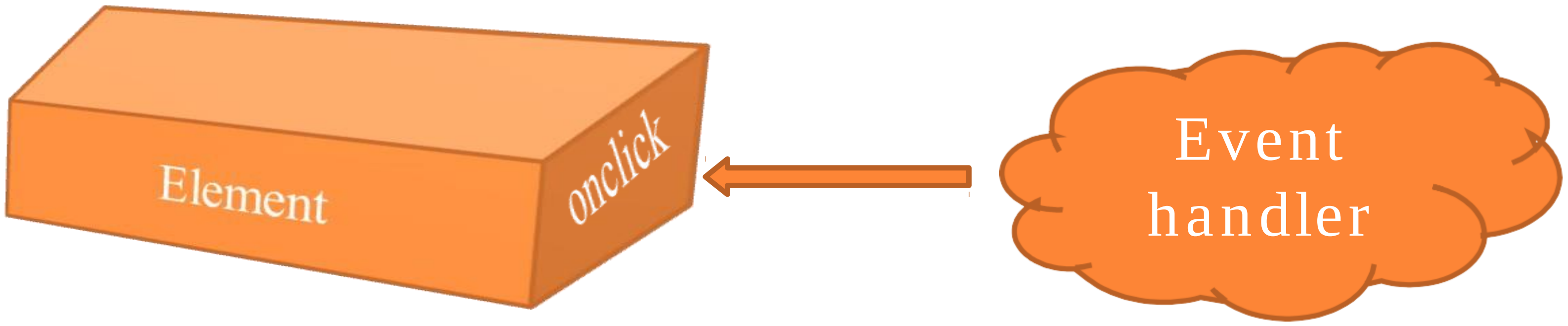
EVENTS

- JS enables us to write scripts that are triggered by **events** that occur during the user’s interaction with the page, like clicking a hyperlink, scrolling the browser’s viewport, typing a value into a form field or submitting a form.

EVENT HANDLERS

Simplest way to run JS code in response to an event is to use an **event handler** (function).



How to attach?: element.on*event* = eventHandler

Note: I didn't provide parentheses to eventHandler

DEFAULT ACTIONS AND THIS

- Default actions: Things the browser normally does in response to events.
- How to prevent?: return **false** or **true** to cancel or let the default action follow.
- this keyword. **this** is an object reference that you get when executing a method on an object. When browser invokes the event handler on some event for an element, then within the event handler **this** points to the element on which event is fired.

We can’t set the value of **this** object.

