

# JavaScript: the language of browser interactions

**Claudia Hauff**

TI1506: Web and Database Technology

[ti1506-ewi@tudelft.nl](mailto:ti1506-ewi@tudelft.nl)



**Densest Web lecture  
of this course.**

**Coding takes time.**

**Be friendly with  
Codecademy & Co.**

# At the end of this lecture, you should be able to ...

- **Employ** OO principles in JavaScript coding
- **Explain** the principle of callbacks
- **Write** interactive Web applications based on click, mouse and keystroke events
- **Translate** jQuery-based code into jQuery-less code

# Chapter 4 of the Web course book

- How to include JavaScript in your Web app
- **Essential JavaScript built-in types** & control structures
- How to **declare** variables & functions
- The purpose of `console.log()`
- How to work with **arrays**
- How to use basic **jQuery** features

# JavaScript's reputation

- Until fairly recently it was considered more of a toy language
- Today: (most) important language of the **modern Web stack**
  - **Tooling** has vastly improved (debuggers, testing frameworks, etc.)
  - JavaScript **runtime engines** are efficient (especially **V8**)
  - JavaScript tracks **ECMAScript**

# A language in flux

ES5 6 2016+ next intl non standard compatibility table

Sort by:  Show obsolete platforms:  Show unstable platforms:

Legend: ■ V8 ■ SpiderMonkey ■ JavaScriptCore ■ Chakra ■ Other

Feature size:  Minor difference (1 point)  Small feature (2 points)  Medium feature (4 points)  Large feature (8 points)

