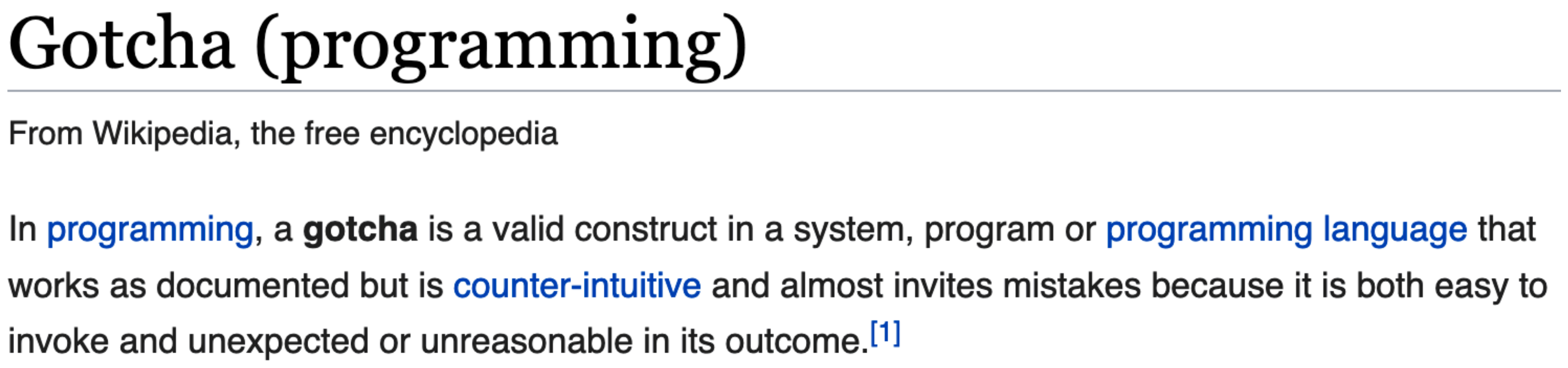
CS2003 W10
JS Gotchas (and Testing)

Özgür Akgün
ozgur.akgun@st-andrews.ac.uk



Gotcha (programming)

From Wikipedia, the free encyclopedia

In programming, a **gotcha** is a valid construct in a system, program or programming language that works as documented but is counter-intuitive and almost invites mistakes because it is both easy to invoke and unexpected or unreasonable in its outcome.[1]

https://en.wikipedia.org/wiki/Gotcha_(programming)

- Every language has these, but JavaScript seems to have a bit more than usual…
- I claim
- Decide for yourselves after this lecture…

