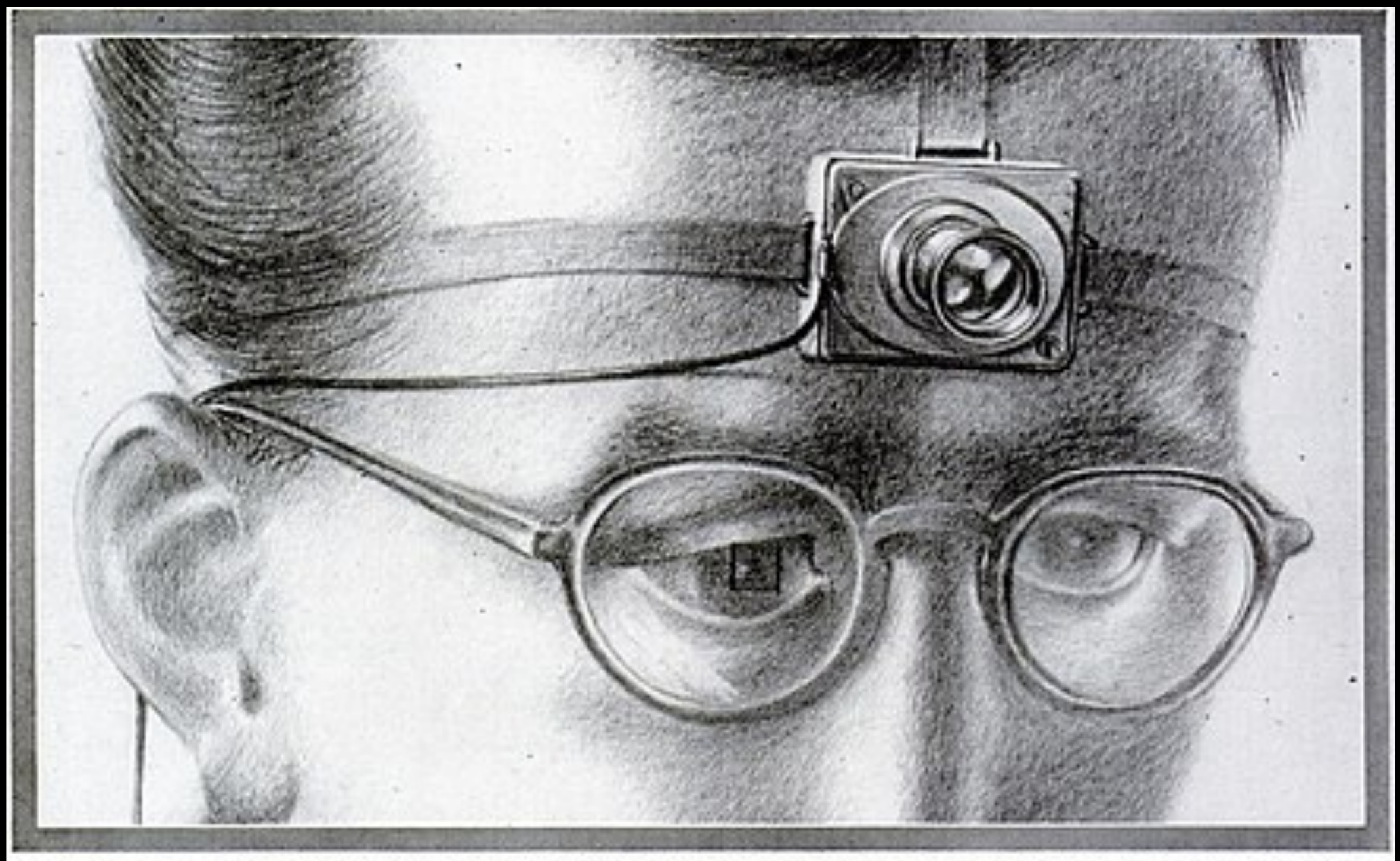


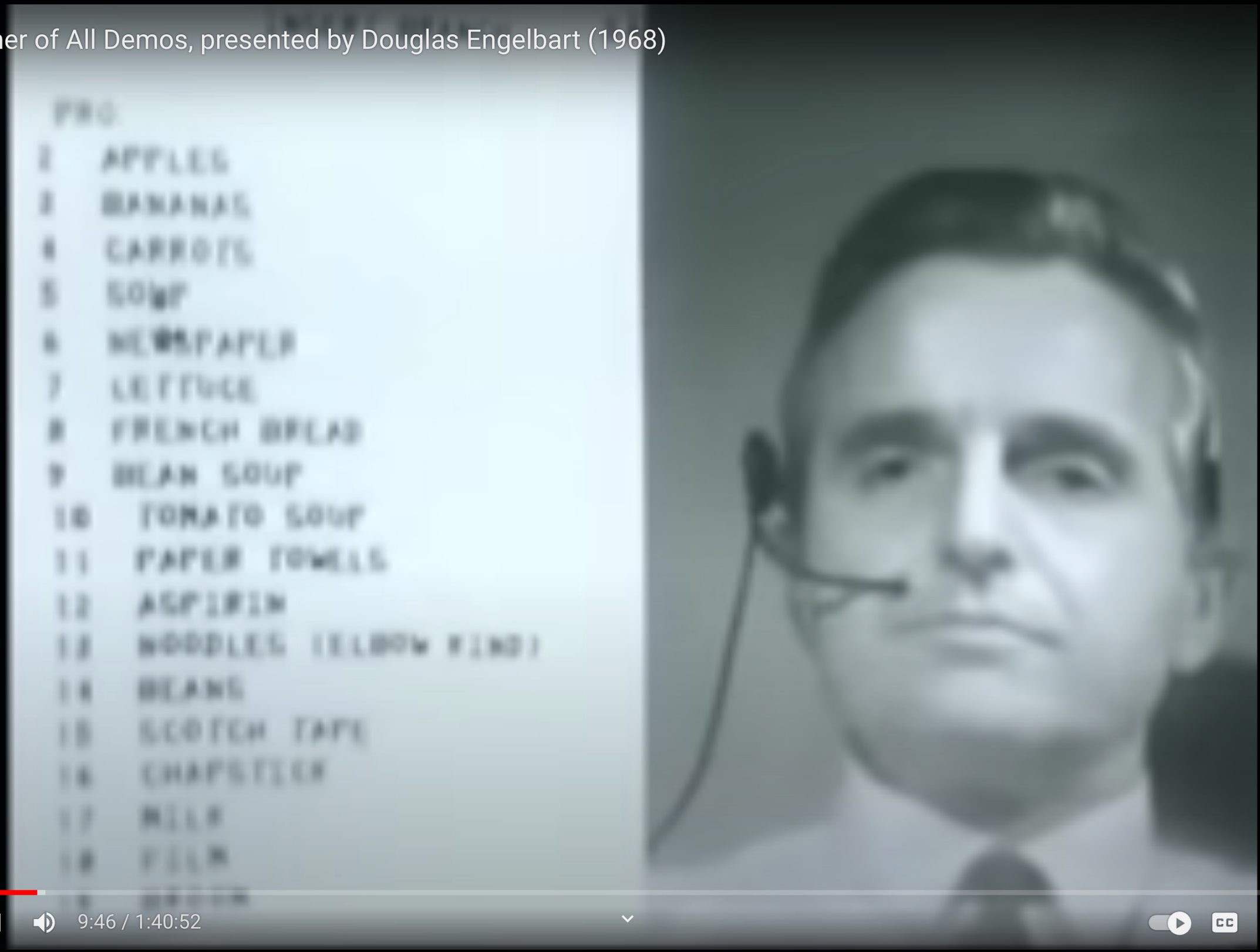
A SCIENTIST OF THE FUTURE RECORDS EXPERIMENTS WITH A TINY CAMERA FITTED WITH UNIVERSAL-FOCUS LENS. THE SMALL SQUARE IN THE EYEGLASS AT THE LEFT SIGHTS THE OBJECT

