web programming

intro to JavaScript & the DOM

oli

rec

agenda

1. intro to JavaScript
2. language features
3. the DOM
4. JavaScript events

1. intro to JavaScript

what is JavaScript?

- JS
- it's a **programming language**, *yay!* 🎉
- it's a programming language that your browser can execute natively

scripts in web pages

```
<!DOCTYPE html>
<html>

  <head>
    ...
    <script src="filename.js"></script>
  </head>

  <body>
    ...
  </body>

</html>
```

how does it work?

- the browser requests the referenced JavaScript file
- the server sends the file to the browser
- the browser executes the file **immediately**

JavaScript execution

- **no "main" method**
  - executed from top to bottom
- **no compilation** by the developer
  - compiled and executed on the fly by the browser (*)

(*): this is called Just-In-Time (JIT) compilation

2. language features

generalities

- multi-paradigm, dynamic language
  - with types and operators, standard built-in objects
- syntax based on Java and C
- supports OOP and FP

types

- `Number`
- `String`
- `Boolean`
- `Object`
- `Function`
- `Symbol`

numbers

- floating point real numbers
  - no integer type
- special values: `NaN`, `+Infinity`, `-Infinity`

number examples

```
const a = 1;
const b = 4.0001;
const c = 2e3;

// numbers are also objects.
(1).toString();  // => '1'
```

strings

- represent text
- sequences of Unicode characters
- enclosed in quotes (single, double, or backticks)
- can check size via `length` property

string examples

```
let message = 'hello';
message += ', world';

message.length;  // => 12

const multiline = 'multi\nline?';

const interpolated = `${message}!`;
```

boolean

- possible values: `true` or `false`
- ANY value can be converted to a boolean:
    - `false`, `0`, `''`, `null`, `undefined`, and `NaN` become `false` (**falsy** values)
    - all other values become `true` (**truthy** values)

boolean examples

```
const a = true;
const b = false;
const c = b || !a;
const d = b && a;
```

arrays

- special type of object
- used to create lists of data
- `0`-based indexing
- can check size via `length` property

array examples

```
const colors = ['aquamarine'];

colors[0];  // => 'aquamarine'

colors.push('deeppink');

colors[1];  // => 'deeppink'

colors.length;  // => 2
```

objects

- everything in JavaScript is an object
- collections of name-value pairs
  - name is a JavaScript string
  - value can be any JavaScript value

object examples

```
const greetings = {
  common: 'hello',
  cool: 'hey',
  rec: 'ola k ase',
};

greetings.cool;  // => 'hey'.

const name = 'rec';
greetings[name];  // => 'ola k ase'.
```

`null` and `undefined`

- `null` indicates a deliberate non-value
- `undefined` indicates an uninitialized value, e.g. variables defined without value

operators

- numeric operators: **`+`**, **`-`**, **`*`**, **`/`**, and **`%`**
  - **`+`** also does string concatenation
- assignments: **`=`**
  - compound assignments, e.g. **`+=`**
- increment and decrement: **`++`** and **`--`**

more operators

- logical operators: **`&&`**, **`||`**, and **`!`**
  - they use short-circuit logic
- ternary operator: **`?`**
- comparisons: **`<`**, **`>`**, **`<=`**, and **`>=`**
  - equality is not that straightforward

equality

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

equality

- `==` and `!=` are basically broken
  - they do an implicit type conversion
- `===` and `!==` were added to keep the existing behavior of `==` and `!=`
- **always use `===` and `!==`**

equality

```
   '' === '0'           // false
   '' === 0             // false
    0 === '0'           // false
  NaN === NaN           // false(*)
 [''] === ''            // false
false === undefined     // false
false === null          // false
 null === undefined     // false
```

(*): still weird

variables

- declared using `let`, `const`, or `var`
- **`let`** declares block-scoped variables
- **`const`** declares block-scoped variables that cannot be reassigned
- **`var`** declares function-scoped variables

`let` example

```
for (let i = 0; i < 5; i++) {
  // i is only visible in here.
}

// i is not visible out here.
```

const example

```
const pi = 3.14;
pi = 3.14159;  // throws an error 🤮

const constants = { pi };
constants.golden = 1.61;  // does not throw.
```

