `true`, optionally automatically close the debugger window with application window (which usually is inconvenient during debugging because on application reloads the ComponentJS debugger window is recreated with default width/height at default position instead of reused). Parameters `width` and `height` can be used to change the initial window size. Parameter `natural` controls whether the component tree is drawn with the root component at the bottom (`true`) or at the top (`false`).

```
cs.debug_window({
    enable: true,
    autoclose: false,
    name "My App"
    width: 800,
    height: 800,
    natural: true
})
```

Code Structuring

ComponentJS internally uses a few code structuring utility functions for implementing class method parameters and class attributes. Those utility functions are also exposed for consumption by application developers, but they are NOT(!) required for using ComponentJS. Especially, it is NOT(!) required that component backing objects are defined by `cs.clazz`!

```
ComponentJS.ns(path: String[, leaf: Object = {}]): Object
```

Classes and traits should be structured through namespaces. A namespace is a JavaScript (hash) object, potentially itself part of a parent namespace object. The top-most implicit namespace object is `window`. A namespace has a dot-separated fully-qualified symbol path like `foo.bar.quux`. This method allows to create the fully-qualified path of nested objects through the dot-separated *path* of object names, optionally assign the right-most/leaf object to *leave* and finally return the right-most/leaf *Object*.

```
cs.ns("my.app"); my.app.ui = cs.clazz({ ... })  /* standard    */
cs.ns("my.app").ui = cs.clazz({ ... })          /* alternative */
cs.ns("my.app.ui", cs.clazz({ ... })            /* alternative */
```

```
ComponentJS.select(object: Object, path: String[, value: Object]): Object
```

Dereference into (and this way subset) *object* according to the *path* specification and either return the dereferenced value or set a new *value*. Object has to be a hash or array object. The *path* argument has to follow the following grammar (which is a direct JavaScript dereferencing syntax):

```
path       ::=segment segment*
segment    ::=bybareword | bykey
bybareword::="."? identifier
bykey      ::="[" key "]"
identifier ::=/[_a-zA-Z$][_a-zA-Z$0-9]*/
key        ::=number | squote | dquote
number     ::=/[0-9]+/
dquote     ::=/"(?:\\"|.)*?"/
squote     ::=/'(?:\\'|.)*?'/
```

Setting the *value* to `undefined` effectively removes the dereferenced value. If the dereferenced parent object is a hash, this means the value is `delete`'ed from it. If the dereferenced parent object is an array, this means the value is `splice`'ed out of it.

```
cs.select({ foo: { bar: { baz: [ 42, 7, "Quux" ] } } }, "foo['bar'].baz[2]") → "Quux"
```

```
ComponentJS.validate(object: Object, spec: String): Boolean
```

Validate an arbitrary nested JavaScript object *object* against the specification *spec*. The specification *spec* has to be either a RegExp object for *String* validation, a validation function of signature

"*spec*(*Object*): *Boolean*" or a string following the following grammar (which is a mixture of JSON-like structure and RegExp-like quantifiers):

```
spec    ::=not | alt | hash | array | any | primary | class | special
not     ::="!" spec
alt     ::="(" spec ("|" spec)* ")"
hash    ::="{" (key arity? ":" spec ("," key arity? ":" spec)*)? "}"
array   ::="[" (spec arity? ("," spec arity?)*)? "]"
arity   ::="?" | "*" | "+" | "{" number "," (number | "oo") "}"
number::=/^[0-9]+$/
key     ::=/^[_a-zA-Z$][_a-zA-Z$0-9]*$/ | "@"
any     ::="any"
primary::=/^(?:null|undefined|boolean|number|string|function|object)$/
class   ::=/^[A-Z][_a-zA-Z$0-9]*$/
special ::=/^(?:clazz|trait|component)$/
```

The special key "@" can be used to match an arbitrary hash element key.

```
cs.validate({ foo: "Foo", bar: "Bar", baz: [ 42, 7, "Quux" ] },
     "{ foo: string, bar: any, baz: [ number+, string* ], quux?: any }")
```

`ComponentJS.`**`params`**`(`*name*`: String, `*args*`: Object[], `*spec*`: Object): Object`

Handle positional and named function parameters by processing a function's `arguments` array. Parameter *name* is the name of the function for use in exceptions in case of invalid parameters. Parameter *args* usually is the JavaScript `arguments` pseudo-array of a function. Parameter *spec* is the parameter specification: each key is the name of a parameter and the value has to be an *Object* with the following possible fields: *pos* for the optional position in case of positional usage, *def* for the default value (of not required and hence optional parameters), *req* to indicate whether the parameter is required and *valid* for type validation (either a string accepted by the **`validate`**() method, or a valid regular expression C object for validating a *String* against it or an arbitrary validation callback function of signature "*valid*(*Object*): *Boolean*".

```
function config () {
    var params = $cs.params("config", arguments, {
        scope: { pos: 0, req: true,      valid: "boolean"           },
        key:   { pos: 1, req: true,      valid: /^[a-z][a-z0-9_]*$/ },
        value: { pos: 2, def: undefined, valid: "object"            },
        force: {         def: false,     valid: "boolean"           }
    });
    var result = db_get(params.scope, params.key);
    if (typeof params.value !== "undefined")
        db_set(params.scope, params.key, params.value, params.force);
    return result;
}
var value = config("foo", "bar");
config("foo", "bar", "quux");
config({ scope: "foo", key: "bar", value: "quux", force: true });
```

`ComponentJS.`**`attribute`**`({ name: String, def: Object, valid: Object }): Function`

`ComponentJS.`**`attribute`**`(`*name*`: String, `*def*`: Object, `*valid*`: Object): Function`