AS WE MAY THINK

A TOP U. S. SCIENTIST FORESEES A POSSIBLE FUTURE WORLD IN WHICH MAN-MADE MACHINES WILL START TO THINK

Vannear Bush 1945

1968

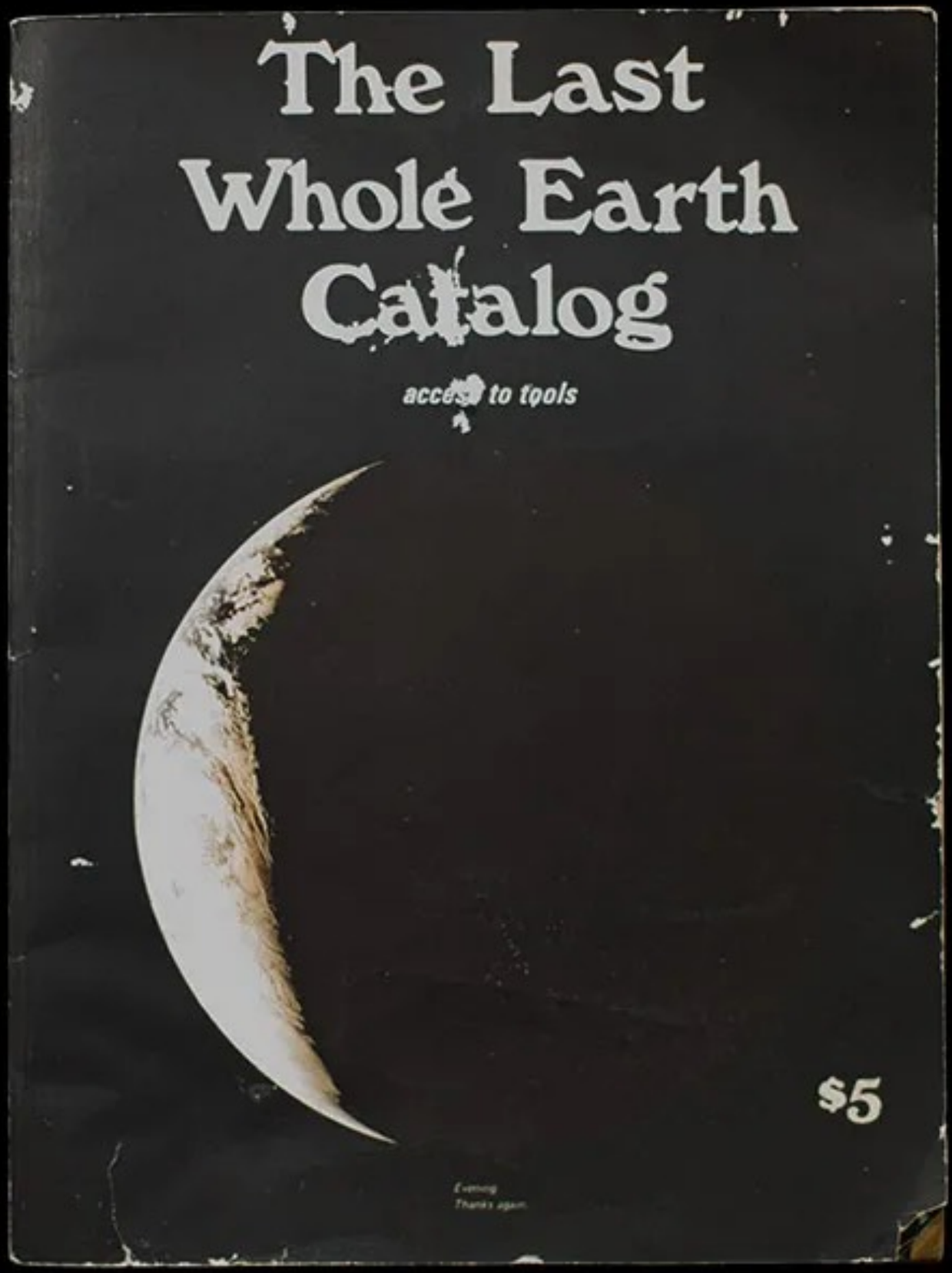


[The Mother of All Demos]

Douglas 1968

Earthrise

Apollo8, 1968



The whole Earth Catalogue

“Information wants to be free.” – Stuart Brand

Whole Earth Catalog created a peer-to-peer network that connected and galvanized like-minded people. In doing so, it created a legacy as an early link between the counterculture movement and the cresting personal tech industry.

It spoke to a growing generation of do-it-yourselfers and innovators—particularly, oh, those entrepreneurs who've found themselves in California over the years.

Indeed, Steve Jobs cited the catalog in his 2006 commencement address at Stanford, calling it "sort of like Google in paperback form, 35 years before Google came along." Its Silicon Valley canonization should also stand as a reminder of the corresponding risks that come with viewing ourselves as gods.

web 1.0:

1989 - 2004

"Ours is a world that is both everywhere and nowhere, but it is not where bodies live."
John Perry Brown

essay: A Declaration of the Independence of Cyberspace, Electronic Frontier Foundation

(AKA one of last week's readings)

Original New Yorker cartoon,
July 5, 1996

The Telecommunications Act of 1996

- Deregulates cable television service
- Allows local telephone companies to provide
- allows local telephone companies to provide cable television service
- requires v-chips in new televisions, which allow parents to block access to objectionable and adult programming
- increases the number of television stations that a single company may own
- bans the knowing transmission of indecent material to minors on the Internet
- Significantly reduced regulations on media concentration and crosss-ownership of media outlets. Less competition, lead to AOL/ Time Warner and Viacom to purchase multiple media outlets in local markets.

browser wars

2nd: 2004 - 2017

Web 2.0

[thefacebook]

quick search go

My Profile [edit]
My Groups
My Friends
My Messages
My Away Message
My Mobile Info
My Account
My Privacy

Scott Pearson's Profile (This is you)

Scranton

Picture [edit]

Visualize My Friends
Edit My Profile
My Account Preferences
My Privacy Preferences

Connection

This is you.

