<epam>

DOM Events. BOM

2022

Agenda

1. INTRO
2. EVENTS
3. CUSTOM EVENTS
4. BOM
5. WINDOW OBJECT. NAVIGATION. LOCATION
6. TIMING EVENTS

INTRO

Document Object Model (DOM)

The Document Object Model (DOM) is a representation — a model — of a document and its content.

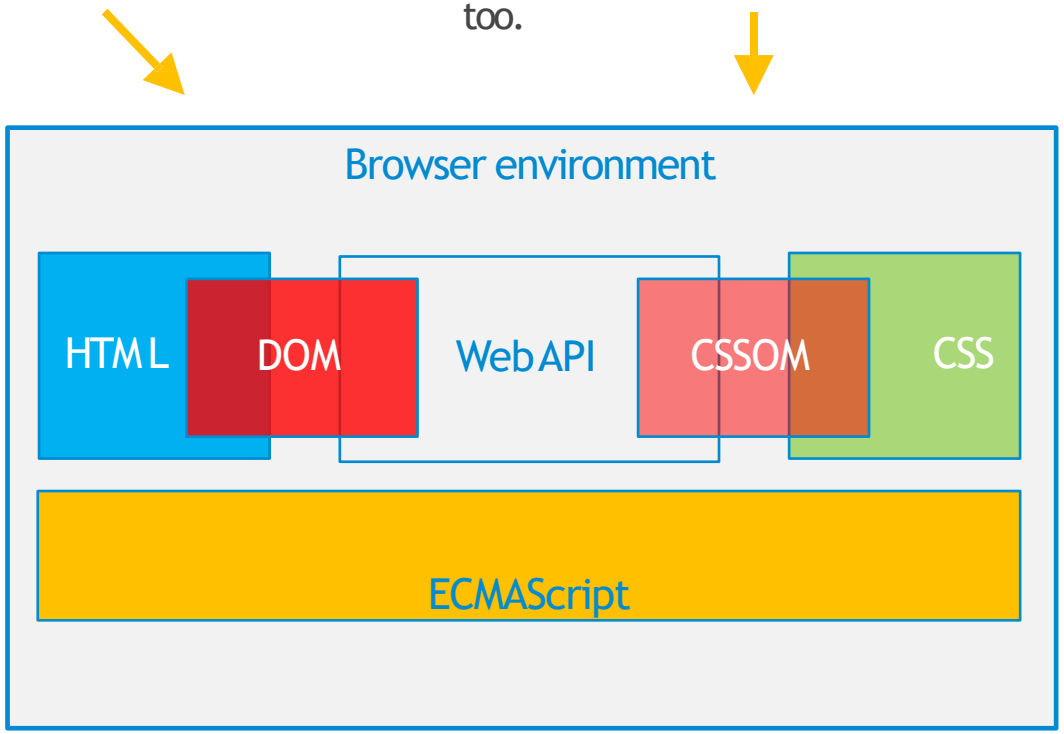
It represents the page so that programs can change the document structure, style, and content. The DOM represents the document as nodes and objects.

DOM is a web standard.

But why do we need to access the HTML markup in the first place? To understand that, we need a bit of a history lesson...

It is 2021 - you probably won't work with the Document Object Model directly, yet it is something you need to know.

Also, we have a CSS Object Model, for manipulating CSS from JavaScript.
We deliberately won't touch this, as you should not touch this in production code, too.



DOM: full access to the document

With the DOM, we can change anything in the entire tree

But is it a problem? - I hear you ask.

Let's say, that the branches of this tree is a component (or a function), and the leaves are variables. Now, just think about, in this system, you can access all of the variables in all of the components from any component.

Everything is globally accessible in this system.

Also, let me show you a real-world example of a DOM tree ...

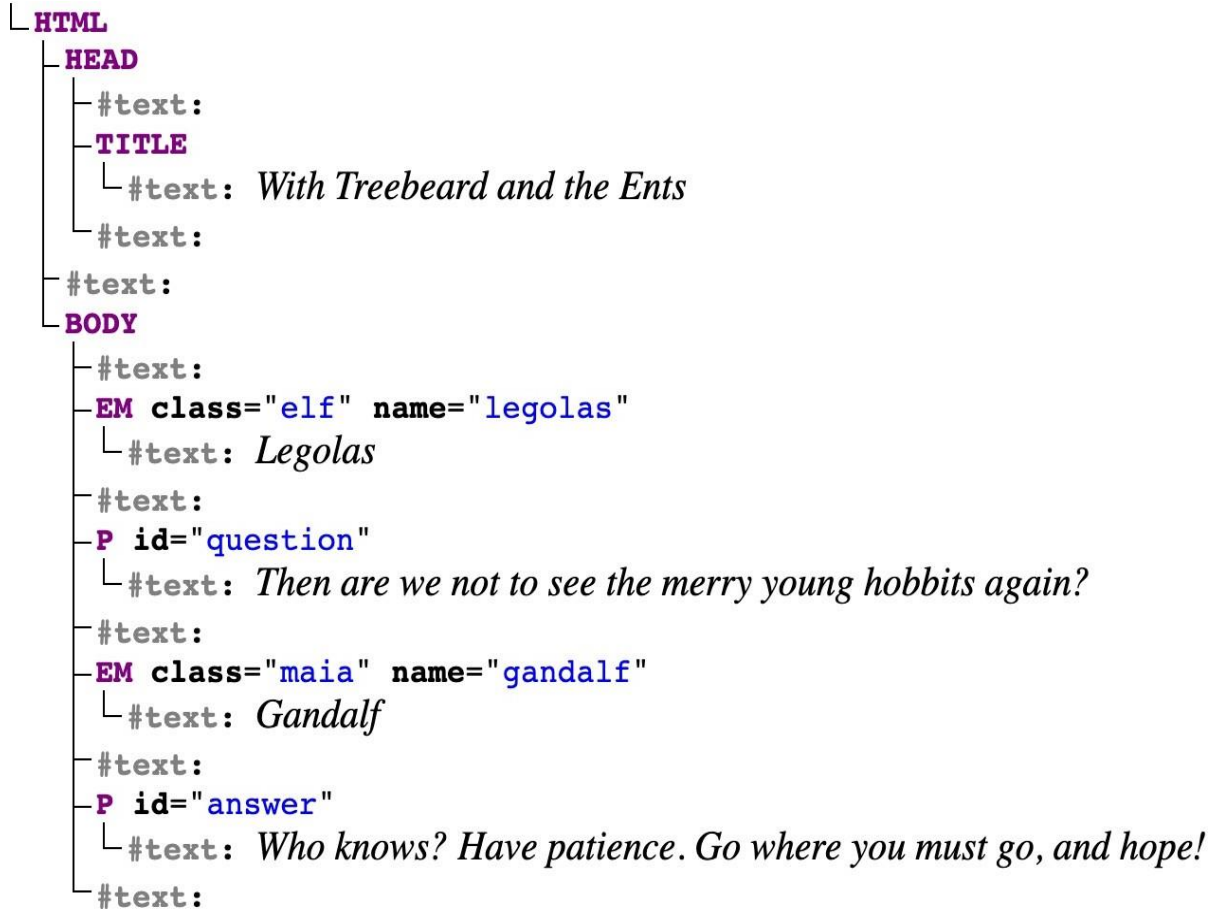
DOM is a tree.

DOM tree

When a web page is loaded, the browser creates the Document Object Model of the page: a tree

In this tree everything is a node, and every node is an object. The DOM can represent HTML or XML documents.

```
•••<html> == $0
  ▼<head>
     <title>With Treebeard and the Ents</title>
   </head>
  ▼<body>
     <em class="elf" name="legolas">Legolas</em>
     <p id="question">Then are we not to see the
              merry young hobbits again?</p>
     <em class="maia" name="gandalf">Gandalf</em>
     <p id="answer">Who knows? Have patience.
              Go where you must go, and hope!</p>
   </body>
 </html>
```



DOM - elements

Let's break down the DOM!

There are different types of nodes, and the methods return different types of collections.

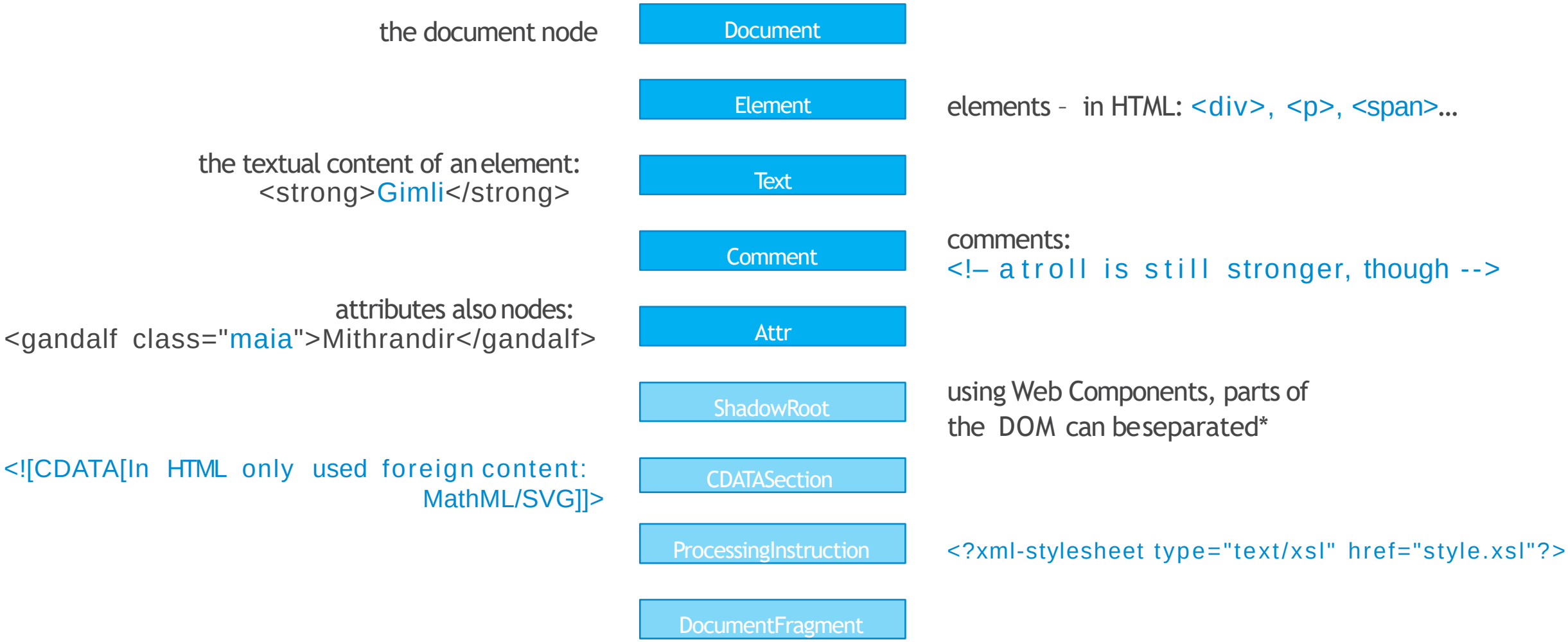
document is HTMLDocument

nodeList is HTMLCollection

node is HTMLElement

```
> document
<-  ▼#document
      <html>
      ▶<head>…</head>
      ▼<body>
          <em class="elf" name="legolas">Legolas</em>
          <p id="question">Then are we not to see the
                  merry young hobbits again?</p>
          <em class="maia" name="gandalf">Gandalf</em>
          <p id="answer">Who knows? Have patience.
                  Go where you must go, and hope!</p>
        </body>
      </html>
> document.toString();
<- "[object HTMLDocument]"
> let emNodes = document.getElementsByTagName("em");
<- undefined
> emNodes.toString();
<- "[object HTMLCollection]"
> emNodes.length;
<- 2
> emNodes[0].toString();
<- "[object HTMLElement]"
> emNodes[0];
<-    <em class="elf" name="legolas">Legolas</em>
```

