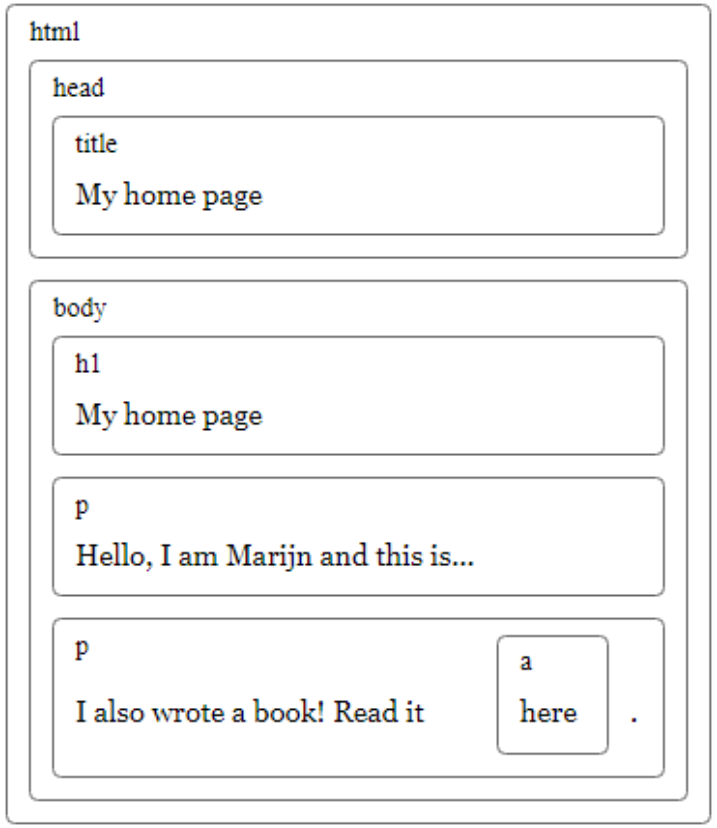
The Document Object Model

Document structure



- imagine an HTML document as a nested set of boxes

```
<!doctype html>
<html>
  <head>
    <title>My home page</title>
  </head>
  <body>
    <h1>My home page</h1>
    <p>Hello, I am Marijn and this is my home page.</p>
    <p>I also wrote a book! Read it
      <a href="http://marijinbook.com">here</a>.</p>
  </body>
</html>
```

