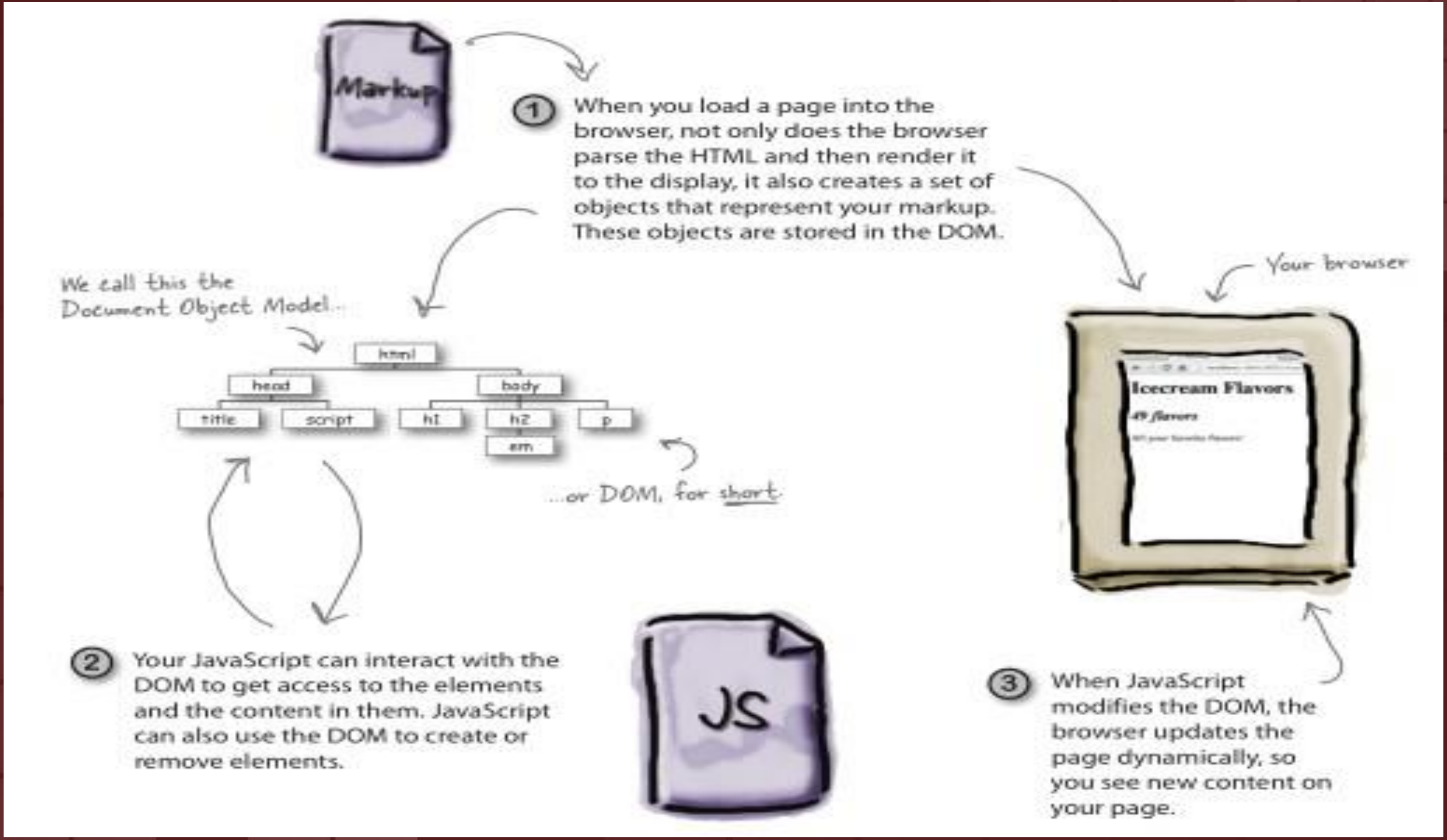
Web Technologies II

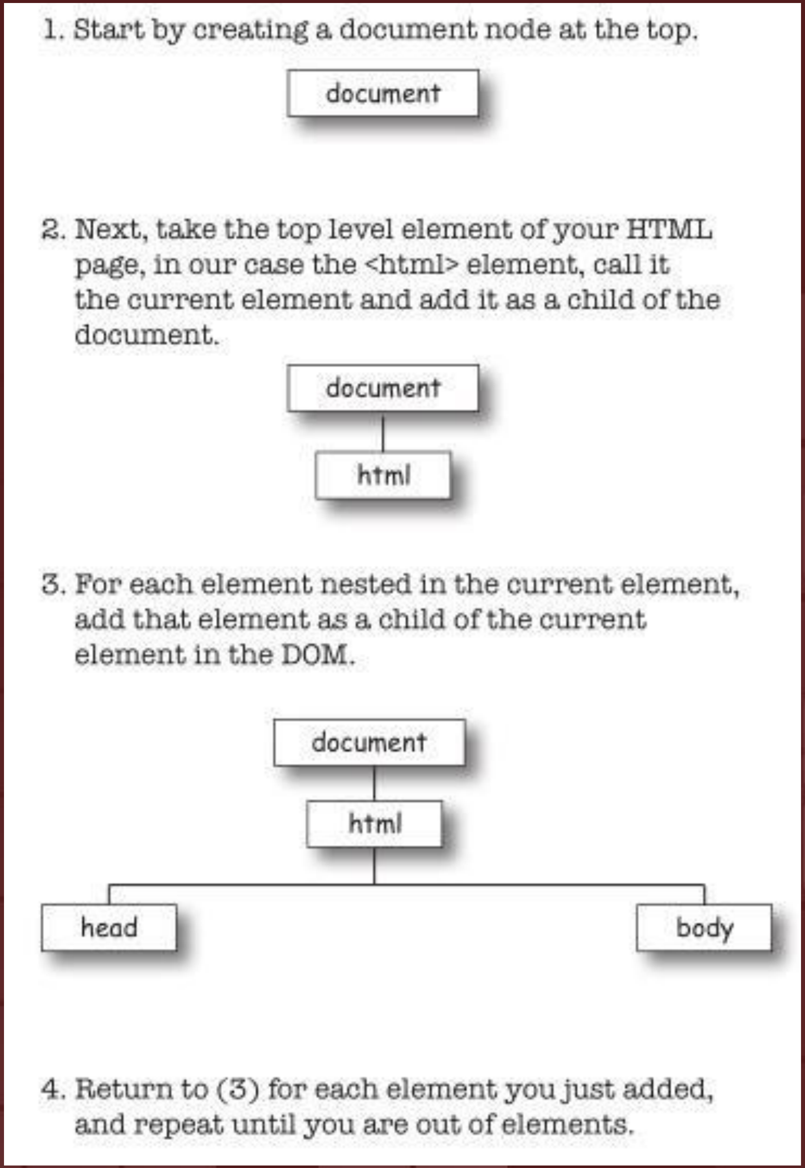
Document Object Model (DOM) – Part A

JavaScript and HTML are two different things. HTML is markup and JavaScript is code. So how do they interact? It all happens through a representation of your page, called the document object model, or the DOM for short. Where does the DOM come from? It's created when the browser loads your page. Here's how:



Markup
1 When you load a page into the browser, not only does the browser parse the HTML and then render it to the display, it also creates a set of objects that represent your markup. These objects are stored in the DOM.
We call this the Document Object Model...
html
head
body
title
script
h1
h2
p
em
...or DOM, for short.
Your browser
Icecream Flavors
2 Your JavaScript can interact with the DOM to get access to the elements and the content in them. JavaScript can also use the DOM to create or remove elements.
JS
3 When JavaScript modifies the DOM, the browser updates the page dynamically, so you see new content on your page.

Let's take some markup and create a DOM for it. Here's a simple recipe for doing that:

