/01

Learn to Code with JavaScript

Document Object Model (DOM)

L. Hernández | 2023

Document Object Model

The *Document Object Model (DOM)* represents an HTML document as a network of connected *nodes* that form a tree-like structure.

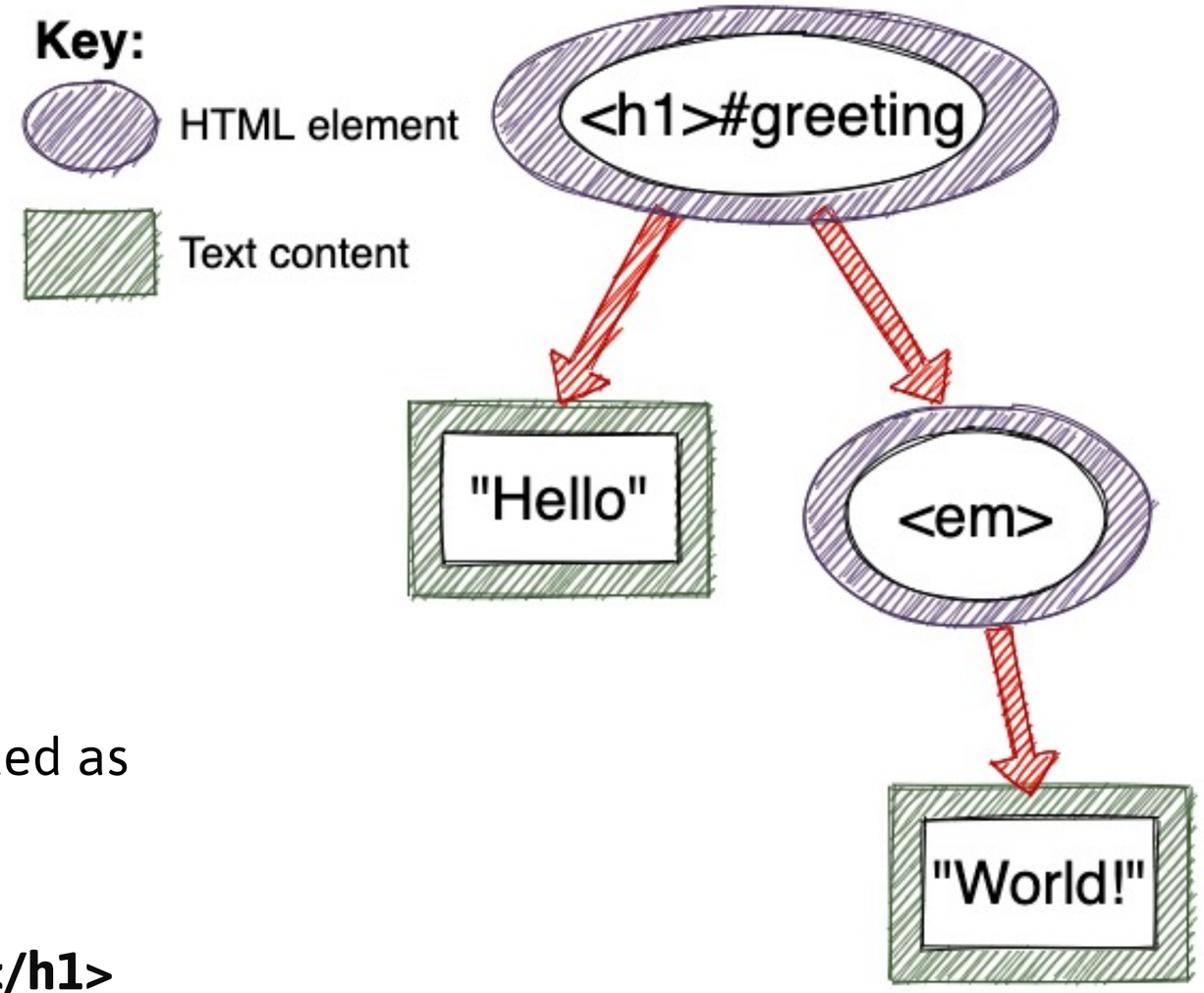
DOM represents *HTML tags* as *element nodes* and any text inside these *tags* as *text nodes*.

All these *nodes* are connected to make a node-tree that describes the overall structure of a web page.

Document Object Model

The following HTML code can be represented as this node-tree diagram:

