JavaScript “Best Practices”

Christian Heilmann | http://wait-till-i.com | http://scriptingenabled.org

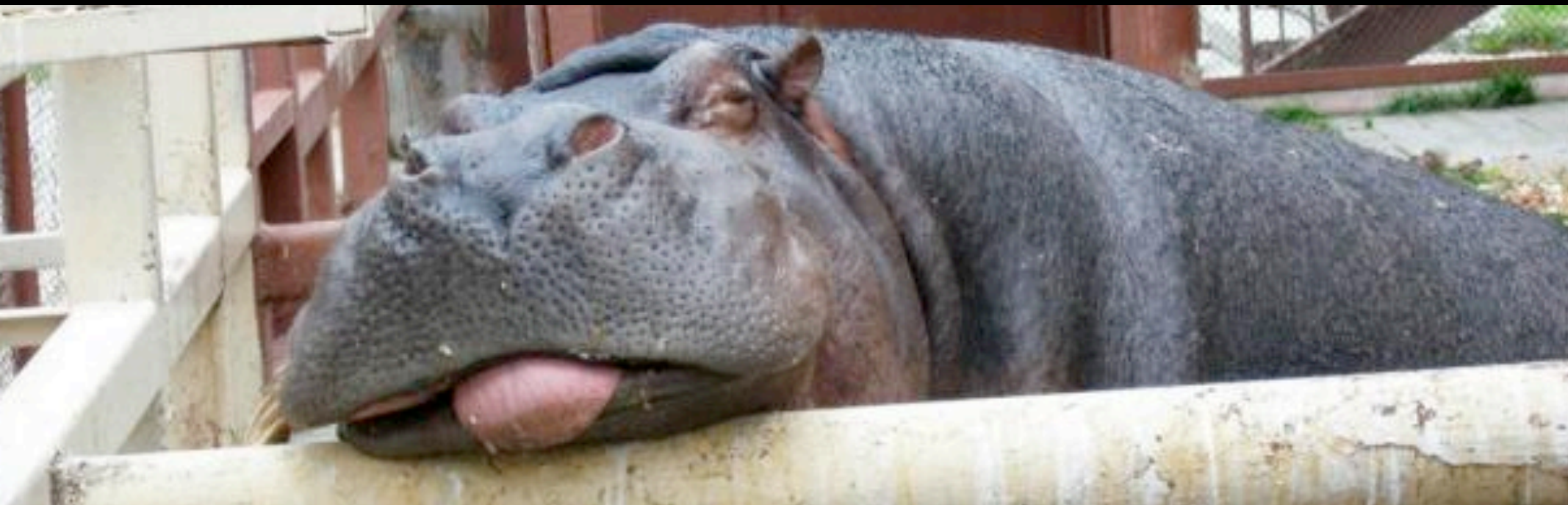
Bangalore, India, Yahoo internal training, February 2009

“Best Practice” presentations are hard to do.

Trying to tell developers to change their ways and follow your example is a tough call.

I also consider it a flawed process.

I am bored of discussions of syntax details.




This is not about
forcing you to
believe what I
believe.

This is about telling you what worked for me and might as well work for you.

I’ve collected a lot of ideas and tips over time how to be more effective.

- Make it understandable
- Avoid globals
- Stick to a strict coding style
- Comment as much as needed but not more
- Avoid mixing with other technologies
- Use shortcut notations
- Modularize
- Enhance progressively
- Allow for configuration and translation
- Avoid heavy nesting
- Optimize loops
- Keep DOM access to a minimum
- Don't yield to browser whims
- Don't trust any data
- Add functionality with JavaScript, not content
- Build on the shoulders of giants
- Development code is not live code

Make it understandable!

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

Bad variable names:

`x1 fe2 xbqne`

Also bad variable names:

`incrementorForMainLoopWhichSpansFromTenToTwenty`

`createNewMemberIfAgeOverTwentyOneAndMoonIsFull`

Avoid describing a value with your variable or function name.

