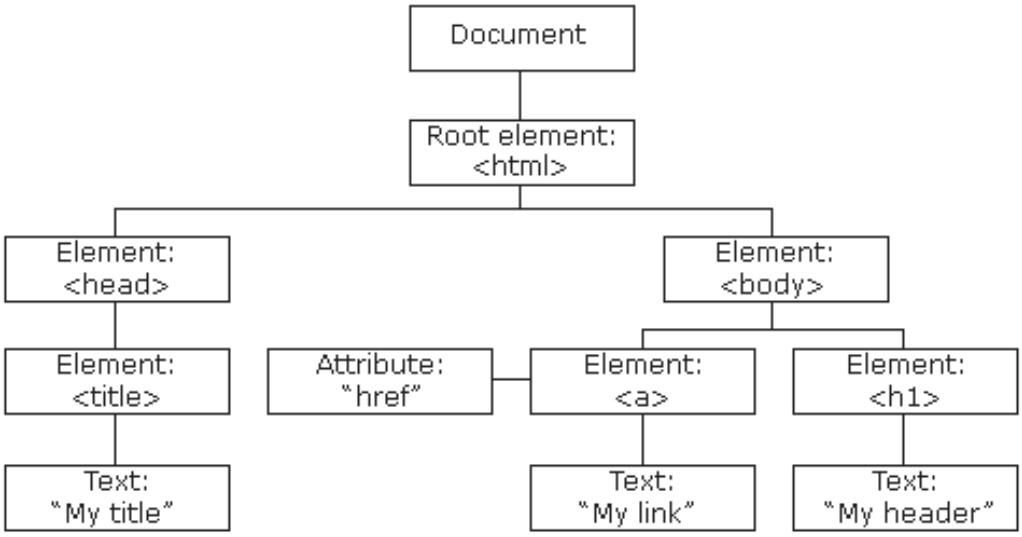
2 Document Object Model (DOM) Basics

With objects in mind, we will now turn our attention to HTML's **Document Object Model (DOM)**.

> **Document Object Model (from MDN)**
> The **Document Object Model (DOM)** is a programming interface for web documents (HTML). It represents the page so that programs (JavaScript) can change the document structure, style, and content. The DOM represents the document as nodes and objects; that way, programming languages can interact with the page.



The HTML DOM Tree of Objects

HTML's DOM provides us a structured system for JavaScript to access and manipulate HTML objects. This include, but not limited to, manipulating text, adding HTML elements, removing HTML elements, programming actions of buttons, and specifying mouse actions.

2.1 Accessing HTML Elements

HTML DOM provides methods for accessing elements:

1. `getElementsByTagName()`
2. `getElementsByClassName()`
3. `querySelector()` and `querySelectorAll()`
4. `getElementById()` and `getElementsByName()`

2.1.1 getElementsByTagName()

Consider the following setup:

```
<body>
    <p> Top bread slice </p>
    <p> Filling </p>
    <p> Bottom bread slice </p>
</body>
```

If we'd like to access the three paragraph <p> tags in our file, then we can use the getElementsByTagName() method:

```
<body>
    <p> Top bread slice </p>
    <p> Filling </p>
    <p> Bottom bread slice </p>
</body>
```

```
let paras=document.getElementsByTagName("p");
console.log(paras);
```

Console Output:

```
>>> [p, p, p]
```

Notice that the console outputs an **array** with three paragraph (p) objects, one for each <p> tag.

To read the contents within the tag, we can use the innerHTML property:

```
<body>
    <p> Top bread slice </p>
    <p> Filling </p>
    <p> Bottom bread slice </p>
</body>
```

```
let paras=document.getElementsByTagName("p");

for (let i = 0; i < paras.length; ++i){
    console.log(paras[i].innerHTML);
}
```

Console Output:

```
>>> Top bread slice
>>> Filling
>>> Bottom bread slice
```

Important to also note is the outerHTML property:

```
<body>
    <p> Paragraph 1 </p>
    <p> Paragraph 2 </p>
    <p> Paragraph 3 </p>
</body>
```

```
let paras=document.getElementsByTagName("p");

for (let i = 0; i < paras.length; ++i){
    console.log(paras[i].outerHTML);
}
```

Console Output:

```
>>> <p>Top bread slice</p>
>>> <p>Filling</p>
>>> <p>Bottom bread slice</p>
```

This gives us more control over exactly what we are accessing in the HTML document through the DOM.

2.1.2 `getElementsByClassName()`

A similar effect can be achieved through accessing the **class** names of HTML objects:

```
<body>
    <p class="bread"> Top bread slice </p>
    <p class="pb">    Peanut butter </p>
    <p class="bread"> Bottom bread slice </p>
</body>
```

```
let bread = document.getElementsByClassName(
    "bread");
for (let i = 0; i < bread.length; ++i){
    console.log(bread[i].innerHTML);
}
```

Console Output:

```
>>> Top bread slice
>>> Bottom bread slice
```

2.1.3 `querySelector()` and `querySelectorAll()`

With classes comes CSS rule sets. If we are targeting specific HTML elements under certain desired CSS rule sets, `querySelector()` and `querySelectorAll()` come in handy.

- `querySelector()`: returns the first DOM element that matches the specified *selector*.
- `querySelectorAll()`: returns an array of all the DOM elements that match the specified *selector*.

Note that the *selector* argument is written in CSS formatting.

The following is an example with `querySelector()`. Notice that even though the bottom piece of bread is also has the CSS selector string `"p.bread"`, since `querySelector()` only returns the first matched DOM element, it finds the top piece of bread.

```
<body>
    <p class="bread"> Top bread slice </p>
    <p class="pb">    Peanut butter </p>
    <p class="bread"> Bottom bread slice </p>
</body>
```

```
// Only gets one value
let bread = document.querySelector("p.bread");

console.log(bread.innerHTML);
```

Console Output:

```
>>> Top bread slice
```

Here `querySelectorAll()` will return an array with all DOM elements with the CSS selector string `"p.bread"`.

```
<body>
    <p class="bread"> Top bread slice </p>
    <p class="pb">    Peanut butter </p>
    <p class="bread"> Bottom bread slice </p>
</body>
```

```
let bread = document.querySelectorAll(
    "p.bread");
for (let i = 0; i < bread.length; ++i){
    console.log(bread[i].innerHTML);
}
```

Console Output:

```
>>> Top bread slice
>>> Bottom bread slice
```

`getElementByClassName()` vs `querySelectorAll()`

You may notice that `getElementByClassName()` vs `querySelectorAll()` behave in very similar ways, and you are correct—both will search through your HTML document for all matching *class name* selectors strings.

There is one major difference:

- `getElementByClassName()` returns a **dynamic/live** list of DOM elements.
- `querySelectorAll()` returns a **static** list of DOM elements.

The differences between **static** and **dynamic** lists are not too important for now as we're only starting out JavaScript. However, being exposed to this idea will help us when we see this idea again down the list. Let's look at what **static** and **dynamic** objects mean, and see the difference in an example.

> **Dynamic vs Static Objects**
> **Dynamic / Live** objects are updated immediately when the DOM changes.
> **Static** objects are not updated even when the DOM changes, and will retain its values at the time of the query.

