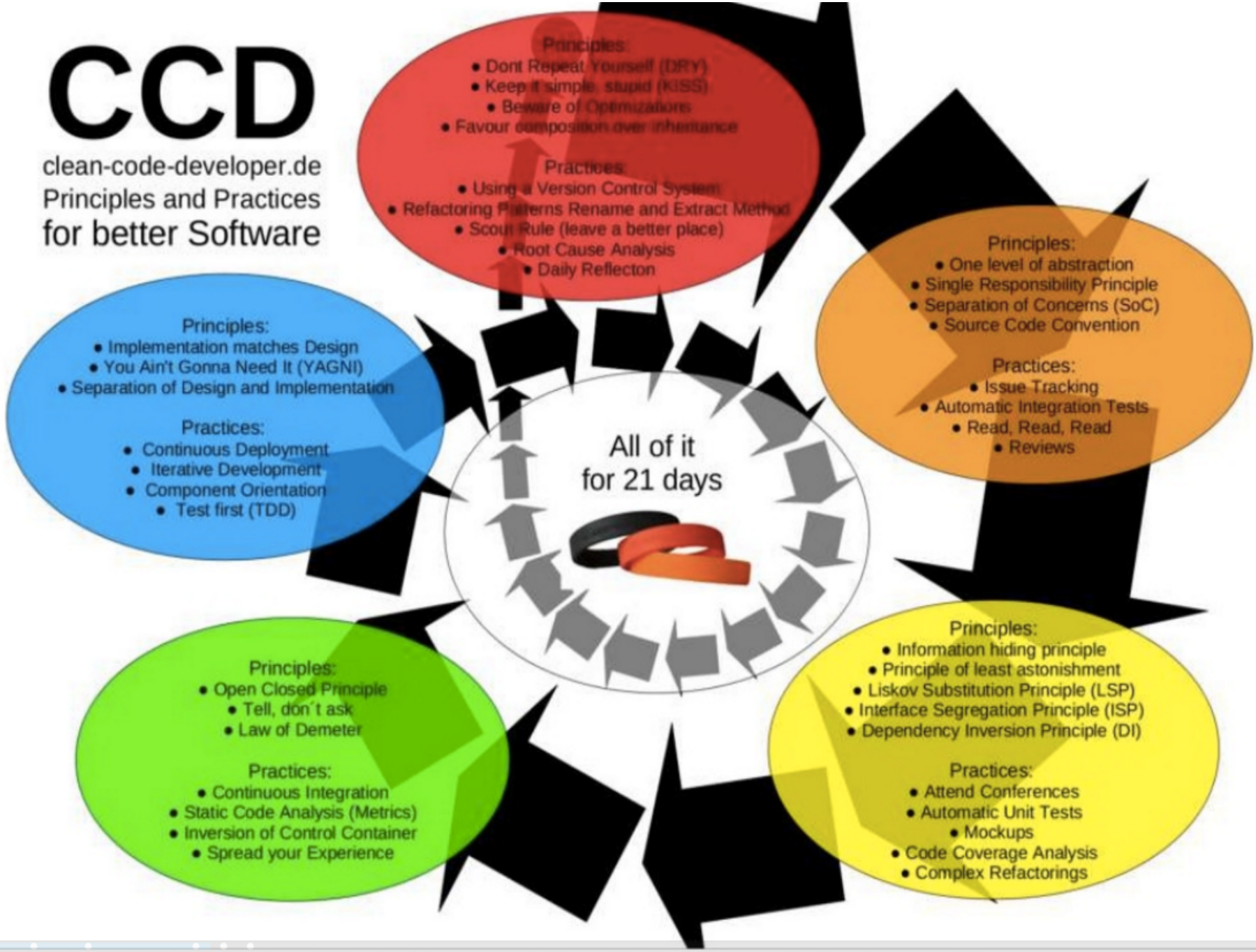
Recovery Plan



Graphic by Michael Hönnig http://michael.hoennig.de/2009/08/08/clean-code-developer-ccd/

Diving In - Orient

- Find out what you have to work with
  - File and Directory List
    - Purpose
    - Sizing
  - Read the docs and comments
    - Ask questions if you have someone to ask
  - Class Diagram
  - Understand the basic flow
  - Get it to "work"
    - Quotes are purposeful, if it is junk your thought should be never making it worse than before

Diving In - Evaluate

- Evaluate
  - Code Metrics
    - LOC
    - Duplicates
    - Complexity
    - Linters
  - Ask questions if you can
  - Trying to see where we are going to "poke first"
    - General Tactic
      - Something small "off to the side" but used
      - Prove you can do something and not break it
    - Triage Tactic
      - Core method/class first, dangerous but might be applicable for a certain situation

