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 by called after 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()
```

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.debug_instrumented()) {
    cs.debug(9)
```

```
        cs.debug_window({ ... })
}
```

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

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)
```

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 shadow 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 throught 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.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 the string returned by JavaScript `typeof` operator, or the string `"array(...)"` for arrays or a valid regular expression `/.../` for validating a *String* against it.

```
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, validate: Object }): Function
```

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

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

```
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] })
```

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 instanciated and which can further initialize the dynamic fields of the class. On each instanciation, all fields which are specified with `dynamics` are cloned and instanciated 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, baz) {
        this._bar = bar;
        this._baz = baz;
    },
    dynamics: {
        _bar: "bar",
        _baz: 42
    },
    protos: {
        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] })
```

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 instanciated and which can further initialize the dynamic fields of the Class. On each instanciation, all fields which are specified with `dynamics` are cloned and instanciated 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 instanciation) 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. 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 instanciated class constructor (*Function*) or an already instanciated object (*Object*).

```
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()
```

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 shadow *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 (//) 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 implicity 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 shadow object *object*. 5. Lookup *Component* object via the *component* itself (no-operation).

```
cs("/foo/bar")           /* absolute */
cs(comp, "model/view")   /* relative to component */
cs(this, "model/view")   /* relative to component via shadow 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.**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 shadow object method *enter* is called. When leaving the target state, the optional component shadow 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("prepared",     "prepare", "cleanup", "#eabc43");
cs.transition("materialized", "render",  "release", "#6699cc");
cs.transition("visible",      "show",    "hide",    "#669933");
```

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 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).

```
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 **register**/**unregister** and Hook **latch**/**unlatch**. 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 a 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* = this,] func: *Function* [args: *Array(Object)* = new Array()] }): *Void*

component.**spool**(*name*: *String*, *func*: *Function*): *Void*

Remember action "ctx.func(args)" on spool named name.

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

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

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

Check whether any actions 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")
}
```

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",] [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: on each attempt to resolve the property, first the scoped variant is tried. This means, if a property was set with name `"bar@prop"` on component `/foo`, if you resolve the property with `cs("/foo/bar/baz", "prop")` you the value, but if you resolve the property with `cs("/foo/quux", "prop")` you do not get the value. This allows you to set the same property with different values for different child components.

```
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.

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

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

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`.

```
var ui = $(...);
cs(this).socket({
    ctx:    ui,
    plug:   function (el) { $(this).append(el); },
    unplug: function (el) { $(el).remove(); }
})
```

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

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

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

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

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*. Returns an identifier for use with the corresponding **`unplug`**() operation.

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

component.**unplug**({ id: *String* }): *Void*

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

Unplugs the object previously plugged under `id`. 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, `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: "array(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*, [value: *Object*,] [force: *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.

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

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

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

Observe the value of *component*'s model element named `name` for `operation` operations (by default set/change operations). For "get" operations, the callback function `func` has to be of signature `func`(*ev*: *Event*, *value*: *Object*): *Void*. For "set" operations, the callback function `func` has to be of signature `func`(*ev*: *Event*, *value-new*: *Object*, *value-old*: *Object*): *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: *String* }): *Void*

component.**unobserve**(*id*: *String*): *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, Services and Hooks are all 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 and 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`], [origin: *Boolean* = `false`], [spool: *String* = `null`] })

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

Subscribe to event `name` on component *component* and execute callback `func` as `func`(*ev*: *Event*, *args*: *Object[]*) once the event is dispatched to *component* after it was published. Option `ctx` allows FIXME. By default an event is dispatched in the targeting and bubbling phases. Setting option `capture` to `true` indicates that the event should be dispatched in the capturing phase.

FIXME

Services

FIXME

FIXME

FIXME

FIXME

Hooks

FIXME

FIXME

FIXME

FIXME