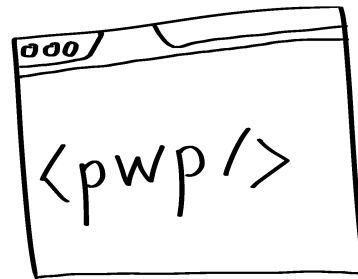


Practical Course: Web Development
Good Practice
Winter Semester 2016/17

Tobias Seitz



The content of this session mostly originates from this book:



Weniger Schlecht
Programmieren

Kathrin Passig & Johannes
Jander

<https://www.oreilly.de/buecher/120174/9783897215672-weniger-schlecht-programmieren.html>

“Any fool can write code that a computer can understand. Good programmers write code that humans can understand.”

Martin Fowler, “Refactoring”

Conventions & Style

Conventions

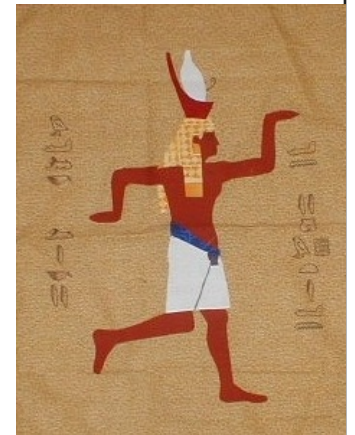
- Sticking to code-conventions helps maintaining a good atmosphere among team members
- Follow the conventions of
 - your chosen **language**: Inform yourself about commonly used coding style and try to adapt it.
 - your existing **team**: See what the others are doing, then do the same.
- Language conventions, e.g. JS style guides
 - AirBnB: <https://github.com/airbnb/javascript>
 - Google: <http://google.github.io/styleguide/jsguide.html>
 - @felixge NodeJS: <https://github.com/felixge/node-style-guide>

Conventions: Code Formatting

```
function getDivs() {  
}
```

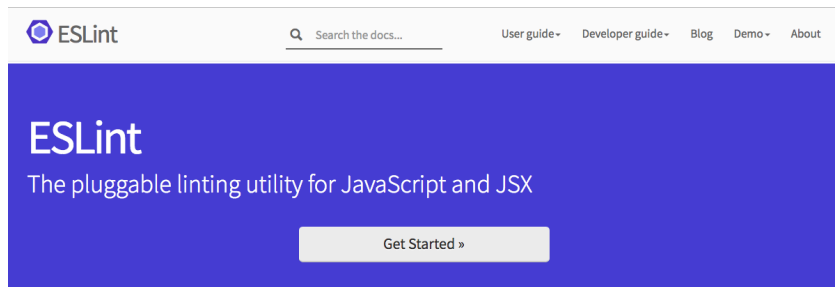
// versus

```
function getDivs (  
{  
}
```



Conventions: “Enforcement”

- Use a Linter (e.g. ESLint, JSHint)
- Configure your IDE / editor to **automatically** lint your code
- Configure **short-cuts** to reformat your code
- Share Lint-style with your team members, in the root of your repository.



The screenshot shows the ESLint website homepage. At the top, there is a navigation bar with the ESLint logo, a search bar, and links for 'User guide', 'Developer guide', 'Blog', 'Demo', and 'About'. Below the navigation bar is a large blue banner with the text 'ESLint' and 'The pluggable linting utility for JavaScript and JSX'. A 'Get Started' button is centered at the bottom of the banner.

Welcome

ESLint is an open source project originally created by [Nicholas C. Zakas](#) in June 2013. Its goal is to provide a pluggable linting utility for JavaScript.

Latest News

- [ESLint v3.11.1 released](#) 28 November 2016



The screenshot shows the JSHint website documentation page. At the top, there is a navigation bar with links for 'JSHint', 'About', 'Docs', 'Install', 'Contribute', and 'Blog'. On the right side, there is a search bar labeled 'Option name' and a 'Jump to docs' button.

Documentation

JSHint is a program that flags suspicious usage in programs written in JavaScript. The core project consists of a library itself as well as a CLI program distributed as a Node module.