Let's consider our HTML sandwich again. Both `querySelectorAll()` and `getElementsByClassName()` return a list size of 2, which is expected since there are 2 elements with class name `"bread"`.

```
<body>
    <p class="bread"> Top bread slice </p>
    <p class="pb">    Peanut butter </p>
    <p class="bread"> Bottom bread slice </p>
</body>
```

```
let q = document.querySelectorAll("p.bread");
let g = document.getElementsByClassName(
    "bread");

console.log(`There are ${q.length} in query`);
console.log(`There are ${g.length} in get`);
```

Console Output:

```
>>> There are 2 in query
>>> There are 2 in get
```

Let's now dynamically add a new paragraph element with class name `"bread"` in JavaScript (we will look at how this works in more detail in a later lecture). Note that this new paragraph element is not in the original HTML, but a newly created element in JavaScript. There are now 3 `"bread"` elements, but list returned by `querySelectorAll()` still says there are 2 elements since it is a **static** list; the list returned by `getElementsByClassName()` will instead show there are 3 elements as it is a **dynamic** list.

```
<body>
    <p class="bread"> Top bread slice </p>
    <p class="pb">    Peanut butter </p>
    <p class="bread"> Bottom bread slice </p>
</body>
```

```
let q = document.querySelectorAll("p.bread");
let g = document.getElementsByClassName(
    "bread");
// create new p element
let newP = document.createElement("p");
newP.innerHTML = "new bread";
newP.className = "bread";
document.body.appendChild(newP);
console.log(`There are ${q.length} in query`);
console.log(`There are ${g.length} in get`);
```

Console Output:

```
>>> There are 2 in query
>>> There are 3 in get
```

2.1.4 `getElementsByName()`

Much similar to `getElementsByClassName()`, this will search through the DOM for all elements of matching *name* selector strings. This will be helpful when we start working with forms.

```
<body>
    <p name="bread"> Top bread slice </p>
    <input type="text" name="filling"
        value="Peanut butter">
    <p name="bread"> Bottom bread slice </p>
</body>
```

```
let filling = document.getElementsByName(
    "filling");

// Only one element with name="filling"
console.log(filling[0].value);
```

Console Output:

```
>>> Peanut butter
```

2.1.5 `getElementById()`

Finally, we can access elements by their unique identifying *id*. Since the *id* field should be unique identifiers, it will return one and only one DOM element of matching selector string.

```
<body>
    <p id="topBread"> Top bread slice </p>
    <p id="filling"> Peanut Butter </p>
    <p id="botBread"> Bottom bread slice </p>
</body>
```

```
let filling = document.getElementById(
    "filling");

// Returns an element, not array
console.log(filling.innerHTML);
```

Console Output:

```
>>> Peanut Butter
```

`class` vs `name` vs `id`

You might be a little overwhelmed with all the possible DOM element accessing methods—they all seem to be doing the same thing. However keep in mind that **class**, **name**, and **id** tag attributes server very different purposes in HTML. As a brief refresher:

- **class** is used to specify CSS classes for CSS formatting.
- **name** is used to identify elements in a forms (`input` tags).
- **id** is used to *uniquely* identify one and only one single HTML element.

2.2 Modifying HTML Elements

Accessing DOM elements and checking their `innerHTML` is fun and all, but we want to manipulate the HTML page, not just read it. The way we do this is quite straightforward: simply set a new value to the `innerHTML` (and to a lesser extent `outerHTML`) properties.

2.2.1 `innerHTML` and `outerHTML` Properties

Both `innerHTML` and `outerHTML` are properties of DOM elements, the difference being whether or not the string returned will contain the HTML tags.

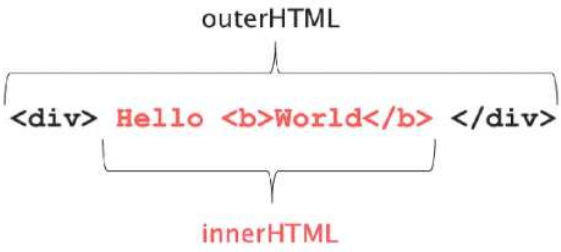


Image courtesy of Ravi Sah (https://www.ravisah.in/).

Example 2.1 *Change the sandwich filling to jelly.*

```
<body>
    <p class="bread">   White bread   </p>
    <p class="filling"> Peanut Butter </p>
    <p class="bread">   White bread   </p>
</body>
```

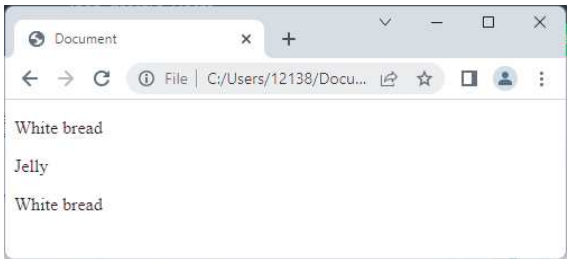
Here we want to change the filling of our HTML sandwich from "peanut butter" to "jelly". We can access the `filling` class with `getElementsByClassName()`.

```
let filling = document.getElementsByClassName("filling");
```

We can now access the `innerHTML` of the filling element and set it to "jelly". Take careful note that `getElementByClassName()` returns an array of elements, thus we need to index it.

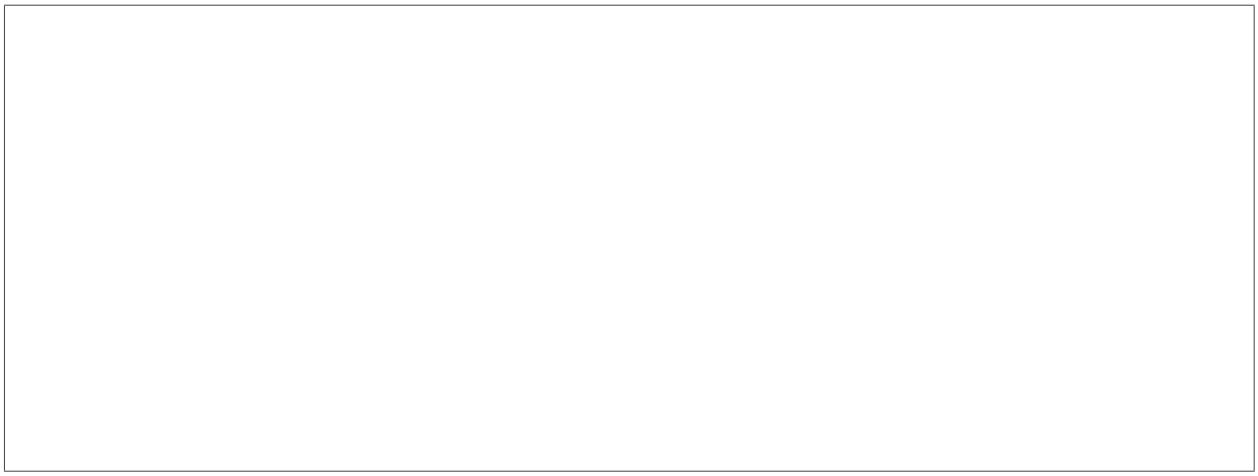
```
let filling = document.getElementsByClassName("filling");
filling[0].innerHTML = "Jelly";
```

