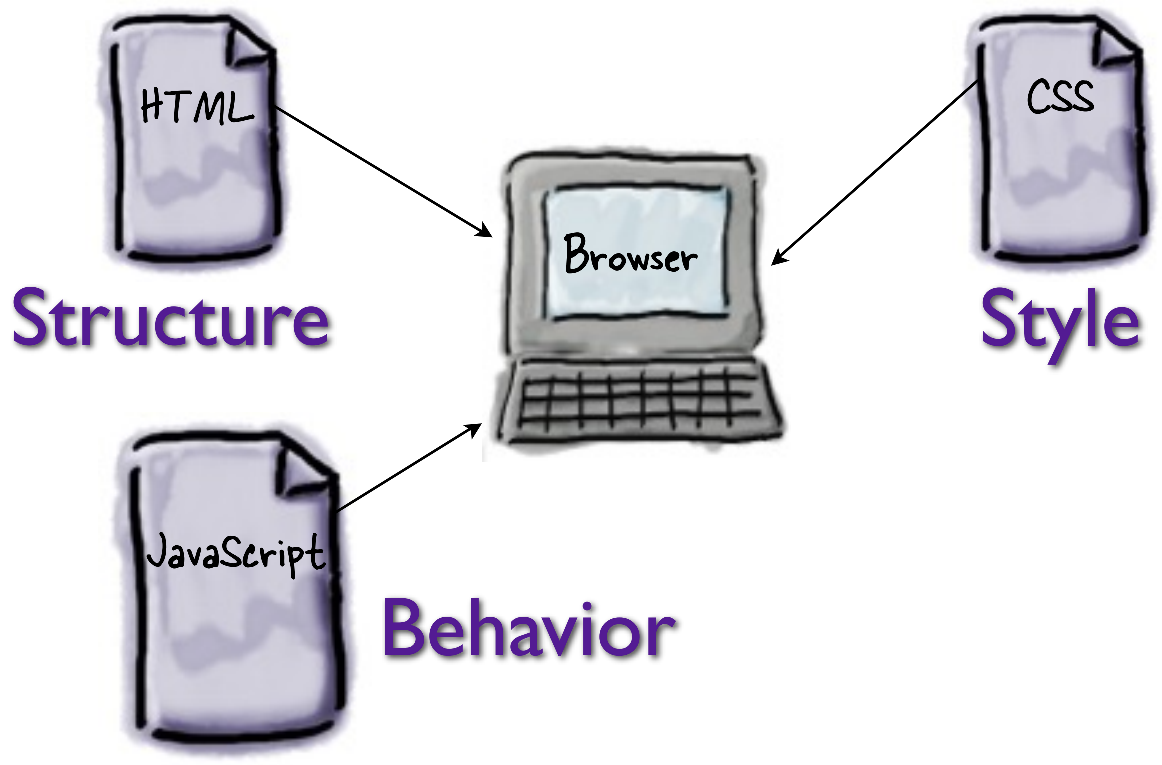
Introduction to Programming with Javascript

Introductions!

http://amzn.to/1nBluI0

What IS JavaScript?



What IS JavaScript?

Core language

This is the syntax and semantics of the programming language, JavaScript.

Host objects

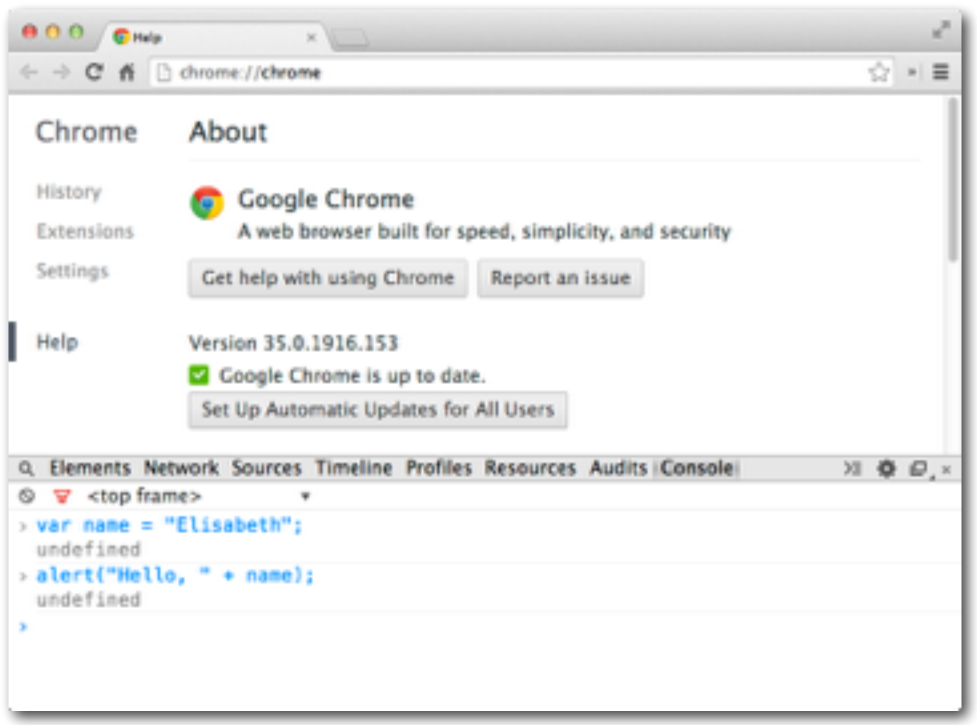
These are extra things that allow JavaScript to interact with its environment.

The environment could be the browser, or it could also be Photoshop or node.js.

Let's fire up a console...

- Start the browser of your choice. I personally like to use Chrome for web development.
- Figure out what version you have installed and make sure it's up to date.
- Access the console.
  - Chrome: View > Developer > JavaScript Console (Mac).
  - IE: Tools > F12 Developer > Script Tab
  - Firefox: Tools > Web Developer > Web Console
  - Safari: Develop > Show Error Console (turn on developer tools first)

Every browser has a console


Help
chrome://chrome
Chrome
About
History
Extensions
Settings
Help
Google Chrome
A web browser built for speed, simplicity, and security
Get help with using Chrome
Report an issue
Version 35.0.1916.153
Google Chrome is up to date.
Set Up Automatic Updates for All Users
Elements Network Sources Timeline Profiles Resources Audits Console
<top frame>

```
> var name = "Elisabeth";
  undefined
> alert("Hello, " + name);
  undefined
>
```

Try This

At the prompt (>), type:

```
var name = "Elisabeth";
```

Press return, and then type:

```
alert("Hello, " + name);
```

Now, try this too:

```
3 + 4
```

Console is similar to IRB ruby command line
Notice we end JavaScript statements with ;
Don't forget this!!!

More Variables

```
var name = "Elisabeth";

var isTeacher = true;

var isMale = false;

var firstAndLast = "Bob" + " " + "Smith";

var age = 10/2;

var ageInDogYears = age * 7;

var numStudents;
```

Similar to Ruby – dynamically typed.

Value Types

Number

```
var age = 5;
```

String

```
var name = "Elisabeth";
```

Boolean

```
var isTeacher = true;

var isMale = false;
```

Value Types

undefined

```
var numStudents;
```

- undefined means we haven't assigned a value to a variable yet.
- undefined is itself a value! (one that means "no value").

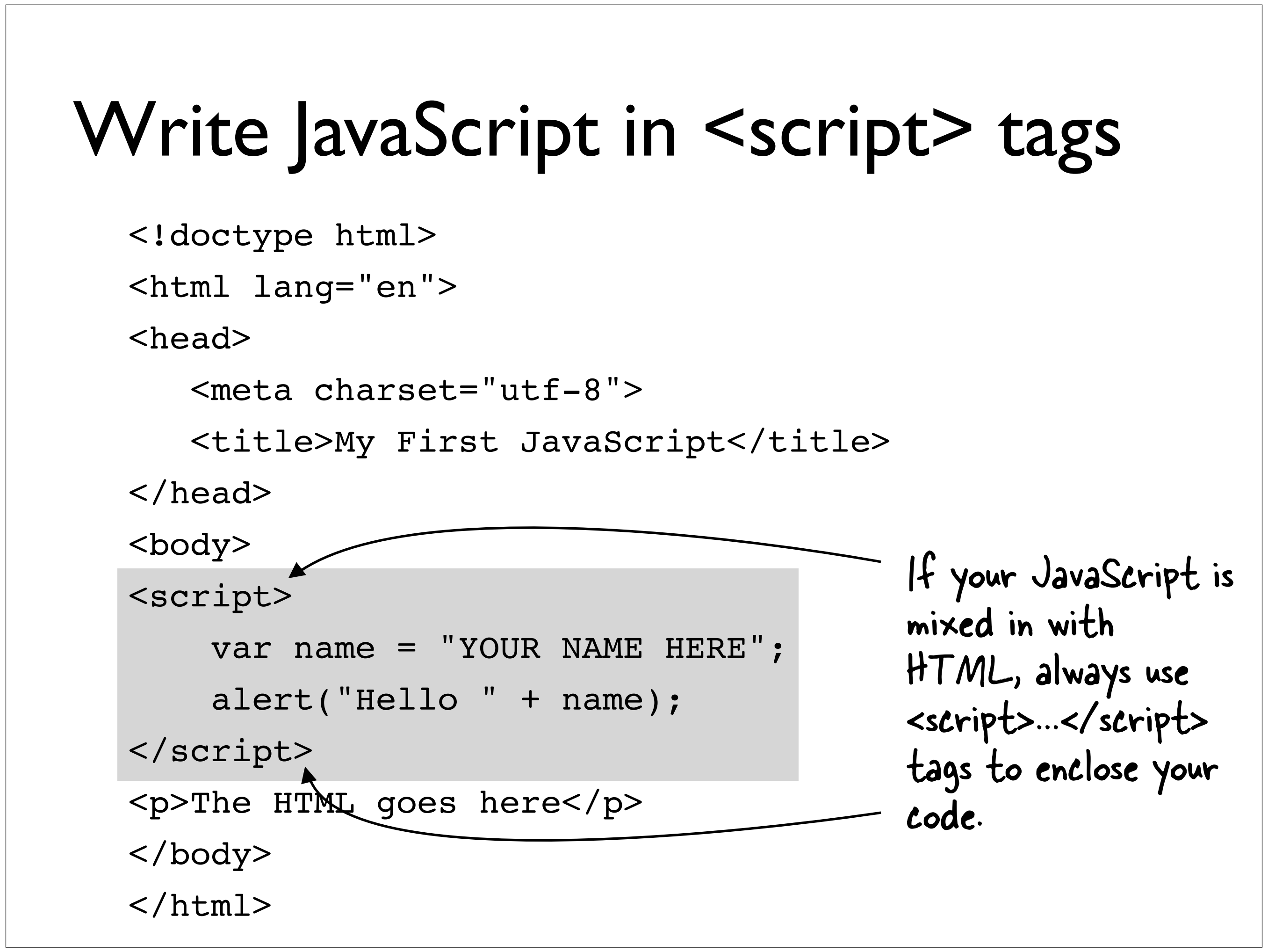
Aren't we supposed to be putting JavaScript in web pages???

Adding JavaScript to a web page

First, you need a text editor...

- A development editor, like Sublime Text, Atom, WebStorm, Coda, TextMate, or one of many others...
- Any editor that has a "plain text mode", like TextEdit or NotePad
- DON'T USE: Microsoft Word or Dreamweaver

Watch out for curly quotes!

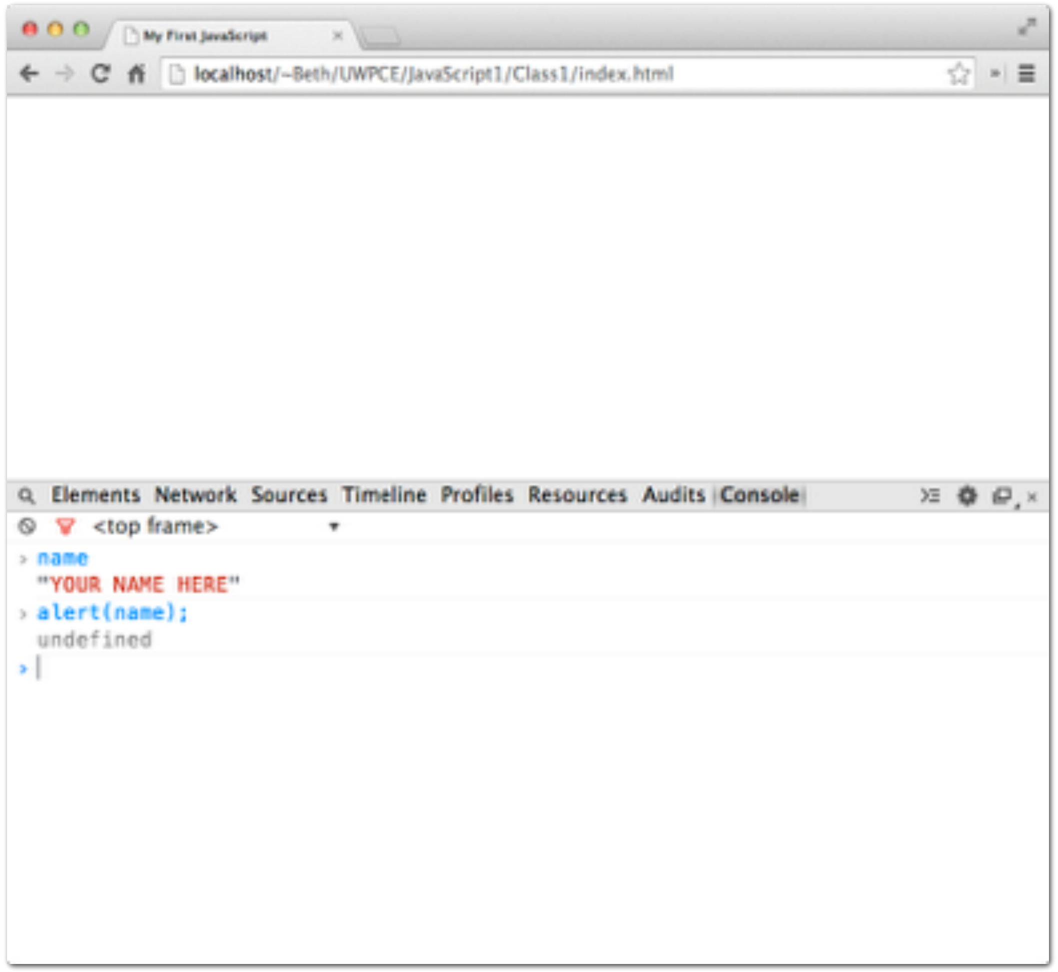


Write JavaScript in <script> tags

```
<!doctype html>
<html lang="en">
<head>
   <meta charset="utf-8">
   <title>My First JavaScript</title>
</head>
<body>
<script>
    var name = "YOUR NAME HERE";
    alert("Hello " + name);
</script>
<p>The HTML goes here</p>
</body>
</html>
```

If your JavaScript is mixed in with HTML, always use <script>...</script> tags to enclose your code.

Save the file as "index.html" on your computer.
Load the web page in your browser and see what happens.


My First JavaScript
localhost/~Beth/UWPCE/JavaScript1/Class1/index.html
Elements Network Sources Timeline Profiles Resources Audits Console
<top frame>

```
> name
  "YOUR NAME HERE"
> alert(name);
  undefined
>
```

You can access the top level variables you define in the web page you load, in the console

Where does JavaScript code go?

```
<!doctype html>
<html lang="en">
<head>
   <meta charset="utf-8">
   <title>My First JavaScript</title>
</head>
<body>
<script>
    var name = "YOUR NAME HERE";
    alert("Hello " + name);
</script>
<p>The HTML goes here</p>
</body>
</html>
```

It can go anywhere in your HTML between the <body>...</body> tags. Like at the top, or in the middle...

Where does JavaScript code go?

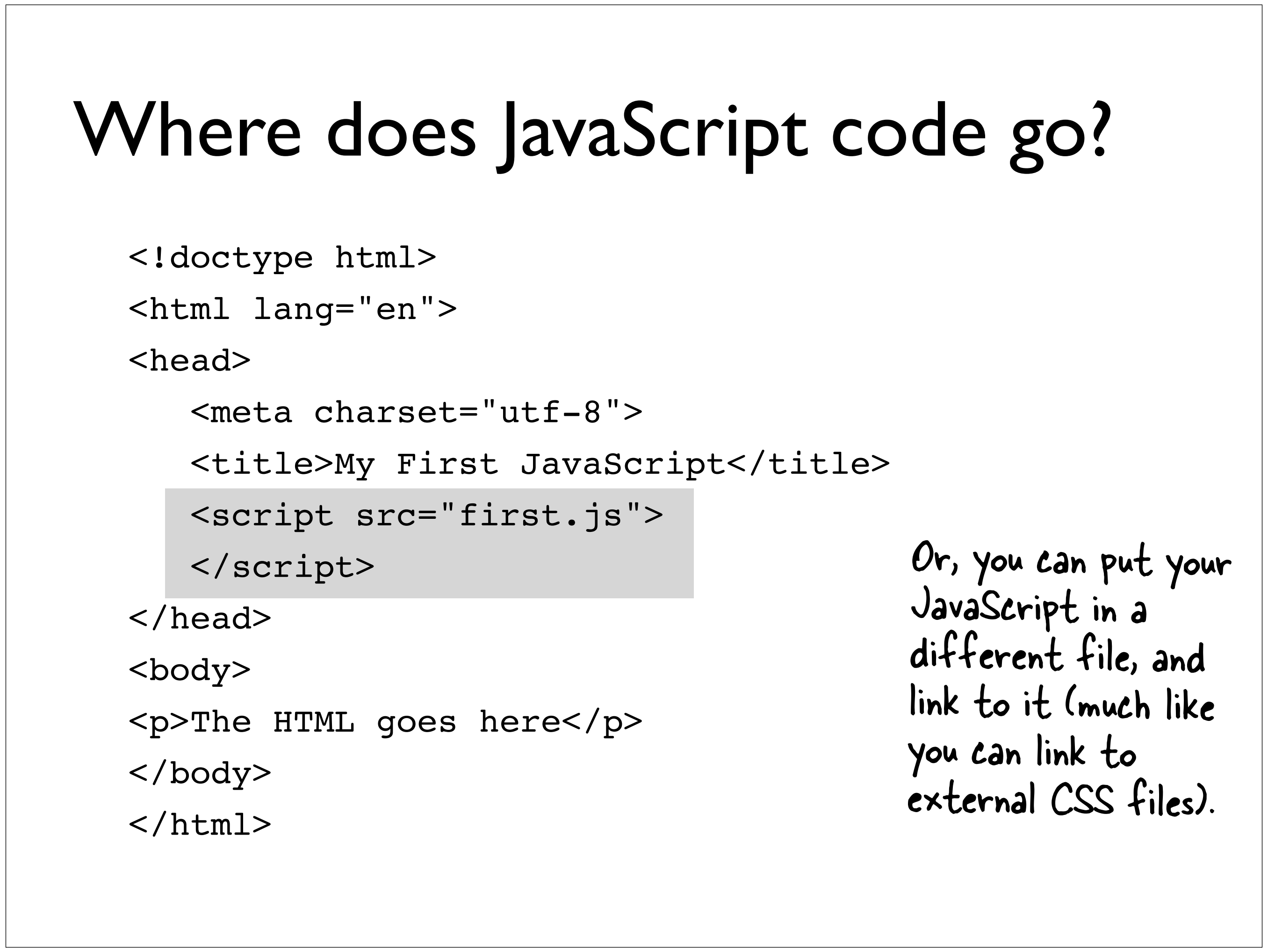
```
<!doctype html>
<html lang="en">
<head>
   <meta charset="utf-8">
   <title>My First JavaScript</title>
</head>
<body>
<p>The HTML goes here</p>
<script>
    var name = "YOUR NAME HERE";
    alert("Hello " + name);
</script>
</body>
</html>
```

... or at the bottom of your page, just before the </body> tag.

Where does JavaScript code go?

```
<!doctype html>
<html lang="en">
<head>
   <meta charset="utf-8">
   <title>My First JavaScript</title>
   <script>
     var name = "YOUR NAME HERE";
     alert("Hello " + name);
   </script>
</head>
<body>
<p>The HTML goes here</p>
</body>
</html>
```

Or, it can go in the <head>...</head> of your page, below the <title>.



Where does JavaScript code go?

```
<!doctype html>
<html lang="en">
<head>
   <meta charset="utf-8">
   <title>My First JavaScript</title>
   <script src="first.js">
   </script>
</head>
<body>
<p>The HTML goes here</p>
</body>
</html>
```

Or, you can put your JavaScript in a different file, and link to it (much like you can link to external CSS files).

Move your JavaScript to another file by:
1) Create a separate file named first.js
2) Copy the JavaScript code (NOT including the script tags) from your HTML
3) Paste the JavaScript code into the new file, first.js
4) Delete the JavaScript code from your HTML

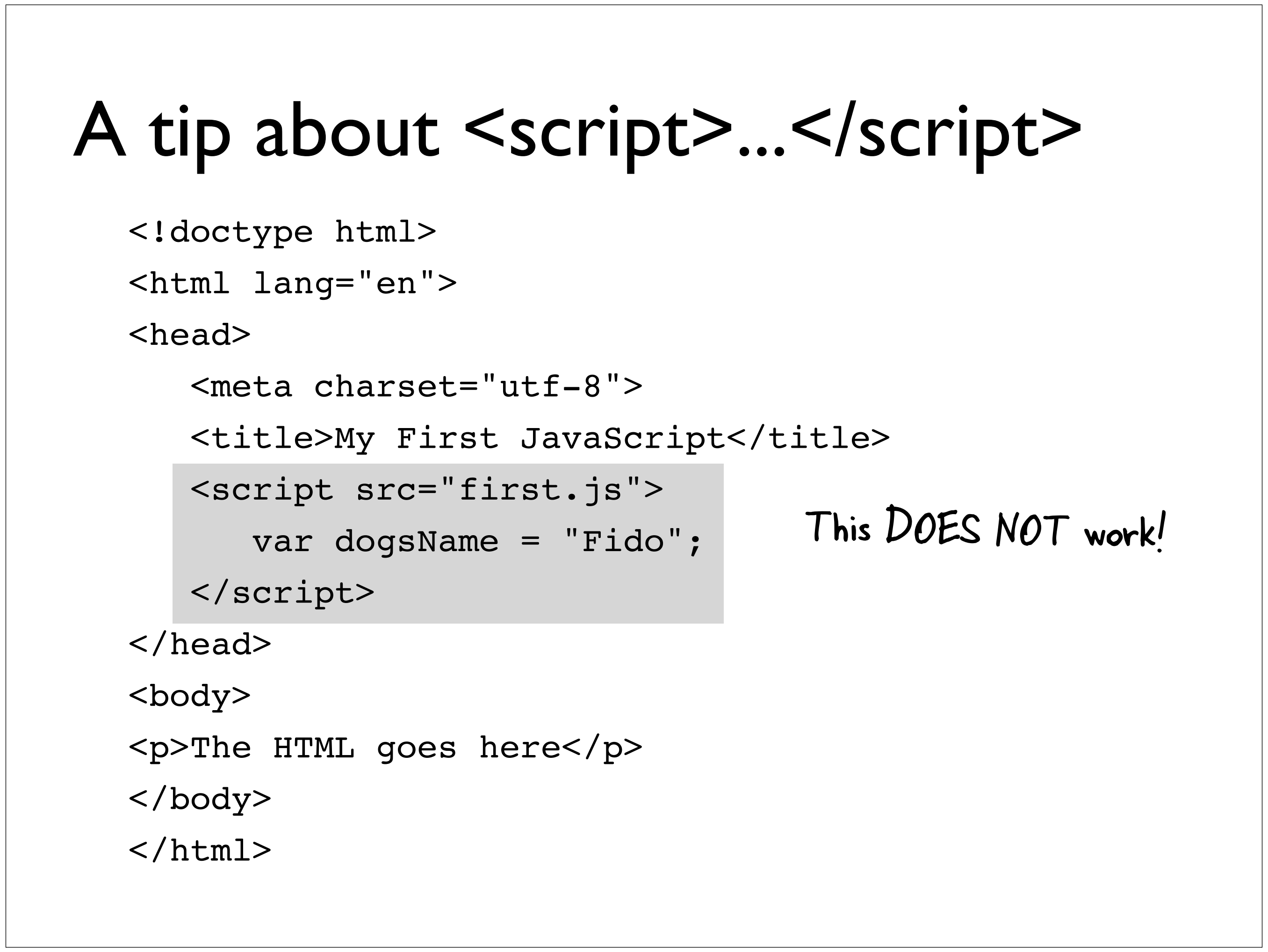
Your JavaScript should look like this:

```
var name = "YOUR NAME HERE";
alert("Hello " + name);
```

No <script>...</script> tags when you put your JS in another file.

Reload the index.html file.

Any difference?



A tip about <script>...</script>

```
<!doctype html>
<html lang="en">
<head>
   <meta charset="utf-8">
   <title>My First JavaScript</title>
   <script src="first.js">
      var dogsName = "Fido";
   </script>
</head>
<body>
<p>The HTML goes here</p>
</body>
</html>
```

This DOES NOT work!

You can't put external and inline JavaScript in the same script tag.

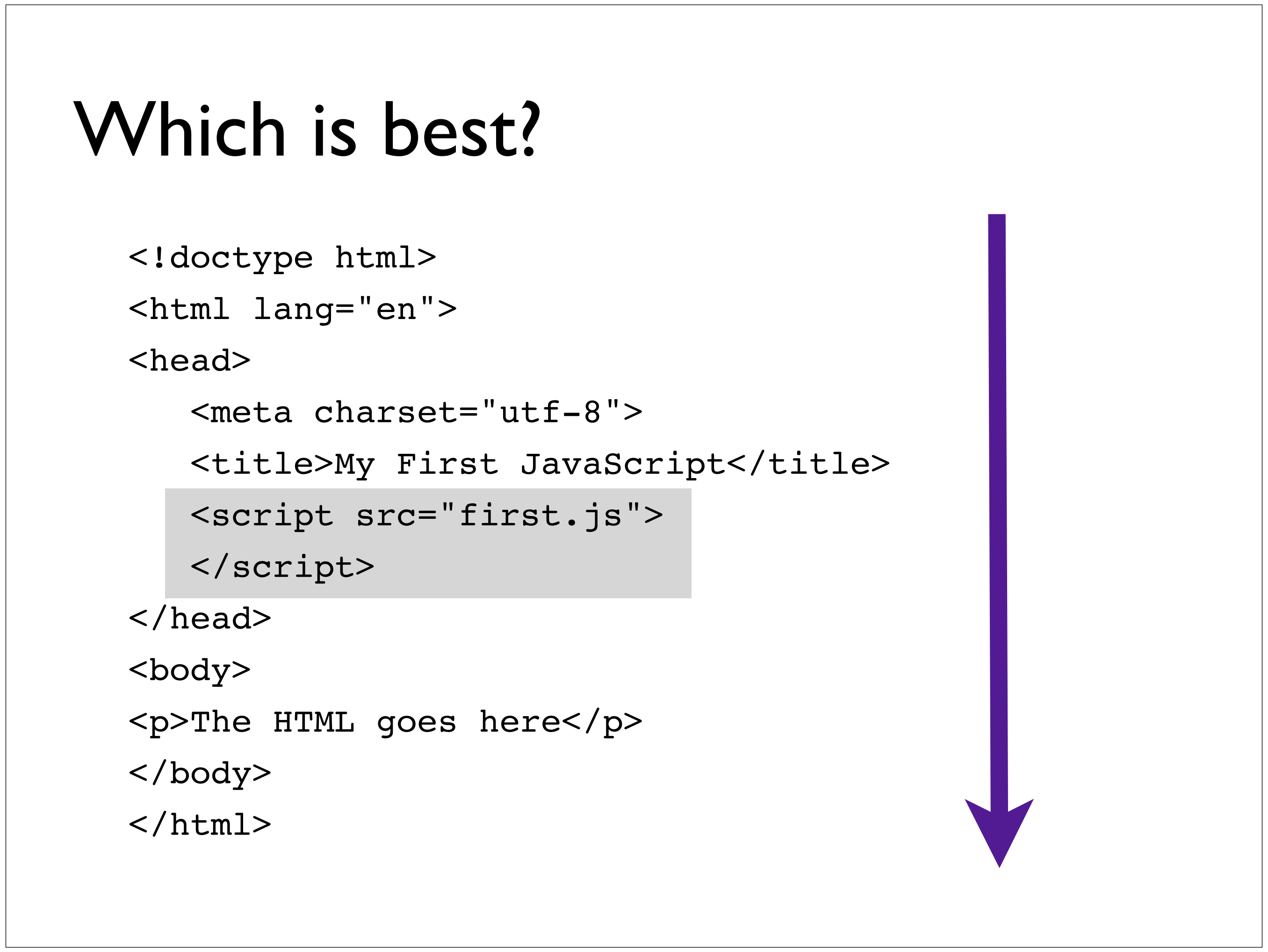
A tip about <script>...</script>

```
<!doctype html>
<html lang="en">
<head>
   <meta charset="utf-8">
   <title>My First JavaScript</title>
   <script src="first.js"></script>
   <script>
      var dogsName = "Fido";
   </script>
</head>
<body>
<p>The HTML goes here</p>
</body>
</html>
```

Do this instead.

And don't forget: code is executed top down!

Also note that the JavaScript in first.js will be executed BEFORE the JavaScript below it.
So the JavaScript in first.js can't depend on the variable dogsName.
dogsName won't be defined until AFTER the code in first.js is done executing.



Which is best?

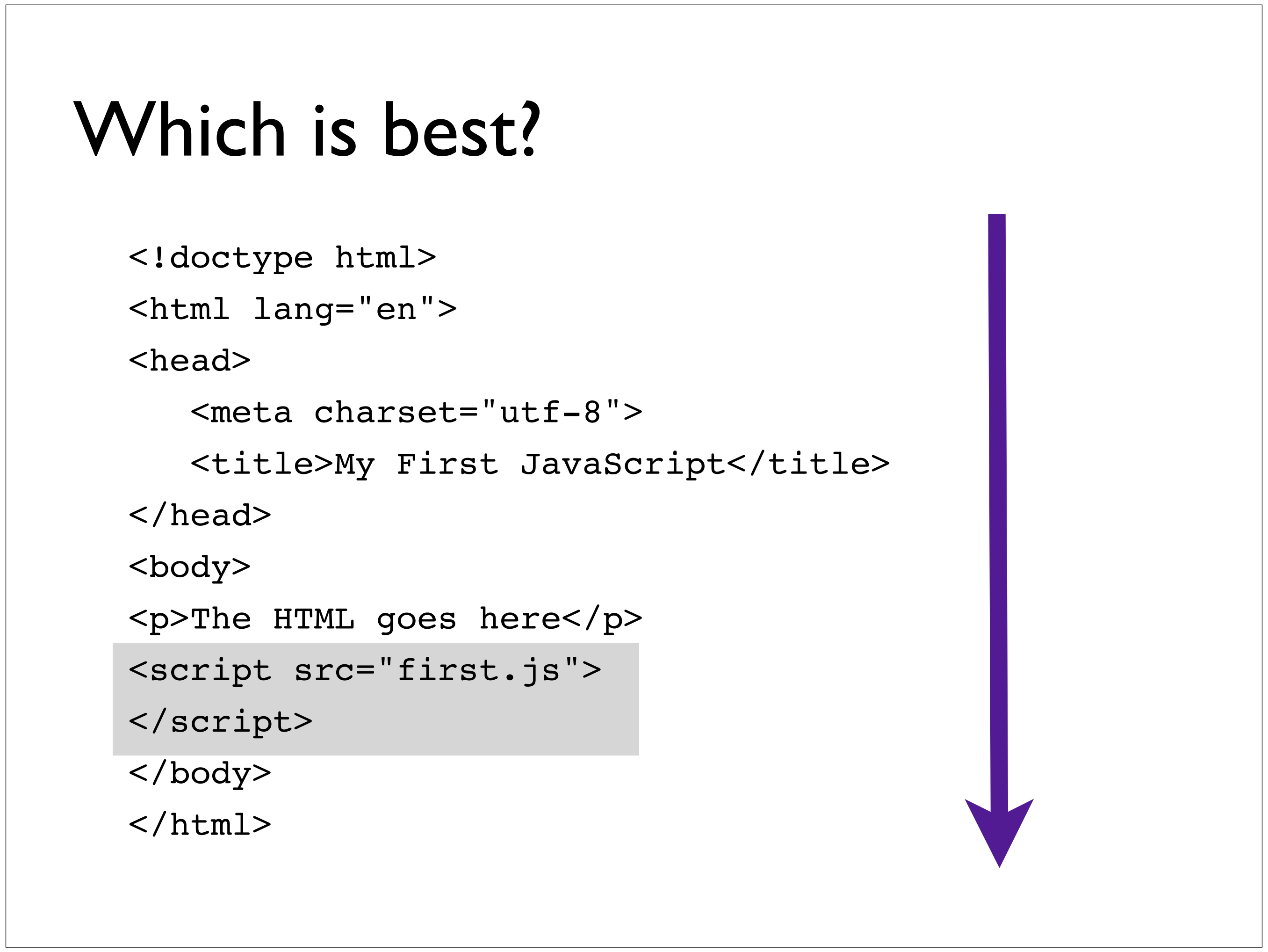
```
<!doctype html>
<html lang="en">
<head>
   <meta charset="utf-8">
   <title>My First JavaScript</title>
   <script src="first.js">
   </script>
</head>
<body>
<p>The HTML goes here</p>
</body>
</html>
```

When the browser loads the web page, it starts reading the HTML at the top.
We say it reads the page "top down". You may already be familiar with this if you've worked a lot with HTML and CSS.

If you put your script in the <head>...</head>, either by link, or directly ("inline" as we say), then your JavaScript code will be executed before the browser reads the rest of the HTML.

The upside is that you may want to initialize some values before the rest of the page loads.
The downside is that if you have a lot of JS code, the user will have to sit and wait to see the page while the browser loads (and interprets) the code.

So...



Which is best?

```
<!doctype html>
<html lang="en">
<head>
   <meta charset="utf-8">
   <title>My First JavaScript</title>
</head>
<body>
<p>The HTML goes here</p>
<script src="first.js">
</script>
</body>
</html>
```

Some people say putting the code at the bottom of the page is better because that way, the HTML loads first, and so the user has something to look at while the JS code is loading.

The downside is that if the user tries to interact with the page before it's completely loaded, partially loaded code may fail.

So it really depends on your use case... and we'll look at some strategies for how to deal with this later!

