

JavaScript: the language of browser interactions

Claudia Hauff
cse1500-ewi@tudelft.nl

Web technology overview

1. HTTP: the language of web communication
2. HTML & web app design
3. **JavaScript**: interactions in the browser
4. **Node.js**: JavaScript on the server
5. **CSS**: adding style
6. Node.js: advanced topics
7. Cookies & sessions
8. Web security

Learning goals

- **Employ** JavaScript objects.
- **Employ** the principle of callbacks.
- **Write** interactive web applications based on click, mouse and keystroke events.
- **Explain** and use jQuery.

Take-aways of book chapter 4

- JavaScript basics (variables, functions, ...)
- JavaScript as part of a web application
- Where to place <script> and why
- “use strict”
- DOM
- jQuery basics

No emphasis on
JavaScript objects.



A bit of context

JavaScript's reputation

- Ten years ago it was considered a toy language
- Now: most important language of the **modern Web stack**
 - **Tooling** has vastly improved (debuggers, testing frameworks, etc.)
 - JavaScript **runtime engines** are efficient (V8, SpiderMonkey, Chakra)
 - JavaScript tracks **ECMAScript (ES)**

JavaScript's reputation

- Ten years ago
- Now: most **Web stack**
- **Tooling** & testing frameworks
- JavaScript (V8, SpiderMonkey)
- JavaScript



Frontend devs used to just need HTML, CSS, and JS! Now apparently they need to learn about node, npm, grunt, gulp, webpack, babel...wait up...



what the heck do I need node for on the frontend?



“Learning modern JavaScript is tough if you haven’t been there since the beginning.”

A language in flux

COMPAT ES ECMAScript 5 6 2016+ next intl non-standard compatibility table Flattr by kangax &

Sort by Engine types Show obsolete platforms Show unstable platforms

V8 SpiderMonkey JavaScriptCore Chakra Carakan KJS Other

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

98% 56% 71% 71% 50% 68% 17% 5% 11% 96% 96% 98% 98% 98% 98% 98% 98% 98% 98% 98% 98% 98% 98% 99% 99% 99%

ES5 was published in 2009.

ES6 was published in 2015 (ES2015).

ES7 was published in 2016.

ES.Next points to the version to come.

A language in flux

A language in flux

compat
ES

ECMAScript 5 6 2016+ next intl non-standard compatibility table

This table shows proposals which have not yet been included in the current ECMAScript standard, but are at one of the maturity stages of the [TC39 process](#).

Sort by Engine types ▾ Show obsolete platforms □ Show unstable platforms ✓

Legend: V8 SpiderMonkey JavaScriptCore Chakra Other
 Minor difference (1 point) Small feature (2 points) Medium feature (4 points) Large feature (8 points)

Feature name	Current browser	Compilers/polyfills										Desktop browsers												Servers/runtime				
		Traceur	Babel 6+ core-js	Babel 7+ core-js	Closure 2018.10	TypeScript + core-js	Edge 17	Edge 18	Edge 19 Preview	FF 60 ESR	FF 62	FF 63	FF 64 Beta	FF 65 Nightly	CH 69 OP 56	CH 70 OP 57	CH 71 OP 58	CH 72 OP 59	SF 11.1	SF 12	SF TP	WK	PJS	Node ≥v6.5 <7 ^[2]	Node ≥v8.10 <9 ^[2]	Node ≥v10.13 <11 ^[2]		
Candidate (stage 3)																												
• string trimming	►	4/4	0/4	4/4	4/4	0/4	4/4	2/4	2/4	2/4	2/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	2/4	4/4	4/4	2/4	2/4	4/4	4/4			
• globalThis	►	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2			
• String.prototype.matchAll	●	No	No	Yes ^[4]	Yes ^[4]	No	Yes ^[5]	No	No	No	No	No	No	No	Flag ^[8]	Flag ^[8]	Flag ^[8]	Flag ^[8]	No	No	No	No	No	No	Flag ^[8]			
• instance class fields	►	0/3	1/3	1/3	1/3	0/3	1/3	0/3	0/3	0/3	0/3	0/3	0/3	0/2	0/3	0/3	0/3	0/3	0/3	0/3	0/3	0/3	0/3	0/3	0/3			
• static class fields	►	0/2	1/2	1/2	1/2	0/2	1/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	1/2	0/2	0/2	0/2	0/2	0/2	0/2			
• Function.prototype.toString revision ↗	►	7/7	0/7	0/7	0/7	0/7	4/7	4/7	4/7	7/7	7/7	7/7	7/7	7/7	7/7	7/7	7/7	7/7	4/7	4/7	4/7	1/7	4/7	4/7	7/7			
• Array.prototype{flat,flatMap} ^[9]	►	2/2	0/2	1/2	1/2	0/2	1/2	0/2	0/2	0/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	0/2	2/2	2/2	0/2	0/2	0/2	0/2			
• Symbol.prototype.description ↗	●	Yes	No	No	No	No	No	No	No	No	Yes	Yes	Yes	Yes	Flag ^[8]	Yes	Yes	Yes	No	Yes	Yes	No	No	No	No			
• BigInt	►	0/8	0/8	0/8	0/8	0/8	0/8	0/8	0/8	0/8	0/8	0/8	0/8	0/8	8/8	8/8	8/8	8/8	0/8	0/8	0/8	0/8	0/8	0/8	8/8			
• Object.fromEntries ↗	●	Yes	No	No	No	No	No	No	No	No	Yes	Yes	Yes	No	No	No	No	No	Yes	Yes	No	No	No	No	No			
• Well-formed JSON.stringify	●	No	No	No	No	No	No	No	No	No	No	No	No	Yes	Yes	No	No	No	Yes	No	No	No	No	No	No			
Draft (stage 2)																												
• Generator function.sent Meta Property	●	No	No	Yes	Yes	No	No	No	No	No	No	No	No	No	No	No	No	No	No	No	No	No	No	No	No			
• Class and Property Decorators	►	0/1	0/1	0/1	0/1	0/1	1/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1			
• Realms	●	No	No	No	No	No	No	No	No	No	No	No	No	No	No	No	No	No	No	No	No	No	No	No	No			
• weak references	●	No	No	No	No	No	No	No	No	No	No	No	No	No	No	No	No	No	No	No	No	No	No	No	No			
• throw expressions	►	0/4	0/4	4/4	0/4	0/4	0/4	0/4	0/4	0/4	0/4	0/4	0/4	0/4	0/4	0/4	0/4	0/4	0/4	0/4	0/4	0/4	0/4	0/4	0/4			
• numeric separators	●	No	No	Yes	No	Yes	No	No	No	No	No	No	No	No	Flag ^[8]	Flag ^[8]	Flag ^[8]	Flag ^[8]	No	No	No	No	No	No	Flag ^[8]			
• Set methods	►	0/4	0/4	0/4	0/4	0/4	0/4	0/4	0/4	0/4	0/4	0/4	0/4	0/4	0/4	0/4	0/4	0/4	0/4	0/4	0/4	0/4	0/4	0/4	0/4			
Proposal (stage 1)																												
• do expressions	●	No	No	Yes	Yes	No	No	No	No	No	No	No	No	No	No	No	No	No	No	No	No	No	No	No	No			
• Observable	►	0/7	0/7	7/7	7/7	0/7	7/7	0/7	0/7	0/7	0/7	0/7	0/7	0/7	0/7	0/7	0/7	0/7	0/7	0/7	0/7	0/7	0/7	0/7	0/7			
• Frozen Realms API	●	No	No	No	No	No	No	No	No	No	No	No	No	No	No	No	No	No	No	No	No	No	No	No	No			