With this, we see that our HTML webpage no longer has "Peanut Butter" as the middle text, but now reads "Jelly":



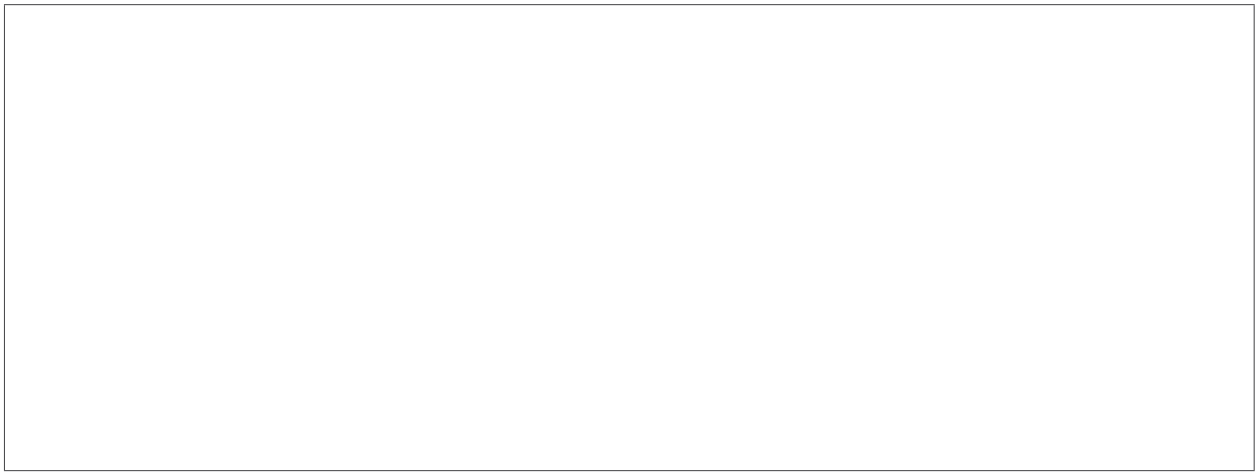
Exercise 2.1 *Change the sandwich bread to wheat bread.*

```
<body>
    <p class="bread"   id="topBread"> White bread   </p>
    <p class="filling" id="filling">  Peanut Butter </p>
    <p class="bread"   id="botBread"> White bread   </p>
</body>
```



Exercise 2.2 *Change only the top slice of bread to wheat bread.*

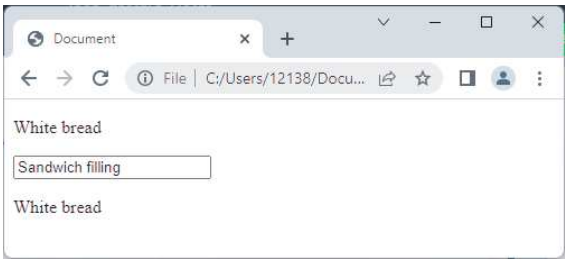
```
<body>
    <p class="bread"   id="topBread"> White bread   </p>
    <p class="filling" id="filling">  Peanut Butter </p>
    <p class="bread"   id="botBread"> White bread   </p>
</body>
```



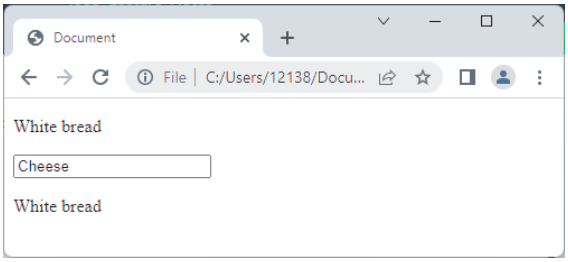
2.2.2 Input Fields and `value` Property

Consider the following sandwich setup with an input field instead of a paragraph tag. This allows the user to input their own choice of sandwich filling.

```
<body>
    <p class="bread">   White bread   </p>
    <input type="text" name="filling" value="Sandwich filling">
    <p class="bread">   White bread   </p>
</body>
```



Let's say the user types in their choice of sandwich filling:



How would we access the contents of this input text field? Notice that the `input` is an *inline* element—it lacks the ending tag that paragraphs do (`<p>` and `</p>`. This means the `input` element does not have an `innerHTML` nor `outerHTML` property. Instead, the `input` element has the `value` property instead—we will access this property instead.

```
// Acceses the input element via name
let filling = document.getElementsByName("filling");

console.log(filling[0].value);
```

Console Output (theoretically):

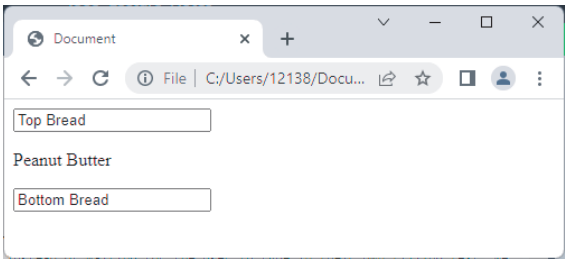
```
>>> Cheese
```

There is a slight issue here right now. Our JavaScript script will run immediately when the webpage loads up, therefore grabbing the default value of "Sandwich filling" instead of waiting for the user to type in their own filling text. We will remedy this with **buttons** and **event listeners** in the next section.

For the following exercise, work under the pretense that our JavaScript script will run only after the user has finished typing into the two input text fields.

Exercise 2.3 *Write a JavaScript function that will check whether or not the two pieces of bread are exactly the same.*

```
<body>
    <input type="text" name="bread" value="Top Bread">
    <p class="filling"> Peanut Butter </p>
    <input type="text" name="bread" value="Bottom Bread">
</body>
```




```
// Takes no inputs
// Outputs true if top and bottom bread are the same
// Outputs false otherwise
function checkBread(){









}
```

2.3 Event Listeners Basics

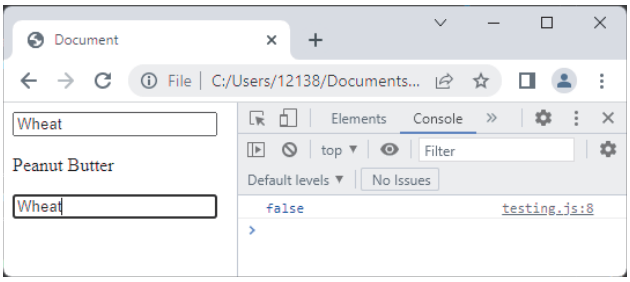
Consider our last exercise on checking if the top and bottom bread are the same:

```
<body>
    <input type="text" name="bread"
        value="Top Bread">
    <p id="filling"> Peanut Butter </p>
    <input type="text" name="bread"
        value="Bottom Bread">
</body>
```

```
function checkBread(){
    const bread = document.getElementsByName(
        "bread");
    const topBread = bread[0].value;
    const botBread = bread[1].value;
    return topBread === botBread;
}
```

The issue we saw is that the JavaScript script will immediately run, meaning the moment the HTML loads, our `checkBread()` function will run, grab the value from the two input fields, and compare them, all before the user can do anything:

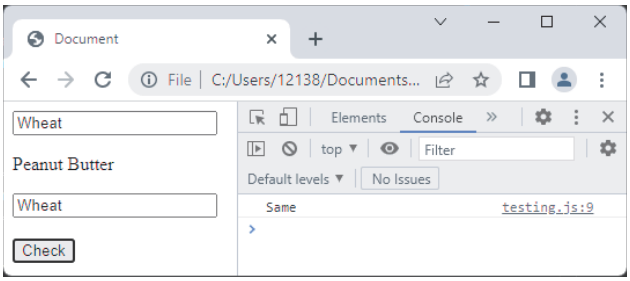
```
// In main body of the .js file
// This runs immediately, will not run even after we change the bread
console.log(checkBread());
```



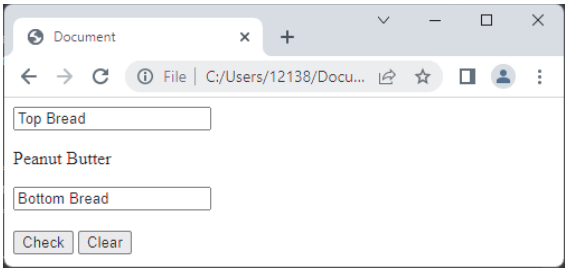
What we would like to happen is for the `checkBread()` function to run only after the user clicks some button. Let's add a button to our HTML page. With this button, we can attach function to the `"click"` event for the button's **event listener**. Let's call this function `checkClick()`:

```
<body>
    <input type="text" name="bread"
        value="Top Bread">
    <p id="filling"> Peanut Butter </p>
    <input type="text" name="bread"
        value="Bottom Bread">
    <br> <br>
    <button> Check </button>
</body>
```

```
function checkBread(){
    const bread = document.getElementsByName("
        bread");
    const topBread = bread[0].value;
    const botBread = bread[1].value;
    return topBread === botBread;
}
// Called when button is clicked
function checkClick(){
    if (checkBread()) console.log("Same");
    else console.log("Different");
}
const btn = document.getElementsByTagName("
    button");
btn[0].addEventListener("click", checkClick);
```



Exercise 2.4 *Add and implement the functionality for a* `clear` *button.*



.html

```
<body>
    <input type="text" name="bread" value="Top Bread">
    <p id="filling"> Peanut Butter </p>
    <input type="text" name="bread" value="Bottom Bread">
    <br> <br>
    <button id ="checkBtn"> Check </button>
    <button id ="clearBtn"> Clear </button>



<body>
```

.js

```
// Takes no inputs
// Clears both bread input fields
function clearClick(){









}

const clearBtn = document.getElementById("clearBtn");

clearBtn.
```

2.4 Styling Elements

So far our HTML pages has been quite plain in its styling, pretty much a toast sandwich. To add flare and personality, we can certainly add a `.css` file for **class styling**.

However, a `.css` file is for *initial styling* — how the webpage looks like when it's first loaded in. But now we have JavaScript, we have the ability to change the HTML page dynamically. Not just the text and contents of the page, but the styling as well.

2.4.1 Styling with CSS (brief review)

`.css` files allows us to specify the styling for **tags**, **.classes**, or `#ids`:

```
/* style applies to every <p> */
p{
    /* property styling */
}

/* style applies to every class "bread" */
.bread{
    /* property styling */
}

/* style applies the element with id "topBread" */
#topBread{
    /* property styling */
}

/* style applies to every <p> of class "filling" */
p.filling{
    /* property styling */
}
```

With `.css`, we can style specific properties of our HTML elements:

```
button.myBtn{
    color: blue;
    background-color: #008800;
    font-size: 48px;
    height: 128px;
    width: 256px;
    margin: 8px;
    padding: 8px;
    border: 4px solid red;
    border-radius: 16px;
}
```

For the full syntax, functionality, and properties of CSS, please refer to MDN's CSS documentation.

2.4.2 Styling with DOM

Styling with the DOM is done through the `.style` of DOM elements. There are slight differences between `.css` syntax and `element.style` syntax, so please make sure to take careful notice:

```
const btn = document.getElementById("btn");
btn.style.color = "blue";
btn.style.backgroundColor = "#008800";
btn.style.fontSize = "48px";
btn.style.height = "128px";
btn.style.width = "256px";
btn.style.margin = "8px";
btn.style.padding = "8px";
btn.style.border = "4px solid red";
btn.style.borderRadius = "16px";
```

Consider the following HTML webpage:

```
<body>
    <p class="bread">Bread</p>
    <p id="filling"> </p>
    <p class="bread">Bread</p>
    <button id="strBtn">Strawberry</button> <button id="graBtn">Grape</button> <button id="
        orgBtn">Orange</button> <button id="bluBtn">Blueberry</button>
</body>
```

Let's say our font here is too small; we'd like to change everything to size 16 font. Let's first change our <p>'s:

Example 2.2 *Using JavaScript, change the font size of <p>'s to size 16 font.*

```
const p = document.getElementsByTagName("p");
for (let i = 0; i < p.length; ++i){
    p[i].style.fontSize = "16px";
}
```

Exercise 2.5 *using JavaScript, change the font size of each button to size 16 font.*

If we use DOM styling in conjunction with **event listeners**, we're able to change the HTML styling in response to user actions.

Example 2.3 *Change the background color of the filling* `<p>` *according to its filling*

```
// get filling DOM element
const fill = document.getElementById("filling");

// click functions for buttons
function strClick(){
    fill.innerHTML = "Strawberry";
    fill.style.backgroundColor = "red";
}
function graClick(){
    fill.style.backgroundColor = "purple";
    fill.innerHTML = "Grape";
}
function orgClick(){
    fill.innerHTML = "Orange";
    fill.style.backgroundColor = "orange";
}
function bluClick(){
    fill.innerHTML = "Blueberry";
    fill.style.backgroundColor = "blue";
}

// attach the functions to the buttons
document.getElementById("strBtn").addEventListener("click", strClick);
document.getElementById("graBtn").addEventListener("click", graClick);
document.getElementById("orgBtn").addEventListener("click", orgClick);
document.getElementById("bluBtn").addEventListener("click", bluClick);
```

This will work as is, however certainly not optimal in terms of organizing our code. We will look at how we can use loops and anonymous functions to make this more efficient.

2.4.3 Styling multiple DOM Elements with Loops (`querySelectorAll.forEach()`)

This definitely works, but we run into a familiar issue: there is a lot of repetition. There are four `click()` functions that all have the same logic of assigning values to `.innerHTML` and `.style.backgroundColor`. Each of these `click()` functions are then attached to a button through `.addEventListener`.

The remedy to all this repetition is **loops**. First, let's put put the functions in an array to it's iterable.

```
// store the functions in an array
const clickFunctions = [strClick, graClick, orgClick, bluClick];
```

We also need the buttons to be iterable. Here we'll going to specifically use `querySelectorAll()` in anticipation for the upcoming section:

```
// store the functions in an array
const clickFunctions = [strClick, graClick, orgClick, bluClick];

// buttons is an array
const buttons = document.querySelectorAll("button");
```

Now we can use a loop to iterate through the buttons and functions at the same time

```
// store the functions in an array
const clickFunctions = [strClick, graClick, orgClick, bluClick];

// buttons is an array
const buttons = document.querySelectorAll("button");

// loop through and add flick functions to the buttons
for (let i = 0; i < buttons.length; ++i){
    buttons[i].addEventListener("click", clickFunctions[i]);
}
```

This alleviates the repetition of all the `.addEventListeners`. Nice.

