

Learning Modern JavaScript

© L. Hernández, 2023

Finding Elements in the DOM

HTML, CSS, and JavaScript

What the browser displays is a *collision* of **HTML**, **CSS**, and **JavaScript** working together to create what gets shown.

JavaScript is a language that allows us to add kinds of interactive *things* people are looking for.

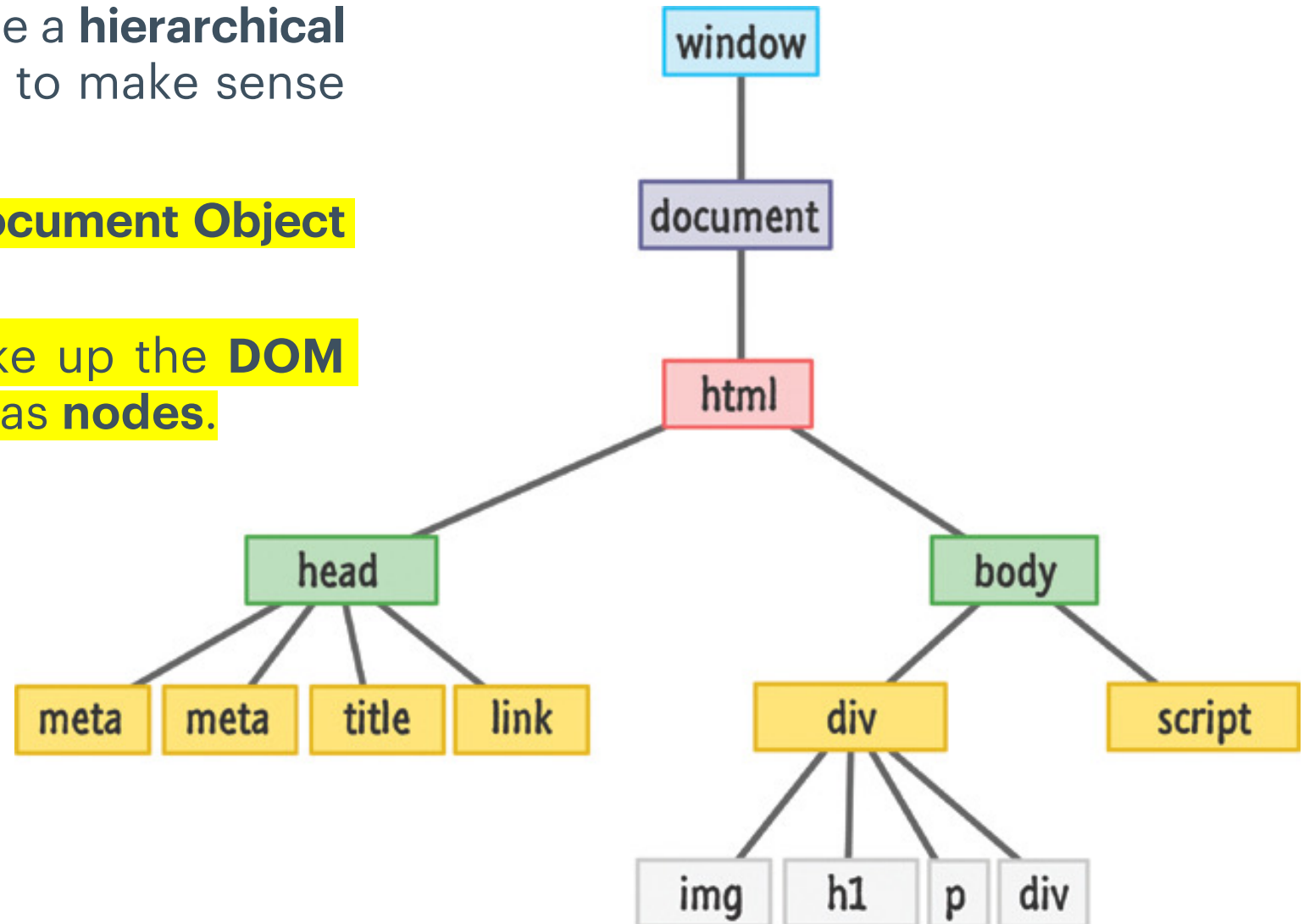


Meet the DOM

Under the covers, there will be a **hierarchical structure** that browser uses to make sense of *everything* going on.

This structure is known as **Document Object Model** or the **DOM**.

All of those *things* that make up the **DOM** are more generically known as **nodes**.



The window Object

The **root** of this hierarchy is the **window** object, which contains many properties and methods that help us work with the **browser**.



The window Object

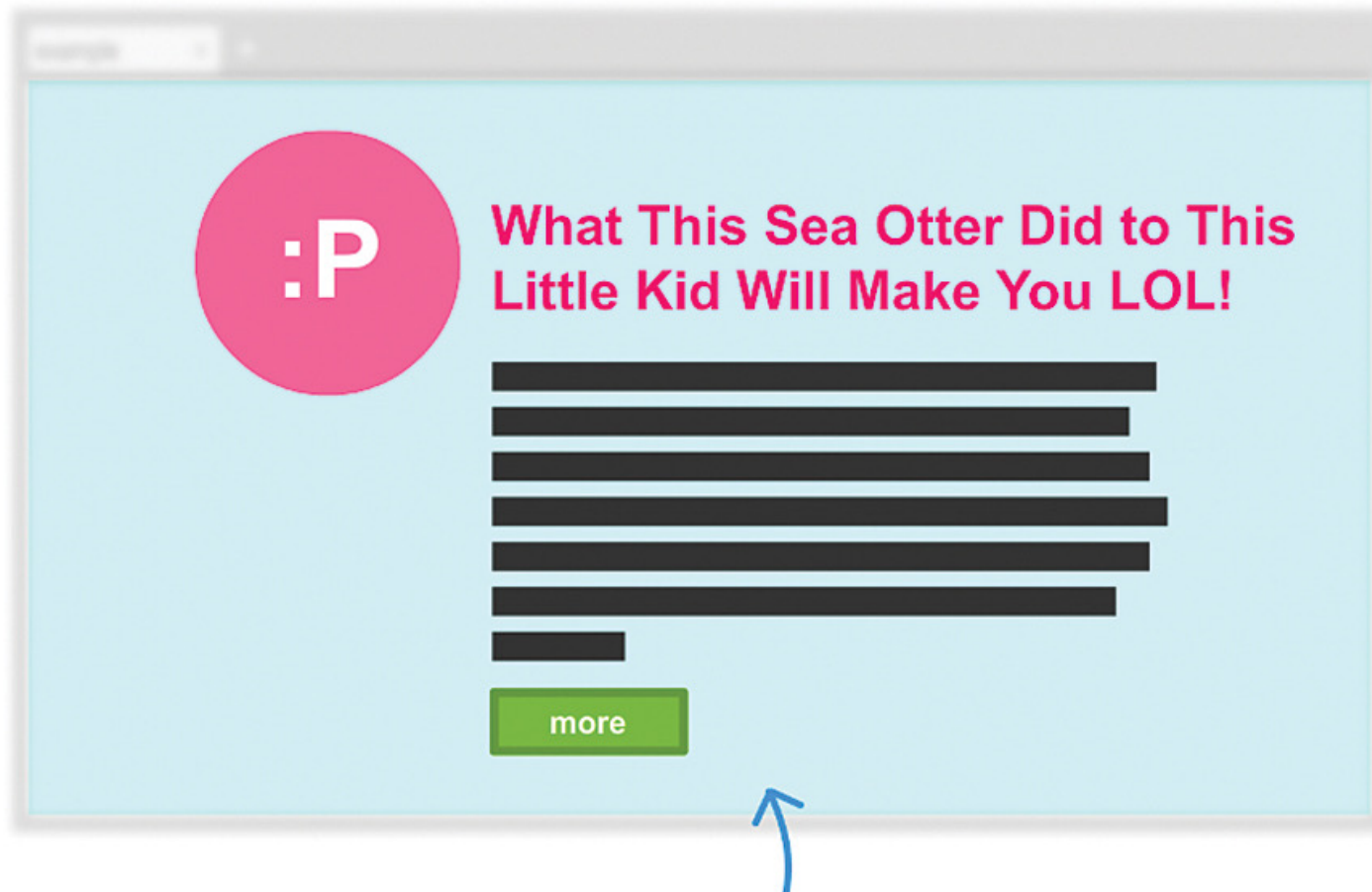
Some of *things* we can do with the help of **window** object:

- Accessing the current URL.
- Getting information about any frames in a page.
- Using **local storage**.
- Seeing information about the screen.
- Using the scroll bar or setting the status bar text.

All sorts of *things* that are applicable to the **container** the web page is displayed in.

The document Object

The **document** object is the *gateway* to all the **HTML elements** that make up what gets shown.



The document Object

The **document** object does not simply represent a *read-only version* of the **HTML document**; it is a *two-way street*, where we can read as well as manipulate the **HTML document** at will.

Any change we make to the **DOM** via **JavaScript** is reflected in what gets shown in the **browser**.

We also will use functionality the **document** object provides for listening to and reacting to **events**.

Finding Elements in the DOM

There are many ways to find these **HTML elements** that they are in the **DOM**.

The **querySelector** function takes as argument a string that represents **CSS selector** for **HTML element** we wish to find.

What gets returned by **querySelector** is first **element** it finds (even if other elements exist) that could get targeted by the **selector**.

Meet the querySelector Family

```
<div id="main">  
  <div class="pictureContainer">  
      
  </div>  
  <div class="pictureContainer">  
      
  </div>  
  <div class="pictureContainer">  
      
  </div>  
  <div class="pictureContainer">  
      
  </div>  
</div>
```

The querySelector Function

Taking **HTML** from example, let's access the **div** whose **id** is **main**.

```
let element = document.querySelector("#main");
```

Similarly, let's specify the **selector** for the **pictureContainer** class.

```
let element = document.querySelector(".pictureContainer");
```

What gets returned is first **div** whose **class value** is **pictureContainer**.

Other **div** elements with the **class value** of **pictureContainer** will simply be ignored.

The `querySelectorAll` Function

The **`querySelectorAll`** function returns all **elements** it finds that match whatever **selector** we provide.

What gets returned is not a **single element**; instead, what gets returned is an **array-like container of elements**.

With exception of number of **elements** returned, we've described about **`querySelector`** applies to **`querySelectorAll`** as well.

The `querySelectorAll` Function

Taking the **HTML** from example, we can use **`querySelectorAll`** to display the **`src` attribute** of all the **`img` elements** that contain the **`class` value `theImage`**.

Using **`getAttribute`** we can read **`attribute values`** from **`elements`**.

```
let images = document.querySelectorAll(".theImage");
```

```
for (let i = 0; i < images.length; i++) {  
  let image = images[i];  
  console.log(image.getAttribute("src"));  
}
```

The CSS Selector Syntax

Any **complex expression** we can specify for a **selector** in any **CSS document** can specify as an **argument** to either **querySelector** or **querySelectorAll**.

To target all **img elements** without having to specify the **class value**:

```
let images = document.querySelectorAll("img");
```

To target only the image whose **src attribute** is set to **meh.png**:

```
let images = document.querySelectorAll("img[src='meh.png']");
```

To target only the immediate **img element** from its **parent div**:

```
querySelector("div + img")
```

The getElement Family

The **getElementById**, **getElementsByTagName**, and **getElementsByClassName** functions were functions we could have used to find **elements** in the **DOM**.

These functions still exist today, but the reasons for using them are limited.

One good reason is if we are looking for a list of **DOM nodes** that are *live* as opposed to *static*.

The getElement Family

```
<div id="container">  
  <span></span>  
  <span></span>  
  <span></span>  
</div>
```

```
<script>  
  let container = document.getElementById('container');  
  let liveSpans = container.getElementsByTagName('span');  
  console.log(liveSpans.length); // 3  
  container.appendChild(document.createElement('span'));  
  console.log(liveSpans.length); // 4  
</script>
```


The getElement Family

Methods like **getElementsByTagName()** and **getElementsByClassName()** return a collection of **HTML objects**.

These **HTML objects** are "live" and will update as the **DOM** updates.

The **querySelectorAll()** method, alternatively, returns a *static node list* which is not a live and will not update with the **DOM**.

The getElement Family

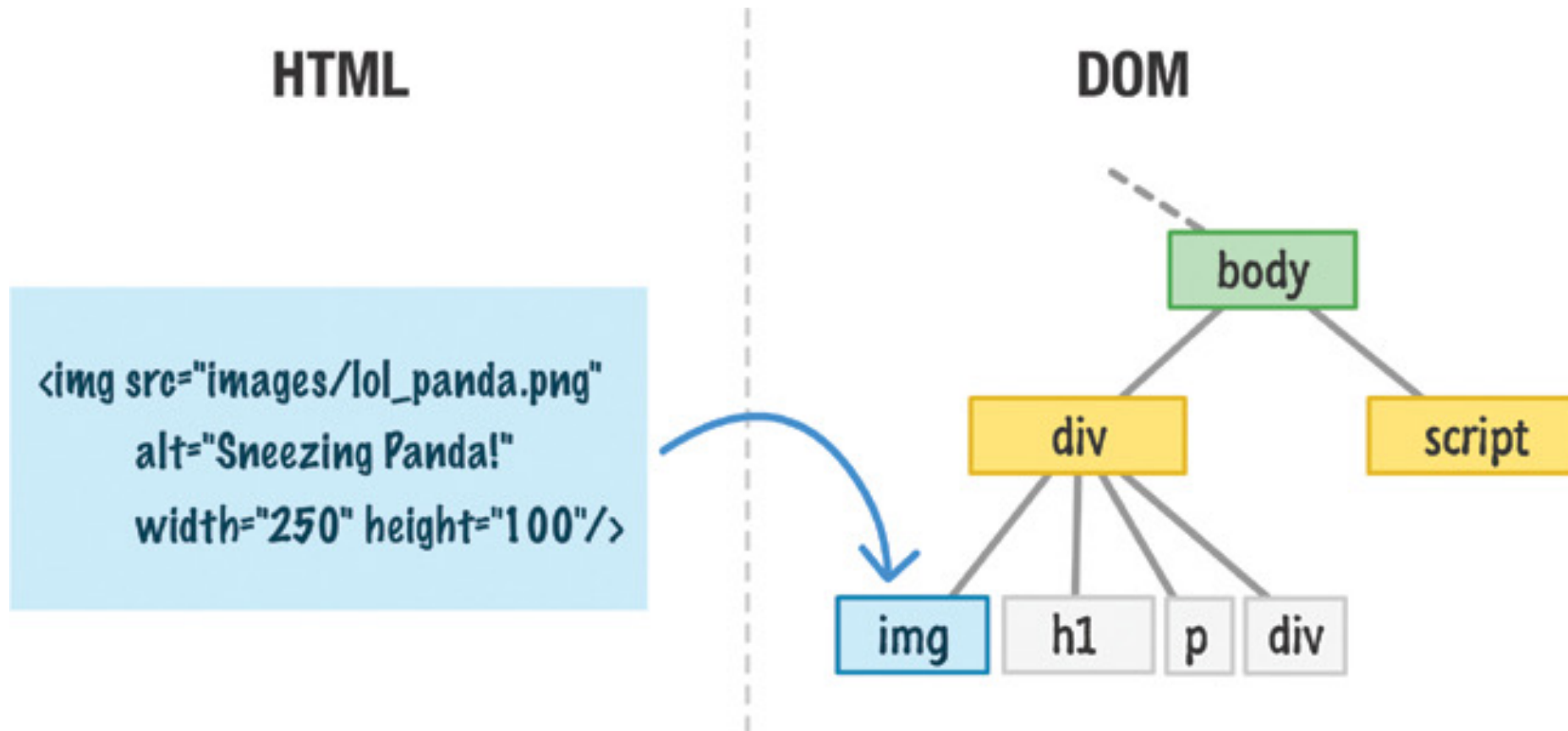
```
<div id="container">  
  <span></span>  
  <span></span>  
  <span></span>  
</div>
```

```
<script>  
  let container = document.getElementById('container');  
  let staticSpans = container.querySelectorAll('span');  
  console.log(staticSpans.length); // 3  
  container.appendChild(document.createElement('span'));  
  console.log(staticSpans.length); // 3  
</script>
```

Modifying DOM Elements

DOM Elements are Objects

Ability to use **JavaScript** to modify what gets shown by browser is made possible because every **HTML tag, style rule, or any other HTML element** that goes into a page has a representation in the **DOM**.