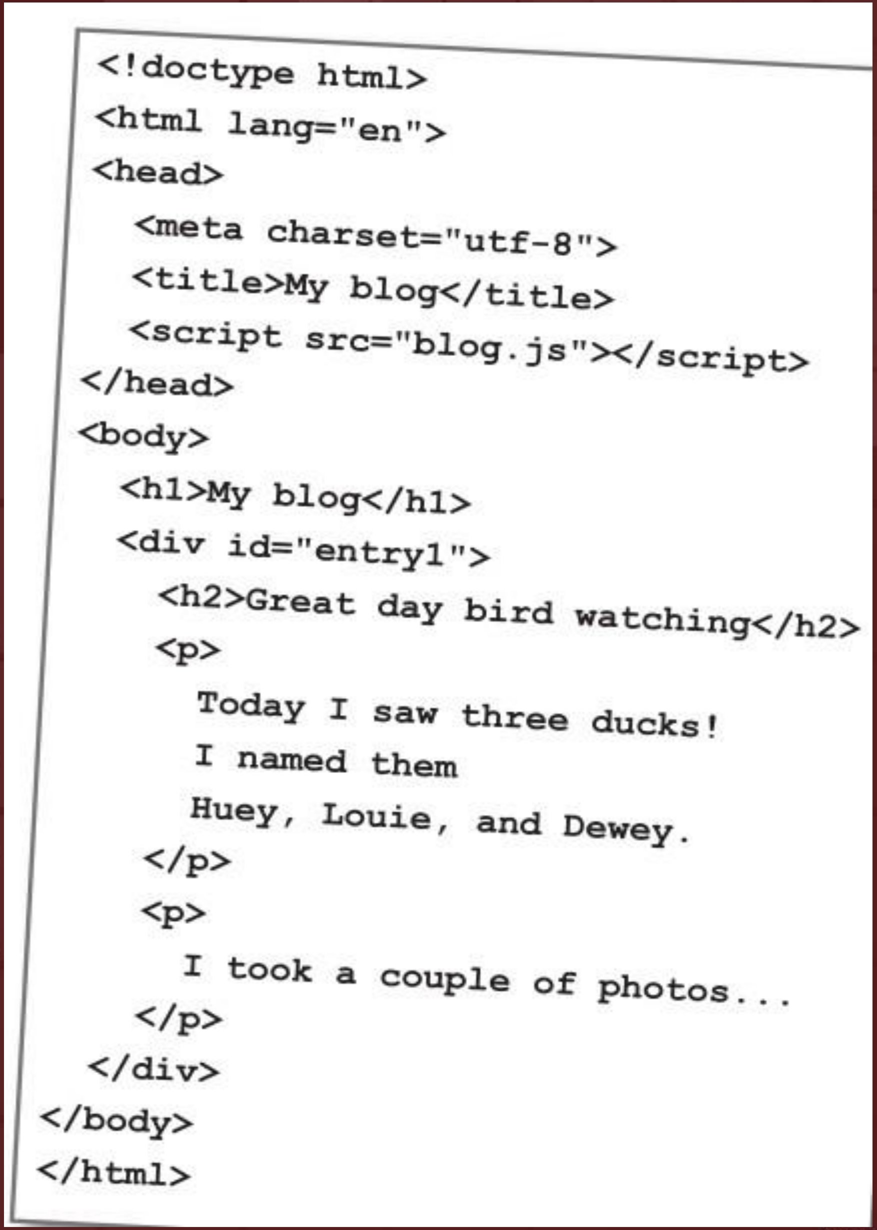


1. Start by creating a document node at the top.

2. Next, take the top level element of your HTML page, in our case the <html> element, call it the current element and add it as a child of the document.

3. For each element nested in the current element, add that element as a child of the current element in the DOM.

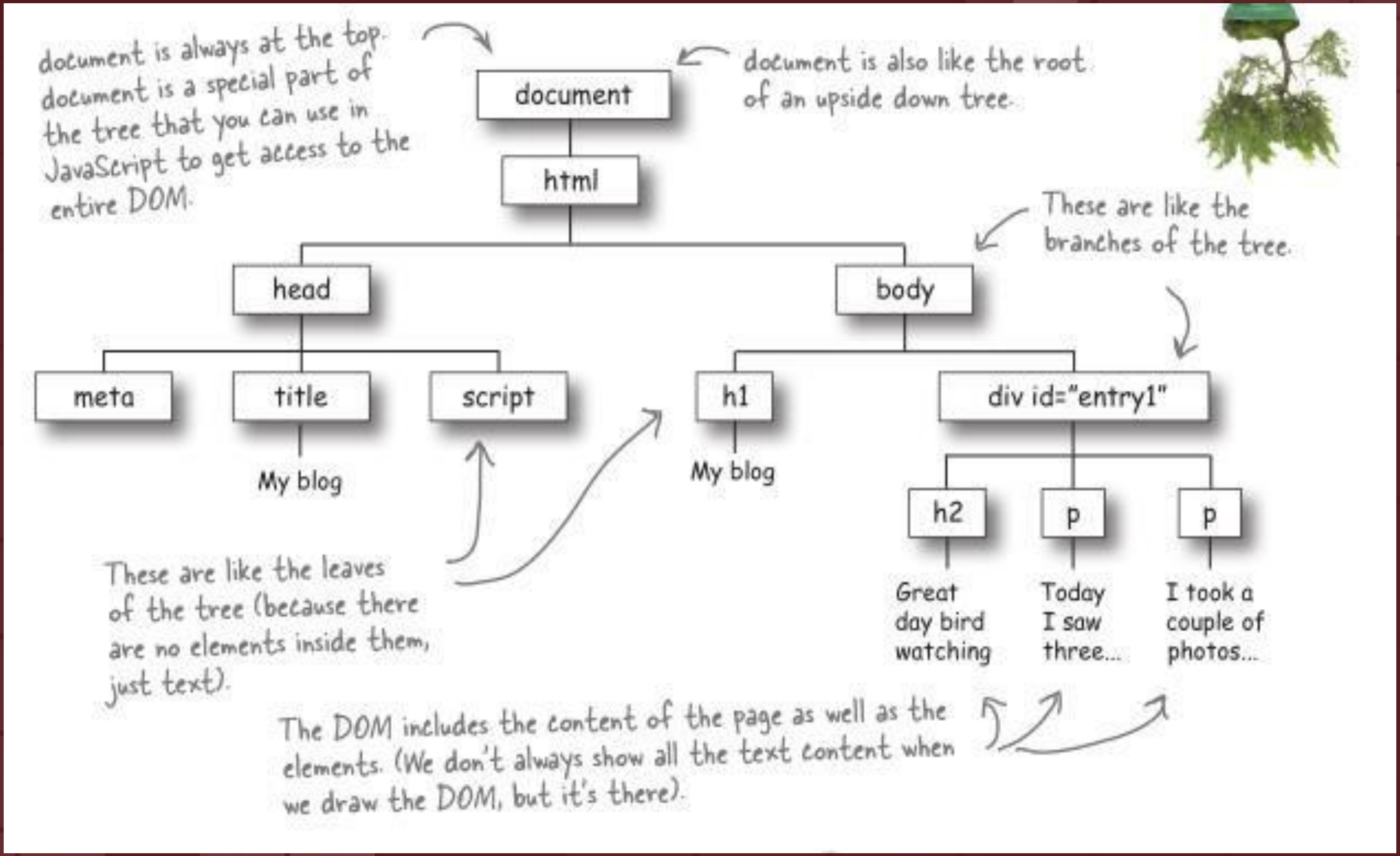
4. Return to (3) for each element you just added, and repeat until you are out of elements.



```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>My blog</title>
  <script src="blog.js"></script>
</head>
<body>
  <h1>My blog</h1>
  <div id="entry1">
    <h2>Great day bird watching</h2>
    <p>
      Today I saw three ducks!
      I named them
      Huey, Louie, and Dewey.
    </p>
    <p>
      I took a couple of photos...
    </p>
  </div>
</body>
</html>
```

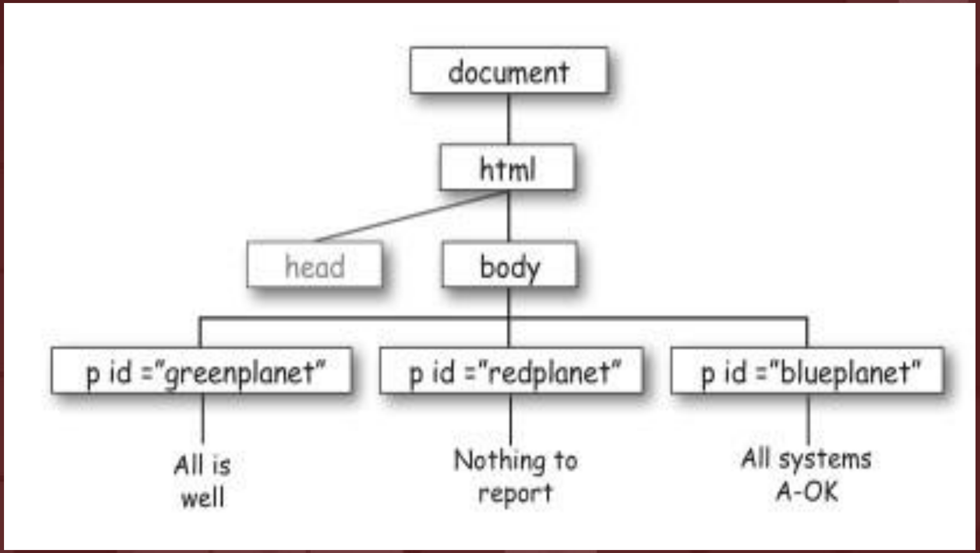
A first taste of the DOM

If you follow the recipe for creating a DOM you'll end up with a structure like the one below. Every DOM has a document object at the top and then a tree complete with branches and leaf nodes for each element in the HTML markup. Let's take a closer look.