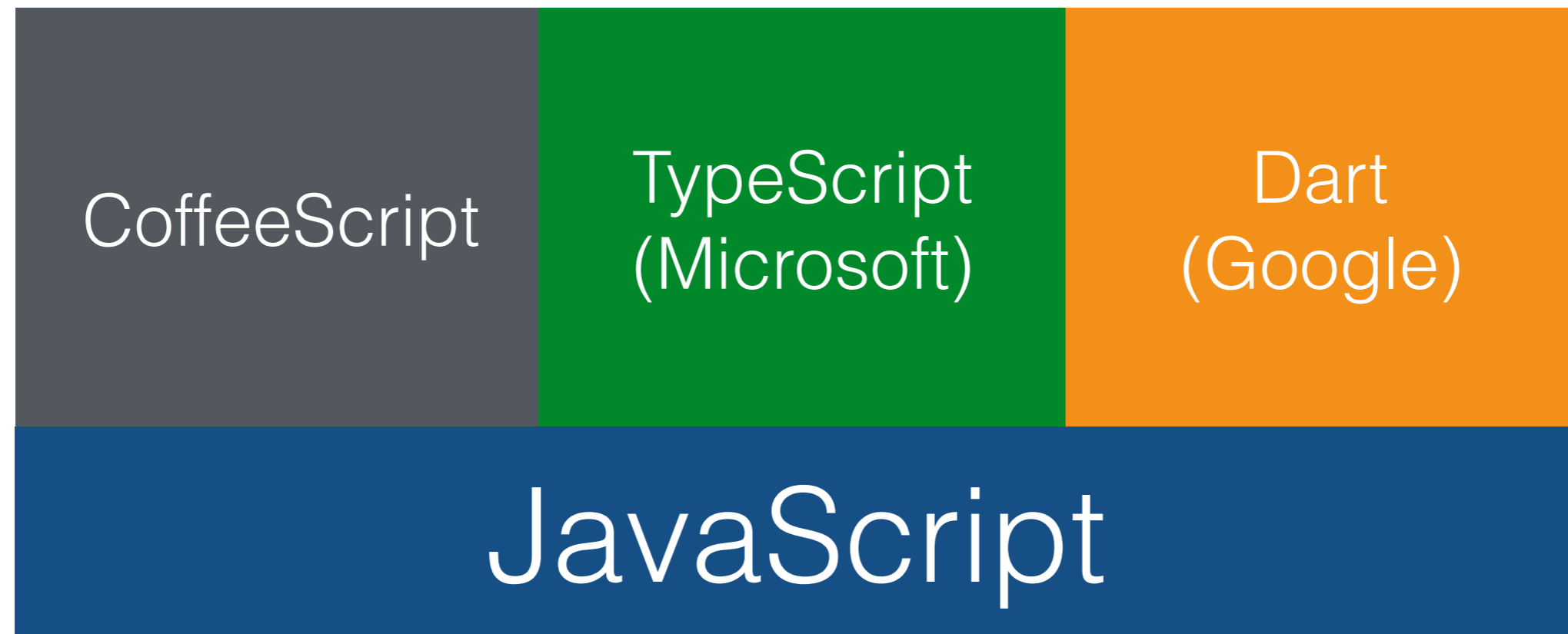
Feature name	Current browser	Compilers/polyfills							Desktop browsers														Safari				
		Traceur	Babel + core-js (2)	Closure	TypeScript + rollup	es7-shim	IE 11	Edge 15	Edge 16	Edge 17 Preview	FF 52 ESR	FF 56	FF 57	FF 58 Beta	FF 59 Nightly	CH 61 OP 48 (1)	CH 62 OP 48 (1)	CH 63 OP 50 (1)	CH 64 OP 51 (1)	SF 10.1	SF 11	SFTF		WK	PJS	Node 4 (1)	
<b>2016 features</b>																											
exponentiation (**) operator	3/3	2/3	3/3	3/3	2/3	0/3	0/3	3/3	3/3	3/3	3/3	3/3	3/3	3/3	3/3	3/3	3/3	3/3	3/3	3/3	3/3	3/3	3/3	3/3	0/3	0/3	
Array.prototype.includes	3/3	0/3	3/3	1/3	3/3	2/3	0/3	3/3	3/3	3/3	3/3	3/3	3/3	3/3	3/3	3/3	3/3	3/3	3/3	3/3	3/3	3/3	3/3	3/3	0/3	0/3	
<b>2016 misc</b>																											
generator functions can be used with "new"	Yes	No	No	No	No	No	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	No	
generator throw caught by inner generator	Yes	No	No	No	No	No	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	
strict mode non-strict non-simple params is error	Yes	No	No	No	No	No	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	No	
nested rest destructuring declarations	Yes	No	Yes	Yes	Yes	No	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	No	
nested rest destructuring parameters	Yes	No	Yes	Yes	Yes	No	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	No	
Proxy "ownKeys" handler removed	Yes	No	No	No	No	No	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	No	
Proxy normal calls, Array.prototype.includes	Yes	No	No	No	No	No	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	No	
<b>2017 Features</b>																											
Class static methods	4/4	0/4	4/4	3/4	4/4	3/4	0/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	0/4	0/4	
String padding	2/2	0/2	2/2	2/2	2/2	2/2	0/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	0/2	0/2	
trailing commas in function syntax	2/2	0/2	2/2	2/2	2/2	0/2	0/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	0/2	0/2	
async functions	15/15	3/15	3/15	3/15	2/15	0/15	0/15	15/15	15/15	15/15	15/15	15/15	15/15	15/15	15/15	15/15	15/15	15/15	15/15	15/15	15/15	15/15	15/15	15/15	0/15	0/15	
shared memory and atomics	17/17	0/17	0/17	0/17	0/17	0/17	0/17	0/17	17/17	17/17	17/17	0/17	17/17	17/17	17/17	17/17	17/17	17/17	17/17	17/17	17/17	17/17	17/17	17/17	0/17	0/17	
<b>2017 misc</b>																											
Proxy "ownKeys" handler, duplicate keys for non-extendible targets (ES 2017 semantics)	No	No	No	No	No	No	No	No	Yes	Yes	Yes	Yes	Yes	No	No	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	No	
RegExp "u" flag, case folding	Yes	No	No	No	No	No	No	No	No	No	No	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	No
arguments callee removal	Yes	No	No	No	No	No	No	No	Yes	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	No	
<b>2017 annex B</b>																											

ES5 was published in 2009, ES6 in 2015, ES7 in 2016.  
 ES8 (or ES2017) was finalised in June 2017.  
 Yearly releases promised.