DOM Elements are Objects

HTML elements are versatile when viewed via the **DOM** is because they share similarities with **JavaScript objects**.

DOM elements contain **properties** that allow us to get/set values and call methods.

They have a form of *inheritance* where the functionality each **DOM element** provides is spread out across **Node**, **Element**, and **HTMLElement** base types.

Modifying DOM Elements

```
<!DOCTYPE html>
<html>
<head>
  <title>Hello...</title>

  <style>
    .highlight {
      font-family: "Arial";
      padding: 30px;
    }
    .summer {
      font-size: 64px;
      color: #0099FF;
    }
  </style>
</head>
...
```

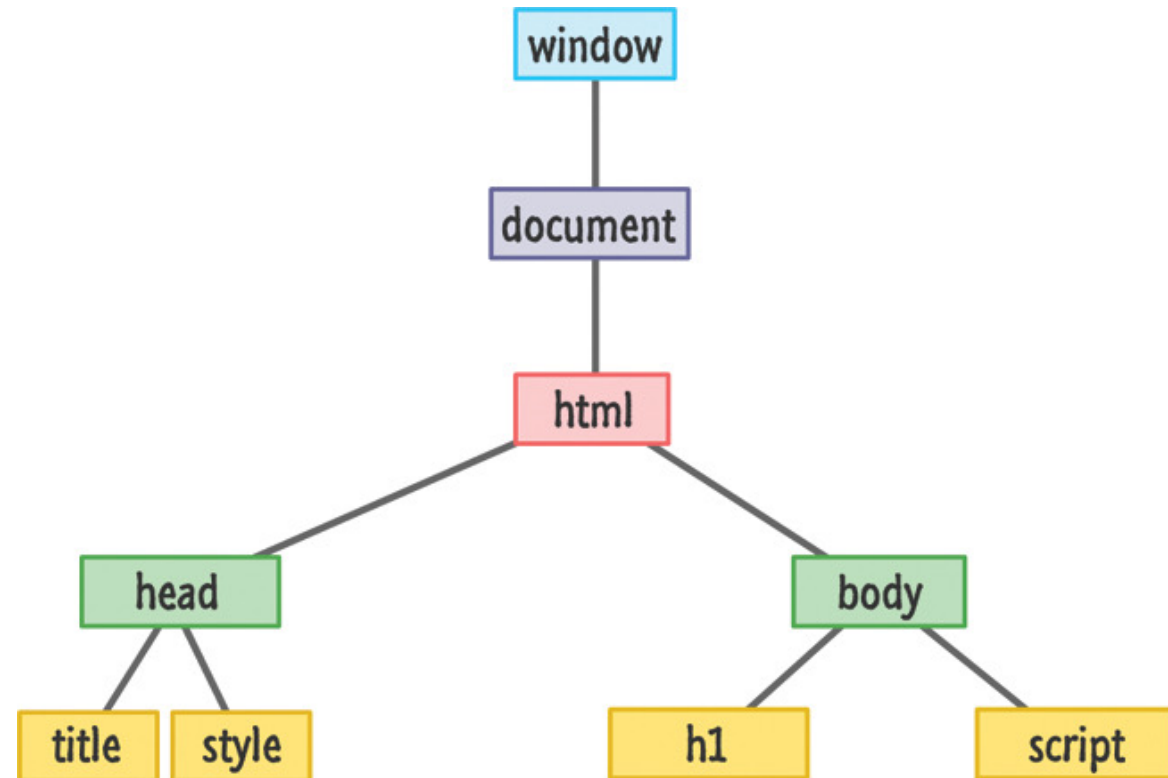
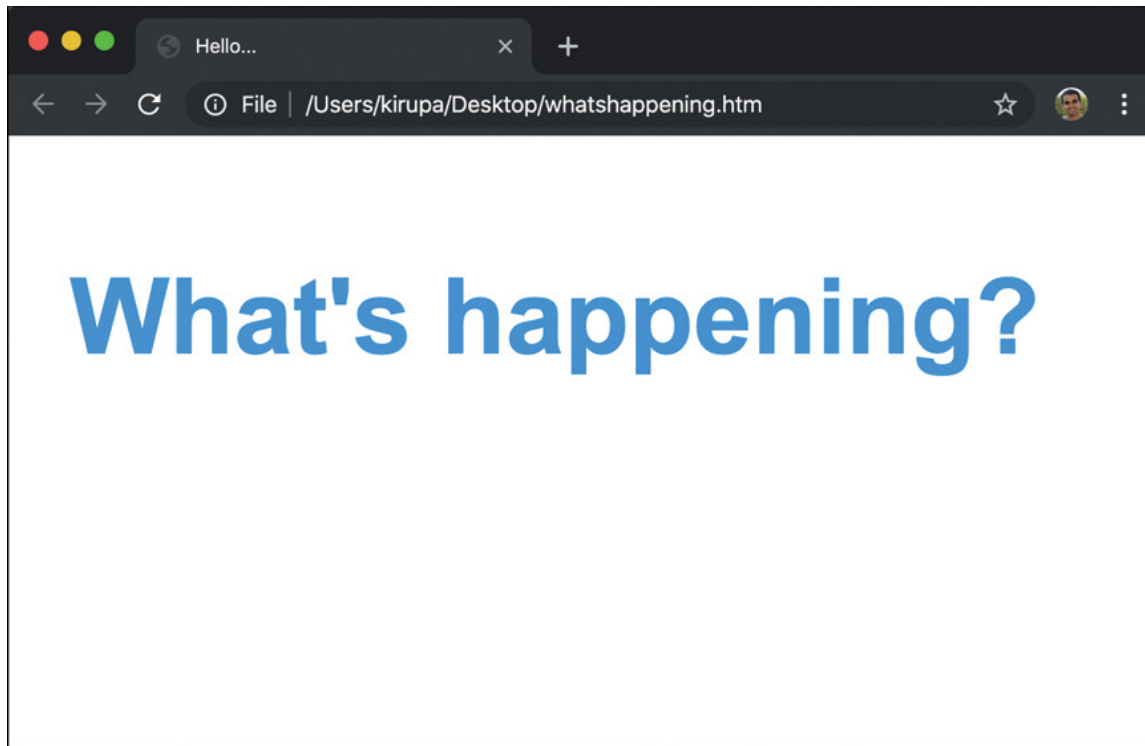
```
...
<body>

  <h1 id="bigMessage" class="highlight summer">
    What's happening?
  </h1>

  <script>
    // javascript code
  </script>
</body>
</html>
```

Modifying DOM Elements

If we preview this **HTML** in the browser, we will see something that looks like this.



Changing an Element's Text Value

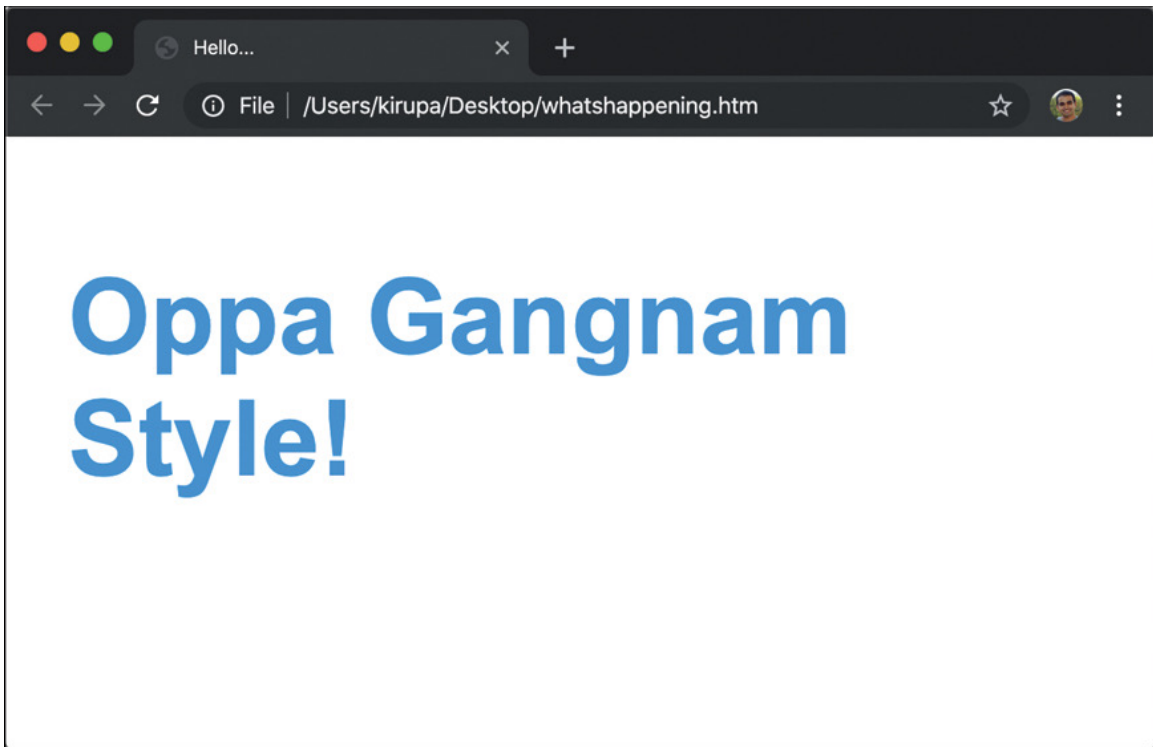
Once we have the reference to the **HTML element**, we can just set the **textContent** property on it to modify its **text value**.

```
let headingElement = document.querySelector("#bigMessage");  
headingElement.textContent = "Oppa Gangnam Style!";
```

The **textContent** property can be read like any variable to show the current value but we can also set the property like we are here to change the value that is stored currently.

Changing an Element's Text Value

If we make this change and preview it in the browser, we will see what is shown here.



Reading Attributes

To access common **attributes** in **JavaScript**, the most popular way is by using **getAttribute** and providing the attribute name as the argument.

Let's say that we have this **HTML element** where the **attributes** on it are **id** and **src**.

```

```

```
let imgElement = document.querySelector("#tv");
```

```
let idValue = imgElement.getAttribute("id");  
let srcValue = imgElement.getAttribute("src");
```

```
console.log(idValue) // tv  
console.log(srcValue) // foo.png
```

Reading Attributes

For **built-in attributes** such as the **id** and **src** attributes on an image element, we can access them directly by just dotting into it.

Another approach could be to use the **attributes** property on the **DOM** and iterate through the attributes.

```
let imgElement = document.querySelector("#tv");
```

```
let idValue = imgElement.id;  
let srcValue = imgElement.src;
```

```
console.log(idValue); // tv  
console.log(srcValue); // <full-path>/foo.png
```

Setting Attributes

Setting attributes is handled by the **setAttribute** method, which takes the attribute name and the new value as its arguments.

Let's use **setAttribute** to change the value of the **src** attribute to be **bar.png**.

```
let imgElement = document.querySelector("#tv");  
imgElement.setAttribute("src", "bar.png");  
  
console.log(imgElement.getAttribute("src")); // bar.png
```

Removing Attributes

The last basic attribute-related activity is how to remove attributes and this task is handled by the **removeAttribute** method.

To use it, we need to provide the name of the attribute to remove.

In this example we are removing the **src** attribute from the image element, so when we will try to access the removed attribute, this method returns a value of **undefined**.

```
let imgElement = document.querySelector("#tv");  
imgElement.removeAttribute("src");  
  
console.log(imgElement.getAttribute("src")); // undefined
```

Styling Content using CSS Properties

Styling Approaches

JavaScript not only lets us **style** the **element** we are interacting with; more importantly, it allows us to **style elements** all over the page.

One way to alter the **style** of an **element** using **JavaScript** is by setting a **CSS property** directly on the **element**.

The other way is by adding or removing **class values** from an **element**, which may result in certain **style rules** getting applied or ignored.

Special Names of CSS Properties

To specify a **CSS property** in **JavaScript** that contains a dash (-), simply remove it.

For example, **background-color** becomes **backgroundColor**, the **border-radius** property transforms into **borderRadius**, and so on.

Also, certain words in **JavaScript** are reserved and can't be used directly.

In **JavaScript**, to use a **property** whose name is reserved, prefix property with **css**, where **float** becomes **cssFloat**.

Setting the Style Directly

Every **HTML element** we access via **JavaScript** has a **style object** and this object allows us to specify a **CSS property** and set its **value**.

For example, this is what setting the background color of an **HTML element** whose **id** value is **superman**.

```
let myElement = document.querySelector("#superman");  
myElement.style.backgroundColor = "#D93600";
```

```
let myElements = document.querySelectorAll(".bar");
```

```
for (let i = 0; i < myElements.length; i++) {  
  myElements[i].style.opacity = 0;  
}
```

Adding and Removing Classes using JavaScript

This approach involves adding and removing **class values** that, in turn, change which **style rules** get applied.

For example, to apply the **.disableMenu** style rule, we need to add **disableMenu** as a **class value** to the **dropDown** element.

```
<ul id="dropDown" class="disableMenu" >  
  <li>One</li>  
  <li>Two</li>  
  <li>Three</li>  
  <li>Four</li>  
  <li>Five</li>  
  <li>Six</li>  
</ul>
```

Adding and Removing Classes using JavaScript

Accomplish this task involves setting the element's **className** property where we are responsible for maintaining current list of **class values** applied.

However, the list of **class values** is returned to us as a **string**.

If we have multiple **class values** we want to add, remove, or just toggle on/off, we have to do a bunch of error-prone string-related trickery that just isn't fun.

Adding Class Values

We have a new **API** that is known as **classList**, and makes adding and removing **class values** from an **element** easy.