Access

Scott is currently logged in from a non-residential location.

Other Schools [edit]

Information [edit]

Account Info:

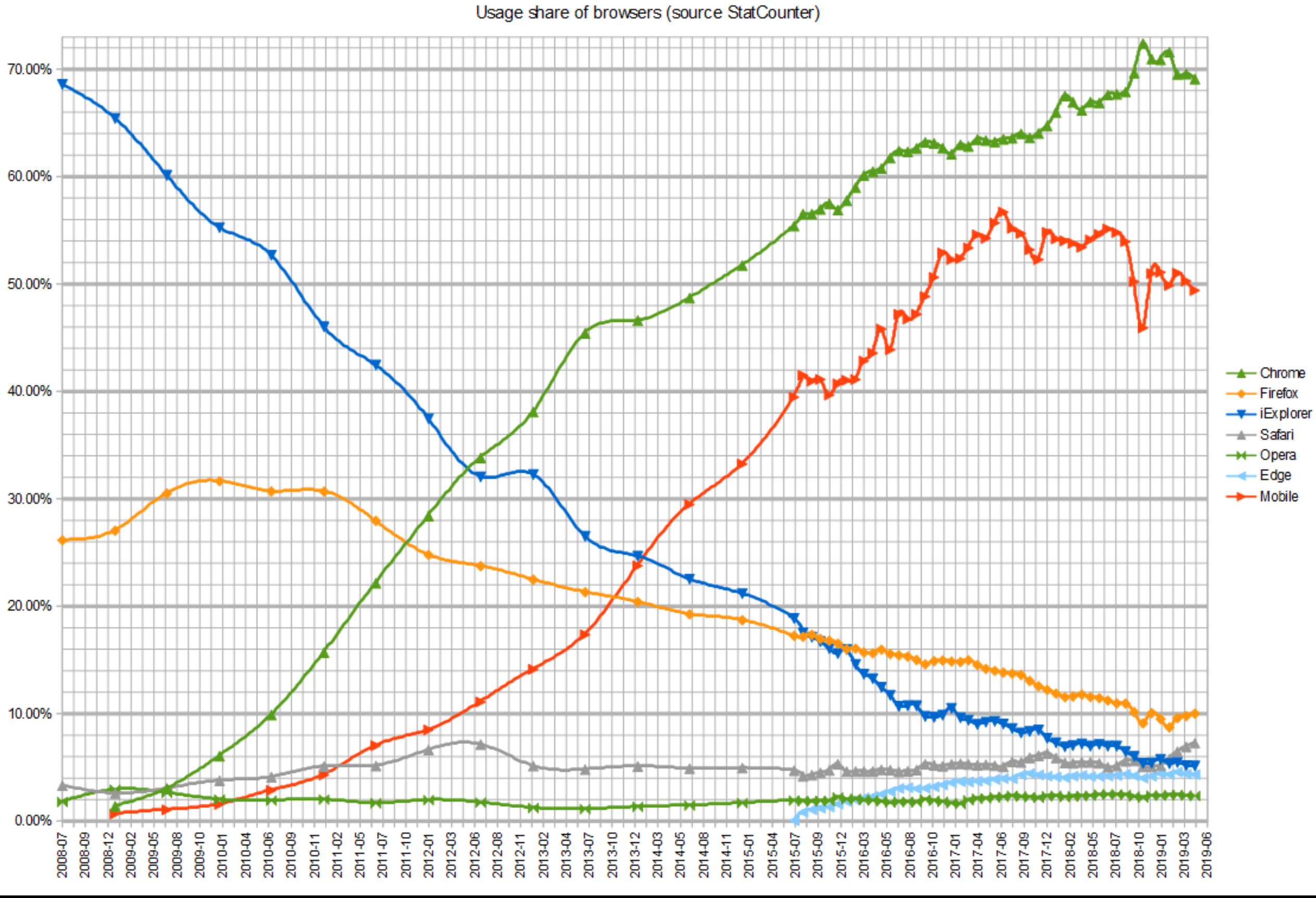
Name:	Scott Pearson, 85
Member Since:	January 12, 2005
Last Update:	February 3, 2005

Basic Info: [edit]

Email:	pearsons2@scranton.edu
Status:	Alumnus/Alumna
Sex:	Male
Year:	2004
Concentration:	Computing Sciences Mathematics
Phone:	570.499.4818
High School:	Dunmore HS '00

Extended Info: [edit]

Screenname:	ScottiePP7
Looking For:	Friendship Dating A Relationship Random play Whatever I can get
Interested In:	Women
Relationship Status:	Single
Political Views:	Liberal
Interests:	Drinking, Football, Basketball, Tennis, saying you'll have that


Usage share of browsers (source StatCounter)
70.00%
60.00%
50.00%
40.00%
30.00%
20.00%
10.00%
0.00%
2008-07
2008-09
2008-12
2009-02
2009-05
2009-08
2009-10
2010-01
2010-04
2010-06
2010-09
2010-11
2011-02
2011-05
2011-07
2011-10
2012-01
2012-03
2012-06
2012-08
2012-11
2013-02
2013-04
2013-07
2013-10
2013-12
2014-03
2014-05
2014-08
2014-11
2015-01
2015-04
2015-07
2015-09
2015-12
2016-03
2016-05
2016-08
2016-10
2017-01
2017-04
2017-06
2017-09
2017-12
2018-02
2018-05
2018-07
2018-10
2019-01
2019-03
2019-06
Chrome
Firefox
iExplorer
Safari
Opera
Edge
Mobile