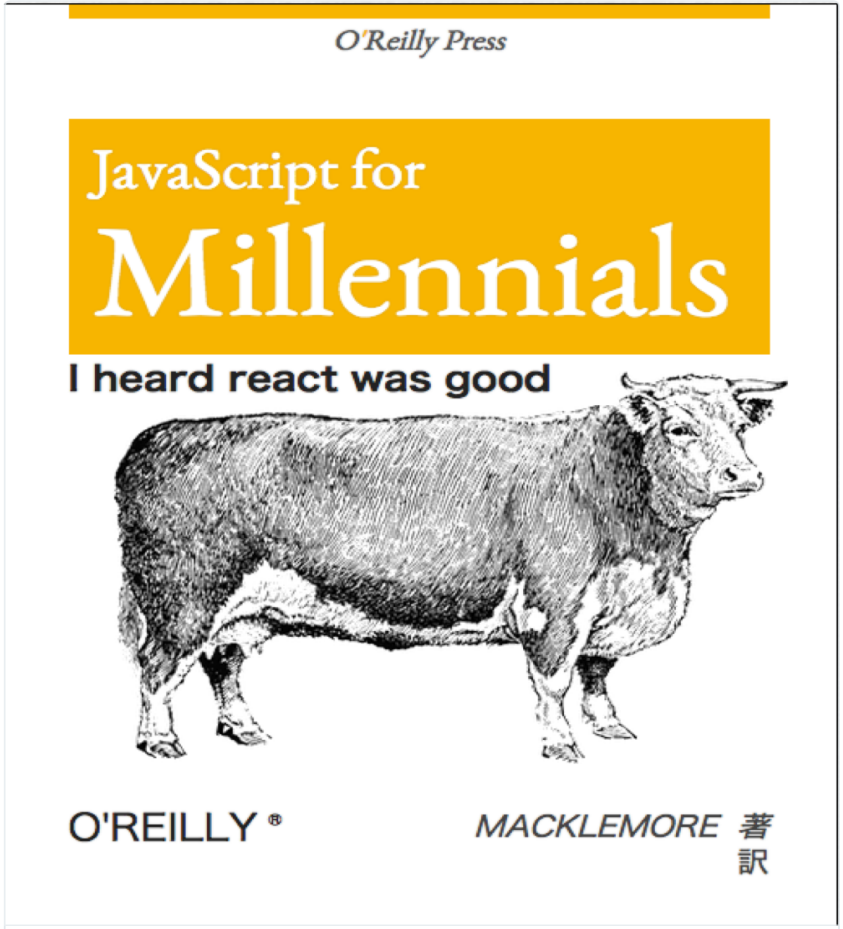
Diving In – Normalize

- Start normalizing the code base so it becomes your own
- Start small
  - File and directory names
  - File organization – top to bottom, props/method order
  - Source Formatting
  - Avoid refactoring like renaming props and methods on the first pass you are going to do many passes
  - You are going to need a coding guideline!
- As you go make sure it still works
  - First test is just does it load and do what it did before

Short JavaScript Aside

JavaScript

- Widely misunderstood weakly typed scripting language dominantly used in browser based applications and Web sites
  - No limits - server-side (ex. ASP, Node.js), desktop, mobile, etc.
  - Think "host" environment - in this case the browser is the host



Wes Bos
@wesbos

Follow

Super excited for this new O'Reilly book

8:17 AM - 12 May 2015

2,115 1,702

JS Versions

- ECMAScript 3, 5, 5.1, [6, 7] (ES 2015]
  - We can use a transpiler to use advanced stuff
- Be warned of the host model chaos
  - Browser based object model
    - BOM, DOM nightmare
  - Reaction: Oldest only, Newest only, Facade/Wrapper/Polyfill/… (aka Abstract away the differences)

JavaScript Points of Interest

- Statements terminate with return or semi-colon
  - Always use ; to avoid problems
- Variables put in global scope by default unless within a function and var is used

```
var x = 'trouble'; // global
function () {
  y = 'more trouble'; // global
  var z = 'ok local';  // local
}
```

Typing Basics

- Primitive: Number, String, Boolean, undefined, null
- Composite/Reference: Object, Array*,Function
- ECMA non-Host: RegExp, Date, Math, Error

Simple Primitives?

- var foo; // undefined
- var name = thomas,num = 3,likeJS = true;
- var name = null;
- But…
  - alert(name.length); // 6
    alert(name.toUpperCase()); //THOMAS
    alert('tom'.toUpperCase()); //TOM

Objects

- document.write('pretty obvious');
- document['write']('not as obvious?');
- In JS is [ ] = { } ? (not quite, but close)
  - JS objects are just hashes
- JS doesn't do class style inheritance, it uses prototypes, but you can simulate whatever you want (PLEASE DON'T!)

It's Harder than You Think

- Given function foo(x,y,z){}
  - How are x,y,z passed?
- Implicit pointers!?!
- Functions are data types
- Closures
- Dynamic
  - Override whatever you want

Nasty Lessons

- Browser Detection
- Capability Detection
- Memory Management
  - No forced garbage collection, only hinting
- Bad tooling
- Hostile environment
- Tons of folklore