To add a **class value** to an **element**, get a reference to element and call the **add** method on it via **classList**.

```
let divElement = document.querySelector("#myDiv");
divElement.classList.add("bar");
divElement.classList.add("foo");
divElement.classList.add("zorb");
divElement.classList.add("baz");

console.log(divElement.classList);
```

Removing Class Values

To remove a **class value**, we call the `remove` method on **classList**.

After this code executes, the **foo** class value will be removed and what we will be left with is just **bar**, **zorb**, and **baz**.

```
let divElement = document.querySelector("#myDiv");  
divElement.classList.remove("foo");  
  
console.log(divElement.classList);
```

Removing Class Values

For many styling scenarios, we can first check if a **class value** on an **element** exists.

If the value exists, we remove it from the **element** but, if the value does not exist, we add that **class value** to the **element**.

To simplify this, **classList** API provides us with the **toggle** method.

```
let divElement = document.querySelector("#myDiv");
divElement.classList.toggle("foo"); // remove foo
divElement.classList.toggle("foo"); // add foo
divElement.classList.toggle("foo"); // remove foo

console.log(divElement.classList);
```

Checking a Class Value Exists

The **contains** method checks to see if the specified **class value** exists on an **HTML element**.

If the value exists, we get **true**, and if the value doesn't exist, we get **false**.

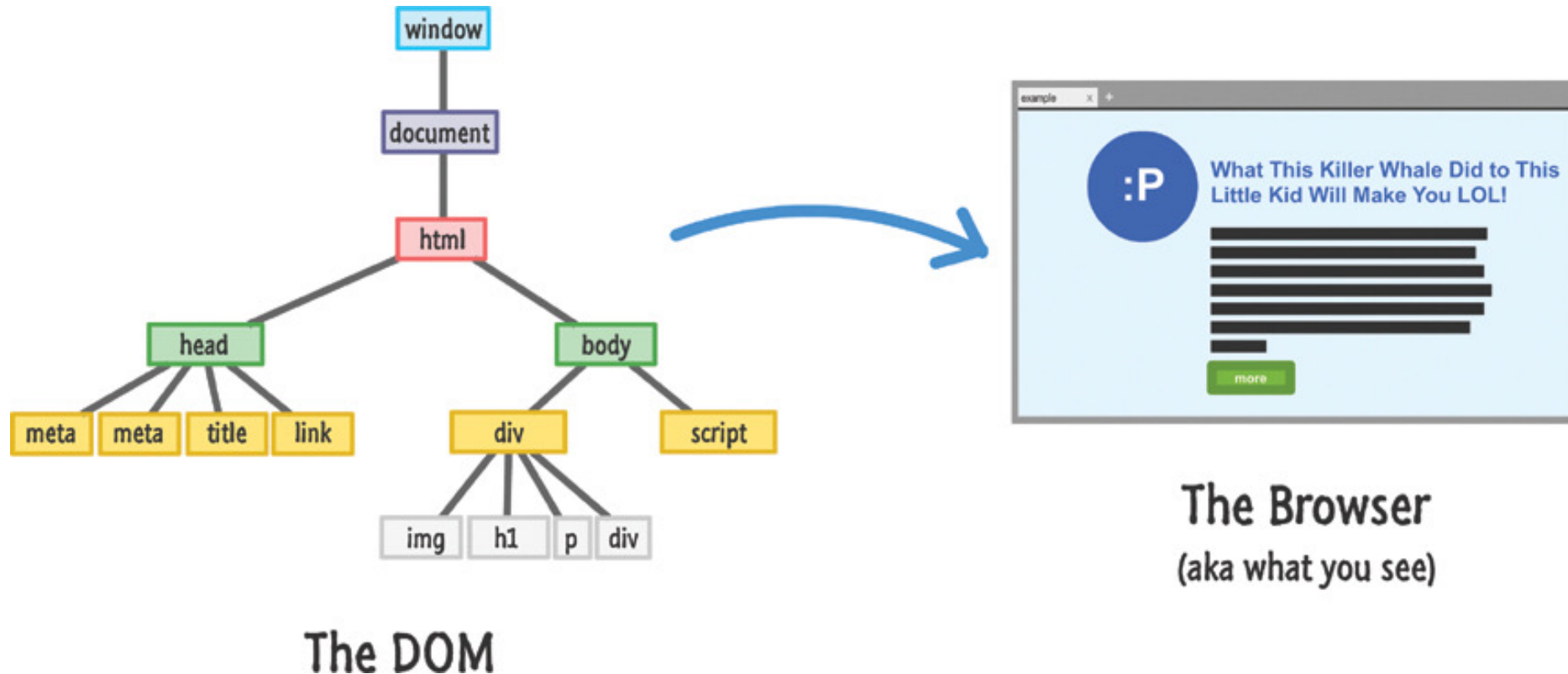
```
let divElement = document.querySelector("#myDiv");
```

```
if (divElement.classList.contains("bar") == true) {  
  // do something  
}
```

Traversing the DOM

Traversing the DOM

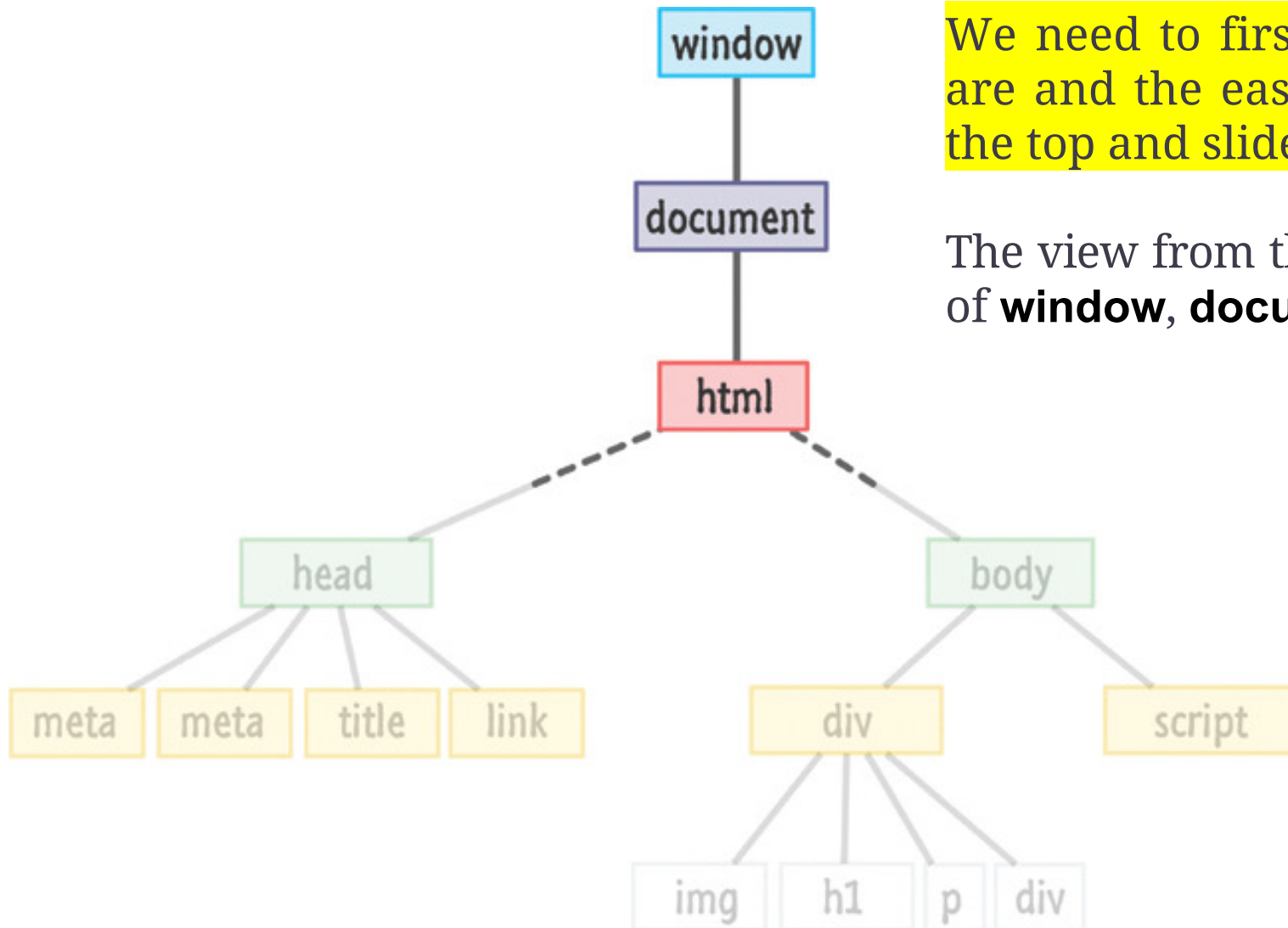
The **elements** in the **DOM** are arranged in a **hierarchy**, that defines what we will eventually see in the **browser**.



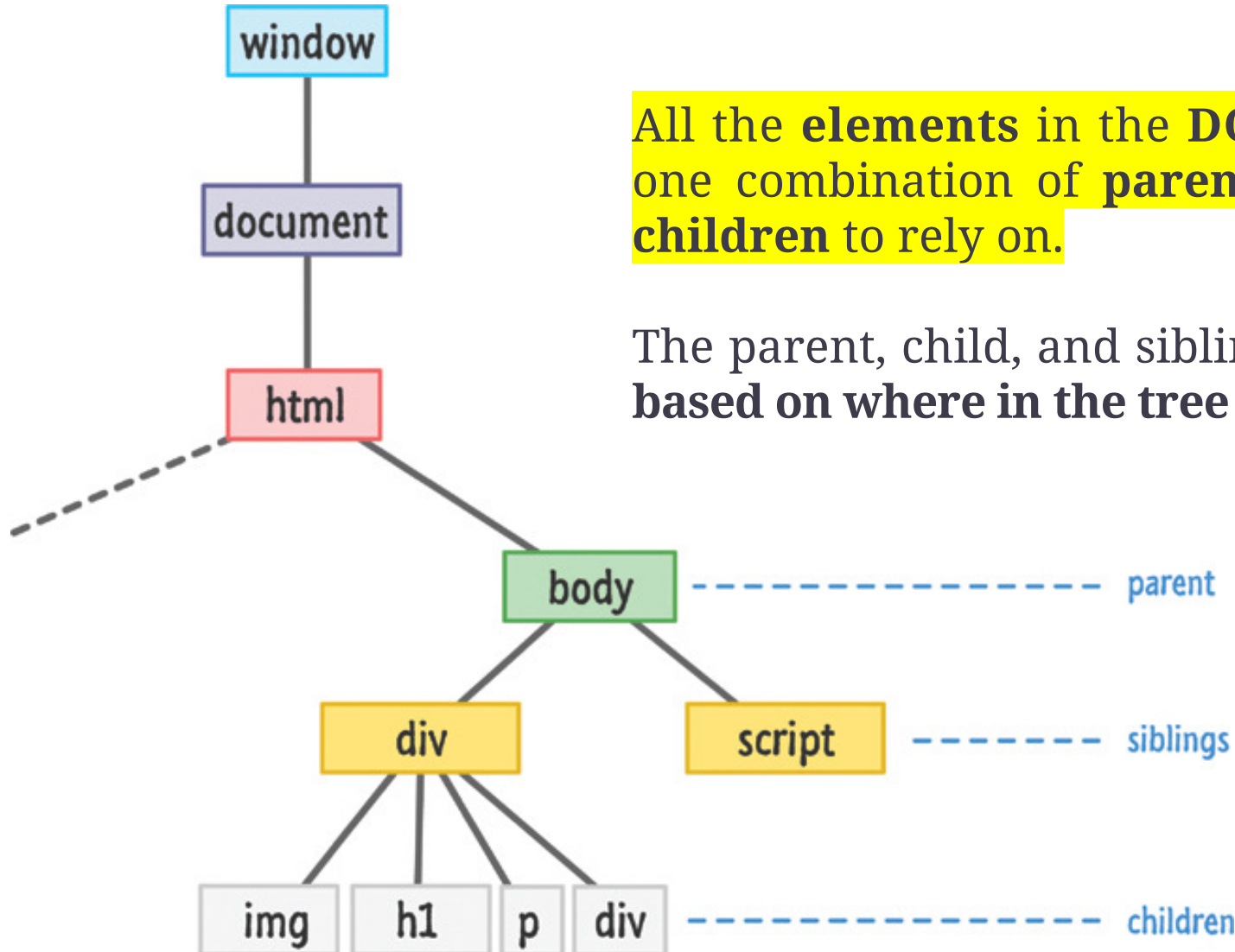
Finding a Way Around

We need to first get to where the **elements** are and the easiest way is to just start from the top and slide all the way down.

The view from the top of the **DOM** is made up of **window**, **document**, and **html** elements.



Finding a Way Around



All the **elements** in the **DOM** have at least one combination of **parents**, **siblings**, and **children** to rely on.

The parent, child, and sibling relationship is **based on where in the tree we are focusing**.

Finding a Way Around

One way of navigating around is by using **querySelector** and **querySelectorAll** to precisely get at these **elements** we are interested in.

If we don't know where we want to go so **querySelector** and **querySelectorAll** methods won't help us here.

To help us through this, we have a handful of **properties** that they are **firstChild**, **lastChild**, **parentNode**, **children**, **previousSibling**, and **nextSibling**.

Creating and Removing DOM Elements

Creating Elements

Having content predefined in our **HTML** is limiting so it's common for interactive sites to dynamically create **HTML elements** and have them live in the **DOM**.

To create elements by using the **createElement** method, we call it via the **document** object and pass in the HTML tag name of the element we wish to create.

When we create this **paragraph element (p)**, the **myElement** variable holds a reference to newly created **element**.