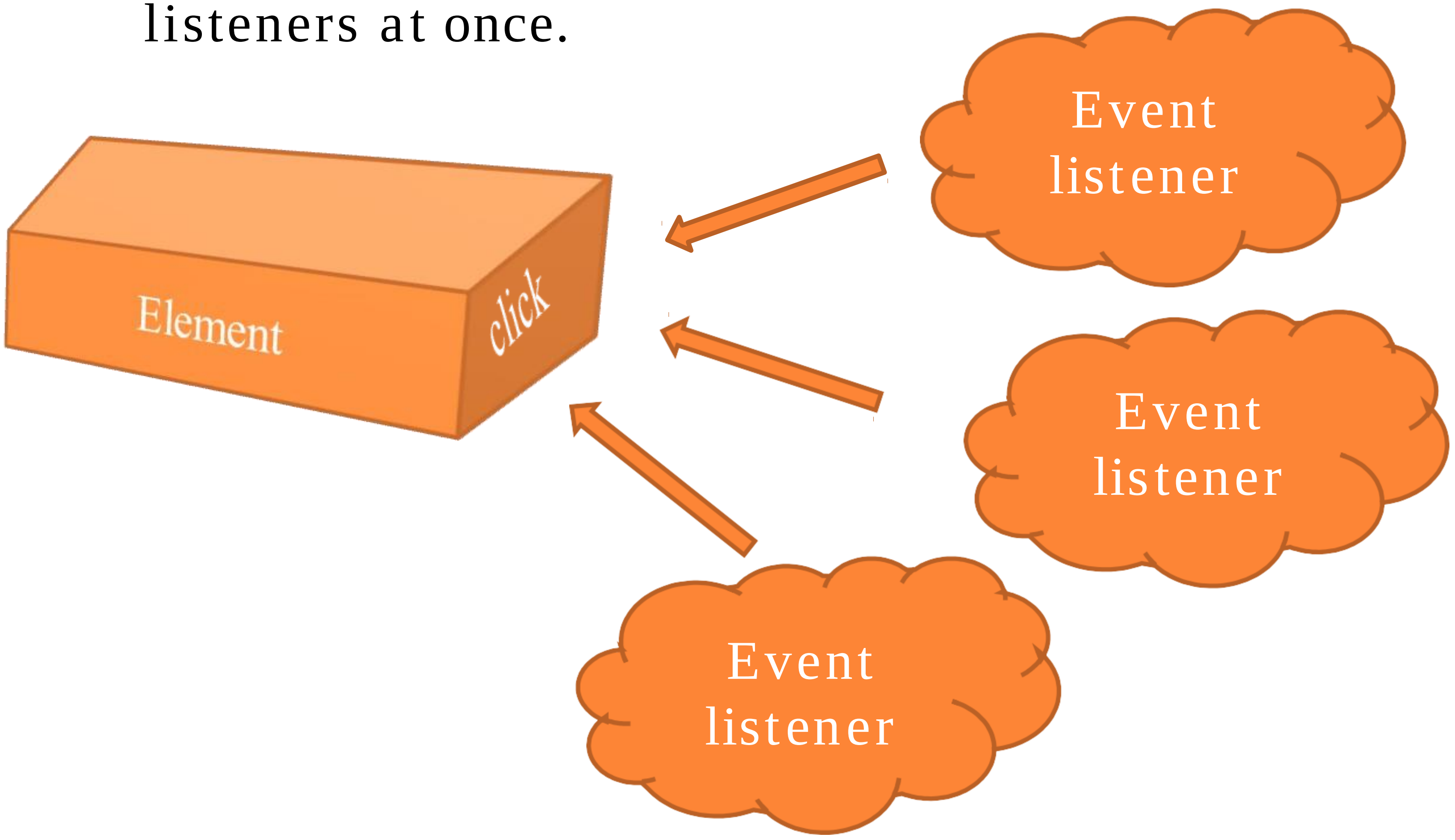
EVENT LISTENERS

- What’s wrong with event handlers? You can’t assign multiple event handlers to an event. i.e.,

  ```
  element.onclick = firstHandler;
  element.onclick = secondHandler;
  ```
- Now only secondHandler will be called, as it has replaced the firstHandler.
- However we can assign as many event listeners as we want to an event of an element, and all of them will be called.

IT'S LIKE HUB

As we can see we can plugin many event listeners at once.



ATTACHING EVENT LISTENERS

- How to attach event listener?: i.e.,

  element.addEventListener(“event”, eventListener, false);

  for IE, element.attachEvent(“onevent”, eventListener);

Object detection:
typeof element.property != “undefined”

EVENT PARAMETER

- Event parameter of the listener: Browser automatically passes the **event** parameter to the event listener function. i.e.,

```
function someClickEventListener (event) {
        // Use event argument's methods
}
```

- **event** parameter has some important methods. Some of them are: **preventDefault()** and **stopPropagation()**.

EVENT OBJECT

- IE has its own way. If you remember IE doesn’t expect third argument. It means it also doesn’t pass **event** parameter to the event listener function. Then how to access it?
- In IE there is a global event object, which is **window.event**. And it has equivalent properties like the event parameter which are: **returnValue** which we can set to **false**, and **cancelBuble** which we can set to **true**. These two settings achieves the same effect as its event parameter counterpart’s methods.

DETACHING EVENT LISTENERS

- How to detach event listener: i.e.,

  element.removeEventListener(“event”, eventListener, false);

  for IE, element.detachListener(“onevent”, eventListener);

EVENT PROPAGATION

- What is event propagation?