`var` example

```
for (var i = 0; i < 5; i++) {
  // i is visible in here.
}

// i is visible out here.

// i is visible to the whole function!
```

variables best practices

- use `const` by default
- use `let` only if rebinding is needed
- **`var` shouldn't be used**

control structures

- similar set to Java or C languages
- conditional statements: `if/else`
- loops: `while`, `do-while`, and `for`
- additional loops: `for..in` and `for..of`

`if/else` example

```javascript
const user = 'John';
const bypassAuth = false;

if (user) {
  // ...
} else if (bypassAuth) {
  // ...
} else {
  // ...
}
```

`while` example

```
let number = 42;
while (number % 13 > 0) {
  number += 1;
}
```

do-while example

```
let answer;
do {
  answer = getAnswer();
} while (answer !== 'y');
```

`for` example

```
const squares = [];
for (let i = 0; i < 5; i++) {
  squares.push(i ** 2);
}
```

for..in example

```
const faces = {
  flipando: '🤯',
  pillo: '😏',
};

for (const face in faces) {
  const emoji = faces[face];
  console.log(`{emoji} is "{face}"`);
}
// => e.g. 😏 is "pillo".
```

for..of example

```
const faces = ['😦', '🤯'];

for (const face of faces) {
  console.log(face);
}
// => e.g. 🤯
```

functions

- first-class objects
- composed of a sequence of statements
- can take parameters
- can use `return` to return a value at any time
  - if nothing is explicitly returned, JavaScript returns `undefined`

function examples

```
const isEven = function(number) {
  return number % 2 === 0;
};

// declaration notation.
function avg(numbers) {
  let sum = 0;
  for (const number of numbers) {
    sum += number;
  }
  return sum / numbers.length;
}
```

exercise

```
/**
 * Returns the longest of the given
 * strings.
 *
 * @param  {Array<String>} strings
 * @return {String}
 */
function longestString(strings) {
  // your code goes here!
}
```

3. the DOM

what is the DOM?

- **D**ocument **O**bject **M**odel
- **representation** of the **web page document** created by the browser
- allows JavaScript to **access the content** and elements of the document **as objects**

