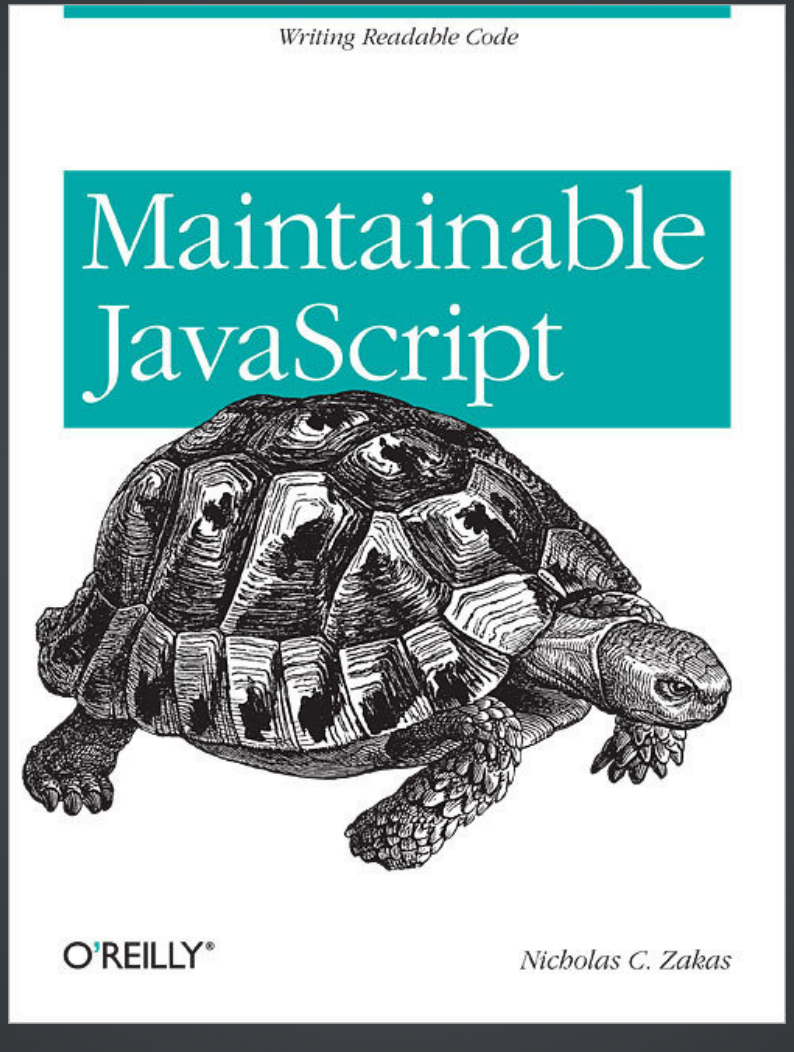
Learning Good Developer Habits

what kind of habits are we talking about?

habits that support writing maintainable code:

- following a coding style guide
- employing best practices and coding conventions
- linting
- writing tests

The topic of writing maintainable code was inspired by Nicolas Zakas, currently the Principal Architect at Box, but more commonly known for his books:


Writing Readable Code
Maintainable JavaScript
O'REILLY®
Nicholas C. Zakas

why should we care?

You should care because you'll spend most of your development time maintaining code.

what's the difference between writing new code and maintaining code?

"When you open your editor and the file is blank, that's when you're writing new code. If you sit down and there's already something there, you're maintaining code."

- Nicholas Zakas

how long should maintainable code last?

"Maintainable code works for five years without major changes."

- Nicholas Zakas

characteristics of maintainable code:

- Intuitive
- Adaptable
- Extendable
- Debuggable
- Testable

approaching writing maintainable code breaks down into two main areas of focus:

- style guidelines/code formatting & appearance
- code conventions/best practices

style guidelines are a type of code convention aimed at the layout of code within a file

The primary tool for implementing style guidelines is the style guide.