We still have a some more repetition though:

```
// get filling DOM element
const fill = document.getElementById("filling");

// click functions for buttons
function strClick(){
    fill.innerHTML = "Strawberry";
    fill.style.backgroundColor = "red";
}
function graClick(){
    fill.style.backgroundColor = "purple";
    fill.innerHTML = "Grape";
}
function orgClick(){
    fill.innerHTML = "Orange";
    fill.style.backgroundColor = "orange";
}
function bluClick(){
    fill.innerHTML = "Blueberry";
    fill.style.backgroundColor = "blue";
}
```

Here we observe that these use of the `click()` functions are very specific — they serve one button and one button only, never to be used in any other contexts. If there are functions that are uniquely tied to a single button's use, we'd like to use **anonymous functions** to remedy this.

Let's look at one `click()` functions:

```
// get filling DOM element
const fill = document.getElementById("filling");

function bluClick(){
    fill.innerHTML = "Blueberry";
    fill.style.backgroundColor = "blue";
}
```

The `fill.innerHTML` string we can get directly by accessing the button's `.innerHTML` since the buttons have the exact text that we want:

```
// button.innerHTML gives us the string we want
<button id="strBtn">Strawberry</button>
<button id="graBtn">Grape</button>
<button id="orgBtn">Orange</button>
<button id="bluBtn">Blueberry</button>
```

However, for `fill.style.backgroundColor`, we want to setup a simple data structure to access the colors. We can either use an **array** or an **object** (dictionary). As much as arrays are convenient, a dictionary here is more versatile for the purposes of mapping the fruit (key) to a color (value). Let's set this up:

```
// get filling DOM element
const fill = document.getElementById("filling");

// Dictionary of fruit to background color
const colorMap = {
    Strawberry: "red",
    Grape: "purple",
    Orange: "orange",
    Blueberry: "blue"
}

// Change this to anonymous functions
for (let i = 0; i < buttons.length; ++i){
    buttons[i].addEventListener("click", clickFunctions[i]);
}
```

Now in place of `clickFunctions[i])`, we can setup **anonymous functions**:

```
// Change this to anonymous functions
for (let i = 0; i < buttons.length; ++i){
    buttons[i].addEventListener("click", function(){
        // set text to the button's text
        fill.innerHTML = buttons[i].innerHTML;
        // set background color by looking up the color in the dictionary
        fill.style.backgroundColor = colorMap[buttons[i].innerHTML];
    });
}
```

With these loops, we've reduced our JavaScript implementation down to:

```
const fill = document.getElementById("filling");
const colorMap = {
    Strawberry: "red",
    Grape: "purple",
    Orange: "orange",
    Blueberry: "blue"
}
const buttons = document.getElementsByTagName("button");
for (let i = 0; i < buttons.length; ++i){
    buttons[i].addEventListener("click", function(){
        fill.innerHTML = buttons[i].innerHTML;
        fill.style.backgroundColor = colorMap[buttons[i].innerHTML];
    });
}
```

The "natural" next question is:

Can we do better?

Of which the answer is:

Yes, yes we can.

Notice that our `for` loop uses a numerical iterator `i`. This numerical iterator `i` is used to index our array of `buttons`.

However, the actual numerical value of `i` has no significance other than indexing. Its sole purpose is to iterate through an array of objects. If we want to iterate through a list of objects, it's more appropriate to use `for...of` loops (see Section 1.6.5) over numerical loops. Therefore the use of a `for` loop with `i` is not the best practice.

The repetitive action of iterating through a DOM objects and applying a function to each of the DOM objects (such as `.addEventListener`) is common enough that we have a specialized loop for it, called a `forEach` loop.

`Arrayl.forEach()` **(from MDN)**
The `forEach()` method executes a provided function once for each array element.

```
forEach(function(element) { ... })
forEach(function(element, index) { ... })
forEach(function(element, index, array){ ... })
forEach(function(element, index, array) { ... }, thisArg)
```

Let's see how we can implement this. First note that we're currently using a a `for` loop to iterate through an array of DOM objects.

```
const fill = document.getElementById("filling");
const colorMap = {
    Strawberry: "red",
    Grape: "purple",
    Orange: "orange",
    Blueberry: "blue"
}
const buttons = document.getElementsByTagName("button");
for (let i = 0; i < buttons.length; ++i){
    buttons[i].addEventListener("click", function(){
        fill.innerHTML = buttons[i].innerHTML;
        fill.style.backgroundColor = colorMap[buttons[i].innerHTML];
    });
}
```

We want to switch this out with a `forEach()`. Here we need to make two important notes:

- The `forEach()` will only work on a **static list**, meaning we **must** to use `querySelectorAll()` instead of `getElementsBy...()` (see Section 2.1.3).
- `forEach()` takes an **anonymous function** as its parameter.

Also, remember that `forEach()` is a method of a static array instead of a standalone structure:

```
document.querySelectorAll("button").forEach(function(element){
    // attach function to each button using addEventListener()
});
```

Here notice that our **anonymous function** as an argument `element`. This `element` argument passes in each individual object from the array returned from `querySelectorAll()` every iteration, effectively acting as our `buttons[i]` without the `i`.

Let's add in our original `for` loop logic, taking special care to replace each `buttons[i]` with `element`. And yes, this is a nested **anonymous function**. Welcome to JavaScript and *functional programming*.

```
// Consts
const fill = document.getElementById("filling");
const colorMap = {
    Strawberry: "red",
    Grape: "purple",
    Orange: "orange",
    Blueberry: "blue"
}

// Iterate with forEach and addEventListeners
document.querySelectorAll("button").forEach(function(btn){
    // attach function to each button using addEventListener()
    btn.addEventListener("click", function(){
        // set text to the button's text
        fill.innerHTML = btn.innerHTML;
        // set background color by looking up the color in the dictionary
        fill.style.backgroundColor = colorMap[btn.innerHTML];
    });
});
```

Admittedly, this is not the nicest looking JavaScript implementation. To a certain extent, this is a "it is what it is so accept it" kind of situation. However, there are some variations to this implementation, so you can find the one that suits your aesthetic and style the best.

Regardless which variation you choose, `forEach()` is preferred over using a numerically-indexed `for` loop as it provides a more rigorous structure to your JavaScript.

2.4.4 Arrow Notation for Anonymous Functions

Let's look at our nested anonymous functions:

```
document.querySelectorAll("button").forEach(function(btn){
    btn.addEventListener("click", function(){
        fill.innerHTML = btn.innerHTML;
        fill.style.backgroundColor = colorMap[btn.innerHTML]
    });
});
```

Maybe you don't like the syntax of all the `function()` declarations. JavaScript provides an alternative syntax for you to minimize the clutter with **arrow notation**. Here you can just remove the `function` keyword and add in an arrow `=>`:

```
document.querySelectorAll("button").forEach((button) => {
    element.addEventListener("click", () => {
        fill.innerHTML = element.innerHTML;
        fill.style.backgroundColor = colorMap[element.innerHTML];
    });
});
```

This is a commonly used syntax to declutter all the `function` text. It also distinctly differentiates between a **named function** — declared using the `function` keyword) — and an **anonymous function** — declared using the `(args...) => {...}` notation.