Create a cloneable attribute capturing getter/setter function with name *name* (for exception handling reasons only), the default value *def* and the value validation *valid*.

```
var id = ComponentJS.attribute("id", "foo", /^[a-z][a-zA-Z0-9_]*/);
id() === "foo"
id("bar") → "foo"
id() → "bar"
```

`ComponentJS.`**`clazz`**`({ [name: String,] [extend: Clazz,] [mixin: Array(Trait),] [cons: Function,] [dynamics: Object,] [protos: Object,] [statics: Object] }): Clazz`

Define a JavaScript Class, optionally stored under the absolute dot-separated object path `name`, optionally extending the base/super/parent Class `extend` and optionally mixing in the functionality of one or more Traits via `mixin`. The class can have a constructor function `cons` which is called once the Class is instantiated and which can further initialize the dynamic fields of the class. On each instantiation, all fields which are specified with `dynamics` are cloned and instantiated and all methods in `protos` are copied into the Class `prototypes` object. The `statics` content is copied into the Class itself only. In case of `extend` and/or `mixin`, both the `cons` and methods of `protos` can call `this.base(...)` for the base/super/parent method.

```
var foo = cs.clazz({
    cons: function (bar) {
        this._bar = bar;
    },
    protos: {
        bar: function (value_new) {
            var value_old = this._bar;
            if (typeof value_new !== "undefined")
                this._bar = value_new;
            return value_old;
        }
        [...]
    }
})
```

It is important to notice how calls to any method resolve and how calls to `this.base()` in any method of a class resolves. When on class Foo and its instanciated object `foo` a method `foo.bar()` is called, the following happens:

- First, a direct property named `bar` on object `foo` is tried. This can exist on `foo` through (in priority order) a `bar` in either the `dynamics` definition of a `mixin` of Foo, or in the `statics` definition of a `mixin` of Foo, or in the `dynamics` definition of Foo, or in the `statics` definition of Foo.
- Second, an indirect prototype-based property named `bar` on object `foo` is tried. This can exist on `foo` through (in priority order) a `bar` in either the `protos` definition of Foo or in the `protos` definition of any `extend` of Foo.

When on class Foo and its instanciated object `foo` in any method `foo.bar()` the `this.base()` is called, the following happens:

- First, a call to the super/base/parent functions in the `mixin` trait chain is attempted. The mixins are traversed in the reverse order of the trait specification in the `mixin` array, i.e., the last trait's mixins are tried first.
- Second, a call to the super/base/parent functions in the `extend` inheritance class chain is attempted. First, the directly `extend` class is attempted, then the `extend` class of this class, etc.

NOTICE: As ComponentJS does not care at all how backing objects of components are defined, you can alternatively use an arbitrary solution for Class-based OO in JavaScript (e.g. TypeScript, JSClass, ExtendJS, DejaVu, Classy, jTypes, etc) or fallback to the also just fine regular Prototype-based OO in JavaScript:

```
var foo = function (bar) {
    this._bar = bar;
}
foo.prototype.bar = function (value_new) {
    var value_old = this._bar;
    if (typeof value_new !=== "undefined")
        this._bar = value_new;
    return value_old;
}
[...]
```

```
ComponentJS.trait({ [name: String,] [mixin: Array(Trait),] [cons: Function,] [setup:
Function,] [dynamics: Object,] [protos: Object,] [statics: Object] }): Trait
```

Define a JavaScript Trait (a Class which can be mixed in), optionally stored under the absolute dot-separated object path `name` and optionally mixing in the functionality of one or more other Traits via `mixin`. The trait can have a constructor function `cons` which is called once the Class the Trait is mixed in is instantiated and which can further initialize the dynamic fields of the Class. On each instantiation, all fields which are specified with `dynamics` are cloned and instantiated and all methods in `protos` are copied into the Class `prototypes` object. The `statics` content is copied into the Class itself only. The optional `setup` function is called directly at the end of Class definition (not instantiation) and can further refine the defined Class.

```
var foo = cs.trait({
    protos: {
        bar: function () {
            [...]
        }
    }
})
```

Component Creation

Components are managed in hierarchical fashion within a component tree. The component tree can be traversed and its components can be created, looked up, state transitioned, communicated on and be destroyed.

`ComponentJS.create(`*abs-tree-spec*`: String, `*class*`: Class[, ...]): Object`

`ComponentJS.create(`*base*`: Component, `*rel-tree-spec*`: String, `*class*`: Class[, ...]): Object`

component`.create(`*rel-tree-spec*`: String, `*class*`: Class[, ...]): Object`

Create one or more components. Their structure is specified by the absolute (*abs-tree-spec*) or relative (*rel-tree-spec*) tree specification which is string containing a set (`{...}`) of slash-separated (`.../...`) paths of component names. In other words, the specification has to follow the following grammar:

```
abs-tree-spec::="/" rel-tree-spec
rel-tree-spec  ::=path | "{" path ("," path)* "}"
path           ::=rel-tree-spec | name ("/" name)*
name           ::=/^[^\/]+$/
```

For instance, the specification `foo/{bar/baz,quux}` is the tree consisting of the two maximum length paths: `foo/bar/baz` and `foo/quux`. For each name from left-to-right in the tree specification you have to give either a to be instantiated class constructor (*Function*) or an already instantiated object (*Object*).