DOM tree and nodes

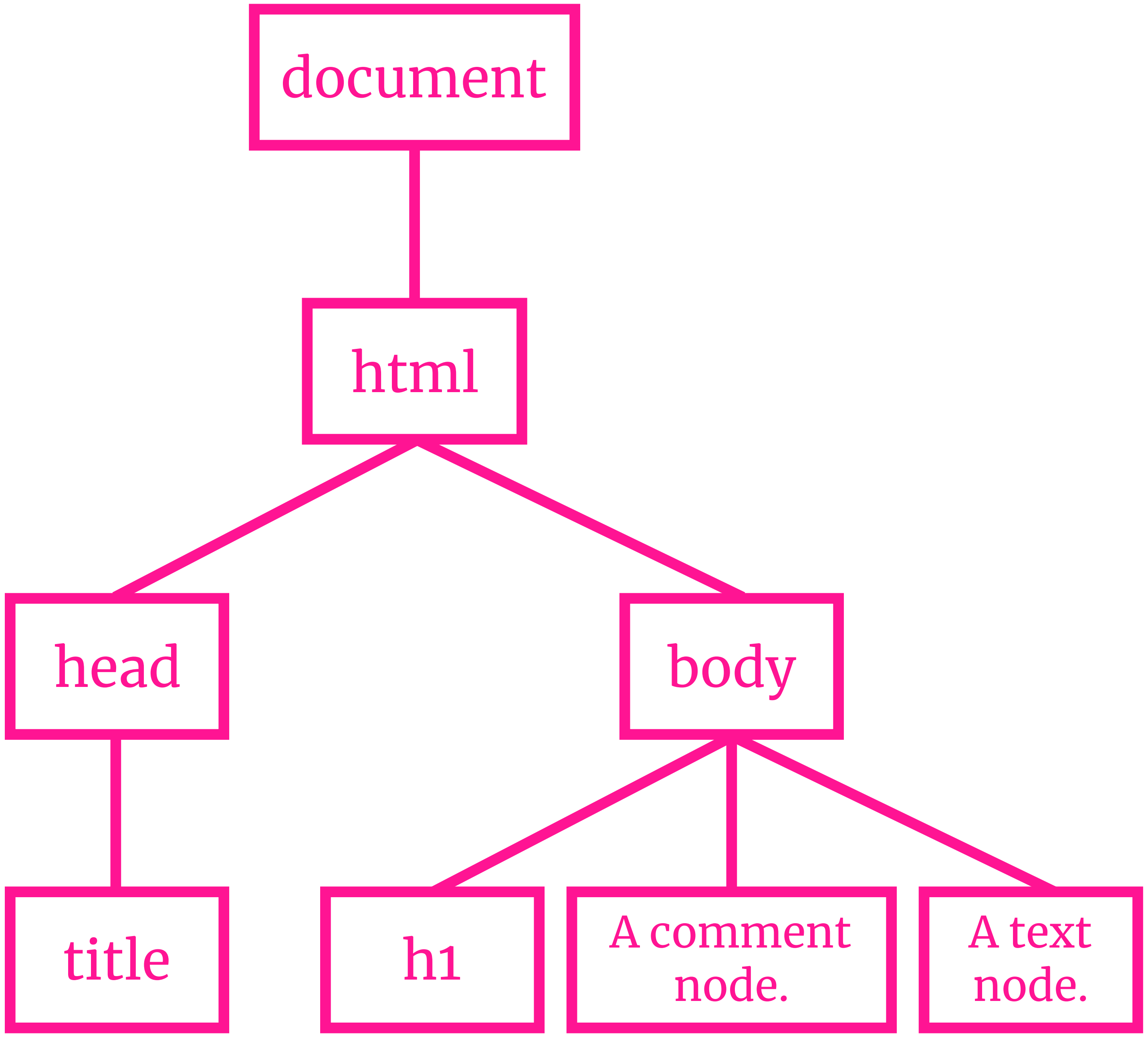
- **tree** of objects called **nodes**
- **document** node at the root
- three main types of nodes:
  - **element** nodes
  - **text** nodes
  - **comment** nodes

```
<!DOCTYPE html>
<html>

<head>
    <title>Web Programming</title>
</head>

<body>
    <h1>An element node</h1>
    <!-- A comment node. -->
    A text node.
</body>

</html>
```



minimal document

property	node	node type
`document`	**`#document`**	`DOCUMENT_NODE`
`document.documentElement`	**`html`** `(*)`	`ELEMENT_NODE`
`document.head`	**`head`** `(*)`	`ELEMENT_NODE`
`document.body`	**`body`** `(*)`	`ELEMENT_NODE`

(*): since these elements are so common, they have their own properties on the document

`document` object

- built-in, property of the `window` (*) object
- **allows to manipulate web pages** via properties and methods, e.g. `document.body` and `document.querySelector('.foo')`

(*): global, top-level object representing a tab in the browser

accessing the DOM

- usually done through **element** nodes
- using methods of the `document` object:
  - `getElementById()`
  - `getElementsByClassName()`
  - `getElementsByTagName()`
  - `querySelector()`
  - `querySelectorAll()`

`getElementById()`

- easiest way to access a single element
- element must have an `id` attribute

```
<div id="nav"></div>

document.getElementById('nav');
```

getElementsByClassName()

- access one or more elements **by class**
- returns an array!

```
<li class="item"></li>
<li class="item"></li>
```

```
document.getElementsByClassName('item');
```

getElementsByTagName()

- less specific way to access elements
- returns an array!

```
<p>Foo</p>
<p>Bar</p>
```

```
document.getElementsByTagName('p');
```

querySelector()

- access a single element that matches a CSS selector
- similar to jQuery's `$('...')`

```
<div id="nav"></div>

document.querySelector('#nav');
```

querySelectorAll()

- access **all** the elements that match a CSS selector
- returns an array!

```
<li class="item"></li>
<li class="item"></li>
```

```
document.querySelectorAll('.item');
```

summary

id	`#foo`	`getElementById('foo')`
class	`.foo`	`getElementsByClassName('foo')`
tag	`p`	`getElementsByTagName('p')`
selector (single)		`querySelector('#foo')`
selector (all)		`querySelectorAll('.foo')`

some `Element` properties

property	description
`id`	the value of the `id` attribute of the element, as a string
`classList`	an object containing the classes applied to the element
`textContent`	the text content of a node and its descendants (inherited from `Node`)
`innerHTML`	the raw HTML between the starting and ending tags of an element, as a string