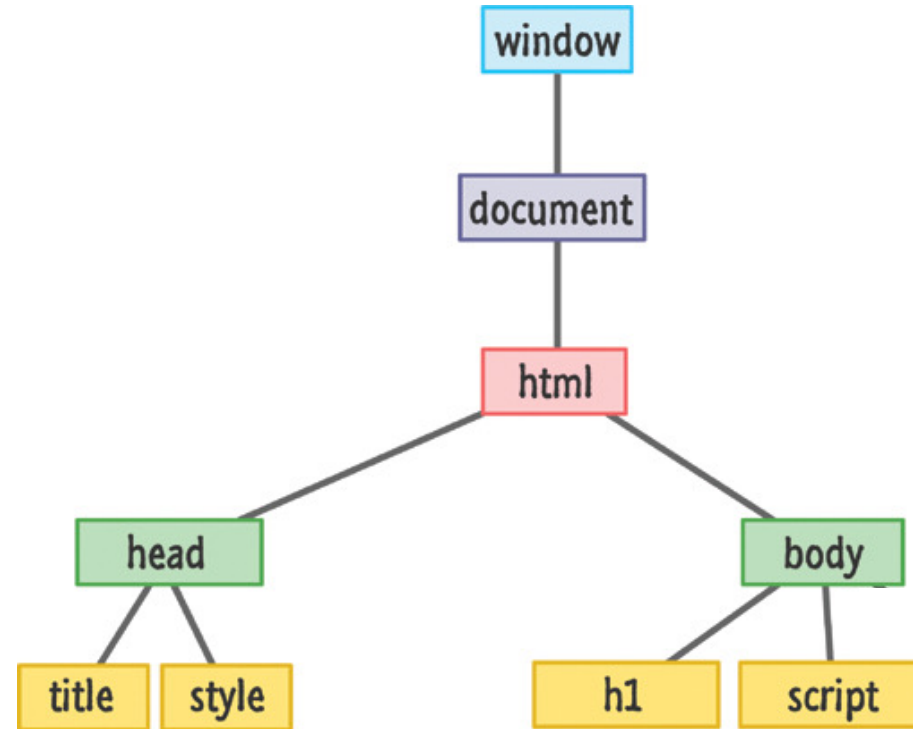
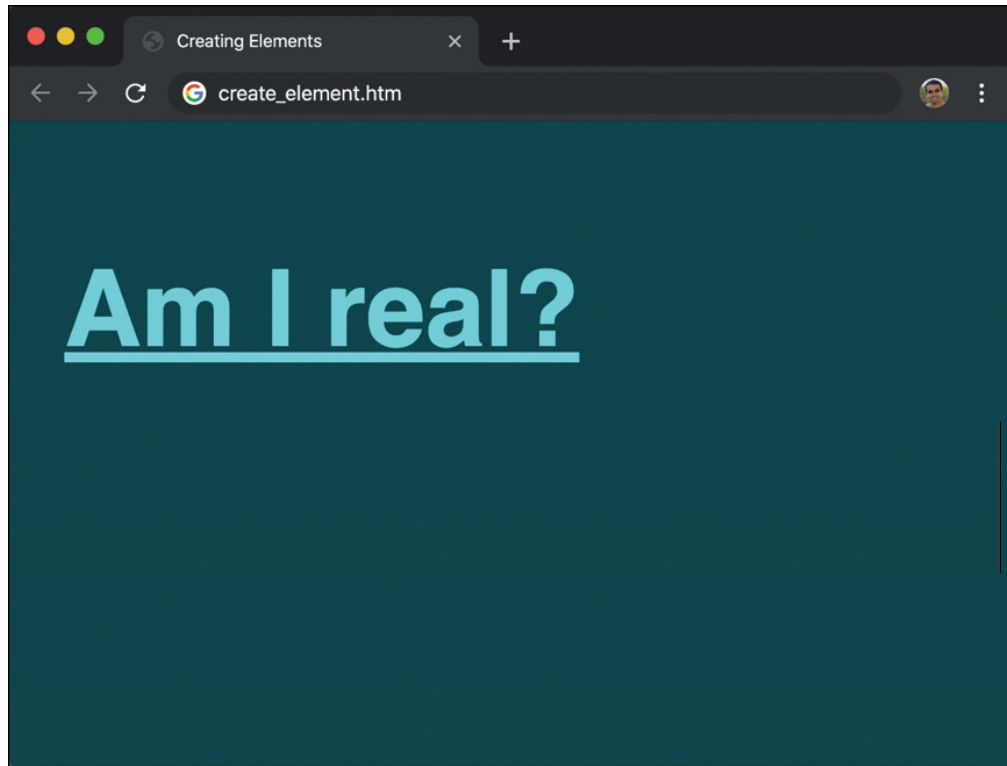
```
let myElement = document.createElement("p");
```

Creating Elements

```
<!DOCTYPE html>
<html>
<head>
  <title>Creating Elements</title>
  <style>
    body {
      background-color: #0E454C;
      padding: 30px;
    }
    h1 {
      color: #14FFF7;
      font-size: 72px;
      font-family: sans-serif;
      text-decoration: underline;
    }
  ...
```

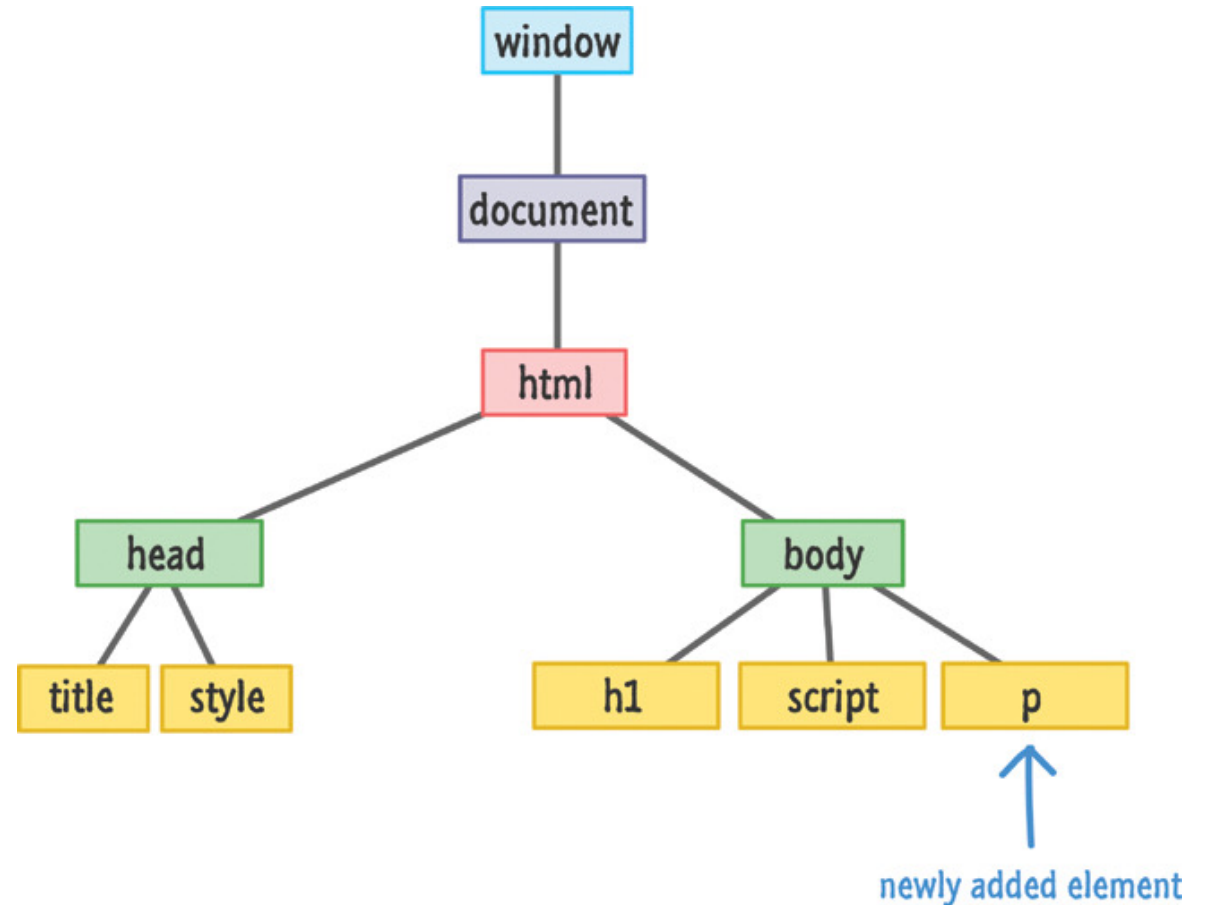
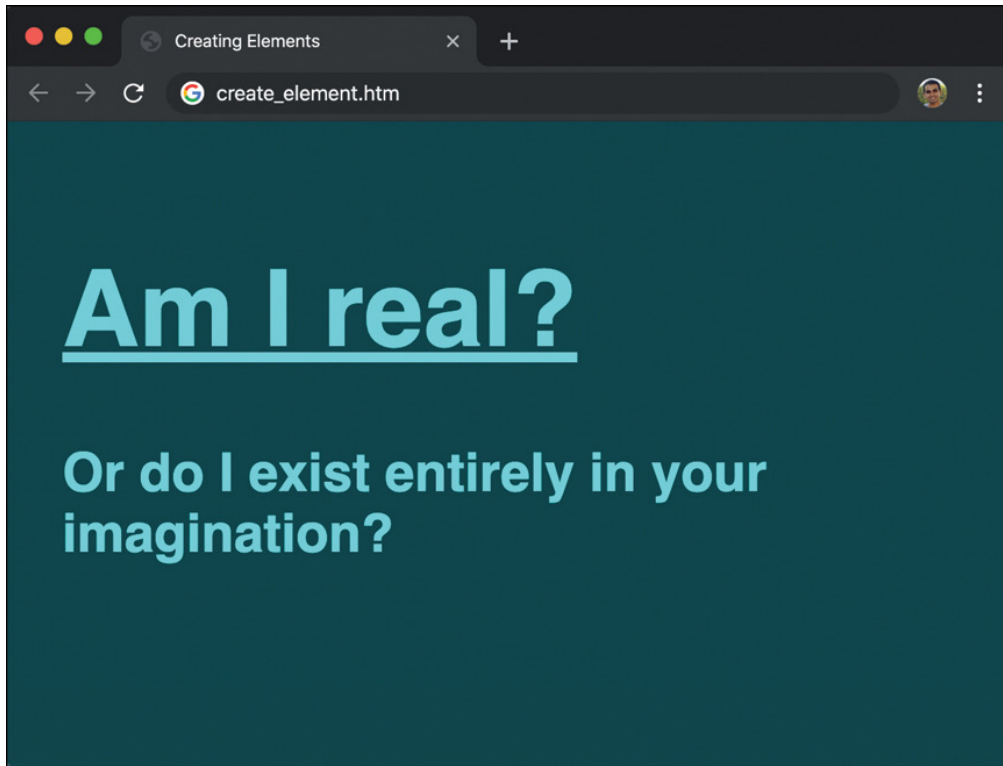
```
...
  p {
    color: #14FFF7;
    font-family: sans-serif;
    font-size: 36px;
    font-weight: bold;
  }
</style>
</head>
<body>
  <h1>Am I real?</h1>
  <script>
    // javascript code
  </script>
</body>
</html>
```

Creating Elements




```
let newElement = document.createElement("p");  
let bodyElement = document.querySelector("body");
```

```
newElement.textContent = "Or do I exist entirely in your imagination?";  
bodyElement.appendChild(newElement);
```



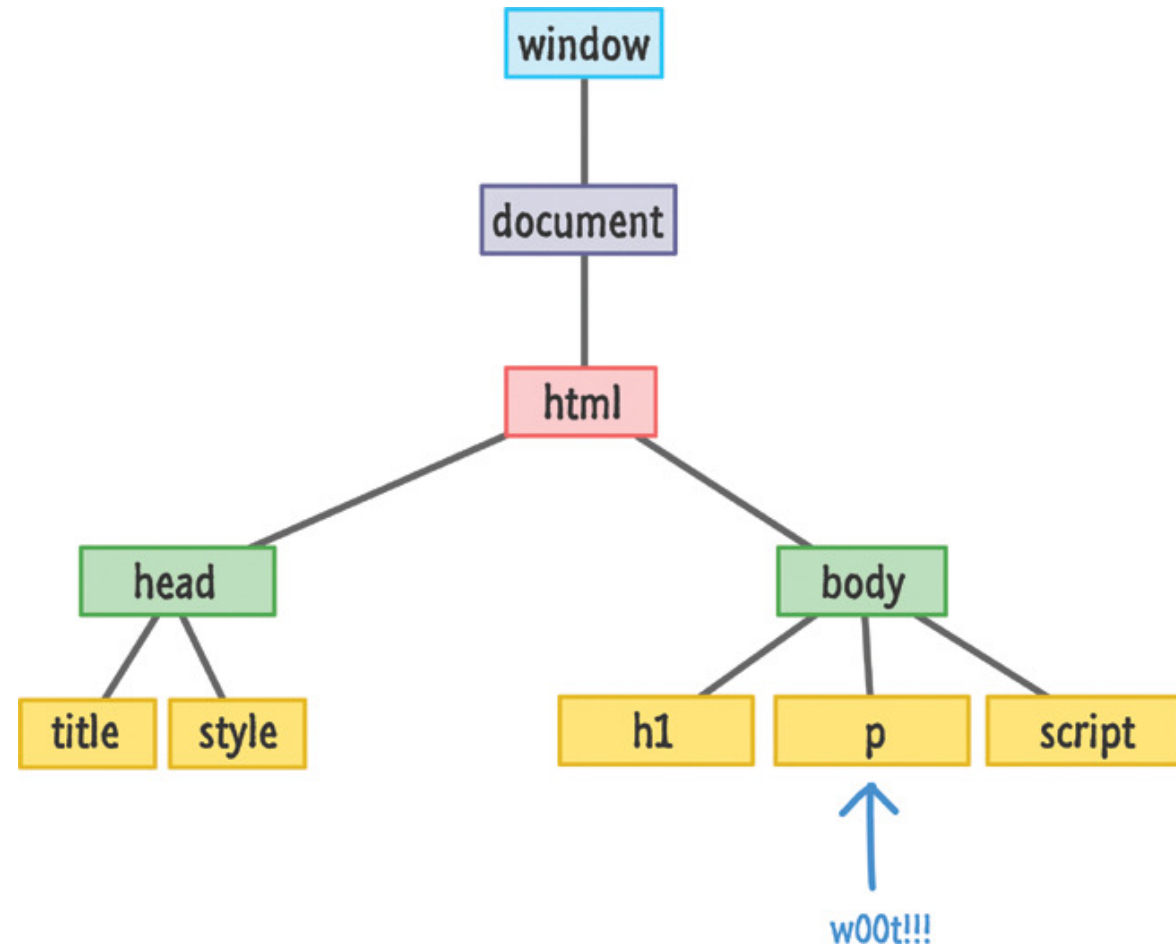
Creating Elements

If we want to insert **newElement** directly after the **h1** tag, we can do so by calling the **insertBefore** function on the parent.