For example
`isOverEighteen()`
might not make sense in some countries,
`isLegalAge()` **however works everywhere.**

Think of your code as a story – if readers get stuck because of an unpronounceable character in your story that isn’t part of the main story line, give it another, easier name.


Avoid globals

Global variables are a terribly bad idea.

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

The workaround is to use closures and the module pattern.

```
var current = null;
var labels = [
  ‘home’:’home’,
  ‘articles’:’articles’,
  ‘contact’:’contact’
];
function init(){
};
function show(){
};
function hide(){
};
```

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

Everything is global and can be accessed

Problem: access is not contained, anything in the page can overwrite what you do.

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

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

Anonymous Module: Nothing is global.

Problem: No access from the outside at all (callbacks, event handlers)

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

Module Pattern: You need to specify what is global and what isn't – switching syntax in between.

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

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

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'
  ];
  function init(){
  };
  function show(){
    current = 1;
  };
  function hide(){
    show();
  };
  return{init:init,show:show,current:current}
}();
module.init();
```

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

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 good code.

Valid code is secure code.

Validate your code:

http://www.jslint.com/

TextMate users: get Andrew's JavaScript Bundle:

http://andrewdupont.net/2006/10/01/javascript-tools-textmate-bundle/

Comment as much as needed but no more

Comments are messages from developer to developer.

“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 though. It is much safer to use /* */ as that 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;
  function init(){
  };
/*
  function show(){
    current = 1;
  };
  function hide(){
    show();
  };
*/
  return{init:init,show:show,current:current}
}();
```

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

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

Comments can be used to write documentation – just check the YUI doc:

http://yuiblog.com/blog/2008/12/08/yuidoc/

However, comments should never go out to the end user in plain HTML or JavaScript.

Back to that later :)

Avoid mixing with other technologies

JavaScript is good for calculation, conversion, access to outside sources (Ajax) and to define the behaviour 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[i].value === ''){
    inputs[i].style.borderColor = '#f00';
    inputs[i].style.borderStyle = 'solid';
    inputs[i].style.borderWidth = '1px';
  }
}
```

Two month 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[i].value === ''){
    inputs[i].className+=’ error’;
  }
}
```

Using classes you keep the look and feel to the CSS designer.

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

Use shortcut notations

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

```
var cow = new Object();
cow.colour = ‘white and black’;
cow.breed = ‘Holstein’;
cow.legs = 4;
cow.front = ‘moo’;
cow.bottom = ‘milk’;
```

is the same as

```
var cow = {
  colour:‘white and black’,
  breed:‘Holstein’,
  legs:4,
  front:‘moo’,
  bottom = ‘milk’
};
```

```
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?'
];
```

