What is Web 2.0

"the network as platform"

- Services, not packaged software, with cost-effective scalability
- Control over unique, hard-to-recreate data sources that get richer as more people use them
- Trusting users as co-developers
- Harnessing collective intelligence
- Leveraging the long tail through customer self-service
- Software above the level of a single device
- Lightweight user interfaces, development models, AND business models

— **Tim O'Reilly**, 2005

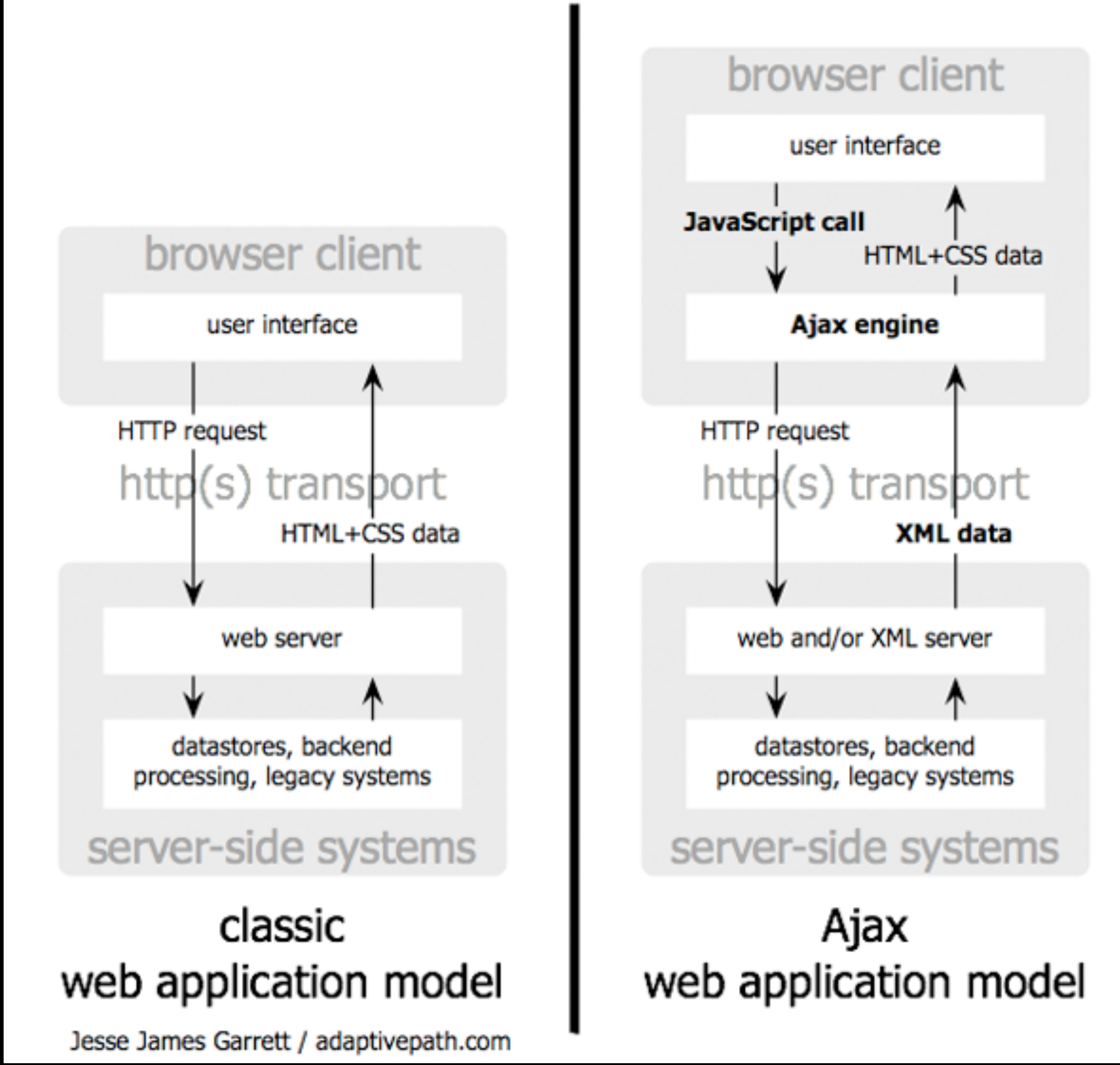
Web Squared

"search as vote"

The collection of technologies used by Google was christened **AJAX**, in a seminal essay by Jesse James Garrett of web design firm Adaptive Path.

"**Ajax** isn't a technology. It's really several technologies, each flourishing in its own right, coming together in powerful new ways.

Ajax incorporates:
- standards-based presentation using **XHTML** and **CSS**;
- dynamic display and interaction using the Document Object Model (DOM);
- data interchange and manipulation using XML and XSLT;
- asynchronous data retrieval using XMLHttpRequest;
- + **JavaScript** binding everything together."



Jesse James Garrett, 2005

(To view this link you'll need the Wayback Machine)

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 They are out of date.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

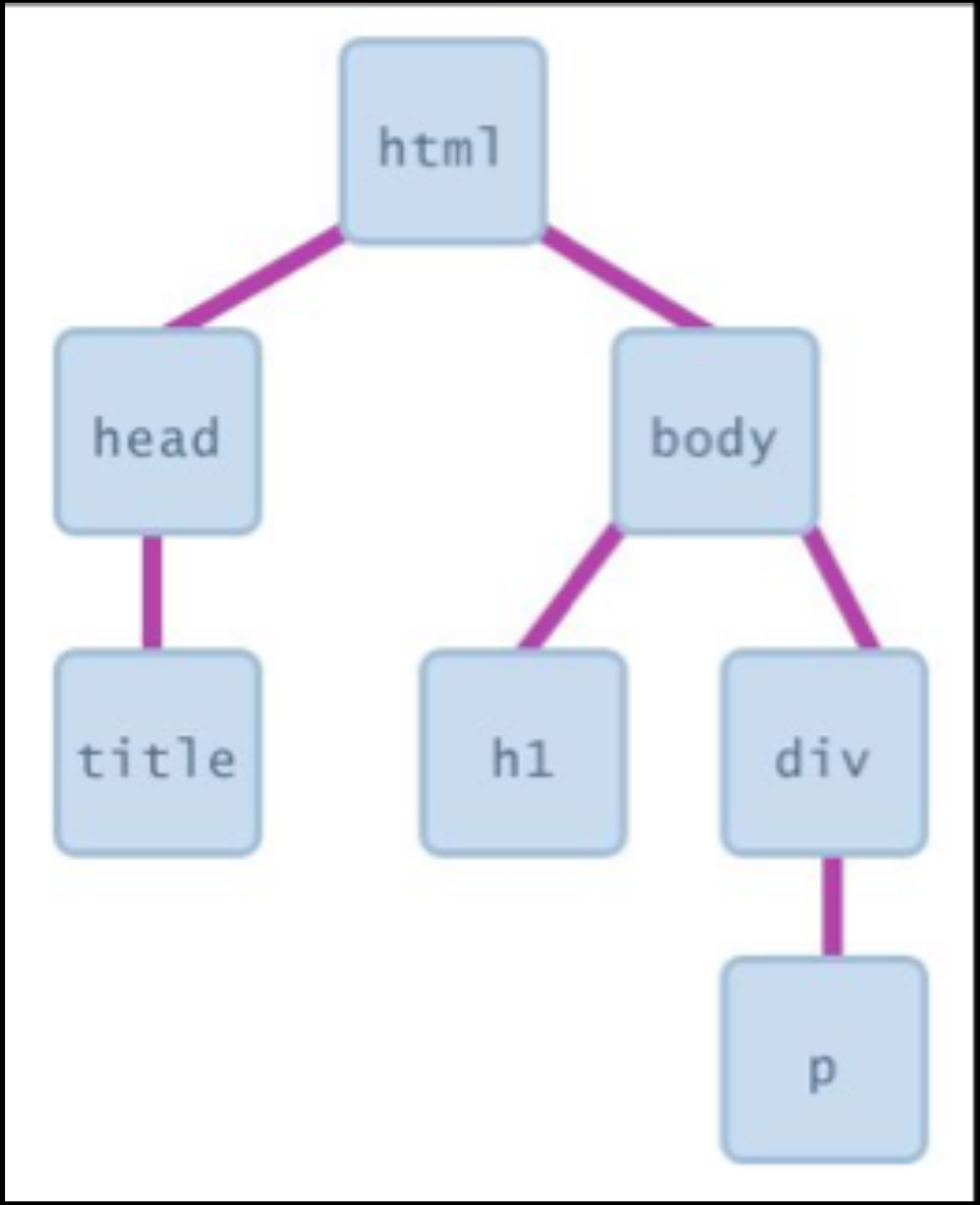
The DOM

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



tech details

The DOM objects that we retrieve from querySelector and querySelectorAll have types:

- Every DOM node is of general type Node (an interface)
- Element implements the Node interface

(FYI: This has nothing to do with NodeJS, if you've heard of that)

- Each HTML element has a specific Element derived class, like HTMLImageElement

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.)

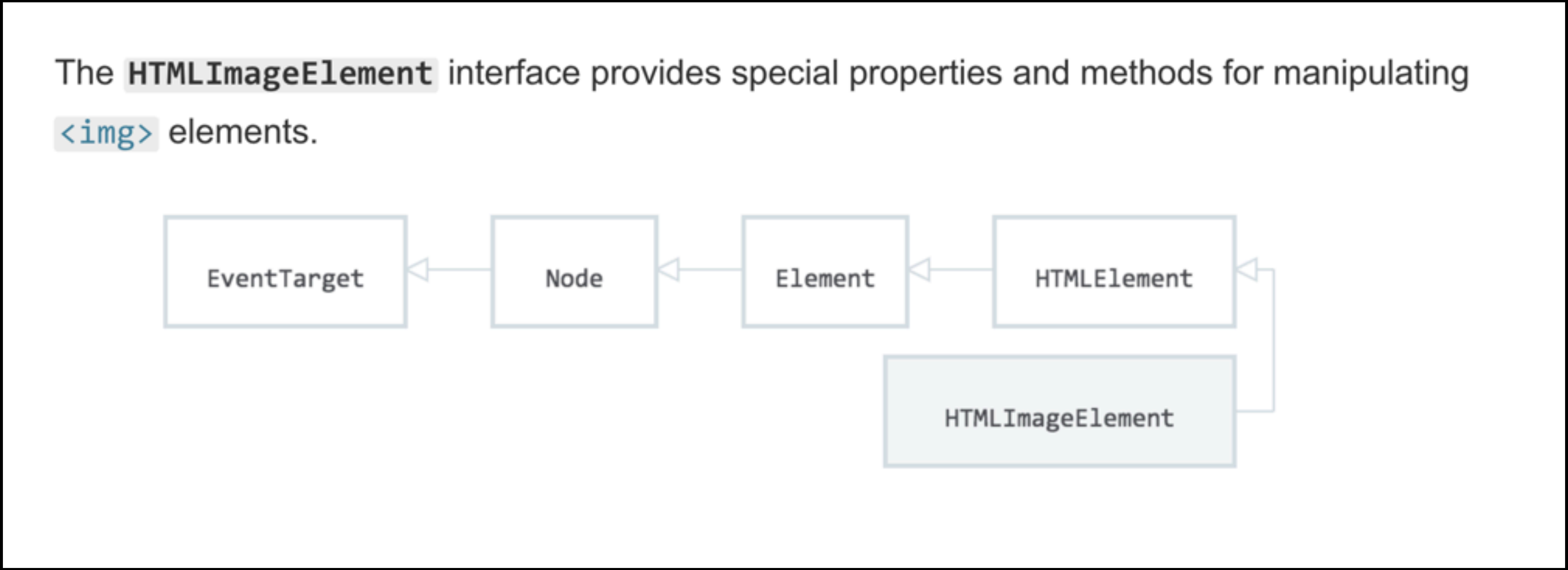
HTML

```
<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.

$$\begin{tikzcd}
\text{EventTarget} & \text{Node} \arrow[l] & \text{Element} \arrow[l] & \text{HTMLElement} \arrow[l] \\
 & & & \text{HTMLImageElement} \arrow[u]
\end{tikzcd}$$

Add elements via DOM

We can create elements dynamically and add them to the web page via createElement and appendChild:

```
document.createElement( HTML tag string )
element.appendChild( another element );
```

Technically you can also add elements to the webpage via **innerHTML**, but it poses a security risk.

