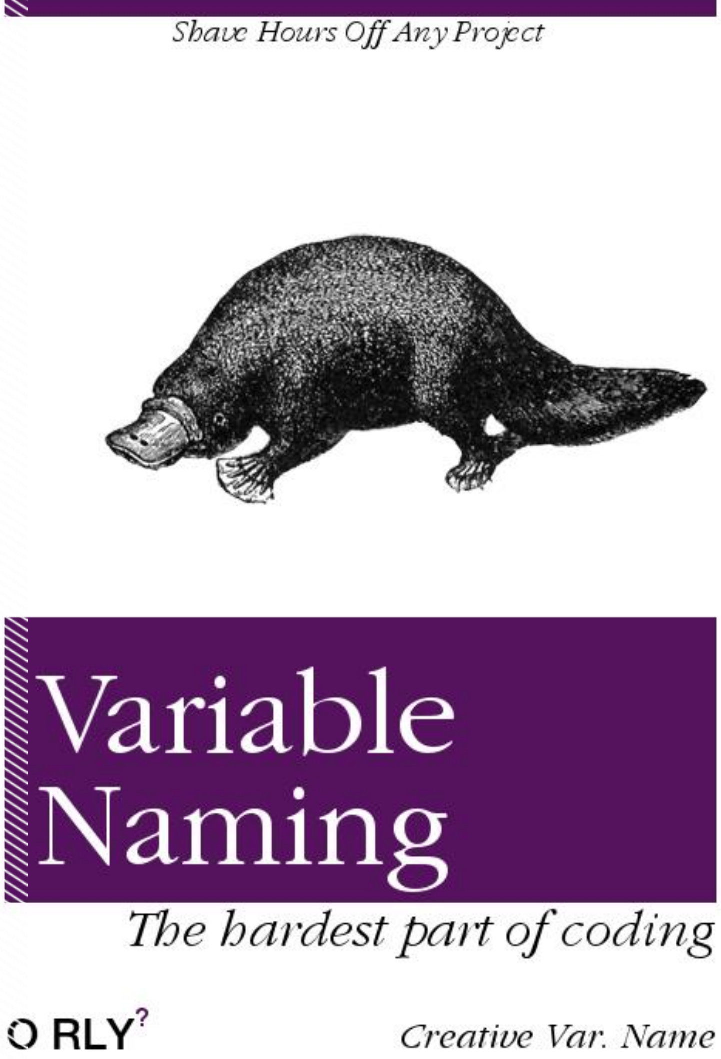
JS Functions



Frontend Junior Program - 2022

Agenda

1 Intro

2 Function

3 Creating a function

4 Arguments and parameters

5 Closure

Preface

At last, we arrived.

This is the part, where your journey really begins. From this point, you will understand what I mean by that: *"Boy, that escalated quickly"*

In fact, it could be your constant feeling. Things will be switched from super easy to very hard to understand, just in a fraction of time.

And that is normal. Please don't give up when you feel confused. We've all been there, done that. However, let me surprise you: it is not that hard.

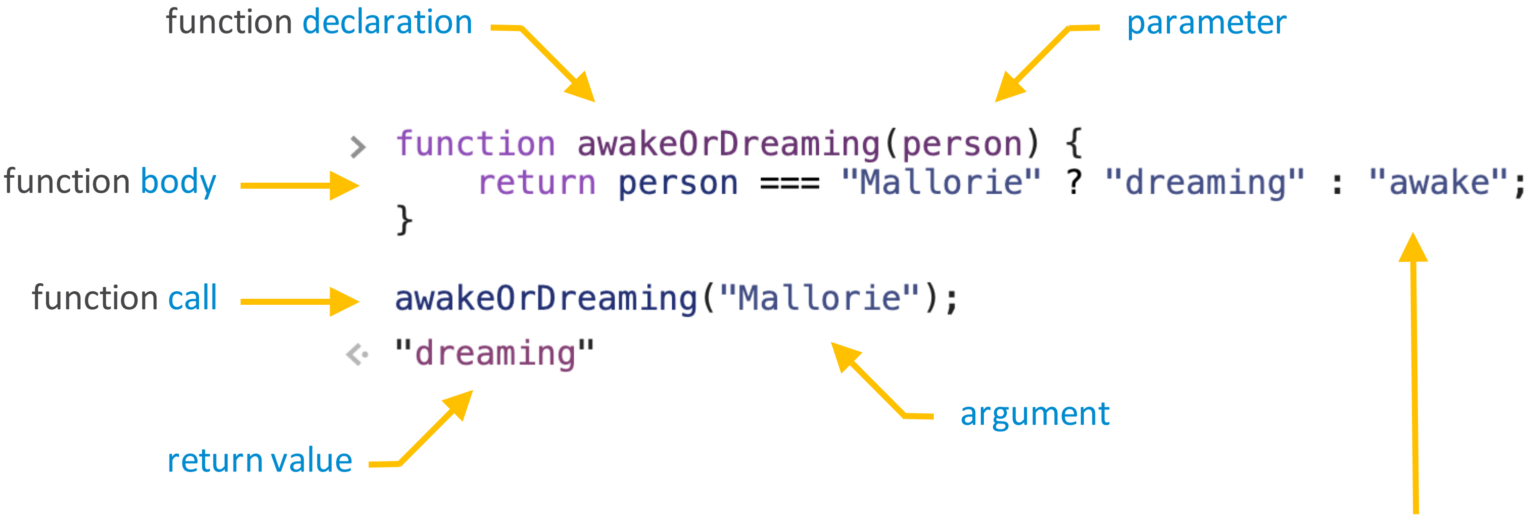
It is just the feeling. If you read it again, if you try the examples and create new ones just to make sure that you've understood correctly …

… then it could serve you as a strong basis to build on. If you do that, it will make your (and your colleagues) life much easier in future.

WHAT'S RECURSION?
IT'S LIKE DREAMING ABOUT YOUR DREAM AND WHEN YOU WAKE UP, YOU'RE STILL IN THAT DREAM
ARE WE DREAMING NOW?

FUNCTION

Function – a bit of terminology



function declaration

parameter

function body

function call

return value

argument

```
> function awakeOrDreaming(person) {
      return person === "Mallorie" ? "dreaming" : "awake";
  }
  awakeOrDreaming("Mallorie");
< "dreaming"
```

A pretty fancy expression using the ternary operator.
It is called ternary, because it does have three operands.
As you guessed, we have binary and unary operators as well.

Function – a first-class citizen

Functions are first-class citizens in JavaScript.