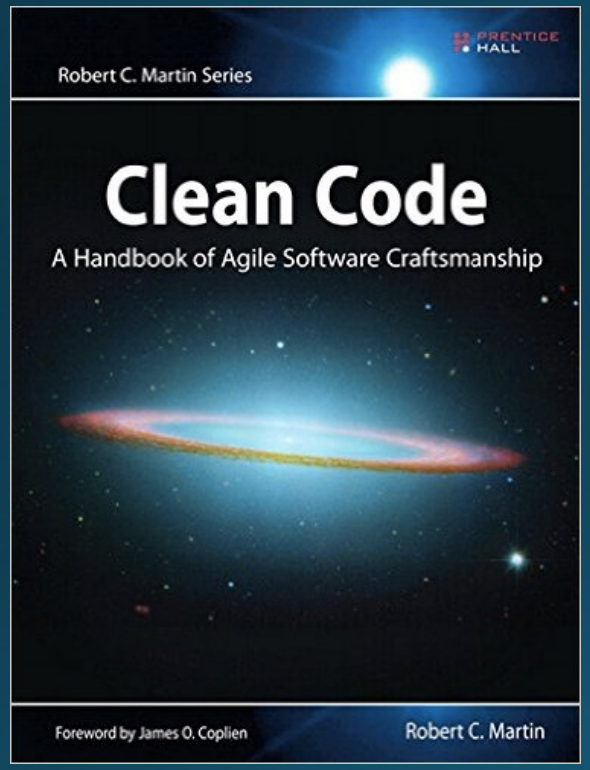
Coding Standards

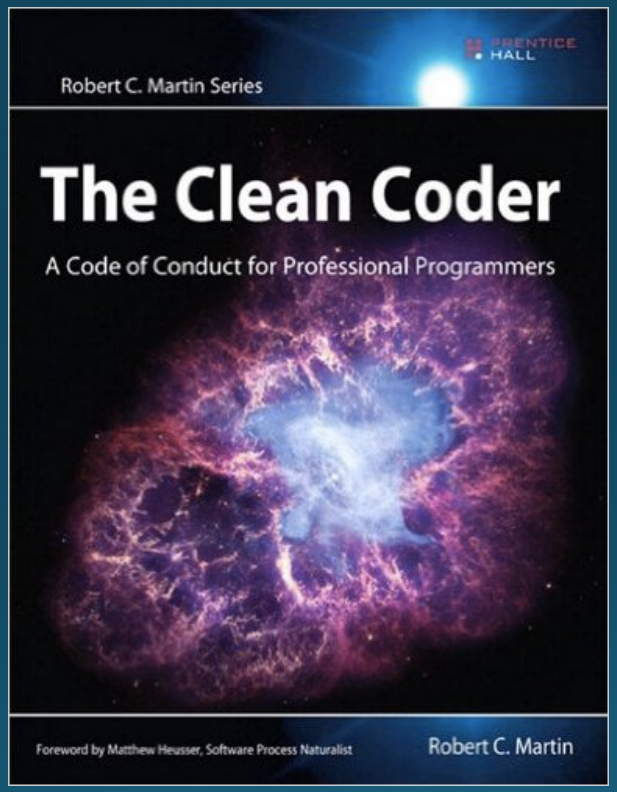
- Being precise in how we are coding can be immensely valuable in the long run
- Ultimate Goal: All code looks like made by same person (even if it isn't)
- Significantly aids in readability once style understood
  - Fact: You read code more than you write code
  - Will reduce speed up front gain much later

