Reminders

- **Homework 1** (not optional) is at **weblab.to/homework1** → please complete it ASAP as you will need certain installations for Wednesday's workshops!
- **Milestone 0** (not optional) can be found at **weblab.to/milestone0** and is due tomorrow, Wednesday 11:59 PM :)
  - If you still need teammates, please check out the **"Search for teammates"** post on piazza!
- **Lunch reminders:**
  - We are serving food to in-person students, but under the agreement that you **eat *outdoors***. No one should be unmasked/eating inside the lecture hall at any point during the day!
  - When you're eating outside, please try to **spread out**, away from other groups. It is important that we be safe about eating, else there's a chance we may get shut down which will be very BAD.

Reminders

- weblab.to/questions
- weblab.to/bukabuka

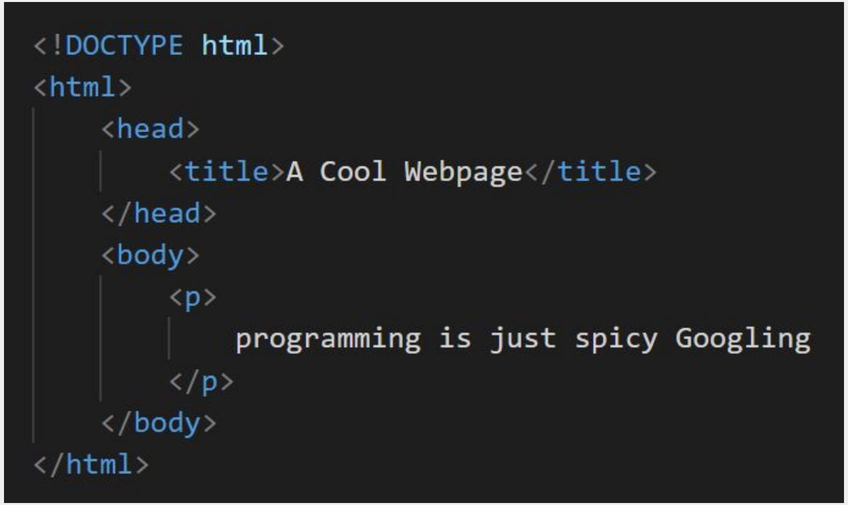
JavaScript

Albert Xing

Recap

HTML

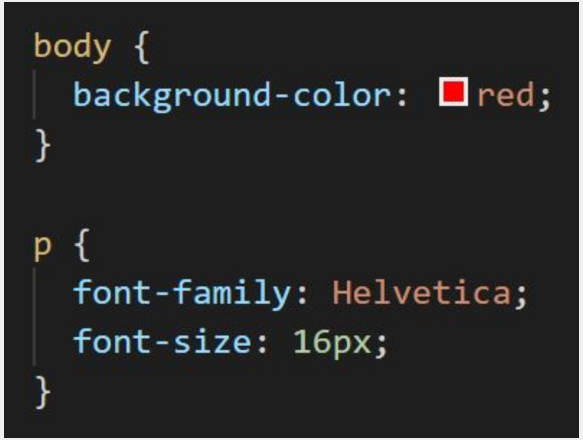
- Describes **content** and **structure**
- What exists? How is it organized?



```
<!DOCTYPE html>
<html>
    <head>
        <title>A Cool Webpage</title>
    </head>
    <body>
        <p>
            programming is just spicy Googling
        </p>
    </body>
</html>
```

CSS

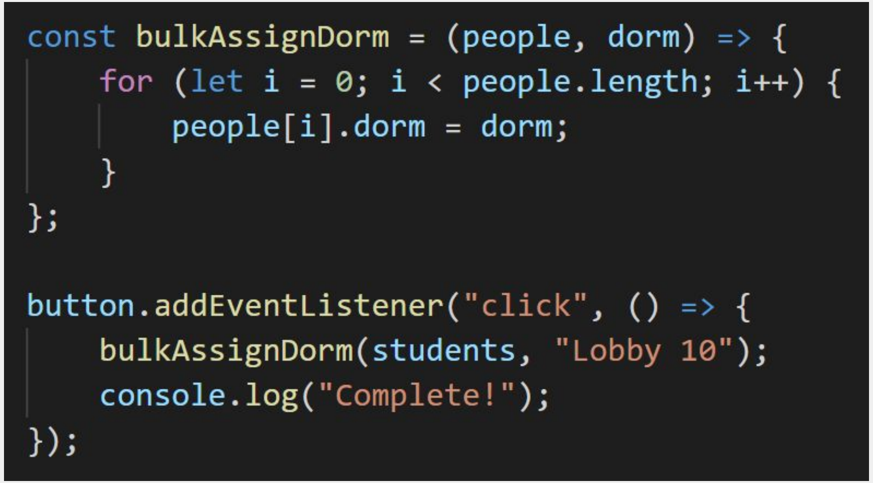
- Describes the **presentation**
- Colors! Fonts! Alignment, margins, borders, shading, and more



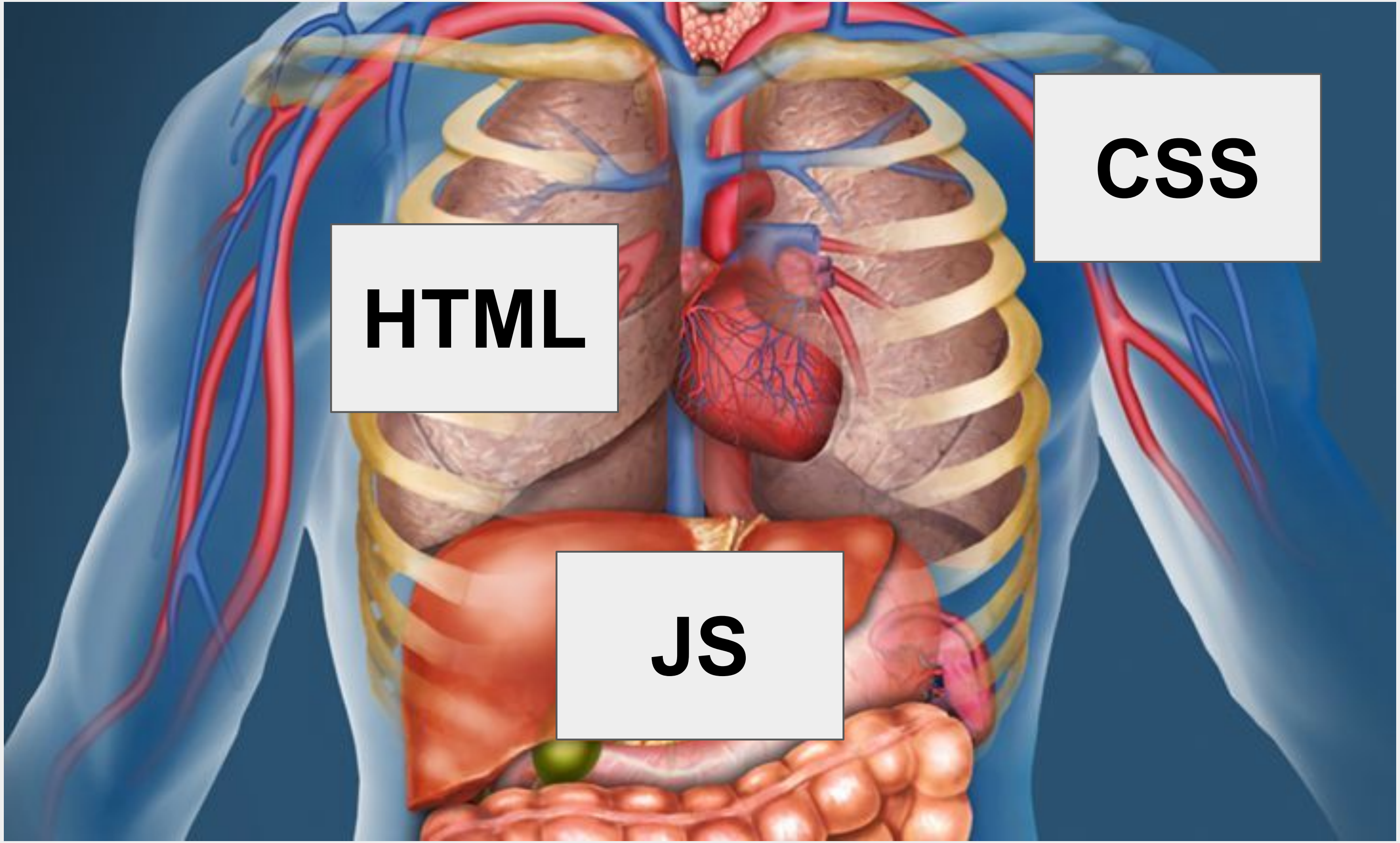
```
body {
  background-color: 🟥red;
}

p {
  font-family: Helvetica;
  font-size: 16px;
}
```

JavaScript is…

- … a programming language that **manipulates** the content of a web page
- … how we take HTML + CSS and make it **interactive**!
- … used by a vast majority of websites and web applications
- … not related to Java 🙃



```
const bulkAssignDorm = (people, dorm) => {
    for (let i = 0; i < people.length; i++) {
        people[i].dorm = dorm;
    }
};

button.addEventListener("click", () => {
    bulkAssignDorm(students, "Lobby 10");
    console.log("Complete!");
});
```


CSS
HTML
JS

Where does it go?

Where can we run JavaScript code?

1. The browser console

   Chrome: Ctrl + Shift + J (on Windows) / Cmd + Option + J (on Mac)

   Firefox: Ctrl + Shift + J (on Windows) / Cmd + Shift + J (on Mac)

2. Tied to our HTML file (more on that later!)

How to JavaScript

Types

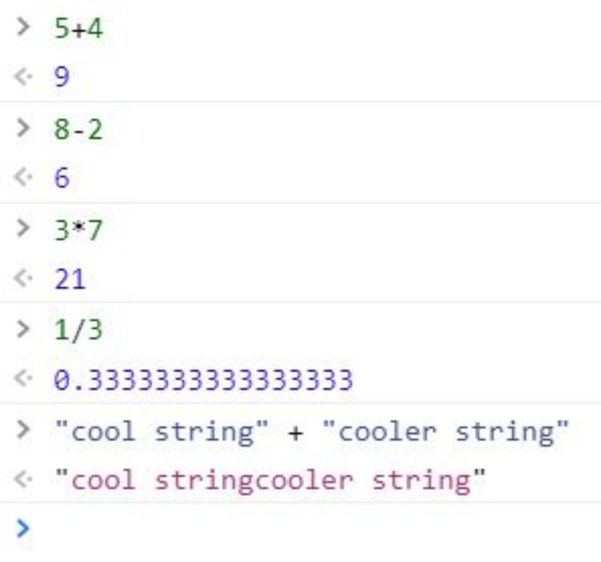
JavaScript has 5 primitive data types:

- Boolean (true, false)
- Number (12, 1.618, -46.7, 0, etc.)
- String (“hello”, “world!”, “12”, “”, etc.)
- Null
- Undefined

Operators

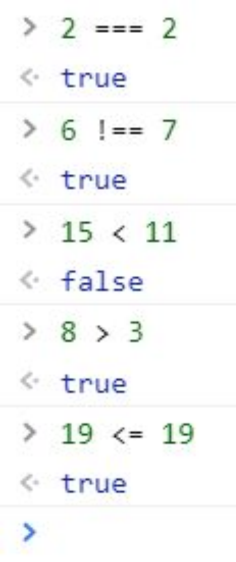
Things (mostly) work how you would expect:

(note the triple equals sign!)



```
> 5+4
< 9
> 8-2
< 6
> 3*7
< 21
> 1/3
< 0.3333333333333333
> "cool string" + "cooler string"
< "cool stringcooler string"
>
```