Semmy Purewal

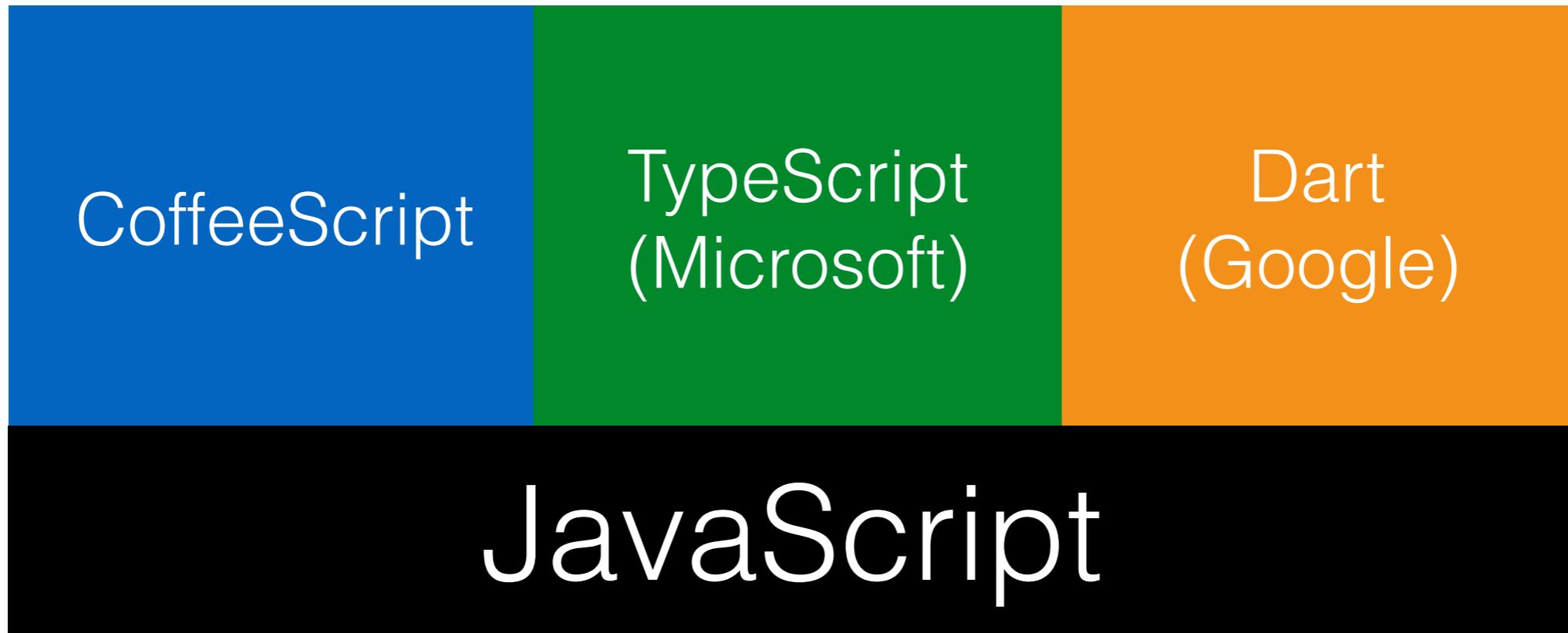
Published in 2014,
thus no ES6.