"Programs are meant to be read by humans and only incidentally for computers to execute"

- Donald Knuth

Some examples of public style-guides used in the industry:

- http://javascript.crockford.com/code.html
- Google
- http://docs.jquery.com/JQuery_Core_Style_Guidelines
- https://github.com/rwaldron/idiomatic.js/
- https://github.com/airbnb/javascript

Be prepared for disagreement when discussing a style guide with your team. Once established, style guidelines allow the team to work at a much higher level.

When a team embraces a style guide, all code looks the same. This allows any developer to work on any file regardless of who wrote it. There's no more time spent on reformatting or deciphering the logic of the file. Consequently, errors become more obvious.

```
if (wl && wl.length) {
                for (i = 0, l = w.length; i < l; ++i) {
            p = wl[i];
            type = Y.Lang.type(r[p]);
            if (s.hasOwnProperty(p)) { if (merge && type == 'object') {

        Y.mix(r[p], s[p]);
} else if (ov || !(p in r)) {
                    r[p] = s[p];
                }
            }
        }
    }
```

Basic Formatting

Indentation rules: two schools of thought:

tabs... or spaces?

just pick one and be consistent!

statement termination:

```
// bad
var name = "James"

function sayMyName() {

    alert(name)
}

// good
var name = "James";

function sayMyName() {

    alert(name);
}
```

```
// open blocks on the same line
function returnSomething() {

    return
    {
        name: "James",
        title: "instructor"
    }
}

// because this is how the parser will see it
function returnSomething() {

    return;
    {
        name: "James",
        title: "instructor"
    };
}
```

```
// works correctly even without semicolons
function returnSomething() {

    return {
        name: "James",
        title: "instructor"
    }
}
```

line length

"The less code on one line, the less likely you'll encounter a merge conflict."

https://twitter.com/slicknet/statuses/169903570047614977

line breaking: indent two levels

```
// bad
aLongFunction(this, element, document, "Are you sure?", false, 3.14, { foo: bar },
    navigator);

// bad
aLongFunction(this, element, document, "Are you sure?", false, 3.14, { foo: bar}
        , navigator);

// good
aLongFunction(this, element, document, "Are you sure?", false, 3.14, { foo: bar },
        navigator);
```

Blank lines

```
$(function () {
    var firstParam = "Jayson";
    var secondParam = "Potter";
    // validate parameters and build result object
    function myFunc (param) {
        if (!myFunc.cache[param]) {
            var result = {};
            result.title = "teaching assistant";
            result.salary = "10000";
            myFunc.cache[param] = result;
        }
        return myFunc.cache[param];
    }
    // test out new function
    myFunc.cache = {};
    myFunc(firstParam);
    myFunc(secondParam);
    sampleObj = myFunc(firstParam + secondParam);
    // iterate through object properties
    for (var prop in sampleObj) {
        discoveredProps.push(sampleObj[prop]);
    }
});
```

```
$(function () {

    var firstParam = "Jayson";
    var secondParam = "Potter";

    // validate parameters and build result object
    function myFunc (param) {

        if (!myFunc.cache[param]) {
            var result = {};
            result.title = "teaching assistant";
            result.salary = "10000";
            myFunc.cache[param] = result;
        }

        return myFunc.cache[param];
    }

    // test out new function
    myFunc.cache = {};
    myFunc(firstParam);
    myFunc(secondParam);

    sampleObj = myFunc(firstParam + secondParam);

    // iterate through object properties
    for (var prop in sampleObj) {
        discoveredProps.push(sampleObj[prop]);
    }
});
```

Naming

variables and functions: use camelCase

```
var myName;
var integerVariable;
var probablyTooLongOfAVariableName;
```

variables: should begin with a noun (or sometimes an adjective)

```
// bad
getCount = 10;
isFound = "James";

// good
var count = 100;
var myName = "James";
var found = true;
```

functions should begin with verbs (describe an action)

```
// bad
function theDescription() {

    return description;
}

// good
function getDescription() {

    return description;
}
```

some common conventions for that verb:

can	returns a boolean
has	returns a boolean
is	returns a boolean
get	returns a non-boolean
set	used to save a value

```
if (isFinished()) {
    setPriority();
}

if (getTeachingAssistant() === "Jayson Potter") {
    payAttention();
}
```

