DOM and REACT

ONE LOVE. ONE FUTURE.

Contents

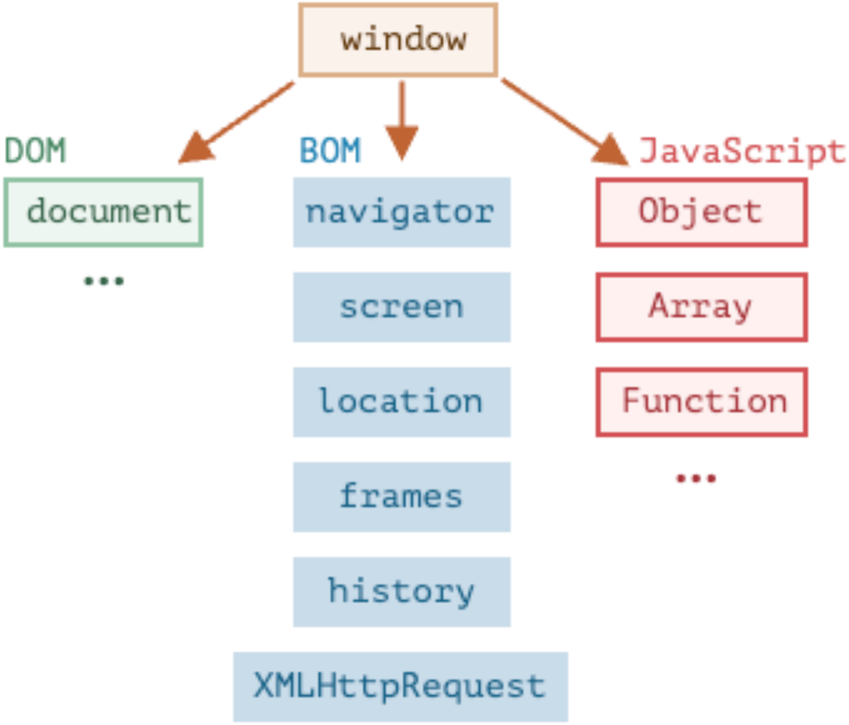
1. Document Object Model
   - Document Object Model (DOM)
   - Browser Object Model (BOM)

2. React
   - Single-Page Application (SPA)
   - React

Document Object Model (DOM)

Browser environment

- JavaScript language was initially created for web browser
- Since then, it has evolved into a language with many platforms
- A **platform** provides its own objects and functions in addition to the language core.
  - Web browsers give a means to control web pages
  - Node.js provides server-side features



What is DOM?

- A W3C (World Wide Web Consortium) standard.

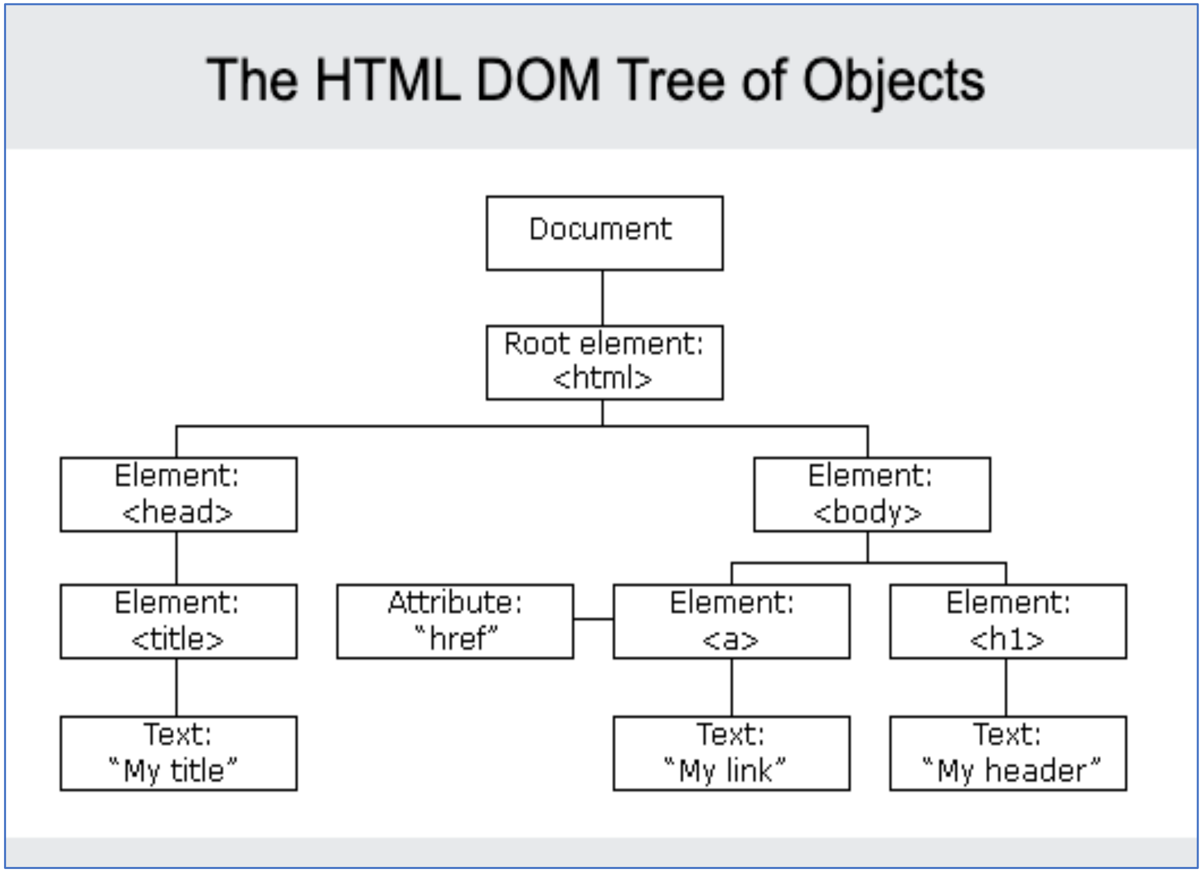
- A programming API for HTML and XML documents. It defines the logical structure of documents and the way a document is accessed and manipulated.

DOM (cont.)

- The W3C DOM standard is separated into 2 parts:
  - DOM Core - standard model for all document types (HTML, XML)
  - DOM HTML - standard model for HTML documents
- The DOM HTML defines:
  - The HTML elements as objects
  - The properties of all HTML elements
  - The methods to access all HTML elements
  - The events for all HTML elements
- In other words: The HTML DOM is a standard for how to get, change, add, or delete HTML elements.

DOM Tree

- When a web page is loaded, the browser creates a tree

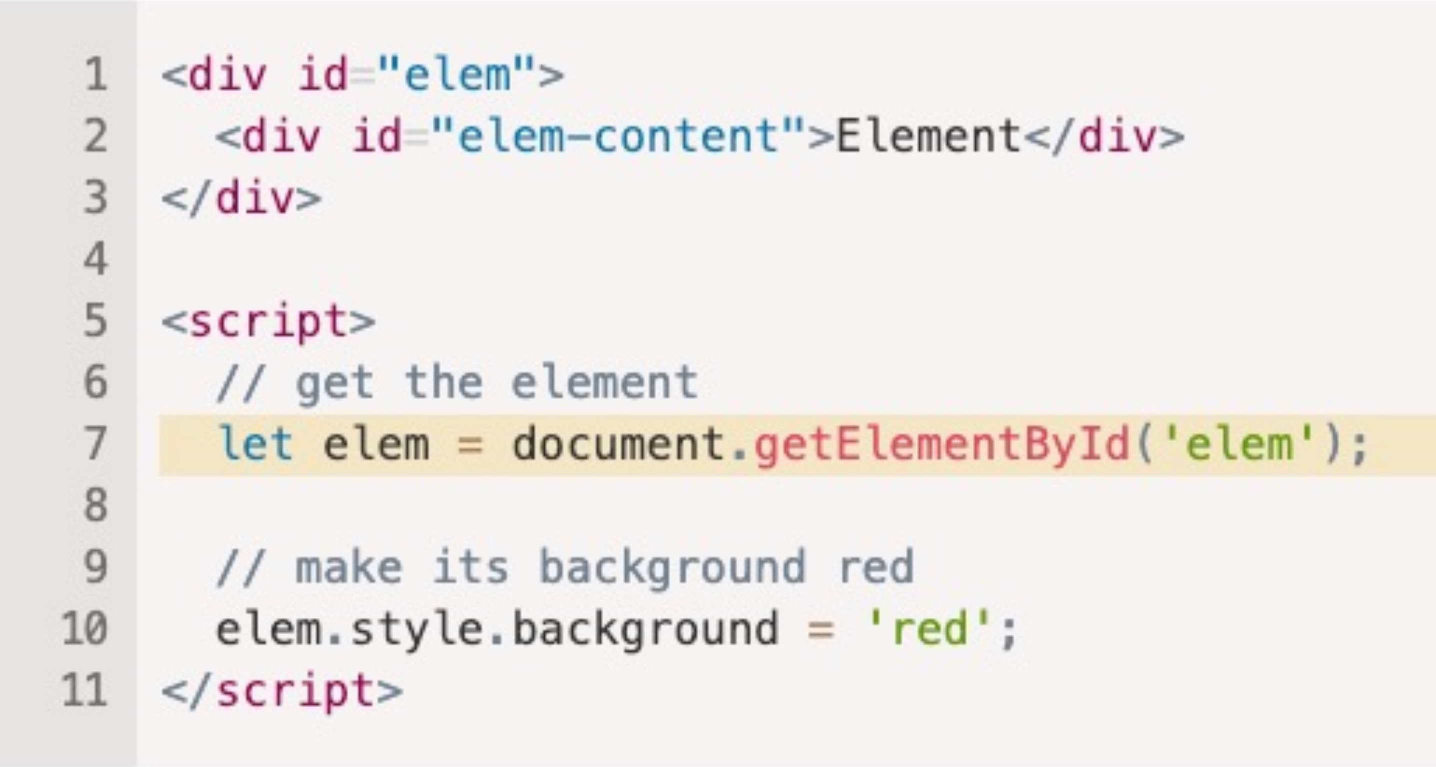


DOM

- With the object model, JavaScript gets all the power it needs to create dynamic HTML:
  - JavaScript can add, remove or change all the HTML **elements** in the page
  - JavaScript can add, remove or change all the HTML **attributes** in the page
  - JavaScript can change all the **CSS styles** in the page
  - JavaScript can create new HTML **events** or react to all existing HTML events in the page

Searching: document.getElementById(id)

- If an element has the id attribute, we can get the element using the method document.getElementById(id)



```
<div id="elem">
  <div id="elem-content">Element</div>
</div>

<script>
  // get the element
  let elem = document.getElementById('elem');

  // make its background red
  elem.style.background = 'red';
</script>
```

Searching: document.getElementById(id)

- The id must be unique:
  - There can be only one element in the document with the given id.
  - If there are multiple elements with the same id, then the behavior of methods that use it is unpredictable

- Only document.getElementById, not anyElem.getElementById
  - The method getElementById can be called only on document object. It looks for the given id in the whole document.