```
// Try not to use innerHTML like this:
element.innerHTML = '<h1>Hooray!</h1>';
```

Remove elements via DOM

We can also call remove elements from the DOM by calling
the **remove( )** method on the DOM object:

```
element.remove( );
```

And actually setting the innerHTML of an element to an empty string is a fine way of removing all children from a parent node:

```
// This is fine and poses no security risk.
    element.innerHTML = '';
```

display: none;

There is yet another super helpful value for display:

```
display: block;
display: inline;
display: inline-block;
display: flex;
display: none;
```

display: none; turns off rendering for the element and all its children. It's treated as if the element were not in the document at all..

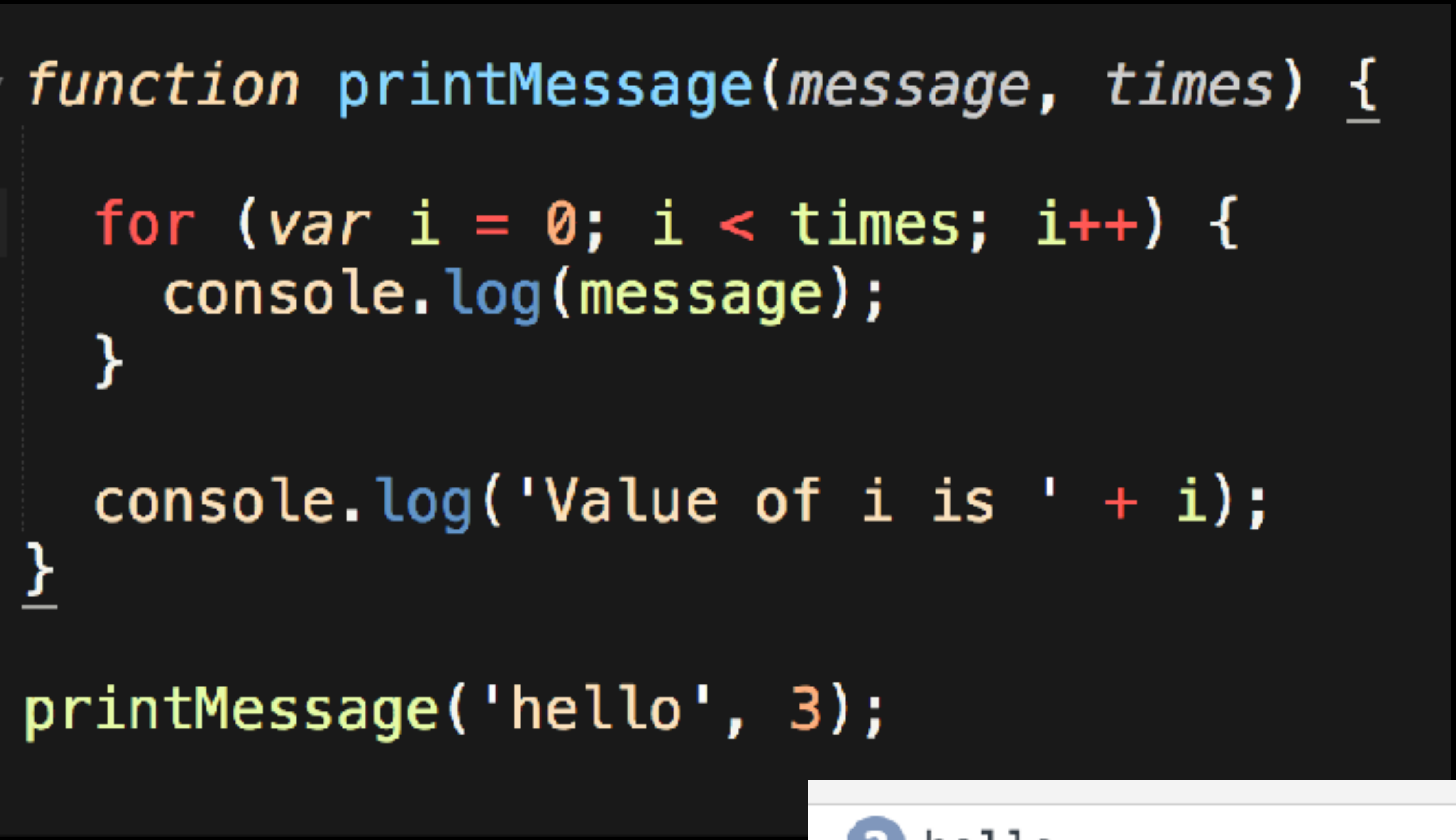
Several Strategies for updating HTML elements in JS:

1. Change content of existing HTML elements in page:
    - Good for simple text updates

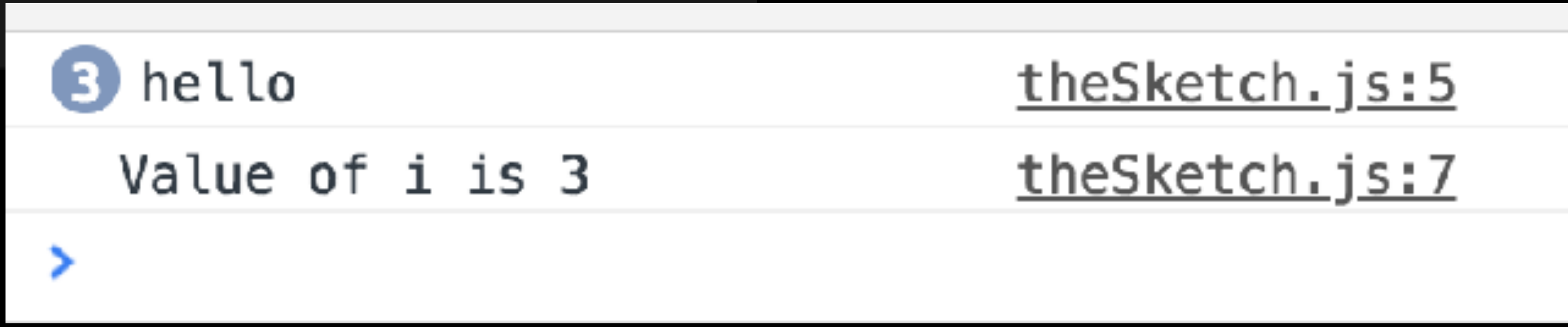
2. Add elements via createElement and appendChild - Needed if you're adding a variable number of elements

3. Put all "views" in the HTML but set inactive ones to hidden, then update display state as necessary.
    - Good when you know ahead of time what element(s) you want to display
    - Can be used in conjunction with (1) and/or (2)

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);
```

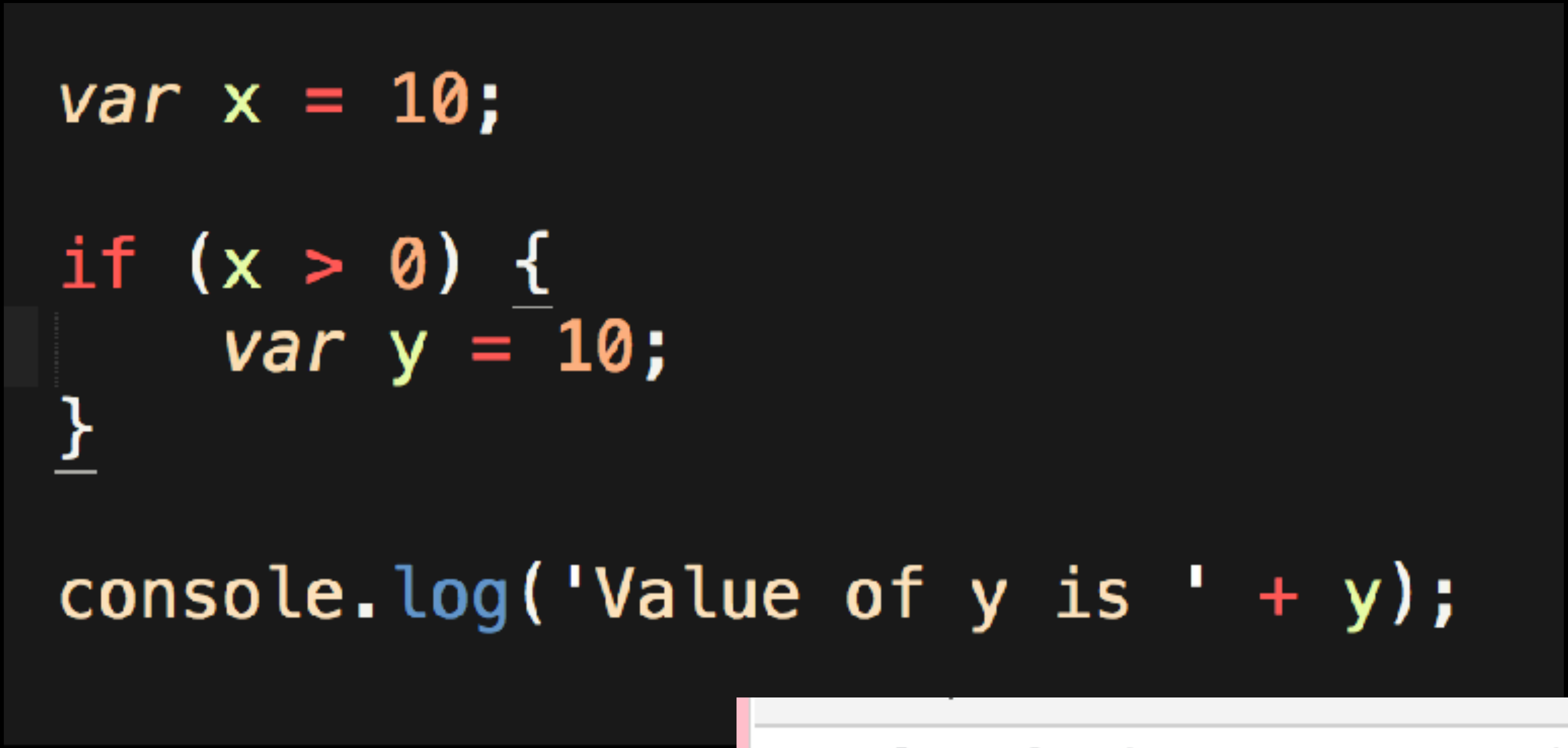


```
3 hello                              theSketch.js:5
  Value of i is 3                    theSketch.js:7
>
```