HTML CSS JS

Interpreted vs. Compiled Programming

Compiled	Interpreted
C++, Java	JavaScript
compiled as machine language	code saved as you write it.
.exe - executable file. Self-standing program	requires library to interpret commands
fast, speed of performance	only language that runs in the browser
Programming language	Scripting language

the DOM

the DOM

When a browser loads a web page, it creates a model of that page

This is called a “DOM tree” and it is stored in the browser’s memory

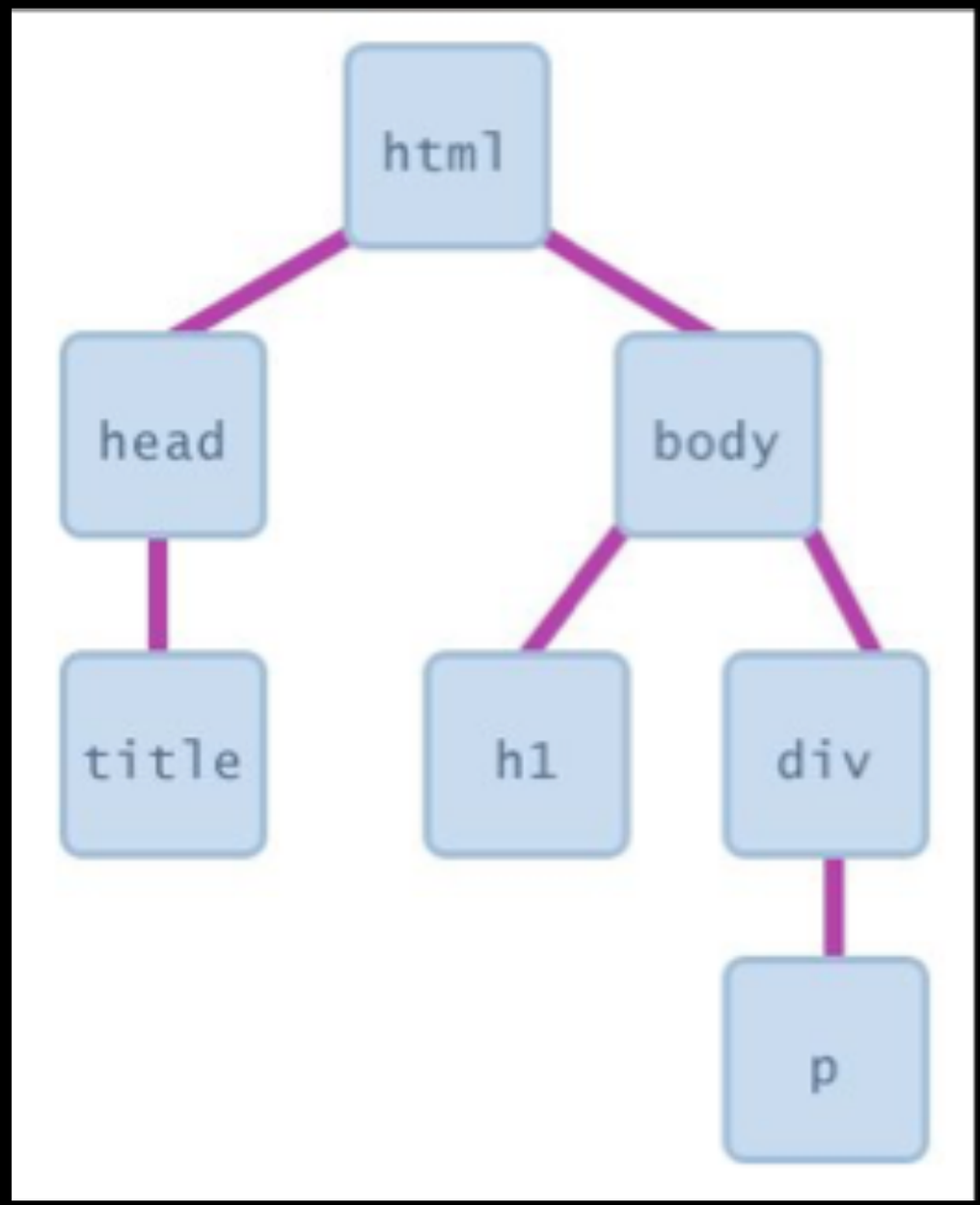
Every element, attribute, and piece of text in the HTML is represented by its own “DOM node”

The DOM

Every element on a page is accessible in JavaScript through the DOM: Document Object Model

The DOM is the tree of nodes corresponding to HTML elements on a page.

Can modify, add and remove nodes on the DOM, which will modify, add, or remove the corresponding element on the page.