More docs: [List of all JSHint options](#) · [Command-line Interface](#) · [API](#) · [Writing your own reporter](#) · [FAQ](#)

Basic usage

First, check out the [installation instructions](#) for details on how to install JSHint in your preferred environment. Both the command line executable and the JavaScript API offer unique ways to configure JSHint's behaviour. The most common usages are:

- [As a command-line tool](#) (via Node.js)
- [As a JavaScript module](#)

Regardless of your preferred environment, you can control JSHint's behavior through specifying any number of [linting options](#). In addition, JSHint will honor any directives declared within the input source code—see the [section on in-line directives](#) for more information.

Configuration

JSHint comes with a default set of warnings but it was designed to be very configurable. There are three main ways to configure your copy of JSHint: you can either specify the configuration file manually via the `--config` flag, use a special file `.jshintrc` or put your config into your projects `package.json` file under the `jshintConfig` property. In case of `.jshintrc`, JSHint will start looking for this file in the same

Example .editorconfig

```
# EditorConfig helps developers define and maintain consistent
# coding styles between different editors and IDEs
# editorconfig.org

root = true

[*]

# Change these settings to your own preference
indent_style = space
indent_size = 2

# We recommend you to keep these unchanged
end_of_line = lf
charset = utf-8
trim_trailing_whitespace = true
insert_final_newline = true

[* .md]
trim_trailing_whitespace = false
```


Example .jscsrc

```
{  
  "preset": "google",  
  "disallowSpacesInAnonymousFunctionExpression": null,  
  "disallowTrailingWhitespace": null,  
  "disallowMultipleVarDecl": false,  
  "maximumLineLength": false,  
  "excludeFiles": ["node_modules/**"]  
}
```

Example .jshintrc

```
{
  "node": true,
  "browser": true,
  "camelcase": true,
  "curly": true,
  "eqeqeq": true,
  "immed": true,
  "indent": 2,
  "latedef": true,
  "noarg": true,
  "quotmark": "single",
  "undef": true,
  "unused": true,
  "newcap": false,
  "globals": {
    "wrap": true,
    "unwrap": true,
    "app": true,
    "google": true,
    "zxcvbn": true,
    "HeatmapOverlay": true
  }
}
```

Adding a Linter to a Task Runner (Gulp)

```
var gulp = require('gulp');
var $ = require('gulp-load-plugins')();

// Lint JavaScript
gulp.task('lint', [], function() {
  return gulp.src([
    '*.js',
    '*.html'
  ])
  .pipe($.if('*.html', $.htmlExtract({strip: true})))
  .pipe($.jshint())
  .pipe($.jscs())
  .pipe($.jscsStylish.combineWithHintResults())
  .pipe($.jshint.reporter('jshint-stylish'))
});
```

Break-Out: Prettify ugly-code.html

ugly-code.html-script-0

```
line 2 col 13 Strings must use singlequote.  
line 2 col 9 Invalid quote mark found  
line 4 col 33 One (or more) spaces required before opening brace for block expressions  
line 6 col 2 Expected indentation of 2 characters  
line 7 col 2 Expected indentation of 2 characters  
line 9 col 0 Expected indentation of 0 characters  
line 11 col 2 Expected indentation of 2 characters  
line 11 col 2 One space required after "if" keyword  
line 12 col 4 Expected indentation of 4 characters  
line 13 col 30 Missing semicolon after statement  
line 13 col 31 Missing semicolon.  
line 14 col 2 Expected indentation of 2 characters  
line 14 col 1 } and else should be on the same line  
line 15 col 2 Expected indentation of 2 characters  
line 16 col 29 Missing semicolon.  
line 16 col 4 Expected indentation of 4 characters  
line 16 col 28 Missing semicolon after statement  
line 17 col 2 Expected indentation of 2 characters  
line 18 col 2 Expected indentation of 2 characters
```

▲ 19 warnings

[09:12:06] Finished 'lint' after 1.66 s

Break-Out: Prettify ugly-code.html

- Open your favorite code editor
- Look for the Keyboard short-cut that allows you to automagically reformat code
- Open the file ugly-code.html from the course material repository (github / gitlab)
- Make the code pretty!