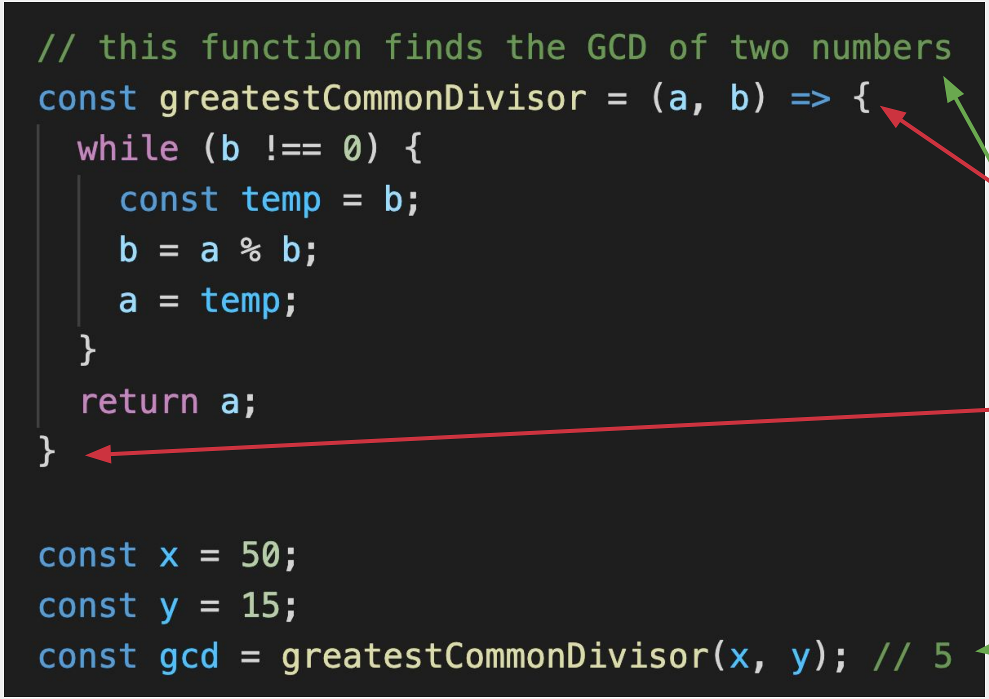
arithmetic operators



```
> 2 === 2
< true
> 6 !== 7
< true
> 15 < 11
< false
> 8 > 3
< true
> 19 <= 19
< true
>
```

comparison operators

Syntax



```
// this function finds the GCD of two numbers
const greatestCommonDivisor = (a, b) => {
  while (b !== 0) {
    const temp = b;
    b = a % b;
    a = temp;
  }
  return a;
}

const x = 50;
const y = 15;
const gcd = greatestCommonDivisor(x, y); // 5
```

Every statement in JavaScript ends with a semicolon;

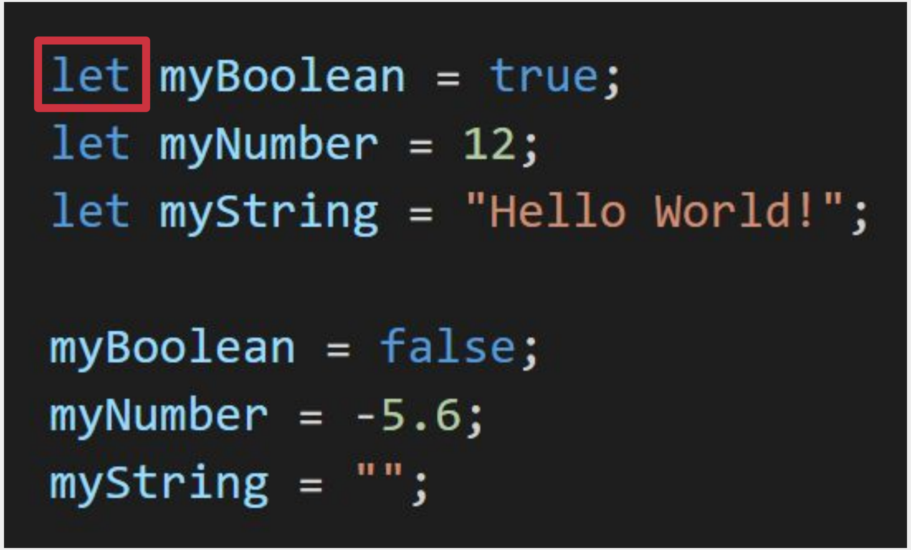
Whitespace is ignored. (but can improve readability)

Curly braces denote where **blocks** begin and end.

These are **comments**. It doesn't affect how the code runs, but you should use them to keep your codebase readable!

Defining variables

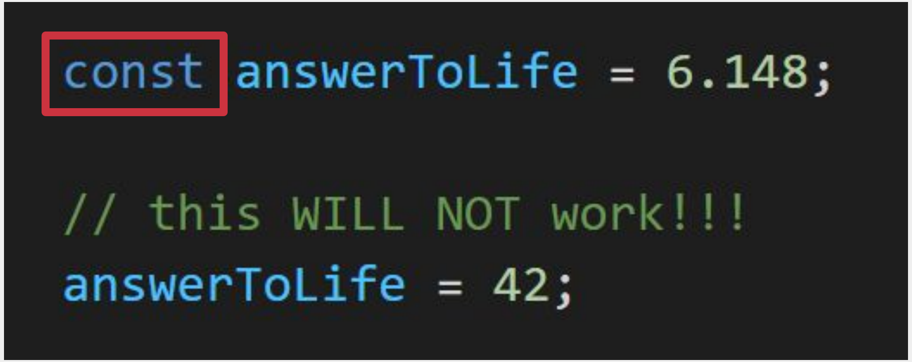
JavaScript convention is to name variables using `camelCase`.



```
let myBoolean = true;
let myNumber = 12;
let myString = "Hello World!";

myBoolean = false;
myNumber = -5.6;
myString = "";
```

Defining constants

To define a variable which *cannot* be re-assigned later:

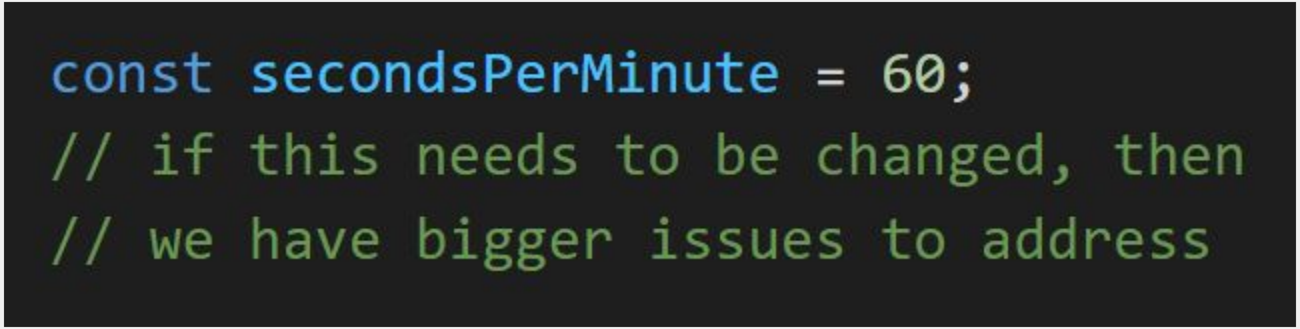


```
const answerToLife = 6.148;

// this WILL NOT work!!!
answerToLife = 42;
```

`let` vs. `const`

Why bother using `const` when `let` exists?

Safe code practices! If something should never be changed, don’t let it change :)

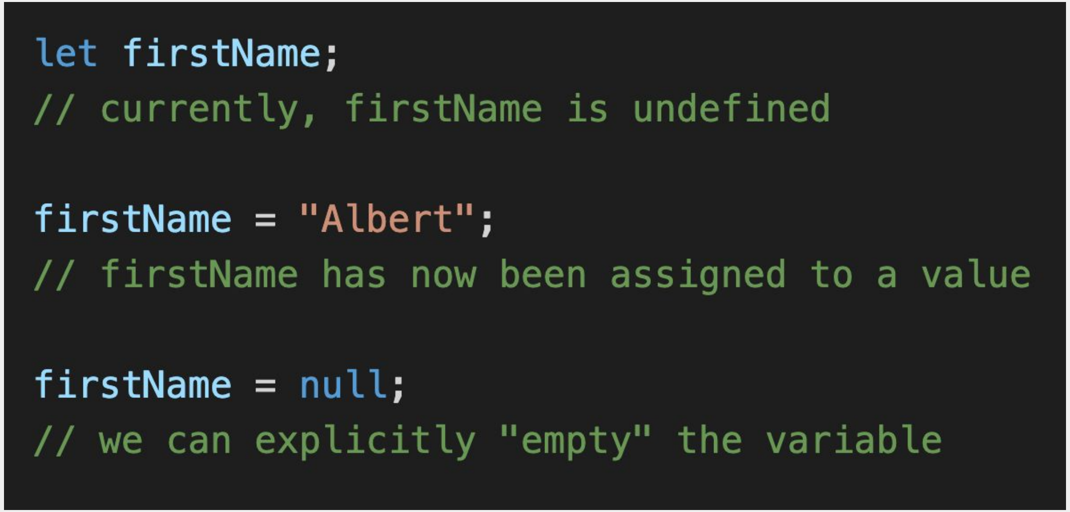


```
const secondsPerMinute = 60;
// if this needs to be changed, then
// we have bigger issues to address
```

null vs. undefined

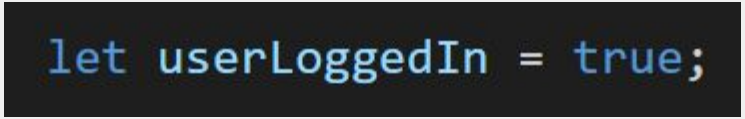
undefined means “declared but not yet assigned a value”

null means “no value”

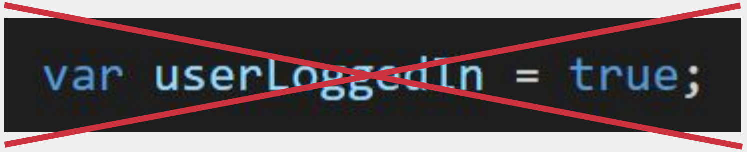


```
let firstName;
// currently, firstName is undefined

firstName = "Albert";
// firstName has now been assigned to a value

firstName = null;
// we can explicitly "empty" the variable
```

let vs. var



```
let userLoggedIn = true;
```



~~var userLoggedIn = true;~~

tl;dr please don't use var

technical details (Google it if you're interested):

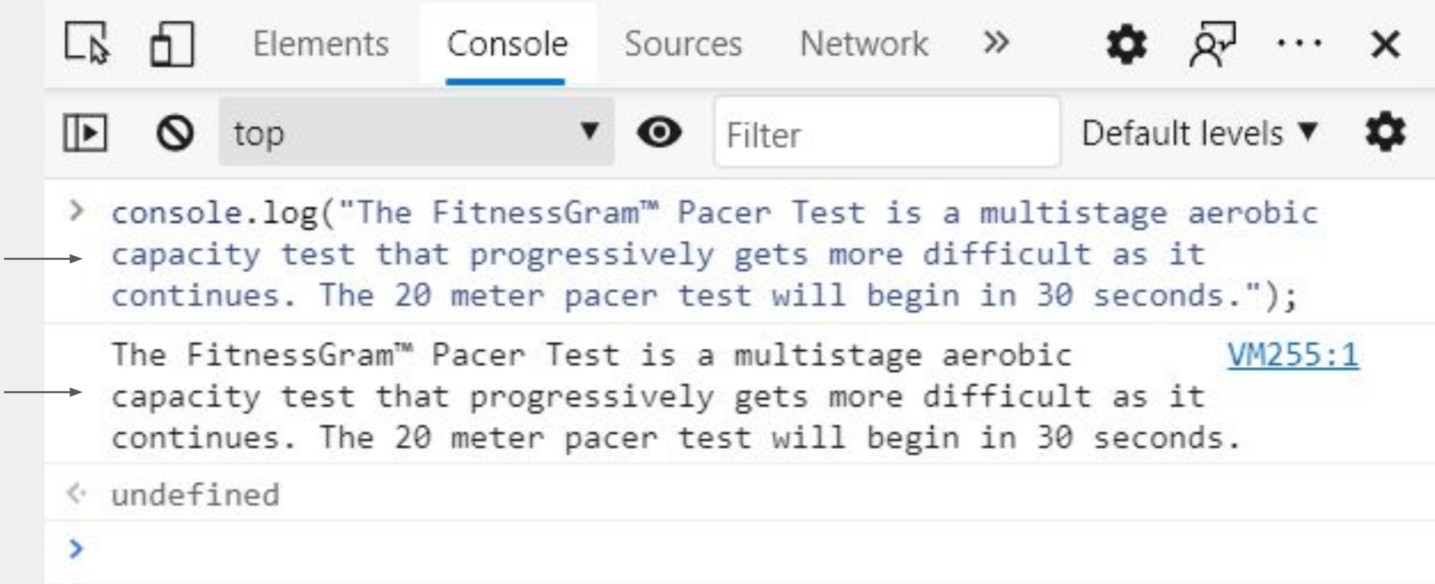
let is block-scoped

var is function-scoped

let exists because people kept getting bugs when trying to use var

Output

`console.log()` writes to the JavaScript console:



input

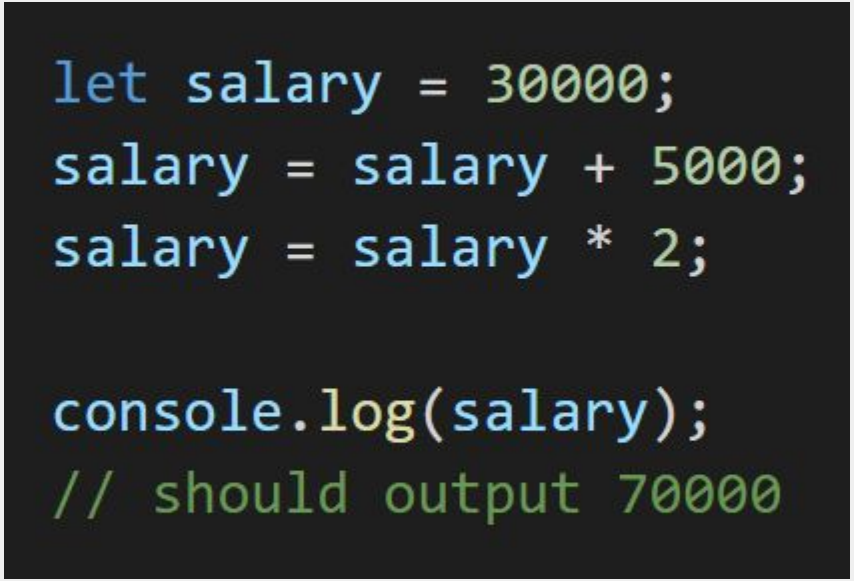
```
> console.log("The FitnessGram™ Pacer Test is a multistage aerobic
  capacity test that progressively gets more difficult as it
  continues. The 20 meter pacer test will begin in 30 seconds.");
```

output

```
  The FitnessGram™ Pacer Test is a multistage aerobic       VM255:1
  capacity test that progressively gets more difficult as it
  continues. The 20 meter pacer test will begin in 30 seconds.
< undefined
>
```

Output

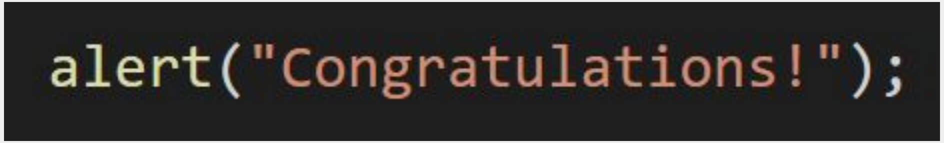
Handy for quick debugging!



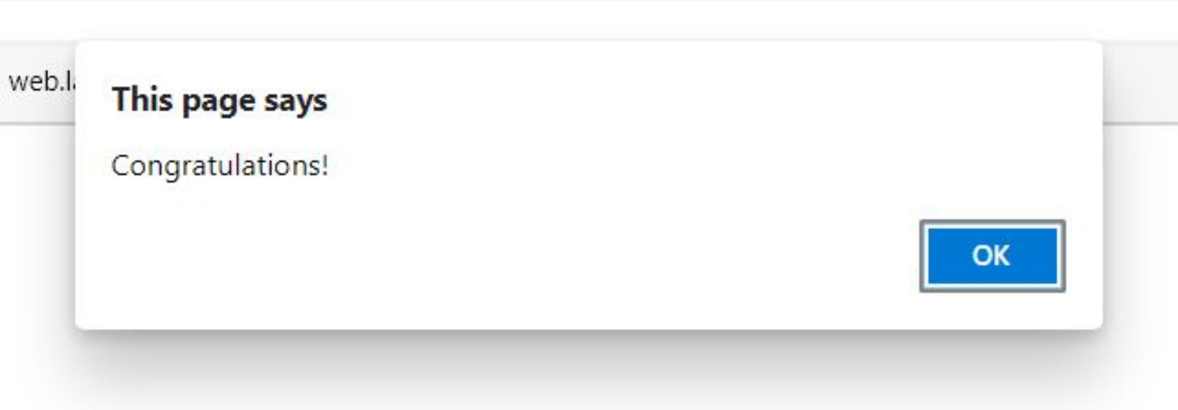
```
let salary = 30000;
salary = salary + 5000;
salary = salary * 2;

console.log(salary);
// should output 70000
```

Alerts

`alert()` generates a pop-up notification with the given content.



```
alert("Congratulations!");
```



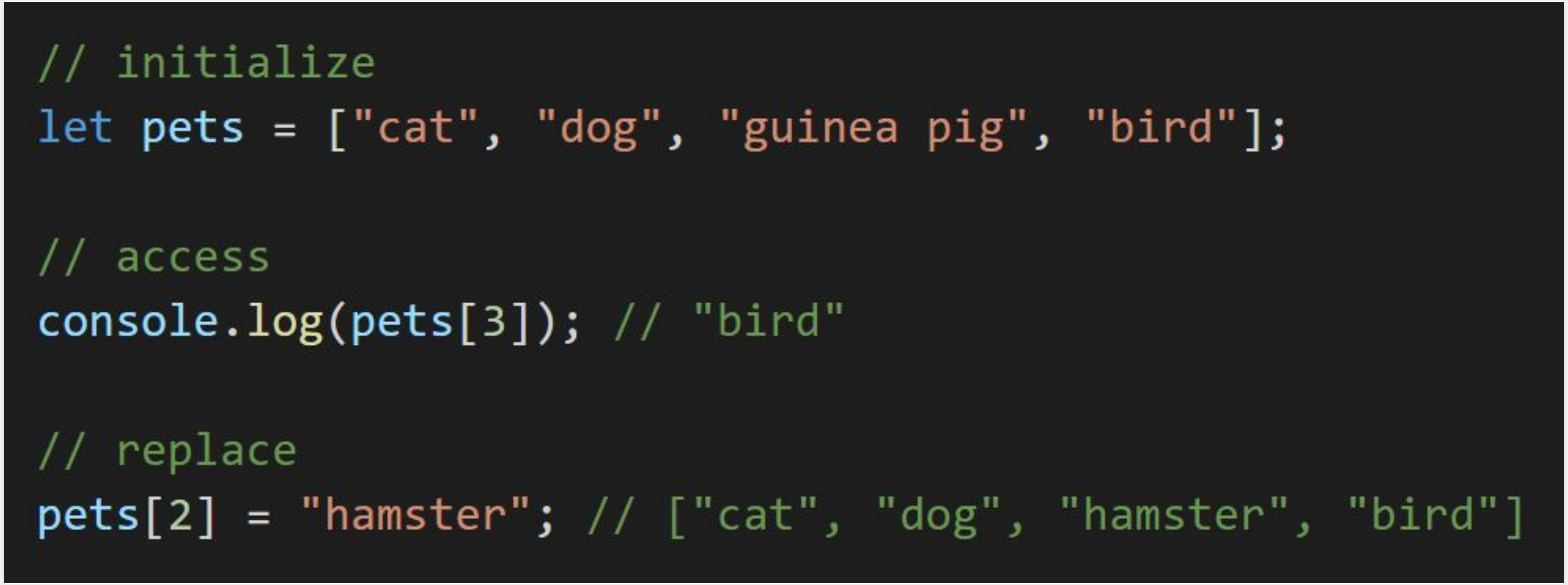
This page says

Congratulations!

Questions?

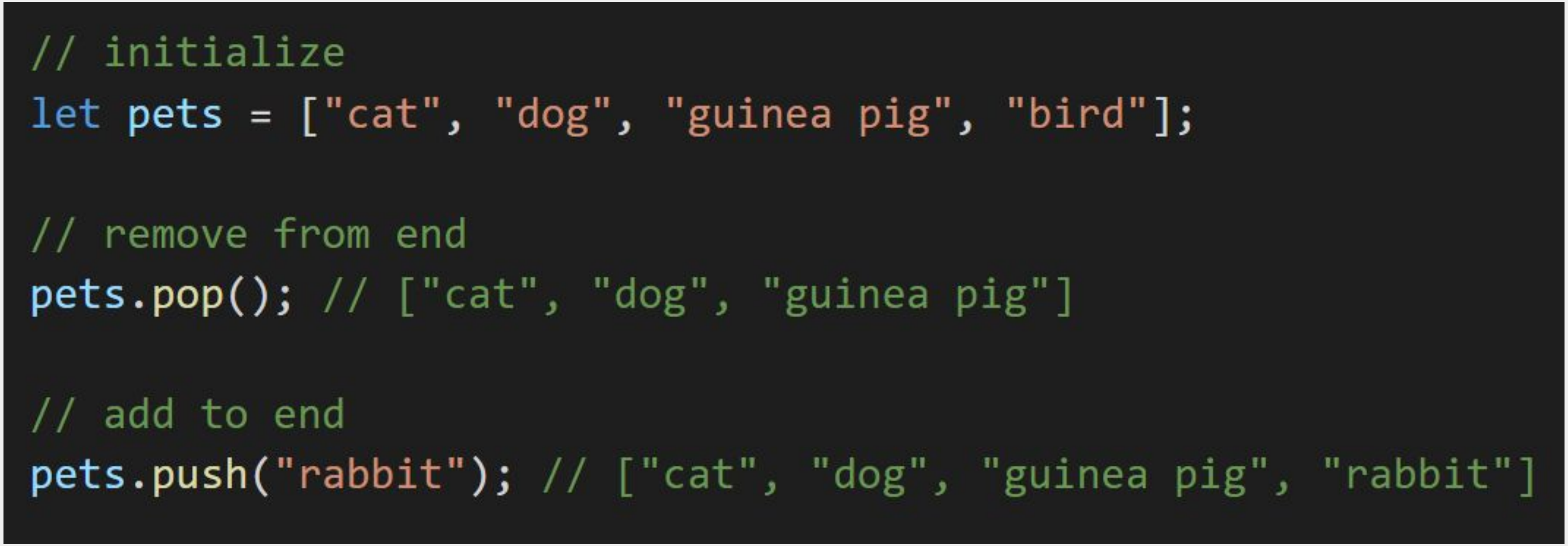
Arrays

For when you want to store a sequence of (ideally similar) items:



```
// initialize
let pets = ["cat", "dog", "guinea pig", "bird"];

// access
console.log(pets[3]); // "bird"

// replace
pets[2] = "hamster"; // ["cat", "dog", "hamster", "bird"]
```

Arrays



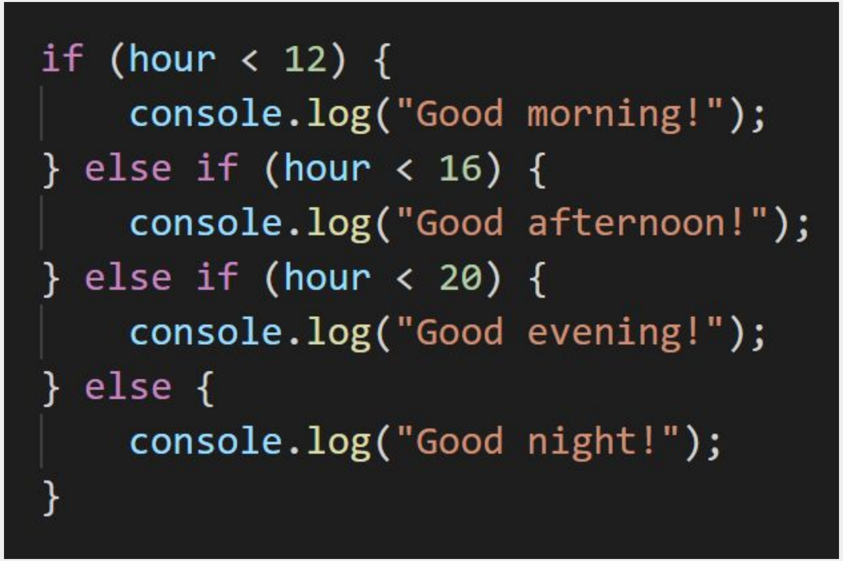
```
// initialize
let pets = ["cat", "dog", "guinea pig", "bird"];

// remove from end
pets.pop(); // ["cat", "dog", "guinea pig"]

// add to end
pets.push("rabbit"); // ["cat", "dog", "guinea pig", "rabbit"]
```

Conditionals

We often want to perform different actions in response to different conditions.

