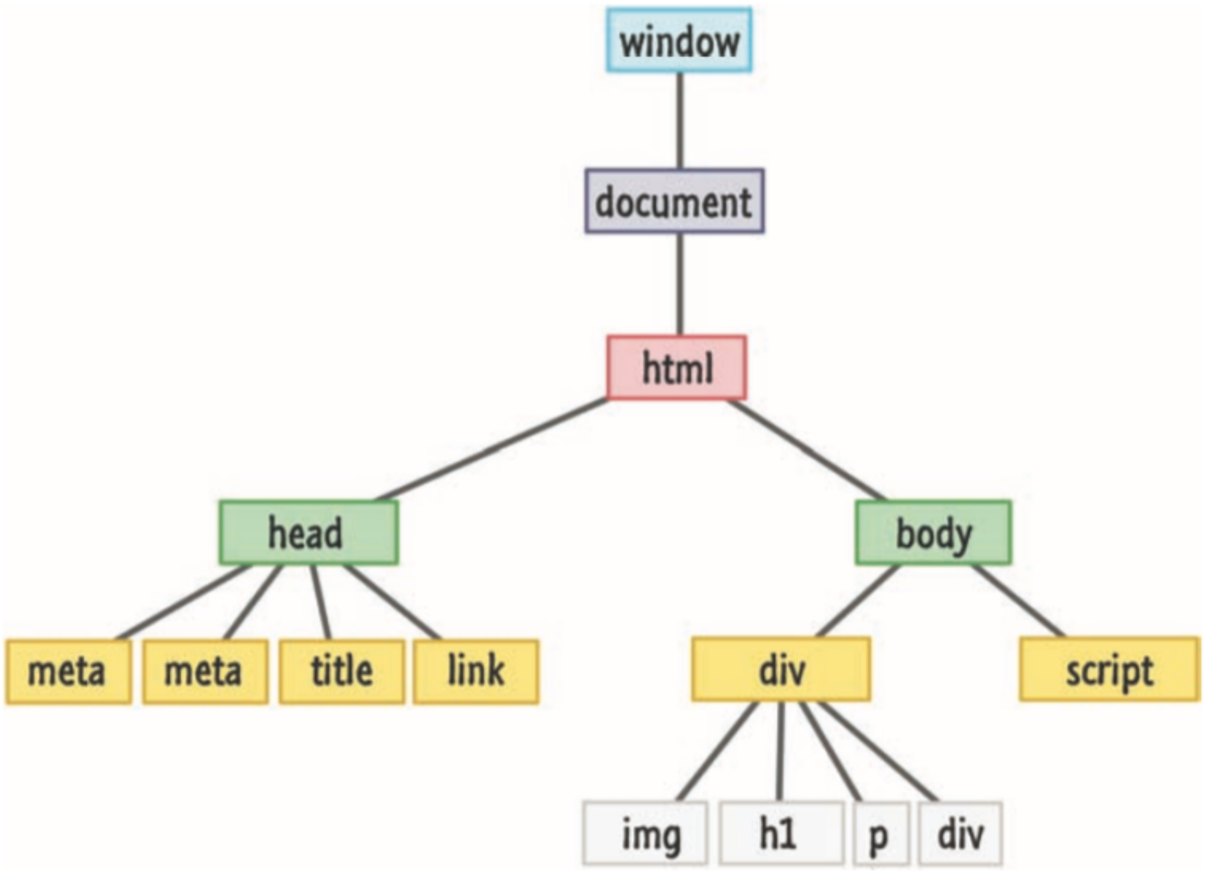
JavaScript Absolute Beginner's Guide Part 3