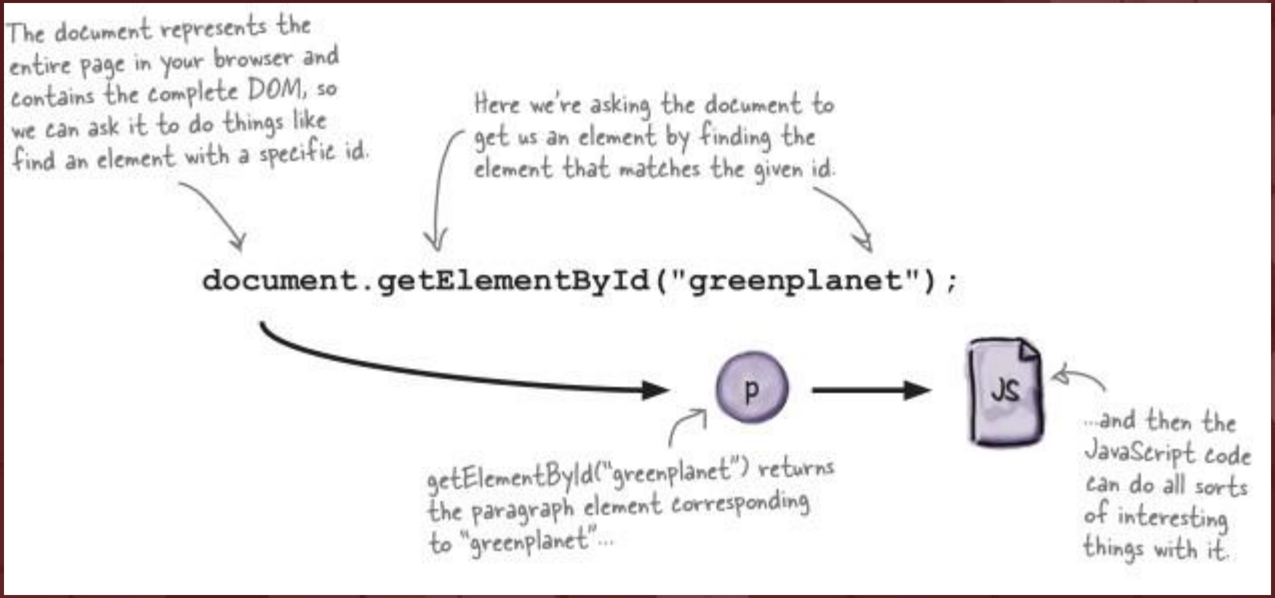


document is always at the top. document is a special part of the tree that you can use in JavaScript to get access to the entire DOM.
document
document is also like the root of an upside down tree.
html
These are like the branches of the tree.
head
body
meta
title
script
h1
div id="entry1"
My blog
My blog
h2
p
p
Great day bird watching
Today I saw three...
I took a couple of photos...
These are like the leaves of the tree (because there are no elements inside them, just text).
The DOM includes the content of the page as well as the elements. (We don't always show all the text content when we draw the DOM, but it's there).

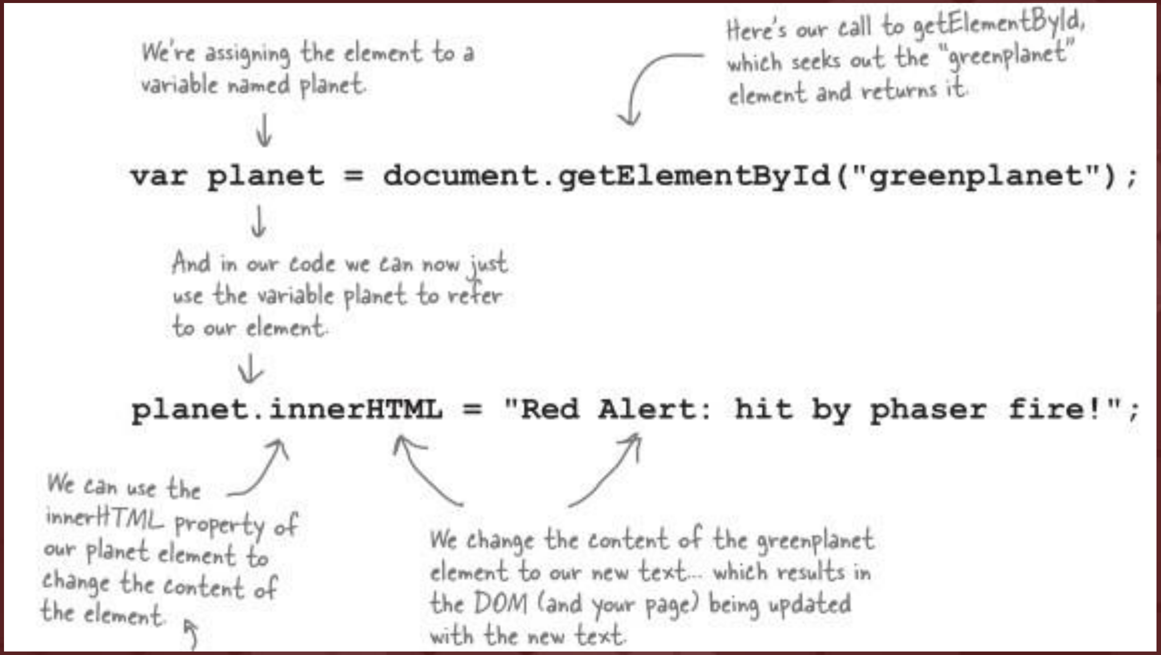
Let’s start with a DOM. Here’s a simple DOM; it’s got a few HTML paragraphs, each with an id identifying it as the green, red or blue planet. Each paragraph has some text as well. Of course there’s a <head> element too, but we’ve left the details out to keep things simpler.



Now let’s use JavaScript to make things more interesting. Let’s say we want to change the greenplanet’s text from “All is well” to “Red Alert: hit by phaser fire!” Down the road you might want to do something like this based on a user’s actions, or even based on data from a web service. We'll get to all that; for now let's just get the greenplanet's text updated. To do that we need the element with the id "greenplanet". Here's some code that does that:


```
document.getElementById("greenplanet");
```
The document represents the entire page in your browser and contains the complete DOM, so we can ask it to do things like find an element with a specific id.
Here we're asking the document to get us an element by finding the element that matches the given id.
p
JS
getElementById("greenplanet") returns the paragraph element corresponding to "greenplanet"...
...and then the JavaScript code can do all sorts of interesting things with it.

Once getElementById gives you an element, you're ready do something with it (like change its text to "Red Alert: hit by phaser fire!"). To do that, we typically assign the element to a variable so we can refer to the element thoughout our code. Let's do that and then change the text:


We're assigning the element to a variable named planet.
Here's our call to getElementById, which seeks out the "greenplanet" element and returns it.
And in our code we can now just use the variable planet to refer to our element.
We can use the innerHTML property of our planet element to change the content of the element.
We change the content of the greenplanet element to our new text... which results in the DOM (and your page) being updated with the new text.

```
var planet = document.getElementById("greenplanet");

planet.innerHTML = "Red Alert: hit by phaser fire!";
```

Getting an element with getElementById

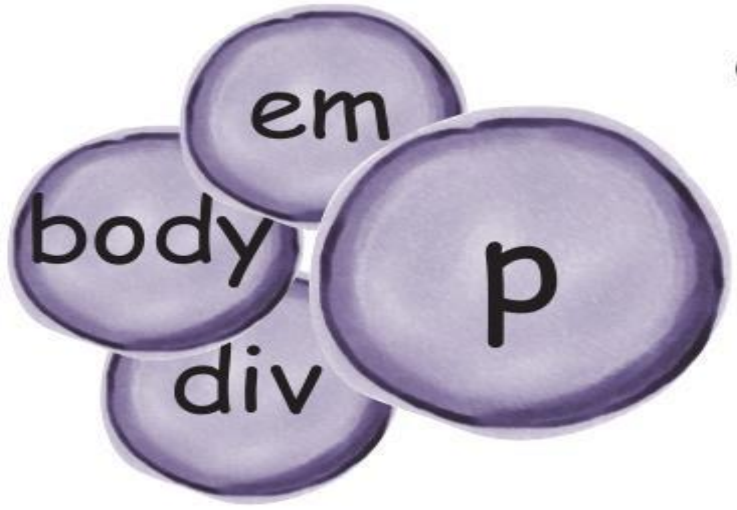
So, what did we just do? Let's step through it in a little more detail. We're using the document object to get access to the DOM from our code. The document object is a built-in object that comes with a bunch of properties and methods, including getElementById , which we can use to grab an element from the DOM. The getElementById method takes an id and returns the element that has that id

Now in the past you've probably used ids to select and style elements with CSS.But here, what we're doing is using an id to grab an element—the <p> element with the id "greenplanet"—from the DOM.

What, exactly, am I getting from the DOM?

When you grab an element from the DOM using getElementById , what you get is an element object, which you can use to read, change or replace the element's content and attributes. And here's the magic: when you change an element, you change what is displayed in your page as well.

But, first things first. Let's take another look at the element object we just grabbed from the DOM. We know that this element object represents the <p> element in our page that has the id "greenplanet" and that the text content in the element is "All is well". Just like other kinds of JavaScript objects, an element object has properties and methods. In the case of an element object, we can use these properties and methods to read and change the element. Here are a few things you can do with element objects:



Get the content (text or HTML).

Change the content.

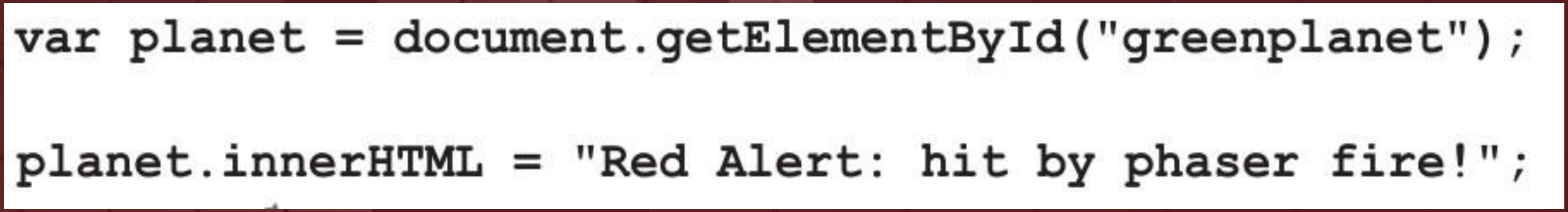
Read an attribute.

Add an attribute.

Change an attribute.

Remove an attribute.

