THINKFUL PRESENTS

Javascript Best Practices, Part 1

Make it Understandable

Choose easy to understand and short names for variables and functions.

Bad variable names:

```
x1 fe2 xbqne
```

Also bad variable names:

```
incrementerForMainLoopWhichSpansFromTenToTwenty
}
createNewMemberIfAgeOverTwentyOneAndMoonIsFull
```

Avoid describing a value with your variable or function name.

Might not make sense in some countries:

```
isOverEighteen()
```

Works everywhere:

```
isLegalAge()
```

Your code is a story - make your storyline easy to follow!

Avoid Globals

Global variables are a terribly bad idea.

Reason: You run the danger of your code being overwritten by any other JavaScript added to the page after yours.

Workaround: use closures and the module pattern

Problem: all variables are global and can be accessed; access is not contained, anything in the page can overwrite what you do.

```
var current = null;
var labels = {
   'home':'home',
   'articles':'articles',
   'contact':'contact'
};
function init(){
};
function show(){
   current = 1;
};
function hide(){
```

```
    show();
};
```

Object Literal: Everything is contained but can be accessed via the object name.

Problem: Repetition of module name leads to huge code and is annoying.

```
demo = {
    current:null,
    labels:{
        'home':'home',
        'articles':'articles',
        'contact':'contact'
    },
    init:function(){
    },
    show:function(){
        demo.current = 1;
    },
    hide:function(){
        demo.show();
    }
}
```

Module Pattern: You need to specify what is global and what isnt - switching syntax in between.

Problem: Repetition of module name, different syntax for inner functions.

```
module = function(){
    var labels = {
        'home':'home',
        'articles':'articles',
        'contact':'contact'
    };
    return {
        current:null,
        init:function(){
        },
        show:function(){
            module.current = 1;
        },
        hide:function(){
            module.show();
        }
    }
}();
```

Revealing Module Pattern: Keep consistent syntax and mix and match what to make global.

```
module = function(){
    var current = null;
    var labels = {
        'home':'home',
        'articles':'articles',
        'contact':'contact'
    };
    var init = function(){
    };
    var show = function(){
        current = 1;
    };
    var hide = function(){
        show();
    }
    return{init:init, show:show, current:current}
}();
module.init();
```

Stick to a Strict Coding Style

Browsers are very forgiving JavaScript parsers. However, lax coding style will hurt you when you shift to another environment or hand over to another developer. Valid code is secure code.

Validate your code: http://www.jslint.com/

Comment as Much as Needed but Not More

“Good code explains itself” is an arrogant myth.

Comment what you consider needed - but don’t tell others your life story.

Avoid using the line comment //. /* */ is much safer to use because it doesn’t cause errors when the line break is removed.

If you debug using comments, there is a nice little trick:

```
module = function(){
   var current = null;
/*
   var init = function(){
   };
   var show = function(){
      current = 1;
   };
   var hide = function(){
      show();
   }
// */
   return{init:init, show:show, current:current}
}();
```

Comments should never go out to the end user in plain HTML or JavaScript. See **Development code is not live code**

Avoid Mixing with Other Technologies

JavaScript is good for calculation, conversion, access to outside sources (Ajax) and to define the behavior of an interface (event handling). Anything else should be kept to the technology we have to do that job.

FOR EXAMPLE:

Put a red border around all fields with a class of “mandatory” when they are empty.

```
var f = document.getElementById('mainform');
var inputs = f.getElementsByTagName('input');
for(var i=0,j=inputs.length;i<j;i++){
   if(inputs[i].className === 'mandatory' && inputs.value === ''){
      inputs[i].style.borderColor = '#f00';
      inputs[i].style.borderStyle = 'solid';
      inputs[i].style.borderWidth = '1px';
   }
}
```

...Two months down the line: All styles have to comply with the new company style guide, no borders are allowed and errors should be shown by an alert icon next to the element.

People shouldn't have to change your JavaScript code to change the look and feel.

```
var f = document.getElementById('mainform');
var inputs = f.getElementsByTagName('input');
for(var i=0,j=inputs.length;i<j;i++){
   if(inputs[i].className === 'mandatory' && inputs.value === ''){
      inputs[i].className+=' error';
   }
}
```

Using CSS inheritance you can avoid having to loop over a lot of elements.

Use Shortcut Notations

Shortcut notations keep your code snappy and easier to read once you get used to it.

This code

```
var lunch = new Array();
lunch[0]='Dosa';
lunch[1]='Roti';
lunch[2]='Rice';
lunch[3]='what the heck is this?';
```

Is the same as...

```
var lunch = [
   'Dosa',
   'Roti',
   'Rice',
   'what the heck is this?'
];
```

This code

```
if(v){
   var x = v;
} else {
   var x =10;
}
```

Is the same as...

```
var x = v || 10;
```

This code

```
var direction;
if(x > 100){
   direction = 1;
} else {
   direction = -1;
```

```
}
```

Is the same as...

```
var direction = (x > 100) ? 1 : -1;
```

Modularize

Keep your code modularized and specialized.

It is tempting and easy to write one function that does everything. However, as you extend the functionality you will find that you do the same things in several functions.

To prevent that, make sure to write smaller, generic helper functions that fulfill one specific task rather than catch-all methods.

At a later stage you can also expose these functions when using the revealing module pattern to create an API to extend the main functionality.

Good code should be easy to build upon without rewriting the core.

Enhance Progressively

Avoid creating a lot of JavaScript dependent code.

DOM generation is slow and expensive.

Elements that are dependent on JavaScript but are available when JavaScript is turned off are a broken promise to our users.

Allow for Configuration and Translation

Everything that is likely to change in your code should not be scattered throughout your code.

This includes labels, CSS classes, IDs and presets.

By putting these into a configuration object and making this one public we make maintenance easy and allow for customization.

For example:

```
carousel = function(){
   var config = {
      CSS:{
         classes:{
            current:'current',
            scrollContainer:'scroll'
         },
         IDs:{
            maincontainer:'carousel'
         }
      },
```

```
        labels:{
            previous:'back',
            next:'next',
            auto:'play'
        },
        settings:{
            amount:5,
            skin:'blue',
            autoplay:false
        },
    };
    function init(){
    };
    function scroll(){
    };
    function highlight(){
    };
    return {config:config,init:init}
}();
```

Summary

In this section, we covered the best practices for naming variables, commenting, and gave a few tips to help you organize your code. Head over to part 2 to learn more about effective ways to deal with loops, nesting and more.

Go To JavaScript Best Practices Part 2

If you enjoyed this guide, you might also like our intro to jQuery guide. Also, if you're interested in learning more about web development, you should take a look at our Frontend Web Development Course or our AngularJS Course.