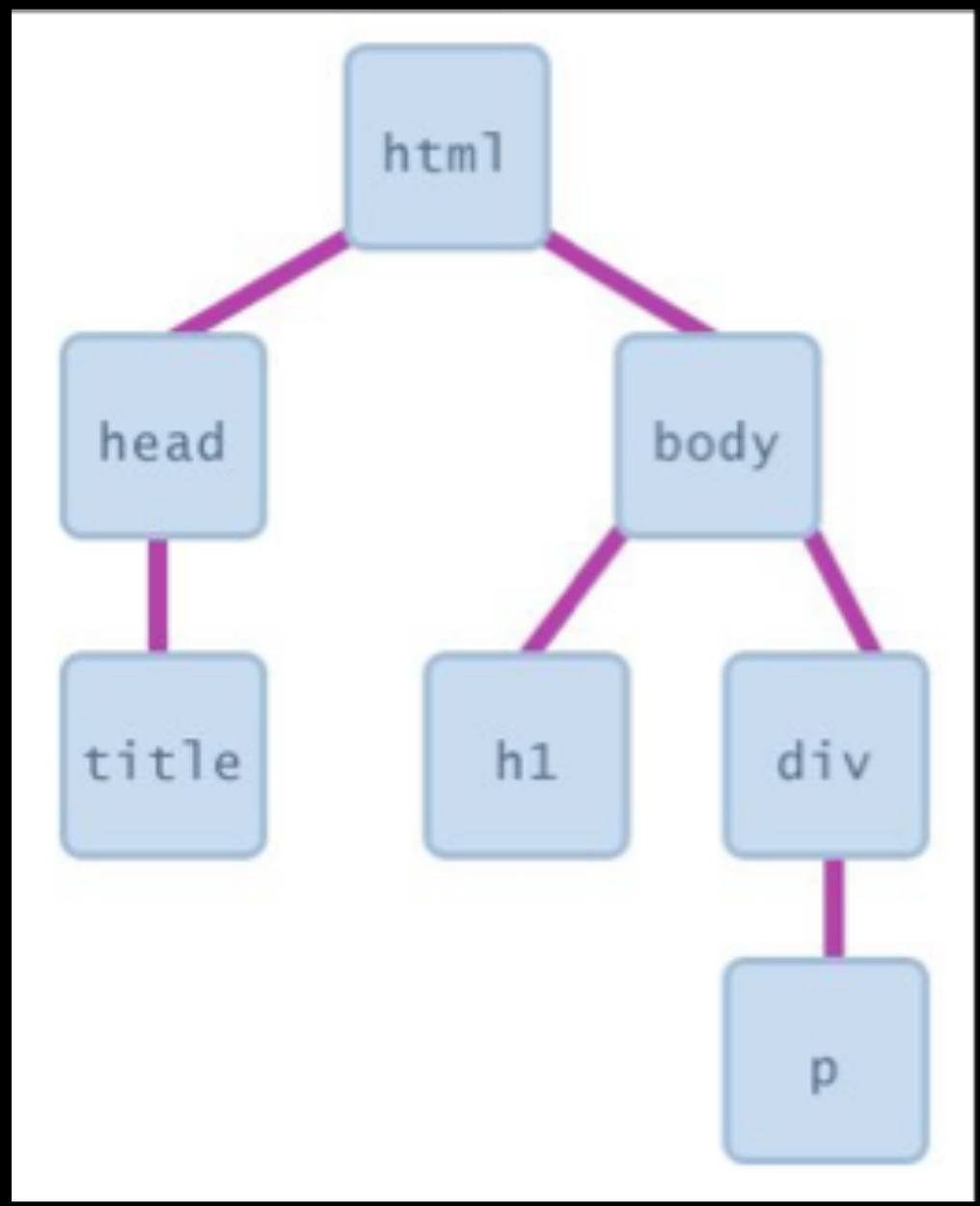
The DOM

The DOM is a tree of node objects corresponding to the HTML elements on a page.

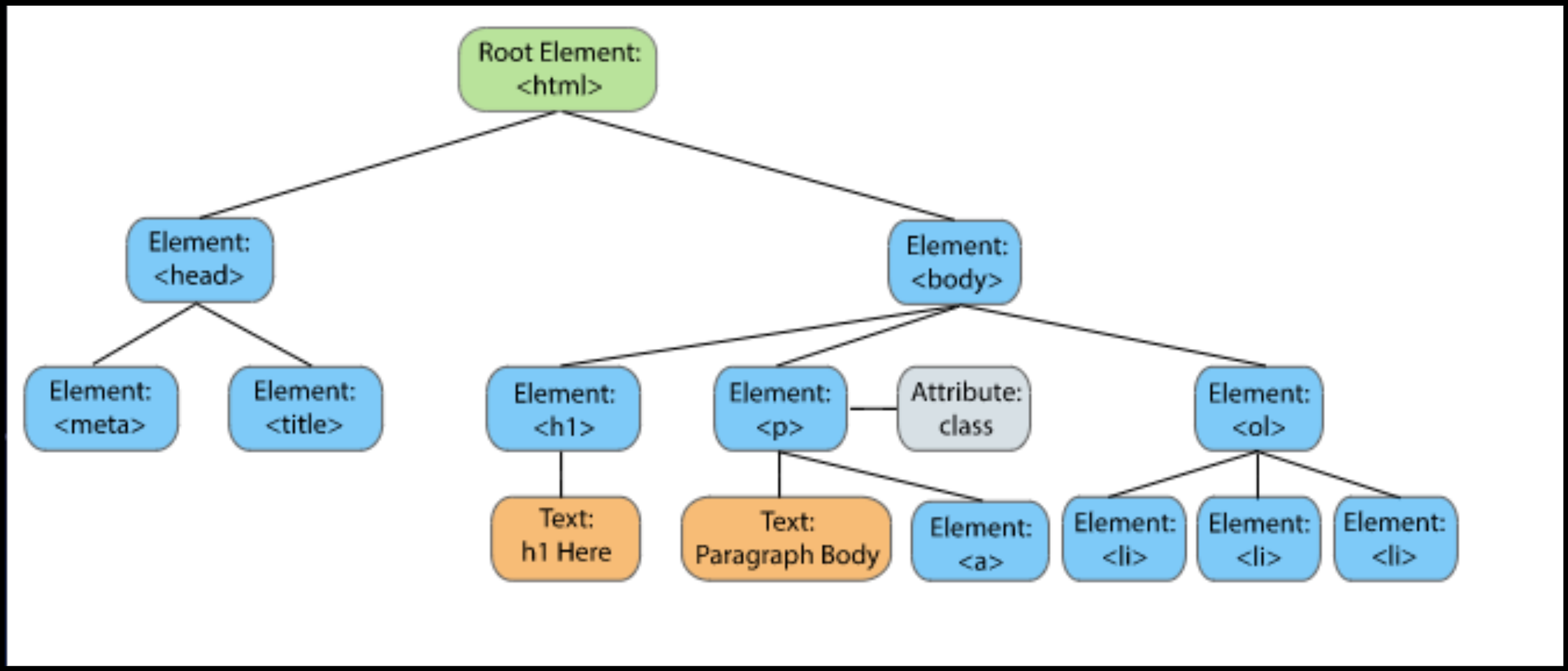
- JS code can examine these nodes to see the state of an element (e.g. to get what the user typed in a text box)

- JS code can edit the attributes of these nodes to change the attributes of an element (e.g. to toggle a style or to change the contents of an <h1> tag)

- JS code can add elements to and remove elements from a web page by adding and removing nodes from the DOM



hello


Root Element:
<html>
Element:
<head>
Element:
<body>
Element:
<meta>
Element:
<title>
Element:
<h1>
Element:
<p>
Attribute:
class
Element:

Text:
h1 Here
Text:
Paragraph Body
Element:
<a>
Element:

Element:

Element:

types of DOM nodes

There are four main types of nodes.

- The **Document** node, which represents the entire page : HTML TAG
- **Element** nodes, which represent individual HTML tags
- **Attribute** nodes, which represent attributes of HTML tags, such as class
- **Text** nodes, which represents the text within an element, such as the *content* of a p tag

We talk about the relationship between element nodes as “parents,” “children,” and “siblings.”

DOM queries

JavaScript methods that find elements in the DOM tree are called “**DOM queries**”

DOM queries may return one element, or they may return a “node list”

Which DOM query you use depends on what you want to do and the scope of browser support required

For Example: JavaScript methods that return a single element node:

```
getElementById(“potato” )

querySelector( )
```

DOM object properties

You can access attributes of an HTML element via a property (field) of the DOM object

```
const theImage = document.querySelector('img');
theImage.src = 'new-picture.png';
```

Exceptions:
Notably, you can't access the class attribute via object.class

Attributes and DOM properties

Roughly every **attribute** on an HTML element is a **property** on its respective DOM object...

HTML

```
<img src =“tree.png“ />
```

JavaScript

```
const myElement = document.querySelector('img');
myElement.src = 'tree.png';
```

(But you should always check the JavaScript spec to be sure. In this case, check the HTMLImageElement.)