Part 3: Working with the DOM

Chapter 21: JS, The Browser, and The DOM

Meet the Document Object Model

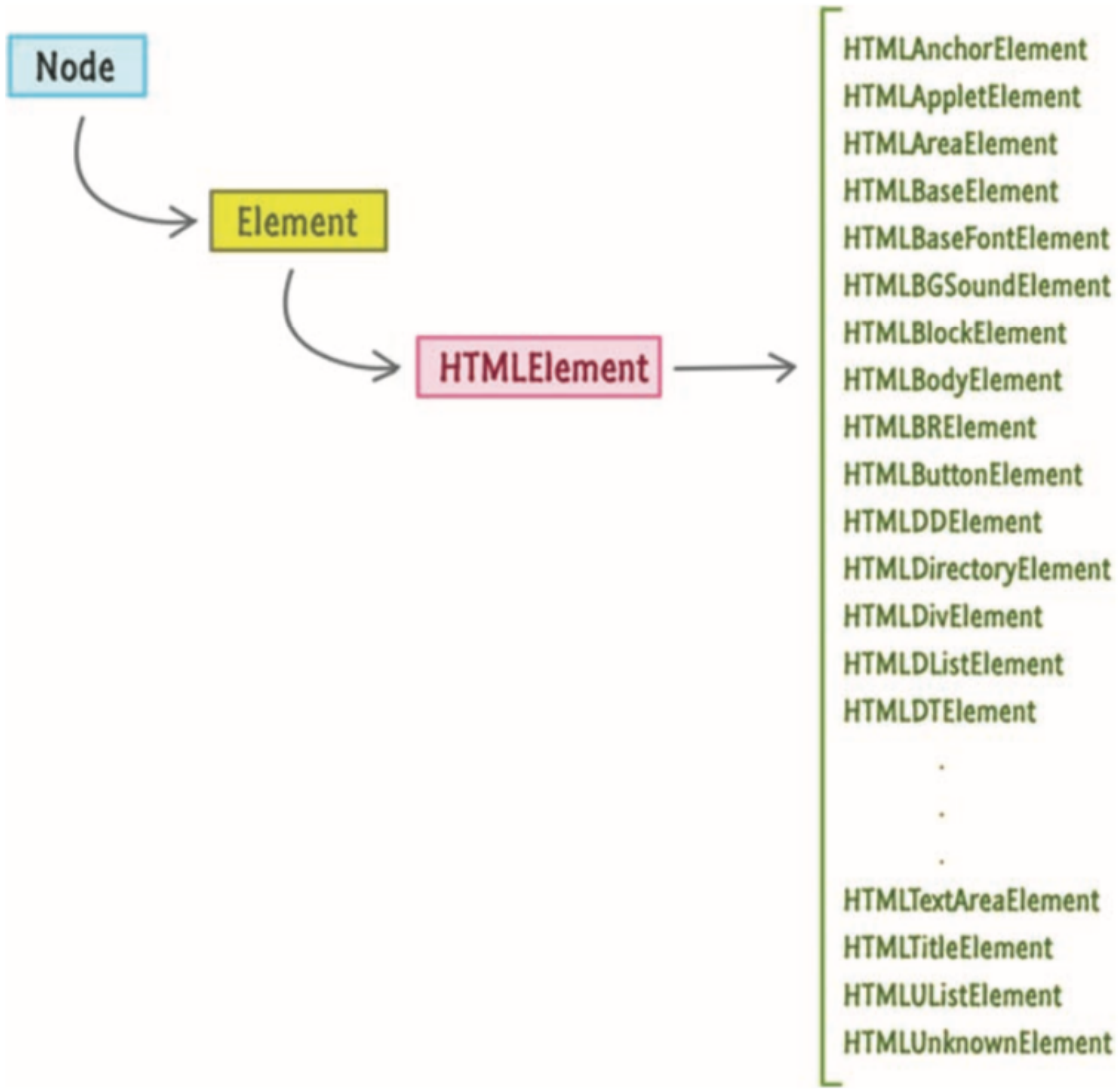
- The DOM (Document Object Model) is the hierarchical structure that your browser uses to make sense of everything going on when summing together all of the HTML, CSS, and JavaScript used to render a webpage.



- Your DOM is actually made up many kinds of things beyond just HTML elements.
    - All of those things that make up your DOM are more generically known as **nodes**.
        - These nodes can be elements, attributes, text content, comments, document-related stuff, and various other things you simply never think about.
        - The only node we have to worry about 99% of the time is the element.
        -

- Every HTML element you want to access has a particular type associated with it, and all of these types extend from the Node base that make up all nodes.

    - Your HTML elements are at the end of a chain that starts with `Node` and continues with `Element` and `HTMLElement` before ending with a type (ie: `HTMLDivElement`, `HTMLHeadingElement`, and so on.) that matches the HTML element itself.
        - The properties and methods you will see for manipulating HTML elements are introduced at some part of this chain.



The Window Object

- In the browser, the root of your hierarchy is the `window` object that contains many properties and methods that help you work with your browser.
    - In other words, the `window` object deals with your browser window.
    - Some of the things you can do with the help of the window object include:
        - accessing the current URL, getting information about any frames in the page,
        - using local storage, seeing information about your screen,

        - fiddling with the scrollbar, - setting the statusbar text,
        - and all sorts of things that are applicable to the container your 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.
    - In other words, the `document` object deals with EVERYTHING that lives in your document.
    - The `document` object does not simply represent a read-only version of the HTML document. It is a two-way street where you can read as well as manipulate your document at will.
        - Any change you make to the DOM via JavaScript is reflected in what gets shown in the browser.
    - You will be relying on functionality the `document` object provides for listening to and reacting to events.