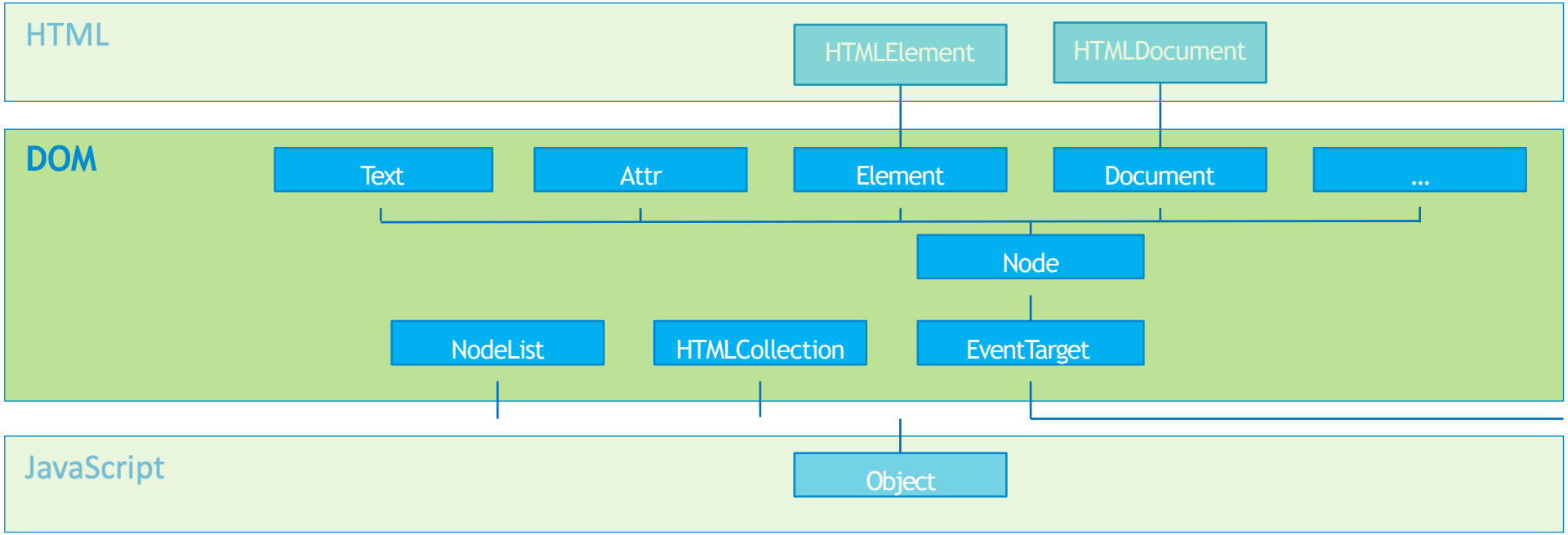
DOM - node types



* it solves a lot of problems...

DOM - object hierarchy

Without getting too deep into the topic, the object hierarchy looks like this:



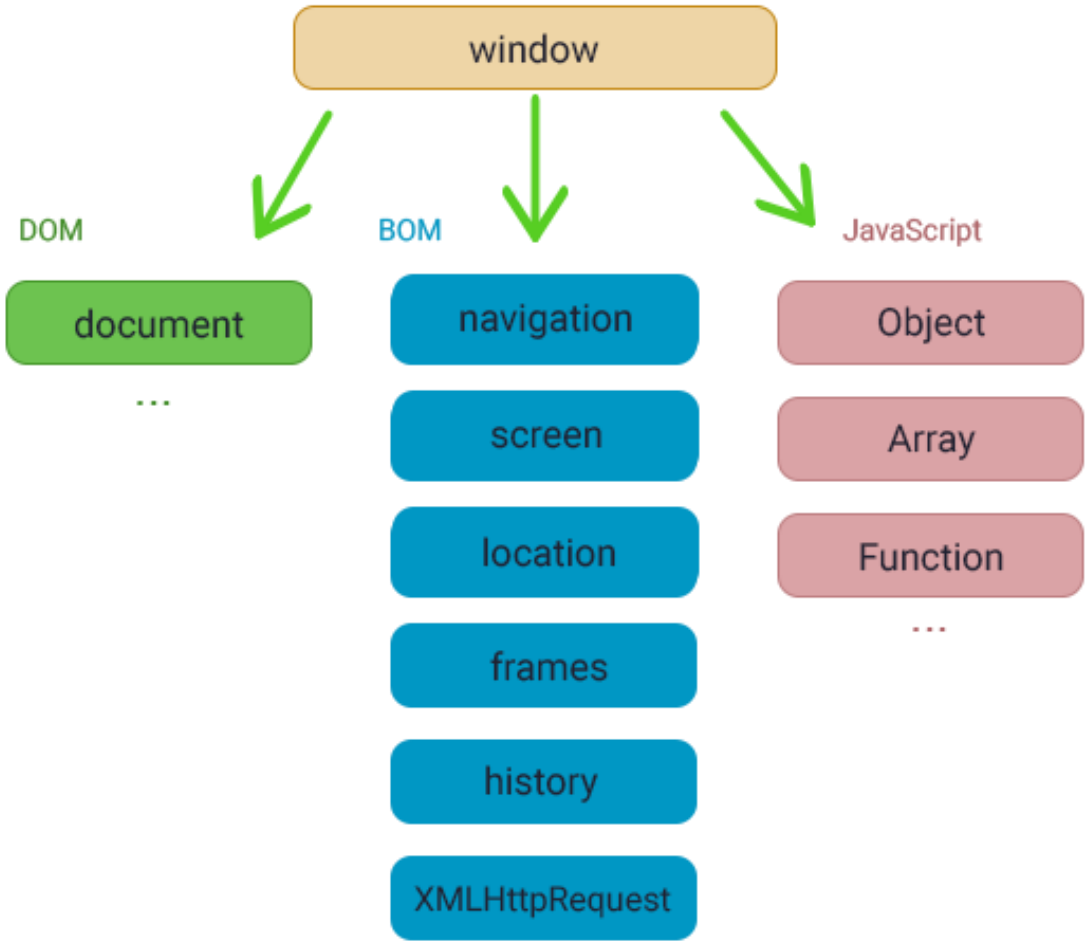
Browser Object Model

The Browser Object Model (BOM) represents additional objects provided by the browser (host environment) for working with everything except the document.

For instance:

- The navigator object provides background information about the browser and the operating system. There are many properties, but the two most widely known are: navigator.userAgent – about the current browser, and navigator.platform – about the platform (can help to differ between Windows/Linux/Mac etc).
- The location object allows us to read the current URL and can redirect the browser to a new one.

```
alert(location.href); // shows current URL
if (confirm("Go to Wikipedia?")) {
  // redirect the browser to another URL
  location.href = "https://wikipedia.org";
}
```

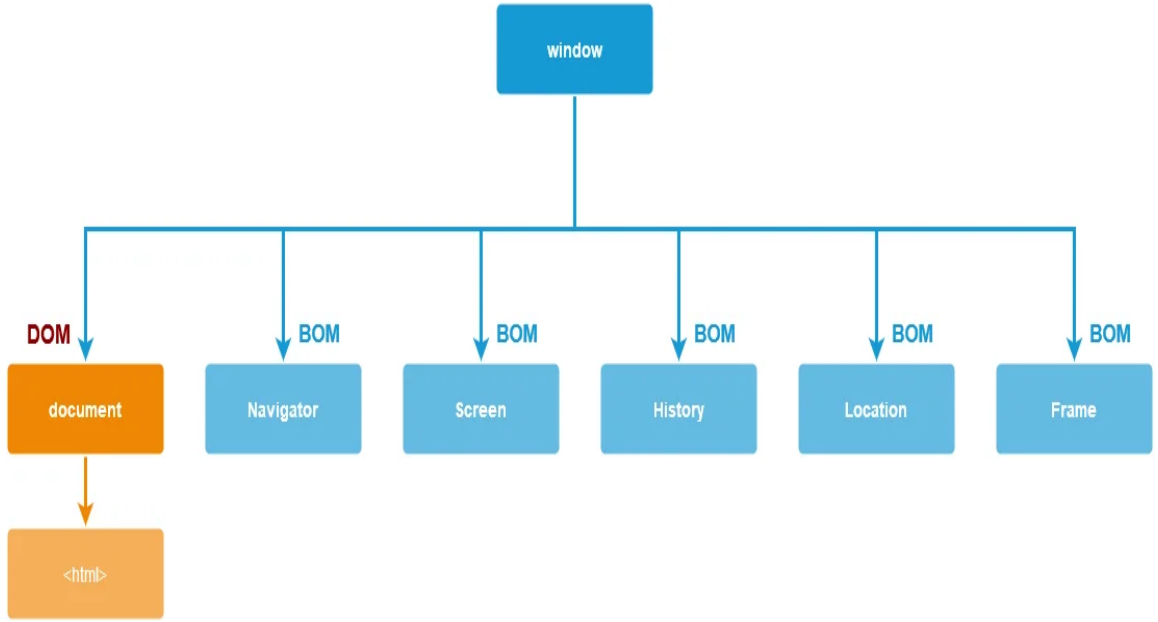


Web APIs

There are many different Web APIs

Most of them are used in specific cases, however, some are utilized on every project: we already know the Webstorage API; now, we are focusing on the Event loop, the Location and the History APIs.

All these are concerns of the host environment (browser) and are not the part of the JavaScript, but the HTML Standard.