`classList`

- control classes applied to an HTML element
- add classes with the `add()` method, e.g. `link.classList.add('active')`
- remove classes with the `remove()` method, e.g. `item.classList.remove('hidden')`

`textContent`

- get or set the text content of an element node
- setting this property on a node **removes all of its children** and replaces them with a single text node with the given value

`innerHTML`

- get or set the HTML content of an element node
- cross-site scripting (XSS) risk, use `textContent` instead ⚠️
- it's ok to use it to remove all children: `element.innerHTML = '';`

traversing the DOM

- **move through the DOM** without specifying each and every element beforehand
- nodes in the DOM are referred to as *parents*, *children*, and *siblings*, depending on their relation to other nodes

parent node

- node that is **one level above** a given node
- accessible via two properties:
  - `parentNode` gets parent node (most common)
  - `parentElement` gets parent **element** node

parent example

HTML

```html
<body>
    <section>
        <h1>Heading</h1>
    </section>
</body>
```

JAVASCRIPT

```javascript
const headingElem = document.querySelector('h1');
headingElem.parentNode; // => <section>
```

children nodes

- nodes that are **one level below** a given node
  - *nodes beyond that level are referred to as "descendants"*
- properties to traverse all nodes: `childNodes`, `firstChild`, `lastChild`
- properties to traverse only element nodes: `children`, `firstElementChild`, `lastElementChild`

children example

HTML

```html
<body>
    <section>
        <h1>Heading</h1>
        <p>Paragraph</p>
    </section>
</body>
```

JAVASCRIPT

```javascript
const sectionElem = document.querySelector('section');
sectionElem.children; // => [<h1>, <p>]
```

sibling nodes

- any node on the **same tree level** as the given node
- properties to traverse all nodes: `previousSibling`, `nextSibling`
- properties to traverse only element nodes: `previousElementSibling`, `nextElementSibling`

sibling example

HTML

```html
<body>
    <section>
        <h1>Heading</h1>
        <p id="p1">1st paragraph</p>
        <p id="p2">2nd paragraph</p>
    </section>
</body>
```

JAVASCRIPT

```javascript
const p1Elem = document.querySelector('#p1');
p1Elem.previousElementSibling; // => <h1>
p1Elem.nextElementSibling; // => <p id="p2">
```

adding/removing elements

- move from a static web page to a **dynamic web page**
- add elements and text with JavaScript

creating new nodes

- two methods:
  - `createElement()` creates a new element node
  - `createTextNode()` creates a new text node
- use `textContent` to add/modify the text of the created nodes