The other thing to think about is whether your code interacts with the page. If it does, then the code can't interact with the page until the page is completely loaded!
There are strategies to deal with this too. And again, more on this later.

Notice, it doesn't matter if you're linking to code in another file, or writing the code inline; the timing of when it's loaded is the same (the browser loads the code when it reaches the <script> tag, reading the page top-down).

Let's write a program to display the lyrics to 99 Bottles of Beer

Photo: https://www.flickr.com/photos/fallenpegasus/2689917493/

The HTML

bottles.html

```
<!doctype html>
<html lang="en">
<head>
   <meta charset="utf-8">
   <title>My First JavaScript</title>
   <script src="bottles.js"></script>
</head>
<body>
<h1>99 Bottles of Beer</h1>
</body>
</html>
```

"bottles.html"
Load the page
Note: 404 Not found error.

The JavaScript

```
// initialize variables
var count = 99;
var word = "bottles";

// loop over all the bottles and display the lyrics
while (count > 0) {
    console.log(count + " " + word + " of beer on the wall");
    console.log(count + " " + word + " of beer,");
    console.log("Take one down, pass it around,");
    count = count - 1;
    if (count > 0) {
       console.log(count + " " + word + " of beer on the wall.");
    }
    else {
       console.log("No more " + word + " of beer on the wall.");
    }
}
```

Making a loop

This is a while statement.

It says test to see if something is true.

```
while (SOMETHING IS TRUE) {

    DO THIS

}
```

If that something is true, then do all the statements between the curly braces

Keep doing the statements between the curly braces as long as that something is true.

Making a decision

This is an if statement.

It says test to see if something is true.

```
if (SOMETHING IS TRUE) {

    DO THIS

} else {

    DO THAT

}
```

If that something is true, then do all the statements between the curly braces

Otherwise do these statements.

The "else" part is optional!

White space and indentation

```
// initialize variables
var count = 99;
var word = "bottles";

// loop over all the bottles and display the lyrics
while (count > 0) {
    console.log(count + " " + word + " of beer on the wall");
    console.log(count + " " + word + " of beer,");
    console.log("Take one down, pass it around,");
    count = count - 1;
    if (count > 0) {
       console.log(count + " " + word + " of beer on the wall.");
    }
    else {
       console.log("No more " + word + " of beer on the wall.");
    }
}
```

Note: matching up curly braces
Note: comments

Try it

White space and indentation

```
// initialize variables
var count = 99;
var word = "bottles";

// loop over all the bottles and display the lyrics
while (count > 0) {
    console.log(count + " " + word + " of beer on the wall");
    console.log(count + " " + word + " of beer,");
    console.log("Take one down, pass it around,");
    count = count - 1;
    if (count > 0) {
       console.log(count + " " + word + " of beer on the wall.");
    }
    else {
       console.log("No more " + word + " of beer on the wall.");
    }
}
```

Note: matching up curly braces
Note: comments

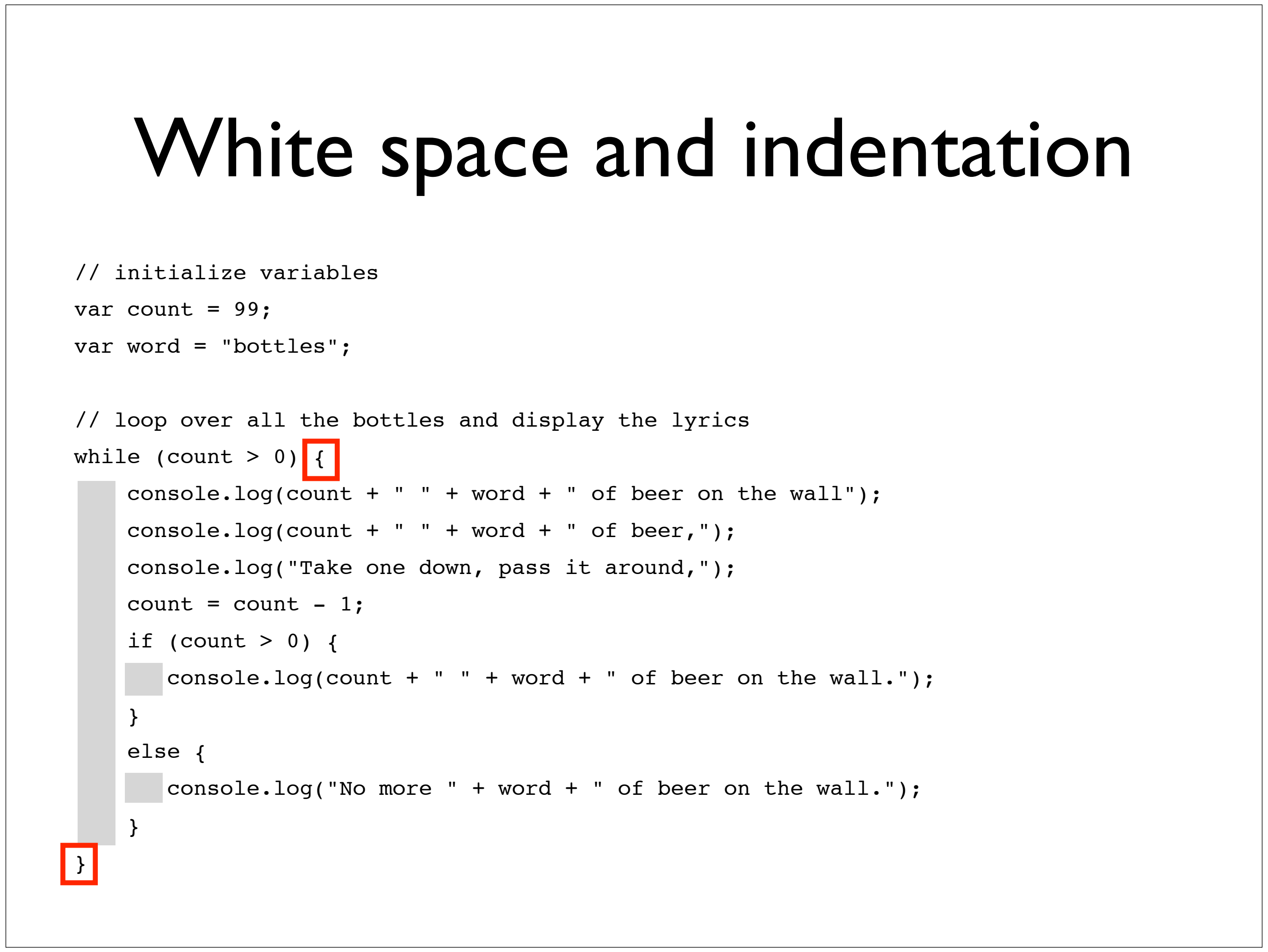
Try it

White space and indentation

```
// initialize variables
var count = 99;
var word = "bottles";

// loop over all the bottles and display the lyrics
while (count > 0) {
    console.log(count + " " + word + " of beer on the wall");
    console.log(count + " " + word + " of beer,");
    console.log("Take one down, pass it around,");
    count = count - 1;
    if (count > 0) {
       console.log(count + " " + word + " of beer on the wall.");
    }
    else {
       console.log("No more " + word + " of beer on the wall.");
    }
}
```

Note: matching up curly braces
Note: comments

Try it

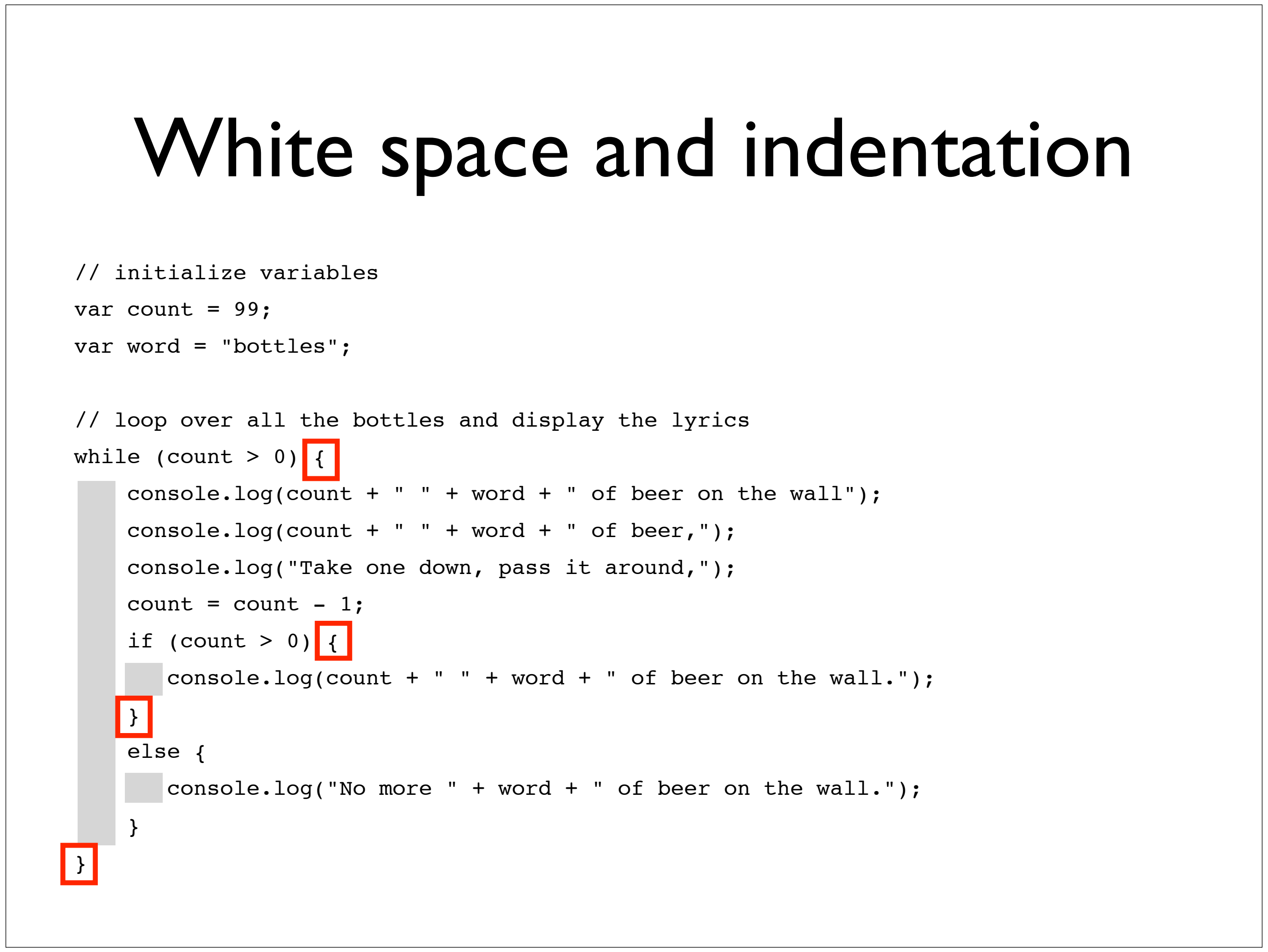


White space and indentation

```
// initialize variables
var count = 99;
var word = "bottles";

// loop over all the bottles and display the lyrics
while (count > 0) {
    console.log(count + " " + word + " of beer on the wall");
    console.log(count + " " + word + " of beer,");
    console.log("Take one down, pass it around,");
    count = count - 1;
    if (count > 0) {
       console.log(count + " " + word + " of beer on the wall.");
    }
    else {
       console.log("No more " + word + " of beer on the wall.");
    }
}
```

Note: matching up curly braces
Note: comments

Try it

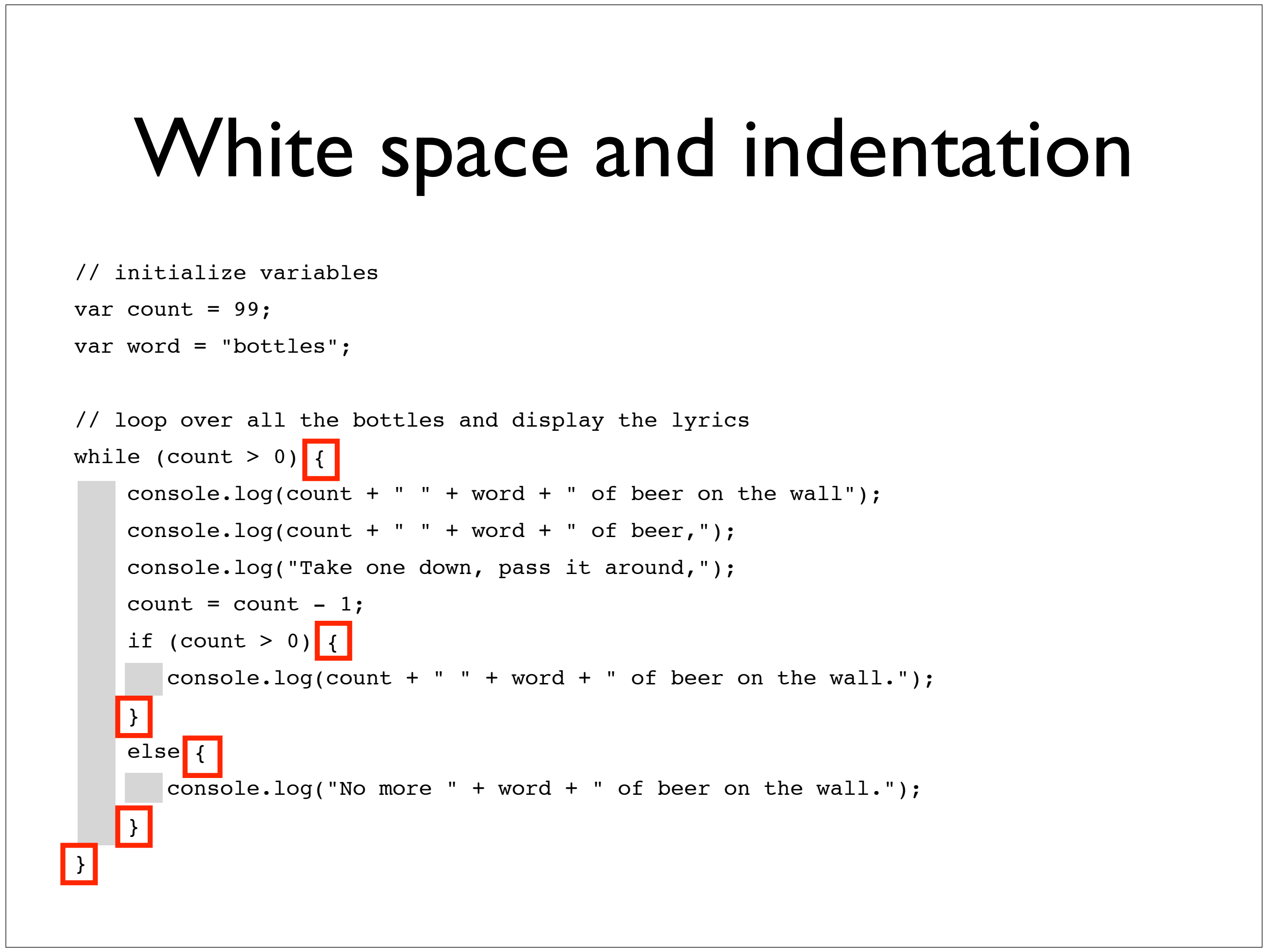


White space and indentation

```
// initialize variables
var count = 99;
var word = "bottles";

// loop over all the bottles and display the lyrics
while (count > 0) {
    console.log(count + " " + word + " of beer on the wall");
    console.log(count + " " + word + " of beer,");
    console.log("Take one down, pass it around,");
    count = count - 1;
    if (count > 0) {
       console.log(count + " " + word + " of beer on the wall.");
    }
    else {
       console.log("No more " + word + " of beer on the wall.");
    }
}
```

Note: matching up curly braces
Note: comments

Try it



White space and indentation

```
// initialize variables
var count = 99;
var word = "bottles";

// loop over all the bottles and display the lyrics
while (count > 0) {
    console.log(count + " " + word + " of beer on the wall");
    console.log(count + " " + word + " of beer,");
    console.log("Take one down, pass it around,");
    count = count - 1;
    if (count > 0) {
        console.log(count + " " + word + " of beer on the wall.");
    }
    else {
        console.log("No more " + word + " of beer on the wall.");
    }
}
```

Note: matching up curly braces
Note: comments

Try it

Fix the bug: make it say "1 bottle of beer on the wall" when there's only one bottle left.

```
// initialize variables
var count = 99;
var word = "bottles";

// loop over all the bottles and display the lyrics
while (count > 0) {
    console.log(count + " " + word + " of beer on the wall");
    console.log(count + " " + word + " of beer,");
    console.log("Take one down, pass it around,");
    count = count - 1;
    if (count > 0) {
       console.log(count + " " + word + " of beer on the wall.");
    }
    else {
       console.log("No more " + word + " of beer on the wall.");
    }
}
```

Time to play Battleship

Photo: https://www.flickr.com/photos/okchomeseller/11272953186

Simplified Battleship

0	1	2	3	4	5	6

```
var location1 = 3;
var location2 = 4;
var location3 = 5;

var guess;
var guesses = 0;

var hits = 0;

var isSunk = false;
```

Simplified Battleship

0	1	2	3	4	5	6

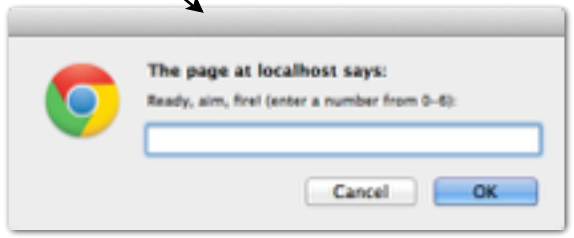
Get a guess from the user. Check to see if the guess matches a location of the ship. If it does mark that location as hit, otherwise mark it as miss. Check to see if all locations of ship are hit. If so, ship is sunk. Otherwise, ask for another guess.

The prompt function

```
var guess = prompt("Ready, aim, fire! (enter a number from 0-6):");
```

The prompt function

```
var guess = prompt("Ready, aim, fire! (enter a number from 0-6):");
```



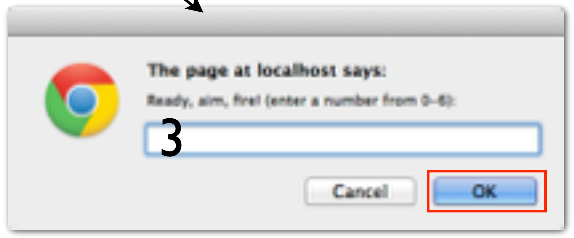
The page at localhost says:

Ready, aim, fire! (enter a number from 0-6):

guess

The prompt function

```
var guess = prompt("Ready, aim, fire! (enter a number from 0-6):");
```



The page at localhost says:

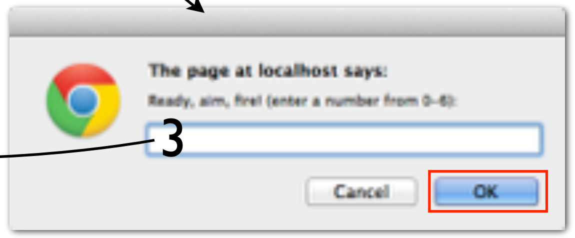
Ready, aim, fire! (enter a number from 0-6):

3

guess

The prompt function

```
var guess = prompt("Ready, aim, fire! (enter a number from 0-6):");
```



The page at localhost says:

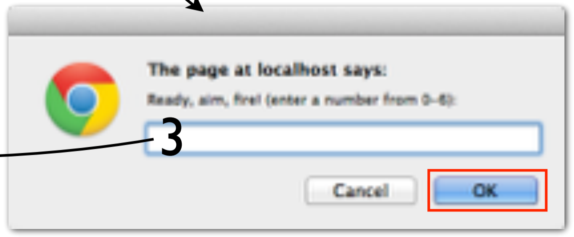
Ready, aim, fire! (enter a number from 0-6):

3

guess

The prompt function

```
var guess = prompt("Ready, aim, fire! (enter a number from 0-6):");
```



The page at localhost says:

Ready, aim, fire! (enter a number from 0-6):

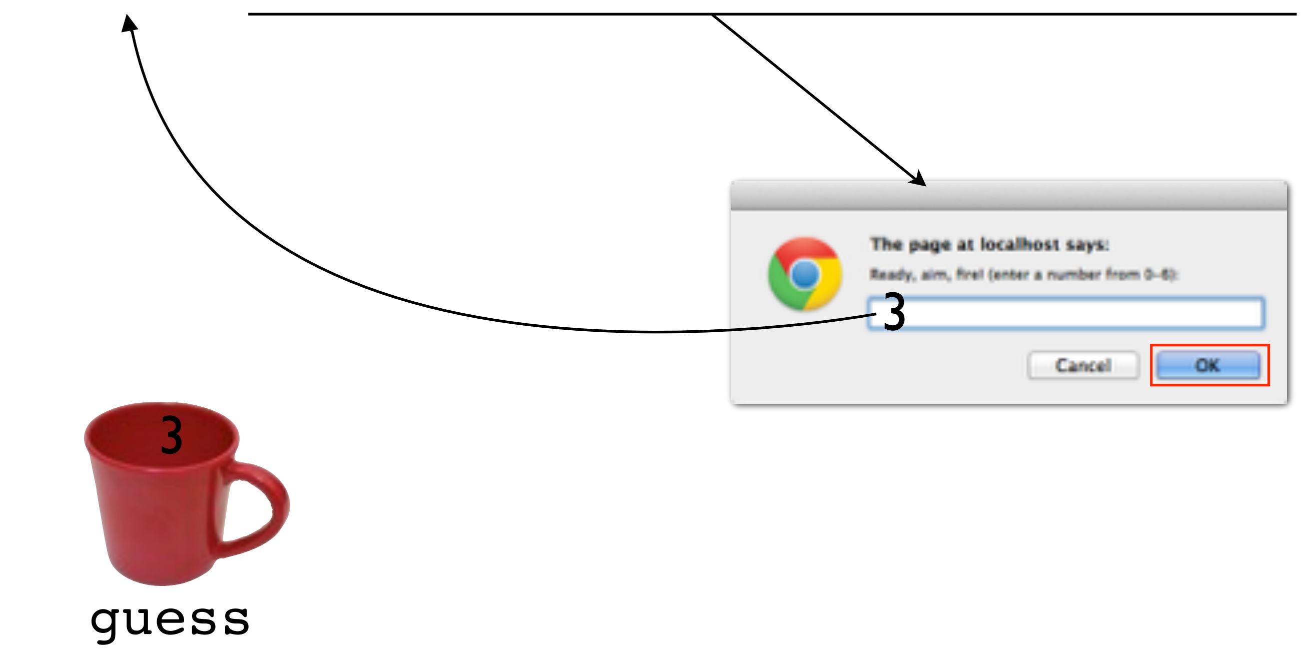
3

3

guess

The prompt function

```
var guess = prompt("Ready, aim, fire! (enter a number from 0-6):");
```


The page at localhost says:
Ready, aim, fire! (enter a number from 0-6):
3
Cancel
OK
3
guess

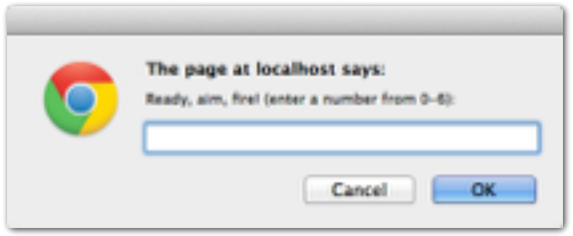
Boolean expressions

```
location1 = 3;
location2 = 4;
location3 = 5;

var guess = prompt(...);




if (guess == location1 || guess == location2 || guess == location3)
{...}
```

$$\begin{tikzcd} \text{true} \quad \text{OR} \quad \text{false} \quad \text{OR} \quad \text{false} \arrow[d] \\ \text{true} \end{tikzcd}$$

Boolean expressions

```
location1 = 3;
location2 = 4;
location3 = 5;

var guess = prompt(...);
```



The page at localhost says:

Ready, aim, fire! (enter a number from 0-6):

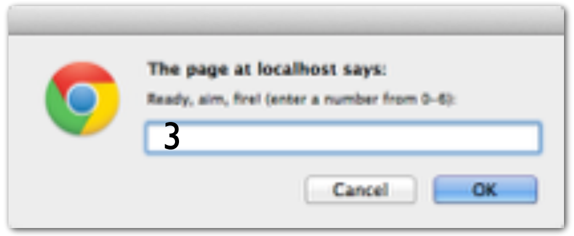
Cancel OK

```
if (guess == location1 || guess == location2 || guess == location3)
{...}
```

$$\underbrace{\text{true} \qquad \text{OR} \qquad \text{false} \qquad \text{OR} \qquad \text{false}}_{\downarrow \\ \text{true}}$$

Boolean expressions

```
location1 = 3;
location2 = 4;
location3 = 5;

var guess = prompt(...);
```



The page at localhost says:

Ready, aim, fire! (enter a number from 0-6):

3

```
if (guess == location1 || guess == location2 || guess == location3)
{...}
```

$$\underbrace{\text{true} \quad \text{OR} \quad \text{false} \quad \text{OR} \quad \text{false}}_{\downarrow}$$

true

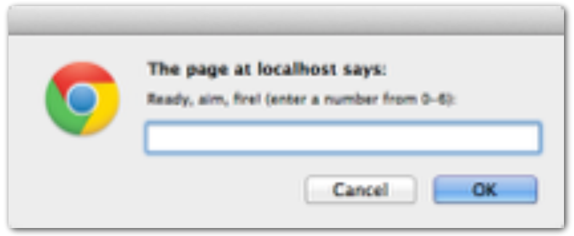
Boolean expressions

```
location1 = 3;
location2 = 4;
location3 = 5;

var guess = prompt(...);




if (guess == location1 || guess == location2 || guess == location3)
{...}
```

$$\begin{tikzcd}[column sep=small] \text{false} & \text{OR} & \text{false} \arrow[dd] & \text{OR} & \text{false} \\ & & & & \\ & & \text{false} & & \end{tikzcd}$$

Boolean expressions

```
location1 = 3;
location2 = 4;
location3 = 5;

var guess = prompt(...);
```


The page at localhost says:
Ready, aim, fire! (enter a number from 0-6):

```
if (guess == location1 || guess == location2 || guess == location3)
{...}
```

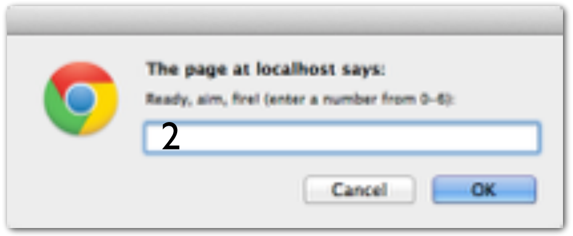
false OR false OR false

$\downarrow$

false

Boolean expressions

```
location1 = 3;
location2 = 4;
location3 = 5;

var guess = prompt(...);
```



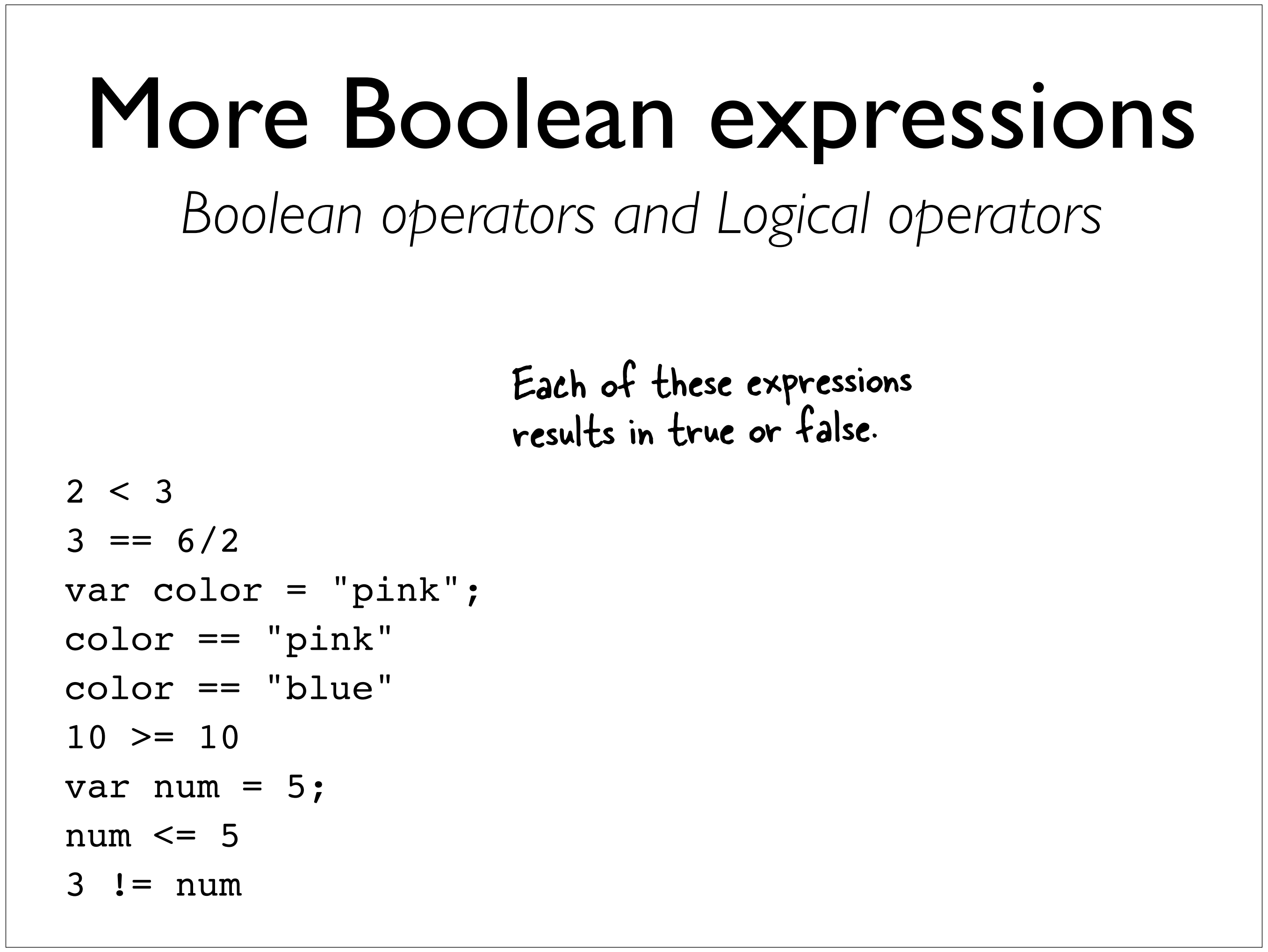
The page at localhost says:

Ready, aim, fire! (enter a number from 0-6):

2

```
if (guess == location1 || guess == location2 || guess == location3)
{...}
```

$$\begin{tikzcd} \text{false} \quad \text{OR} \quad \text{false} \quad \text{OR} \quad \text{false} \arrow[d] \\ \text{false} \end{tikzcd}$$



More Boolean expressions

Boolean operators and Logical operators

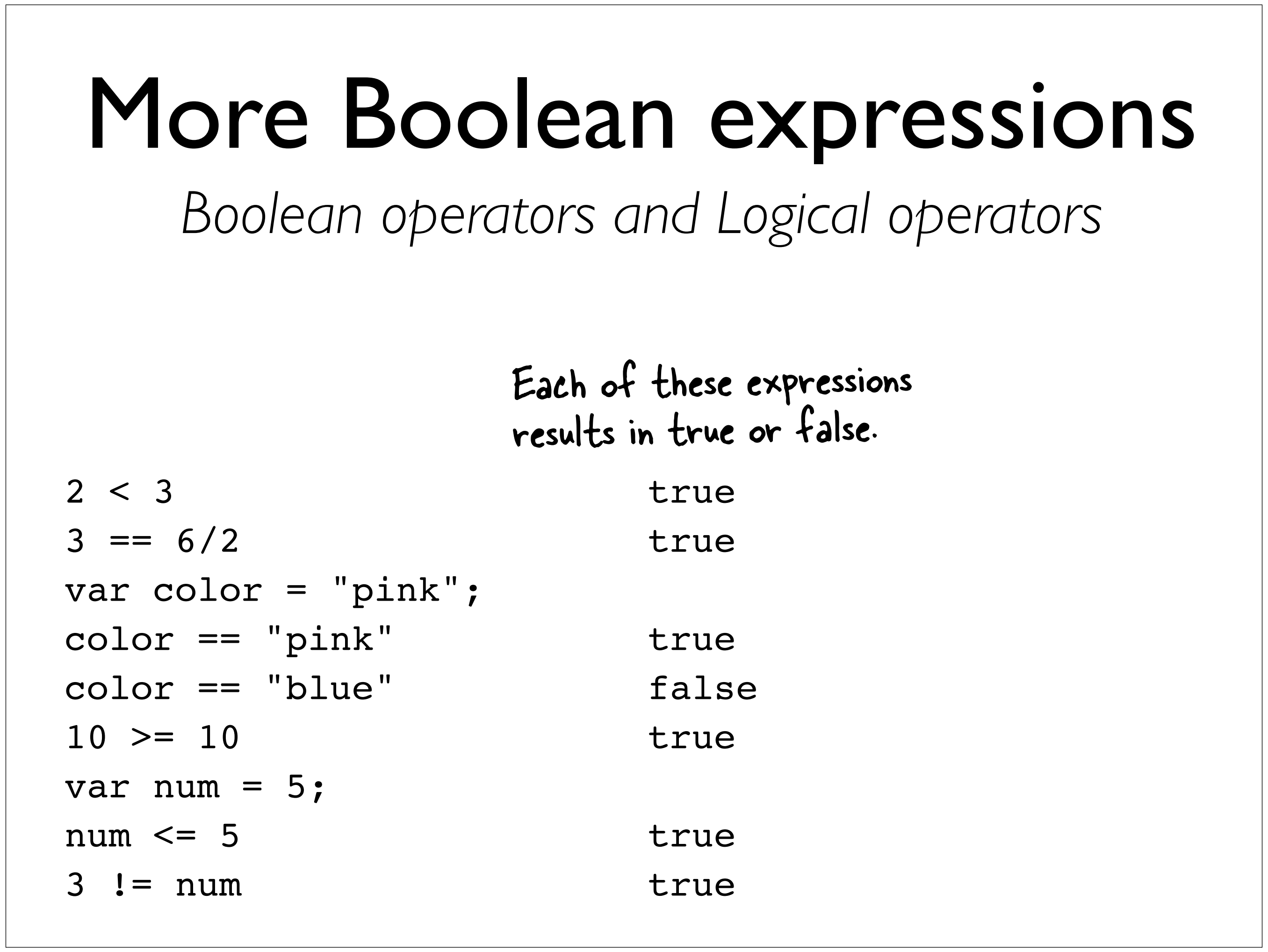
Each of these expressions results in true or false.

```
2 < 3
3 == 6/2
var color = "pink";
color == "pink"
color == "blue"
10 >= 10
var num = 5;
num <= 5
3 != num
```

A Boolean expression is named after George Boole.
Boolean operators compare things: is one thing greater than another, or one thing less than another, for instance. (You've seen these already in conditional expressions).