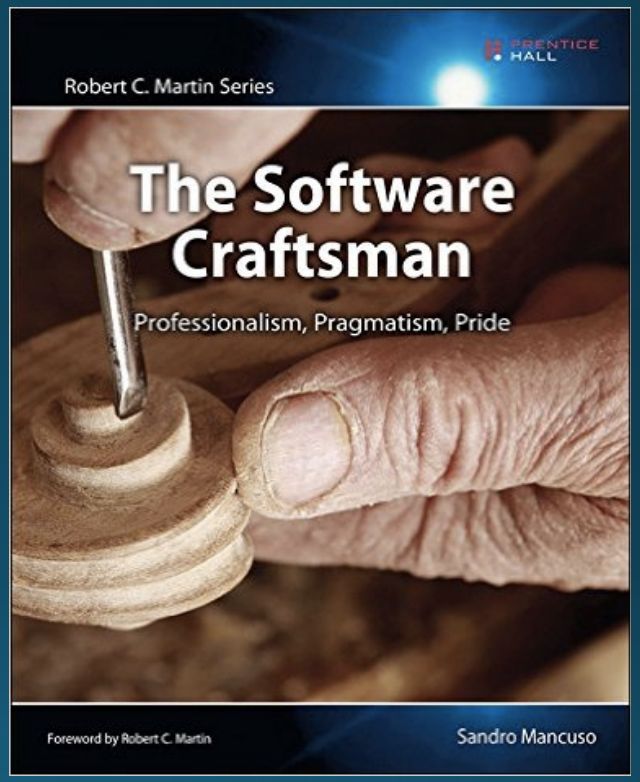
JavaScript Coding Standards

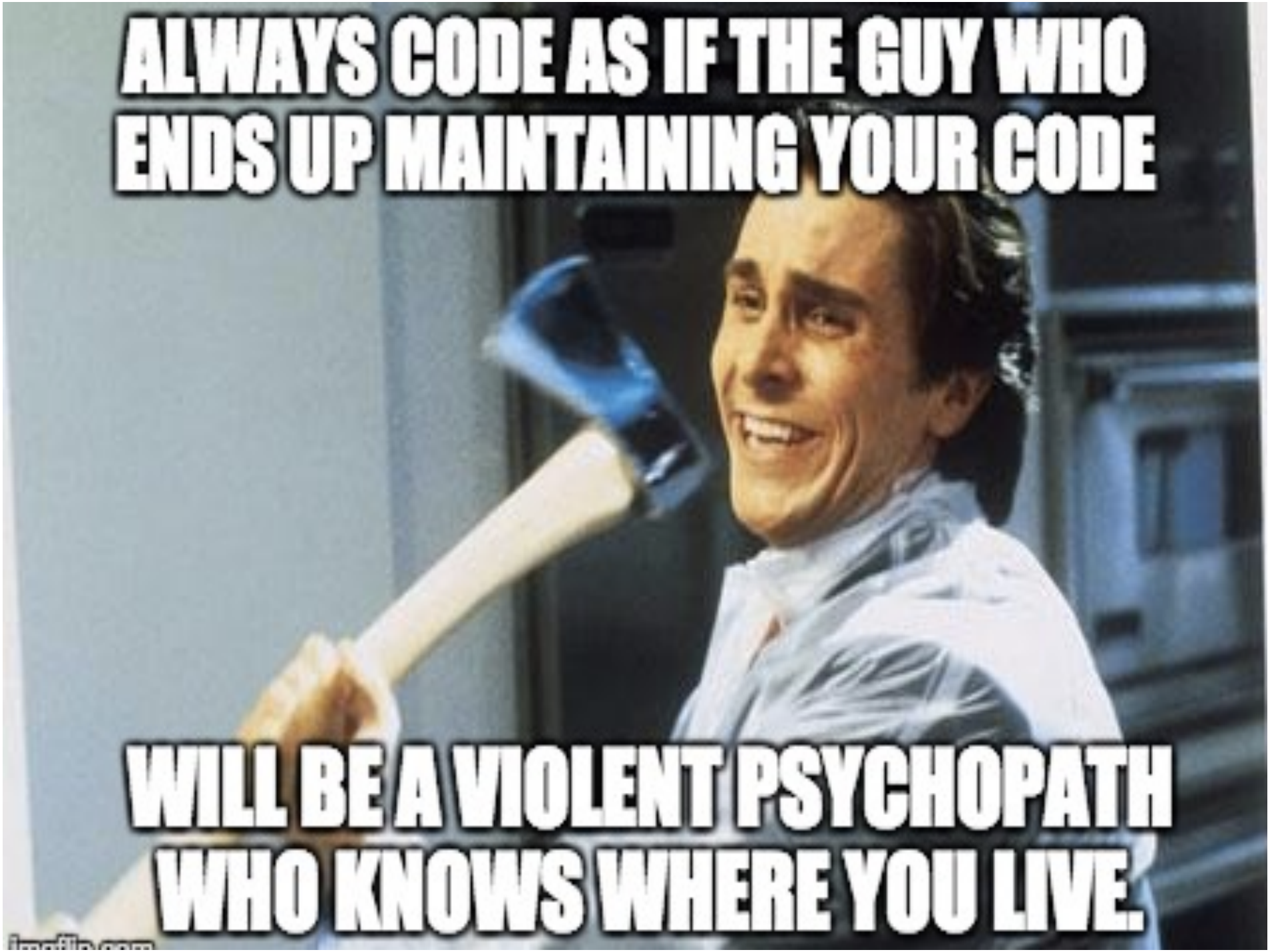
- Since language isn't owned by a sole patron many different standards
  - Google, Yahoo, Mozilla, various authors/bloggers, your company, etc.
- Many standards presented are illustrative and likely will fit many languages.
- Specific JS, DOM, Web browser things will be called out though
- There are many online that you can adopt
  - Nothing will be perfect and may upset your sensibilities
  - Good candidate we've used AirBnB JS guidelines, consider JS version though!
  - General review follows (out of order for how I might teach the course top down!)

Some Resources








ALWAYS CODE AS IF THE GUY WHO ENDS UP MAINTAINING YOUR CODE
WILL BE A VIOLENT PSYCHOPATH WHO KNOWS WHERE YOU LIVE.

Go Slow

- Writing quality code is something that requires you to go slower
  - For construction – measure twice, cut once
  - For programming – think, wait, think again, then code
- Remember going slow doesn't mean you get less done as you might save time in refactoring and bug tracking

Why is Bad Code Written?

- Went to fast?
- Fatigue
- Brain Fart
- Bored, apathy, disinterested, don't like work environment, so no passion
- Got it working, clean it up later, never later because more stuff
- Playing above one's level* (unaware of the bad code)

- Regardless of cause everybody does it, and more often than they think

Naming Basics

- Meaningful Names
  - We all agree that we want names that mean something but it is difficult to get people to agree on what makes sense here.
  - Set **some** approach and just stick to it! We have important things to do!
- Searchable Names
  - How many len, i, tmp, me, options, etc. will be find in our code base if we do a search
- Of course IDEs can help but you have to
  - Code so the IDE and related debuggers can index well and help you
    - Ex: Call stack with Anonymous Function, Anonymous Function, etc. over and over

Naming

- Conventions
  - Coding in general: I, j, k, l, len, etc.
  - Your language: $, _this, el, etc.
  - The domain: session, userAgent, etc.
- Remember read by programmers not by non-programmers, but I contend if you write so a non-programmer or at least a novice can understand what you did you really did a good job
  - Even those of us who understand, don't want to put the mental energy in if we don't have too!