HTML

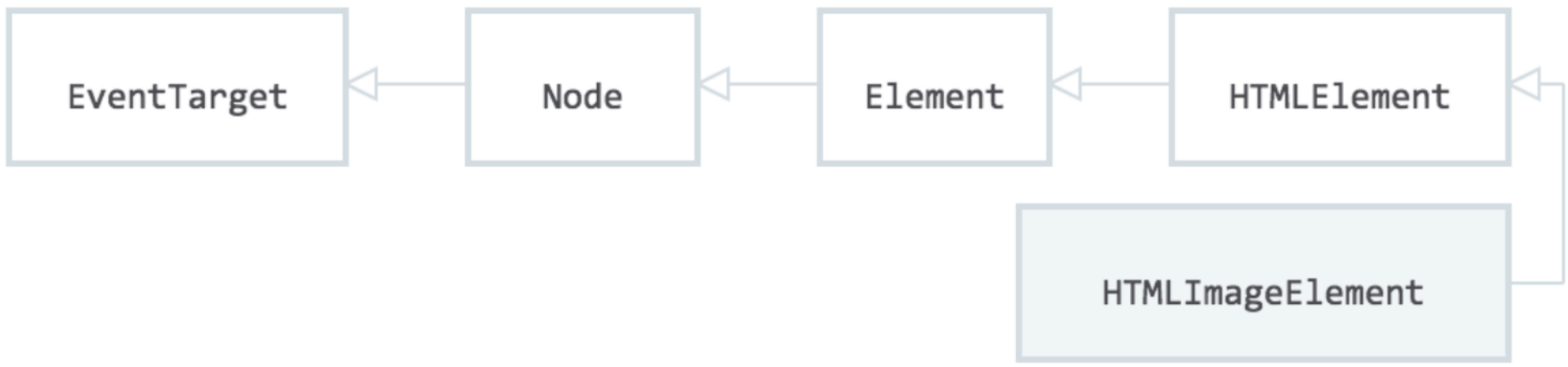
```
<img src =“tree.png” />
```

JavaScript

```
const myElement = document.querySelector('img');
myElement.src = 'tree.png';
```

(But you should always check the JavaScript spec to be sure. In this case, check the HTMLImageElement.)

The **`HTMLImageElement`** interface provides special properties and methods for manipulating `<img>` elements.



Getting DOM objects

We can access an HTML element's corresponding DOM object in JavaScript via the querySelector function:

```
document.querySelector('css selector');
```

This returns the **first** element that matches the given CSS selector
querySelector('h1')

```
document.querySelectorAll('css selector');
```

Returns all the elements that match the given CSS selector

Getting DOM objects

```
let element = document.querySelector('#button');
```

Returns the DOM object for the HTML element with id="button", or null if none exists.

```
let elementList = document.querySelectorAll('.quote, .comment');
```

Returns a list of DOM objects containing all elements that have a "quote" class AND all elements that have a "comment" class.

Adding Event Listeners

Each DOM object has the following function:

addEventListener(event name, function name);

event name is the string name of the JavaScript event you want to listen to

- Common ones: click, focus, blur, etc

function name is the name of the JavaScript function you want to execute when the event fires

Removing Event Listeners

Each DOM object has the following function:

removeEventListener(event name, function name);

event name is the string name of the JavaScript event you want to stop listening to

function name is the name of the JavaScript function you no longer want to execute when the event fires

Event Listeners

Let's print "Clicked" to the Web Console when the user clicks the given button:

```
<button> Click!!</button>
```

We need to add an event listener to the button...

How do we talk to an element in HTML from JavaScript?

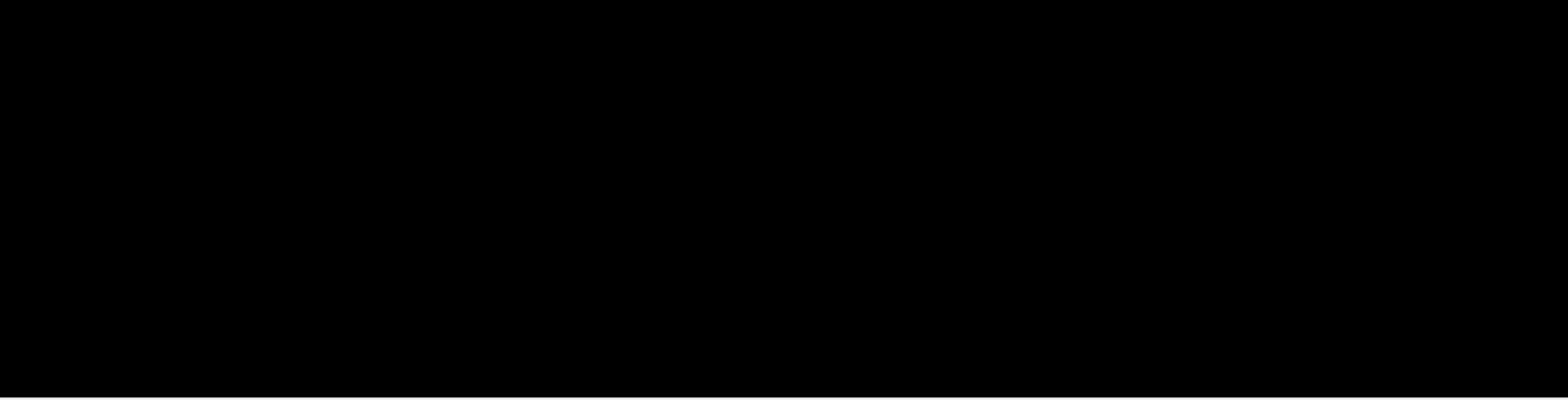
Adding Event Listeners

```
<button id="myButton"> Click!!</button>
```

```
function clickButton() {
    consol.log("!!");

}

const theClick = document.querySelector('#myButton');
theClick.addEventListener('click', clickButton);
```


▶Uncaught TypeError: Cannot read property 'addEventListener' of null
theSketch.js:7
at theSketch.js:7
theSketch.js:7

Error! WHY ?

```
<!DOCTYPE html>

<head>
    <title>Week 7 - JS</title>
    <meta name="viewport" content="user-scalable=no,
    width=device-width, initial-scale=1, maximum-scale=1">
    <link href="style.css" rel="stylesheet" type="text/css">
    <script src="theSketch.js"></script>
    <meta charset="utf-8">

</head>

<body>
    <p>
        this great website
    </p>

        <button id="myButton"> Click!!</button>

</body>

</html>
```

Error! WHY ?

defer

You can add the defer attribute onto the script tag so that the JavaScript doesn't execute until after the DOM is loaded (mdn):

```
<script src="script.js" defer></script>
```

Other old school ways of doing this **(not best practice - don’t do!)**:

- Put the <script> tag at the bottom of the page
- Listen for the “load” event on the window object

You will see tons of examples on the internet that do this. They are out of date. defer is wide supported and better

Types of expressions

Expressions that assign a value to a variable

```
let theMovie = "Blade Runner";
```

Expressions that use two or more values to return a single value

```
let theHeight = 50 * 3;
let theSentence = "My name is " + "Rebecca";
```

Arithmetic Operators in JS

NAME	OPERATOR	PURPOSE & NOTES	EXAMPLE	RESULT
ADDITION	+	Adds one value to another	10+5	15
SUBTRACTION	-	Substracts one value from another	10-5	5
DIVISION	/	Divides two values	10/5	2
MULTIPLICATION	*	Multiplies two values	10*5	50
INCREMENT	++	Adds one to the current number	i=10; i++;	11
DECREMENT	- -	Subtracts one from the current number	i=10; i- -;	9
MODULUS	%	Divides two values and returns the remainder	10%3;	1

String Operators in JS

There is only one string operator **+**

It's used to join the strings together

```
let firstName = "Rebecca";
let lastName = "Leopold";
let fullName = firstName + lastName;
```

Process of joining two or more strings together into a new one is: **concatenation**.