First argument is element we want to insert and second is a reference to the sibling, aka the child of a parent, we want to precede.

```
let newElement = document.createElement("p");  
let bodyElement = document.querySelector("body");
```

```
let scriptElement = document.querySelector("script");  
newElement.textContent = "I exist entirely in your imagination.";  
bodyElement.insertBefore(newElement, scriptElement);
```



Creating Elements

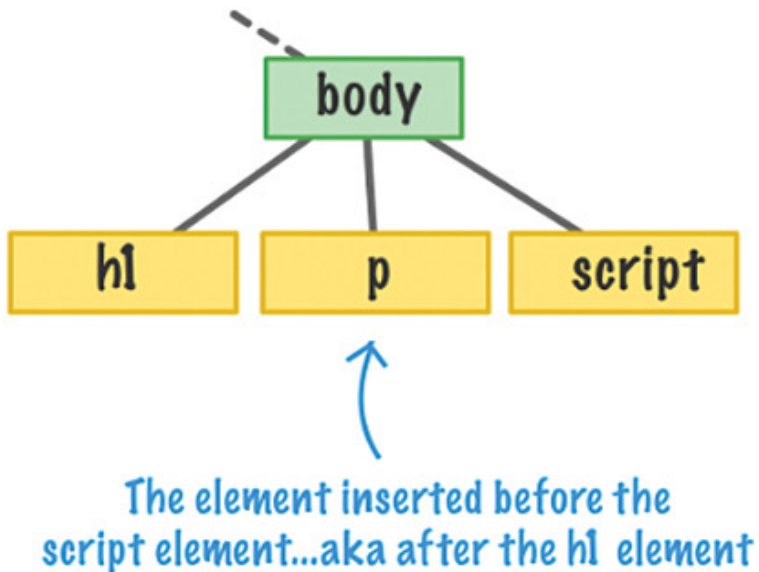
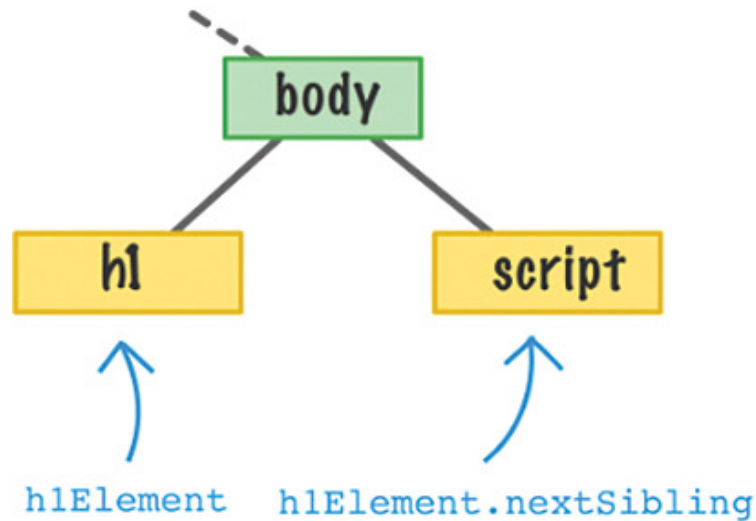
There isn't a supported built-in way of inserting an element after an element instead of before it.

What we can do is trick the **insertBefore** function by telling it to insert an element **an extra element ahead**.

If there is no sibling element to target, this function just appends the element we want to the end automatically.

Creating Elements

```
let newElement = document.createElement("p");  
let bodyElement = document.querySelector("body");  
let h1Element = document.querySelector("h1");  
  
newElement.textContent = "I exist entirely in your imagination."  
bodyElement.insertBefore(newElement, h1Element.nextSibling);
```



Creating Elements

A more generic way of adding children to a parent is by realizing that parent elements treat **children** like entries in an array.

To access this array of children, we have the **children** and **childNodes** properties.

The **children** property only returns **HTML elements**, and **childNodes** property returns the more generic **nodes**.

Removing Elements

Now, we are going to look at **removeChild** function, which, given its name, is all about removing elements.

If the **DOM element** we are removing has many levels of children, all of them will be removed as well.

```
let newElement = document.createElement("p");  
let bodyElement = document.querySelector("body");  
let h1Element = document.querySelector("h1");
```

```
newElement.textContent = "I exist entirely in your imagination."  
bodyElement.appendChild(newElement);
```

```
bodyElement.removeChild(newElement);
```

Removing Elements

There is a variation where we can remove **newElement** by calling **removeChild** on its parent by specifying **newElement.parentNode**.

```
let newElement = document.createElement("p");  
let bodyElement = document.querySelector("body");  
let h1Element = document.querySelector("h1");
```

```
newElement.textContent = "I exist entirely in your imagination.";
```

```
bodyElement.appendChild(newElement);
```

```
newElement.parentNode.removeChild(newElement);
```

Removing Elements

There is a newer and better way to remove an element, where we just call the **remove** method on the element we wish to remove.

```
let newElement = document.createElement("p");  
let bodyElement = document.querySelector("body");  
let h1Element = document.querySelector("h1");
```

```
newElement.textContent = "I exist entirely in your imagination.";
```

```
bodyElement.appendChild(newElement);
```

```
newElement.remove();
```


Cloning Elements

In the **DOM** manipulation technique for cloning elements, we start with one element and create identical replicas of it.

We clone an element by calling the **cloneNode** function on the element we wish to clone, along with providing a **true** or **false** argument to specify whether we want to clone just the element or the element and all of its children.

```
let bodyElement = document.querySelector("body");
```

```
let item = document.querySelector("h1");
```

```
let clonedItem = item.cloneNode(false);
```

```
// add cloned element to the DOM
```

```
bodyElement.appendChild(clonedItem);
```

```
<!DOCTYPE html>
<html>
<head>
  <title>Cloning Elements</title>
  <style>
    body {
      background-color: #60543A;
      padding: 30px;
    }
    h1 {
      color: #F2D492;
      font-size: 72px;
      font-family: sans-serif;
      text-decoration: underline;
    }
    p {
      color: #F2D492;
      font-family: sans-serif;
      font-size: 36px;
      font-weight: bold;
    }
  </style>
</head>
```

...

```
...
<body>
  <h1>Am I real?</h1>
  <p class="message">
    I exist entirely in your imagination.
  </p>

  <script>
    let bodyElement =
      document.querySelector("body");
    let textElement =
      document.querySelector(".message");

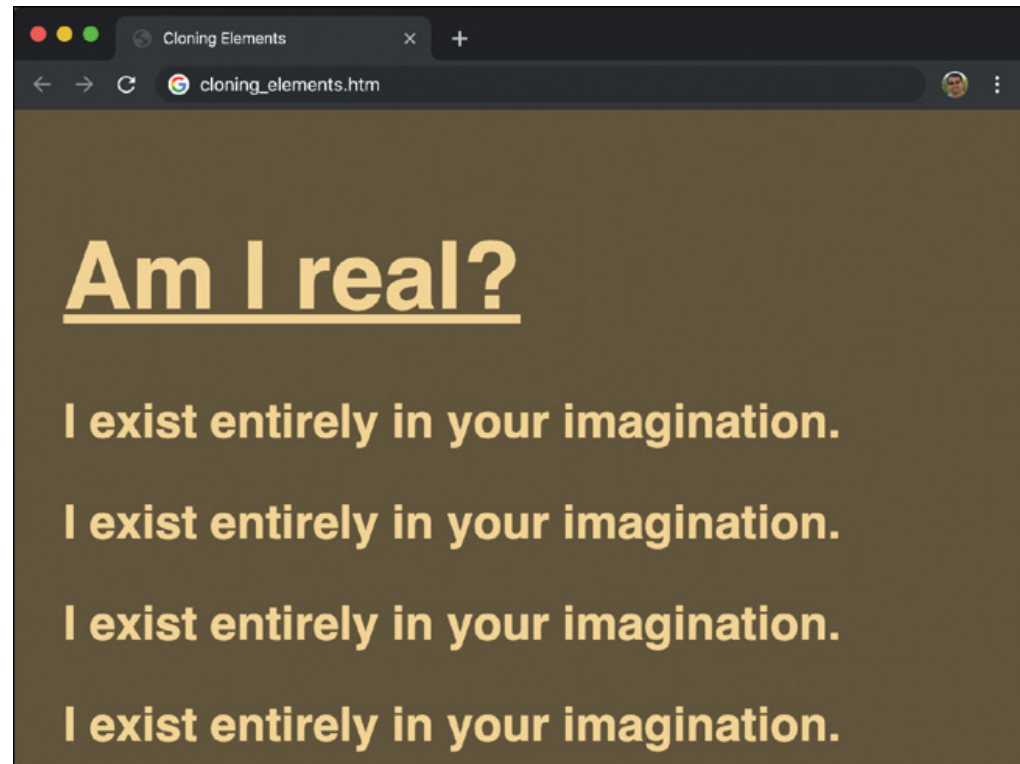
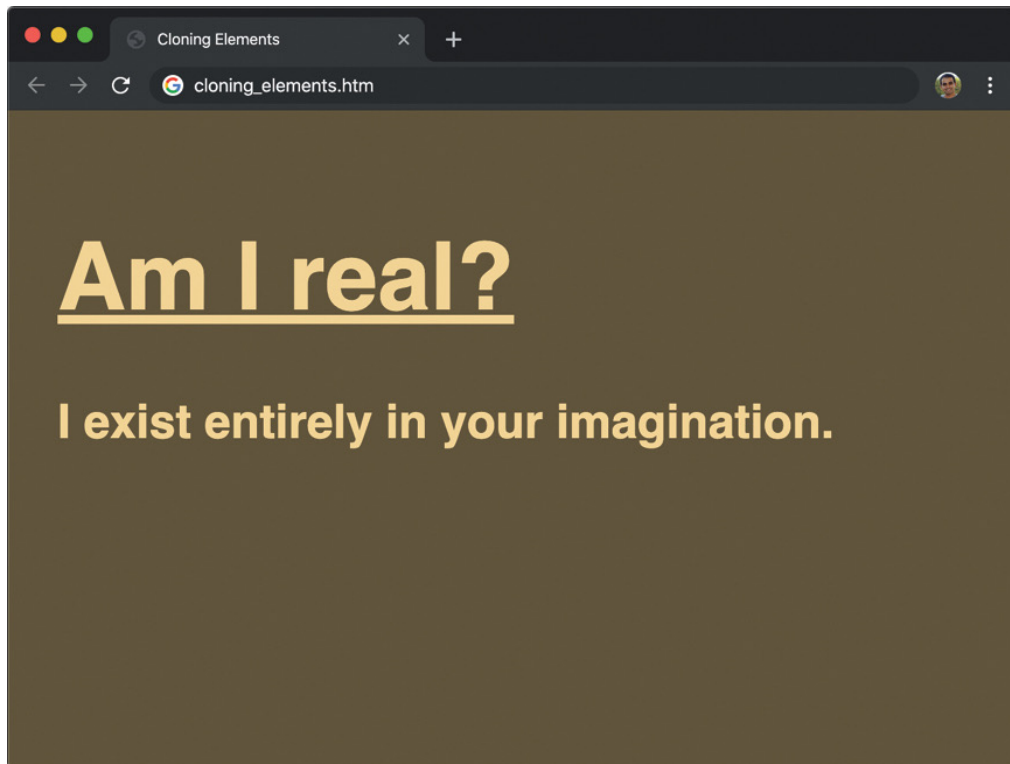
    setInterval(sayWhat, 1000);

    function sayWhat() {
      let clonedText = textElement.cloneNode(true);
      bodyElement.appendChild(clonedText);
    }
  </script>
</body>
</html>
```

Cloning Elements

Once the cloned elements have been added to the **DOM**, we can then use all the techniques we've seen to modify them.

If we preview it in a browser, after a few seconds, the message keeps duplicating.



Cloning Elements

In this example, what we are cloning is a **paragraph element**.

```
<p class="message">I exist entirely in your imagination.</p>
```

We're calling **cloneNode** with the **true** flag to indicate we want to clone all of the children as well.

```
let clonedText = textElement.cloneNode(true);
```

In this case, because distinction between **elements** and **nodes**, this **paragraph element** doesn't have any **child element**, but the text wrapped by the **p** tag is a **child node**.

Adding Many Elements into the DOM

When dealing with a large quantity of elements, **for maximum performance, whether we are adding one element or many elements, make all of the DOM updates at once.**

Quickly Add Elements


Click the **Generate** button below to display *ten thousand* elements that are created entirely in code! Neat, huh?


Generate


Quickly Add Elements


Click the **Generate** button below to display *ten thousand* elements that are created entirely in code! Neat, huh?

Generate

 1850.6109643975367

 5307.608567145799

 7617.747375312354

 8182.932659133995

```
<body>
  <div class="topContent">
    <h1>Quickly Add Elements</h1>
    <p>Click the <b>Generate</b> button below to display <i><b>ten thousand</b></i>
      elements that are created entirely in code! Neat, huh?</p>
    <button id="generateItems">Generate</button>
  </div>
  <div id="container">
  </div>
  <script>
    let container = document.querySelector("#container");
    let generateButton = document.querySelector("#generateItems");
    let emojis = [...];
    generateButton.addEventListener("click", generateContent, false);
    function generateContent(e) {
      // code goes here!
    }
  </script>
</body>
```

The innerHTML Approach

The easiest and fastest approaches for adding a large quantity of elements into the **DOM** is setting the **innerHTML** property with **all of the content we want to add in the form of a string**.

```
function generateContent(e) {  
  let htmlToAdd = "";  
  let numberOfItems = 1000;  
  
  for (let i = 0; i < numberOfItems; i++) {  
    let num = Math.random() * 10000;  
    let emoji = emojis[Math.floor(Math.random() * emojis.length)];  
    htmlToAdd += `<div class="item"><p>${emoji} ${num}</p></div>`;  
  }  
  
  container.innerHTML = htmlToAdd;  
}
```

The innerHTML Approach

This approach is very simple because we just deal with strings and concatenating them.

This approach is extremely fast because we make only one **DOM** update at the very end when we set the **innerHTML** to the **HTML elements** we generated.

On the downside, if we want more fine-grained control over the **HTML** we generate beyond doing string replacement, then working with strings can involve a lot of conditional statements.

The DocumentFragment Approach

In this approach, we will use a variable to store a reference to the **DocumentFragment** object, and we treat this like we would any other **DOM element**.

Because we are dealing with **DocumentFragment** object here, we can use **DOM methods** like **createElement**, **appendChild**, and **classList** to generate the **HTML structure** and build out the **DOM subtree**.

```
function generateContent(e) {  
  let fragment = new DocumentFragment();  
  
  let numberOfItems = 1000;  
  
  for (let i = 0; i < numberOfItems; i++) {  
    let num = Math.random() * 10000;  
    let emoji = emojis[Math.floor(Math.random() * emojis.length)];  
  
    let divElement = document.createElement("div");  
    divElement.classList.add("item");  
  
    let pElement = document.createElement("p");  
    pElement.innerText = `${emoji} ${num}`;  
  
    divElement.appendChild(pElement);  
    fragment.appendChild(divElement);  
  }  
  
  container.appendChild(fragment);  
}
```

The DocumentFragment Approach

Think of a **DocumentFragment** as an invisible, virtual container that gives us all the helper methods and capabilities that regular **DOM elements** provide **without actually creating a new DOM element**.

All of the **DOM manipulation** we are doing that would typically be expensive when done on a **live DOM tree** doesn't apply here.

The DocumentFragment Approach

Once we have finished building this **DOM subtree**, it's time to commit these changes to the **DOM** and visualize the result of all this data being mapped to **HTML elements**.

This step we do **exactly once** after the loop has finished, and the **DOM subtree stored by this fragment** is appended to the **container** element.

Because this acted like a **temporary container** to build the **DOM subtree**, only the **content** of the **DocumentFragment** survived and became a part of the parent element.

The DocumentFragment Approach

If we want to remove a large quantity of elements instead, we just need to clear out the entire **DOM subtree** of content.

A way to do this is by using on the **replaceChildren** method.

