

JavaScript

JavaScript

A yellow square containing the letters 'JS' in a bold, black, sans-serif font.

- Historiquement permet de programmer des interactions au sein des navigateurs
 - **Interagir** : savoir qu'un bouton a été cliqué
 - **Afficher** : manipuler la page web pour rendre visible des nouvelles parties
 - **Communiquer** : envoyer ou recevoir des requêtes
- **C'est le seul langage disponible côté navigateur**
- Mais est aussi disponible côté serveur via NodeJS

Exemple

```
<!doctype html>
<html lang="fr">
  <head>
    <title>Exemple</title>
    <script>
      var msg = "hello";
      alert(msg);
    </script>
  </head>
  <body>
  </body>
</html>
```



Best practice

```
<!doctype html>  
<html lang="fr">  
  <head>  
    <title>Example</title>  
    <script src="script.js"></script>  
  </head>  
  <body>  
  </body>  
</html>
```



Messages de log

L'instruction qui permet d'afficher des messages de log en JavaScript est :

script.js

```
console.log('Hello, world!');
```

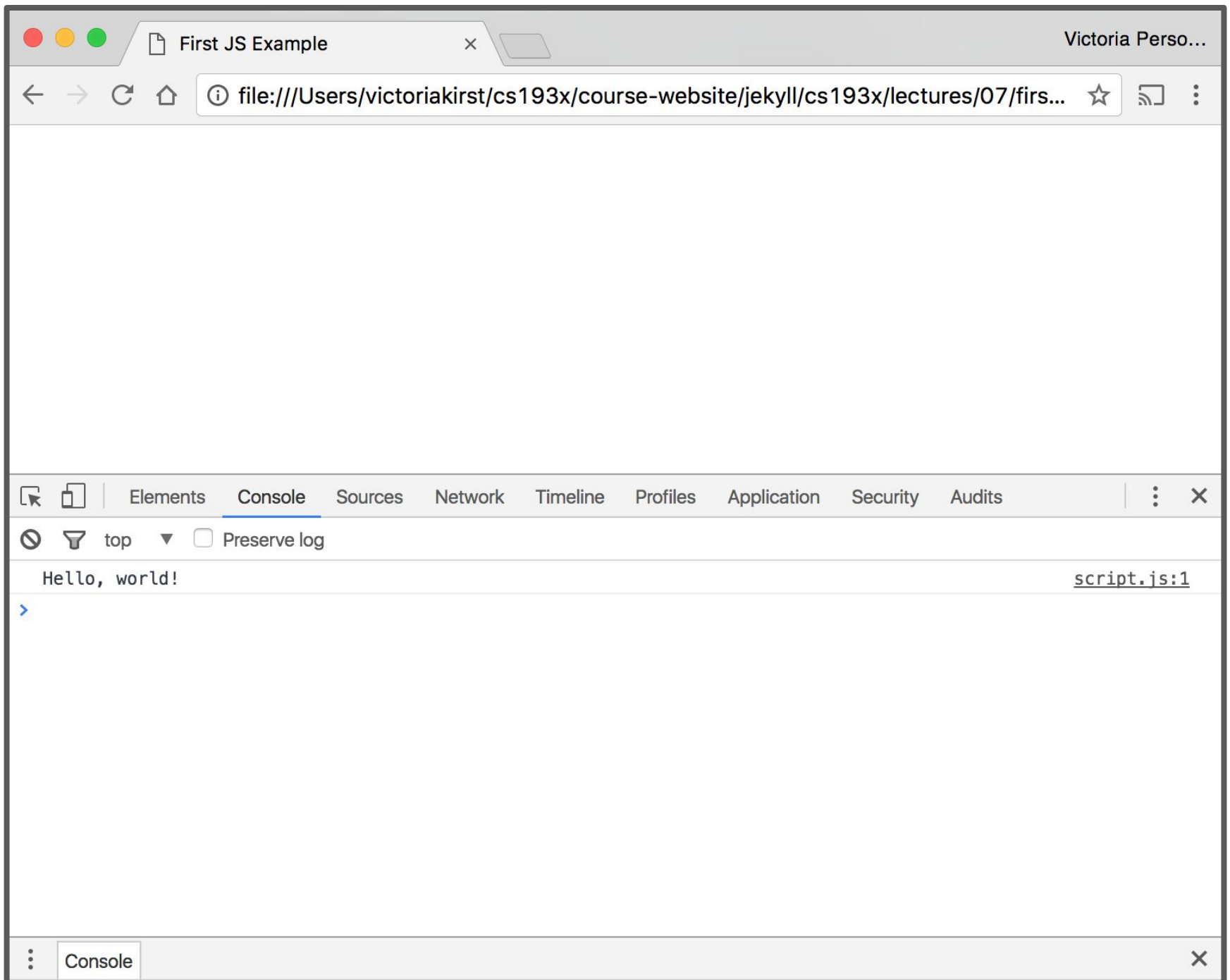
Exécution de JavaScript

Il n'y a pas de "**main method**"

- Le script est exécuté de haut en bas

Il n'y a pas de **compilation** par le développeur

- JavaScript est compilé et exécuté à la volée par le navigateur



Similarités avec Java, C, ...

for-loops:

```
for (let i = 0; i < 5; i++) { ... }
```

while-loops:

```
while (notFinished) { ... }
```

comments:

```
// comment or /* comment */
```

conditionals (if statements):

```
if (...) {  
    ...  
} else {  
    ...  
}
```

Fonctions

La syntaxe suivante est une des manières de définir une fonction en JavaScript

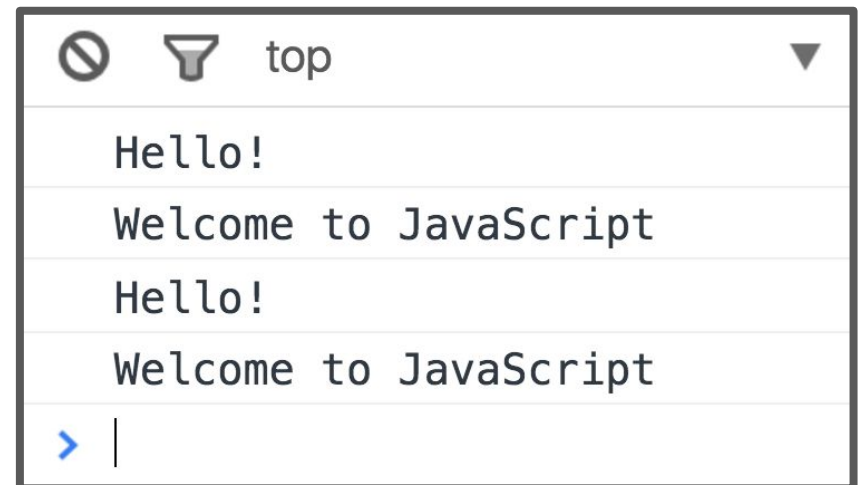
```
function name() {  
    statement;  
    statement;  
    ...  
}
```

Exemple

```
hello();  
hello();  
  
function hello() {  
  console.log('Hello!');  
  console.log('Welcome to JavaScript');  
}
```

Cela fonctionne car les déclarations de fonctions sont ***hoisted*** : déplacées au sommet du scope dans lesquelles elles sont définies

il faut éviter de se reposer sur ce mécanisme



Variables

Trois façons de déclarer des variables en JS

```
// Function scope variable  
var x = 15;  
// Block scope variable  
let fruit = 'banana';  
// Block scope constant; cannot be reassigned  
const isHungry = true;
```

Le langage est dynamiquement typé

Paramètres de fonctions

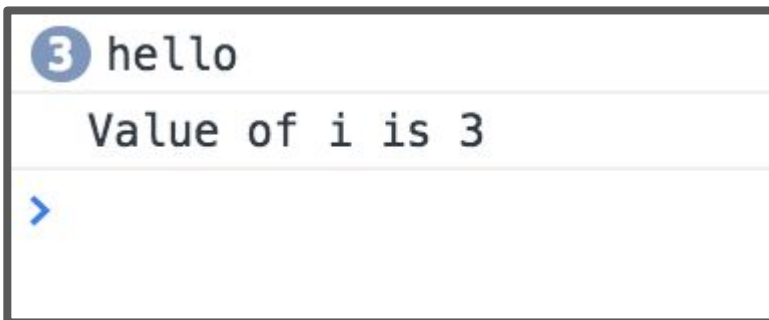
```
function printMessage(message, times) {  
  for (var i = 0; i < times; i++) {  
    console.log(message);  
  }  
}
```

Les paramètres de fonctions ne sont pas déclarés à l'aide de `let`, `const` ou `var`

Comprendre var

```
function printMessage(message, times) {  
  for (var i = 0; i < times; i++) {  
    console.log(message);  
  }  
  console.log('Value of i is ' + i);  
}
```

```
printMessage('hello', 3);
```



A screenshot of a JavaScript console window. It shows three lines of output: a blue circle with the number 3 followed by the text 'hello', the text 'Value of i is 3', and a blue prompt character '>' on the third line.

```
3 hello  
Value of i is 3  
>
```

La valeur de `i` est accessible hors de la boucle `for` car `var` déclare des variables avec un scope *fonction*

Comprendre let

```
function printMessage(message, times) {  
  for (let i = 0; i < times; i++) {  
    console.log(message);  
  }  
  console.log('Value of i is ' + i);  
}
```

```
printMessage('hello', 3);
```

3 hello

✖ ▶ Uncaught ReferenceError: i is not defined
 at printMessage (script.js:5)
 at script.js:8

>

La valeur de `i` n'est pas accessible hors de la boucle `for` car `let` déclare des variables avec un scope *block*

Comprendre const

```
const y = 10;  
y = 0;           // error!  
y++;            // error!  
const list = [1, 2, 3];  
list.push(4);    // OK
```

Les variables déclarées avec `const` ne peuvent pas être réaffectées

Cependant il reste possible de modifier l'objet sous jacent

- Ressemble au `final` de Java

Bonnes pratiques

- Utiliser `const` partout où vous pouvez
- Si vous avez besoin d'une variable réaffectable, utilisez `let`
- **N'utilisez pas `var`.**
 - `const` et `let` sont maintenant bien supportés par les browsers

Types

Les **variables** JS n'ont pas de types, mais leurs **valeurs** si

Il y a plusieurs types primitifs:

- **Boolean** : true et false
- **Number** : tout est de type double (pas d'entiers)
- **String**: avec 'single' ou "double-quotes"
- **Null**: null une valeur qui signifie “ceci n’a pas de valeur”
- **Undefined**: la valeur d’une variable non affectée

Il y a aussi les types Object, comme Array, Date, et même Function!

Transtypage booléen

Les valeurs non booléennes peuvent être utilisées dans les instructions de contrôle, elles sont converties en "truthy" ou "falsy"

- `null`, `undefined`, `0`, `NaN`, `' '` évaluent à `false`
- Les autres valeurs évaluent à `true`

```
if (username) {  
    // username is defined  
}
```

Egalité

`==` et `!=` font une conversion implicite de type avant comparaison

```
' ' == '0' // false
' ' == 0   // true
0  == '0'  // true
NaN == NaN // false
[ ' ' ] == ' ' // true
false == undefined // false
false == null      // false
null == undefined  // true
```

Opérateurs === et !==

=== et !== sont les véritables opérateurs de comparaison,
toujours les utiliser!

```
' ' === '0'    // false
' ' === 0      // false
0 === '0'      // false
NaN === NaN    // still weirdly false
[ ' ' ] === ' ' // false
false === undefined // false
false === null  // false
null === undefined // false
```

null et undefined

Quelle différence?