**Not a bug, but a
feature!**

ES6 introduced
many features that
only make sense
knowing the “old”
JavaScript.

We use two ES6
features: `let`, `const`

Compiling to JavaScript



All major (and not so major) languages compile to JS

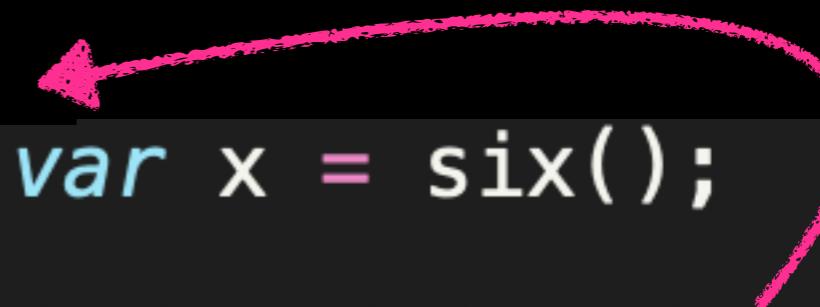


JavaScript has quirks

Hoisting principle:
declarations are
processed before any
code is executed.

Declarations are
hoisted to the top (also
inside of functions).

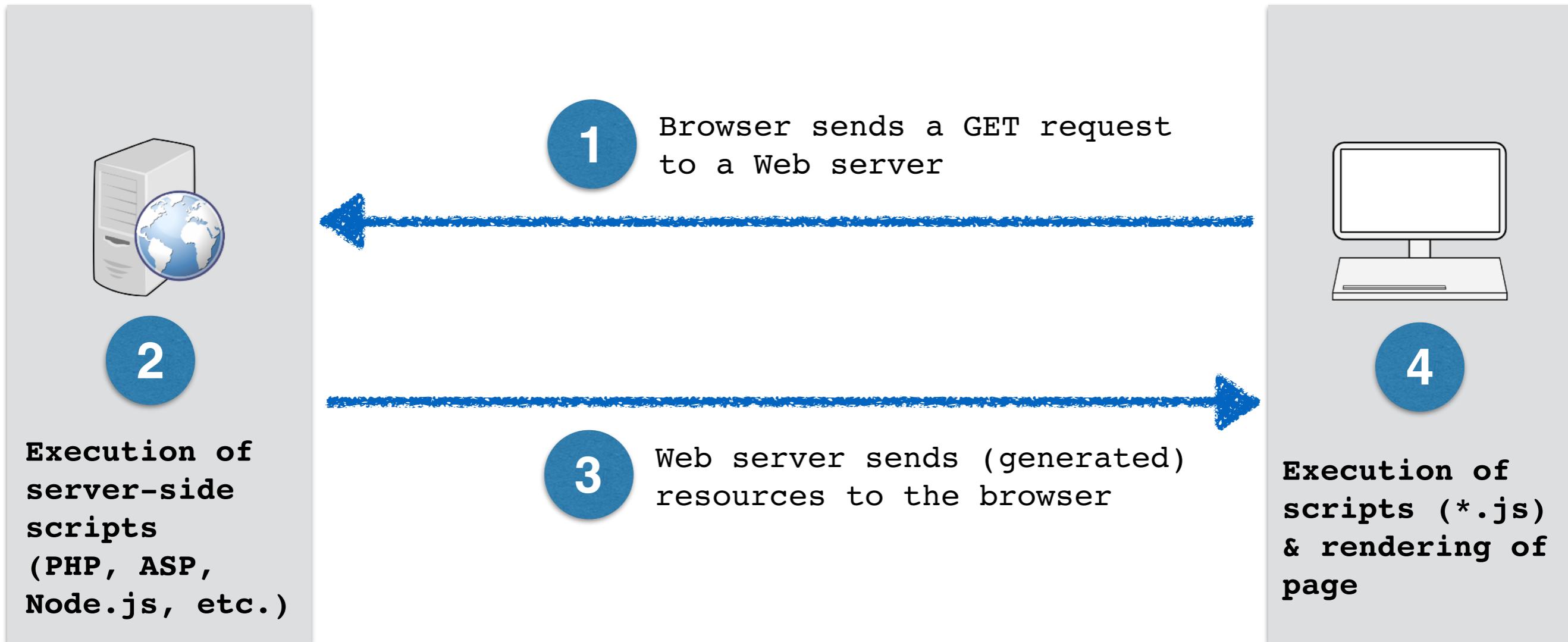
Expressions are not
hoisted.



```
1 var x = six();
2
3 //function declaration
4 function six(){
5     return 6;
6 }
7
8 var y = seven()
9
10 //function expression
11 var seven = function(){
12     return 7;
13 }
14
15 console.log(x+" "+y);
```

Scripting overview

Interactive web applications



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

Server-side scripting

- Source code is **private**, only the result of a script's execution is returned
- HTML can be rendered by **any browser**
- Server-side scripts can **access additional resources** (e.g. databases)
- Server-side scripts can use **non-standard language features**

Client-side scripting

- Source code is **visible** to everyone
- Client-side script execution **reduces server load**
- All **raw data** needs to be downloaded and processed by the client
- JavaScript is **event-driven**: code blocks executed in response to user actions