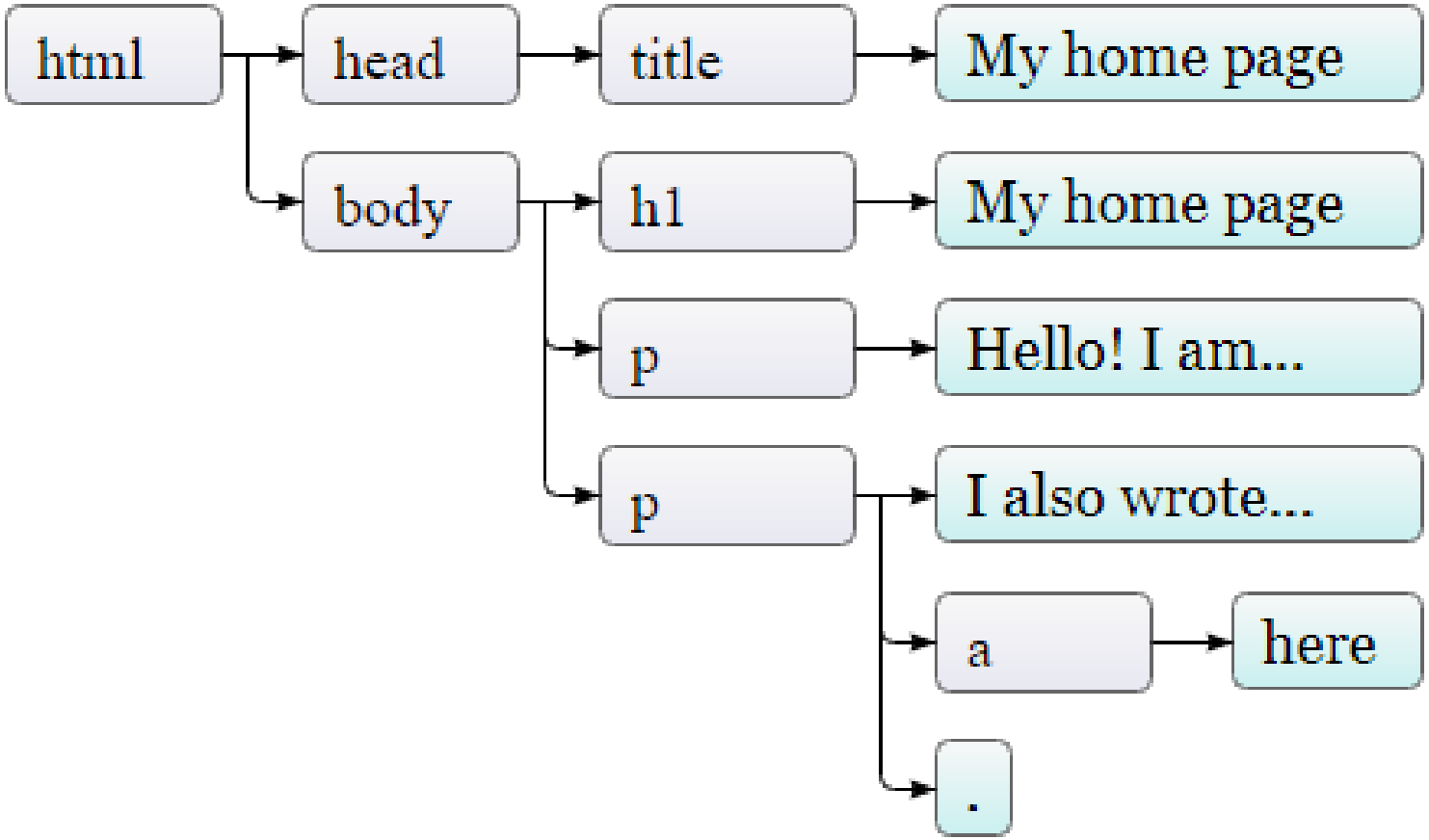
Document Structure (cont.)

