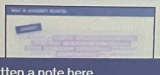
# HAT IS JAVASCRIPT: REVISITED

## JAVASCRIPT

JAVASCRIPT IS A HIGH-LEVEL PROTOTYPE-BASED OBJECT-ORIENTED MULTI-PARADIGM INTERPRETED OR JUST-IN-TIME COMPILED DYNAMIC SINGLE-THREADED GARBAGE-COLLECTED PROGRAMMING LANGUAGE WITH FIRST-CLASS FUNCTIONS AND A NON-BLOCKING EVENT LOOP CONCURRENCY MODEL 🤔 🤯 🤣



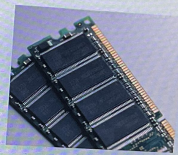
people have written a note here.

JS

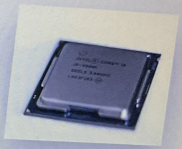
# DECONSTRUCTING THE MONSTER DEFINITION

- High-level
- Garbage-collected
- Interpreted or just-in-time compiled
- Multi-paradigm
- Prototype-based object-oriented
- First-class functions
- Dynamic
- Single-threaded
- Non-blocking event loop

Any computer program needs resources:



+



LOW-LEVEL

HIGH-LEVEL

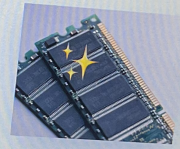
Developer has to manage resources manually

Developer does NOT have to worry, everything happens automatically

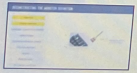
tell how much storage comp should reserve

# DECONSTRUCTING THE MONSTER DEFINITION

- High-level
- Garbage-collected
- Interpreted or just-in-time compiled
- Multi-paradigm
- Prototype-based object-oriented
- First-class functions
- Dynamic
- Single-threaded
- Non-blocking event loop



Cleaning the memory so we don't have to



# RECONSTRUCTING THE MONSTER DEFINITION

- High-level
- Garbage-collected
- Interpreted or just-in-time compiled
- Multi-paradigm
- Prototype-based object-oriented
- First-class functions
- Dynamic
- Single-threaded
- Non-blocking event loop

1 people have written a note here.

```
document.querySelector(".again").addEventListener("click", () => {
  document.querySelector(".message").textContent = "Start guessing...";
  document.querySelector(".number").textContent = "?";
  score = 20;
  document.querySelector(".score").textContent = score;
  number = Math.floor(Math.random() * 20) + 1;
});
```

Abstraction over 0s and 1s

CONVERT TO MACHINE CODE = COMPILING

Happens inside the JavaScript engine

More about this **Later in this Section**

# DECONSTRUCTING THE MONSTER DEFINITION

- High-level
- Garbage-collected
- Interpreted or just-in-time compiled
- Multi-paradigm
- Prototype-based object-oriented
- First-class functions
- Dynamic
- Single-threaded
- Non-blocking event loop

**Paradigm**: An approach and mindset of structuring code, which will direct your coding style and technique.

+ types

classify

The one we've been using so far

1. Procedural programming
2. Object-oriented programming (OOP)
3. Functional programming (FP)

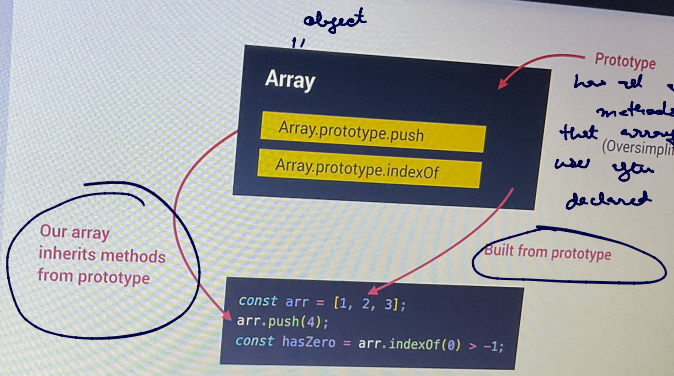
- Imperative vs.
- Declarative

JS is flexible
can be all 3 above

More about this later in **Multiple Sections**

# DECONSTRUCTING THE MONSTER DEFINITION

- High-level
- Garbage-collected
- Interpreted or just-in-time compiled
- Multi-paradigm
- Prototype-based object-oriented
- First-class functions
- Dynamic
- Single-threaded
- Non-blocking event loop