Client-side scripting



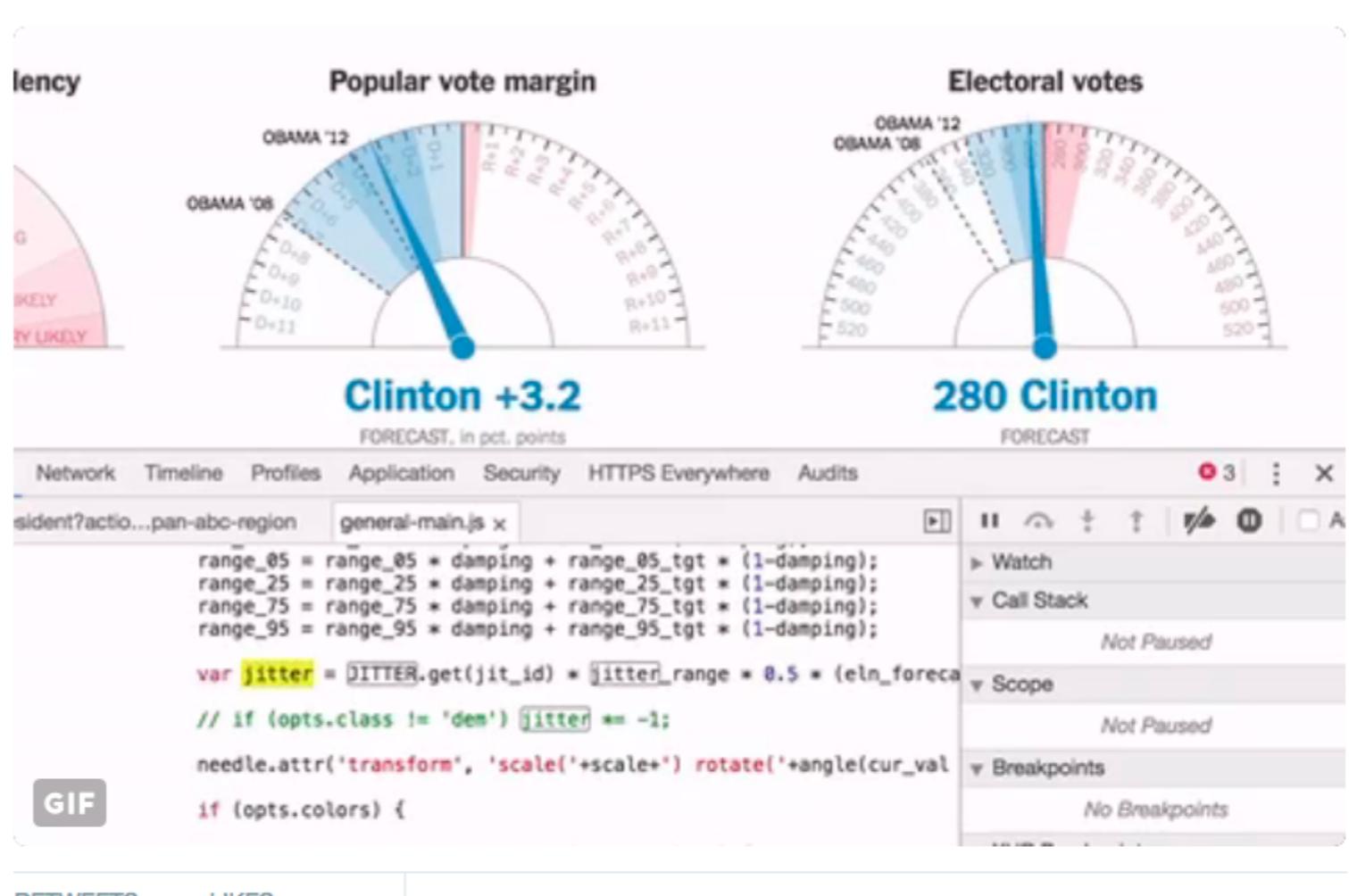
Alp Toker

@atoker



Follow

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

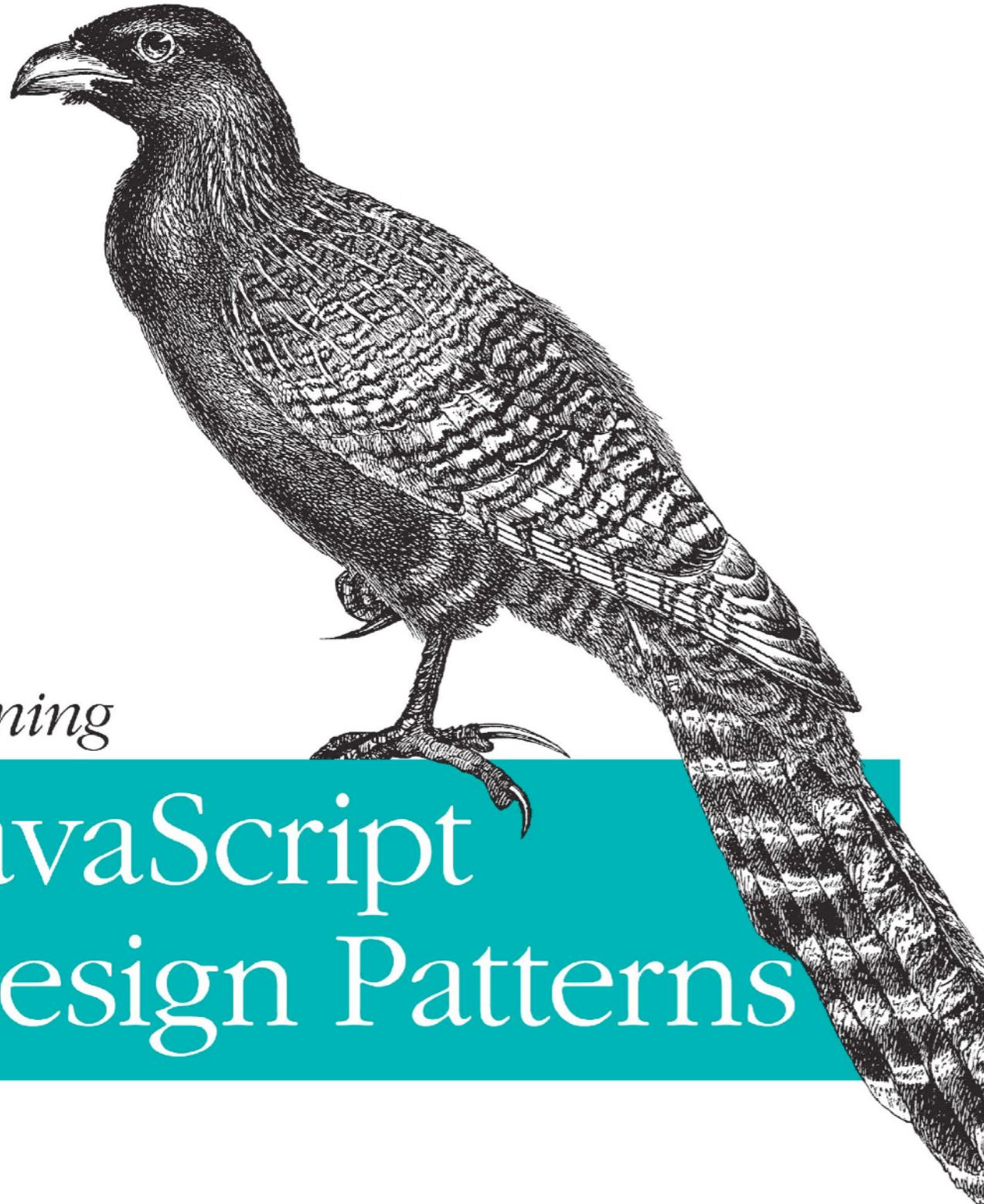


RETWEETS
2,323

LIKES
2,339



Learning
JavaScript
Design Patterns



O'REILLY®

Addy Osmani

Objects in JavaScript

Basic constructor

**Prototype-based
constructor**

Module pattern

Executing this JavaScript code snippet yields what output?

```
1 function giveMe(x){  
2     return function(y){  
3         return x*y;  
4     }  
5 }  
6 var giveMe5 = giveMe(5);  
7  
8 console.log( giveMe5(10) );