Notice that I use a semi-colon after a statement, and no semi-colon after an expression.
Because an expression is usually part of a statement.

Try this code in the console and determine the value of each expression.
Make sure you see why!



More Boolean expressions

Boolean operators and Logical operators

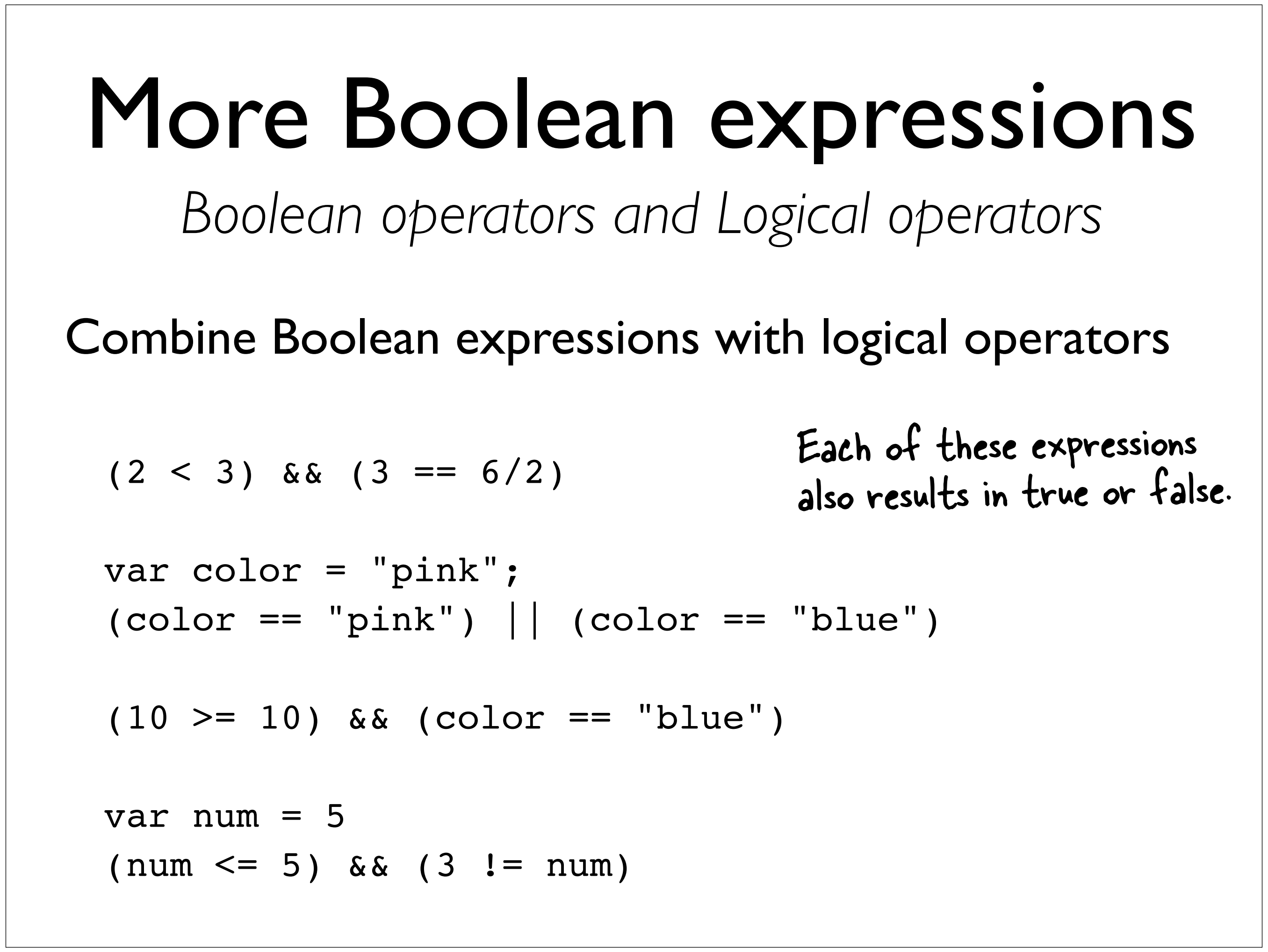
Each of these expressions results in true or false.

```
2 < 3                          true
3 == 6/2                       true
var color = "pink";
color == "pink"                true
color == "blue"                false
10 >= 10                       true
var num = 5;
num <= 5                       true
3 != num                       true
```

A Boolean expression is named after George Boole.
Boolean operators compare things: is one thing greater than another, or one thing less than another, for instance. (You've seen these already in conditional expressions).

Notice that I use a semi-colon after a statement, and no semi-colon after an expression.
Because an expression is usually part of a statement.

Try this code in the console and determine the value of each expression.
Make sure you see why!



More Boolean expressions

Boolean operators and Logical operators

Combine Boolean expressions with logical operators

```
(2 < 3) && (3 == 6/2)

var color = "pink";
(color == "pink") || (color == "blue")

(10 >= 10) && (color == "blue")

var num = 5
(num <= 5) && (3 != num)
```

Each of these expressions also results in true or false.

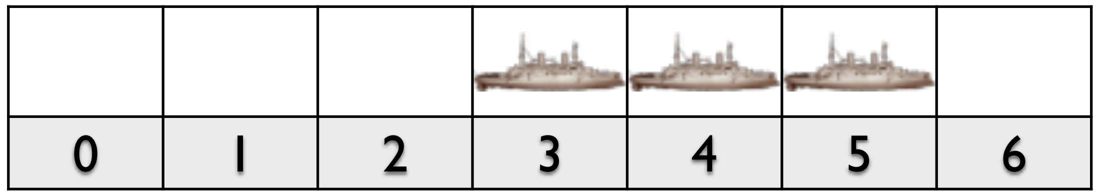
A logical operator combines true and false values.
AND – if this AND that are true then the whole expression is true
OR – if this OR that is true, then the whole expression is true

AND – if this is true AND that is false, then the whole expression is false
OR – if this is true OR that is false, then the whole expression is true
if this is false OR that is false, then the whole expression is false

See if you can figure out the answer in your head, and then
Try in the console. What is the answer for each?

```
var location1 = 3
var location2 = 4
var location3 = 5
var guess;
var hits = 0;
var guesses = 0;
var isSunk = false;

while (isSunk == false) {
    guess = prompt("Ready, aim, fire! (enter a number from 0-6):");
    if (guess < 0 || guess > 6) {
        alert("Please enter a valid cell number!");
    } else {
        guesses = guesses + 1;
        if (guess == location1 || guess == location2 || guess == location3) {
            alert("HIT!");
            hits = hits + 1;
            if (hits == 3) {
                isSunk = true;
                alert("You sank my battleship!");
            }
        } else {
            alert("MISS");
        }
    }
}
var stats = "You took " + guesses + " guesses to sink the battleship, " +
            "which means your shooting accuracy was " + (3/guesses);
alert(stats);
```

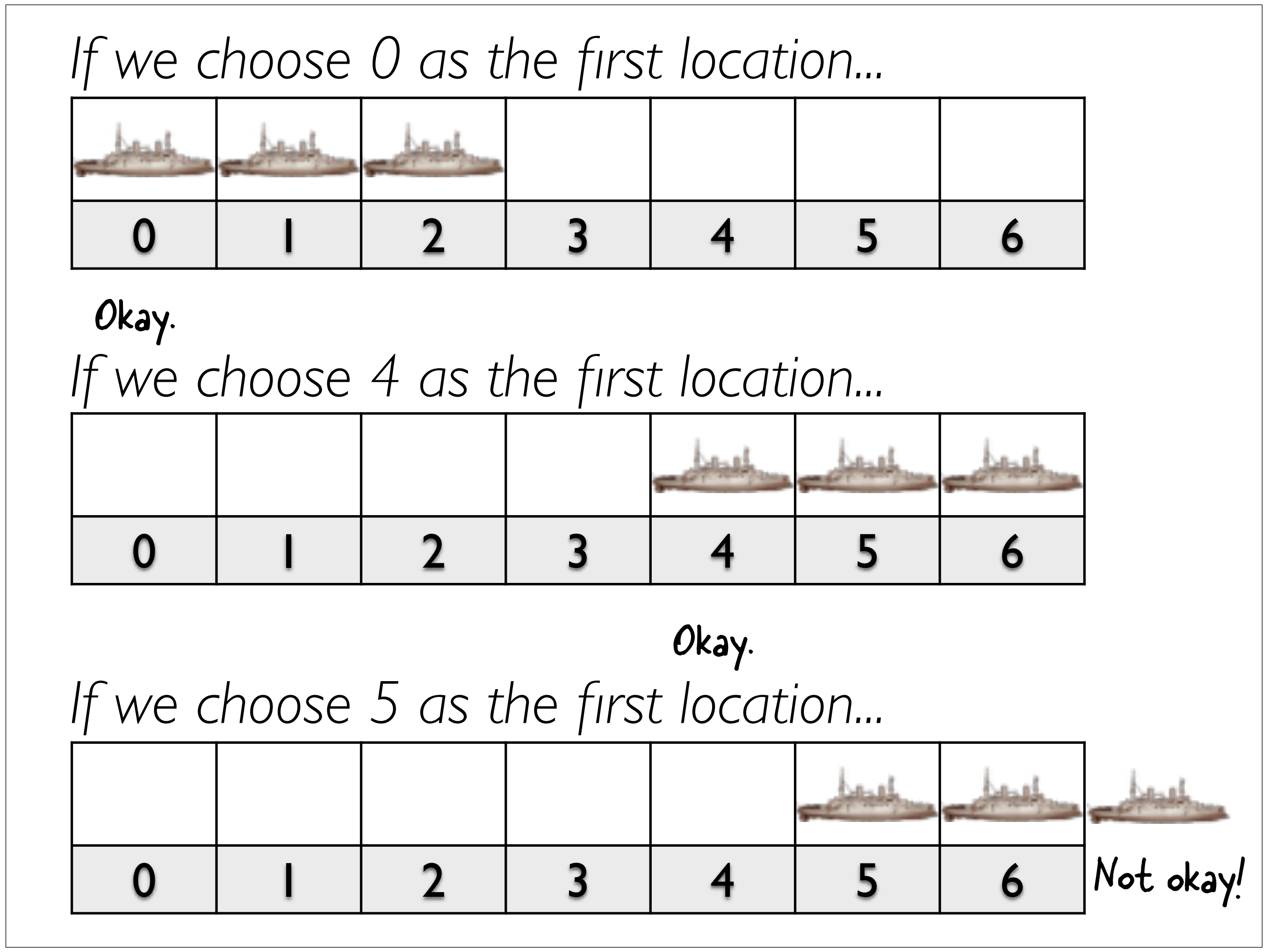
Place the ship randomly



We want to place the ship randomly on the board.

- Randomly pick the first location for the ship on the board. E.g. 3
- Then, the second location is the first location + 1
- The third location is the second location + 1

Let's randomly choose the position of the first location of the ship.


If we choose 0 as the first location...
0
1
2
3
4
5
6
Okay.
If we choose 4 as the first location...
0
1
2
3
4
5
6
Okay.
If we choose 5 as the first location...
0
1
2
3
4
5
6
Not okay!

But we need to make sure the first location of the ship is not greater than 4 (in other words it's 5 or less) so the ship doesn't fall off the end of the board.

Making a random number

```
var randomLoc = Math.random() * 5;
```

Now, randomLoc is a real number, like 3.98352

```
var randomLoc = Math.floor(randomLoc);
```

0 <= number <= 4

Now, randomLoc is an integer: 0, 1, 2, 3 or 4

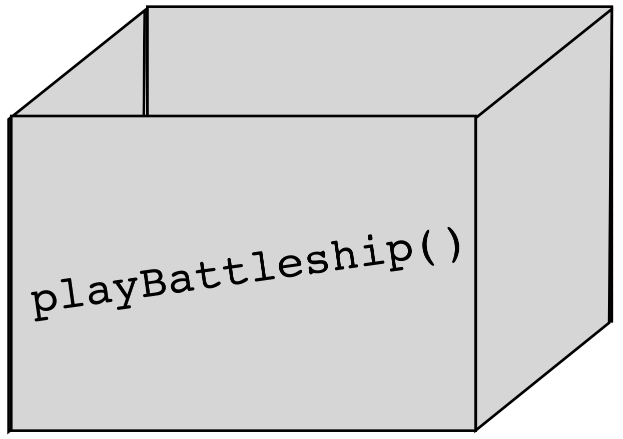
```
var randomLoc = Math.floor(Math.random() * 5);
var location1 = randomLoc;
var location2 = location1 + 1;
var location3 = location1 + 2;
var guess;
var hits = 0;
var guesses = 0;
var isSunk = false;

while (isSunk == false) {
    guess = prompt("Ready, aim, fire! (enter a number from 0-6):");
    if (guess < 0 || guess > 6) {
        alert("Please enter a valid cell number!");
    } else {
        guesses = guesses + 1;
        if (guess == location1 || guess == location2 || guess == location3) {
            alert("HIT!");
            hits = hits + 1;
            if (hits == 3) {
                isSunk = true;
                alert("You sank my battleship!");
            }
        } else {
            alert("MISS");
        }
    }
}
var stats = "You took " + guesses + " guesses to sink the battleship, " +
            "which means your shooting accuracy was " + (3/guesses);
alert(stats);
```

Also:
Best practices! Notice:
* variable declaration and initialization
* variable names
* semicolons
* braces
* indentation
* how to split up strings (at the + (concatenation) operator)
* this code should have comments! Comment your code...

NOTE: When you're playing this game, see if you can find the bug – yes, there's one left! ;-)

Make a function

A packaged up piece of code you can use over and over again.



```
playBattleship();

alert("Hi");

console.log("Hey there.");

var random = Math.random();
```

Defining and calling a function

```
function bark(name, weight) {
    if (weight > 20) {
        console.log(name + " says WOOF WOOF");
    } else {
        console.log(name + " says woof woof");
    }
}

bark("Spot", 13);
bark("Fido", 25);
```

Returning a value from a function

```
function tempFtoC(tempF) {
    var tempC = (tempF - 32) * 5/9;
    return tempC;
}

var tempInCelsius = tempFtoC(32);
console.log("Temp in celsius: ", tempInCelsius);
```

Photo: https://www.flickr.com/photos/renaissancechambara/4938639714/

In a function, parameters are just like other variables, except their values come from the arguments passed in when the function is called.

```
function calculateArea(r) {
    var area;
    if (r <= 0) {
        return 0;
    } else {
        area = Math.PI * r * r;
        return area;
    }
}

var radius = 5.2;
var theArea = calculateArea(radius);
console.log("The area is: ", theArea);
```

NOTE: r is a variable just like area
But no "var" needed.
But we do need "var" for area to declare is as a variable in the function calculateArea

JavaScript is *pass-by-value*.

That means when you call a function and pass an argument to a parameter, the value is *copied*.

```
var myAge = 29;
```

myAge contains the value 29.

```
birthday(myAge);
```

When we call birthday, we copy the value 29 into the parameter age.

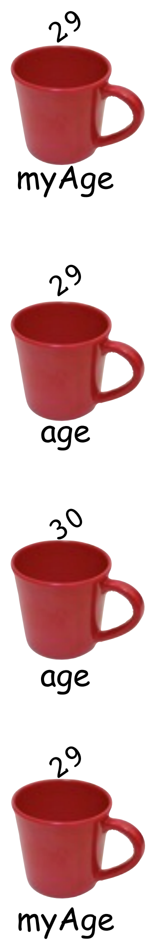
copy!

```
function birthday(age) {
    age = age + 1;
    console.log("You're now " +
            age + " years old!");
}
```

We add 1 to age.

```
console.log("Your age is: ", myAge);
```

The value of myAge hasn't changed!

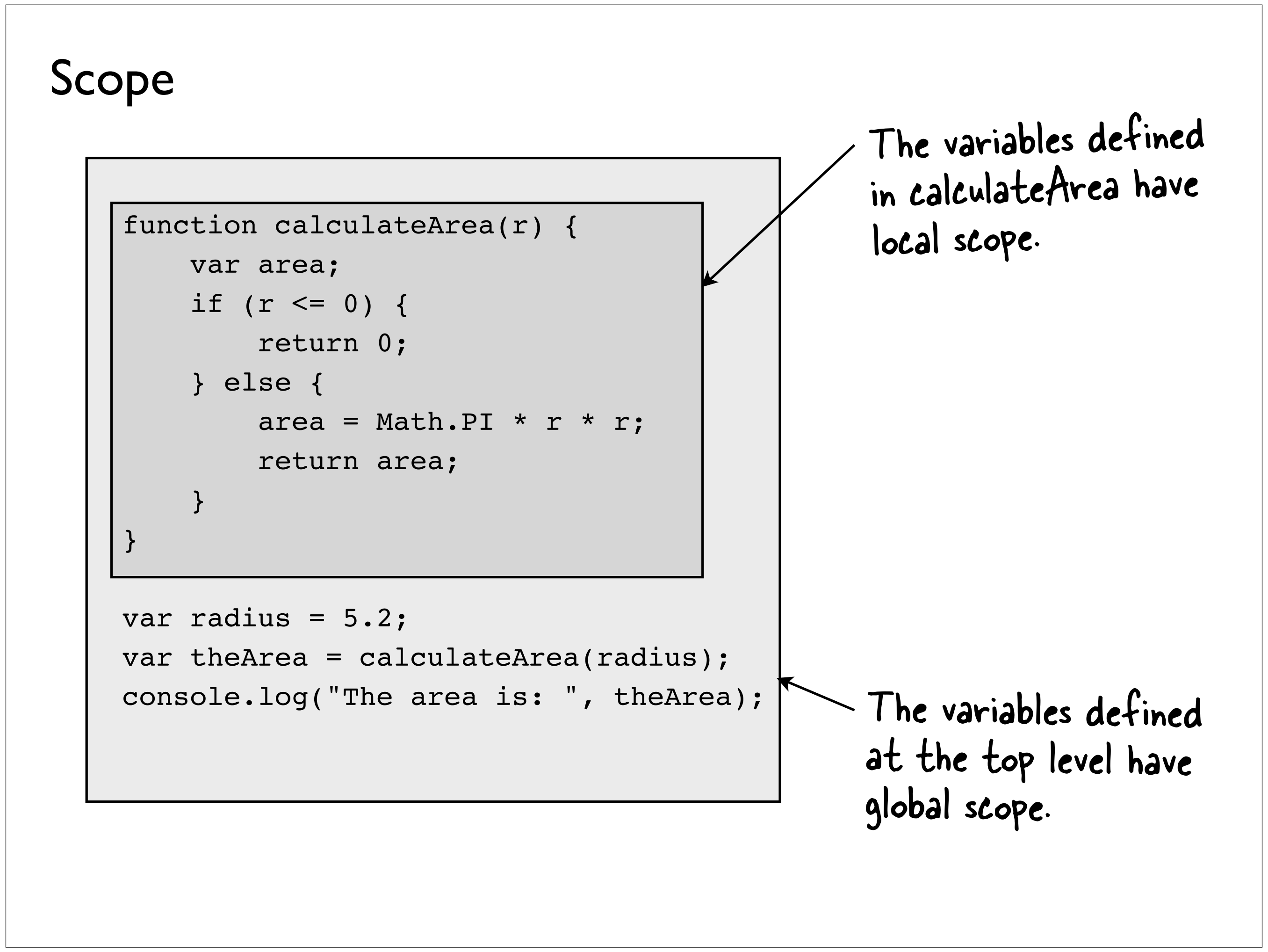


Scope describes where a variable is visible.

A *global variable* is visible everywhere in your code.

A *local variable* is visible only within its local scope.

A *function* creates a local scope.



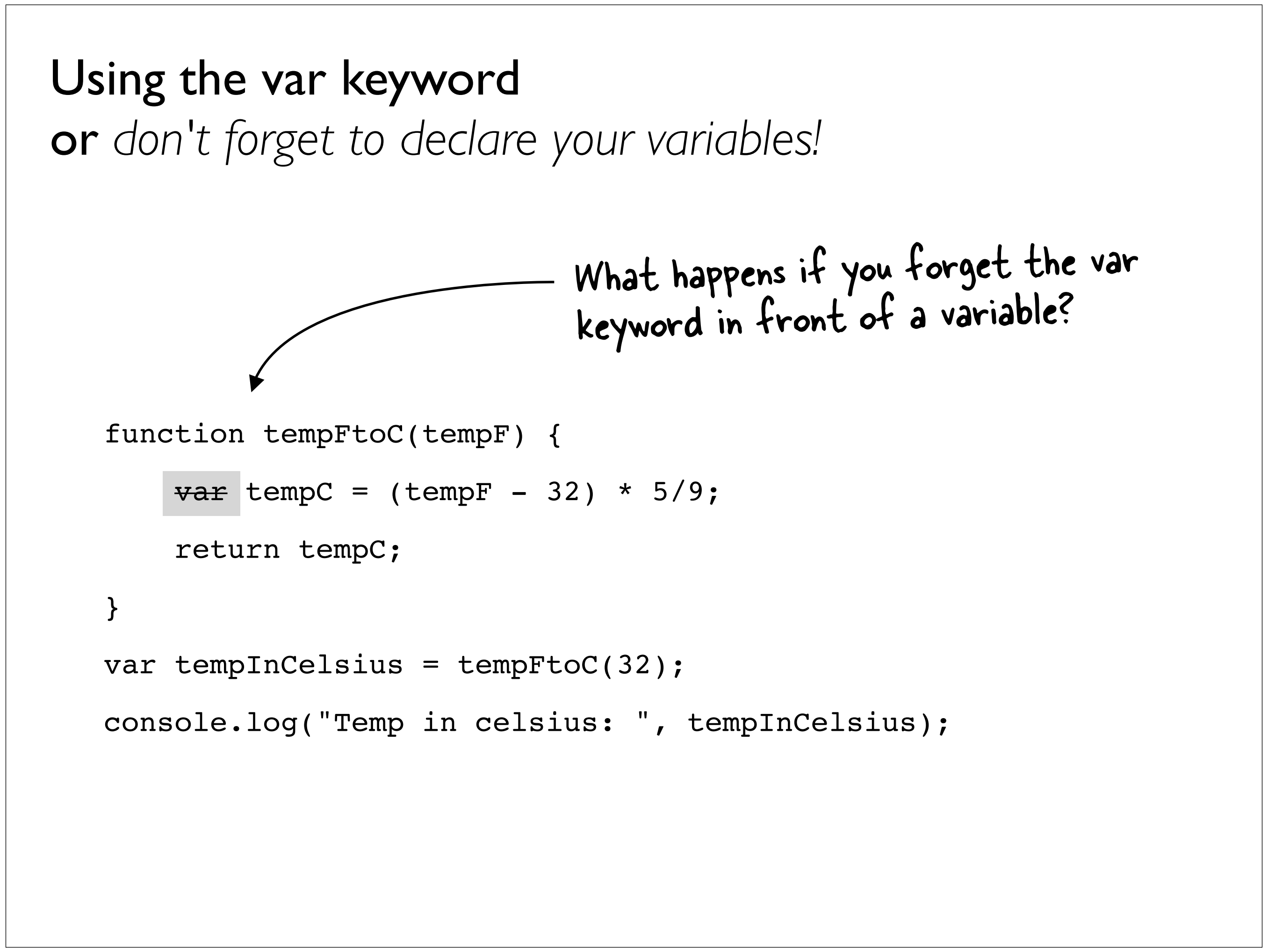
Scope

```
function calculateArea(r) {
    var area;
    if (r <= 0) {
        return 0;
    } else {
        area = Math.PI * r * r;
        return area;
    }
}

var radius = 5.2;
var theArea = calculateArea(radius);
console.log("The area is: ", theArea);
```

The variables defined in calculateArea have local scope.

The variables defined at the top level have global scope.

NOTE: radius and theArea are global variables
NOTE: r and area are local variables
NOTE: Math.PI (comes with JavaScript) is a global constant



Using the var keyword
or *don't forget to declare your variables!*

What happens if you forget the var keyword in front of a variable?

```
function tempFtoC(tempF) {
    ~~var~~ tempC = (tempF - 32) * 5/9;
    return tempC;
}
var tempInCelsius = tempFtoC(32);
console.log("Temp in celsius: ", tempInCelsius);
```

Try it and see! Remove the var keyword.

Do you get an error?

When you write:

```
function tempFtoC(tempF) {
    tempC = (tempF - 32) * 5/9;
    return tempC;
}
```

It's as if you wrote:

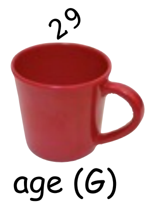
```
var tempC;
function tempFtoC(tempF) {
    tempC = (tempF - 32) * 5/9;
    return tempC;
}
```

What's the big deal if we forget the var keyword? It works...

- JavaScript has only one global "namespace". That means all global variables exist in the same space.
- That goes for external JavaScript files you link to, including libraries, like jQuery and underscore.
- So if you inadvertently create a global variable, you might overwrite the value of another global variable when you didn't mean to.
- This is known as "variable name clashing".
- So, it's better to keep your variables local.
- To make sure your variables are declared in the right place, ***always*** use the var keyword!

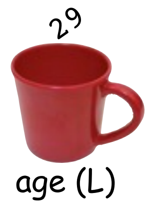
```
var age = 29;
```

age (global) contains the value 29.



```
birthday(age);
```

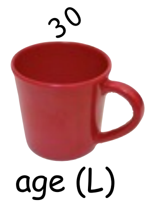
When we call birthday, we copy the value 29 into the parameter age (local). Local age is a different variable!!



copy!

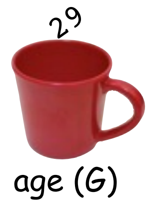
```
function birthday(age) {
    age = age + 1;
    console.log("You're now " +
            age + " years old!");
}
```

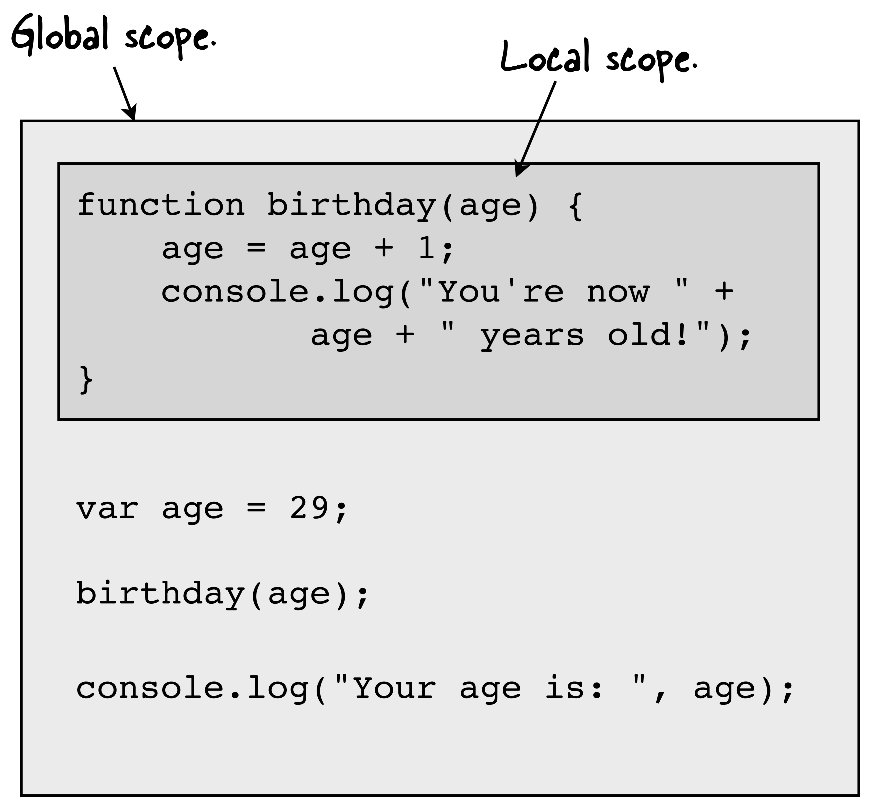
We add 1 to age (local).



```
console.log("Your age is: ", age);
```

The value of age (global) hasn't changed!



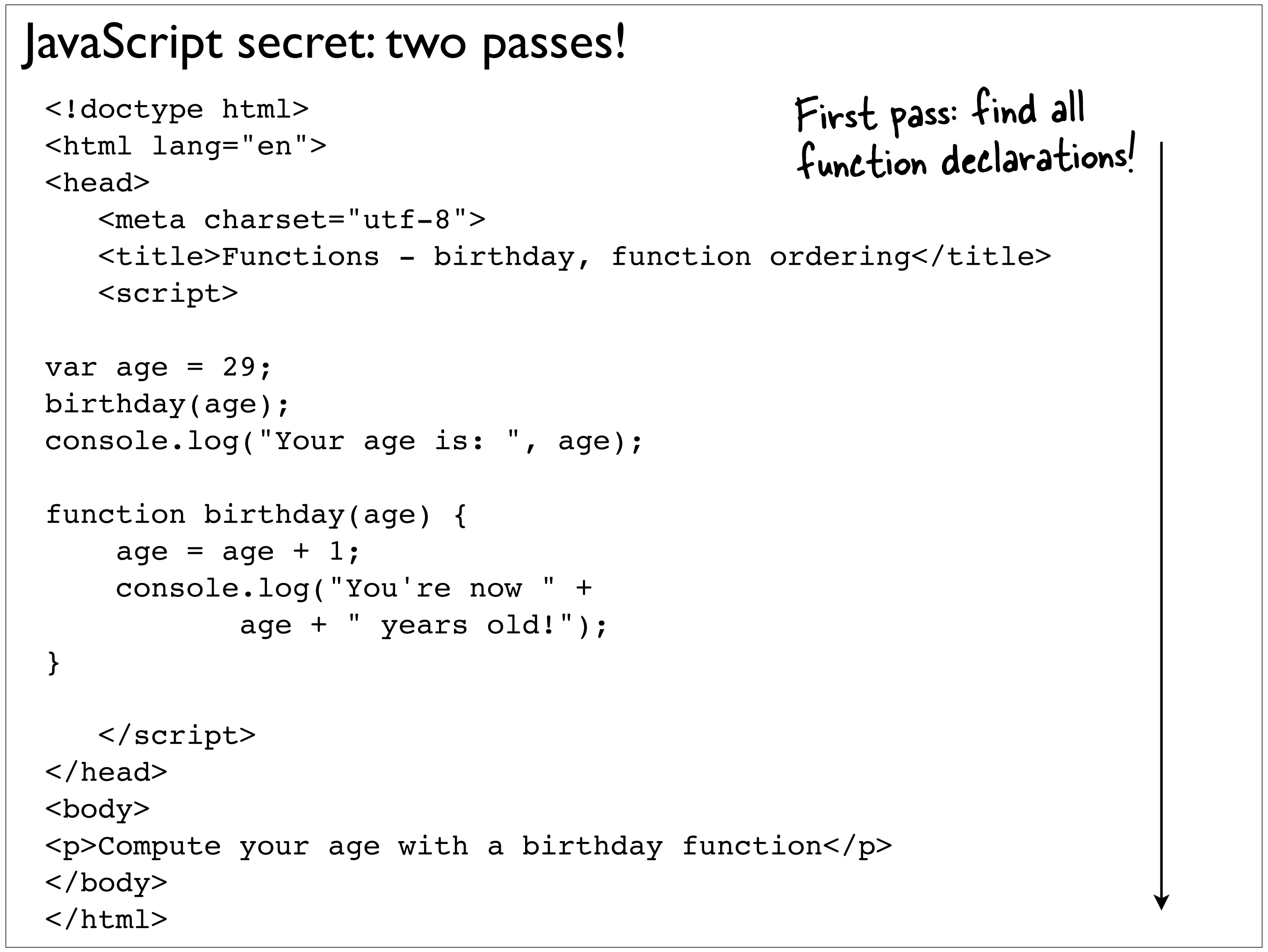


```
function birthday(age) {
    age = age + 1;
    console.log("You're now " +
           age + " years old!");
}

var age = 29;

birthday(age);

console.log("Your age is: ", age);
```

The age in birthday is a different variable from the age in the global scope.

We say the age in birthday "shadows" the global variable age because we can't "see" the global age inside birthday.

We can only "see" the local age inside birthday.



JavaScript secret: two passes!

First pass: find all function declarations!

```
<!doctype html>
<html lang="en">
<head>
   <meta charset="utf-8">
   <title>Functions - birthday, function ordering</title>
   <script>

var age = 29;
birthday(age);
console.log("Your age is: ", age);

function birthday(age) {
    age = age + 1;
    console.log("You're now " +
           age + " years old!");
}

   </script>
</head>
<body>
<p>Compute your age with a birthday function</p>
</body>
</html>
```

Here's a secret (okay, it's not really a secret! but might surprise you):

The browser actually makes TWO passes over your JavaScript code.

The first pass looks for all function declarations.
In this pass, all functions declared at the top level of your code are defined. That means the names of the functions (and what they're defined as) are now visible throughout your code (because functions defined at the top level are global).

JavaScript secret: two passes!

Second pass:
execute the code.

```
<!doctype html>
<html lang="en">
<head>
   <meta charset="utf-8">
   <title>Functions - birthday, function ordering</title>
   <script>

var age = 29;
birthday(age);
console.log("Your age is: ", age);

function birthday(age) {
    age = age + 1;
    console.log("You're now " +
           age + " years old!");
}

   </script>
</head>
<body>
<p>Compute your age with a birthday function</p>
</body>
</html>
```

In the second pass, the browser executes your JavaScript code.

So, if you call a function that is declared at the top level, the browser already knows about it, and so your function call works!

So, where should you put your functions?