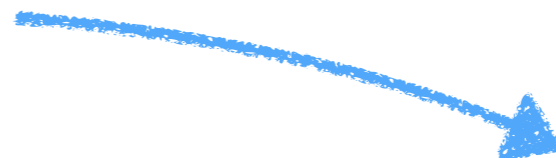
<http://kangax.github.io/compat-table/>

# Compiling to JavaScript

this course: plain JavaScript



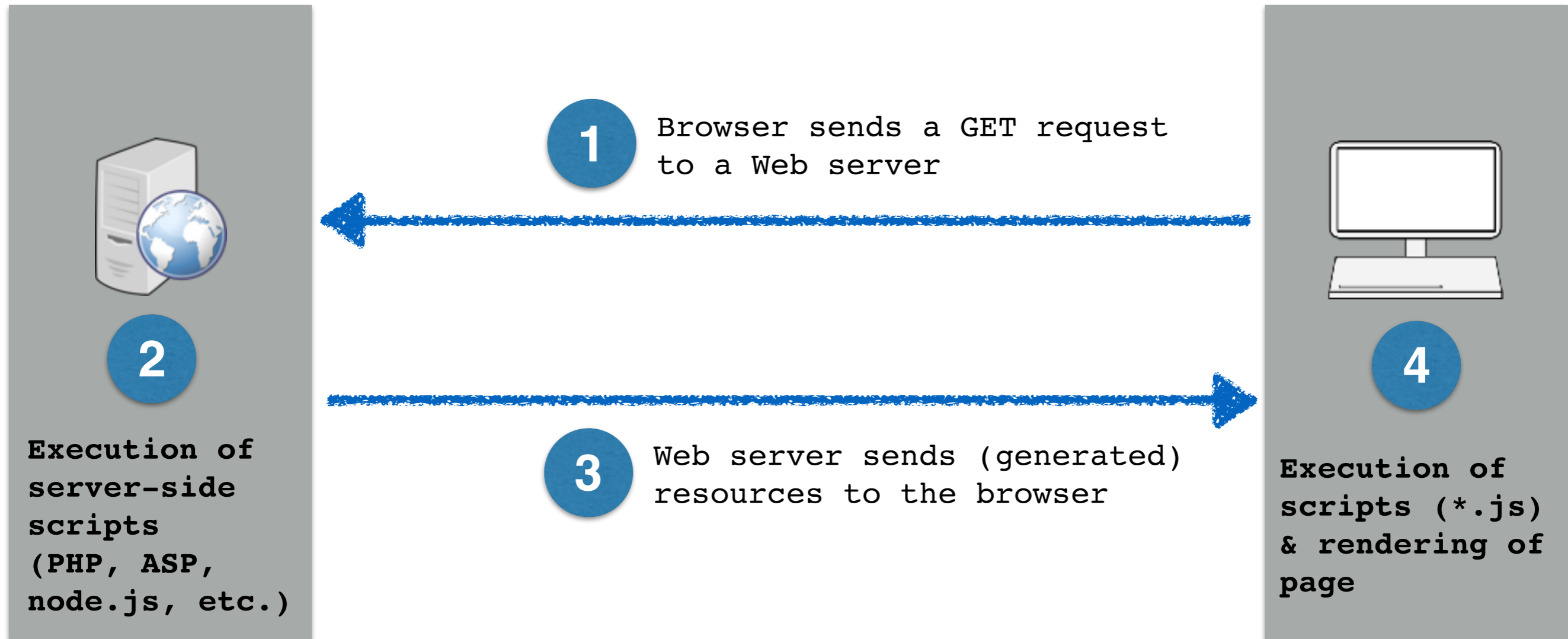
All major languages compile to JS



# Scripting overview



# Requesting & processing a Web page in 4 steps



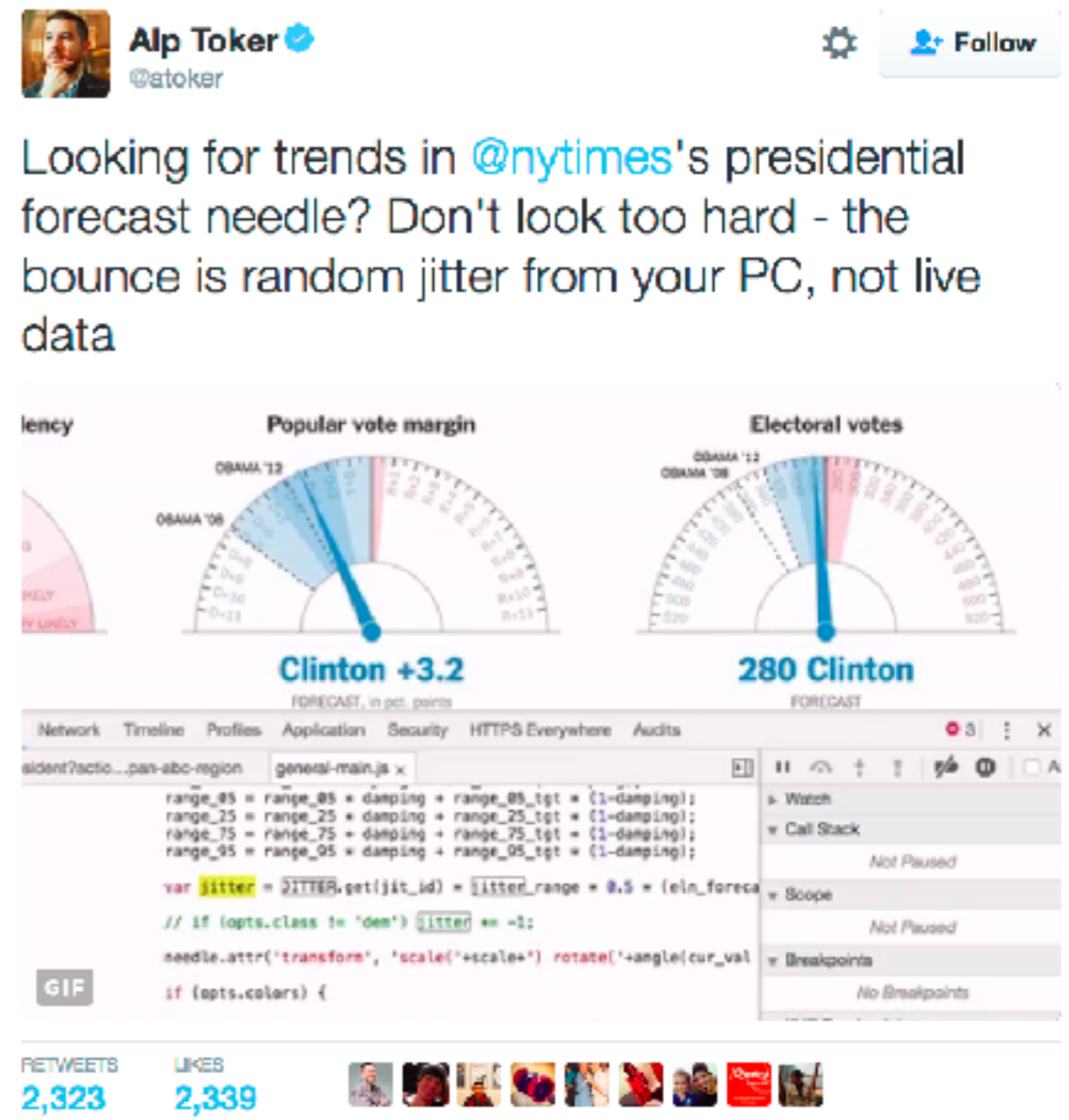
JavaScript makes Web apps **interactive** and **responsive** to user actions.

# Server-side scripting

- Source code is **private**, result of script execution is returned (in HTML), not the script itself
- HTML can be rendered by **any browser**
- Server-side scripts can **access additional resources** (including databases)
- Server-side scripts can use **non-standard language features** (you know your server's software)

# Client-side scripting

- Source code is **visible** to everyone



**Alp Toker** @atoker

Looking for trends in @nytimes's presidential forecast needle? Don't look too hard - the bounce is random jitter from your PC, not live data

lency

Popular vote margin

Electoral votes

Clinton +3.2

280 Clinton

```
range_05 = range_05 * damping + range_05_tgt * (1-damping);
range_25 = range_25 * damping + range_25_tgt * (1-damping);
range_75 = range_75 * damping + range_75_tgt * (1-damping);
range_95 = range_95 * damping + range_95_tgt * (1-damping);

var jitter = JITTER.get(jit_id) * jitter_range * 0.5 * (in_foreca

// if (opts.class != 'dem') jitter += -1;