What we want to do with our <p> element—which, remember, is the <p> element with the id “greenplanet”—is change the content “All is well” to “Red Alert: hit by phaser fire!”. We’ve got the element object stashed in the planet variable in our code; let’s use that to modify one of its properties, innerHTML

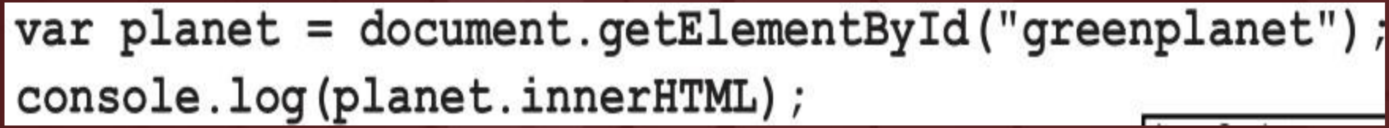


```
var planet = document.getElementById("greenplanet");

planet.innerHTML = "Red Alert: hit by phaser fire!";
```

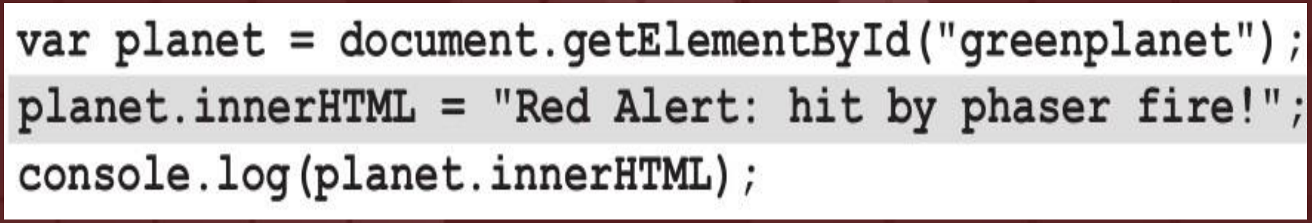
Finding your inner HTML

The innerHTML property is an important property that we can use to read or replace the content of an element. If you look at the value of innerHTML then you'll see the content contained within the element, not including the HTML element tags. The "withIN" is why it's called "inner" HTML. Let's try a little experiment. We'll try displaying the content of the planet element object in the console by logging the innerHTML property. Here's what we get:

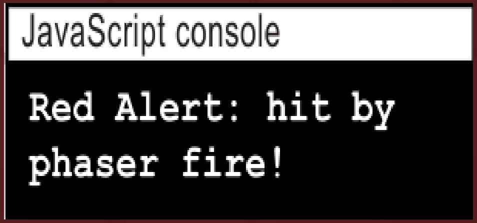


```
var planet = document.getElementById("greenplanet");
console.log(planet.innerHTML);
```

Now let's try changing the value of the innerHTML property. When we do this, we're changing the content of the "greenplanet" <p> element in the page, so you'll see your page change too!



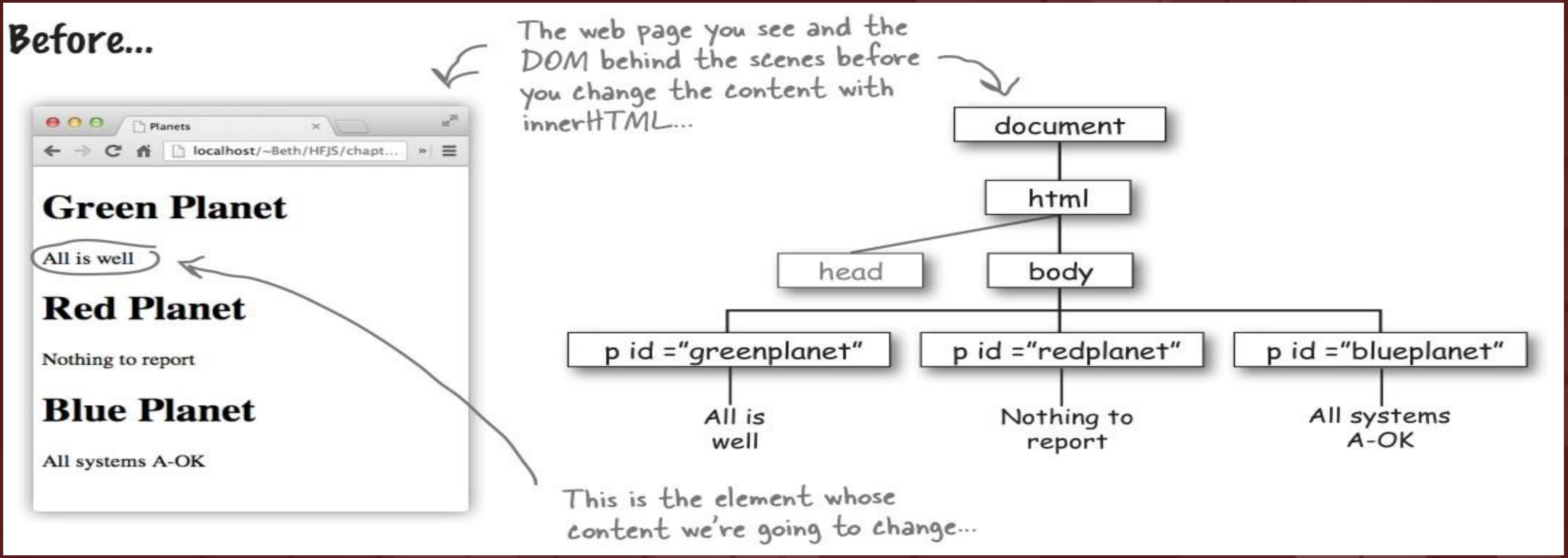
```
var planet = document.getElementById("greenplanet");
planet.innerHTML = "Red Alert: hit by phaser fire!";
console.log(planet.innerHTML);
```