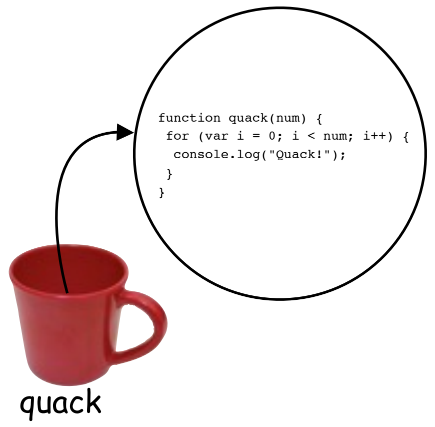
- It actually doesn't matter where you declare your top level functions, because they'll be visible everywhere.
- But most people like to group them together at the top or bottom of their code.
- As you continue to work with JavaScript you'll get a sense of how to organize your functions.
- Eventually, you might even split up functions into separate files, depending on their role.

- BUT: if you use *function expressions*, then you need to more careful! What's a function expression?

There are two ways to define functions:

function declarations

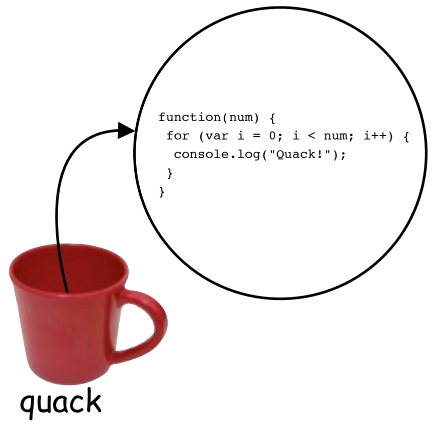
```
function quack(num) {
    for (var i = 0; i < num; i++) {
        console.log("Quack!");
    }
}
```



```
function quack(num) {
 for (var i = 0; i < num; i++) {
  console.log("Quack!");
 }
}
```

function expressions
assigned to variables

```
var quack = function quack(num) {
    for (var i = 0; i < num; i++) {
        console.log("Quack!");
    }
};
```



```
function(num) {
 for (var i = 0; i < num; i++) {
  console.log("Quack!");
 }
}
```

What's the difference? Is one better than the other?

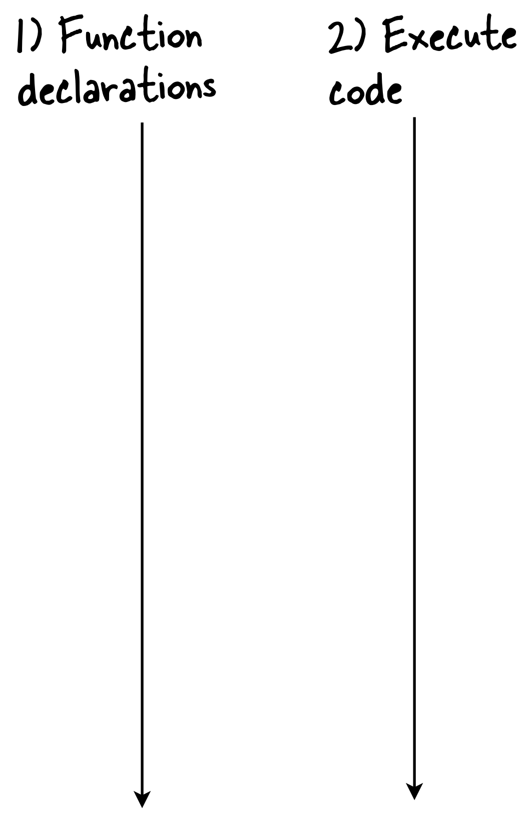
These two ways of defining functions basically accomplish the same thing:
They both create a function object and store a reference to that object in a variable, in this case, a variable named quack. That variable is created for you behind the scenes when you declare a function. You create and assign a function to the variable when you create a function with a function expression.

The main difference is WHEN the function is defined, and the SCOPE of its definition.
* (top level) function declarations are defined in the first pass, and their scope is all your code
* function expressions are assigned to variables when the code that does the assignment is executed, and the variable is defined to that function only BELOW where that assignment is done (just like any other variable assignment).

Notice one small difference: the function object made by the function declaration has a function NAME: quack. This is the same name as the variable, quack, where that function object is stored.
There is a difference between the variable name and the function name. Right now that's not important, but we'll come back to this later!!! (For now, don't worry about it at all).

Remember we said JavaScript takes two passes over the code.

```
var migrating = true;
var fly = function(num) {
    for (var i = 0; i < num; i++) {
        console.log("Flying!");
    }
}

function quack(num) {
    for (var i = 0; i < num; i++) {
        console.log("Quack!");
    }
}

if (migrating) {
    quack(4);
    fly(4);
}
```



First pass: define functions that are declared

1) Function declarations

```
var migrating = true;
var fly = function(num) {
    for (var i = 0; i < num; i++) {
        console.log("Flying!");
    }
}

function quack(num) {
    for (var i = 0; i < num; i++) {
        console.log("Quack!");
    }
}

if (migrating) {
    quack(4);
    fly(4);
}
```

Second pass execute the code

2) Execute code

```
var migrating = true;
var fly = function(num) {
    for (var i = 0; i < num; i++) {
        console.log("Flying!");
    }
}

function quack(num) {
    for (var i = 0; i < num; i++) {
        console.log("Quack!");
    }
}

if (migrating) {
    quack(4);
    fly(4);
}
```

Second pass execute the code

2) Execute code

```
var migrating = true;
var fly = function(num) {
    for (var i = 0; i < num; i++) {
        console.log("Flying!");
    }
}

function quack(num) {
    for (var i = 0; i < num; i++) {
        console.log("Quack!");
    }
}

if (migrating) {
    quack(4);
    fly(4);
}
```

Second pass execute the code

```
var migrating = true;
var fly = function(num) {
    for (var i = 0; i < num; i++) {
        console.log("Flying!");
    }
}

function quack(num) {
    for (var i = 0; i < num; i++) {
        console.log("Quack!");
    }
}

if (migrating) {
    quack(4);
    fly(4);
}
```

2) Execute code

Second pass execute the code

2) Execute code

```
var migrating = true;
var fly = function(num) {
    for (var i = 0; i < num; i++) {
        console.log("Flying!");
    }
}

function quack(num) {
    for (var i = 0; i < num; i++) {
        console.log("Quack!");
    }
}

if (migrating) {
    quack(4);
    fly(4);
}
```

Second pass execute the code

2) Execute code

```
var migrating = true;
var fly = function(num) {
    for (var i = 0; i < num; i++) {
        console.log("Flying!");
    }
}

function quack(num) {
    for (var i = 0; i < num; i++) {
        console.log("Quack!");
    }
}

if (migrating) {
    quack(4);
    fly(4);
}
```

quack was defined in the first pass.

Second pass execute the code

2) Execute code

```
var migrating = true;
var fly = function(num) {
    for (var i = 0; i < num; i++) {
        console.log("Flying!");
    }
}

function quack(num) {
    for (var i = 0; i < num; i++) {
        console.log("Quack!");
    }
}

if (migrating) {
    quack(4);
    fly(4);
}
```

fly was defined above, when we assigned the function expression to the variable fly.

Arrays

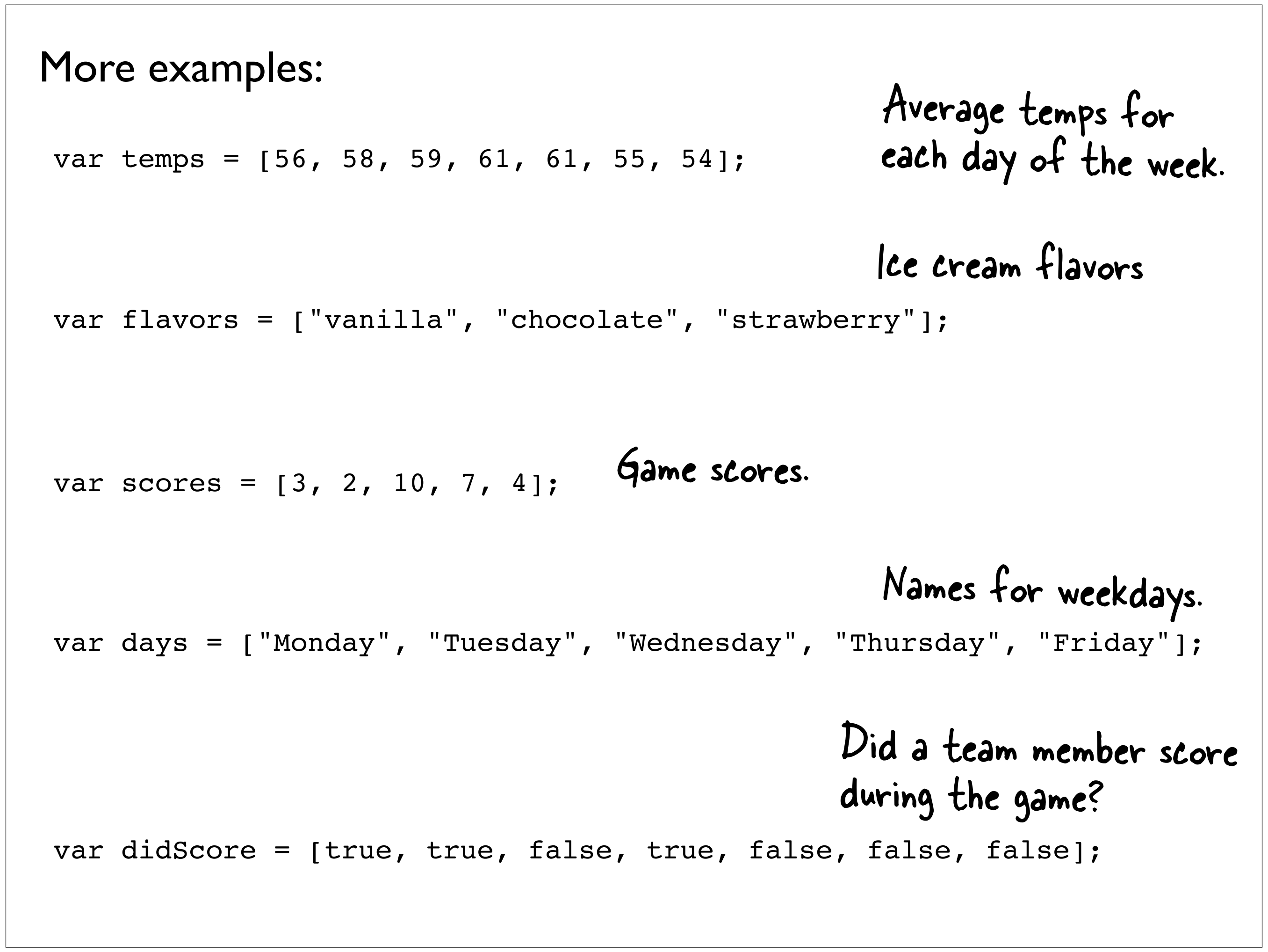
Arrays

- Arrays are for storing groups of related values. For instance, high temperatures for the week.

```
var temps = [56, 58, 59, 61, 61, 55, 54];
```

We're storing 7 values in one variable, temps.

This array has seven values, so we say its length is 7.



More examples:

```
var temps = [56, 58, 59, 61, 61, 55, 54];
```

Average temps for each day of the week.

Ice cream flavors

```
var flavors = ["vanilla", "chocolate", "strawberry"];
```

```
var scores = [3, 2, 10, 7, 4];
```

Game scores.

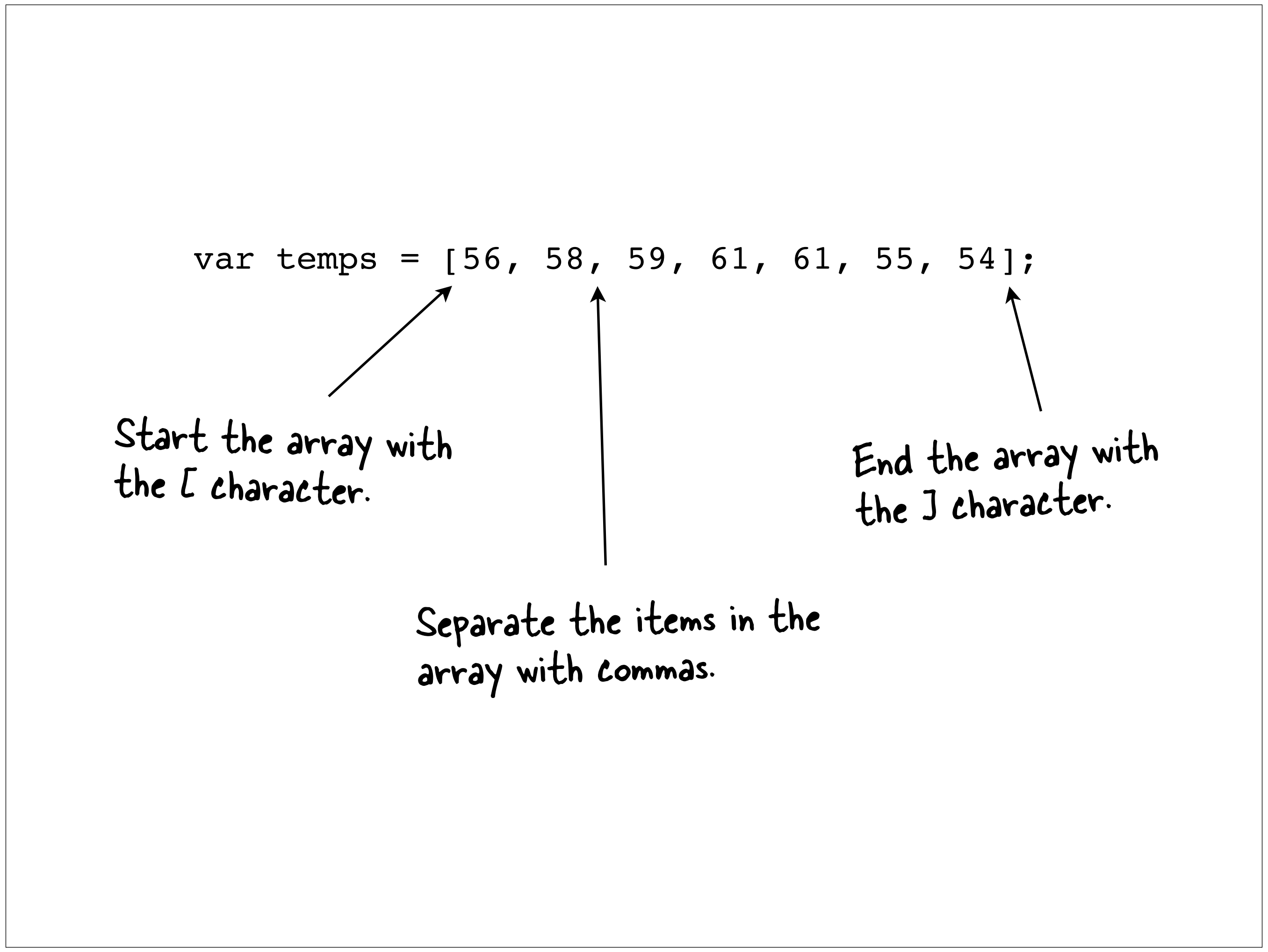
Names for weekdays.

```
var days = ["Monday", "Tuesday", "Wednesday", "Thursday", "Friday"];
```

Did a team member score during the game?

```
var didScore = [true, true, false, true, false, false, false];
```

Notice we can store numbers, strings, or booleans (the primitives we learned about before) in an array.

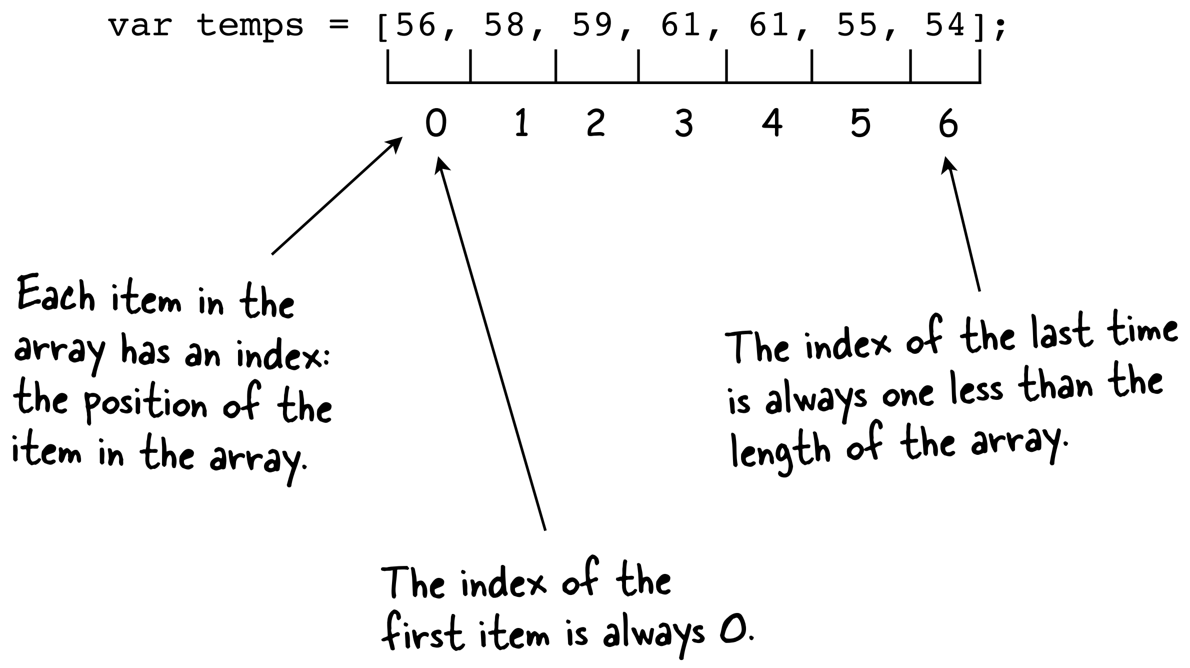


```
var temps = [56, 58, 59, 61, 61, 55, 54];
```

Start the array with the [character.

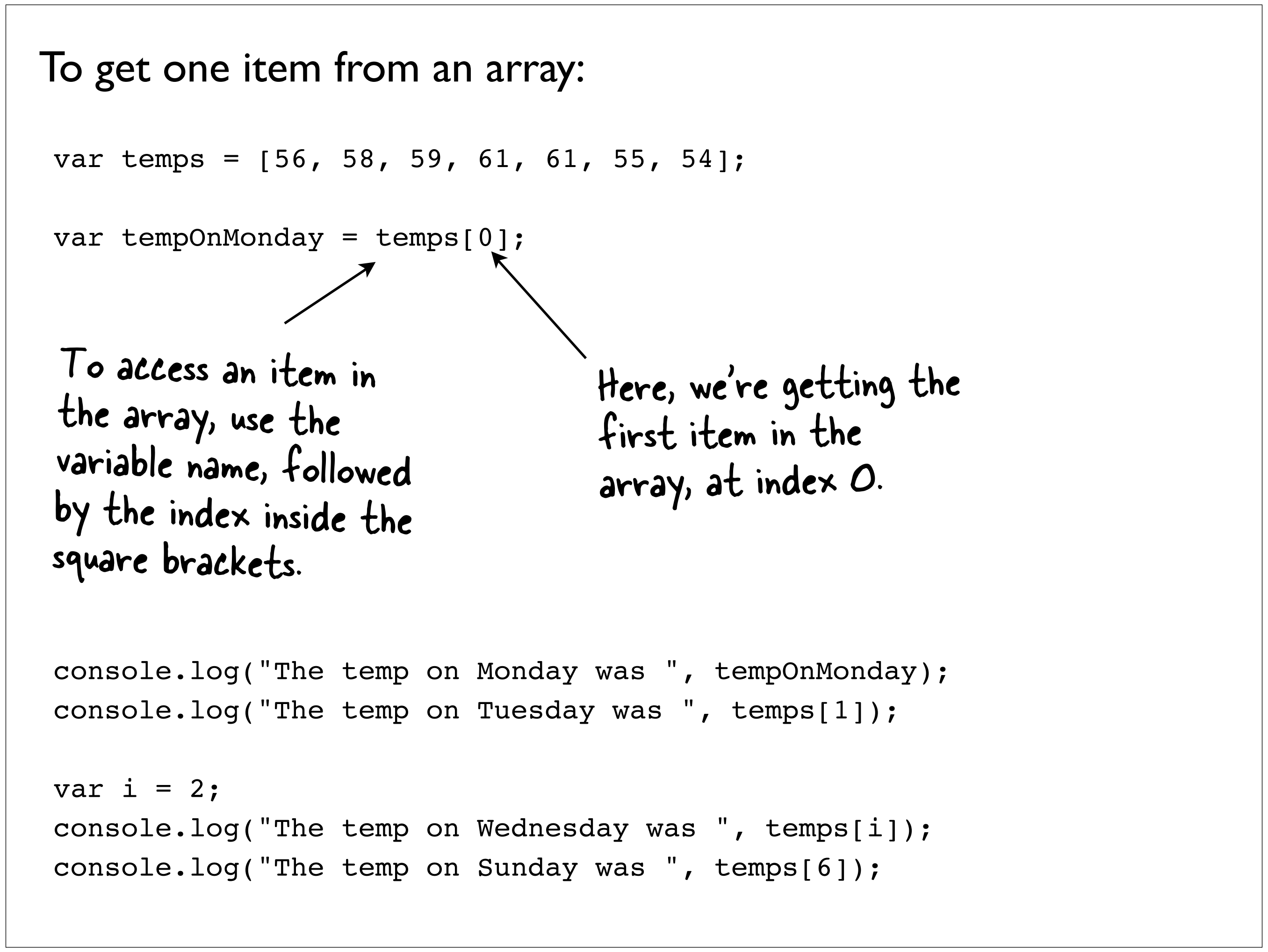
Separate the items in the array with commas.

End the array with the] character.



```
var temps = [56, 58, 59, 61, 61, 55, 54];
```

0 1 2 3 4 5 6
Each item in the array has an index: the position of the item in the array.
The index of the first item is always 0.
The index of the last time is always one less than the length of the array.



To get one item from an array:

```
var temps = [56, 58, 59, 61, 61, 55, 54];

var tempOnMonday = temps[0];
```

To access an item in the array, use the variable name, followed by the index inside the square brackets.

Here, we're getting the first item in the array, at index 0.

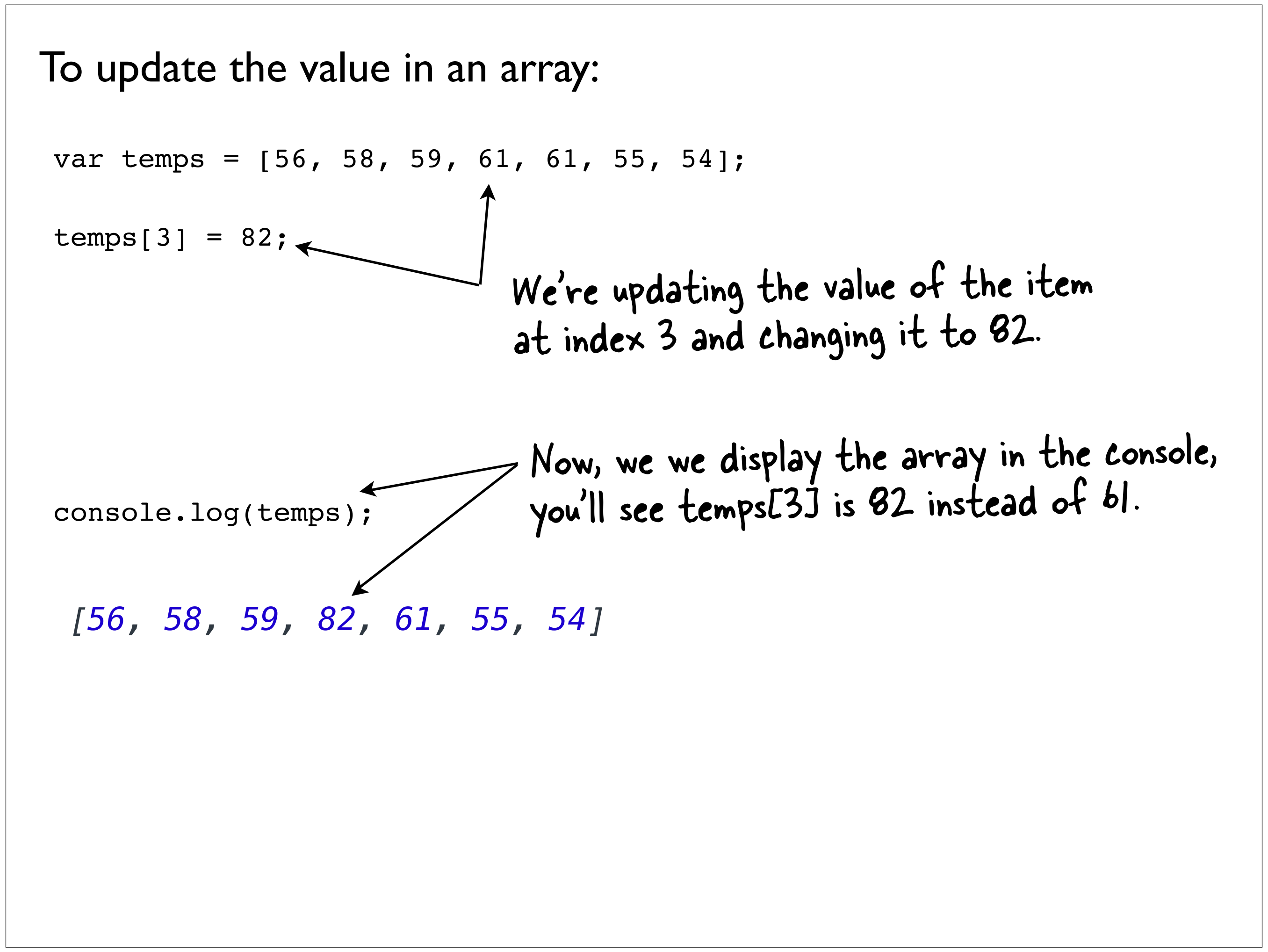
```
console.log("The temp on Monday was ", tempOnMonday);
console.log("The temp on Tuesday was ", temps[1]);

var i = 2;
console.log("The temp on Wednesday was ", temps[i]);
console.log("The temp on Sunday was ", temps[6]);
```

Try this now:
Type in an array into the console, and try accessing different values.

Notice we don't have to store a value from an array in a variable before using it;
we can use the value directly just by using the syntax to access an item in the array.

Notice that we can use a variable in place of a number for the index.

You may hear the following terms for values in an array: an array "item", an array "element".
They both mean the the same thing.



To update the value in an array:

```
var temps = [56, 58, 59, 61, 61, 55, 54];

temps[3] = 82;
```

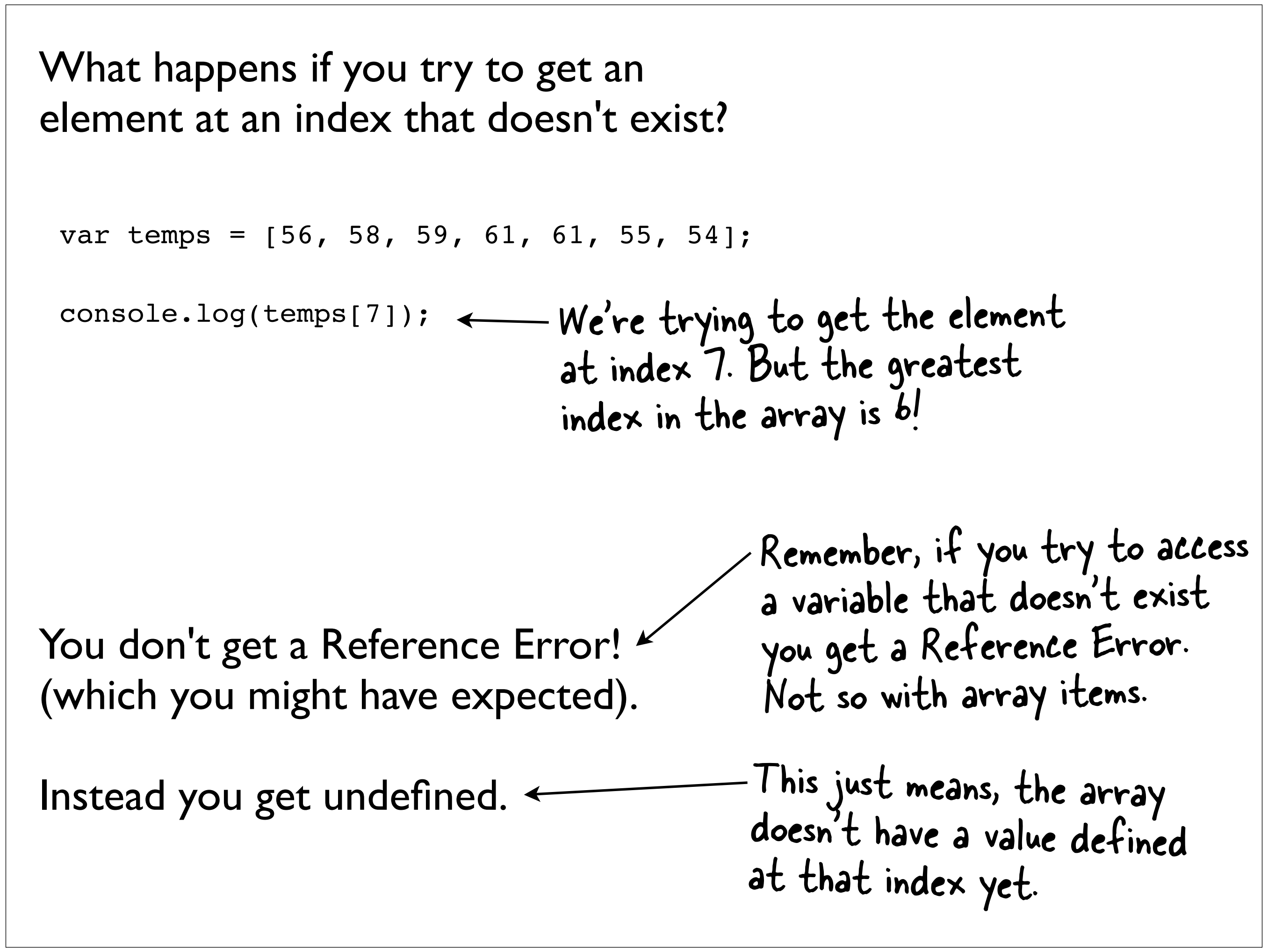
We're updating the value of the item at index 3 and changing it to 82.

```
console.log(temps);
```

Now, we we display the array in the console, you'll see temps[3] is 82 instead of 61.

```
[56, 58, 59, 82, 61, 55, 54]
```

Try changing other values in the array.
Try changing every value in the array.



What happens if you try to get an element at an index that doesn't exist?

```
var temps = [56, 58, 59, 61, 61, 55, 54];

console.log(temps[7]);
```

We're trying to get the element at index 7. But the greatest index in the array is 6!

Remember, if you try to access a variable that doesn't exist you get a Reference Error. Not so with array items.

You don't get a Reference Error! (which you might have expected).

Instead you get undefined.

This just means, the array doesn't have a value defined at that index yet.

This behavior might surprise you if you're coming from a language like Java where arrays have pre-defined lengths, and you get an access or reference error if you try to access an item at an index that doesn't exist.