needle.attr('transform', 'scale('+scale+') rotate('+angle|cur_val

if (opts.colors) {
```

RETWEETS 2,323 LIKES 2,339

# Client-side scripting

- Source code is **visible** to everyone
- Script execution by the browser **reduces load** on the Web server
- All **raw data** necessary (e.g. for visualizations) needs to be downloaded and **processed by the client**
- JavaScript is **event-driven**: code blocks executed in response to user actions (click, hover, move, etc.)



*Learning*

# JavaScript Design Patterns

O'REILLY®

*Addy Osmani*

Objects in JavaScript

**Basic constructor**

**Prototype-based  
constructor**

**Module pattern**

# OO for JavaScript

“A value has first-class status if it can be **passed** as a **parameter**, **returned** from a **subroutine**, or **assigned** into a **variable**.” (Michael L. Scott)

- JavaScript has **functions** as **first-class citizens**
- OO groups together **related data and behaviour**
- Built-in objects: String, Number, Array, etc.
- **Objects can be created in different ways** (do not get confused, stick to one way)

# Design patterns

“Design patterns are reusable solutions to commonly occurring problems in software design.” — **Addy Osmani**

- Many design patterns exist, we focus on **three** (the most important ones for our use case)
- Design patterns develop over time
- Design patterns often hold across programming languages