- `null` est la valeur représentant l'absence de valeur (comme `null` en Java)
- `undefined` est la valeur d'une variable n'ayant pas reçu de valeur

```
let x = null;  
let y;  
console.log(x);  
console.log(y);
```

`null`

`undefined`



Tableaux

Les tableaux sont les objets pour définir des listes

```
// Creates an empty list
const list = [];
const groceries = ['milk', 'cocoa puffs'];
groceries[1] = 'kix';

// For each loop
for (let item of groceries) {
  console.log(item);
}
```

Objets

Collection de paires clé - valeur, peut être utilisé comme une hashmap

```
const prices = {};  
const scores = {  
  peach: 100,  
  mario: 88,  
  luigi: 91  
};  
console.log(scores['peach']);    // 100  
console.log(scores.peach);      // 100  
scores.peach = 20;  
console.log(scores.peach);      // 20
```

Itérer les propriétés d'un objet

Il est possible d'itérer sur les propriétés d'un objet

```
const scores = {  
  peach: 100,  
  mario: 88,  
  luigi: 91  
};
```

```
for (let name in scores) {  
  console.log(name + ':' + scores[name]);  
}
```

Fonctions de premier ordre

En JavaScript, les fonctions sont des objets comme les autres

```
var add = function(a, b) {  
    return a + b;  
}
```

```
add(2, 2); // 4
```

Ajout d'une fonction sur un objet

On peut ajouter une fonction dans un objet, `this` désigne l'objet receveur

```
const scores = {  
  peach: 100,  
  mario: 88,  
  luigi: 91,  
  total: function() {  
    return this.peach + this.mario + this.luigi;  
  }  
};
```

Callbacks

Les fonctions peuvent être passées en paramètres d'autres fonctions

```
const groceries = ['milk', 'cocoa puffs'];

// For each loop
for (let item of groceries)
  console.log(item);

// Callback style
groceries.forEach(function(item) {
  console.log(item);
});

// Nerdy callback style (use that!)
groceries.forEach(item => {
  console.log(item);
});
```

Mais que fait forEach?

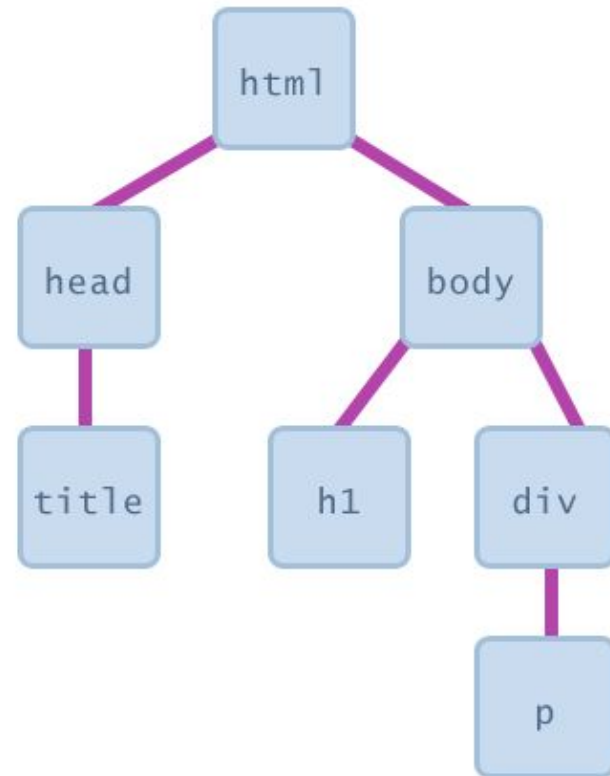
```
class Array {  
  forEach(callback) {  
    for (let i = 0; i < this.length; i++) {  
      callback(this[i], i, this);  
    }  
  }  
}
```

forEach itère sur tous les éléments du tableau, pour chaque élément, elle applique la fonction callback (passée en paramètre de forEach) en lui fournissant en paramètre l'élément courant **forEach calls callback back!**

DOM

On accède aux éléments composant la page web en JavaScript au travers du **Document Object Model**

```
<html>
  <head>
    <title></title>
  </head>
  <body>
    <h1></h1>
    <div>
      <p></p>
    </div>
  </body>
</html>
```



DOM et JavaScript

JavaScript peut :

- **examiner** les noeuds du DOM pour en inspecter l'état (ex. voir le texte saisi par un utilisateur)
- **editer** les attributs des noeuds du DOM (ex. changer le style d'un élément <h1>)
- **ajouter ou supprimer** des noeuds du DOM (ex. ajouter un texte de statut quelque part)

HTML et JavaScript

Mais n'oubliez pas raison !

Fonctionnalité « formulaire » marche avec HTML+HTTP
GOV.UK — vrais utilisateurs sur XBox/Playstation/etc.

Pas de bons navigateurs !

Mais un site bien fait est utilisable

[https://shkspr.mobi/blog/2021/01/
the-unreasonable-effectiveness-of-simple-html/](https://shkspr.mobi/blog/2021/01/the-unreasonable-effectiveness-of-simple-html/)

Accéder aux noeuds du DOM

L'accès aux noeuds du DOM se fait via la fonction `querySelector`:

```
document.querySelector('css selector');
```

- Retourne le **premier** noeud qui matche la règle CSS

```
document.querySelectorAll('css selector');
```

- Retourne **tous** les éléments qui matchent la règle CSS

Quelques propriétés des noeuds du DOM

Property	Description
id	la valeur de l'attribut id de l'élément
innerHTML	le HTML entre le noeud ouvrant et fermant de l'élément vu comme une chaîne de caractères
textContent	Le contenu texte d'un noeud ainsi que celui de ses descendants
classList	Un objet contenant toute la liste des classes dans lesquelles l'élément se situe

Créer des éléments dans le DOM

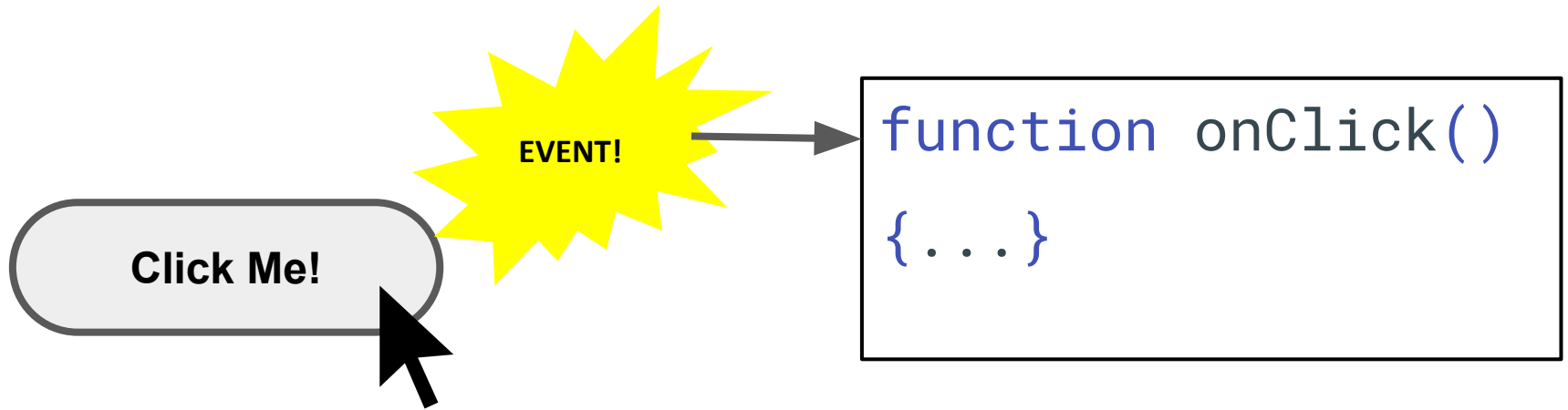
Il est possible de créer des éléments dynamiquement via `createElement` et `appendChild`:

```
document.createElement(tag string)  
    element.appendChild(element);
```

On peut supprimer des éléments via `remove`

```
element.remove();
```

Évènements



Exemple : un élément de page avec lequel on peut interagir. Quand on clique sur le bouton, un événement est déclenché. L'utilisateur peut enregistrer des callbacks sur les événements de son choix

```
<html>
  ▼<head>
    <meta charset="utf-8">
    <title>First JS Example</title>
    <script src="script.js" defer</script>
  </head>
  ▼<body>
    <button>Click Me!</button>
  </body>
</html>
```

```
function onClick() {
  console.log('clicked');
}

const button = document.querySelector('button');
button.addEventListener('click', onClick);
```

Click Me!



Elements

Console



top

clicked



JavaScript events in detail

Events in JavaScript

If you put a "click" event listener on an element, what happens if the user clicks a *child* of that element?

```
<div class="show-details">  
    
    
    
  Click me!  
  <div id="inner">  
    No, click me!  
  </div>  
</div>
```

([CodePen](#))

```
const outer = document.querySelector('#outer');  
const inner = document.querySelector('#inner');  
outer.addEventListener('click', onOuterClick);  
inner.addEventListener('click', onInnerClick);
```

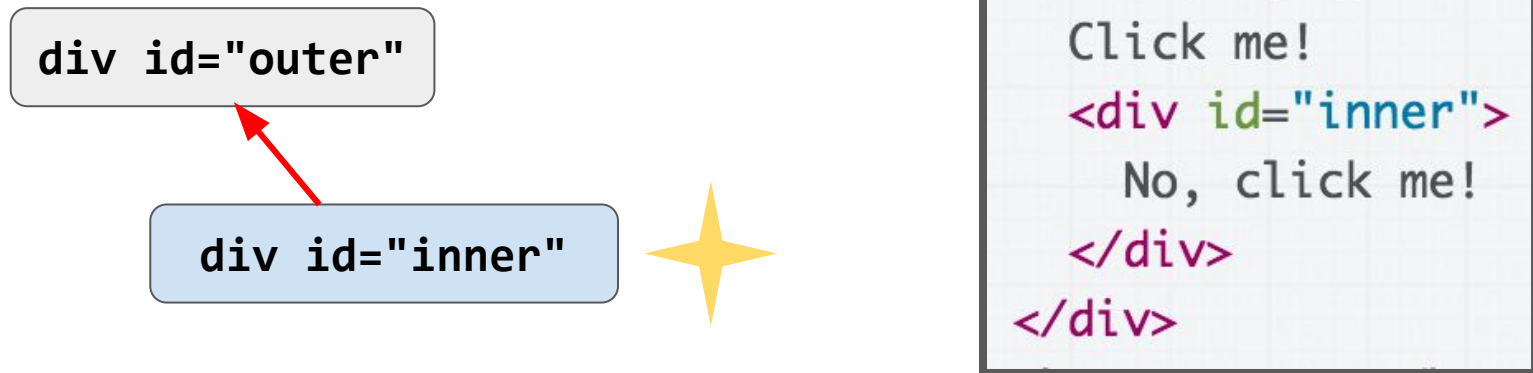
Click me!

No, click me!

Reset

Event bubbling

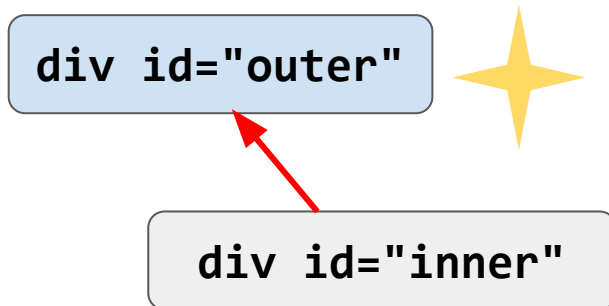
- Both events fire if you click the inner element
- By default, the event listener on the inner-most element fires first



This event ordering (inner-most to outer-most) is known as **bubbling**. ([CodePen](#))

Event bubbling

- Both events fire if you click the inner element
- By default, the event listener on the inner-most element fires first



```
<div id="outer">  
  Click me!  
  <div id="inner">  
    No, click me!  
  </div>  
</div>
```

This event ordering (inner-most to outer-most) is known as **bubbling**. ([CodePen](#))

stopPropagation()

We can stop the event from bubbling up the chain of ancestors by using ***event***.stopPropagation():

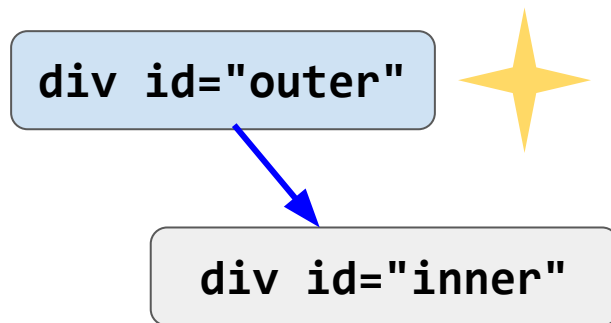
```
function onInnerClick(event) {  
    inner.classList.add('selected');  
    console.log('Inner clicked!');  
    event.stopPropagation();  
}
```

See [default behavior](#) vs with [stopPropagation](#)

Event capturing

To make event propagation go the opposite direction, add a [3rd parameter](#) to `addEventListener`:

```
event.addEventListener(  
  'click', onClick, { capture: true } );
```



```
<div id="outer">  
  Click me!  
  <div id="inner">  
    No, click me!  
  </div>  
</div>
```

This event ordering (outer-most to inner-most) is known as **capturing**. ([CodePen](#))

Event capturing

To make event propagation go the opposite direction, add a [3rd parameter](#) to `addEventListener`:

```
event.addEventListener(  
    'click', onClick, { capture: true } );
```

`div id="outer"`



`div id="inner"`



```
<div id="outer">  
  Click me!  
  <div id="inner">  
    No, click me!  
  </div>  
</div>
```

This event ordering (outer-most to inner-most) is known as **capturing**. ([CodePen](#))

stopPropagation()

We can also use ***event***.stopPropagation() in capture-order:

```
function onOuterClick(event) {  
  outer.classList.add('selected');  
  console.log('Outer clicked!');  
  event.stopPropagation();  
}
```

See [default behavior](#) vs with [stopPropagation](#)

Some technical details...