Chapter 22: Finding Elements in The DOM

Meet the `querySelector` Family

- To help explain `querySelector` and `querySelectorAll` take a look at the following HTML:

```
<div id="main">
    <div class="pictureContainer">
        <img class="theImage" src="smiley.png" height="300" width="150"
/>
    </div>
    <div class="pictureContainer">
        <img class="theImage" src="tongue.png" height="300" width="150"
/>
    </div>
    <div class="pictureContainer">
        <img class="theImage" src="meh.png" height="300" width="150" />
    </div>
    <div class="pictureContainer">
        <img class="theImage" src="sad.png" height="300" width="150" />
    </div>
</div>
```

    - In this example, you have one `div` with an `id` of **`main`**, and then you have four `div` and `img` elements each with a class value of **`pictureContainer`** and **`theImage`** respectively

`querySelector`

- The `querySelector` function basically works as follows:

```
var element = document.querySelector("CSS selector");
```

- The `querySelector` function takes an argument, and this argument is a CSS selector for the element you wish to find.
- What gets returned by `querySelector` is the **first element** it finds, even if other elements exist that could get targeted by the selector.

- Taking the HTML from our earlier example, if we wanted to access the `div` whose `id` is `main`, you would write the following:

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

  - Because `main` is the `id`, the selector syntax for targeting it would be `#main`.

- Similarly, let's specify the selector for the `pictureContainer` class:

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

  - What gets returned is the first `div` whose class value is `pictureContainer`. The other `div` elements with the class value of `pictureContainer` will simply be ignored because they are not first.

`querySelectorAll`

- The `querySelectorAll` function returns all elements it finds that match whatever selector you provide:

```
var element = document.querySelectorAll("CSS selector");
```

  - What gets returned is an **array-like container of elements**, as opposed to just the first occurence of the element like with `querySelector`

- Continuing to use the HTML from earlier, here is what our JavaScript would look like if we wanted to use `querySelectorAll` to help us display the `src` attribute of all the `img` elements that contain the class value `theimage`:

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

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

- You can use the full range of CSS Selector Syntax variations as a function argurment to `querySelector` and `querySelectorAll`

- If you wanted to target all of the `img` elements without having to specify the class value, here is what our `querySelectorAll` call could look like:

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

- If you wanted to target only the image whose `src` attribute is set to `meh.png`, you can do the following:

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

- **NOTE:** Missing in all of this element-finding excitement were the `getElementById`, `getElementsByTagName`, and `getElementsByClassName` functions. Back in the day, these were the functions you would have used to find elements in your DOM. The `querySelector` and `querySelectorAll` functions are the present and future solutions for finding elements, so don't worry about the `getElement*` functions anymore.
  - As of right now, the only slight against the `querySelector` and `querySelectorAll` functions is performance.
    - The `getElementById` function is still pretty fast, and you can see the comparison for yourself here: http://jsperf.com/getelementbyid-vs-queryselector.

Chapter 23: Modifying DOM Elements

- Disclaimer: the DOM was never designed to mimic the way Objects work. Many of the things you can do with objects you can certainly do with the DOM, but that is because the browser vendors help ensure that.
- The example HTML document we will refer to:

  ```
  <!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="theTitle" class="highlight summer">
        What's happening?
    </h1>
    <script>
    </script>
</body>
</html>
```

Changing an Element's Text Value

- The way you modify the text value is by setting the `textContent` property.
  - The `textContent` property can also be read like any variable to show the current value.
- Example of chainging text value:

```
<script>
    var title = document.querySelector("#theTitle");
    title.textContent = "Oppa Gangnam Style!";
</script>
```

  - If you make this change in the HTML example above preview it in a browser, you will see the words "Oppa Gangnam Style!" show up.

