

"the others"



The shock of recognition! In an electric information environment, minority groups can no longer be contained-ignored. Too many people know too much about each other. Our new environment compels commitment and participation. We have become irrevocably involved with, and responsible for, each other.

there
is
absolutely
no
inevitability
as
long
as
there
is
a
willingness
to
contemplate
what
is
happening

the medium is the message
— Marshal McLuhan

THE MEDIUM
IS THE
MESSAGE

THE MEDIUM
IS THE
MESSAGE

THE MEDIUM
IS THE
MESSAGE

THE MEDIUM
IS THE
MESSAGE

THE



THE MEDIUM
IS THE
MESSAGE

THE MEDIUM
IS THE
MESSAGE

THE MEDIUM
IS THE
MESSAGE

THE



Winnip

The electric light is pure information. It is a medium without a message, as it were, unless it is used to spell out some verbal ad or name. This fact, characteristic of all media, means that the "content" of any medium is always another medium. The content of writing is speech, just as the written word is the content of print, and print is the content of the telegraph.

The electric light is pure information. It is a medium without a message, as it were, unless it is used to spell out some verbal ad or name. This fact, characteristic of all media, means that the "content" of any medium is always another medium. The content of writing is speech, just as the written word is the content of print, and print is the content of the telegraph.

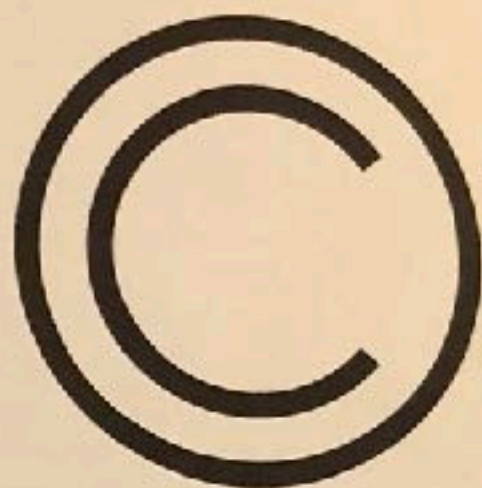
If it is asked, "What is the content of speech?," it is necessary to say, "It is an actual process of thought, which is in itself nonverbal." An abstract painting represents direct manifestation of creative thought processes as they might appear in computer designs. What we are considering here, however, are the psychic and social consequences of the designs or patterns as they amplify or accelerate existing processes. For the "message" of any medium or technology is the change of scale or pace or pattern that it introduces into human affairs.

the medium is the message

An inventory of Effects

— Marshal McLuhan + Quentin Fiore

Innumerable confusions and profound feeling of despair invariably emerge in period of great technological and cultural transitions. Our "Age of Anxiety" is, in great part, the result of trying to do today's job with yesterday's tools - with yesterday's concepts.



"Authorship"—in the sense we know it today, individual intellectual effort related to the book as an economic commodity—was practically unknown before the advent of print technology. Medieval scholars were indifferent to the precise identity of the "books" they studied. In turn, they rarely signed even what was clearly their own. They were a humble service organization. Procuring texts was often a very tedious and time-consuming task. Many small texts were transmitted into volumes of miscellaneous content, very much like "jottings" in a scrapbook, and, in this transmission, authorship was often lost.

The invention of printing did away with anonymity, fostering ideas of literary fame and the habit of considering intellectual effort as private property. Mechanical multiples of the same text created a public—a reading public. The rising consumer-oriented culture became concerned with labels of authenticity and protection against theft and piracy. The idea of copyright—"the exclusive right to reproduce, publish, and sell the matter and form of a literary or artistic work"—was born.

Xerography—every man's brain-picker—heralds the times of instant publishing. Anybody can now become both author and publisher. Take any book on any subject and custom-make your own book by simply xeroxing a chapter from this one, a chapter from that one—instant steal!

As new technologies come into play, people are less and less convinced of the importance of self-expression. Teamwork succeeds private effort.

A ditto, ditto device.