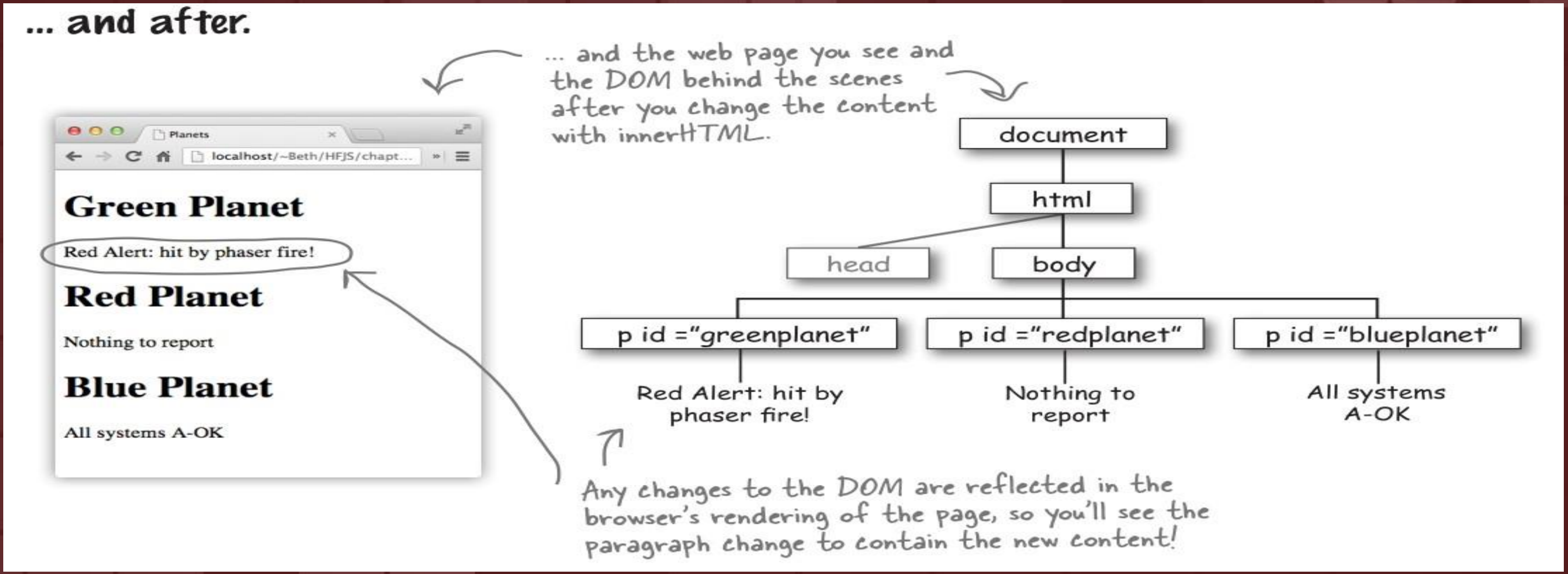


JavaScript console

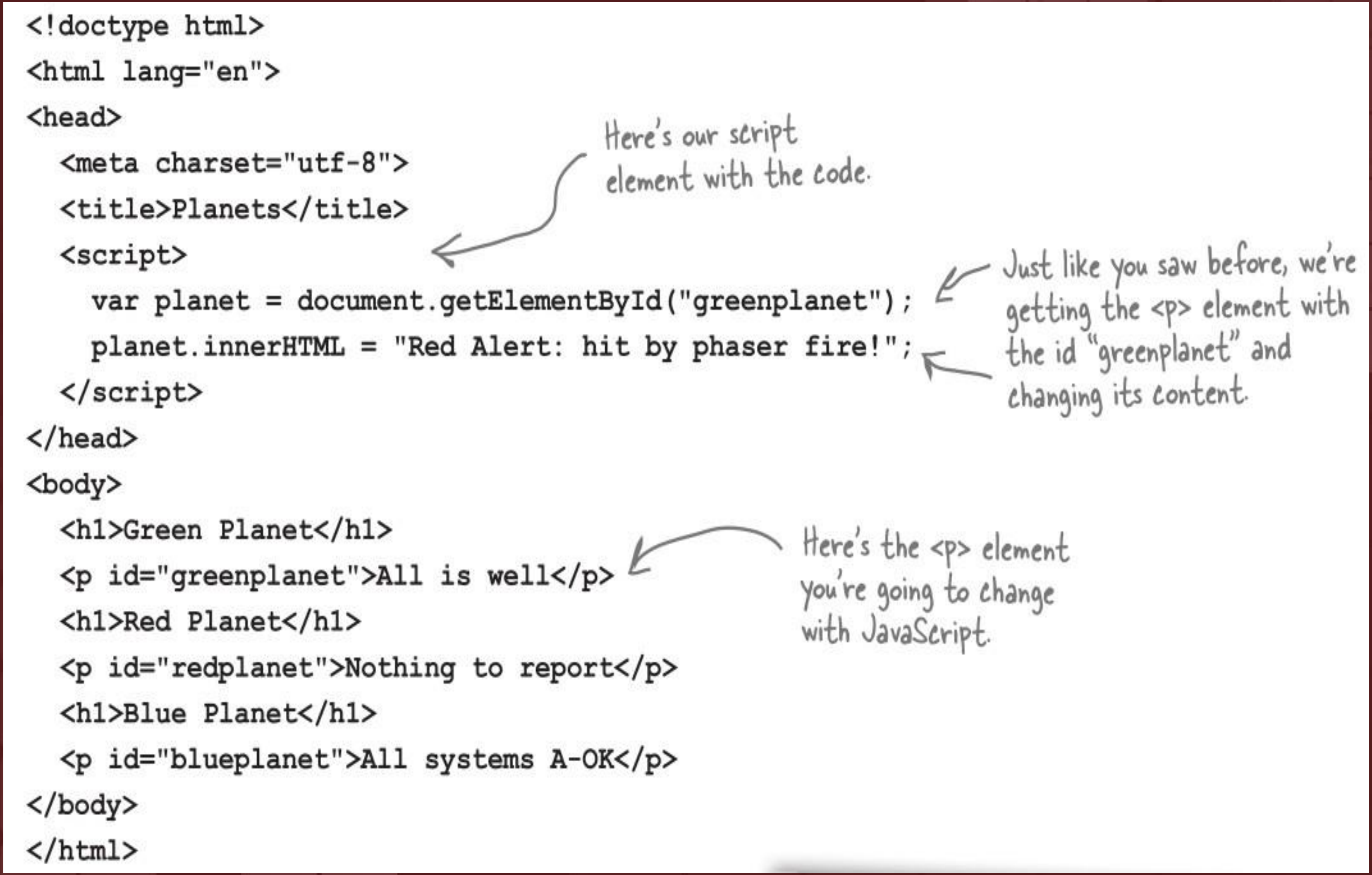
```
Red Alert: hit by
phaser fire!
```

What happens when you change the DOM




... and after.
... and the web page you see and the DOM behind the scenes after you change the content with innerHTML.
Planets
localhost/~Beth/HFJS/chapt...
Green Planet
Red Alert: hit by phaser fire!
Red Planet
Nothing to report
Blue Planet
All systems A-OK
document
html
head
body
p id ="greenplanet"
p id ="redplanet"
p id ="blueplanet"
Red Alert: hit by phaser fire!
Nothing to report
All systems A-OK
Any changes to the DOM are reflected in the browser's rendering of the page, so you'll see the paragraph change to contain the new content!

Here's the HTML for the planets. We've got a <script> element in the <head> where we'll put the code, and three paragraphs for the green, red, and blue planets. If you haven't already, go ahead and type in the HTML and the JavaScript to update the DOM:



```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Planets</title>
  <script>
    var planet = document.getElementById("greenplanet");
    planet.innerHTML = "Red Alert: hit by phaser fire!";
  </script>
</head>
<body>
  <h1>Green Planet</h1>
  <p id="greenplanet">All is well</p>
  <h1>Red Planet</h1>
  <p id="redplanet">Nothing to report</p>
  <h1>Blue Planet</h1>
  <p id="blueplanet">All systems A-OK</p>
</body>
</html>
```

Here's our script element with the code.

Just like you saw before, we're getting the <p> element with the id "greenplanet" and changing its content.

Here's the <p> element you're going to change with JavaScript.

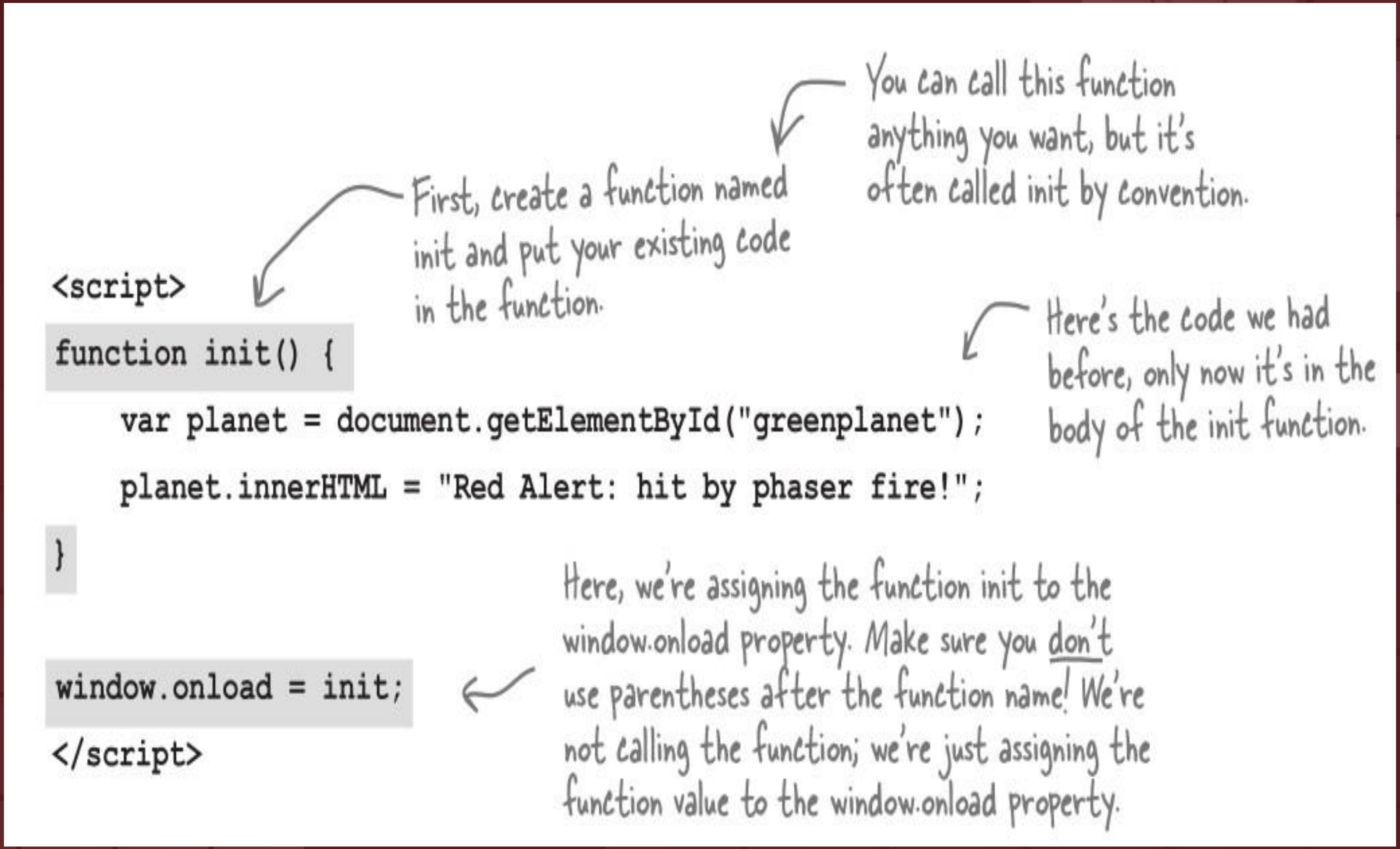
When you're dealing with the DOM it's important to execute your code only after the page is fully loaded. If you don't, there's a good chance the DOM won't be created by the time your code executes.

Let's think about what just happened: we put code in the <head> of the page, so it begins executing before the browser even sees the rest of the page. That's a big problem because that paragraph element with an id of "greenplanet" doesn't exist, yet.

So what happens exactly? The call to getElementById returns null instead of the element we want, causing an error, and the browser, being the good sport that it is, just keeps moving and renders the page anyway, but without the change to the green planet's content.

How do we fix this? Well, we could move the code to the bottom of the <body> , but there's actually a more foolproof way to make sure this code runs at the right time; a way to tell the browser "run my code after you've fully loaded in the page and created the DOM."

Here's how it works: first create a function that has the code you'd like executed once the page is fully loaded. After you've done that, you take the window object, and assign the function to its onload property.



```
<script>
function init() {
    var planet = document.getElementById("greenplanet");
    planet.innerHTML = "Red Alert: hit by phaser fire!";
}

window.onload = init;
</script>
```

You can call this function anything you want, but it's often called init by convention.
First, create a function named init and put your existing code in the function.
Here's the code we had before, only now it's in the body of the init function.
Here, we're assigning the function init to the window.onload property. Make sure you don't use parentheses after the function name! We're not calling the function; we're just assigning the function value to the window.onload property.

Let’s think about how onload works just a bit more, because it uses a common coding pattern you’ll see over and over again in JavaScript. Let's say there’s a big important event that’s going to occur, and you definitely want to know about it. Say that event is the “page is loaded” event. Well, a common way to deal with that situation is through a callback, also known as an event handler.