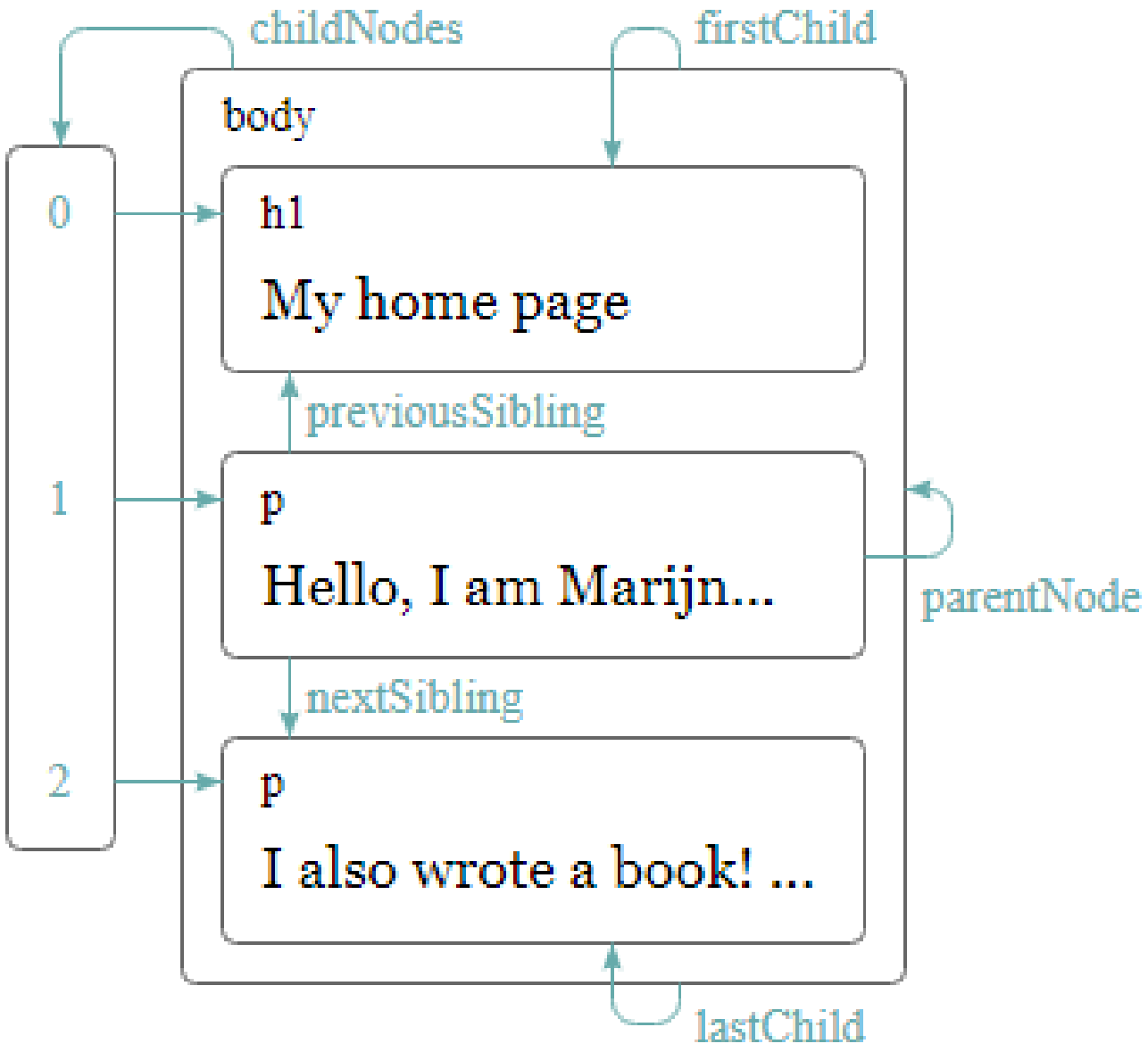
- For each box, there is an object
  - which we can interact with to find out things such as what HTML tag it represents and which boxes and text it contains.
- This representation is called the *Document Object Model*, or DOM for short.
- The global binding `document` gives us access to these objects.
  - Its `documentElement` property refers to the object representing the <html> tag.
  - Since every HTML document has a head and a body, it also has `head` and `body` properties, pointing at those elements.

Node Type

- Each DOM node object has a `nodeType` property
- `nodeType` property contains a code (number) that identifies the type of node.
  - Elements have **code 1**, which is also defined as the constant property `Node.ELEMENT_NODE`
  - Text nodes, representing a section of text in the document, get **code 3** (`Node.TEXT_NODE`)
  - Comments have **code 8** (`Node.COMMENT_NODE`)

DOM as a Tree


html
head
title
My home page
body
h1
My home page
p
Hello! I am...
p
I also wrote...
a
here
.


childNodes
firstChild
body
0
h1
My home page
previousSibling
1
p
Hello, I am Marijn...
parentNode
nextSibling
2
p
I also wrote a book! ...
lastChild

children Property

- `children` property is like `childNodes` but contains only element (type 1) children, not other types of child nodes.
- This can be useful when you aren’t interested in text nodes.

Finding Elements

- Navigating these links among parents, children, and siblings is often useful.
- But if we want to find a specific node in the document, reaching it by starting at document.`body` and following a fixed path of properties is a bad idea.
  - Doing so bakes assumptions into our program about the precise structure of the document—a structure you might want to change later.
- Another complicating factor is that text nodes are created even for the whitespace between nodes.
  - The example document's <body> tag does not have just three children (<h1> and two <p> elements) but actually has seven: those three, plus the spaces before, after, and between them.

Finding Elements (cont.)