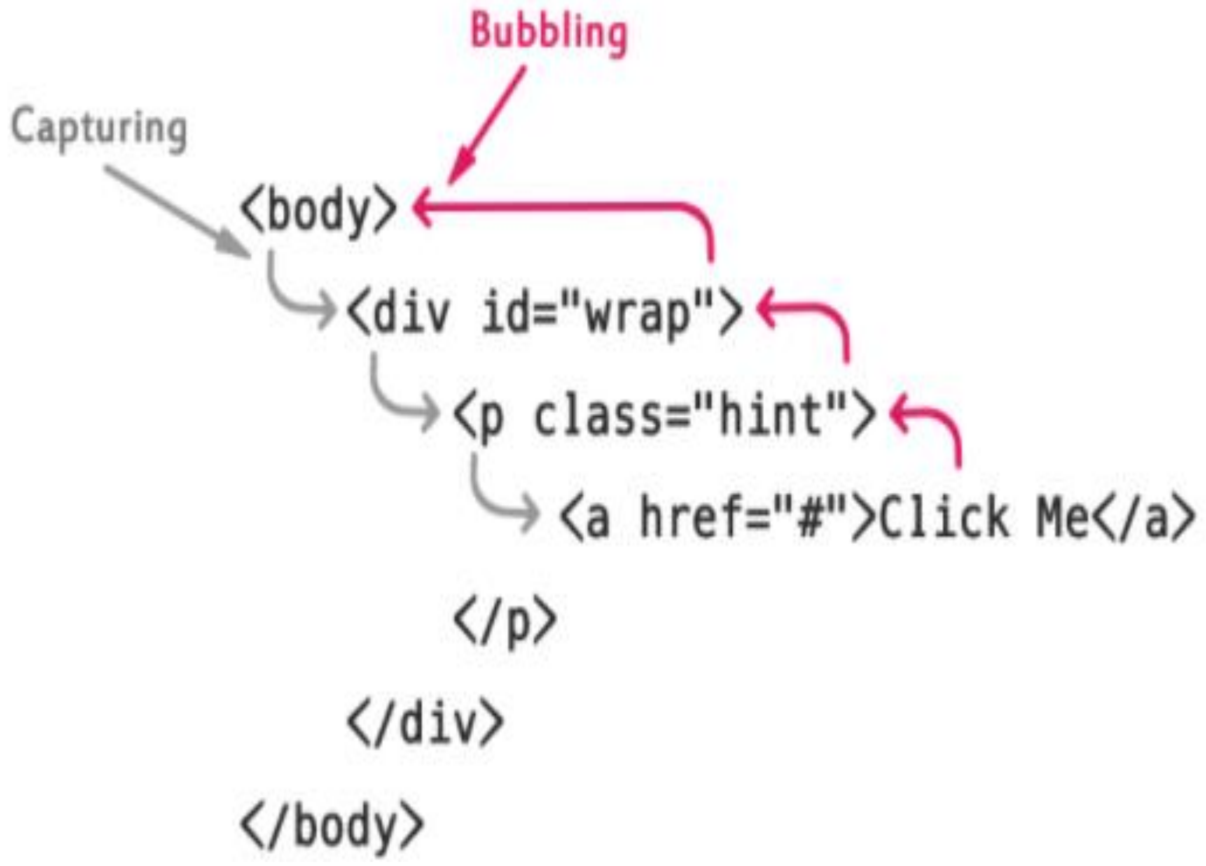
Event propagation is a mechanism that defines how events propagate or travel through the DOM tree to arrive at its target and what happens to it afterward.

Event propagation runs in three phases:

- Capture phase: Document to element.
- Target phase: Element which triggered the event.
- Bubbling phase: Element to Document.

NOTE: - The stopPropagation() method prevents propagation of the same event from being called. Propagation means bubbling up to parent elements or capturing down to child elements.

EVENT PROPAGATION



```
<body>
    <div id="wrap">
        <p class="hint">
            <a href="#">Click Me</a>
        </p>
    </div>
</body>
```

EVENT PROPAGATION

- Let's understand this with the help of an example, suppose you have assigned a click event handler on a hyperlink (i.e. <a> element) which is nested inside a paragraph (i.e. <p> element). Now if you click on that link, the handler will be executed. But, instead of link, if you assign the click event handler to the paragraph containing the link, then even in this case, clicking the link will still trigger the handler. That's because events don't just affect the target element that generated the event—they travel up and down through the DOM tree to reach their target. This is known as event propagation
- In modern browser event propagation proceeds in two phases: capturing, and bubbling phase. Before we proceed further, take a look at the following illustration:

Callback

- A callback is a function passed as an argument to another function
- This technique allows a function to call another function
- A callback function can run after another function has finished

NOTE:

- JavaScript functions are executed in the sequence they are called. Not in the sequence they are defined.
- In the real world, callbacks are most often used with asynchronous functions.