For this, we use the **conditional operators** `if`, `else`, and `else if`:

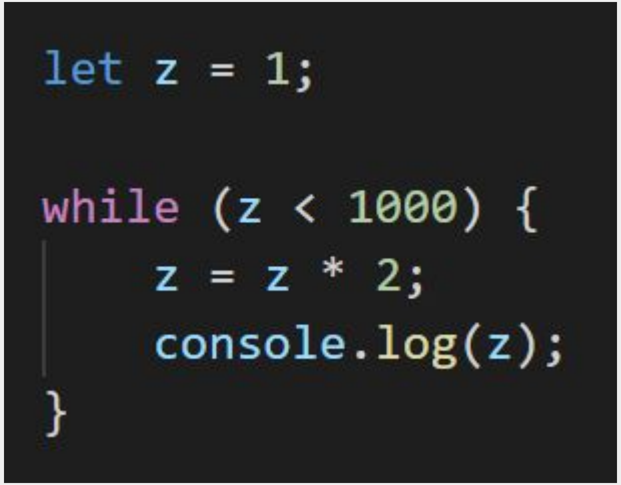
Note the indent (tab)!
It's not necessary, but
it will make your code
much more readable.



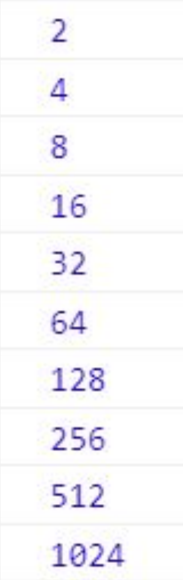
```
if (hour < 12) {
    console.log("Good morning!");
} else if (hour < 16) {
    console.log("Good afternoon!");
} else if (hour < 20) {
    console.log("Good evening!");
} else {
    console.log("Good night!");
}
```

While loops

What if we want to repeat an action *as long as* some condition is satisfied?



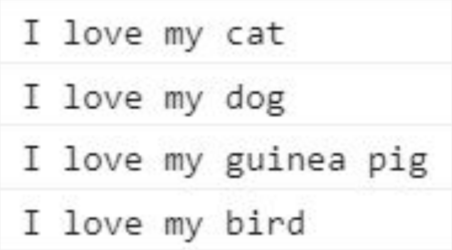
```
let z = 1;

while (z < 1000) {
    z = z * 2;
    console.log(z);
}
```



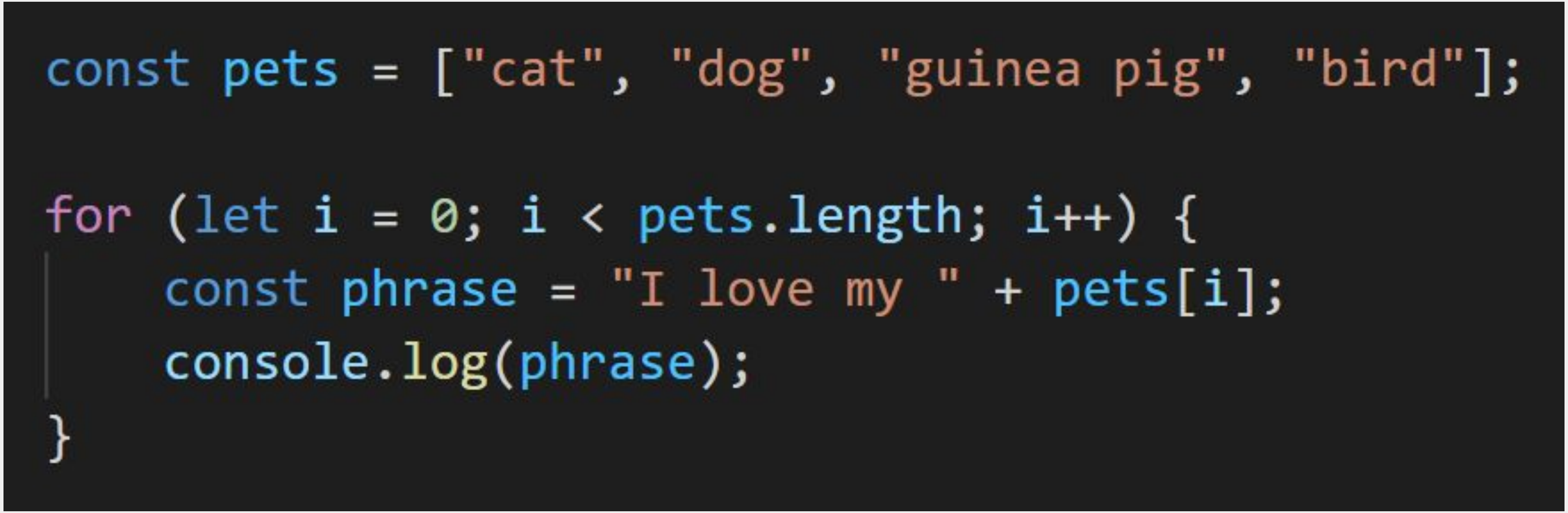
```
2
4
8
16
32
64
128
256
512
1024
```

For loops

Useful when we want to iterate through indices:



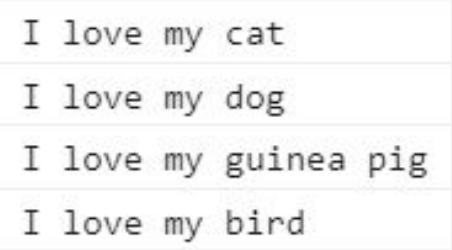
```
I love my cat
I love my dog
I love my guinea pig
I love my bird
```



```javascript
const pets = ["cat", "dog", "guinea pig", "bird"];

for (let i = 0; i < pets.length; i++) {
    const phrase = "I love my " + pets[i];
    console.log(phrase);
}
```

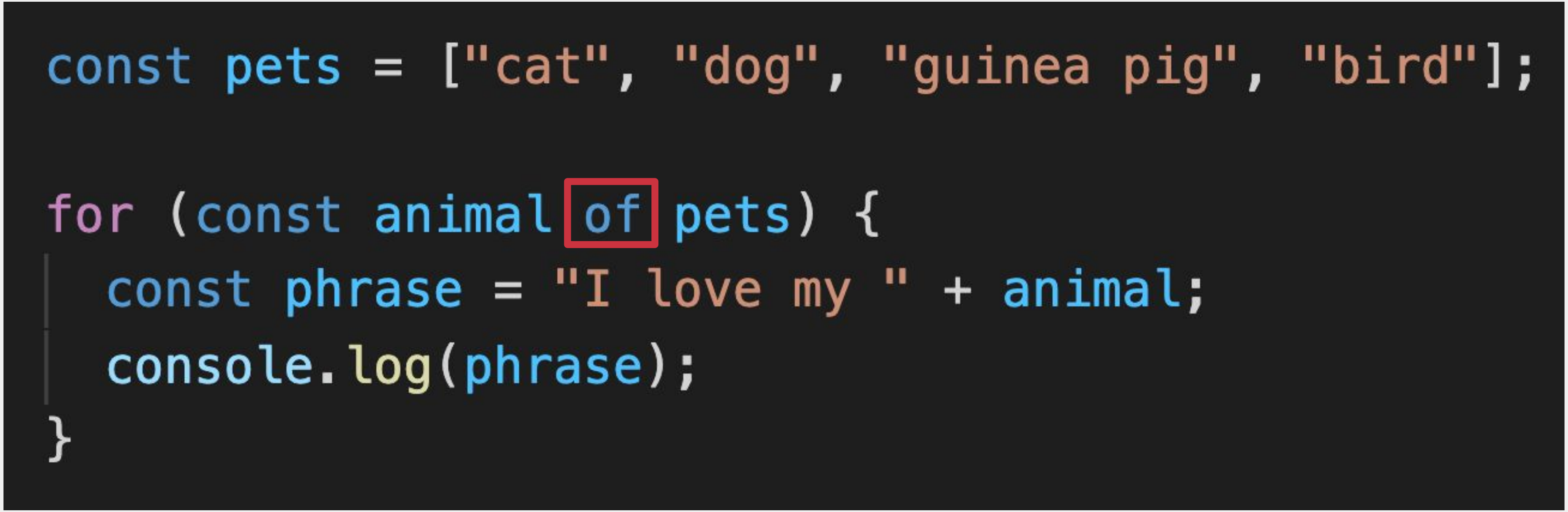
For … of …

A more “pythonic” way of iterating:



```
I love my cat
I love my dog
I love my guinea pig
I love my bird
```

Requires the keyword **of** instead of **in**

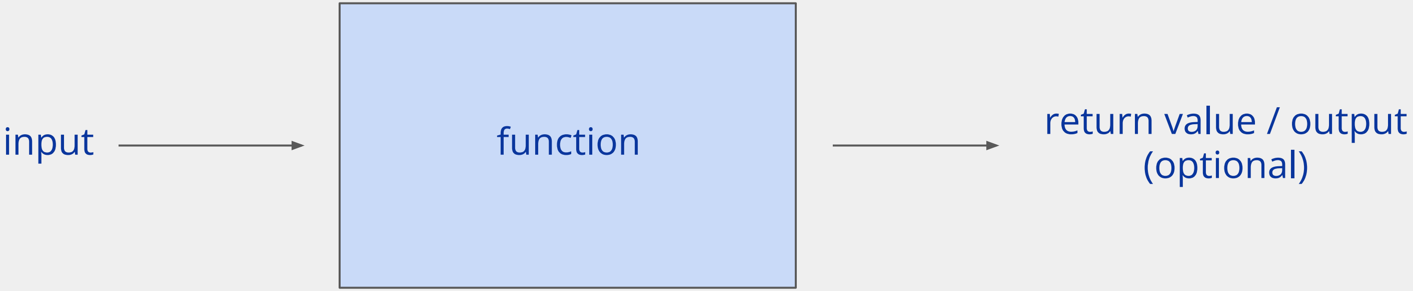


```
const pets = ["cat", "dog", "guinea pig", "bird"];

for (const animal of pets) {
  const phrase = "I love my " + animal;
  console.log(phrase);
}
```

Questions?

Functions

A **function** is a compartmentalized block of code which can be given input and asked to perform a set of instructions on that input.



Sometimes, we want the function to **return** an output value.

