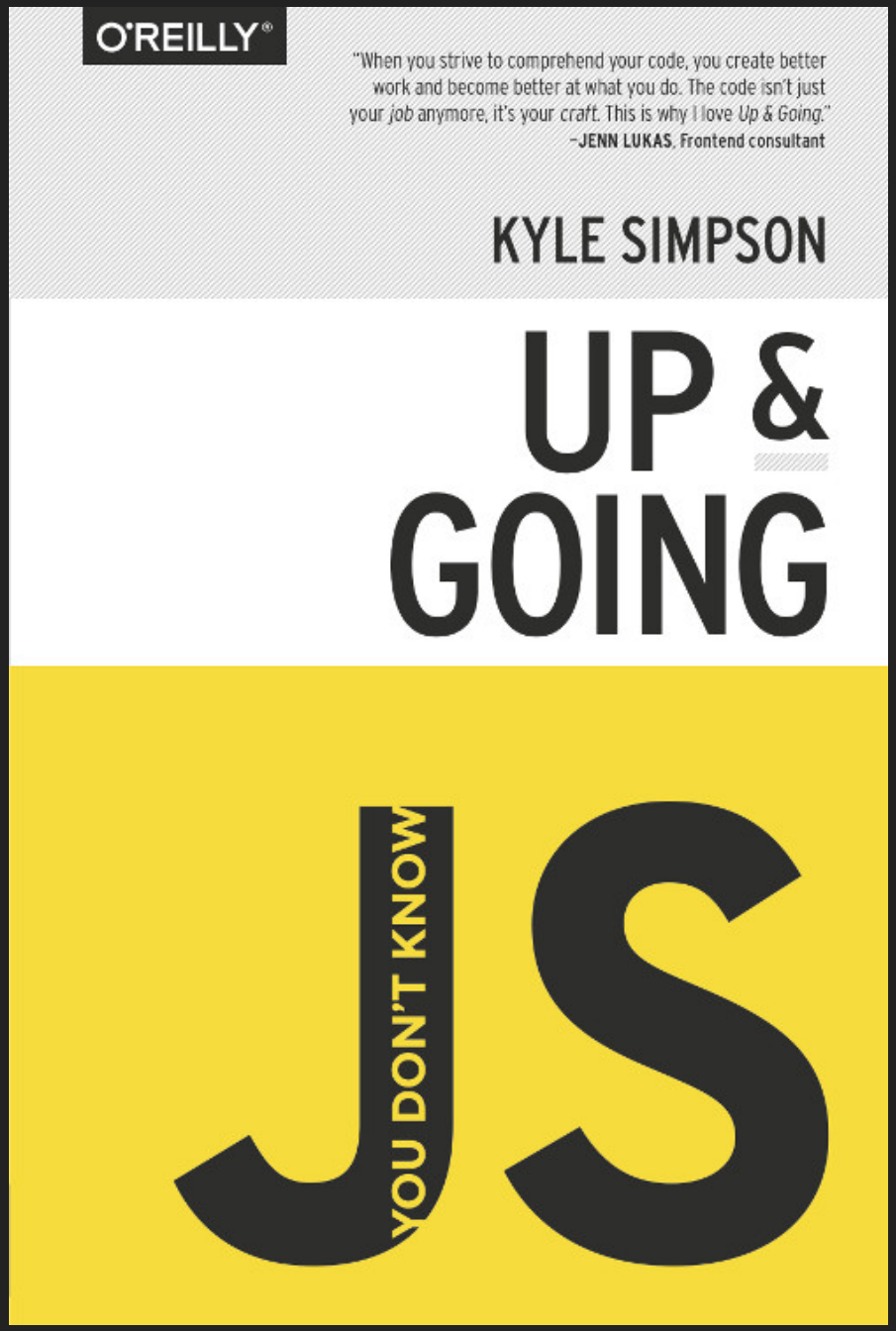
KYLE SIMPSON GETIFY@GMAIL.COM

GETTING INTO JS



https://github.com/getify/You-Dont-Know-JS

```
1 var teacher = "Kyle";
2 var twitterHandle = "getify";
3 var age = 39;
4
5 function whoAmI(myName,myNickname,myAge) {
6     console.log(`
7         Hi, I'm ${myName} (aka ${myNickname}),
8         and I'm ${myAge} years old.
9     `);
10 }
11
12 whoAmI(teacher,twitterHandle,age);
```

Course Overview

- Programming Primer (in JS)
- Three Pillars of JS:
  - Types / Coercion
  - Scope / Closures
  - this / Prototypes

...but before we begin...

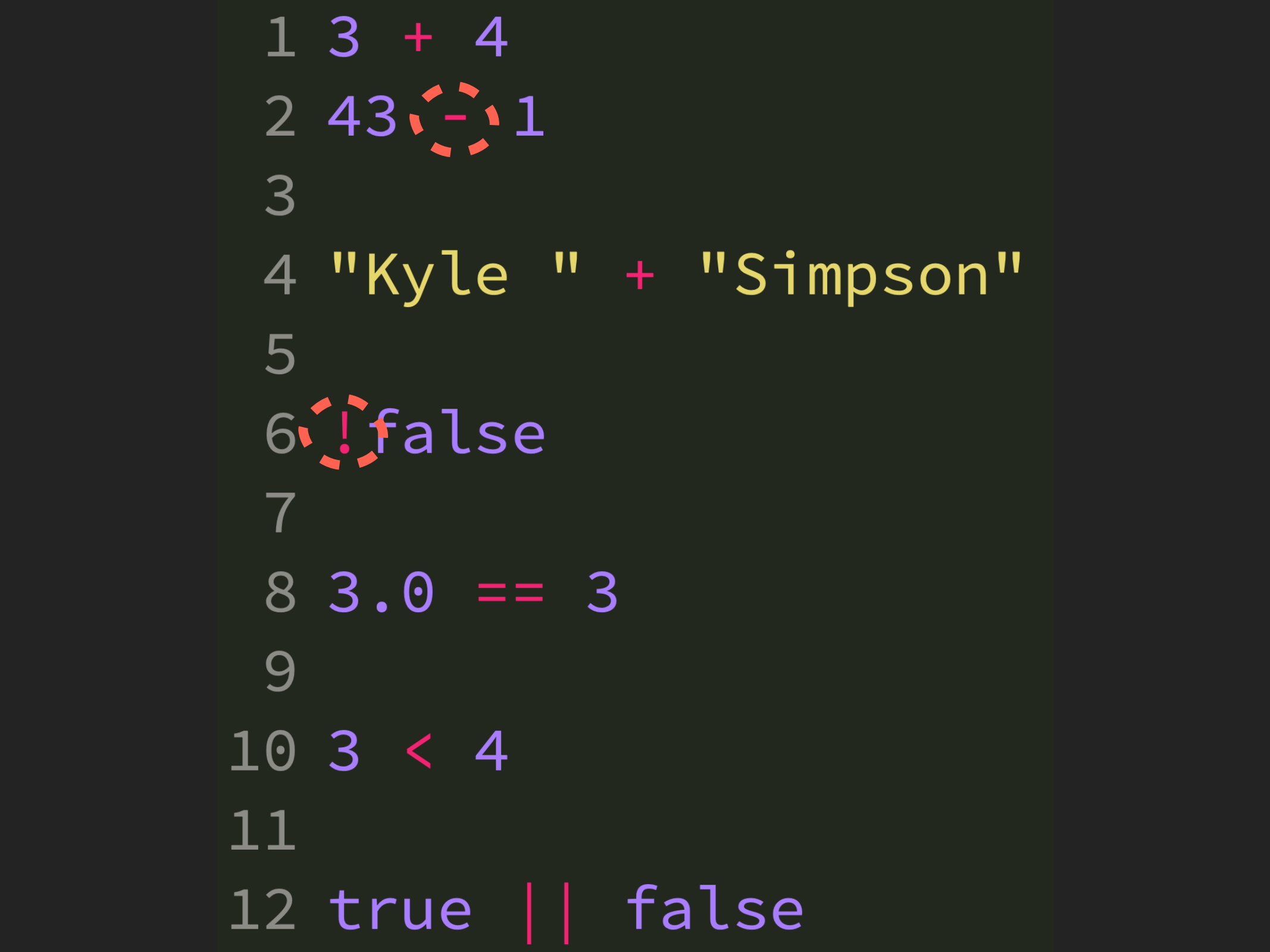
Programming Primer (in JS)