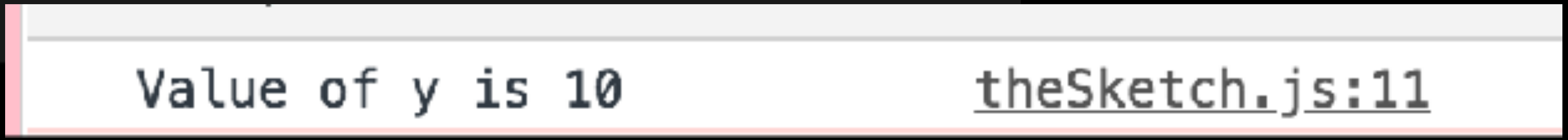
var

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);
```

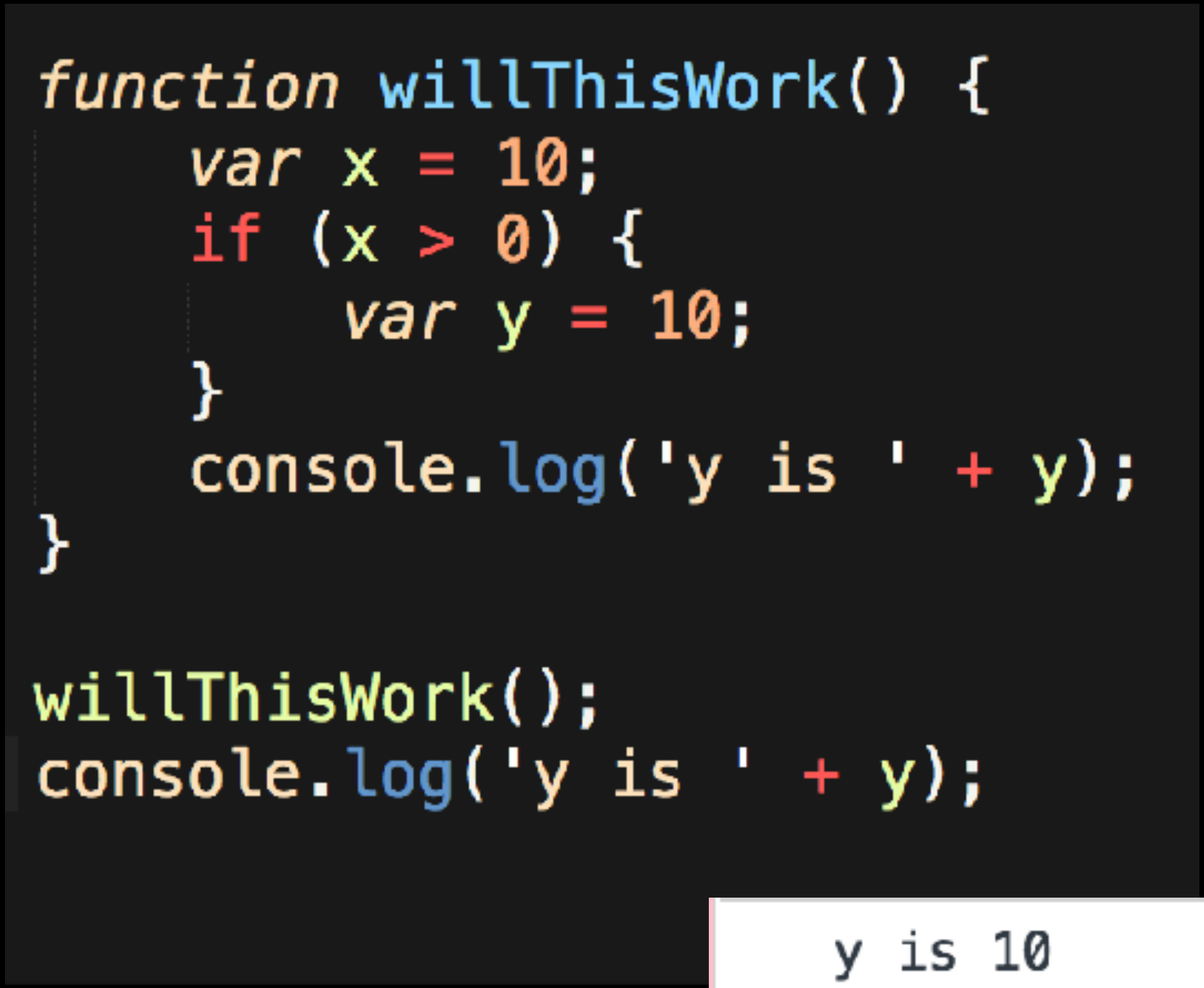


```
Value of y is 10                theSketch.js:11
```

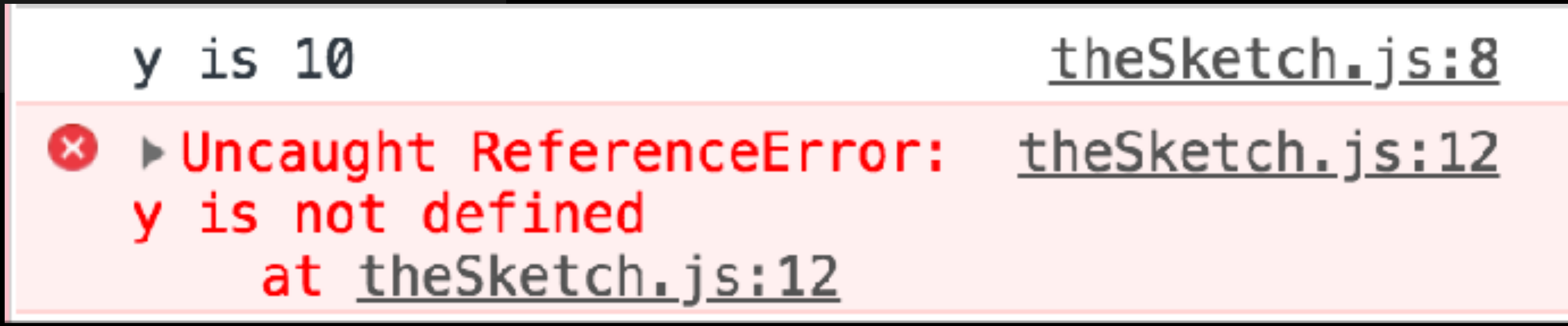
Variables declared with var have function-level scope and do not go out of scope at the end of blocks; only at the end of functions

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



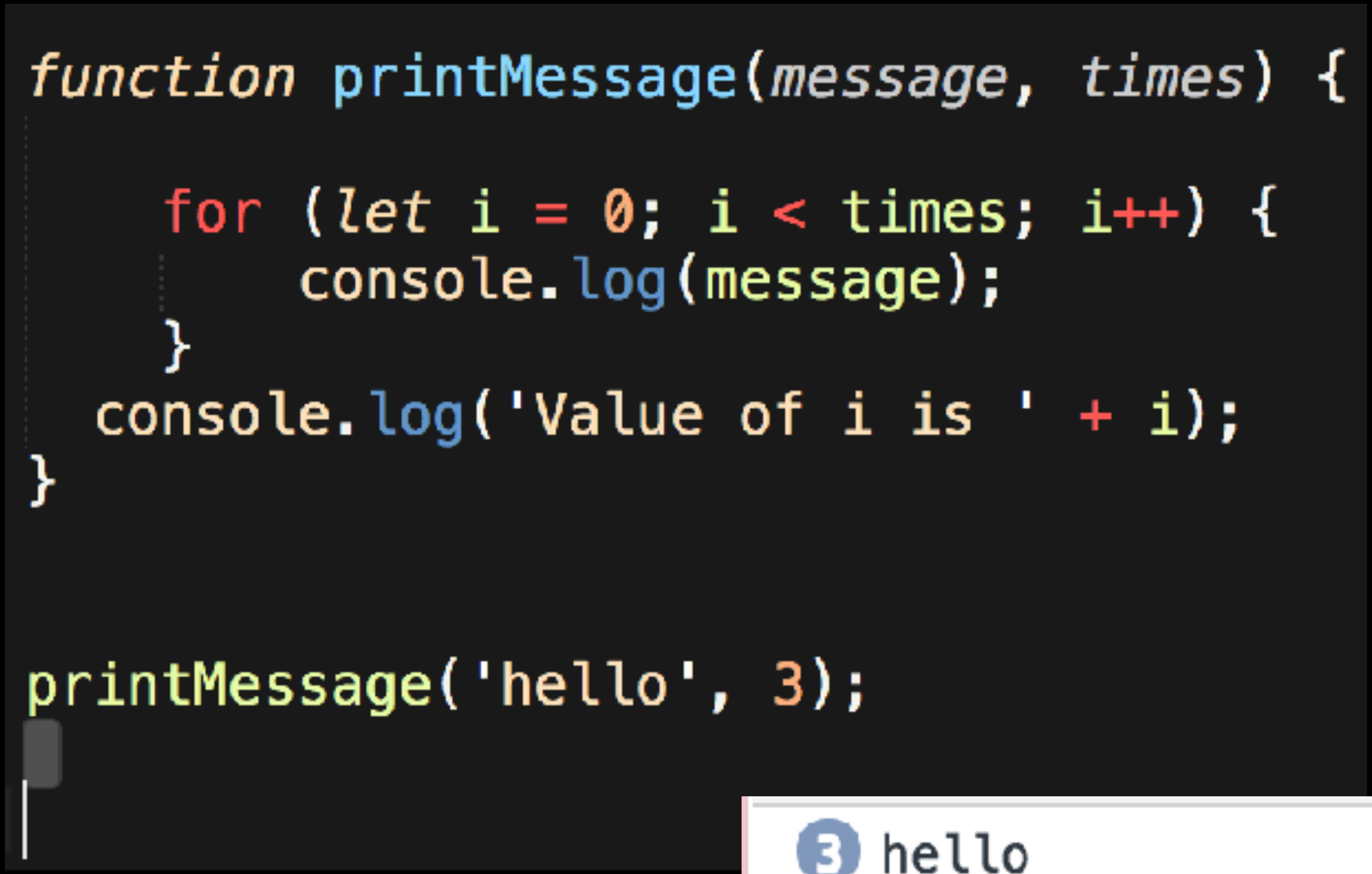
```
function willThisWork() {
    var x = 10;
    if (x > 0) {
        var y = 10;
    }
    console.log('y is ' + y);
}