Constants

```
var MAX_GRADE = 1500;
var URL = "https://canvas.uw.edu/courses/985046";
```

Constructors

```
function Student(firstName, lastName) {

    this.firstName = firstName;
    this.lastName = lastName;
};

Student.prototype.assignGrade(grade) {

    this.grade = grade;
}

var sampleStudent = new Student("Jayson Potter");
sampleStudent.assignGrade('CNC');
```

null - where it should be used:

- to initialize a variable that may later be assigned an object value
- to compare against an uninitialized variable that may or may not have an object value
- to pass into a function where an object is expected
- to return from a function where an object is expected

```
// good
var student = null;

// good
function getStudent() {

    if (found) {
        return new Student("Jayson");
    } else {
        return null;
    }
}

// good
var student = getStudent();
if (student !== null) {
    assignHomework();
}

// bad
var student;
if (student !== null) {
    assignHomework();
}

// bad
function assignHomework(week, module) {

    if (module !== null) {
        addToQueue();
    }
}
```

undefined - avoid using in code

This value is meant to indicate a variable that has not yet been assigned a value. However, when using typeof with a variable to check this state, it can cause confusion:

```
// both console.log calls will return "undefined"
var student;
console.log(typeof student);
console.log(typeof instructor);

// initialize variables rather than checking for undefined
var student = null;
console.log(student === null);
```

Comments

single-line comments:

- goes on it's own line, explaining the line following the comment
- should be preceded by an empty line
- same indentation level as the code it explains
- avoid same-line comments
- most editors use for commenting out blocks of code

```
// bad
if (found) {
    // the treasure has been found, proceed to dig!
    beginDigging();
}

// bad
if (found) {

// the treasure has been found, proceed to dig!
    beginDigging();
}

// bad* (variables)
if (found) {
    beginDigging(); // the treasure has been found, proceed to dig!
}

// good
if (found) {

    // the treasure has been found, proceed to dig!
    beginDigging();
}
```

multiline comments

- use when a single comment will span more than one line
- should be preceded by a blank line
- should immediately precede the code it explains
- should have the same indentation level as that code

```
/*
 * here is an example of a multiline comment
 * that uses the Java style multiline comment pattern
 * and while not required, makes identifying and reading
 * the comment easier
 */
(function () {

    // a short block of code
    leaveComment({
        verbose: true,
        date: "March 31, 2015"
    });
}());

/* this is also an example of a multiline comment that is also
   acceptable - while it doesn't use the Java style, it's still
   better than using the single-line comment format for multiple lines */

(function () {

    // a shorter block of code
    exitDiscussion(true);
}());
```

when to comment:

- when something isn't clear
- when there is difficult to understand code
- when code could appear to have an error
- when using browser-specific hacks

```
// bad

// initialize count
var timeout = 3000;

// good

// timeout must be at least 3 seconds to handle latency
var timeout = 3000;
```

statements and expressions

placement of braces:

```
// bad
if (found)
    startDigging();

// bad
if (found) startDigging();

// bad
if (found) { startDigging(); }

// good
if (found) {
    startDigging();
}

// omitting braces can cause subtle bugs
if (found)
    startDigging();
    beginSmelting();
```

brace alignment:

```
// bad
if (found)
{
    startDigging();
}
else
{
    leaveNow();
}

// good
if (found) {
    startDigging();
} else {
    leaveNow();
}
```

for loops:

```
var scores = [97, 82, 100, 86, 91, 70],
var i, l;

// cache the length rather than re-calculating with each loop iteration
for (i = 0, l = scores.length; i < l; i += 1) {
    submitScores(scores[i]);
}
```

for-in loops:

```
var prop;

// remember to check for own properties to filter out prototype properties
for (prop in obj) {
    if (obj.hasOwnProperty(prop)) {
        cacheValue(obj[prop]);
    }
}
```