Behind the scenes

Technically, the browser will go through **both** a capture phase and a bubbling phase when an event occurs:

```
<html>
  <head>
    <meta charset="utf-8">
    <title>JS Events: Two event listeners</title>
  </head>
  <body>
    <div id="outer">
      Click me!
      <div id="inner">
        No, click me!
      </div>
    </div>
    <button>Reset</button>
  </body>
</html>
```

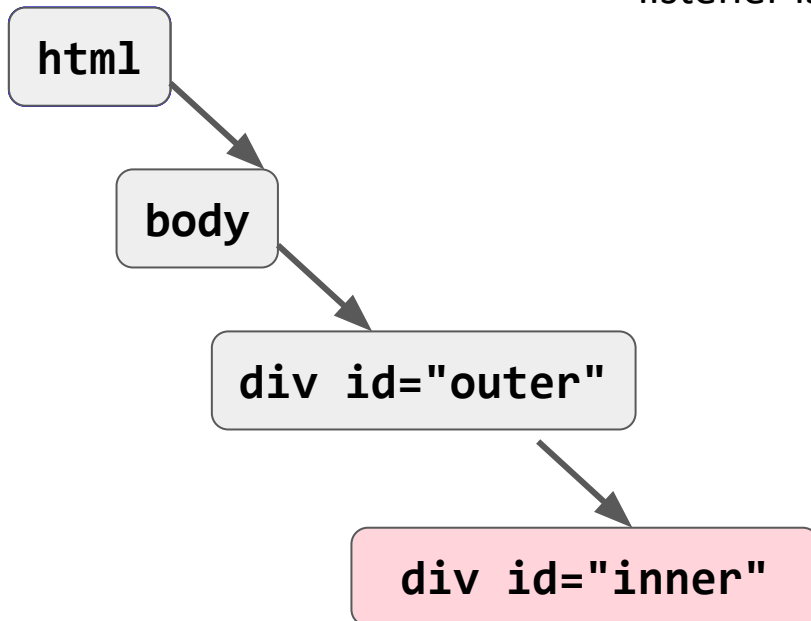


If we click on the div with id="inner"...

Behind the scenes

The browser creates the target's "**propagation path**," or the list of its ancestors up to root ([w3c](#))

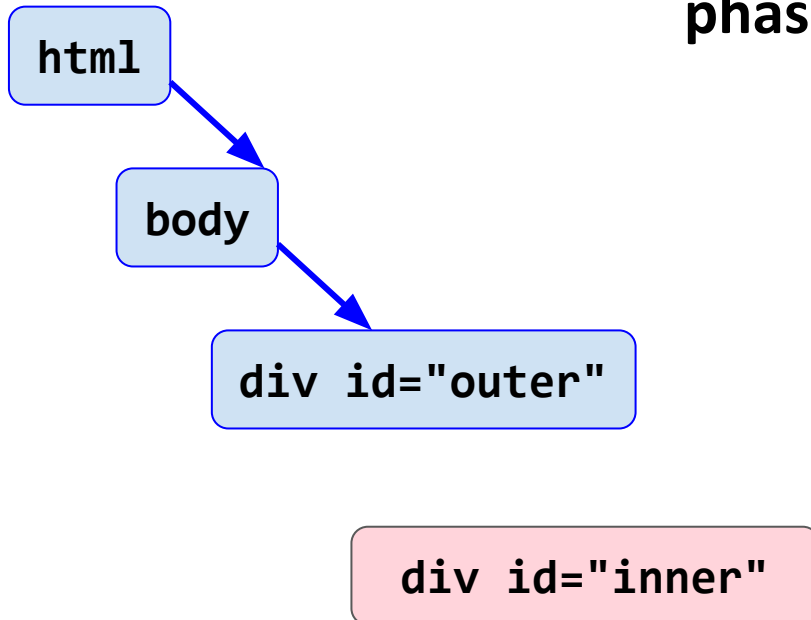
(target meaning the thing you clicked; not necessarily the element the event listener is attached to)



```
<html>
  <head>
    <meta charset="utf-8">
    <title>JS Events: Two event list
  </head>
  <body>
    <div id="outer">
      Click me!
      <div id="inner">
        No, click me!
      </div>
    </div>
    <button>Reset</button>
  </body>
```

"Capture phase"

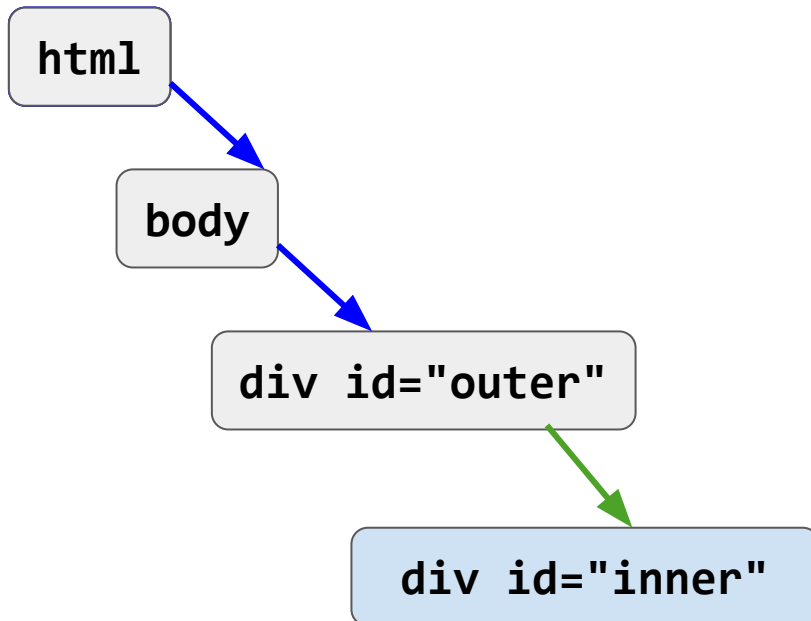
The browser begins at the top of the propagation path and invokes any event listeners that have `capture="true"`, in path order until it gets to the target. This is the "**capture phase**" ([w3c](#))



```
<html>
  <head>
    <meta charset="utf-8">
    <title>JS Events: Two event list
  </head>
  <body>
    <div id="outer">
      Click me!
      <div id="inner">
        No, click me!
      </div>
    </div>
    <button>Reset</button>
  </body>
```

"Target phase"

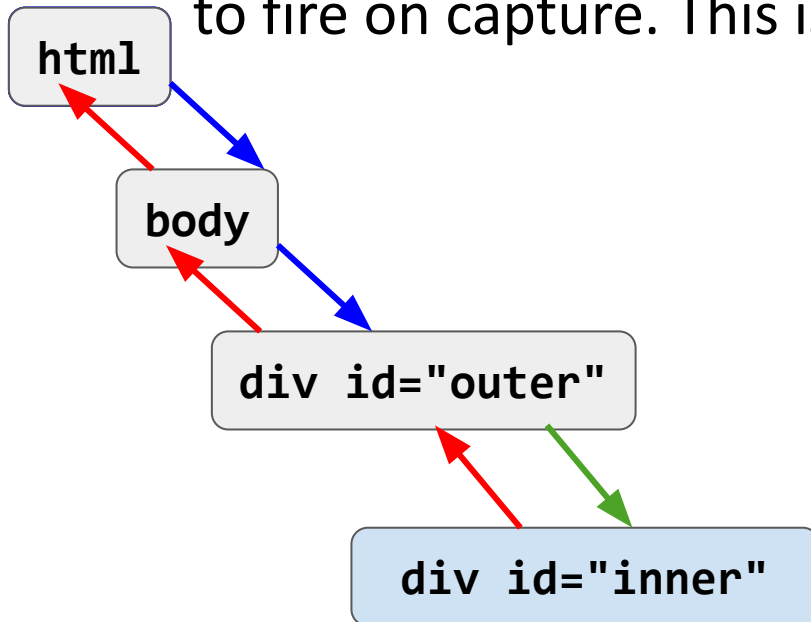
Then the browser invokes any event listener that was set on the target itself. This is the "target phase" ([w3c](https://www.w3c.org))



```
<html>
  <head>
    <meta charset="utf-8">
    <title>JS Events: Two event list
  </head>
  <body>
    <div id="outer">
      Click me!
      <div id="inner">
        No, click me!
      </div>
    </div>
    <button>Reset</button>
  </body>
```

"Bubble phase"

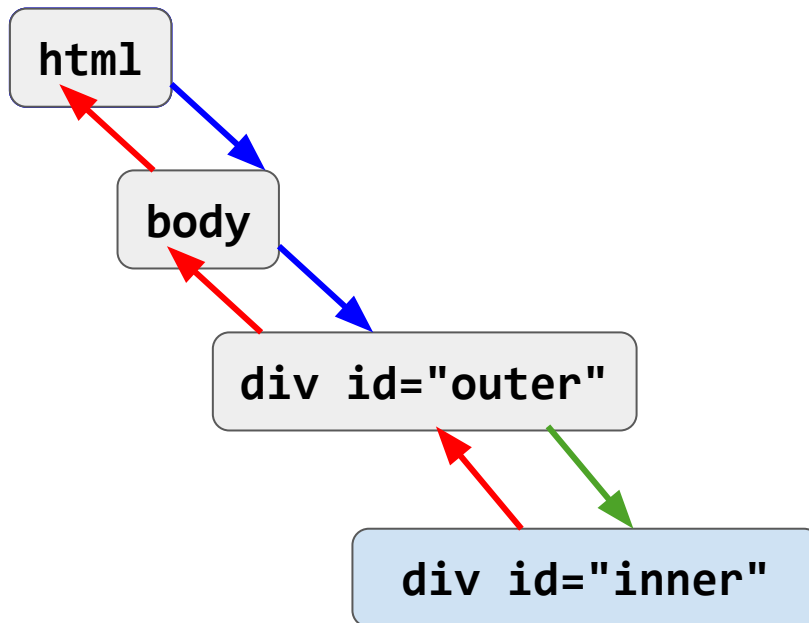
If the event type has `bubbles=true` (see [click](#), e.g.) the browser goes back up the propagation path in reverse order and invokes any event listener that wasn't supposed to fire on capture. This is the "**bubble phase**" ([w3c](#))



```
<html>
  <head>
    <meta charset="utf-8">
    <title>JS Events: Two event list
  </head>
  <body>
    <div id="outer">
      Click me!
      <div id="inner">
        No, click me!
      </div>
    </div>
    <button>Reset</button>
  </body>
```

stopPropagation()

Therefore `stopPropagation()` actually stops the rest of the 3-phase dispatch from executing



In Practice

Don't worry about:

- You never need to use capture order - you can always use bubbling
- You don't really need to know how the browser goes through "capture phase", "target phase", then "bubble phase"

Do worry about:

- **You do need to understand bubbling, though**
- `stopPropagation()` also comes in handy

JS prototypes

Classical (for JS) approach to objects/inheritance

Classes added to ECMA script 2015–2019

Prototype-based code is used a lot, and still being written

You're going to need to debug this one day

JS Objects

Object — associative array, dictionary, hash table

Can create just as a literal: `x={}`

Can always add a field: `x.y = 1`

Can index: `f="y"; x[f]`

JS functions

`this` refers to the current context

In object methods: the current instance

In browser: might be current window

Prototypes: idea

Object — associative array, dictionary, hash table

If `x.z` does not exist?