More about this in S… Oriented Programming

# DECONSTRUCTING THE MONSTER DEFINITION

- High-level
- Garbage-collected
- Interpreted or just-in-time compiled
- Multi-paradigm
- Prototype-based object-oriented
- First-class functions
- Dynamic
- Single-threaded
- Non-blocking event loop

In a language with **first-class functions**, functions are simply **treated as variables**. We can pass them into other functions, and return them from functions.

```
const closeModal = () => {
  modal.classList.add("hidden");
  overlay.classList.add("hidden");
};

overlay.addEventListener("click", closeModal);
```

Passing a function into another function as an argument: First-class functions!

More about this in Section **A Closer Look at Functions**

# DECONSTRUCTING THE MONSTER DEFINITION

- High-level
- Garbage-collected
- Interpreted or just-in-time compiled
- Multi-paradigm
- Prototype-based object-oriented
- First-class functions
- Dynamic
- Single-threaded
- Non-blocking event loop

**Dynamically-typed language:**

No data type definitions. Types becomes known at runtime

Data type of variable is automatically changed

```
let x = 23;
let y = 19;
x = "Jonas";
```


manually assign data type

# DECONSTRUCTING THE MONSTER DEFINITION

- High-level
- Garbage-collected
- Interpreted or just-in-time compiled
- Multi-paradigm
- Prototype-based object-oriented
- First-class functions
- Dynamic
- Single-threaded
- Non-blocking event loop

- **Concurrency model**: how the JavaScript engine handles multiple tasks happening at the same time.

↓ Why do we need that?

- JavaScript runs in one **single thread**, so it can only do one thing at a time.

↓ So what about a long-running task?

- Sounds like it would block the single thread. However, we want non-blocking behavior!

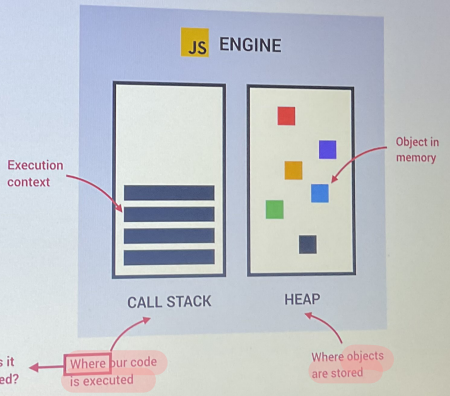
↓ How do we achieve that?

- By using an **event loop**: takes long running tasks, executes them in the "background", and puts them back in the main thread once they are finished.

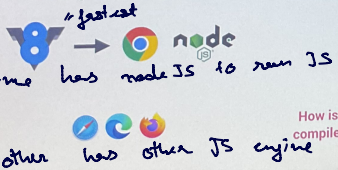
# WHAT IS A JAVASCRIPT ENGINE?

**JS ENGINE**

PROGRAM THAT **EXECUTES** JAVASCRIPT CODE.

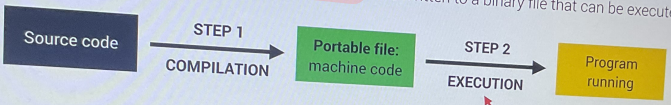



Example: V8 Engine



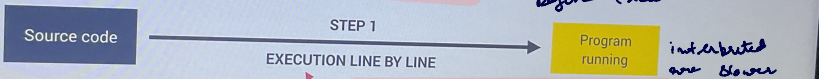
# COMPUTER SCIENCE SIDENOTE: COMPILATION VS. INTERPRETATION

**Compilation:** Entire code is converted into machine code at once, and written to a binary file that can be executed by a computer.