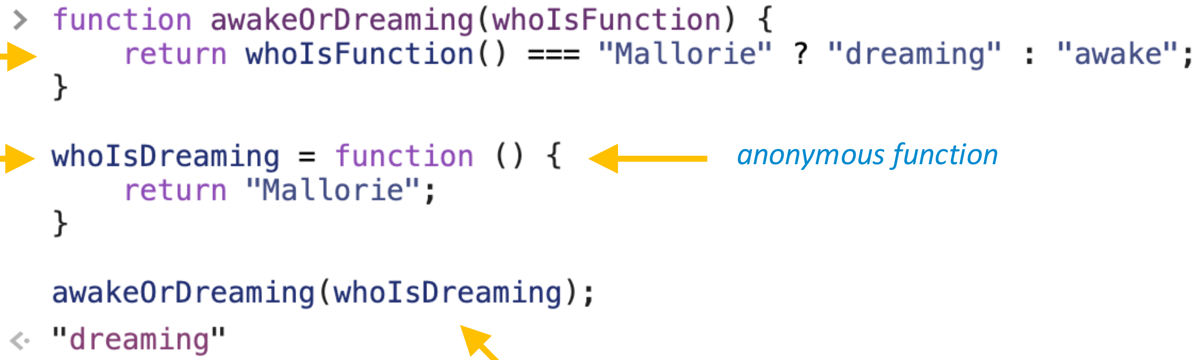
It means that they act as variables: we can assign them to a variable, we can pass them as an argument, they can be returned as a value, etc.* This is the basis of the functional programming, and that's why JavaScript is still modern and so cool today.

a higher-order function,
it eats functions for breakfast

whoIsDreaming
will be called only here

function expression

anonymous function



```
> function awakeOrDreaming(whoIsFunction) {
      return whoIsFunction() === "Mallorie" ? "dreaming" : "awake";
  }

  whoIsDreaming = function () {
      return "Mallorie";
  }

  awakeOrDreaming(whoIsDreaming);
< "dreaming"
```

look again! we pass a function as an argument, but we don't call it yet

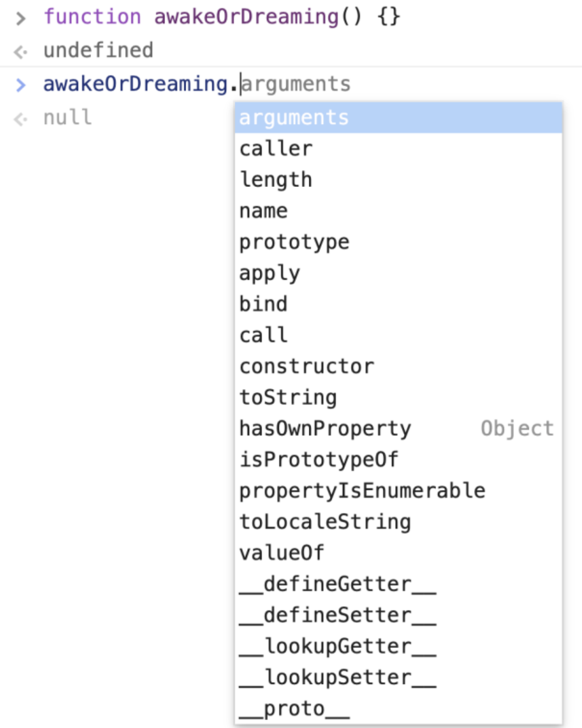
* *having anonymous functions also help in this regard.*

Function - object

Functions are objects

The Function object does have some predefined properties and methods, most of them focused on the object-oriented part. We will open that pandora’s box later.

prototype
bind, apply, call
hasOwnProperty



```
> function awakeOrDreaming() {}
< undefined
> awakeOrDreaming.arguments
< null
```

```
arguments
caller
length
name
prototype
apply
bind
call
constructor
toString
hasOwnProperty        Object
isPrototypeOf
propertyIsEnumerable
toLocaleString
valueOf
__defineGetter__
__defineSetter__
__lookupGetter__
__lookupSetter__
__proto__
```

Scope

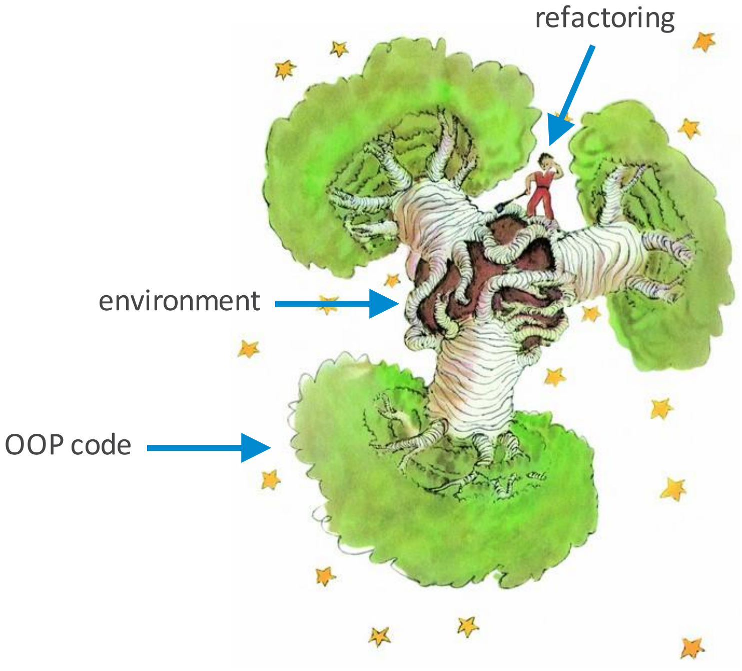
Think about scope as an environment where the code exists.

Essentially, *an ECMAScript scope contains variable, constant, let, class, module, import, and/or function declarations.*

This is a pretty complex area, and it is definitely not required to know in detail.

However, if you are still interested, here is the spec and an excellent summary about it.

You were warned ;)

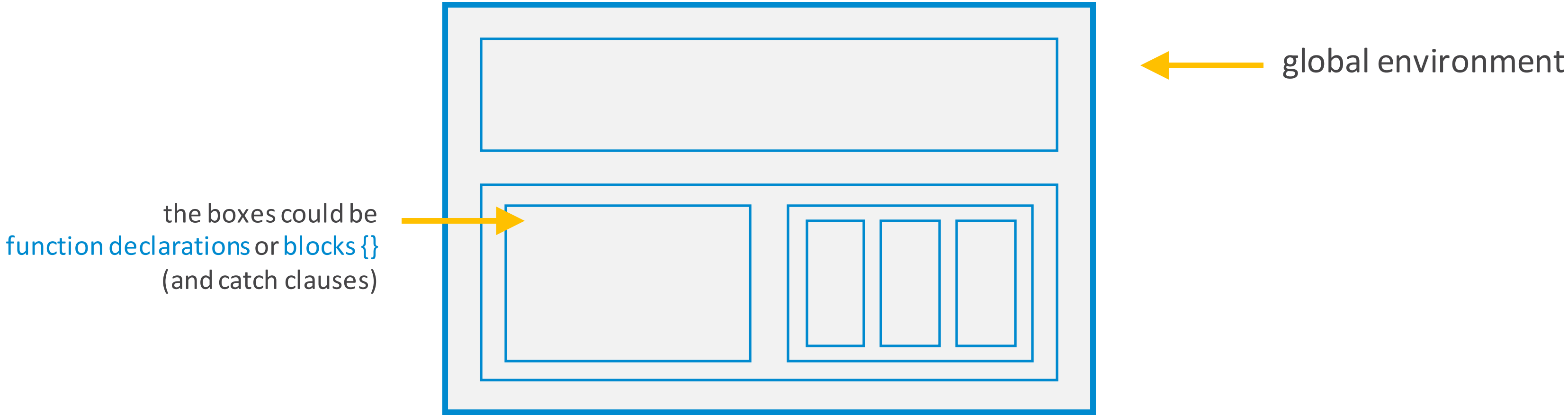


JavaScript for impatient programmers: Scope

Environment

Environments are like boxes.