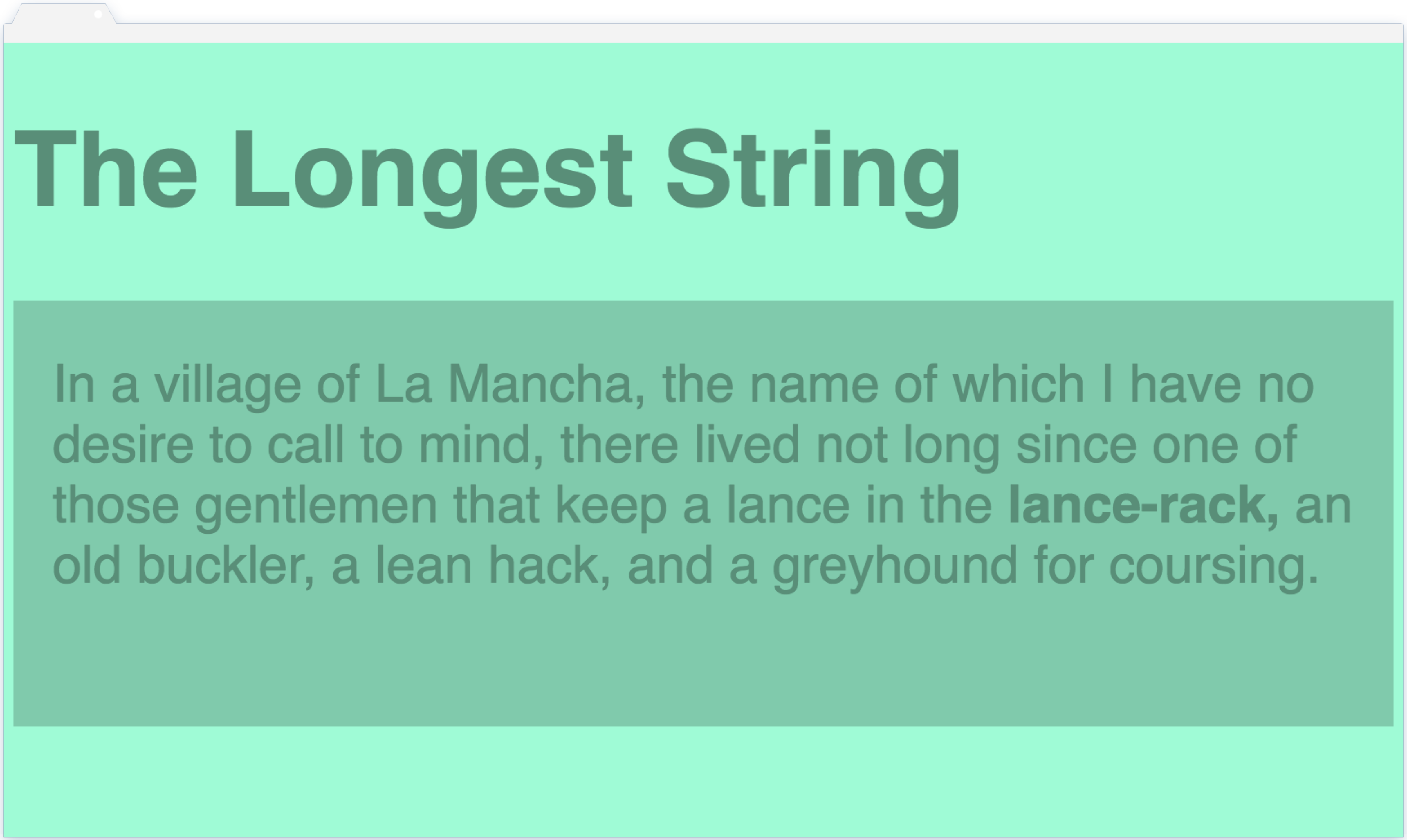
inserting nodes

- three methods:
  - `appendChild()` adds a node as the last child of the parent node
  - `insertBefore()` insert a node into the parent element before the given sibling node
  - `replaceChild()` replace an existing node with a new node

removing nodes

- two methods:
  - `removeChild()` removes the given child node
  - `remove()` removes the node

exercise



The Longest String

In a village of La Mancha, the name of which I have no desire to call to mind, there lived not long since one of those gentlemen that keep a lance in the **lance-rack,** an old buckler, a lean hack, and a greyhound for coursing.

`defer` 😏

- `<script>` element attribute (boolean)
- indicates to the browser that the script is meant to be executed **after the document has been parsed**

```
<script src="..." defer></script>
```

4. JavaScript events

events

- actions that take place in the browser
- initiated by the user, e.g. user clicks a button, or by the browser itself, e.g. page finishes loading
- **we can make web pages interactive by writing code that responds to events**

event handlers

- JavaScript function that is executed when an event fires
- assigned to elements in two ways:
    - attribute
    - event listener

event handler attribute

```html
<body>
  <button onclick="sayHi()">Click me!</button>
</body>
```

event listeners

- watch for events on the element they are attached to
- can have multiple handlers for the same event (in contrast to attribute handlers)
- can control events not necessarily attached to an element

adding listeners

```
const buttonElem = document.getElementById(‘submit-button');

buttonElem.addEventListener('dblclick', function(event) {
  console.log('Double clicked!');
});
```

adding listeners

```
const buttonElem = document.getElementById(‘submit-button');

buttonElem.addEventListener('dblclick', function(event) {
  console.log('Double clicked!');
});
```

adding window listeners

```
window.addEventListener('scroll', function(event) {
  console.log(`Page position now at ${event.pageY}`);
});
```