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



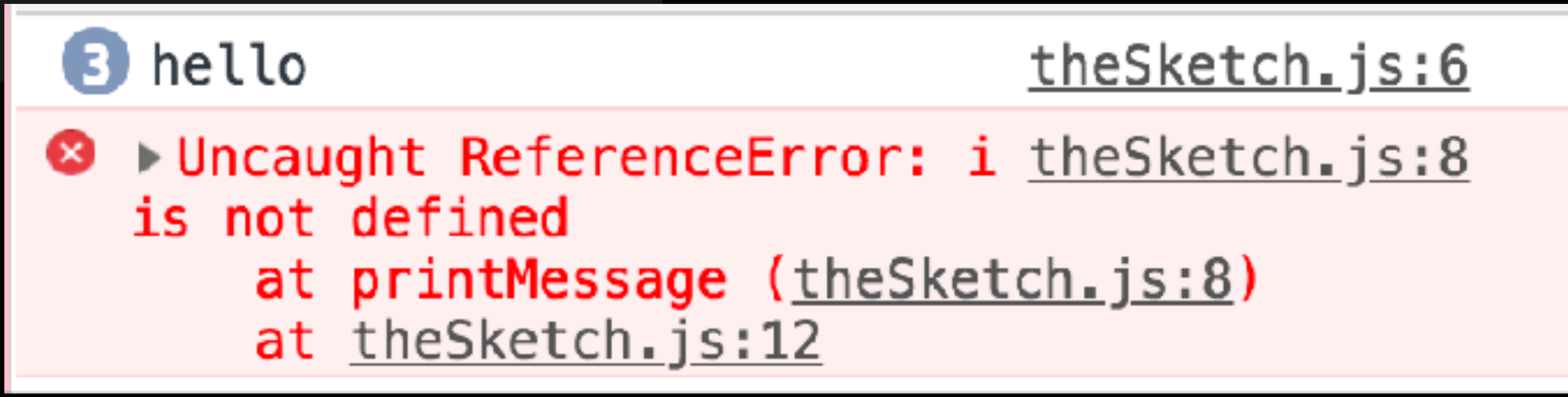
```
y is 10                                          theSketch.js:8
▸Uncaught ReferenceError:                        theSketch.js:12
y is not defined
    at theSketch.js:12
```

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

scope w/ let



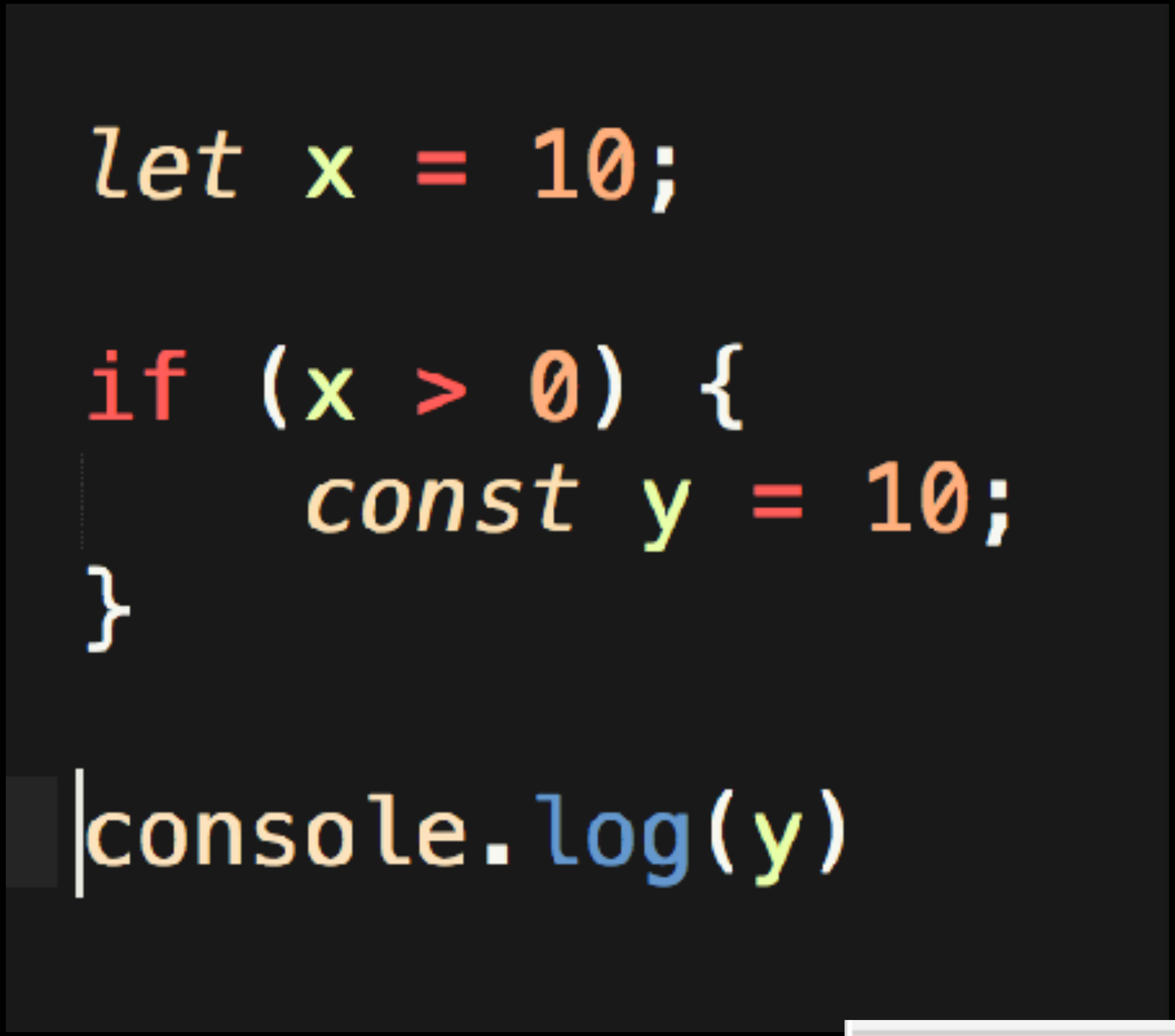
```
function printMessage(message, times) {

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


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



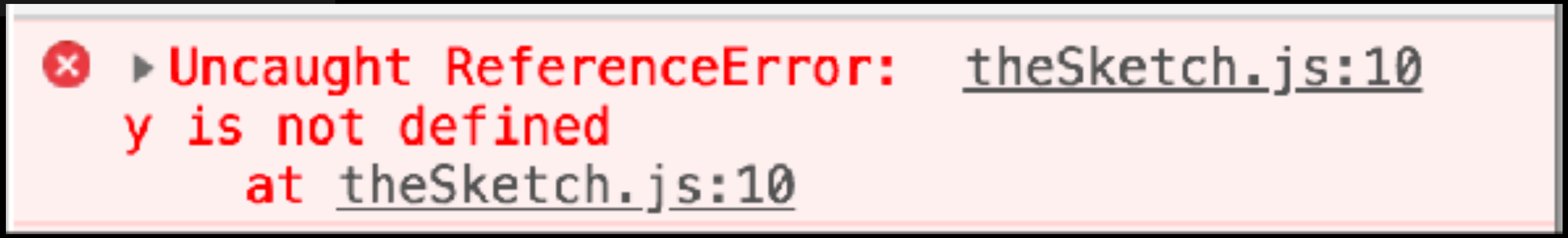
```
3 hello                                  theSketch.js:6
▶Uncaught ReferenceError: i             theSketch.js:8
is not defined
    at printMessage (theSketch.js:8)
    at theSketch.js:12
```

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)
```

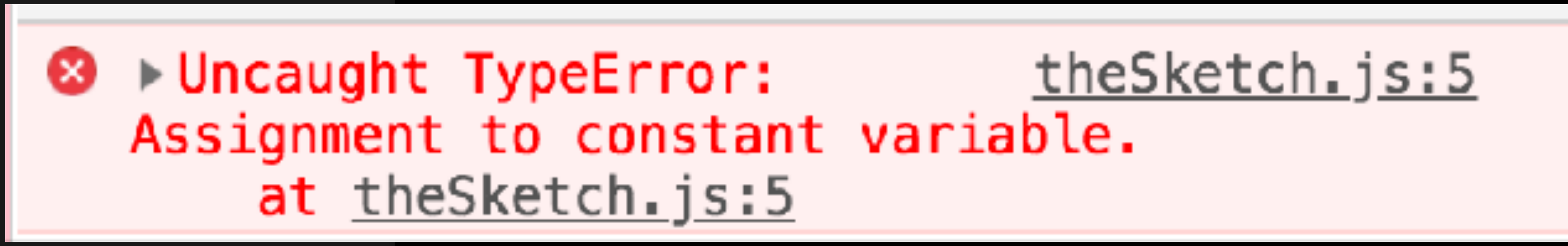


```
Uncaught ReferenceError:   theSketch.js:10
y is not defined
    at theSketch.js:10
```