- So if we want to get the `href` attribute of the link in that document,
  - we don't want to say something like "Get the second child of the sixth child of the document body".
  - It'd be better if we could say "Get the first link in the document". And we can.

```
let link = document.body.getElementsByTagName("a")[0];
console.log(link.href)
```

Finding Elements (cont.)

- All element nodes have a `getElementsByTagName` method, which collects all elements with the given tag name that are descendants (direct or indirect children) of that node and returns them as an array-like object.

- To find a specific single node, you can give it an id attribute and use document.`getElementById` instead.

```
<p>My ostrich Gertrude:</p>
<p><img id="gertrude" src="img/ostrich.png"></p>
<script>
  let ostrich = document.getElementById("gertrude");
  console.log(ostrich.src);
</script>
```

- A third, similar method is getElementsByClassName, which, like getElementsByTagName, searches through the contents of an element node and retrieves all elements that have the given string in their class attribute.

Changing The Document

- Almost everything about the DOM data structure can be changed.
- The shape of the document tree can be modified by changing parent-child relationships.
- Nodes have a remove method to remove them from their current parent node.
- To add a child node to an element node, we can use `appendChild`,
  - which puts it at the end of the list of children,
  - or `insertBefore`, which inserts the node given as the first argument before the node given as the second argument.

Changing The Document (cont.)

```
<p>One</p>
<p>Two</p>
<p>Three</p>

<script>
  let paragraphs =
document.body.getElementsByTagName("p");
  document.body.insertBefore(paragraphs[2],
paragraphs[0]);
</script>
```

- A node can exist in the document in only one place.
  - Thus, inserting paragraph Three in front of paragraph One will first remove it from the end of the document and then insert it at the front, resulting in Three/One/Two.
  - All operations that insert a node somewhere will, as a side effect, cause it to be removed from its current position (if it has one).

Changing The Document (cont.)

- The `replaceChild` method is used to replace a child node with another one.
- It takes as arguments two nodes: a new node and the node to be replaced.
- The replaced node must be a child of the element the method is called on.
- Note that both `replaceChild` and `insertBefore` expect the new node as their first argument.

Creating Nodes

- Say we want to write a script that replaces all images (<`img`> tags) in the document with the text held in their alt attributes, which specifies an alternative textual representation of the image.
  - This involves not only removing the images but adding a new text node to replace them. Text nodes are created with the document.`createTextNode` method.

```
<p>The <img src="img/cat.png" alt="Cat"> in the
  <img src="img/hat.png" alt="Hat">.</p>

<p><button onclick="replaceImages()">Replace</button></p>

<script>
  function replaceImages() {
    let images = document.body.getElementsByTagName("img");
    for (let i = images.length - 1; i >= 0; i--) {
      let image = images[i];
      if (image.alt) {
        let text = document.createTextNode(image.alt);
        image.parentNode.replaceChild(text, image);
      }
    }
  }
</script>
```

Creating Nodes (cont.)

- Given a string, createTextNode gives us a text node that we can insert into the document to make it show up on the screen.
- The loop that goes over the images starts at the end of the list.
  - This is necessary because the node list returned by a method like getElementsByTagName (or a property like childNodes) is live.
    - That is, it is updated as the document changes.
- If we started from the front, removing the first image would cause the list to lose its first element so that the second time the loop repeats, where i is 1, it would stop because the length of the collection is now also 1.

Creating Nodes (cont.)

- If you want a solid collection of nodes, as opposed to a live one, you can convert the collection to a real array by calling `Array.from`.

```
let arrayish = {0: "one", 1: "two", length: 2};
let array = Array.from(arrayish);
console.log(array.map(s => s.toUpperCase()));
// → ["ONE", "TWO"]
```

Creating Nodes (cont.)