Higher Level Constructs

- Functions
  - Well named
  - Should be small
  - Should do ~1 thing! (1 idea)
    - Error handling is a thing so consider extraction?
  - Smallish argument list
    - Priority arguments
    - Optional arguments
    - Overflow / growth arguments
  - No side-effects

- Classes themselves follow same principle
  - Should be small (this seems to be a relative term!)
  - Single Responsibility Principle (SRP) → Do 1 thing again (higher level)
  - SQL -> SELECT, UPDATE

- What about packages/modules?
  - Should be small (where have I heard that before)
  - SRP
  - Do ~1 things (even higher level)

Basic High Level Affordances

- Where is the entry point of the code
- Where are things defined
  - Is it consistent
  - Avoid doing it as you go along, put in one place generally the top
- Consider a pattern for your classes for example
  - Constructor, properties (public), methods (public) , properties (private), methods (private)

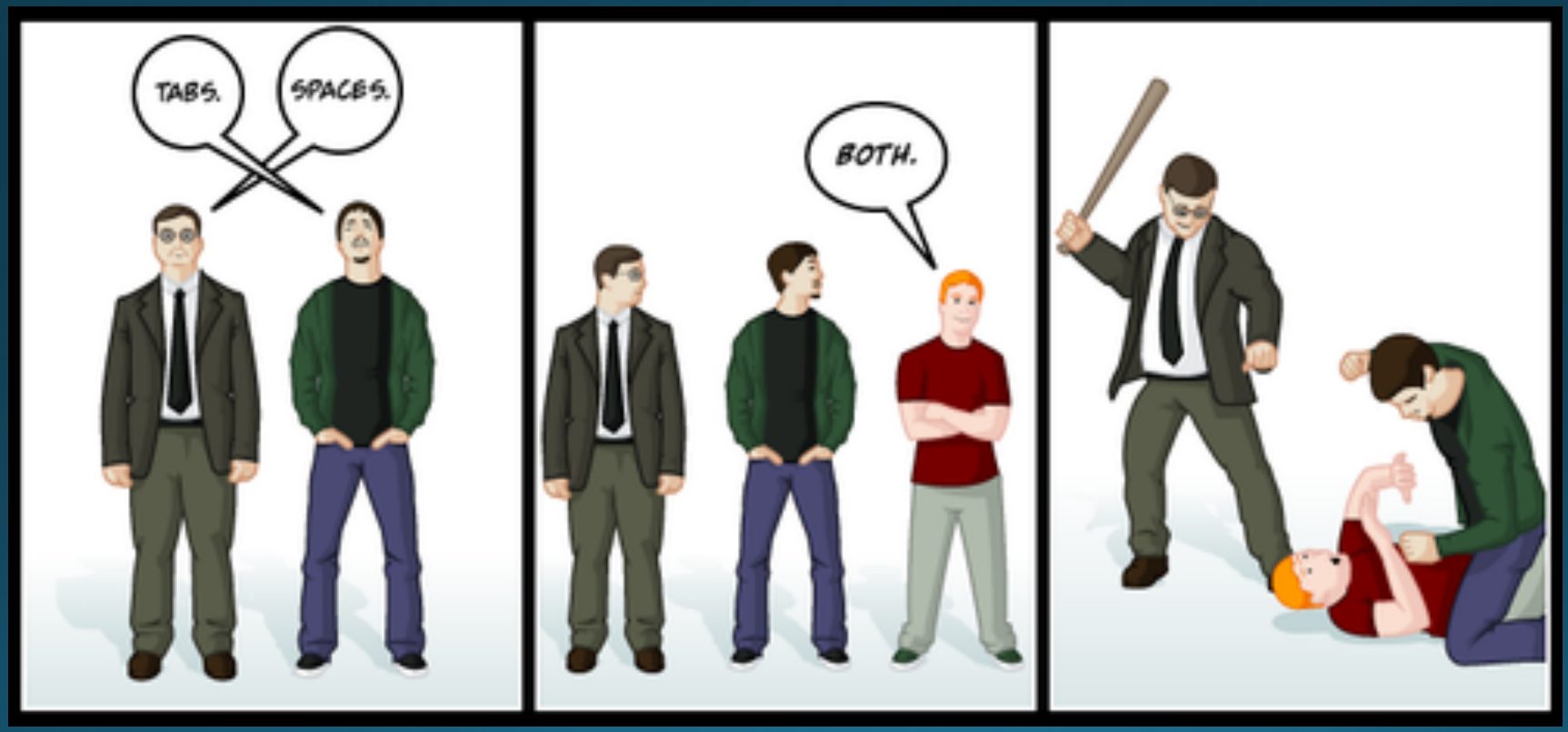
Commenting

- Don't do it if you don't need to
- Aim to be self-understood and in isolation if so commenting doesn't add much
- Required if you can't do better, if you can keep refactoring until the comment is useless
- Comments can be warning – "Don't touch", "Hot Wire", etc.
- Comments can be work flow – "TODO, REFACTOR, etc.
  - HACK is a common one, but seems more judgemental than anything
- Comments can be historical
  - // TAP added this – 1/5/2016 (maybe should have let revision control do!)
  - // BUG #14567 http://tracker.com/id=14567 (this can be useful it allows us to avoid putting context into the comment