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

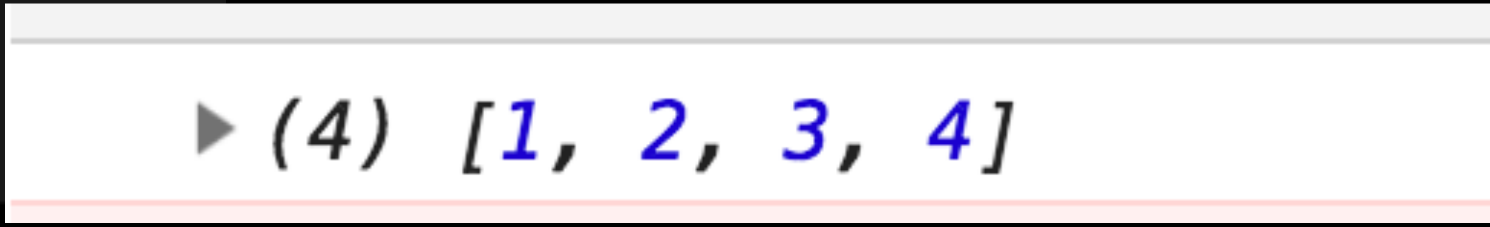
const

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



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

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

console.log(myList);
```



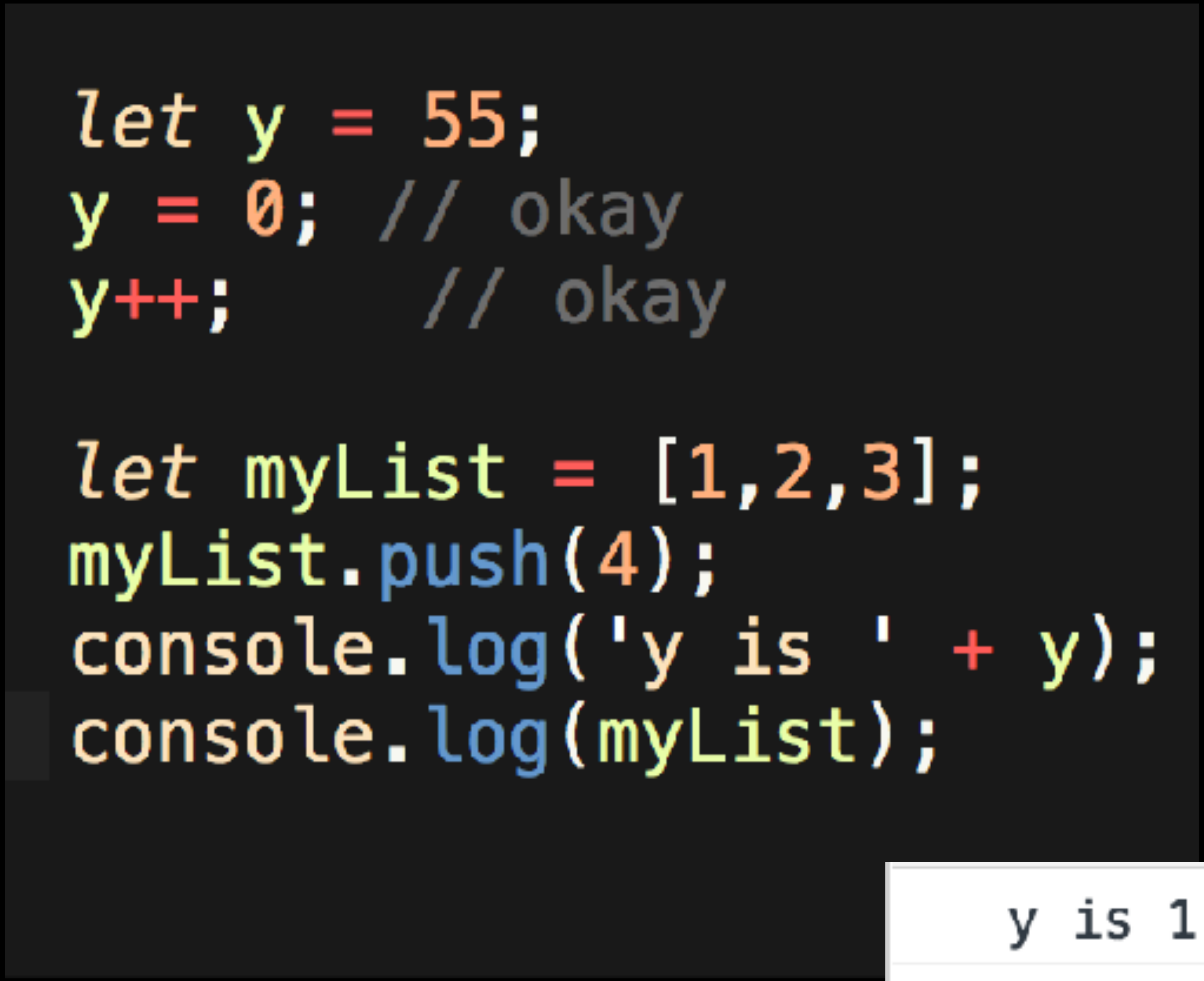
```
▸(4) [1, 2, 3, 4]
```

const declared variables cannot be reassigned.

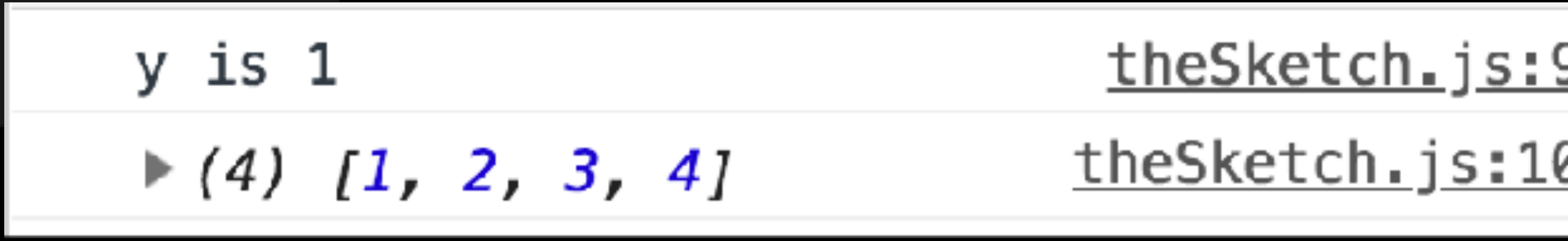
However, it doesn't provide true const correctness, so you can still modify the underlying object

- (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")

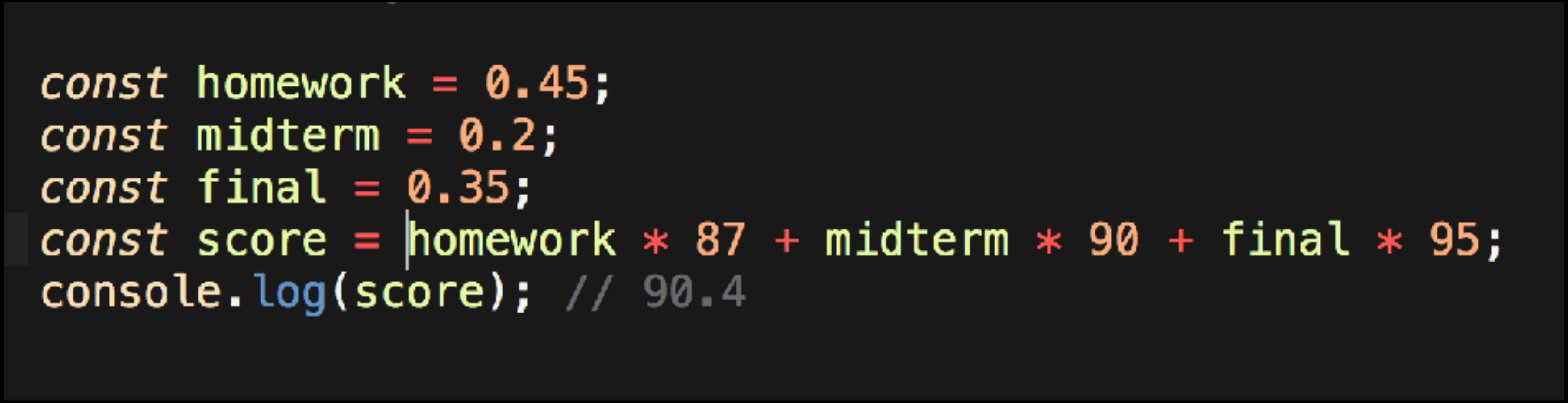
JS Variables do not have types, but the values do.

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 primitiv type), etc.

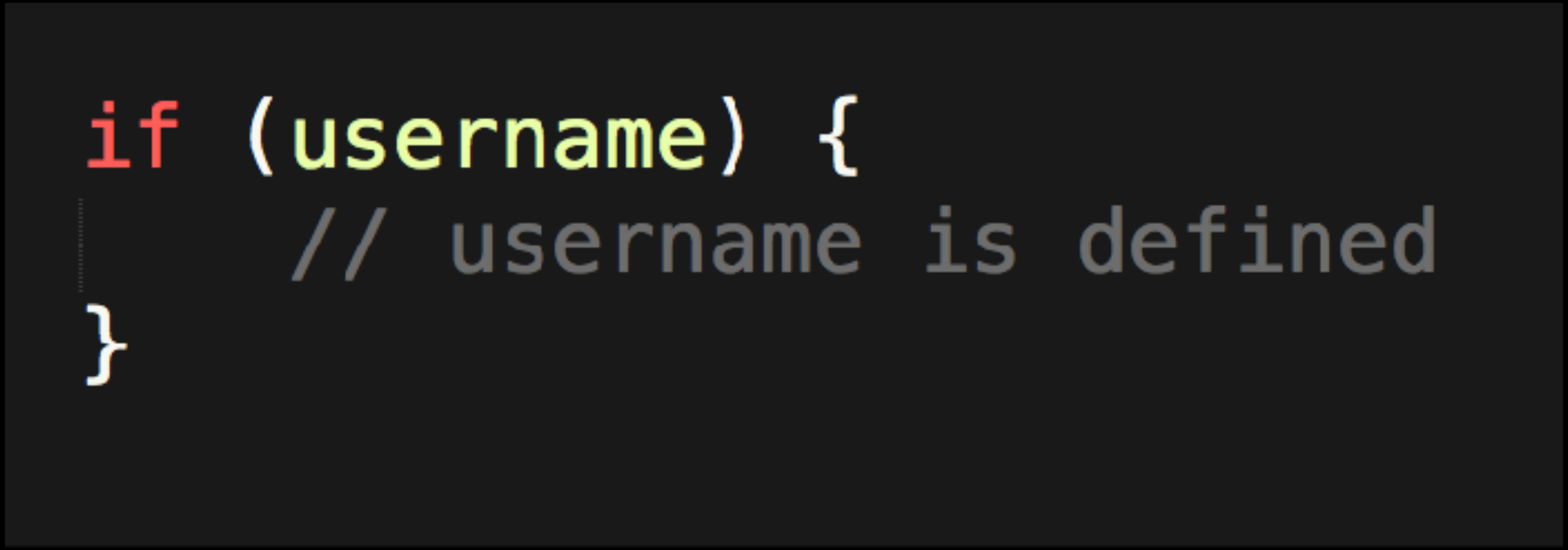
Data Type: Numbers



```javascript
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)