A callback works like this: give a function to the object that knows about the event. When the event occurs, that object will call you back, or notify you, by calling that function. You’re going to see this pattern in JavaScript for a variety of events.

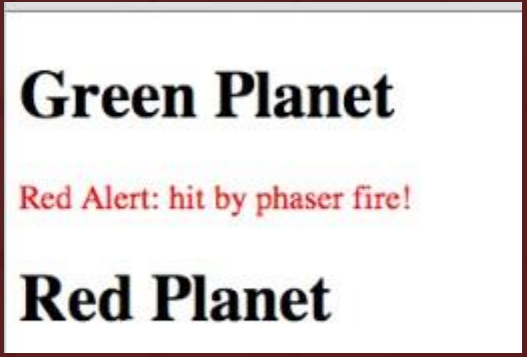
That’s right, and there are many kinds of events you can handle if you want to. Some events, like the load event, are generated by the browser, while others are generated by a user interacting with the page, or even by your own JavaScript code.

You’ve seen an example of “the page is loaded” event, which we handle by setting the onload property of the window object. You can also write event handlers that handle things like “call this function every five seconds,” or "some data arrived from a web service that we need to deal with," or "the user clicked a button and we need to get some data from a form," and there are many more.

Let's think for a second about what you just did: you took a static web page and you dynamically changed the content of one of its paragraphs using code. It seems like a simple step, but this is really the beginning of making a truly interactive page.

Let's take the second step: now that you know how to get your hands on an element in the DOM, let's set an attribute of an element with code.

Why would that be interesting? Well, take our simple planets example. When we set the paragraph to read "Red Alert," we could also set the paragraph's color to red. That would certainly more clearly communicate our message.



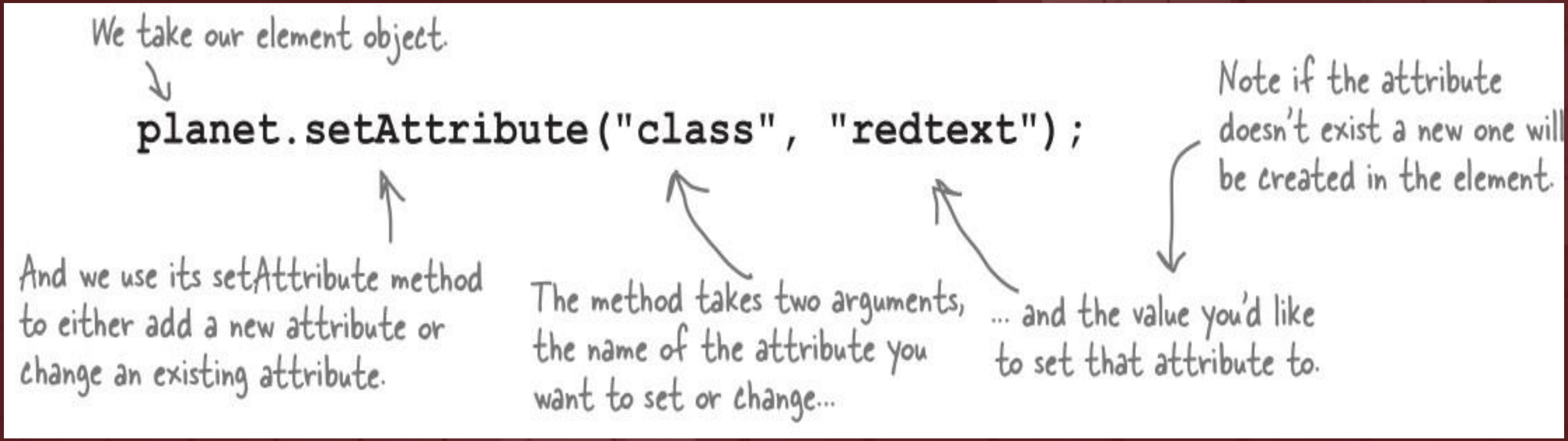
Green Planet

Red Alert: hit by phaser fire!

Red Planet

How to set an attribute with set Attribute

Element objects have a method named setAttribute that you can call to set the value of an HTML element's attribute. The setAttribute method looks like this:



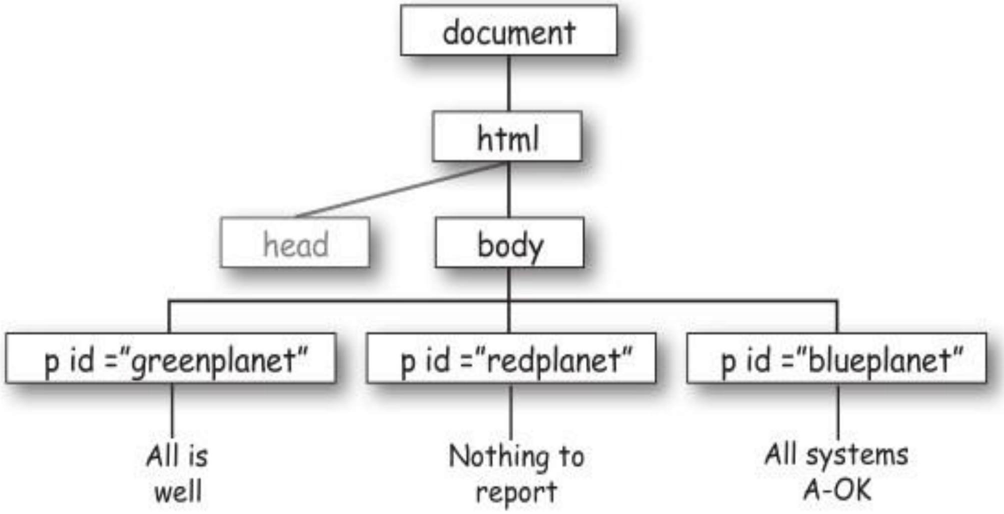
```
planet.setAttribute("class", "redtext");
```

We take our element object.
And we use its setAttribute method to either add a new attribute or change an existing attribute.
The method takes two arguments, the name of the attribute you want to set or change...
... and the value you'd like to set that attribute to.
Note if the attribute doesn't exist a new one will be created in the element.

We can call setAttribute on any element to change the value of an existing attribute, or, if the attribute doesn't already exist, to add a new attribute to the element.As an example, let's check out how executing the code above affects our DOM.

Before...

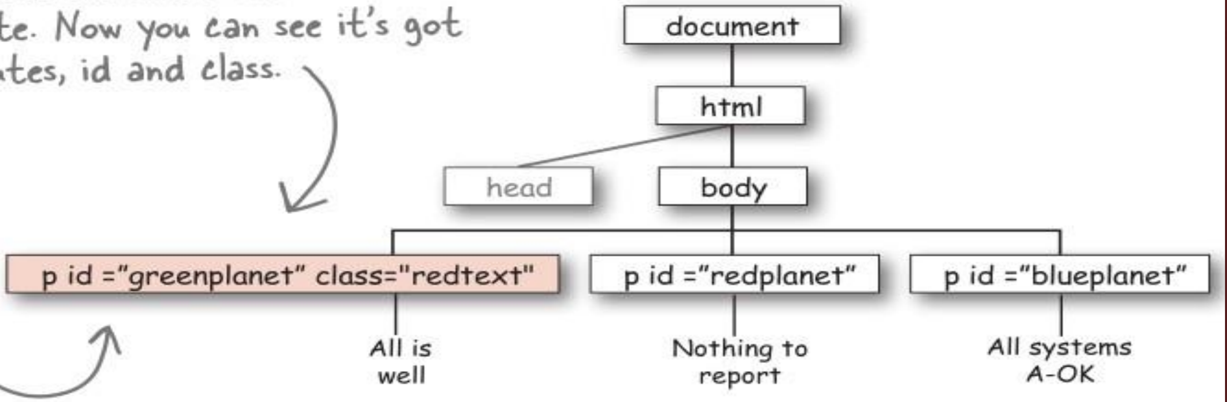
Here's the element before we call the setAttribute method on it. Notice this element already has one attribute, id.



And After

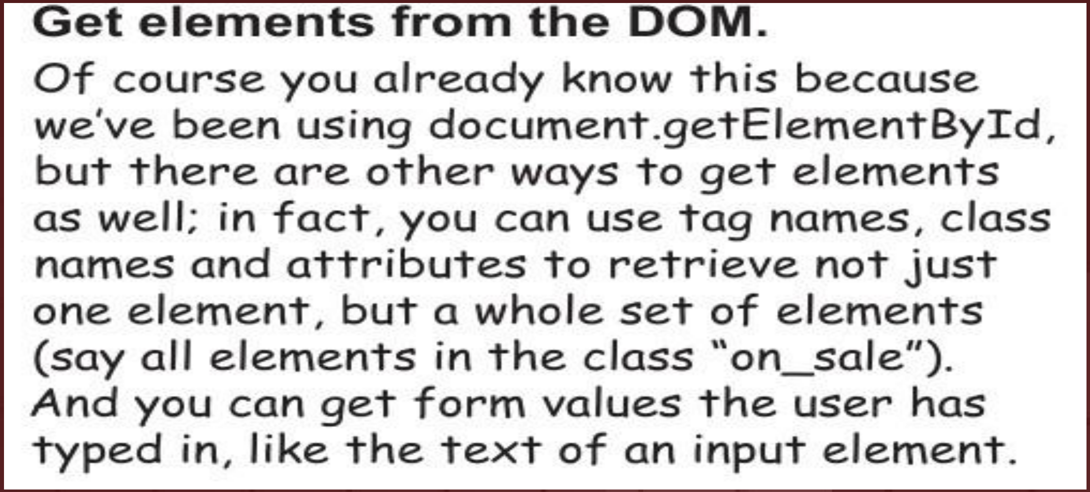
And here's the element after we call setAttribute. Now you can see it's got two attributes, id and class.