You can put them to each other, and there is nothing outside the larger box (global environment).



“boxception”

Sure, but why is this important?

Basically, for 2 reasons:

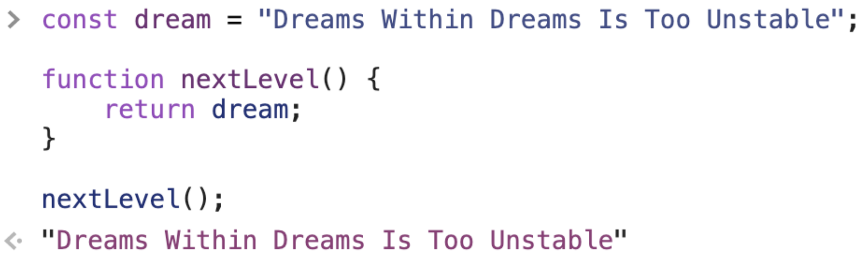
1. when you declare a variable, you need to understand the places, where it can have an effect.
2. when you access an outer variable, you need to be aware of the consequences.*

you can see outside, but you cannot see into the house

* *a sneak peek to closures: if you open the window and grab a flower, then this connection will prevent destroying your house.*

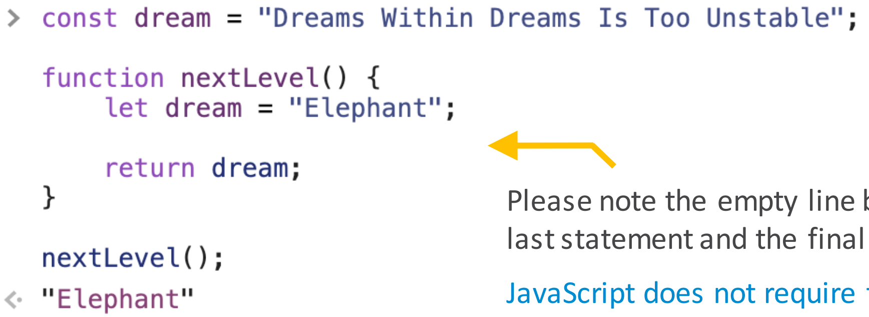
Shadowing

A function can access the outer scope



```
> const dream = "Dreams Within Dreams Is Too Unstable";

  function nextLevel() {
      return dream;
  }

  nextLevel();
<· "Dreams Within Dreams Is Too Unstable"
```

Outer variables can be shadowed



```
> const dream = "Dreams Within Dreams Is Too Unstable";

  function nextLevel() {
      let dream = "Elephant";

      return dream;
  }

  nextLevel();
<· "Elephant"
```

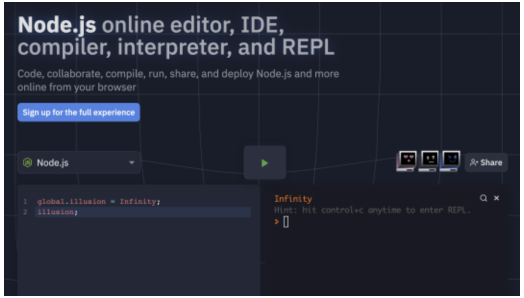
Please note the empty line between the last statement and the final return.

JavaScript does not require that. We do.

Global object

Also, we have a global object.

The global object (window, or global in Node.js) is bound to the global environment, and that's why the global object's properties appears as a global variables.



Node.js online editor, IDE, compiler, interpreter, and REPL

Code, collaborate, compile, run, share, and deploy Node.js and more online from your browser

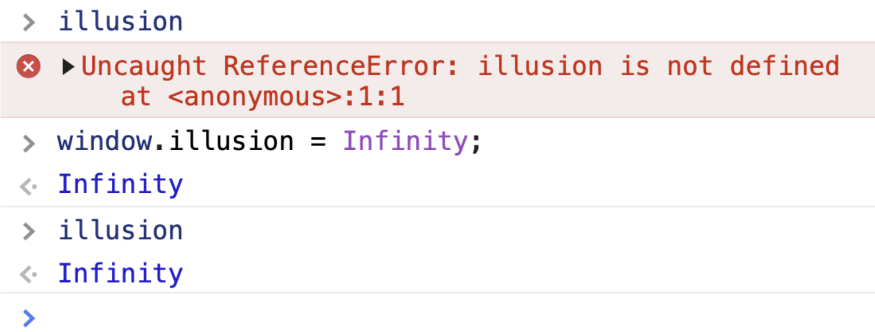
```
global.illusion = Infinity;
illusion;
```

```
Infinity
Hint: hit control+c anytime to enter REPL.
>
```

a Node.js (and other) REPL, in case you wanna play.

we does not have an illusion →

now we have →



```
> illusion
✖ ▶Uncaught ReferenceError: illusion is not defined
      at <anonymous>:1:1
> window.illusion = Infinity;
< Infinity
> illusion
< Infinity
>
```

CREATING A FUNCTION

Function declaration vs function expression

function declarations are hoisted

Can be called anywhere in the current scope.
A function name is required: there is no such a thing like "~~anonymous function declaration~~".

```
printParam('print me!')
function printParam(param) {
    console.log(param);
}
```

function expressions are like variables

While the identifier will be hoisted as well, the value is not
- can't be called before the assignment. A name is not required (anonymous), but it is possible to have.

```
let printParam = function (param) {
    console.log(param);
}
printParam('print me!')
```

IIFE (Immediately Invoked Function Expression)

The () turns the function declaration into an expression.

```
(function (param) {
    console.log(param);
})('print me!')
```

Named Function Expression

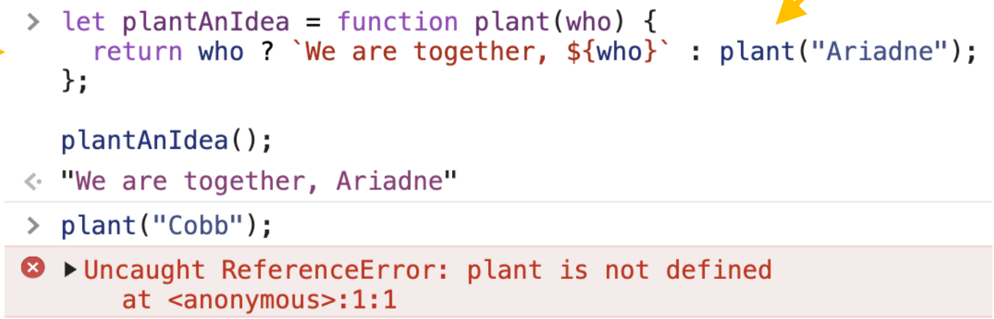
Named Function Expression (NFE) is a term for function expressions that have a name.