Asynchronous

- Functions running in parallel with other functions are called asynchronous like reading/writing a file
- A good example is JavaScript setTimeout()

setTimeout()
When using the JavaScript function setTimeout(), you can specify a callback function to be executed on time-out:

setTimeout(myFunction, time in milliseconds);

setInterval()
When using the JavaScript function setInterval(), you can specify a callback function to be executed for each interval

setInterval(myFunction, time in milliseconds);

Promises

Promises are used to handle asynchronous operations in JavaScript. They are easy to manage when dealing with multiple asynchronous operations where callbacks can create callback hell leading to unmanageable code.

- Prior to promises events and callback functions were used but they had limited functionalities and created unmanageable code.
- Multiple callback functions would create callback hell that leads to unmanageable code.
- Events were not good at handling asynchronous operations.

Promises

Benefits of Promises

- Improves Code Readability
- Better handling of asynchronous operations
- Better flow of control definition in asynchronous logic
- Better Error Handling

Promise has four states:

- **fulfilled**: Action related to the promise succeeded
- **rejected**: Action related to the promise failed
- **pending**: Promise is still pending i.e not fulfilled or rejected yet
- **settled**: Promise has fulfilled or rejected

A promise can be created using Promise constructor.
Syntax

```
var promise = new Promise(function(resolve, reject){
    //do something
});
```

Promises

A promise is an object that may produce a single value some time in the future: either a resolved value, or a reason that it’s not resolved (e.g., a network error occurred).

A promise may be in one of 3 possible states: fulfilled, rejected, or pending.
A promise is an object which can be returned synchronously from an asynchronous function. It will be in one of 3 possible states:

- Fulfilled: onFulfilled() will be called (e.g., resolve() was called)
- Rejected: onRejected() will be called (e.g., reject() was called)
- Pending: not yet fulfilled or rejected

Promises Cosumers

Promise Consumers

Promises can be consumed by registering functions using .then and .catch methods.

- then() is invoked when a promise is either resolved or rejected.

Parameters:

- then() method takes two functions as parameters.

First function is executed if promise is resolved and a result is received.

Second function is executed if promise is rejected and an error is received. (It is optional and there is a better way to hanlde error using .catch() method

Syntax:

```
.then(function(result){
      //handle success
   }, function(error){
      //handle error
   })
```

Async and Await

Javascript is a Synchronous which means that it has an event loop that allows you to queue up an action that won’t take place until the loop is available sometime after the code that queued the action has finished executing.

But there’s a lot of functionalities in our program which makes our code Asynchronous.

- Async/Await is the extension of promises
- *async and await make promises easier to write"*
- **async** makes a function return a Promise
- **await** makes a function wait for a Promise

Async

The keyword async before a function makes the function return a promise

```
async function myFunction() {
  return "Hello";
}
```

This is same like below

```
async function myFunction() {
  return Promise.resolve("Hello");
}
```

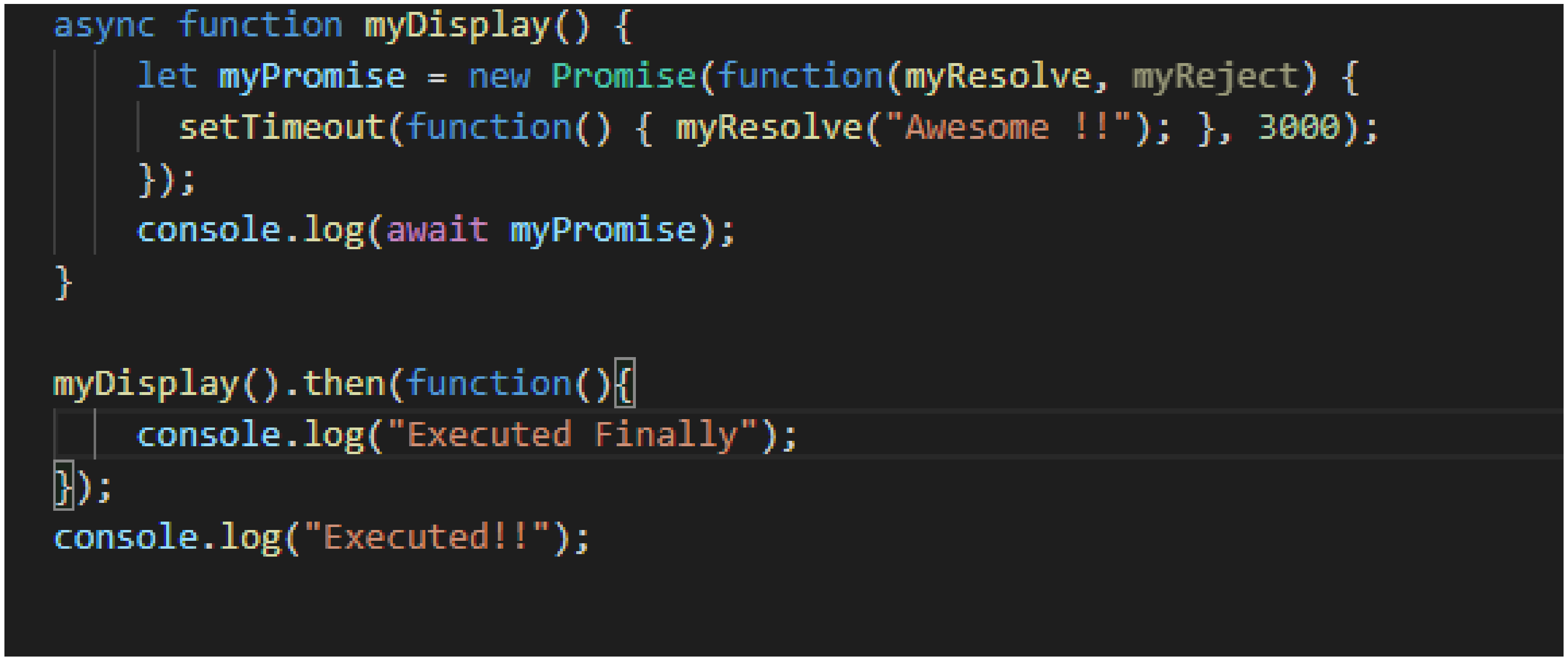
And, how to use the promise

```
myFunction().then(
  function(value) { /* code if successful */ },
  function(error) { /* code if some error */ }
);
```