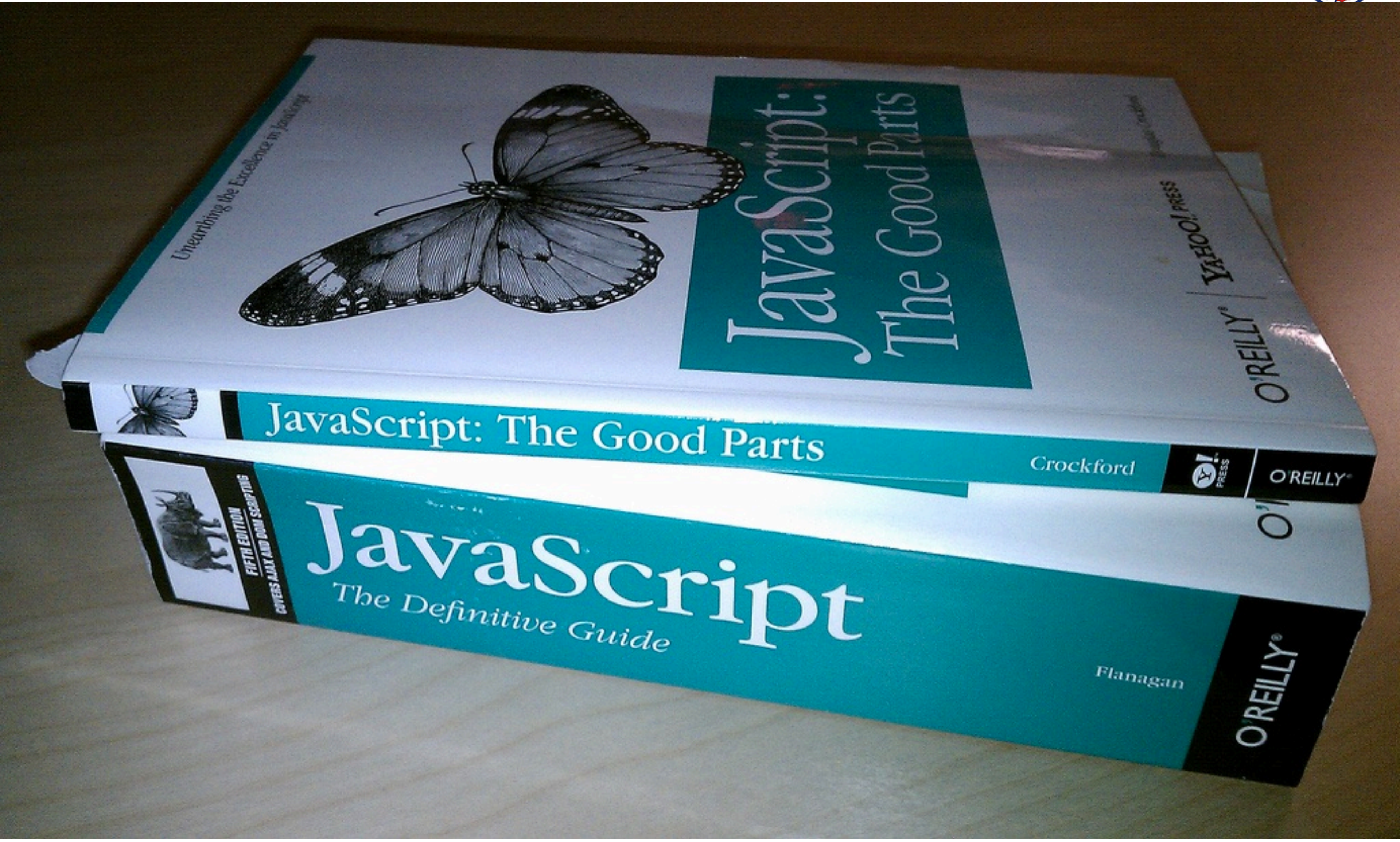

Unearthing the Excellence in JavaScript
JavaScript: The Good Parts
YAHOO! PRESS
O'REILLY
JavaScript: The Good Parts
Crockford
O'REILLY
FIFTH EDITION
COVERS AJAX AND DOM SCRIPTING
JavaScript
The Definitive Guide
Flanagan
O'REILLY

Image from: https://blog.klipse.tech/javascript/2016/09/21/valueOf-js.html

JavaScript good - JavaScript bad

- We focused on the good parts of JavaScript so far
  - Can run in the browser, on the server
  - First class functions
  - Good libraries
- Anything “bad” about it?
  - Dynamic typing? Weak typing?
  - Scoping
  - Counter-intuitive choices in the functionality of the standard library
    - Trying to do too much, maybe?

JavaScript good - JavaScript bad

- JS offers enough good properties
  - Hence used very heavily in the industry
- To deal with the “bad” parts
  - Test, test, test
  - Even more than you might test a program written in, say, Java

Resources

- JavaScript the Good Parts by Douglas Crockford
    - Appendices A & B
- Various bits I collected over the years
- Just a fun thing for you to watch
    - https://www.destroyallsoftware.com/talks/wat

Testing

- There are a few unit testing libraries for JS
- I'll show you JEST
  - Quite similar to JUnit
  - Property based unit testing
- Think about
  - Testing with randomized inputs
  - Full system testing
    - Automated? For a web application?

JEST

```
// file: sum.js

function sum(a, b) {
    return a + b;
}
module.exports = sum;
```

```
// file: sum.test.js

const sum = require('./sum’);
test('adds 1 + 2 to equal 3', () => {
    expect(sum(1, 2)).toBe(3);
});
```

```
// file: package.json

{
    "scripts": {
        "test": "jest”
    }
}
```

- npm install jest
- npm run test
- See: https://jestjs.io

Global variables

- 3 ways to define them
  - var foo = value;
  - window.foo = value; // available as ‘foo’ as well
  - foo = value; // forgot to initialise?

Scope

- C-like syntax, different scoping rules

```
for (var i = 0; i <= 10; i++) {
    j = i*i
    console.log(i,j)
}

console.log(i)        // still defined???
console.log(j)        // this one too.
```

Semicolon insertion

- Are these the same?
- If not, what's the difference?