It allows to reference the function inside, and it is not visible outside of the function. Basically, there are 2 reasons for this: supporting the debugging process, and to call these functions recursively.

Sometimes, we add a name just for clarity.

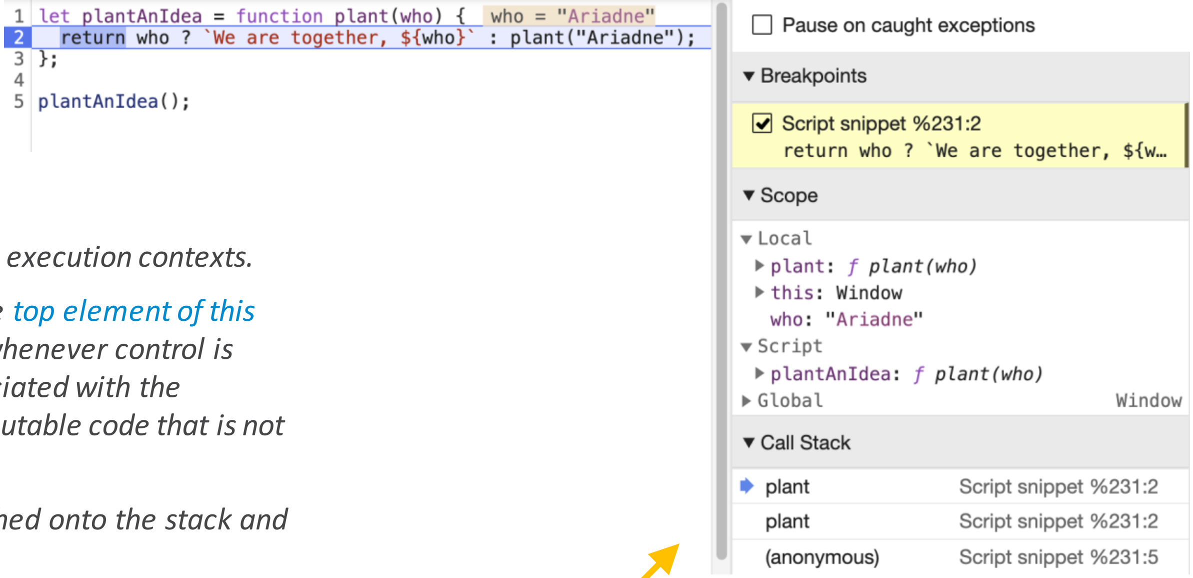
a recursive call

with recursive calls, it is always good idea to have a condition, unless you want to test how deep the ~~rabbit hole~~ call stack



```
> let plantAnIdea = function plant(who) {
    return who ? `We are together, ${who}` : plant("Ariadne");
  };

  plantAnIdea();
<· "We are together, Ariadne"
> plant("Cobb");
✖ ▶Uncaught ReferenceError: plant is not defined
      at <anonymous>:1:1
```

Call Stack (execution context stack)



```
let plantAnIdea = function plant(who) {  who = "Ariadne"
  return who ? `We are together, ${who}` : plant("Ariadne");
};

plantAnIdea();
```

```
☐ Pause on caught exceptions
▼ Breakpoints
  ☑ Script snippet %231:2
    return who ? `We are together, ${w…
▼ Scope
  ▼ Local
    ▶ plant: ƒ plant(who)
    ▶ this: Window
      who: "Ariadne"
  ▼ Script
    ▶ plantAnIdea: ƒ plant(who)
  ▶ Global                                  Window
▼ Call Stack
  ➡ plant                  Script snippet %231:2
    plant                  Script snippet %231:2
    (anonymous)            Script snippet %231:5
```

Execution context stack*

The execution context stack is used to track execution contexts.

The running execution context is always the top element of this stack. A new execution context is created whenever control is transferred from the executable code associated with the currently running execution context to executable code that is not associated with that execution context.

The newly created execution context is pushed onto the stack and becomes the running execution context.

the rabbit hole
here is 2 levels deep

* While it seems to be a bit convoluted, the Standard is pretty clear here.

Execution context

Execution contexts are like rooms

However, these are magical rooms: a room is created only when you open the door (when you call a function) and will be destroyed* after you left the room (return).

You can open doors and walk from one room to another, but you must leave the room where you entered.

Even when you call a function inside the same function (recursion), a new room will be created.

When you open doors from one room to another, your rooms will be created and placed on top of each other (call stack), and when you go backward, these will be removed one by one.

looks like you missed to add a condition for a recursive call…

** except a special case, when you grab something outside (a flower, or a variable). In this case, the room will exist, while it is needed.*

Do we use recursion in projects a lot?

Nope. In fact, it is pretty rare. There are many reasons for that:
first, usually it is not needed. Second, it adds cognitive complexity to the code.

The golden rule here is the same as with other solutions (e.g., flexbox):

If it is the easiest, natural way
to achieve your goal, then use it.

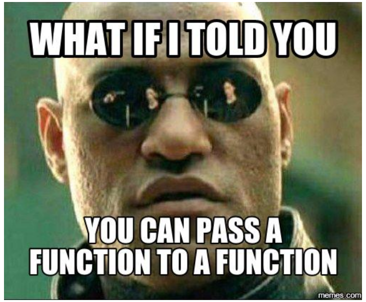
That being said, now you need to build your algorithmic
mindset, and recursion is really a fun way to do that.

Also, it could be requested in interviews,
so please master it (by practicing).

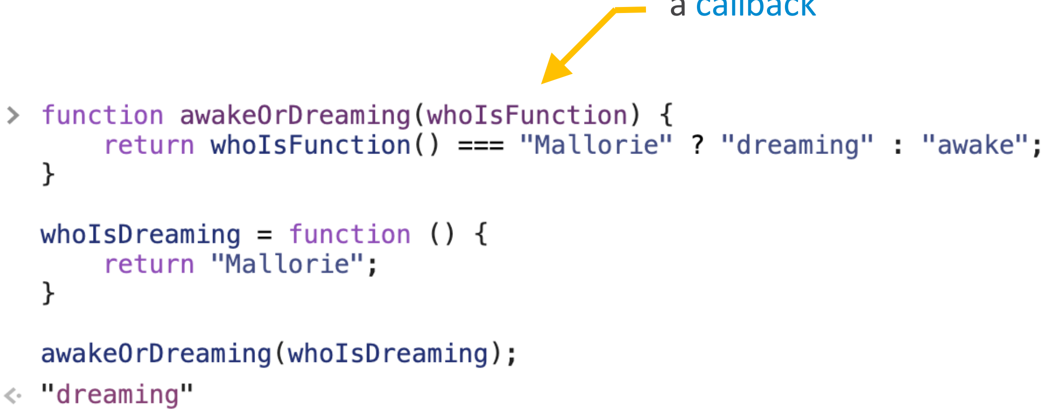
Callback functions

A callback is a function passed into another function as an argument, which is then invoked inside the outer function to complete some kind of routine or action.