2.4.5 Additional `forEach()` Arguments

The function passed into the `forEach()` requires at least one argument, that of `element` (in this case, the elements in our array are all `buttons`). However this callback function can have additional arguments in case you need access to them. The full syntax for this callback function is:

```
function callback(element, index, array){};
```

Where the first argument `element` passes the the current iterated element of the `forEach()` function; the second argument `index` passes the index of the current iterated element; and the third argument `array` passes the original full array as an argument.

So, using the second `index` argument, we can technically implement:

```
// indexable arrays
const fruits = ["strawberry", "grape", "orange", "blueberry"];
const colors = ["red", "purple", "orange", "blue"];

document.querySelectorAll("button").forEach((btn, i) => {
    btn.addEventListener("click", () => {
        fill.innerHTML = fruits[i];
        fill.style.backgroundColor = colors[i];
    });
});
```

This does completely undermine the original use of `forEach()`, however if the index number is an absolute necessity in the implementation, `forEach()` will pass the index as a second argument for us to use.

2.5 Event Listeners: Callback and Other Events

Here we will look at some more nuanced details when working with event listeners in conjunction with `forEach()`.

2.5.1 Callback Functions

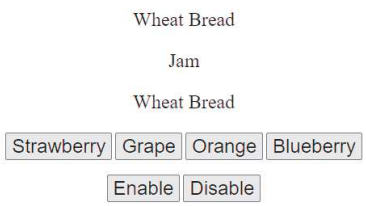
Consider again our sandwich jam implementation:

```
// Consts
const fill = document.getElementById("filling");
const colorMap = {
    Strawberry: "red",
    Grape: "purple",
    Orange: "orange",
    Blueberry: "blue"
}

// Iterate with forEach and addEventListeners
document.querySelectorAll("button").forEach((btn) => {
    // attach function to each button using addEventListener()
    btn.addEventListener("click", () => {
        // set text to the button's text
        fill.innerHTML = btn.innerHTML;
        // set background color by looking up the color in the dictionary
        fill.style.backgroundColor = colorMap[btn.innerHTML];
    });
});
```

Let's say now we want to add enable and disable functionalities to our jam buttons:

- Disable button will remove all clicking functionality from the jam buttons.
- Enable button will allow the jam buttons to change the jam.



This means we no longer can use an anonymous function for our click functionalities — the jam click functionality needs to be named in order for `addEventListener()` and `removeEventListener()` to add/remove the button's functionality.

Let's extract out the click anonymous functionality and name it:

```
// Consts
const fill = document.getElementById("filling");
const colorMap = {
    Strawberry: "red",
    Grape: "purple",
    Orange: "orange",
    Blueberry: "blue"
}
// function now named for add/remove event listener
function jamClick(){
    // set text to the button's text
    fill.innerHTML = btn.innerHTML;
    // set background color by looking up the color in the dictionary
    fill.style.backgroundColor = colorMap[btn.innerHTML];
}
// Iterate with forEach and addEventListeners
document.querySelectorAll("button.jam").forEach((btn) => {
    // attach function to each button using addEventListener()
    btn.addEventListener("click", jamClick);
});
```

With now a named function `jamClick()`, we can now implement our enable and disable functions:

```
// enable and disable functions
document.getElementById("enableBtn").addEventListner("click", () => {
    // addEventListener to each button
    document.querySelectorAll("button.jam").forEach((btn) => {
        btn.addEventListener("click", jamClick);
    });
});
document.getElementById("disableBtn").addEventListner("click", () => {
    // removeEventListner to each button
    document.querySelectorAll("button.jam").forEach((btn) => {
        btn.removeEventListner("click", jamClick);
    });
});
```

However here we have a problem: the `jamClick()` function used to reference the `btn` argument from the anonymous function used in `forEach()`:

```
// function now named for add/remove event listener
function jamClick(){
    // set text to the button's text
    fill.innerHTML = btn.innerHTML;
    // set background color by looking up the color in the dictionary
    fill.style.backgroundColor = colorMap[btn.innerHTML];
}
```

Since we migrated this anonymous function out in order to implement our enable and disable functions, this named function no longer has access to `btn`, meaning it cannot reference its corresponding button's fruit text.

There is a fix to this. Every function passed into `addEventListner()` can take in an additional argument `event`. This additional argument tell JavaScript to interpret `jamClick()` as a **callback** function: a function that is being used as an argument for another function. The argument `event` allows us to access the caller event. In this case, it's the button!

Let's setup `jamClick()` to become a **callback function**:

```
// adding the "event" argument will allow
// JavaScript to interpret this function as a callback function
function jamClick(event){
    // set text to the button's text
    fill.innerHTML = btn.innerHTML;
    // set background color by looking up the color in the dictionary
    fill.style.backgroundColor = colorMap[btn.innerHTML];
}
```

From here we have 2 variations to use `event`:

Variation 1: `event.target`

`jamClick()` is going to be attached to the buttons via `addEventListner()`. In other words, the button is the **target** element of `jamClick()`. As such, the `target` property of `event` provides us access to the button.

Let's swap out `btn` for `event.target`:

```
// adding the "event" argument will allow
// JavaScript to interpret this function as a callback function
function jamClick(event){
    // set text to the button's text
    fill.innerHTML = event.target.innerHTML;
    // set background color by looking up the color in the dictionary
    fill.style.backgroundColor = colorMap[event.target.innerHTML];
}
```

Variation 2: `this` Keyword

A shorthand for `event.target` is to use the `this` keyword. This will behave the exact same way just with different syntax. Note that we still need `event` as an argument, we just won't be using it in favor of `this`:

```
// adding the "event" argument will allow
// JavaScript to interpret this function as a callback function
function jamClick(event){
    // set text to the button's text
    fill.innerHTML = this.innerHTML;
    // set background color by looking up the color in the dictionary
    fill.style.backgroundColor = colorMap[this.innerHTML];
}
```

With **callback** functions we're able to *trickle up* and access the calling element's properties. Understanding how to *trickle up* (callback functions) and *trickle down* (accessing `.properties`) will require a clear understanding of the hierarchy of DOM elements. Understanding and practicing this process of *trickling up/down* will allow you to access the full functionality potential of JavaScript.