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

is the same as

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

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

is the same as

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

/* Avoid nesting these! */
```


Modularize

Keep your code modularized and specialized.

It is very tempting and easy to write one function that does everything.

As you extend the functionality you will however 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 when using the revealing module pattern to create an API to extend the main functionality.

Good code should be easy to build upon without re-writing the core.

Enhance progressively

There are things that work on the web.

Use these rather than 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.

Example: TV tabs.

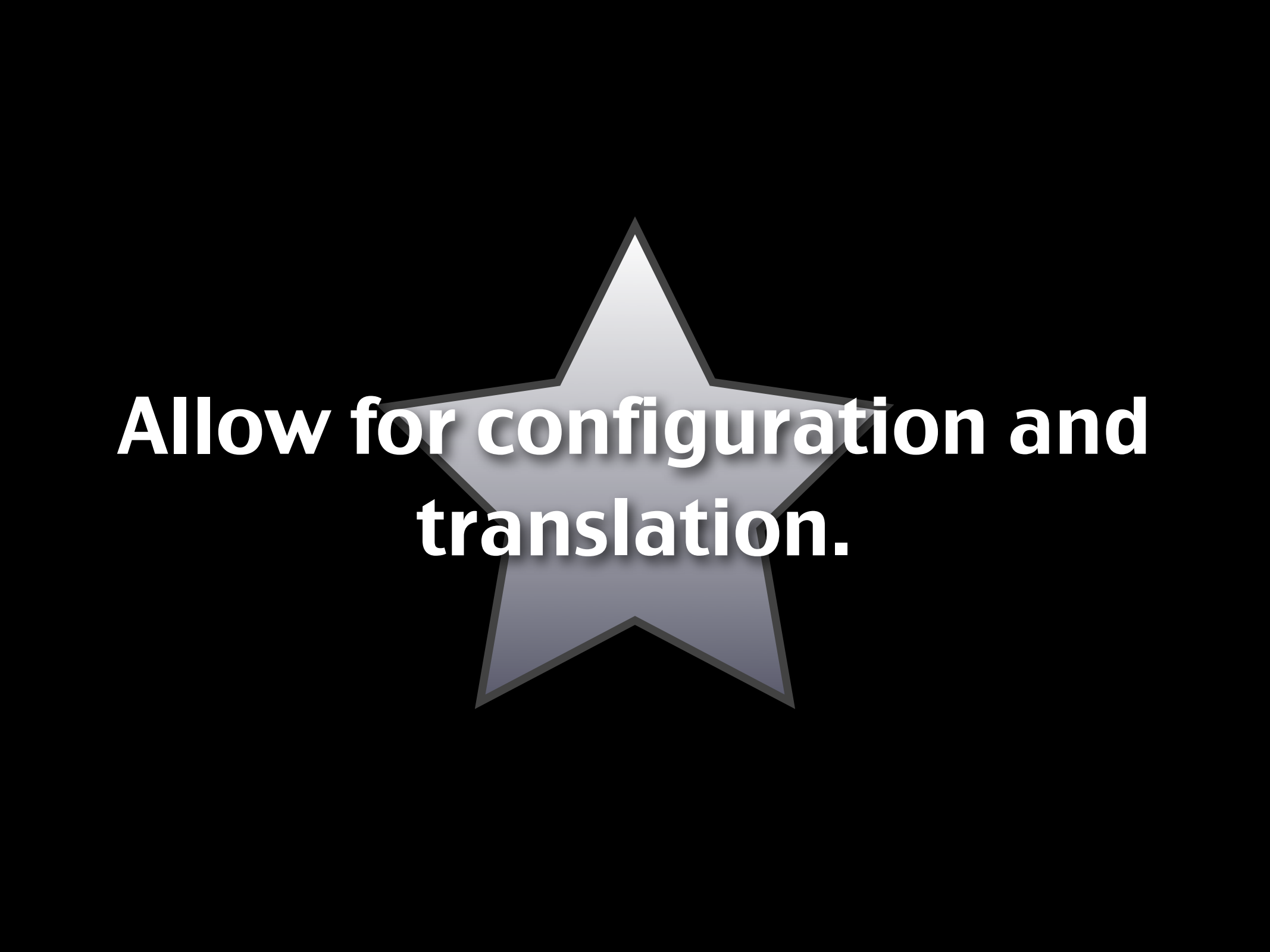
Web Images | Video | Audio

SEARCH

Search

- The web
- For images
- For video
- For audio

SEARCH


Allow for configuration and translation.

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

This includes labels, CSS classes, IDs and presets.

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

```
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}
}();
```

Avoid heavy nesting

Code gets unreadable after a certain level of nesting – when is up to your personal preference and pain threshold.

A really bad idea is to nest loops inside loops as that also means taking care of several iterator variables (i,j,k,l,m...).

You can avoid heavy nesting and loops inside loops with specialized tool methods.