In our example, each time we click the **Generate** button, we don't want to keep adding a lot of elements each time.

We want to clear out current **DOM elements** before adding new ones.

The DocumentFragment Approach

```
function generateContent(e) {  
  container.replaceChildren();  
  
  let htmlToAdd = "";  
  let numberOfItems = 1000;  
  
  for (let i = 0; i < numberOfItems; i++) {  
  
    let num = Math.random() * 10000;  
    let emoji = emojis[Math.floor(Math.random() * emojis.length)];  
  
    htmlToAdd += `<div class="item"><p>${emoji} ${num}</p></div>`;  
  }  
  
  container.innerHTML = htmlToAdd;  
}
```

Dealing with Events

Listening for Events

Almost *everything* we do inside an website results in an **event** getting fired:

- 1) Website can **fire events automatically**, such as when it loads.
- 2) Website can **fire events as a reaction** to us interacting with it.

This task of **listening to the right event** is handled entirely by a function called **addEventListener**.

An **event listener** is responsible for **listening** so that it can **notify** another part of website when an interesting **event** gets fired.

Listening for Events

Event	When the Event is Fired
click	When we press down and release the primary mouse button, trackpad, and so on.
mousemove	Whenever we move the mouse cursor.
mouseover	When we move the mouse cursor over an element .
mouseout	When the mouse cursor moves outside the boundaries of an element .
dblclick	When we quickly click twice.
DOMContentLoaded	When the document's DOM has fully loaded.
load	When the entire document (DOM, external stuff like images, scripts, and so on) has fully loaded.
keydown	When we press down on a key on the keyboard.
keyup	When we stop pressing down on a key on the keyboard.
scroll	When an element is scrolled around.
wheel / DOMMouseScroll	Every time we use the mouse wheel to scroll up or down.

Listening for Events

Listening to events is handled by **event listener** but what to do after an event is overheard is handled by the **event handler**.

In a function that is designated as an **event handler**, this function is **specifically called out by name** from the **event listener** and receives an **Event** object as its argument.

```
document.addEventListener("click", doSomething, false);
```

```
function doSomething(event) {  
  // code goes here!  
}
```

A Simple Example

```
<!DOCTYPE html>
<html>
<head>
  <title>Click Anywhere!</title>
</head>

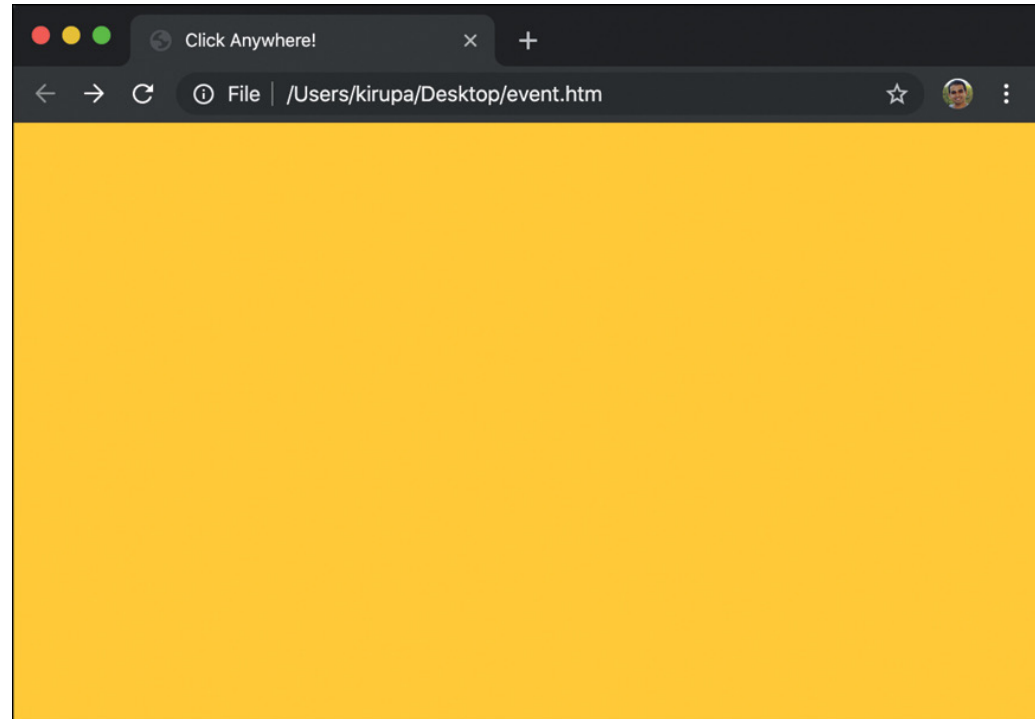
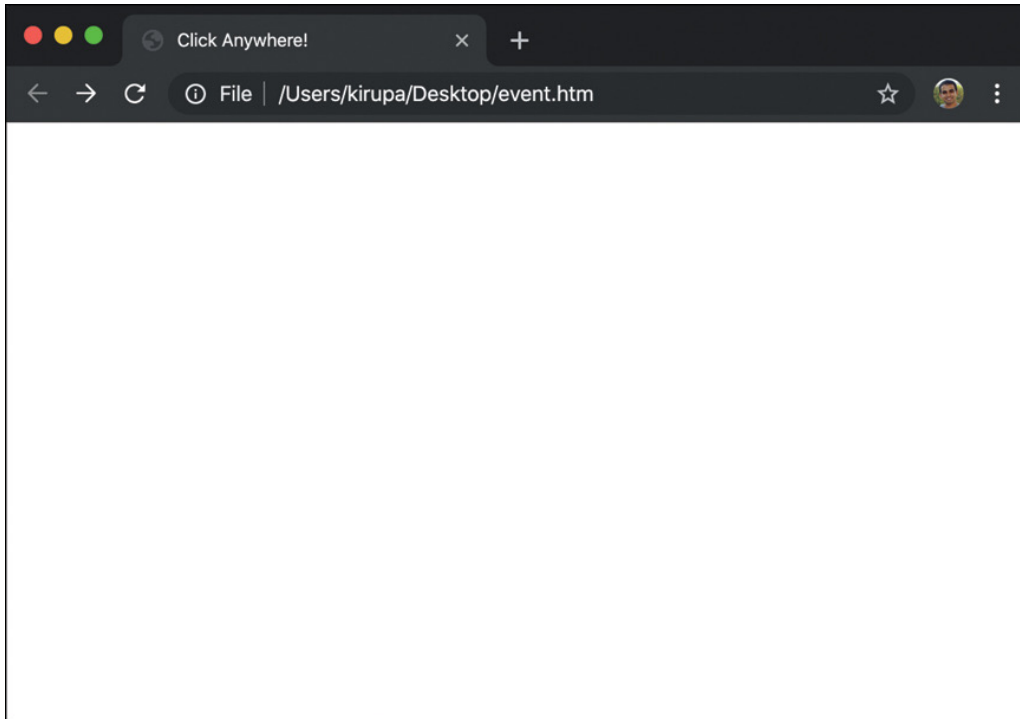
<body>
  <script>
    document.addEventListener("click", changeColor, false);

    function changeColor() {
      document.body.style.backgroundColor = "#FFC926";
    }
  </script>
</body>
</html>
```

A Simple Example

If we preview this document in the browser, we'll initially just see a blank page.

When we click anywhere on the page, the page's background will change from being white to a yellowish color.



The Event Arguments and Type

This **Event** type object contains properties that are **relevant to the event that was fired** and most events will have their own specialized behavior.

An event triggered by a mouse click will have different properties when compared to an event triggered by a key press, a page load, and so on.

```
function doSomething(event) {  
    // code goes here!  
}
```

Removing an Event Listener

To remove an **event listener**, we can use the **removeEventListener** function where we need to specify the exact same arguments.

```
document.addEventListener("click", changeColor, false);
```

```
function changeColor() {  
    document.body.style.backgroundColor = "#FFC926";  
}
```

```
...
```

```
document.removeEventListener("click", changeColor, false);
```

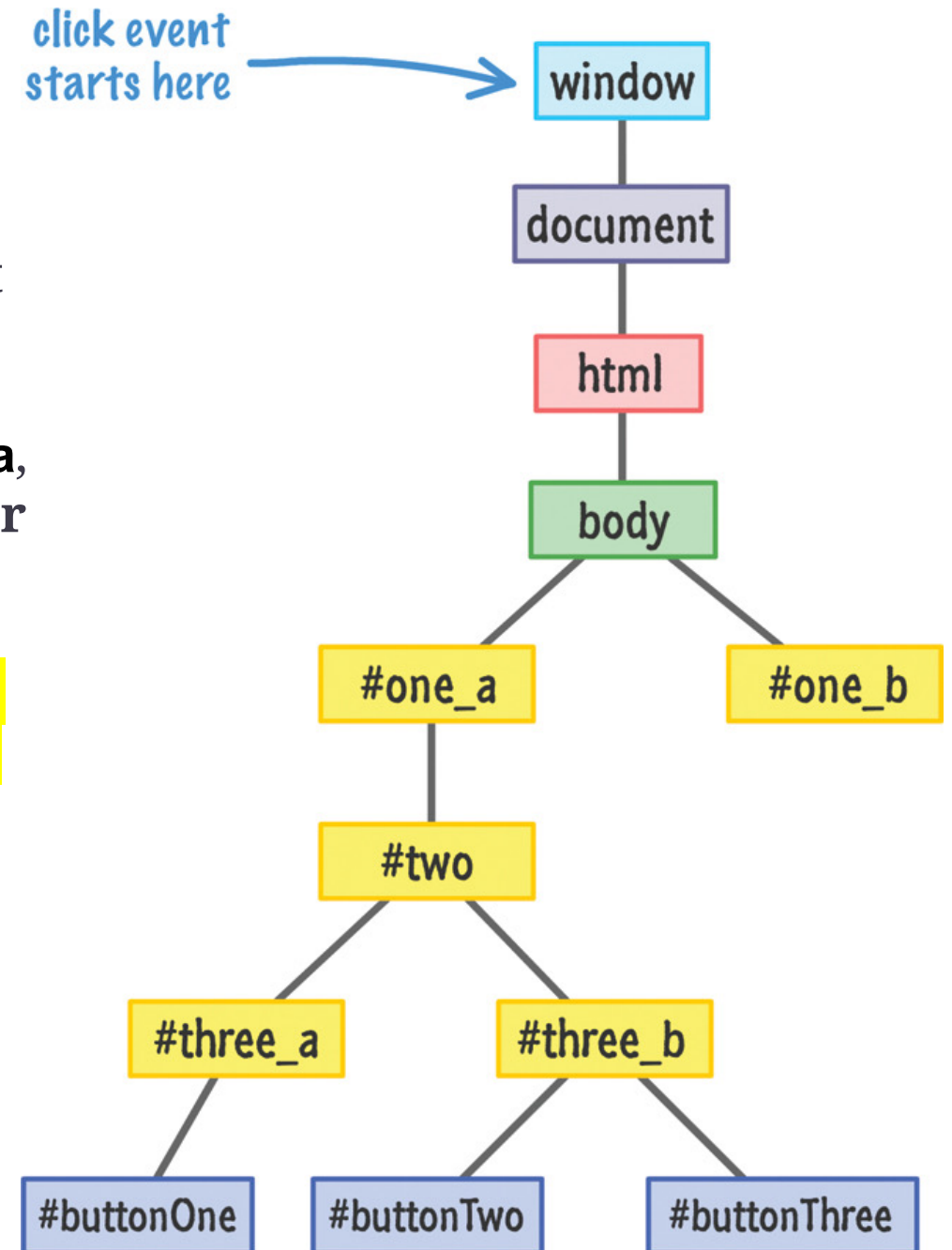
Event Bubbling and Capturing

Event Goes Down and Up

When we click the **buttonOne** element, a **click** event is going to be fired.

If we were to listen for a **click** event on **body**, **one_a**, **two**, or **three_a**, then the associated **event handler** will get fired.

This **click** event (like almost every other **JavaScript event**) does not actually originate at the **element** we interacted with.

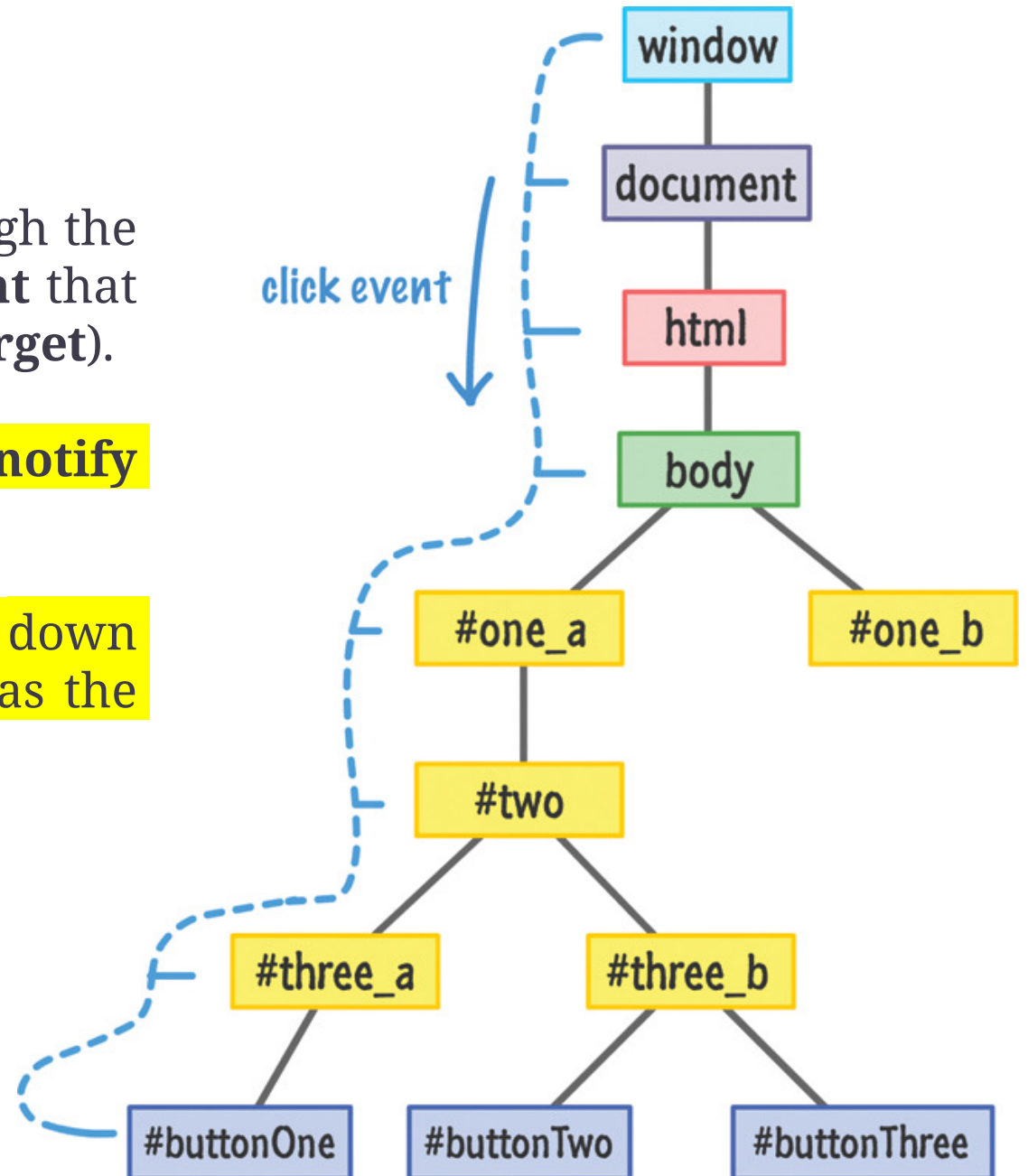


Event Goes Down and Up

From the **root**, the **event** makes its way through the pathways of the **DOM** and stops at the **element** that triggered the event (also known as the **event target**).