- To create element nodes, you can use the document.`createElement` method.
  - This method takes a tag name and returns a new empty node of the given type.
- The following example defines a utility elt, which creates an element node and treats the rest of its arguments as children to that node.
  - This function is then used to add an attribution to a quote.

```
<blockquote id="quote">
  No book can ever be finished. While working on it we learn
  just enough to find it immature the moment we turn away
  from it.
</blockquote>
```

Creating Nodes (cont.)

```
<script>
  function elt(type, ...children) {
    let node = document.createElement(type);
    for (let child of children) {
      if (typeof child != "string") node.appendChild(child);
      else node.appendChild(document.createTextNode(child));
    }
    return node;
  }

  document.getElementById("quote").appendChild(
    elt("footer", "—",
        elt("strong", "Karl Popper"),
        ", preface to the second edition of ",
        elt("em", "The Open Society and Its Enemies"),
        ", 1950"));
</script>
```

Example: createElement() Method

Initially:

After pressing Create Paragraph button:

Example: createElement() Method (cont.)

```html
<!DOCTYPE html>
<html>

<head>
    <script>
        function createparagraph() {
            var x = document.createElement("p");
            var t =
                document.createTextNode("Paragraph is created.");
            x.appendChild(t);
            document.body.appendChild(x);
        }
    </script>
</head>

<body>
    <button onclick="createparagraph()">CreateParagraph</button>
</body>

</html>
```

Example: removeChild() Method

DOM removeChild() Method

Sorting Algorithm

- Insertion sort
- Merge sort
- Quick sort

Click Here!

After click on the button:

DOM removeChild() Method

Sorting Algorithm

- Merge sort
- Quick sort

Click Here!

Example: removeChild() Method (cont.)

```html
<!DOCTYPE html>
<html>
    <head>
        <title>
            HTML DOM removeChild() Method
        </title>
    </head>

    <body>

        <h1 style="color: green;">
            GeeksforGeeks
        </h1>

        <h2>
            DOM removeChild() Method
        </h2>

        <p>Sorting Algorithm</p>
        <ul id = "listitem"><li>Insertion sort</li>
            <li>Merge sort</li>
            <li>Quick sort</li>
        </ul>

        <button onclick = "Geeks()">
            Click Here!
        </button>

        <script>
            function Geeks() {
                var doc = document.getElementById("listitem");
                doc.removeChild(doc.childNodes[0]);
            }
        </script>

    </body>
</html>
```

After Click on the button:

Welcome To GeeksforGeeks

HTML DOM insertBefore() Method

- C++
- Java
- Python

Click on the button to insert an element before Python

Insert Node

Before Click on the button:

Welcome To GeeksforGeeks

HTML DOM insertBefore() Method

- C++
- Python

Click on the button to insert an element before Python

Insert Node

Example: insertBefore() Method

```html
<!DOCTYPE html>
<html>
    <head>
        <title>
            HTML DOM removeChild() Method
        </title>
    </head>

    <body>

        <h1 style="color: green;">
            GeeksforGeeks
        </h1>

        <h2>
            DOM removeChild() Method
        </h2>

        <p>Sorting Algorithm</p>
        <ul id = "listitem"><li>Insertion sort</li>
            <li>Merge sort</li>
            <li>Quick sort</li>
        </ul>

        <button onclick = "Geeks()">
            Click Here!
        </button>

        <script>
            function Geeks() {
                var doc = document.getElementById("listitem");
                doc.removeChild(doc.childNodes[0]);
            }
        </script>

    </body>
</html>
```

Example: insertBefore() Method (cont.)

Before click on the button:

GeeksforGeeks

DOM replaceChild() Method

Sorting Algorithm

- Insertion sort
- Merge sort
- Bubble sort

Click Here!

After click on the button:

GeeksforGeeks

DOM replaceChild() Method

Sorting Algorithm

- Quick sort
- Merge sort
- Bubble sort

Click Here!

```
<h1 style="color: green;">
    GeeksforGeeks
</h1>

<h2>
    DOM replaceChild() Method
</h2>

<p>Sorting Algorithm</p>
<ul id = "listitem"><li>Insertion sort</li>
    <li>Merge sort</li>
    <li>Bubble sort</li>
</ul>

<button onclick ="Geeks()">
    Click Here!
</button>

<script>
function Geeks() {
    var doc = document.createTextNode("Quick sort");
    var list = document.getElementById("listitem").childNodes[0];
    list.replaceChild(doc, list.childNodes[0]);
}
</script>
```

Attributes

- Some element attributes, such as href for links, can be accessed through a property of the same name on the element's DOM object.
  - This is the case for most commonly used standard attributes.
- But HTML allows you to set any attribute you want on nodes.
  - This can be useful because it allows you to store extra information in a document.
- If you make up your own attribute names, though, such attributes will not be present as properties on the element's node.
  - Instead, you have to use the `getAttribute` and `setAttribute` methods to work with them.