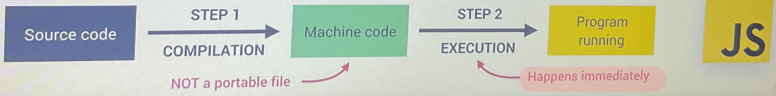

**Interpretation:** Interpreter runs through the source code and executes it line by line.

Code compile first before execution

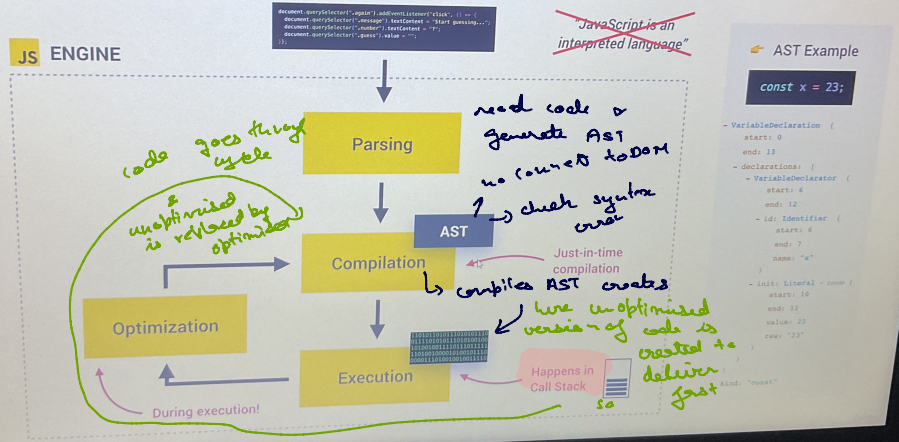



interpreted are slower than compiled

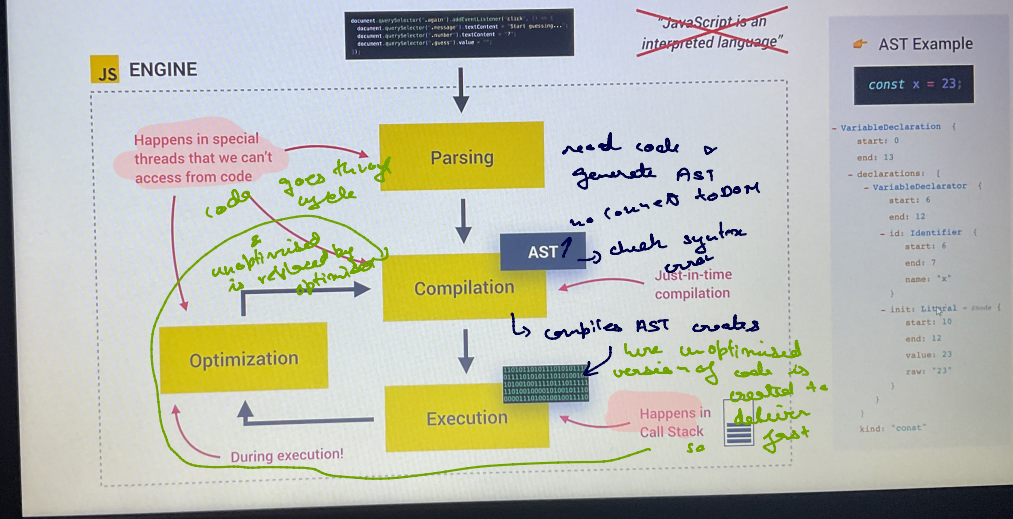
**Just-in-time (JIT) compilation:** Entire code is converted into machine code at once, then executed immediately.



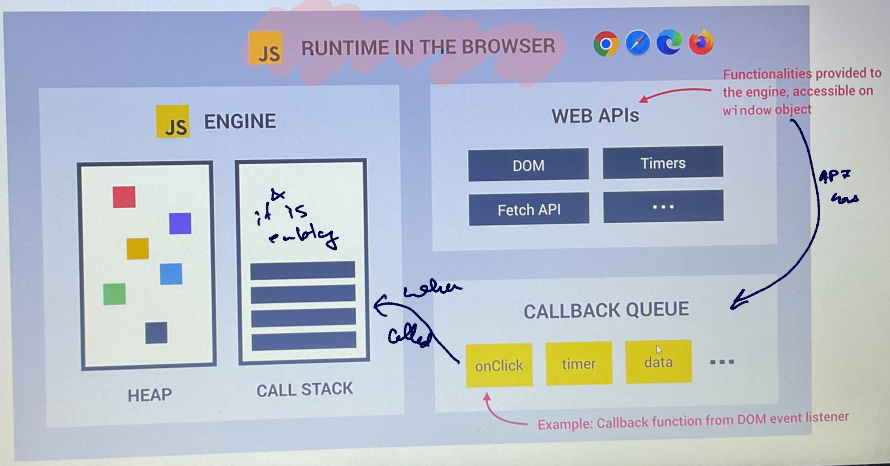
# MODERN JUST-IN-TIME COMPILATION OF JAVASCRIPT