The path this **event** takes is direct, but **it does notify every element along that path.**

Where we initiate the **event** and this moves down the **DOM** from the **root** to **target** is known as the **event capturing phase.**

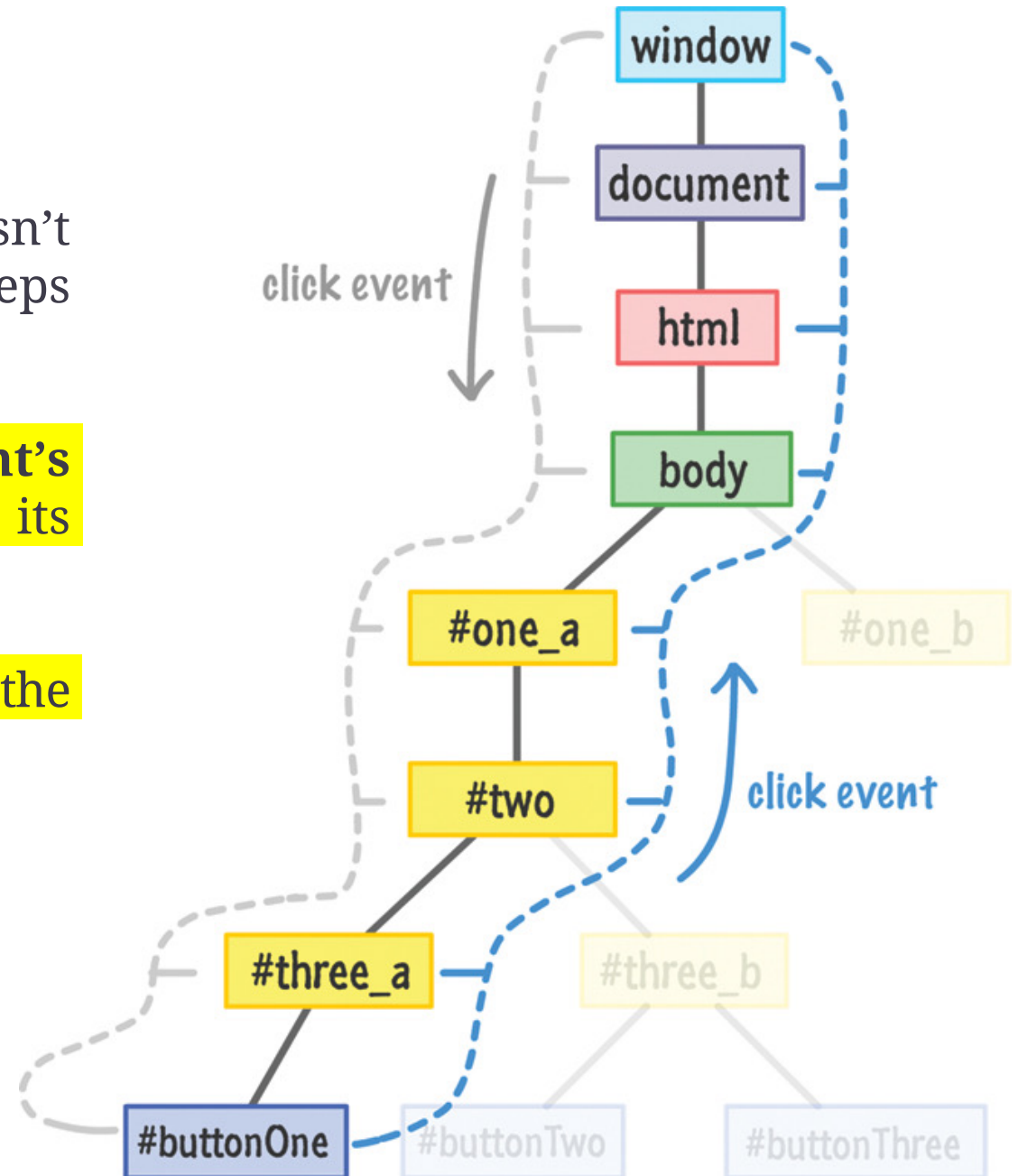


Event Goes Down and Up

Now, once this **event** reaches its **target**, it doesn't stop, the event keeps going by retracing its steps and returning back to the **root**.

Just like before, every **element** along the **event's path** as it is moving on up gets notified about its existence.

The phase, where this **event** bubbles back up to the **root**, is known as the **event bubbling phase**.



Meet the Event Phases

Choosing the **event phase** will be a detail that we specify with a **true** or **false** as part of the **addEventListener** call.

```
item.addEventListener("click", doSomething); // false
```

An argument of **true** means that we want to listen to the **event** during the **capture phase**.

```
item.addEventListener("click", doSomething, true);
```

If we specify **false**, this means we want to listen for the **event** during the **bubbling phase**.

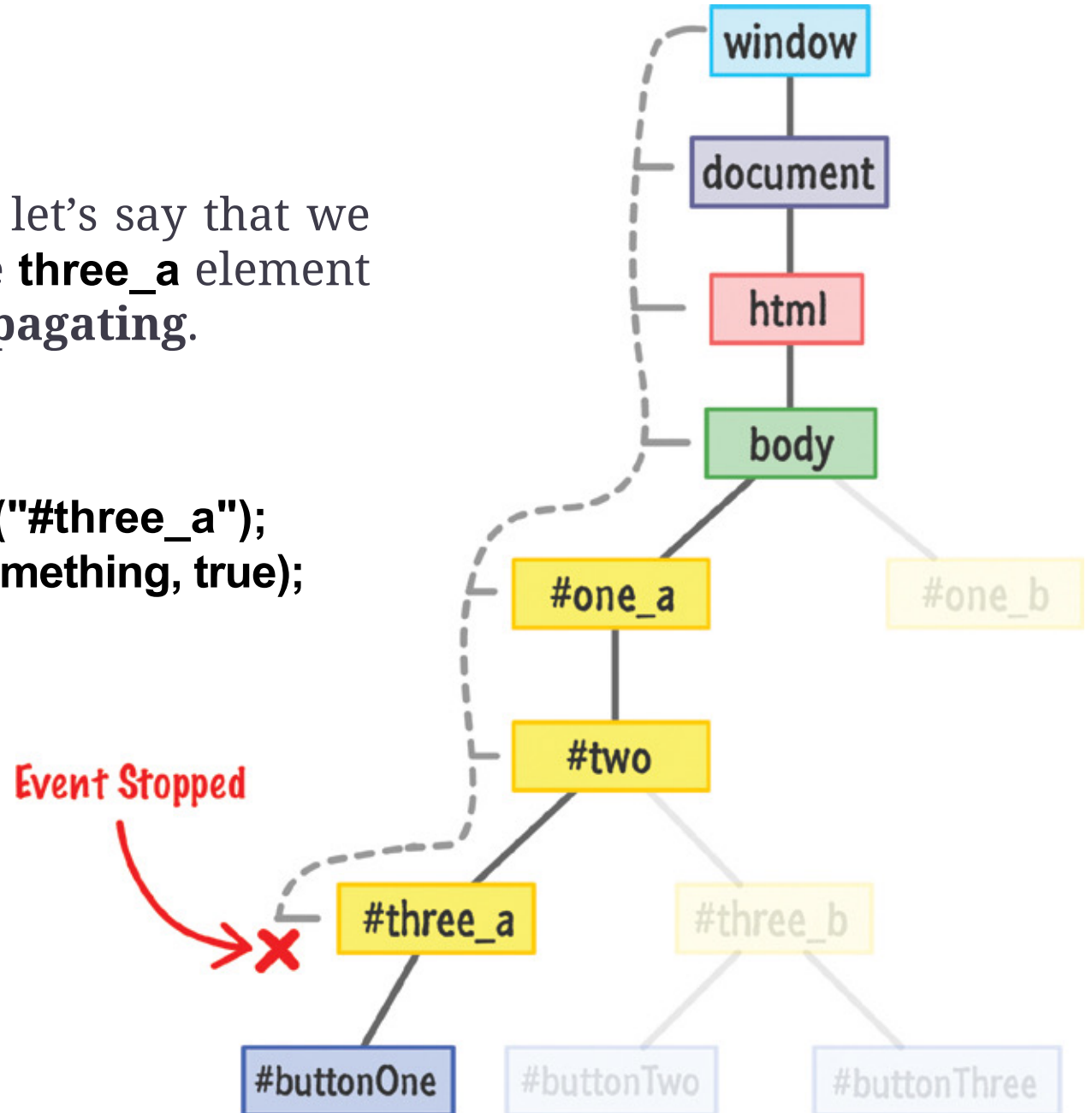
```
item.addEventListener("click", doSomething, false);
```

Interrupt an Event

Continuing with our earlier example, let's say that we are listening for the **click** event on the **three_a** element and **wish to stop the event from propagating**.

```
let element = document.querySelector("#three_a");  
element.addEventListener("click", doSomething, true);
```

```
function doSomething(e) {  
  e.stopPropagation();  
}
```



Interrupt an Event

There are many examples of **built-in reactions to events** any browser instinctively knows how to handle.

If we want to turn off this **default behavior**, we can call the **preventDefault** function.

This function needs to be called when reacting to an **event** on the **element** whose **default reaction** we want to ignore.

```
function overrideScrollBehavior(e) {  
    e.preventDefault();  
  
    // do something  
}
```

Page Load Events and Script Elements

What Happen during Page Load

When we click a link or press "Enter" after typing in a URL, the page will be loaded.

In that short period of time between we wanting to load a page and this loading, many relevant and interesting *stuff* happens that we need to know more about.

There are many factors that affect what the “right time” really is to run our code.

The DOMContentLoaded and load Events

If we have any code that relies on working with the **DOM**, such as a code that uses **querySelector** or **querySelectorAll** functions, we want to **ensure the code runs only after the DOM has been fully loaded**.

If we try to access the **DOM** before it has fully loaded, we may get incomplete results or no results at all.

The **DOMContentLoaded** and **load** Events

While a page loads, we can use **DOMContentLoaded** and **load** events to time **when exactly we want our code to run.**

The **DOMContentLoaded** event fires when the raw **markup** and the **DOM** of the page has been loaded and **the page's DOM is fully parsed.**

The **load** event fires once all **external resources** has been downloaded **the page has fully loaded** and we can interact with it.

The DOMContentLoaded and load Events

A way to ensure we never get into a situation where our code runs **before the DOM is ready** is to listen for the **DOMContentLoaded** event and let all of the code that relies on the **DOM** to run only after that **event** is overheard.

```
document.addEventListener("DOMContentLoaded", theDomHasLoaded, false)
```

```
function theDomHasLoaded(e) {  
  let headings = document.querySelectorAll("h2");  
  
  // do something with the images  
}
```

The DOMContentLoaded and load Events

For cases where we want our code to run **only after the page has fully loaded**, use the **load** event.

We never had too much use for the **load** event at the **document level**, outside of checking the final dimensions of a loaded image or creating a crude progress bar to indicate progress.

```
window.addEventListener("load", pageFullyLoaded, false);
```

```
function pageFullyLoaded(e) {  
  // do something again  
}
```

Scripts Location in the DOM

It doesn't matter if the **script** contains **inline code** or **references external resources**.

All **scripts** are treated the same and run in the order in which they appear in the page.

Because the **DOM tree** gets parsed from top to bottom, the **script** element has access to all of the **DOM elements** that were already parsed.

```
<!DOCTYPE html>
<html>
<body>
  <h1>Example</h1>
  <script>
    console.log("inline 1");
  </script>
  <script src="external1.js"></script>
  <script>
    console.log("inline 2");
  </script>
  <script src="external2.js"></script>
  <script>
    console.log("inline 3");
  </script>
</body>
</html>
```

Scripts Location in the DOM

If we really want to have the **script** elements at the top of the **DOM**, we need to ensure that all of the code that relies on the **DOM** runs after the **DOMContentLoaded** event gets fired.

If we can guarantee that the **scripts** will appear toward the end of the **document** after the **DOM elements**, we can avoid following the whole **DOMContentLoaded** approach described.

Scripts Location in the DOM

When a **script** element is being parsed, the browser stops *everything* else on the page from running while the code is executing.

If we have a really **long-running script** or an **external script** takes its time in getting downloaded, the **DOM** can be only partially parsed at this point and then the page will look incomplete in addition to being frozen.

This is another reason besides the **DOM** access why we recommend having the **scripts** live toward the bottom of the page.

Script Elements

If a **script** element is being parsed, it could block the browser from being responsive and usable.

The **async** attribute allows a script to run asynchronously.

```
<script async src="someRandomScript.js"></script>
```

By setting this **attribute** on a **script** element, the **script** will run whenever it is able to, but **it won't block the rest of the browser from doing its tasks.**

Script Elements

The **scripts** marked as **async** will not always run in order.

We could have a case where several **scripts** marked as **async** will run in an order different from what was specified in the **markup**.

The only guarantee we have is that these **scripts** marked with **async** will start running at some point before the **load** or **DOMContentLoaded** event gets fired.

Script Elements

Scripts marked with **defer** run in the order in which they were defined, but they only get executed at the end, just a moment before the **DOMContentLoaded** event gets fired.

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

Script Elements

In this example, **scripts** will execute in this order: **inline1**, **external2**, **inline2**, **inline3**, **external1**, and **external3**.

Because **external1** and **external3** are marked as **defer**, they will appear at the end, despite being declared in different locations in the **markup**.

```
<!DOCTYPE html>
<html>
<body>
  <h1>Example</h1>
  <script defer src="external1.js"></script>
  <script>
    console.log("inline 1");
  </script>
  <script src="external2.js"></script>
  <script>
    console.log("inline 2");
  </script>
  <script defer src="external3.js"></script>
  <script>
    console.log("inline 3");
  </script>
</body>
</html>
```

Making HTTP Requests in JavaScript

Making HTTP Requests in JavaScript

The **HTTP protocol** provides a common language that allows the browser to communicate with all the **servers** in Internet.

The requests the browser makes using the **HTTP protocol** are known as **HTTP requests**, and these requests go well beyond simply loading a new page as we are navigating.

A common set of use cases can be revolved around updating an existing page with **data** resulting from an **HTTP request**.