variable declarations:

```
// bad
var name = "James"
    , title = "instructor"
    , city = "Seattle"
    , present = true
    , color = "green";

// problematic
var name = "James",
    title = "instructor,
    city = "Seattle";
    present = true,
    color = "green";


// preferred "
var name = "James";
var title = "instructor";
var city = "Seattle";
var present = true;
var color = "green";
```

function declarations: avoid hoisting confusion

```
// bad
startDigging();

function startDigging() {
    alert("I'm digging!");
}

// declare functions before using
function startDigging() {
    alert("I'm digging!");
}

startDigging();
```

spacing: functions, blocks, etc.

```
// function declarations have no space between the name and parens
function startDigging() {
    alert("I'm digging!");
}

// function expressions do (both have a space between closing paren and opening brace
var startDigging = function () {
    alert("I'm diggin!");
}

// blocks have a space between block keyword and opening paren
while (found) {
    found = digMore();
}

for (i = 0, l = treasures.length; i < l; i += 1) {
    digForTreasure(treasures[i]);
}
```

strict mode

```
// bad - global strict
"use strict";

function startDigging() {
    alert("I'm digging!");
}

// good - function strict
function startDigging() {

    "use strict";

    alert("I'm digging!");
}

// good - if you don't want to have to repeat it
(function () {

    "use strict";

    function startDigging() {
        alert("I'm digging!");
    }
    function stopDigging() {
        alert("I'm not digging anymore");
    }
}());
```

equality and type coercion

```
// always avoid using "==" to determine equality - actually avoid it altogethe
var count = 5;
var guess = "5";

if (count == guess) {
    loseEverything = true;
}
```

Programming Practices

These are another type of code convention, but whereas style guidelines are concerned with the appearance of code, programming practices are concerned with the outcome.

loose coupling of UI layers

- HTML is used to define the data and semantics of the page
- CSS is used to style the page
- JavaScript is used to add behavior to the page, making it interactive

what's meant by loose coupling?

Tight coupling of components is when one component has direct knowledge of another in such a way that a change to one of the components often necessitates a change to the other component.

Loose coupling is achieved when you're able to make changes to a single component without making changes to other components.

what does this mean for us?

For front-end development, this means keeping JavaScript and CSS out of our markup, and keeping our CSS and markup out of our JavaScript. This is also considered separation of concerns, where css and JavaScript lives in their own files.

keeping CSS out of JavaScript

```
// bad
element.style.color = "blue";
element.style.left = "20px";
element.style.top = "50px";

// instead this information should live in a class that lives in a css file
.dialog {
    color: blue;
    left: 20px;
    top: 50px;
}

// good - using JavaScript to manipulate the class if the style change is dynamic
element.className += " dialog";

// or with HTML5
element.classList.add("dialog");

// or with jQuery
$(element).addClass("dialog");
```

keeping JavaScript out of HTML

```
// bad
<button onclick="startDigging()" id="dig-btn">Start digging!</button>

// good - JavaScript in a separate file
function startDigging() {
    alert("I'm digging!");
}

document.load = function () {

    var btn = document.getElementById("dig-btn");

    btn.addEventListener("click", startDigging, false);
}

// or, with jQuery
$(function () {
    $('#dig-btn').on('click', startDigging);
});
```

keeping HTML out of JavaScript

```
// bad
var div = document.getElementById("zip_code_field");
div.innerHTML = "<p><span class="error_state">Error</span><span>Invalid zip code.</span></p>";
```

Markup constructed in your code in this fashion is a bad idea for a few reasons:

- complicates tracking down text and structural issues - makes debugging difficult
- maintainability - when you need to change text or markup, you want to be able to do it in one place
- less error prone to edit markup than JavaScript

Loading markup and text from the server, or using templates is more maintainable and flexible.

Avoid globals - the problems:

- naming collisions - the likelihood increases the more globals are introduced, especially problematic when dealing with 3rd party scripts
- code fragility - globals tightly couple to the environment - changing the environment, the code is likely to break
- difficulty testing - functions relying on globals means you have to set them in your testing environment

Globals can be managed through namespaces and modules - something we'll address later.