Remember, when we call the setAttribute method, we're changing the element object in the DOM, which immediately changes what you see displayed in the browser.



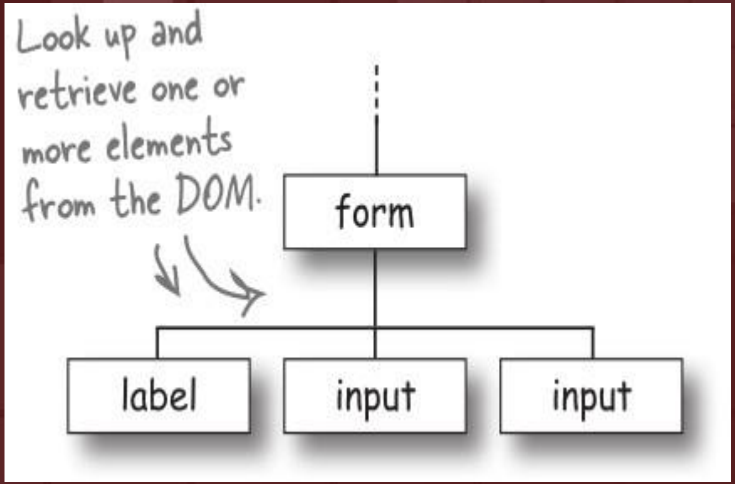
So what else is a DOM good for anyway?

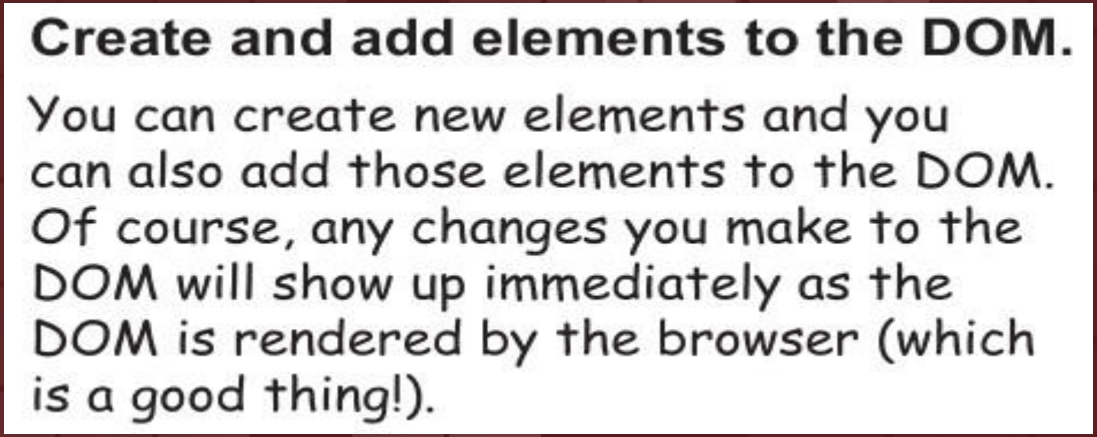
The DOM can do a fair bit more than we've seen so far and we'll be seeing some of its other functionality as we move forward in the course, but for now let's just take a quick look so you've got it in the back of your mind:



Get elements from the DOM.

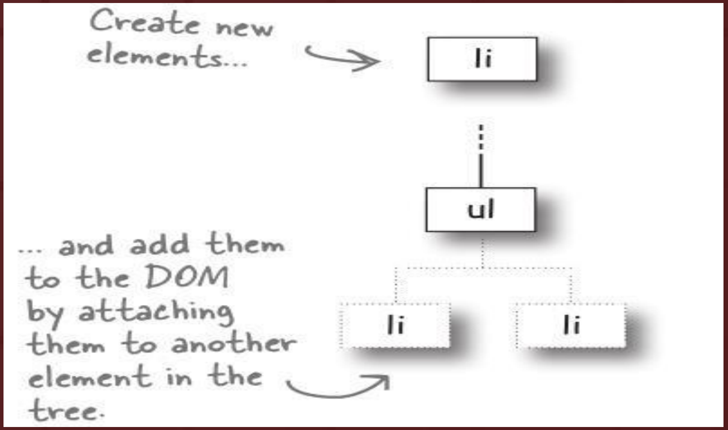
Of course you already know this because we've been using document.getElementById, but there are other ways to get elements as well; in fact, you can use tag names, class names and attributes to retrieve not just one element, but a whole set of elements (say all elements in the class "on_sale"). And you can get form values the user has typed in, like the text of an input element.





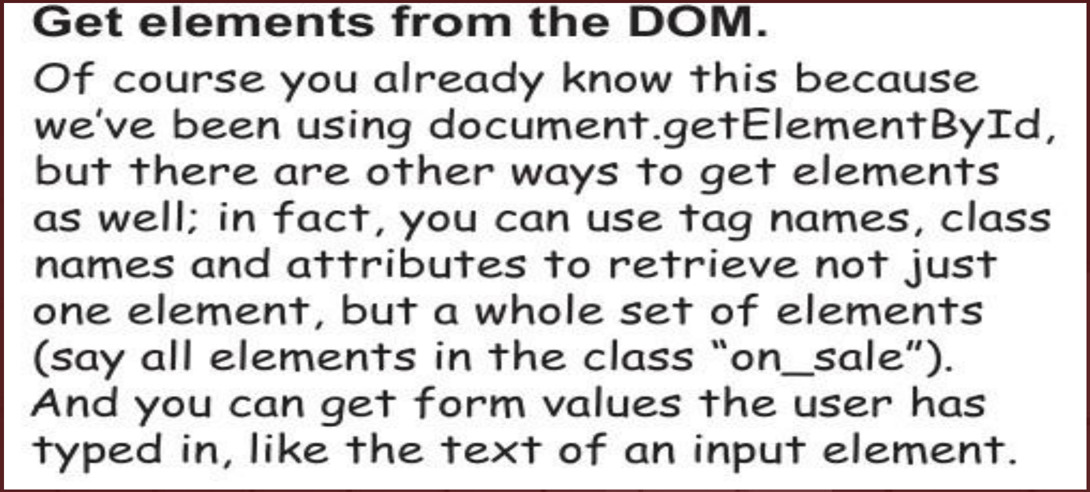
Create and add elements to the DOM.

You can create new elements and you can also add those elements to the DOM. Of course, any changes you make to the DOM will show up immediately as the DOM is rendered by the browser (which is a good thing!).


Create new elements...
li
ul
... and add them to the DOM by attaching them to another element in the tree.
li
li

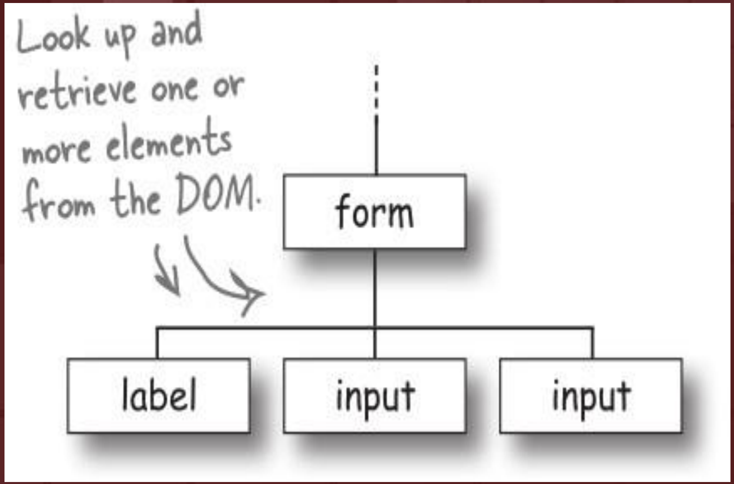
So what else is a DOM good for anyway?

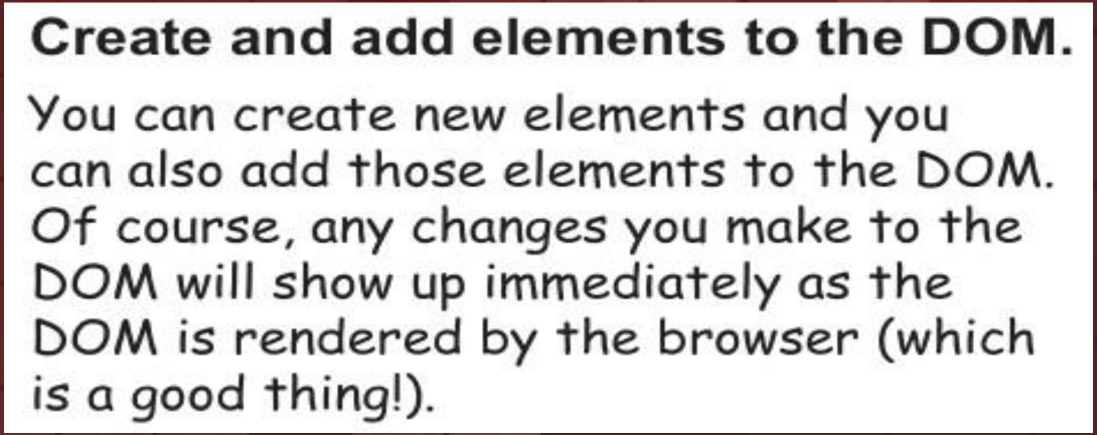
The DOM can do a fair bit more than we've seen so far and we'll be seeing some of its other functionality as we move forward in the course, but for now let's just take a quick look so you've got it in the back of your mind:



Get elements from the DOM.