<h1 id='greeting'>Hello, World!</h1>



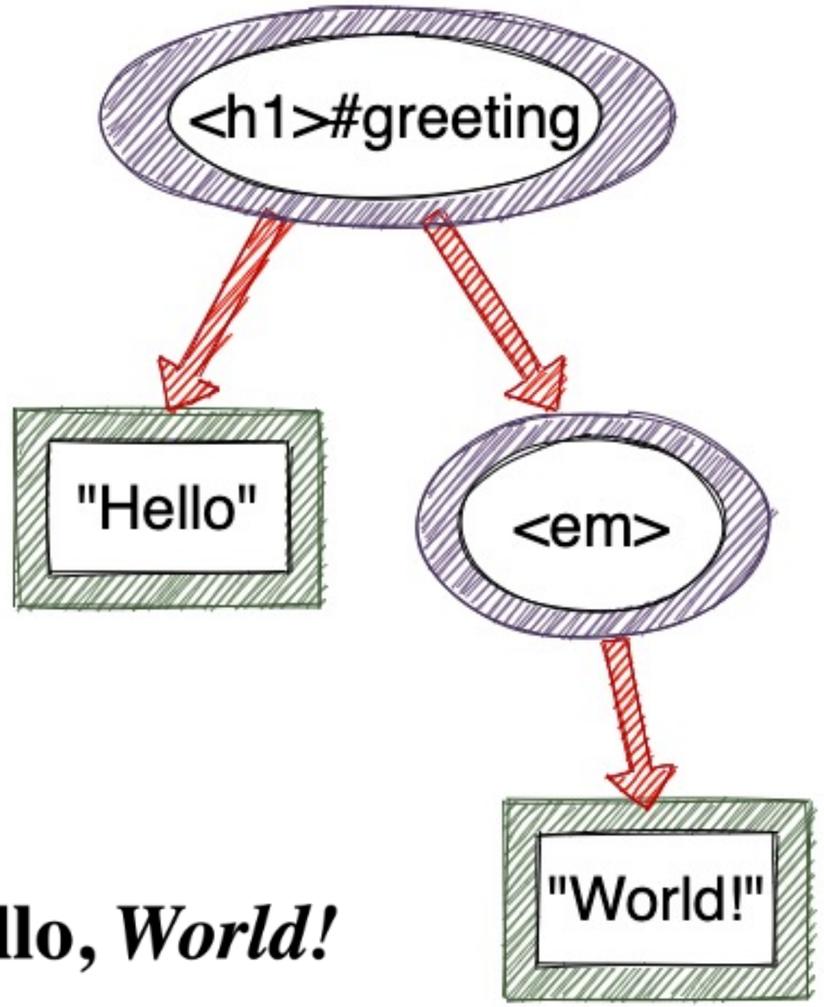
Document Object Model

The **<h1>** tag appears as an *element node* (purple) at the top of the *node tree*.

The word “Hello” is text, so this is a *child text node* (green).

The **** element is inside the **<h1>** tag, so it is a *child element node*.

This makes **<h1>** *element node* a *parent node* of these *child nodes*. The text inside the **** tags is a *text child node* of the **** *element node*.



Hello, *World!*

HTML document vs DOM

When we visit a web page, browser first downloads page document with all its HTML, text, images and so on.

Browser then creates a representation (or model) of that document, which is the *Document Object Model (DOM)*.

Finally, browser uses that the DOM to display web page on our device.

JavaScript allows changes to be made to web page, but those changes are made to the DOM, rather than to the original, hard-coded HTML.

Getting an Element

The DOM provides a method called **getElementById** that returns a *reference* to element with a particular **id** *attribute*.

For example, we can get a reference to **<h1>** heading element.

```
<h1 id='greeting'>Hello, <em>World!</em></h1>
const hello = document.getElementById('greeting');
```

If no element exists with the **id** provided, **null** is returned.

Updating HTML

The easiest way to update the HTML on a page is to use the **innerHTML** property.

This will return all the HTML that's enclosed inside that element's tags as a string.

The **innerHTML** property is that it's also writable, so it gives us a convenient way to insert any HTML inside an element.

Updating HTML

To demonstrate this, let’s add some JavaScript code to give a more personalized greeting.

First of all, we need to have a *reference* to the heading:

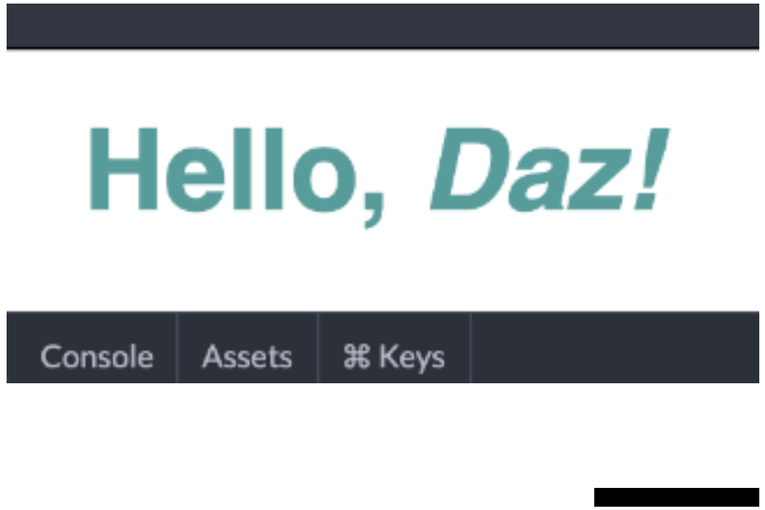
const hello = document.getElementById('greeting');

Next, we’ll store the user name in a variable:

const name = 'Daz';

Last of all, we will replace **innerHTML** property with our own personalized greeting using this variable:

hello.innerHTML = `Hello, <em>${name}!</em>`;



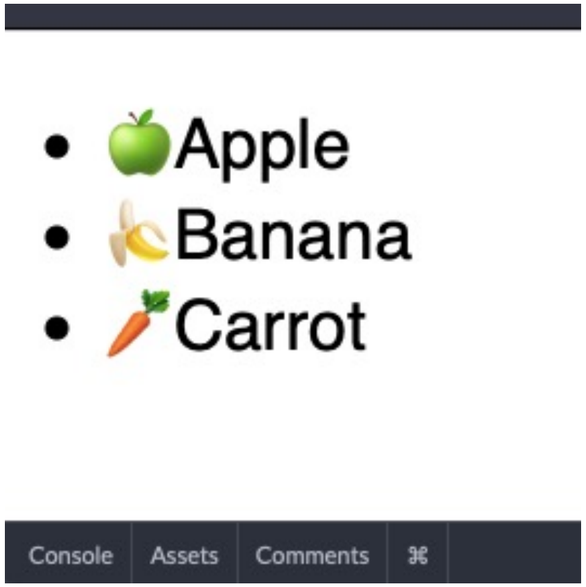
Updating HTML

We've used a *template literal* to produce this HTML.

These are useful when creating any HTML code to dynamically insert into a web page, as they allow variables to be inserted directly into them.