Commenting Troubles

- Excessive commenting leads to code bloat
  - Visual Noise that makes seeing the code difficult
  - Counter Argument can be a visual anchor to break apart sections
    - IDEs can help here with code folding but again do you use properly
- Commented out code is a form of code rot. Very short term it is ok as you might test things but
  - You can use your local VCS to avoid doing if you get good a stashing, shelving, etc.
  - You can use scratch files to deal with this in Webstorm for example
  - You can make a convention and catch it on check in, in case you forget
- It is a form of code level Yak Shaving sometimes in service of making it pass some coding guidline

Coding Guidelines



There are two types of people.

Programmers will know.

Indentation

- Holy War
  - How? Tabs or spaces
  - How many? 2, 3, 4
- Whatever picked keep same
- If we fight horizontal why not vertical?

Semi-colons

- JavaScripts does allow return instead of ;
- Always use semi-colons
  - Avoid ASI because it can cause some subtle errors
    - Copy-paste
    - Poor minification breakages

Line Length

- Aim for lines of no more than ~100 chars
- What about big monitors?
  - Font bigger? Same rule
  - Small font? Beware normal human limits
- If lines must break, break on operator and indent appropriately

Variable Naming

- JavaScript promotes camelcase
  - `var myName, myLastName, veryLongLongName;`
  - Short names (ex. `var i,j,k`)
    - Result of minification, loop use
  - Long names
    - Descriptive: `var strUserName;`
    - Obfuscated: `var _10101001010;`

Adding _ to the mix?

- Standard camelCase

```
var myLastName;
```

- With _ (snake case)

```
var my_Last_Name;
```

- Value: Distinguish between your properties and vars and the environment

Return of Hungarian Style?

- JavaScript weak typing can be a challenge
- Hungarian notation may be useful
  - `sMyName` or `strMyName` (Strings)
  - `bLikeJS` or `boolLikeJS` (Booleans)
  - `nAge` or `numAge` (Numbers)
  - `aStooges` or `arrStooges` (Arrays)
  - `oConfig` or `objConfig` (Objects)
- My view on using it is not widely held

Constants

- JavaScript natively doesn't support constants
  - Wide browser support though and if we use a transpiler like Babel we could use them

- Regardless of approach use all caps to signify a "constant"

```
var MAX_REMINDERS = 3,
MAGIC_NUMBER = 9;
```

- Note the native language does this also `Math.PI`

Function Naming

- a vs b(); // no worries pretty obvious
- Since functions are first class distinguishing difference of vars/props/methods/functions can be challenging
  - var myX = myY; // is myY a function?
- Also I think we should consider our names versus environmental names

Action Naming

Prefix	Returns	Example
can	Boolean	`canSend()`
has	Boolean	`hasAppt()`
is	Boolean	`isCancelled()`
get	Non-Boolean	`getAppt()`
set	Saves value	`setAppt()`

Style Could Reveal

- If using Hungarian anything w/o would be a function. Unless you decided to decorate function names with return types!
- If you named variables with _ then w/o is a function (Snake_case Style)
- `like_JS` (var) vs `likeJS` (function)
- `bLikeJS` (var) vs `likeJS` (function)
- `b_Like_JS` (var) vs `bLikeJS` (function)

Construction Functions

- Initial Cap, but can be camel cased

```
function Appointment( ) {

} // appointment Constructor

var anAppt = new Appointment() ;

var bad = Appointment();

var maybe = objAppointment();

var maybe = new objAppointment();
```

Private Methods & Props

- Commonly we may name private values and methods with a _ prefix

```
var _statusCode;

function _logTransaction() { }
```

- Trouble when using _ naming idea?

```
var _status_Code;  // private

var status_Code;   // public

// Is there enough variation?
```

Single Line Comments

1. Before something

```
// Use bitmask to select used features

tricky =  nValue & MASK;
```

2. End of a line

```
var nTotal = (nSubTotal * (1 + TAX)); // Calc total with tax
```

Single Line Comments

3. To Comment Out Code Blocks

```
// /* doThings: A sample function
//  *   Inputs: none
//  *   Outputs: none
//  */
//
// function doThings() {
//
//  if (bSomething) {
//    doThis(); // still ok
//    doThat();
//  }
// } /* doThings */  // why did I put that there?
```

Multi-Line Comments

- Generally put at function / file top
- Doc generators can use tokens to auto generate API pages

```
/**
 * @method isWellFormedEmail
 *  Checks if passed value is well-formed email
 *  address syntax wise, does not mean address is
 *  currently working
 *
 *
 * @parameter {string} address - address to check
 * @return {boolean} - true if well formed
 *
 */
```