- Values
- Operations
- Variables
- Expressions and Statements
- Decisions
- Loops
- Functions

Values

```
42
3.14

"Hello, friend."

true
false

null
undefined

[ 1, 2, 3 ]
{ name: "Kyle" }
```

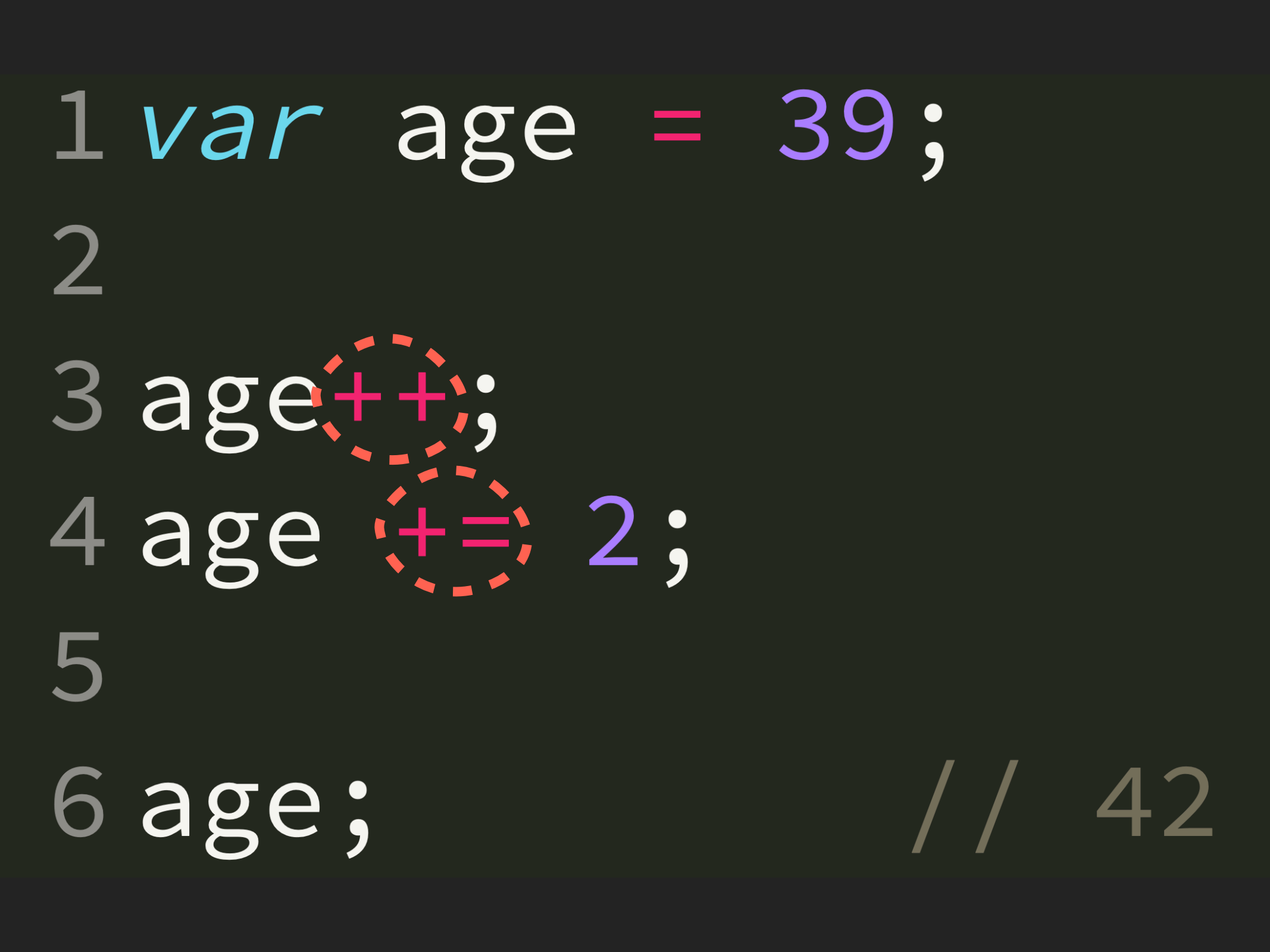
Operations


1 3 + 4
2 43 - 1
3
4 "Kyle " + "Simpson"
5
6 !false
7
8 3.0 == 3
9
10 3 < 4
11
12 true || false