Searching: getElementsBy*

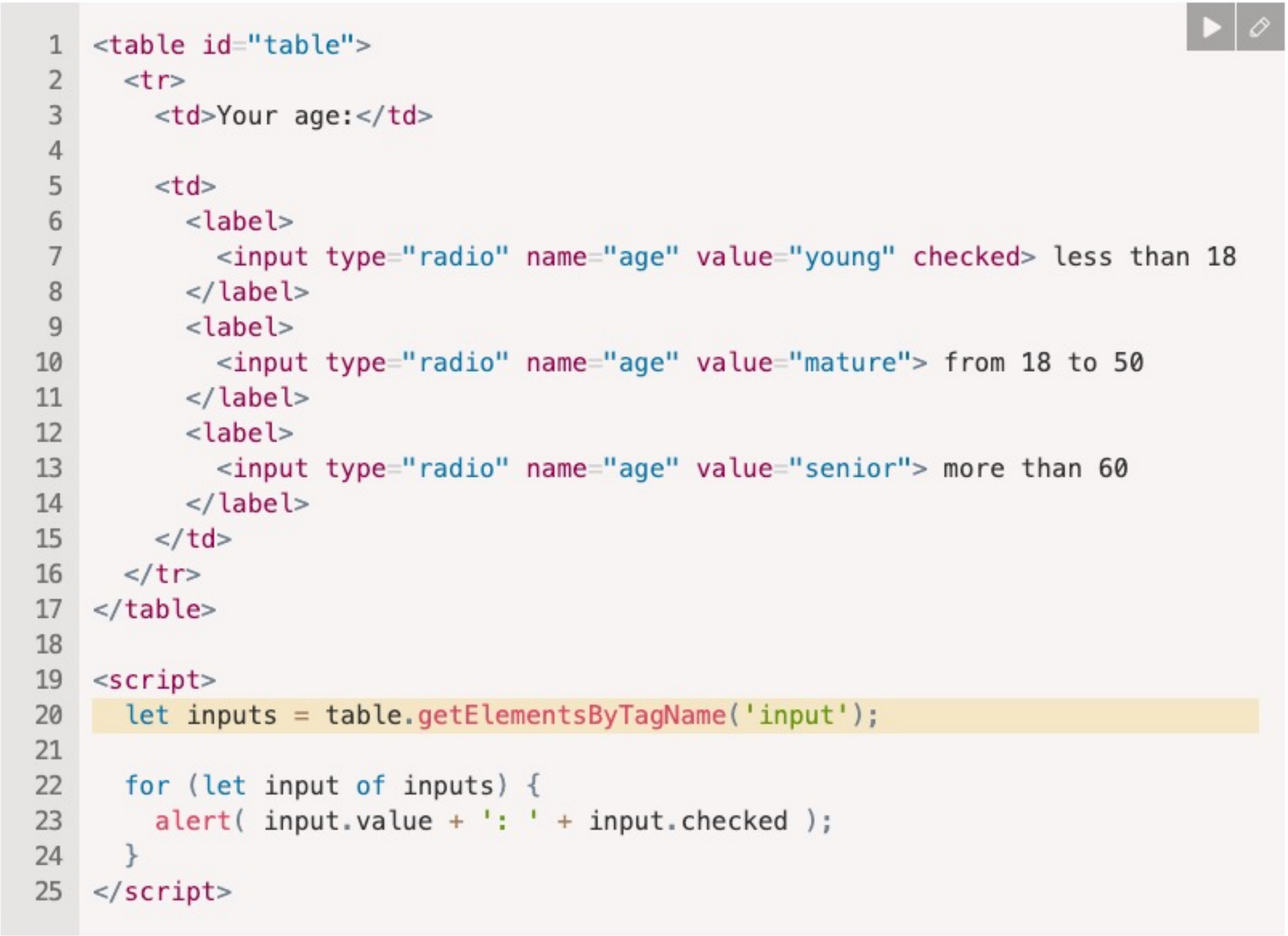
- There are also other methods to look for nodes
- Today, they are mostly history, as querySelector is more powerful and shorter to write.
  - elem.getElementsByTagName(tag)

    ```
    e.g., table.getElementsByTagName("td");
    ```
  - elem.getElementsByClassName(className)

    ```
    e.g., document.getElementsByClassName("example");
    ```
  - document.getElementsByName(name)

    ```
    e.g.,
    <input name="animal" type="checkbox" value="Cats">
    document.getElementsByName("animal");
    ```

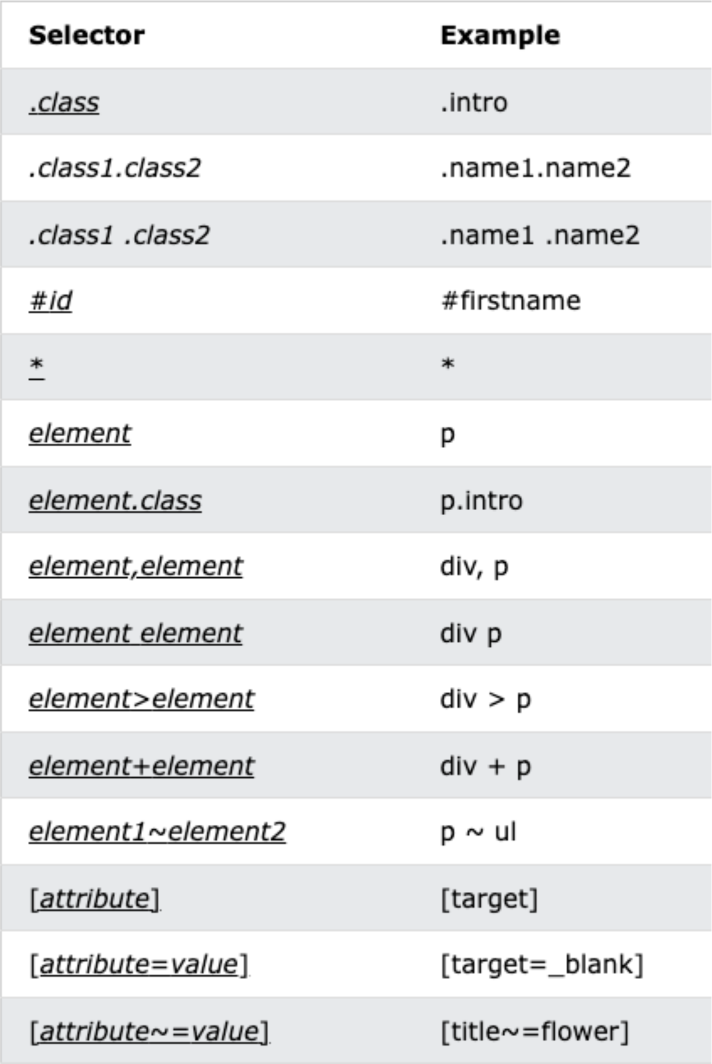
Searching: getElementsBy*



```
<table id="table">
  <tr>
    <td>Your age:</td>

    <td>
      <label>
        <input type="radio" name="age" value="young" checked> less than 18
      </label>
      <label>
        <input type="radio" name="age" value="mature"> from 18 to 50
      </label>
      <label>
        <input type="radio" name="age" value="senior"> more than 60
      </label>
    </td>
  </tr>
</table>

<script>
  let inputs = table.getElementsByTagName('input');

  for (let input of inputs) {
    alert( input.value + ': ' + input.checked );
  }
</script>
```

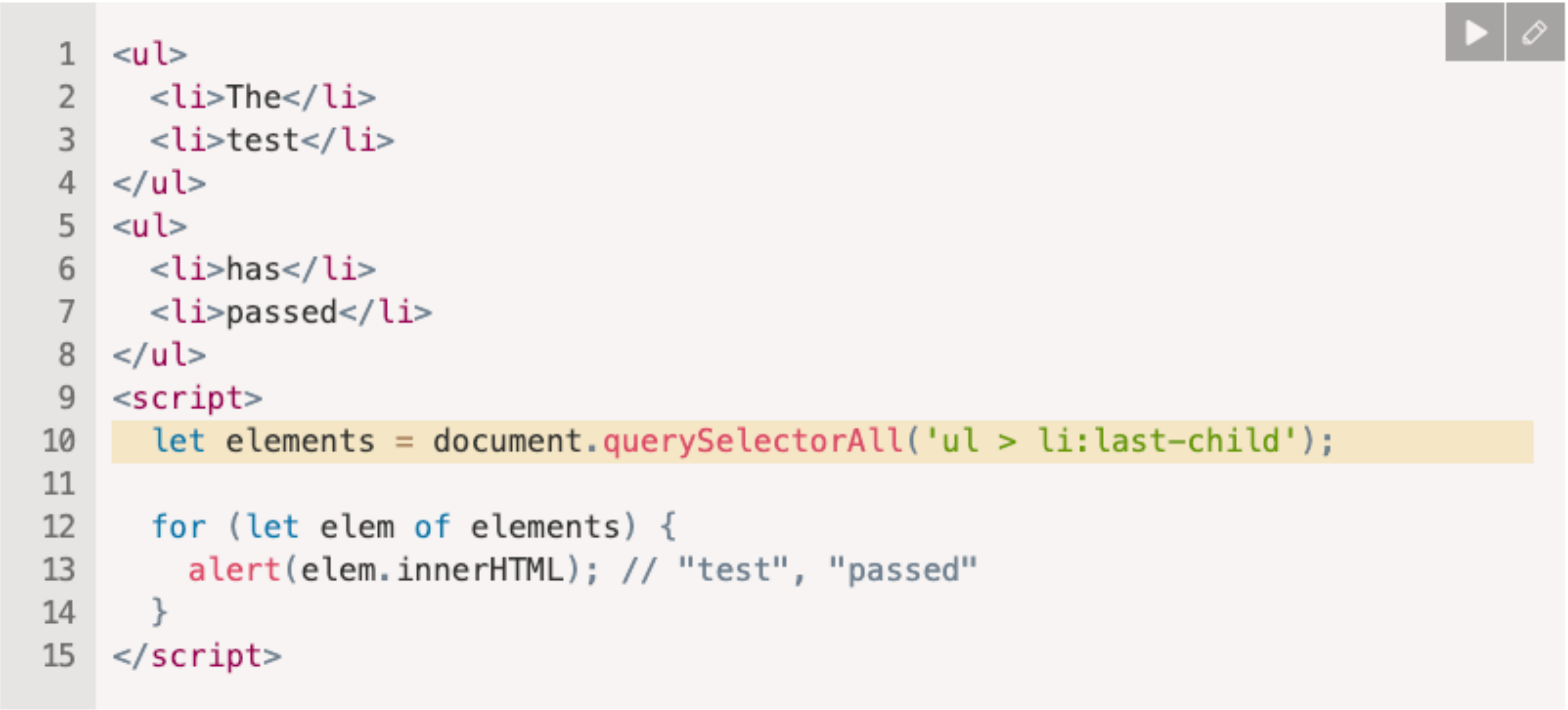
Searching: querySelectorAll

- The most versatile method: elem.querySelectorAll(css)
- Any CSS selector can be used



Selector	Example
.class	.intro
.class1.class2	.name1.name2
.class1 .class2	.name1 .name2
#id	#firstname
*	*
element	p
element.class	p.intro
element,element	div, p
element element	div p
element>element	div > p
element+element	div + p
element1~element2	p ~ ul
[attribute]	[target]
[attribute=value]	[target=_blank]
[attribute~=value]	[title~=flower]