The **`create`**() method returns the last created component, i.e., the right-most component in the tree specification.

```
cs.create("/{sv,ui/{one,two}}", my.sv, {}, my.ui.one, my.ui.two);
cs.create(this, "model/view", model, view);
cs(this).create("model/view", model, view);
```

`ComponentJS.destroy(`*abs-path*`: String): Void`

component`.destroy(): Void`

component`.destroy(): Void`

Destroy the component uniquely identified by *abs-path* or the *component* on which this method is called upon.

```
cs.destroy("/foo/bar")
cs.destroy(comp, "foo/bar")
cs("/foo/bar").destroy()
```

Component Information

Components carry a few distinct information. They can be accessed via the following getter/setter-style methods.

component.**id**(): *String*

component.**id**(*id*: *String*): *String*

Get current unique id of *component* or set new *id* on *component* and return the old id. Setting the id of a component should be not done by the application as it is done by ComponentJS internally on component creation time.

```
cs(this).id() → "00000000000000000000000000000001"
```

component.**name**(): *String*

component.**name**(*name*: *String*): *String*

Get current non-unique name of *component* or set new *name* on *component* and return the old name. Setting the name of a component should be not done by the application as it is done by ComponentJS internally on component creation time.

```
cs("/foo/bar").name() === "bar"
```

component.**obj**(): *Object*

Retrieve the backing *Object object* to the corresponding *Component*.

```
cs(this).obj() === this
```

component.**cfg**(): *Array(String)*

component.**cfg**(*key*: *String*): *Object*

component.**cfg**(*key*: *String*, *value*: *Object*): *Object*

component.**cfg**(*key*: *String*, undefined): *Object*

Components can have key/value pairs attached for application configuration purposes. Four use cases exists for this method: 1. get array of all key strings, 2. get current configuration property identified by *key*, 3. set configuration property identified by *key* to new value *value* and return the old value, and 4. delete the configuration property identified by *key*.

```
var value = cs("/foo/bar").cfg("quux")
cs("/foo/bar").cfg("quux", value)
cs("/foo/bar").cfg("quux", undefined)
```

Component Lookup

Before performing certain operations on a component, it first have to be looked up in the component tree. As this is one of the most prominent functionalities of ComponentJS, it is directly exposed through the global API symbol.

ComponentJS(*abs-path*: *string*): *Component*

ComponentJS(*component*: *Component*, *rel-path*: *String*): *Component*

ComponentJS(*object*: *Object*, *rel-path*: *String*): *Component*

ComponentJS(*component*: *Component*): *Component*

ComponentJS(*object*: *Object*): *Component*

Components can be looked up by absolute/relative paths from root/base components. A path is a string of slash-separated component names with four special names allowed: "." for current component name, ".." for parent component name, "*" for any component name and an empty name (C) for any component trees between current and following components. In any case, the result has to uniquely identify a single component. The following usages exist: 1. Lookup *Component* by absolute path *path* (this is usually never done explicitly, but occurs implicitly if the input parameter is already a *Component*). 2. Lookup *Component* by path *path*, relative to *Component* *component*. 3. Lookup *Component* by path *path*, relative to the *Component* corresponding to *Object* *object*. 4. Lookup *Component* object via backing object *object*. 5. Lookup *Component* object via the *component* itself (no-operation). The paths have to follow the following grammar:

abs-path::="/" rel-path
rel-path ::=name ("/" name)*
name ::="" | "*" | /^[^\/]+$/

```
cs("/foo/bar")           /* absolute */
cs(comp, "model/view")   /* relative to component */
cs(this, "model/view")   /* relative to component via backing object */
cs("//bar")              /* full-tree lookup */
cs(comp, "//bar")        /* sub-tree lookup */
cs(this, "*/view")       /* wildcard lookup */
cs(this, "..//view")     /* parent sub-tree lookup */
```

component.**exists**(): *Boolean*

Check whether a (usually previously looked up) *component* (either a real existing on or the special pre-existing singleton component with name "<none>") really exists in the component tree.

```
if (cs("//quux").exists()) ...
if (cs("//quux").name() !== "<none>") ...
```

Component Tree

Components are managed within a component tree. The following functions allow you to traverse this tree.

component.**path**(): *Array(Component)*

component.**path**(*separator*: *String*): *String*

Either retrieve as an array all *Components* from the current *component* up to and including the root component, or get the slash-separated component name path *String* from the root component down to and including the current *component*.

```
cs("/foo/bar").path("/") → "/foo/bar"
cs("/foo/bar").path() → [ cs("/foo/bar"), cs("/foo"), cs("/") ]
```

component.**parent**(): *Component*

Return the parent component of *component*, or `null` if *component* is the root or none component.

```
cs(this).parent() === cs(this, "..")
```

component.**children**(): *Array(Component)*

Return the array of child components of *component*.

```
cs(this).children()
```

component.**attach**(*parent*: *Component*): *Void*

Attach *component* as a child to the *parent* component. In case it is already attached to an old parent component, it automatically calls **detach**() before attaching to the new parent component. Internally used by ComponentJS on **create**(), but can be also used by application when moving a sub-tree within the component tree.

```
/*  migrate all children from our view1 onto our view2  */
var view1 = cs(this, "model/view1")
var view2 = cs(this, "model/view2")
view1.children().forEach(function (child) {
    var state = child.state({ state: "created", sync: true })
    child.detach()
    child.attach(view2)
    child.state(state)
})
```

component.**detach**(): *Void*

Detach *component* as a child from its parent component. Internally used by ComponentJS on **destroy**(), but can be also used by application when moving components within the component tree.

```
cs(this).detach()
```

component.**walk_up**(*callback*: *Function*, *ctx*: *Object*): *Object*

Walk the component tree upwards from the current component (inclusive) to the root component (inclusive). The *callback Function* has to be of signature *callback*(*depth*: *Number*, *component*: *Component*, *ctx*: *Object*): *Object* and for each component it is called like "`ctx = callback(depth++, comp, ctx)`" where initially `ctx`=*ctx*, `comp`=*component* and `depth=0` was set.

```
var path = cs(this).walk_up("", function (depth, comp, ctx) {
    return "/" + comp.name() + ctx;
}, "")
```

component.**walk_down**(*callback*: *Function*, *ctx*: *Object*): *Object*

Walk the component tree downwards from the current component (inclusive) to all the transitive child components (inclusive). The *callback Function* has to be of signature *callback*(*ctx*: *Object*, *component*: *Component*, *depth*: *Number*, *depth_first*: *Boolean*): *Object* and for each component it is called twice(!): once like "`ctx = callback(depth, comp, ctx, false)`" when entering the component (before all children will be visited) and once like "`ctx = callback(depth, comp, ctx, true)`" when leaving a component (after all children were visited). Initially `ctx`=*ctx*, `comp`=*component* and `depth=0` is set.

```
var output = cs(this).walk_down(
    function (depth, comp, output, depth_first) {
        if (!depth_first) {
            for (var n = 0; n < depth; n++)
                output += "    ";
            output += "\"" + comp.name() + "\"\n";
        }
        return output;
    },
"")
```