/10

Getting Multiple Elements



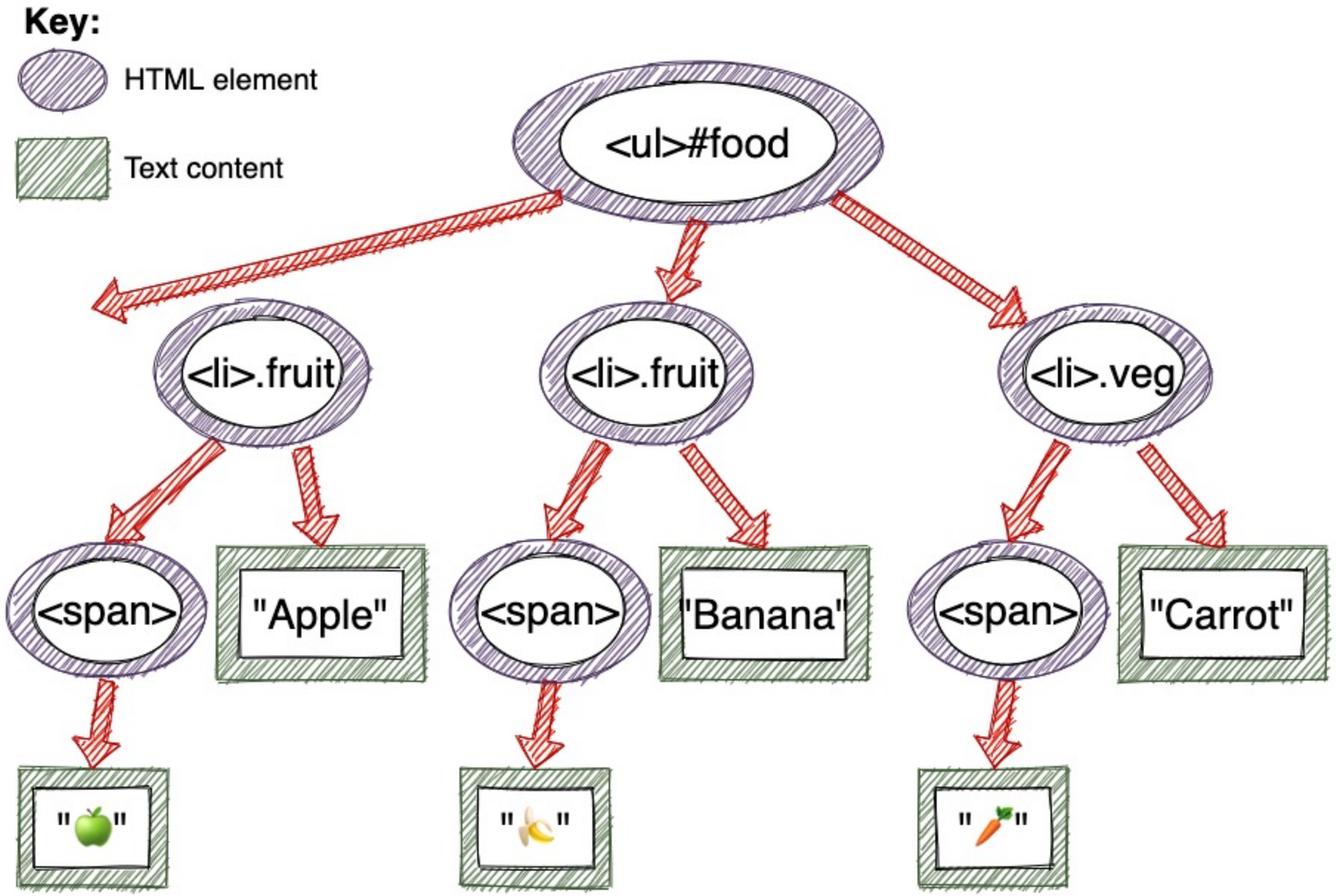
Let’s have a look at how to select more than one element at once.

```
<ul id='food'>
  <li class='fruit'><span>🍏</span>Apple</li>
  <li class='fruit'><span>🍌</span>Banana</li>
  <li class='veg'><span>🥕 </span>Carrot</li>
</ul>
```

L. Hernández | 2023

Getting Multiple Elements

The *DOM tree* for this HTML is shown in this diagram.



Getting Multiple Elements

We can gain access to **<ul>** element using this code:

const food = document.getElementById('food');

But how can we access a group of elements, such as this list items?

The DOM also provides a few methods that allow us to access *groups of elements*.

Getting Multiple Elements

We can use **getElementsByTagName()** to return a collection of all elements with *tag name* provided as an argument.

For example, we can get all list items (HTML tag **<li>**) in document using this code:

const items = document.getElementsByTagName('li');

This variable now contains a collection of all the list-item elements.

We can access each item in the collection using the *square bracket index notation* (**items[0]**).

/14

Getting Elements by Class Name

We can use **getElementsByClassName()** method to return a collection of all elements that have a particular *class name*.

```
<ul id='food'>
  <li class='fruit'><span>🍏</span>Apple</li>
  <li class='fruit'><span>🍌</span>Banana</li>
  <li class='veg'><span>🥕 </span>Carrot</li>
</ul>
```

For example, this code will return a collection of all elements with the class of **fruit**:

```
const fruit = document.getElementsByClassName('fruit');
```

Getting Elements by Class Name

If there are no elements with given class, a collection will still be returned, but it will contain no items and have a length of 0.

Getting Elements by Class Name

We can only get one element by ID (**getElementById**) but multiple elements by *class name* (**getElementsByClassName**).

We are only allowed to use a particular ID once per HTML document, while we can use a particular *class name* multiple times in the same document.

Query Selectors

Another way to get elements in the DOM it to use *query selectors* and they allow us to use *CSS notation* to target specific elements.

The **querySelector()** method allows us to find the first element in the document that matches the *CSS selector* provided.

Query Selectors

For example, instead of using **getElementById** to get a *reference* to the element with an ID, we could use **querySelector()**:

```
const food = document.querySelector('#food');
```

The **querySelectorAll()** method also uses *CSS notation*, but returns a list of all elements in document that match the *CSS query selector*.

These two statements are identical and return the same *node list*:

```
document.getElementsByClassName('fruit')`;
document.querySelectorAll('.fruit');
```

Query Selectors

Query selectors are powerful methods that can emulate all the previous methods.

They allow us more *fine-grained control* over which *element nodes* are returned.

Query Selectors

CSS pseudo-selectors can be used to pinpoint a particular element.

This code, for example, will return only the last list item in this list:

const carrot = document.querySelector('ul#food li:last-child');

L. Hernández | 2023

Navigating DOM Tree

DOM nodes have a number of properties and methods for navigating around *document tree*.