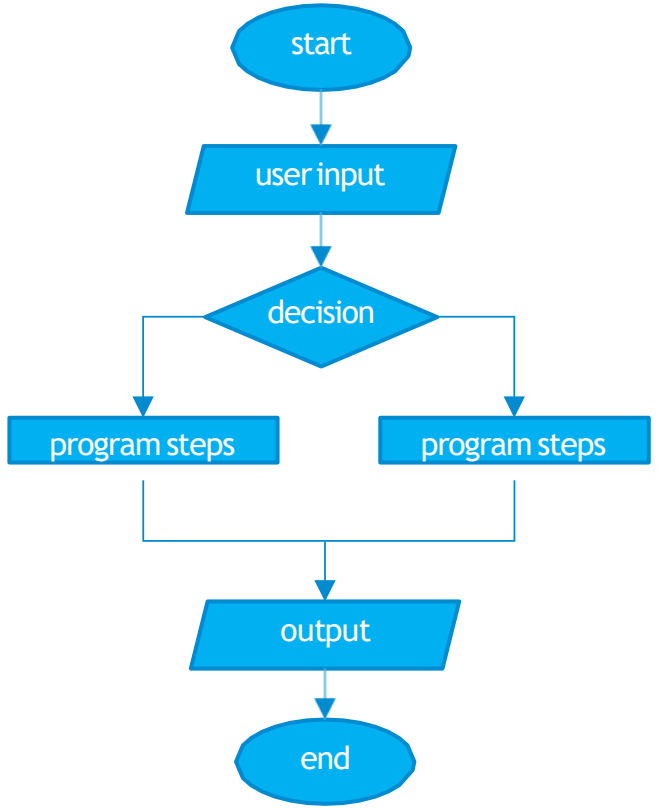
Program flow

What are events and why do we need them?

The real question is why would we need anything other than events, as programming is, in its essence, about event management.

This, however, is not obvious at first, if you consider programs, like this:

Usually, however, we develop different kind of programs...



a program flow depicted in programming books

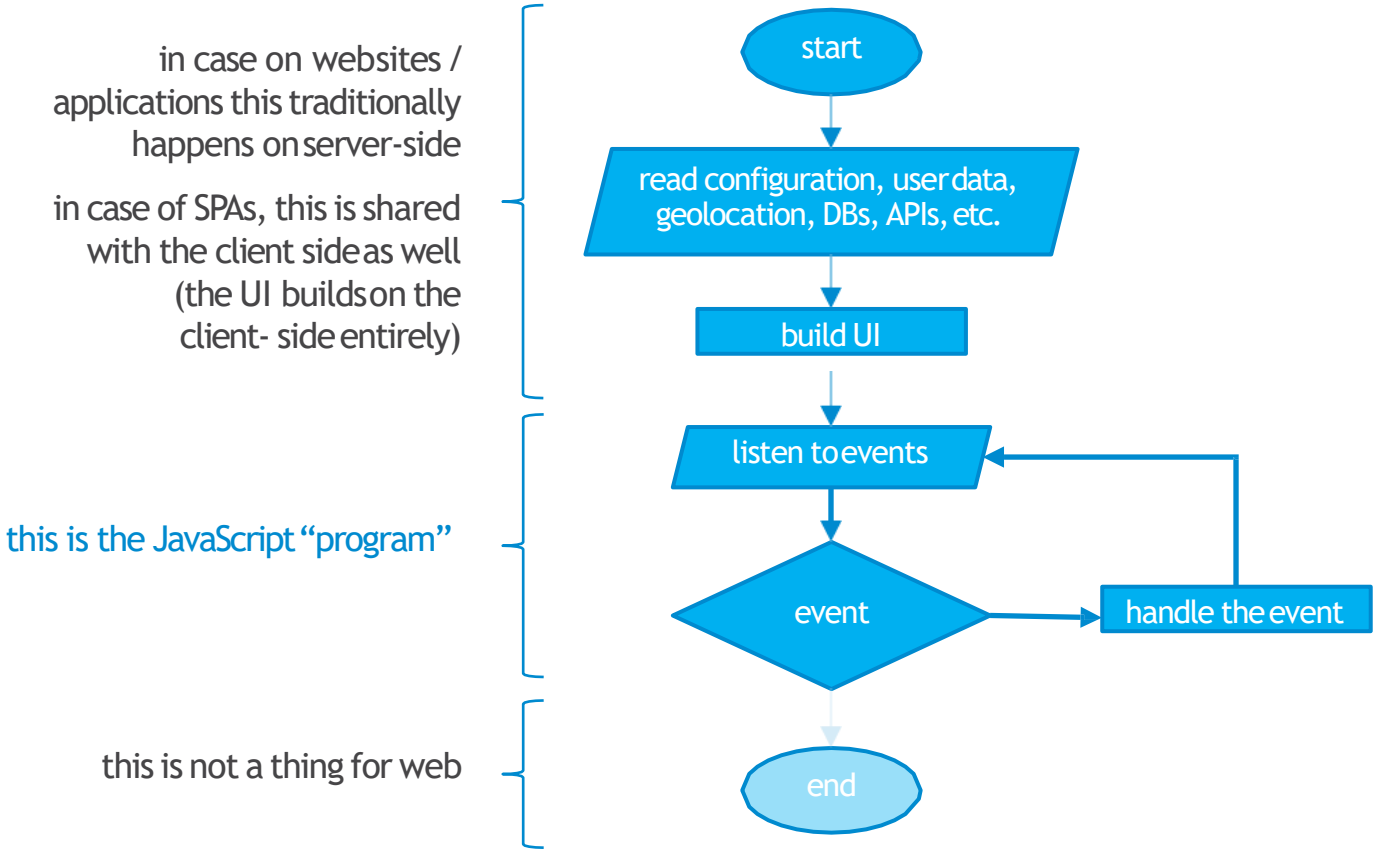
Program flow – application with User Interface

Real-life* applications rely heavily on user input

The application itself is nothing else than configuring the UI components (buttons, input fields, scrollbars) and setting up the event handlers for these.

In past, writing event handlers was *the "programming"* in JavaScript, but with SPAs, the site-build part is also done on client-side.

Still, writing event handlers is a crucial part of the web development.



a general program flow of every app with UI

**we less often write programs for mars rovers or for nuclear power plants*

EVENTS

Events

An event is a signal that something has happened that the application need to react on

Events can be user input / UI events (keyboard, pointer, input field, click, touch, scroll, submit, focus), environmental (history, network, sockets, messaging) or application dispatched events.

Event names starts with "on".

This snippet returns the available events in your browser:

```
function getEvents(obj, events =[]) {
  for (var prop in obj) if (prop.indexOf('on') ===0) events.push(prop);
  return events;
}

function getAllEvents(events ={}) {
  events['window'] = getEvents(window);
  Object.getOwnPropertyNames(window).forEach(prop => {
    try {
      const arr =getEvents(window[prop].prototype);
      events ={...events, ...(arr.length &&{[prop]: arr})};
    }
    catch {}
  });

  return events;
}
console.log(getAllEvents());
```

one simply cannot declare a variable in the parameter list in prod code, believe me ;)

this is, however, a pretty common way to add a property conditionally

Event handlers

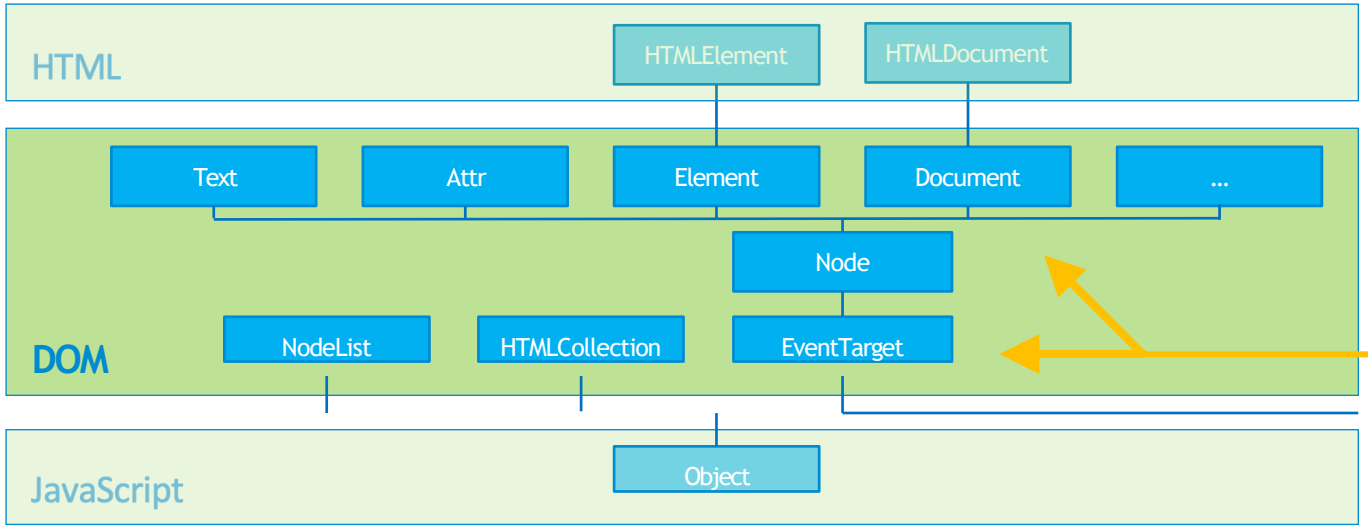
An event handler is a function, which can be assigned to the relevant property on a HTMLelement

In this lecture, we are primarily focusing on browsers as a host environment. There are events in Node.js as well, but those are different from many aspects – and Node.js definitely does not connected to any DOM. Remember this chart?

With vanilla JS you will never use this method, because your HTML and JavaScript should be separated.

With React, you will almost always use this. ¯_(ツ)_/¯

```
> document.body
<-    <body onclick="console.log('Snap!')"></body>
  Snap!
```



the document object itself is an EventTarget

Event handlers - running in a different time and space

Let me stop here just for a moment – because this *snap* will have significant consequences

The code part that we assign to the event handler, will run later – we don't know exactly when. Also, we don't know that the state we expect (DOM element, data values, user state, etc.) will exist at all when the event finally will be fired.

What we do know, that all these could be completely different, even in a way we are not prepared for.

an event occurs here...

then somewhere in the universe things start to happen...

... so that will end in an unintended way.