" " " "

A ditto, ditto device.

" " " "

A ditto, ditto device.

" " " "

Authorship"—in the sense we know it today, individual intellectual effort related to the book as an economic commodity—was practically unknown before the advent of print technology. Medieval scholars were indifferent to the precise identity of the "books" they studied. In turn, they rarely signed even what was clearly their own. They were a humble service organization. Procuring texts was often a very tedious and time-consuming task. Many small texts were transmitted into volumes of miscellaneous content, very much like "jottings" in a scrapbook, and, in this transmission, authorship was often lost.

The invention of printing did away with anonymity, fostering ideas of literary fame and the habit of considering intellectual effort as private property. Mechanical multiples of the same text created a public—a reading public. The rising consumer oriented culture became concerned with labels of authenticity and protection against theft and piracy. The idea of copyright—"the exclusive right to reproduce, publish, and sell the matter and form of a literary or artistic work"—was born.

Xerography—every man's brain-picker—heralds the times of instant publishing. Anybody can now become both author and publisher. Take any books on any subject and custom-make your own book by simply xeroxing a chapter from this one, a chapter from that one—instant steal!

As new technologies come into play, people are less and less convinced of the importance of selfexpression. Teamwork succeeds private effort.

A ditto, ditto device.

“ “ “ ”

A ditto, ditto device.

“ “ “ ”

A ditto, ditto device.

“ “ “ ”

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

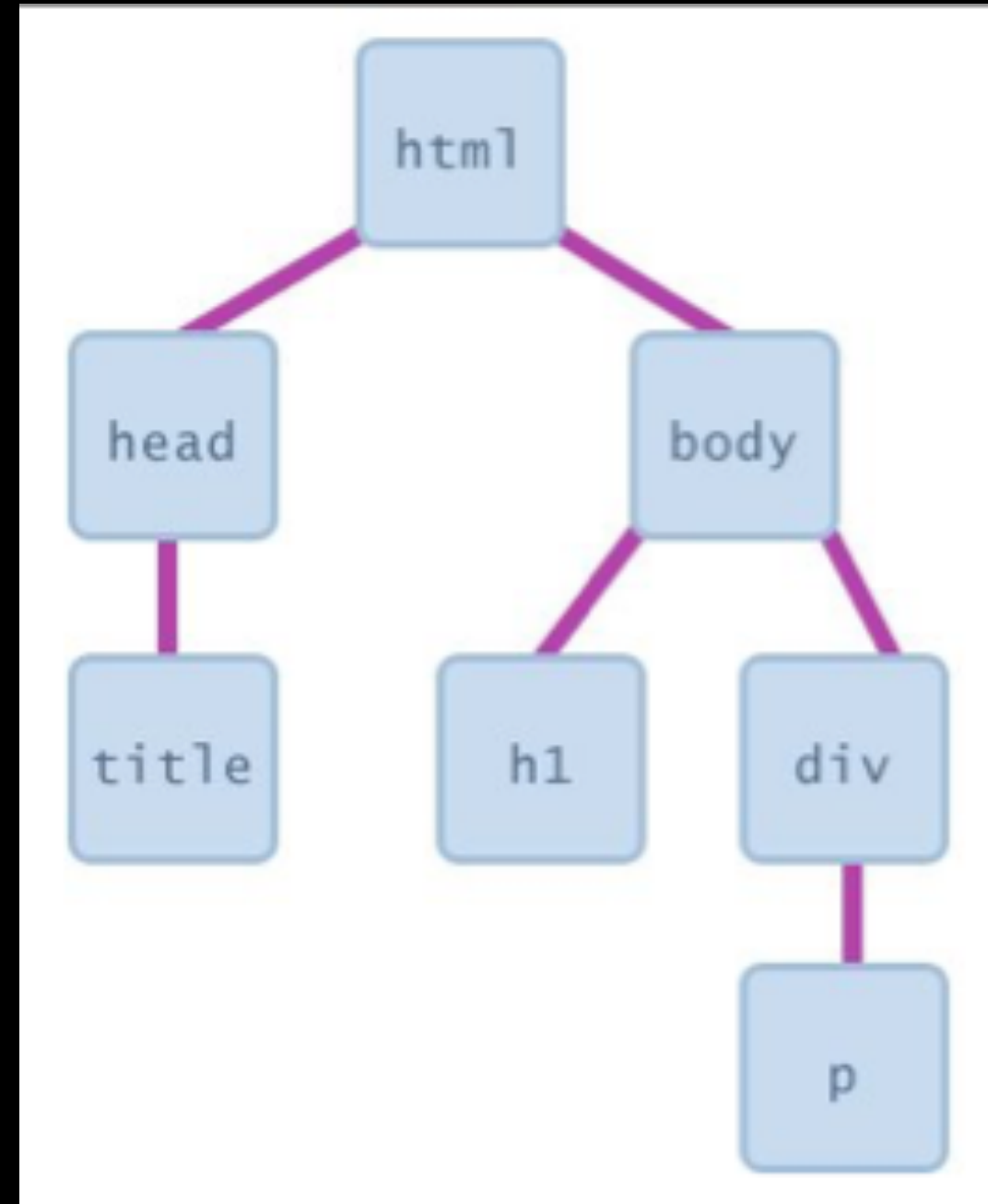
This is called a “DOM tree” and it is stored in the browser’s memory