→ undefined

Fallback: prototype

```
A = {a: 1}
```

```
A.b // undefined
```

```
B = {b: 2}
```

```
B.b // 2
```

```
A.__proto__ = B
```

```
A.b // 2
```

Chained:

```
C = {c: 3}
```

```
B.__proto__ = C
```

```
A.c // 3
```

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Inheritance_and_the_prototype_chain

Prototypes: idea

Object — associative array, dictionary, hash table

If `x.z` does not exist?

→ undefined

Fallback: prototype

```
A = {a: 1}
```

```
A.b // undefined
```

```
B = {b: 2}
```

```
B.b // 2
```

```
A.__proto__ = B
```

```
A.b // 2
```

Chained:

```
C = {c: 3}
```

```
B.__proto__ = C
```

```
A.c // 3
```

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Inheritance_and_the_prototype_chain

Prototypes: idea

Object — associative array, dictionary, hash table

If `x.z` does not exist?

→ undefined

Fallback: prototype

```
A = {a: 1}
```

```
A.b // undefined
```

```
B = {b: 2}
```

```
B.b // 2
```

```
A.__proto__ = B
```

```
A.b // 2
```

Chained:

```
C = {c: 3}
```

```
B.__proto__ = C
```

```
A.c // 3
```

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Inheritance_and_the_prototype_chain

Classical «new»

```
function Car(model) {  
  this.model = model;  
}
```

```
Car.prototype.colour = "black"; // not __proto__
```

```
const car1 = new Car("308");
```

```
car1.model; // "308"
```

```
car1.colour; // "black"
```

Classical «new»: definition

`new f(a1, a2)`

- Create new object

- Set its `__proto__` to `f.prototype`

 - If not `Object` — fallback to `Object.prototype`

- Make this object `this` for execution of call to `f`

- Result is normally that new object

 - If `f` returns non-primitive value, use it

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/new>

Classes in JavaScript

Amateur JavaScript

Your code may look like this:

- Mostly all in one file
- All global functions
- Global variables to save state between events

It would be nice to write code in a **modular** way...

```
1 //
2 // Album view functions
3 //
4 let currentIndex = null;
5 function onThumbnailClick(event) {
6   currentIndex = event.currentTarget.dataset.index;
7   const image = createImage(event.currentTarget.src);
8   showFullSizeImage(image);
9   document.body.classList.add('no-scroll');
10  modalView.style.top = window.pageYOffset + 'px';
11  modalView.classList.remove('hidden');
12 }
13
14 //
15 // Photo view functions
16 //
17 function createImage(src) {
18   const image = document.createElement('img');
19   image.src = src;
20   return image;
21 }
22
23 function showFullSizeImage(image) {
24   modalView.innerHTML = '';
25
26   image.addEventListener('pointerdown', startDrag);
27   image.addEventListener('pointermove', duringDrag);
28   image.addEventListener('pointerup', endDrag);
29   image.addEventListener('pointercancel', endDrag);
30   modalView.appendChild(image);
31 }
32
33 let originX = null;
34 function startDrag(event) {
35   event.preventDefault();
36   // Needed so clicking on picture doesn't cause modal dialog to close.
37   event.stopPropagation();
38
39   originX = event.clientX;
40   event.target.setPointerCapture(event.pointerId);
41 }
42
43 function duringDrag(event) {
44   if (originX) {
45     const currentX = event.clientX;
46     const delta = currentX - originX;
47     const element = event.currentTarget;
48     element.style.transform = `translateX(${delta + 'px'})`;
49   }
50 }
51
52 function endDrag(event) {
53   if (!originX) {
54     return;
55   }
56
57   const currentX = event.clientX;
58   const delta = currentX - originX;
59   originX = null;
60
61   let nextIndex = currentIndex;
62   if (delta < 0) {
63     nextIndex++;
64   } else {
65     nextIndex--;
66   }
67
68   if (nextIndex < 0 || nextIndex == PHOTO_LIST.length) {
69     event.currentTarget.style.transform = '';
70     return;
71   }
72 }
```

ES6 classes

We can define **classes** in JavaScript using a syntax that is similar to Java or C++:

```
class ClassName {  
  constructor(params) {  
    ...  
  }  
  methodName() {  
    ...  
  }  
  methodName() {  
    ...  
  }  
}
```

These are often called "**ES6 classes**" or "**ES2015 classes**" because they were introduced in the EcmaScript 6 standard, the 2015 release

- Recall that EcmaScript is the standard; JavaScript is an implementation of the EcmaScript standard

Wait a minute...

Wasn't JavaScript created in 1995?

And classes were introduced... 20 years later in 2015?

**Q: Was it seriously not possible to create
classes in JavaScript before 2015?!**

Objects in JavaScript

In JavaScript, there are several ways to create blueprints for objects. Two broad approaches:

1. Functional

- a. This approach has existed since the creation of the JavaScript
- b. Weird syntax for people used to languages like Java, C++, Python
- c. Doesn't quite behave the same way as objects in Java, C++, Python

2. Classical

- a. This is the approach that just got added to the language in 2015
- b. Actually just "[syntactic sugar](#)" over the functional objects in JavaScript, so still a little weird
- c. But syntax is much more approachable

Objects in JavaScript

In JavaScript, there are several ways to create blueprints for objects. Two broad approaches:

1. Functional

- a. This approach has existed since the creation of the JavaScript
- b. Weird syntax for people used to languages like Java, C++, Python
- c. Doesn't quite behave the same way as objects in Java, C++, Python

2. Classical

- a. This is the approach that just got added to the language in 2015
- b. Actually just "[syntactic sugar](#)" over the functional objects in JavaScript, so still a little weird
- c. But syntax is much more approachable

This approach is quite controversial.

Class controversy

"There is one thing I am certain is a bad part, a very terribly bad part, and that is the new class syntax [in JavaScript]... [T]he people who are using `class` will go to their graves never knowing how miserable they were." ([source](#))

-- Douglas Crockford, author of *JavaScript: The Good Parts*; prominent speaker on JavaScript; member of [TC39](#) (committee that makes ES decisions)



Public methods

```
class ClassName {  
    constructor(params) {  
        ...  
    }  
    methodName() {  
        ...  
    }  
    methodName() {  
        ...  
    }  
}
```

constructor is optional.

Parameters for the constructor and methods are defined in the same way they are for global functions.

You do not use the `function` keyword to define methods.

Public methods

```
class ClassName {  
    constructor(params) {  
        ...  
    }  
    methodOne() {  
        this.methodTwo();  
    }  
    methodTwo() {  
        ...  
    }  
}
```

Within the class, you must always refer to other methods in the class with the **this.** prefix.

Public methods

```
class ClassName {  
    constructor(params) {  
        ...  
    }  
    methodName() {  
        ...  
    }  
    methodName() {  
        ...  
    }  
}
```

All methods are **public**, and you **cannot** specify private methods... yet.

Public fields

```
class ClassName {  
    constructor(params) {  
        this.fieldName = fieldValue;  
        this.fieldName = fieldValue;  
    }  
  
    methodName() {  
        this.fieldName = fieldValue;  
    }  
}
```

Define public fields by setting **this.*fieldName*** in the constructor... or in any other function.

Class fields : ES2019

```
class MyClass {  
  name = "toto";  
  
  constructor(age) {  
    this.age = age;  
  }  
}
```

Class fields are public

Only since ES2019 - check compatibility!

Public fields

```
class ClassName {  
  constructor(params) {  
    this.someField = someParam;  
  }  
  methodName() {  
    const someValue = this.someField;  
  }  
}
```

Within the class, you must always refer to fields with the **this.** prefix.

Private fields : ES2019

```
class MyClass {  
  #name = "toto";  
  constructor(age) {  
    this.age = age;  
  }  
  display() {  
    console.log(this.#name + " : " + this.age);  
  }  
}  
  
let m = new MyClass(22);  
m.display();  
  
// console.log(m.#name + " : " + m.age); // ERROR
```

Only since ES2019 - check compatibility!

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes/Private_class_field

Getter and setter : since 2015

```
class Lang {  
  log = [];  
  
  set current(name) {  
    this.log.push(name);  
  }  
  
  get latest() {  
    if (this.log.length === 0) {  
      return undefined;  
    }  
  
    return this.log[this.log.length - 1];  
  }  
}
```

```
let lang = new Lang();  
console.log(lang.latest);  
lang.current = 'FR';  
lang.current = 'EN';  
console.log(lang.latest);  
console.log(lang.log);
```

Ouvrons nos cadeaux

<> index.html > ...

```
1  <!DOCTYPE html>
2  <html lang="en">
3    <head>
4      <meta charset="UTF-8" />
5      <title>Simple class: present</title>
6      <script src="scripts/main.js" defer></script>
7      <style>img { width: 200px; }</style>
8    </head>
9    <body>
10     <div id="presents"></div>
11   </body>
12 </html>
```

Ouvrons nos cadeaux

scripts > JS main.js > ...

```
1  class Present {
2      constructor(containerElement) {
3          this.containerElement = containerElement;
4
5          // Create image and append to container.
6          const image = document.createElement('img');
7          image.src = 'https://s3-us-west-2.amazonaws.com/s.cdpn.io/1083533/gift-icon.png';
8          image.addEventListener('click', this.openPresent);
9          this.containerElement.append(image);
10     }
11
12     openPresent(event) {
13         const image = event.currentTarget;
14         image.src = 'https://media.giphy.com/media/27ppQU0xe7KlG/giphy.gif';
15         image.removeEventListener('click', this.openPresent);
16     }
17 }
18
19 const presents = document.querySelector('#presents');
20 const present = new Present(presents);
```

Don't forget this

scripts > JS main.js > ...

```
1  class Present {
2    constructor(containerElement) {
3      this.containerElement = containerElement;
4
5      // Create image and append to container.
6      const image = document.createElement('img');
7      image.src = 'https://s3-us-west-2.amazonaws.com/s.cdpn.io/1083533/gift-icon.png';
8      image.addEventListener('click', this.openPresent);
9      this.containerElement.appendChild(image);
10   }
11
12   openPresent() {
13     // ...
14   }
15 }
16
17 }
18
19 const presents = document.querySelector('#presents');
20 const present = new Present(presents);
```

If the event handler function you are passing to `addEventListener` is a method in a class, you must pass `"this.functionName"`

this in event handler

```
12  openPresent(event) {  
13      const image = event.currentTarget;  
14      image.src = 'https://media.giphy.com/media/27ppQU0xe7KlG/giphy.gif';  
15      image.removeEventListener('click', this.openPresent);  
16  }
```

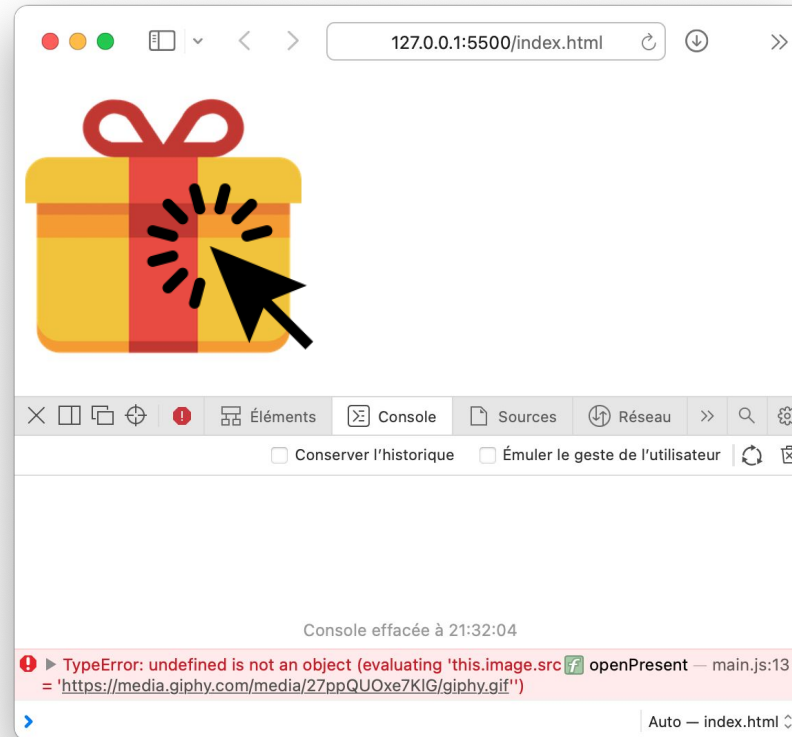
Right now we access the image we create in the constructor in **openPresent** via **event.currentTarget**

this in event handler

```
12 openPresent(event) {  
13     this.image.src = 'https://media.giphy.com/media/27ppQU0xe7KlG/giphy.gif';  
14     this.image.removeEventListener('click', this.openPresent);  
15 }
```

Q: What if we make the `image` a field and access it in `openPresent` via `this.image` instead of `event.currentTarget`?

this in event handler



Error message!