Naming

DOs

- function names should describe what the function does, not how it does it
`validatePassword()`
- Private variables are usually prefixed with an underscore
`_untouchableThing`
- Include units:
`delaySeconds`
- Function naming structure
`verbAdjectiveNounDatatype`

DONTs

- don't use any other abbreviations than `num`, `post`, `len`, `max`, `min`, `temp`, `val`.
- don't be funny and use incomprehensible names
`superCat = 1`

Language

- German / Native Language:
 - easier to find variable names
 - fewer typos
 - bad English can negatively influence the readability of your code

- .. but: Do yourself and anyone else a favor and use **English!**
 - statements and other syntax in English → otherwise weird mix
 - the community in your country might not be as big as the whole world. → asking for help is easier with code snippets.

Comments & Documentation

“Always code as if the guy who ends up maintaining your code is a violent psychopath who knows where you live.”

(origin unclear, probably one of: John Woods / Martin Golding / Rick Osborne)

Comments

- “Mangelhafter, gründlich kommentierter Code ist mangelhaftem unkommentiertem Code klar vorzuziehen (Passig & Jander)
- “Don’t document bad code – rewrite it” (Kernigham & Pike)
- When to comment:
 - If there’s unexpected behavior
 - if you “temporarily” comment out code
 - if you failed with a solution: say in the comment how you failed and why
 - if you found a good solution that looks scary / complicated
 - if you assume things might break
 - if you paste code from other sources

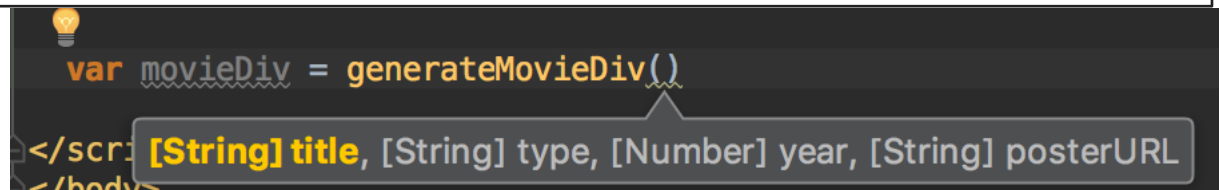
Problematic Comments / Heuristics

- If you need more than 3-4 sentences to describe what a function does, this could mean it does too much
- Comments refer to parameters / variables that were renamed.
- Comment includes non-standard abbreviations
- Comment should actually be the commit message (or vice versa)
- Comment does not address the readership

JSDoc - <http://usejsdoc.org/>

- Similar to JavaDoc. Addresses a central “problem” of JavaScript: dynamic typing.
- Proper JSDoc allows editors to display live-help when you try to use the function.

```
/**  
 * generates a div that displays a movie poster (or placeholder)  
 * a title, type and the year.  
 * @param {String} title of the movie  
 * @param {String} type of the move (movie/series/episode)  
 * @param {Number} year  
 * @param {String} posterURL  
 * @return {jQuery|HTMLElement} with class "movie flexChild card"  
 */  
function generateMovieDiv(title, type, year, posterURL) {}
```



TODO

- If you need to postpone a task, mark it with a TODO
- Example
*// **TODO reformat this code***
- Usually, code editors allow you to scan and find TODOs, or even highlight them
- Before you commit: resolve TODOs (not always possible, but at least try)
- Important:
Stick to TODO / FIXME / CHANGEBACK / XXX / !!!!!!!
- Don't: TODO Fix this.