A classic example

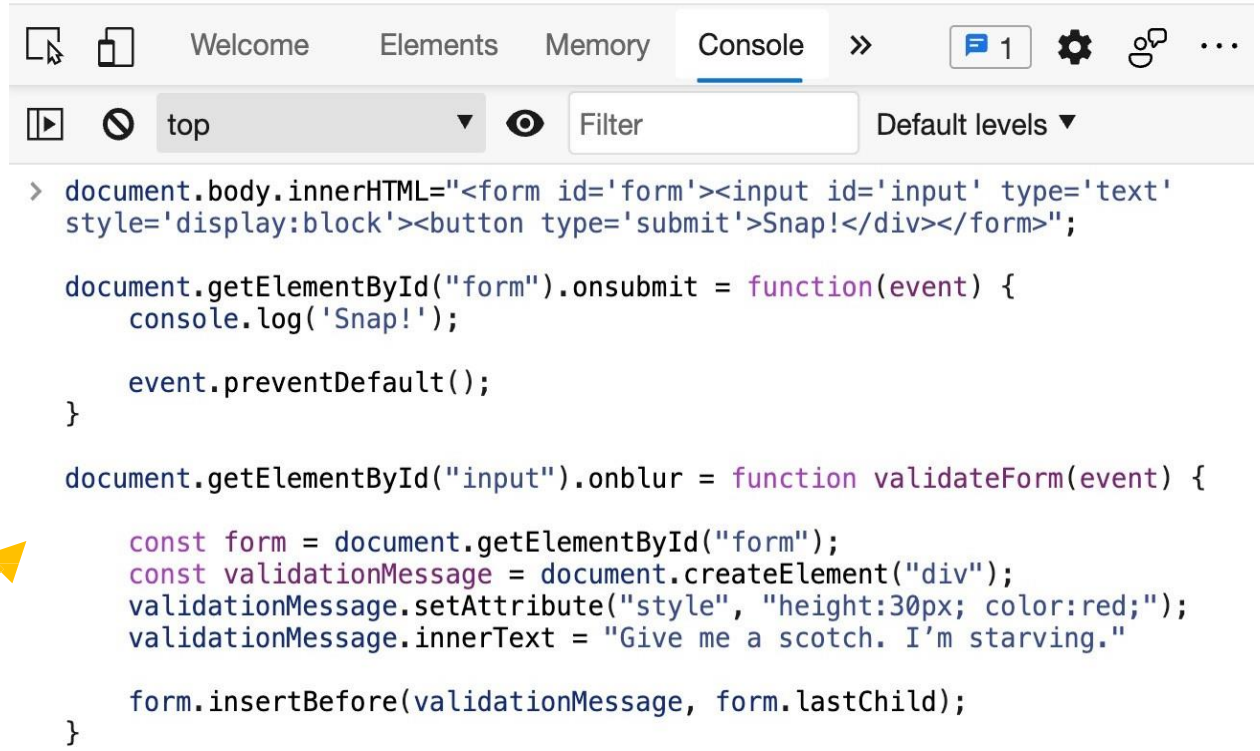
Should you be curious, here you are an example – a form

The expectation is, that when clicking on the submit button, the *onsubmit* event should fire; not necessarily to submit the form, but we probably need some additional check, or logging.

So, the *onsubmit* event should fire no matter what.

Now, guess what will happen if we write something into the input field, then click on the "Snap!" button?

a form, an input field and a button – nothing special...



... also, we have a validation message, which triggers on the onblur event of the input field – again, pretty usual

A classic example - the result

What happened?

Well, the *onsubmit* event did not fire; why? Let's go through step-by-step:

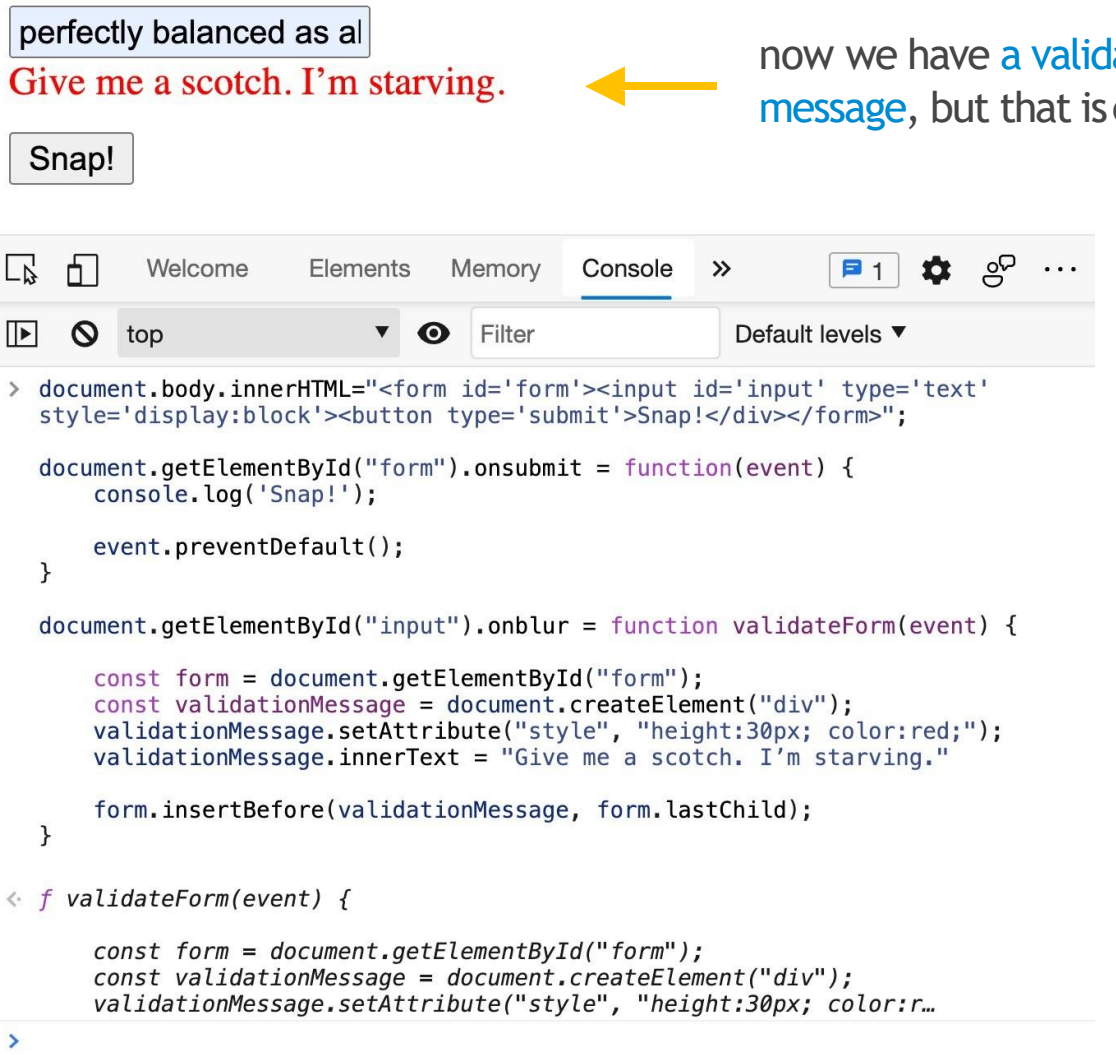
1., we click the "Snap!" button

2., but before that, we actually leave the input field, so the *onblur* event fires first

3., we show a validation message

4., but the message pushes down the button 5., so when the click would arrive to the button 6., the button is not there anymore

7., so there is no click, there is no *onsubmit* event



now we have a validation message, but that is expected

Let's analyse this!

the onclick attribute... →

will be set as property... →

therefore we can overwrite it →

```
> document.body
    <body onclick="console.log('Snap!')"></body>
> document.body.onclick
ƒ onclick(event) {
console.log('Snap!')
}
> document.body.onclick = function() {
  console.log('Another snap!');
}
ƒ () {
  console.log('Another snap!');
}
Another snap!
```

Not this way

the onclick property...

```
> document.body
    <body></body>
> document.body.onclick = function() {
    console.log("snap!");
  }
  document.body.id = "thanos";
"thanos"
snap!
> document.body
    <body id="thanos"></body>
```

...won't be synchronized with the onclick attribute. It is not obvious, because many properties (e.g. id) will.

Event object parameter

Events receives an event object as a parameter

This object's properties provide useful information about the event.

```
> document.body.onclick = function(event) {
    console.log(event);
  }
< f (event) {
    console.log(event);
  }
```

here we have a MouseEvent 

```
▼MouseEvent {isTrusted: true, screenX: 762, screenY: 294, clientX: 223, clientY: 156, …}
    altKey: false
    bubbles: true
    button: 0
    buttons: 0
    cancelBubble: false
    cancelable: true
    clientX: 223
    clientY: 156
```

CUSTOM EVENTS

.addEventListener

Event handlers can be added with .addEventListener as well

With this method, multiple handlers can be registered.


```
> document.body.addEventListener("click", function() {
    console.log('Thanos: Snap!');
  });
< undefined
> document.body.addEventListener("click", function() {
    console.log('Iron man: Hold my beer...');
  });
< undefined
  Thanos: Snap!
  Iron man: Hold my beer...
```

.removeEventListener

Event handlers can be removed as well

it is cleaner to define a handler function separately and register that

```
> const thanosSnapHandler = function() {
    console.log("Snap!");
  };

  document.body.addEventListener("click", thanosSnapHandler);
<· undefined

  Snap!

> const ironManSnapHandler = function() {
    console.log("Hold my beer...");
  };

  document.body.addEventListener("click", ironManSnapHandler);

  document.body.removeEventListener("click", thanosSnapHandler);
<· undefined

  Hold my beer...
```

Events can be created with the Event constructor as follows:

```
const event = new Event('build');

// Listen for the event.
elem.addEventListener('build', (e) => { /* … */ }, false);

// Dispatch the event.
elem.dispatchEvent(event);
```

To add more data to the event object, the CustomEvent interface exists and the detail property can be used to pass custom data. For example, the event could be created as follows:

```
const event = new CustomEvent('build', { detail: elem.dataset.time });

function eventHandler(e) {
  console.log(`The time is: ${e.detail}`);
}
```

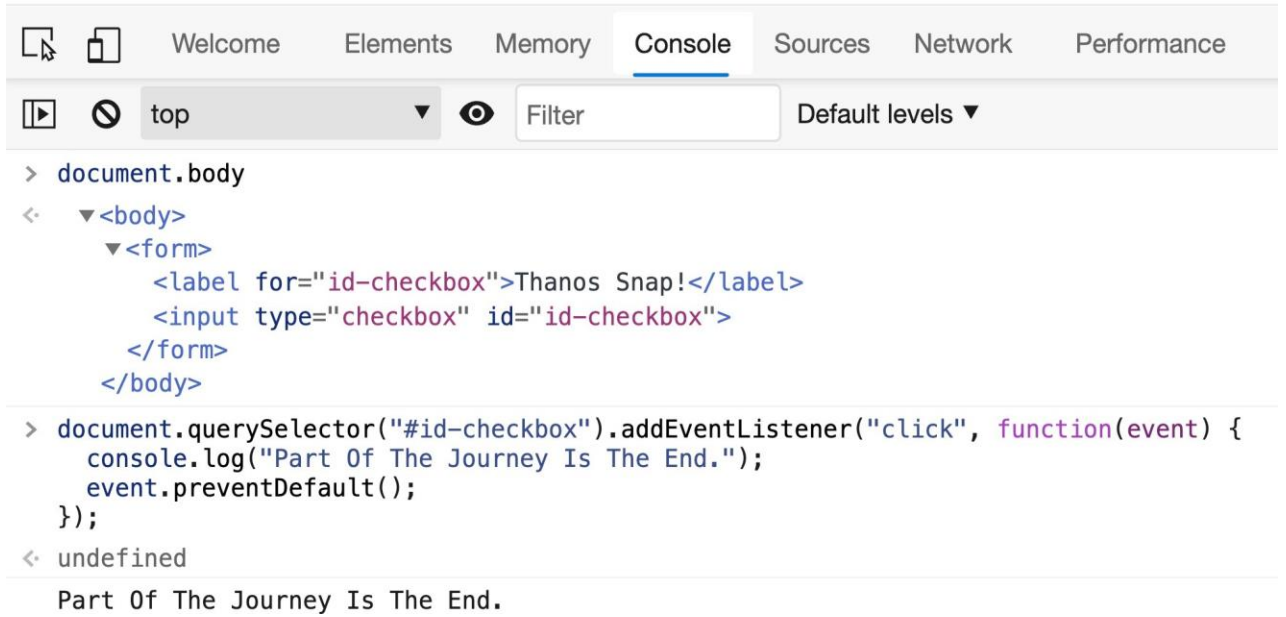
.preventDefault

The browser's default event can be prevented

this won't check now

Thanos Snap!