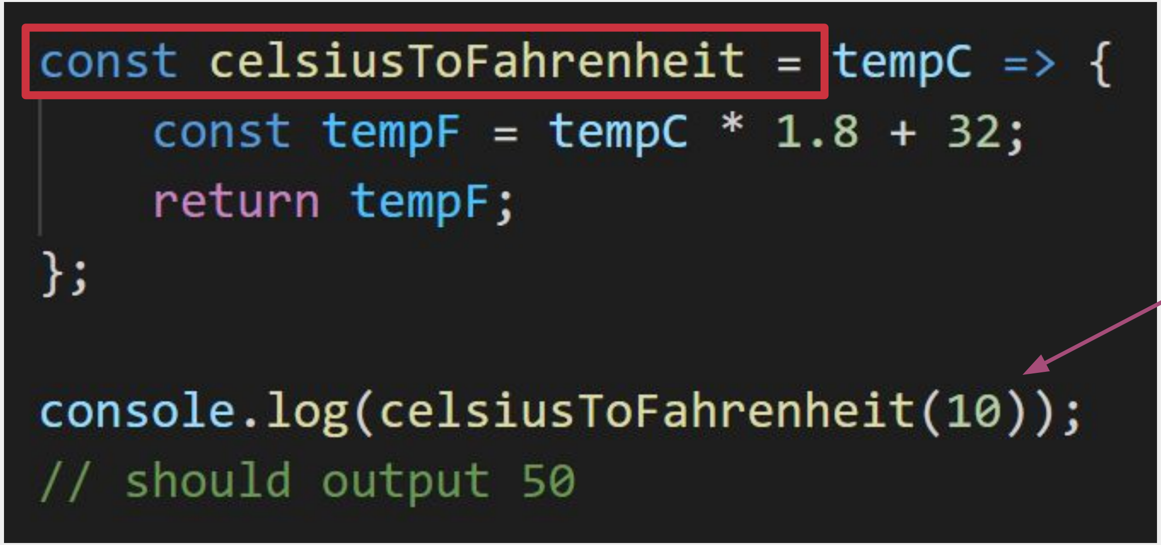
Other times, what matters more is what happens inside the “box” (the function body).

Functions

JavaScript functions can be defined using this cute little arrow =>

Syntax: `(parameters) => { body };`

parentheses are optional if there’s only one parameter



```
const celsiusToFahrenheit = tempC => {
    const tempF = tempC * 1.8 + 32;
    return tempF;
};

console.log(celsiusToFahrenheit(10));
// should output 50
```

The actual parameter value(s) given to the function are called **arguments**.

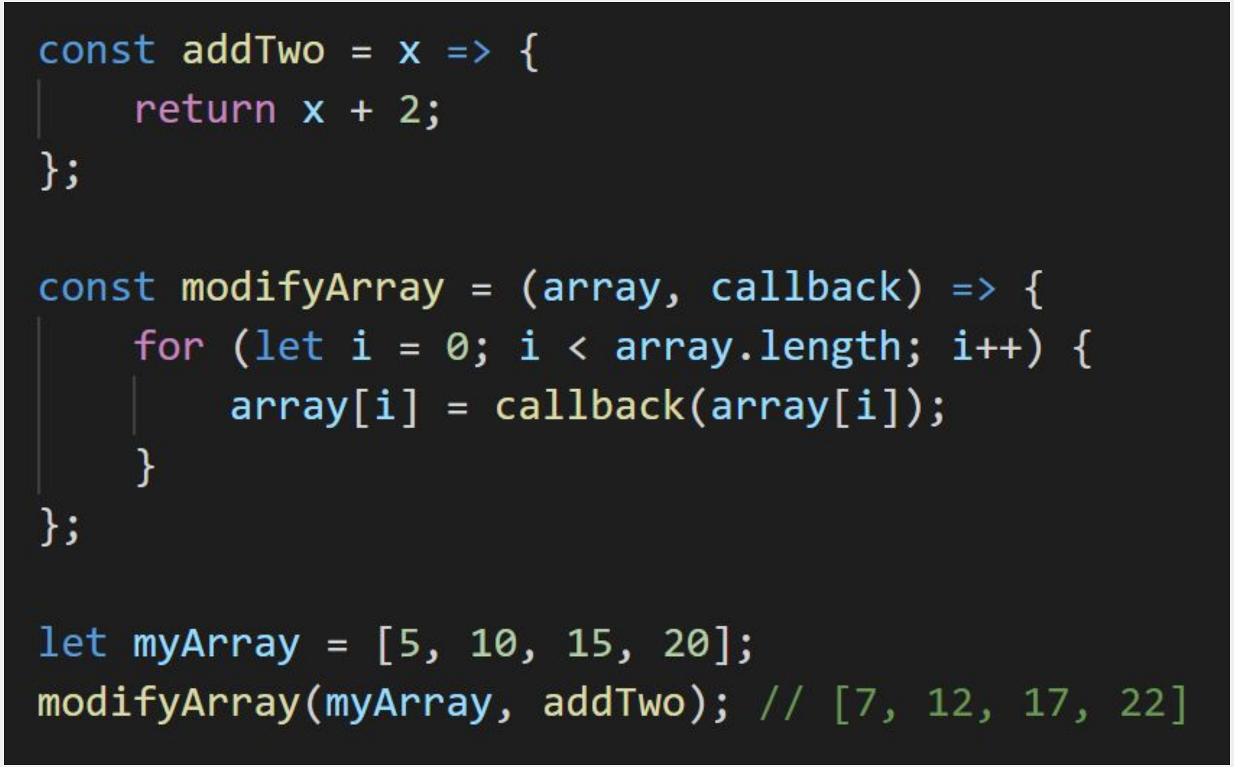
Callback functions

In JavaScript, functions can be passed around like any other variable.

This means we can give a **“callback”** function as an argument to another function!

Why might we do this?

Callback functions



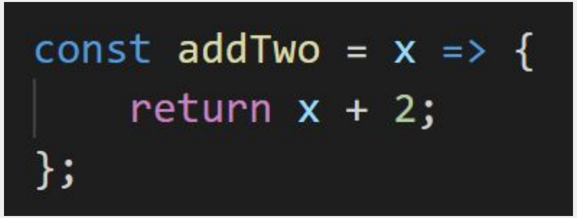
```
const addTwo = x => {
    return x + 2;
};

const modifyArray = (array, callback) => {
    for (let i = 0; i < array.length; i++) {
        array[i] = callback(array[i]);
    }
};

let myArray = [5, 10, 15, 20];
modifyArray(myArray, addTwo); // [7, 12, 17, 22]
```

A common mistake with callback functions

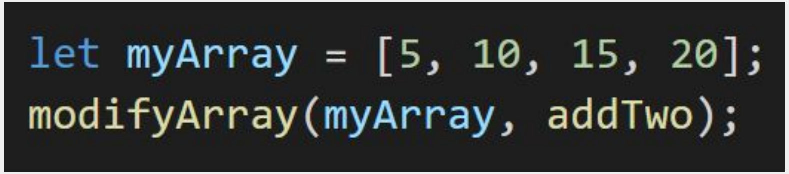
`addTwo` is a function.

`addTwo(x)` is the *value* that gets returned when you run `addTwo` on `x`.

`modifyArray` needs to be given the actual function in order to use it!

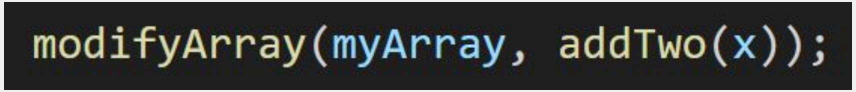


```
const addTwo = x => {
    return x + 2;
};
```



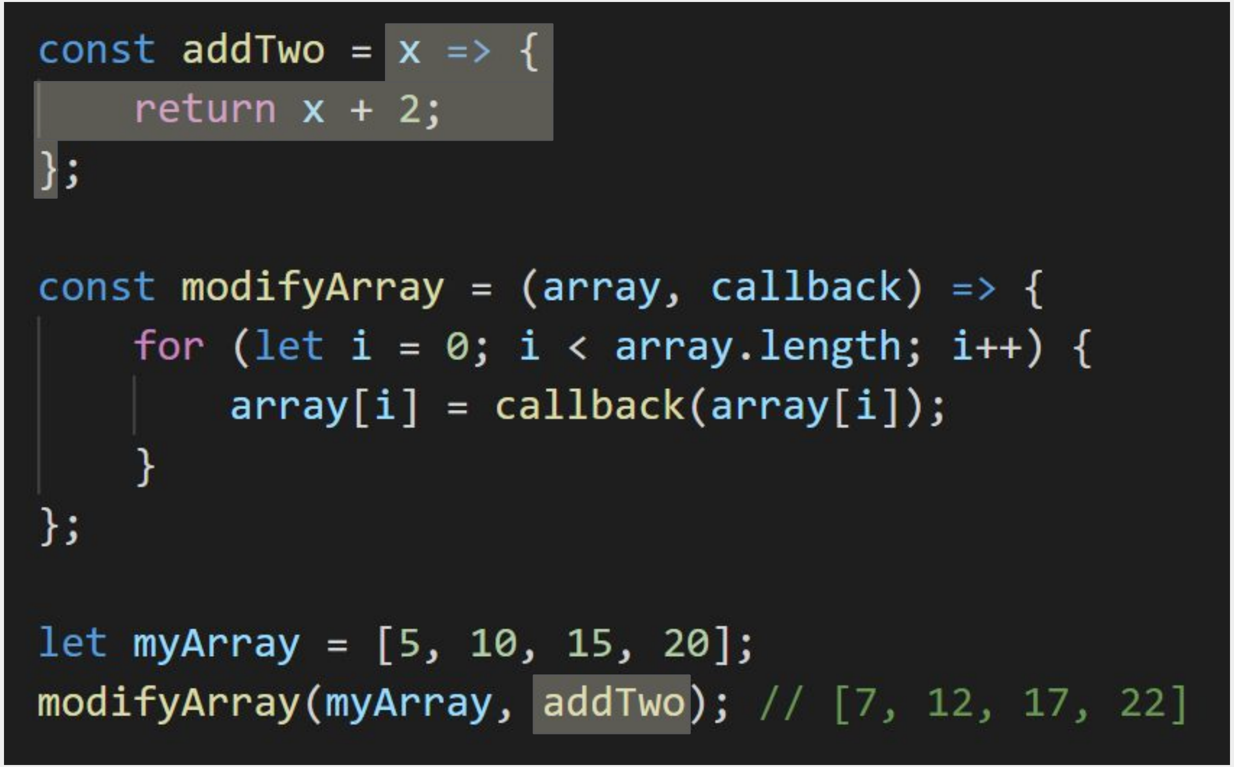
```
let myArray = [5, 10, 15, 20];
modifyArray(myArray, addTwo);
```

Don’t do this: 🚫



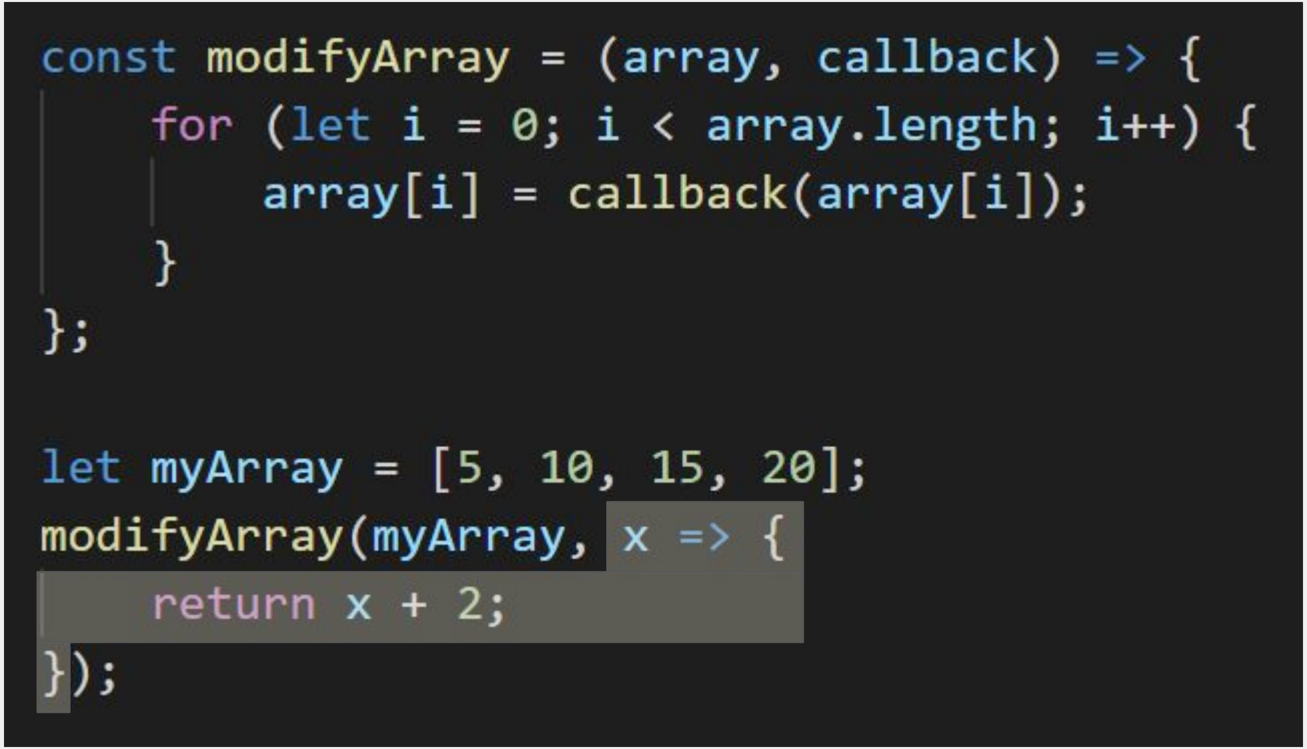
```
modifyArray(myArray, addTwo(x));
```