# Objects in JavaScript

- `new Object()` produces **an empty object**, ready to receive name/value pairs
  - Name: any string
  - Value: anything (String, array, Number, etc.) apart from `undefined`
- **Members** are accessed through
  - `[ name ]` (bracket notation)
  - `.name` (dot notation)

```
1 var note1 = new Object();
2 note1["type"] = 1;
3 note1["note"] = "Math homework due";
4 console.log(note1["type"]); /* prints out: 1 */
5 console.log(note1.note); /* prints out: "Math homework due" */
```



# Another way: object literals

```
1 var note1 = new Object();
2 note1["type"] = 1;
3 note1["note"] = "Math homework due";
4 console.log(note1["type"]); /* prints out: 1 */
5 console.log(note1.note); /* prints out: "Math homework due" */
```



```
1 var note2 = {
2   type: 2,
3   note: "Math homework due" /* no comma at the last entry */
4 };
```

# Adding a method

```
1 var note1 = new Object();
2 note1["type"] = 1;
3 note1["note"] = "Math homework due";
4 note1["toString"] = function(){
5     note1.toString() /* this refers to the current object */
6     return "Note: "+this.note+", type: "+this.type;
7     };
```

```
1 var note2 = {
2     type: 2,
3     message: "Math homework due",
4     toString: function() {
5         /* this refers to the current object */
6         return "Note: "+this.note+", type: "+this.type;
7     }
8 };
9     note2.toString()
```

# Object literals can be complex

```
1 var paramModule = {
2   /* parameter object */
3   Param : {
4     minType : 1,
5     maxType : 5,
6     maxNoteLen : 100,
7   },
8
9   getParams : function() {
10    var s = "Here all parameters should be listed ... ";
11    return s;
12  }
13 };
```

inner object  
Param

# Are object literals enough?

```
1 var note2 = {
2     type: 2,
3     message: "Math homework due",
4     toString: function() {
5         /* this refers to the current object */
6         return "Note: "+this.note+", type: "+this.type;
7     }
8 };
```

What happens if we need **1,000 objects** of this kind?  
What happens if a **method** needs to be **added to all objects**?

# Design Pattern (I): Basic constructor

# Recall: constructors in Java

```
1 public class Note {
2     private String note; /* encapsulate private members */
3     private int type;
4
5     /* constructor: a special method to initialize a new object */
6     public Note(String n, int t) {
7         this.note = n; /* this: reference to the current object */
8         this.type = t;
9     }
10    public void setType(int t) {this.type = t;}
11
12    public int getType() {return this.type;}
13
14    public String getNote() {return this.note;}
15
16    public String toString() {
17        return "Note: "+this.note+", type: "+this.type;
18    }
19 }
```

```
1 Note note1 = new Note("Maths homework assignment", 1);
2 note1.setType(2);
```

# Basic constructor in JavaScript

In JavaScript, **functions** are **first-class citizens**.

```
1 function Note( note, type ) {
2   this.note = note; /* this: reference to the current object */
3   this.type = type;
4
5   this.setType = function(t) {this.type = t;};
6
7   this.getType = function() {return this.type;};
8
9   this.getNote = function() {return this.note;};
10
11  this.toString = function () {
12    return "Note: "+this.note+", type: "+this.type;
13  };
14 }
```

```
1 var note1 = new Note("Maths homework assignment", 1);
2 note1.setType(2);
3 note1.toString();
4 var note2 = new Note("English homework due"); /* what happens to type? */
5
```

# Basic constructor

- An object constructor is just **a normal function**
- What does JavaScript do with **new**?
  - new anonymous empty object is created and used as **this**
  - **returns** new object at the end of the function

common error: forgetting "new"


```
1 var note1 = new Note("Maths homework assignment", 1);
2 note1.setType(2);
3 note1.toString();
4 var note2 = new Note("English homework due"); /* what happens to type? */
5 var note3 = Note("Music homework due", 3); /* what now? */
```

**this can refer to anything!**




# Basic constructor

```
1 /* Remember that JavaScript is loosely typed */
2 var note1 = new Note("Maths homework", "IMPORTANT");
3 note1.type; /* "IMPORTANT" */
4 var note2 = new Note("English homework", 1);
5 note2.type; /* 1 */
6 note2.dueDate = "1-1-2015"; /* new members added on the fly */
7 note1.toString(); /* "Note: Maths homework, type: IMPORTANT" */
8 note1.toString = function(){return this.type;};
9 note1.toString(); /* "IMPORTANT" */
```