```

- A. Error: giveMe5 is not a function
- B. Error: x is undefined
- C. 10
- D. 50 ✓

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** (which are also objects) as **first-class citizens**
- Objects group together **related data and behaviour**
- Built-in objects: **String**, **Number**, **Array**, etc., but also **document** (when we talk about JS in the browser)
- **Objects can be created in ~7 different ways**

Design patterns

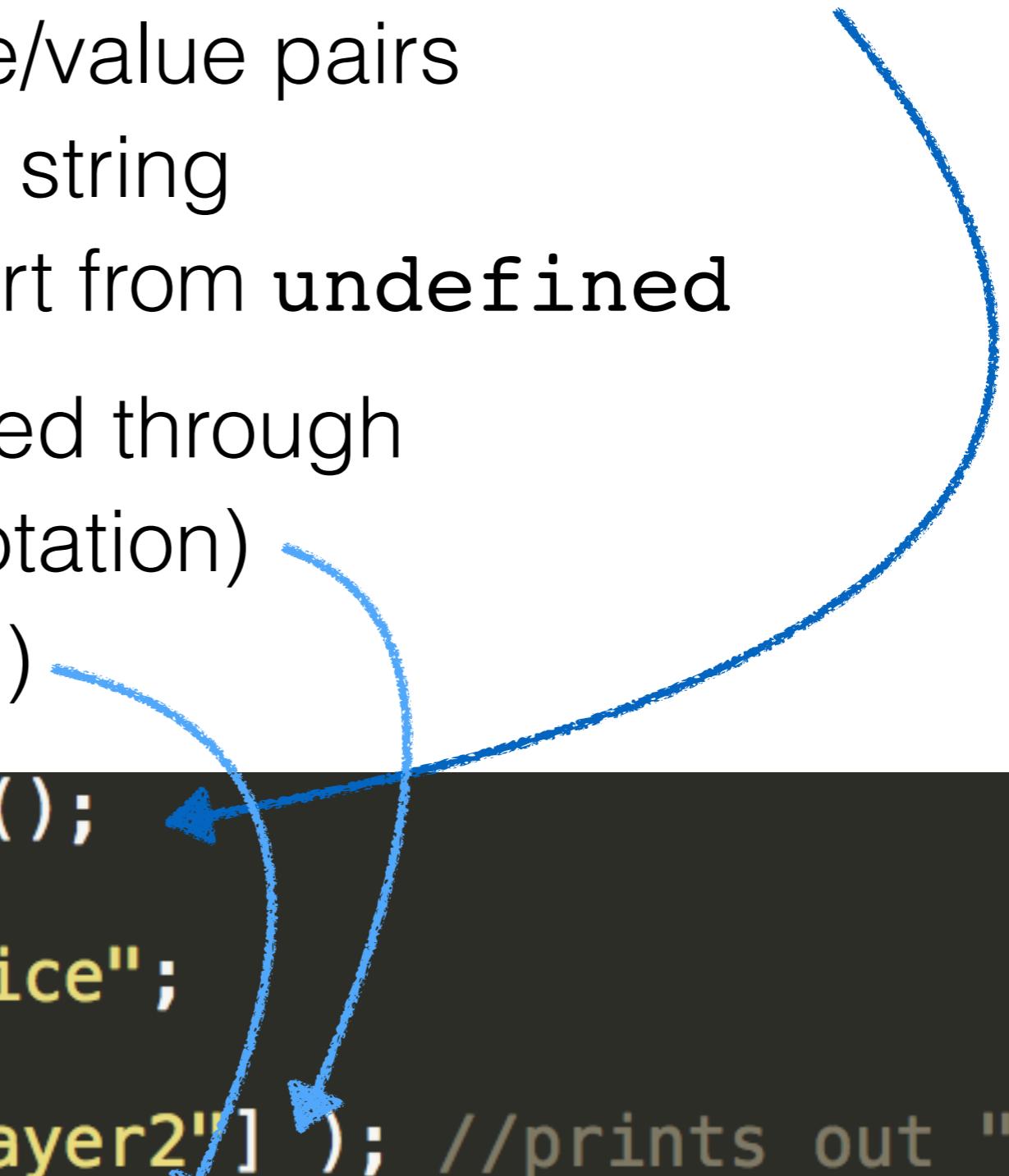
“Design patterns are **reusable solutions** to commonly occurring problems in software design.” — **Addy Osmani**