Functions can take functions as arguments and can be returned by other functions. Functions that do this are called higher-order functions. Any function that is passed as an argument is called a callback function.



a callback

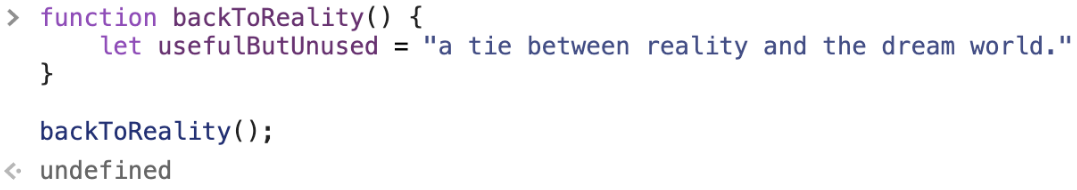


```
> function awakeOrDreaming(whoIsFunction) {
      return whoIsFunction() === "Mallorie" ? "dreaming" : "awake";
  }

  whoIsDreaming = function () {
      return "Mallorie";
  }

  awakeOrDreaming(whoIsDreaming);
<· "dreaming"
```

Returning

missing the return means

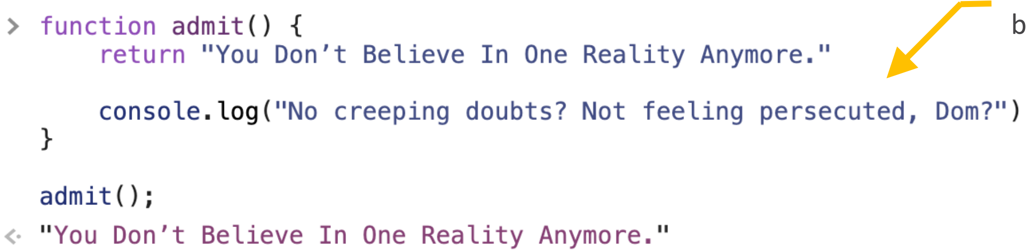
… returning undefined anyway



```
> function backToReality() {
      let usefulButUnused = "a tie between reality and the dream world."
  }

  backToReality();
<· undefined
```

the return *returns**

*stops the execution.



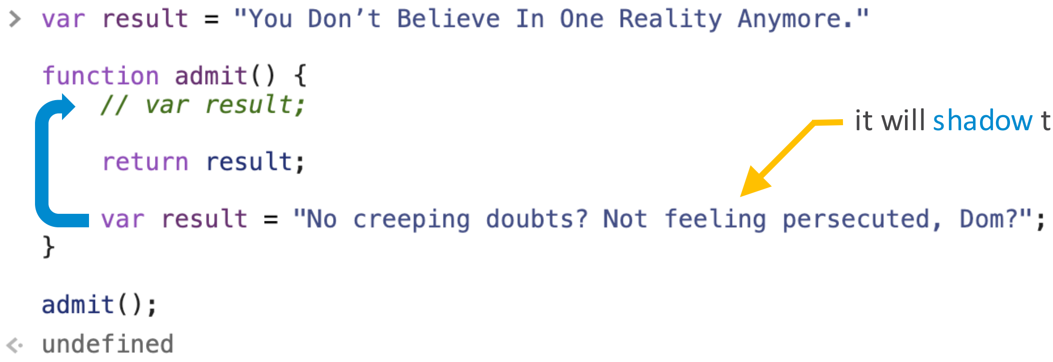
```
> function admit() {
      return "You Don’t Believe In One Reality Anymore."

      console.log("No creeping doubts? Not feeling persecuted, Dom?")
  }

  admit();
<· "You Don’t Believe In One Reality Anymore."
```

this is not executed, but…

How to create bugs? Part 4523

Just combine different parts of JavaScript,

which are not that simple alone: hoisting and shadowing.



```
> var result = "You Don’t Believe In One Reality Anymore."

  function admit() {
      // var result;

      return result;

      var result = "No creeping doubts? Not feeling persecuted, Dom?";
  }

  admit();
<· undefined
```

… but at the same time it will be hoisted as well

it will shadow the result variable …

How can you prevent that? Avoid both!

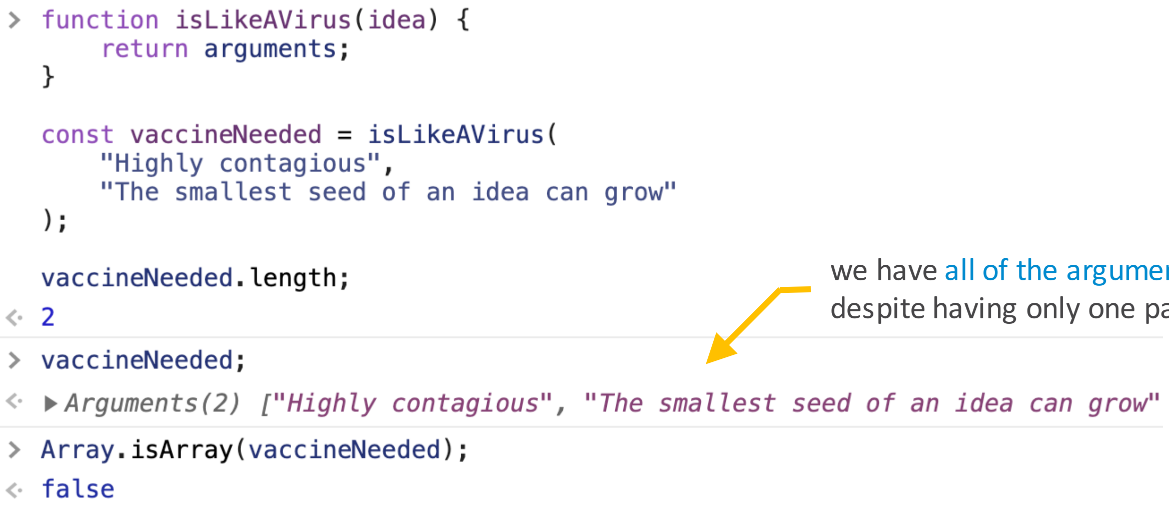
- *use let*
- *don’t rely on global variables*

ARGUMENTS AND PARAMETERS

Arguments object

arguments is an array-like object

A built-in object, containing all the function parameters.
Sadly, you won't use it* (use the rest parameters, instead).



```
> function isLikeAVirus(idea) {
      return arguments;
  }

  const vaccineNeeded = isLikeAVirus(
      "Highly contagious",
      "The smallest seed of an idea can grow"
  );

  vaccineNeeded.length;
< 2
> vaccineNeeded;
< ▶Arguments(2) ["Highly contagious", "The smallest seed of an idea can grow"]
> Array.isArray(vaccineNeeded);
< false
```