Truck Factor

How many of your team would have to be run over by a **truck** to make the project stand still?

https://commons.wikimedia.org/wiki/File:Kenworth_W900_semi_in_red.jpg

Solutions to Increase the Truck Factor

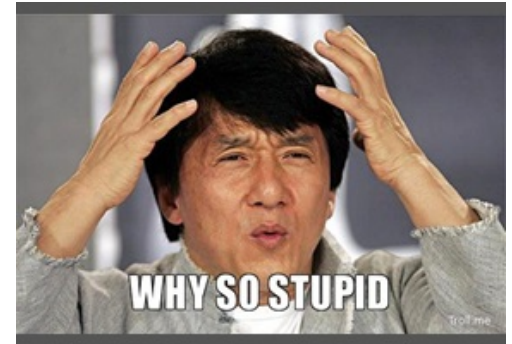
- Issue Tracker (built into GitLab)
- Responsibilities: Who is responsible for what?
- Agile Process: everybody at least has a high-level understanding of what's going on in other “departments”
- Good Documentation
- Code Reviews / Extreme Programming
- Decide up front who your replacement is.

Debugging

Debugging Approaches (Sample)

- Usually useful:
 - Debugger to pause program during execution
 - Rubber-Duck Debugging <http://www.developerduck.com/>
 - Other inspection tools (HTML, CSS)
- Sometimes useful:
 - Console.log Debugging (Printline Debugging)
 - Stackoverflow-Debugging (next slides)
 - Sophisticated logging (multiple levels)
 - Code reviews / pair programming / Think aloud
- After all: Take a break. Get some sleep.

Common Stupidities (we've all been there)



- The file that I edit, is not the file I execute / view.
- I use a feature that is unsupported by older browsers (<http://caniuse.com/>)
- I accidentally use a variable name for a local variable that collides with higher-scope variable.
- I treat undefined, 'undefined', null and false as the same thing (hint: they are not)
- Off-By-One Errors.
“There are two hard problems in computer science: cache invalidation, naming things, and off-by-one errors”
(Leon Bambrick)

Checklist for when to get Help

- Did you copy paste the error-message into a search engine?
- Did you use the English version of the error message?
- Did you lint your code?
- Did you enable error messages in your programming environment?
- Did you check the official documentation for what you are trying to achieve?
- Did you look on stackoverflow.com?



Stackoverflow Debugging

1. What is your goal?
2. What have you done?
3. What have you tried?
4. What were the results?
5. What did you expect?

If you can honestly answer these five questions and have not found the answer during this process, go and ask the question on stack overflow.

Bad Code

A Bad Programmer's 7 Arguments

1. Nobody else is going to see my code
2. The software isn't going to be used by anyone else but me.
3. I'll redo it properly later.
4. It's a complex problem. I can only solve it with 8 stacked loops.
5. I simply remember not to enter that.
6. I'm going to uncomment that again.
7. It's just a small project.

Bad Code Heuristics – Part 1

- Files too big
- Function bodies too long
- High indentation level
- Control statements with more than 5 checks
- Magic Numbers
- complex arithmetic without a comment
- Global variables
- Code that introduces a „hack“ to make things work.

Bad Code Heuristics – Part 2

- Third party functions are implemented again
- Inconsistent code style
- Functions with more than 5 parameters
- Code duplication / Redundancy
- Suspicious file names
- Reading labyrinth (instead of top to bottom)
- Many methods and member variables
- Old, commented out code blocks
- Suspicious keyboard sounds.

Break Out: Past Behavior

- Step 1: From the things you have heard about today, what have you done “wrong” in the past?
- Step 2: Use a style guide and spot things that you have done wrong in the past.

Hands-On: Refactor Foreign Code

- Clone / Fork this repository:
<https://github.com/MIMUC-MMN/assignments-16-17>
- pick one of the folders, e.g.
04 - jquery basics or 06 - jquery ajax
- Take a look of the code in there and try to
 - understand it
 - write down things that you don't understand
 - improve it
 - document what you improved.
- Time frame: 15 minutes, discussion afterwards.

You don't learn to walk by following rules. You learn by doing, and by falling over.
(Richard Branson)

Links 'n' Stuff

- <http://jscs.info/overview>
- <http://jshint.com/docs/>
- <http://eslint.org/>
- <http://editorconfig.org/>
- <https://blog.codinghorror.com/new-programming-jargon/>
- <http://www.journaldev.com/240/my-25-favorite-programming-quotes-that-are-funny-too>