```
function renderProfiles(o){
  var out = document.getElementById(‘profiles’);
  for(var i=0;i<o.members.length;i++){
    var ul = document.createElement(‘ul’);
    var li = document.createElement(‘li’);
    li.appendChild(document.createTextNode(o.members[i].name));
    var nestedul = document.createElement(‘ul’);
    for(var j=0;j<o.members[i].data.length;j++){
      var datali = document.createElement(‘li’);
      datali.appendChild(
        document.createTextNode(
          o.members[i].data[j].label + ‘ ‘ +
          o.members[i].data[j].value
        )
      );
      nestedul.appendChild(datali);
    }
    li.appendChild(nestedul);
  }
  out.appendChild(ul);
}
```

```
function renderProfiles(o){
  var out = document.getElementById('profiles');
  for(var i=0;i<o.members.length;i++){
    var ul = document.createElement('ul');
    var li = document.createElement('li');
    li.appendChild(document.createTextNode(data.members[i].name));
    li.appendChild(addMemberData(o.members[i]));
  }
  out.appendChild(ul);
}
function addMemberData(member){
  var ul = document.createElement('ul');
  for(var i=0;i<member.data.length;i++){
    var li = document.createElement('li');
    li.appendChild(
      document.createTextNode(
        member.data[i].label + ' ' +
        member.data[i].value
      )
    );
  }
  ul.appendChild(li);
  return ul;
}
```

Think of bad editors and small screens.

Optimize loops

Loops can get terribly slow in JavaScript.

Most of the time it is because you're doing things in them that don't make sense.

```
var names = ['George','Ringo','Paul','John'];
for(var i=0;i<names.length;i++){
  doSomeThingWith(names[i]);
}
```

This means that every time the loop runs, JavaScript needs to read the length of the array.

You can avoid that by storing the length value in a different variable:

```
var names = ['George','Ringo','Paul','John'];
var all = names.length;
for(var i=0;i<all;i++){
  doSomeThingWith(names[i]);
}
```

An even shorter way of achieving this is to create a second variable in the pre-loop condition.

