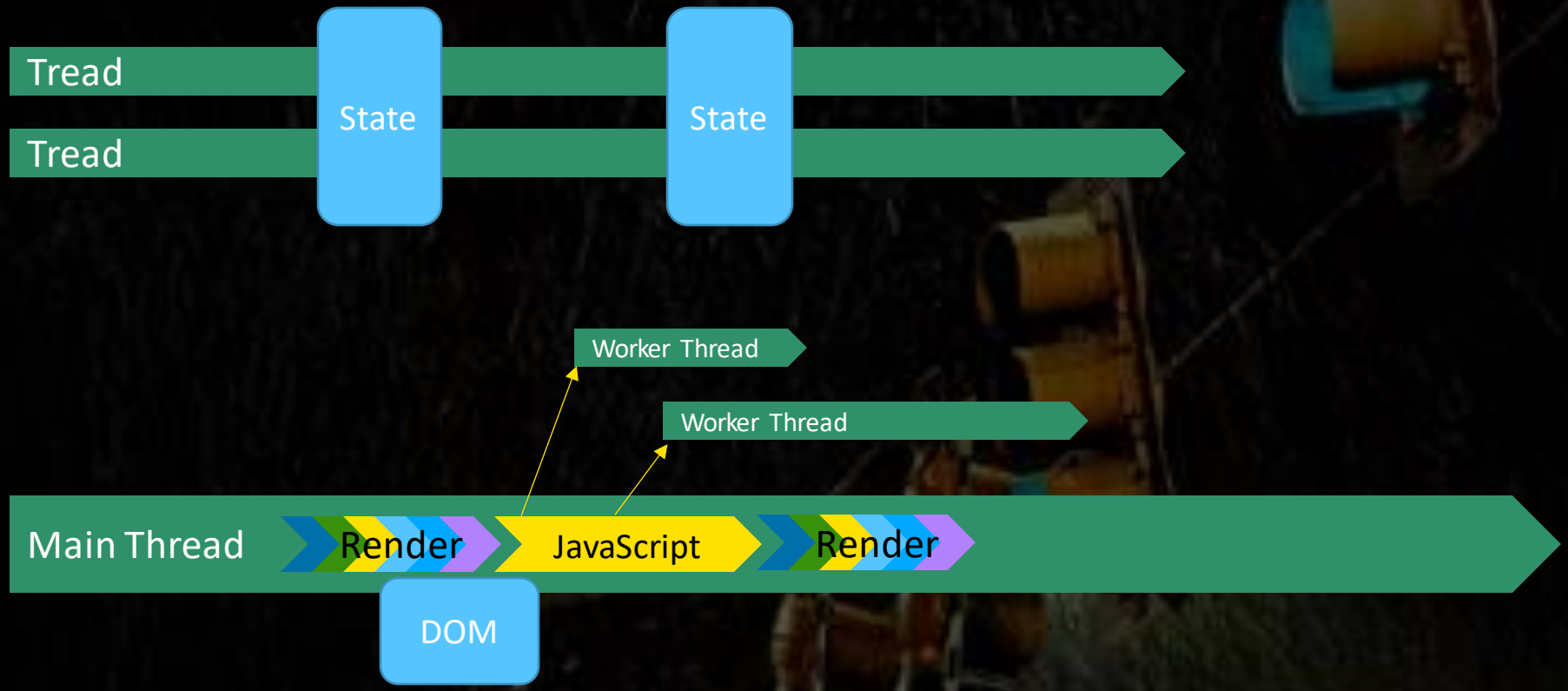
Concurrency Model and the Event Loop

Event driven programming

Concurrency Model

Specifies how threads in the system colaborate to complete the tasks given.



Call Stack

LIFO

stack

foo()

```
a: 10
b: 21
```

bar()

```
y: 3
x: 7
```

(anonymous)

```
foo: function
bar: function
res: 42
```

stack frames

```
function foo (b) {
 const a = 10
 // throw new Error()
 return a + b + 11
}

function bar (x) {
 const y = 3
 return foo(x * y)
}

let res = console.log(bar(7))
```

```
Uncaught Error                stack.js:9
    at foo (stack.js:9)
    at bar (stack.js:21)
    at stack.js:24
```