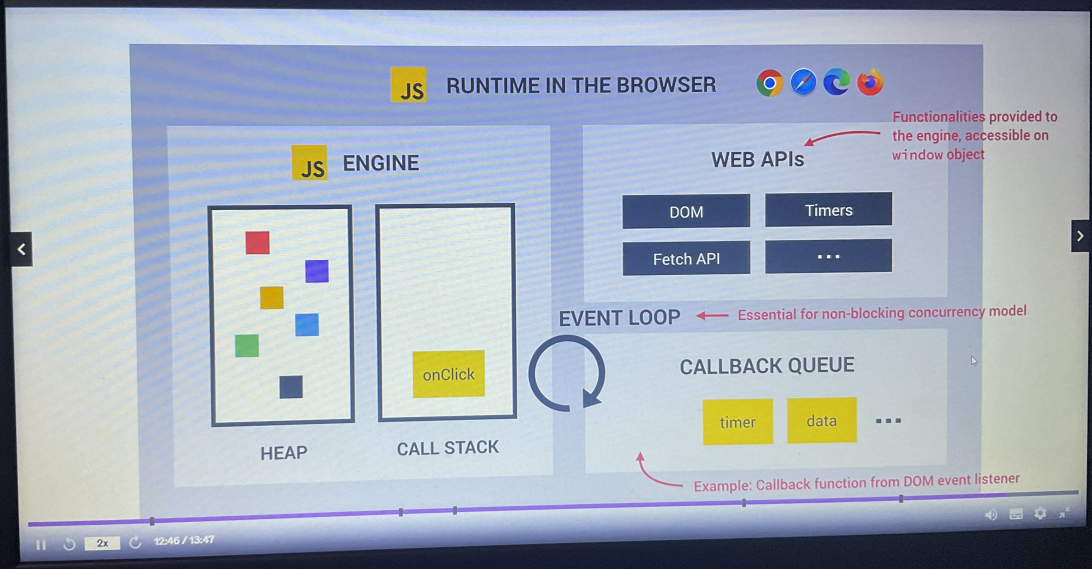
# MODERN JUST-IN-TIME COMPILATION OF JAVASCRIPT



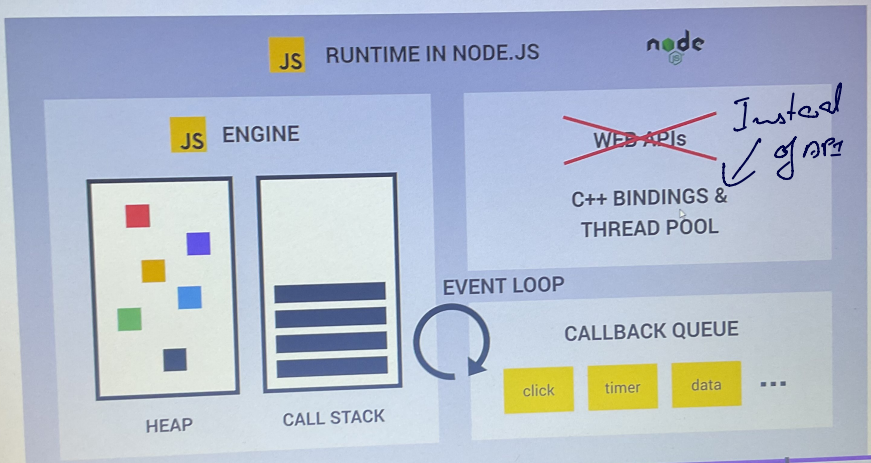
# THE BIGGER PICTURE: JAVASCRIPT RUNTIME



90. The JavaScript Engine and Runtime



90. The JavaScript Engine and Runtime



# WHAT IS AN EXECUTION CONTEXT?

Human-readable code:

```
const name = 'Jonas';

const first = () => {
  let a = 1;
  const b = second();
  a = a + b;
  return a;
};

function second() {
  var c = 2;
  return c;
}
```

Top level code

only executed when called

Function body only executed when called!