- Many design patterns exist, we focus on **three**
- 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 (**property**): a string
 - Value: anything apart from `undefined`
- **Members** are accessed through
 - `[name]` (bracket notation)
 - `.name` (dot notation)

```
1 var game = new Object();
2 game["id"] = 1;
3 game["player1"] = "Alice";
4 game.player2 = "Bob";
5 console.log( game["player2"] ); //prints out "Bob"
6 console.log( game.player1 ); //prints out "Alice"
```



Another way: object literals

```
1 var game = new Object();
2 game["id"] = 1;
3 game["player1"] = "Alice";
4 game.player2 = "Bob";
5 console.log( game["player2"] ); //prints out "Bob"
6 console.log( game.player1 ); //prints out "Alice"
```



```
1 var game = {
2   id: 1,
3   player1: "Alice",
4   player2: "Bob"
5 };
```

Adding a method

```
1 var game = new Object();
2 game["id"] = 1;
3 game["player1"] = "Alice";
4 game.player2 = "Bob";
5
6 game["won lost"] = "1 12";
7
8 game.printID = function(){
9     console.log( this.id );
10}
11 game["printID"](); // prints out "1"
12 game.printID(); //prints out "1"
```

```
1 var game = {
2     id: 1,
3     player1: "Alice",
4     player2: "Bob",
5     "won lost": "1 12",
6     printID: function(){
7         console.log(this.id);
8     }
9};
```

Object literals can be complex

```
1 var paramModule = {  
2     /* parameter literal */  
3     Param : {  
4         minGames: 1,  
5         maxGames: 100,  
6         maxGameLength: 30  
7     },  
8     printParams: function(){  
9         console.table(this.Param);  
10    }  
11};
```

inner object
Param

Are object literals enough?

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

Constructors in Java vs. JS

```
1 public class Game {  
2     /* encapsulate private  
3      * members  
4      */  
5     private int id;  
6  
7     /* constructor */  
8     public Game(int id){  
9         this.id = id;  
10    }  
11  
12    public int getID(){  
13        return this.id;  
14    }  
15  
16    public void setID(int id){  
17        this.id = id;  
18    }  
19 }
```

- Functions as constructors
- new to initialise a new object

```
1 function Game( id){  
2     this.id = id;  
3     this.getID = function(){  
4         return this.id;  
5     };  
6     this.setID = function(id){  
7         this.id = id;  
8     };  
9 }  
10  
11 //creating objects  
12 var g1 = new Game(1);  
13 g1.getID();  
14 g1.setID(2);  
15  
16 var g2 = new Game(3);
```

Basic constructor

- An object constructor is a **normal function**
- JavaScript runtime reacts to **new** keyword:

1. anonymous empty object is created and used as **this**
2. returns new object at the end of the function

common error: forgetting "new"

- Object properties can be **added on the fly**
- Objects come with default properties (**prototype chaining**)

The screenshot shows a browser's developer tools console with tabs for Inspector and Console. The Console tab is active, displaying the following JavaScript code and its execution results:

```
>> function Game(id){  
    this.id = id;  
}  
< undefined  
>> var g1 = new Game(1);  
< undefined  
>> var g2 = Game(2);  
< undefined  
>> g1  
< Object { id: 1 }>  
>> g2  
< undefined>  
>> window.id  
< 2 > this can refer to anything!  
>> g1.getID = function(){  
    return this.id;  
}  
< function getID()>  
>> g1.getID()  
< 1 >  
>> g1.toString()  
< "[object Object]">
```

Annotations with blue arrows highlight several parts of the code and output:

- An arrow points from the word "this" in the annotation "this can refer to anything!" to the `this.id = id;` line in the `Game` constructor.
- An arrow points from the word "new" in the annotation "common error: forgetting 'new'" to the `new Game(1)` line.
- An arrow points from the word "this" in the annotation "this can refer to anything!" to the `this.id = id;` line in the `Game` constructor.
- An arrow points from the word "get" in the annotation "common error: forgetting 'new'" to the `g1.getID = function(){` line.