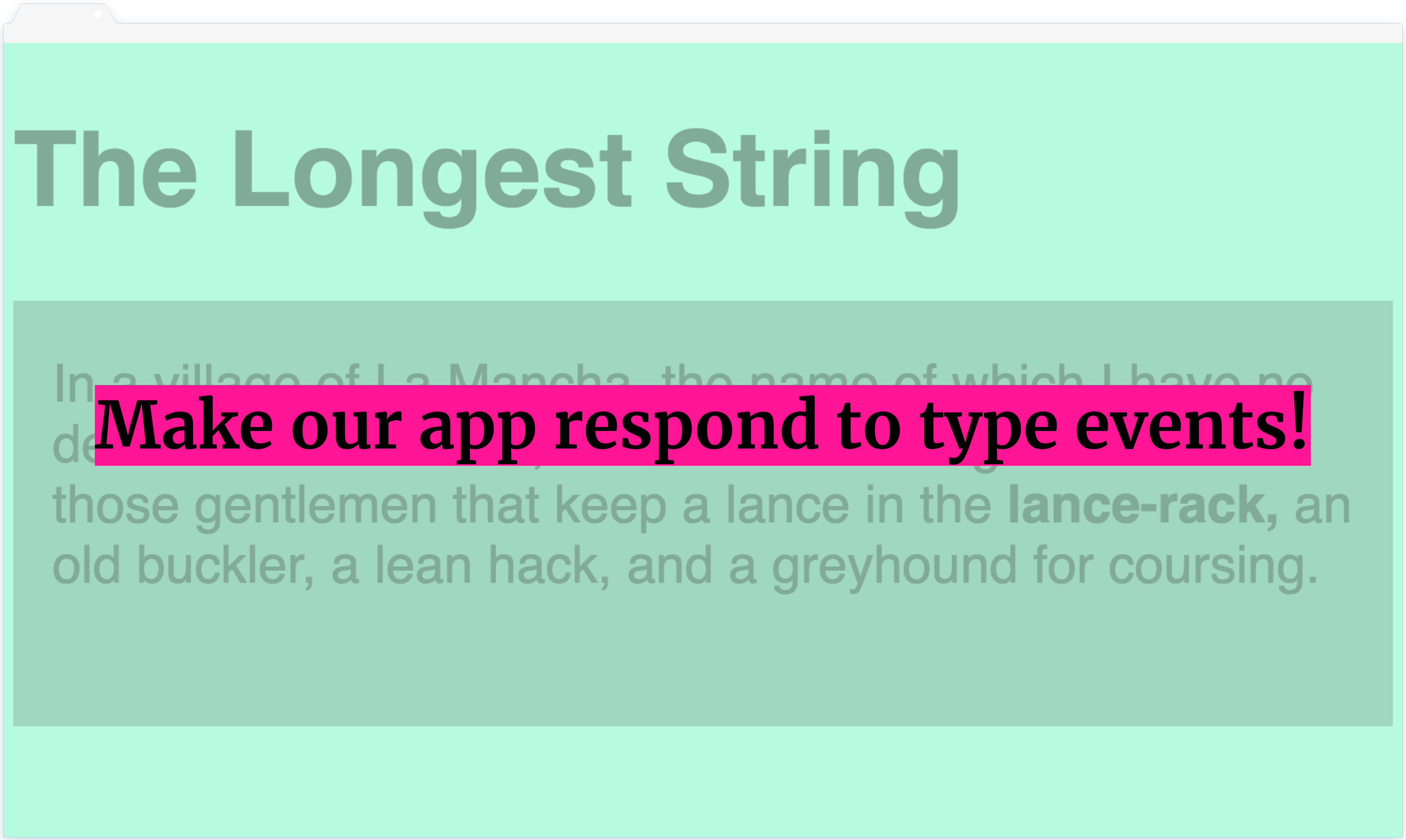
removing listeners

```
const buttonElem = document.getElementById('submit-button');

const eventHandler = function (event){
  console.log('Double clicked!');
};

buttonElem.removeEventListener('dblclick', eventHandler);
```

exercise



The Longest String

In a village of La Mancha, the name of which I have no de

those gentlemen that keep a lance in the **lance-rack,** an old buckler, a lean hack, and a greyhound for coursing.

Make our app respond to type events!

thanks!