Searching: querySelectorAll



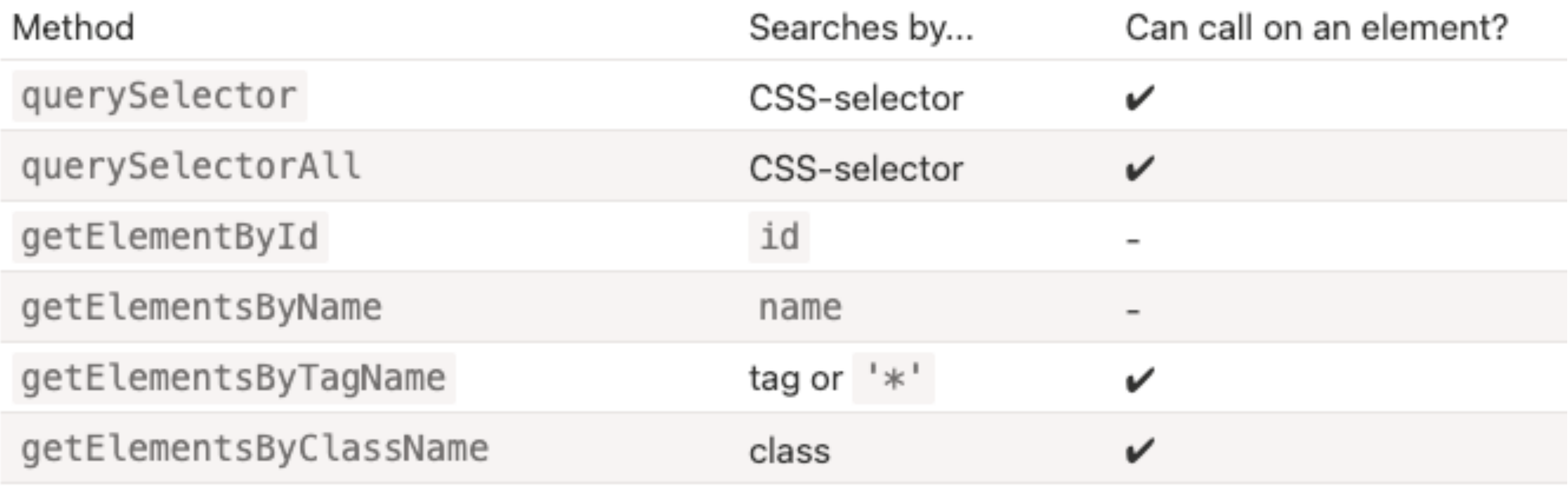
```
<ul>
  <li>The</li>
  <li>test</li>
</ul>
<ul>
  <li>has</li>
  <li>passed</li>
</ul>
<script>
  let elements = document.querySelectorAll('ul > li:last-child');

  for (let elem of elements) {
    alert(elem.innerHTML); // "test", "passed"
  }
</script>
```

Searching: querySelector

- The call to elem.querySelector(css) returns the first element for the given CSS selector

- The result is the same as elem.querySelectorAll(css)[0]
  - The latter is looking for all elements and picking one
  - elem.querySelector just looks for one. So it’s faster and also shorter to write.

Searching: Summary



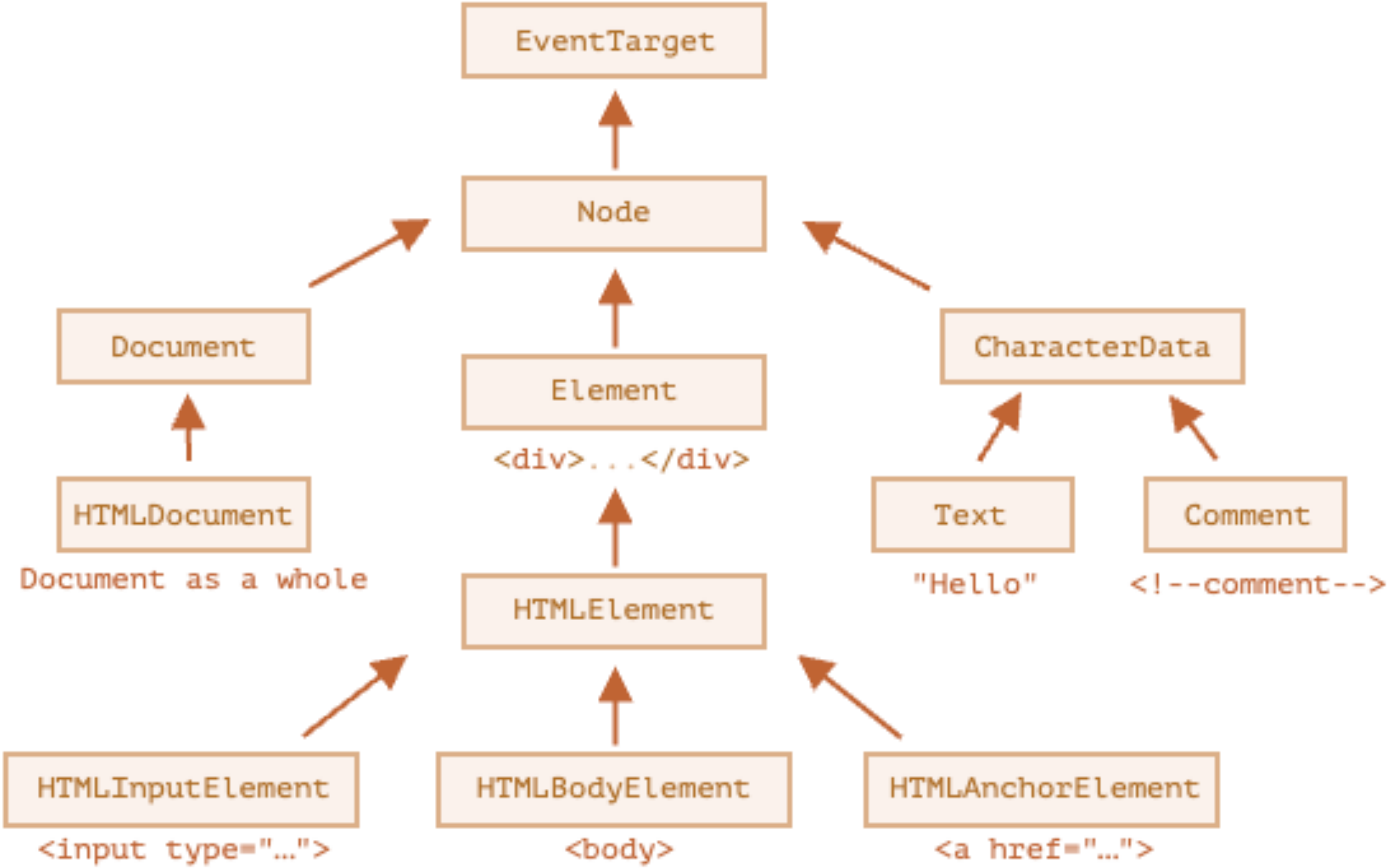
Method	Searches by...	Can call on an element?
`querySelector`	CSS-selector	✔
`querySelectorAll`	CSS-selector	✔
`getElementById`	`id`	-
`getElementsByName`	`name`	-
`getElementsByTagName`	tag or `'*'`	✔
`getElementsByClassName`	class	✔

Node properties

- Different DOM nodes may have different properties.
  - An element node corresponding to tag <a> has link-related properties
  - The one corresponding to <input> has input-related properties and so on.
  - Text nodes are not the same as element nodes.

- But there are also common properties and methods between all of them

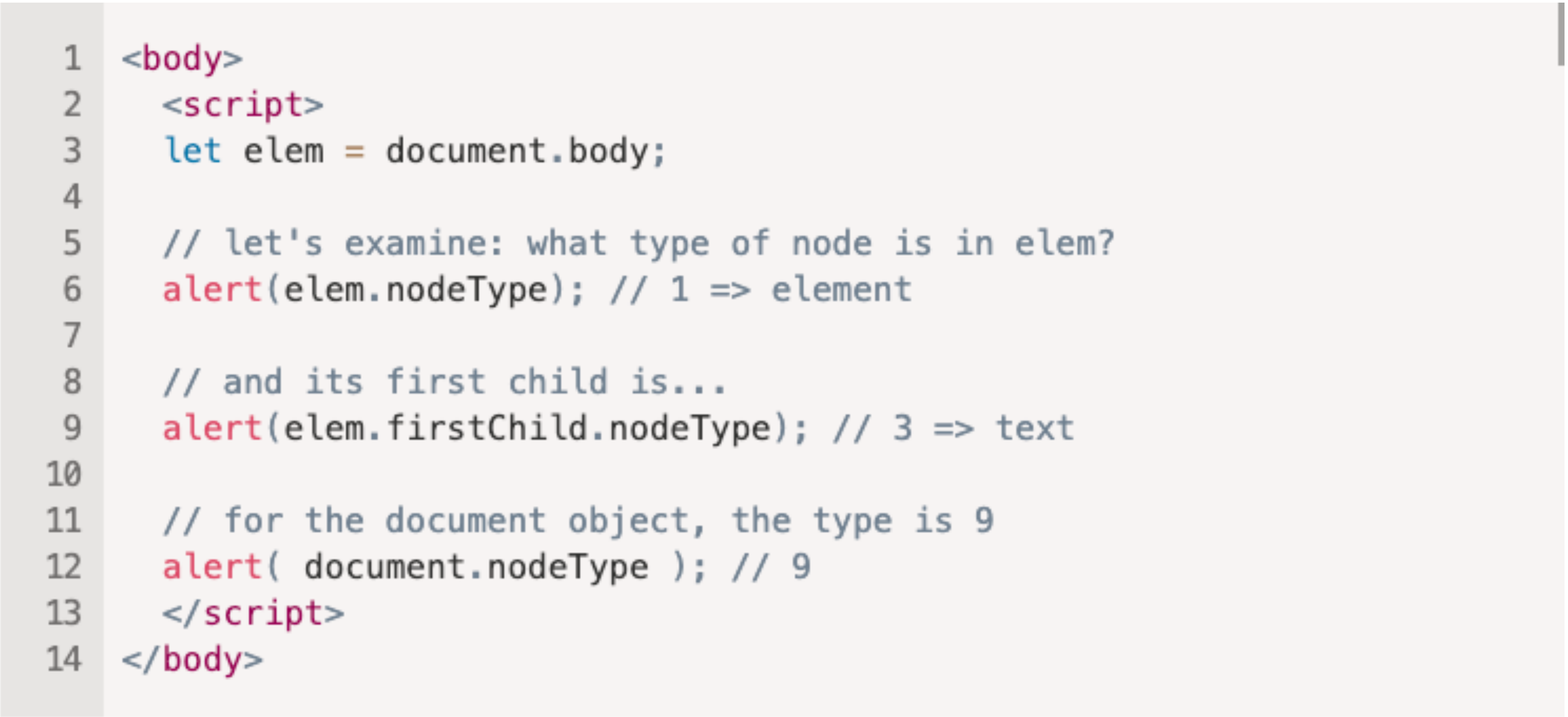
Node properties: hierarchy



The “nodeType” property

- The nodeType property provides one more, “old-fashioned” way to get the “type” of a DOM node.
- It has a numeric value:
    - elem.nodeType == 1 for element nodes,
    - elem.nodeType == 3 for text nodes,
    - elem.nodeType == 8 for comment nodes,
    - elem.nodeType == 9 for the document object

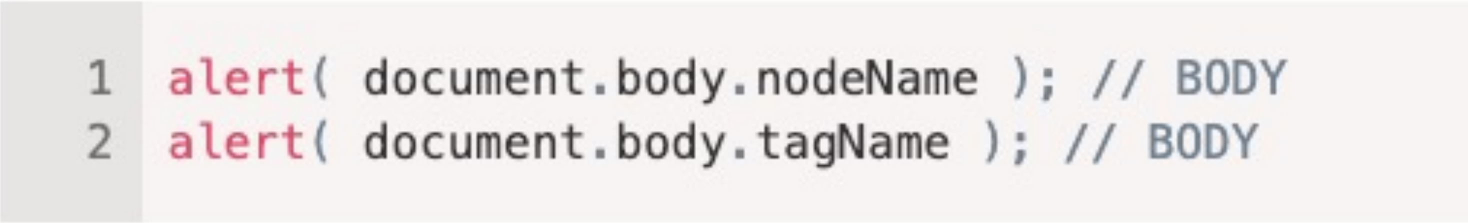
“nodeType” property example



```
<body>
  <script>
  let elem = document.body;

  // let's examine: what type of node is in elem?
  alert(elem.nodeType); // 1 => element

  // and its first child is...
  alert(elem.firstChild.nodeType); // 3 => text

  // for the document object, the type is 9
  alert( document.nodeType ); // 9
  </script>
</body>
```