Attribute Values

- One of the primary ways your HTML elements distinguish themselves is through their attributes and the values these attributes store.
- For example, the src and alt attributes are what distinguish the following three img elements:

```
<img src="images/lol_panda.png" alt="Sneezing Panda!"/>
<img src="images/cat_cardboard.png" alt="Cat sliding into box!"/>
<img src="images/dog_tail.png" alt="Dog chasing its tail!"/>
```

- Every HTML attribute (including custom ones) can be accessed via JavaScript.
- To help you deal with attributes, your elements expose the `getAttribute` and `setAttribute` methods.
- The `getAttribute` method allows you to specify the name of an attribute on the element it is living on. If the attribute is found, this method will then return the value associated with that attribute.
- See following example:

```
<script>
    var title = document.querySelector("h1");
    alert(title.getAttribute("id"));
</script>
```

- In this snippet, notice that we are getting the value of the `id` attribute on our `h1` element.
  - If you specify an attribute name that doesn't exist, you will get a nice value of `null`.
- You use `setAttribute` on the element that you want to affect and specifying both the attribute name as well as the value that attribute will store.
- Here is an example of `setAttribute` at work:

```
<script>
    document.body.setAttribute("class", "bar foo");
</script>
```

- We are setting the class attribute on the body element to `bar foo`.
- The `setAttribute` function doesn't do any validation to ensure that the attribute you are setting is valid for the element you are setting it on.
- Nothing prevents you from doing something silly as follows:

```
<script>
    document.body.setAttribute("src", "http://www.kirupa.com");
</script>
```

  - The `body` element doesn't contain the `src` attribute, but you can get away with specifying it.
  - When your code runs, your `body` element will sport the `src` attribute...probably very uncomfortably.
- **NOTE:** Because of how common setting `id` and `class` attributes are, your HTML elements expose the `id` and `className` properties directly:

```
<script>
    var title = document.querySelector("h1");
    alert(title.id);
    document.body.className = "bar foo";
</script>
```

Chapter 24: Styling Your Content

A Tale of Two Styling Approaches

1. One way is by setting a CSS property directly on the element.
2. The other way is by adding or removing class values from an element which may result in certain style rules getting applied or ignored.

Setting the Style Directly

- Every HTML element that you access via JavaScript has a `style` object.
    - The `style` object allows you 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` looks like:

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

- To affect many elements, you can do something as follows:

```
var myElements = document.querySelectorAll(".bar");
for (var i = 0; i < myElements.length; i++) {
    myElements[i].style.opacity = 0;
}
```

- To style elements directly using JavaScript, the first step is to access the element.
    - The second step is just to find the CSS property you care about and give it a value.
        - Remember, many values in CSS are actually strings.
        - Also remember that many values require a unit of measurement like `px` or `em` or something like that to actually get recognized.

IMPORTANT NOTES:

- To specify a CSS property in JavaScript that contains a dash, simply remove the dash and capitalize the first letter of the second word.
    - 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.
    - One example of a CSS property that falls into this special category is `float`. In CSS it is a layout property. In JavaScript, it stands for something else.
        - To use a property whose name is entirely reserved, prefix the property with css where `float` becomes `cssFloat`.

Adding and Removing Classes Using `classList`

- A common way to style elements is by adding and removing `class` values on their `class` attribute.
- Let's say we have the following `div` element:

```
<div id="myDiv" class="bar foo zorb"> ... </div>
```

- The `classList` API is great because it is simple and provides the following methods to manipulate class values:
  - `add`
  - `remove`
  - `toggle`
  - `contains`

Adding Class Values

- To add a class value to an element, call the `add` method on `classList`:

```
var divElement = document.querySelector("#myDiv");
divElement.classList.add("baz");
alert(divElement.classList);
```

  - After this code runs, our `div` element will have the following class values: `bar`, `foo`, `zorb`, `baz`.
    - The `classList` API takes care of ensuring that spaces are added between class values and all the other sort of stuff that CSS expects from your HTML content.
    - If you specify an invalid class value, the `classList` API will throw an exception and not add it.
    - If you tell the `add` method to add a class that already exists on the element, your code will run without exception but the duplicate class value will not get added.

Removing Class Values

- To remove a class value, just call the `remove` method on `classList`:

```
var divElement = document.querySelector("#myDiv");
divElement.classList.remove("foo");
alert(divElement.classList);
```

  - After this code executes, the `foo` class value will be removed. What you will be left with is just `bar` and `zorb`.

Toggling Class Values

- The toggle method, as its name implies, adds or removes the specified class value on the element each time it is called:

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

  - In our case, the `foo` class is removed the first time the `toggle` method is called. The second time, the `foo` class is added. The third time, the `foo` class is removed...