- The one on the left returns undefined…

```
return
{
    status: true
};
```

```
return {
    status: true
};
```

Reserved words

- `abstract boolean break byte case catch char class const continue debugger default delete do double else enum export extends false final finally float for function goto if implements import in instanceof int interface long native new null package private protected public return short static super switch synchronized this throw throws transient true try typeof var volatile void while with`

- So many! But most not actually used.

Reserved words

- How many of these are legal? None, some, all?

```
var method;
var class;
object = {box: value};
object = {case: value};
object = {'case': value};
object.box = value;
object.case = value;
object['case'] = value;
```

Reserved words

- 5/8, apparently…

```
var method;                    // ok
var class;                     // illegal
object = {box: value};         // ok
object = {case: value};        // illegal
object = {'case': value};      // ok
object.box = value;            // ok
object.case = value;           // illegal
object['case'] = value;        // ok
```

Reserved words

- “They cannot be used to name variables or parameters.
- When reserved words are used as keys in object literals, they must be quoted.
- They cannot be used with the dot notation, so it is sometimes necessary to use the bracket notation instead.”

```
var method;                  // ok
var class;                   // illegal
object = {box: value};       // ok
object = {case: value};      // illegal
object = {'case': value};    // ok
object.box = value;          // ok
object.case = value;         // illegal
object['case'] = value;      // ok
```

parseInt

- What does this return?
  - `parseInt("230 miles")`

-
```
parseInt("230 miles") / parseInt("5 hours")
> 46
```

NaN

1. `- '2'`	`-2`	
2. `+ '0'`	`0`	
3. `+ 'oops'`	`NaN`	
4. `typeOf NaN`	`"number"`	
5. `isNaN(NaN)`	`true`	makes sense
6. `isNan(0)`	`false`	it is indeed a number
7. `isNaN('oops')`	`true`	string, not a number, ok
8. `isNaN('0')`	`false`	???

setTimeout calls eval!

- `eval` is bad
  - Let's you convert a string to code, and run it
  - Code injection
- `setTimeout("console.log('test')", 1000);`
- setTimeout implicitly calls eval :(

How big is null?

1. `null >= 0`	`true`
2. `null <= 0`	`true`
3. `null == 0`	`false`
4. `null != 0`	`true`
5. `null > 0`	`false`
6. `null < 0`	`False`
7. `null === 0`	`false`
8. `null !== 0`	`true`

The way list.map works

From: https://medium.com/dailyjs/parseint-mystery-7c4368ef7b21

- `['1', '7', '11'].map(x => x + 1)`
  `> ['11', '71', '111']`
- Not surprising… What about:
- `['1', '7', '11'].map(parseInt)`
  - A: `[1, 7, 11]`
  - B: Something else
  - How about `[1, NaN, 3]`???

The way list.map works

- ['1', '7', '11'].map(console.log)
  - 1 0 ['1', '7', '11']
  - 7 1 ['1', '7', '11']
  - 11 2 ['1', '7', '11']
- `parseInt` takes 2 arguments. Second (optional) is radix/base
- Call `parseInt` with those 3 arguments…
- It ignores the third…
- We get:
  - `parseInt(1, 0) = 1`
  - `parseInt(7, 1) = NaN`
  - `parseInt(11,2) = 3`

Come on, a last one

- How many of these are true? None, some, all?
  - `[] + [] == []`
  - `[] + {} == {}`
  - `{} + [] == {}`
  - `{} + {} == {}`
  - All but the last one - NaN
- Using ===?
  - None are true!
- See Examples/web/gotchas/plus.js

Take away message?

- Try to write code that is
  - Easy to understand – No guesswork!
  - Easy to convince yourself that it is correct
  - Easy to test
- Write tests!
- These are good guidelines in general and for JS

Resources

- JavaScript the Good Parts by Douglas Crockford
  - Appendices A & B

- Just a fun thing for you to watch
  - https://www.destroyallsoftware.com/talks/wat

- There is more!
  - A long list here

- Don't forget to test your code!
  - https://jestjs.io

Wrapping up – thanks!

- This is our last lecture for the Web stream
- Programming with JavaScript for the web
  - HTML
  - CSS
  - Async programming, callbacks, promising
  - AJAX, Websockets
  - Frameworks and component-based architecture
- Had to deal with remote working issues :(
  - nginx proxies…
- This is an introductory course, much more to learn for the interested student