```
var names = ['George','Ringo','Paul','John'];
for(var i=0,j=names.length;i<j;i++){
  doSomeThingWith(names[i]);
}
```

Keep computation-heavy code outside of loops.

This includes regular expressions but first and foremost DOM manipulation.

You can create the DOM nodes in the loop but avoid inserting them to the document.

Keep DOM access to a minimum

If you can avoid it, don’t access the DOM.

The reason is that it is slow and there are all kind of browser issues with constant access to and changes in the DOM.

Write or use a helper method that batch-converts a dataset to HTML.

Seed the dataset with as much as you can and then call the method to render all out in one go.

Don't yield to browser whims!

What works in browsers today might not tomorrow.

Instead of relying on flaky browser behaviour and hoping it works across the board...

...avoid hacking around and analyze the problem in detail instead.

Most of the time you'll find the extra functionality you need is because of bad planning of your interface.

Don't trust any data

The most important thing about good code is that you cannot trust any data that comes in.

Don’t believe the HTML document – any user can meddle with it for example in Firebug.

Don’t trust that data that gets into your function is the right format – test with `typeof` and then do something with it.

Don’t expect elements in the DOM to be available – test for them and that they indeed are what you expect them to be before altering them.

And never ever use JavaScript to protect something – it is as easy to crack as it is to code :)

Add functionality with JavaScript, don't create content.

If you find yourself creating lots and lots of HTML in JavaScript, you might be doing something wrong.

It is not convenient to create using the DOM...

...flaky to use innerHTML (IE's Operation Aborted error)...

...and it is hard to keep track of the quality of the HTML you produce.

If you really have a massive interface that only should be available when JavaScript is turned on...

...load the interface as a static HTML document via Ajax.

That way you keep maintenance in HTML and allow for customization.

Build on the shoulders of giants

JavaScript is fun, but writing JavaScript for browsers is less so.

JavaScript libraries are specifically built to make browsers behave and your code more predictable by plugging browser holes.

Therefore if you want to write code that works without keeping the maintenance overhead...

... of supporting current browsers and those to come to yourself...

... start with a good library. (YUI)

Development code is not live code.

Last but not least I want you to remember that some things that work in other languages are good in JavaScript, too.

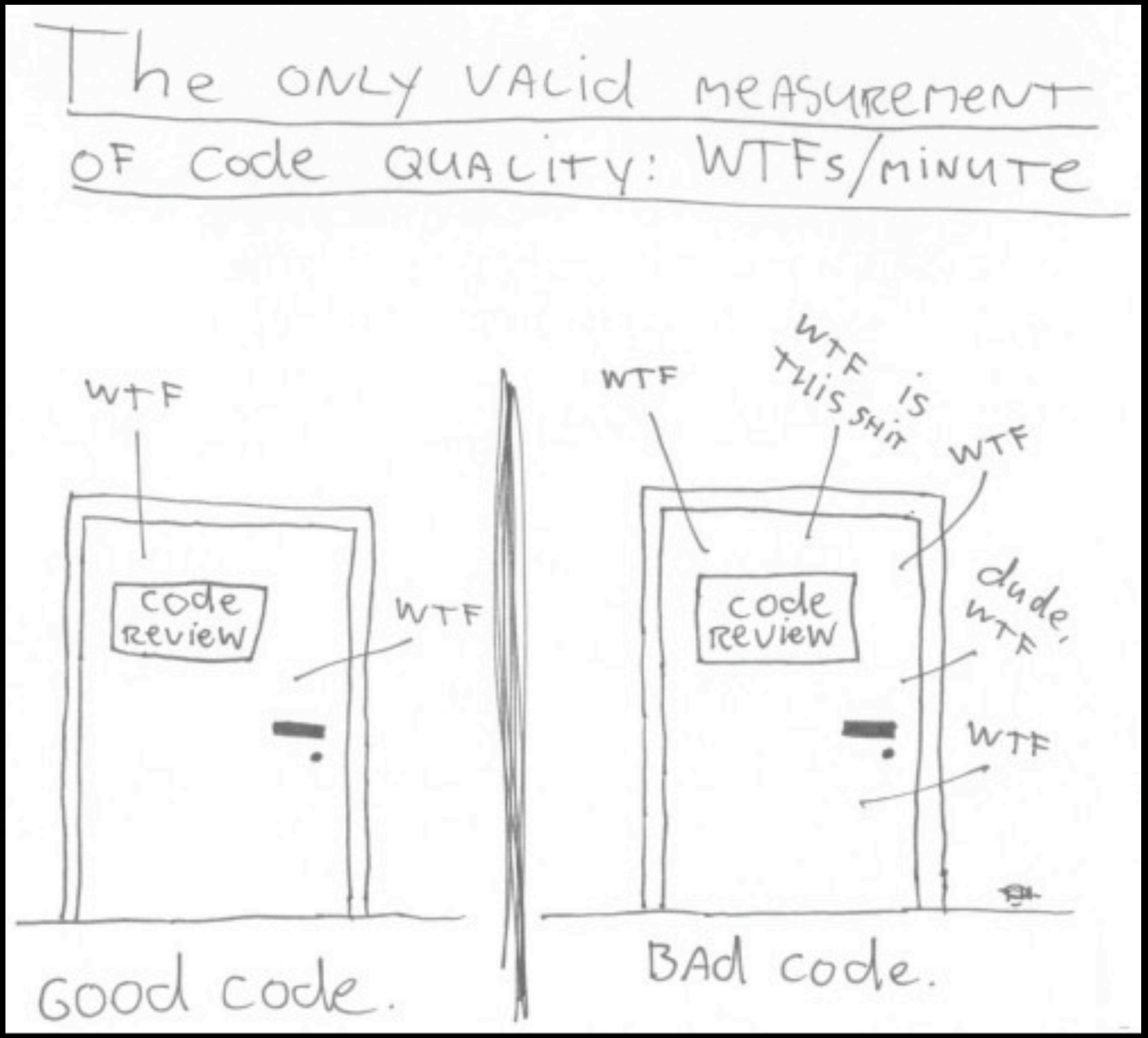
Live code is done for machines.

Development code is done for humans.

Collate, minify and optimize your code in a build process.

Don’t optimize prematurely and punish your developers and those who have to take over from them.

If we cut down on the time spent coding we have more time to perfect the conversion to machine code.



T H A N K S !

Keep in touch:

Christian Heilmann

http://wait-till-i.com

http://scriptingenabled.org

http://twitter.com/codepo8

http://delicious.com/codepo8/jscodetips