Attributes (cont.)

```
<p data-classified="secret">The launch code is 00000000.</p>
<p data-classified="unclassified">I have two feet.</p>

<script>
  let paras = document.body.getElementsByTagName("p");
  for (let para of Array.from(paras)) {
    if (para.getAttribute("data-classified") == "secret") {
      para.remove();
    }
  }
</script>
```

Attributes (cont.)

- It is recommended to prefix the names of such made-up attributes with `data-` to ensure they do not conflict with any other attributes.

Layout

- You may have noticed that different types of elements are laid out differently.
  - Some, such as paragraphs (<p>) or headings (<h1>), take up the whole width of the document and are rendered on separate lines.
    - These are called block elements.
  - Others, such as links (<a>) or the element, are rendered on the same line with their surrounding text.
    - Such elements are called inline elements.
- For any given document, browsers are able to compute a layout, which gives each element a size and position based on its type and content.
  - This layout is then used to actually draw the document.

Layout (cont.)

- The size and position of an element can be accessed from JavaScript.
- The `offsetWidth` and `offsetHeight` properties give you the space the element takes up in pixels.
  - A pixel is the basic unit of measurement in the browser.
  - It traditionally corresponds to the smallest dot that the screen can draw, but on modern displays, which can draw very small dots, that may no longer be the case, and a browser pixel may span multiple display dots.
- Similarly, `clientWidth` and `clientHeight` give you the size of the space inside the element, ignoring border width.

Layout (cont.)

```
<p style="border: 3px solid red">
  I'm boxed in
</p>

<script>
  let para =
document.body.getElementsByTagName("p")[0];
  console.log("clientHeight:", para.clientHeight);
  console.log("offsetHeight:", para.offsetHeight);
</script>
```

```
<p><span id="one"></span></p>
<p><span id="two"></span></p>

<script>
  function time(name, action) {
    let start = Date.now(); // Current time in milliseconds
    action();
    console.log(name, "took", Date.now() - start, "ms");
  }

  time("naive", () => {
    let target = document.getElementById("one");
    while (target.offsetWidth < 2000) {
      target.appendChild(document.createTextNode("X"));
    }
  });
  // → naive took 32 ms

  time("clever", function() {
    let target = document.getElementById("two");
    target.appendChild(document.createTextNode("XXXXX"));
    let total = Math.ceil(2000 / (target.offsetWidth / 5));
    target.firstChild.nodeValue = "X".repeat(total);
  });
  // → clever took 1 ms
</script>
```

Styling

- We have seen that different HTML elements are drawn differently.
  - Some are displayed as blocks, others inline.
  - Some add styling—`<strong>` makes its content bold, and <a> makes it blue and underlines it.
- The way an `<img>` tag shows an image or an `<a>` tag causes a link to be followed when it is clicked is strongly tied to the element type.
- But we can change the styling associated with an element, such as the text color or underline.
  - Here is an example that uses the style property:

```
<p><a href=".">Normal link</a></p>
<p><a href="." style="color: green">Green link</a></p>
```

Styling (cont.)

- A style attribute may contain one or more declarations, which are a property (such as color) followed by a colon and a value (such as green).
  - When there is more than one declaration, they must be separated by semicolons, as in "`color: red; border: none`".
- A lot of aspects of the document can be influenced by styling.
  - For example, the display property controls whether an element is displayed as a block or an inline element.

```
This text is displayed <strong>inline</strong>,
<strong style="display: block">as a block</strong>, and
<strong style="display: none">not at all</strong>.
```

Styling (cont.)