heap

```
function foo (b) {
  …
}
```

```
function bar (x) {
  …
}
```

Garbage Collector

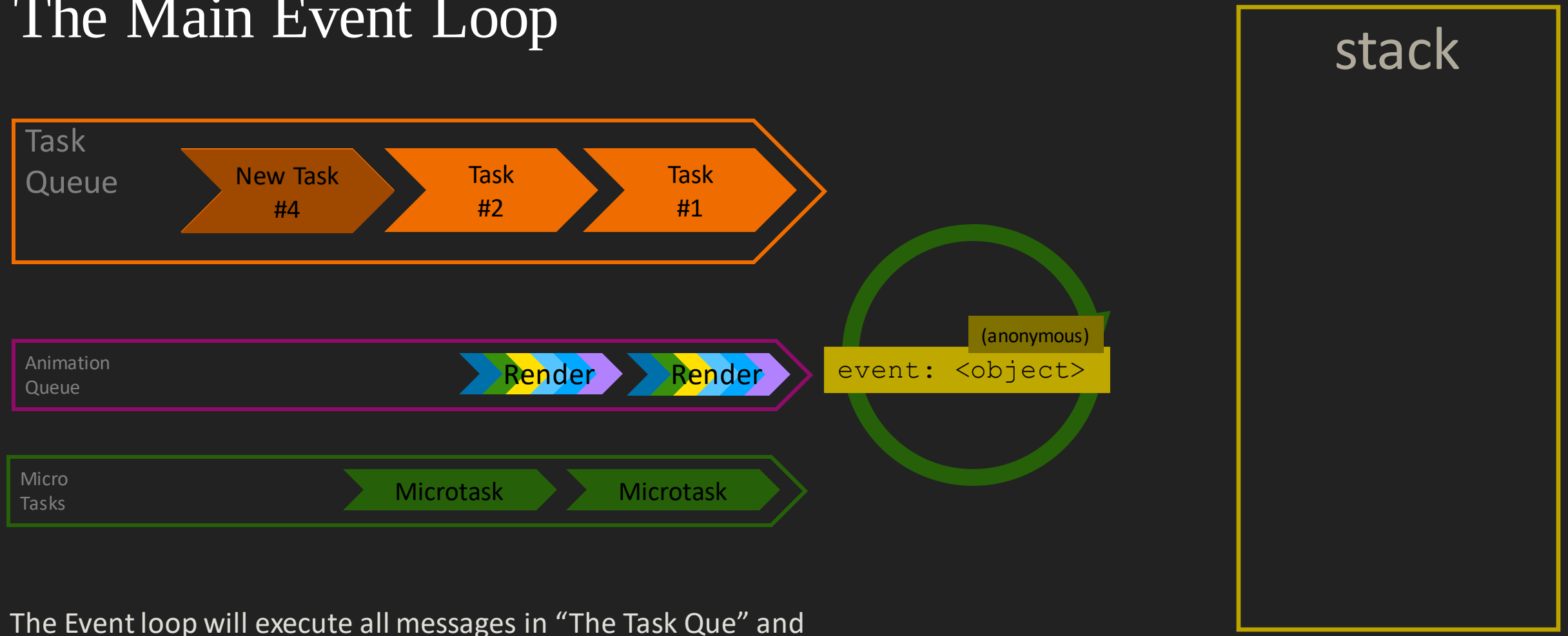
The Task Queue

FIFO

Tasks are added to the queue when:

- New Javascript code is executed
- An event is triggered, adding a message to the queue
- A TimeOut or Interval triggers
- A Worker is adding a message to the queue