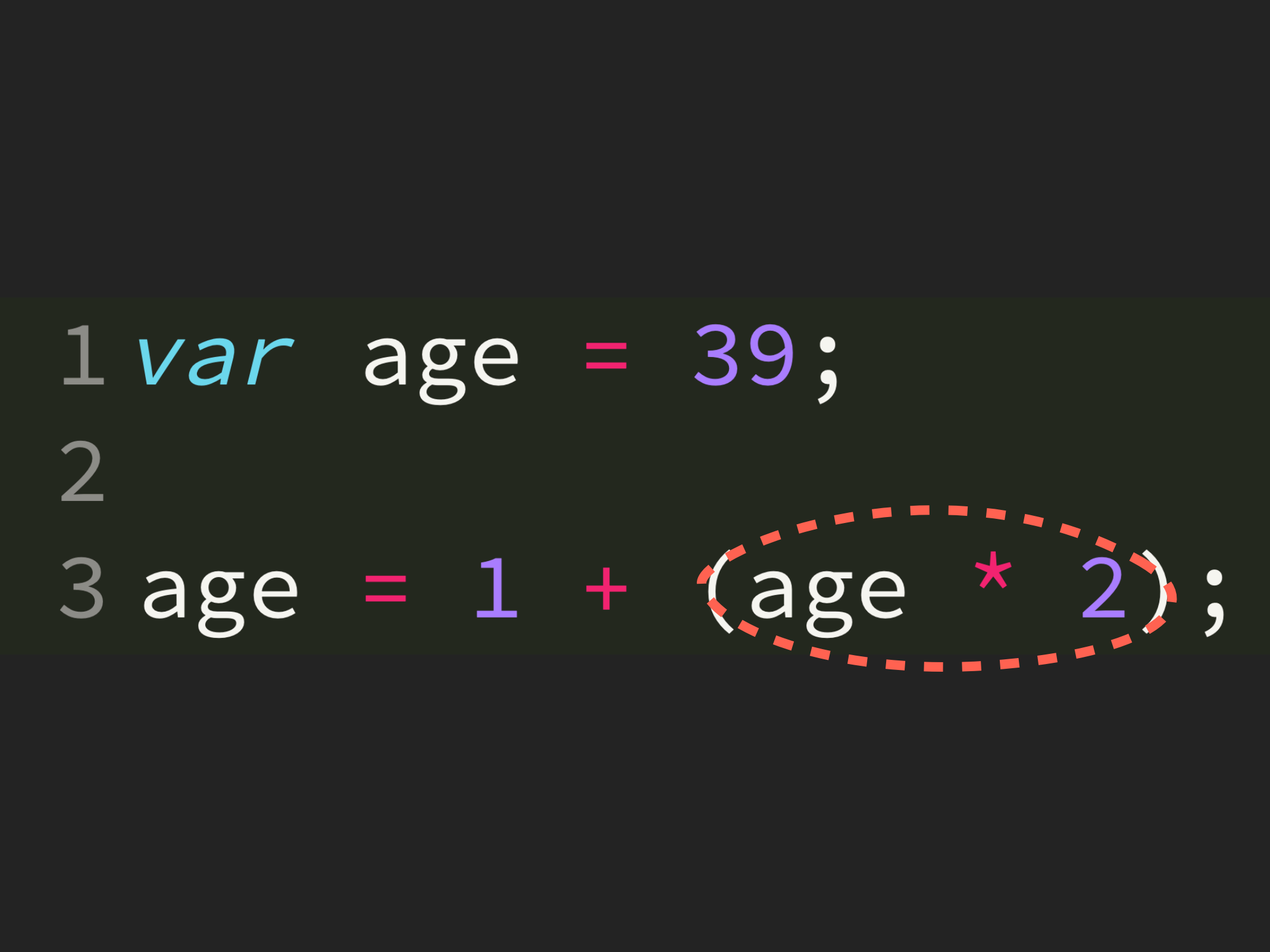
```
typeof 42               // "number"

typeof "Kyle"           // "string"

typeof true             // "boolean"

typeof undefined        // "undefined"

typeof { age: 39 }      // "object"


typeof null             // "object"  !?!?
typeof [1,2,3]          // "object"
```

Variables


var name = "Kyle Simpson";
var age;
age = 39;
var friends = ["Brandon", "Marc"];
console.log(friends.length);
console.log(friends[1]);


1 var age = 39;
2
3 age++;
4 age += 2;
5
6 age; // 42

Expressions and Statements


1 var age = 39;
2
3 age = 1 + (age * 2);

Decisions

```
1 var age = 39;
2
3 if (age >= 18) {
4     goVote();
5 }
```

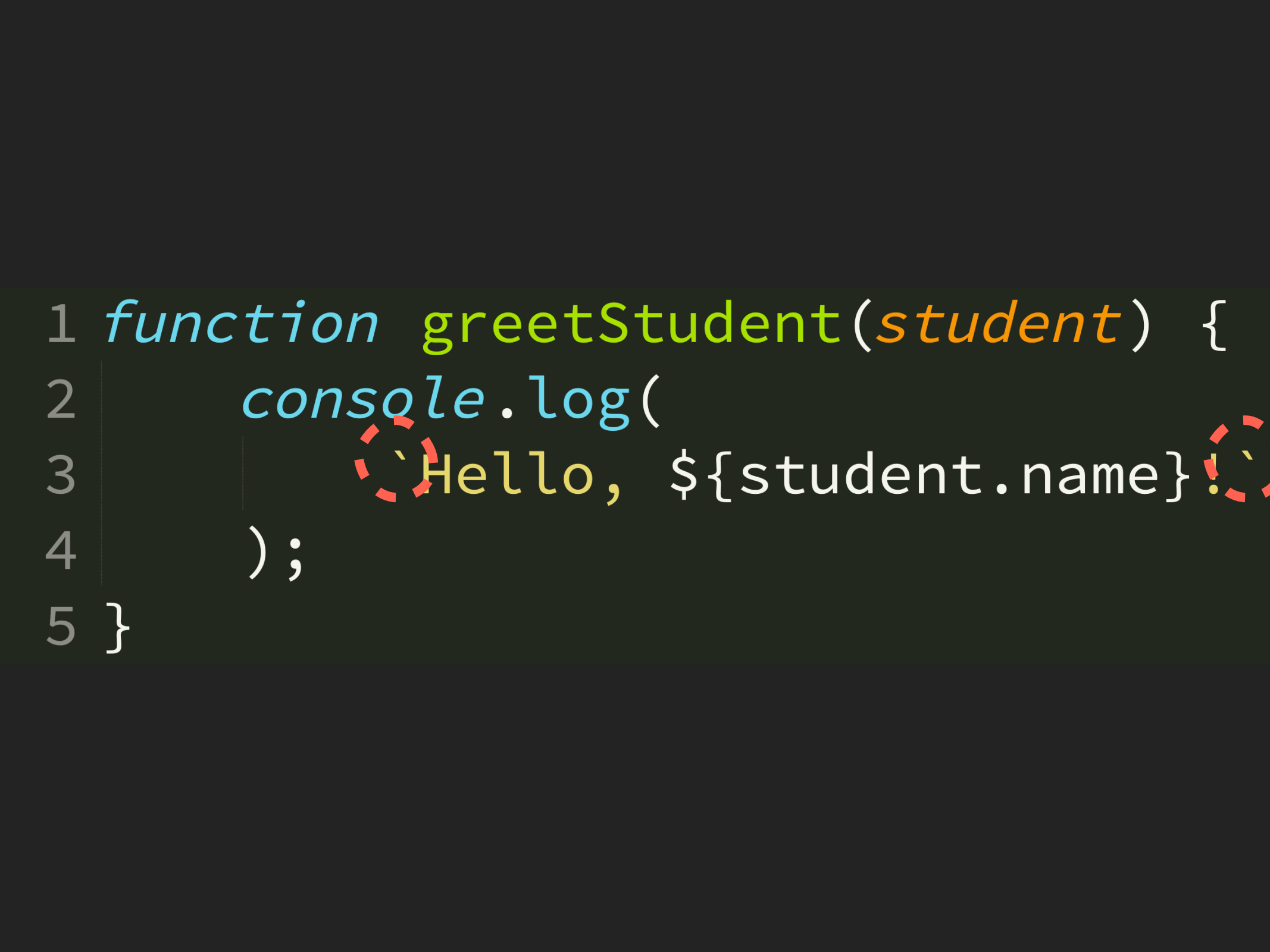
```
1 if (isEnrolled()) {
2     takeClass();
3 }
4 else {
5     enrollFirst();
6 }
```