Once we have a *reference* to an element, we can walk along *document tree* to find other *nodes*.

Parent Node

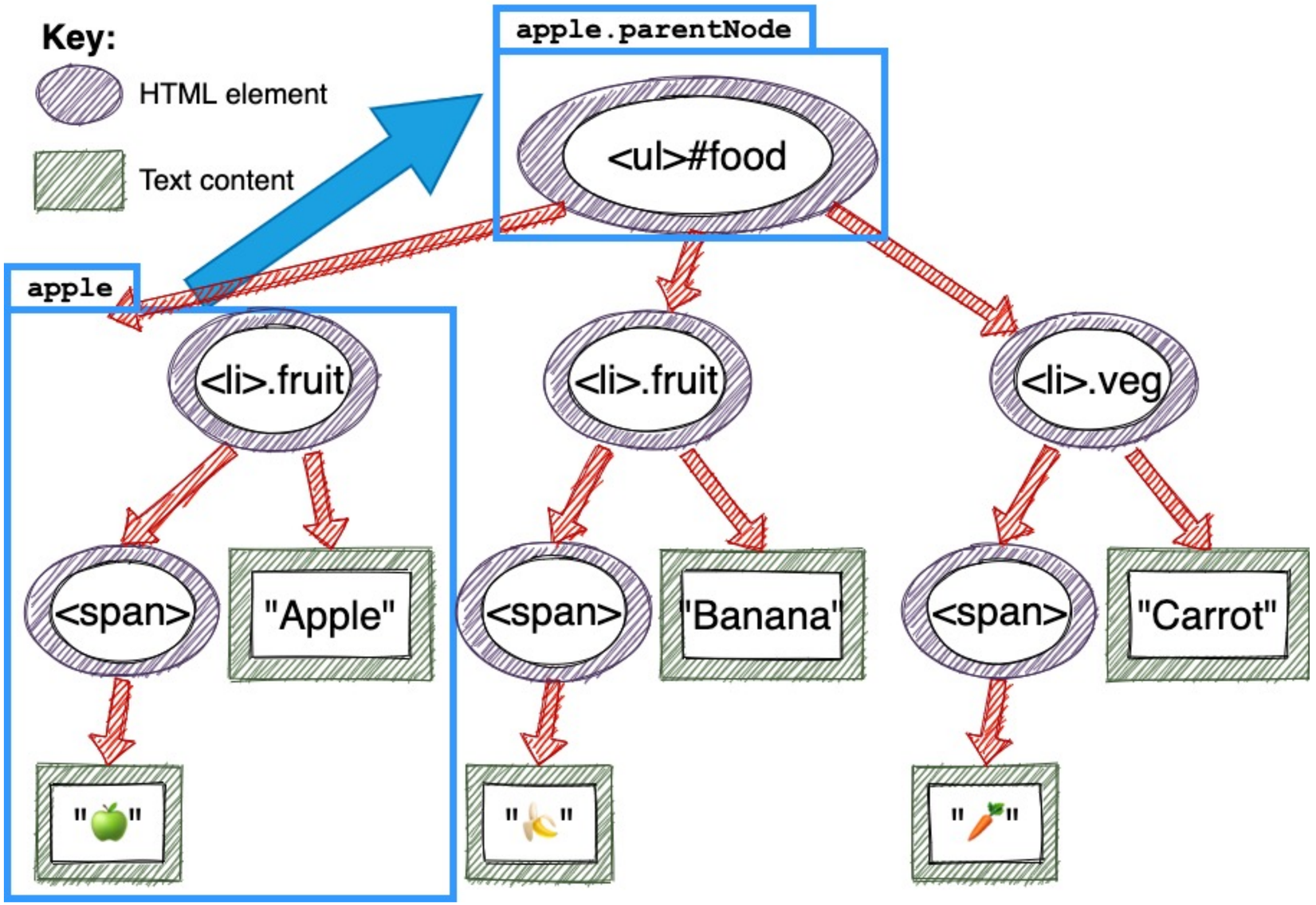
The **parentNode** *property* returns the *parent node* of an element.

In this example, it will return the **food** *node* because it's the *parent* of the **apple** *node*:

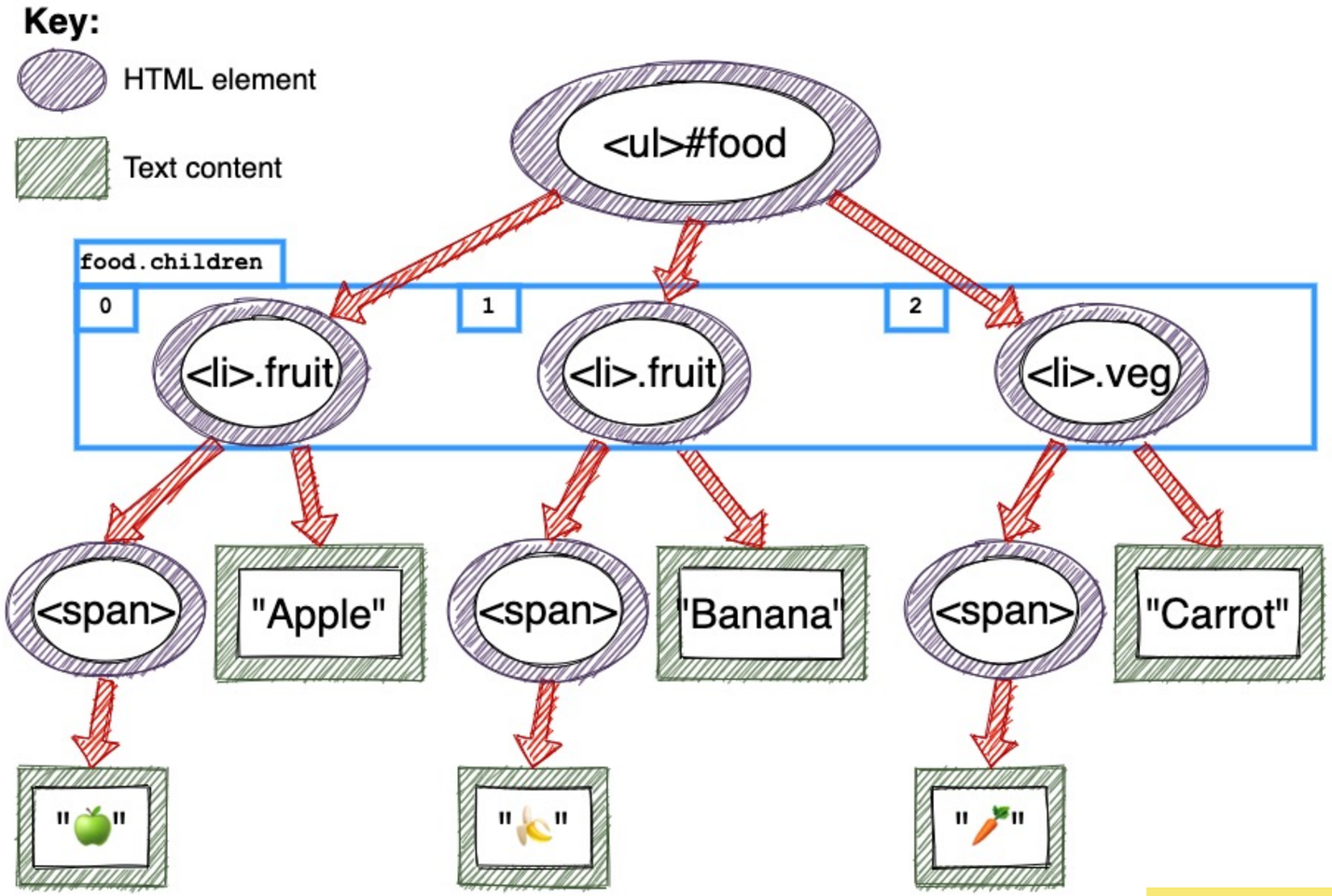
```
<ul id='food'>
<li id='apple '><span>🍏</span>Apple</li>
<li id='banana'><span>🍌</span>Banana</li>
<li id='carrot'><span>🥕 </span>Carrot</li>
</ul>
```

/23

Parent Node



L. Hernández | 2023



Child Nodes

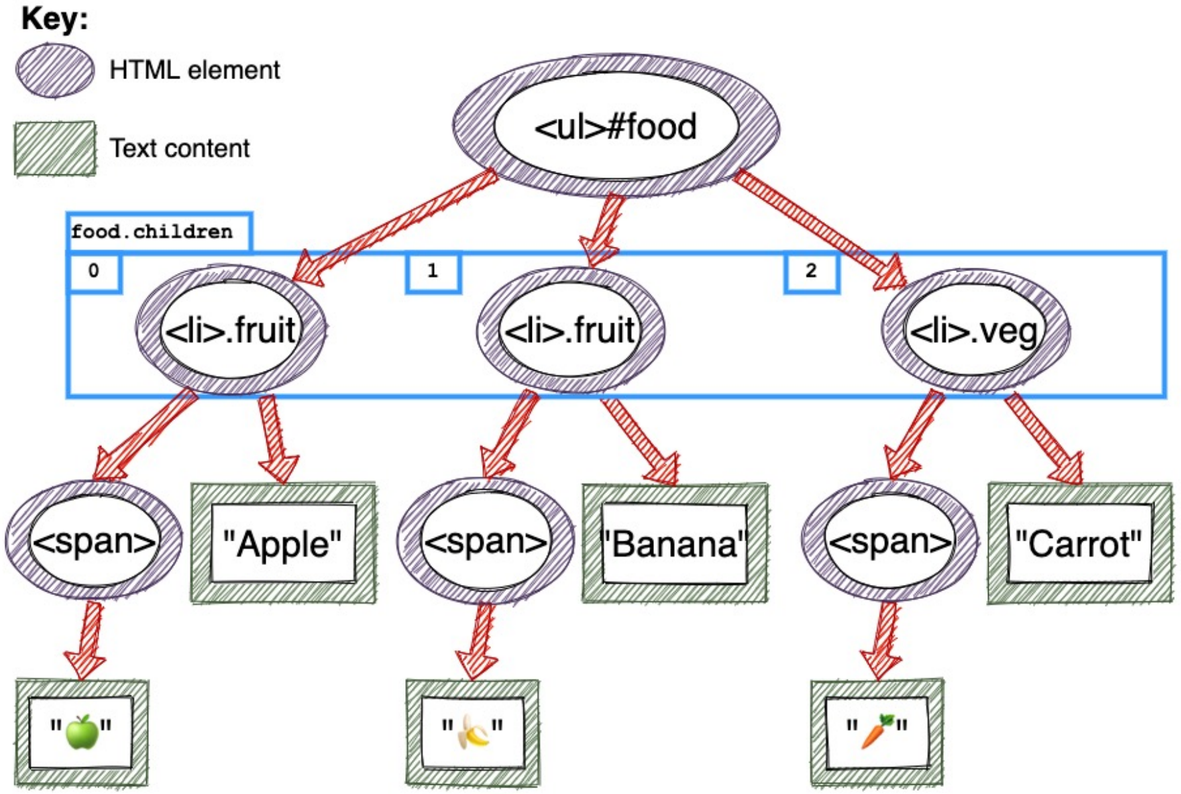
We can get a collection of all *child elements* of an element using the **children** *property*.

In this example, it will return a *node list* of all *child elements* of **<ul>** element that has an ID of **food**.

Child Nodes

Using the **children** *property*, we can create some *references* to these *child nodes* in this example:

const apple = food.children[0];
const banana = food.children[1];
const carrot = food.children[2];



/26

Creating Dynamic Markup

We've looked at how to gain access to different elements of a web page and find out information about them.

Real power of the *DOM* is its ability to dynamically update the *markup*, with the **innerHTML** *property*.

We're going to create new elements and add them to the page, update elements that already exist, and remove any unwanted elements from the page.

The **document** *object* has a **createElement()** method that takes a *tag name* as a parameter and returns that element.