- The block tag will end up on its own line since block elements are not displayed inline with the text around them.
- The last tag is not displayed at all—`display: none` prevents an element from showing up on the screen.
  - This is a way to hide elements.
  - It is often preferable to removing them from the document entirely because it makes it easy to reveal them again later.

Styling (cont.)

- JavaScript code can directly manipulate the style of an element through the element's style property.
  - This property holds an object that has properties for all possible style properties.
  - The values of these properties are strings, which we can write to in order to change a particular aspect of the element's style.

```
<p id="para" style="color: purple">
  Nice text
</p>
<script>
  let para = document.getElementById("para");
  console.log(para.style.color);
  para.style.color = "magenta";
</script>
```

Styling (cont.)

- Some style property names contain hyphens, such as `font-family`.
- Because such property names are awkward to work with in JavaScript (you'd have to say style["font-family"]), the property names in the style object for such properties have their hyphens removed and the letters after them capitalized (`style.fontFamily`).

Cascading Style

Now ***strong text*** is italic and gray.

Now **strong text** is italic and gray.

Cascading Style (cont.)

- The cascading in the name refers to the fact that multiple such rules are combined to produce the final style for an element.
  - In the example, the default styling for `<strong>` tags, which gives them `font-weight: bold`, is overlaid by the rule in the `<style>` tag, which adds `font-style` and `color`.
- When multiple rules define a value for the same property, the most recently read rule gets a higher precedence and wins.
  - So if the rule in the <style> tag included font-weight: normal, contradicting the default font-weight rule, the text would be normal, not bold.
  - Styles in a style attribute applied directly to the node have the highest precedence and always win.

Cascading Style (cont.)

- It is possible to target things other than tag names in CSS rules.
  - A rule for .abc applies to all elements with "abc" in their class attribute.
  - A rule for #xyz applies to the element with an id attribute of "xyz" (which should be unique within the document).

```
.subtle {
  color: gray;
  font-size: 80%;
}
#header {
  background: blue;
  color: white;
}
/* p elements with id main and with classes a and b */
p#main.a.b {
  margin-bottom: 20px;
}
```

Cascading Style (cont.)

- The precedence rule favoring the most recently defined rule applies only when the rules have the same specificity.
- A rule's specificity is a measure of how precisely it describes matching elements, determined by the number and kind (tag, class, or ID) of element aspects it requires.
  - For example, a rule that targets `p.a` is more specific than rules that target `p` or just `.a` and would thus take precedence over them.
- The notation `p > a {…}` applies the given styles to all `<a>` tags that are direct children of `<p>` tags.
  - Similarly, `p a {…}` applies to all `<a>` tags inside `<p>` tags, whether they are direct or indirect children.

Query Selectors

- The notation used in style sheets to determine which elements a set of styles apply to—is that we can use this same mini-language as an effective way to find DOM elements.
- The `querySelectorAll` method, which is defined both on the document object and on element nodes, takes a selector string and returns a `NodeList` containing all the elements that it matches.

Query Selectors (cont.)

```
<p>And if you go chasing <span class="animal">rabbits</span></p>
<p>And you know you're going to fall</p>
<p>Tell 'em a <span class="character">hookah smoking
  <span class="animal">caterpillar</span></span></p>
<p>Has given you the call</p>
<script>
  function count(selector) {
    return document.querySelectorAll(selector).length;
  }
  console.log(count("p"));            // All <p> elements
  console.log(count(".animal"));      // Class animal
  console.log(count("p .animal"));    // Animal inside of <p>
  console.log(count("p > .animal")); // Direct child of <p>
</script>
```

Query Selectors (cont.)

- Unlike methods such as `getElementsByTagName`, the object returned by `querySelectorAll` is not live.
  - It won't change when you change the document.
  - It is still not a real array, though, so you still need to call `Array.from` if you want to treat it like one.
- The `querySelector` method (without the `All` part) works in a similar way.
  - This one is useful if you want a specific, single element. It will return only the first matching element or null when no element matches.