Tag: nodeName and tagName

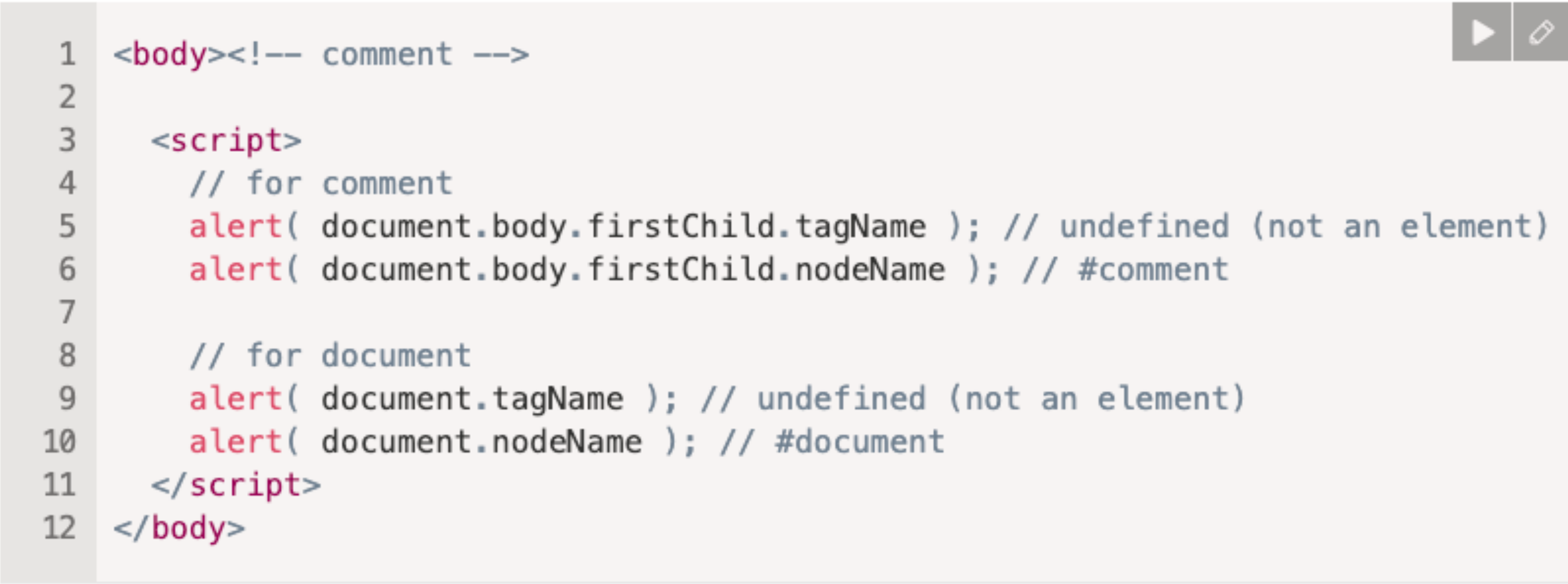
- Given a DOM node, we can read its tag name from nodeName or tagName properties:



```
alert( document.body.nodeName ); // BODY
alert( document.body.tagName ); // BODY
```

- Is there any difference between tagName and nodeName?
  - The tagName property exists only for Element nodes.
  - The nodeName is defined for any Node:
    - for elements it means the same as tagName.
    - for other node types (text, comment, etc.) it has a string with the node type.

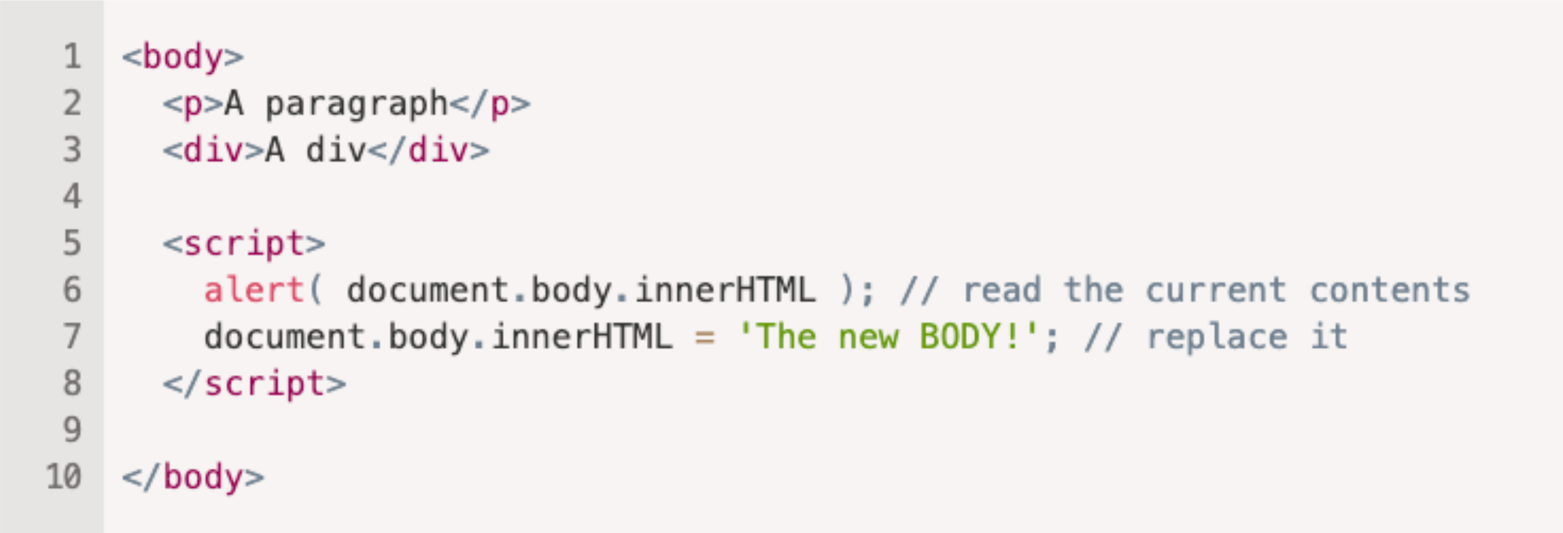
Tag: nodeName and tagName



```
<body><!-- comment -->

  <script>
    // for comment
    alert( document.body.firstChild.tagName ); // undefined (not an element)
    alert( document.body.firstChild.nodeName ); // #comment

    // for document
    alert( document.tagName ); // undefined (not an element)
    alert( document.nodeName ); // #document
  </script>
</body>
```

innerHTML: the contents

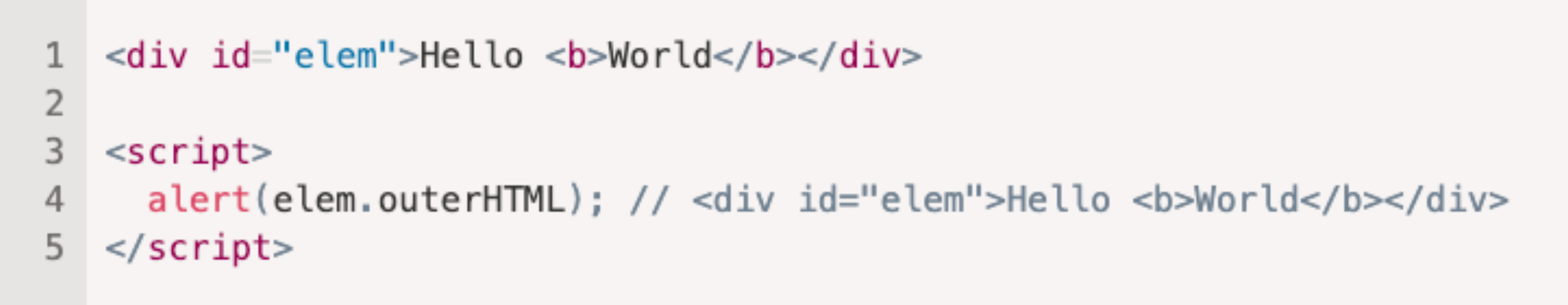
- The innerHTML property allows to get the HTML inside the element as a string.
- We can also modify it. So it’s one of the most powerful ways to change the page.
- The example shows the contents of document.body and then replaces it completely:



```
<body>
  <p>A paragraph</p>
  <div>A div</div>

  <script>
    alert( document.body.innerHTML ); // read the current contents
    document.body.innerHTML = 'The new BODY!'; // replace it
  </script>

</body>
```

outerHTML: full HTML of the element

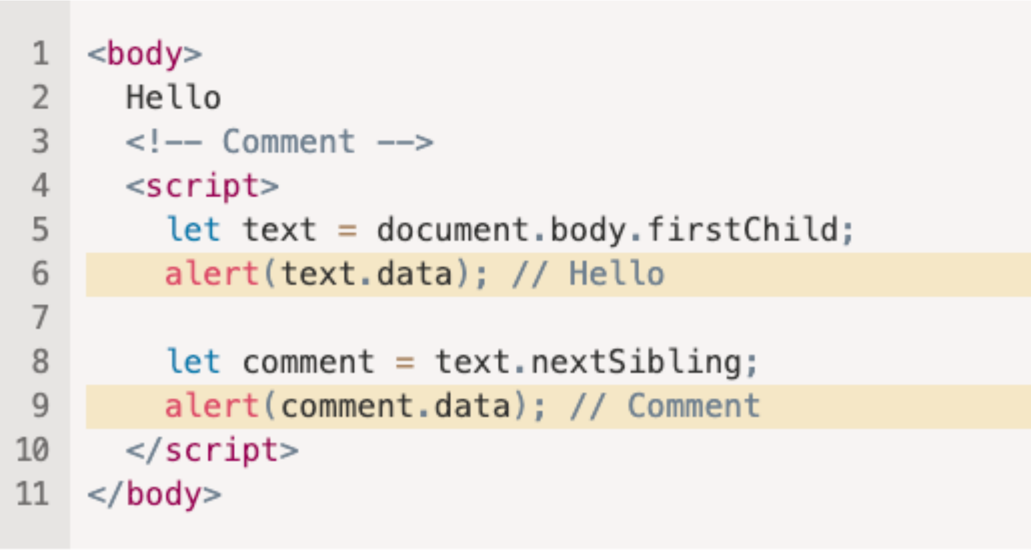
- The outerHTML property contains the full HTML of the element. That’s like innerHTML plus the element itself.



```
<div id="elem">Hello <b>World</b></div>

<script>
  alert(elem.outerHTML); // <div id="elem">Hello <b>World</b></div>
</script>
```