For example, we could create a new item for our list as a *DOM fragment* in memory:

```
const melon = document.createElement('li');
```

We'll use **innerHTML** *property* to add this HTML content:

```
melon.innerHTML = `<span>🍉</span>Melon`;
```

Creating Dynamic Markup

There's also a **textContent** *property* that can be used to add text to an element, but we can't use any *HTML elements* in it.

If we tried to add **** tags around the emoji, it wouldn't parse the HTML.

// this would be fine:
melon.textContent = '🍉Melon';

// this code won't parse the tags ...
melon.textContent = '🍉Melon';

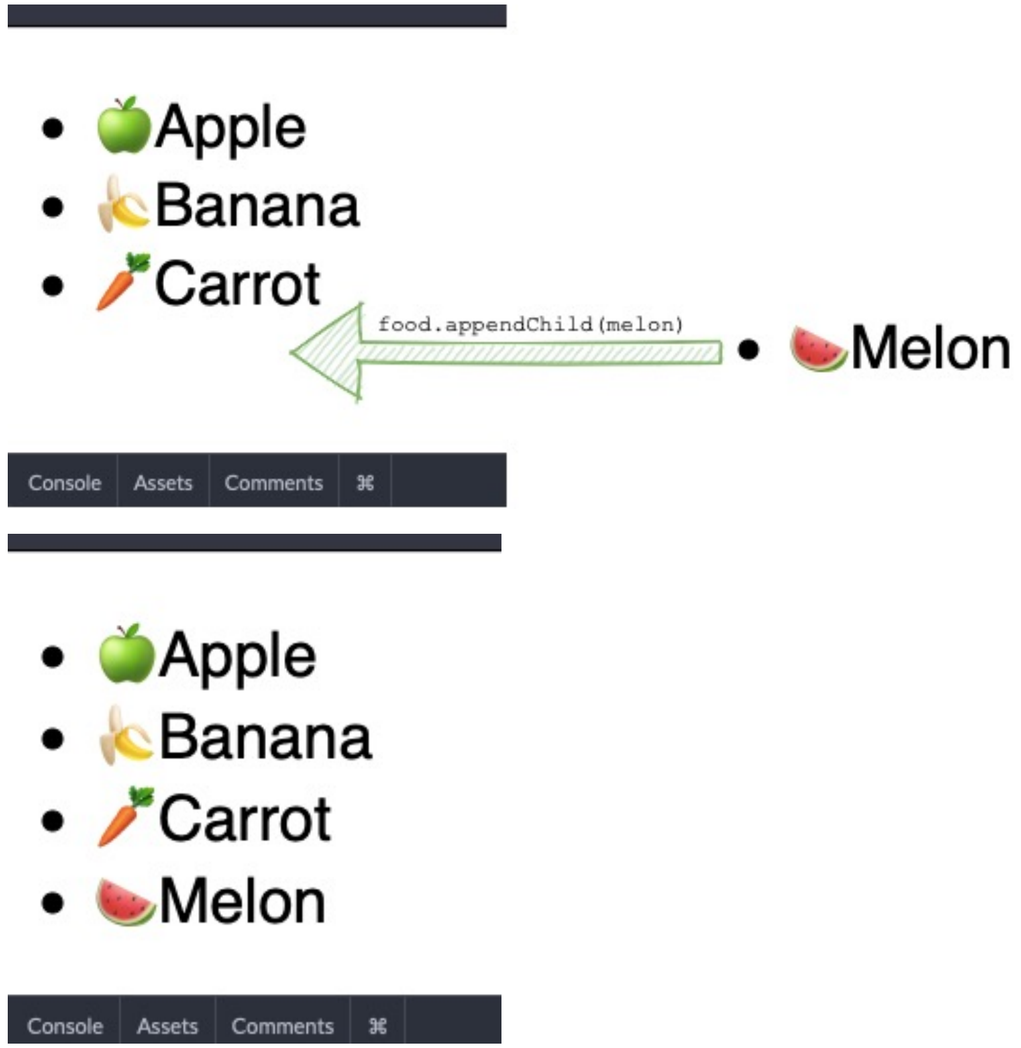
- 🍉Melon

Adding Elements to Page

Every *DOM node* has an **appendChild()** method that will add another *node* (given as an argument) as a *child node*.

This example will add the **melon** element we created to the end of our list:

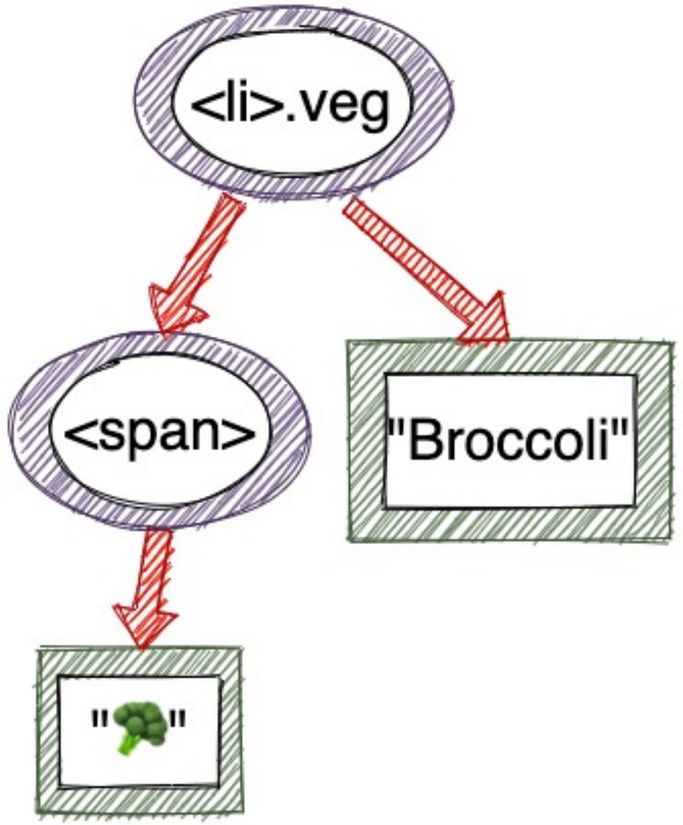
food.appendChild(melon);