Checking Whether a Class Value Exists

- The last thing we are going to look at is the contains method:

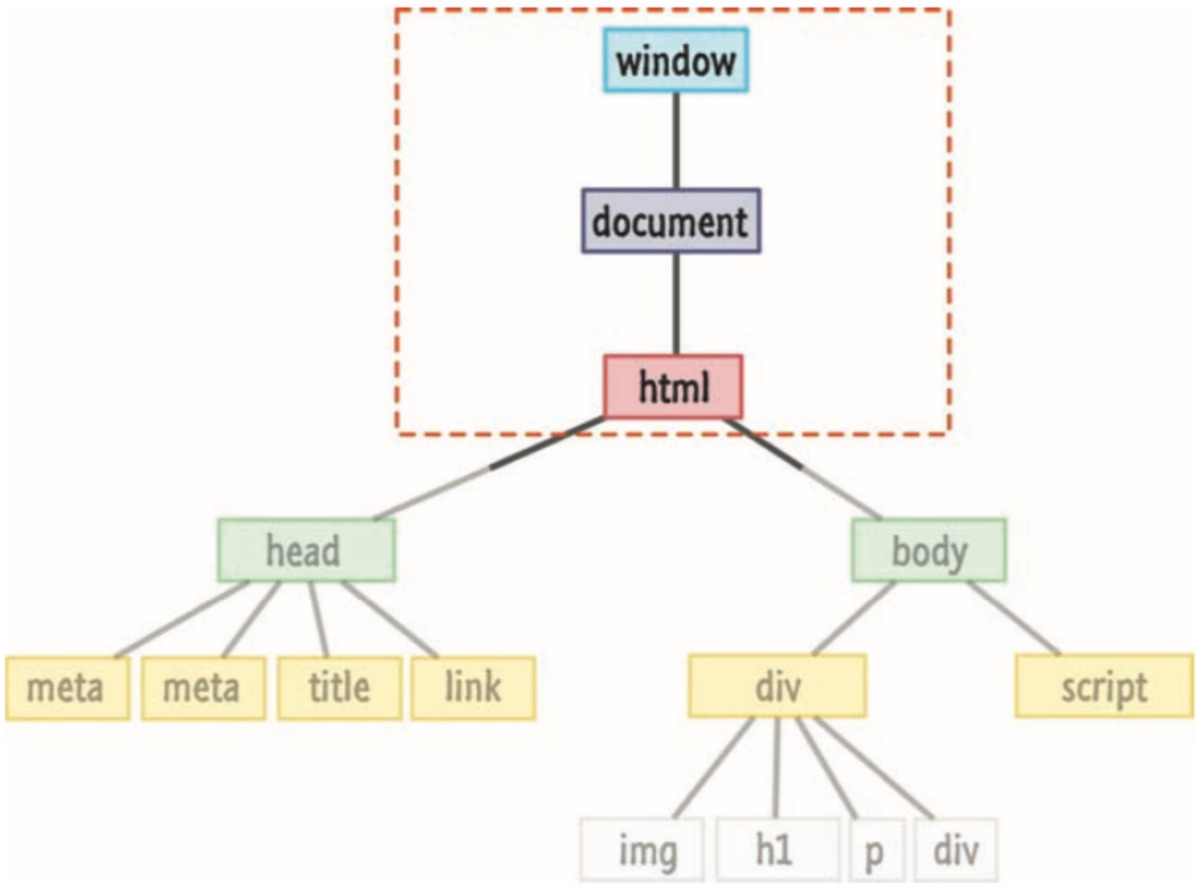
  ```
  var divElement = document.querySelector("#myDiv");
  if (divElement.classList.contains("bar") == true) {
      // do something
  }
  ```

  - This method checks to see if the specified class value exists on the element. If the value exists, you get `true`. If the value doesn't exist, you get `false`.
- More information on using the `classList` API: https://www.kirupa.com/html5/using_the_classlist_api.htm