.preventDefault()



TYPE OF EVENTS

The browser triggers many events. A full list is available in MDN, but here are some of the most common event types and event names:

- **mouse events** (`MouseEvent`): mousedown, mouseup, click, dblclick, mousemove, mouseover, mousewheel, mouseout, contextmenu
- **touch events** (`TouchEvent`): touchstart, touchmove, touchend, touchcancel
- **keyboard events** (`KeyboardEvent`): keydown, keypress, keyup
- **form events**: focus, blur, change, submit
- **window events**: scroll, resize, hashchange, load, unload

The **MouseEvent** interface represents events that occur due to the user interacting with a pointing device (such as a mouse). Common events using this interface include click, dblclick, mouseup, mousedown.
MouseEvent derives from UIEvent, which in turn derives from Event. Though the MouseEvent.initMouseEvent() method is kept for backward compatibility, creating of a MouseEvent object should be done using the MouseEvent() constructor.

The **TouchEvent** interface represents an UIEvent which is sent when the state of contacts with a touch-sensitive surface changes. This surface can be a touch screen or trackpad, for example. The event can describe one or more points of contact with the screen and includes support for detecting movement, addition and removal of contact points, and so forth.
Touches are represented by the Touch object; each touch is described by a position, size and shape, amount of pressure, and target element. Lists of touches are represented by TouchList objects.

KeyboardEvent objects describe a user interaction with the keyboard; each event describes a single interaction between the user and a key (or combination of a key with modifier keys) on the keyboard. The event type (keydown, keypress, or keyup) identifies what kind of keyboard activity occurred.

Note: KeyboardEvent events just indicate what interaction the user had with a key on the keyboard at a low level, providing no contextual meaning to that interaction. When you need to handle text input, use the input event instead. Keyboard events may not be fired if the user is using an alternate means of entering text, such as a handwriting system on a tablet or graphics tablet.

Event interfaces

The browser provide several event interfaces available through objects

These events are browser dependent, several of them available only in a particular browser.

this snippet returns the available event objects of your browser:

```
Object.getOwnPropertyNames(window)
  .filter(e => window[e] && Object.getPrototypeOf(window[e]).prototype === Event.prototype)
  .sort();
```

AnimationEvent
AudioProcessingEvent
BeforeInputEvent
BeforeUnloadEvent
BlobEvent
ClipboardEvent
CloseEvent
CompositionEvent
CSSFontFaceLoadEvent
CustomEvent
DeviceLightEvent
DeviceMotionEvent
DeviceOrientationEvent
DeviceProximityEvent
DOMTransactionEvent
DragEvent
EditingBeforeInputEvent
ErrorEvent
FetchEvent
FocusEvent
GamepadEvent
HashChangeEvent
IDBVersionChangeEvent
InputEvent
KeyboardEvent
MediaStreamEvent
MessageEvent
MouseEvent
MutationEvent
OfflineAudioCompletionEvent
OverconstrainedError
PageTransitionEvent
PaymentRequestUpdateEvent
PointerEvent
PopStateEvent
ProgressEvent
RelatedEvent
RTCDataChannelEvent
RTCIdentityErrorEvent
RTCIdentityEvent
RTCPeerConnectionIceEvent
SensorEvent
StorageEvent
SVGEvent
SVGZoomEvent
TimeEvent
TouchEvent
TrackEvent
TransitionEvent
UIEvent
UserProximityEvent
WebGLContextEvent
WheelEvent

EVENT PHASE

Events are bubbling

Events are bubbling (by default)

If event handlers were registered for the same event for both the parent and the child, then both handler will run: first the child, then the parent.

This will continue, until the the browser will reach the root element.

we still don't use innerHTML in prod

```
document.body.innerHTML="<div style='height:100%'></div>";

const thanosSnapHandler = function() {
  console.log("Snap!");
};

const ironManSnapHandler = function() {
  console.log("Hold my beer...");
};

document.getElementsByTagName("div")[0].addEventListener("click", thanosSnapHandler);
document.body.addEventListener("click", ironManSnapHandler);
document.body;
```

```
▼<body>
   <div style="height:100%"></div>
 </body>
```

```
Snap!
Hold my beer...
```

first the child's then the parent's handler run: this is the event bubbling

Events can be capturing as well

There is an opposite mechanism as well: the capturing

By setting the third parameter of the addEventListener, the process can be reversed: first the parent's handler will run, and it will continue, until we reach the target element.

```
> document.body.innerHTML="<div style='height:100%'></div>";

  const thanosSnapHandler = function() {
    console.log("Snap!");
  };

  const ironManSnapHandler = function() {
    console.log("Hold my beer...");
  };

  document.getElementsByTagName("div")[0].addEventListener("click", thanosSnapHandler, true);
  document.body.addEventListener("click", ironManSnapHandler, true);
  document.body;

<· ▼<body>
     <div style="height:100%"></div>
   </body>

  Hold my beer...

  Snap!
```

the useCapture parameter

when the parent's handler run first
that is the capturing

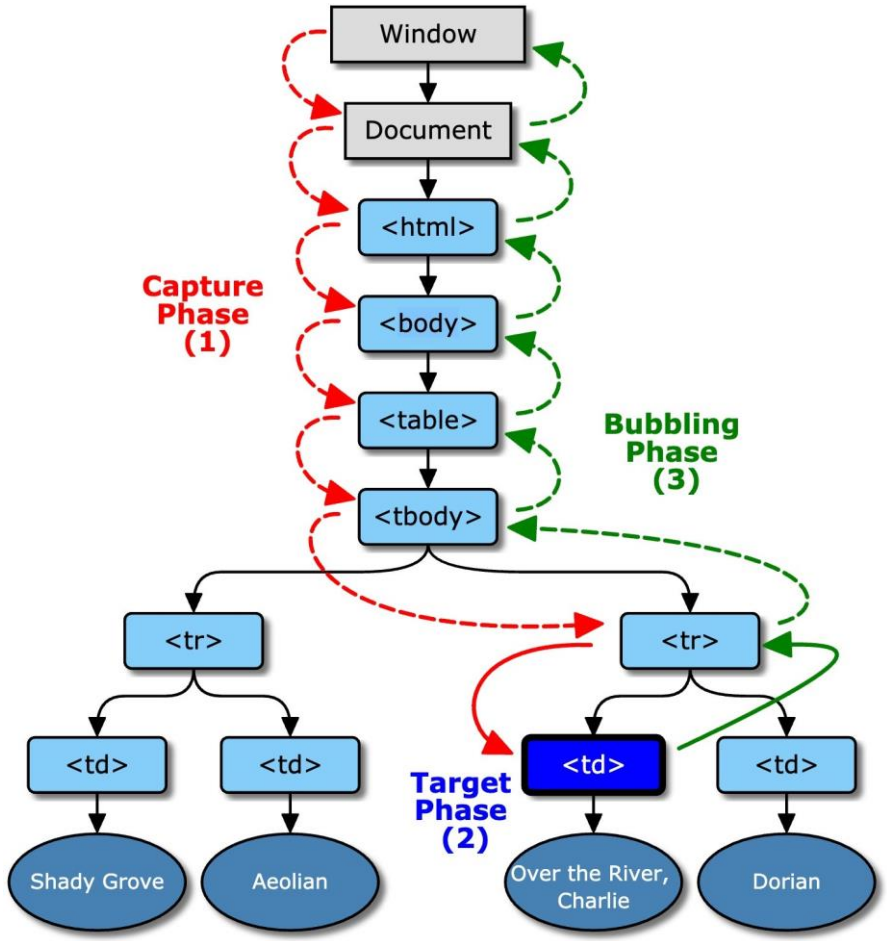
Event phases - bubbling, capturing, target

There are 3 different event phases defined in the Standard

The capture phase: The event object propagates through the target's ancestors from the Window to the target's parent.

The target phase: The event object arrives at the event object's event target. This phase is also known as the at-target phase. If the event type indicates that the event doesn't bubble, then the event object will halt after completion of this phase.

The bubble phase: The event object propagates through the target's ancestors in reverse order, starting with the target's parent and ending with the Window. This phase is also known as the bubbling phase.



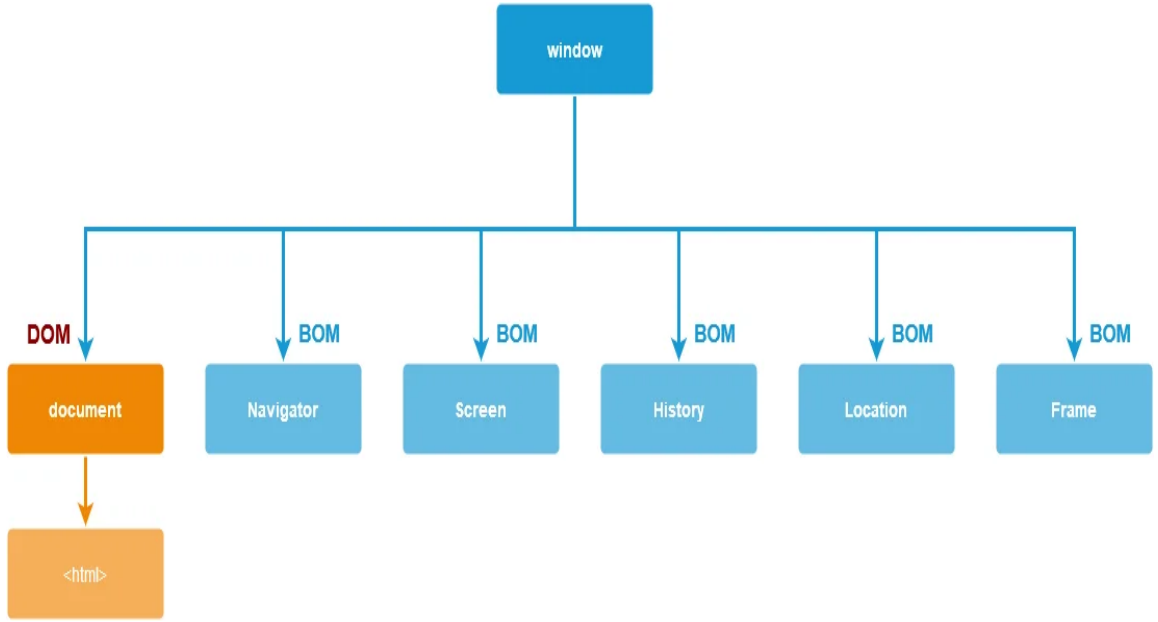
BOM

Web APIs

There are many different Web APIs

Most of them are used in specific cases, however, some are utilized on every project: we already know the Webstorage API; now, we are focusing on the Event loop, the Location and the History APIs.