States

Components, during their life-cycle, are in various particular states. Components can be triggered to change their state. During those state transitions, enter and leave methods are called accordingly.

```
ComponentJS.transition(null)
```

```
ComponentJS.transition(target: String, enter: String, leave: String, color: String, [source: String])
```

```
ComponentJS.transition({ target: String, enter: String, leave: String, color: String, [source: String] })
```

Clear all (if passed just a single `null` parameter) or add one state transition to target state *target*, either at the top of the transition stack or in the middle, above the source state *source*. When entering the target state, the optional component backing object method *enter* is called. When leaving the target state, the optional component backing object method *leave* is called. The *color* is a "#RRGGBB" string used for visualizing the state in the debugger view. The default state transition definitions are given as an example.

```
cs.transition(null);
cs.transition("created",      "create",  "destroy",  "#cc3333");
cs.transition("configured",   "setup",   "teardown", "#eabc43");
cs.transition("prepared",     "prepare", "cleanup",  "#f2ec00");
cs.transition("materialized", "render",  "release",  "#6699cc");
cs.transition("visible",      "show",    "hide",     "#669933");
cs.transition("enabled",      "enable",  "disable",  "#336600");
```

```
component.state(): String
```

```
component.state(state: String[, callback: Function]): String
```

```
component.state({ state: String, [callback: Function = undefined,] [sync: Boolean = false,] }): String
```

Determine the current state or request a transition to a new state of *component*. By default a state transition is performed asynchronously, but you can request a synchronous transition with `sync`. For asynchronous transitions you can await the transition finish with `callback`. The old state is returned on state transitions. On each state transition, for each transitively involved component and each target or intermediate state, a non-capturing/non-bubbling event is internally published named "`ComponentJS:state:state:enter`" or "`ComponentJS:state:state:leave`". You can subscribe to those in order to react to state transitions from outside the component, too.

```
cs("/ui").state("visible")
```

```
component.state_compare({ state: String }): Number
```

```
component.state_compare(state: String): Number
```

Compare the state of *component* with `state`. If *component* is in a lower state than `state`, a negative number is returned. If *component* is in same state than `state`, a zero is returned. If *component* is in a higher state than `state`, a positive number is returned.

```
if (cs(this).state_compare("visible") < 0) ...
```

```
component.state_auto_increase(increase: Boolean): Boolean
```

```
component.state_auto_increase(): Boolean
```

Get or set component *component* to automatically transition to same higher/increased state than its parent component.

```
cs(this).state_auto_increase(true)
```

```
component.state_auto_decrease(decrease: Boolean): Boolean
```

```
component.state_auto_decrease(): Boolean
```

Get or set component *component* to automatically transition to same lower/decreased state than its child components. Notice that this means that a child can drag down the parent component and this way implicitly also all of its other sibling child components. Hence, use with care!

```
cs(this).state_auto_decrease(true)
```

component.**guard**({ method: *String*, level: *Number* }): *Void*

component.**guard**(*method*: *String*, *level*: *Number*): *Void*

Guard component *component* from calling the state enter/leave method `method` and this way prevent it from entering/leaving the corresponding state. The `level` can be increased and decreased. Initially it should be set to a positive number to activate the guard. Then it should be set to a negative number to (potentially) deactivate the guard. A usage with an initial call of +1 and then followed by a `-1` is a boolean guard. An initial call of +N and then followed by N times a `-1` call is a Semaphore-like guard which ensures that only after the Nth `-1` call the guard is finally deactivated again. This is useful if you activate the guard in order to await N asynchronous operations. Then the guard should be deactivated once the last asynchronous operation is finished (independent which one of the N operations this is). A guard *level* of `0` resets the guard, independent what its current level is.

```
var self = this;
cs(self).guard("render", +2)
$.get(url1, function (data) {
    self.data1 = data;
    cs(self).guard("render", -1)
});
$.get(url2, function (data) {
    self.data2 = data;
    cs(self).guard("render", -1)
});
```

Spools

In ComponentJS there are at least 4 resource allocating operations which have corresponding deallocation operations: Model **observe**()/**unobserve**(), Socket **plug**()/**unplug**(), Event **subscribe**()/**unsubscribe**(), Service and **register**()/**unregister**(). For correct run-time operation it is required that each allocation operation, performed in a state enter method, is properly reversed with the corresponding deallocation operation in the state leave method. As this is extremely cumbersome (especially because you have to store the identifiers returned by the allocation operations as you need them for the deallocation operation), ComponentJS provides

- convenient spool mechanism which all of the above allocation operations support and which also can be used by the
application itself.

component.**spool**({ name: *String*, ctx: *Object*, func: *Function*, [args: *Array(Object)* = new Array()] }): *Void*

component.**spool**(*name*: *String*, *ctx*: *Object*, *func*: *Function*, [*args*: *Object*, ...]): *Void*