Callback functions



```javascript
const addTwo = x => {
    return x + 2;
};

const modifyArray = (array, callback) => {
    for (let i = 0; i < array.length; i++) {
        array[i] = callback(array[i]);
    }
};

let myArray = [5, 10, 15, 20];
modifyArray(myArray, addTwo); // [7, 12, 17, 22]
```

Anonymous functions

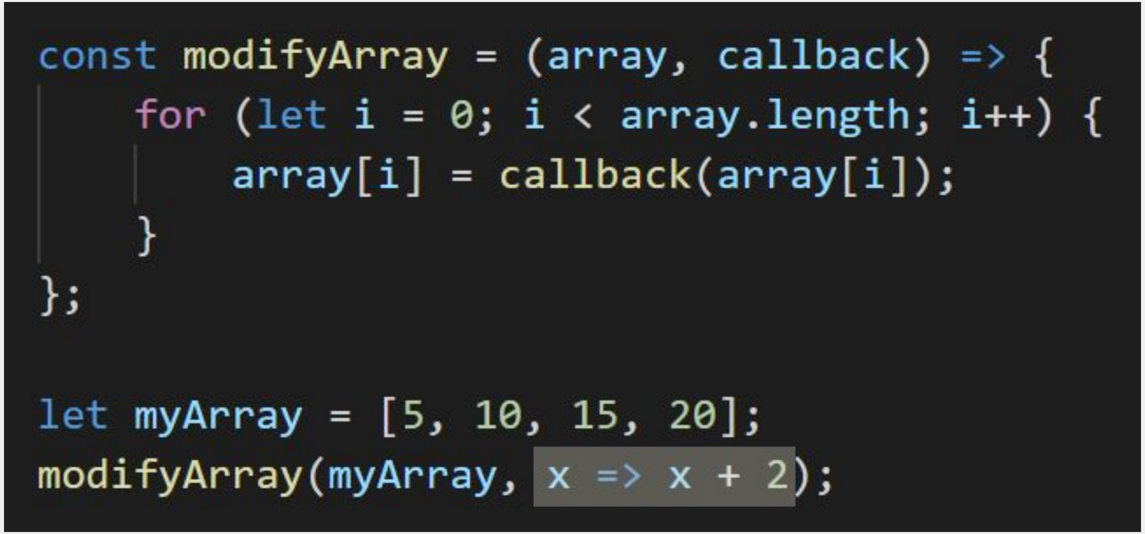


```javascript
const modifyArray = (array, callback) => {
    for (let i = 0; i < array.length; i++) {
        array[i] = callback(array[i]);
    }
};

let myArray = [5, 10, 15, 20];
modifyArray(myArray, x => {
    return x + 2;
});
```

Anonymous functions

If your function is simple enough, you can use the following shorthand.

Syntax: `(parameters) => output;`



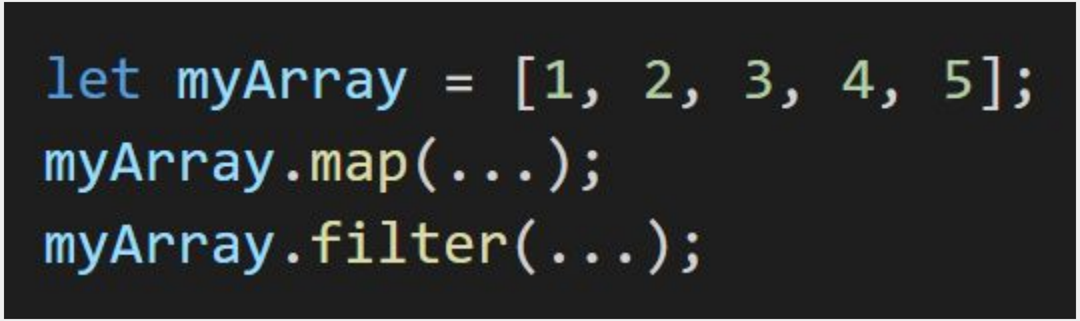
```
const modifyArray = (array, callback) => {
    for (let i = 0; i < array.length; i++) {
        array[i] = callback(array[i]);
    }
};

let myArray = [5, 10, 15, 20];
modifyArray(myArray, x => x + 2);
```

Other built-in array functions

If it seems common enough, there’s probably a built-in function for it!

For arrays, we’ve seen `push` and `pop`, which mutate the target array *in-place*.

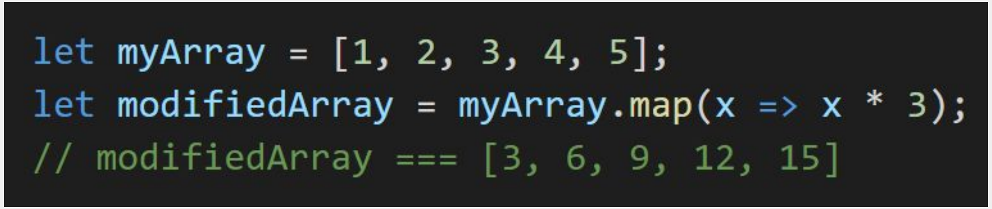
We also have `map` and `filter`, which produce a *new* array based on some instruction. (This “instruction” is going to be a callback function!)



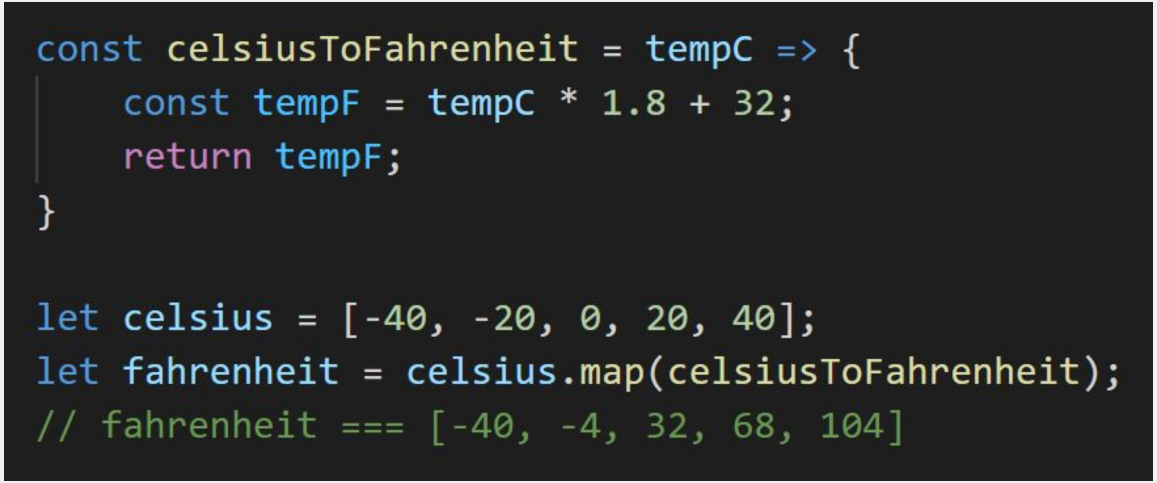
```
let myArray = [1, 2, 3, 4, 5];
myArray.map(...);
myArray.filter(...);
```

map(...)

Creates a new array by applying the callback function to every element of the starting array.



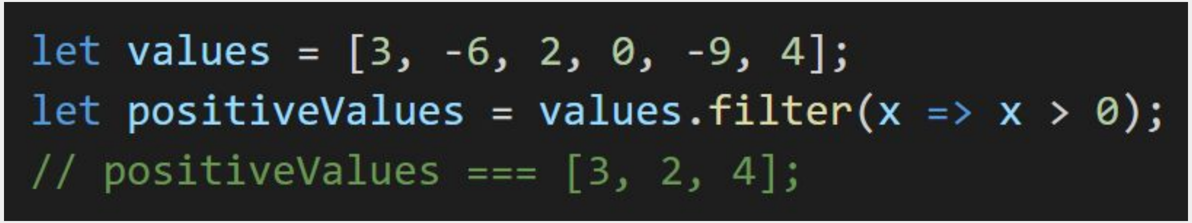
```
let myArray = [1, 2, 3, 4, 5];
let modifiedArray = myArray.map(x => x * 3);
// modifiedArray === [3, 6, 9, 12, 15]
```



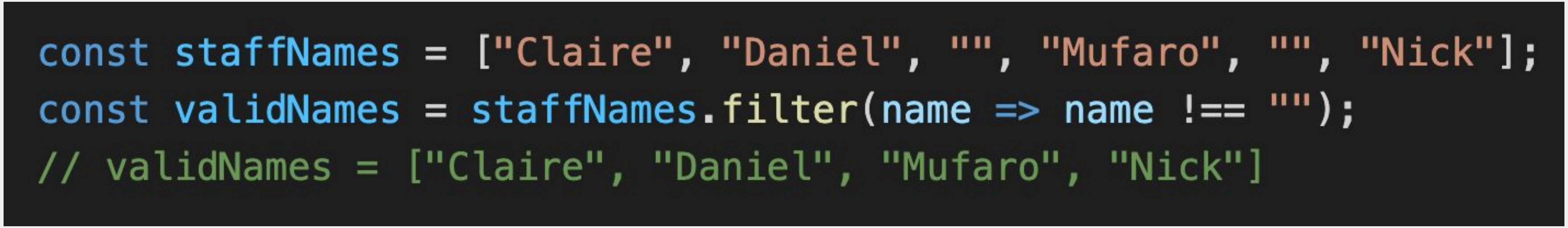
```
const celsiusToFahrenheit = tempC => {
    const tempF = tempC * 1.8 + 32;
    return tempF;
}

let celsius = [-40, -20, 0, 20, 40];
let fahrenheit = celsius.map(celsiusToFahrenheit);
// fahrenheit === [-40, -4, 32, 68, 104]
```

filter(...)

Creates a new array by selecting the elements in the starting array which pass the given “test” (i.e. *filtering out* the “bad” elements and keeping the “good” ones).



```
let values = [3, -6, 2, 0, -9, 4];
let positiveValues = values.filter(x => x > 0);
// positiveValues === [3, 2, 4];
```



```
const staffNames = ["Claire", "Daniel", "", "Mufaro", "", "Nick"];
const validNames = staffNames.filter(name => name !== "");
// validNames = ["Claire", "Daniel", "Mufaro", "Nick"]
```

Questions?

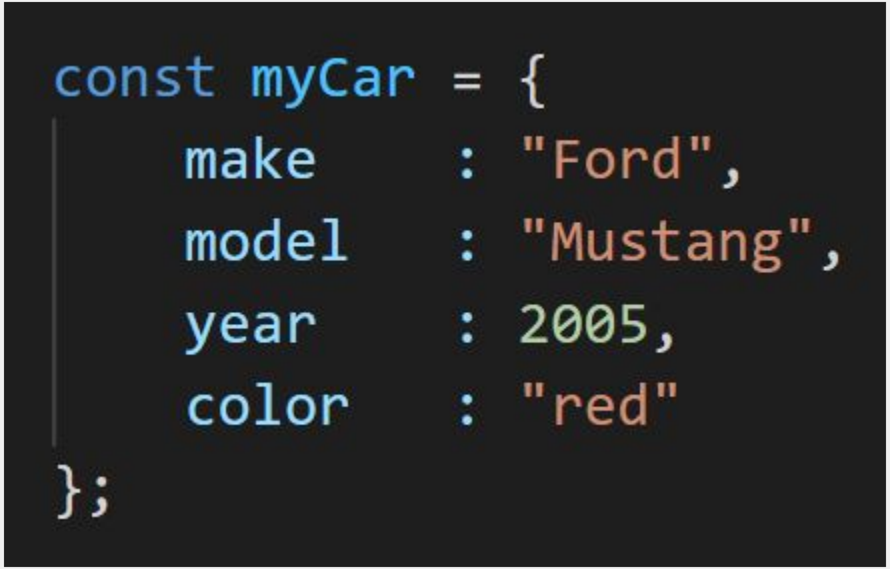
Comments?