What's going on?

JavaScript `this`

The `this` keyword in JavaScript is **dynamically assigned**, or in other words: `this` means different things in different contexts ([mdn list](#))

- In our constructor, `this` refers to the instance
- When called in an event handler, `this` refers to... the element that the event handler was attached to ([mdn](#)).

this in event handler

```
12  openPresent(event) {  
13      this.image.src = 'https://media.giphy.com/media/27ppQU0xe7KlG/giphy.gif';  
14      this.image.removeEventListener('click', this.openPresent);  
15  }
```

That means `this` refers to the `<img>` element, not the instance variable of the class...

Console effacée à 21:32:04

❗ ▶ **TypeError: undefined is not an object (evaluating 'this.image.src = 'https://media.giphy.com/media/27ppQU0xe7KlG/giphy.gif')'**

openPresent — main.js:13

...which is why we get this error message.

Solution: bind

To make `this` always refer to the instance object for a method in the class (i.e. to get `this` to behave as you'd expect), you can add the following line of code in the constructor:

```
this.methodName = this.methodName.bind(this);
```

```
scripts > JS main.js >  Present >  constructor
```

```
1  class Present {  
2      constructor(containerElement) {  
3          this.containerElement = containerElement;  
4  
5          // Bind event listeners  
6          this.openPresent = this.openPresent.bind(this);
```

Solution: bind

Now **this** in the **openPresent** method refers to the instance object

```
12 openPresent(event) {  
13     this.image.src = 'https://media.giphy.com/media/27ppQU0xe7KlG/giphy.gif';  
14     this.image.removeEventListener('click', this.openPresent);  
15 }
```



Moral of the story:

**Don't forget to `bind()`
event listeners in your
constructor!!**

scripts > JS main.js > Present > constructor

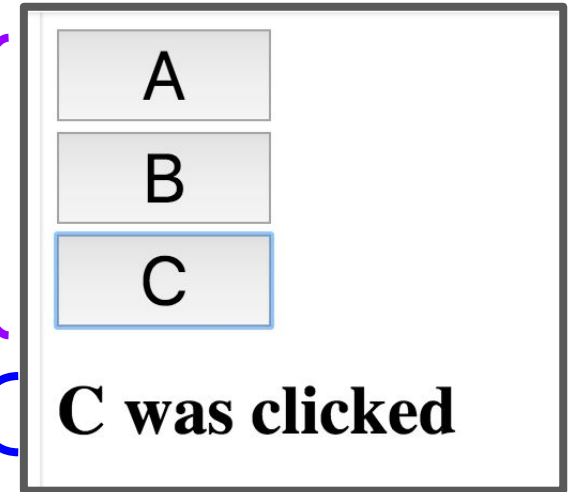
```
1  class Present {  
2      constructor(containerElement) {  
3          this.containerElement = containerElement;  
4  
5          // Bind event listeners  
6          this.openPresent = this.openPresent.bind(this);
```

One more time:

**Don't forget to `bind()`
event listeners in your
constructor!!**

Example: Buttons

```
<html>
  <head>
    <meta charset="utf-8">
    <title>Menu and buttons examples</title>
  </head>
  <body>
    <div id="menu"></div>
    <h1 id="status-bar"></h1>
  </body>
</html>
```



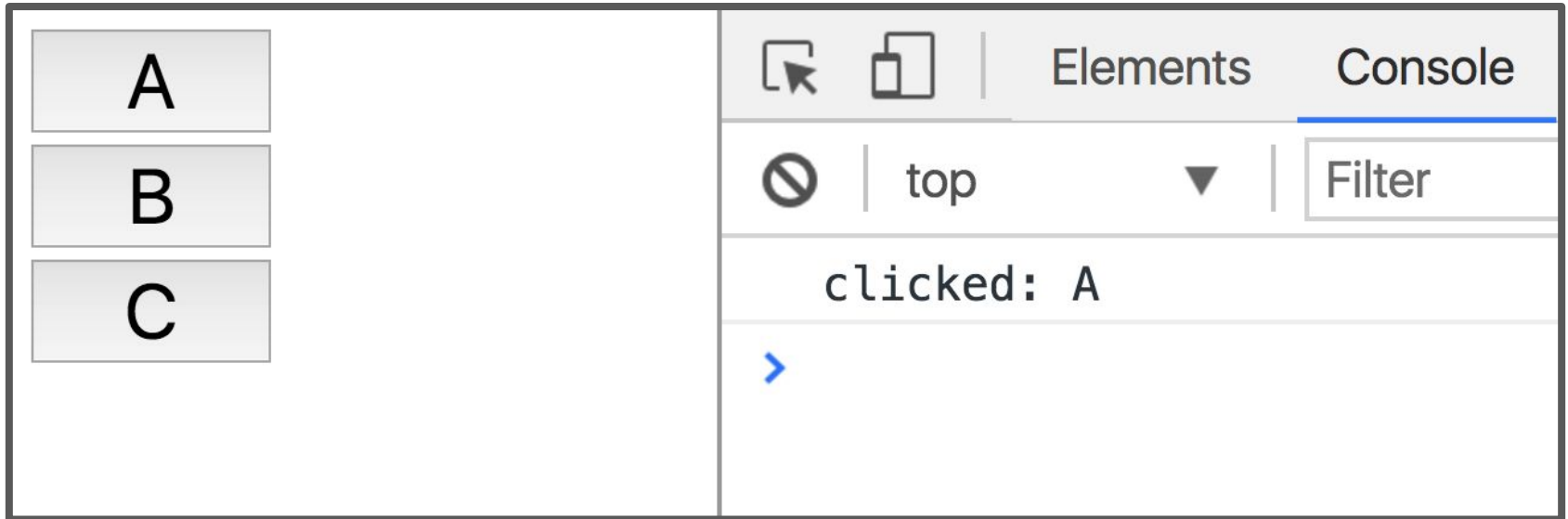
We want to:

- Fill the `<div id="menu"></div>` with buttons A, B, and C
- Update the `<h1>` with the button that was clicked
- [Live example](#)

```
class Button {  
  constructor(containerElement, text) {  
    this.containerElement = containerElement;  
  
    const button = document.createElement('button');  
    button.textContent = text;  
    this.containerElement.append(button);  
  }  
}  
  
const buttonContainer = document.querySelector('#menu');  
const button1 = new Button(buttonContainer, 'A');  
const button2 = new Button(buttonContainer, 'B');  
const button3 = new Button(buttonContainer, 'C');
```

First step: Create a Button class and create three Buttons. ([CodePen](#))

Click handler for Button



Let's make it so that every time we click a button, we print out which button was clicked in the console. ([Live](#))

```
class Button {  
  constructor(containerElement, text) {  
    this.containerElement = containerElement;  
  
    const button = document.createElement('button');  
    button.textContent = text;  
    this.containerElement.append(button);  
  }  
}
```

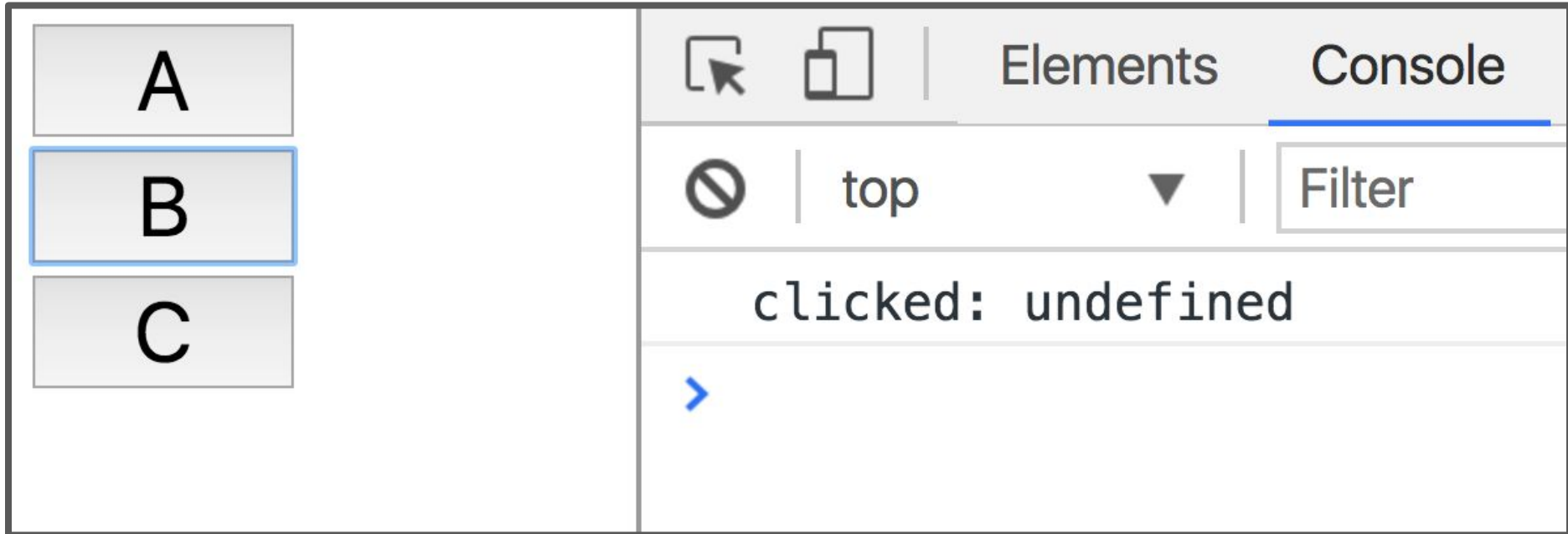
Starting with this definition of Button...

```
class Button {  
  constructor(containerElement, text) {  
    this.containerElement = containerElement;  
    this.text = text;  
  
    const button = document.createElement('button');  
    button.textContent = text;  
    button.addEventListener('click', this.onClick);  
    this.containerElement.append(button);  
  }  
  
  onClick() {  
    console.log('clicked: ' + this.text);  
  }  
}
```

An initial attempt might look like this. ([CodePen](#))

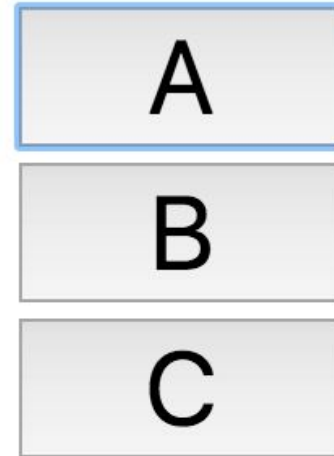
```
class Button {  
  constructor(containerElement, text) {  
    this.containerElement = containerElement;  
    this.text = text;  
  
    const button = document.createElement('button');  
    button.textContent = text;  
    button.addEventListener('click', this.onClick)  
    this.containerElement.append(button);  
  }  
  
  onClick() {  
    console.log('clicked: ' + this.text);  
  }  
}
```

An initial attempt might look like this. ([CodePen](#))



But when we run it, that gives us "clicked: undefined" ([CodePen](#)) **Why?**

```
class Button {  
  constructor(containerElement, text) {  
    this.containerElement = containerElement;  
    this.text = text;  
  
    const button = document.createElement('button');  
    button.textContent = text;  
    button.addEventListener('click', this.onClick);  
    this.containerElement.append(button);  
  }  
  
  onClick() {  
    console.log('clicked: ' + this.text);  
    console.log(this);  
  }  
}
```



```
clicked: undefined  
<button>A</button>  
>
```

That's because the value of `this` in `onClick` is not the `Button` object; it is the `<button>` element to which we've attached the `onClick` event handler.

What?!?

this in JavaScript

this in the constructor

```
class Point {  
    constructor(x, y) {  
        this.x = x;  
        this.y = y;  
    }  
}
```

In the constructor of a class, `this` refers to the new object that is being created.