Loops

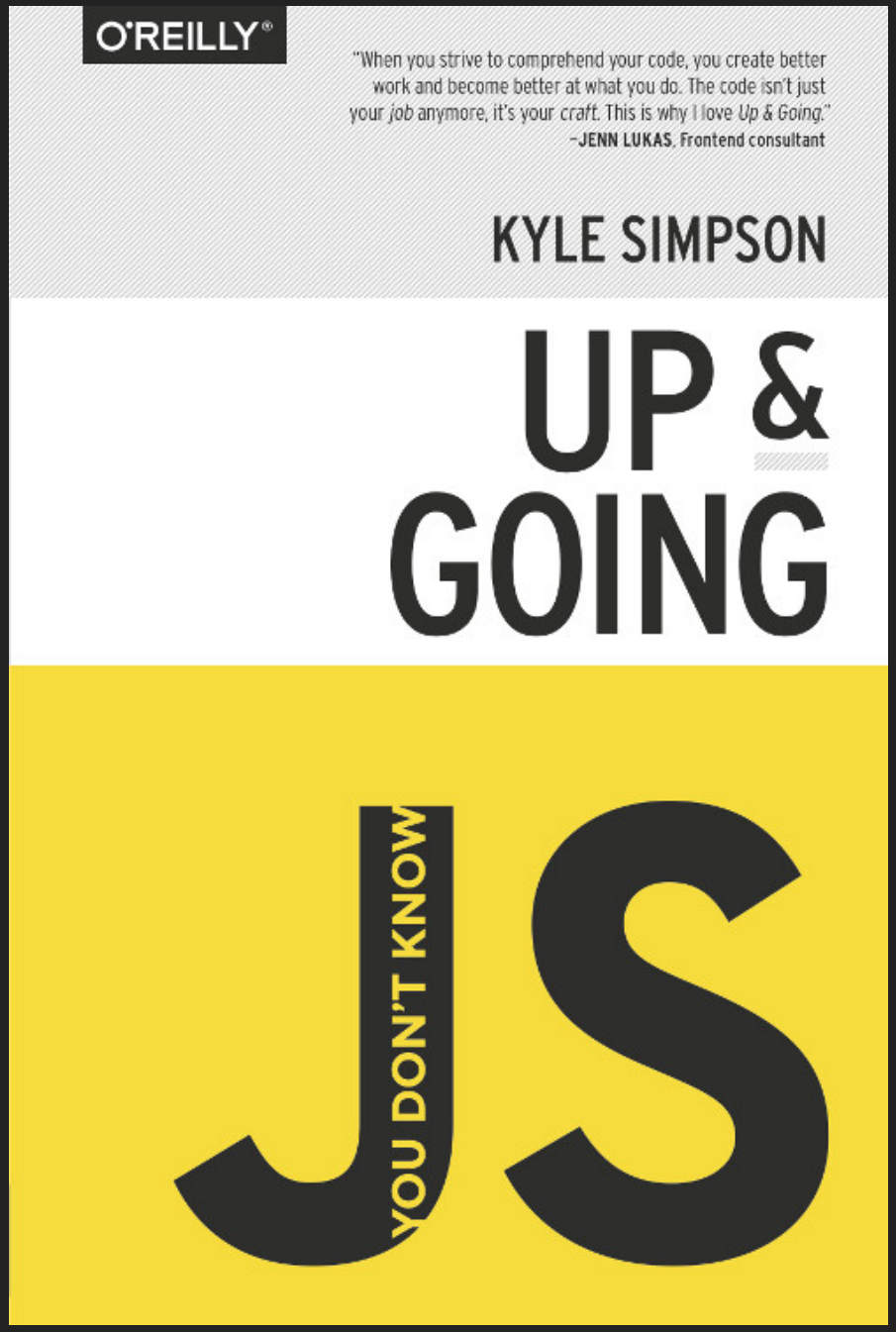
```
var students = [ /*..*/ ];

for (let i = 0; i < students.length; i++) {
    greetStudent( students[i] );
}

for (let student of students) {
    greetStudent( student );
}
```

```
1 var students = [ /*..*/ ];
2
3 while (students.length > 0) {
4     let student = students.pop();
5     greetStudent(student);
6 }
```

Functions


1 function greetStudent(student) {
2 console.log(
3 `Hello, ${student.name}!`
4);
5 }

```
1 function timeRemaining(timeElapsed,endTime) {
2     return endTime - timeElapsed;
3 }
4
5 var left = timeRemaining(42,240);
6
7 left;          // 198
```


O'REILLY®
"When you strive to comprehend your code, you create better work and become better at what you do. The code isn't just your *job* anymore, it's your *craft*. This is why I love *Up & Going*."
–JENN LUKAS, Frontend consultant
KYLE SIMPSON
UP & GOING
YOU DON'T KNOW
JS

Chapter 1

Three Pillars of JS

1. Types / Coercion
2. Scope / Closures
3. this / Prototypes

Types / Coercion

- Primitive Types
- Converting Types
- Checking Equality

Primitive Types

"In JavaScript, everything is an object."

false

- undefined
- string
- number
- boolean
- object
- symbol

- null?
- function?
- array?

Primitive Types

In JavaScript, variables don't have types, values do.

```
1  var v;
2  typeof v;                    // "undefined"
3  v = "1";
4  typeof v;                    // "string"
5  v = 2;
6  typeof v;                    // "number"
7  v = true;
8  typeof v;                    // "boolean"
9  v = {};
10 typeof v;                    // "object"
11 v = Symbol();
12 typeof v;                    // "symbol"
```

Primitive Types: typeof

```
 1 typeof doesntExist;       // "undefined"
 2
 3 var v = null;
 4 typeof v;                 // "object"  OOPS!
 5
 6 v = function(){};
 7 typeof v;                 // "function"  hmmm?
 8
 9 v = [1,2,3];
10 typeof v;                 // "object"  hmmm?
```

Primitive Types: typeof

NaN (“~~not a number~~”)

```
1 var greeting = "Hello, class!";
2
3 var something = greeting / 2;    // ?!?!?
4
5 something;                       // NaN
6 Number.isNaN( something );       // true
7
8 Number.isNaN( greeting );        // false
```

NaN

Use new:

- Object()
- Array()
- Function()
- Date()
- RegExp()
- Error()

Don't use new:

- String()
- Number()
- Boolean()

```
1 var yesterday = new Date("March 6, 2019");
2 yesterday.toUTCString();
3 // "Wed, 06 Mar 2019 06:00:00 GMT"
4
5 var myGPA = String(transcript.gpa);
6 // "3.54"
```

Fundamental Objects

Converting Types

The way to convert from one type to another: coercion

```
1 var msg1 = "There are ";
2 var numStudents = 16;
3 var msg2 = " students.";
4 console.log(msg1 + numStudents + msg2);
5 // "There are 16 students."
```

Coercion: string concatenation (number to string)

```
1 var numStudents = 16;
2
3 console.log(
4     `There are ${numStudents+""} students.`
5 );
6 // "There are 16 students."
```

Coercion: string concatenation (number to string)

Number + Number = Number

Number + String = String

String + Number = String

String + String = String

```
1 function addAStudent(numStudents) {
2     return numStudents + 1;
3 }
4
5 addAStudent(
6     Number(studentsInputElem.value)
7 );
8 // 17
```

Coercion: string to number

Falsy	Truthy
“”	“foo”
0, -0	23
null	{ a:1 }
NaN	[1,3]
false	true
undefined	function(){..}
	...