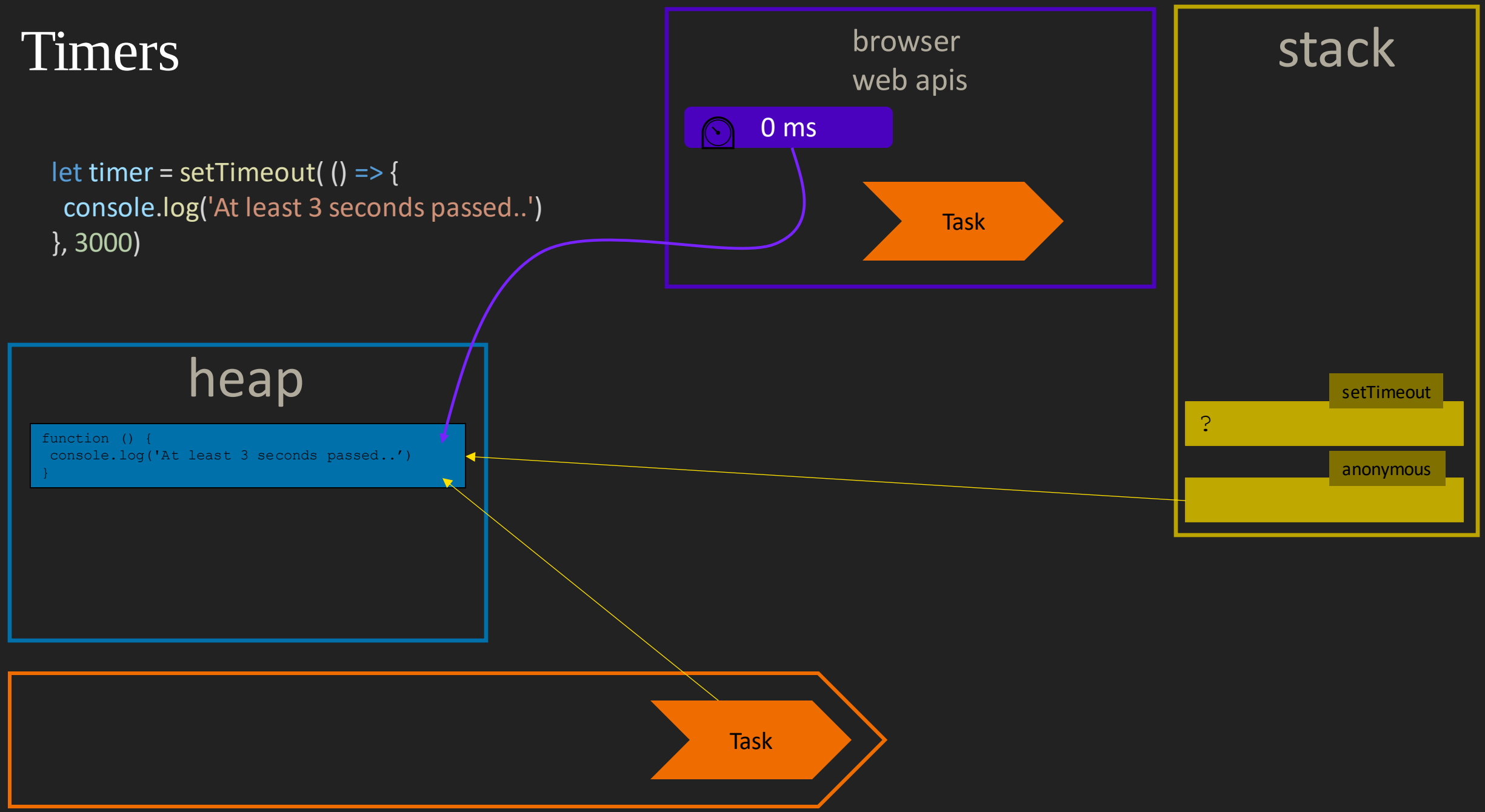
The Main Event Loop



The Event loop will execute all messages in “The Task Que” and “Animation Queue” before continuing.

Timers

```
let timer = setTimeout( () => {
  console.log('At least 3 seconds passed..')
}, 3000)
```



Timers and Interval

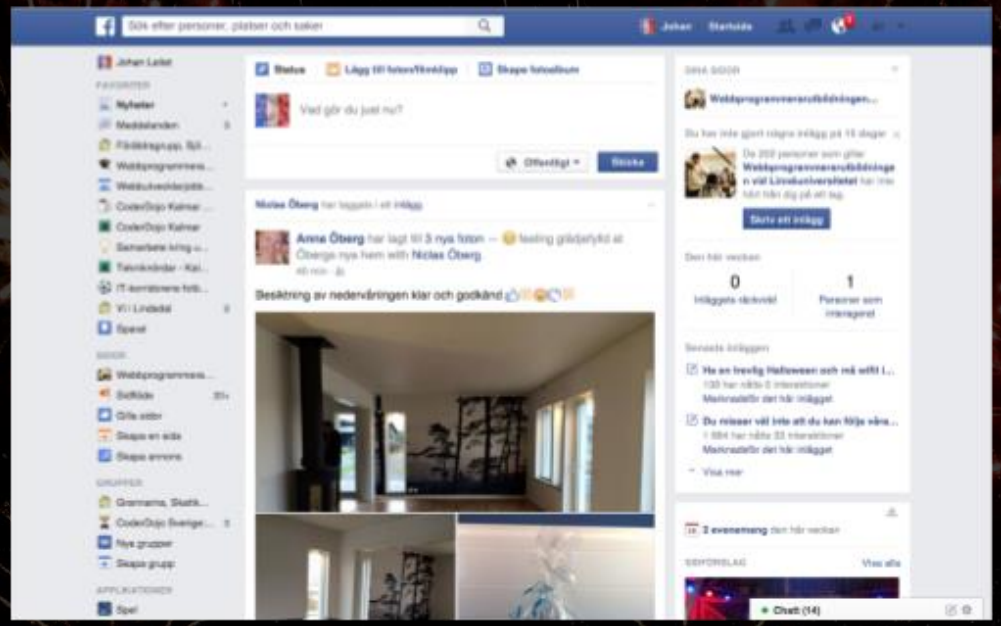
```
const timer = window.setTimeout( () => {
  console.log('At least 3 seconds passed..')
}, 3000)
```

```
window.clearTimeout(timer) // Stops the timer
```

```
const interval = window.setInterval( () => {
  console.log('At least 3 seconds passed..')
}, 3000)
```

```
window.clearInterval(interval) // Stops the interval
```

Event



```
637  function createBackgrounds() {
638
639      var printMode = isPrintingPDF();
640
641      // Clear prior backgrounds
642      dom.background.innerHTML = '';
643      dom.background.classList.add( 'no-transition' );
644
645      // Iterate over all horizontal slides
646      toArray( dom.wrapper.querySelectorAll( HORIZONTAL_SLIDES_SELECTOR ) ).forEach( fu
647
648          var backgroundStack;
649
650          if( printMode ) {
651              backgroundStack = createBackground( slideh, slideh );
652          }
653          else {
654              backgroundStack = createBackground( slideh, dom.background );
655          }
656
657          // Iterate over all vertical slides
658          toArray( slideh.querySelectorAll( 'section' ) ).forEach( function( slidev ) {
659
660              if( printMode ) {
661                  createBackground( slidev, slidev );
662              }
663              else {
664                  createBackground( slidev, backgroundStack );
665              }
666
667              backgroundStack.classList.add( 'stack' );
668
669          } );
670
671      } );
672
673      // Add parallax background if specified
674      if( config.parallaxBackgroundImage ) {
675
676          dom.background.style.backgroundImage = 'url("' + config.parallaxBackgroundImag
677          dom.background.style.backgroundSize = config.parallaxBackgroundSize;
678
679          // Make sure the below properties are set on the element - these properties ar
680          // needed for proper transitions to be set on the element via CSS. To remove
681          // annoying background slide-in effect when the presentation starts, apply
682          // these properties after short time delay
683          setTimeout( function() {
684              dom.wrapper.classList.add( 'has-parallax-background' );
685          }, 1 );
686
687      }
688      else {
```