All these are concerns of the host environment (browser) and are not the part of the JavaScript, but the HTML Standard.



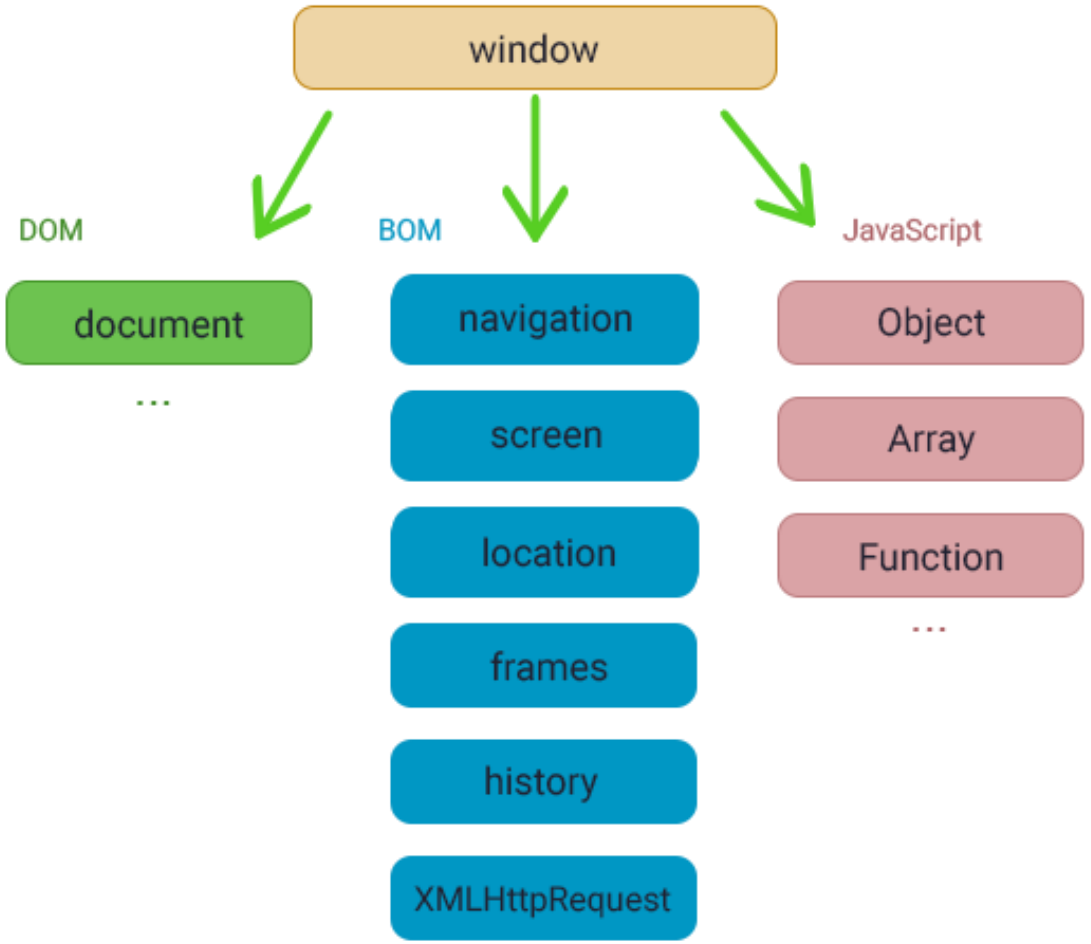
Browser Object Model

The Browser Object Model (BOM) represents additional objects provided by the browser (host environment) for working with everything except the document.

For instance:

- The navigator object provides background information about the browser and the operating system. There are many properties, but the two most widely known are: navigator.userAgent – about the current browser, and navigator.platform – about the platform (can help to differ between Windows/Linux/Mac etc).
- The location object allows us to read the current URL and can redirect the browser to a new one.

```
alert(location.href); // shows current URL
if (confirm("Go to Wikipedia?")) {
  // redirect the browser to another URL
  location.href = "https://wikipedia.org";
}
```



There are no official standards for the Browser Object Model (BOM).

Since modern browsers have implemented (almost) the same methods and properties for JavaScript interactivity, it is often referred to, as methods and properties of the BOM.

The Window Object

The window object is supported by all browsers. It represents the browser's window.

All global JavaScript objects, functions, and variables automatically become members of the window object.

- Global variables are properties of the window object.

- Global functions are methods of the window object.

Even the document object (of the HTML DOM) is a property of the window object

```
window.document.getElementById("header");
```

is the same as:

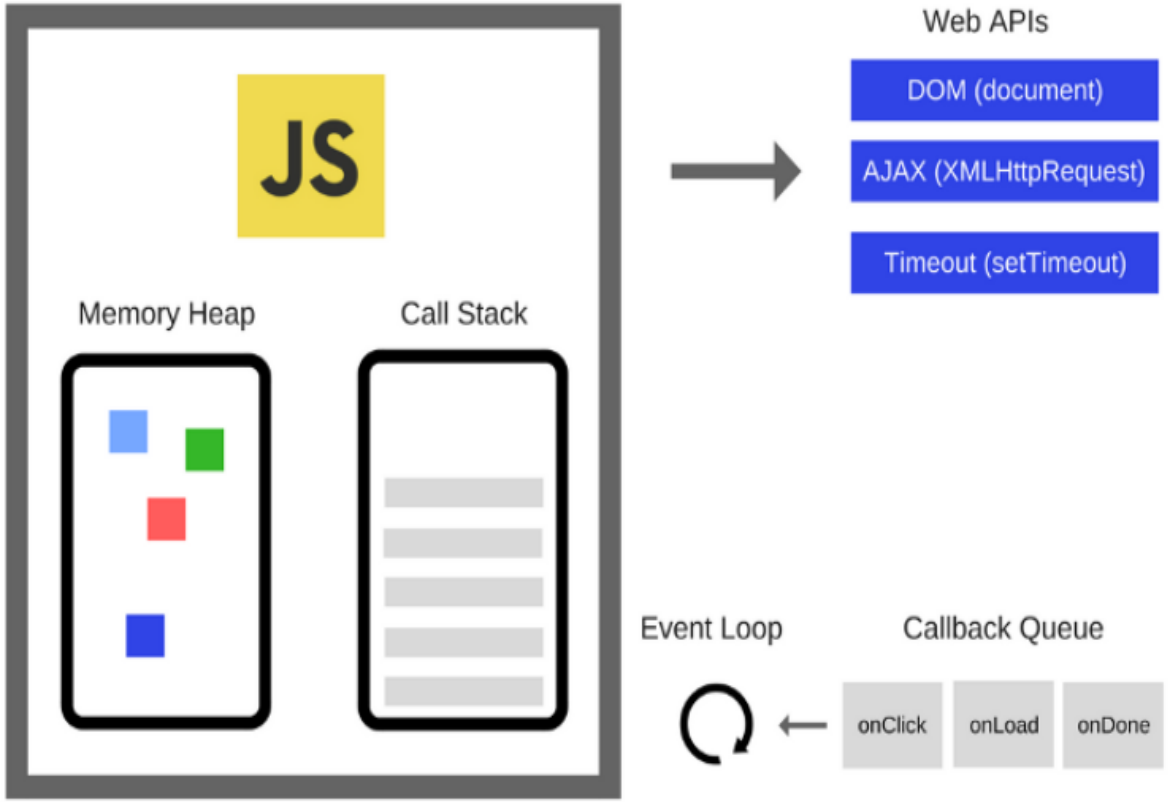
```
document.getElementById("header");
```

WHAT IS EVENT LOOP

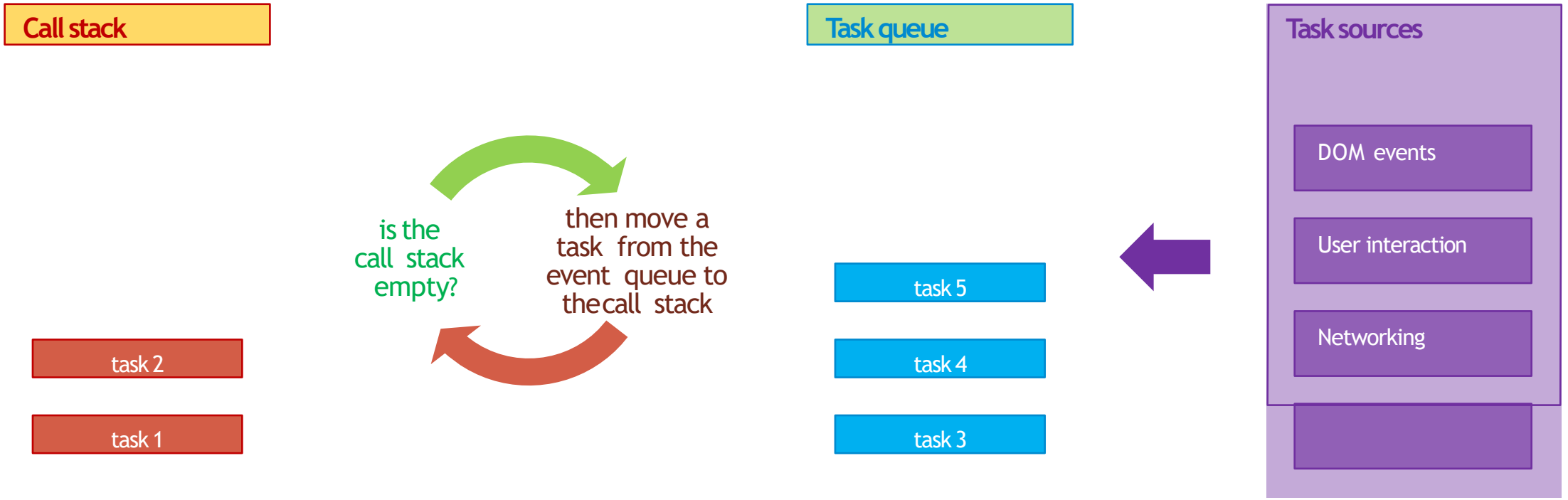
Event Loop

JavaScript has a concurrency model based on an event loop, which is responsible for executing the code, collecting and processing events, and executing queued sub-tasks. This model is quite different from models in other languages like C and Java.