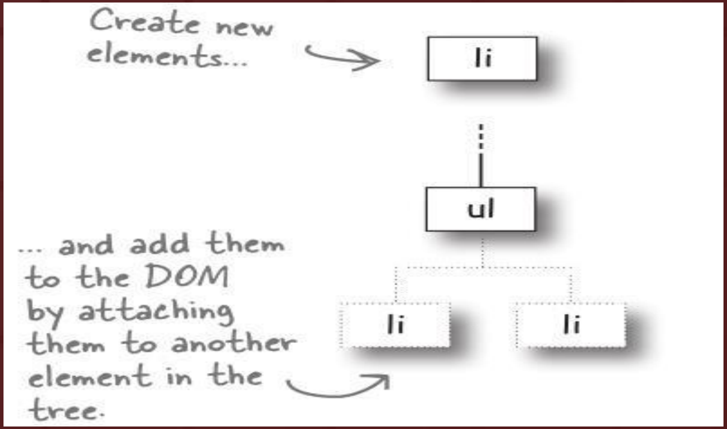
Of course you already know this because we've been using document.getElementById, but there are other ways to get elements as well; in fact, you can use tag names, class names and attributes to retrieve not just one element, but a whole set of elements (say all elements in the class "on_sale"). And you can get form values the user has typed in, like the text of an input element.

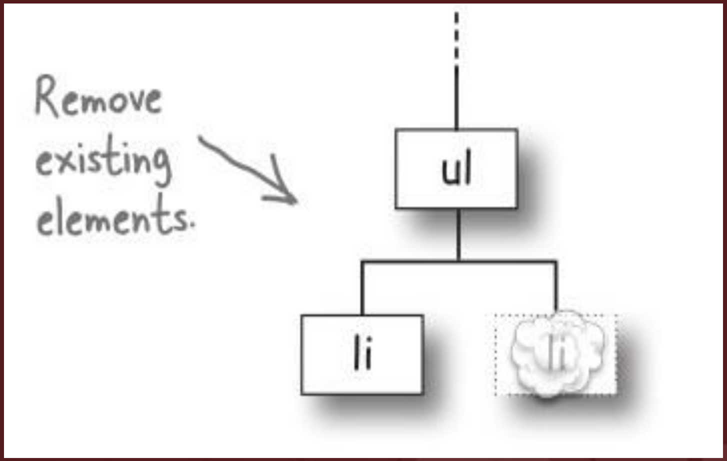




Create and add elements to the DOM.

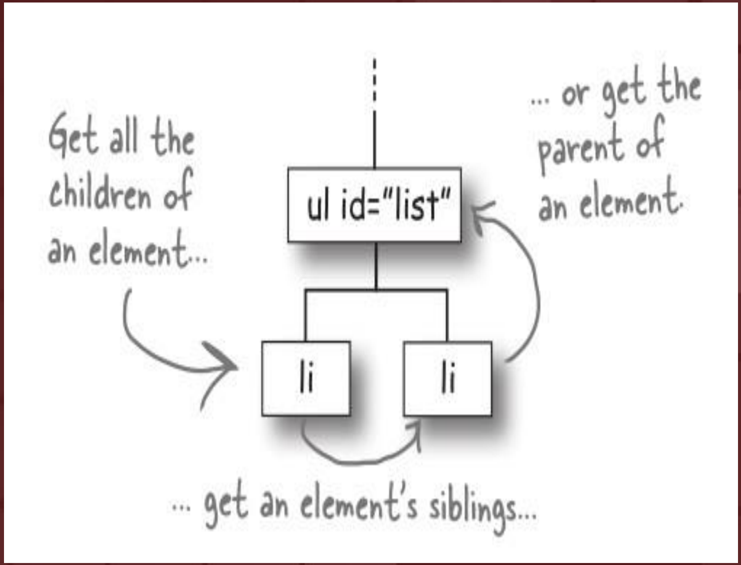
You can create new elements and you can also add those elements to the DOM. Of course, any changes you make to the DOM will show up immediately as the DOM is rendered by the browser (which is a good thing!).


Create new elements...
li
ul
li
li
... and add them to the DOM by attaching them to another element in the tree.



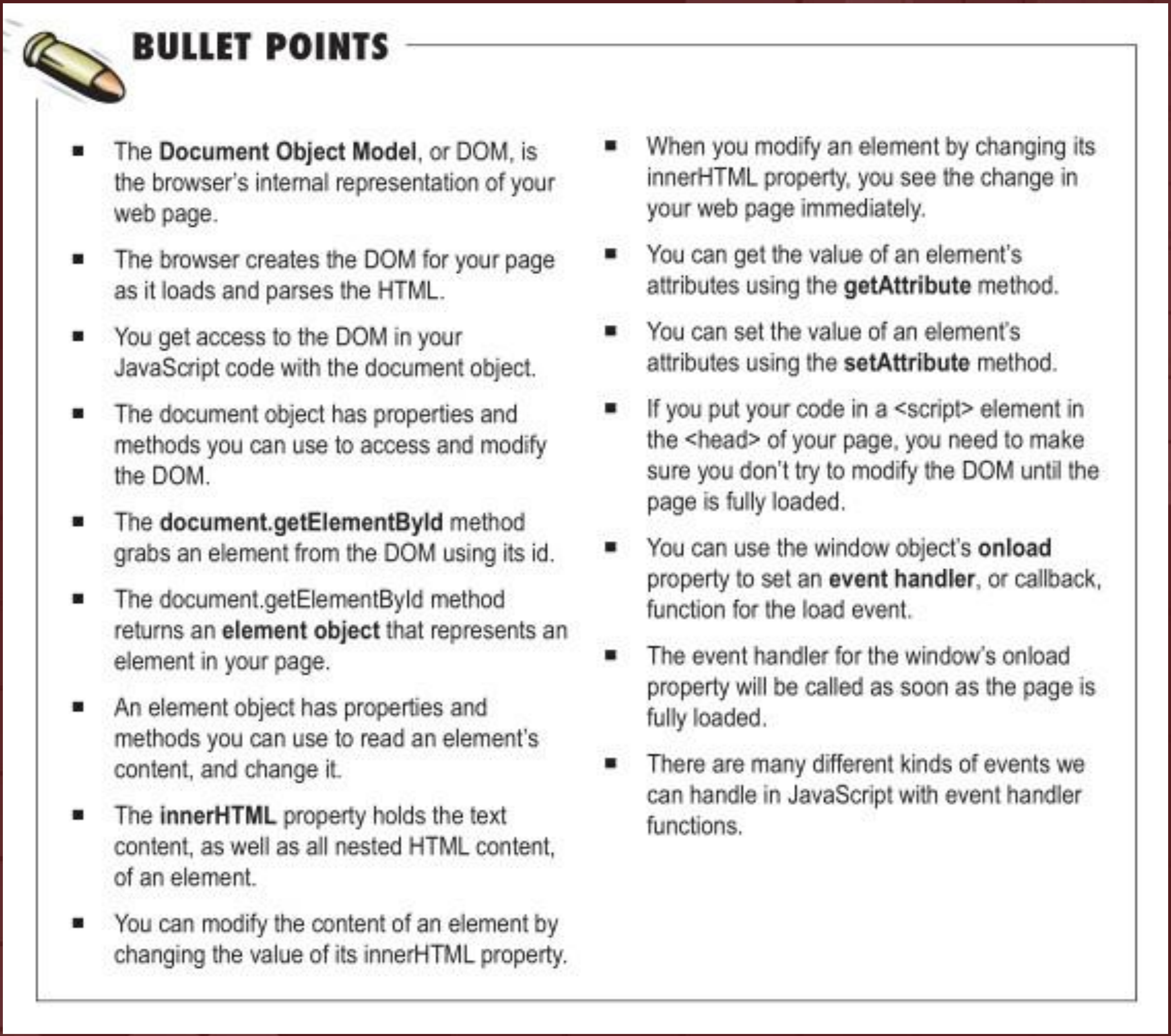
Remove elements from the DOM.

You can also remove elements from the DOM by taking a parent element and removing any of its children. Again, you'll see the element removed in your browser window as soon as it is deleted from the DOM.


Get all the children of an element...
ul id="list"
li
li
... or get the parent of an element.
... get an element's siblings...

Traverse the elements in the DOM.

Once you have a handle to an element, you can find all its children, you can get its siblings (all the elements at the same level), and you can get its parent. The DOM is structured just like a family tree!



BULLET POINTS

- The **Document Object Model**, or DOM, is the browser's internal representation of your web page.
- The browser creates the DOM for your page as it loads and parses the HTML.
- You get access to the DOM in your JavaScript code with the document object.
- The document object has properties and methods you can use to access and modify the DOM.
- The **document.getElementById** method grabs an element from the DOM using its id.
- The document.getElementById method returns an **element object** that represents an element in your page.
- An element object has properties and methods you can use to read an element's content, and change it.
- The **innerHTML** property holds the text content, as well as all nested HTML content, of an element.
- You can modify the content of an element by changing the value of its innerHTML property.
- When you modify an element by changing its innerHTML property, you see the change in your web page immediately.
- You can get the value of an element's attributes using the **getAttribute** method.
- You can set the value of an element's attributes using the **setAttribute** method.
- If you put your code in a <script> element in the <head> of your page, you need to make sure you don't try to modify the DOM until the page is fully loaded.
- You can use the window object's **onload** property to set an **event handler**, or callback, function for the load event.
- The event handler for the window's onload property will be called as soon as the page is fully loaded.
- There are many different kinds of events we can handle in JavaScript with event handler functions.

Questions ?