L. Hernández | 2023

Building Elements Node by Node

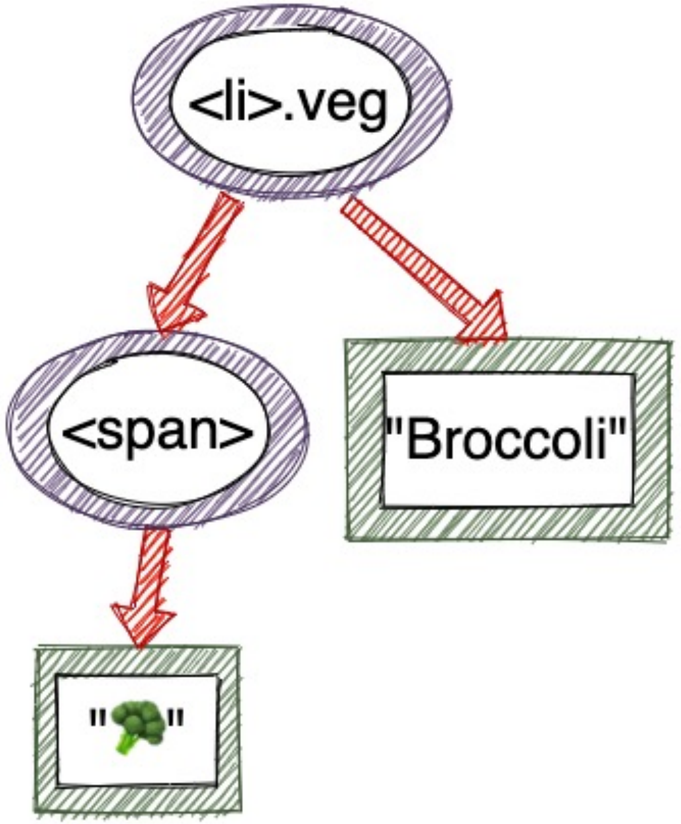
An alternative to using **innerHTML** to populate content of an element is to build each *node* individually and then use **appendChild()** method to put them all together.



Building Elements Node by Node

We've already seen **createElement()** method that's used to create *element nodes* (purple).

There's also a **createTextNode()** method that we can use to create *text nodes* (green).

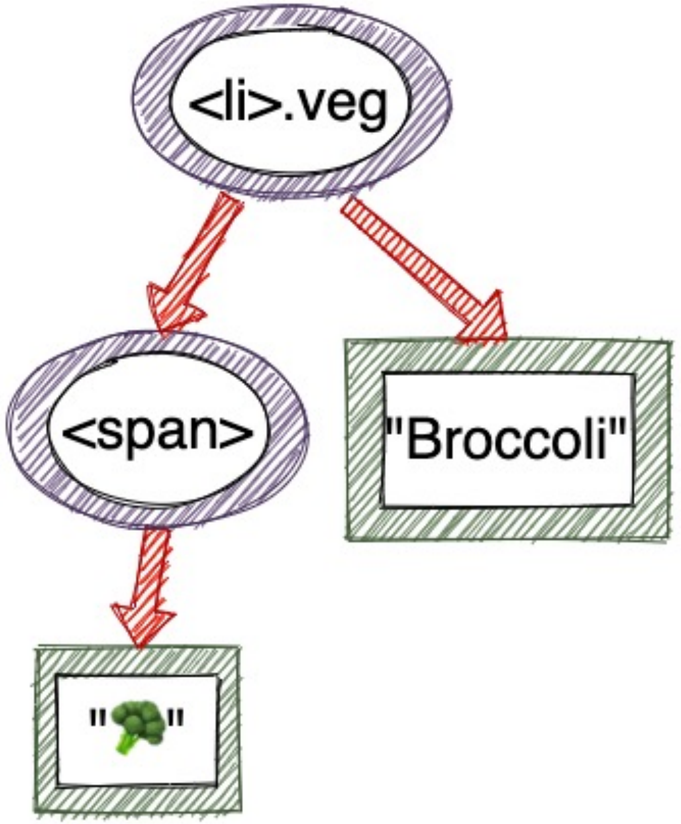


Building Elements Node by Node

Using **createElement()** and **createTextNode()** we can first create all *nodes* and then put them together to form a list item.

```
const broccoli = document.createElement('li');
const text = document.createTextNode('Broccoli');
const span = document.createElement('span');
const emoji = document.createTextNode('🥦');

span.appendChild(emoji);
broccoli.appendChild(span)
broccoli.appendChild(text);
```



Insert Before

Now, we need to insert this new list item into the HTML.

The **appendChild()** method is useful as we want to add a new *element* to the bottom of a list.

But what if we want to place a new *element* in between two existing *elements*?

Insert Before

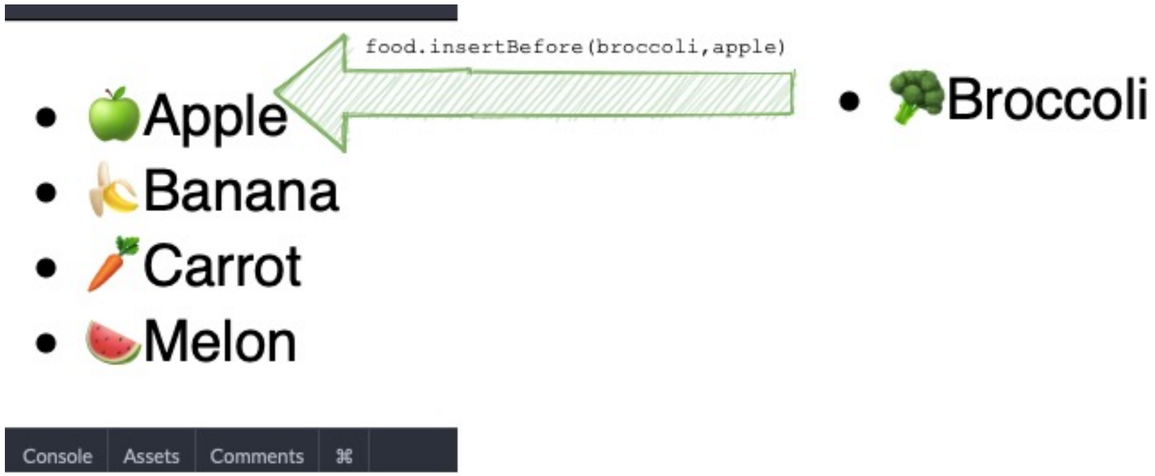
The **insertBefore()** method will place a new *element* before another *element* in the *markup*.