Await

The keyword await before a function makes the function wait for a promise

- The await keyword can only be used inside an async function



```
async function myDisplay() {
    let myPromise = new Promise(function(myResolve, myReject) {
      setTimeout(function() { myResolve("Awesome !!"); }, 3000);
    });
    console.log(await myPromise);
}

myDisplay().then(function(){
    console.log("Executed Finally");
});
console.log("Executed!!");
```

Ajax

- Read data from a web server - after the page has loaded
- Update a web page without reloading the page
- Send data to a web server - in the background

The keystone of AJAX is the XMLHttpRequest object.

- Create an XMLHttpRequest object
- Define a callback function
- Open the XMLHttpRequest object
- Send a Request to a server

AJAX

- AJAX acronym for Asynchronous JavaScript And XML.
- This technology is used for **asynchronous** information transfer between browser and server in **bite-size** chunks.
- It is supported using the **XMLHttpRequest** object built right into the browser.
- Firstly implemented by IE 5 and 6 and they did using the **ActiveX** object.
- IE 7+ don’t use ActiveX object, instead they use XMLHttpRequest.

INITIALIZE

- How to initialize?: i.e.,

  var requester = new XMLHttpRequest();

  For IE,

  var requester = new ActiveXObject(“Microsoft.XMLHTTP”);

PROPER INSTANTIATION

- Example of instantiating the XMLHttpRequest:

```
try {
    var requester = new XMLHttpRequest();
}catch (error) {
    try {
        var requester = new
   ActiveXObject(“Microsoft.XMLHTTP”);
    }catch (error) {
        var requester = null;
    }
}
```

USING THE XHR

- Using the XMLHttpRequest:

requester.setRequestHeader(“Content-Type”, “application/x-www-form-urlencoded”); // Optional

requester.open(“GET”, “/url.xml”, true);

requester.send(null);

requester.onreadystatechange = function() {

// Use requester.readyState property, which is 0 to 4, uninitialized, loading, loaded, interactive, **complete**.

// Also now check requester.state property, which contains HTTP codes of response. For success use 200 or 304, other are failures.