Remember action "`func`.apply(`ctx`, `args`)" on spool named `name`. The `name` parameter can be either just a plain spool-name "`name`" or a combination of (relative) component-path and spool-name "`path:name`". This allows one to spool on a component different from *component* (usually a relative path back to the component of the caller of the **spool**() operation).

```
cs(this).spool({
    name: "foo",
    ctx: this,
    func: function (num, str) { ... },
    args: [ 42, "foo" ]
});
```

component.**spooled**({ name: *String* }): *Number*

component.**spooled**(*name*: *String*): *Number*

Return the number of actions which are spooled under spool named `name`. Usually done before calling **unspool**() as it would throw an exception if there are no spooled actions at all.

```
if (cs(this).spooled("foo"))
    cs(this).unspool("foo")
```

component.**unspool**({ name: *String* }): *Void*

component.**unspool**(*name*: *String*): *Void*

Perform all actions previously spooled on spool `name` in reverse spooling order (those spooled last are executed first).

```
release: function () {
    cs(this).unspool("materialized")
}
```

Markers

An object can be "marked" with a set of names. ComponentJS internally does not use those markers at all, but the ComponentJS Debugger plugin at least uses markers named "service", "model", "view" and "controller" on components' backing object to render those components in different colors.

ComponentJS.**mark**(*obj*: *Object*, *name*: *String*): *Void*

component.**mark**(*name*: *String*): *Void*

Mark object *obj* with marker named *name*. An arbitrary number of markers can be added to an object. An an alternative and for convenience reasons, but only if the component classes are defined through ComponentJS' optional Class/Trait system, the traits `cs.marker.{service,model,view,controller}` can be mixed into.

```
app.ui.panel.view = cs.clazz({
    create: function () {
        cs(this).mark("view");
    }
    ...
});
```

```
app.ui.panel.view = cs.clazz({
    mixin: [ cs.marker.view ]
    ...
});
```

ComponentJS.**marked**(*obj*: *Object*, *name*: *String*): *Boolean*

component.**marked**(*name*: *String*): *Boolean*

Checks whether object *obj* is marked with marker named *name*. This is usually interesting for ComponentJS plugin developers only.

```
if (cs("/").marked("controller")) {
    ...
}
```

Properties

Every component can have an arbitrary number of key/value based properties attached to it. The keys have to be of type *String*, the values can be of any type. A property is set on a target component but is resolved on both the target component and all parent components (up to and including the root component). This way properties feel like inherited and overrideable values which can be used for both storing component-local information and to communicate information to foreign components.

component.**property**({ name: *String*, [value: *Object* = undefined,] [def: *Object* = undefined,] [scope: *String* = undefined,] [bubbling: *Boolean* = true,] [targeting: *Boolean* = true,] [returnowner: *Boolean* = false] }): *Object*

component.**property**(*name*: *String*, *value*: *Object*): *Object*

component.**property**(*name*: *String*): *Object*

Get or set property with name `name` and value `value` on component *component*. If `bubbling` is set to `false` a property get operation does not resolve on any parent components ("it does not bubble up to the root"). If `targeting` is set to `false` a property get operation does not resolve on the target component *component* (resolving starts on parent component). If `returnowner` is set to `true` instead of the property value, the owning component is returned. Finally, properties can be scoped with a child component name or even a descendant component name path: on each attempt to resolve the property, first the scoped variants are tried. This means, if a property was set with `name` `"quux@bar"` (or with `name` `"quux"` and an explicitly `scope` set to `"bar"`) on component `/foo`, if you resolve the property with `cs("/foo/bar", "quux")` you get the value, but if you resolve the property with `cs("/foo/baz", "quux")` you do not get the value. This allows you to set the same property with different values for different child components. Additionally the scope can be a partial component path, too. If a property was set with name `"quux@bar/baz"` on component `/foo`, if you resolve the property with `cs("/foo/bar/baz", "quux")` you get the value, but if you resolve the property with `cs("/foo/bar/baz2", "quux")` you do not get the value. This allows you for instance to skip so-called intermediate namespace-only components. Setting `value` to `"null"` removes the property. If no property `name` is found at all, `def` (by default the value `undefined`) is returned.

```
cs(this).property("foo")
```

Sockets

Sockets are a special form of component Properties with callback functions as the values. They are intended to link Views of child/descendant components into the View of a parent/ancestor component. In contrast to regular Properties, Sockets are never resolved directly on the target component. Instead they always start to resolve on the parent component because the sockets on the target component are intended for its child/ancestor components and not for the target component itself. So, please remember to never plug a socket directly onto the target component!

component.**socket**({ [name: *String* = "default",] [scope: *Object* = null,] ctx: *Object*, plug: *Function*, unplug: *Function* [, spool: *String*] }): *Number*

component.**socket**(*ctx*: *Object*, *plug*: *Function*, *unplug*: *Function*): *Number*

Create a socket on *component*, named `name` and optionally scoped for the child component named `scope`, where **plug**() and **unplug**() calls on child/ancestor components execute the supplied `plug`/`unplug` functions with `ctx` supplied as `this`, the `object` parameter of **plug**()/**unplug**() as first argument and *component* as the second argument. The **socket**() method returns an id which uniquely identifies the socket. Instead of having to manually release the socket later via **unsocket**() you can use the spool mechanism and spool the corresponding **unsocket**() operation via option `spool`.

```
var ui = $(...);
cs(this).socket({
    ctx:    ui,
```

```
    plug:   function (el) { $(this).append(el); },
    unplug: function (el) { $(el).remove(); }
})
```

component.**unsocket**({ id: *Number* }): *Void*

component.**unsocket**(*id*: *Number*): *Void*

Destroy the socket identified by id, previously created by a call to **socket**(). This is usually done implicitly through the spooling mechanism.

```
cs(this).unsocket(id)
```

component.**link**({ [name: *String* = "default",] [scope: *Object* = null,] target: *Object*, socket: *String* [, spool: *String*] })

component.**link**(*target*: *Object*, *socket*: *String*)