Comment Annotations

- TODO - Code to complete, indicate details if possible

```
// TODO - Add complicated regex for real email check
```

- HACK - Shortcut or ugly workaround. Indicate owner, date, and details of the hack

```
// HACK - TAP (1/2015) Used giant sparse array
```

- XXX - Indicates problem code to be fixed ASAP, but you can make your own obviously!

```
// XXX - Leaking memory here, replace event handler code
```

Comment Annotations

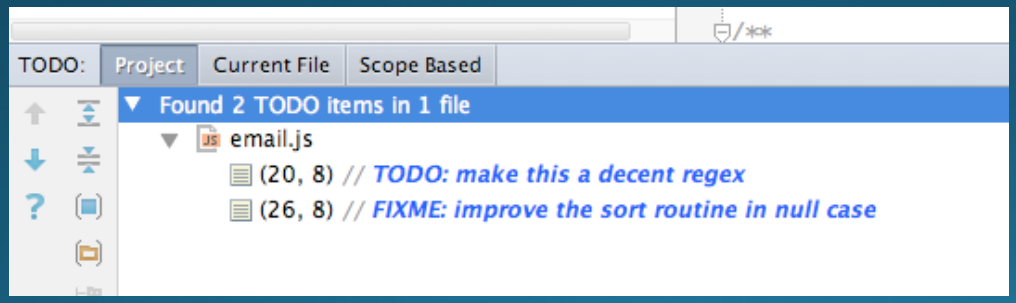
- FIXME - Indicates lower priority problems

```
// FIXME - Safari cosmetic rendering quirk
```

- REVIEW - Indicates code needs review

```
// REVIEW - Sending algorithm seems a bit wonky
```

- Some editors can add easy navigation with these comments



Keep JS out of CSS

- In older IEs we had JS expressions in CSS
- Huge performance hit
- Generally not an issue now but CSS is starting to take on logic features (animations, variables, list counters)
- Wouldn't be a stretch to see JS returning to CSS

Keep CSS out of JS

- Anti-Pattern

```
function changeStyle(el) {

 var element =
document.getElementById(el);

 // bad example
 // set in Halloween style
 element.style.color = "black";
 element.style.fontSize = "24px";
 element.style.backgroundColor = "orange";
 element.style.visibility = "visible";
}
```

Keep CSS out of JS

- Anti-Pattern

```
function changeStyle(el) {

 var element =
document.getElementById(el);

 // still a bad example
 // set in Halloween style
 element.style.cssText = "color: black;
font-size: 24px; background-color: orange;
visibility: visible";

}
```

Keep CSS out of JS

- Suggestion: Use Class Names

```
function changeStyle(el) {

 var element =
document.getElementById(el);

 if (element.classList) {
   element.classList.add("halloween");
 }
 else {
  element.className += " halloween";
 }
}
```

Keep CSS out of JS

- In CSS file some place

```
.halloween {
  color: black;
  font-size: 24px;
  background-color: orange;
  visibility: visible;
}
```

- Moved to more logical place, separation of concerns, but still is coupled
- Not discussing reflows
- Can promote class-itis

Keep JS out of HTML

- Common practice particular in old days

```
<button onclick="alert('really just say
no!');">Bad!</button>
```

- Better to name and bind

```
<button id= "myBtn">Better</button>


 // simple example - w/o cross browser nightmare

  var myBtn =
document.getElementById("myBtn");
  myBtn.onclick = function() {
    alert('ok a little better');
  };
```

Keep HTML out of JS

- Common practice even today

```
var output =
document.getElementById("errorMsg");
output.innerHTML = "<b>Error</b> Bury
my HTML in my code!";
```

- First thought just extract it

```
output.innerHTML = ERROR_MESSAGE;

// strings file contains
var ERROR_MESSAGE = "<b>Error</b> Bury my HTML in my
code!";
```

- It is still in the code though

Keep HTML out of JS

- Use some HTML template/fragment system

```
<b>{{error_type}}</b> {{error_msg}}
```

- But where to keep template in our HTML?

```
// hidden region
<div style="display: none;">
  <b>{{error_type}}</b> {{error_msg}}
</div>

// script tag overload
<script type="text/some-template-lang">
  <b>{{error_type}}</b> {{error_msg}}
</script>

// soon?
<template><b>{{error_type}}</b> {{error_msg}}</template>
```

Global Variables

- JavaScript makes them by default :-(
- ```
  badOutside = true;  // global
  var badOutside2 = true; // global
  function foo() {
    bad = true;         // this is global!
    var good = true;  // this is not
  }
  ```
  ```

Globals are bad …

- Promotes leaky abstractions (non-modular)
- May collide with environmental variables

```
var location = "San Diego";

window.location // safe or gone?
```

- May collide with other scripts

```
var temp;  // yeah nobody every uses that

var $ = function () { };  // zoinks!
```

- Stemming employed to reduce risk

```
function MM_SwapImage() { };  //
Macromedia lives
```

One Global Scheme

- Limit your global name space pollution with a single wrapper object. Many ways to do, IIFE return, simple singleton declaration, etc.