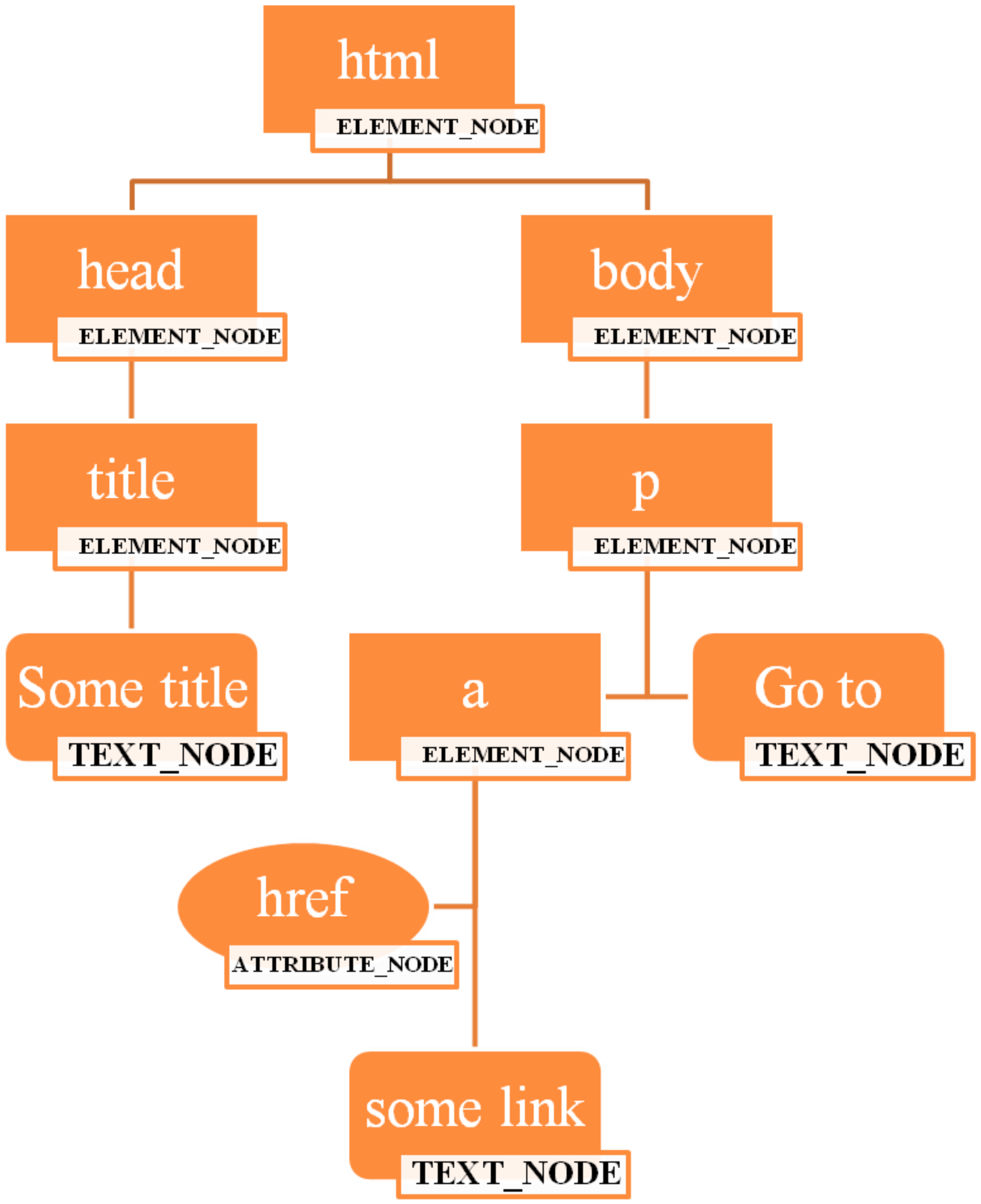
READ THE DATA FROM XHR

- Where is the data:
  - responseXML: If the server responsed with **content-type** set to **text/xml** then we have DOM. We can traverse it as usual to get the data. It also populates the **responseText** property, just for an alternative.
  - responseText: If the server responsed with the **content-type** set to **text/plain** or **text/html** then we have single string as the data. But now the **responseXML** will be empty.