Event

User initialized
- Mouse scrolling, keyboard ...

Browser initialized
- Page loaded, DOM Changed ...

Network
- Content loaded ...

Synthetic events
- Your custom events

Anti-pattern

```
<a href="page.html" onclick="doSomething()">The link</a>
```

Add event listener

```
const a = document.querySelector('#theatag')

a.addEventListener('click', buttonClicked)

function buttonClicked(event){
  console.log('You clicked the link!')
}
```

```
const a = document.querySelector('#theatag')

a.addEventListener('click', function(event){
 console.log('You clicked the link!')
})
```

```
const a = document.querySelector('#theatag')

a.addEventListener('click', (event) => {
 console.log('You clicked the link!')
})
```

Remove event listener

```
const a = document.querySelector('#theatag')

a.addEventListener('click', function buttonClicked(event){
  console.log('You can only click me once!’)

  a.removeEventListener('click', buttonClicked)
})
```

What triggered the event?

Inside an event handler *this* will reference the element which triggered the event.

```
const a = document.querySelector('#atag')

a.addEventListener('click', function(event){
  a === this // true
  a === event.target // true
  this === event.target // true
})
```

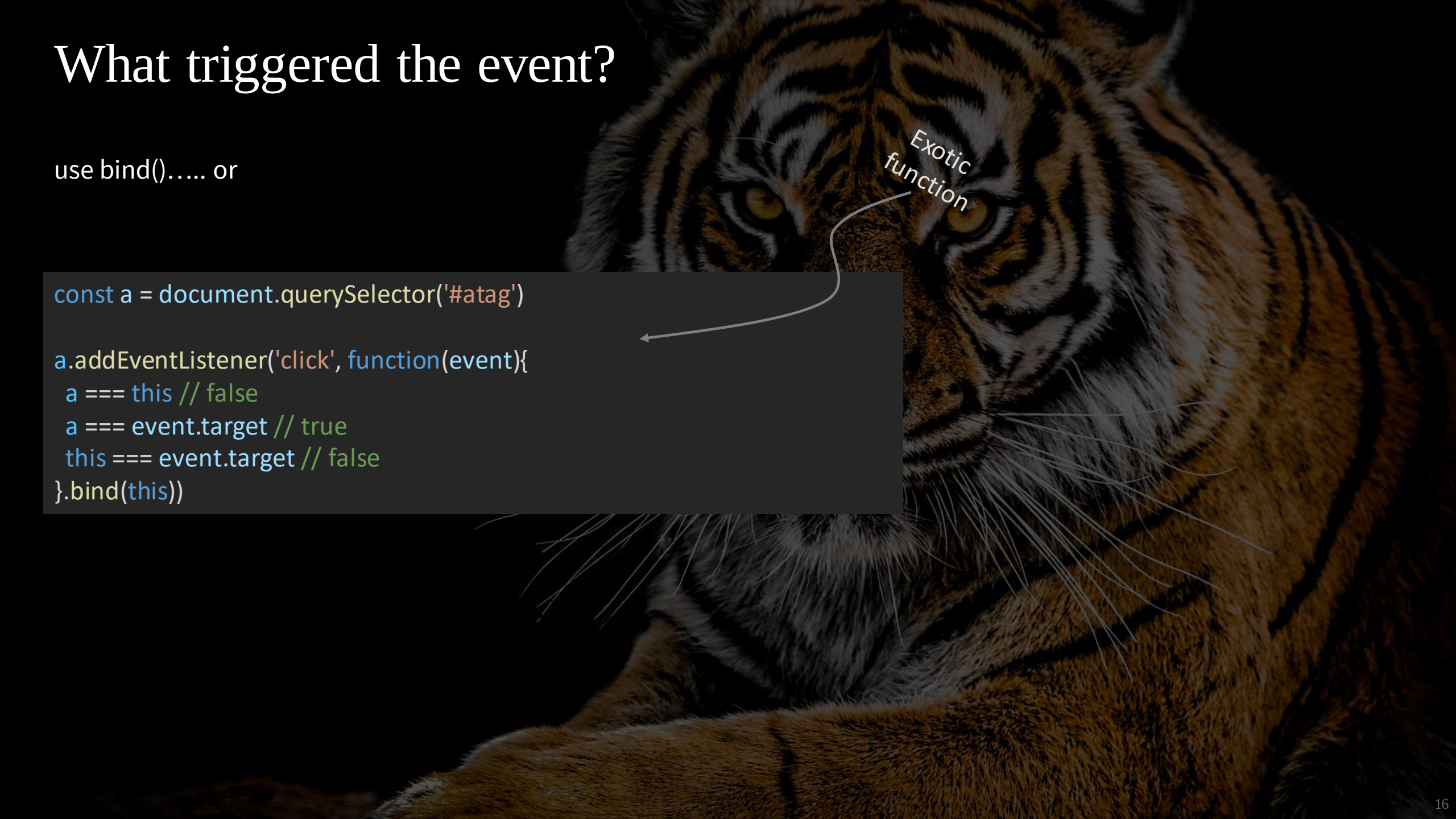
What triggered the event?