Making HTTP Requests in JavaScript

A page might not have initially this data, but it will be information the browser will request as part of we interacting with the page.

Then the **server** will respond with the **data** and have the page update with that information.

To **asynchronously request and process data** from a **server** without requiring a page navigation/reload, we'll use **AJAX** (stands for **Asynchronous JavaScript and XML**).

An Example

Let's have some **JavaScript** that makes an **HTTP request** to a **service** (**ipinfo.io**) that returns a whole bunch of **data** about our connection.

Using **JavaScript**, we process all that returned **data** in order to get the IP address that we so will display here.



185.202.220.65

(This is your IP address...probably :P)

Meet the Fetch API

The newest block for making **HTTP requests** is the **fetch API**.

To use **fetch** in its most basic form, all we need to do is provide the **URL** to send our **request** to.

Once the **request** has been made, a **response** will be returned that we can then process.

Meet the Fetch API

Once we send the **request** to the URL **ipinfo.io/json**, the **service** running on **ipinfo.io** will send us some **data**.

Because the **response** returned by **fetch** is a **promise**, these **then** blocks are part of **how promises work asynchronously** to allow us to process that **data**.

```
{
  "ip": "66.87.125.72",
  "hostname": "ip-66-87-125-72.spfdma.spcsdns.net",
  "city": "Springfield",
  "region": "Massachusetts",
  "country": "US",
  "loc": "42.1015,-72.5898",
  "org": "AS10507 Sprint Personal Communications Systems",
  "postal": "01101",
  "timezone": "America/New_York"
}
```


Meet the Fetch API

Getting back to our example, in the first **then** block, we specify that we want the raw **JSON data** that the **fetch** call returns.

In the next **then** block (called after the previous one completes), we process the returned **data** to read the property that will give us the IP address and then printing it to the console.

```
fetch("https://ipinfo.io/json")  
  .then(function (response) {  
    return response.json();  
  })  
  .then(function (json) {  
    console.log(json.ip);  
  })
```

Meet the Fetch API

Let's go ahead and add the few missing details to get a page looking like our example.

Also, instead of printing the IP address to the console, we're instead displaying the IP address inside the **ipText** paragraph element.



185.202.220.65

(This is your IP address...probably :P)

```
<!DOCTYPE html>
<html>
<head>
  <title>Display IP Address</title>
  <style>
    body {
      background-color: #FFCC00;
    }
    h1 {
      font-family: sans-serif;
      text-align: center;
      padding-top: 140px;
      font-size: 60px;
      margin: -15px;
    }
    p {
      font-family: sans-serif;
      color: #907400;
      text-align: center;
    }
  </style>
</head>
```

```
<body>
  <h1 id="ipText"></h1>
  <p>( This is your IP address...probably :P )</p>
  <script>
    fetch("https://ipinfo.io/json")
      .then(function (response) {
        return response.json();
      })
      .then(function (myJson) {
        document
          .querySelector("#ipText")
          .innerHTML = json.ip;
      })
      .catch(function (error) {
        console.log("Error: " + error);
      });
  </script>
</body>
</html>
```

Meet the XMLHttpRequest Object

The other (more traditional) option for allowing us to send and receive **HTTP requests** is the **XMLHttpRequest** object.

- Send a **request** to a **server**.
- Check on the **status** of a **request**.
- Retrieve and parse the **response** from the **request**.
- Listen for the **onreadystatechange** (to react to **status** of the **request**).

Creating the Request

First, we're going to do is initialize the **XMLHttpRequest** object.

Then, the **xhr** variable will be the *gateway* to all the properties and methods the **XMLHttpRequest** object provides for allowing us to make **HTTP requests**.

The **open** method is what allows us to specify the details of the **request** we would like to make.

```
let xhr = new XMLHttpRequest();  
xhr.open('GET', "https://ipinfo.io/json", true);
```

Creating the Request

In the **open** method, first argument specifies which **HTTP method** to use to process the **request** where the values we can specify are **GET**, **PUT**, **POST**, and **DELETE**.

In this case, we are interested in **receiving information**, so this argument is going to be **GET**.

Next, we specify the **URL** to send the **request** to and, for this example, the path we will specify is **ipinfo.io/json**.

The last argument specifies **whether we want the request to run asynchronously**.

Sending the Request

We've initialized the **XMLHttpRequest** object and constructed the **request**, but we haven't sent it out yet.

Then, the **send** method is responsible for sending the **request**.

If we set the **request** to be **asynchronous**, the **send** method immediately returns and the rest of the code continues to run.

```
let xhr = new XMLHttpRequest();  
xhr.open('GET', "https://ipinfo.io/json", true);  
xhr.send();
```

```
// do something
```

Asynchronous Calls and Events

We have the **readystatechange** event that is fired by the **XMLHttpRequest** object whenever the **request** hits any **notification**.

We need to listen for this **event** and call the **processRequest** function (the **event handler**) when the event is overheard.

```
let xhr = new XMLHttpRequest();  
xhr.open('GET', "https://ipinfo.io/json", true);  
xhr.send();
```

```
xhr.addEventListener("readystatechange", processRequest, false);
```


Processing the Request

We have the **event listener** all ready, and all we need is the **event handler** (the **processRequest** function) where we can add the code to parse the result of the **HTTP request**.

```
let xhr = new XMLHttpRequest();  
xhr.open('GET', "https://ipinfo.io/json", true)  
xhr.send();
```

```
xhr.onreadystatechange = processRequest;
```

```
function processRequest(e) {  
    // do something  
}
```

Processing the Request

To do this, we use the **readystatechange** event being tied to the **XMLHttpRequest** object's **readyState** property.

This **readyState** property chronicles the path the **HTTP request** takes, and each change in its value results in the **readystatechange** event getting fired.

Value	State	Description
0	UNSENT	The open method hasn't been called yet
1	OPENED	The send method has been called
2	HEADERS_RECEIVED	The send method has been called and the HTTP request has returned the status and headers
3	LOADING	The HTTP request response is being downloaded
4	DONE	Everything has completed

Processing the Request

The **status code** we care about with **HTTP requests** is **200** because it is returned by the **server** when the **request** was successful.

Only if the request has completed (**readyState == 4**) and is successful (**status == 200**) the request did what we wanted it to do.

```
function processRequest(e) {  
    if (xhr.readyState == 4 && xhr.status == 200) {  
        // do something  
    }  
}
```

Processing the Request

To convert the current data returned as a **JSON-like string** into an actual **JSON object**, we pass in the result of **xhr.responseText** into the **JSON.parse** method.

```
function processRequest(e) {  
    if (xhr.readyState == 4 && xhr.status == 200) {  
        let response = JSON.parse(xhr.responseText);  
  
        document.querySelector("#ipText").innerHTML = response.ip;  
    }  
}
```

Storing Data using Web Storage API

Why Storage for the Web

Let's say we have a simple website that allows us to maintain a *to-do list*.

If we navigate to another page or close the tab this page was on, default behavior when we return to this page would be to empty the *to-do list*.

My To-Do List

1. Finish local storage article.
2. Mow the lawn.
3. Save the world.

SaveReset

My To-Do List

SaveReset

Why Storage for the Web

Web pages do **not persist data by default** but we have a many approaches we can take for solving this problem.

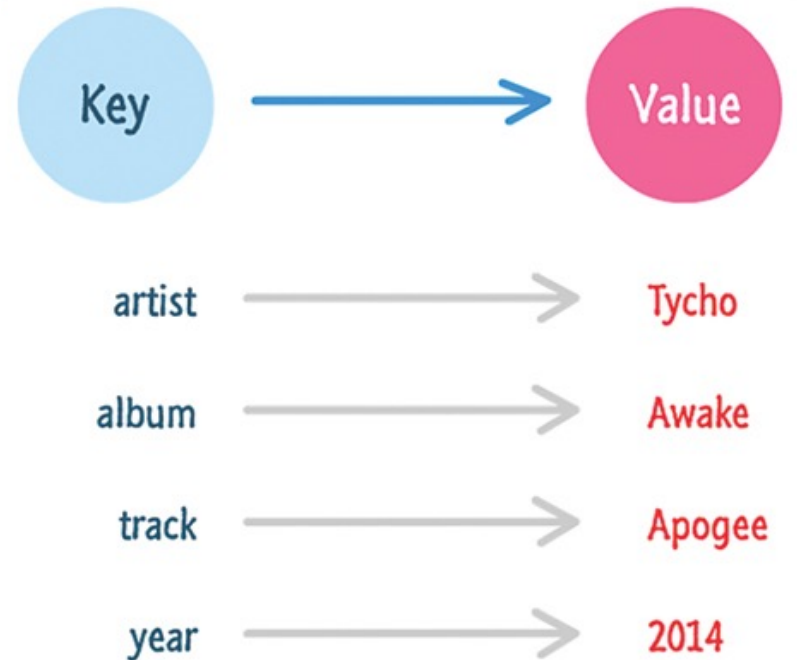
We're going to look at a way of **storing data** by relying on what is known as the **Web Storage API**.

This **API** allows us to write just a few lines of **Javascript** code to handle many storage situations.

Adding Data

The method we use for adding data lives off of the **localStorage** object, it is called **setItem** and it takes the **key** and **value** arguments.

```
localStorage.setItem("artist", "Tycho");  
localStorage.setItem("album", "Awake");  
localStorage.setItem("track", "Apogee");  
localStorage.setItem("year", "2014");
```



Adding Data

If we specify an existing **key**, the existing **value** the key is pointing to will be overwritten with the new value.

```
// overwriting some data  
localStorage.setItem("track", "Apogee");  
localStorage.setItem("track", "L");
```

Because **all keys and values must be strings**, to store a **custom object** for retrieval later, basically we need to use **string values** and rely heavily on methods like **toString()** and **JSON.stringify()** to ensure we're storing any **complex data** in the form of a **string** we can un-stringify later.

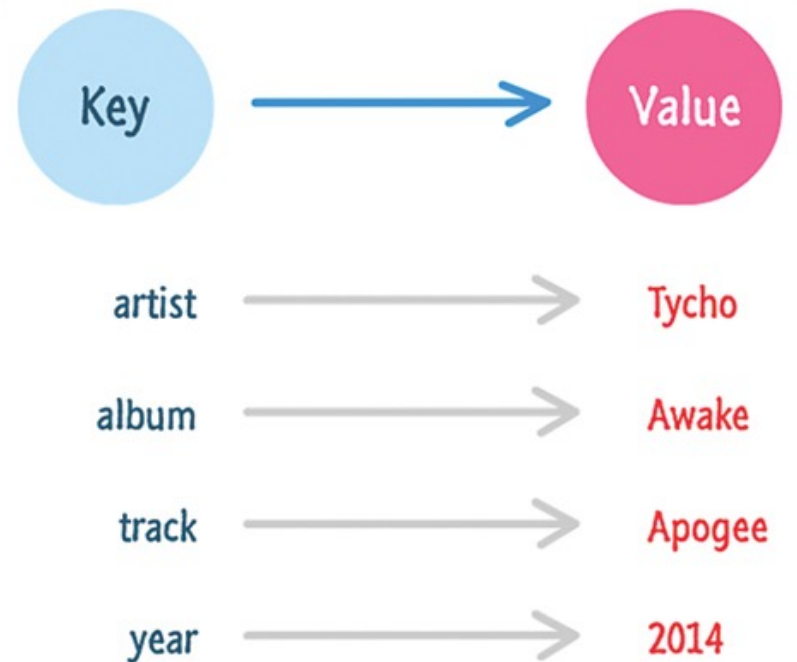
Retrieving Data

To retrieve any data stored in the **localStorage** object, we use the **getItem** method.

This method takes only the **key** as an argument, and it returns the **value** associated with that **key**.

If the **key** we pass as an argument in does not exist, a value of **undefined** is returned instead.

```
let artist = localStorage.getItem("artist");  
console.log(artist); // will print out 'Tycho'
```



A Shortcut for Adding and Removing

We can bypass **getItem** or **setItem** methods by using this notation for setting and retrieving data.

```
// storing data  
localStorage["key"] = value;
```

```
// retrieving data  
let data = localStorage["key"];
```

Removing Data

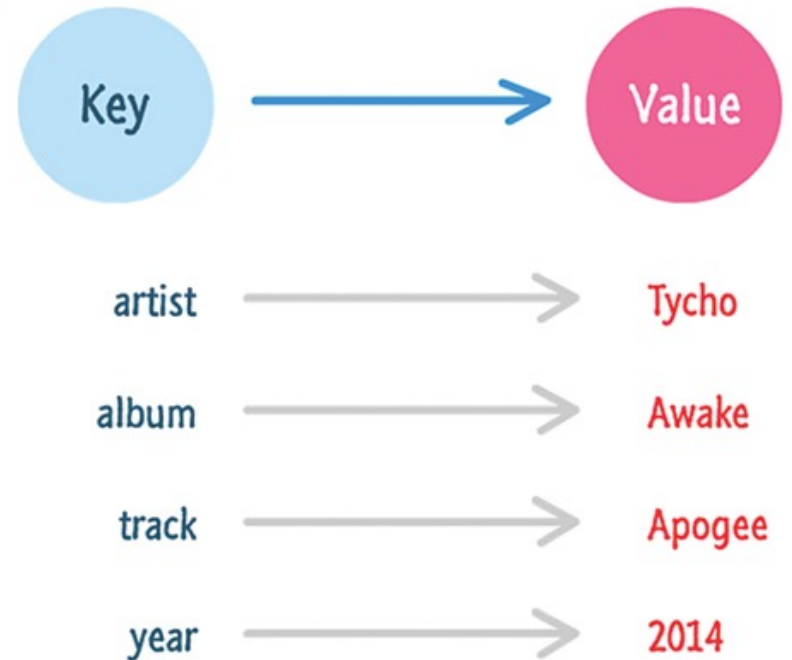
We can remove all data from the **local storage**, or we can selectively remove a **key** and **value** pair individually.

To remove any data from the **local storage**, we call the **clear** method on the **localStorage** object.

localStorage.clear();

To remove only select key and value entries from **local storage**, we use **removeItem()** and pass in the key name associated with the data we wish to remove.

localStorage.removeItem("year");



What about Session Storage?

When we store data using **local storage**, the data is available across multiple **browsing sessions**.

If we close the browser and come back later, any data we had stored will still be accessible by the page.

When we are storing data via **session storage**, the data is **only available for that browsing session**.

There is **no long-term persistence** so if we close the browser, come back a few moments later, we'll find that all of the data stored in the **sessionStorage** object is gone.

What about Session Storage?

For the most part, we use **local storage** and **session storage** the same way, so everything we saw for the **localStorage** object applies to the **sessionStorage** object as well.

The way we add, update, and remove items is even unchanged.

// adding items

```
sessionStorage.setItem("artist", "Tycho");  
sessionStorage.setItem("album", "Awake");  
sessionStorage.setItem("track", "Apogee");  
sessionStorage.setItem("year", "2014");
```

// removing item

```
sessionStorage.removeItem("album");
```