DOCUMENT ACCESS

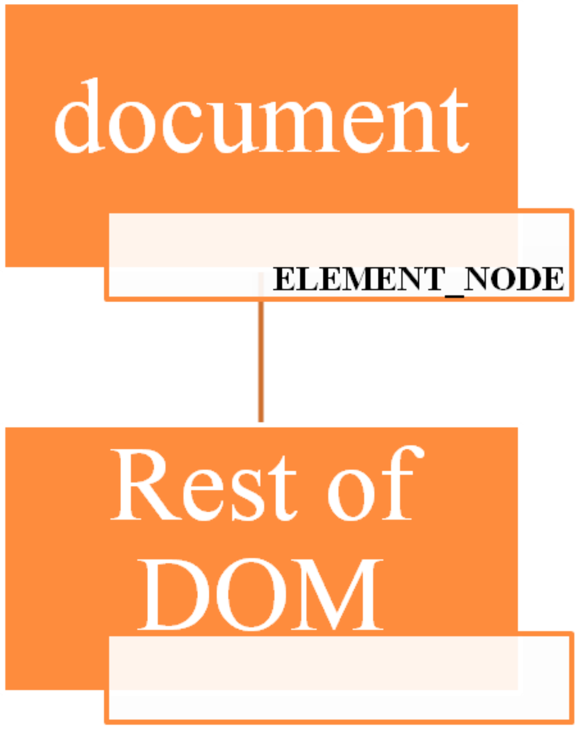
- **Document Object Model**: Mapping your HTML. The browser stores its interpreted HTML code as a structure of JavaScript objects, called **DOM**.
- DOM consists of Nodes. There are currently 12 types of nodes. Most of which we use are **ELEMENT_NODE (1), ATTRIBUTE_NODE (2), TEXT_NODE (3)**.
- We can use the **nodeType** property to check the type of the node. i.e., document.getElementsByTagName("h1")[0].nodeType; // 1

DOM Structure:



GRAND NODE

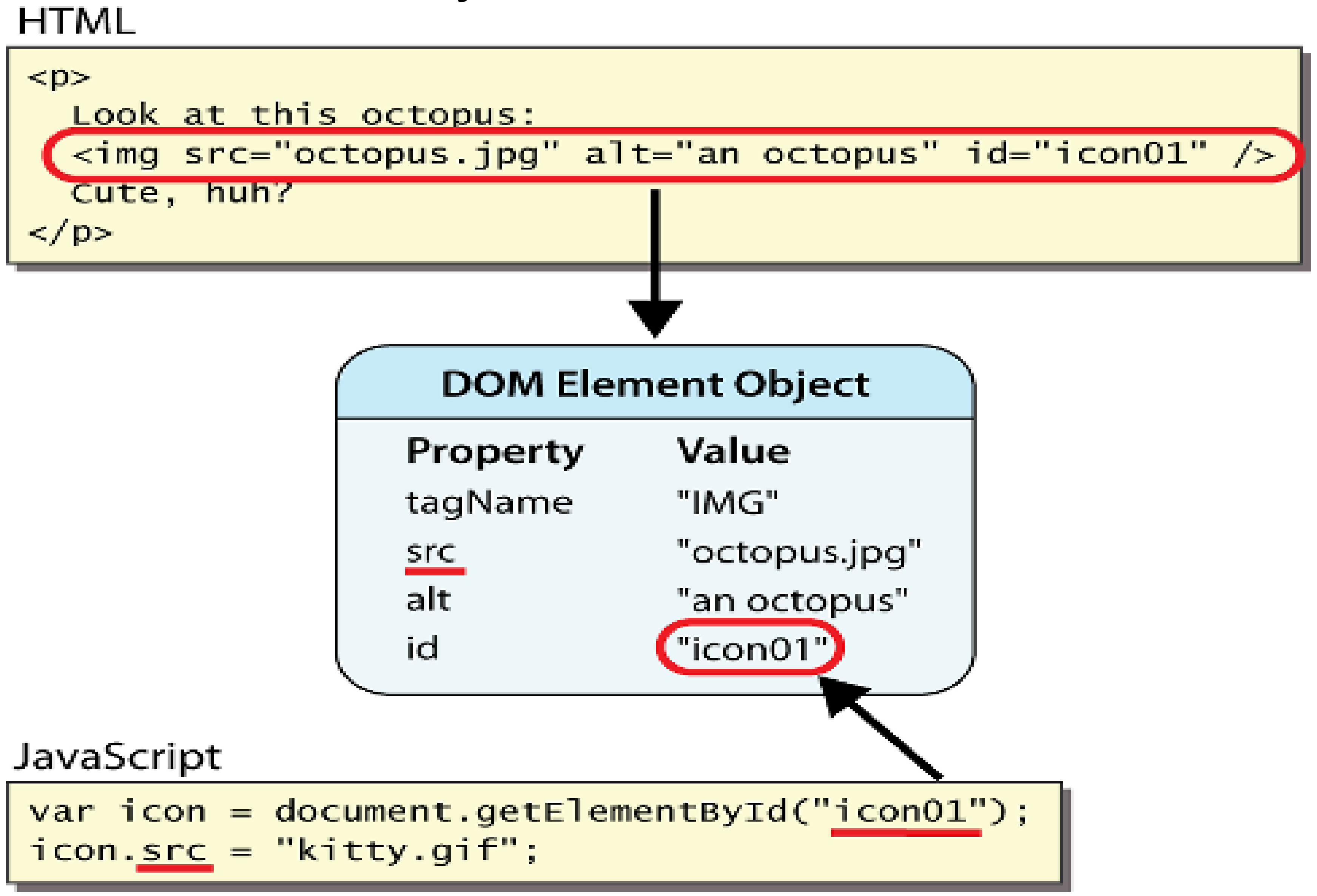
In fact there is an universal node. **document** node.This node exists even if the document is blank.