The old fashion way. (do not do this… anymore)

```
const a = document.querySelector('#atag’)
const that = this

a.addEventListener('click', function(event){
  // Use that to refer to this outside of the function.

})
```

What triggered the event?

use bind()….. or




```
const a = document.querySelector('#atag')

a.addEventListener('click', function(event){
  a === this // false
  a === event.target // true
  this === event.target // false
}.bind(this))
```

What triggered the event?

… use Arrow Functions (if possible)

```
const a = document.querySelector('#atag')

a.addEventListener('click', (event) => {
  a === this // false
  a === event.target // true
  this === event.target // false
})
```

Closures can cause headache

```
<div id="links">
  <a href="#">The first link</a>
  <a href="#">The second link</a>
  <a href="#">The third link</a>
  <a href="#">The forth link</a>
</div>
```

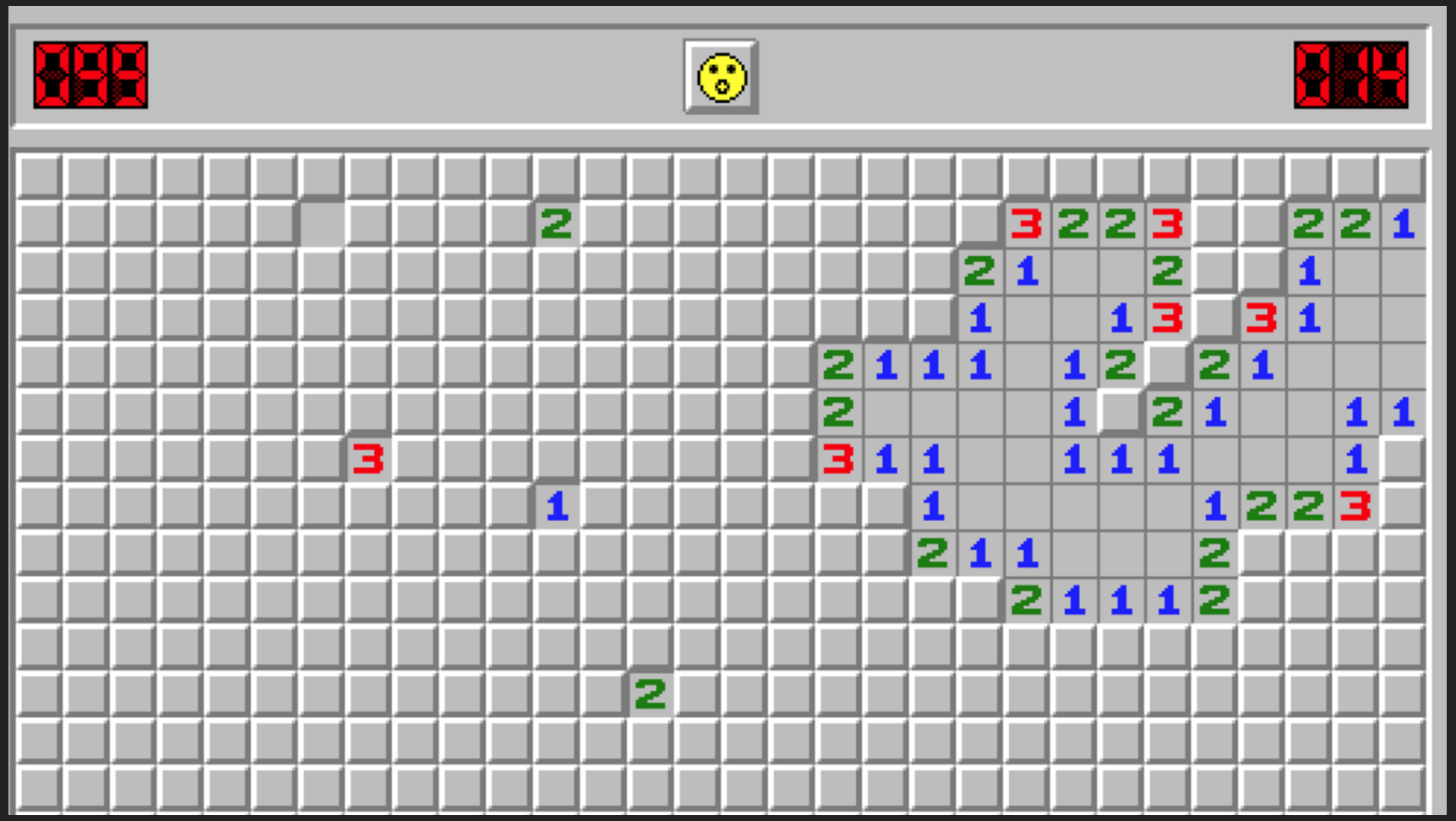
```
const aTags = document.querySelectorAll('#links a')

for(let i = 0 ; i < aTags.length; i += 1) {

  aTags[i].addEventListener('click', (event) => {
    console.log('Clicked link ', i) // 4
    console.log(event.target.textContent) // Correct ('The first link'...)

  })
}
```

