classjs Documentation

Release 1.0

Angelo Dini

December 30, 2015

Contents

Class.js is a JavaScript library for building class based object-oriented programs using JavaScript's prototypal inheritance. class.js strives to mimic classical class inheritance provided by other languages such as Python, Java or C. Its syntax is heavily inspired by MooTools class implementation.

The latest stable version is 1.0 - https://github.com/finalangel/classjs

CHAPTER 1

Introduction

1.1 Why class.js

I started learning JavaScript roughly in 2002 without the use of any framework. Back than Prototype was the big player and I did not understand its concept. I also had issues with Prototype's none-modular approach splitting animation (script-aculo-us) from the core library.

After I finishing my study I came across MooTools and quickly adapted my code to a more modular and re-usable style. At a certain point of my career the requirements for use of jQuery grew. jQuery's popularity grow and we had to adapt.

I am not a big fan of jQuery, especially jQuery's UI approach. It is missing any modular structure and is almost impossible to extend. The fame of jQuery makes it hard to find a plugin for your case. If you end up finding one you end up changing the code to work for your specific case. This is wrong. I should be able to take the plugin and extend it, leaving the code untouched. This enables me to easily update and maintain the plugin without re-implementing your own changes.

There is also a second reason. A lot of developers do not understand the principles of prototypal inheritance. All in all JavaScript is one of the few popular languages which adapts this concept. This is confusing and leads to misinterpretation.

This is why I created class.js.

1.2 How to implement

Download the production version of class.js (minified) and implement it in your html code.

For **HTML5** use:

```
<script src="/path/class.js"></script>
```

For **HTML** and **XHTML** versions use:

```
<script type="text/javascript" src="/path/class.js"></script>
```

You will be able to call `Class.version` within your preferred browsers console if class.js is implemented or initialized correctly. Feel free to explore the source.

1.3 How to update

1.3.1 Update from Beta to 1.0

There are no special adjustments required for this version. Beta releases will not be covered by any adaptation. Therefore consult the Changelog for more details.

1.3.2 Update from classy.js to class.js

There is not **yet** an official adapter from classy.js (Divio internal version) to class.js. You need to change your code to be compatible with class.js. Please follow this guide for a smooth transition:

Comparison from old to new:

```
var Animal = Class.$extend(obj); // old
var Animal = new Class(obj); // new

var Tiger = Animal.$extend(obj); // old
var Tiger = Animal.extend(obj); // new

Class.$classyVersion; // old
Class.version; // new

Class.$withData(obj); // old
Class.setOptions(obj); // new

Class.$noConflict(); // old
Class.noConflict(); // new

this.$super(); // old
this.parent(); // new

initialize: function () {} // identical in both versions

implement: [array] // identical in both version
```

Additionally class.js provides the following features:

```
Class.getOptions();
Class.implement([array]);
```

CHAPTER 2

Class API

2.1 methods

The following methods can be attached to a new class.

2.1.1 new Class(object);

`(returns object)` - This is the class constructor and passes the provided object to the instances prototype.

Create a new Class:

```
var Animal = new Class({
    initialize: function (name, age) {
        this.name = name;
        this.age = age;
    },
    eat: function () {
        alert(this.name + ' is eating now.');
    }
});
```

For a better explanation to `initialize` consult the class.js properties section.

2.1.2 Class.extend(object)

`(returns object)` - The object or class passed through extend will be **merged** with the assigned class following the previous example.

Extend the class:

```
Animal.extend({
    die: function () {
        alert(this.name + ' died at age ' + this.age);
    }
});

var cat = new Animal('Sora', 4);
    cat.die(); // alerts "Sora died at age 4"
```

You can create a new instance and extend it, without modififying the parent class:

```
var MyAnimal = new Class(Animal);
    MyAnimal.extend({
        clone: function () {
           alert(this.name + ' can clone itself now.');
        }
    });

// can't clone
var sora = new Animal('Sora', 4);
var pitschy = new MyAnimal('Pitschy', 4);
    // sora.clone(); fails
    pitschy.clone(); // can clone
```

2.1.3 Class.implement(array)

`(returns object)` - Each object or class within the array will be **added** to the assigned class following the previous example.