nodeValue/data: text node content

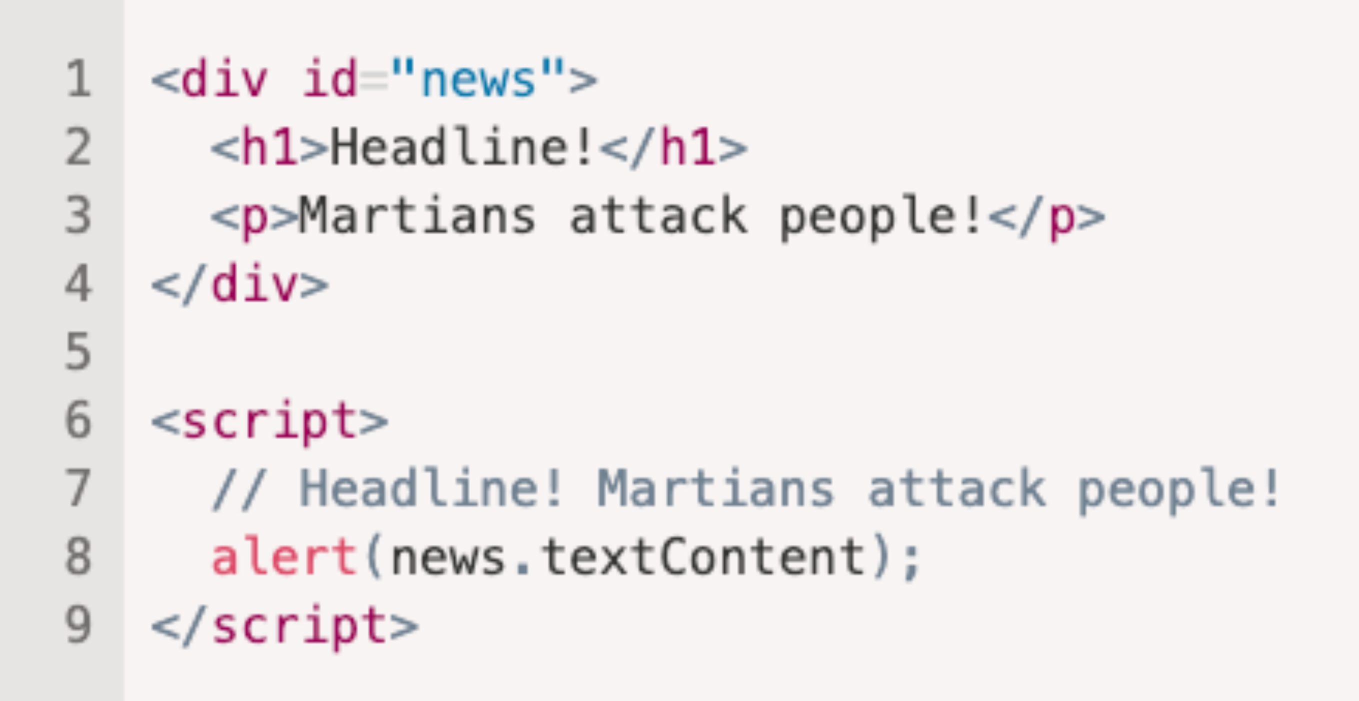
- The innerHTML property is only valid for element nodes.
- Other node types, such as text nodes, have their counterpart: nodeValue and data properties.
- These two are almost the same for practical use.



```
<body>
  Hello
  <!-- Comment -->
  <script>
    let text = document.body.firstChild;
    alert(text.data); // Hello

    let comment = text.nextSibling;
    alert(comment.data); // Comment
  </script>
</body>
```

textContent: pure text

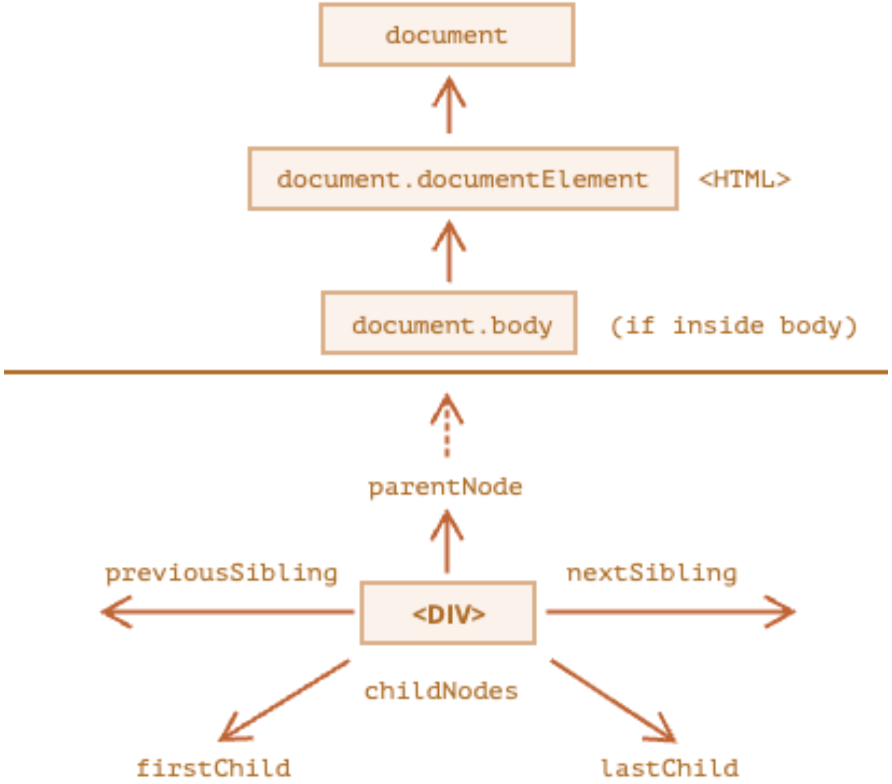
- The textContent provides access to the text inside the element: only text, minus all <tags>.



```
<div id="news">
  <h1>Headline!</h1>
  <p>Martians attack people!</p>
</div>

<script>
  // Headline! Martians attack people!
  alert(news.textContent);
</script>
```

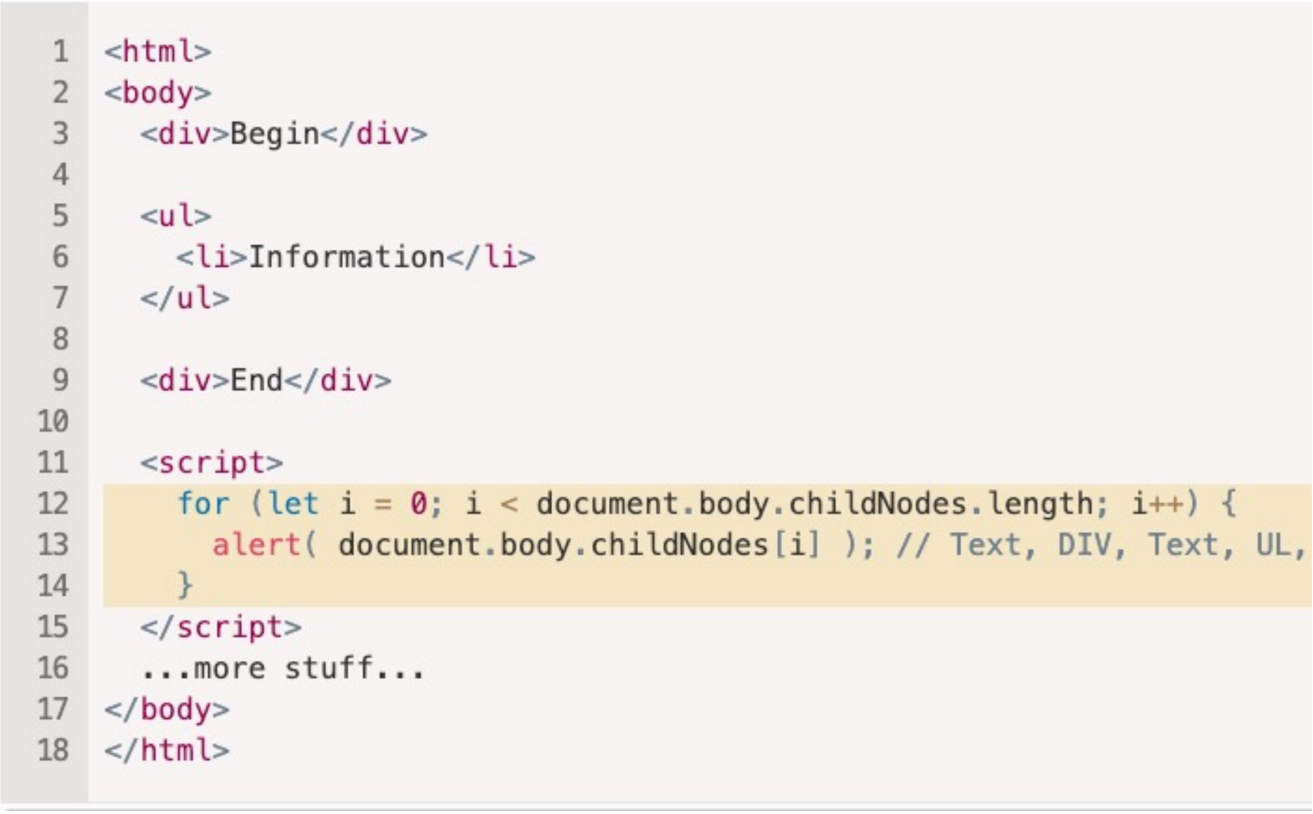
Walking the DOM

- **Child nodes** (or children): elements that are direct children
- **Descendants** – all elements that are nested in the given one, including children, their children and so on.



Children: childNodes

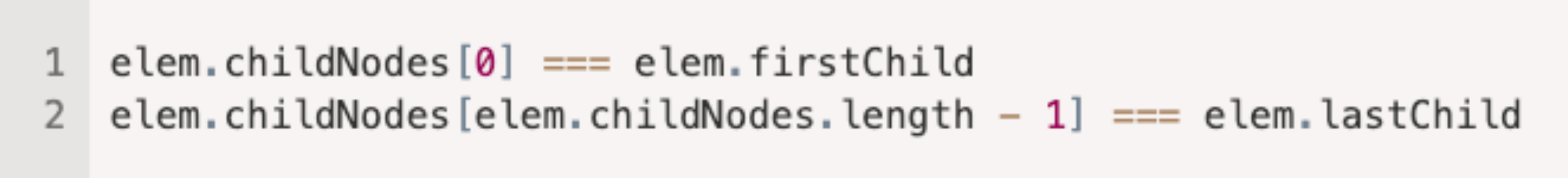
- The childNodes collection lists all child nodes, including text nodes.



```
<html>
<body>
  <div>Begin</div>

  <ul>
    <li>Information</li>
  </ul>

  <div>End</div>

  <script>
    for (let i = 0; i < document.body.childNodes.length; i++) {
      alert( document.body.childNodes[i] ); // Text, DIV, Text, UL,
    }
  </script>
  ...more stuff...
</body>
</html>
```

Children: firstChild, lastChild

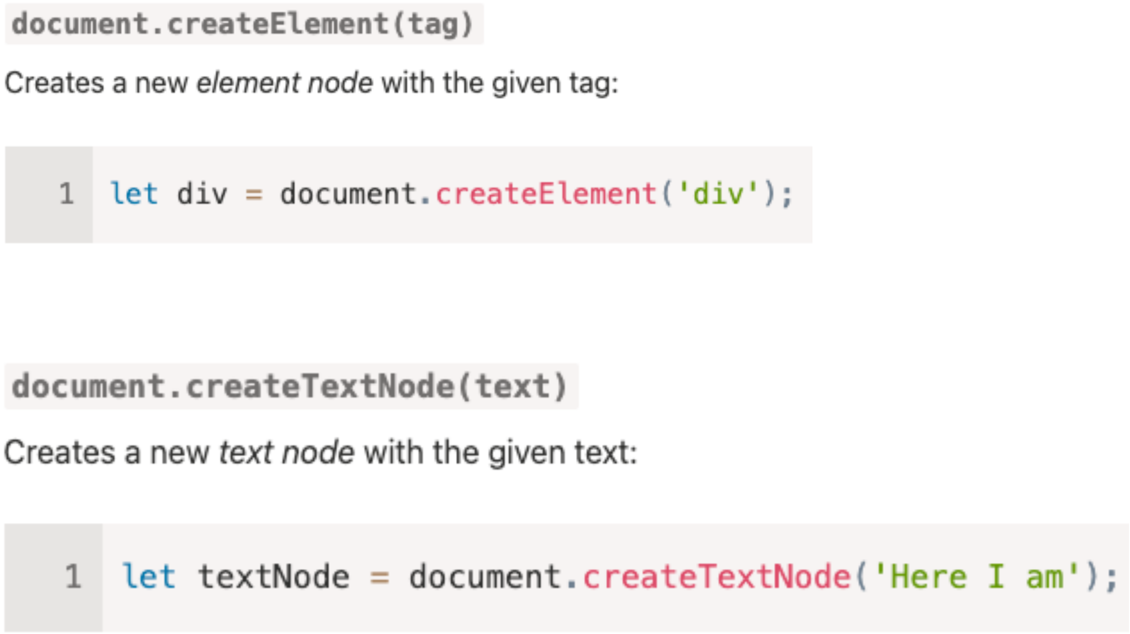
- Properties firstChild and lastChild give fast access to the first and last children
- If there exist child nodes, then the following is always true:



```
elem.childNodes[0] === elem.firstChild
elem.childNodes[elem.childNodes.length - 1] === elem.lastChild
```