New variables and objects **can be added on the fly**.

```
12 note1.hasOwnProperty("dueDate"); /* false */
13 note1.hasOwnProperty("type"); /* true */
```



Objects come with **default methods** (prototype chaining)

# Summary: basic constructor

- Advantage: **very easy to use**
- Issues:
  - Not obvious how to use **inheritance** (e.g. `NoteWithDueDate`)
  - **Objects do not share functions**
    - function `toString()` is not shared between `note1` and `note2`
  - **All members are public**
    - Any piece of code can access/change/delete(!) members `type` and `note`

# Design Pattern (2): Prototype-based constructor

# Prototype chaining explained

Objects have a **secret pointer** to another object - the object's **prototype**

- Properties of the constructor's prototype are also accessible in the new object
- If a member is not defined in the object, the **prototype chain** is followed

```
var name = "Daisy";  
typeof(name); // "string"
```

```
name.charAt(1)
```

\_\_proto\_\_

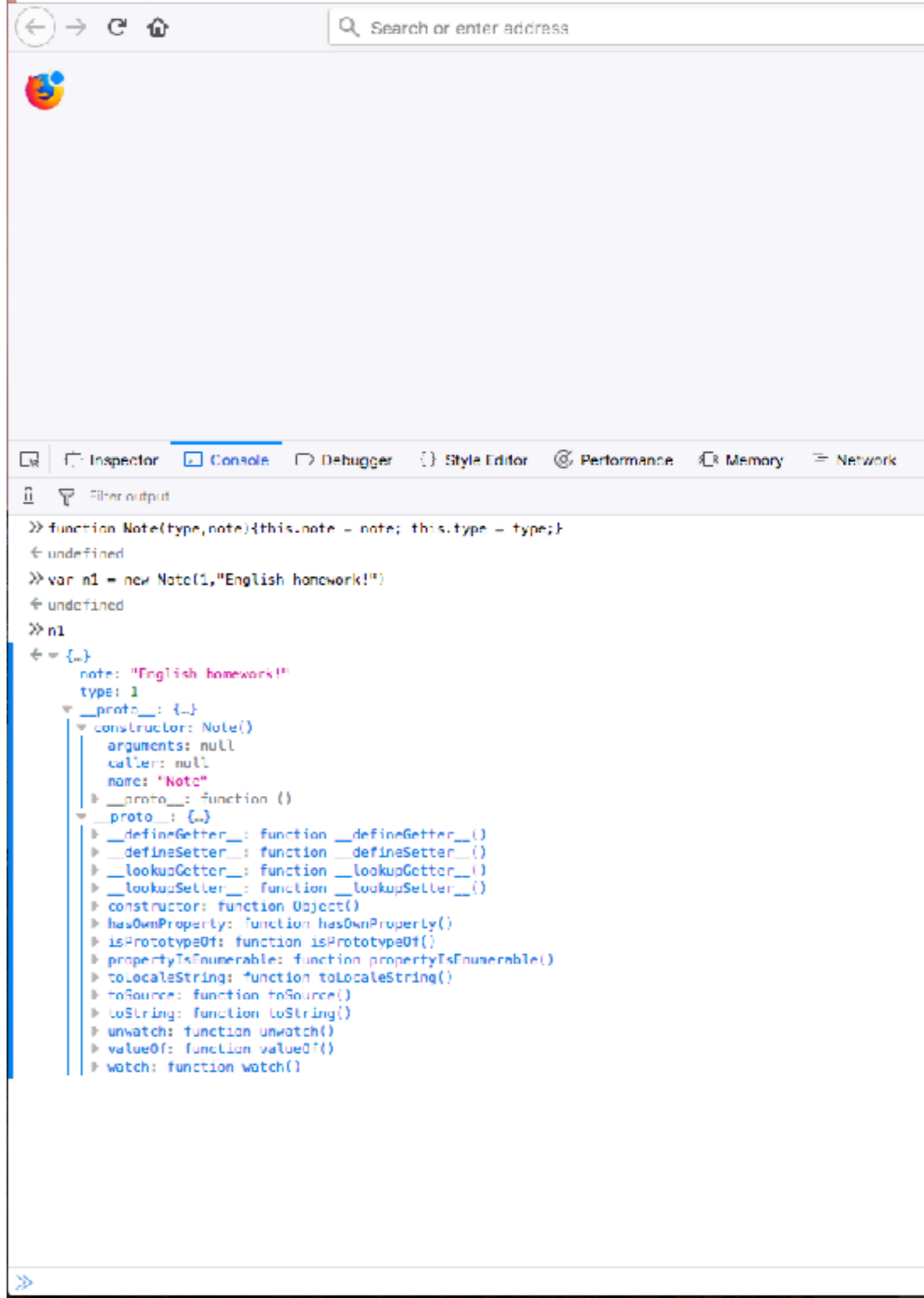
```
String.prototype  
charAt()  
indexOf()  
...
```