JavaScript arrays are dynamic: we can add new values at any time!

```
var flavors = ["vanilla", "chocolate", "strawberry"];

console.log("Ice cream flavors: ", flavors);

flavors[3] = "pistachio";

console.log("Ice cream flavors: ", flavors);

flavors[4] = "lemon";

console.log("Ice cream flavors: ", flavors);
```

See how the new items are added on the end?

```
Ice cream flavors:  ["vanilla", "chocolate", "strawberry"]
Ice cream flavors:  ["vanilla", "chocolate", "strawberry", "pistachio"]
Ice cream flavors:  ["vanilla", "chocolate", "strawberry", "pistachio", "lemon"]
```

So, how do you know how long your array is at any given time?

```
var flavors = ["vanilla", "chocolate", "strawberry"];

console.log("length: ", flavors.length);
```

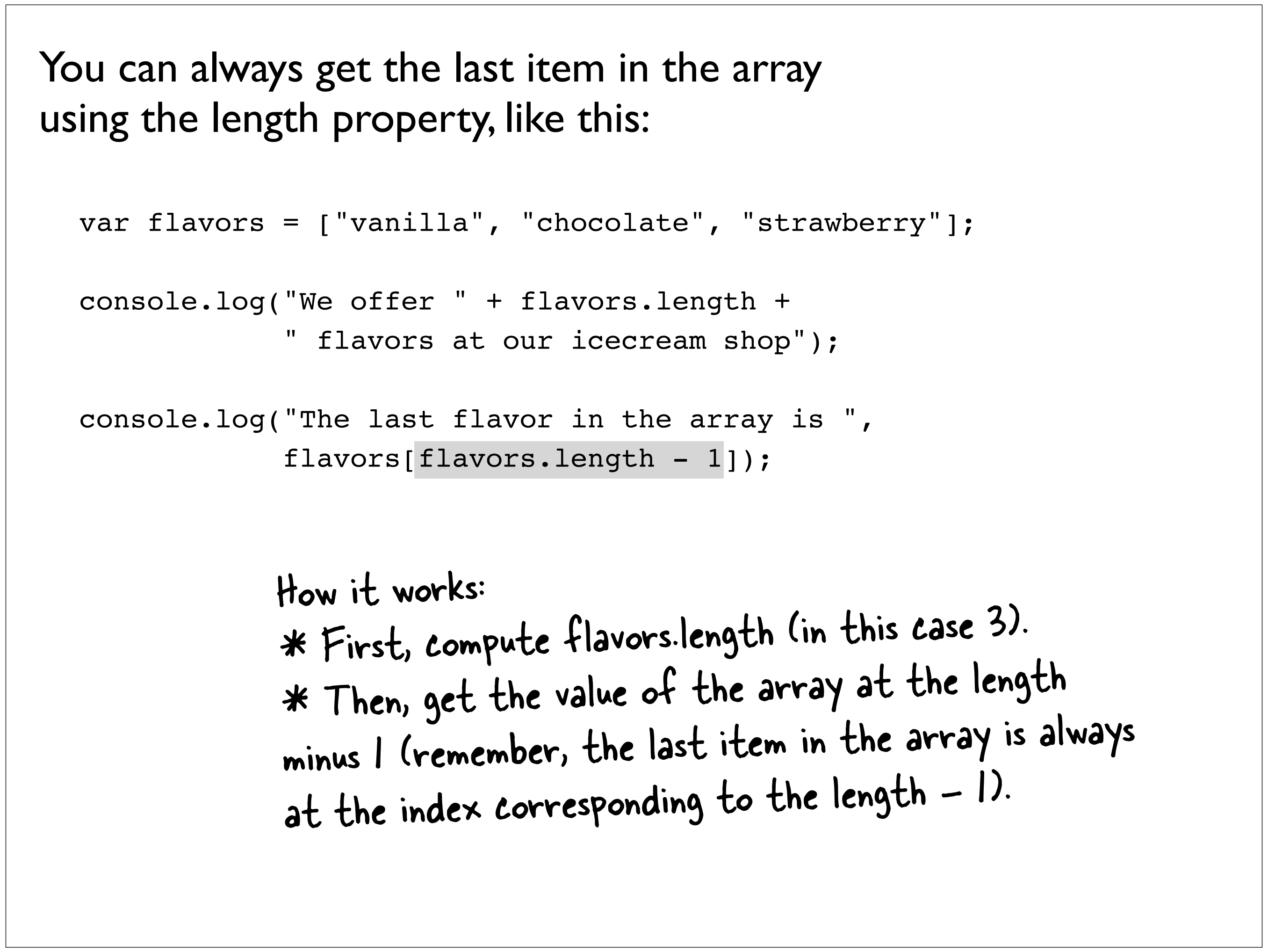
Every array has a property, length, that tells you the length of the array.

```
length:  3
```

The length of the array grows if you add new items to it, so the length property changes:

```
var flavors = ["vanilla", "chocolate", "strawberry"];

console.log("length: ", flavors.length);

flavors[3] = "pistachio";

console.log("length: ", flavors.length);

flavors[4] = "lemon";

console.log("length: ", flavors.length);
```

```
length:  3
length:  4
length:  5
```



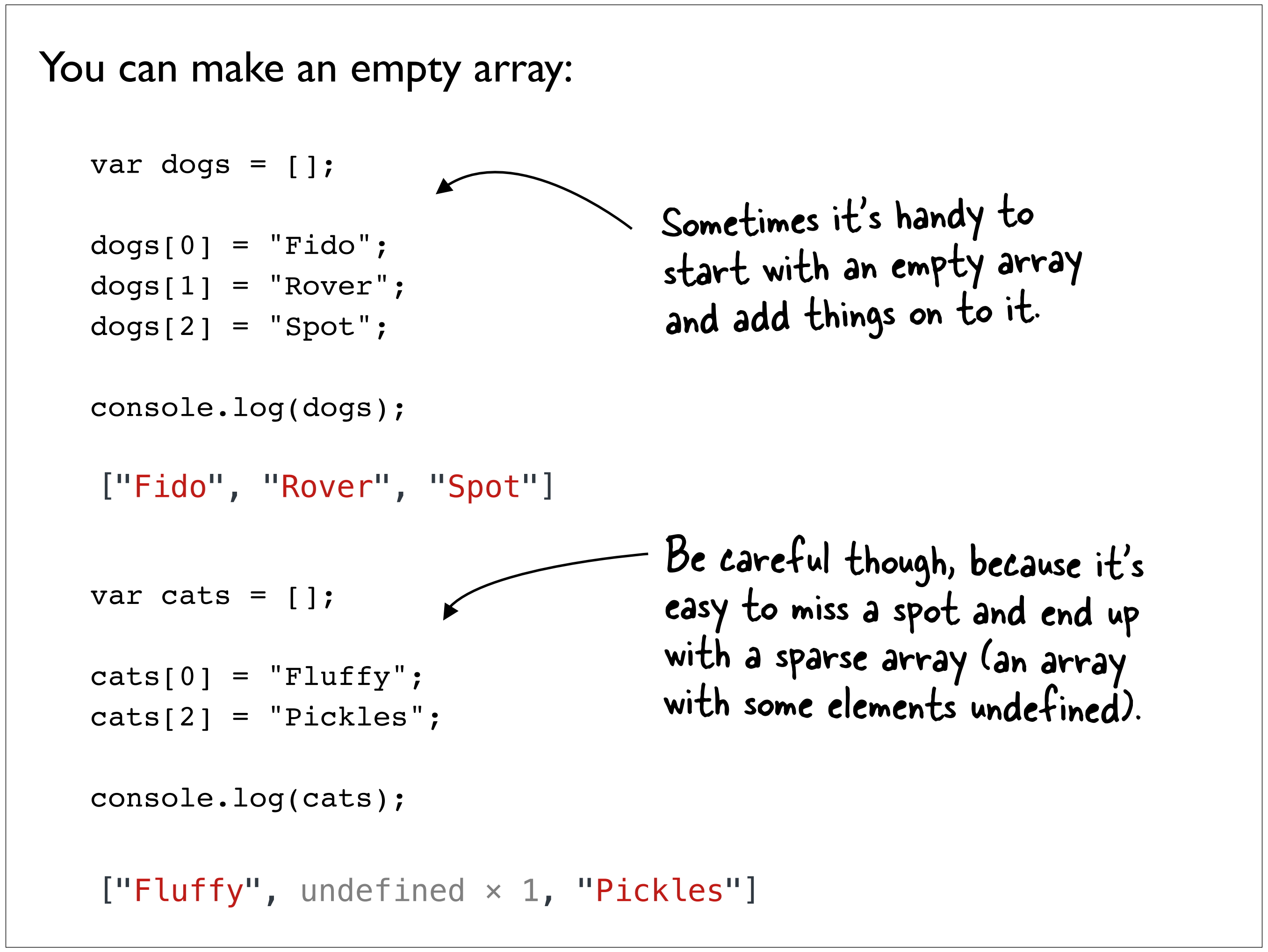
You can always get the last item in the array using the length property, like this:

```
var flavors = ["vanilla", "chocolate", "strawberry"];

console.log("We offer " + flavors.length +
            " flavors at our icecream shop");

console.log("The last flavor in the array is ",
            flavors[flavors.length - 1]);
```

How it works:
* First, compute flavors.length (in this case 3).
* Then, get the value of the array at the length minus 1 (remember, the last item in the array is always at the index corresponding to the length – 1).

Type this in and make sure you understand why we use flavors.length – 1!

Notice that flavors.length – 1 is an expression.
The expression is evaluated, and the result is 2
The value 2 is used to access the array at index 2: flavors[2]



You can make an empty array:

```
var dogs = [];

dogs[0] = "Fido";
dogs[1] = "Rover";
dogs[2] = "Spot";

console.log(dogs);

["Fido", "Rover", "Spot"]
```

Sometimes it's handy to start with an empty array and add things on to it.

```
var cats = [];

cats[0] = "Fluffy";
cats[2] = "Pickles";

console.log(cats);

["Fluffy", undefined × 1, "Pickles"]
```

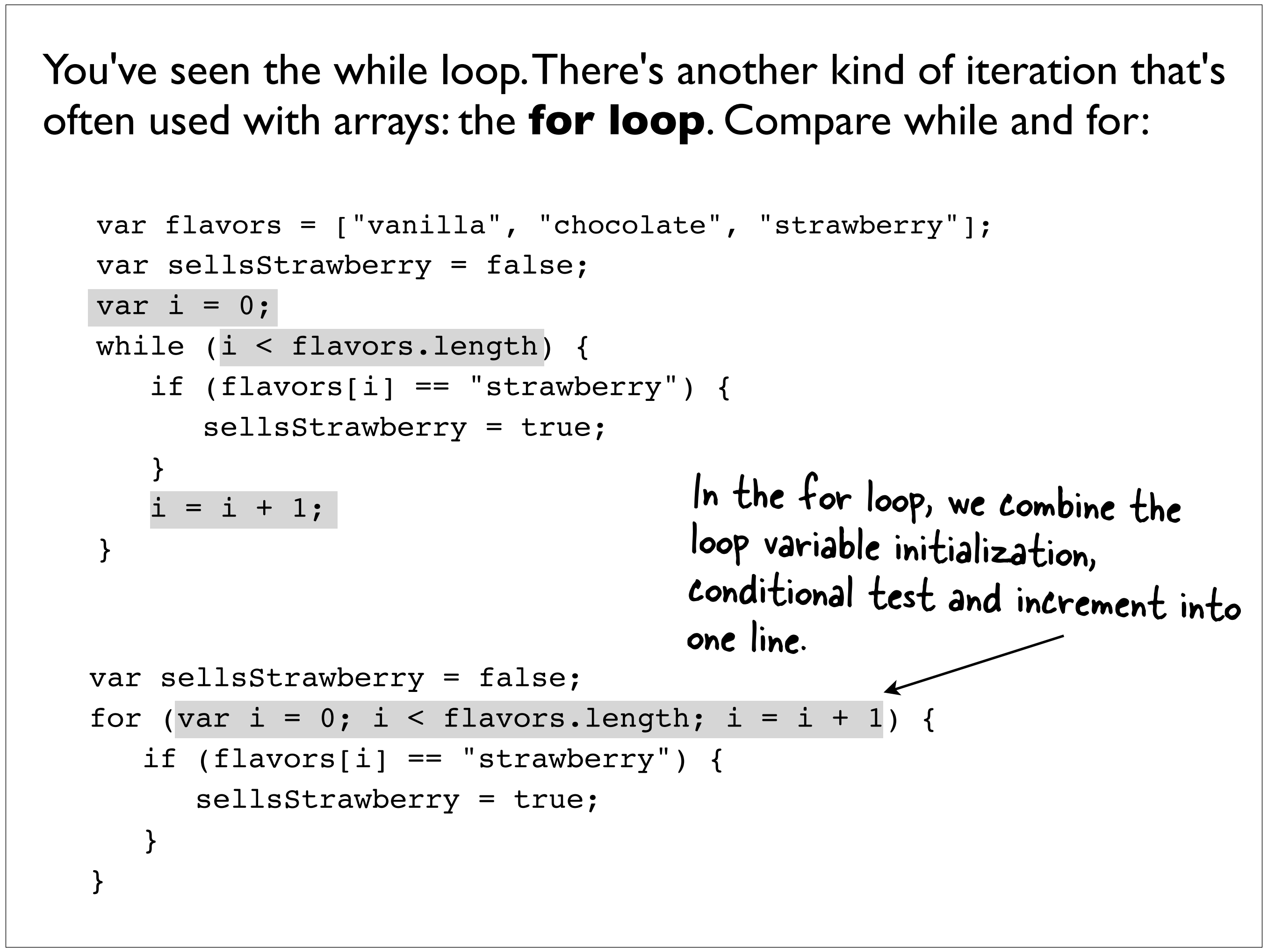
Be careful though, because it's easy to miss a spot and end up with a sparse array (an array with some elements undefined).

The length of an empty array is 0.
We start by creating an empty array and adding new elements to it.

In the second example, we provided values for cats[0], and cats[2], but not cats[1].
So the value of cats[1] is undefined.

Try typing this in.
What's the length of the cats array?

Well talk more about adding items onto an array and dealing with sparse arrays later.



You've seen the while loop. There's another kind of iteration that's often used with arrays: the **for loop**. Compare while and for:

```
var flavors = ["vanilla", "chocolate", "strawberry"];
var sellsStrawberry = false;
var i = 0;
while (i < flavors.length) {
   if (flavors[i] == "strawberry") {
      sellsStrawberry = true;
   }
   i = i + 1;
}
```

In the for loop, we combine the loop variable initialization, conditional test and increment into one line.

```
var sellsStrawberry = false;
for (var i = 0; i < flavors.length; i = i + 1) {
   if (flavors[i] == "strawberry") {
      sellsStrawberry = true;
   }
}
```

Developers often use the for loop when iterating through arrays because it's a bit more concise.

The variable i gets initialized once.
Then we compare i to flavors.length. If i is less than flavors.length we
Execute the body of the loop (everything between { and }).
Then we increment i, and test i again.
When i = flavors.length, then we stop iterating.

Post-increment (and other variations)

- You'll often see for loops written like this:

```
for (var i = 0; i < flavors.length; i++) {
   if (flavors[i] == "strawberry") {
      sellsStrawberry = true;
   }
}
```

- `i++` is shorthand for `i = i + 1`
- `i-- is shorthand for i = i - 1`
- Try the following code in the console:

```
var i = 0;
i++
i
i--
i
var x = i++;
x
i
```

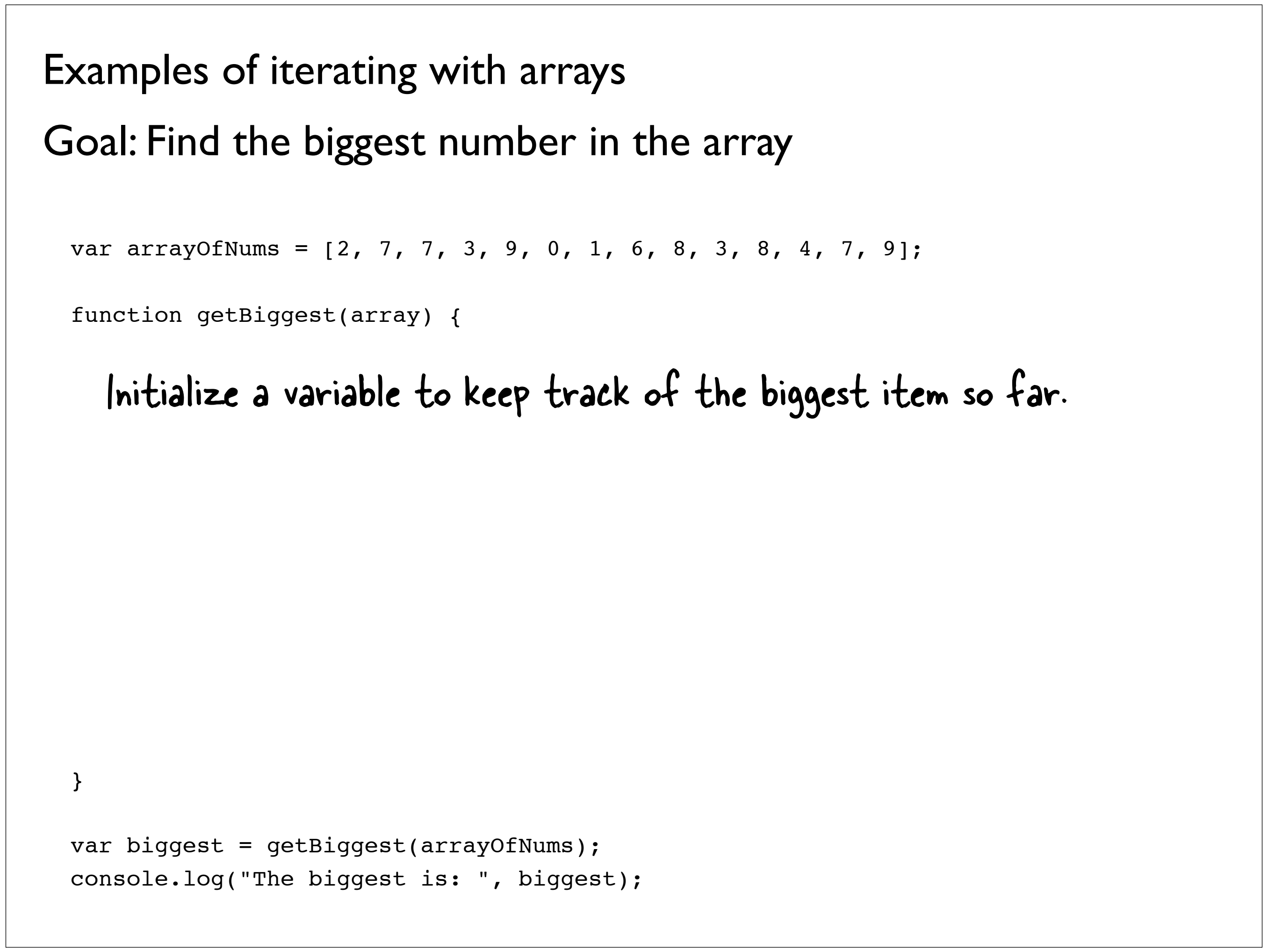
Examples of iterating with arrays

Goal: Find the biggest number in the array

```
var arrayOfNums = [2, 7, 7, 3, 9, 0, 1, 6, 8, 3, 8, 4, 7, 9];

function getBiggest(array) {












}

var biggest = getBiggest(arrayOfNums);
console.log("The biggest is: ", biggest);
```

Take a look at the arrayOfNums array
Our goal is to write a function, getBiggest, that takes an array, and finds the biggest number in the array.
We'll call getBiggest and pass in arrayOfNums.
As you can see, 9 is the biggest number in the array.
So we should expect to see 9 as the result.

Take a few minutes now and see if you can write this code.
Work with your neighbor.



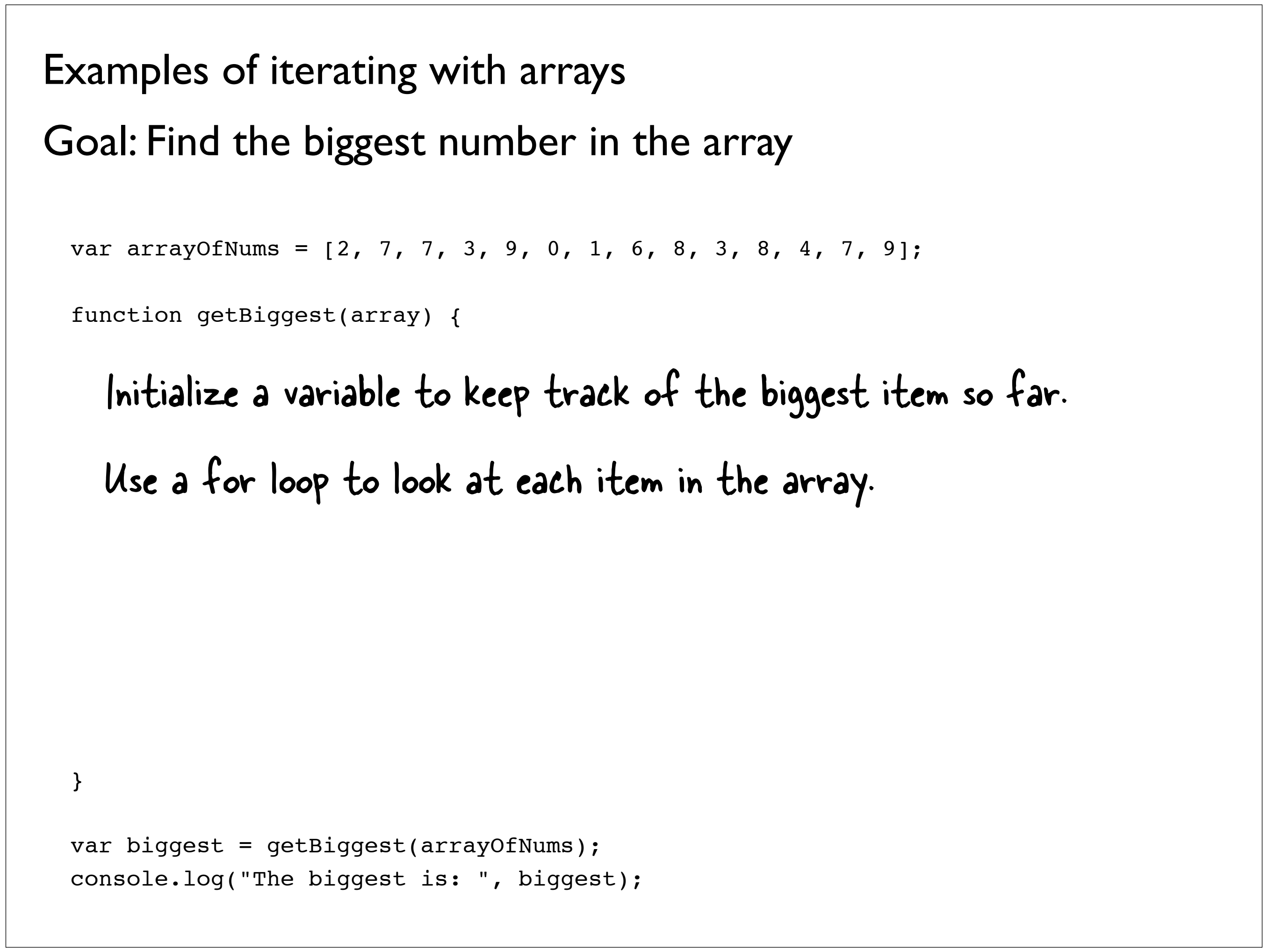
Examples of iterating with arrays

Goal: Find the biggest number in the array

```
var arrayOfNums = [2, 7, 7, 3, 9, 0, 1, 6, 8, 3, 8, 4, 7, 9];

function getBiggest(array) {

   Initialize a variable to keep track of the biggest item so far.









}

var biggest = getBiggest(arrayOfNums);
console.log("The biggest is: ", biggest);
```

Take a look at the arrayOfNums array
Our goal is to write a function, getBiggest, that takes an array, and finds the biggest number in the array.
We'll call getBiggest and pass in arrayOfNums.
As you can see, 9 is the biggest number in the array.
So we should expect to see 9 as the result.

Take a few minutes now and see if you can write this code.
Work with your neighbor.



Examples of iterating with arrays

Goal: Find the biggest number in the array

```
var arrayOfNums = [2, 7, 7, 3, 9, 0, 1, 6, 8, 3, 8, 4, 7, 9];

function getBiggest(array) {

   Initialize a variable to keep track of the biggest item so far.

   Use a for loop to look at each item in the array.

}

var biggest = getBiggest(arrayOfNums);
console.log("The biggest is: ", biggest);
```

Take a look at the arrayOfNums array
Our goal is to write a function, getBiggest, that takes an array, and finds the biggest number in the array.
We'll call getBiggest and pass in arrayOfNums.
As you can see, 9 is the biggest number in the array.
So we should expect to see 9 as the result.

Take a few minutes now and see if you can write this code.
Work with your neighbor.

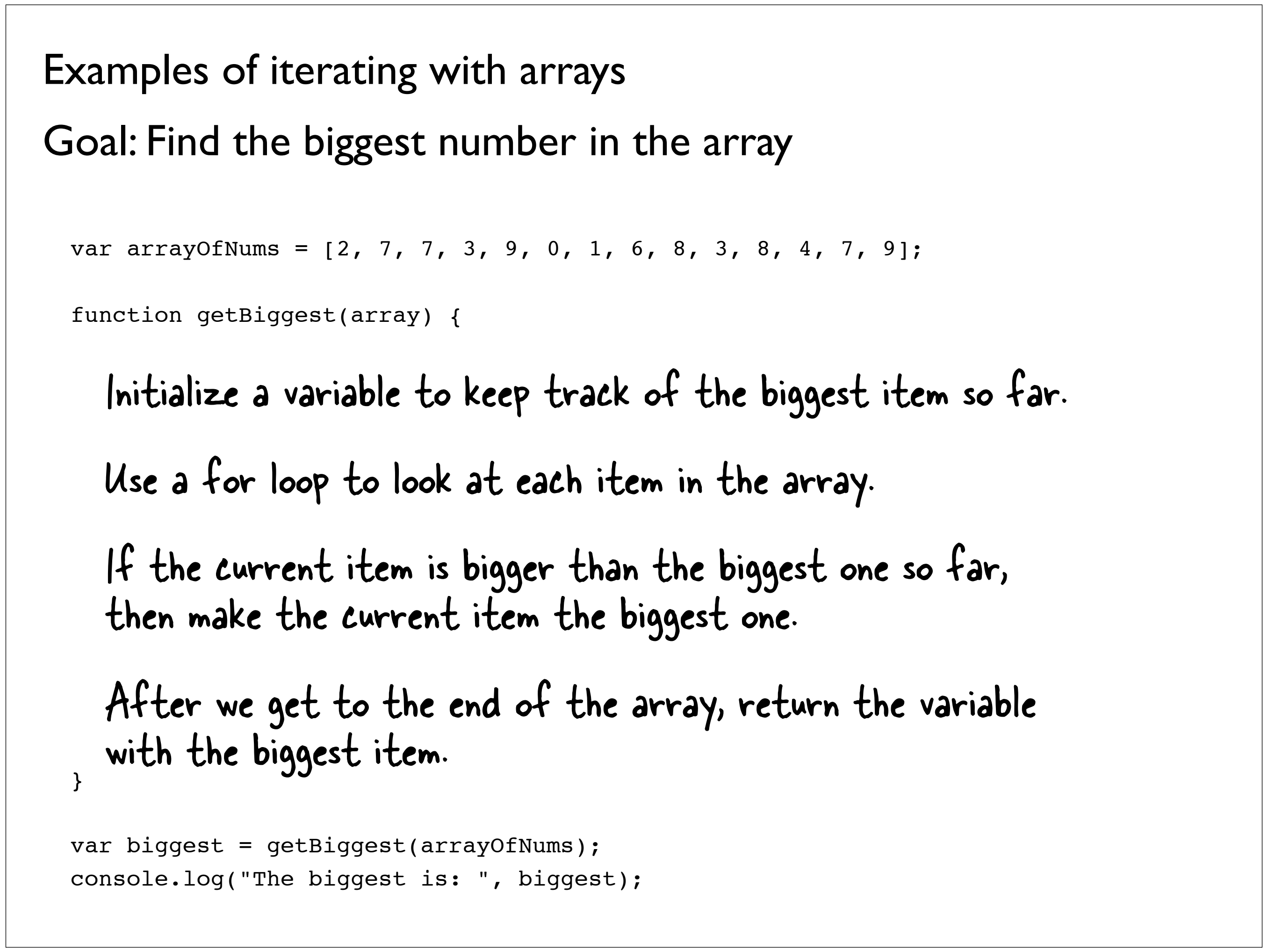
Examples of iterating with arrays

Goal: Find the biggest number in the array

```
var arrayOfNums = [2, 7, 7, 3, 9, 0, 1, 6, 8, 3, 8, 4, 7, 9];

function getBiggest(array) {

   Initialize a variable to keep track of the biggest item so far.

   Use a for loop to look at each item in the array.

   If the current item is bigger than the biggest one so far,
   then make the current item the biggest one.



}

var biggest = getBiggest(arrayOfNums);
console.log("The biggest is: ", biggest);
```

Take a look at the arrayOfNums array
Our goal is to write a function, getBiggest, that takes an array, and finds the biggest number in the array.
We'll call getBiggest and pass in arrayOfNums.
As you can see, 9 is the biggest number in the array.
So we should expect to see 9 as the result.

Take a few minutes now and see if you can write this code.
Work with your neighbor.



Examples of iterating with arrays

Goal: Find the biggest number in the array

```
var arrayOfNums = [2, 7, 7, 3, 9, 0, 1, 6, 8, 3, 8, 4, 7, 9];

function getBiggest(array) {

   Initialize a variable to keep track of the biggest item so far.

   Use a for loop to look at each item in the array.

   If the current item is bigger than the biggest one so far,
   then make the current item the biggest one.

   After we get to the end of the array, return the variable
   with the biggest item.
}

var biggest = getBiggest(arrayOfNums);
console.log("The biggest is: ", biggest);
```

Take a look at the arrayOfNums array
Our goal is to write a function, getBiggest, that takes an array, and finds the biggest number in the array.
We'll call getBiggest and pass in arrayOfNums.
As you can see, 9 is the biggest number in the array.
So we should expect to see 9 as the result.

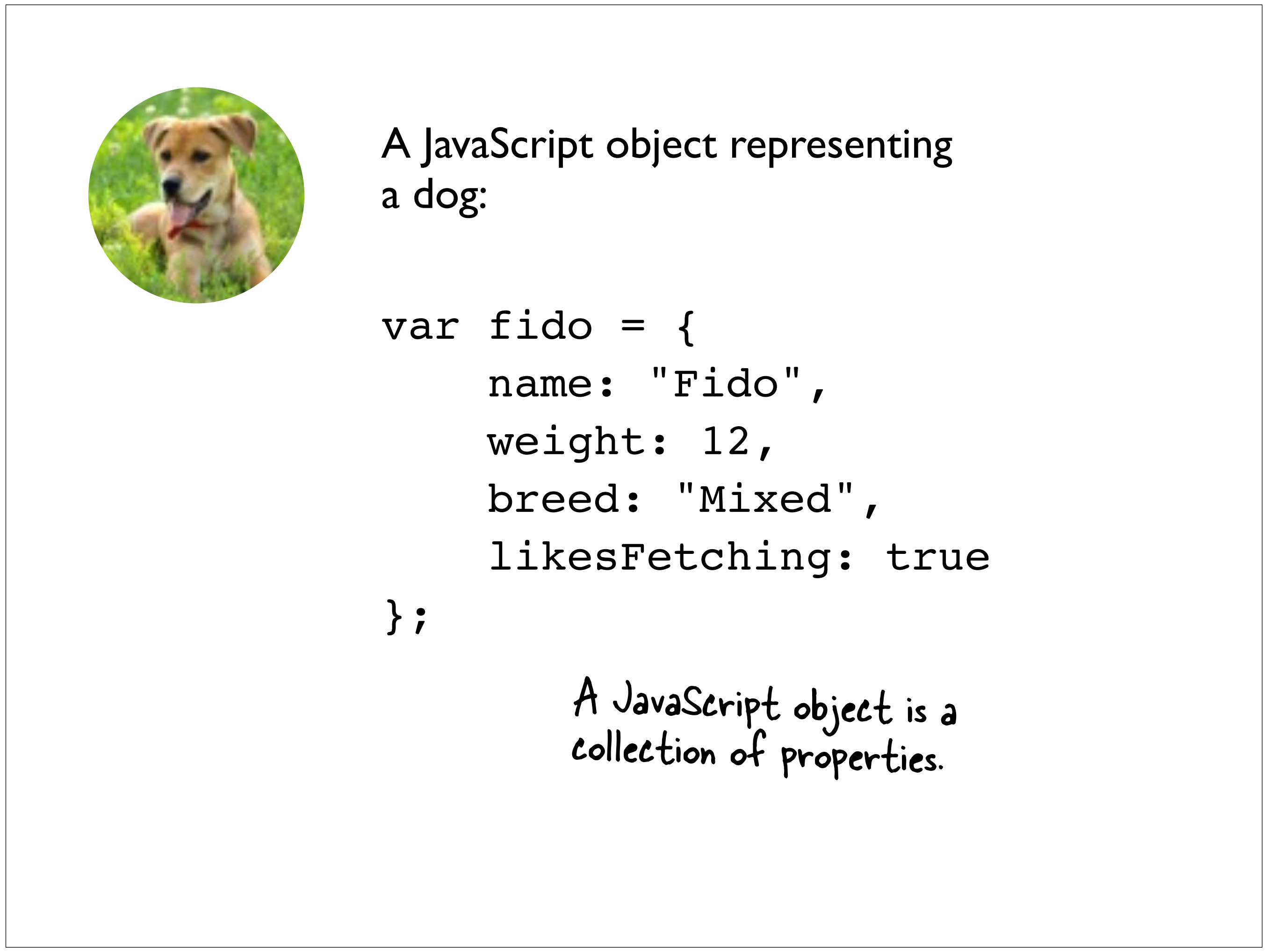
Take a few minutes now and see if you can write this code.
Work with your neighbor.

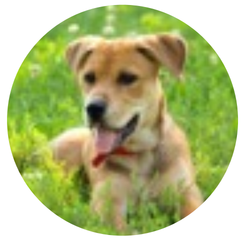
Examples of iterating with arrays

Goal: Find the biggest number in the array

```
var arrayOfNums = [2, 7, 7, 3, 9, 0, 1, 6, 8, 3, 8, 4, 7, 9];

function getBiggest(array) {
    var biggest = -Infinity;
    for (var i = 0; i < array.length; i++) {
        if (array[i] > biggest) {
            biggest = array[i];
        }
    }
    return biggest;
}

var biggest = getBiggest(arrayOfNums);
console.log("The biggest is: ", biggest);
```

Objects



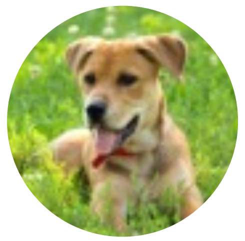


A JavaScript object representing a dog:

```
var fido = {
    name: "Fido",
    weight: 12,
    breed: "Mixed",
    likesFetching: true
};
```

A JavaScript object is a collection of properties.

Try typing this into the console now (or create a simple HTML file and add this code to a script).

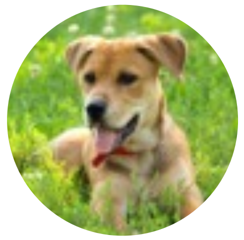


We can group properties together into an object, like this:

```
var fido = {
    name: "Fido",
    weight: 12,
    breed: "Mixed",
    likesFetching: true
};
```

Start...

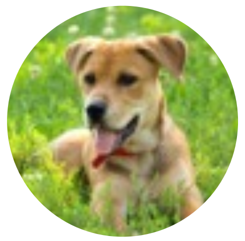
...and end with braces.



We can group properties together into an object, like this:

Each property of an object is a name/value pair.

```
var fido = {
    name: "Fido",
    weight: 12,
    breed: "Mixed",
    likesFetching: true
};
```

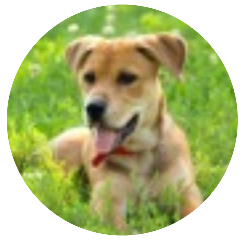


We can group properties together into an object, like this:

```
var fido = {
    name: "Fido",
    weight: 12,
    breed: "Mixed",
    likesFetching: true
};
```

Each property of an object is a name/value pair.

Separate each name/value pair with a comma.



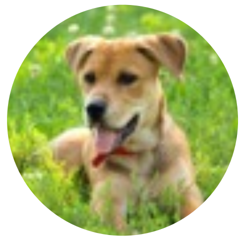
We can group properties together into an object, like this:

```
var fido = {
    name: "Fido",
    weight: 12,
    breed: "Mixed",
    likesFetching: true
};
```

Each property of an object is a name/value pair.

Separate each name/value pair with a comma.

Don't need a comma on the last property!



We can group properties together into an object, like this:

Separate the name and value of a property with a colon.

```
var fido = {
    name: "Fido",
    weight: 12,
    breed: "Mixed",
    likesFetching: true
};
```

Easy mistakes to make:
* Use a semi-colon instead of a comma to separate the properties
* Use = instead of : to define properties

Your turn: make a JavaScript object
for this dog from these values:

Her name is: Rainbow
Her weight is: 16
Her breed is: Beagle
She likes fetching balls

Can you think of any other
properties you'd like to add?

Photo: https://www.flickr.com/photos/onefromrome/174097822/

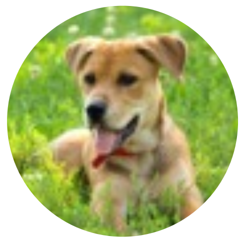
Try this now before going on...