Implementing new methods:

```
// preparing a class
var Mammal = new Class({
    swim: function () {
        alert(this.name + ' can swim now.');
    },
    dive: function () {
        alert(this.name + ' can dive now.');
    }
});

// preparing a normal object
var Bird = {
    fly: function () {
        alert(this.name + ' can fly now.');
    }
};

Animal.implement([Mammal, Bird]);

var cat = new Animal('Sora', 4);
    cat.swim(); // alerts "Sora can swim now."
    cat.dive(); // alerts "Sora can dive now."
    cat.fly(); // alerts "Sora can fly now."
```

Note: implement simply **copies** object methods over into the new class and breaks the prototypal chain. It does not create a parent link nor does it copy `initialize` into the new class. Implemented methods cannot be overwritten to prevent accidental conflicts. Use extend to modify available class methods.

2.1.4 Class.getOptions()

`(returns object)` - Gathers assigned options and returns them.

Getting options from a class:

```
var Animal = new Class({
    options: {
            'name': '',
            'age': null
    },
    initialize: function (name, age) {
            this.name = this.options.name || name;
            this.age = this.options.age || age;
    }
});

Animal.getOptions(); // returns { 'name': '', 'age': null }
```

It is not possible to get the options once an instance has been created. You can access the instance objects through their options name `cat.options.name`.

2.1.5 Class.setOptions(object)

`(returns object)` - Sets and merges a given options object to the classes internal options object.

Setting options for a class:

```
var Animal = new Class({
    options: {
            'name': '',
            'age': null
    },
    initialize: function (name, age) {
            this.name = this.options.name || name;
            this.age = this.options.age || age;
    }
});

Animal.setOptions({
    'name': undefined,
    'dead': false
});

Animal.getOptions(); // returns { 'name': undefined, 'age': null, 'dead': false }
```

It is not possible to change the options once an instance has been created. You can access the instance objects through their options name `cat.options.name`.

2.1.6 Class.noConflict()

`(returns Class)` - Removes the class object from the window object and restores what was there before class.js was loaded.

Using class.js with multiple libraries:

```
// loading MooTools
var Classy = Class.noConflict();

var Animal = new Classy({
    initialize: function (name, age) {
            this.name = name;
            this.age = age;
```

```
    },
    eat: function () {
        alert(this.name + ' is eating now.');
    }
});

var cat = new Animal('Sora', 4);
    cat.eat(); // alerts "Sora is eating now."
```

2.1.7 Class.version

`(returns string)` - Returns the current running class.js version as a string.

alert(Class.version); // alerts current class.js version

2.2 properties

The following properties can be used within the Classes passing object.

2.2.1 initialize: function

`(returns constructor)` - The initialize function serves as constructor on instance creation.

Create a new Class:

```
var Animal = new Class({
    initialize: function(name, age) {
        this.name = name;
        this.age = age;
    }
});

    // the passing options will be passed through the constructor initialize
new Animal('Satan', 21);
```

2.2.2 implement: array

`(returns object)` - Each object or class within the array will be **added** to the assigned class.

Implementing new methods:

```
// preparing a class
var Mammal = new Class({
    swim: function () {
        alert(this.name + ' can swim now.');
    },
    dive: function () {
        alert(this.name + ' can dive now.');
    }
});

// preparing a normal object
var Bird = {
```

```
    fly: function () {
        alert(this.name + ' can fly now.');
    }
};

var Animal = new Class({
    implement: [Mammal, Bird],
    initialize: function (name, age) {
        this.name = name;
        this.age = age;
    },
    eat: function () {
        alert(this.name + ' is eating now.');
    }
});

var cat = new Animal('Sora', 4);
    cat.swim(); // alerts "Sora can swim now."
    cat.dive(); // alerts "Sora can dive now."
    cat.fly(); // alerts "Sora can fly now."
```

This property works identical to the Class.implement method.

2.2.3 options: object

`(returns object)` - The options object represents a JSON structure.