# Prototype-based constructor

```
1 function Note( note, type ) {
2   this.note = note; /* this: reference to the current object */
3   this.type = type;
4 }
5
6 /* member methods are defined once in the prototype */
7 Note.prototype.setType = function(t) {this.type = t;};
8 Note.prototype.getType = function() {return this.type;};
9 Note.prototype.getNote = function() {return this.note;};
10 Note.prototype.toString = function () {
11   return "Note: "+this.note+", type: "+this.type;
12 };
13
14 // Using it:
15 var note1 = new Note("Maths homework due", "IMPORTANT");
16 var note2 = new Note("English homework", 2);
17 note1.getType(); /* "IMPORTANT" */
18 note2.getNote(); /* "Maths homework due" */
```

# Getting to grips with JavaScript

**WebConsole** is your friend!



The screenshot shows a web browser's developer console with the following content:

```
>> function Note(type,note){this.note = note; this.type = type;}  
← undefined  
>> var n1 = new Note(1,"English homework!")  
← undefined  
>> n1  
← {  
  note: "English homework!"  
  type: 1  
  __proto__: {  
    constructor: Note()  
    arguments: null  
    caller: null  
    name: "Note"  
    __proto__: function ()  
    __proto__: {  
      __defineGetter__: function __defineGetter__()  
      __defineSetter__: function __defineSetter__()  
      __lookupGetter__: function __lookupGetter__()  
      __lookupSetter__: function __lookupSetter__()  
      constructor: function Object()  
      hasOwnProperty: function hasOwnProperty()  
      isPrototypeOf: function isPrototypeOf()  
      propertyIsEnumerable: function propertyIsEnumerable()  
      toLocaleString: function toLocaleString()  
      toSource: function toSource()  
      toString: function toString()  
      unwatch: function unwatch()  
      valueOf: function valueOf()  
      watch: function watch()  
    }  
  }  
}
```

# Prototype-based constructor

Prototype changes are also reflected in existing objects!

```
1 function Note( note, type ) {
2   this.note = note; /* this: reference to the current object */
3   this.type = type;
4 }
5
6 /* member method setType() defined */
7 Note.prototype.setType = function(t) {this.type = t;};
8
9 var note1 = new Note("Maths homework due", "IMPORTANT");
10 note1.setType(2); /* OK */
11 note1.getType(); /* TypeError: note1.getType isn't a function */
12
13 /* lets define the method */
14 Note.prototype.getType = function() {return this.type;}
15
16 note1.getType(); /* 2 */
```

# Prototype-based constructor

## Inheritance through prototyping.

```
1 function Note(note, type){
2   this.note = note;
3   this.type = type;
4 }
5
6 Note.prototype.setType = function(t){this.type = t;}
7 Note.prototype.getType = function(){return this.type;}
8
9 /* constructor */
10 function NoteWithDeadline(note, type, dueDate){
11   Note.call(this,note,type); /* ensures proper setting of 'this' */
12   this.dueDate = dueDate;
13 }
14
15 /* redirect prototype */
16 NoteWithDeadline.prototype = Object.create(Note.prototype);
17 /* redirect the constructor */
18 NoteWithDeadline.prototype.constructor = NoteWithDeadline;
19
20 /* using it */
21 var nw = new NoteWithDeadline("Algorithms", 1, "1-1-2017");
22 nw.setType(2); /* this works, setType is defined in Note */
```

call() calls a function with a given this value and arguments (one by one)

1. create a new constructor

2. redirect the prototype

= prototype chain



# Summary: prototype-based constructor

- Advantages:
  - **Inheritance is easy** to achieve
  - **Objects share functions**
- Issue:
  - All members are **public**, i.e. any piece of code can access/change/delete members type and note

# Design Pattern (3): Module

# JavaScript scoping

- All JavaScript code enters the **same namespace**
- JavaScript has **limited scoping**
  - `var` in function: **local**, limited scope
  - `var` outside of a function: **global** scope
  - no `var`: **global** scope  
(holds for function names too)
  - `let` (ES6): block scope
  - `const` (ES6): block scope, no reassignment or redeclaration

# JavaScript scoping

```
1 var note1 = new Note("Maths", 1); /* global */
2 var note2 = new Note("English", 3); /* global */
3
4 function calcMinType(n1,n2) { /* global */
5     var t1 = Number(n1.type); /* local */
6     t2 = Number(n2.type); /* global */
7     return Math.min(t1,t2);
8 }
9
10 t1; /* ReferenceError: t1 is not defined */
11 t2; /* ReferenceError: t2 is not defined */
12
13 calcMinType(note1,note2);
```

What if another JavaScript library used in the project defines `note1`?