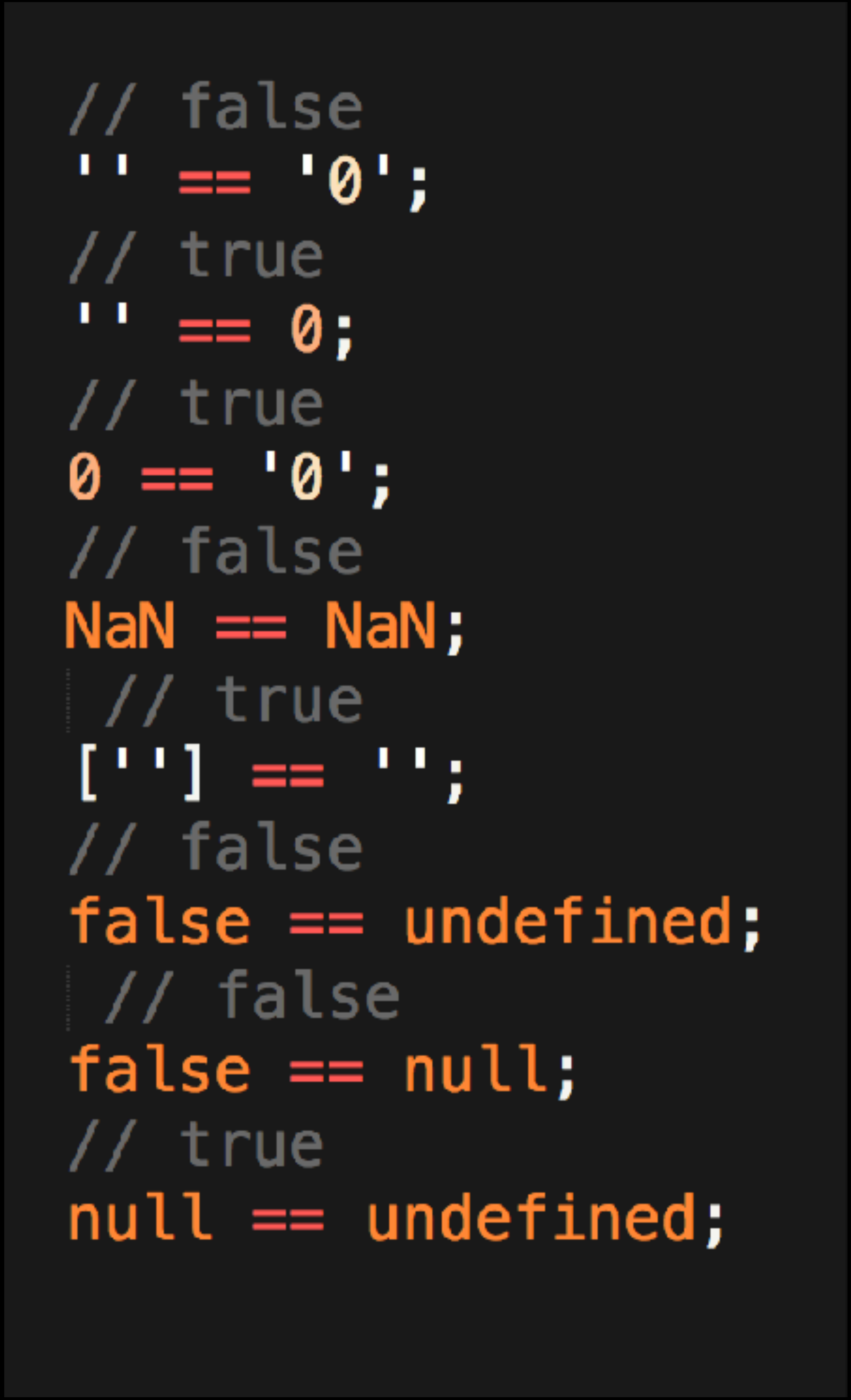
Compares two values to check that both the data and value are the same

!== (strict not equal to)

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

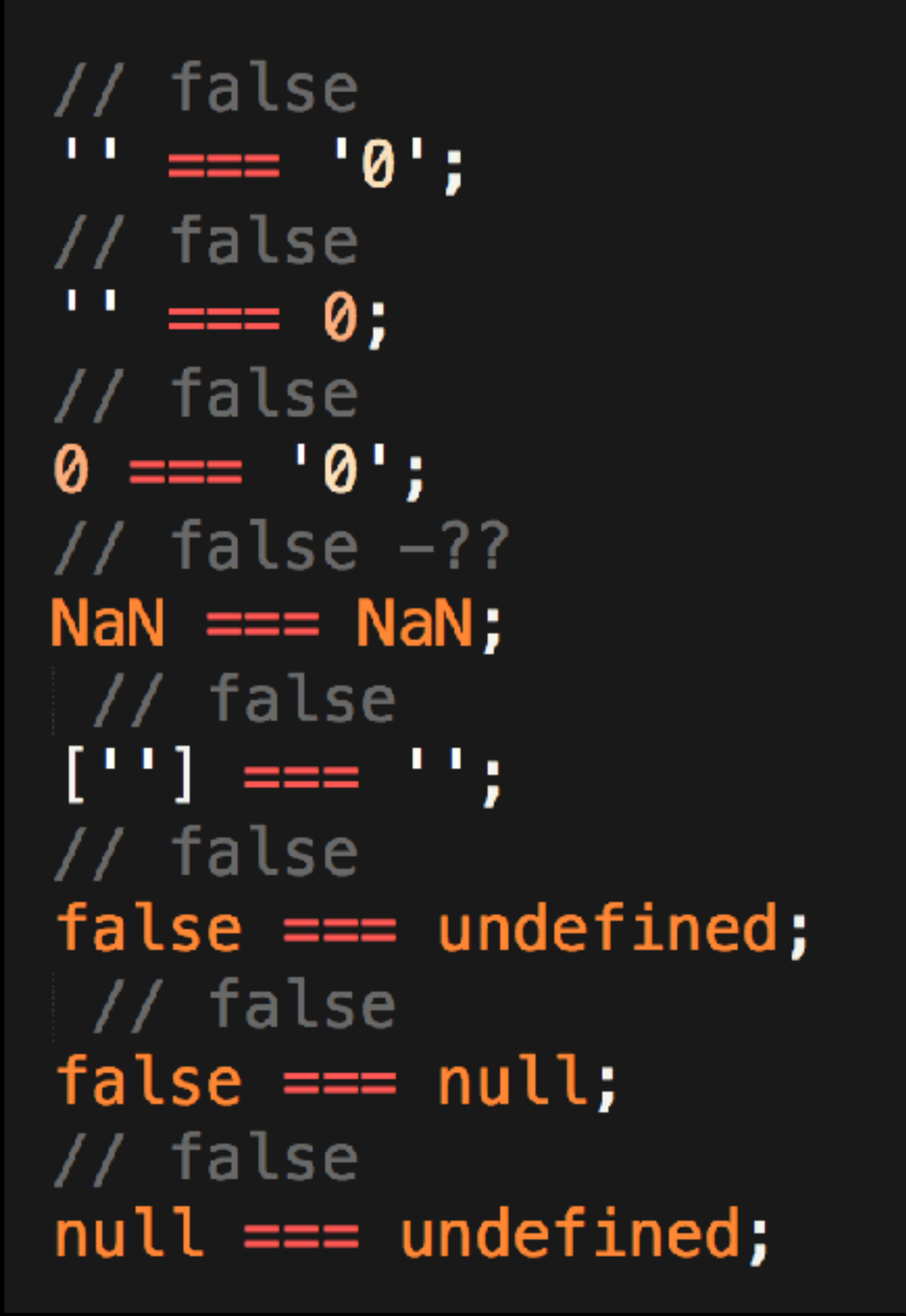
Equality

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



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

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

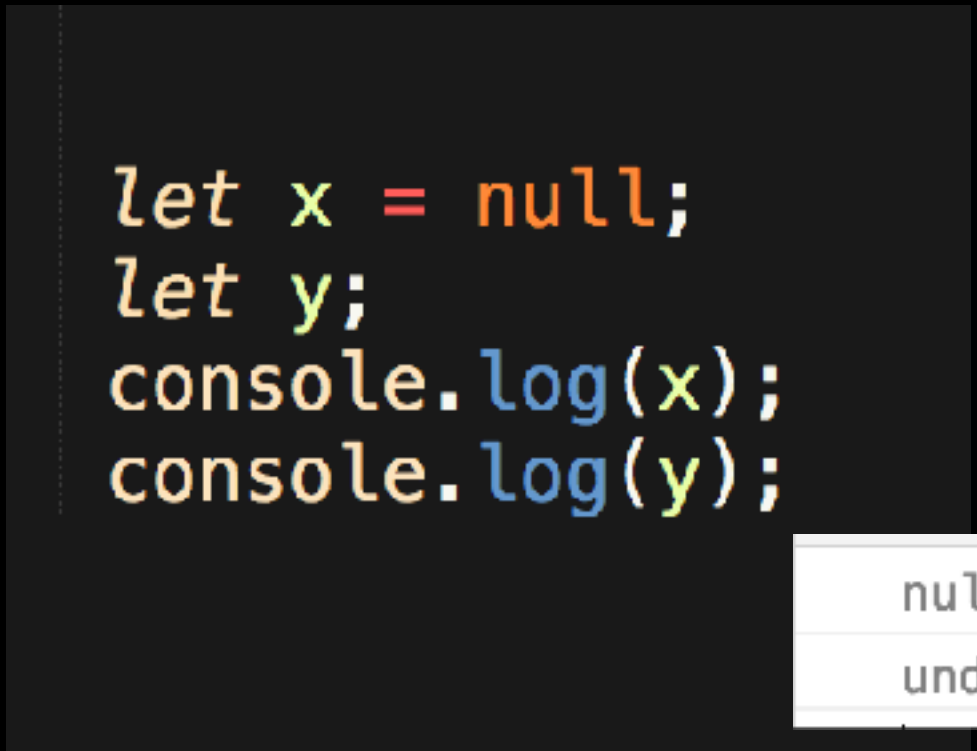
Always use === and !== and don't use == or !=

Null + Undefined

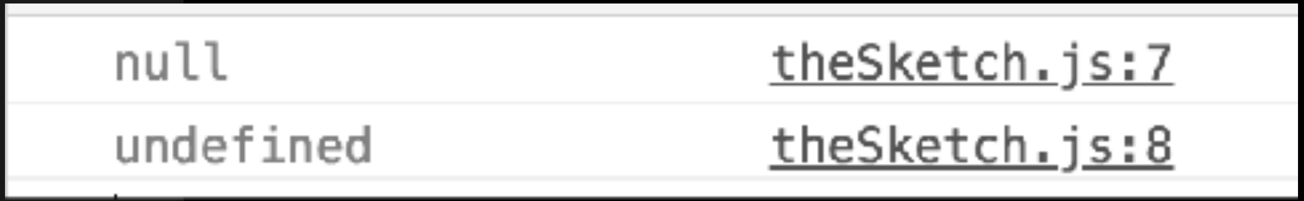
What's the difference?