Modifying the document

- DOM modification is the key to creating “live” pages.
- Here we’ll see how to create new elements “on the fly” and modify the existing page content.
- To create DOM nodes, there are two methods:



`document.createElement(tag)`

Creates a new *element node* with the given tag:

```
let div = document.createElement('div');
```

`document.createTextNode(text)`

Creates a new *text node* with the given text:

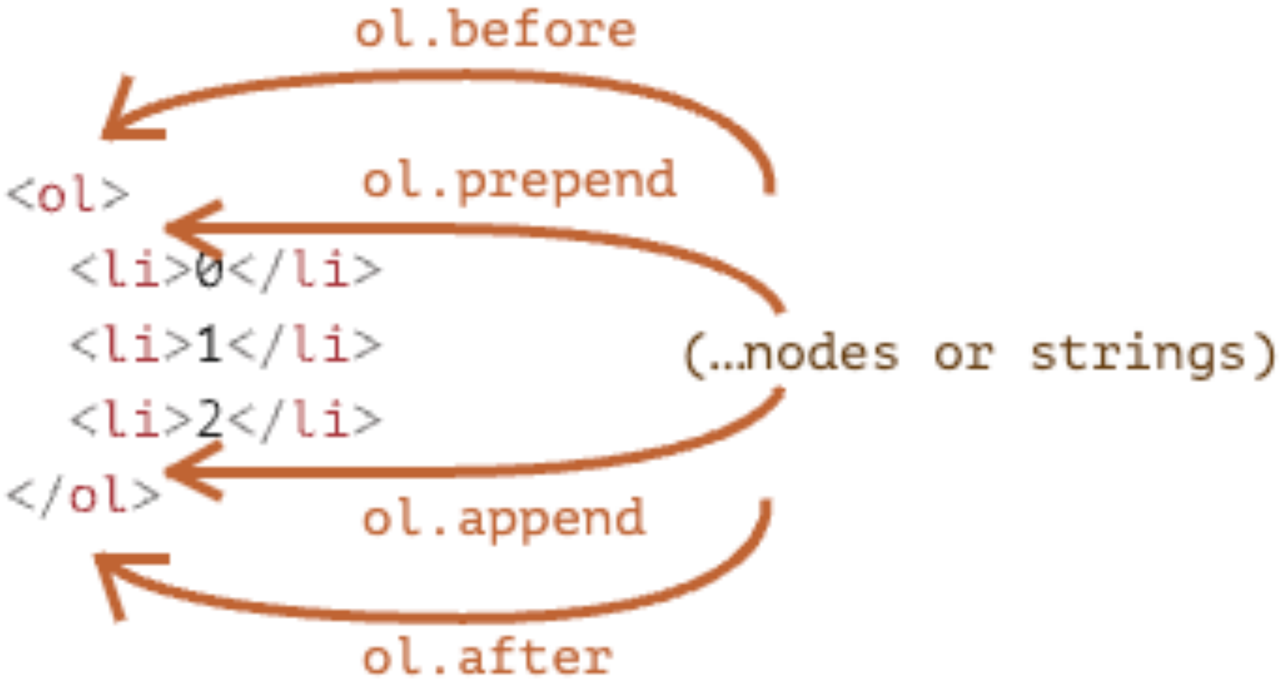
```
let textNode = document.createTextNode('Here I am');
```

Insertion methods

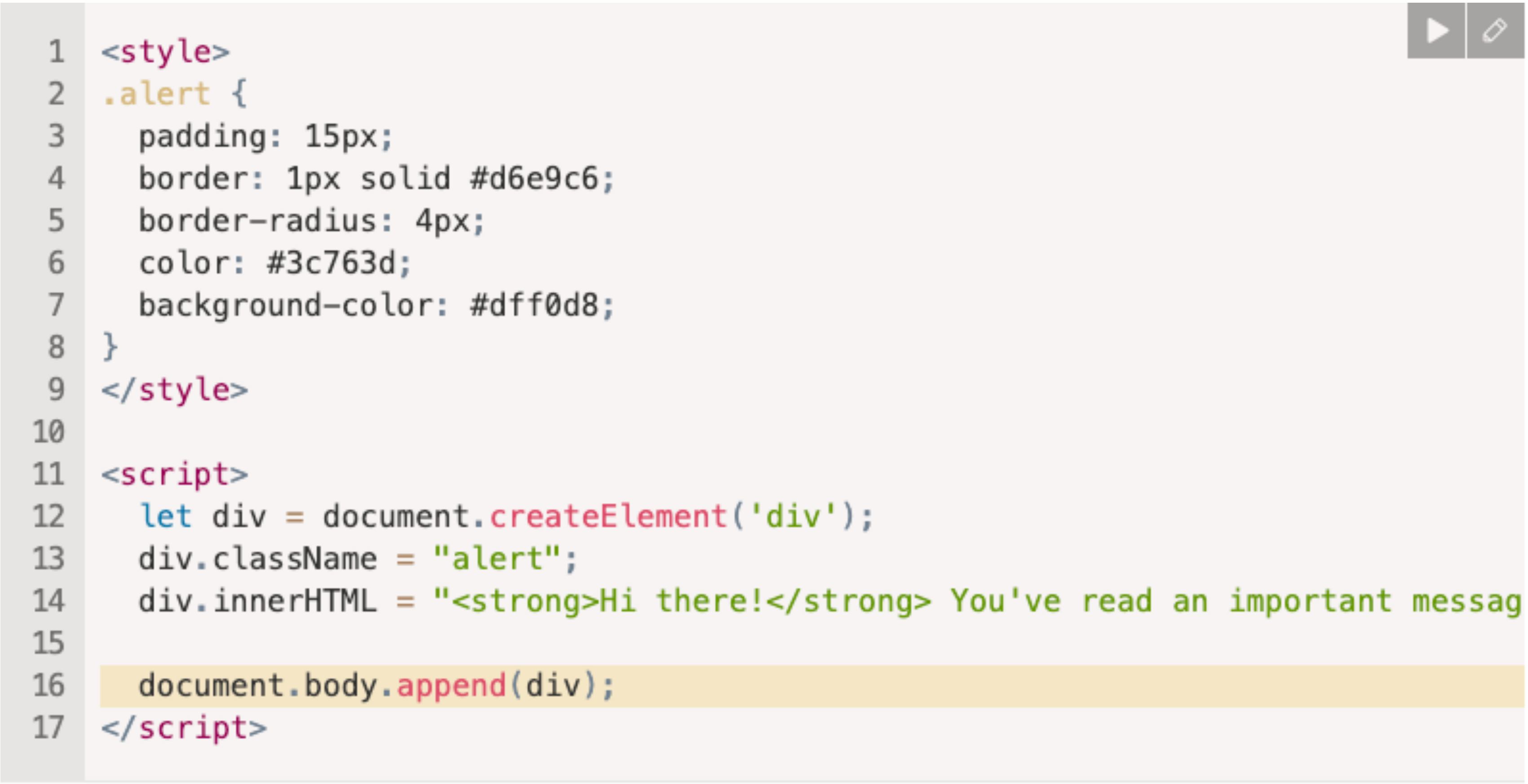
- Here are more insertion methods, they specify different places where to insert:
  - **node.append**(...nodes or strings) – append nodes or strings *at the end* of node,
  - **node.prepend**(...nodes or strings) – insert nodes or strings *at the beginning* of node,
  - **node.before**(...nodes or strings) – insert nodes or strings *before* node,
  - **node.after**(...nodes or strings) – insert nodes or strings *after* node,
  - **node.replaceWith**(...nodes or strings) – replace node with the given nodes or strings.

Insertion methods

- Here’s a visual picture of what the methods do:



Modifying the document: Example



```
<style>
.alert {
  padding: 15px;
  border: 1px solid #d6e9c6;
  border-radius: 4px;
  color: #3c763d;
  background-color: #dff0d8;
}
</style>

<script>
  let div = document.createElement('div');
  div.className = "alert";
  div.innerHTML = "<strong>Hi there!</strong> You've read an important messag

  document.body.append(div);
</script>
```

Exercise

- Run example exercises at: https://www.w3schools.com/js/js_htmldom.asp

Browser Object Model (BOM)

Browser Object Model (BOM)

- 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.

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 Size

- Two properties can be used to determine the size of the browser window, both properties return the sizes in pixels:
  - window.innerHeight - the inner height of the browser window (in pixels)
  - window.innerWidth - the inner width of the browser window (in pixels)
- Other Window Methods:
  - window.open() - open a new window
  - window.close() - close the current window
  - window.moveTo() - move the current window
  - window.resizeTo() - resize the current window

Window Screen

- The window.screen object contains information about the user's screen.
  - The window.screen object can be written without the window prefix.
- Properties:
  - screen.width
  - screen.height
  - screen.availWidth
  - screen.availHeight
  - screen.colorDepth
  - screen.pixelDepth

Window Location

- The window.location object can be used to get the current page address (URL) and to redirect the browser to a new page.
- https://www.w3schools.com/js/tryit.asp?filename=tryjs_loc_href
  - window.location.href returns the URL of the current page
  - window.location.hostname returns the domain name
    - `www.w3schools.com`
  - window.location.pathname returns the path of the current page:
    - `/js/tryit.asp`
  - window.location.protocol returns the web protocol used
    - `https`

Window History, Window Navigator

- Window Navigator:
  - navigator.appVersion
  - navigator.userAgent
  - navigator.platform
  - language
  - onLine
  - javaEnabled()
  - …

Window History:

- history.back() - same as clicking back in the browser
- history.forward() - same as clicking forward in the browser

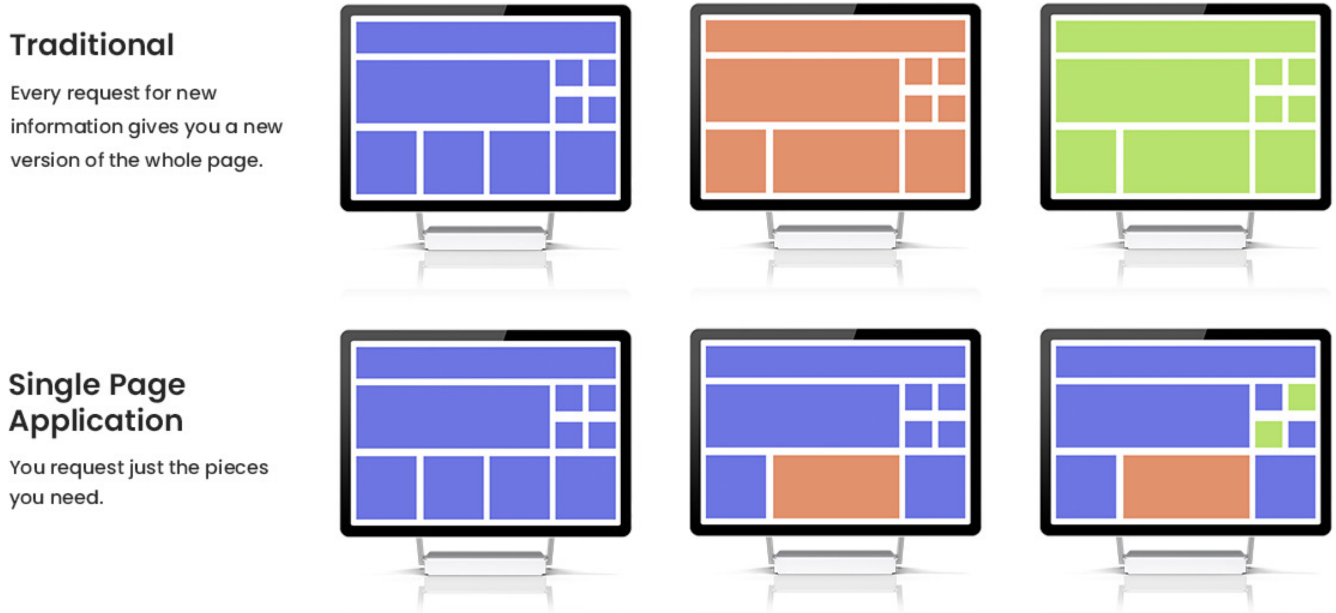
Exercises

- Run example exercises at https://www.w3schools.com/js/js_window.asp

Single-Page Application (SPA)

Multi-Page Application vs Single-Page Application

- Traditional web applications perform most of the application logic on the server
- Single-page applications (SPAs) perform most of the user interface logic in a web browser, communicating with the web server primarily using web API



Traditional

Every request for new information gives you a new version of the whole page.