If you'll try to add other arithmetic operators on a string, it will return NaN

```
let fullName = firstName * lastName;
returns: "Not a Number"
```

Logical Operators

LOGIC	OPERATOR	EXAMPLE	NOTES
AND	**&&**	exprss1 **&&** express2	Returns expr1 if it can be converted to **false**; otherwise, returns expr2. Thus, when used with Boolean values, **&&** returns **true** if both operands are **true**; otherwise, returns **false**.
OR	**||**	exprss1 **||** express2	Returns expr1 if it can be converted to **true**; otherwise, returns expr2. Thus, when used with Boolean values, **||** returns **true** if either operand is **true**.
NOT	**!**	**!** express	Returns **false** if its single operand can be converted to **true**; otherwise, returns **true**.

An expression results in a single value (produces a value and is written whenever a value is expected).

Arrays

An array is a special type of variable. It doesn't just store one value; it stores a list of values.

You should use an array whenever you're working with a list of values that are related to each other. (ex: an array of images you want the user to click through, an array of colors to randomize a design)

Create an array and give it a name just like any other variable:

```
let theMovies = [ ];
```

2. Create an array using array literal technique:

```
let theMovies = [“Blade Runner”, “Sorry to Bother You”, “Groundhog Day”];
```

3. Create an array using array constructor technique:

```
let theMovies = new Array (“Blade Runner”, “Sorry to Bother You”, “Groundhog Day”);
```

Note: values in an array do **not have to be the same data type** (could be string, number, boolean in one array).

Note2: array literal is the preferred way to create an array in JS.

Values in Arrays

Values in an array are accessed through their numbers they are assigned in the list. The number is called index and starts from 0.

```
let theMovies = [“Blade Runner”, “Sorry to Bother You”, “Groundhog Day”];
                      index [0]          index [1]              index [2]
```

You can check the number of items in an array using **length** property

```
let theMoviesLength = theMovies.length;
```

Changing Values in Arrays

Let's say we want to update the value of the second item (change "Sorry to Bother You" to something else)

To access the current third value, we have to call the name of the array followed by the index number for that value inside the square brackets:

```
let theMovies = ["Blade Runner", "Sorry to Bother You", "Groundhog Day"];
theMovies[1]
```

After we select a value, we can assign a new value to it:

```
theMovies[1] = "Clueless";
```

When we log the updated array, we see updated values:

```
console.log(theMovies) ;
    ["Blade Runner", "Clueless","Groundhog Day"];
```

Adding and removing values from the array

You can add values to the array using .**push()** method:

```
let theMovies = [“Blade Runner”, “Clueless”,“Groundhog Day”];
theMovies.push(“The Shining”);
```

You can remove values from the array using **.splice()** method:

```
let theMovies =  [“Blade Runner”, “Clueless”,“Groundhog Day”, “The Shining”];
theMovies.splice(0,1); (will remove “Blade Runner” movie)
```

Here - 0 is an index at what position an item should be removed and 1 how many items should be removed (in this case only one movie)

For Loops

A For loop uses a counter as a condition.
It instructs code to run a specified number of times.
Good to use when the number of repetitions is known, or can be supplied by the user.

```
let theYear = [“January”, “February”,  “March”, “April”, “May”, “June”, “July”, “August”, “September”, “October”, “November”, “December”];
```

Keyword **Condition (counter)**

```
for (let i = 0; i < theYear.length; i ++){
        console.log(theYear[i]);
    }
```

code to execute during loop

For Loops

We can use for loops to programmatically work through arrays + get their values.

```
let theYear = [“January”, “February”,  “March”, “April”, “May”, “June”, “July”, “August”, “September”, “October”, “November”, “December”];
```

Keyword

Condition (counter)

```
for (var theIndex = 0; theIndex < theYear.length; theIndex ++){
        console.log(theYear[theIndex]);
    }
```

code to execute during loop

While Loops

Good to use in applications with numeric situations and when we don’t know how many tim
the code should run.
In other words: the loop repeats until a certain “condition” is met.
If the condition is false at the beginning of the loop, the loop is never executed.

```
let theIndex = 1;

while ( theIndex < 10 ){
  console.log( theIndex );
  theIndex ++;
}
```

Functions group a series of statements together to perform a specific task.

Prgrmmng Vocabulary

When you ask function to perform its task, you **call** a function

Some functions need to be provided with information in order to achieve a given task - pieces of information passed to a function are called **parameters**

When you write a function and expect it to provide you with an answer, the response is known as **return value**

Declaring a function

To create a function you give it a name and write statements inside to achieve a task you want it to achieve

function keyword **function name**

```
function buttonClicked() {

    console.log("hello");


}
```

code statement

code block inside curly braces

Calling a function

You call a function by writing it's name somewhere in the code.

```
function buttonClicked() {
    console.log("hello");
}

buttonClicked() // code after
```

Declaring functions with parameters

Sometimes a function needs specific information to perform it's task **(parameters)**
Inside the function the parameters act as variables

parameters

```
function countTotal(itemNumber, price) {
    return itemNumber * price;
}
```

parameters are used like variables inside the function

Calling functions with parameters

When you call a function that has **parameters,** you need to specify the values it should take in. Those values are called **arguments.**

Arguments are written inside parentheses when you call a function + can be provided as values or as variables

```
function countTotal(itemNumber, price) {
        return itemNumber * price;
}

countTotal(7,15); //will return 105

(itemNumber = 7 * price = 15) countTotal(itemNumber, price);
```

Using “return” in a function

return is used to return a value to the code that called the function

The interpreter leaves the function when return is used and goes back to the statement that called it

```
function countTotal(itemNumber, price) {
  return itemNumber * price;
  // interpreter will skip any code found after return
}
```

Variable Scope

Where you declare a variable affects where it can be used within your code

If it's declared inside a function, it can only be used inside that function

It's known as variable's **scope**

Local + Global Variables

When a variable is created inside a function
It's called **local variable** or **function-level variable**

When a variable is created outside a function
It's called **global variable** can be called anywhere
the in code + will take up more memory

```
var salesTax = .08

function countTotal(itemNumber, price) {
      var theSum = itemNumber * price;
      var theTax = theSum * salesTax;
      var theTotal = theSum + theTax;
      return theTotal;
}

console.log(typeof salesTax);
```

CSS variables

variables are a relatively new CSS feature.