Function FTW, or?

```
<div id="links">
  <a href="#">The first link</a>
  <a href="#">The second link</a>
  <a href="#">The third link</a>
  <a href="#">The forth link</a>
</div>
```

```
const aTags = document.querySelectorAll('#links a’)

  aTags.forEach( (a, i) => {
    a.addEventListener('click', (event) => {
    console.log(i) // Correct (0 ... 1 ... 2)
    console.log(event.target.textContent) // Correct
  })
})
```

Event delegation

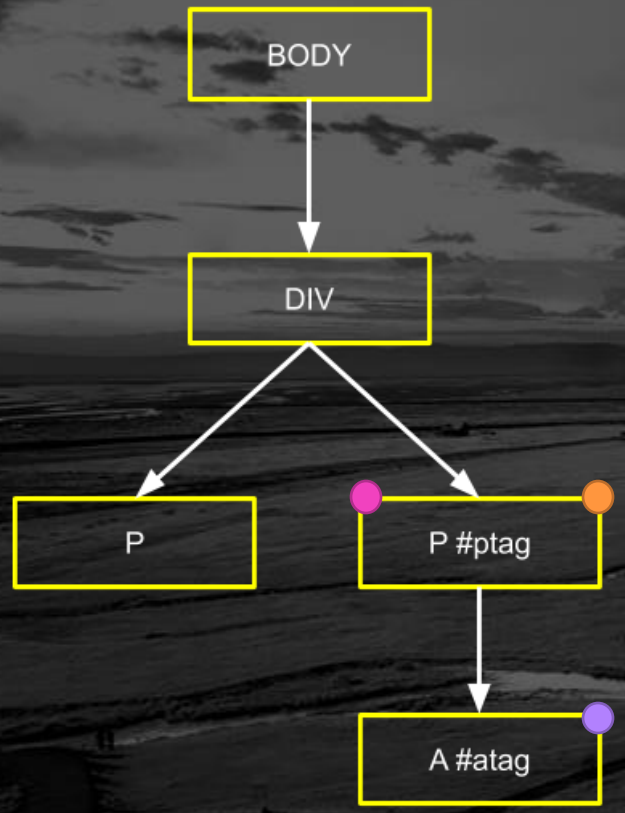


Event delegation

```html
<div id="gameboard">
  <button id='m1'></button>
  <button id='m2'></button>
  ...
  <button id='m500'></button>
</div>
```

```js
const board = document.querySelector('#gameboard')

board.addEventListener('click', (event) => {

  console.log(event.target.textContent) // <button>#m32

  event.currentTarget === board //true

})
```

Propagation




```
const a = document.querySelector('#atag')
const p = document.querySelector('#ptag')

a.addEventListener('click', (event) => {
  console.log('You clicked the link!')

  event.stopPropagation()
})

p.addEventListener('click', (event) => {
  console.log('Damn you a! I will not be called.')
})
```

```
p.addEventListener('click', (event) => {
  console.log('Hihi! Captured it!')
}, {capture: true})
```

Stop the default action


```
const a = document.querySelector('#atag')

a.addEventListener('click', (event) => {
  console.log('I got this. Don´t activate the link!')
  event.preventDefault()

})
```

Another alternative is to *return false* from the listener.

Web Workers API

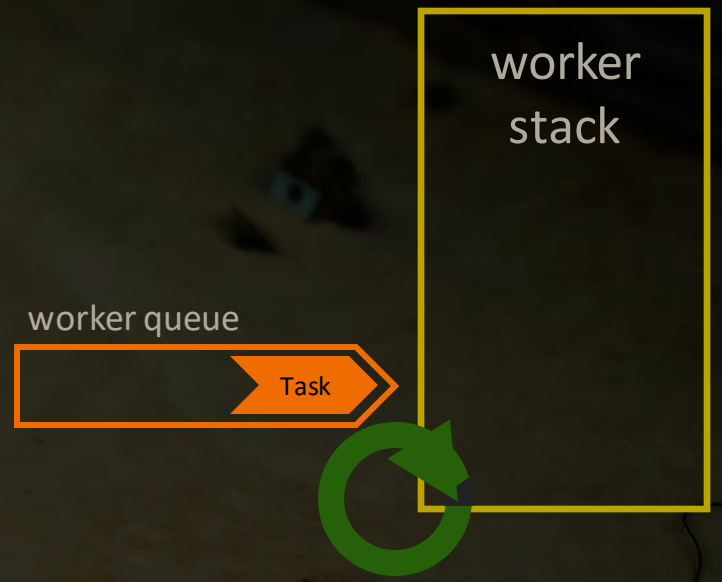
Used when we want tor run code seperated from the main event loop.

```
The highest prime number discovered so far is: <span id="pn"></span>
```