Coercion: boolean

```
1 if (studentsInputElem.value) {
2     numStudents =
3         Number(studentsInputElem.value);
4 }
```

```
1 while (newStudents.length) {
2     enrollStudent(newStudents.pop());
3 }
```

Coercion: boolean

```
1 if (!!studentsInputElem.value) {
2     numStudents =
3         Number(studentsInputElem.value);
4 }
```

```
1 while (newStudents.length > 0) {
2     enrollStudent(newStudents.pop());
3 }
```

Coercion: boolean

```
1  var workshopEnrollment1 = 16;
2  var workshopEnrollment2 = workshop2Elem.value;
3
4  if (Number(workshopEnrollment1) < Number(workshopEnrollment2)) {
5      // ..
6  }
7
8  if (workshopEnrollment1 < workshopEnrollment2) {
9      // ..
10 }
```

Coercion: implicit can be good (sometimes)

A quality JS program embraces coercions, making sure the types involved in every operation are clear.

~~"If a feature is sometimes useful and sometimes dangerous and if there is a better option then always use the better option."~~

-- "The Good Parts", Crockford

Useful: when the reader is focused on what's important

Dangerous: when the reader can't tell what will happen

Better: when the reader understands the code

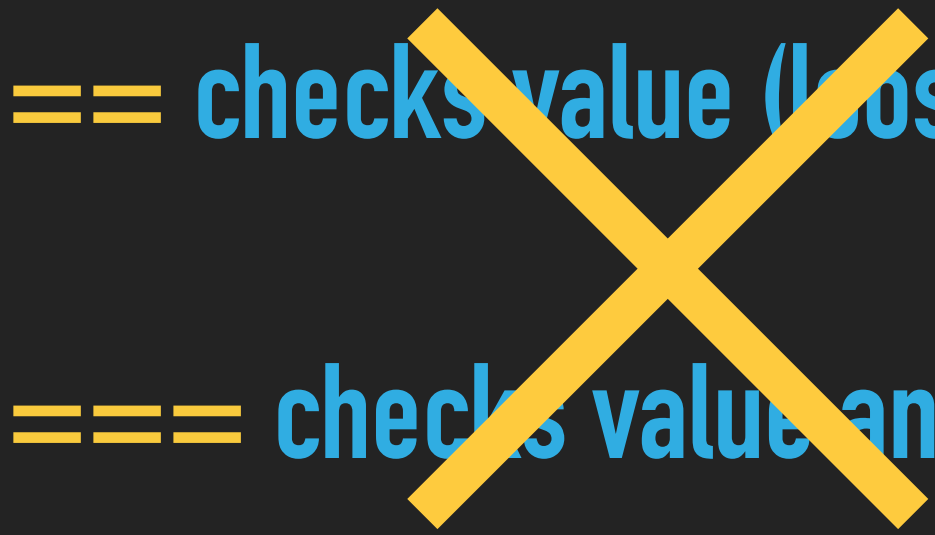
Checking Equality

== vs. ===

== checks value (loose)

=== checks value and type (strict)

?



== allows coercion (types different)

=== disallows coercion (types same)

Coercive Equality vs. Non-Coercive Equality

```
1  var studentName1 = "Frank";
2  var studentName2 = `${studentName1}`;
3
4  var workshopEnrollment1 = 16;
5  var workshopEnrollment2 = workshopEnrollment1 + 0;
6
7  studentName1 == studentName2;                    // true
8  studentName1 === studentName2;                   // true
9
10 workshopEnrollment1 == workshopEnrollment2;   // true
11 workshopEnrollment1 === workshopEnrollment2; // true
```

Coercive Equality: == and ===

```
1  var workshop1 = { topic: null };
2  var workshop2 = {};
3
4  if (
5      (workshop1.topic === null || workshop1.topic === undefined) &&
6      (workshop2.topic === null || workshop2.topic === undefined) &&
7  ) {
8      // ..
9  }
10
11 if (
12     workshop1.topic == null &&
13     workshop2.topic == null
14 ) {
15     // ..
16 }
```

Coercive Equality: null == undefined

Like every other operation, is coercion helpful in an equality comparison or not?

== is not about comparisons with unknown types

== is about comparisons with known type(s), optionally where conversions are helpful

JavaScript has a (dynamic) type system, which uses various forms of coercion for value type conversion, including equality comparisons

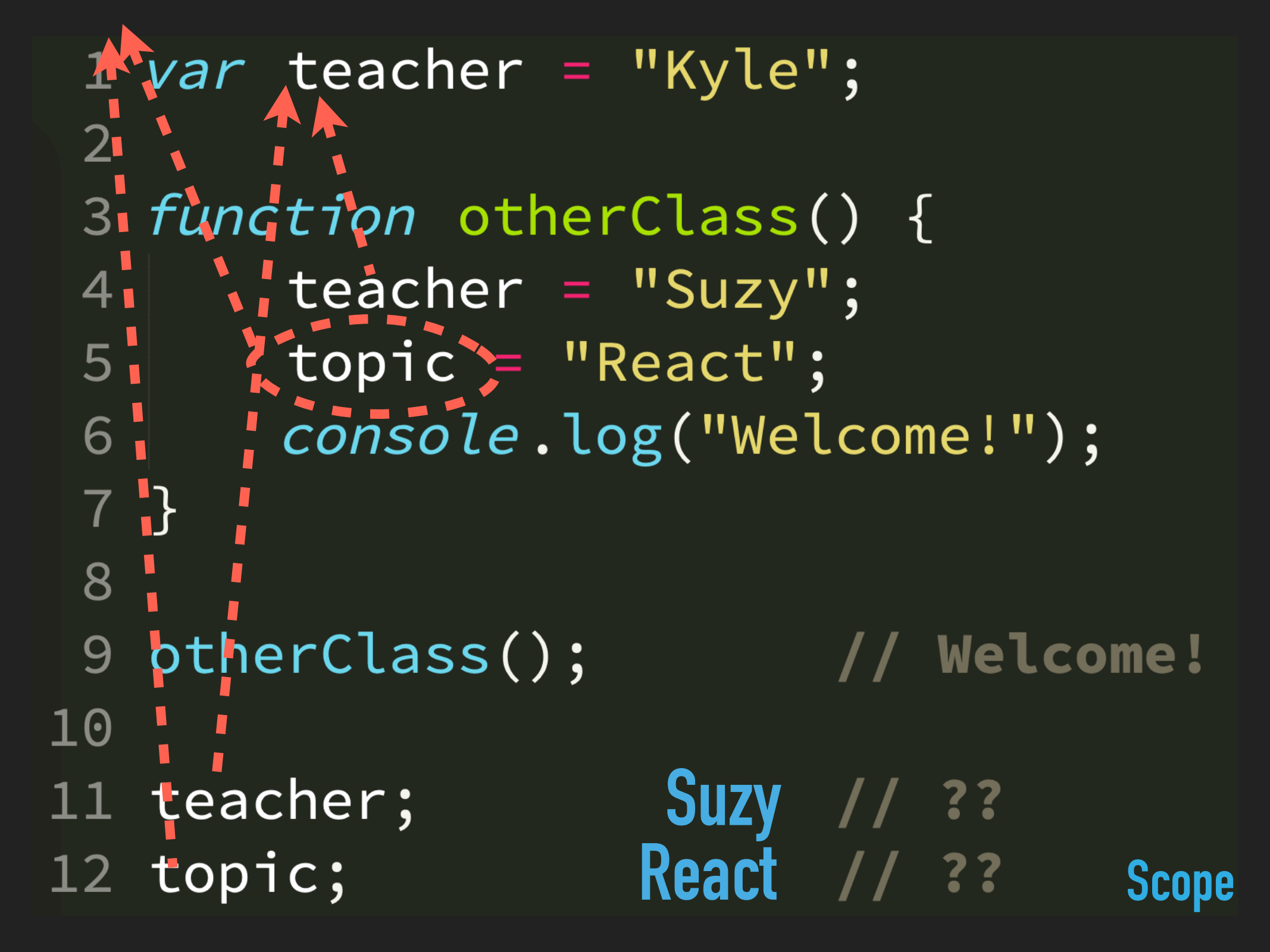
You simply cannot write quality JS programs without knowing the types involved in your operations.