This method is called on the *parent node* and it takes two parameters: first is new *node* to be added, and second is *node* that we want it to go before.

Insert Before

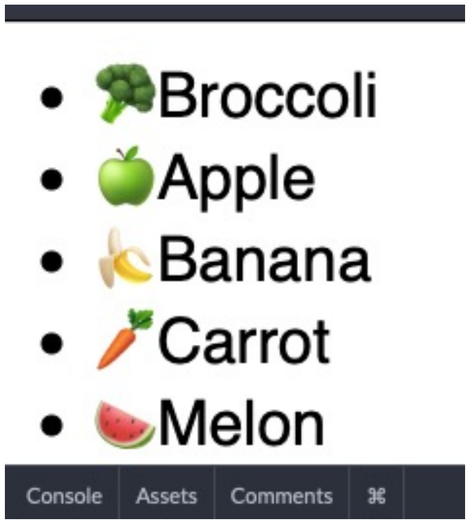
For example, we can place our new **broccoli** element before **apple** element with this code:

food.insertBefore(broccoli,apple);



This will place the **broccoli** element at the top of list, before **apple** element.

There's no **insertAfter()** method, so we need to ensure that we have access to correct elements to place an element exactly where we want it.



Insert Before

/37

Removing Elements from a Page

An *element* can be removed from a page using the **remove()** method.

This method is called on *node* to be removed and it returns a *reference* to removed node.

For example, if we wanted to remove **carrot** *element*, we would use this code:

carrot.remove();

Removing Elements from a Page

JavaScript has removed *element* list item from the *DOM*, but not from the actual HTML.

The **carrot** list item has been removed from *rendered view* (what we see in the browser), but original **carrot** code is still there in HTML.

The **replaceChild()** method can be used to replace one *node* with another.

It's called on the *parent node* and has two parameters: new *node*, and *node* that's to be replaced.

For example, if we wanted to change content of third list item, we could replace this *text node* with a new one:

```
const lemon = document.createElement('li');
lemon.innerHTML = `<span>🍋</span>Lemon`;
food.replaceChild(lemon,banana);
```

Replacing Elements on a Page

Getting and Setting Attributes

All *HTML elements* have a large number of possible *attributes*, such as **class**, **id**, **src**, and **href**.

The *DOM* has a number of methods that can be used to get or set current *attributes* from *HTML elements*.

Getting an Element's Attributes

The **getAttribute()** method returns the value of *attribute* provided as an argument.

For example, we can find out the *class* of **apple** *element* by using this code:

apple.getAttribute('class');

Getting an Element's Attributes

If an *element* doesn't have the given *attribute*, it returns **null**.

For example, if we use this code, we can see that **broccoli** *element* doesn't have a **src** *attribute*:

broccoli.getAttribute('src');

The **setAttribute()** method can change value of an *element's attribute*.

It takes two arguments: *attribute* that we wish to change, and the new value of that *attribute*.

For example, if we want to add *class* of **veg** to **broccoli** *element*, we can use this code:

```
broccoli.setAttribute('class', 'veg');
```

Setting an Element's Attributes

Setting an Element's Attributes

We can modify the *class name* of an *element* using the **setAttribute()** method.

When **setAttribute()** is used to update the **class** *attribute*, it will overwrite all classes that this element has.

Using **className** Property

There's a **className** *property* that be able to used to find out the value of **class** *attribute*.

apple.className;

We can also use **className** *property* to set the **class** *attribute* of an *element*.

melon.className = 'fruit';

Using **className** Property

As with **setAttribute**, changing the **className** property of an element by assignment will also overwrite all other classes that have already been set on element.

This problem can be avoided by using **classList** property instead.

Using **classList** Property

The **classList** property is a list of all *classes* an *element* has.

This also has a number of methods that can be used to modify the *class* of an *element*.

Using **classList** Property

The **add()** method can be used to add a *class* to an *element* without overwriting any *classes* that already exist.

For example, we could add a *class* of **fruit** to the **lemon** *element* with this code:

```
lemon.classList.add('fruit');
```

Using **classList** Property

The **remove()** method will remove a specific *class* from an *element*.

For example, we could remove the *class* of **fruit** just added this code:

```
lemon.classList.remove('fruit');
```

The **contains()** method will check to see if an *element* has a particular *class*.

```
apple.classList.contains('fruit');
```

Every *element node* has a **style** *property*.

This can be used to dynamically modify the presentation of any *element* on a web page:

apple.style.border = "red 2px solid";

Doing it with **style**

Any *CSS property names* that are separated by hyphens must be written in *camelCase notation* when referenced in JavaScript.

For example, *CSS property* **background-color** becomes **backgroundColor**, and **font-size** becomes **fontSize**.

Being Classy

While it can be useful to edit *styles* of *elements* on the fly, it can get messy if we want to change a large number of *styles* all at once.

A better alternative is to dynamically change the *class* of an *element* and have different *styles* for each *class* in the CSS.

Being Classy

For example, if we wanted to add a red border around the **apple** *element*, we could also add a *class* of highlighted to the **apple** *element*:

```
apple.classList.add('highlighted');
```

Then we need to add this new *style rule* to the CSS section:

```
.highlighted {
  border: red 2px solid;
}
```

Being Classy

This method gives us more flexibility if we decide later on to change how we style *elements*.

We might want to use a blue background and bold text instead of a red border, for example.

All we'd need to do is change code for highlighted *class* in this CSS section.