The event loop is a programming construct or design pattern that waits for and dispatches events or messages in a program. The event loop works by making a request to some internal or external "event provider" (that generally blocks the request until an event has arrived), then calls the relevant event handler ("dispatches the event"). The event loop is also sometimes referred to as the message dispatcher, message loop, message pump, or run loop.



Event loop once again



the setTimeout callback must wait until the entire call stack will be empty

the setTimeout callback will be placed here *after the delay time*

the events come from different sources, and the browser could prioritise between them

TIMING EVENTS

setTimeout

It is easier to understand what is going on if we play a bit with setTimeout.

This code seems pretty straightforward: the browser tries to execute the function after the timer expires.

```
> setTimeout("console.log('This is eval.'.replace('a', 'i'))", 500);
< 1
  This is evil.
```

works with a string as well, but you won't use it

The callback function to be executed.

the delay is 500 ms

returns a timer ID
(with this the timer can be cancelled)

after a delay, it "~~executes~~"* the code

```
> const cageSays = function() {
      console.log("What I am about to tell you sounds crazy.");
  }

  setTimeout(cageSays, 500);

< 1
  What I am about to tell you sounds crazy.
```

* *actually, this is not the case - we will see the details now*

setTimeout - does not wait, it is async!

Things start to be complicated when we realize that setTimeout does not actually wait

When we set a callback to be executed it actually does what the name suggests: it only sets up a timer and the execution continues without any waiting.

So, when it will actually run? Well, after a 500ms delay, that's for sure, but when exactly?

You may ask: how to sleep the execution in JavaScript then? The short answer is: there is no internal function for sleep - you have to implement that.

this will be immediately executed →

tadam! →

```
> const cageSays = function() {
      console.log("What I am about to tell you sounds crazy.");
  }

  setTimeout(cageSays, 500);

  console.log("But you have to listen to me.");

  But you have to listen to me.
< undefined
  What I am about to tell you sounds crazy.
```

a sleep() function at you disposal

Here you are, a simple sleep function

What is interesting here, however, is not that awesome sleep function,
but while the browser is working on that loop, it stops doing anything else.

It does it because the JavaScript is a single threaded language.

try to click as much as you can!

it does not really care...
until it finishes that pretty while loop

the good news is the event loop is
still registering the click events
(will be explained a bit later)

```
> (function sleep(delay, initialTime = Date.now()) {
    while (Date.now() < initialTime + delay);
    console.log("For you? Judgement day.");
})(10000);

window.onclick = function() { console.log("What day is it?"); }

For you? Judgement day.
ƒ () { console.log("What day is it?"); }
90 What day is it?
```

setTimeout - runs async, later, but much how later?

Let's develop it heuristically!

I am sure that after a bit of trial and error we will figure it out, shall we? Our presumptions are:

we schedule 500ms waiting here

500ms should be finished for now!

so every console.logs should run according to the numbers, right?

```
> function sleep(delay, initialTime = Date.now()) {
    while (Date.now() < initialTime + delay);
}

const cageSays = function() {
    console.log("1. What I am about to tell you sounds crazy.");
};

(function () {
    setTimeout(cageSays, 500);

    sleep(1000);

    console.log("2. But you have to listen to me.");
})();

console.log("3. Your very lives depend on it.");
```

setTimeout - could run much later

Wrong!

Hmm, it seems it waits until the function finishes...

But wait, the console.log placed outside the function runs before as well!

```
const cageSays = function() {
    console.log("1. What I am about to tell you sounds crazy.");
};

(function () {
    setTimeout(cageSays, 500);

    sleep(1000);

    console.log("2. But you have to listen to me.");
})();

console.log("3. Your very lives depend on it.");
2. But you have to listen to me.
3. Your very lives depend on it.
<· undefined
1. What I am about to tell you sounds crazy.
```

these look in order... →

but our poor callback runs last →

setTimeout - later, and we don't know, when

All these are the result of the event loop

Once we understand how the event loop works, it will all make sense!

```
const cageSays = function() {
    console.log("5. …that we've had this conversation.");
};

(function () {
    (function () {
        (function () {
            setTimeout(cageSays, 500);
                                                    this ends up in 4000 ms
            sleep(1000);
            console.log("1. What I am about to tell you sounds crazy.");
        })();
        sleep(1000);
        console.log("2. But you have to listen to me.");
    })();
    sleep(1000);
    console.log("3. Your very lives depend on it.");
})();

sleep(1000);
console.log("4. You see this isn't the first time… ");
```

```
1. What I am about to tell you sounds crazy.
2. But you have to listen to me.
3. Your very lives depend on it.
4. You see this isn't the first time…
undefined
5. …that we've had this conversation.
```

setTimeout - breakdown

these all are built on the call stack already

```
const cageSays = function() {
    console.log("5. …that we've had this conversation.");
};

(function () {
    (function () {
        (function () {
            setTimeout(cageSays, 500);

            sleep(1000);
            console.log("1. What I am about to tell you sounds crazy.");
        })();
        sleep(1000);
        console.log("2. But you have to listen to me.");
    })();
    sleep(1000);
    console.log("3. Your very lives depend on it.");
})();

sleep(1000);
console.log("4. You see this isn't the first time… ");
```

the callback won't run, just will be moved to the task queue after the delay

```
1. What I am about to tell you sounds crazy.
2. But you have to listen to me.
3. Your very lives depend on it.
4. You see this isn't the first time…
undefined
5. …that we've had this conversation.
```

setTimeout - async calls if meet...

Let's see a real world situation: we have to execute a task after something has been finished

Can we use the setTimeout for that?

it could be anything (server call, component rendering), we can't see its internals, we just have to wait for that

```
> const _cageSaysFirst = function() {
      console.log("1. What I am about to tell you sounds crazy.");
  };

  const cageSaysFirst = function() {
      setTimeout(_cageSaysFirst, 1000);
  };

  const cageSaysSecond = function() {
      console.log("2. But you have to listen to me.");
  };

  cageSaysFirst();
  setTimeout(cageSaysSecond, 500);
< 2
  2. But you have to listen to me.
  1. What I am about to tell you sounds crazy.
```

so we wait a bit

but it could not be enough - and while *it may work* on your workstation, it could fail at the visitor

A key takeaway –

“soldier, you never use setTimeout to wait for anything async, am I clear?”

Use a callback or an event for that – but setTimeout is never a solution.

What result would we get?

```
for (var i = 0; i < 10; i++) {
    setTimeout(function () {
        console.log(i);
    }, 100);
}
```

```
for (var i = 0; i < 10; i++) {
    (function (i) {
        setTimeout(function () {
            console.log(i);
        }, 100);
    })(i)
}
```

```
for (var i = 0; i < 10; i++) {
    setTimeout(function (i) {
        console.log(i);
    }, 100, i);
}
```

```
for (let i = 0; i < 10; i++) {
    setTimeout(function () {
        console.log(i);
    }, 100);
}
```

[The most common solutions]

setInterval

setInterval schedules a task for running periodically

And it does it really, but due to the event loop, it could lead into surprises.

100ms interval could mean...

Just ~10ms between the end of the task and the start of the task.

If the scheduled task takes longer, then the interval loop will consume the whole CPU time

now it is fine, but you never rely on that the task will be finished on time

```
let ticks = 0;
let sleepTime = 90;

const cageSays = function() {
    console.log(ticks, "starting", new Date().getMilliseconds());
    sleep(sleepTime);
    console.log(ticks, "ending", new Date().getMilliseconds());

    if (ticks === 3) {
        sleepTime = 0;
        console.log("Where's your helmet?");
    }
    if (ticks === 6) {
        clearInterval(intervalID);
    }

    ticks++;
};

const intervalID = setInterval(cageSays, 100);
```

a long (90ms) task here

```
undefined
0 "starting" 227
0 "ending" 317
1 "starting" 325
1 "ending" 415
2 "starting" 426
2 "ending" 517
3 "starting" 524
3 "ending" 614
Where's your helmet?
4 "starting" 626
4 "ending" 626
5 "starting" 724
5 "ending" 724
6 "starting" 829
6 "ending" 829
```

instead of setInterval - use recursive setTimeout

It is a usual pattern using recursive setTimeout calls instead of setIntervals

```
let ticks = 0;
let sleepTime = 90;

const cageSays = function() {
    console.log(ticks, "starting", new Date().getMilliseconds());
    sleep(sleepTime);
    console.log(ticks, "ending", new Date().getMilliseconds());

    if (ticks < 3) {
        setTimeout(cageSays, 100);
    }

    ticks++;
};

cageSays();
```

a recursive setTimeout call

remember: if you have a recursion, you need a condition as well to stop

proper 100ms intervals between the end and the start

```
0 "starting" 800
0 "ending" 890
undefined
1 "starting" 992
1 "ending" 82
2 "starting" 187
2 "ending" 279
3 "starting" 382
3 "ending" 472
```

The location API is important basically in 2 cases:

1. when you want to know your current position

2. when you want to set your target

NAVIGATOR

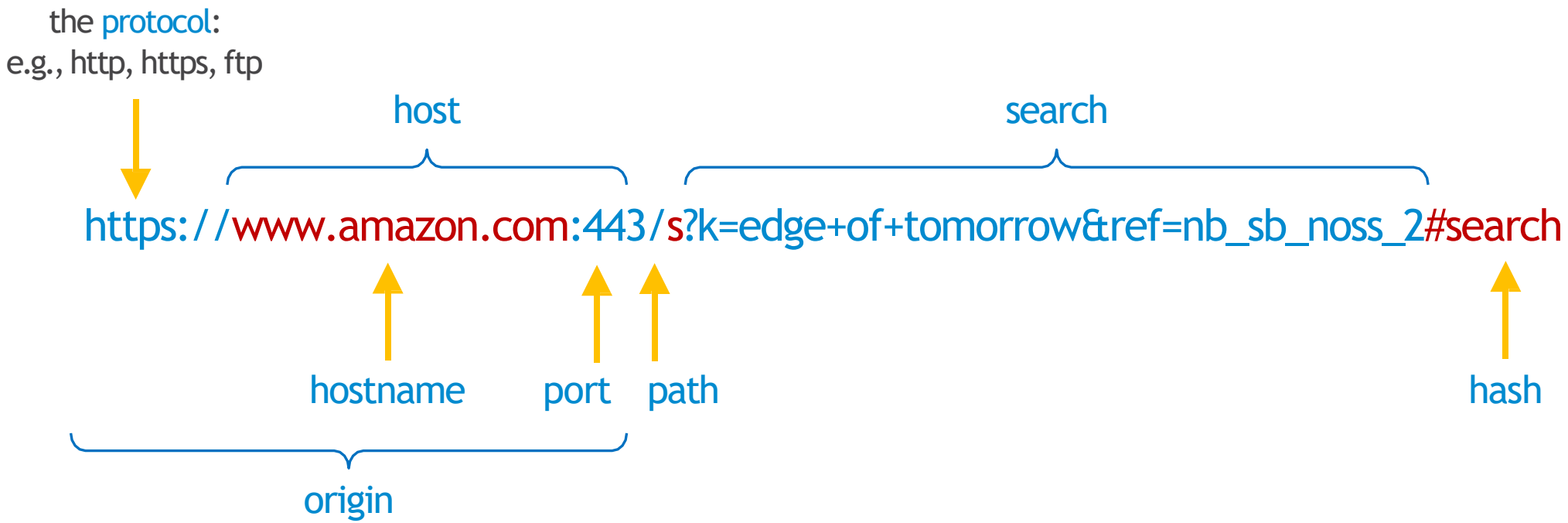
The ***window.navigator*** object contains information about the visitor's browser. It's the part of window object.