Your turn: make a JavaScript object
for this dog from these values:

Her name is: Rainbow
Her weight is: 16
Her breed is: Beagle
She likes fetching balls

```
var rainbow = {
    name: "Rainbow",
    weight: 16,
    breed: "Beagle",
    likesFetching: true
};
```

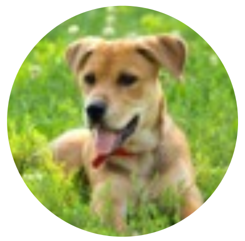
Photo: https://www.flickr.com/photos/onefromrome/174097822/



Values you can store in objects:

```
var fido = {
    name: "Fido",
    weight: 12,
    breed: "Mixed",
    likesFetching: true,
    walkTimes: ["8am", "1pm", "9pm"]
};
```

You can store any of the types of values we've seen so far: numbers, strings, booleans, and arrays. You'll see later, we can store other types of values too, like other objects and even functions!

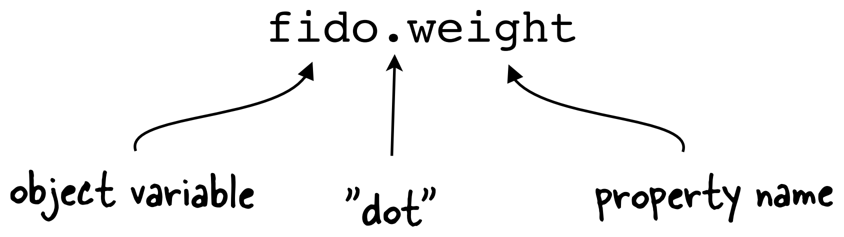


To access the properties in an object, we use the *dot notation.*

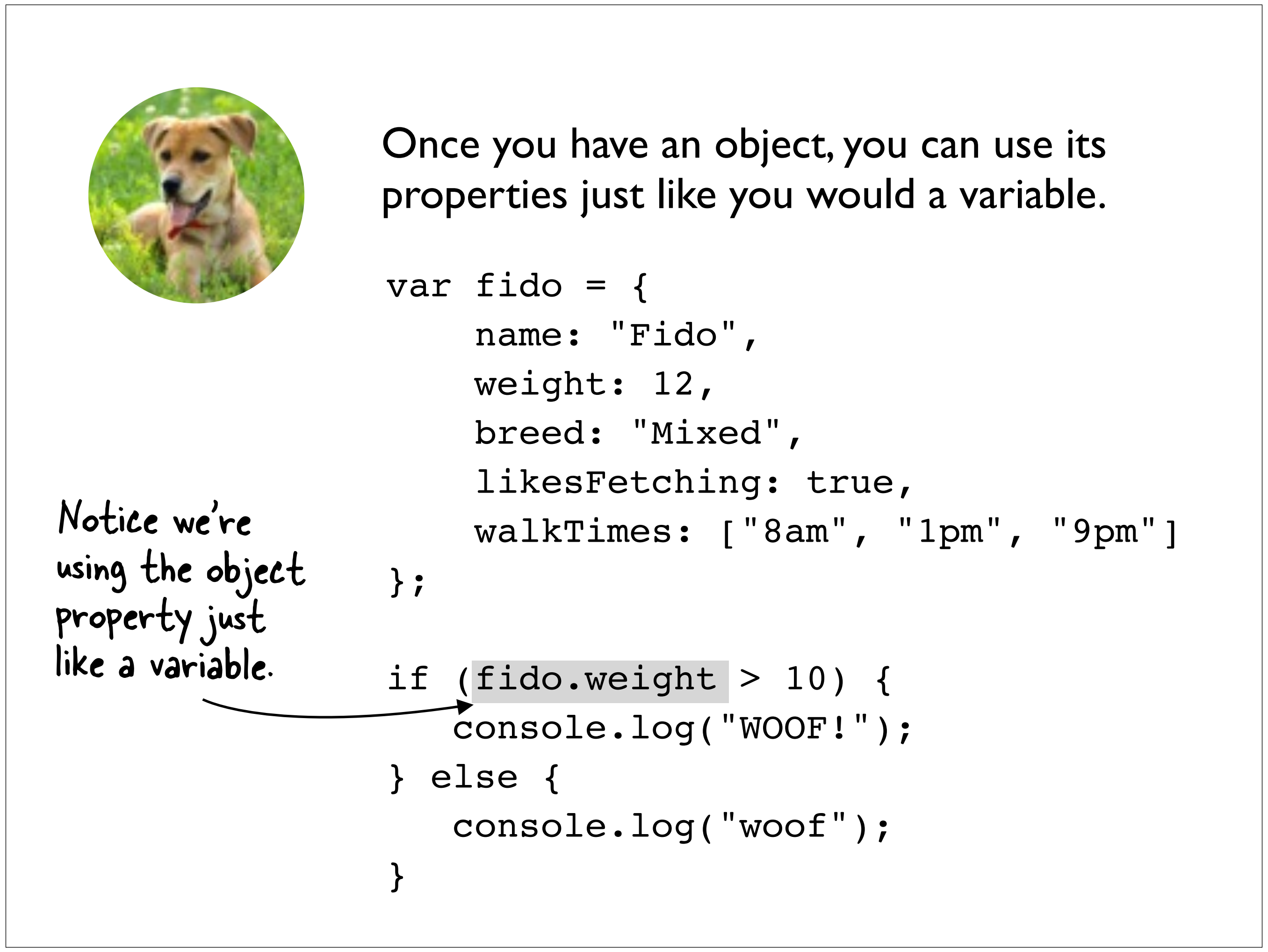
```
var fido = {
    name: "Fido",
    weight: 12,
    breed: "Mixed",
    likesFetching: true,
    walkTimes: ["8am", "1pm", "9pm"]
};
console.log("My name is " + fido.name);
console.log("My weight is " +
            fido.weight);
```

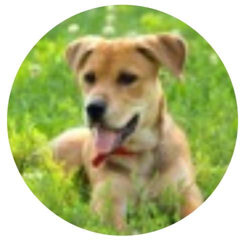
object variable "dot" property name

This says: "*Give me the value of the weight property in the fido object*".



Try this now in the console.
(Assuming you've already typed in the fido object).





Once you have an object, you can use its properties just like you would a variable.

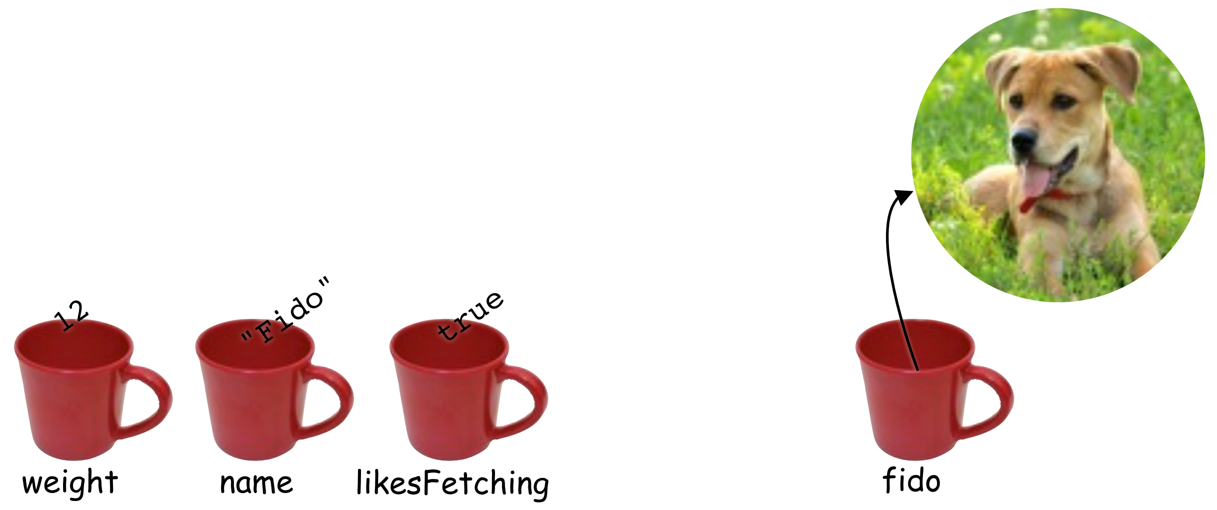
```
var fido = {
    name: "Fido",
    weight: 12,
    breed: "Mixed",
    likesFetching: true,
    walkTimes: ["8am", "1pm", "9pm"]
};

if (fido.weight > 10) {
   console.log("WOOF!");
} else {
   console.log("woof");
}
```

Notice we're using the object property just like a variable.

You can use an object's property anywhere you would a variable. They work exactly the same way.

Your turn: try writing a loop to display all the values of the fido.walkTimes array in the console.

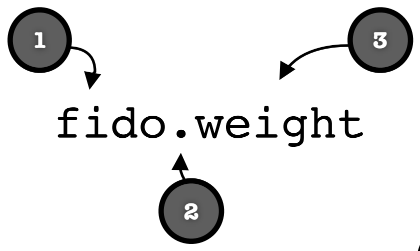
Objects are reference variables: we store a *reference* to the object the variable:

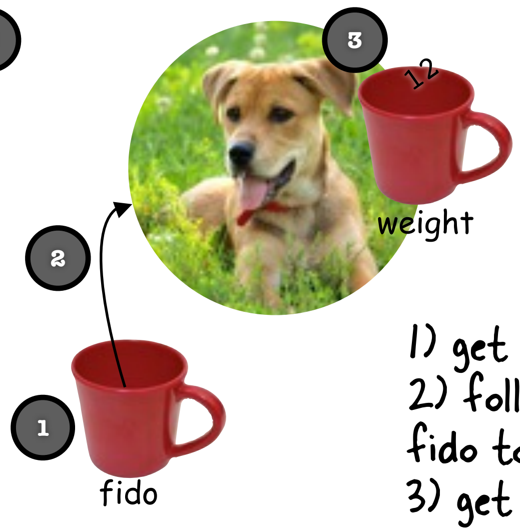


The variable fido doesn't contain the dog object; it contains a reference to the dog object.

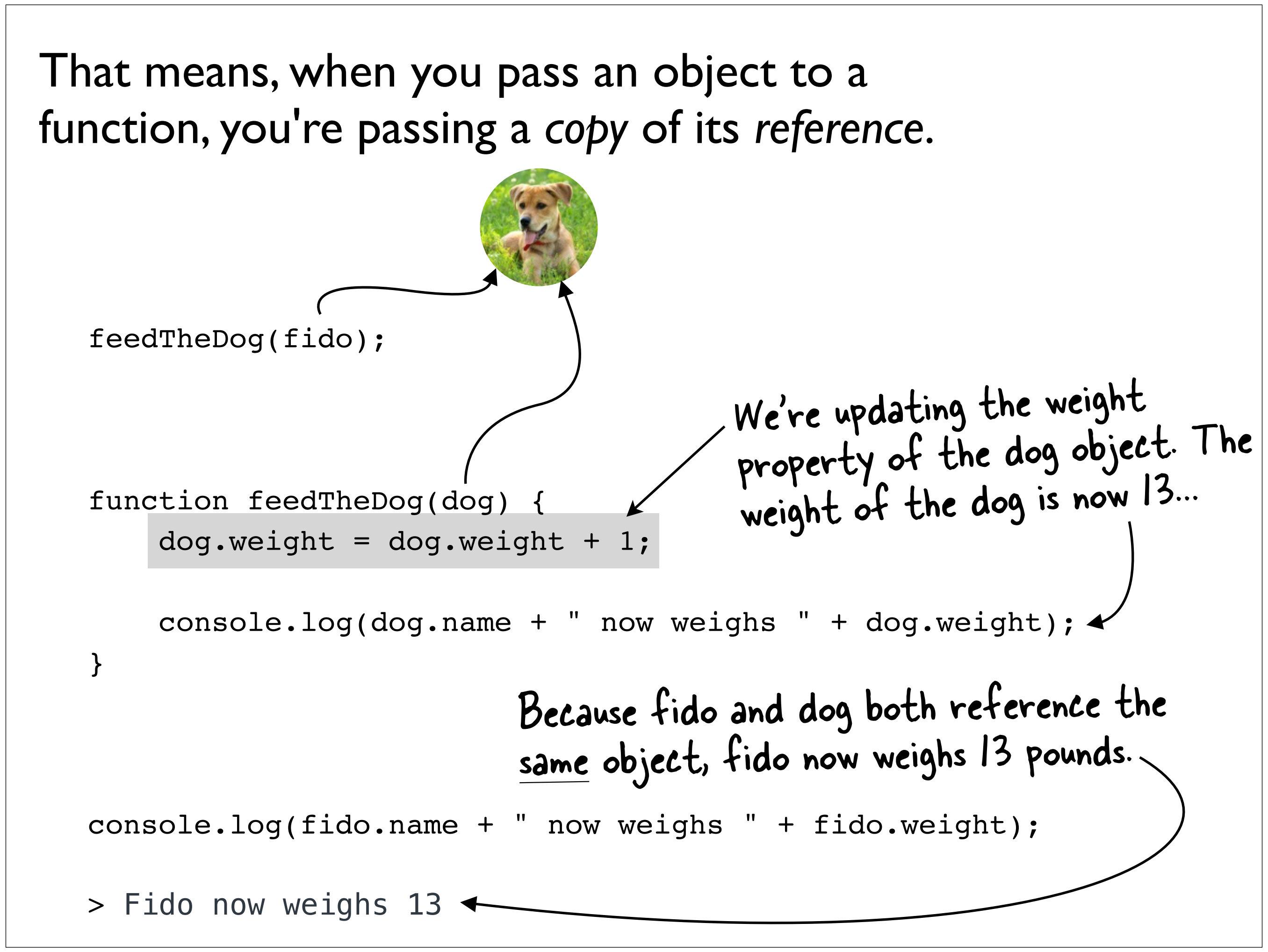
You can think of a reference like a pointer.
Instead of containing the dog object itself, fido contains a value that points to the dog object.

When you use the dot notation to access an object's property value, you're asking JavaScript to follow the reference to the object to find the property, and get its value:





1) get fido, the variable.
2) follow the reference in fido to the object.
3) get the weight property, and return the value in the property.



That means, when you pass an object to a function, you're passing a *copy* of its *reference*.

```
feedTheDog(fido);



function feedTheDog(dog) {
    dog.weight = dog.weight + 1;

    console.log(dog.name + " now weighs " + dog.weight);
}
```

We're updating the weight property of the dog object. The weight of the dog is now 13...

Because fido and dog both reference the same object, fido now weighs 13 pounds.

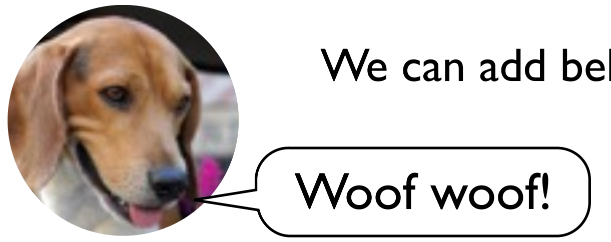
```
console.log(fido.name + " now weighs " + fido.weight);

> Fido now weighs 13
```

The same rule applies to functions that applied before:
we "pass-by-value" which means we pass a COPY of the reference to the dog object
(That means we do NOT pass a copy of the object itself! Just the reference to the object.)

So when we change the value of dog.weight, we're also changing the value of fido.weight
because fido and dog point to the same object.

Review carefully and make sure you understand why. Ask questions if you don't!!!

We can add behavior to objects with *methods*:

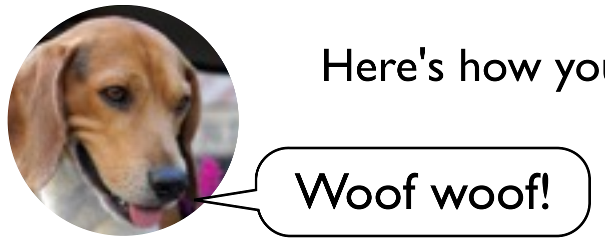


bark is another property of the object. Its value is a function.

```
var rainbow = {
    name: "Rainbow",
    weight: 16,
    breed: "Beagle",
    likesFetching: true,
    bark: function bark() {
        console.log("Woof woof!");
    }
};
```

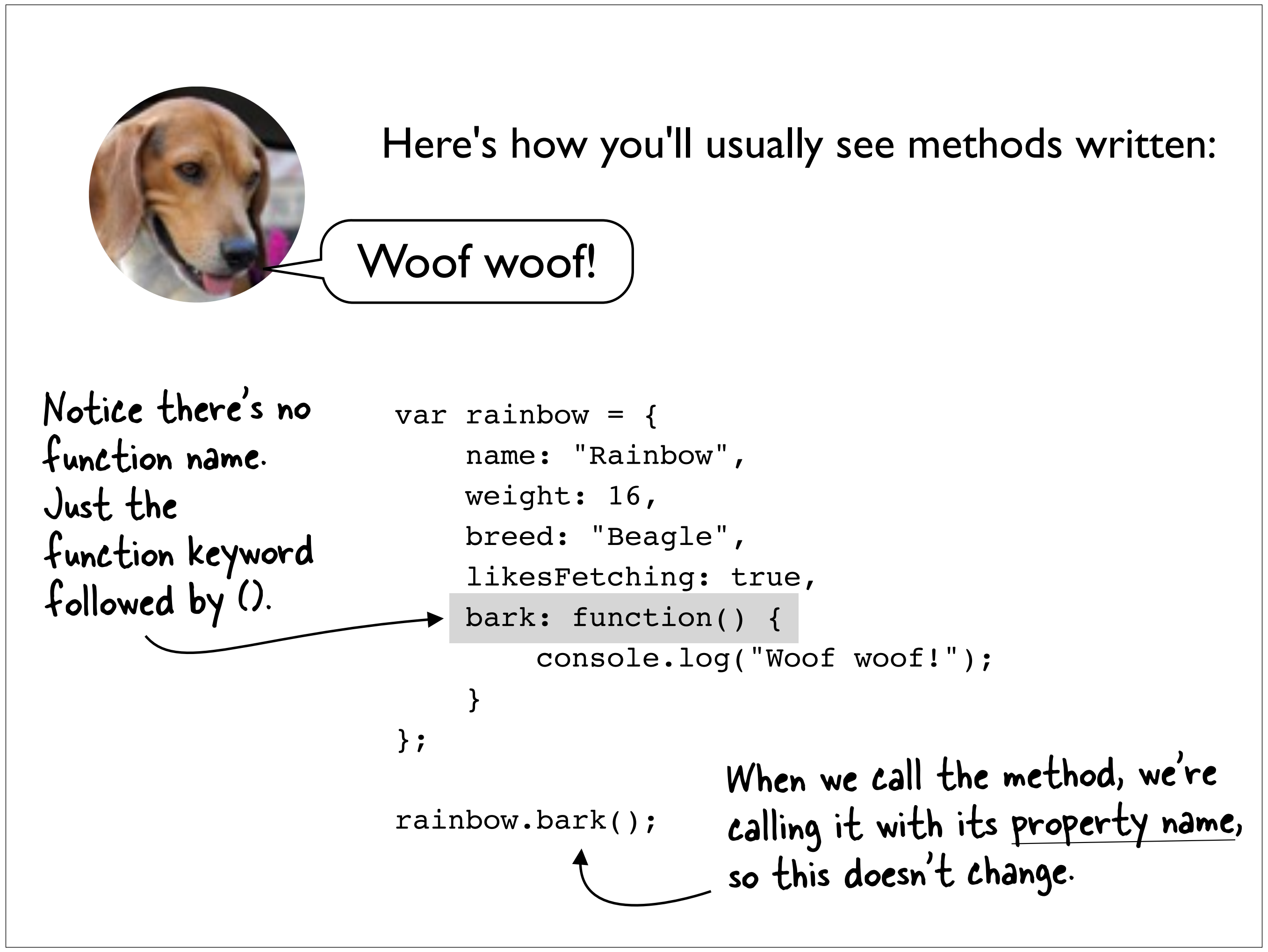
When we have a function in an object, we call that function a "method".

Here's how you call a method:

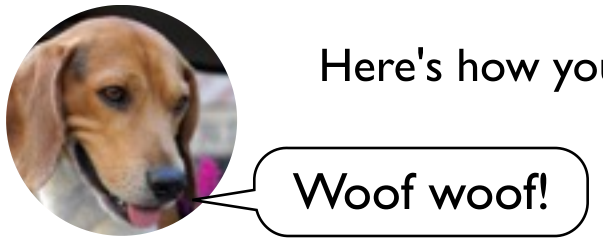


We access the method property just like we do any other property, and call the function like normal.

```
var rainbow = {
    name: "Rainbow",
    weight: 16,
    breed: "Beagle",
    likesFetching: true,
    bark: function bark() {
        console.log("Woof woof!");
    }
};

rainbow.bark();
```



Here's how you'll usually see methods written:



Notice there's no function name. Just the function keyword followed by ().

```
var rainbow = {
    name: "Rainbow",
    weight: 16,
    breed: "Beagle",
    likesFetching: true,
    bark: function() {
        console.log("Woof woof!");
    }
};

rainbow.bark();
```

When we call the method, we're calling it with its property name, so this doesn't change.

Most of the time, when you see methods in objects, you won't see a function name.
Instead you'll just see "function() { ... }".
This is known as an "anonymous function" and we'll be coming back to this later...

For now, just get used to seeing methods written this way!

Even if a method doesn't have a function name, you still call it the same way: using the name of the property.

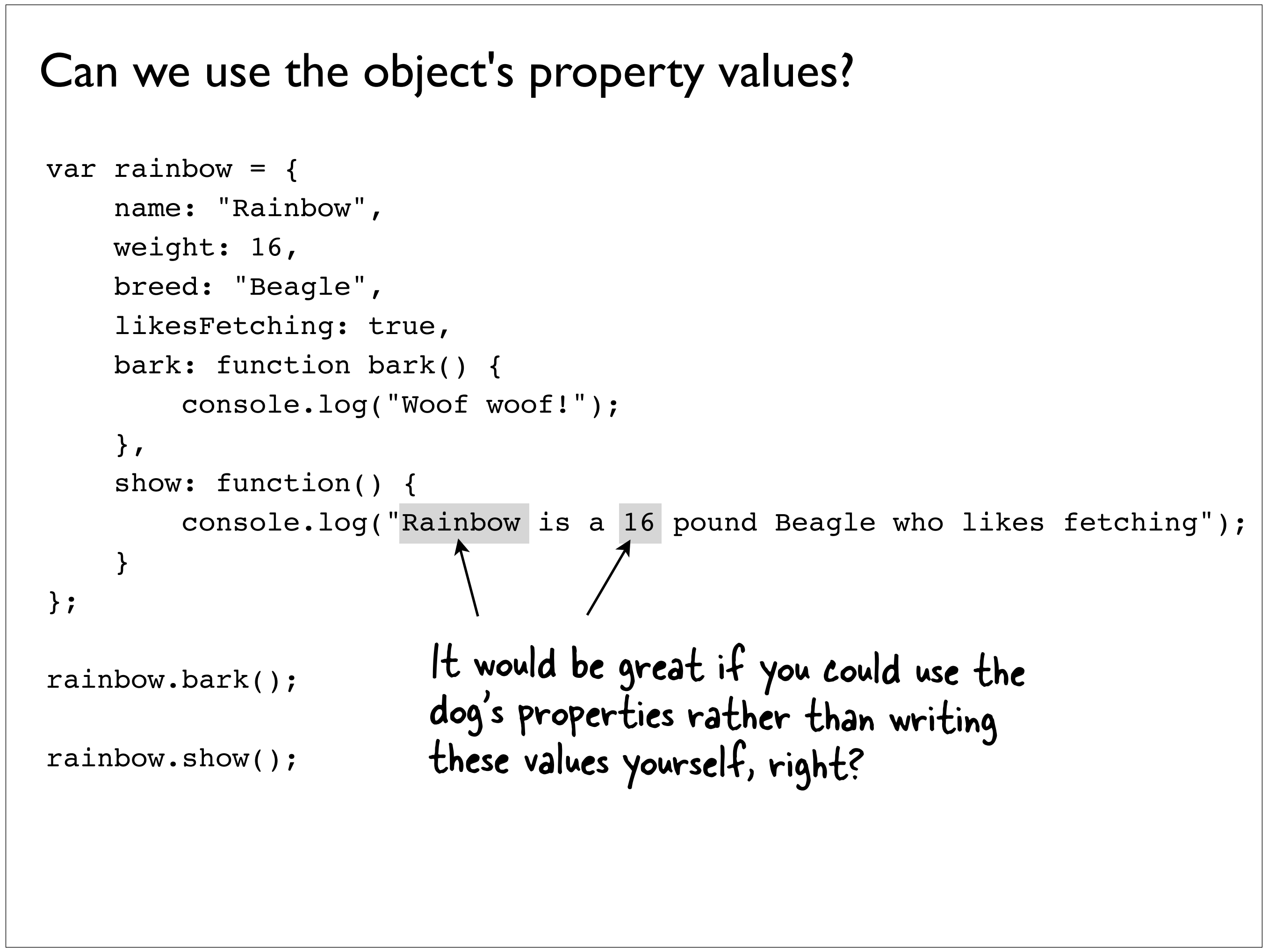
Try typing this object with the method into the console now, and then call the object's method.
Make sure you can get this working.

Add a show method:

```
var rainbow = {
    name: "Rainbow",
    weight: 16,
    breed: "Beagle",
    likesFetching: true,
    bark: function bark() {
        console.log("Woof woof!");
    },
    show: function() {
        console.log("Rainbow is a 16 pound Beagle who likes fetching");
    }
};

rainbow.bark();

rainbow.show();
```

Here's what I did.

(Did you remember to add a "," after the previous method when you added the new one????)



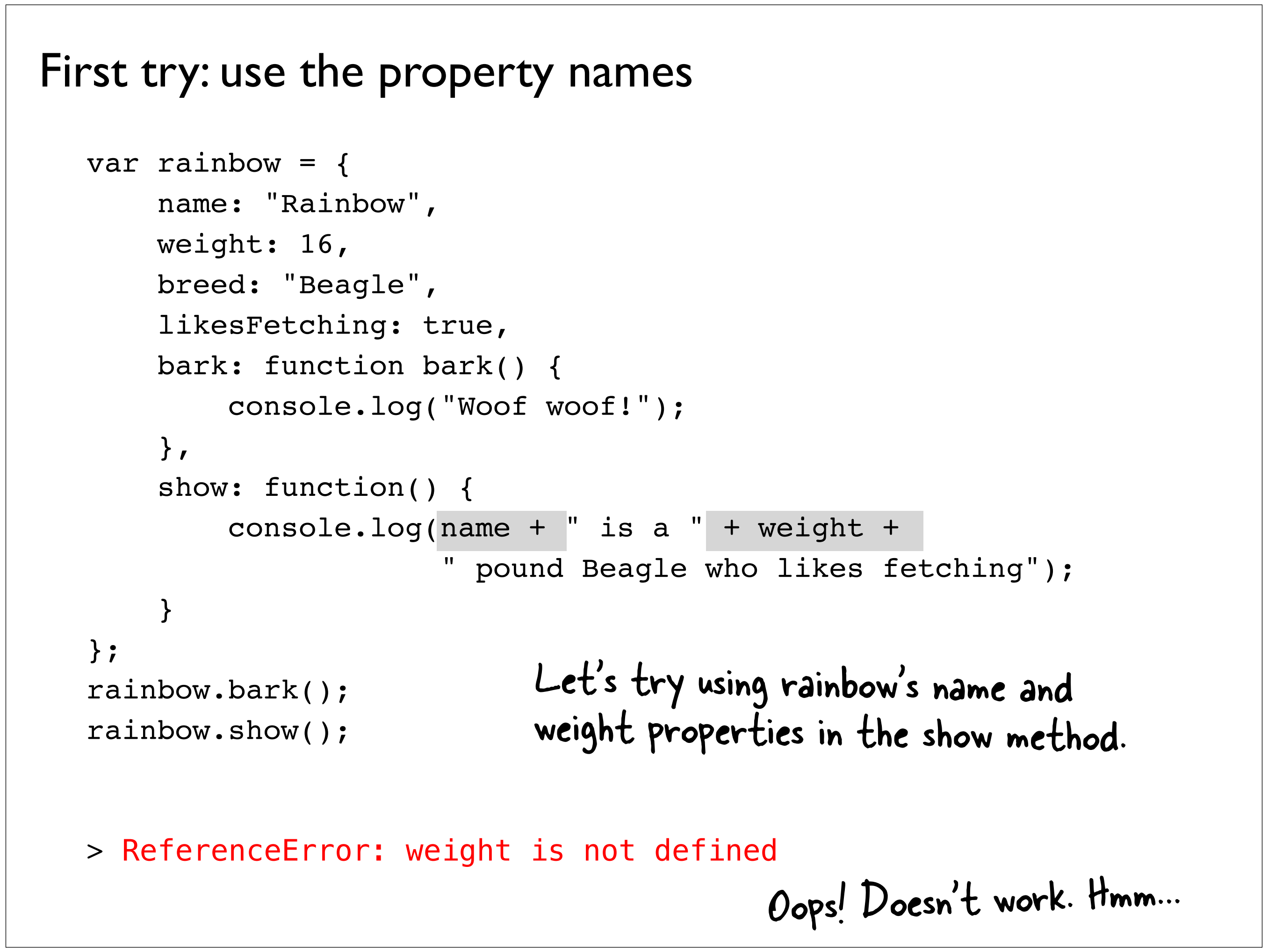
Can we use the object's property values?

```
var rainbow = {
    name: "Rainbow",
    weight: 16,
    breed: "Beagle",
    likesFetching: true,
    bark: function bark() {
        console.log("Woof woof!");
    },
    show: function() {
        console.log("Rainbow is a 16 pound Beagle who likes fetching");
    }
};

rainbow.bark();

rainbow.show();
```

It would be great if you could use the dog's properties rather than writing these values yourself, right?

Here's what I did.

(Did you remember to add a "," after the previous method when you added the new one????)



First try: use the property names

```
var rainbow = {
    name: "Rainbow",
    weight: 16,
    breed: "Beagle",
    likesFetching: true,
    bark: function bark() {
        console.log("Woof woof!");
    },
    show: function() {
        console.log(name + " is a " + weight +
                    " pound Beagle who likes fetching");
    }
};
rainbow.bark();
rainbow.show();
```

Let's try using rainbow's name and weight properties in the show method.

```
> ReferenceError: weight is not defined
```

Oops! Doesn't work. Hmm...

You might have tried using the property names.

If you try this, you'll find it doesn't work.

WHY?
* name, weight are not local variables, or parameters
* name, weight are not global variables either.
* they are properties of the rainbow object

Somehow we need a way to say "use the properties name and weight from THIS object"

First try: use the property names

```
var rainbow = {
    name: "Rainbow",
    weight: 16,
    breed: "Beagle",
    likesFetching: true,
    bark: function bark() {
        console.log("Woof woof!");
    },
    show: function() {
        console.log(this.name + " is a " + this.weight +
                    " pound Beagle who likes fetching");
    }
};
rainbow.bark();
rainbow.show();
```

We need to use the keyword "this".

```
> Woof woof!
Rainbow is a 16 pound Beagle who likes fetching
```

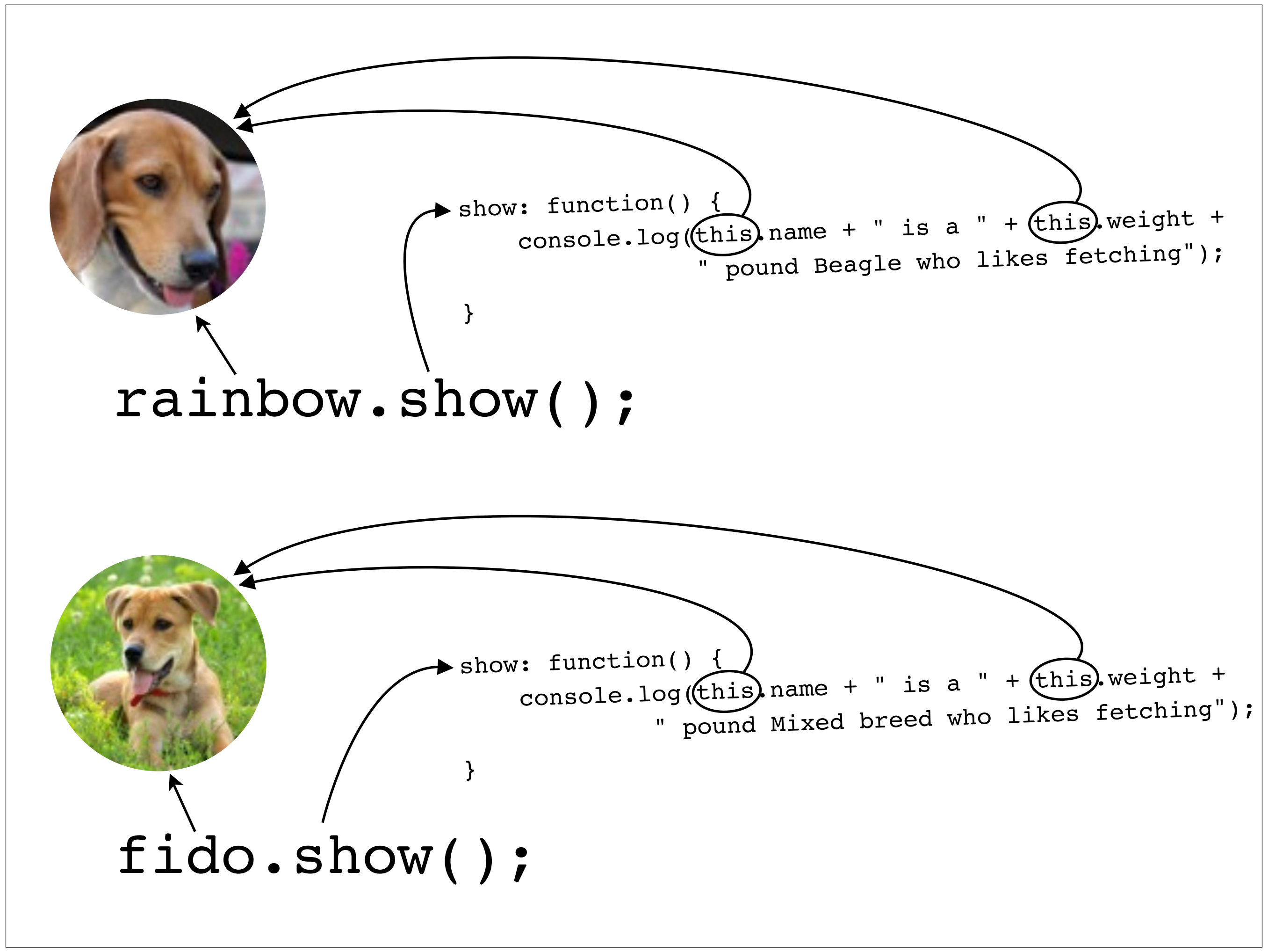
```
show: function() {
    console.log(this.name + " is a " + this.weight +
                " pound Beagle who likes fetching");

}
```

```
rainbow.show();
```

Add the show property to fido:

```
var fido = {
    name: "Fido",
    weight: 12,
    breed: "Mixed",
    likesFetching: true,
    walkTimes: ["8am", "1pm", "9pm"],
    show: function() {
        console.log(this.name + " is a " + this.weight +
                    " pound Mixed breed who likes fetching");
    }
};
```

Notice we don't have to change "this.name" and "this.weight", but we do change "Beagle" to "Mixed breed".



```
show: function() {
    console.log(this.name + " is a " + this.weight +
                " pound Beagle who likes fetching");

}
```

```
rainbow.show();
```

```
show: function() {
    console.log(this.name + " is a " + this.weight +
            " pound Mixed breed who likes fetching");

}
```

```
fido.show();
```

When you call a method of an object, "this" in the method is always* set to the objects whose method is called.