Scope / Closures

- Nested Scope
- Closure

Scope: where to look for things

```
1 x = 42;
2 console.log(y);
```


var teacher = "Kyle";
function otherClass() {
teacher = "Suzy";
topic = "React";
console.log("Welcome!");
}
otherClass(); // Welcome!
teacher; // ??
topic; // ??
Suzy
React
Scope

undefined
vs.
undeclared

Function Expressions

```
1 var clickHandler = function(){
2     // ..
3 };
4
5 var keyHandler = function keyHandler(){
6     // ..
7 };
```

Named Function Expressions

```
1  var ids = people.map(person => person.id);
2
3  var ids = people.map(function getId(person){
4      return person.id;
5  });
6
7  // **********************
8
9  getPerson()
10 .then(person => getData(person.id))
11 .then(renderData);
12
13 getPerson()
14 .then(function getDataFrom(person){
15     return getData(person.id);
16 })
17 .then(renderData);
```

Arrow Functions?

```
1 var teacher = "Kyle";
2
3 ( function anotherTeacher() {
4     var teacher = "Suzy";
5     console.log(teacher);    // Suzy
6 } )();
7
8 console.log(teacher);    // Kyle
```

http://benalman.com/news/2010/11/immediately-invoked-function-expression/

Function Scoping: IIFE

Block Scoping

Instead of an IIFE?

```
1 var teacher = "Kyle";
2
3 ( function anotherTeacher() {
4     var teacher = "Suzy";
5     console.log(teacher);    // Suzy
6 } )();
7
8 console.log(teacher);    // Kyle
```

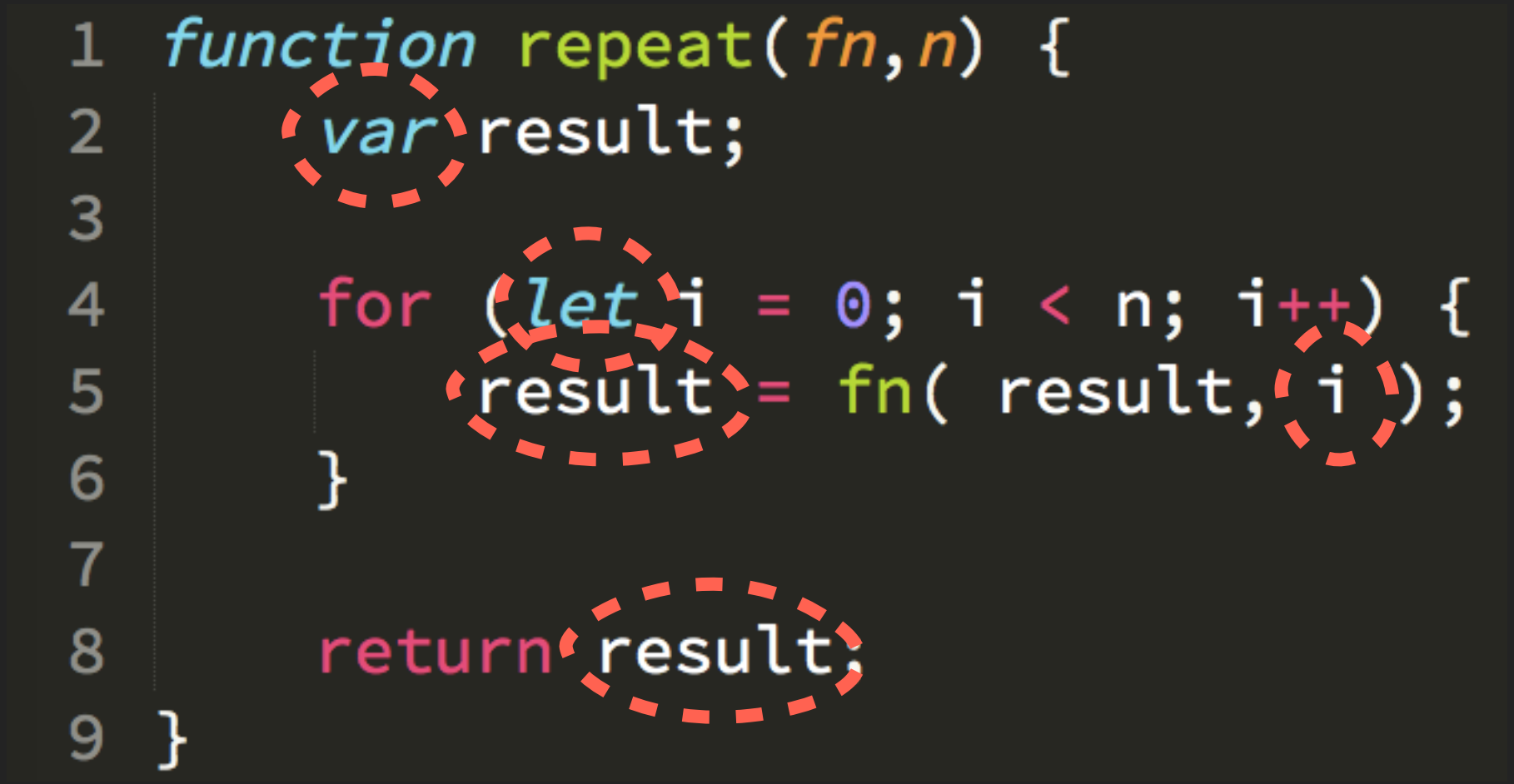
Block Scoping: encapsulation

```
1 var teacher = "Kyle";
2
3 {
4     let teacher = "Suzy";
5     console.log(teacher);    // Suzy
6 }
7
8 console.log(teacher);    // Kyle
```

Block Scoping: encapsulation

```
1 function diff(x,y) {
2     if (x > y) {
3         let tmp = x;
4         x = y;
5         y = tmp;
6     }
7
8     return y - x;
9 }
```

Block Scoping: let



Block Scoping: let + var

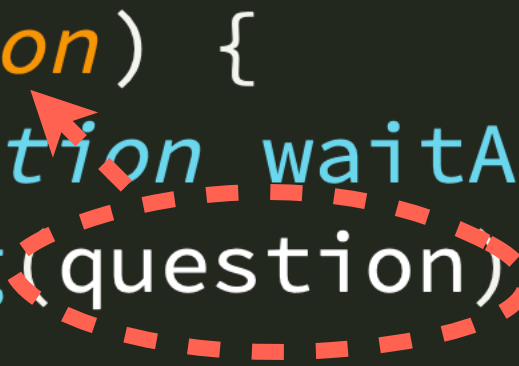
```
1  function formatStr(str) {
2      { let prefix, rest;
3          prefix = str.slice( 0, 3 );
4          rest = str.slice( 3 );
5          str = prefix.toUpperCase() + rest;
6      }
7
8      if (/^FOO:/.test( str )) {
9          return str;
10     }
11
12     return str.slice( 4 );
13 }
```