---

Chapter 25: Traversing The DOM

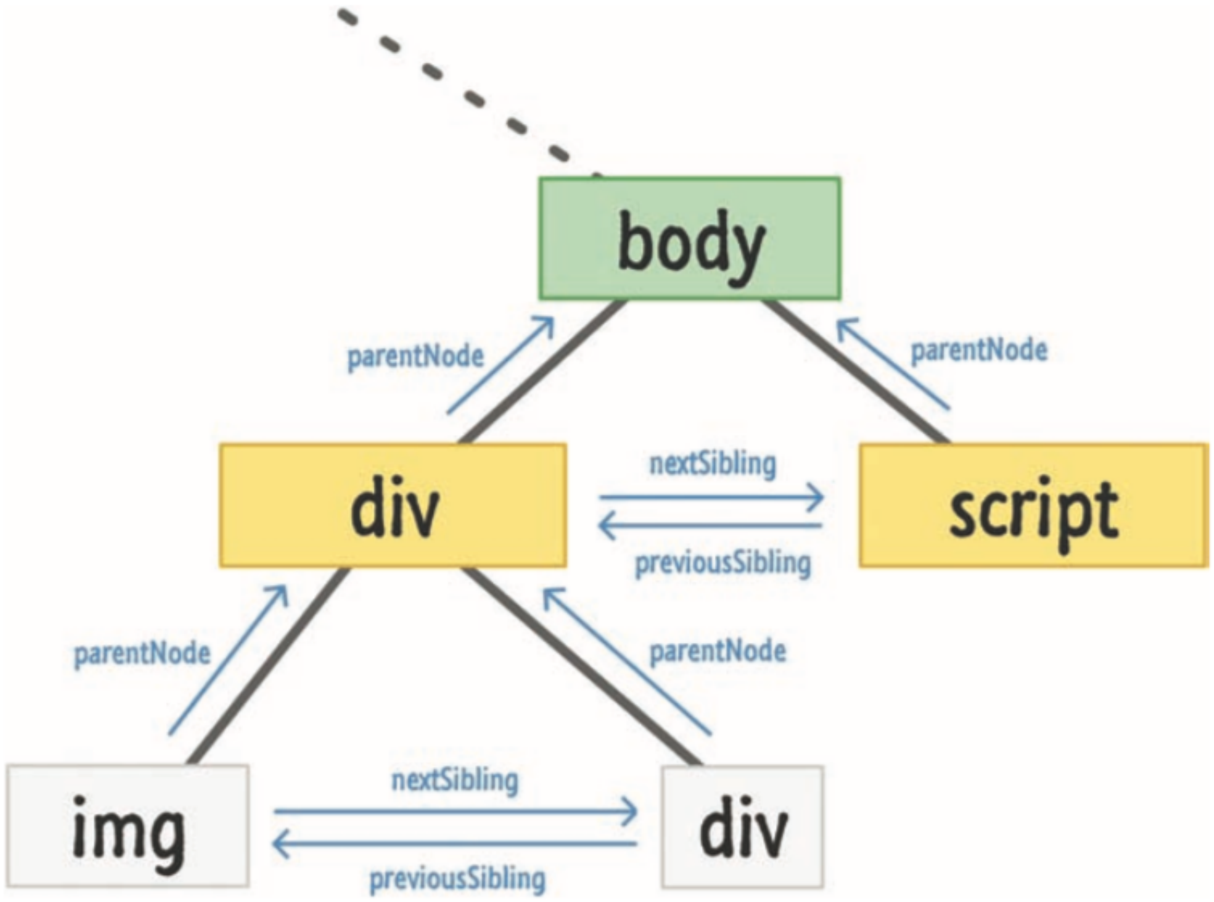
- The view from the top of your DOM is made up of your `window`, `document`, and `html` elements:



- Note that both `window` and `document` are global properties.
- Sometimes, you don't know where you want to go. The `querySelector` and `querySelectorAll` methods won't help you here. You just want to get in the car and drive...and hope you find what you are looking for.
  - In the example diagram above we see that Both the `div` and `script` elements are siblings. The reason they are siblings is because they share the `body` element as their parent.
    - The `script` element has no children, but the `div` element does.
    - The `img`, `h1`, `p`, and `div` are children of the `div` element, and all children of the same parent are siblings as well.
  - Properties including `firstChild`, `lastChild`, `parentNode`, `children`, `previousSibling`, and `nextSibling` exist to help you navigate the DOM when you do no know exactly where you are going.
    - Reference: https://www.w3schools.com/jsref/