* Not always - there are ways to change what the value of "this" is in a method, but we'll get to that later!

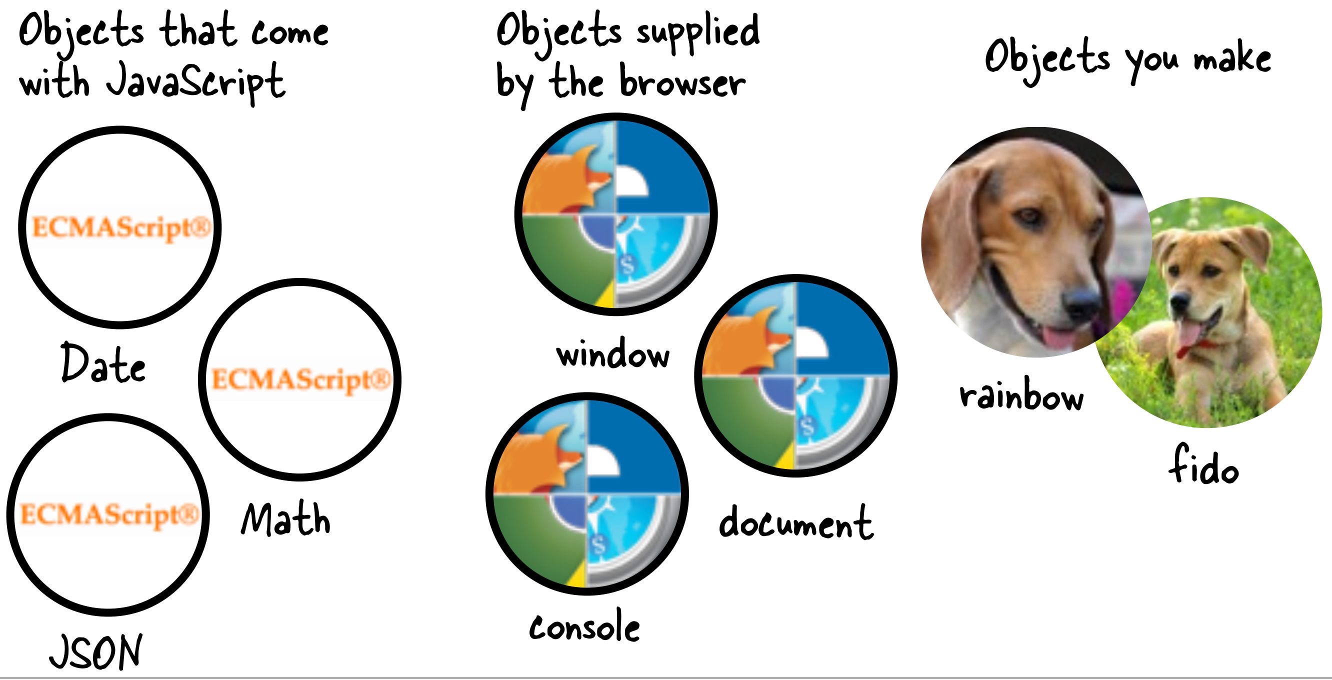
Create a to do object, with the following properties:

a task (a string) - a description of the thing to do
who (a string) - the name of a person to do it
done (a boolean) - is the task done or not?
isDone (a method) - get the value of done
setDone (a method) - set the value of done

Create 2 or 3 of these to do objects.

JavaScript is full objects!

In fact, almost everything in JavaScript *is* an object.
(We'll come back to this later.)



Browser image: http://www.acsa-caah.ca/tag/safari-browser
ECMAScript image: http://www.ecma-international.org/ecma-262/5.1/

You've seen examples of all three kinds of objects already:

Math

```
this.width = Math.floor(this.width +
               (this.width * (percentBigger/100)));
```

Console

```
console.log("My name is " + fido.name);
```

```
var fido = {
    name: "Fido",
    weight: 12,
    breed: "Mixed",
    likesFetching: true,
    walkTimes: ["8am", "1pm", "9pm"],
    bark: function bark() {
        console.log("Woof woof!");
    }
};
fido.bark();
```

So far, we've been creating objects with *object literals*

```
var fido = {
    name: "Fido",
    weight: 12,
    breed: "Mixed",
    likesFetching: true
};
```

```
var rainbow = {
    name: "Rainbow",
    weight: 16,
    breed: "Beagle",
    likesFetching: true
};
```

We've "literally" been writing out
each object with it's properties
whenever we want a new object.

And, what if we want a whole bunch of dogs? We're going to get awfully tired of writing out all those object literals...

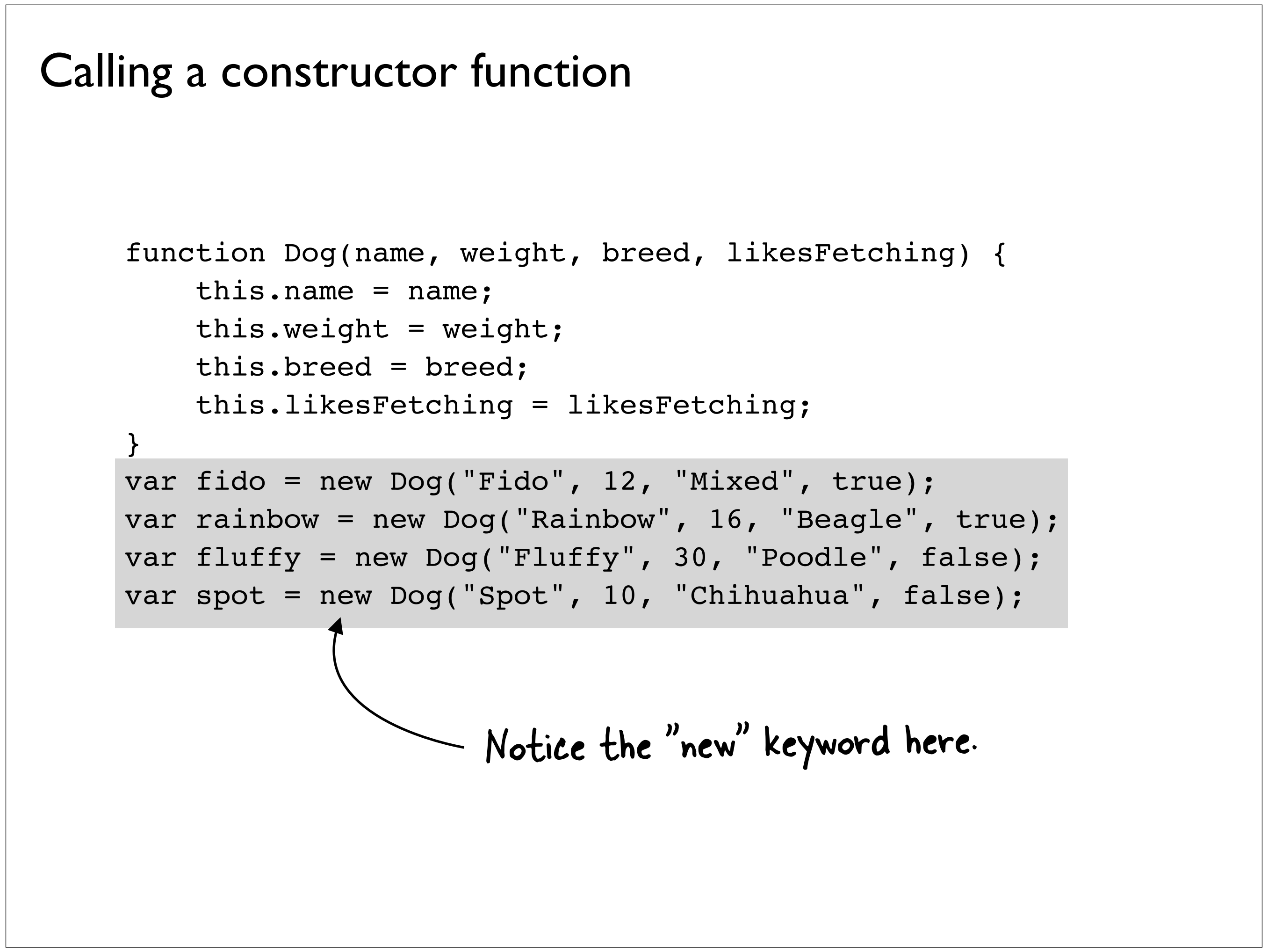
https://www.flickr.com/photos/cathedraljack/14206861858
https://www.flickr.com/photos/robertszlivka/2741653860
https://www.flickr.com/photos/tambako/821161211
https://www.flickr.com/photos/soldiersmediacenter/396544434

We need an easier way to construct objects that all have the same properties.

```
function Dog(name, weight, breed, likesFetching) {
    this.name = name;
    this.weight = weight;
    this.breed = breed;
    this.likesFetching = likesFetching;
}
```

Things to note:
* this is just a function... nothing special about the function itself.
* Convention: capitalize the name of the function.
* parameters match the properties we want to customize for the dog
* property names don't have to match the parameter names, but we usually write it like this to be clear about which parameters match up to which properties.
* Notice the function contains statements, ending in a semi-colon. Different from a comma-separated list of property name/value pairs like we see in object literals.
* NOTE: this.name is NOT the same as the parameter variable name!! Likewise with the rest of the properties/parameters.
* The function doesn't return anything.

Go ahead and type this into a new file, dog.html



Calling a constructor function

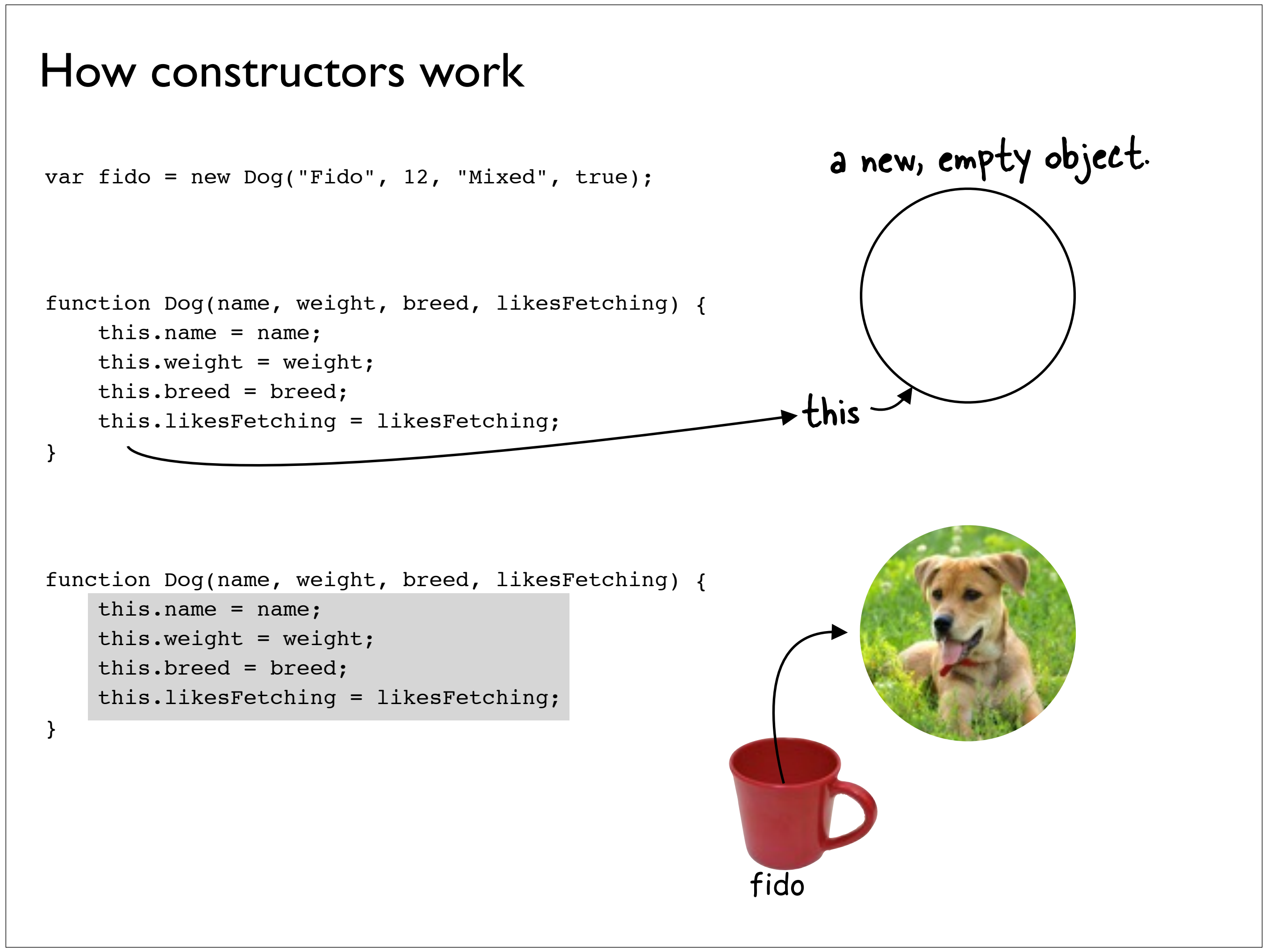
```
function Dog(name, weight, breed, likesFetching) {
    this.name = name;
    this.weight = weight;
    this.breed = breed;
    this.likesFetching = likesFetching;
}
var fido = new Dog("Fido", 12, "Mixed", true);
var rainbow = new Dog("Rainbow", 16, "Beagle", true);
var fluffy = new Dog("Fluffy", 30, "Poodle", false);
var spot = new Dog("Spot", 10, "Chihuahua", false);
```

Notice the "new" keyword here.

To call a constructor function, you use the keyword "new"

(We'll talk about what happens if you *forget* "new" later!)

Try typing this new code in and running the program by loading dog.html into your browser.



How constructors work

```
var fido = new Dog("Fido", 12, "Mixed", true);
```

```
function Dog(name, weight, breed, likesFetching) {
    this.name = name;
    this.weight = weight;
    this.breed = breed;
    this.likesFetching = likesFetching;
}
```

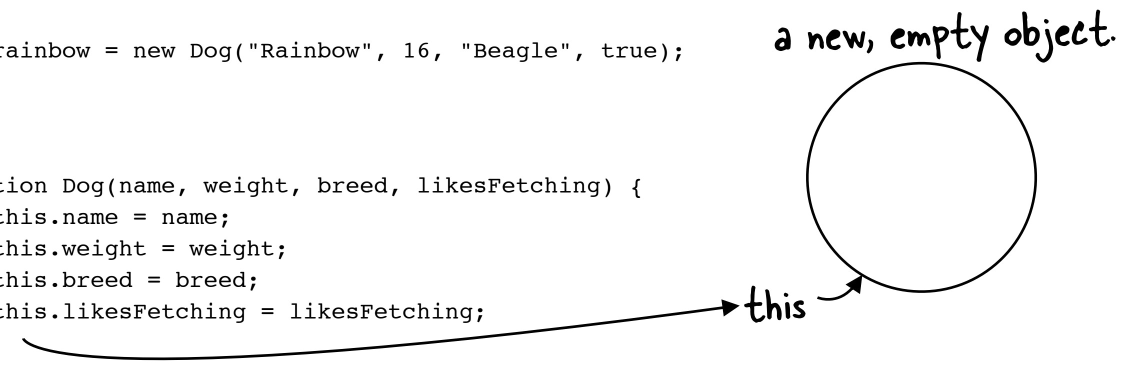
```
function Dog(name, weight, breed, likesFetching) {
    this.name = name;
    this.weight = weight;
    this.breed = breed;
    this.likesFetching = likesFetching;
}
```

When you call a constructor function with "new"
1) new creates an empty object
2) new then sets up the "this" variable inside the constructor function to point to the new, empty object.
3) the body of the constructor function adds properties and property values to the new object, using "this".
4) at the end of the function, the object is returned (actually a reference to the object!)
5) In this case, that object's reference is stored in the fido variable.

Same thing happens for any dog you want to create

```
var rainbow = new Dog("Rainbow", 16, "Beagle", true);
```

```
function Dog(name, weight, breed, likesFetching) {
    this.name = name;
    this.weight = weight;
    this.breed = breed;
    this.likesFetching = likesFetching;
}
```



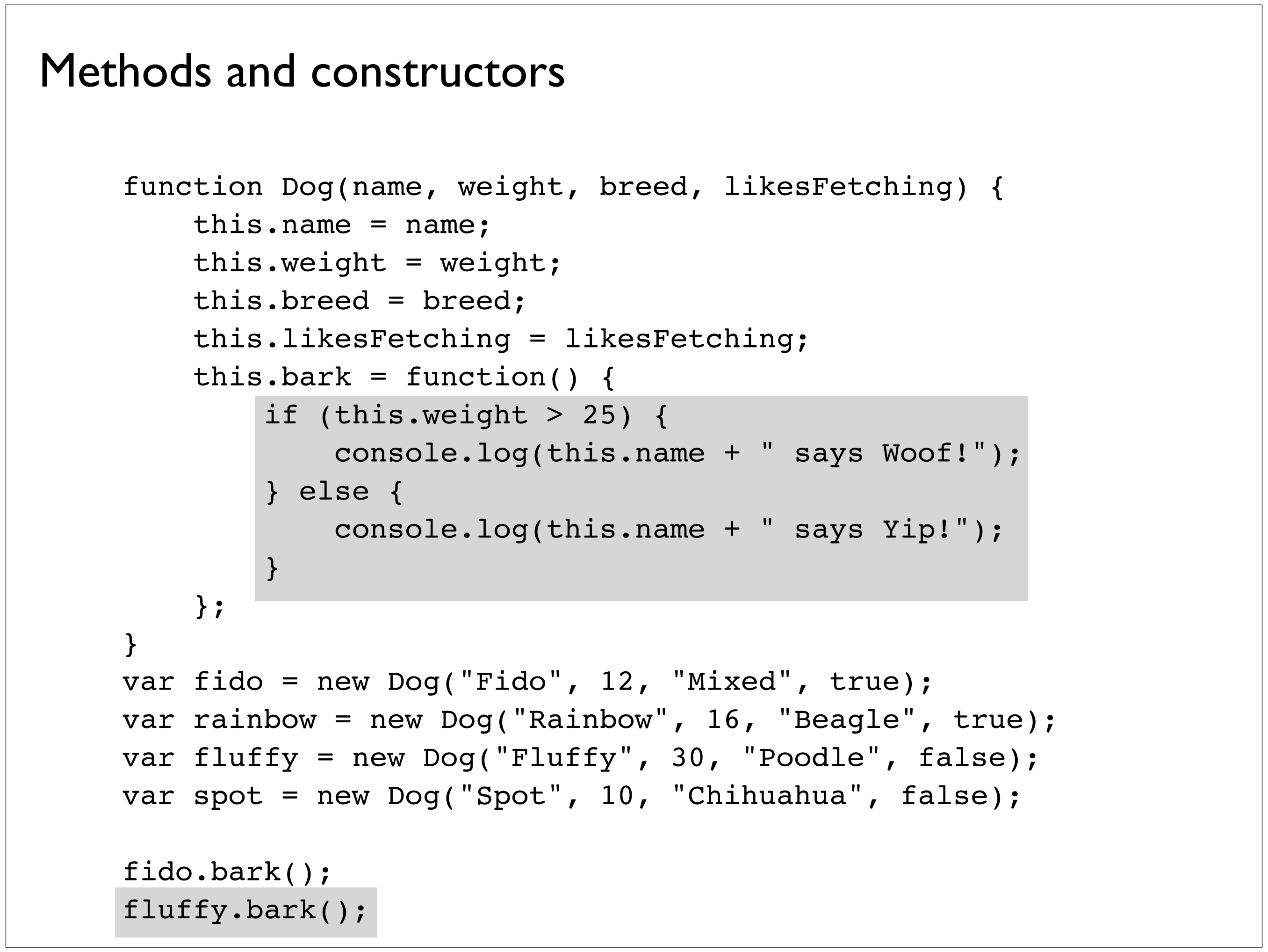
```
function Dog(name, weight, breed, likesFetching) {
    this.name = name;
    this.weight = weight;
    this.breed = breed;
    this.likesFetching = likesFetching;
}
```

Questions so far??

Methods and constructors

```
function Dog(name, weight, breed, likesFetching) {
    this.name = name;
    this.weight = weight;
    this.breed = breed;
    this.likesFetching = likesFetching;
    this.bark = function() {
        console.log("Woof, woof!");
    };
}
var fido = new Dog("Fido", 12, "Mixed", true);
var rainbow = new Dog("Rainbow", 16, "Beagle", true);
var fluffy = new Dog("Fluffy", 30, "Poodle", false);
var spot = new Dog("Spot", 10, "Chihuahua", false);

fido.bark();
rainbow.bark();
```

try adding this code



Methods and constructors

```
function Dog(name, weight, breed, likesFetching) {
    this.name = name;
    this.weight = weight;
    this.breed = breed;
    this.likesFetching = likesFetching;
    this.bark = function() {
        if (this.weight > 25) {
            console.log(this.name + " says Woof!");
        } else {
            console.log(this.name + " says Yip!");
        }
    };
}
var fido = new Dog("Fido", 12, "Mixed", true);
var rainbow = new Dog("Rainbow", 16, "Beagle", true);
var fluffy = new Dog("Fluffy", 30, "Poodle", false);
var spot = new Dog("Spot", 10, "Chihuahua", false);

fido.bark();
fluffy.bark();
```

Just like in object literals, if we want to refer to the object whose method we called,
we use "this" in the body of the method.

Question: what's the difference between the "this" used to initialize the properties, and the "this" in the body of the method???

Think about it while you update your code and try it...

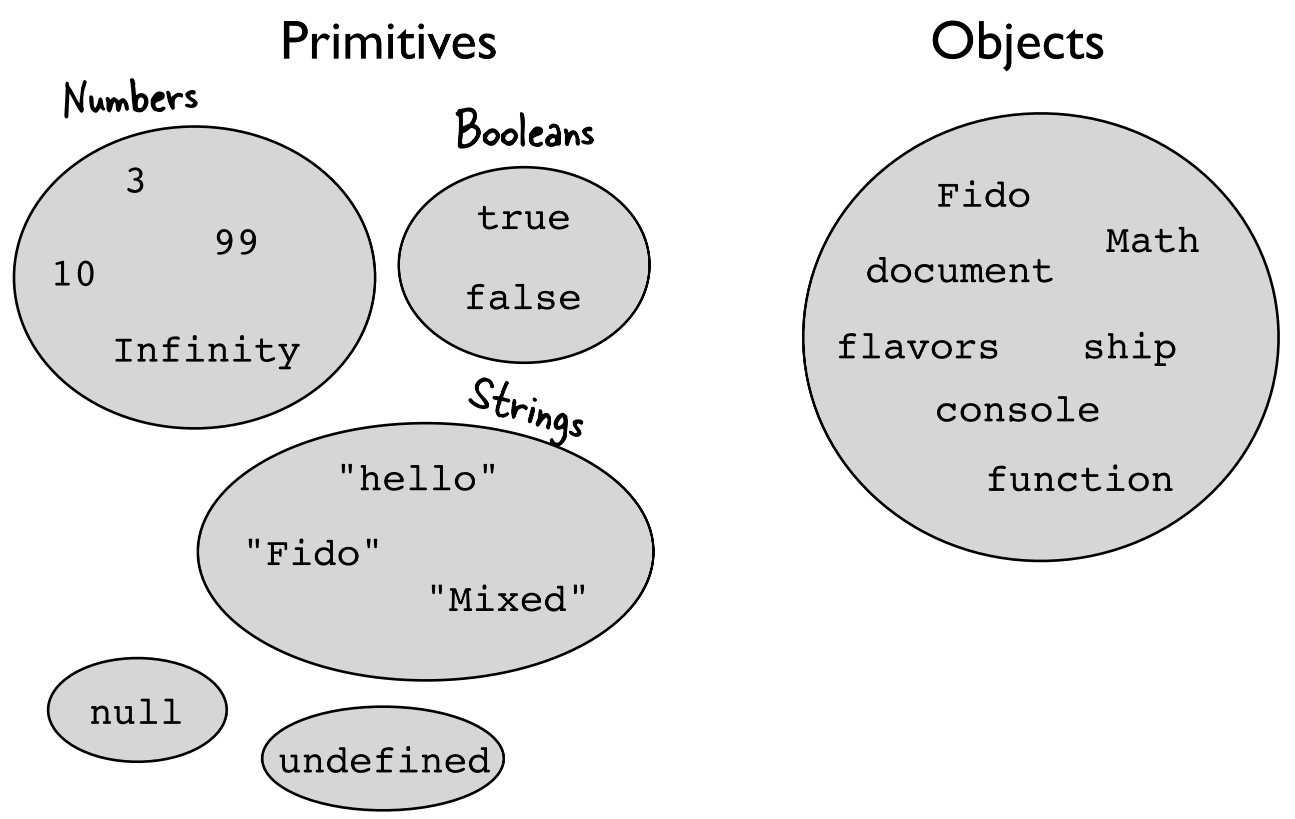
Your turn: create a constructor for a todo object that has the following properties:

- task: this will be a string, passed into the constructor
- who: also a string, also passed into the constructor
- done: a boolean, initially false
- isDone: a method that returns the value of done
- setDone: sets the value of done to true

Create a few todo objects.
Try using the methods.

Types

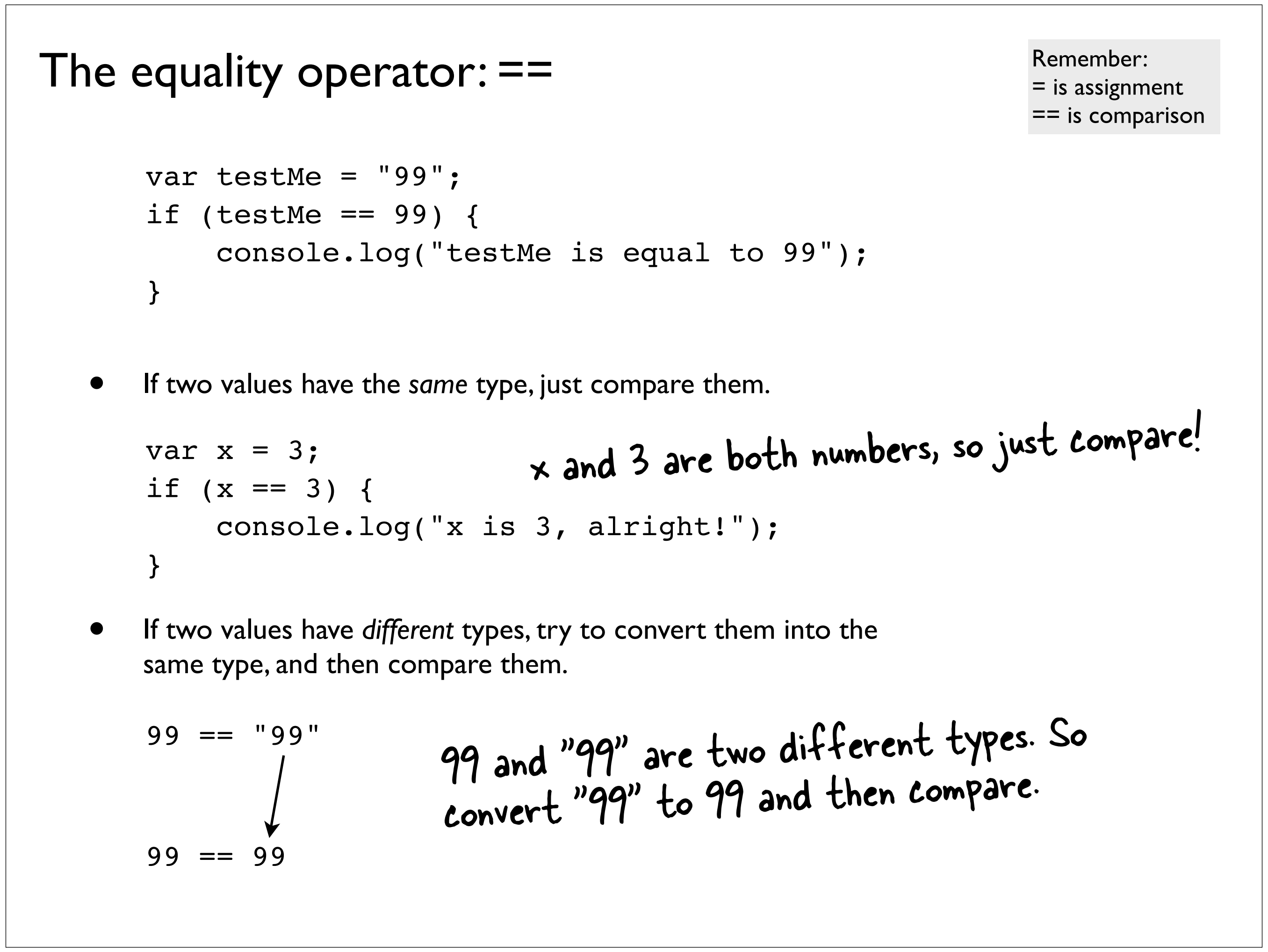
Every value in JavaScript is a primitive or an object.



Your turn!

```
var test1 = "abcdef";
var test2 = 123;
var test3 = true;
var test4 = {};
var test5 = [];
var test6;
var test7 = {"abcdef": 123};
var test8 = ["abcdef", 123];
function test9(){
    return "abcdef"
};
var test10 = null;
console.log("test1: ", typeof test1);
console.log("test2: ", typeof test2);
console.log("test3: ", typeof test3);
console.log("test4: ", typeof test4);
console.log("test5: ", typeof test5);
console.log("test6: ", typeof test6);
console.log("test7: ", typeof test7);
console.log("test8: ", typeof test8);
console.log("test9: ", typeof test9);
console.log("test10: ", typeof test10);
```

Type conversion



The equality operator: ==

Remember:
= is assignment
== is comparison

```
var testMe = "99";
if (testMe == 99) {
    console.log("testMe is equal to 99");
}
```

- If two values have the *same* type, just compare them.

```
var x = 3;
if (x == 3) {
    console.log("x is 3, alright!");
}
```

x and 3 are both numbers, so just compare!

- If two values have *different* types, try to convert them into the same type, and then compare them.

```
99 == "99"

99 == 99
```

99 and "99" are two different types. So convert "99" to 99 and then compare.

When you compare two different types of values using the equality operator (==),
JavaScript tries to convert the values to the same type before doing the comparison.

These rules describe how JavaScript does this.
Note, you're not doing these conversions; JavaScript is doing it behind the scenes.
Also, the values you have stored in variables don't change when this happens; again, all this happens behind the scenes.

What are the rules?

1. Comparing a number and a string? Convert the string into a number and then compare.

$$\begin{tikzcd}
\texttt{99 == "99"} \arrow[d] & \texttt{99 == "vanilla"} \arrow[d] \\
\texttt{99 == 99} \arrow[d] & \texttt{99 == NaN} \arrow[d] \\
\texttt{true} & \texttt{false}
\end{tikzcd}$$

2. Comparing a boolean with any other type? Convert the boolean to a number, and then compare.

$$\begin{tikzcd}
\texttt{1 == true} \arrow[d] & \texttt{"1" == true} \arrow[d] \\
\texttt{1 == 1} \arrow[d] & \texttt{"1" == 1} \arrow[d] \\
\texttt{true} & \texttt{1 == 1} \arrow[d] \\
 & \texttt{true}
\end{tikzcd}$$

true is converted to 1
false is converted to 0

3. Comparing null and undefined? This always evaluates to true (when using ==).

A few loose ends

The same conversions happen when you use >=, <=, and !=

```
1 == ""
       ↓
99 == 0
    ↓
 false
```

"" is the empty string (there is nothing in it, not even a space).

```
"pizza" + 3              "3" + 4
           ↓                    ↓
"pizza" + "3"            "3" + "4"
         ↓                   ↓
    "pizza3"               "34"
```

+ defaults to concatenation (not addition) when there is a string in the expression

```
true * 99                "3" * 4
 ↓                        ↓
1 * 99                   3 + 4
 ↓                        ↓
99                        12
```

"But I don't want my types to convert!"

Then use *strict equality*: ===
One, easy rule:

- Two values are strictly equal only if they have the same type *and* the same value.

  So...

```
99 == "99"              99 === "99"
  ↓                        ↓
 true          but       false

null == undefined      null === undefined
       ↓                      ↓
     true        but        false
```

There is !==

But there is NOT <== and >==

Remember: null and undefined have different types!

=== is going to look weird at first!

Use == if you WANT type conversion to happen (sometimes this is convenient).
Use === if you DO NOT WANT type conversion to happen (you're in complete control).
Some people say always use === to compare values.
I say, depends on the situation.