Block Scoping: explicit let block

Closure

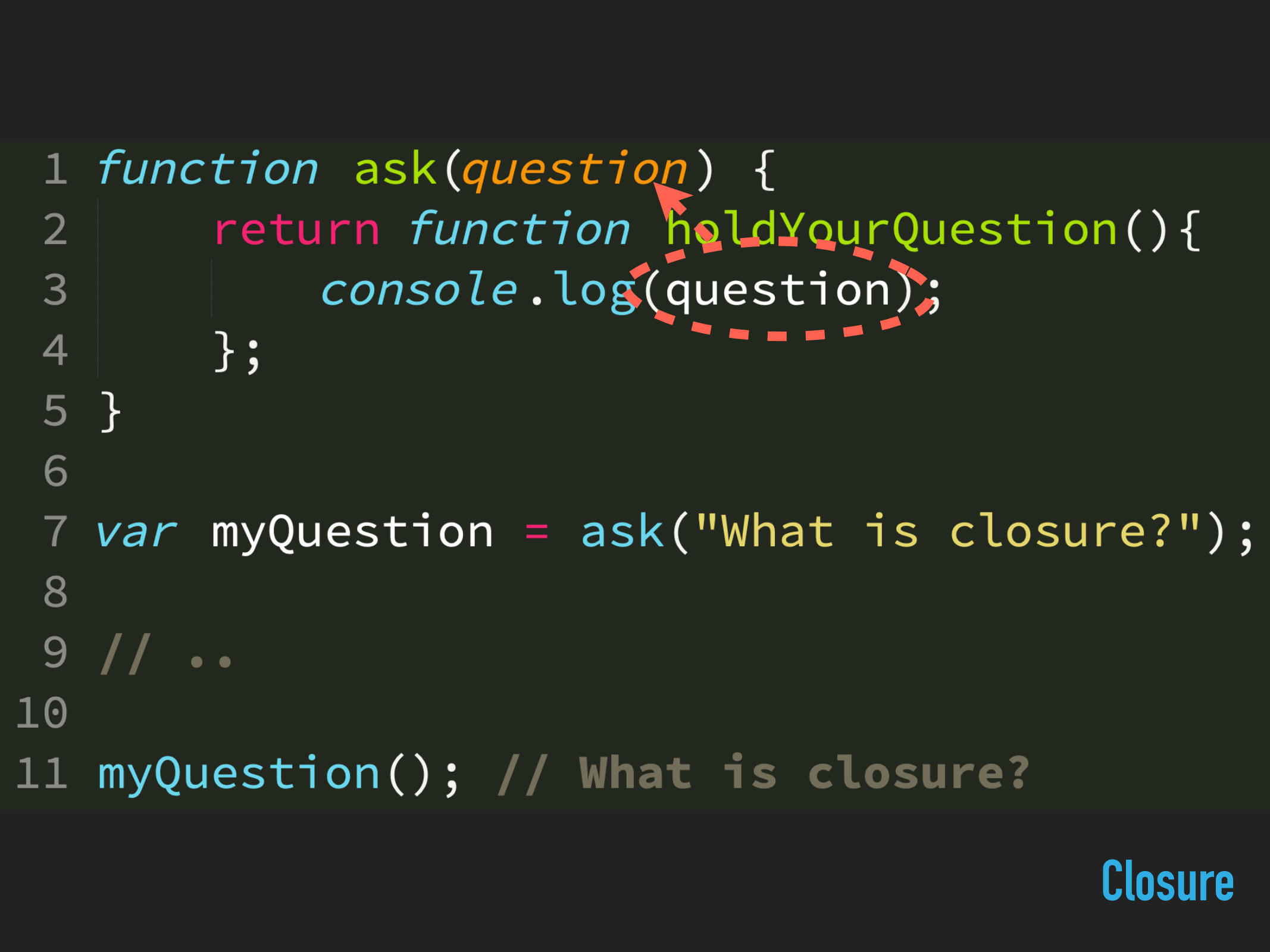
Closure is when a function “remembers” the variables outside of it, even if you pass that function elsewhere.

```
1 function ask(question) {
2     setTimeout(function waitASec(){
3         console.log(question);
4     },100);
5 }
6
7 ask("What is closure?");
8 // What is closure?
```



Closure

```
1  function ask(question) {
2      return function holdYourQuestion(){
3          console.log(question);
4      };
5  }
6
7  var myQuestion = ask("What is closure?");
8
9  // ..
10
11 myQuestion(); // What is closure?
```