That's the same meaning as `this` in Java or C++.

this in the constructor

// Java

```
public class Point {  
    public Point(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
  
    public int x;  
    public int y;  
}
```

Here's roughly the equivalent code in Java. `this` refers to the new object that is being created.

this in Java

```
// Java
```

```
public class Point {  
    ...  
  
    String toString() {  
        return this.x + ", " + this.y;  
    }  
}
```

In Java, **this** **always** refers to the new instance being created, no matter what method you're calling it from, or how that method is invoked.

this in JavaScript

```
class Point {  
    ...  
  
    toString() {  
        return this.x + ", " + this.y;  
    }  
}
```

But in JavaScript, **this** can have a different meaning if used outside of the constructor, depending on the **context** in which the function is called.

this in JavaScript

```
toString() {  
    return this.x + ", " + this.y;  
}
```

In JavaScript, `this` is:

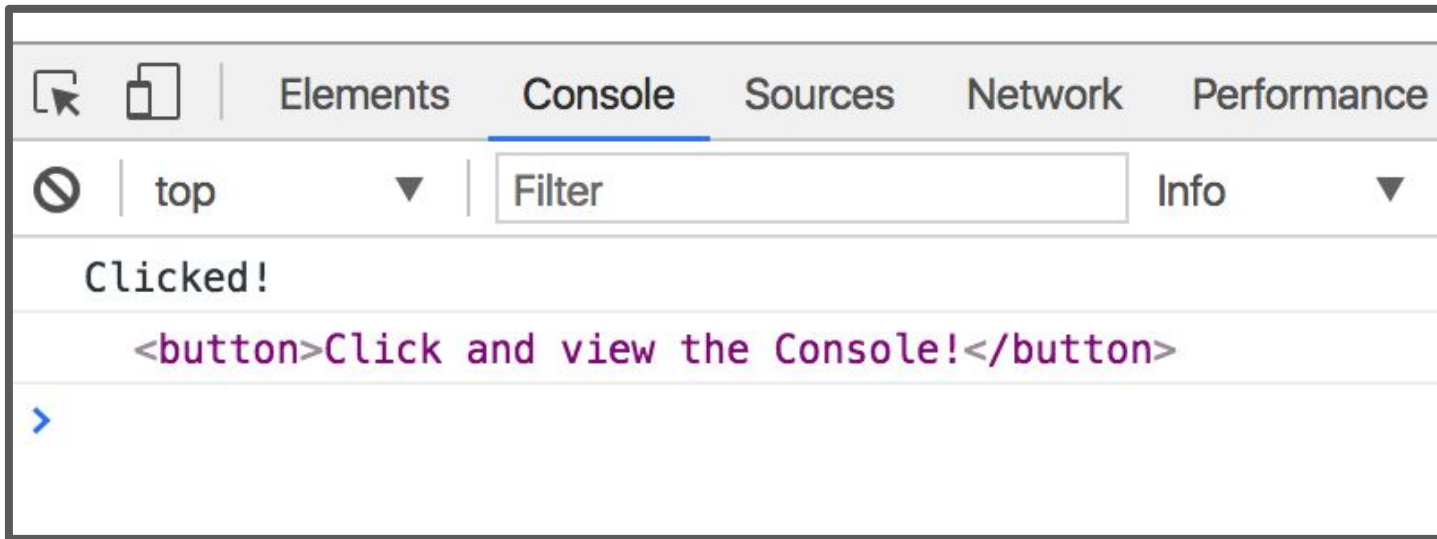
- A implicit **parameter** that is passed to **every JavaScript function**, including functions not defined in a class!
- The value of the `this` parameter changes depending on how it is called.

this in addEventListener

```
function onClick() {  
  console.log('Clicked!');  
  console.log(this);  
}
```

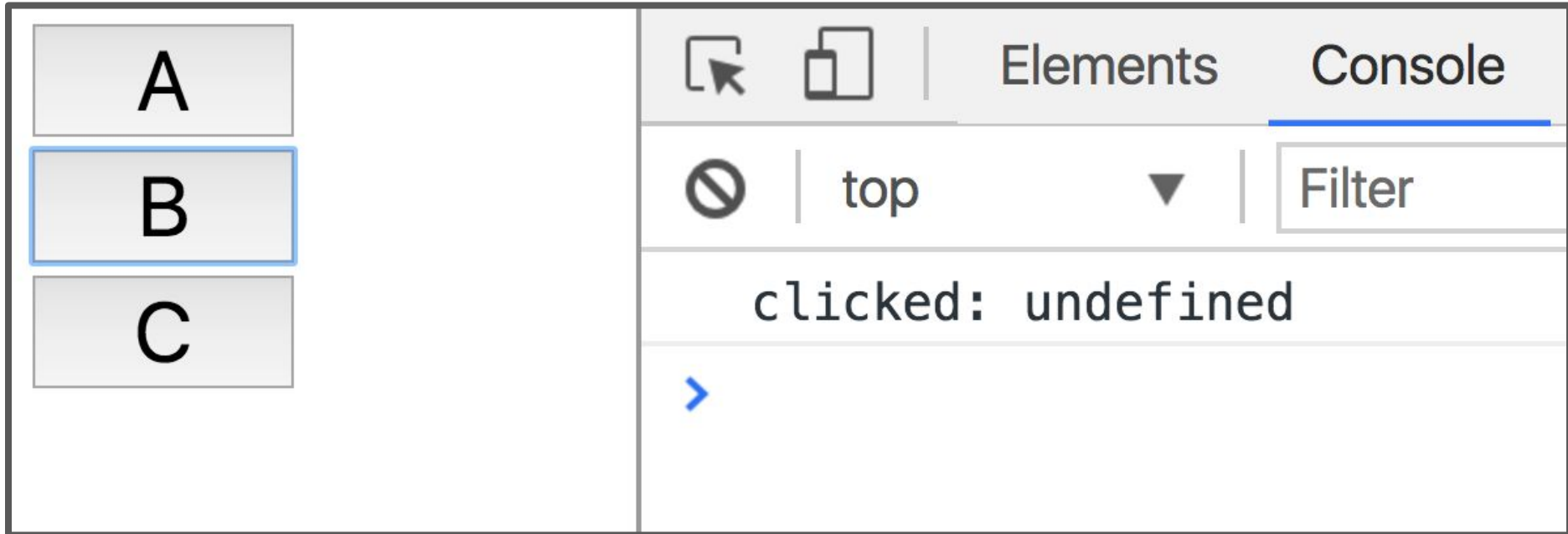
```
const button = document.querySelector('button');  
button.addEventListener('click', onClick);
```

When used in an event handler, **this** is set to the **element to which that the event was added**. ([mdn](#) / [CodePen](#) / [live](#))



```
function onClick() {  
  console.log('Clicked!');  
  console.log(this);  
}  
const button = document.querySelector('button');  
button.addEventListener('click', onClick);
```

In `onClick`, `this` refers to `<button>` because it `onClick` was invoked by `addEventListener`.



Let's revisit our undefined text... ([CodePen](#))

```
class Button {  
  constructor(containerElement, text) {  
    this.containerElement = containerElement;  
    this.text = text;  
  
    const button = document.createElement('button');  
    button.textContent = text;  
    button.addEventListener('click', this.onClick);  
    this.containerElement.append(button);  
  }  
}
```

In the constructor, `this` refers to the new object we're creating. No problems here.

```
onClick() {  
  console.log('clicked: ' + this.text);  
}  
}
```

But in `onClick`, `this` will mean something different depending on how the function is called.

That is because we are using `this` in a function that is **not** a constructor.

```
button.addEventListener('click', this.onClick);  
this.containerElement.append(button);  
}  
  
onClick() {  
  console.log('clicked: ' + this.text);  
}  
}
```

Specifically, because `onClick` is attached to the `<button>` via `addEventListener...`

```
button.addEventListener('click', this.onClick);
this.containerElement.append(button);
}

onClick() {
  ➡ console.log('clicked: ' + this.text);
}
}
```

...we know the value of `this` will be the `<button>` element when the click event is fired and invokes `onClick`.

Since [HTMLButtonElement](#) doesn't have a `text` property, `this.text` is undefined.

```
class Button {  
  constructor(containerElement, text) {  
    this.containerElement = containerElement;  
    this.text = text;  
  }  
}
```

...

```
onClick() {  
  console.log('clicked: ' + this.text);  
}  
}
```

It'd be nice if we could set the value of "this" in onClick to be the Button object, like it is in the constructor.

"Bind" the value of this

```
class Button {  
  constructor(containerElement, text) {  
    this.containerElement = containerElement;  
    this.text = text;  
  
    this.onClick = this.onClick.bind(this);  
  }  
}
```

That is what this line of code does:

"Hey, use the current value of `this` in `onClick`"

(And the current value of `this` is the new object, since we're in the constructor)

[CodePen](#) / [Live](#)

bind in classes

```
constructor() {  
  const someValue = this;  
  this.methodName = this.methodName.bind(someValue);  
}
```

This is saying:

- Make a copy of *methodName*, which will be the exact same as *methodName* except this in *methodName* is always set to the someValue
- The value of someValue is this to bind(), which is the value of the new object since we are in the constructor

bind in classes

```
constructor() {  
  this.methodName = this.methodName.bind(this);  
}
```

And of course, you don't need the intermediate `someValue` variable.

[CodePen](#) / [Live](#)

One more time...

this in the constructor

this in the constructor refers to the new object you are creating.

```
constructor(x, y) {  
  this.x = x;  
  this.y = y;  
}
```

this in a function

this in a function that is **not** a constructor has a different value, depending on **how the function is called**.

```
onClick() {  
  console.log(this.x);  
  console.log(this.u);  
}
```

- When invoked as a response to an event, the `this` in `onClick` will be `Event.targetElement`, or the element onto which the `onClick` event handler was attached.

A consistent this

```
class Point {  
  constructor(x, y) {  
    this.x = x;  
    this.y = y;  
  }  
  
  onClick() {  
    console.log(this.x);  
    console.log(this.u);  
  }  
}
```

Right now, **this** in the constructor always refers to the new object we're creating...

A consistent this

```
class Point {  
  constructor(x, y) {  
    this.x = x;  
    this.y = y;  
  }  
  
  onClick() {  
    console.log(this.x);  
    console.log(this.u);  
  }  
}
```

But `this` in `onClick` function refers to a different value, depending on how `onClick` is called.

A consistent this

```
class Point {  
  constructor(x, y) {  
    this.x = x;  
    this.y = y;  
  }  
  
  onClick() {  
    console.log(this.x);  
    console.log(this.y);  
  }  
}
```

It'd be nice if we could make the "this" value in onClick:

- Refer to the new object we're constructing, instead of things like the dom element, etc
- And make it always refer to the new object we're constructing

A consistent `this`

```
class Point {  
  constructor(x, y) {  
    this.x = x;  
    this.y = y;  
    this.onClick = this.onClick.bind(someParam);  
  }  
}
```

That's what **bind** does:

- It is saying, "Hey, in the `onClick` function, I want the `this` value to always be ***someParam***," i.e. the value that we are passing as a parameter to `bind`.

A consistent this

```
class Point {  
  constructor(x, y) {  
    this.x = x;  
    this.y = y;  
    this.onClick = this.onClick.bind(someParam);  
  }  
}
```

```
  onClick() {  
    console.log(this.x);  
    console.log(this.u);  
  }  
}
```

We want the value of `this` in `onClick` to be the value of the new object being created.

In other words, we want ***someParam*** to be the value of the new object being created.

A consistent this

```
class Point {  
  constructor(x, y) {  
    this.x = x;  
    this.y = y;  
    this.onClick = this.onClick.bind(someParam);  
  }  
  
  onClick() {  
    console.log(this.x);  
    console.log(this.u);  
  }  
}
```

In the constructor, how do we access the new object being created?