```
var UCSD=  { };
UCSD= true;
UCSD.method = function () {    };
```

- Obviously it can be blasted though it is a full failure as opposed to partial failure which is harder to deal with

When you use globals

- Why would we ever want to use globals?
- Can there be a reason besides laziness or ignorance?

- So if used at least indicate via a coding convention

```
var gMagicVar = 3;  // necessary evil

var gNumMagicVar = 3; // hungarian global
var g_Num_Magic_Var = 3; // with snake_style
```

Suggestion: Automation

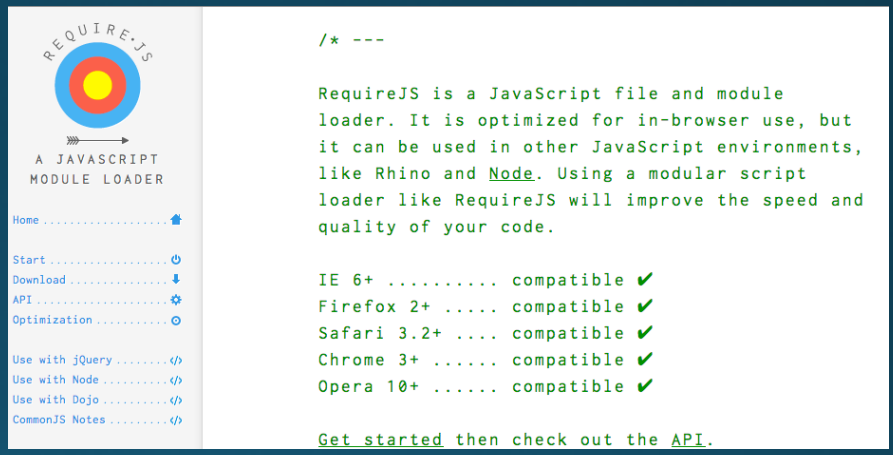
- Think of JS as laking make, include, etc.
- We need tools and libs to help us
- A popular one is Grunt http://gruntjs.com/
- Newer is Gulp - http://gulpjs.com/

Suggestion: Linting

- JsLint - http://www.jslint.com/
- JsHint - http://jshint.com
- Editors can give advice too (ex. WebStorm)
- Make part of build process?
  - https://github.com/gruntjs/grunt-contrib-jshint
- Trouble: If seems too strict then ignore

Suggestion: Add Modules Somehow

- Jury is out here since it is coming to native JS but …
- Require.js http://requirejs.org/
- You'll notice angular has modules too but is this a different issue?



Suggestion: Watch Code Complexity

- LOC, number of parameters in functions/methods, cyclomatic complexity, various other heuristics
- complexityReport.js https://github.com/philbooth/complexityReport.js and http://jscomplexity.org/
- jsmeter http://jsmeter.info/
- Possibility: Auto run of this on build?

Back to the Story

Diving In – Improve

- After your first clean-up pass some things are still going to make the linter/inspector unhappy
  - Many things will probably make you unhappy!
- Start first with simple renaming
  - Use refactor tools where possible!
  - Follow the coding guidelines strictly
  - For high level changes like major class or method names talk it over with the team

Diving In – Improve Contd.

- As you go you will find ugly things. If they aren't for this pass add a // TODO comment.
  - Consider making a new TODO style like // REFACTOR or // REWRITE or // REVIEW to organize the types of things found
- Next locate duplicates
  - See if you can condense things
  - Use tools if you can
- Next find large files, classes, and methods
  - See if you can break things into pieces – no adding or removing just breaking up
  - Use tools if you can

Diving In - Automate

- Now as you proceeded in your improving you probably have been repeating some mindless tasks over and over again, those are things you can automate

- Also as you have been doing the refactor you might wonder that will the app still works in the sanity case does it work beyond that?
  - It may not, but laying in all the tests first wouldn't have worked
  - Time to start laying in a few tests at a time

Diving In - Accelerate

- You need to accelerate the cycle of building and testing
- Two places to do this
  - In editor – see the play button
  - On build server (TeamCity)
- Your general expectation is that your team will need to do some localized testing, linting, etc. and then when they check in you might do some more significant testing

Diving In - Improve

- Now that you have made a few passes over the code and made it your own with some reformatting and basic refactoring you might finally have enough insight to start making some real decisions

- Questions
  - What is the state of the code base after the clean-up
    - The mess behind the mess?
    - Should we go look at the other repo, should we consider just taking pieces or start again?
    - Be very very very careful of the sunk cost fallacy
  - Does the framework make sense for what is being done?
    - Is it being leveraged? If not, can it be used or removed?
  - Where are the trouble spots? If we started making some moves where would we go first.
  - If you can do this maybe you can start planning out your first forward movement sprint

If you crash…

- It is possible that you move too fast and break things as you go
- Admit it, you did that to yourself. You have to move only as far and as fast as you can do safely. Undo until last saved checkpoint (maybe the start) and do again moving more cautiously
- Slow coding in unknown code bases seems to work better believe it or not!
  - Or at least the process of doing it that way is less risky