Basic constructor

In Java, `this` refers to the **current object**. In **JavaScript** what `this` refers to is dependent on how the function containing this was called.



The screenshot shows a browser's developer tools console tab labeled "Console". The console output is as follows:

```
>> function Game(id){  
    this.id = id;  
}  
< undefined  
>> var g1 = new Game(1);  
< undefined  
>> var g2 = Game(2);  
< undefined  
>> g1  
< ► Object { id: 1 }  
>> g2  
< undefined  
>> window.id this can refer to anything!  
< 2  
>> g1.getID = function(){  
    return this.id;  
}  
< ► function getID()  
>> g1.getID()  
< 1  
>> g1.toString()  
< "[object Object]"
```

A vertical blue line highlights the word "this" in the last line of the console output, which is annotated with the text "this can refer to anything!" in pink.

Summary: basic constructor

- Advantage: **easy to use**
- Issues:
 - **Inheritance** (e.g. NoteWithDueDate)
 - **Objects do not share functions**
 - All members are **public**: any code can access, change or delete an object's properties

next ...

Design Pattern (2): Prototype-based constructor

Prototype chaining explained

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

- Properties of the **constructor's prototype** are also accessible in the new object
- If a property 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 Game(id){  
2     this.id = id;  
3 }  
4  
5 /* new member functions are defined once in the prototype */  
6 Game.prototype.getID = function(){ return this.id; };  
7 Game.prototype.setID = function(id){ this.id = id; };  
8  
9 //using it  
10 var g1 = new Game("1");  
11 g1.setID("2"); //that works!  
12  
13 var g2 = new Game("2");  
14 g2.setID(3); //that works too!  
15  
16 g1["setID"] = function(id){  
17     this.id = "ID"+id;  
18 }  
19 //g1 and g2 now point to different setID properties  
20 g1.setID(4);  
21  
22 console.log(g1.getID()); //ID4  
23 console.log(g2.getID()); //3
```

Executing this JavaScript code snippet yields what output?

```
1 function Game(id){  
2     this.id = id;  
3 }  
4  
5 Game.prototype.getID = function(){ return this.id; };  
6  
7 var g1 = new Game("1");  
8  
9 Game.prototype.getID = function(){  
10    return "0"+this.id;  
11 }  
12  
13 console.log( g1.getID() );
```

- A. 1
- B. 01 ✓ Prototype changes are reflected in existing objects!
- C. Error: a prototype property can only be assigned once.

Prototype-based constructor

Inheritance through prototyping.

```
1 function Game(id){  
2     this.id = id;  
3 }  
4  
5 /* new member functions are defined once in the prototype */  
6 Game.prototype.getID = function(){ return this.id; };  
7 Game.prototype.setID = function(id){ this.id = id; };  
8  
9 /* constructor */  
10 function TwoPlayerGame(id, p1, p2){  
11     /*  
12      * call(...) calls a function with a given this value and arguments.  
13      */  
14     Game.call(this, id);  
15     this.p1 = p1;  
16     this.p2 = p2;  
17 }  
18  
19 /* redirect prototype */  
20 TwoPlayerGame.prototype = Object.create(Game.prototype);  
21 TwoPlayerGame.prototype.constructor = TwoPlayerGame;  
22  
23 /* use it */  
24 var TPGame = new TwoPlayerGame(1, "Alice", "Bob");  
25 console.log( TPGame.getID() ); //prints out "1"  
26 console.log( TPGame.p1 ); //prints out "Alice"
```

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**: any code can access, change or delete an object's properties

Design Pattern (3): Module

JavaScript scoping

- All code enters the **same namespace**
- **Limited scoping** (=context in which values and expressions are visible):
 - `var` in function: **local**, limited scope
 - `var` outside of a function: **global** scope
 - no `var`: **global** scope
 - `let (ES6)`: **block** scope (block: `{...}`)
 - `const (ES6)`: **block** scope, no reassignment or redeclaration

JavaScript scoping

`setTimeout(fn, d)`: sets a timer and executes function `fn` once after `d` milliseconds

Task: print the numbers 1 to 10, each with a second delay

```
1 for (var i = 1; i <= 10; i++) {  
2     setTimeout(function() {  
3         console.log(i);  
4     }, 1000);  
5 }
```



11 is printed
ten times

```
1 function fn(i) {  
2     setTimeout(function() {  
3         console.log(i);  
4     }, 1000 * i);  
5 }  
6  
7 for (var i = 1; i <= 10; i++)  
8     fn(i);
```



```
1 for (let i = 1; i <= 10; i++)  
2     setTimeout(function() {  
3         console.log(i)  
4     }, 1000 * i)
```



JavaScript scoping

```
1 function Note(n, t){  
2     this.note = n;  
3     this.type = t;  
4 }  
5  
6 var note1 = new Note("Algorithms",1);  
7 var note2 = new Note("Databases",3);  
8  
9 note1.setDueIn = function(x){  
10    this.dueIn = x;  
11 }  
12  
13 note2.setDueIn = function(x){  
14    this.dueIn = x;  
15 }  
16  
17 note1.setDueIn == note2.setDueIn; //true or false?  
18 note1.setDueIn.toString() == note2.setDueIn.toString(); //true or false?
```

what if another JavaScript library used in the project defines Note?

Module

- Goals:
 - **Do not declare global variables** unless required
 - Emulate **private/public** membership
 - Expose only the **necessary** properties (as API)
- Results:
 - Reduce potential **conflicts** with other JS libraries
 - Public API **minimizes** unintentional side-effects

Module