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
```

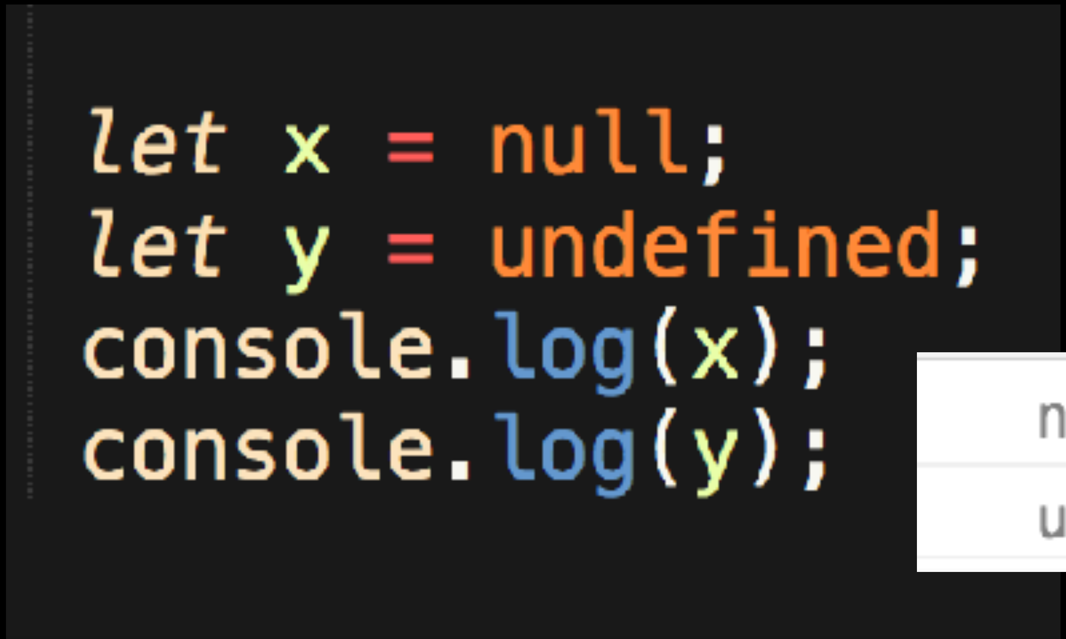
Null + Undefined

What's the difference?

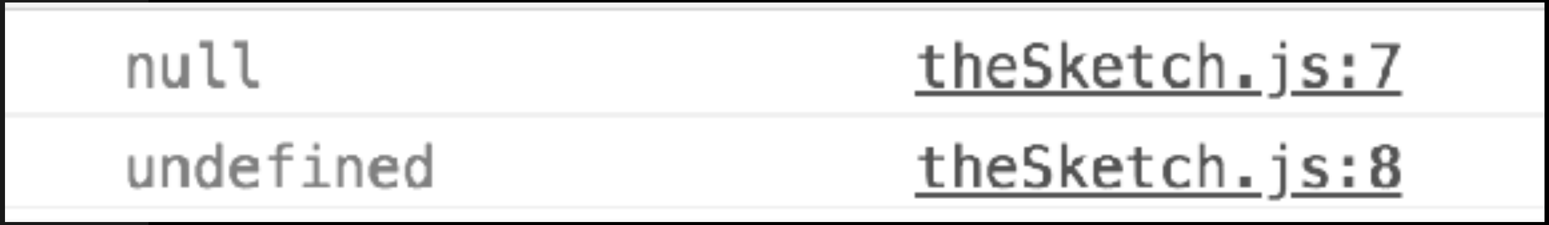
null is a value representing the absence of a value, similar to null in Java and nullptr in C++.

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
```

Object properties + methods

In an object:

1. **Variables** become known as **properties**
2. **Functions** become known as **methods**

Properties tell us about the object, such as the height of a chair and how many chairs there are

Methods represent tasks that are associated with the object, e.g. count the height of all chairs by adding all heights together

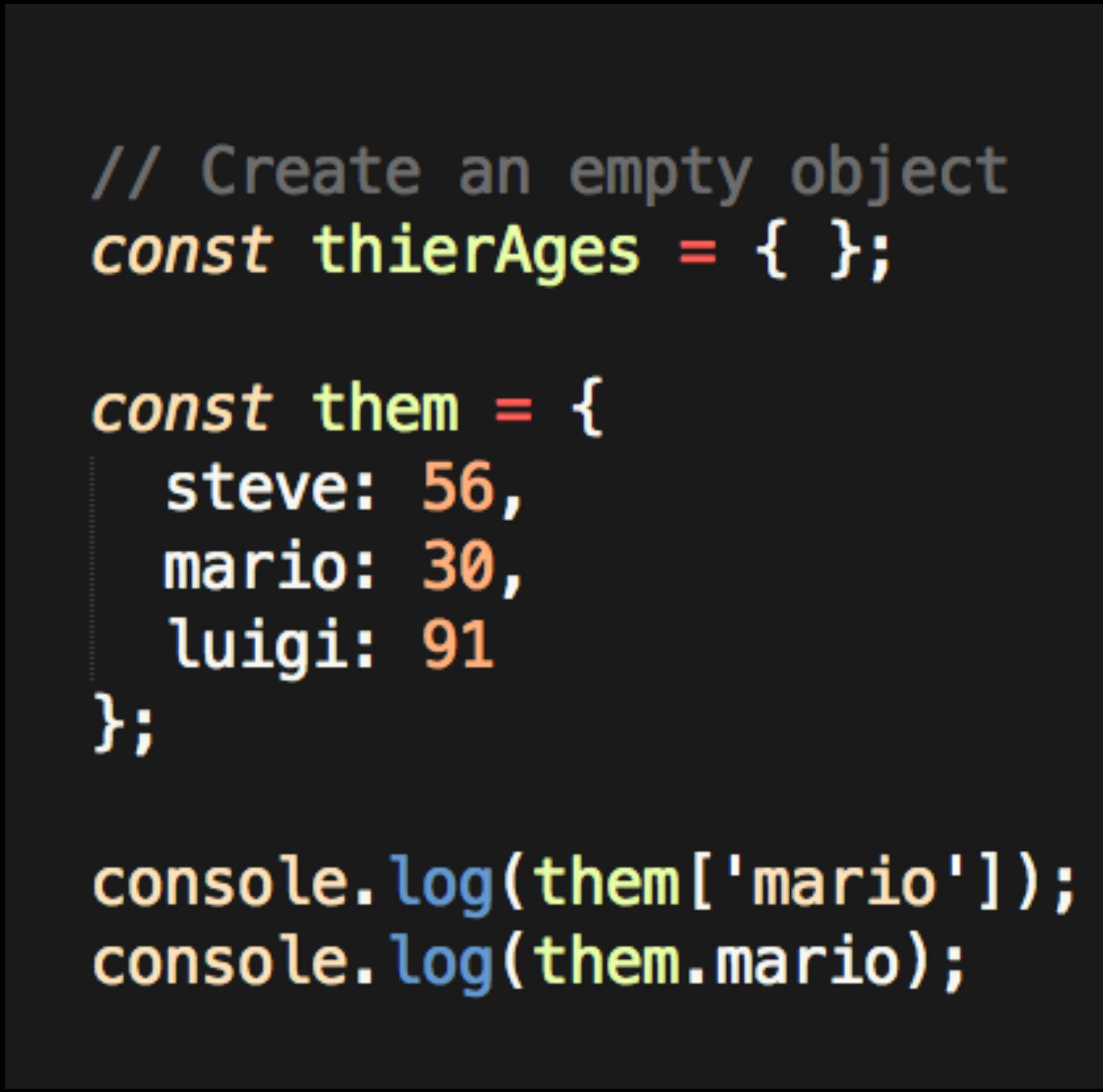
Maps through objects

There are two ways to access the value of a property:

1. objectName[property]
2. objectName.property

(2 only works for string keys.)

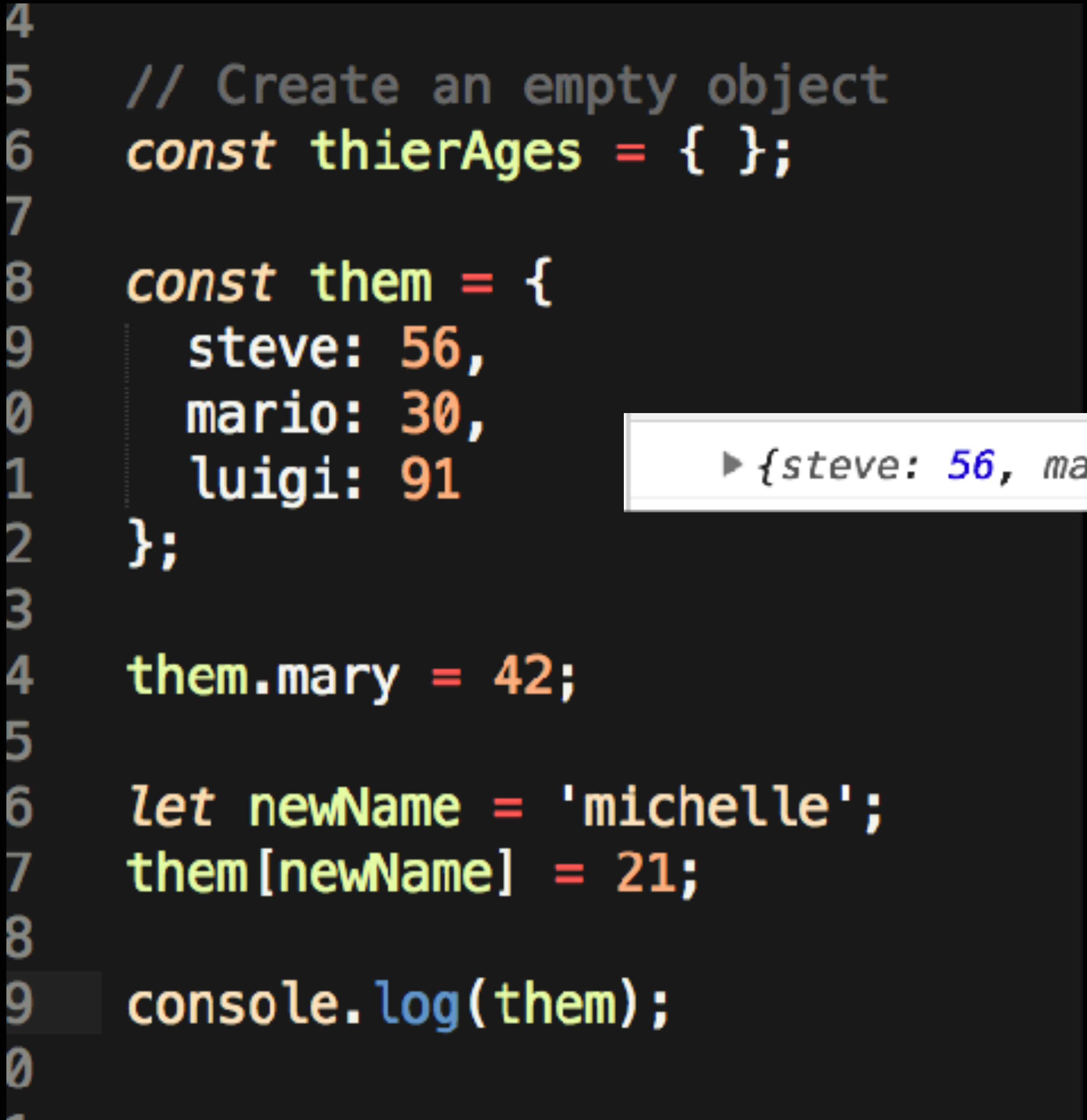
Generally prefer style (2), unless the property is stored in a variable, or if the property is not a string.