CSS variables are entities defined by CSS authors that contain specific values to be reused throughout a document. They are set using custom property notation (e.g., --main-color: black;) and are accessed using the **var( )** function (e.g., color: var(—main-color);).

CSS variables are subject to the cascade and inherit their value from their parent.

```
:root {
  --thisGreat-color: black: hotpink;
}

h1 {
  background-color: var(--thisGreat-color);
}
```

function scope w/ var

```
function printMessage(message, times) {

  for (var i = 0; i < times; i++) {
    console.log(message);
  }

  console.log('Value of i is ' + i);
}

printMessage('hello', 3);
```

```
3 hello                          theSketch.js:5
  Value of i is 3                theSketch.js:7
>
```

var

The value of "i" is readable outside of the for-loop because variables declared with **var** have function scope.

function scope w/ var

```
var x = 10;

if (x > 0) {
    var y = 10;
}

console.log('Value of y is ' + y);
```

```
Value of y is 10                    theSketch.js:11
```

Variables declared with **var** have function-level scope and do not go out of scope at the end of blocks; only at the end of functions

Therefore you can refer to the same variable after the block has ended (e.g. after the loop or if-statement in which they are declared)

function scope w/ var

```
function willThisWork() {
    var x = 10;
    if (x > 0) {
        var y = 10;
    }
    console.log('y is ' + y);
}

willThisWork();
console.log('y is ' + y);
```

```
y is 10                                          theSketch.js:8
▶ Uncaught ReferenceError:                       theSketch.js:12
y is not defined
    at theSketch.js:12
```

But you can't refer to a variable outside of the function in which it's declared.

scope w/ let

```
function printMessage(message, times) {

    for (let i = 0; i < times; i++) {
        console.log(message);
    }
  console.log('Value of i is ' + i);
}
```

```
printMessage('hello', 3);
```

```
3 hello                                          theSketch.js:6
▶ Uncaught ReferenceError: i                     theSketch.js:8
is not defined
    at printMessage (theSketch.js:8)
    at theSketch.js:12
```

let has block-scope so this results in an error

scope w/ const

```
let x = 10;

if (x > 0) {
    const y = 10;
}

console.log(y)
```

```
▶Uncaught ReferenceError: y is not defined
    at theSketch.js:10
```


like, **let** - **const** has block-scope, so accessing the variable outside the block results in an error

const

```
const y = 10;
y = 0; //error
y++; //error
```

```
Uncaught TypeError:                    theSketch.js:5
Assignment to constant variable.
    at theSketch.js:5
```

```
const myList = [1, 2, 3];
myList.push(4); //okay

console.log(myList);
```

```
▶ (4) [1, 2, 3, 4]
```

const declared variables cannot be reassigned.

However, it doesn't provide true **const** correctness, so you can still modify the underlying object

- (In other words, it behaves like Java's final keyword and not C++'s const keyword)

const vs. let

```
let y = 55;
y = 0; // okay
y++;     // okay

let myList = [1,2,3];
myList.push(4);
console.log('y is ' + y);
console.log(myList);
```

```
y is 1                                  theSketch.js:9
▶ (4) [1, 2, 3, 4]                      theSketch.js:10
```

let can be reassigned, which is the difference between **const** **and** **let**

JS Syntax - Variables best practices

- Use **const** whenever possible.

- If you need a variable to be reassignable, use **let**.

- Don't use **var.**
  You will see a ton of example code on the internet with var since const and let are relatively new.
  However, **const** and **let** are well-supported, so there's no reason not to use them.
  (This is also what the Google and AirBnB JavaScript Style Guides recommend.)

You do not declare the datatype of the variable before using it ("dynamically typed")

JS Variables do not have types, but the values do.

There are six primitive types (mdn):

- **Boolean** : true and false
- **Number** : everything is a double (no integers)
- **String**: in 'single' or "double-quotes"
- **Symbol:** (skipping this today)
- **Null:** null: a value meaning "this has no value"
- **Undefined:** the value of a variable with no value assigned

There are also Object types, including Array, Date, String (the object wrapper for the primitive type), etc.

Data Type: Numbers

```
const homework = 0.45;
const midterm = 0.2;
const final = 0.35;
const score = homework * 87 + midterm * 90 + final * 95;
console.log(score); // 90.4
```

- All numbers are floating point real numbers. No integer type.
- Operators are like Java or C++.
- Precedence like Java or C++.
- A few special values: **NaN** (not-a-number), **+Infinity**, **-Infinity**
- There's a Math class: **Math**.floor, **Math**.ceil, etc.

Data Type: Boolean

```
if (username) {
    // username is defined
}
```

Non-boolean values can be used in control statements, which get converted to their "**truthy**" or "**falsy**" value:

- null, undefined, 0, NaN, '', "" evaluate to false
- Everything else evaluates to true

Equality

== (is equal to)

Compares two values to see if they are the same

!= (is not equal to)

Compares two values to see if they are not the same

=== (strict equal to)

Compares two values to check that both the data and value are the same

!== (strict not equal to)

Compares two values to check that both the data and value are not the same

Equality

JavaScript's `==` and `!=` are basically broken: they do an implicit type conversion before the comparison.

```
// false
'' == '0';
// true
'' == 0;
// true
0 == '0';
// false
NaN == NaN;
// true
[''] == '';
// false
false == undefined;
// false
false == null;
// true
null == undefined;
```

Equality

Instead d of fixing `==` and `!=` ,
the ECMAScript standard kept
the existing behavior but added
`===` and `!==`

```
// false
'' === '0';
// false
'' === 0;
// false
0 === '0';
// false -??
NaN === NaN;
 // false
[''] === '';
// false
false === undefined;
 // false
false === null;
// false
null === undefined;
```

Always use `===` and `!==` and don't use `==` or `!=`

Null + Undefined

What's the difference?

null is a value representing the absence of a value, similar to null in Java and nullptr in C++.

undefined is the value given to a variable that has not been a value.

```
let x = null;
let y;
console.log(x);
console.log(y);
```

```
null                    theSketch.js:7
undefined               theSketch.js:8
```

Null + Undefined

What's the difference?

null is a value representing the absence of a value, similar to null in Java and nullptr in C++.

undefined is the value given to a variable that has not been a value.

... however, you can also set a variable's value to undefined bc … javascript.

```
let x = null;
let y = undefined;
console.log(x);
console.log(y);
```

```
null                    theSketch.js:7
undefined               theSketch.js:8
```