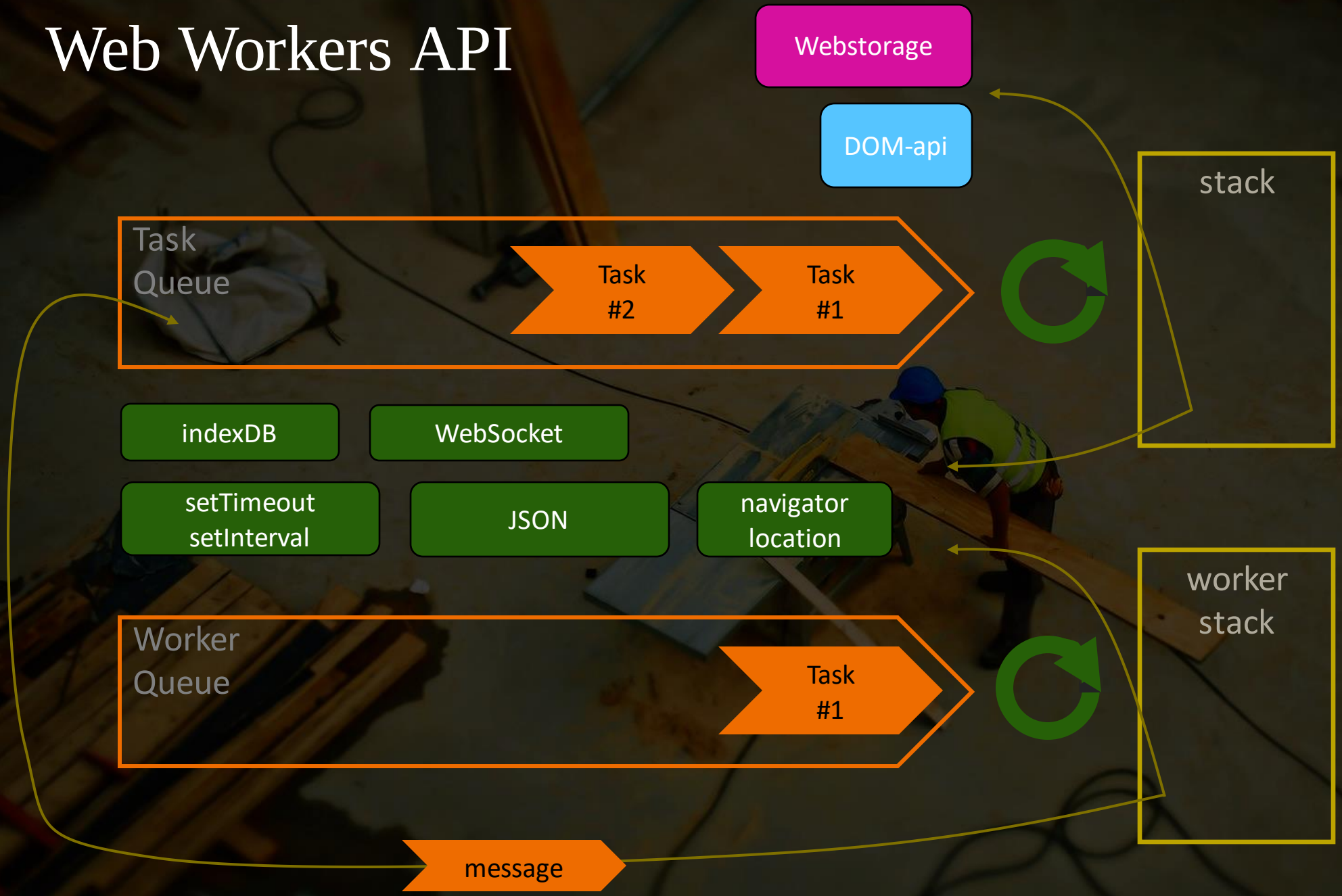
```
const worker = new window.Worker('worker.js’)
let dataContainer = document.querySelector('#pn')

worker.addEventListener('message', (event) => {
  dataContainer.textContent = event.data
})
```

worker.js

```
let n = 1

search: while(true) {
  n += 1
  for(let i = 2; i <= Math.sqrt(n); i += 1)
    if (n % i === 0)
      continue search

  postMessage(n)
}
```