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

The DOM

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



types of DOM nodes

There are four main types of nodes.

- The **Document** node, which represents the entire page
- **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()

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

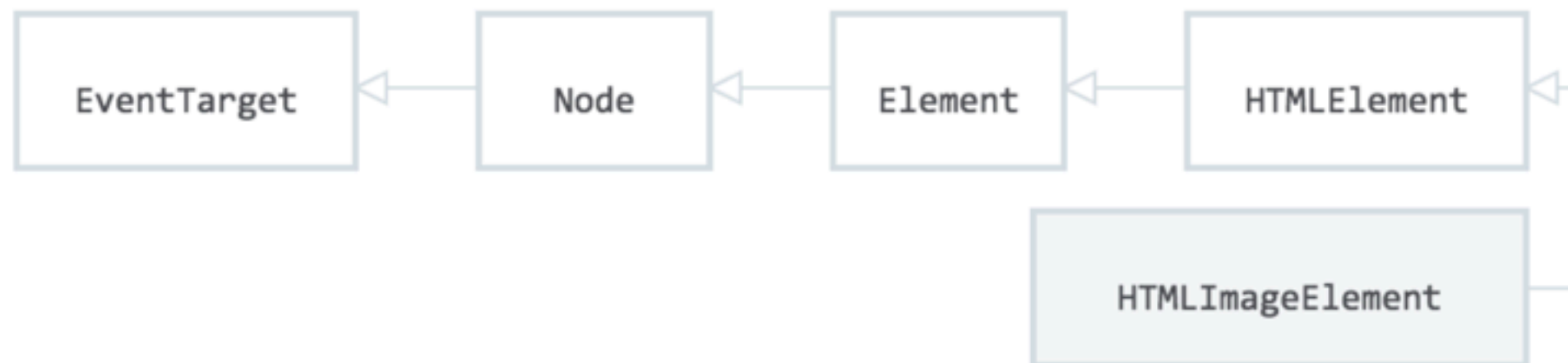
```
<img src = "tree.png" />
```

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 **** elements.



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

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	-	Subtracts 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 .

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 (var 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)
<pre>for (var theIndex = 0; theIndex < theYear.length; theIndex ++){ console.log(theYear[theIndex]); }</pre>	
	code to execute during loop

While Loops

Good to use in applications with numeric situations and when we don't know how many times 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() {
```

code statement

```
    console.log("hello");
```

```
}
```

code block inside curly braces

Calling a function

You call a function by writing its 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 its 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);
```

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	<u>theSketch.js:5</u>
Value of i is 3	<u>theSketch.js:7</u>
>	

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: theSketch.js:10
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: 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 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