acts like an array →

we have all of the arguments, despite having only one parameter

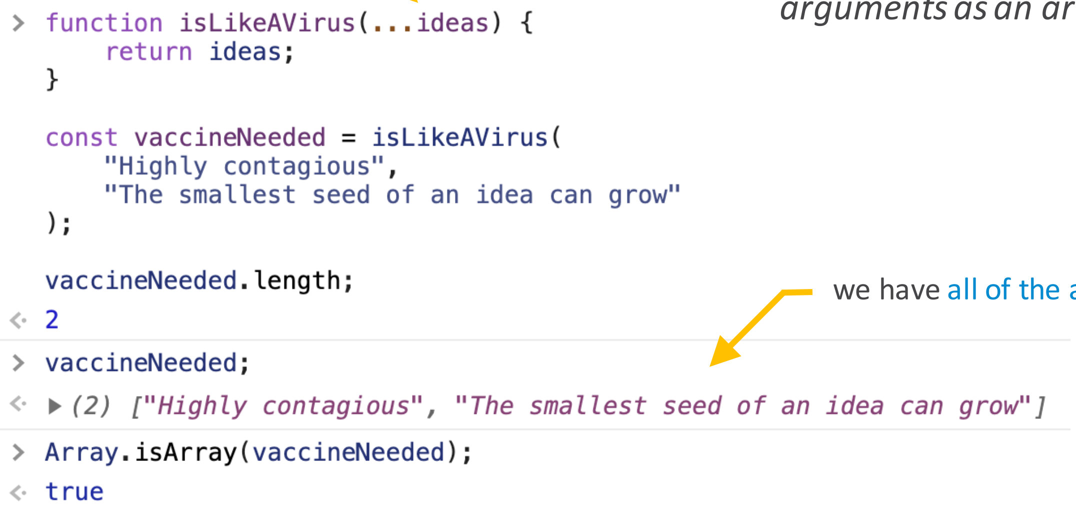
but it is not →

* Basically, there are 2 reasons for that. First: it is magic. Second: it behaves differently with "arrow functions". This will be covered later.

Rest parameters

the rest parameter

The rest parameter allows a function to accept an indefinite number of arguments as an array

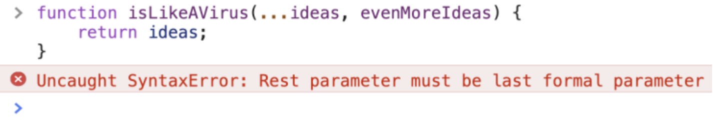


```
> function isLikeAVirus(...ideas) {
      return ideas;
  }

  const vaccineNeeded = isLikeAVirus(
      "Highly contagious",
      "The smallest seed of an idea can grow"
  );

  vaccineNeeded.length;
< 2
> vaccineNeeded;
< ▶ (2) ["Highly contagious", "The smallest seed of an idea can grow"]
> Array.isArray(vaccineNeeded);
< true
```

it is an array

we have all of the arguments as well

really is



```
> function isLikeAVirus(...ideas, evenMoreIdeas) {
      return ideas;
  }
⊗ Uncaught SyntaxError: Rest parameter must be last formal parameter
>
```

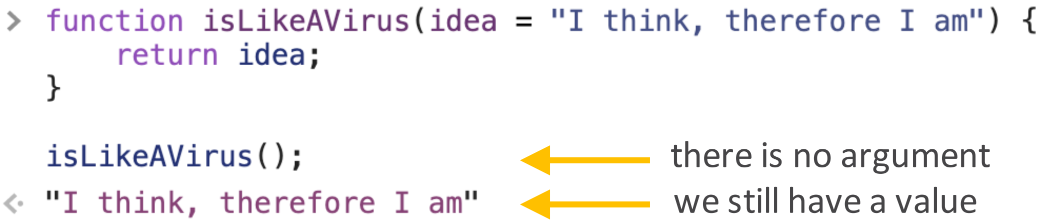
⚠

Default parameters

default parameters initialize a parameter with a value

… even when the argument was not provided. Works well with the values of zero and false as an argument, too.

the default parameter



```
function isLikeAVirus(idea = "I think, therefore I am") {
    return idea;
}

isLikeAVirus();
"I think, therefore I am"
```

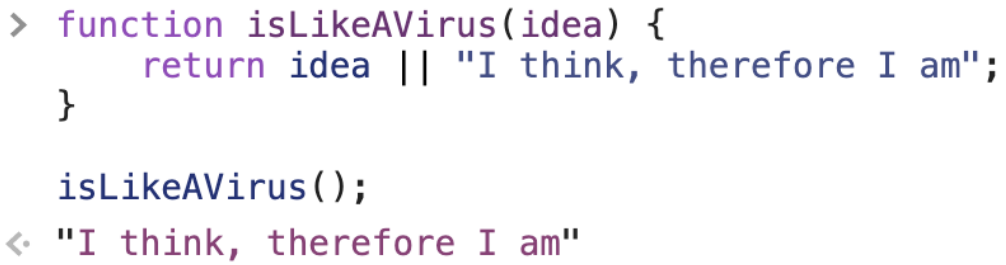
there is no argument

we still have a value

Defaulting original

The old defaulting pattern

Not that safe, think about the curious case of a number parameter and zero as an argument. ⚠



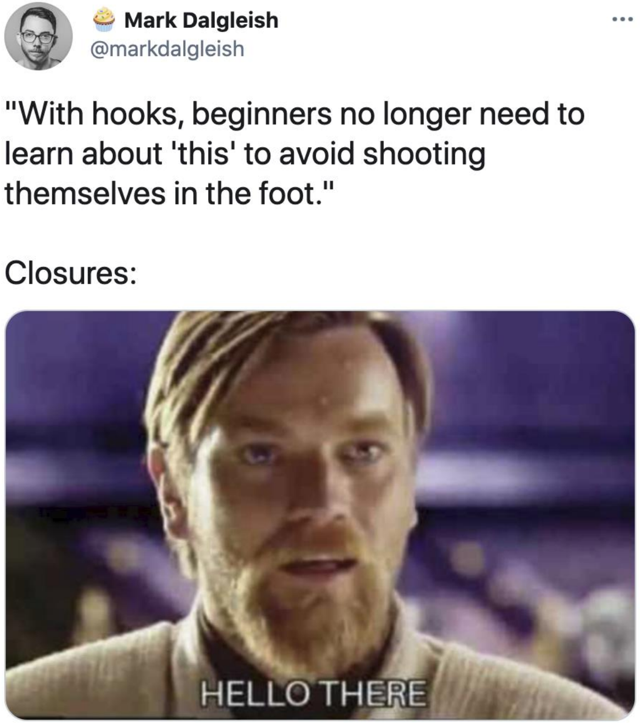
```
> function isLikeAVirus(idea) {
      return idea || "I think, therefore I am";
  }

  isLikeAVirus();
<· "I think, therefore I am"
```

CLOSURES

Closures

Closures. Here we are.

Let us build this step by step.



Mark Dalgleish
@markdalgleish

"With hooks, beginners no longer need to learn about 'this' to avoid shooting themselves in the foot."