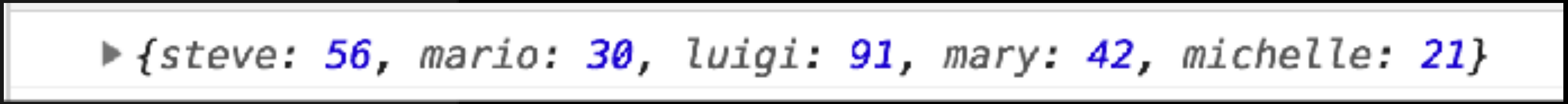
```
// Create an empty object
const thierAges = { };

const them = {
  steve: 56,
  mario: 30,
  luigi: 91
};

console.log(them['mario']);
console.log(them.mario);
```

Maps through objects

To add a property to an object, name the property and give it a value:



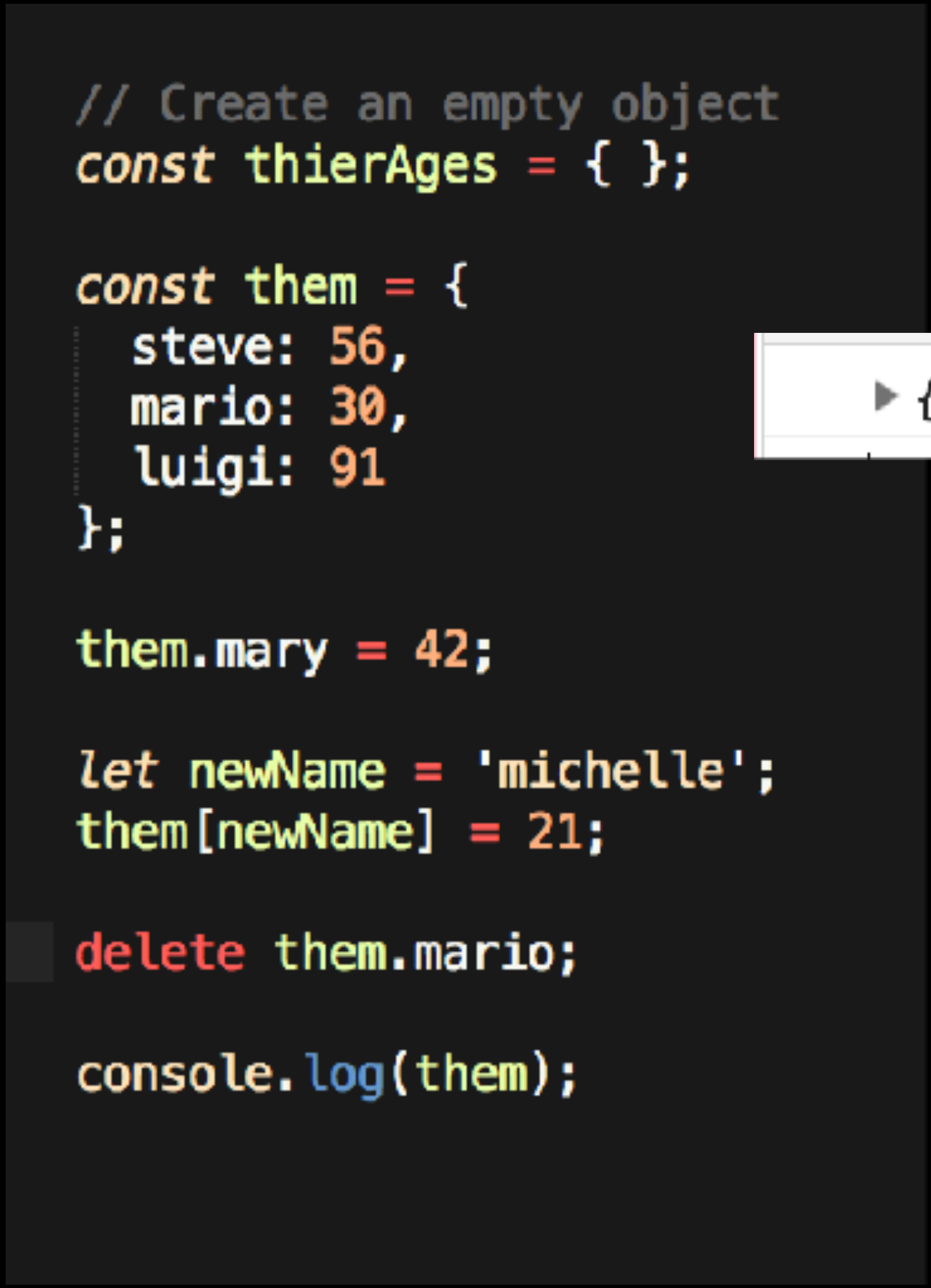
```
// Create an empty object
const thierAges = { };

const them = {
  steve: 56,
  mario: 30,
  luigi: 91
};

them.mary = 42;

let newName = 'michelle';
them[newName] = 21;

console.log(them);
```



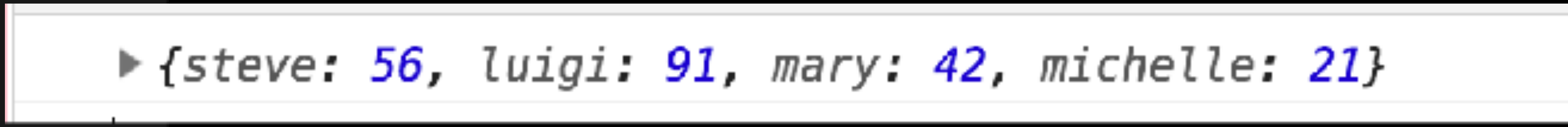
```
▶{steve: 56, mario: 30, luigi: 91, mary: 42, michelle: 21}
```

Maps through objects

To remove a property to an object, use **delete**:

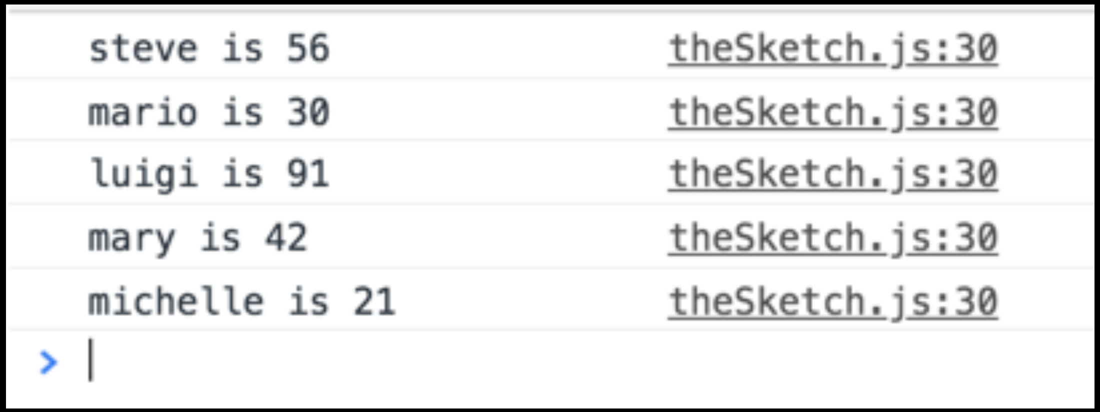


```
// Create an empty object
const thierAges = { };

const them = {
  steve: 56,
  mario: 30,
  luigi: 91
};

them.mary = 42;

let newName = 'michelle';
them[newName] = 21;

delete them.mario;

console.log(them);
```

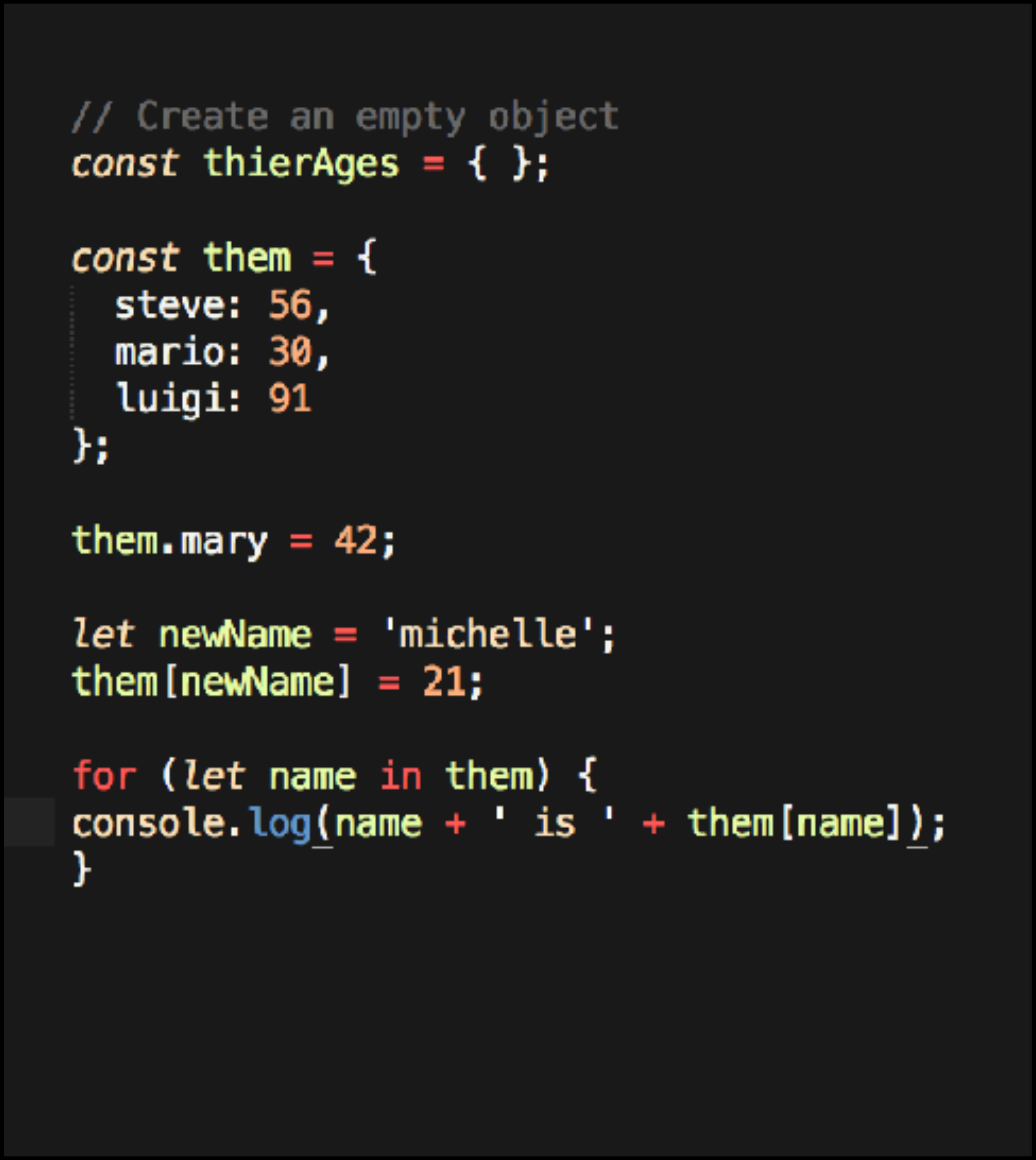


```
▶ {steve: 56, luigi: 91, mary: 42, michelle: 21}
```

Iterating through Map



```
steve is 56          theSketch.js:30
mario is 30          theSketch.js:30
luigi is 91          theSketch.js:30
mary is 42           theSketch.js:30
michelle is 21       theSketch.js:30
> |
```



```
// Create an empty object
const thierAges = { };

const them = {
  steve: 56,
  mario: 30,
  luigi: 91
};

them.mary = 42;

let newName = 'michelle';
them[newName] = 21;

for (let name in them) {
console.log(name + ' is ' + them[name]);
}
```

- You can't use **for...in** on lists; only on **object types**
- You can't use **for...of** on objects; only on **list types**