Create a socket on *component*, named name and optionally scoped for the child component named scope, and pass-through the **plug**()/**unplug**() calls to the target component target and its socket named socket. Usually used by Controller components to link their default socket (for the View below itself) to a particular socket of a parent component (because a View should be reusable and hence is not allowed to know the particular socket intended for it). The **link**() method returns an id which uniquely identifies the linked socket. Instead of having to manually release the socket later via **unlink**() you can use the spool mechanism and spool the corresponding **unlink**() operation via option spool.

```
cs(this).link({ name: "default", target: this, socket: "menu1" })
```

component.**unlink**({ id: *Number* }): *Void*

component.**unlink**(*id*: *Number*): *Void*

Destroy the linked socket identified by id, previously created by a call to **link**(). This is usually done implicitly through the spooling mechanism.

```
cs(this).unlink(id)
```

component.**plug**({ [name: *String* = "default",] object: *Object*, [spool: *String*,] [targeting: *Boolean*] }): *Number*

component.**plug**(*object*: *Object*): *Number*

Plugs object into the socket named name provided by any parent/ancestor component of *component*. Optionally spool the corresponding **unplug**() operation on spool spool attached to *component*. Optionally (in case of targeting set to true) start the operation on *component* instead of its parent component. Returns an identifier for use with the corresponding **unplug**() operation.

```
cs(this).plug({ object: ui, spool: "materialized" })
```

component.**unplug**({ id: *Number*[, targeting: *Boolean*] }): *Void*

component.**unplug**(*id*: *Number*): *Void*

Unplugs the object previously plugged under id from the socket providing parent/ancestor component of *component*. Optionally (in case of targeting set to true) start the operation on *component* instead of its parent component. This is usually performed indirectly through the Spool mechanism.

```
cs(this).unplug(id)
```

Models

When using Model/View/Controller roles for components, the Model component needs a so-called Presentation Model: an abstraction of presentation onto which both View and Controller components attach via Observer pattern. The Controller component for provisioning business information into the Model and triggering business services upon Model changes. The View component for displaying the Model information and storing events into it.

component.**model**(*spec*: *Object*): *Object*

Define a model through the specification in *spec*. Each key is the name of a model element and the value has to be an *Object* with the following possible fields: `value` (*Object*) for the default value, `valid` (*String*/*RegExp*) for validating the values (based on the underlying validation language of the **validate**() method), `autoreset` (*Boolean*) for indicating that on each value write, the value should be automatically reset to the initial `value`, and `store` (*Boolean*) for indicating that the value should be persistently stored in the browser's `localStorage`. Multiple calls to the **model**() method on the same component incrementally add model elements.

```
cs(this).model({
    "param:realms":         { value: [],      valid: "[string*]" },
    "data:realm":           { value: "",      valid: "string", store: true },
    "data:username":        { value: "",      valid: "string", store: true },
    "data:password":        { value: "",      valid: "string" },
    "state:username":       { value: "empty", valid: "string" },
    "state:username-hint":  { value: "",      valid: "string" },
    "state:password":       { value: "empty", valid: "string" },
    "state:password-hint":  { value: "",      valid: "string" },
    "state:hashcode-col":   { value: 0,       valid: "number" },
    "state:hashcode-txt":   { value: "",      valid: "string" },
    "state:button-enabled": { value: false,   valid: "boolean" },
    "event:button-clicked": { value: false,   valid: "boolean", autoreset: true }
})
```

component.**value**({ name: *String*, [op: *String*,] [value: *Object*,] [force: *Boolean*,] [injected: *Boolean*] })

component.**value**(*name*: *String*, [*value*: *Object*,] [*force*: *Boolean*])

Get the value of *component*'s model element named `name` or set the value of *component*'s model element named `name` to `value`. As each value change causes observers to be triggered, by default changing a value to the same value does not trigger anything. But if `force` is `true` even setting a model element to its current value triggers observers. Setting the option `injected` to `true` should be done by plugins only and prevents model value observers from rejecting the (already injected) value.

```
var val = cs(this).value("foo")
cs(this).value("foo", "bar")
```

If you store arbitrary sub-structured values, you can make `name` a path full specification based on the language supported by the **select**() method:

```
var val = cs(this).value("foo.bar[1].baz['the-quux']")
cs(this).value("foo.bar[1].baz['the-quux']", "bar")
```

In addition to the basic get/set operations on scalar values, you can also use array and hash operations on collections by using the `op` option. Supported `op` values are `"get"`, `"set"`, `["splice",offset,remove]`, `"delete"`, `"push"`, `"pop"`, `"unshift"` and `"shift"`. The last four array operations are internally translated to the corresponding `splice` operation. The arguments to the `splice` operation are the same as for JavaScript's `Array.prototype.splice`: "offset" is the 0-based offset into the array to operate at and "remove" is the number of elements to remove at "offset" (before the `value` is added). The operations `get`/`set`/`delete` operate on collection elements while the operations `splice`/`push`/`pop`/`unshift`/`shift` operate on collections, hence you have to provide a path in `name` which is suitable for them. The operations `get`/`set`/`delete` can operate on both array and hash elements while `splice`/`push`/`pop`/`unshift`/`shift` can operate on array objects only.

To illustrate the functionality see the following comparisons between the standard JavaScript variable access code and the ComponentJS model value access code.