# Module

- Goals:
  - **Do not declare any global variables** or functions unless required
  - Emulate **private/public** membership
  - Expose only the **necessary** members to the public (as API)
- Results:
  - Potential **conflicts** with other JavaScript libraries are **reduced**
  - Public API **minimizes** unintentional side-effects when wrongly used

# Module

```
1 var notesModule = (function () {  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12  
13  
14 })(); /* function is called */
```

# Module

```
1 var notesModule = (function () {
2
3   /* 'private' members */
4   var noteCounter = 0;
5   ...
6
7   /* 'public' members; return accessible object */
8   return {
9     incrNoteCounter : function () {
10      noteCounter++;
11    },
12    ...
13  };
14 })(); /* function is called */
```

# Module

```
1 var notesModule = (function () {
2
3   /* 'private' members */
4   var noteCounter = 0;
5   ...
6
7   /* 'public' members; return accessible object */
8   return {
9     incrNoteCounter : function () {
10      noteCounter++;
11    },
12    ...
13  };
14 }());
```

1 notesModule.incrNoteCounter();

2 notesModule.noteCounter; /\* un

public

private



} ) ( ) ;

# Module

the pattern  
can be  
arbitrarily  
complex;

```
1 var notesModule = (function () {
2     /* 'private' members */
3     var noteCounter = 0;
4     var logCounter = function() {
5         console.log("notesModule counter: "+noteCounter);
6     };
7     /* 'public' members; return accessible object */
8     return {
9         incrNoteCounter : function () {
10            noteCounter++;
11        },
12        decrNoteCounter : function () {
13            if(noteCounter>0) {
14                noteCounter--;
15            }
16        },
17        getNoteCounter : function() {
18            logCounter();
19            return noteCounter;
20        }
21    };
22 })();/* function is called */
```

# Module

The encapsulating function can also contain arguments

```
1 var notesModule = (function (startingCount) {
2     /* 'private' members */
3     var noteCounter = startingCount;
4     var logCounter = function() {
5         console.log("notesModule counter: "+noteCounter);
6     };
7     /* 'public' members; return accessible object */
8     return {
9         incrNoteCounter : function () {
10            noteCounter++;
11        },
12        decrNoteCounter : function () {
13            if(noteCounter>0) {
14                noteCounter--;
15            }
16        },
17        getNoteCounter : function() {
18            logCounter();
19            return noteCounter;
20        }
21    };
22 })(5);/* function is called */
```

# Summary: module

- Advantages:
  - **Encapsulation** is achieved
  - Object members are either **public** or **private**
- Issues:
  - Changing the type of membership (public/private) costs time
  - **Methods added on the fly later on cannot access 'private' members**

# Events & the DOM

# A look at book chapter 4

```
1 var main = function () {
2   "use strict";
3   $(".comment-input button").on("click", function (event) {
4     var $new_comment = $("<p>"),
5     comment_text = $(".comment-input input").val();
6     $new_comment.text(comment_text);
7     $(".comments").append($new_comment);
8   });
9 };
10 $(document).ready(main);
```

- Uses **jQuery** extensively (a big time saver)
- **Important to understand** what jQuery "covers up"
- Decide for yourself whether you want to use jQuery in the assignments (*other JavaScript libraries are not allowed*)

# A look at book chapter 4

```
1 /* jQuery's way of accessing DOM elements */
2 $(".comment-input button").on("click", function (event) {
3     ...
4 });
```

- With jQuery: no matter if `class` or `id` or ..., the access pattern is the same, i.e. `$()`
- **Callback principle**: we define what should happen when an event fires

\$ ( )



} ) ;

**“CALLBACK HELL”**

} ) ;

} ) ;

} ) ;

} ) ;

} ) ;

} ) ;

# Step-by-step: making a responsive UI control

1. Pick a control (e.g. a button)

2. Pick an event (e.g. a click on a button)

3. Write a JavaScript function: what should happen when the event occurs? (e.g. a popup appears)

4. Attach the function to the event ON the control

# Client-side JS examples

[HTML SLIDES]