DOM MANIPLUATION

- Access elements: Use **document.get*()** methods.
- Alter properties. It depends on the type ofnode.
- If it's some html node like div, h1, span, label then you would set its **innerHTML**property or sometimes **textContent** property, if element is**li**.
- If it's some input element then you would set its **value** property.
- Traversal: We use **nextSibling, previousSibling, parent, children** etc. properties.

DOM element objects



```
<p>
  Look at this octopus:
  <img src="octopus.jpg" alt="an octopus" id="icon01" />
  Cute, huh?
</p>
```

Property	Value
tagName	"IMG"
src	"octopus.jpg"
alt	"an octopus"
id	"icon01"

```
var icon = document.getElementById("icon01");
icon.src = "kitty.gif";
```

Accessing elements: `document.getElementById`

```
var name = document.getElementById("id");
                                                        JS
```

```
<button onclick="changeText();">Click me!</button>
<span id="output">replace me</span>
<input id="textbox" type="text" />                    HTML
```

```
function changeText() {
      var span = document.getElementById("output");
      var textBox = document.getElementById("textbox");

       textbox.style.color = "red";

}                                                       JS
```

Accessing elements: `document.getElementById`

- document.getElementById returns the DOM object for an element with a given id
- can change the text inside most elements by setting the innerHTML property
- can change the text in form controls by setting the value property

Changing element style: `element.style`

Attribute	**Property or style object**
color	color
padding	padding
background-color	backgroundColor
border-top-width	borderTopWidth
Font size	fontSize
Font famiy	fontFamily

MANIPLUATING STYLE AND CLASSES

- Altering style: We can use the **style** property to alter the style of any element. i.e.,

  document.getElementsByTagName (“body”)[0].style.background = “#ddd”;

  document.getElementByTagName(“label”)[0].style.position = “absolute”;

  document.getElementByTagName(“label”)[0].style.left = “300px”;

- We can also use **className** property to access or set a CSS class directly. It’s directly available on the element.

BROWSER ACCESS

- Browser Object Model: It targets the browser environment rather than the document. It offers some objects allow us to handle the browser by our script.
- window
- location
- navigator
- screen
- history

FORMS

- Forms are there to collect the data. And JavaScript is known for its form validations. But as we know JavaScript is more than that. However forms are still integral part. Whenever there is a single input box it must be contained inside a form.
- Some DOM Methods for HTML Form Controls

Method	Element(s)	Description
blur	input, select, textarea	Removes keyboard focus
click	input	Simulates a mouse click
focus	input, select, textarea	Gives keyboard focus

- …Continued

reset	form	Reset all control's value to default
select	input, textarea	Selects all the contents
submit	form	Submits the form without triggering submit event

- Some DOM Properties for HTML Form Controls

Property	Element(s)	Description
elements	form	A node list containing all the form controls

…Continued

checked	input	True only for checkbox and radio inputs if checked else false
disabled	button, input, optgroup, option, select, textarea	While true controls is unavailable to user
form	button, input, option, select, textarea	A reference to the form containing this control
index	option	Index of this option control within the select control that contains it (0 for first)

- …Continued

options	select	A node list containing all the options controls
selected	option	True if this option is selected else false.
selectedIndex	select	Index of the currently selected option (0 for the first)
value	button, input, option, select, textarea	Current value of this control, as it would be submitted to the server

- Some DOM Events for HTML Form Controls

Event	Element(s)	Triggered when…
change	input, select, textarea	Control lost focus after its value has changed
select	input, textarea	User has selected some text
submit	form	User has requested to submit the form