First, working with scalars:

```
//  val = foo.bar
val = cs(this).value("foo.bar")
val = cs(this).value({ name: "foo.bar", op: "get" })

//  foo.bar = "quux"
cs(this).value("foo.bar", "quux")
cs(this).value({ name: "foo.bar", op: "set", value: "quux" })
```

Second, working with Arrays:

```
//  foo.bar = []
cs(this).value("foo.bar", [])
cs(this).value({ name: "foo.bar", value: [] })

//  len = foo.bar.length
len = cs(this).value("foo.bar").length

//  val = foo.bar[42]
val = cs(this).value("foo.bar[42]")
val = cs(this).value({ name: "foo.bar[42]", op: "get" })

//  foo.bar[42] = "quux"
cs(this).value("foo.bar[42]", "quux")
cs(this).value({ name: "foo.bar[42]", op: "set", value: "quux" })

//  foo.bar.splice(1, 0, "quux")
cs(this).value({ name: "foo.bar", op: [ "splice", 1, 0 ], value: "quux" })

//  foo.bar.push("foo")
cs(this).value({ name: "foo.bar", op: "push", value: "foo" })

//  val = foo.bar.pop()
val = cs(this).value({ name: "foo.bar", op: "pop" })

//  foo.bar.unshift("bar")
cs(this).value({ name: "foo.bar", op: "unshift", value: "bar" })

//  val = foo.bar.shift()
val = cs(this).value({ name: "foo.bar", op: "shift" })
```

Third, working with hashes:

```
//  foo.bar = {}
cs(this).value("foo.bar", {})
cs(this).value({ name: "foo.bar", value: {} })

//  keys = Object.keys(foo.bar)
keys = Object.keys(cs(this).value("foo.bar"))

//  val = foo.bar.baz
//  val = foo.bar["baz"]
val = cs(this).value("foo.bar.baz")
val = cs(this).value("foo.bar['baz']")
val = cs(this).value({ name: "foo.bar.baz",    op: "get" })
val = cs(this).value({ name: "foo.bar['baz']", op: "get" })

//  foo.bar.baz = "quux"
//  foo.bar["baz"] = "quux"
cs(this).value("foo.bar.baz",    "quux")
cs(this).value("foo.bar['baz']", "quux")
cs(this).value({ name: "foo.bar.baz",    op: "set", value: "quux" })
```

```
cs(this).value({ name: "foo.bar['baz']", op: "set", value: "quux" })

//  delete foo.bar.baz
//  delete foo.bar["baz"]
cs(this).value({ name: "foo.bar.baz",    op: "delete" })
cs(this).value({ name: "foo.bar['baz']", op: "delete" })
```

component.**touch**({ name: *String*, })

component.**touch**(*name*: *String*)

Touches the value of *component*'s model element named `name`, without changing the value but with triggering all its "get" observers (its "changed" observers are not triggered). This can be useful for firing "set" observers manually.

```
cs(this).touch("foo")
```

component.**observe**({ name: *String*, func: *Function*, [touch: *Boolean* = false,] [op: *String* = "set",] [spool: *String* = null] }): *Number*

component.**observe**(name: *String*, func: *Function*): *Number*

Observe the value of *component*'s model element named `name` for `op` operations (by default "set" operations). For "get" operations, the callback function `func` has to be of signature `func`(*ev*: *Event*, *value*: *Object*): *Void*. For "set" and "changed" operations, the callback function `func` has to be of signature `func`(*ev*: *Event*, *value-new*: *Object*, *value-old*: *Object*, *op*: *Object*, *path*: *String*): *Void*. Both types of callbacks can override the value by using *ev*.**result**(*value*). The **observe**() method returns an id which uniquely identifies the observation. Instead of having to manually release the observation later via **unobserve**() you can use the spool mechanism and spool the corresponding **unobserve**() operation via `spool`.

```
id = cs(this).observe("state:username", function (ev, username) {
    ...
})
```

component.**unobserve**({ id: *Number* }): *Void*

component.**unobserve**(*id*: *Number*): *Void*

Release the observation identified by `id`, previously acquired by a call to **observe**(). This is usually done implicitly through the spooling mechanism.

```
cs(this).unobserve(id)
```

Events

The Event mechanism is a central one in ComponentJS. Both Models and Services are internally based on the Events mechanism. An Event is an object published towards a target component. It is delivered in 4 phases:

- In phase 1 (the "capturing" phase) the Event is delivered to all components on the path from the root component (inclusive) towards the target component (exclusive).
- In phase 2 (the "targeting" phase) the Event is delivered to the target component.
- In phase 3 (the "spreading" phase) the Event is delivered to all descendant components of the target component in a depth-first traversal order.
- In phase 4 (the "bubbling" phase) the Event is delivered (again) to all components on the path from the target component (exclusive) to the root component (inclusive).

Event objects are implicitly created by the **publish**() operation and they provide various getter/setter methods:

- **target**() (*Component*): target component the event is send to

- **propagation**() (*Boolean*): whether event propagation should continue
- **processing**() (*Boolean*): whether final default event processing should be performed
- **dispatched**() (*Boolean*): whether event was dispatched at least once to a subscriber
- **decline**() (*Boolean*): whether event was declined by subscriber
- **state**() (*Boolean*): state of dispatching: `capturing`, `targeting`, `spreading` or `bubbling`
- **result**() (*Object*): optional result value event subscribers can provide
- **async**() (*Boolean*): whether event is dispatched asynchronously

component.**subscribe**({ name: *String*, [spec: *Object* = {}], [ctx: *Object* = *component*,] func: *Function*, [args: *Object[]* = []], [capturing: *Boolean* = false], [spreading: *Boolean* = false], [bubbling: *Boolean* = true], [noevent: *Boolean* = false], [exclusive: *Boolean* = false], [spool: *String* = null] }): *Number*

component.**subscribe**(*name*: *String*, *func*: *Function*, [*args*: *Object*, ...]): *Number*

Subscribe to event name (optionally sub-specified via spec) on component *component* and execute callback func as func(*ev*: *Event* , *args*: *Object*, ..., *sargs*: *Object*, ...) once the event is dispatched to *component* after it was published. By default an event is dispatched in the (mandatory) targeting and (optional) bubbling phases.