Concerns?

Wait, we don’t have a primitive data type for this

Objects

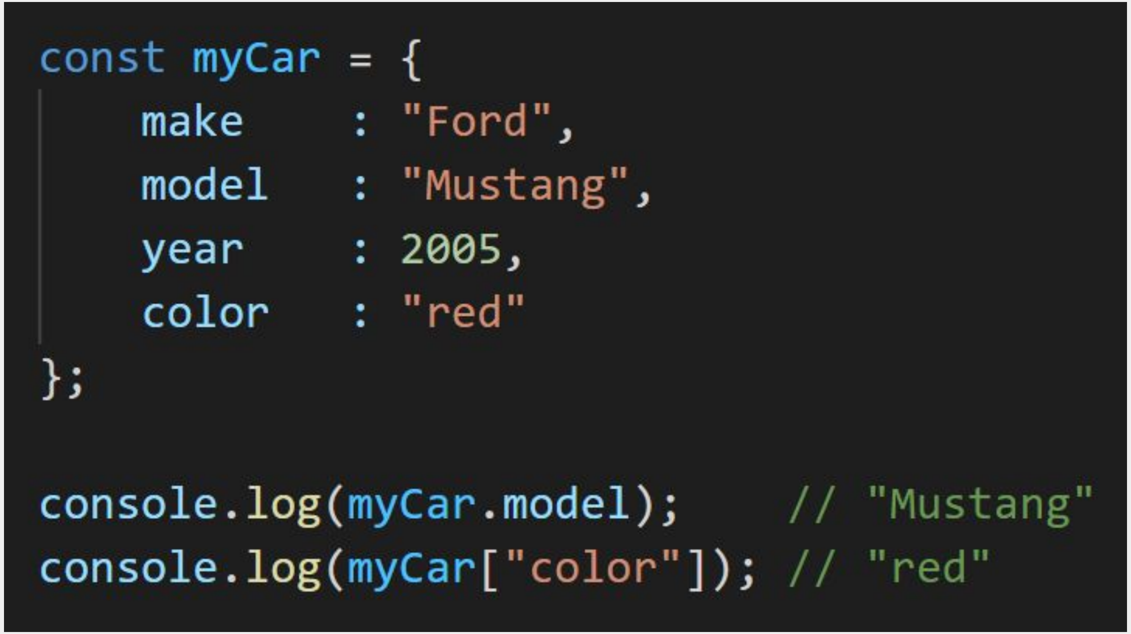
A JavaScript **object** is a collection of `name:value` pairs.



```
const myCar = {
    make    : "Ford",
    model   : "Mustang",
    year    : 2005,
    color   : "red"
};
```

Accessing properties

There are two ways to access object properties, if you know the property name:

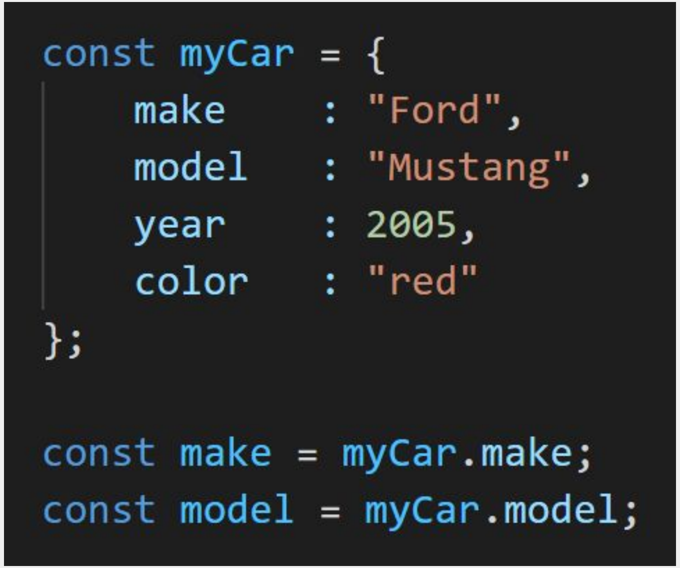


```
const myCar = {
    make    : "Ford",
    model   : "Mustang",
    year    : 2005,
    color   : "red"
};

console.log(myCar.model);    // "Mustang"
console.log(myCar["color"]); // "red"
```

Object destructuring

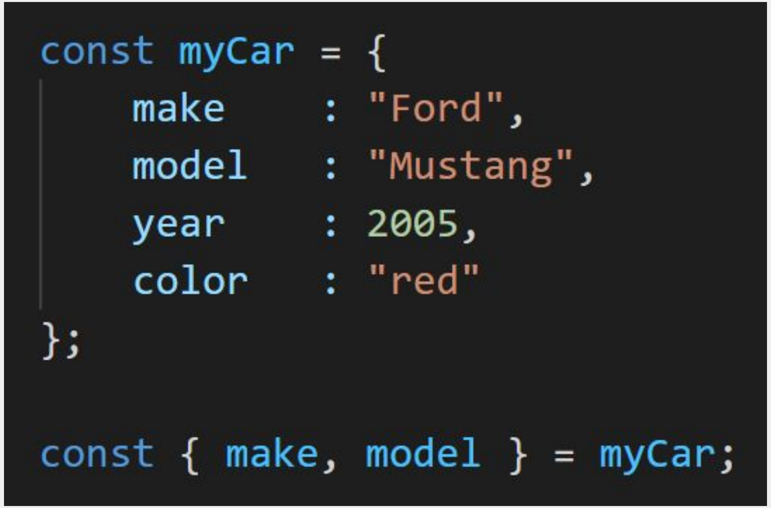
Object destructuring is a shorthand to obtain multiple properties at once.

without object destructuring



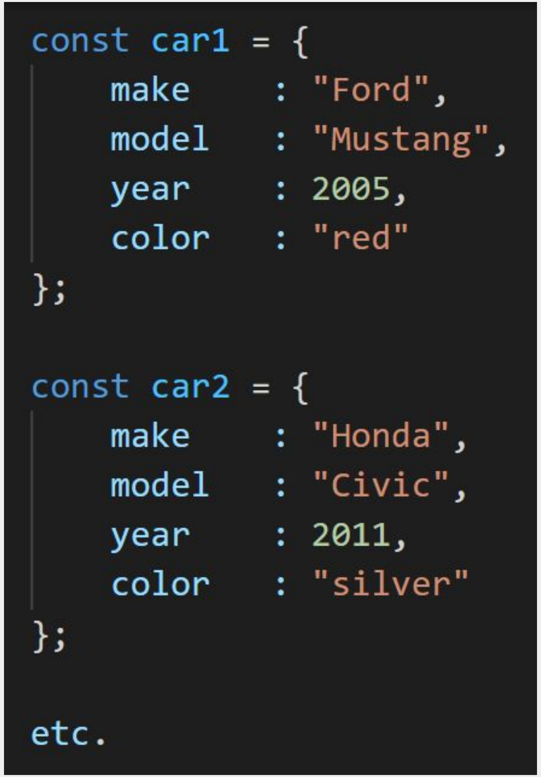
```
const myCar = {
    make    : "Ford",
    model   : "Mustang",
    year    : 2005,
    color   : "red"
};

const make = myCar.make;
const model = myCar.model;
```

with object destructuring



```
const myCar = {
    make    : "Ford",
    model   : "Mustang",
    year    : 2005,
    color   : "red"
};

const { make, model } = myCar;
```

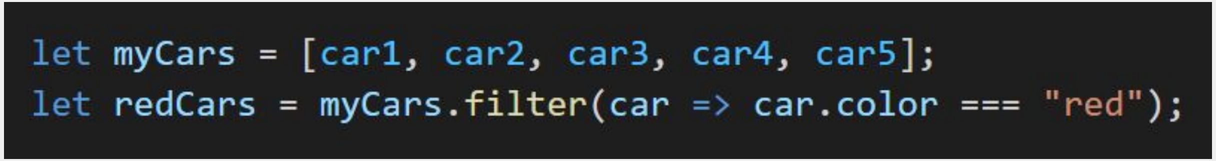
Using objects



```
const car1 = {
    make    : "Ford",
    model   : "Mustang",
    year    : 2005,
    color   : "red"
};

const car2 = {
    make    : "Honda",
    model   : "Civic",
    year    : 2011,
    color   : "silver"
};

etc.
```

We can treat objects like any other variable!

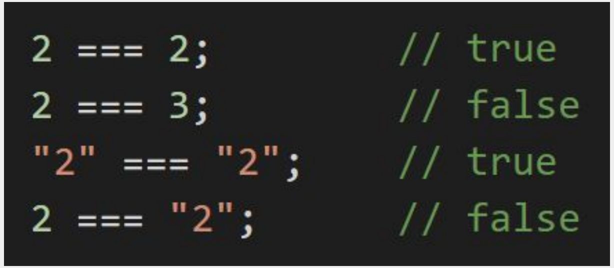
For example, given an array of car objects, we can apply a filter to just keep the red ones:



```
let myCars = [car1, car2, car3, car4, car5];
let redCars = myCars.filter(car => car.color === "red");
```

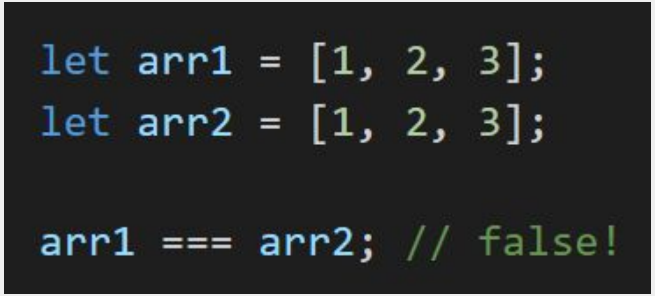
Equality...?

We use === to check if two *primitive* variables are equal in JavaScript.

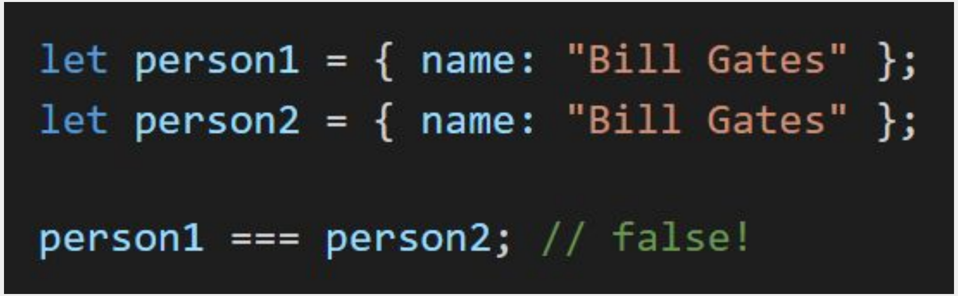


```
2 === 2;        // true
2 === 3;        // false
"2" === "2";    // true
2 === "2";      // false
```

But what does === mean for arrays and objects?