A consistent this

```
class Point {  
  constructor(x, y) {  
    this.x = x;  
    this.y = y;  
    this.onClick = this.onClick.bind(this);  
  }  
  
  onClick() {  
    console.log(this.x);  
    console.log(this.y);  
  }  
}
```

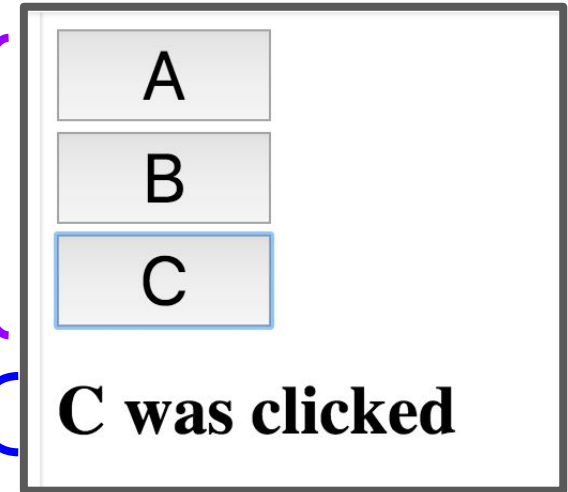
In the constructor, the new object is referenced by **this**.

Now the **this** in `onClick` always referring to the new object.

What were we
trying to do again?

Example: Buttons

```
<html>
  <head>
    <meta charset="utf-8">
    <title>Menu and buttons examples</title>
  </head>
  <body>
    <div id="menu"></div>
    <h1 id="status-bar"></h1>
  </body>
</html>
```



We want to:

- Fill the `<div id="menu"></div>` with buttons A, B, and C
- Update the `<h1>` with the button that was clicked
- [Live example](#)

(Contrived) OO example

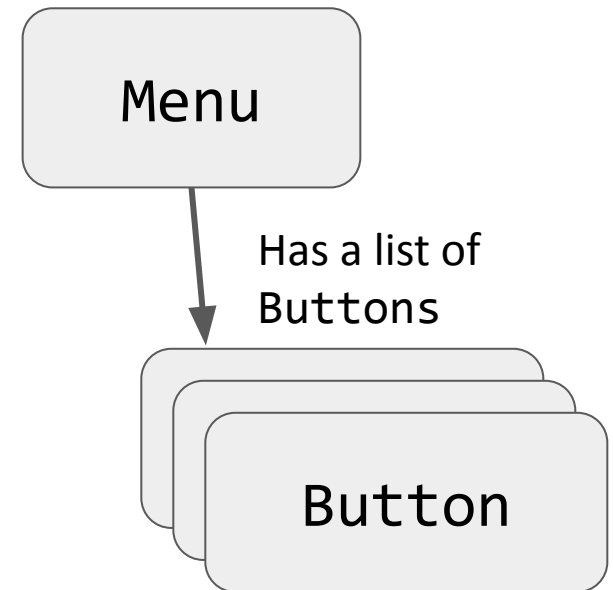
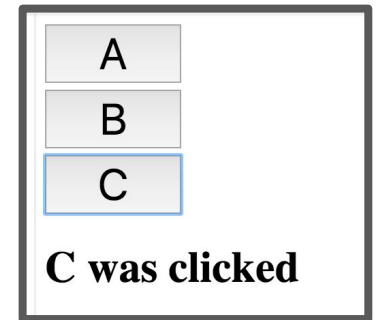
For practice, we'll write this using 2 classes:

Menu:

- Has an array of Buttons
- Also updates the <h1> with what was clicked

Button:

- Notifies Menu when clicked, so that Menu can update the <h1>



```
class Menu {  
  constructor() {  
    this.buttonContainer = document.querySelector('#menu');  
    this.statusBar = document.querySelector('#status-bar');  
  
    this.buttons = [  
      new Button(this.buttonContainer, 'A'),  
      new Button(this.buttonContainer, 'B'),  
      new Button(this.buttonContainer, 'C')  
    ];  
  }  
}
```

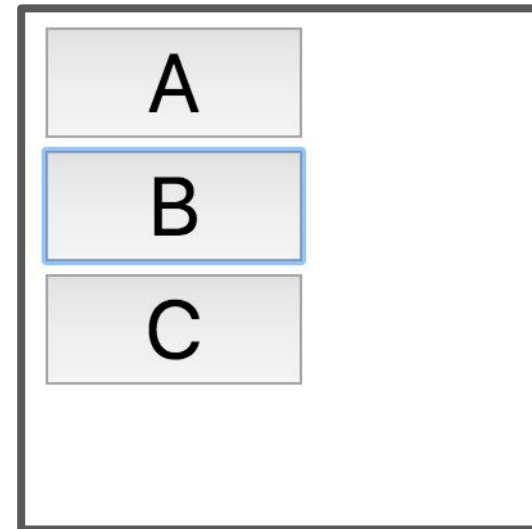
Partial solution: We create a Menu class, which creates the Buttons ([CodePen](#))

```
const menu = new Menu();
```

Then we create the Menu (and the menu creates the Buttons) when the page loads. ([CodePen](#))

Update Menu when Button clicked

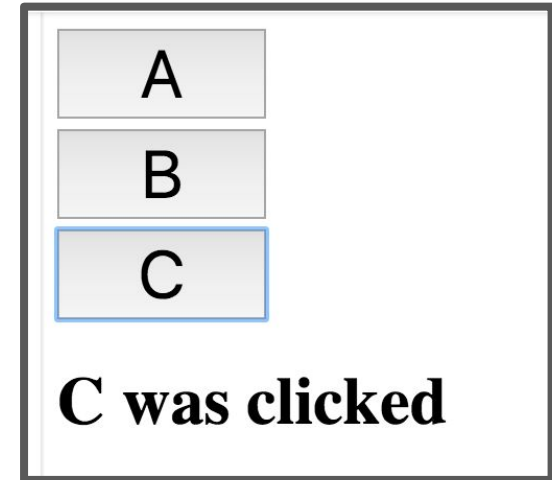
```
class Menu {  
  constructor() {  
    this.buttonContainer = document.querySelector('#menu');  
    this.statusBar = document.querySelector('#status-bar');  
  
    this.buttons = [  
      new Button(this.buttonContainer, 'A'),  
      new Button(this.buttonContainer, 'B'),  
      new Button(this.buttonContainer, 'C')  
    ];  
  }  
}
```



Our current Menu doesn't do much.

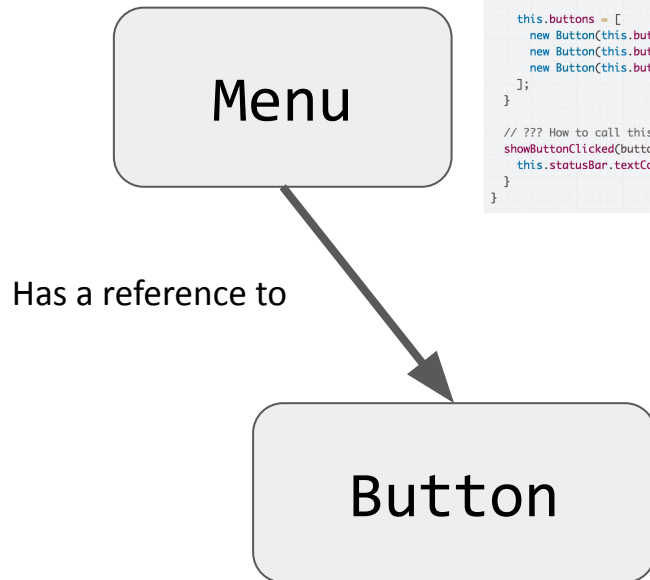
Update Menu when Button clicked

```
class Menu {  
  constructor() {  
    this.buttonContainer = document.querySelector('#menu');  
    this.statusBar = document.querySelector('#status-bar');  
  
    this.buttons = [  
      new Button(this.buttonContainer, 'A'),  
      new Button(this.buttonContainer, 'B'),  
      new Button(this.buttonContainer, 'C')  
    ];  
  }  
  
  // ??? How to call this?  
  showButtonClicked(buttonName) {  
    this.statusBar.textContent = buttonName + ' was clicked';  
  }  
}
```



We want the Menu to update the `<h1>` when one of the Buttons are clicked. **How do we do this?**

Communicating upstream



```
class Menu {
  constructor() {
    this.buttonContainer = document.querySelector('#menu');
    this.statusBar = document.querySelector('#status-bar');

    this.buttons = [
      new Button(this.buttonContainer, 'A'),
      new Button(this.buttonContainer, 'B'),
      new Button(this.buttonContainer, 'C')
    ];
  }

  // ??? How to call this?
  showButtonClicked(buttonName) {
    this.statusBar.textContent = buttonName + ' was clicked';
  }
}
```

Button is the thing that knows it was clicked...

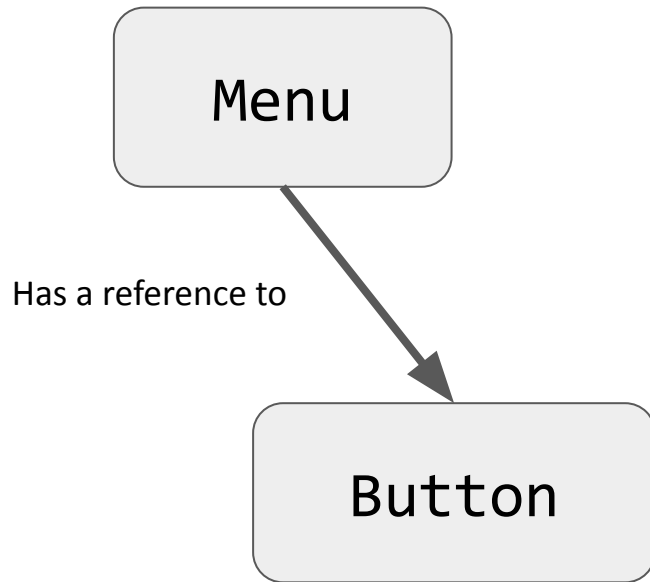
```
class Button {
  constructor(containerElement, text) {
    this.containerElement = containerElement;
    this.text = text;

    this.onClick = this.onClick.bind(this);

    const button = document.createElement('button');
    button.textContent = text;
    button.addEventListener('click', this.onClick);
    this.containerElement.append(button);
  }

  onClick() {
    console.log('clicked: ' + this.text);
  }
}
```

Communicating upstream



But Menu is the thing that
can update the header.

```
class Menu {
  constructor() {
    this.buttonContainer = document.querySelector('#menu');
    this.statusBar = document.querySelector('#status-bar');

    this.buttons = [
      new Button(this.buttonContainer, 'A'),
      new Button(this.buttonContainer, 'B'),
      new Button(this.buttonContainer, 'C')
    ];
  }

  // ??? How to call this?
  showButtonClicked(buttonName) {
    this.statusBar.textContent = buttonName + ' was clicked';
  }
}
```

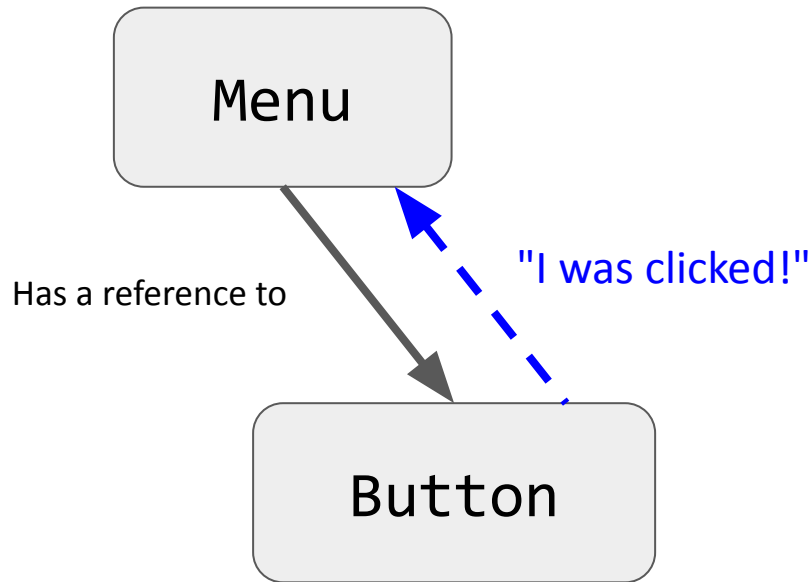
```
class Button {
  constructor(containerElement, text) {
    this.containerElement = containerElement;
    this.text = text;

    this.onClick = this.onClick.bind(this);

    const button = document.createElement('button');
    button.textContent = text;
    button.addEventListener('click', this.onClick);
    this.containerElement.append(button);
  }

  onClick() {
    console.log('clicked: ' + this.text);
  }
}
```

Communicating upstream



It needs to be possible for a Button to tell the Menu that it has been clicked.

```
class Menu {
  constructor() {
    this.buttonContainer = document.querySelector('#menu');
    this.statusBar = document.querySelector('#status-bar');

    this.buttons = [
      new Button(this.buttonContainer, 'A'),
      new Button(this.buttonContainer, 'B'),
      new Button(this.buttonContainer, 'C')
    ];
  }

  // ??? How to call this?
  showButtonClicked(buttonName) {
    this.statusBar.textContent = buttonName + ' was clicked';
  }
}
```

```
class Button {
  constructor(containerElement, text) {
    this.containerElement = containerElement;
    this.text = text;

    this.onClick = this.onClick.bind(this);

    const button = document.createElement('button');
    button.textContent = text;
    button.addEventListener('click', this.onClick);
    this.containerElement.append(button);
  }

  onClick() {
    console.log('clicked: ' + this.text);
  }
}
```

One strategy for doing this:
Custom events

Custom Events

You can listen to and dispatch Custom Events to communicate between classes ([mdn](#)):

```
const event = new CustomEvent(  
    eventNameString, optionalParameterObject);  
  
element.addEventListener(eventNameString,  
functionName);  
  
element.dispatchEvent(eventNameString);
```

Custom Events on document

CustomEvent **can only be listened to / dispatched on HTML elements**, and not on arbitrary class instances.