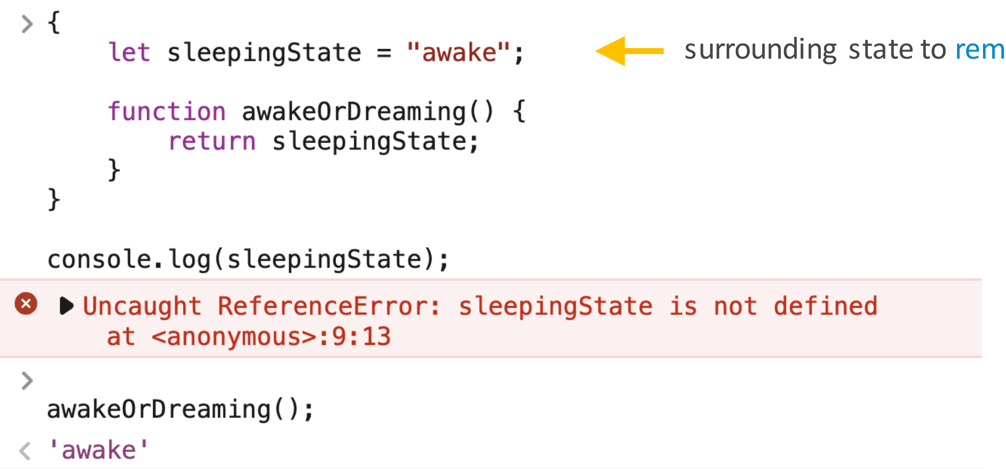
Closures:

no worries, we will learn about this

Creating a closure

A closure is when a function remembers its surrounding state.

A really simple example:



```
> {
      let sleepingState = "awake";

      function awakeOrDreaming() {
          return sleepingState;
      }
  }

  console.log(sleepingState);
✖ ▶Uncaught ReferenceError: sleepingState is not defined
      at <anonymous>:9:13
>
  awakeOrDreaming();
< 'awake'
```

surrounding state to remember

sleepingState was lost…

… or was not? awakeOrDreaming() keeps it alive!

closures →

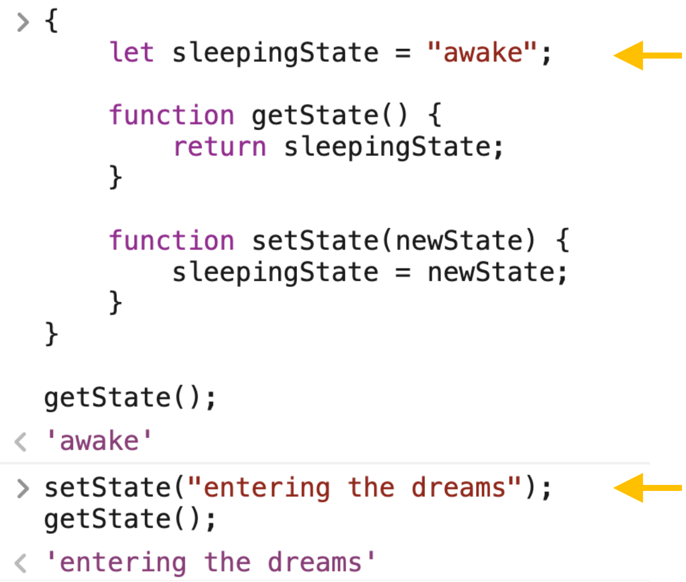
value →

different state →

~~INCEPTION~~
SELLING:
The Art of Planting ~~an Idea~~ in
Your ~~Potential Customer's Mind~~

The living state

It is not simply the *value* of sleepingState that is remembered, but the state itself:



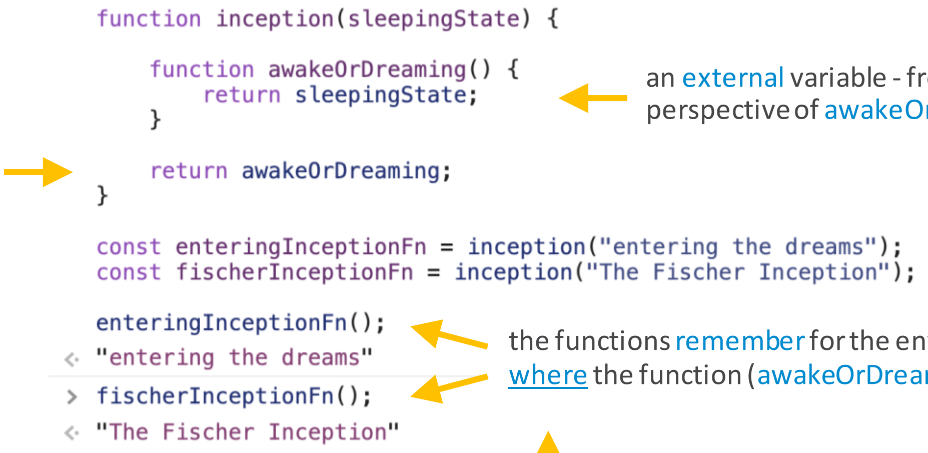
```
> {
      let sleepingState = "awake";

      function getState() {
          return sleepingState;
      }

      function setState(newState) {
          sleepingState = newState;
      }
  }

  getState();
< 'awake'
> setState("entering the dreams");
  getState();
< 'entering the dreams'
```

surrounding state to remember

it is remembered

and now we can change it!

A more realistic example, how we use it usually:



```
function inception(sleepingState) {

    function awakeOrDreaming() {
        return sleepingState;
    }

    return awakeOrDreaming;
}

const enteringInceptionFn = inception("entering the dreams");
const fischerInceptionFn = inception("The Fischer Inception");

  enteringInceptionFn();
<· "entering the dreams"
> fischerInceptionFn();
<· "The Fischer Inception"
```

an external variable - from the perspective of awakeOrDreaming()