Event handling

Event handlers should be small and only handle the event

```
// bad
function handleClick(event) {
    var dialog = document.getElementById('dialog');

    dialog.style.left = event.clientX + 'px';
    dialog.style.top = event.clientY + 'px';
    dialog.className += ' reveal';
}

// better, but still bad
function handleClick(event) {
    showDialog(event);
}
function showDialog(event) {
    var dialog = document.getElementById('dialog');

    dialog.style.left = event.clientX + 'px';
    dialog.style.top = event.clientY + 'px';
    dialog.className += ' reveal';
}
```

Don't pass the event object around

```
// still bad - showDialog now has a dependency on entire event object
function handleClick(event) {

    showDialog(event);
}
function showDialog(event) {

    var dialog = document.getElementById('dialog');

    dialog.style.left = event.clientX + 'px';
    dialog.style.top = event.clientY + 'px';
    dialog.className += ' reveal';
}

// good - showDialog has been de-couple from the event object dependency
function handleClick(event) {

    showDialog(event.clientX, event.clientY);
}
function showDialog(x, y) {

    var dialog = document.getElementById('dialog');

    dialog.style.left = x + 'px';
    dialog.style.top = x + 'px';
    dialog.className += ' reveal';
}
```

Don't modify objects you don't own

```
// don't add new methods
Array.prototype.holla = function () {

    alert('Holla!');
};

// don't override methods
Array.prototype.toString = function () {

    alert('Holla at ya!');
};
```

Avoid null comparisons for reference values:

```
// null is usually not enough info to draw a safe conclusion
takeSteps = function (stuff) {

    if (stuff !== null) {
        stuff.sort();
        stuff.forEach(function (doStuff) {
            // do stuff
        });
    }
};

// instead use instanceof
takeSteps = function (stuff) {

    if (stuff instanceof Array) {
        stuff.sort();
        stuff.forEach(function (doStuff) {
            // do stuff
        });
    }
};
```

Separate configuration data

any hard-coded value in an application.

hard-coded strings are a maintenance issue and a source of bugs

```
function validate(value) {

    if (!value) {
        alert('Invalid value');
        location.href = '/errors/invalid.html';
    }
}

function toggleSelected(element) {

    if (hasClass(element, 'selected')) {
        removeClass(element, 'selected');
    } else {
        addClass(element, 'selected');
    }
}
```

configuration data should be externalized

```
var configData = {
    MSG_INVALID_VALUE = "invalid value",
    URL_INVALID = "/errors/invalid.html",
    SELECTED_CLASS = "selected"
};

function validate(value) {

    if (!value) {
        alert(configData.MSG_INVALID_VALUE);
        location.href = configData.URL_INVALID;
    }
}

function toggleSelected(element) {

    if (hasClass(element, configData.SELECTED_CLASS)) {
        removeClass(element, configData.SELECTED_CLASS;
    } else {
        addClass(element, configData.SELECTED_CLASS);
    }
}
```

Linting

Linting tools are just programs that analyze code for potential errors. They can help identify problematic styles and patterns in your code.

linting tools

- JSLint easy to use, but configurability is limited
- JSHint configurable, but not very customizable
- ESLint configurable, customizable - requires some setup

ESLint becoming an industry standard

ESLint is the linting tool we use in this course.

ESLint will help you create a configuration file:

```
eslint --init
```

Don't Repeat Yourself

Making your code D.R.Y. means avoiding repeating the same code. Repeated code makes for a maintenance headache, is inefficient, and, in the world of web development, means more code which = worse performance. Common code should be refactored into single modules/functions

Single Responsibility Principle

Every module/class/function should have responsibility over a single part of the functionality provided by the software and that responsibility should be encapsulated by the unit (module/class/function).

You Aren't Gonna Need It

YAGNI: don't add functionality until it's necessary. Doing so leads to wasting time on things you may discard or you may eventually take a different approach to - this principle comes from extreme programming and it's behind the practice of "do the simplest thing that could possibly work".