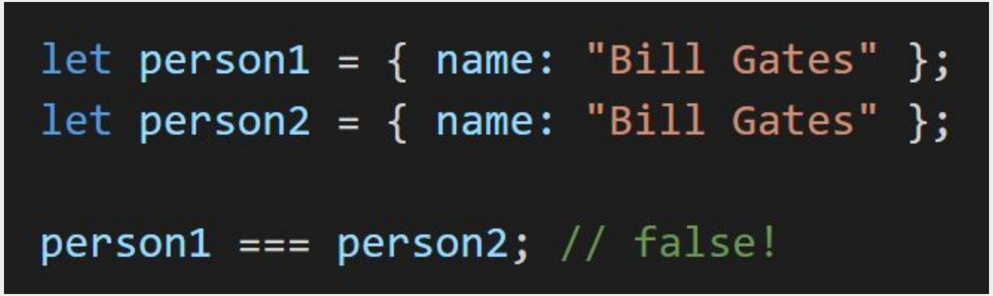
```
let arr1 = [1, 2, 3];
let arr2 = [1, 2, 3];

arr1 === arr2; // false!
```



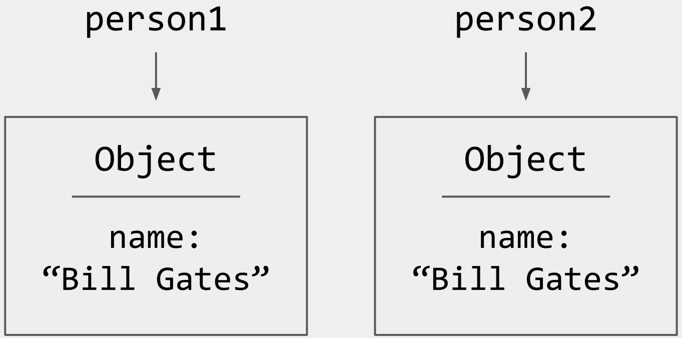
```
let person1 = { name: "Bill Gates" };
let person2 = { name: "Bill Gates" };

person1 === person2; // false!
```

Object references

Object variables are **references** – they point to where the data is actually stored.



```
let person1 = { name: "Bill Gates" };
let person2 = { name: "Bill Gates" };

person1 === person2; // false!
```



person1

Object

name:
“Bill Gates”

person2

Object

name:
“Bill Gates”

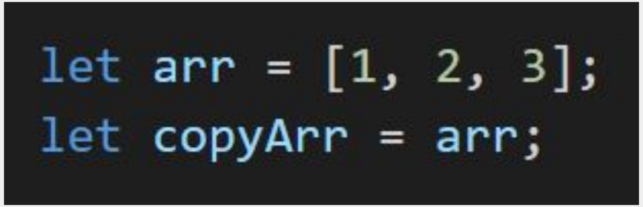
=== checks if the *references* are equal.

Two objects created separately are stored separately, so their references are different!

Same goes for arrays – two arrays created separately have different references.

How to copy arrays and objects

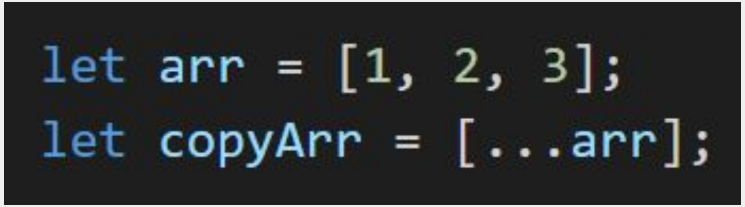
It’s not as simple as



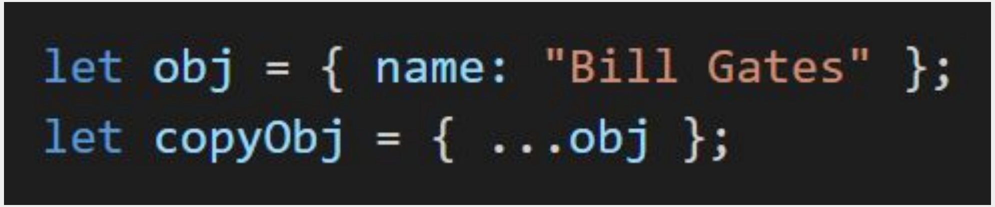
```
let arr = [1, 2, 3];
let copyArr = arr;
```

(Why not?)

One way to copy arrays and objects is to use the **spread** operator (`...`) like so:



```
let arr = [1, 2, 3];
let copyArr = [...arr];
```



```
let obj = { name: "Bill Gates" };
let copyObj = { ...obj };
```

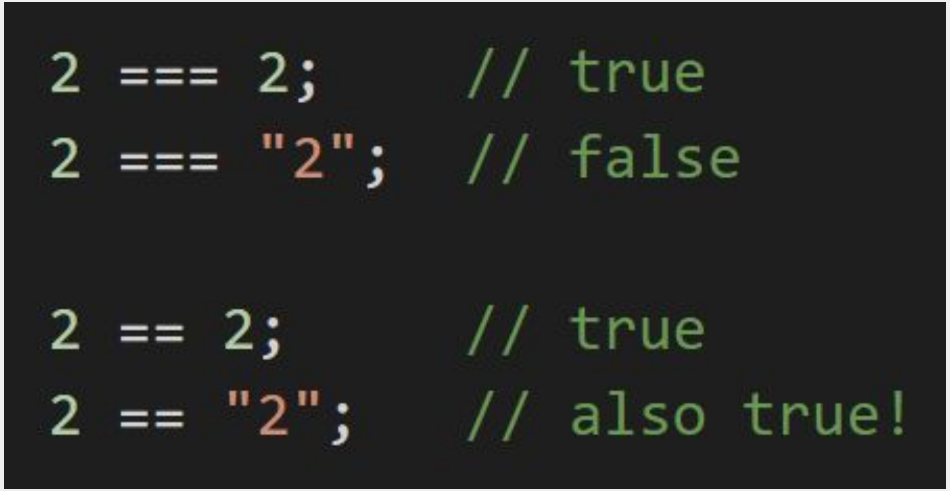
You could also manually copy over every item / property. But where’s the fun in that?

Why we don’t use ==

So, we use === to check equality in JavaScript.

But what does == do?

It performs *type coercion* (i.e. forces the arguments to be of the same type before comparing them)



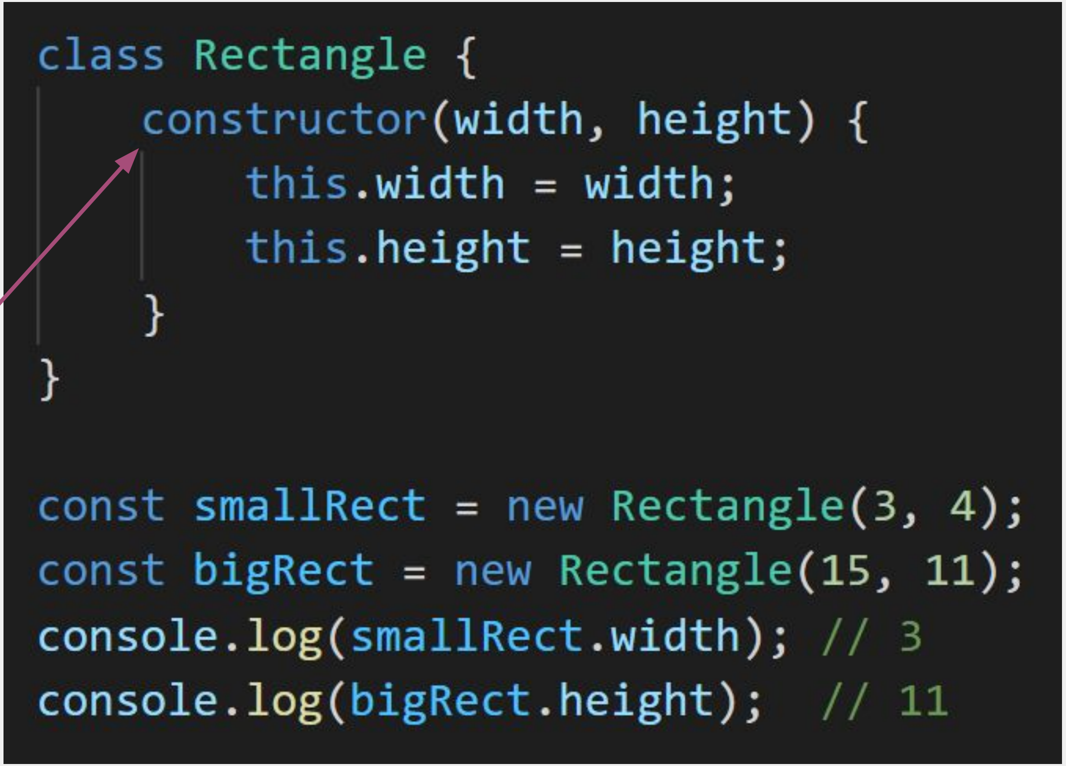
```
2 === 2;      // true
2 === "2";    // false

2 == 2;       // true
2 == "2";     // also true!
```

tl;dr don’t use ==

Classes

If you want multiple entities that are guaranteed to have shared behavior, use classes!

Every class has a **constructor** which tells it how to create a specific **instance** of that entity (in this case, a rectangle).

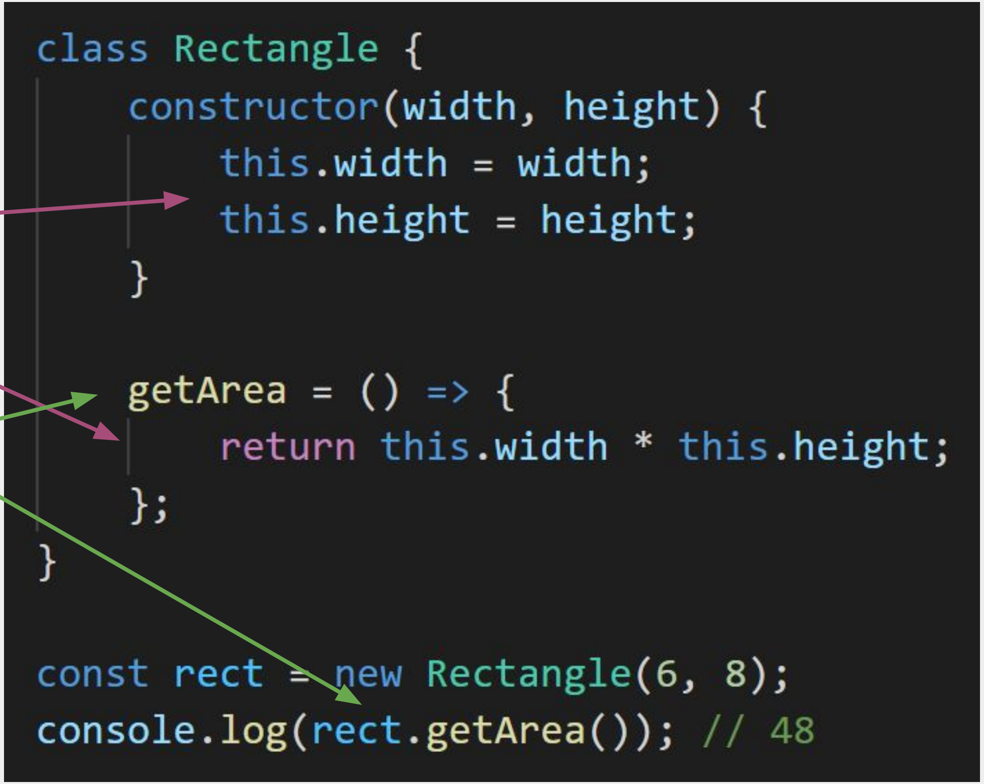


```
class Rectangle {
    constructor(width, height) {
        this.width = width;
        this.height = height;
    }
}

const smallRect = new Rectangle(3, 4);
const bigRect = new Rectangle(15, 11);
console.log(smallRect.width); // 3
console.log(bigRect.height);  // 11
```

Classes

Classes have **instance properties** which are specific to each instance. Instance properties are accessed with the keyword `this`.

Classes may also contain **methods** (functions) which can access and manipulate instance properties. The same methods exist in every instance of the class!



```
class Rectangle {
    constructor(width, height) {
        this.width = width;
        this.height = height;
    }

    getArea = () => {
        return this.width * this.height;
    };
}

const rect = new Rectangle(6, 8);
console.log(rect.getArea()); // 48
```

Summary

JavaScript is how we make things happen!

- Declare variables using let, const.
- boolean, number, string, null, undefined
- functions, arrays, objects, classes
- if, else, while, for

Up next: hands-on JavaScript workshop!

Questions?

W1: Javascript

Albert Xing

Agenda: Make Something With JS

Demo

Things We Need

1. Game setup
2. Snake
3. Respond to inputs
4. Food
5. Snake die

Let’s get started!

cd into your catbook-react folder

Run *git fetch*

Run *git reset --hard*

Run *git checkout w1-starter*