- Option ctx allows you to give "this" a particular value for the callback func. Option args allows you to pass additional parameters to func (before those passed by **publish**().
- Option noevent does not pass the *ev*: *Event* parameter to func.
- Setting option capturing to "true" indicates that the event should be also dispatched in the capturing phase.
- Setting option spreading to "true" indicates that the event should be also dispatched in the spreading phase.
- Setting option bubbling to "false" indicates that the event should not be dispatched in the bubbling phase.
- Option exclusive can be set to "true" for an exclusive subscription, i.e., a subscription which prevents any subsequent subscriptions.

The **subscribe**() method returns an id which uniquely identifies the subscription. Instead of having to manually release the subscription later via **unsubscribe**() you can use the spool mechanism and spool the corresponding **unsubscribe**() operation via option spool.

```
cs(self).subscribe({
    name: "data-loaded",
    spool: "prepared",
    func: function (ev, data, info) {
        ...
    }
})
```

component.**unsubscribe**({ id: *Number* }): *Void*

component.**unsubscribe**(*id*: *Number*): *Void*

Release the subscription identified by id, previously acquired by a call to **subscribe**(). This is usually done implicitly through the spooling mechanism.

```
cs(this).unsubscribe(id)
```

component.**publish**({ name: *String*, [spec: *Object* = {},] [async: *Boolean* = false,] [capturing: *Boolean* = true,] [spreading: *Boolean* = false,] [bubbling: *Boolean* = true,] [completed: *Function*,] [resultinit: *Object* = undefined,] [resultstep: *Function*,] [directresult: *Boolean* = false,] [noresult: *Boolean* = false,] [firstonly: *Boolean* = false,] [silent: *Boolean* = false,] [args: *Object[]* = []] }): *Object*

component.**publish**(*name*: *String*, *args*...: *Object*): *Object*

Publishes an *Event* to component *component* named name and with optional arguments args. By default the event is intended to be dispatched in the (mandatory) targeting and (optional) capturing and bubbling phases. The following options allow you to further control the event publishing process:

- Option `spec` allows you to sub-specify/parametrize the event with arbitrary key/value pairs in case the `name` is too generic.
- Option `async` allows the event processing to occur asynchronously.
- Setting option `capturing` to "`false`" indicates that the event should not be intended to be dispatched in the capturing phase.
- Setting option `spreading` to "`true`" indicates that the event should also be intended to be dispatched in the spreading phase.
- Setting option `bubbling` to "`false`" indicates that the event should not be intended to be dispatched in the bubbling phase.
- Option `completed` executes the specified callback function once the event was dispatched to subscribers in all possible phases. This allows you to react at the end of `async`=`true` events.
- Option `resultinit` and `resultstep`

```
cs(this).publish("data-loaded", data, info)
```

Services

Services are loosely coupled method calls across components. The functionality provider does **`register`**() the service and the functionality consumer does **`call`**() the service.

component.**`register`**({ `name`: *String*, [`ctx`: *Object* = *component*,] `func`: *Function*, [`args`: *Object[]* = [],] [`spool`: *String*,] [`capturing`: *Boolean* = `false`,] [`spreading`: *Boolean* = `false`,] [`bubbling`: *Boolean* = `true`] }): *Number*

component.**`register`**(*name*: *String*, *func*: *Function*): *Number*

Register a service `name` on *component* with the implementing callback function `func`. The function returns an identifier for **`unregister`**(). The following options can be used to control the later service calls:

- Option `ctx` can be used to set the `this` pointer for `func`.
- Option `args` can be used to pass additional parameters to `func` (before the `args` of **`call`**()!).
- Option `spool` can be used to spool the corresponding **`unregister`**() call.
- Option `capturing` can be set to `true` to provide the service also in the "capturing" phase.
- Option `spreading` can be set to `true` to provide the service also in the "spreading" phase.
- Option `bubbling` can be set to `false` to not provide the service in the "bubbling" phase.

```
var id = cs(this).register({
    name: "load-entity",
    args: [ em ],
    func: function (em, clazz, id) {
        return em.findById(clazz, id);
    }
})
```

component.**`unregister`**({ `id`: *Number* }): *Void*

component.**`unregister`**(*id*: *Number*): *Void*

Release the registration identified by `id`, previously acquired by a call to **`register`**(). This is usually done implicitly through the spooling mechanism.

```
cs(this).unregister(id)
```

component.**`callable`**({ `name`: *String*[, `value`: *Boolean*] }): *Boolean*

component.**`callable`**(*name*: *String*[, *value*: *Boolean*]): *Boolean*

Checks whether a registered service is callable/enabled or enable/disable a registered service. On every change to the "callable" status of a service, an internal event named

"ComponentJS:service:name:callable" is published with two arguments: the new and old boolean value.

```
cs(this).subscribe("ComponentJS:service:load-person:callable", function (old, new) {
    if (new) {
        /* react on now callable service  */
    }
})
cs(this).callable("load-person", false)
cs(this).callable("load-person", true)
```

component.**call**({ name: *String*, [args: *Object[]* = [],] capturing: *Boolean* = false,] spreading: *Boolean* = false,] bubbling: *Boolean* = true] }): *Object*

component.**call**(*name*: *String* [, *args*...: *Object*]): *Object*

Call service named name on component *component*, optionally passing it the arguments args (after the optional args of **register**()!). The following options can be used to control the service call:

- Option capturing can be set to true to deliver the underlying service event also in the "capturing" phase.
- Option spreading can be set to true to deliver the underlying service event also in the "spreading" phase.
- Option bubbling can be set to false to not deliver the underlying service event in the "bubbling" phase.

```
var person = cs("/sv").call("load-entity", "Person", 42)
```