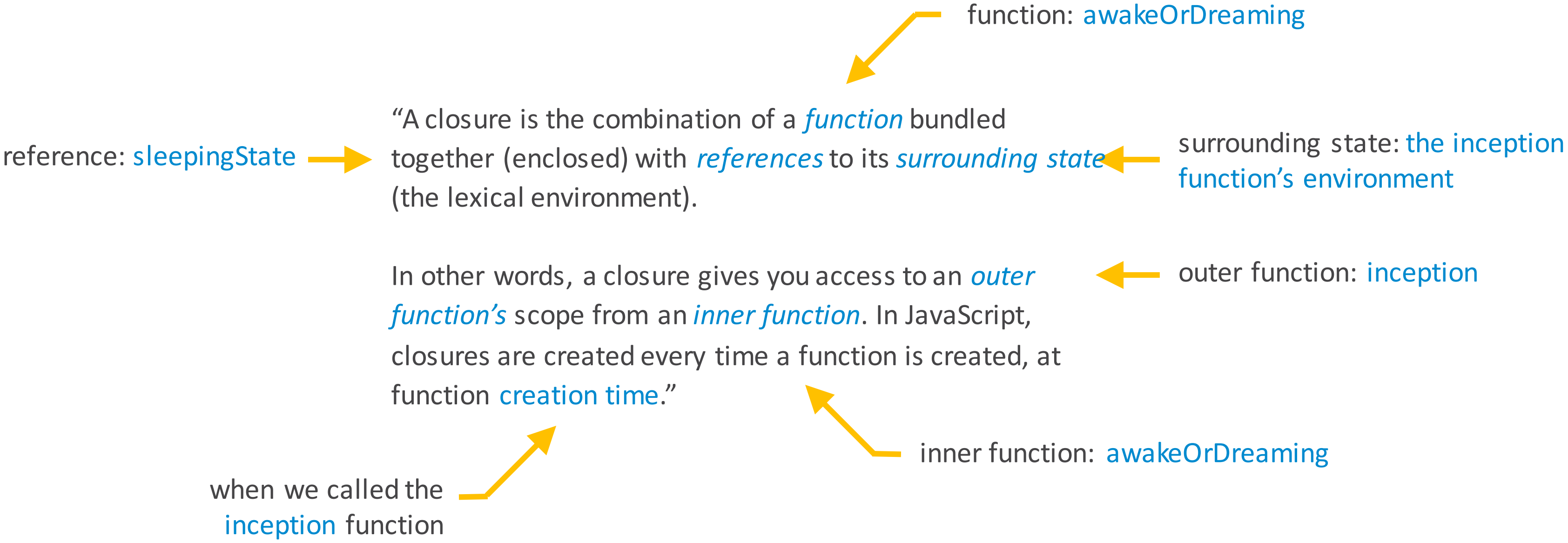
important part: returning a function

we call inception() in different ways, keeping alive 2 different states

the functions remember for the environment (variables, parameters, etc...) where the function (awakeOrDreaming) was declared.

to put it simply, the *lexical environment* means this: *where*

And now the [definition]() could be clear:



function: awakeOrDreaming

reference: sleepingState

“A closure is the combination of a *function* bundled together (enclosed) with *references* to its *surrounding state* (the lexical environment).

surrounding state: the inception function’s environment

In other words, a closure gives you access to an *outer function’s* scope from an *inner function*. In JavaScript, closures are created every time a function is created, at function creation time.”

outer function: inception

inner function: awakeOrDreaming

when we called the inception function

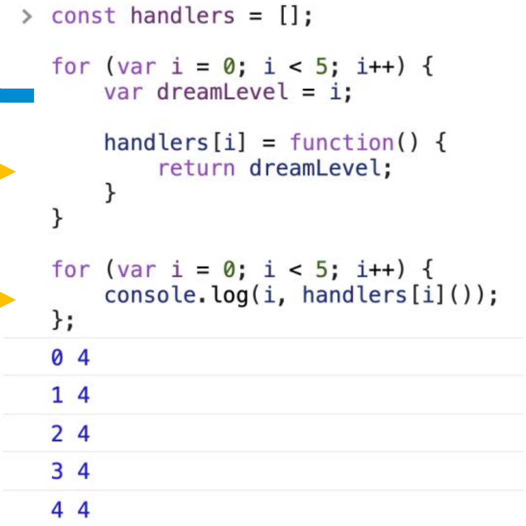
In the last episode… why let is useful?

Remember the closure in loop issue?

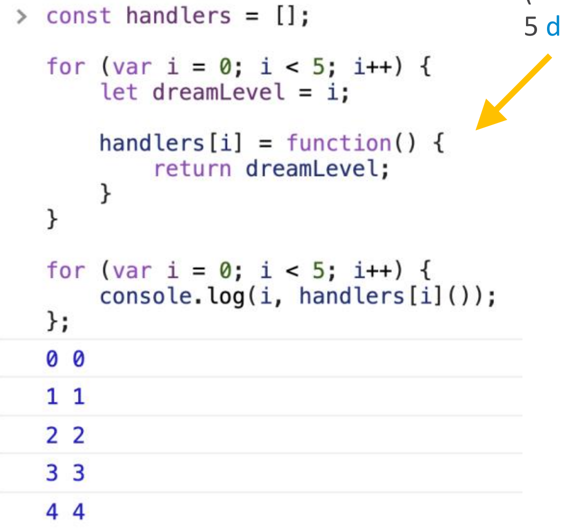
dreamLevel is hoisted

it will be evaluated…

… only here, and now dreamLevel is 4



```
> const handlers = [];

  for (var i = 0; i < 5; i++) {
      var dreamLevel = i;

      handlers[i] = function() {
          return dreamLevel;
      }
  }

  for (var i = 0; i < 5; i++) {
      console.log(i, handlers[i]());
  };
  0 4
  1 4
  2 4
  3 4
  4 4
```



```
> const handlers = [];

  for (var i = 0; i < 5; i++) {
      let dreamLevel = i;

      handlers[i] = function() {
          return dreamLevel;
      }
  }

  for (var i = 0; i < 5; i++) {
      console.log(i, handlers[i]());
  };
  0 0
  1 1
  2 2
  3 3
  4 4
```

dreamLevel now exists
(and remembered) in
5 different environments

Do we use closures in projects?

Yes.

It is very important to understand the concept, and depending on the project and the tech stack, sometimes it is used a lot.

And when it is not that frequent, you still will run into closures here and there, and then it will be critical to understand it.

Naming a function

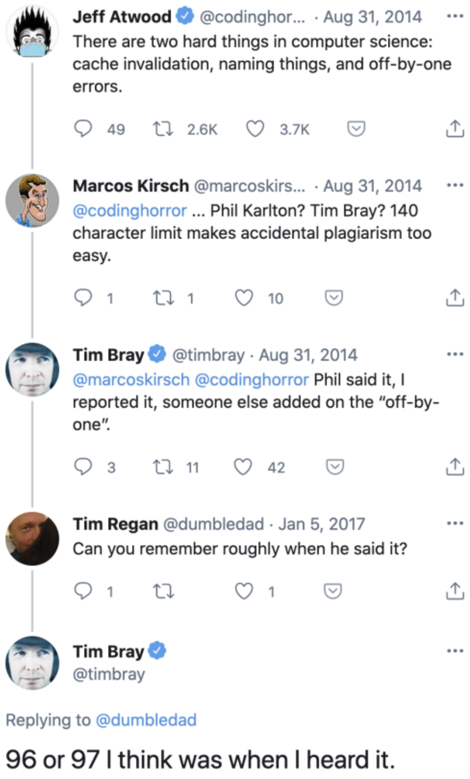
Functions are actions. So, their name is usually a verb.

Function names starts with like these …

- "get…" – returns a value,
- "calc…" – calculate something,
- "create…" – create something,
- ”is…" – check something and return a boolean, etc.

The most fun part is when you get something, but you modify that at the same time, still, you would not consider it as creation.

```
showMessage(..)     // shows a message
getAge(..)          // returns the age (gets it somehow)
createForm(..)      // creates a form (and returns it)
calcSum(..)         // calculates a sum and returns
checkPermission(..) // checks a permission
```



Jeff Atwood @codinghor… · Aug 31, 2014

There are two hard things in computer science: cache invalidation, naming things, and off-by-one errors.

49 2.6K 3.7K

Marcos Kirsch @marcoskirs… · Aug 31, 2014

@codinghorror … Phil Karlton? Tim Bray? 140 character limit makes accidental plagiarism too easy.

1 1 10

Tim Bray @timbray · Aug 31, 2014

@marcoskirsch @codinghorror Phil said it, I reported it, someone else added on the “off-by-one”.

3 11 42

Tim Regan @dumbledad · Jan 5, 2017

Can you remember roughly when he said it?

1 1

Tim Bray @timbray

Replying to @dumbledad

96 or 97 I think was when I heard it.

Q&A

edu_hu@epam.com