



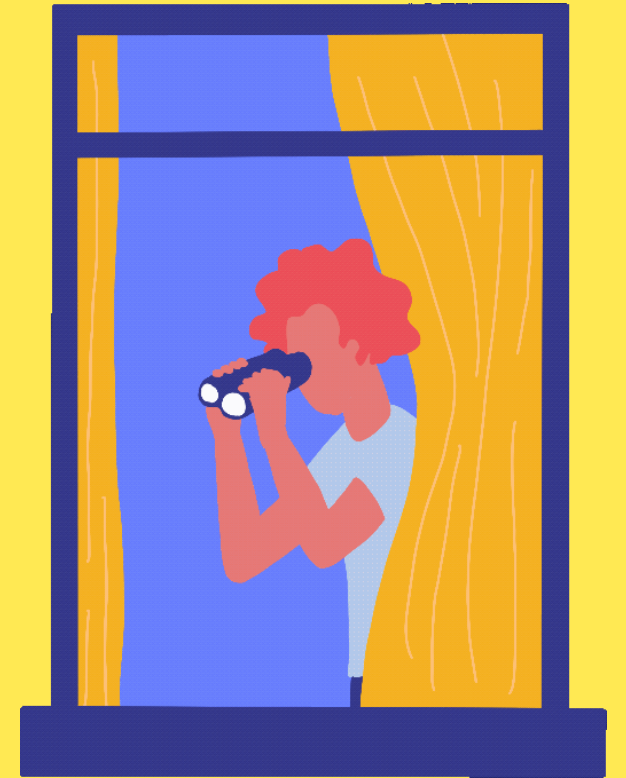
# FUNCTIONS IN DETAIL

Important things  
you should know  
about functions

# SCOPE

## Variable "visibility"

The location where a variable is defined dictates where we have access to that variable



# FUNCTION SCOPE

```
function helpMe(){  
    let msg = "I'm on fire!";  
    msg; //"I'm on fire";  
}
```

*msg* is scoped to the  
*helpMe* function

```
msg; //NOT DEFINED!
```

# FUNCTION SCOPE



```
let bird = 'mandarin duck';  
  
function birdWatch(){  
    let bird = 'golden pheasant';  
    bird; //'golden pheasant'  
}  
  
bird; //'mandarin duck'
```

*bird* is scoped to  
birdWatch function

# BLOCK SCOPE

```
let radius = 8;

if(radius > 0){
    const PI = 3.14;
    let circ = 2 * PI * radius;
}

console.log(radius); //8
console.log(PI); //NOT DEFINED
console.log(circ); //NOT DEFINED
```


*PI & circ* are  
scoped to the  
BLOCK

# LEXICAL SCOPE



```
function outer() {  
  let hero = "Black Panther";  
  
  function inner() {  
    let cryForHelp = `${hero}, please save me!`  
    console.log(cryForHelp);  
  }  
  
  inner();  
}
```

# FUNCTION EXPRESSIONS



```
const square = function (num) {  
  return num * num;  
}  
square(7); //49
```



**FUNCTIONS  
ARE...  
OBJECTS!**





# HIGHER ORDER FUNCTIONS




# HIGHER ORDER FUNCTIONS

Functions that operate on/with other functions.

They can:


- Accept other functions as arguments
- Return a function

# FUNCTIONS AS ARGUMENTS



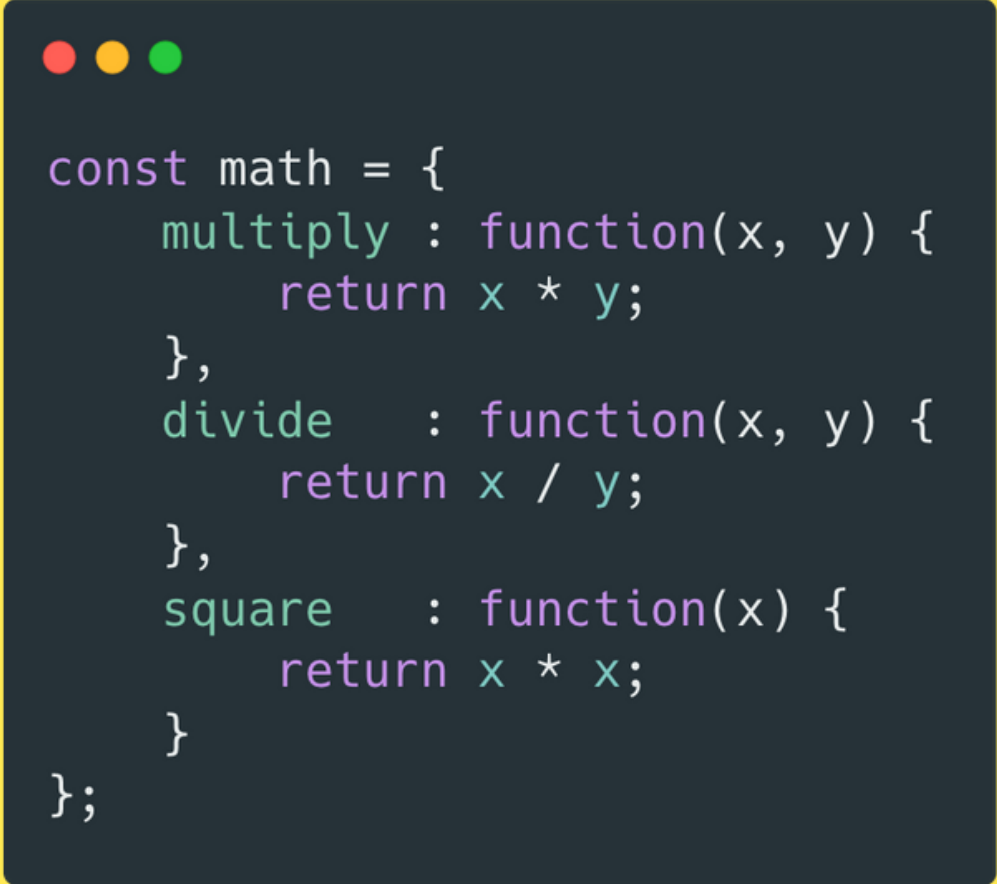
```
function callTwice(func) {  
  func();  
  func();  
}  
  
function laugh() {  
  console.log("HAHAHAHAHAHAHAHAHAHAHAHA");  
}  
callTwice(laugh) //pass a function as an arg!  
// "HAHAHAHAHAHAHAHAHAHAHAHA"  
// "HAHAHAHAHAHAHAHAHAHAHAHA"
```

# RETURNING FUNCTIONS



```
function makeBetweenFunc(min, max) {  
  return function (val) {  
    return val >= min && val <= max;  
  }  
}  
  
const inAgeRange = makeBetweenFunc(18, 100);  
  
inAgeRange(17); //false  
inAgeRange(68); //true
```

# METHODS




```
const math = {  
  multiply : function(x, y) {  
    return x * y;  
  },  
  divide   : function(x, y) {  
    return x / y;  
  },  
  square   : function(x) {  
    return x * x;  
  }  
};
```

We can add functions as properties on objects.

We call them methods!

# SHORTHAND



```
const math = {  
  blah: 'Hi!',  
  add(x, y) {  
    return x + y;  
  },  
  multiply(x, y) {  
    return x * y;  
  }  
}  
math.add(50, 60) //110
```

We do this so often that there's a new shorthand way of adding methods.

# 'THIS' IN METHODS

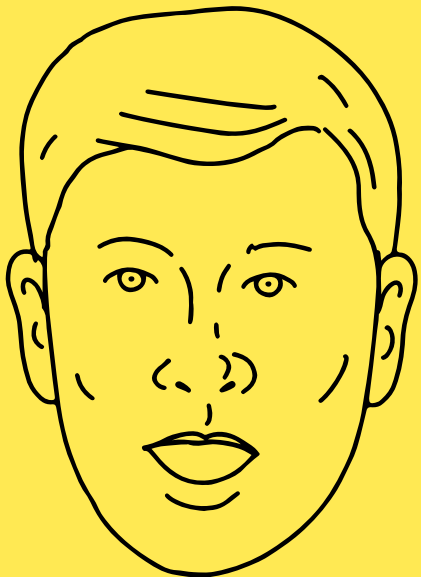
Use the keyword `this` to access other properties on the same object.



```
const person = {  
  first: 'Robert',  
  last: 'Herjavec',  
  fullName() {  
    return `${this.first} ${this.last}`  
  }  
}  
  
person.fullName(); // "Robert Herjavec"  
person.last = "Plant";  
person.fullName(); // "Robert Plant"
```



The value of ***this*** depends on  
the invocation context of  
the function it is used in.



# SAME FUNCTION



```
const person = {  
  first: 'Robert',  
  last: 'Herjavec',  
  fullName() {  
    return `${this.first} ${this.last}`  
  }  
}
```



```
person.fullName();  
// "Robert Herjavec"
```

# DIFFERENT RESULT???



```
const func = person.fullName;  
func()  
// "undefined undefined"
```

The value of ***this*** depends on  
the **invocation context** of  
the function it is used in.