Closure

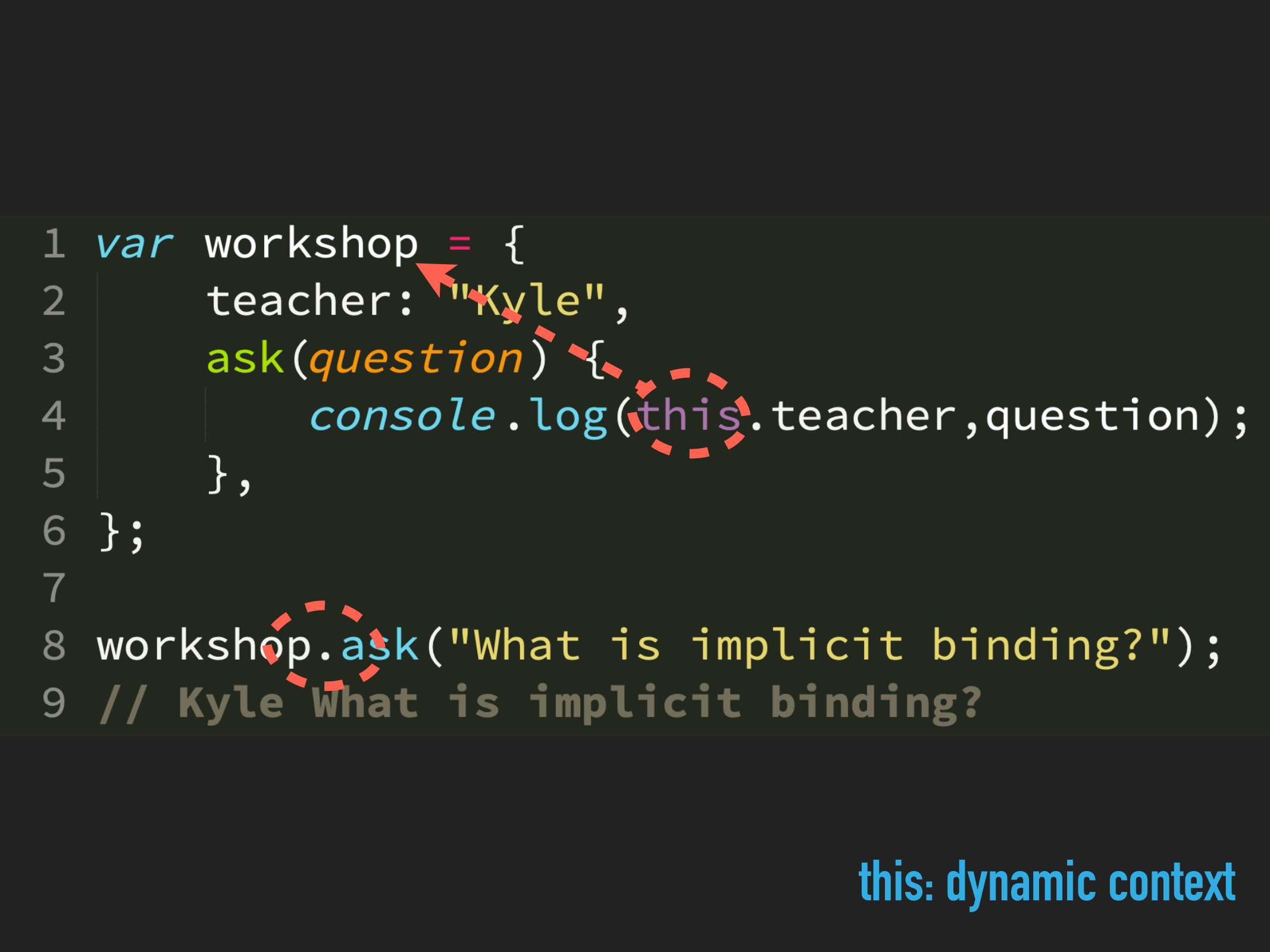
this / Prototypes

- this
- Prototypes
- class { }

this

A function's this references the execution context for that call, determined entirely by how the function was called.

A this-aware function can thus have a different context each time it's called, which makes it more flexible & reusable.


var workshop = {
teacher: "Kyle",
ask(question) {
console.log(this.teacher,question);
},
};
workshop.ask("What is implicit binding?");
// Kyle What is implicit binding?
this: dynamic context

this: dynamic context

Prototypes

```
1  function Workshop(teacher) {
2      this.teacher = teacher;
3  }
4  Workshop.prototype.ask = function(question){
5      console.log(this.teacher,question);
6  };
7
8  var deepJS = new Workshop("Kyle");
9  var reactJS = new Workshop("Suzy");
10
11 deepJS.ask("Is 'prototype' a class?");
12 // Kyle Is 'prototype' a class?
13
14 reactJS.ask("Isn't 'prototype' ugly?");
15 // Suzy Isn't 'prototype' ugly?
```

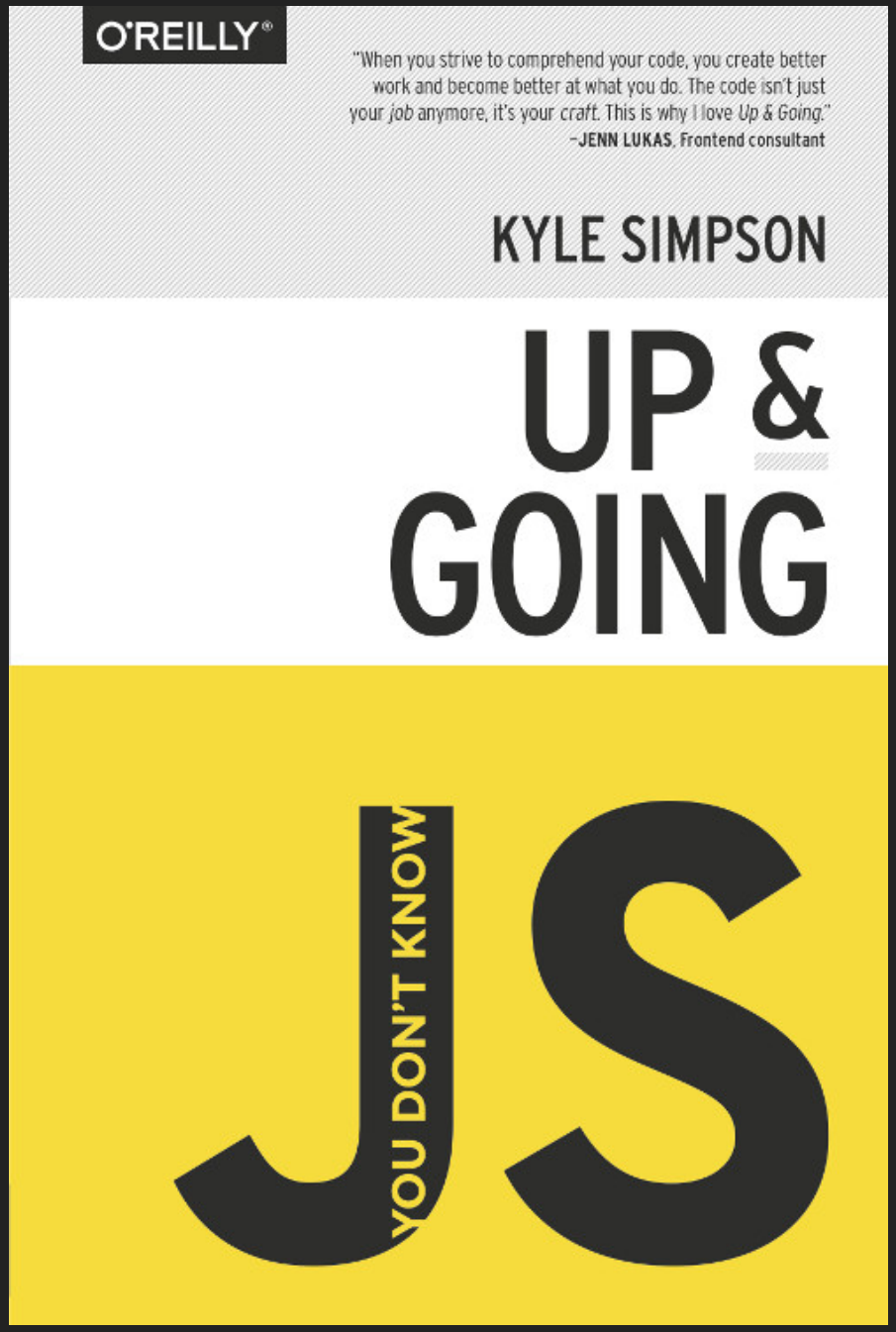
Prototypes: as "classes"

class { }

ES6

```
class Workshop {
    constructor(teacher) {
        this.teacher = teacher;
    }
    ask(question) {
        console.log(this.teacher,question);
    }
}

var deepJS = new Workshop("Kyle");
var reactJS = new Workshop("Suzy");

deepJS.ask("Is 'class' a class?");
// Kyle Is 'class' a class?

reactJS.ask("Is this class OK?");
// Suzy Is this class OK?
```

ES6 class


O'REILLY®
"When you strive to comprehend your code, you create better work and become better at what you do. The code isn't just your *job* anymore, it's your *craft*. This is why I love *Up & Going*."
–JENN LUKAS, Frontend consultant
KYLE SIMPSON
UP & GOING
YOU DON'T KNOW
JS

Chapter 2

The best way to learn JS is to get in and write it!

THANKS!!!!

KYLE SIMPSON GETIFY@GMAIL.COM

GETTING INTO JS