Web Workers API

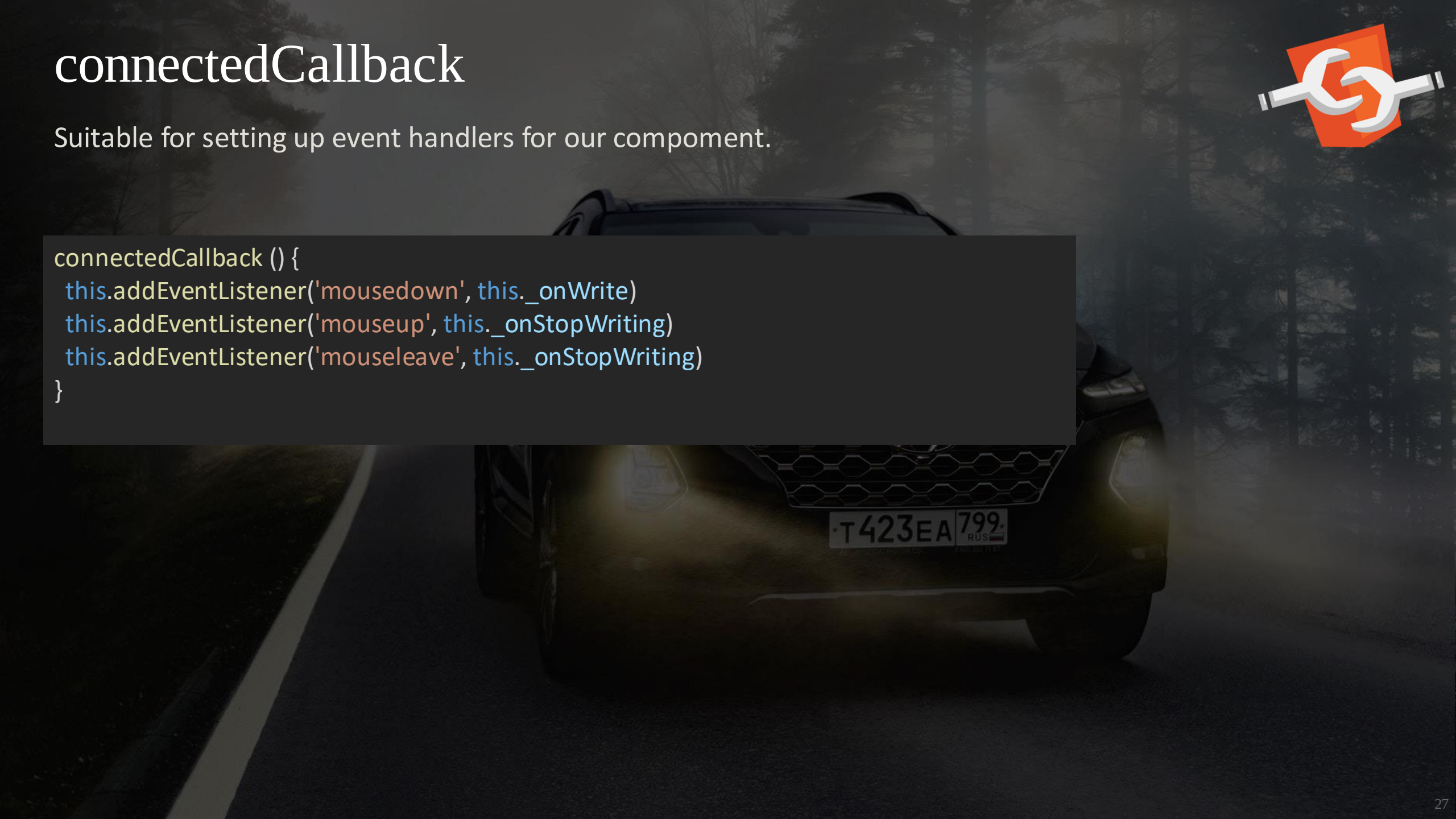


Lifecycle callbacks (custom element reactions)

- **constructor**
  Called by the browser when an element is created or upgraded
- **attributeChangedCallback**
  Called by the browser when the value of an observed attribute change (add/remove/change) and when an element is created or upgraded
- **connectedCallback**
  Called by the browser when the element is added to the DOM (or a new DOM)
- **disconnectedCallback**
  Called by the browser when the element is removed from the DOM
- adaptedCallback
  Called by the browser when the element is moved to another document.

connectedCallback

Suitable for setting up event handlers for our compoment.



```
connectedCallback () {
  this.addEventListener('mousedown', this._onWrite)
  this.addEventListener('mouseup', this._onStopWriting)
  this.addEventListener('mouseleave', this._onStopWriting)
}
```

disconnectedCallback

Suitable for removing event handlers for our compoment. *(Should not be needed)*

```
disconnectedCallback () {
  this.removeEventListener('mousedown', this._onWrite)
  this.removeEventListener('mouseup', this._onStopWriting)
  this.removeEventListener('mouseleave', this._onStopWriting)
}
```

Synthetic Events (Custom Events)

src/components/my-component/

```
const signupEvent = new window.CustomEvent('signup', { detail: myInput.value })

myCustomElement.dispatchEvent(signupEvent) // This will "trigger"/dispatch the event
```

```
let element = document.querySelector('#theCustomElement')

element.addEventListener('signup', event => {
  console.log(event.detail)
})
```

src/index.js