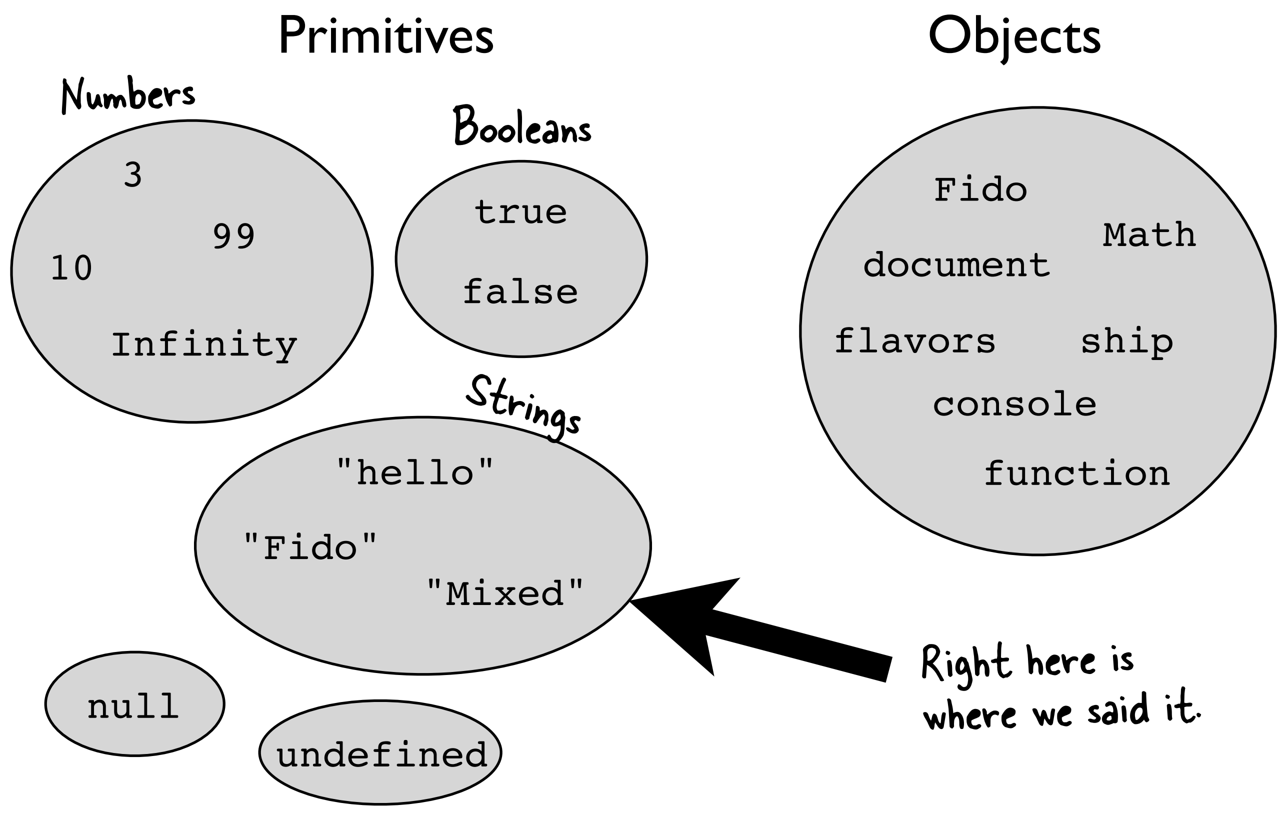
Your turn: Which of these expressions are true and which are false? Why? Make sure you understand why...

```
"42" == 42

"42" === 42

"0" == 0

"0" === 0

"0" == false

"0" === false

"true" == true

"true" === true

true == (1 == "1")

true === (1 === "1")
```

Strings

Photo: https://www.flickr.com/photos/jason-samfield/5169617997/

You probably noticed earlier, we said strings are primitives:

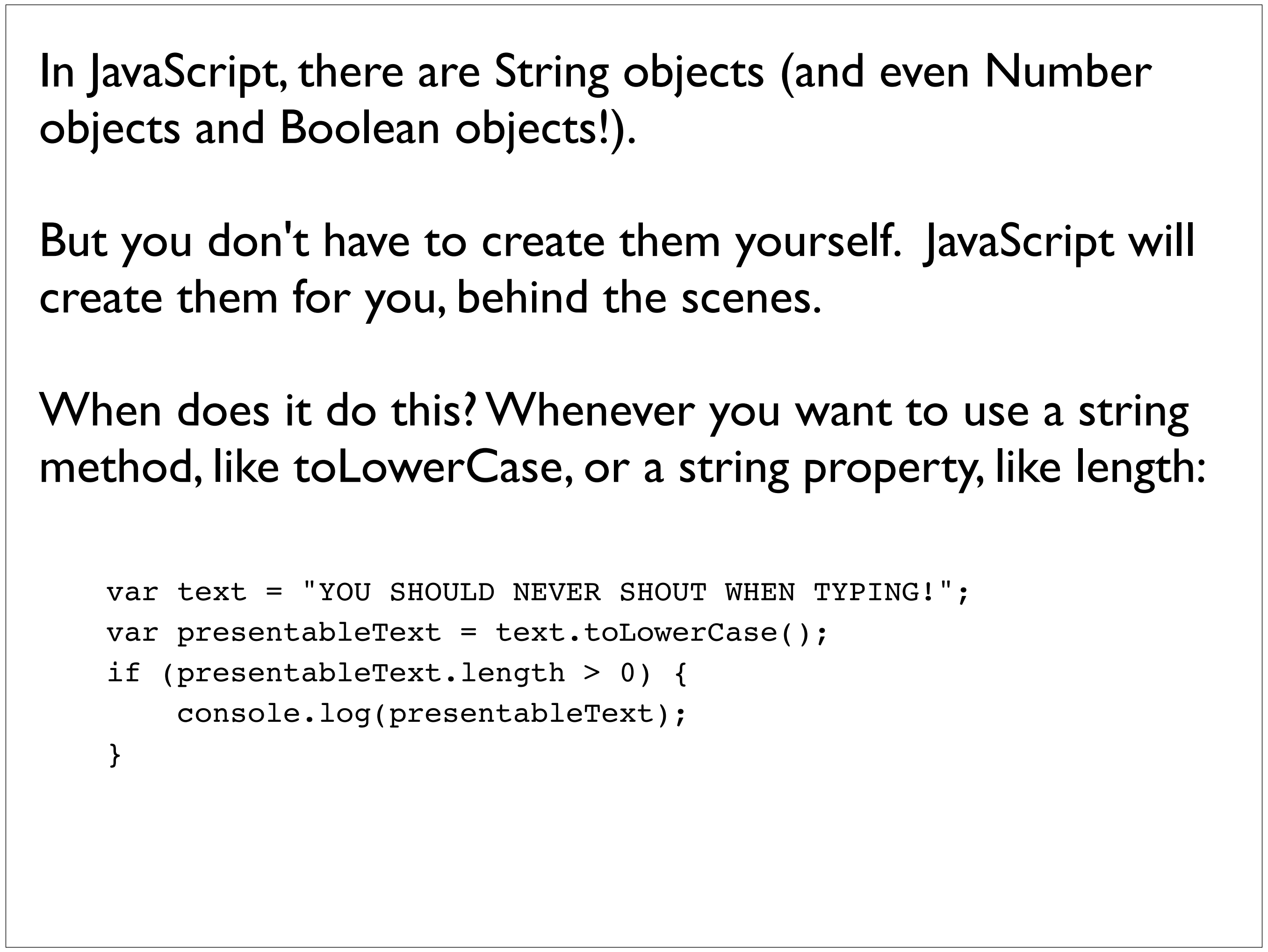


But strings actually have a double life...

...they're also objects! (sometimes). Try this:

```
var text = "YOU SHOULD NEVER SHOUT WHEN TYPING!";
var presentableText = text.toLowerCase();
if (presentableText.length > 0) {
    console.log(presentableText);
}
```

These look like object method calls, don't they?



In JavaScript, there are String objects (and even Number objects and Boolean objects!).

But you don't have to create them yourself. JavaScript will create them for you, behind the scenes.

When does it do this? Whenever you want to use a string method, like toLowerCase, or a string property, like length:

```
var text = "YOU SHOULD NEVER SHOUT WHEN TYPING!";
var presentableText = text.toLowerCase();
if (presentableText.length > 0) {
    console.log(presentableText);
}
```

To see what properties a string has, try this:

Type:
var myString = "Any string you like";
myString.

After you type the period, you'll see a popup list of all the properties a string has.

Here's how it works.

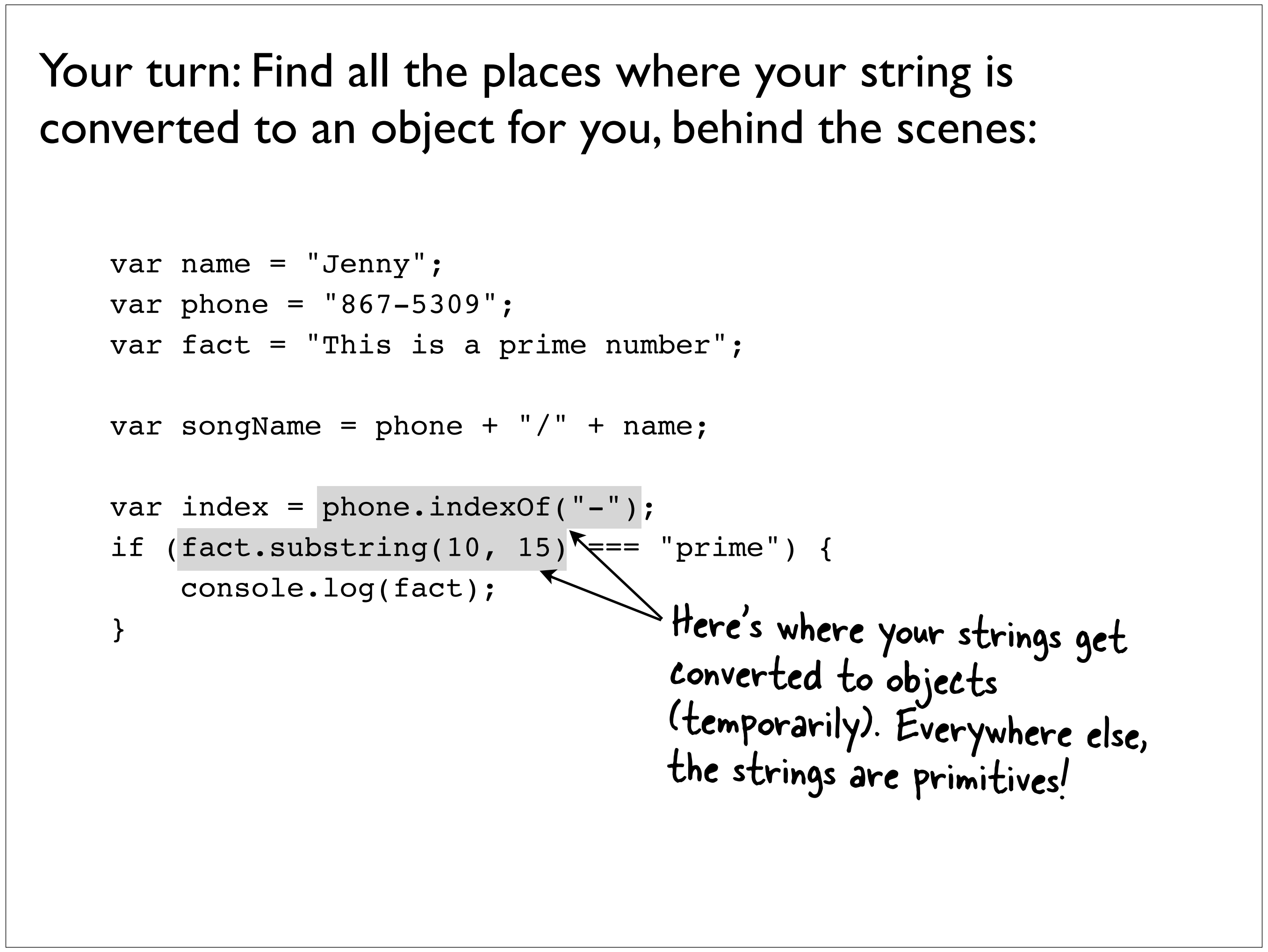
```
var text = "YOU SHOULD NEVER SHOUT WHEN TYPING!";
var presentableText = text.toLowerCase();
if (presentableText.length > 0) {
    console.log(presentableText);
}
```

1. We declare a variable text, which is a primitive string.
2. We call a string method toLowerCase on text.
3. JavaScript converts the primitive string, text, into a temporary String object.
4. The method is called, which converts all the characters into lowercase.
5. Once the method has been called, text is converted back to a primitive string again.

NOTE: Steps 3, 4, 5 all happen behind the scenes! You don't have to worry about it.

Your turn: Find all the places where your string is converted to an object for you, behind the scenes:

```
var name = "Jenny";
var phone = "867-5309";
var fact = "This is a prime number";

var songName = phone + "/" + name;

var index = phone.indexOf("-");
if (fact.substring(10, 15) === "prime") {
    console.log(fact);
}
```



Your turn: Find all the places where your string is converted to an object for you, behind the scenes:

```
var name = "Jenny";
var phone = "867-5309";
var fact = "This is a prime number";

var songName = phone + "/" + name;

var index = phone.indexOf("-");
if (fact.substring(10, 15) === "prime") {
    console.log(fact);
}
```

Here's where your strings get converted to objects (temporarily). Everywhere else, the strings are primitives!

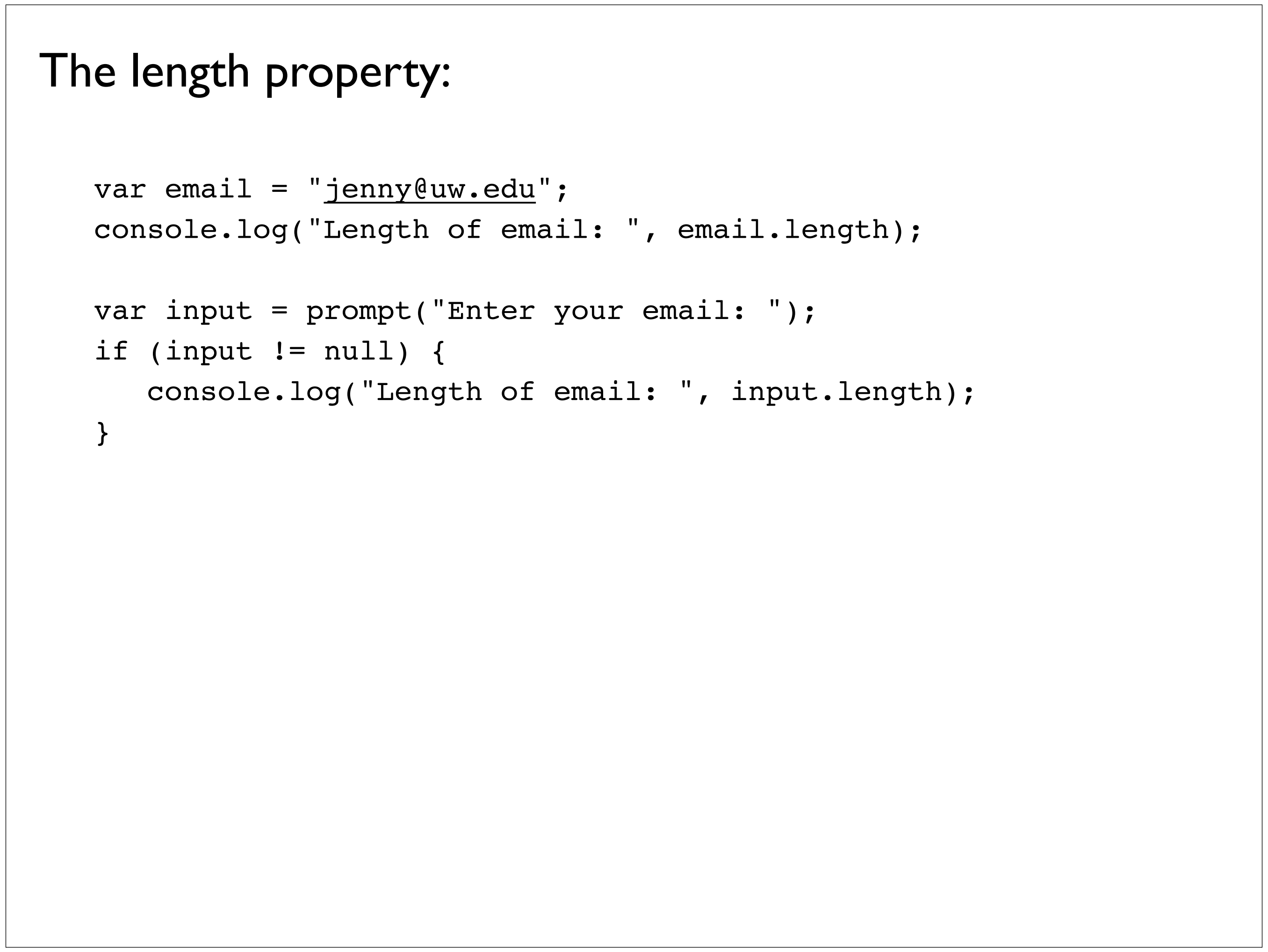
Questions??

Note that while numbers and booleans can also be converted to objects when necessary, in practice you'll rarely need to use numbers or booleans as objects (you won't find you'll use their properties that often).

But string properties are incredibly handy for manipulating and processing strings. We'll look at a few of the most useful ones next...

Useful string properties (ones you'll use often):

- length: this property holds the length of a string.
- charAt: this method takes an integer and returns a string containing the single character at that position in the string (think of the string like an array of characters).
- indexOf: this method takes a string, and returns the index of the first character of the first occurrence of that argument in the string.
- substring: give the substring method two indices, and it will extract and return the string contained within them.
- split: this method takes a character that acts as a delimiter, and breaks the string into parts based on the delimiter (and returns an array containing these parts).

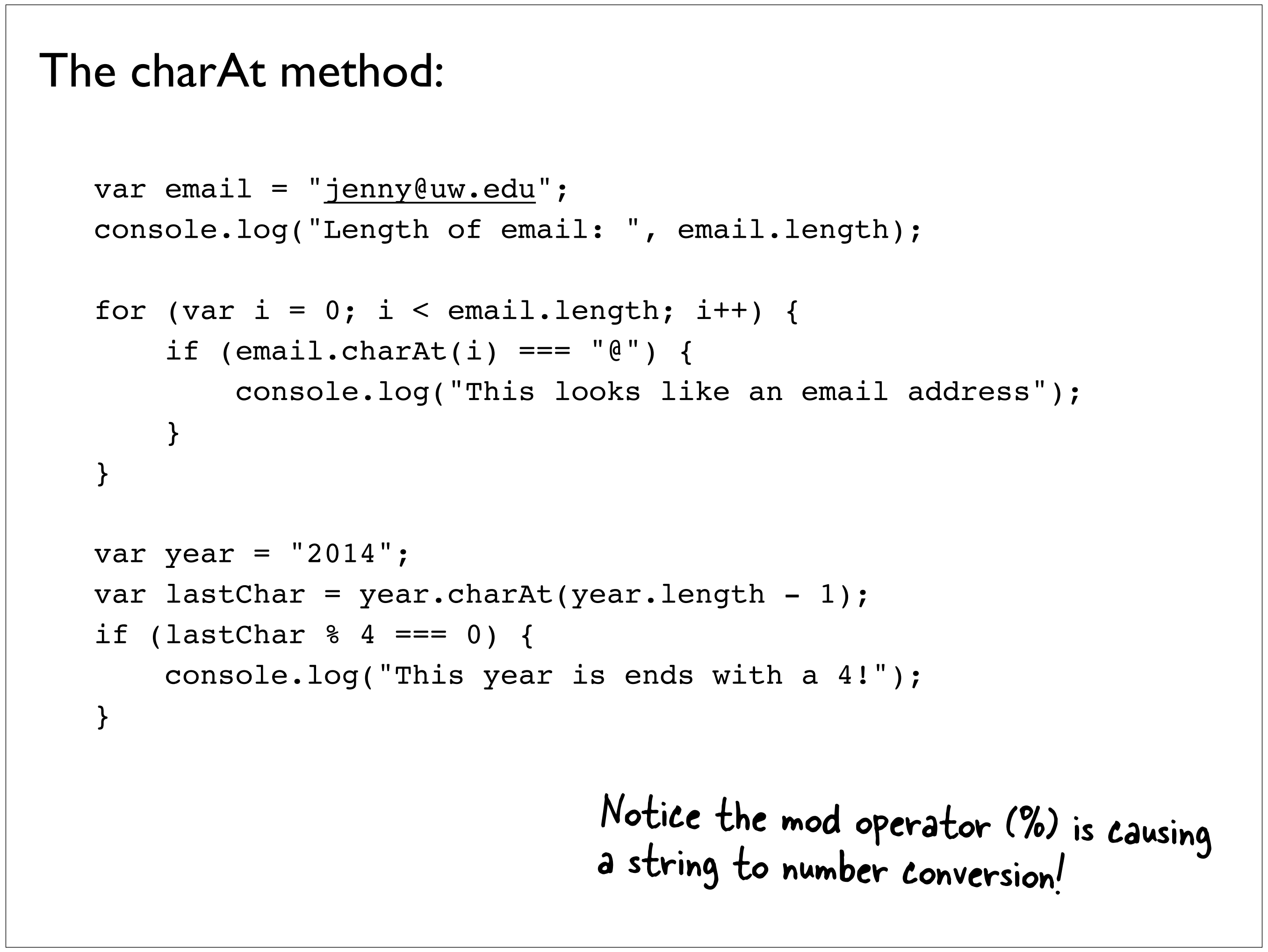


The length property:

```
var email = "jenny@uw.edu";
console.log("Length of email: ", email.length);

var input = prompt("Enter your email: ");
if (input != null) {
   console.log("Length of email: ", input.length);
}
```

Try typing in this code.

Experiment with length.



The charAt method:

```
var email = "jenny@uw.edu";
console.log("Length of email: ", email.length);

for (var i = 0; i < email.length; i++) {
    if (email.charAt(i) === "@") {
        console.log("This looks like an email address");
    }
}

var year = "2014";
var lastChar = year.charAt(year.length - 1);
if (lastChar % 4 === 0) {
    console.log("This year is ends with a 4!");
}
```

Notice the mod operator (%) is causing a string to number conversion!

Try typing in this code.

The indexOf method:

```
var phrase = "the cat in the hat";

var index = phrase.indexOf("cat");
console.log("Index of 'cat': ", index);

index = phrase.indexOf("the");
console.log("(First) Index of 'the': ", index);

index = phrase.indexOf("the", 5);
console.log("Index of 'the' (appearing after index 5): ", index);

index = phrase.indexOf("dog");
console.log("Index of 'dog': ", index);
```

Try typing in this code.

The substring method:

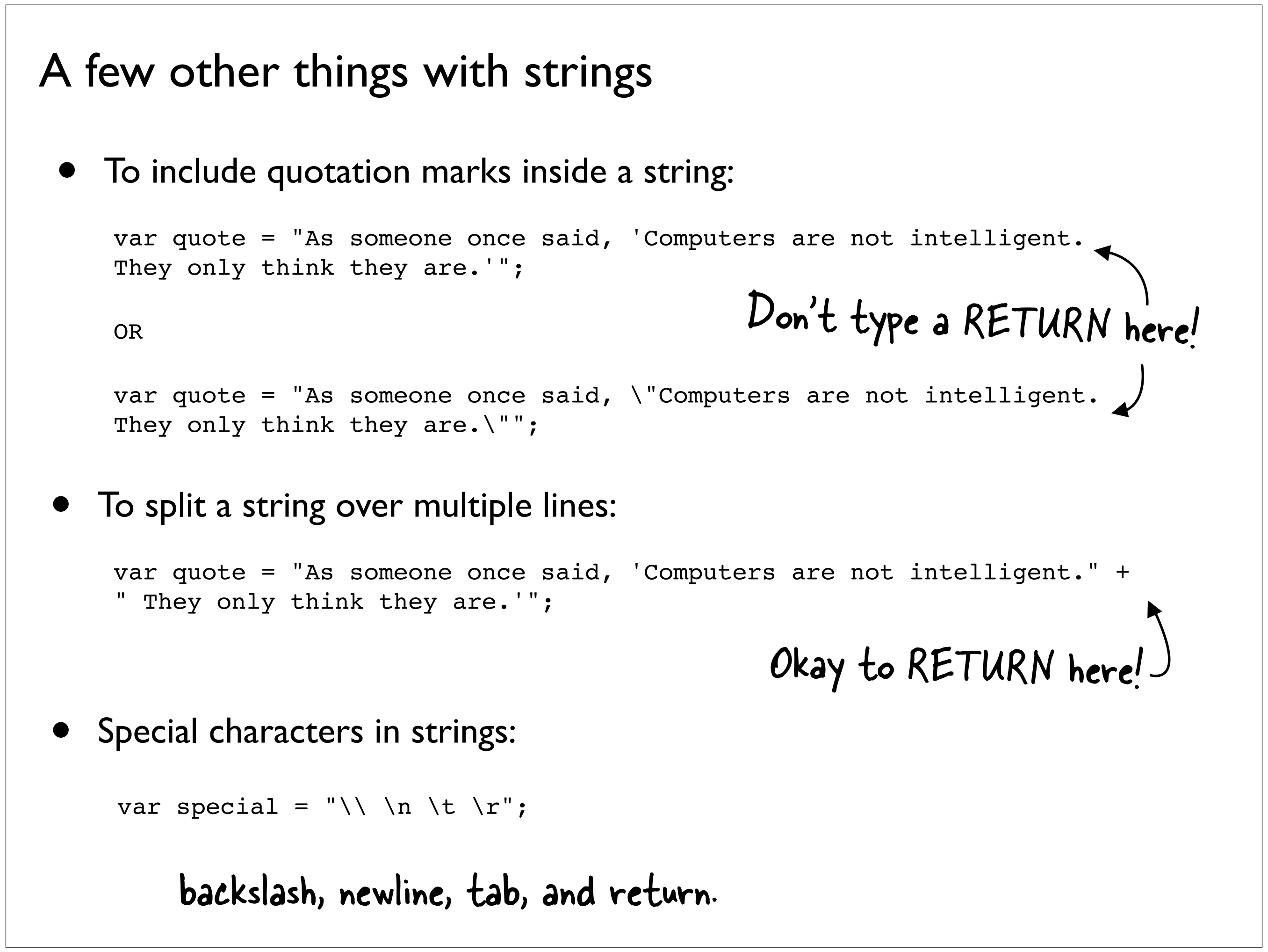
```
var data = "name|phone|address";
var val = data.substring(5, 10);
console.log("Substring from 5 to 10 is: ", val);

val = data.substring(11);
console.log("Substring from 11 to the end is: ", val);
```

Try typing in this code.

The split method:

```
var data = "name|phone|address";
var vals = data.split("|");
console.log("Split array is ", vals);

var sentence = "I love icecream";
var words = sentence.split(" ");
console.log("Words array is ", words);
```

Try typing in this code.



A few other things with strings

- To include quotation marks inside a string:

```
var quote = "As someone once said, 'Computers are not intelligent.
They only think they are.'";

OR

var quote = "As someone once said, \"Computers are not intelligent.
They only think they are.\"";
```

Don't type a RETURN here!

- To split a string over multiple lines:

```
var quote = "As someone once said, 'Computers are not intelligent." +
" They only think they are.'";
```

Okay to RETURN here!

- Special characters in strings:

```
var special = "\\ \n \t \r";
```

backslash, newline, tab, and return.

Try typing this in the console.

You probably won't use special characters very often but there are quite a few if you need them (a good reference is handy!).

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String

Comparing strings

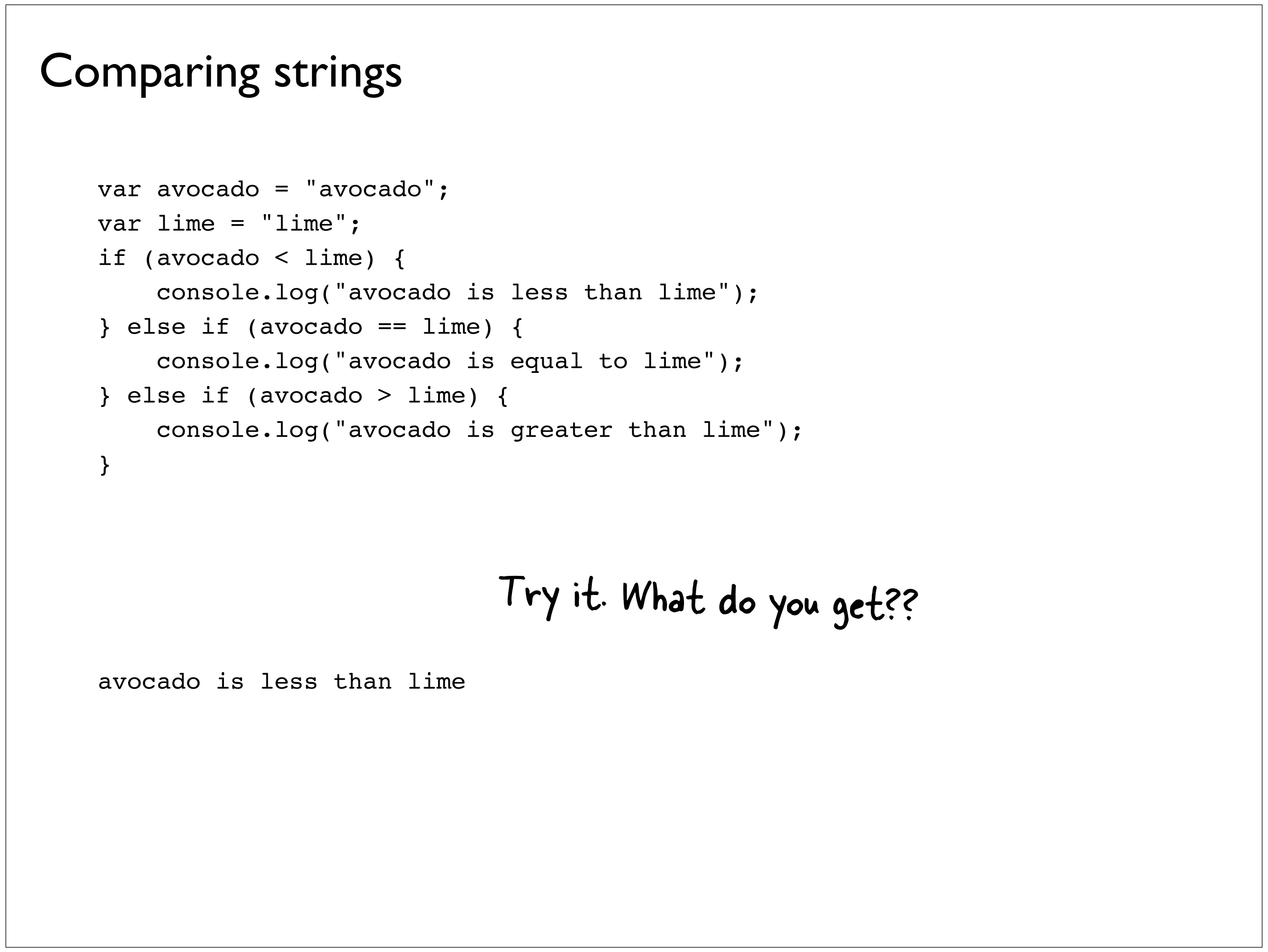
```
var avocado = "avocado";
var lime = "lime";
if (avocado < lime) {
    console.log("avocado is less than lime");
} else if (avocado == lime) {
    console.log("avocado is equal to lime");
} else if (avocado > lime) {
    console.log("avocado is greater than lime");
}
```

Try it. What do you get??

Comparing strings with the comparison operators (==, <=, >=)
The way strings are compared in JavaScript is with the ordering that characters appear in the Unicode specification.

alphabetical: a is less than z
lower case is greater than upper case! (not necessarily intuitive).

If you want to change the basic Unicode ordering (e.g. for sorting strings) you have to implement it yourself!
Just to keep in mind.
We'll focus on the simple cases here.



Comparing strings

```
var avocado = "avocado";
var lime = "lime";
if (avocado < lime) {
    console.log("avocado is less than lime");
} else if (avocado == lime) {
    console.log("avocado is equal to lime");
} else if (avocado > lime) {
    console.log("avocado is greater than lime");
}
```

Try it. What do you get??

```
avocado is less than lime
```

Comparing strings with the comparison operators (==, <=, >=)
The way strings are compared in JavaScript is with the ordering that characters appear in the Unicode specification.

alphabetical: a is less than z
lower case is greater than upper case! (not necessarily intuitive).

If you want to change the basic Unicode ordering (e.g. for sorting strings) you have to implement it yourself!
Just to keep in mind.
We'll focus on the simple cases here.

Comparing strings

```
var lime = "lime";

if (lime == "Lime") {
    console.log("lime is equal to Lime");
} else if (lime < "Lime") {
    console.log("lime is less than Lime");
} else if (lime > "Lime") {
    console.log("lime is greater than Lime");
}
```

Try it. What do you get??

Comparing strings with the comparison operators (==, <=, >=)
The way strings are compared in JavaScript is with the ordering that characters appear in the Unicode specification.

alphabetical: a is less than z
lower case is greater than upper case! (not necessarily intuitive).

If you want to change the basic Unicode ordering (e.g. for sorting strings) you have to implement it yourself!
Just to keep in mind.
We'll focus on the simple cases here.

Comparing strings

```
var lime = "lime";

if (lime == "Lime") {
    console.log("lime is equal to Lime");
} else if (lime < "Lime") {
    console.log("lime is less than Lime");
} else if (lime > "Lime") {
    console.log("lime is greater than Lime");
}
```

Try it. What do you get??

```
lime is greater than Lime
```

Comparing strings with the comparison operators (==, <=, >=)
The way strings are compared in JavaScript is with the ordering that characters appear in the Unicode specification.

alphabetical: a is less than z
lower case is greater than upper case! (not necessarily intuitive).

If you want to change the basic Unicode ordering (e.g. for sorting strings) you have to implement it yourself!
Just to keep in mind.
We'll focus on the simple cases here.

Project

Todo list manager - Silver

- Function to create and return a to do object.
- An array, todoList, to store all your todo items. Hint: you can store objects in an array!
- createNewTodo function to prompt the user for a todo item.
- displayTodoList function to display all your todos.
- A loop to get todo items from the user until they hit "cancel" on prompt. Hint: test to see what is returned by prompt when you enter nothing, when you enter a string, and when you hit cancel.

Todo list manager - Platinum

- Constructor to create new todo items (you've already started this).
- An array, todoList, to store all your todo items. Hint: you can store objects in an array!
- createNewTodo function to prompt the user for a todo item.
- displayTodoList function to display all your todos.
- A loop to get todo items from the user until they hit "cancel" on prompt. Hint: test to see what is returned by prompt when you enter nothing, when you enter a string, and when you hit cancel.