```
1 /* creating a module */
2 var gameStatModule = ( function() {
3
4     /* private members */
5     var gamesStarted = 0;
6     var gamesCompleted = 0;
7     var gamesAbolished = 0;
8
9     /* public members: return accessible object */
10    return {
11        incrGamesStarted : function(){
12            gamesStarted++;
13        },
14        getNumGamesStarted : function(){
15            return gamesStarted;
16        }
17    }
18 })();
19
20 /* using the module */
21 gameStatModule.incrGamesStarted();
22 console.log( gameStatModule.getNumGamesStarted() ) //public
23 console.log( gameStatModule.gamesStarted ); //private
```

public
private

Module

```
1 /* creating a module */
2 var gameStatModule = ( function() {
3
4     /* private members */
5     var gamesStarted = 0;
6     var gamesCompleted = 0;
7     var gamesAbolished = 0;
8
9     /* public members: return accessible object */
10    return {
11        incrGamesStarted : function(){
12            gamesStarted++;
13        },
14        getNumGamesStarted : function(){
15            return gamesStarted;
16        }
17    }
18 })();
19
20 /* using the module */
21 gameStatModule.incrGamesStarted();
22 console.log( gameStatModule.getNumGamesStarted() )public
23 console.log( gameStatModule.gamesStarted ); //prinprivate
```

Only the returned object has access to `gamesStarted`, etc. through JavaScript's concept of **closures**.

A closure is the combination of a function and the lexical environment within which that function was declared.

```
(function() {  
    ...  
})();
```

Immediately Invoked Function Expression (IIFE)

Module

The encapsulating function can contain arguments.

The pattern can be arbitrarily complex.

```
1 /* creating a module */
2 var gameStatModule = function(s, c, a) {
3
4     /* private members */
5     var gamesStarted = s;
6     var gamesCompleted = c;
7     var gamesAbolished = a;
8
9     /* public members: return
10    * accessible object
11    */
12    return {
13        incrGamesStarted : function(){
14            gamesStarted++;
15        },
16        getNumGamesStarted : function(){
17            return gamesStarted;
18        }
19    }
20 }(1, 1, 1);
```

Summary: module

- Advantages:
 - **Encapsulation** is achieved
 - Object properties are **public** or **private**
- Issues:
 - Changing the membership type 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);
```

- Course book uses **jQuery** extensively
- **Important to understand** what jQuery covers up

Which of the following
expressions evaluates to true
when jQuery is being used?

- A. `$ == jQuery` ✓
- B. `$ == window`
- C. `$ == document`
- D. `$ == object`

A look at book chapter 4

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

- **With jQuery:** no matter if class or id or ..., the access pattern is the same, i.e. `$()`
- **Without jQuery:**
 - `document.getElementById`
 - `document.getElementsByClassName`
 - `document.getElementsByTagName`
 - `document.querySelector`
 - `document.querySelectorAll`
- **Callback principle: we define what happens when an event fires**

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 function: what happens when the event fires?
(e.g. a popup appears)
4. Attach the function to the event **on** the control

Browser developer tooling

The screenshot shows a browser window with developer tools open. The top bar includes standard browser controls like back, forward, and search, along with zoom and other settings.

The main content area displays a simple HTML page with a button labeled "Say Hello World" and an input field showing "Hello World!".

The developer tools toolbar at the top includes tabs for Inspector, Console, Debugger, Style Editor, Performance, Memory, Network, Storage, and more.

The left sidebar shows the DOM tree:

```
<!DOCTYPE html>
<html class="gr__"> event
  <head>...</head>
  <body data-gr-c-s-loaded="true">
    <button onclick=" var tb = document.getElementById('out'); tb.value = 'Hello World!'>Say Hello World</button> event
    <input id="out" type="text" value="Hello World!"> click ...tHub/Web-Teaching/demo-code/lecture3-examples/1.html Bubbling DOM0
  </body>
</html>
```

A callout box highlights the `onclick` event handler for the button, showing the JavaScript code:

```
function onclick(event) {
  var tb = document.getElementById('out');
  tb.value = 'Hello World!'
}
```

The right sidebar shows the CSS Rules panel, which lists a single rule for the body element:

```
element { }
```

The bottom status bar shows the path: `html.gr__ > body > button`.

Interaction via the DOM

The DOM is our entry point to interactive web applications. It allows us to:

- **Extract an element's state**
 - Is the checkbox checked?
 - Is the button disabled?
 - Is a `<h1>` appearing on the page?
- **Change an element's state**
 - Check a checkbox
 - Disable a button
 - Create an `<h1>` element on a page if none exists
- **Change an element's style** (material for a later lecture)
 - Change the color of a button
 - Change the size of a paragraph
 - Change the background color of a web application

Client-side JS examples

small and self-contained.

Available in the [Lecture notes](#)
(an alternative to Ch. 4).

getElementsBy

1

mouse events

4

creating DOM nodes

2

crowdsourcing interface

5

this

3

typing game

6

- Read **chapters 5 & 6** (Node.js) of the Web development book **before** this Thursday's lecture!
- Complete **Assignment 1**.
- Get started on **Assignment 2**.