Therefore we are going to be adding/dispatching events on the `document` object, so that events can be globally listened to/dispatched.

```
document.addEventListener(eventNameString,  
functionName);
```

```
document.dispatchEvent(eventNameString);
```

Define a custom event

We'll define a custom event called 'button-click':

Menu will listen for the event:

```
document.addEventListener(  
    'button-click', this.showButtonClicked);
```

Button will dispatch the event:

```
document.dispatchEvent(  
    new CustomEvent('button-click'));
```

```
class Menu {  
  constructor() {  
    this.buttonContainer = document.querySelector('#menu');  
    this.statusBar = document.querySelector('#status-bar');  
  
    this.buttons = [  
      new Button(this.buttonContainer, 'A'),  
      new Button(this.buttonContainer, 'B'),  
      new Button(this.buttonContainer, 'C')  
    ];  
  }  
}
```

A first attempt: We should listen for the custom
'button-click' event in Menu.

```
class Menu {  
  constructor() {  
    this.buttonContainer = document.querySelector('#menu');  
    this.statusBar = document.querySelector('#status-bar');  
  
    this.showButtonClicked = this.showButtonClicked.bind(this);  
  
    this.buttons = [  
      new Button(this.buttonContainer, 'A'),  
      new Button(this.buttonContainer, 'B'),  
      new Button(this.buttonContainer, 'C')  
    ];  
  
    document.addEventListener('button-click', this.showButtonClicked);  
  }  
  
  showButtonClicked(event) {  
    console.log("Menu notified!");  
    const buttonName = event.currentTarget.textContent;  
    this.statusBar.textContent = buttonName + ' was clicked';  
  }  
}
```

A first attempt: Listen for the custom 'button-click' event in Menu. **Note the call to bind!** ([CodePen](#))

```
class Menu {  
  constructor() {  
    this.buttonContainer = document.querySelector('#menu');  
    this.statusBar = document.querySelector('#status-bar');  
  
    this.showButtonClicked = this.showButtonClicked.bind(this);  
  
    this.buttons = [  
      new Button(this.buttonContainer, 'A'),  
      new Button(this.buttonContainer, 'B'),  
      new Button(this.buttonContainer, 'C')  
    ];  
  
    document.addEventListener('button-click', this.showButtonClicked);  
  }  
  
  showButtonClicked(event) {  
    console.log("Menu notified!");  
    const buttonName = event.currentTarget.textContent;  
    this.statusBar.textContent = buttonName + ' was clicked';  
  }  
}
```

A first attempt: Listen for the custom 'button-click' event in Menu. **Note the call to bind!** ([CodePen](#))

```
class Button {  
  constructor(containerElement, text) {  
    this.containerElement = containerElement;  
    this.text = text;  
  
    this.onClick = this.onClick.bind(this);  
  
    const button = document.createElement('button');  
    button.textContent = text;  
    button.addEventListener('click', this.onClick);  
    this.containerElement.append(button);  
  }  
  
  onClick() {  
    console.log('clicked: ' + this.text);  
  }  
}
```

Then we want to dispatch the 'button-click' event in the onClick event handler in Button.

```
class Button {
  constructor(containerElement, text) {
    this.containerElement = containerElement;
    this.text = text;

    this.onClick = this.onClick.bind(this);

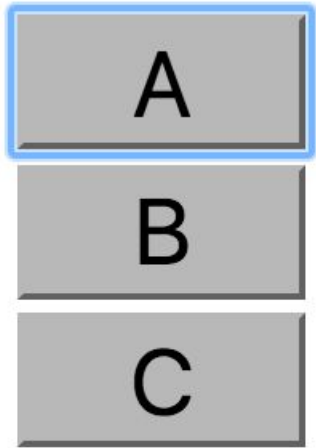
    const button = document.createElement('button');
    button.textContent = text;
    button.addEventListener('click', this.onClick);
    this.containerElement.append(button);
  }

  onClick() {
    console.log('clicked: ' + this.text);
    document.dispatchEvent(new CustomEvent('button-click'));
  }
}
```

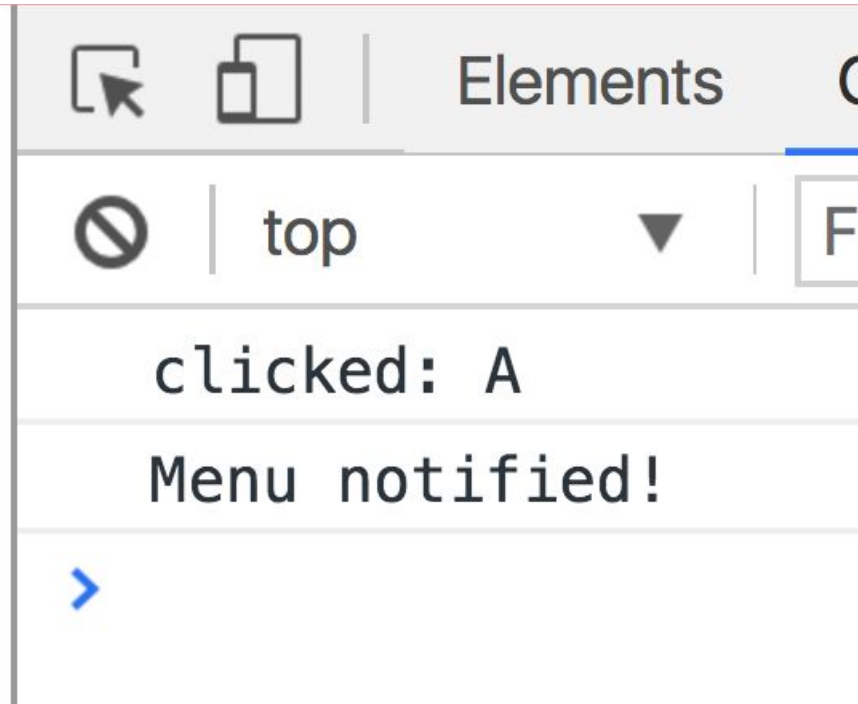
Dispatch the 'button-click' event in the onClick event handler in Button ([CodePen](#)).

```
class Button {  
  constructor(containerElement, text) {  
    this.containerElement = containerElement;  
    this.text = text;  
  
    this.onClick = this.onClick.bind(this);  
  
    const button = document.createElement('button');  
    button.textContent = text;  
    button.addEventListener('click', this.onClick);  
    this.containerElement.append(button);  
  }  
  
  onClick() {  
    console.log('clicked: ' + this.text);  
    document.dispatchEvent(new CustomEvent('button-click'));  
  }  
}
```

Dispatch the 'button-click' event in the onClick event handler in Button ([CodePen](#)).



null was clicked



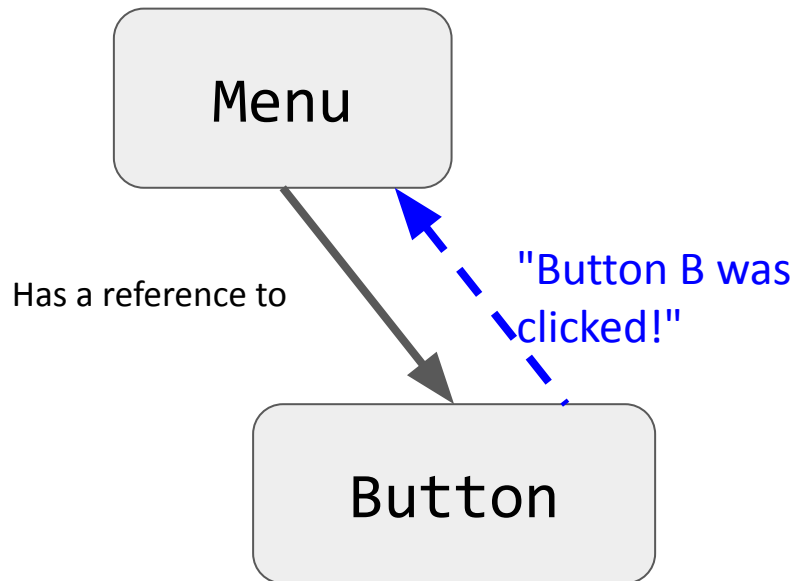
When we try it out, the event dispatching seems to work... but our output is "null was clicked"

([CodePen](#) / [Live](#))

```
class Menu {  
  constructor() {  
    this.buttonContainer = document.querySelector('#menu');  
    this.statusBar = document.querySelector('#status-bar');  
  
    this.showButtonClicked = this.showButtonClicked.bind(this);  
  
    this.buttons = [  
      new Button(this.buttonContainer, 'A'),  
      new Button(this.buttonContainer, 'B'),  
      new Button(this.buttonContainer, 'C')  
    ];  
  
    document.addEventListener('button-click', this.showButtonClicked);  
  }  
  
  showButtonClicked(event) {  
    console.log("Menu notified!");  
    const buttonName = event.currentTarget.textContent;  
    this.statusBar.textContent = buttonName + ' was clicked';  
  }  
}
```

The problem is we are adding custom event listeners to document, meaning `event.currentTarget` is going to be document, and not `<button>`

Communicating upstream



Menu knows some button was clicked... How do we tell the Menu which button was clicked?

```
class Menu {
  constructor() {
    this.buttonContainer = document.querySelector('#menu');
    this.statusBar = document.querySelector('#status-bar');

    this.showButtonClicked = this.showButtonClicked.bind(this);

    this.buttons = [
      new Button(this.buttonContainer, 'A'),
      new Button(this.buttonContainer, 'B'),
      new Button(this.buttonContainer, 'C')
    ];

    document.addEventListener('button-click', this.showButtonClicked);
  }

  showButtonClicked(event) {
    console.log("Menu notified!");
    const buttonName = event.currentTarget.textContent;
    this.statusBar.textContent = buttonName + ' was clicked';
  }
}
```

```
class Button {
  constructor(containerElement, text) {
    this.containerElement = containerElement;
    this.text = text;

    this.onClick = this.onClick.bind(this);

    const button = document.createElement('button');
    button.textContent = text;
    button.addEventListener('click', this.onClick);
    this.containerElement.append(button);
  }

  onClick() {
    console.log('clicked: ' + this.text);
    document.dispatchEvent(new CustomEvent('button-click'));
  }
}
```

CustomEvent parameters

You can add a parameter to your [CustomEvent](#):

- Create an object with a `detail` property
- The value of this `detail` property can be whatever you'd like.

```
onClick() {  
  const eventInfo = {  
    buttonName: this.text  
  };  
  document.dispatchEvent(  
    new CustomEvent('button-clicked', { detail: eventInfo }));  
}
```

CustomEvent parameters

You can add a parameter to your [CustomEvent](#):

- The event handler for your CustomEvent will be able to access this detail property via `Event.detail`

```
document.addEventListener('button-clicked', this.showButtonClicked);  
}  
  
showButtonClicked(event) {  
  this.statusBar.textContent = event.detail.buttonName + ' was clicked';  
}  
}
```

[Finished CodePen](#)