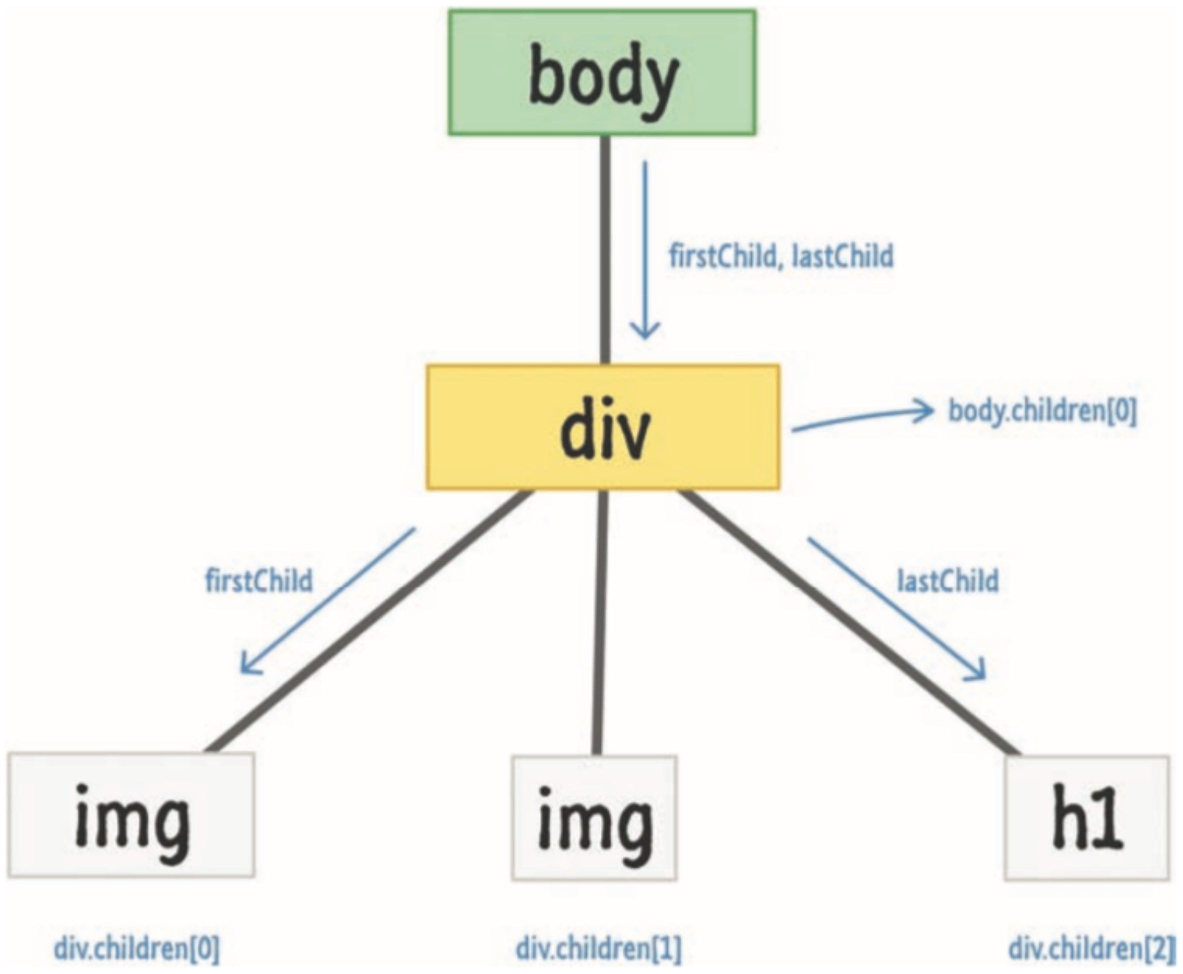
Dealing with Siblings and Parents

- Of the common family properties, the easiest ones to deal with are the parents and siblings. The relevant properties for this are `parentNode`, `previousSibling`, and `nextSibling`:



Dealing with Children

- What is a little less straightforward is how the children fit into all of this, so let's take a look at the `firstChild`, `lastChild`, and `children` by viewing the following diagram:



- When you access the `children` property on a parent, you basically get a collection of the child elements the parent has.
    - This collection is not an Array, but it does have some Array-like powers.
        - Just like with an Array, you can iterate through this collection or access the children individually kind of like what you see in the diagram.
        - This collection also has a `length` property that tells you the count of how many children the parent is dealing with.

Checking Whether a Child Exists

- To check if an element has a child, you can do something like the following:

```
var bodyElement = document.body;
if (bodyElement.firstChild) {
    // do something interesting
}
```

    - This chunk of code will return `null` if there are no children.
    - You could also have used `bodyElement.lastChild` or `bodyElement.children.length` if you enjoy typing.

Accessing All the Child Elements

- If you want to access all of a parent's children, you can always rely on the for loop:

  ```
  var bodyElement = document.body;
  for (var i = 0; i < bodyElement.children.length; i++) {
      var childElement = bodyElement.children[i];

      document.writeln(childElement.tagName);
  }
  ```

  - Notice that I am using the `children` and `length` properties property just like I would an Array. *The thing to remember is that this collection is actually not an Array.*
    - Almost all of the Array methods that you may want to use will not be available in this collection returned by the `children` property.