The full implementation is provided below:

HTML:

```
<body>
    <center>
    <p class="bread">Wheat Bread</p>
    <p id="filling">Jam</p>
    <p class="bread">Wheat Bread</p>
    <button class="jam" id="strBtn">Strawberry</button>
    <button class="jam" id="graBtn">Grape</button>
    <button class="jam" id="orgBtn">Orange</button>
    <button class="jam" id="bluBtn">Blueberry</button>
    <br> <br>
    <button id="enableBtn">Enable</button> <button id="disableBtn">Disable</button>
    </center>
</body>
```

JavaScript:

```
// Unordered data structure
const colorDictionary = {   Blueberry: "blue",
                            Orange: "orange",
                            Strawberry: "red",
                            Grape: "purple"
                        };

// adding "event" argument makes jamClick() a callback function
// use "event.target" or "this" to access to button
function jamClick(event){
    const fill = document.getElementById("filling");
    fill.innerHTML = this.innerHTML;
    fill.style.backgroundColor = colorDictionary[this.innerHTML];
}


// Initial attaching jam button's functionality upon page load
document.querySelectorAll("button.jam").forEach((btn) => {
    // Use callback function to allow enable and disable
    btn.addEventListener("click", jamClick);
});


// enable button's functionality
document.getElementById("enableBtn").addEventListener("click", () => {
    // addEventListener to each jam button
    document.querySelectorAll("button.jam").forEach((btn) => {
        btn.addEventListener("click", jamClick);
    });
});

// disable button's functionality
document.getElementById("disableBtn").addEventListener("click", () => {
    // addEventListener to each jam button
    document.querySelectorAll("button.jam").forEach((btn) => {
        btn.removeEventListener("click", jamClick);
    });
});
```

2.5.2 `mouseenter`, `mouseleave`, and more

We observe the issue that some background colors makes the text difficult to read:

Let's alleviate this by clearing out the background color when our mouse moves over this text. To be more exact, we'd like two separate behaviors:

1. When the mouse **enters** the text, we want to set the background color to white.
2. When the mouse **leaves** the text, we want to set the background color back to its original color.

Let's deal with the **enter** event first. Here we first identify the DOM element that this event will be attached to:

```
const fill = document.getElementById("filling");
```

Now we want this element to listen to a `mouseenter` event:

```
const fill = document.getElementById("filling");
fill.addEventListener("mouseester", ...);
```

And finally we add an anonymous function specifically catered to this event:

```
const fill = document.getElementById("filling");
fill.addEventListener("mouseester", () => {
    fill.style.backgroundColor = "white";
});
```

Now let's finish the second part, the mouse **leave** event. We'll add a second event listener with the `mouseleave` event. We will once again use an anonymous function:

```
const fill = document.getElementById("filling");
fill.addEventListener("mouseester", () => {
    fill.style.backgroundColor = "white";
});
fill.addEventListener("mouseleave", () => {
    // search up correct color with our map
    fill.style.backgroundColor = colorMap[fill.innerHTML];
});
```

Let's add similar functionality to the bread:

```
<body>
    <p class="bread"> Bread </p>
    <p id="filling"></p>
    <p class="bread"> Bread </p>
<\body>
```

Exercise 2.6 *Using JavaScript, set the background color of both the bread* `<p>`*'s to yellow. Using* `querySelectorAll().forEach()` *is recommended here.*

Exercise 2.7 *Using JavaScript, implement the functionality where the background color of the two bread* `<p>`*'s are cleared to white when the mouse hovers over it. Using* `querySelectorAll().forEach()` *is recommended here.*

The full implementation in JavaScript is as follows:

```
// Get the filling element
const fill = document.getElementById("filling");
// For mapping innerHTML text to the corresponding backgroundColor
const colorMap = {Strawberry: "red", Grape: "purple", Orange: "orange", Blueberry: "blue"};

// Add button functionality using forEach
document.querySelectorAll("button").forEach((btn) => {
    // add click event for each iterated btn
    btn.addEventListener("click", () => {
        fill.innerHTML = btn.innerHTML;
        fill.style.background = colorMap[btn.innerHTML];
    });
});

// add mouseenter and mouseleave events to filling
fill.addEventListener("mouseenter", () => {
    fill.style.backgroundColor = "white";
});
fill.addEventListener("mouseleave", () => {
    // search up correct color with our map
    fill.style.backgroundColor = colorMap[fill.innerHTML];
});

// set the backgroundColor of both bread to yellow
document.querySelectorAll("p.bread").forEach((elem) => {
    elem.style.backgroundColor = "yellow";
});

// add mouseenter and mouseleave events to the bread
// iterate both p.bread
document.querySelectorAll("p.bread").forEach((elem) => {
    // add events
    elem.addEventListener("mouseenter", () => {
        elem.style.backgroundColor = "white";
    });
    elem.addEventListener("mouseleave", () => {
        elem.style.backgroundColor = "yellow";
    });
});
```

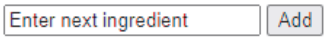
2.6 Adding and Removing DOM Elements

So far we've been able to **access** DOM elements through `getElement(s)By...()` or `querySelectorAll()` functions, and with this access we can **change** it's text (`.innerHTML` or `.value`) and CSS styling.

Let's look at how to dynamically add and remove DOM elements.

2.6.1 `createElement()`, `appendChild()`, and `insertBefore()`

Rather than switching out the bread and filling of a sandwich, let's say we want to build a sandwich piece by piece:



Here we have the barebones HTML. Notice that we've created a `<div>` specifically to place our sandwich in:

```
<center>
    <div id="sandwich">

    </div>
    <input type="text" name="textInput" value="Enter next ingredient">
    <button id="addBtn">Add</button>
</center>
```

Before even worrying about creating and adding new DOM elements, let's get the basic `<input>` reading functionality attached to our add button:

```
function init(){
    // add event to button via anonymous function
    document.getElementById("addBtn").addEventListener("click", () => {
        // read the value in the input
        console.log(document.getElementById("textInput").value);
    });
}

init()
```

With this basic functionality, every time we click the "add" button, our JavaScript will read the input value and print it to the console. This is a good starting point to make sure we have the basics correct.

Our actual desired functionality is:

1. Click the button.
2. A new `<p>` is created with the input text.
3. The new `<p>` is added to our HTML.

We have the first part down where we have a click event setup. Let's implement steps 2 and 3.

Let's create a new <p> element in our button's click event. Make sure we store this new object in a variable:

```
function init(){
    // add event to button via anonymous function
    document.getElementById("addBtn").addEventListener("click", () => {
        // create new <p> element
        const newP = document.createElement("p");
    });
}
```

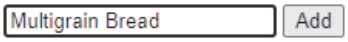
Right now this newly-created <p> is completely empty. Written in HTML it will be:

```
<p></p>
```

Let's add the input text as its .innerHTML:

```
function init(){
    // add event to button via anonymous function
    document.getElementById("addBtn").addEventListener("click", () => {
        // create new <p> element
        const newP = document.createElement("p");
        // set innerHTML to input's text value
        newP.innerHTML = document.getElementById("textInput").value;
    });
}
```

If we type "multigrain bread" in our HTML and click the add button, our HTML equivalent of our newly-created <p> will be:




```
<p>Multigrain Bread</p>
```

However notice that nothing is showing up in our HTML even though we've created a new <p> and updated its .innerHTML. This is because we have yet to add newP into our HTML. We will do this with appendChild(). Remember that we want to add this in our sandwich :

```
function init(){
    // add event to button via anonymous function
    document.getElementById("addBtn").addEventListener("click", () => {
        // create new <p> element
        const newP = document.createElement("p");
        // set innerHTML to input's text value
        newP.innerHTML = document.getElementById("textInput").value;
        // add newP to the sandwich <div>
        document.getElementById("sandwich").appendChild(newP);
    });
}
```

With this we have a working sandwich maker!