Single Page Application

You request just the pieces you need.

What is a Single-Page Application?

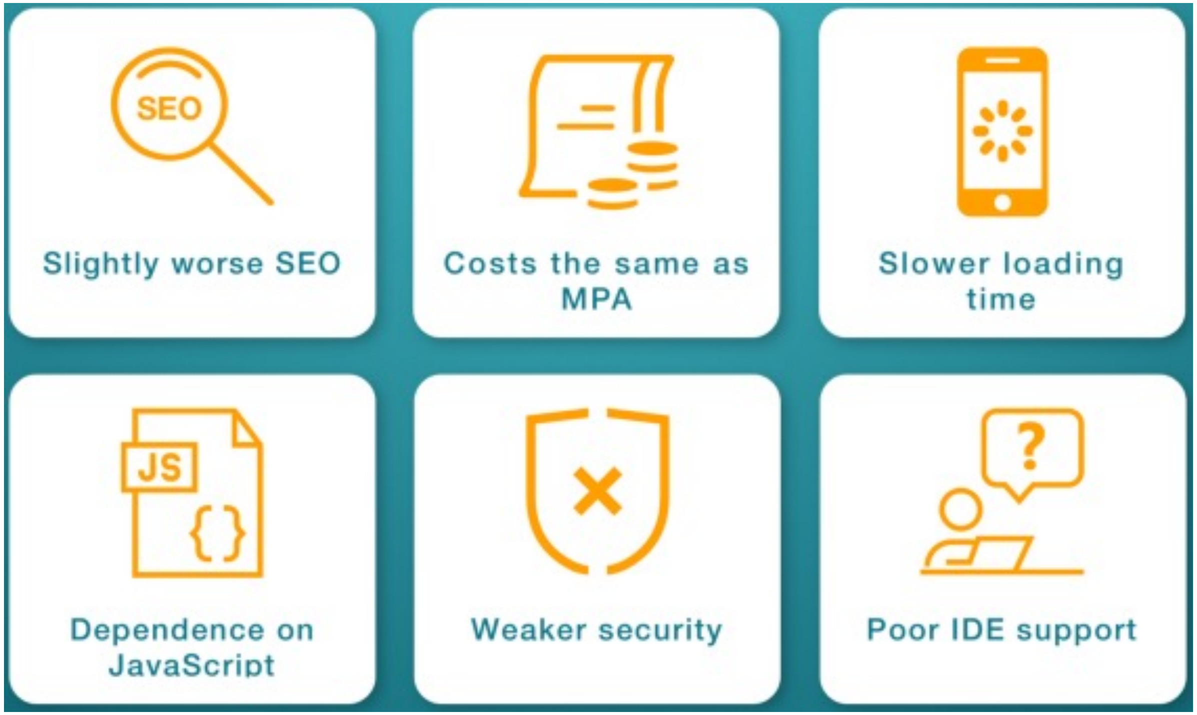
- A multiple-page app
  - Multiple pages with static information (text, image, etc.)
  - Every change requests rendering a new page from the server => browser reloads the content of a page completely and downloads the resources again

- A single-page app
  - Loads only a single page, and then updates the body content of that single document via JavaScript APIs
  - Examples: Gmail, Google Maps, Tiki, etc.

SPA Advantages

- Faster performance: all the resources are loaded during one session, only the necessary data is changed
- Data caching: provides ability to work offline
- Improved user experience: the use of AJAX and JavaScript frameworks (like React, Angular) allows building a more flexiable and responsive interface

SPA Disadvantages



- Slightly worse SEO
- Costs the same as MPA
- Slower loading time
- Dependence on JavaScript
- Weaker security
- Poor IDE support

MPA Advantages

- Faster initial page load.
- MPAs are easy and good for SEO management.
- MPAs provide lots of analytics and data about how a website works.

MPA Disadvantages

- Slow performance: application reloads every time a user clicks on a new tab
- Hard to maintain: developers need to maintain each page separately and regularly
- More development time: application has a higher number of features compared to a SPA, so it requires more effort and resources

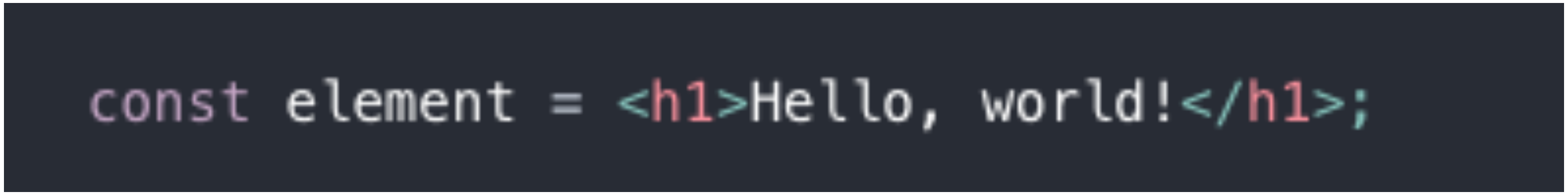
REACT

React

- React (also known as React.js or ReactJS) is a free and open-source front-end JavaScript library for building user interfaces based on UI components.
- It is maintained by Meta (formerly Facebook) and a community of individual developers and companies.
- React can be used as a base in the development of single-page, mobile, or server-rendered applications with frameworks like Next.js.

JSX

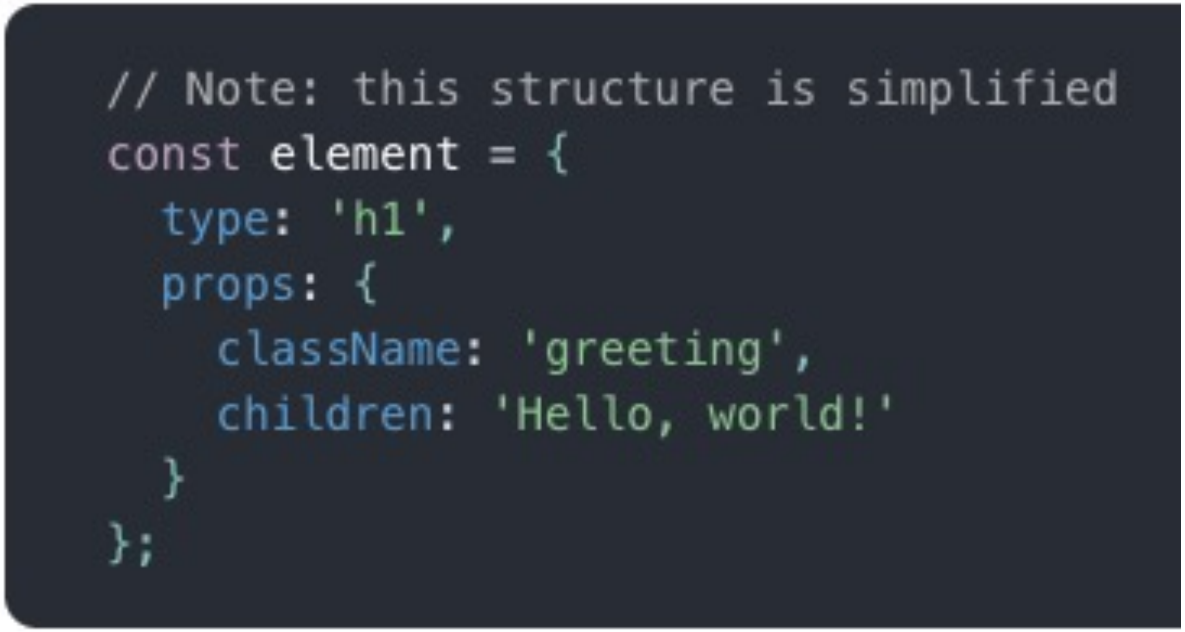
- JSX is a syntax extension to JavaScript.
- Similar in appearance to HTML, JSX provides a way to structure component rendering.



```
const element = <h1>Hello, world!</h1>;
```

JSX Represents Objects

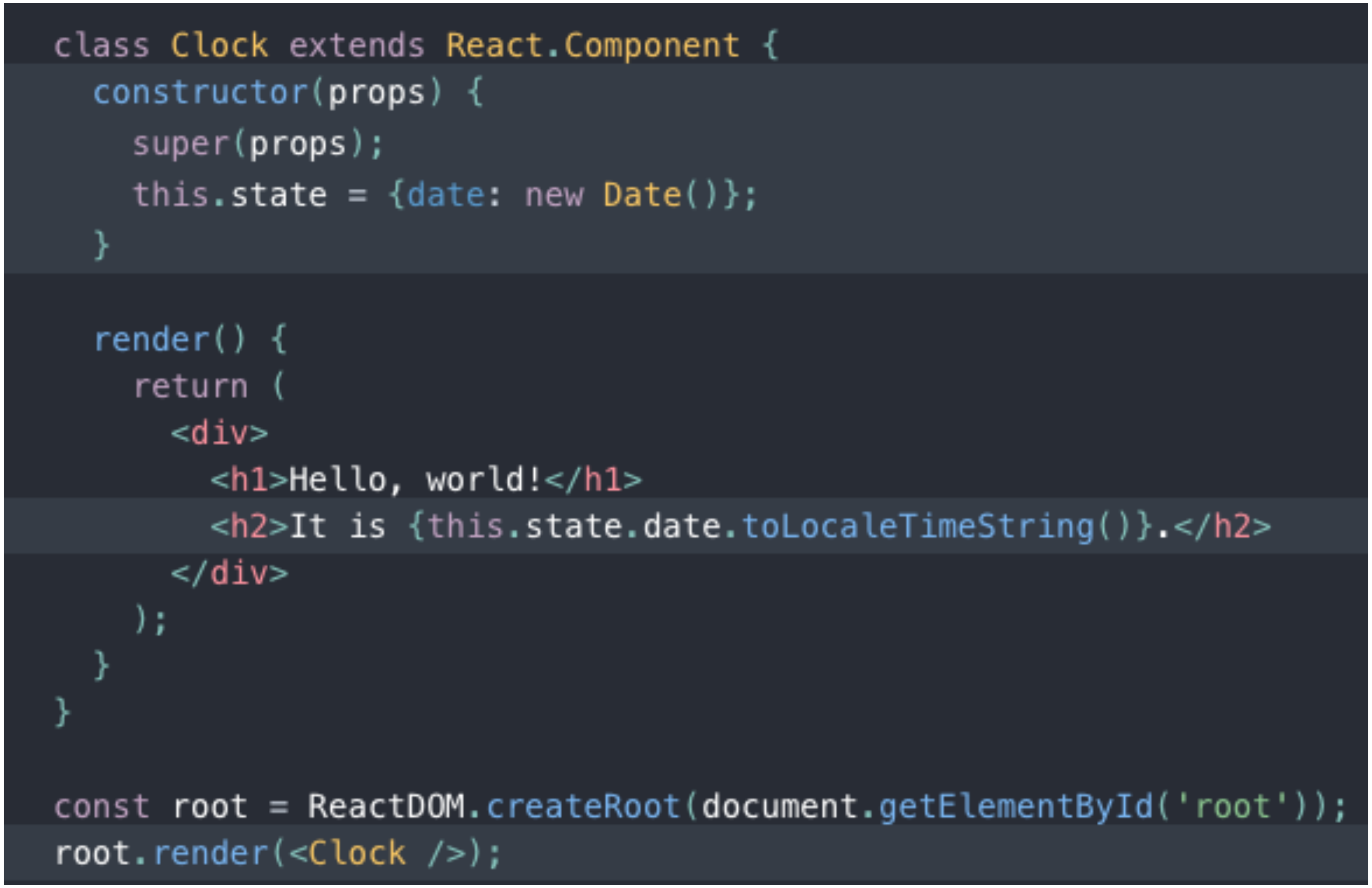
- React reads these objects and uses them to construct the DOM and keep it up to date.



```
// Note: this structure is simplified
const element = {
  type: 'h1',
  props: {
    className: 'greeting',
    children: 'Hello, world!'
  }
};
```