Attach options to a Class:

```
var Animal = new Class({
    options: {
        'name': '',
        'age': null,
        'dead': false
    },
    initialize: function (options) {
        // merging objects using jQuery
        this.options = options;

        // you might want to merge the two ojects using jQuery
        // this.options = jQuery.extend(true, {}, this.options, options)
    }
});

var cat = new Animal({
    'name': 'Sora',
    'age': 4
});

    // call options direct
    alert(cat.options.name); // alerts 'Sora'
```

2.2.4 this.parent()

`(returns function)` - When called from within a class.js function this invokes the parent function of the same name. This is useful when extending an existing class.js plugin.

Example usage of parent:

```
var Animal = new Class({
    initialize: function (name, age) {
            this.name = name;
            this.age = age;
    },
    getName: function () {
            alert(this.name + ' is eating now.');
    }
});

Animal.extend({
    getName: function () {
            // calls previous getName first
            this.parent();
            alert(this.name + ' is ' +  this.age + ' years old.');
    }
});

var cat = new Animal('Sora', 4);
    cat.getName(); // alerts both calls, name and age
```

CHAPTER 3

Contributing

3.1 Contribution guide

All open-source projects live through their community, so does class.js. You are very welcome to contribute bug fixes and feature proposals. Please note the following guidelines:

- class.js is hosted on github: http://github.com/FinalAngel/classjs
- Use github's issues tracker to submit bug fixes or feature proposals
- Whenever possible include a jsfiddle to an issue.
- It is good practice to fork a project and make your changes through your own fork.
- Always send a pull request with a reference to the github ticket.
- Include enough information within your commit messages and pull request to make life easier for us.

Don't feel offended if a pull or feature request has been rejected. We always provide reasonable comments or information's about how your code could be improved.

Start by fixing or extending the documentation ;)

3.2 Plugins

Creating and extending plugins is the entire purpose of class.js. To encourage users to write and extend plugins, we need to establish some common guidelines.

3.2.1 Creating a plugin

First of all, create a JavaScript file and include the following information within the top of your script:

- the authors name
- the plugins version
- the plugins license
- the plugins dependencies

For example:

```
/**
 * @author      Angelo Dini
 * @version     1.0
 * @copyright   Distributed under the BSD License.
 * @requires    class.js, jQuery
 */
```

You can include the information at any style of your choice whatsoever. Just insure that you include the required information within the development and minified versions alike.

Creating a namespace is important to keep the amount of global variables as low as possible.

I prefere to use Cl as the shorthand for **Cl** ass:

```
var Cl = window.Cl || {};
```

You are welcome to use this namespace as well. You might want to crossreference the core plugins for already reservated names like Cl.Lightbox or Cl.Carousel.

Creating a closure allows you to create an environment where you don't have to worry about other plugins namespaces. class.js uses this system by itself:

```
(function(){
    // your code here
})();

// you might want to pass some custom variables
(function($){
    // your code here
})(jQuery);
```

When choosing names be descriptive and use CamelCase for naming conventions:

```
Cl.MyPlugin = new Class({
    initialize: function () {
        return 'My first plugin';
    }
});
```

If we combine those techniques your plugin would look like this:

```
/**
 * @author      Angelo Dini, Distributed under the BSD License.
 * @version     1.0
 * @requires    class.js, jQuery
 */
(function(){
    Cl.MyPlugin = new Class({
        initialize: function () {
            return 'My first plugin';
        }
    });
})();
```

3.2.2 Submitting a plugin

If you successfully created a plugin worth spreading let us know https://twitter.com/finalangel and we will include it within the plugins list. We just require the a website or download link.

3.2.3 Core plugins

Besides the above described plugins we maintain several core plugins like `Cl.Lightbox` or `Cl.Carousel`. Those core plugins differ in various ways:

A core plugin requires tests and documentation preferable within the repository itself, with the option of external documentation. This allows for an easy development process and insures a high level of quality. Those plugins receive a **reserved namespace** like `Cl.AutoComplete` and will be listed separately.

It is also a **must** to provide an example page in which the functionality of the plugin can be demonstrated. Either through using github or a dedicated microsite.

You are encouraged to use classjs as a name prefix when using a subversion system. For example `classjs-lightbox` or `classjs-autocomplete`.

Your plugin also needs to be easily extendable while providing meaningful options.

You can find a full list of the core plugins on the Demo page.