However, who builds a sandwich from the top down? Shouldn't the plate come first, then bread, then build the sandwich upwards? Our current sandwich building direction is downwards because `appendChild()`, similar to `push()` for arrays, adds elements to the end of the list, not the beginning of the list.

To add elements to the top/beginning, we'll use a combination of `firstChild` and `insertBefore()`. Let's first access the sandwich `<div>`:

```
function init(){
    // add event to button via anonymous function
    document.getElementById("addBtn").addEventListener("click", () => {
        // create new <p> element
        const newP = document.createElement("p");
        // set innerHTML to input's text value
        newP.innerHTML = document.getElementById("textInput").value;
        // get the sandwich <div>
        const sandwich = document.getElementById("sandwich");
    });
}
```

With the `<div>`, we now want to insert `newP` at the very beginning of the list, before the `.firstChild`.

```
function init(){
    // add event to button via anonymous function
    document.getElementById("addBtn").addEventListener("click", () => {
        // create new <p> element
        const newP = document.createElement("p");
        // set innerHTML to input's text value
        newP.innerHTML = document.getElementById("textInput").value;

        // get the sandwich <div>
        const sandwich = document.getElementById("sandwich");
        // insert newP at the beginning
        sandwich.insertBefore(newP, sandwich.firstChild);
    });
}
```

JavaScript is smart enough to handle the fact that for an empty `<div>`, the `.firstChild` will return `null`, but recognize and insert the new element without exploding.

2.6.2 `removeChild()`

Let's now turn our attention to removing DOM elements. This will not simply be a case of setting `innerHTML`s to empty strings `""`, where the element still exists but "invisible". We will be completely removing the entire DOM element from the HTML.

Let's first add in a clear button to our HTML:

```
<center>
    <div id="sandwich">

    </div>
    <input type="text" id="textInput" value="Enter next ingredient">
    <button id="addBtn">Add</button>
    <br>
    <button id="clearBtn">Clear</button>
</center>
```

First we'll need to create the event connected to the clear button:

```
document.getElementById("clearBtn").addEventListener("click", () =>{
});
```

To remove elements, we need to call `removeChild()` called from the parent element. So let's access the sandwich `<div>`:

```
document.getElementById("clearBtn").addEventListener("click", () =>{
    // get the parent element
    const parent = document.getElementById("sandwich");
});
```

Now we want to remove every single child of this element. To do so, we want to 1) check if there are still child elements, and if so 2) call `removeChild()`:

```
document.getElementById("clearBtn").addEventListener("click", () =>{
    // get the parent element
    const parent = document.getElementById("sandwich");
    // loop checking if there are still children
    while (parent.firstChild){
        // child still exists, remove it
        parent.removeChild(parent.firstChild);
    }
});
```

With this implementation, we can clear every single DOM element that's added as a child element of the sandwich `<div>`.

2.7 Global Events

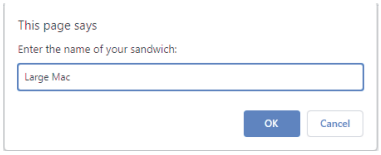
To wrap up our exploration into DOM usage, let's look at some global events. These events are tied directly to the `document` root object in the DOM.

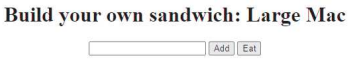
2.7.1 `prompt()`, `alert()`, `confirm()`

At the beginning of this course, we saw the use of `prompt()` as a way to directly read a string input from the user. We temporarily used this before we learned how to use the DOM. We can use `prompt()` to collect precursor information that won't be changed throughout the users' visit to the page.

Consider again our sandwich-builder web page. The bread and contents of the sandwich will always be changing, therefore it does not make sense to use `prompt()` to ask user for the input. However maybe we want the user to name their culinary masterpiece. Since this name is set and should never be changed from the beginning of the web page, it makes more sense to use `prompt()`:

```
const name = prompt("Enter the name of your sandwich:");
document.querySelector("h1").innerHTML += name;
```





If we want a popup window to display a message without any user inputs, then we can use the `alert()` function:

```
// add an alert to the eat/clear button
document.getElementById("clearBtn").addEventListener("click", () => {
    // check if there is anything added to the sandwich
    const div = document.getElementById("sandwich");
    if (div.childNodes.length == 0){
        alert("You cannot eat an empty sandwich!");
    }
});
```

Lastly, we can provide users a boolean window chose between "OK" and "Cancel" and adjust our web page behavior accordingly. This will make use of the `confirm()` function. If the user clicks "OK" then `confirm()` will return `true`; if "Cancel" then `confirm()` will return `false`. Thus `confirm()` is often used inside conditional statements:

```
// add an alert to the eat/clear button
document.getElementById("clearBtn").addEventListener("click", () => {
    // check if there is anything added to the sandwich
    const div = document.getElementById("sandwich");
    if (div.childNodes.length == 0){
        alert("You cannot eat an empty sandwich!");
    }

    // check if user is ready to eat
    if(confirm("Have you washed your hands?")){
        while(div.firstChild){
            div.removeChild(div.firstChild);
        }
    }
});
```

2.7.2 `setTimeout()`

Sometimes instead of waiting for the user to click on the pop-up window, we just want to wait for a specified amount of time before something happens. Let's say for instance we don't want the sandwich to immediately disappear since no one can gobble down a sandwich instantaneously. Instead we want to wait 3 seconds before we clear out all the sandwich text. To do this, we can use the `setTimemout()` function. Note that `setTimeout()` takes two arguments: 1) a function of the action to do after the timeout is finished, and 2) the timeout length in miliseconds.

```
// add an alert to the eat/clear button
document.getElementById("clearBtn").addEventListener("click", () => {
    // check if there is anything added to the sandwich
    const div = document.getElementById("sandwich");
    if (div.childNodes.length == 0){
        alert("You cannot eat an empty sandwich!");
    }

    // check if user is ready to eat
    if(confirm("Have you washed your hands?")){
        // wait 3000 miliseconds before clearing everything out
        setTimeout(() => {
            // after timeout, logic to run (removeChild())
            while(div.firstChild){
                div.removeChild(div.firstChild);
            }
        }, 3000);
    }
});
```

Exercise 2.8 *For the clear function, add in* `setTimeout()` *such that the page will clear one line of the sandwich content every 0.5 seconds*

```
// add an alert to the eat/clear button
document.getElementById("clearBtn").addEventListener("click", () => {
    // check if there is anything added to the sandwich
    const div = document.getElementById("sandwich");
    if (div.childNodes.length == 0){
        alert("You cannot eat an empty sandwich!");
    }

    // check if user is ready to eat
    if(confirm("Have you washed your hands?")){
        // wait 3000 miliseconds before clearing everything out
        setTimeout(() => {
            // after timeout, logic to run (removeChild())
            // remove one line every 0.5 seconds









        }, 3000);
    }
});
```

Now each of these events happen globally, which can cause quite a bit of disruption for the user's experience. Use these global events in seldom.