State and Lifecycle

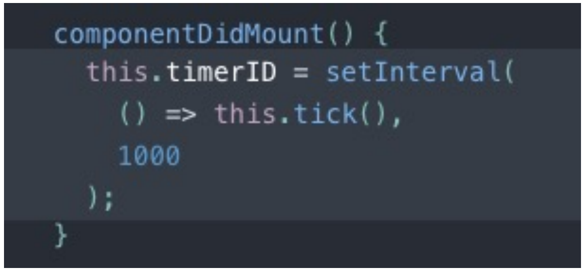
- Clock (class component) with this.state.date



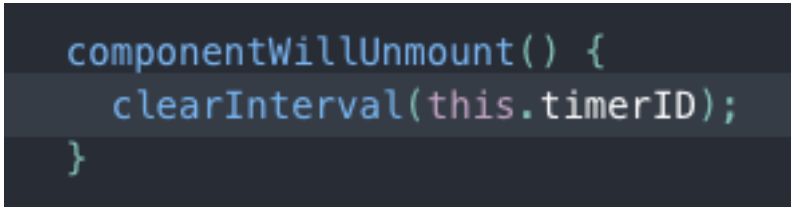
```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }

  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Clock />);
```

State and Lifecycle

- Adding Lifecycle Methods to a Class
  - We want to set up a timer whenever the Clock is rendered to the DOM for the first time. This is called “mounting” in React.
  - We also want to clear that timer whenever the DOM produced by the Clock is removed. This is called “unmounting” in React.
- We can declare special methods on the component class to run some code when a component mounts and unmounts:



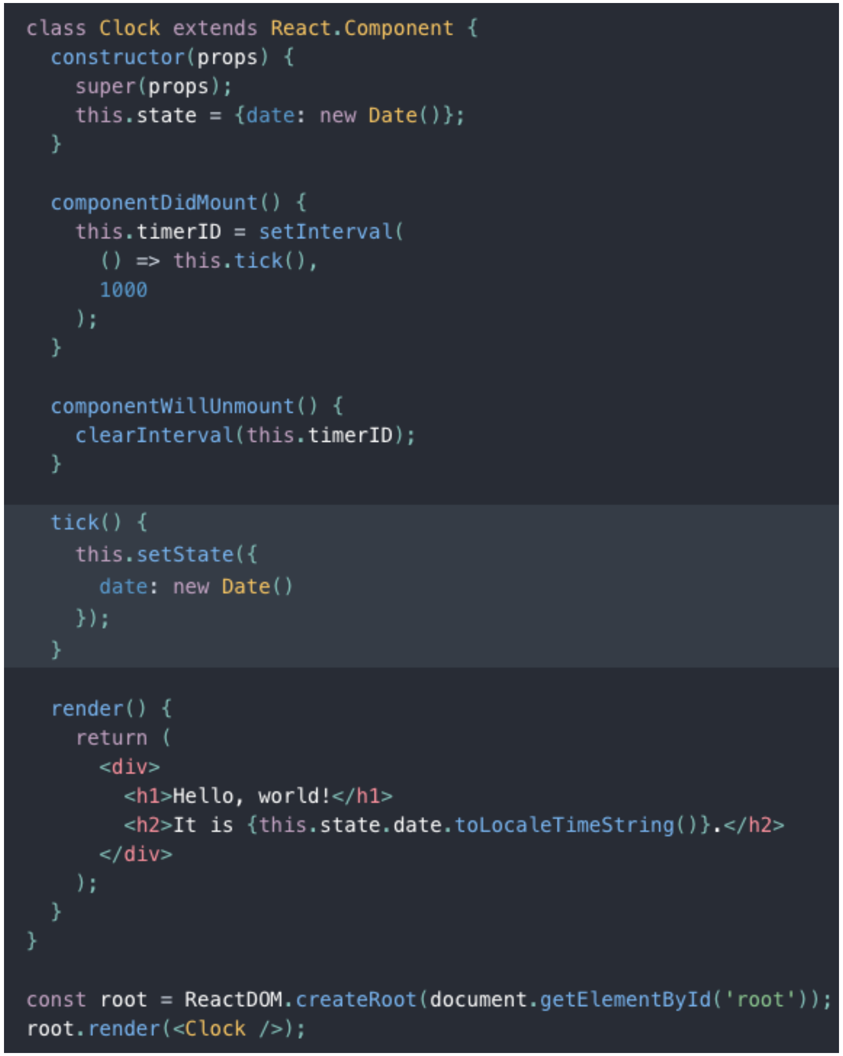
```
componentDidMount() {
  this.timerID = setInterval(
    () => this.tick(),
    1000
  );
}
```



```
componentWillUnmount() {
  clearInterval(this.timerID);
}
```

State and Lifecycle

1. When <Clock /> is passed to root.render(), React calls the constructor of the Clock component
   - it initializes this.state with an object including the current time
2. React then calls Clock component’s render() method
   - This is how React learns what should be displayed on the screen
3. When the Clock output is inserted in the DOM, React calls the componentDidMount() lifecycle method.
4. Every second the browser calls the tick() method.
5. If the Clock component is ever removed from the DOM, React calls the componentWillUnmount() lifecycle method so the timer is stopped.

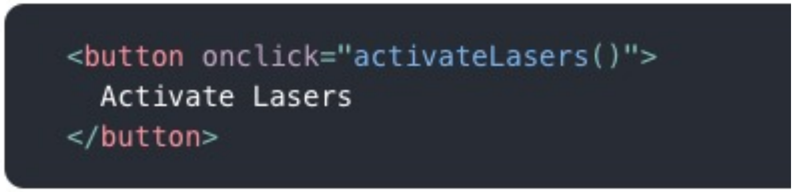


```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }

  componentDidMount() {
    this.timerID = setInterval(
      () => this.tick(),
      1000
    );
  }

  componentWillUnmount() {
    clearInterval(this.timerID);
  }

  tick() {
    this.setState({
      date: new Date()
    });
  }

  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Clock />);
```

Handling Events

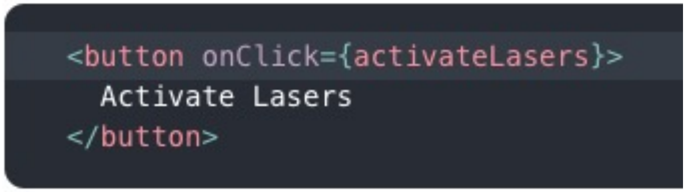
- Handling events with React elements is very similar to handling events on DOM elements. There are some syntax differences:
  - React events are named using camelCase, rather than lowercase.
  - With JSX you pass a function as the event handler, rather than a string.

For example, the HTML:

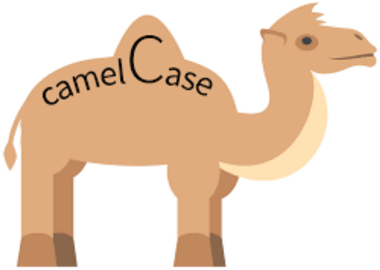


```
<button onclick="activateLasers()">
  Activate Lasers
</button>
```

is slightly different in React:



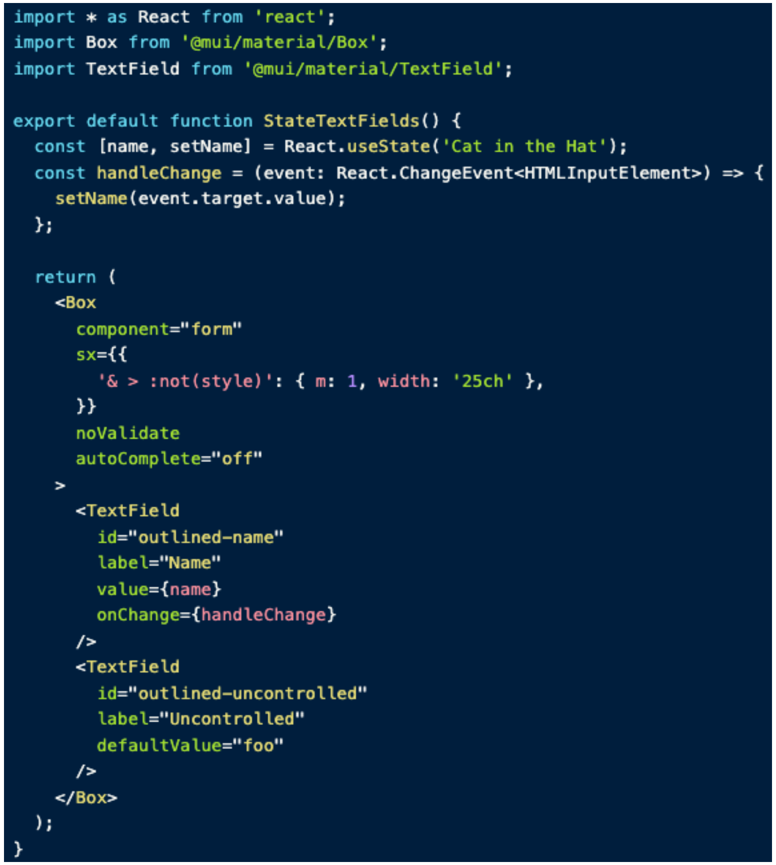
```
<button onClick={activateLasers}>
  Activate Lasers
</button>
```



Summary

A usual React component comprises three parts:

- structures: html
  - represented by JSX
- styles: css
  - how it is expressed depends on the styling solution.
- behaviours: js
  - state variables
  - event handlers



```
import * as React from 'react';
import Box from '@mui/material/Box';
import TextField from '@mui/material/TextField';

export default function StateTextFields() {
  const [name, setName] = React.useState('Cat in the Hat');
  const handleChange = (event: React.ChangeEvent<HTMLInputElement>) => {
    setName(event.target.value);
  };

  return (
    <Box
      component="form"
      sx={{
        '& > :not(style)': { m: 1, width: '25ch' },
      }}
      noValidate
      autoComplete="off"
    >
      <TextField
        id="outlined-name"
        label="Name"
        value={name}
        onChange={handleChange}
      />
      <TextField
        id="outlined-uncontrolled"
        label="Uncontrolled"
        defaultValue="foo"
      />
    </Box>
  );
}
```

Refs

- https://www.w3schools.com/js/js_htmldom.asp
- https://javascript.info/
- https://www.w3schools.com/js/js_window.asp
- https://hygger.io/blog/mpa-vs-spa-traditional-web-apps-or-single-page-applications/
- https://create-react-app.dev/docs/getting-started/
- https://reactjs.org/docs/getting-started.html

Exercises

Ex 1.

- https://www.freecodecamp.org/learn/front-end-development-libraries/#react

- Ex 2.
  - Create a simple game Tic-Tac-Toe
  - https://react.dev/learn/tutorial-tic-tac-toe