Walking the DOM

- This snippet recursively walks the DOM and awkwardly runs into every HTML element it can find:

  ```
  function theDOMElementWalker(node) {
      if (node.nodeType == 1) {
          // do something with the node
          node = node.firstChild;
          while (node) {
              theDOMElementWalker(node);
              node = node.nextSibling;
          }
      }
  }
  ```

  - To see this function in action, simply call it by passing in a node that you want to start your walk from:

    ```
    var texasRanger = document.querySelector("#texas");
    theDOMElementWalker(texasRanger);
    ```

Chapter 26: Creating and Removing DOM Elements

Creating Elements

- The way to create elements is by using the `createElement` method.

  - The way `createElement` works is pretty simple. You call it via your document object and pass in the tag name of the element you wish to create.
- In the following snippet, you are creating a paragraph element represented by the letter `p`:

```
var el = document.createElement("p");
```

  - If you run this line of code as part of your app, it will execute and a p element will get created.
  - You need to actually place this element somewhere in the DOM, for your dynamically created `p` element is just floating around aimlessly right now.
- In order for an element to be a part of the DOM, there are two things we need to do:
  1. Find an element that will act as the parent
  2. Use `appendChild` and add the element you want into that parent element
- The following example shows both of these steps in action:

```
<script>
    var newElement = document.createElement("p");
    newElement.textContent = "I exist entirely in your imagination.";
    document.body.appendChild(newElement);
</script>
```

  - Our parent is going to be the `body` element, which is accessed via `document.body`.
  - On the `body` element, we call `appendChild` and pass in an argument to our newly created element, to which a reference with the `newElement` variable is passed.
- **The `appendChild` function always adds the element to the end of whatever children a parent may have.**
- If you want to insert `newElement` directly after a specific tag, you can do so by calling the `insertBefore` function on the parent.
  - The `insertBefore` function takes two arguments.
    1. The first argument is the element you want to insert.
    2. The second argument is a reference to the sibling (aka child of a parent) you want to precede.
- The following example shoes `insertBefore` put to proper use:

```
<script>
    var newElement = document.createElement("p");
    newElement.textContent = "I exist entirely in your imagination.";

    var scriptElement = document.querySelector("script");
    document.body.insertBefore(newElement, scriptElement);
</script>
```

- There exist no built-in `insertAfter` function. However, you can make your own with the following code:

```
function insertAfter(target, newElement) {
    target.parentNode.insertBefore(newElement, target.nextSibling);
}
```

Removing Elements

- Removing elements is also possible with the `removeChild` method. See the following example:

```
<script>
    var newElement = document.createElement("p");
    newElement.textContent = "I exist entirely in your imagination.";
    document.body.appendChild(newElement);

    document.body.removeChild(newElement);
</script>
```

- Now, let's say that you don't have direct access to an element's parent and don't want to waste time finding it.
    - You can still remove that element very easily by using the parentNode property as follows:

    ```
    <script>
        var newElement = document.createElement("p");
        newElement.textContent = "I exist entirely in your imagination.";
        document.body.appendChild(newElement);
        newElement.parentNode.removeChild(newElement);
    </script>
    ```

        - In this variation `newElement` is removed by calling `removeChild` on its parent by specifying `newElement.parentNode`.

Cloning Elements

- The way you clone an element is by calling the `cloneNode` function on the element you wish to clone along with providing a `true` or `false` argument to specify whether you want to clone just the element or the element and all of its children.

```
<script>
    var share = document.querySelector(".share");
```

```
    var shareClone = share.cloneNode(false);
    document.querySelector("#footer").appendChild(shareClone);
</script>
```

Chapter 27: In-Browser Developer Tools

- The Developer Tools provide you with the ability to:
  - Inspect the DOM
  - Debug JavaScript
  - Inspect objects and view messages via the console
  - Figure out performance and memory issues
  - See the network traffic
  - and a whole lot more!
- Resources:
  - https://www.w3schools.com/js/js_debugging.asp
    - More about the JavaScript Debugger
  - https://developer.mozilla.org/en-US/docs/Tools/Debugger
    - Firefox JavaScript Debugguer help
  - https://www.kirupa.com/html5/examples/randomColorGenerator.htm
    - Demo page to inspect from book