The most common properties:
- ***Navigator.cookieEnabled*** - Returns false if setting a cookie will be ignored and true otherwise
- ***Navigator.geolocation*** - Returns a Geolocation object allowing accessing the location of the device
- ***Navigator.language*** - Returns a string representing the preferred language of the user, usually the language of the browser UI. The null value is returned when this is unknown.

Read more: https://developer.mozilla.org/en-US/docs/Web/API/Navigator

LOCATION

Location - URL



Location

URL

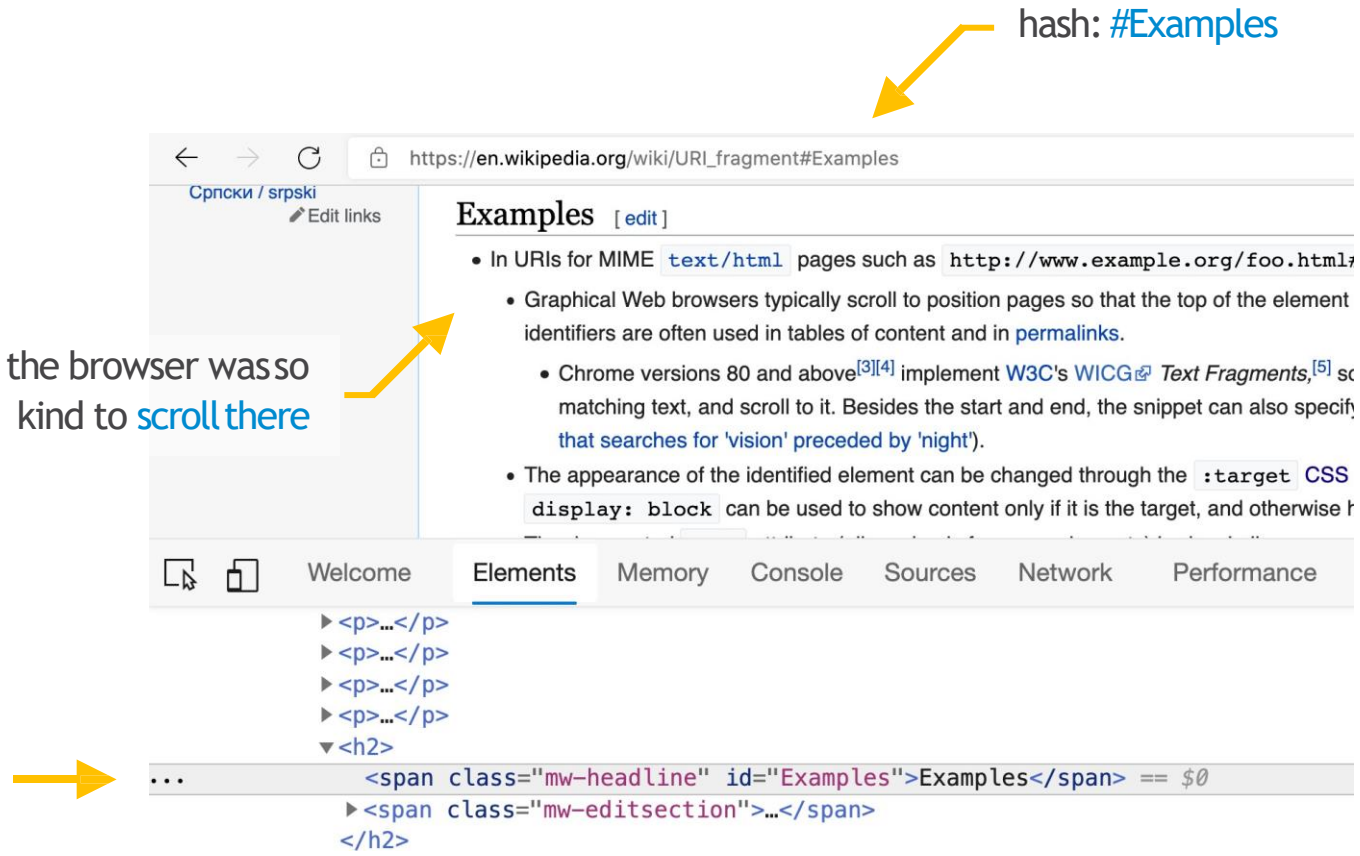
```
> location
< Location {ancestorOrigins: DOMStringList, href: "https://www.amazon.com/s?k=edge+of+tomorrow&ref=nb_sb_noss_2",
  origin: "https://www.amazon.com", protocol: "https:", host: "www.amazon.com", …}
    ▶ ancestorOrigins: DOMStringList {length: 0}
    ▶ assign: ƒ assign()
      hash: ""
      host: "www.amazon.com"
      hostname: "www.amazon.com"
      href: "https://www.amazon.com/s?k=edge+of+tomorrow&ref=nb_sb_noss_2"
      origin: "https://www.amazon.com"
      pathname: "/s"
      port: ""
      protocol: "https:"
    ▶ reload: ƒ reload()
    ▶ replace: ƒ replace()
      search: "?k=edge+of+tomorrow&ref=nb_sb_noss_2"
    ▶ toString: ƒ toString()
    ▶ valueOf: ƒ valueOf()
      Symbol(Symbol.toPrimitive): undefined
    ▶ __proto__: Location
```

the URL parts are accessible via the location object

Location - Hash

Having a hash value in the URL the browser should scroll to the relevant part of the document

This happens at page reload, but hash can be added in JavaScript as well, and when this code runs, the browser should scroll as well.



Location - Hash

Hash seems harmless at the first sight, however, it opens up a whole lot of new possibilities (and bugs)

Sometimes, there is a request from the Product Owner, that the page should scroll to a specified position after the page loading.

While the #hash can be used for that, it is the browser's concern to decide how and when to process that. Relying on the hash scroll could lead to surprises.

window.scrollTo(x-coord, y-coord) also can be used, but you need to be very careful *when to do that* - the page rendering takes time.

Hash also can be used for special purposes (e.g., communication between iframes - many times you will need to integrate 3rd party iframes, and while now there are other methods for that, it is still can be used by those iframes)

even there is an event for that

```
> window.onhashchange = function(event) {
      console.log("My middle name... is... " + location.hash);
  };

  location.hash="Rose";
< "Rose"
```

My middle name... is... #Rose

Location - navigate

When setting the location.href, the browser will navigate to a new URL

Navigation could be possible with location.replace() as well - the difference is, that location.href will save the original URL to the browser's history.

```
> location
  Location {ancestorOrigins: DOMStringList, href: "https://www.amazon.com/s?k=edge
▼ +of+tomorrow&ref=nb_sb_noss_2", origin: "https://www.amazon.com", protocol: "htt
  ps:", host: "www.amazon.com", …} i
> location.href = "https://epam.com";
"https://epam.com"
Navigated to https://www.epam.com/
```

History

The browser history can be modified, also, actions can be performed (back, forward, go)

This is a significant responsibility in Single Page Applications, as the navigation between “pages” in SPAs are not natively supported by browsers (there is only one page from the browser’s perspective).

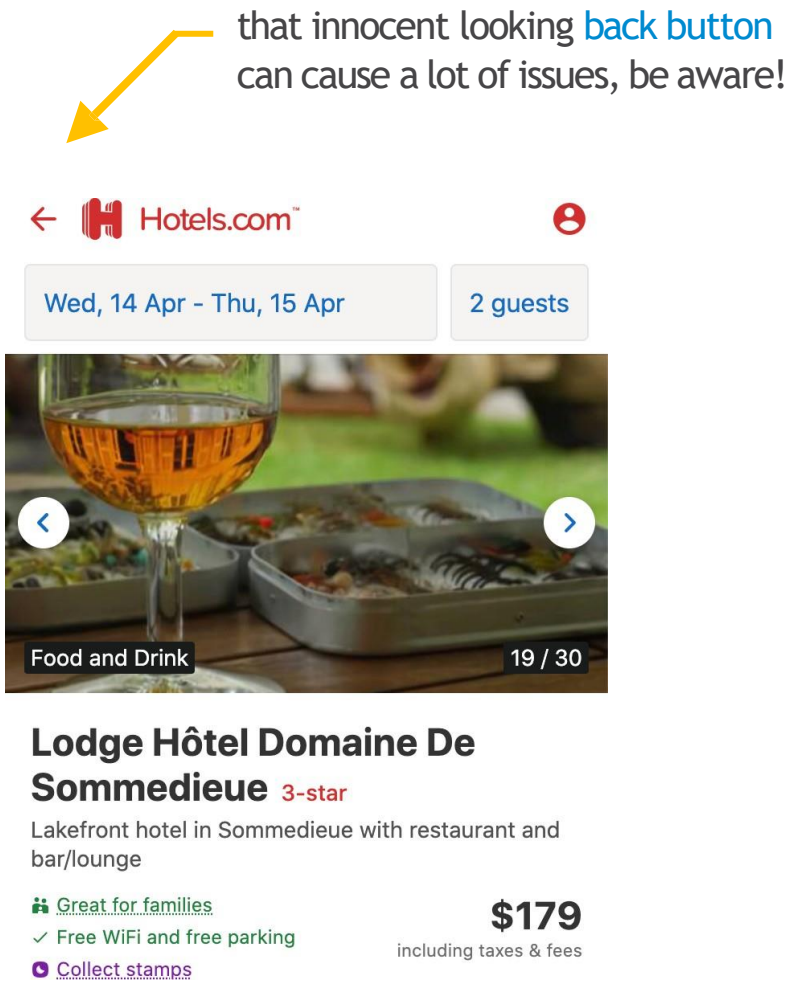
a simple bug in the history management of a web application - no worries, business as usual

```
> location
<· Location {ancestorOrigins: DOMStringList, href: "https://www.epam.com/",
  origin: "https://www.epam.com", protocol: "https:", host: "www.epam.co
  m", …}
> history.back();
<· undefined
Navigated to https://www.amazon.com/s?k=edge+of+tomorrow&ref=nb_sb_noss_2
> history.forward();
<· undefined
Navigated to https://www.epam.com/
```

History

Also, mobile web applications tend to have navigation UI elements

Those would need special attention to handle properly, as there are a lot of edge cases there - please test the application thoroughly.



THANK YOU!