Compilation

Creation of **global execution context** (for top-level code)

NOT inside a function

force must only be executed when called

EXECUTION

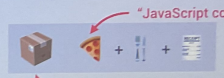
Execution of **top-level code** (inside global EC)

Execution of **functions** and waiting for **callbacks**

Example: click event callback

## EXECUTION CONTEXT

Environment in which a piece of JavaScript is executed. Stores all the necessary information for some code to be executed.




- Exactly **one** global execution context (EC): Default context, created for code that is not inside any function (top-level).
- One execution context **per function**: For each function call, a new execution context is created.

All together make the call stack

same for methods

# EXECUTION CONTEXT IN DETAIL

## WHAT'S INSIDE EXECUTION CONTEXT?

1. Variable Environment
   - let, const and var declarations
   - Functions
   - arguments object
2. Scope chain
3. ~~this~~ keyword

NOT in arrow functions!

how all arguments we pass into function

```
const name = 'Jonas';

const first = () => {
  let a = 1;
  const b = second(7, 9);
  a = a + b;
  return a;
};

function second(x, y) {
  var c = 2;
  return c;
}

const x = first();
```

**Global**

```
name = 'Jonas'
first = <function>
second = <function>
x = <unknown>
```

Literally the function code

Need to run `first()` first

**first()**

```
a = 1
b = <unknown>
```

Need to run `second()` first

**second()**

```
c = 2
arguments = [7, 9]
```

Generated during "creation phase", right before execution

Array of passed arguments. Available in all "regular" functions (not arrow)

(Technically, values only become known during execution)

DELL

91. Execution Contexts and The Call Stack

THE CALL STACK

- Compiled code starts execution

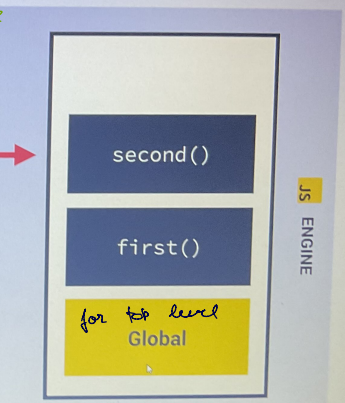
```
const name = 'Jonas';

const first = () => {
  let a = 1;
  const b = second(7, 9);
  a = a + b;
  return a;
};

function second(x, y) {
  var c = 2;
  return c;
}

const x = first();
```

JS can only execute one func at one time

then first() parsed

then this

after second execute




"Place" where execution contexts get stacked on top of each other, to keep track of where we are in the execution → CALL STACK



DELL

91. Execution Contexts and The Call Stack

THE CALL STACK

Compiled code starts execution

```
const name = 'Jonas';

const first = () => {
  let a = 1;
  const b = second(7, 9);
  a = a + b;
  return a;
};

function second(x, y) {
  var c = 2;
  return c;
}

const x = first();
```

over
done

removed

first()

Global

JS ENGINE

"Place" where execution contexts get stacked on top of each other, to keep track of where we are in the execution

CALL STACK

2x 15:42 / 17:45

92. Scope and The Scope Chain

SCOPING AND SCOPE IN JAVASCRIPT: CONCEPTS

EXECUTION CONTEXT

- Variable environment
- Scope chain
- this keyword

## SCOPE CONCEPTS

- **Scoping:** How our program's variables are **organized** and **accessed**. *"Where do variables live?"* or *"Where can we access a certain variable, and where not?"*;
- **Lexical scoping:** Scoping is controlled by **placement** of functions and blocks in the code;
- **Scope:** Space or environment in which a certain variable is **declared** (*variable environment in case of functions*). There is **global** scope, **function** scope, and **block** scope;
- **Scope of a variable:** Region of our code where a certain variable can be **accessed**.

they how variables are organized & accessed

# 92. Scope and The Scope Chain

THE 3 TYPES OF SCOPE

func & variables are almost same

## GLOBAL SCOPE

```
const me = 'Jonas';
const job = 'teacher';
const year = 1989;
```

Top level code

- Outside of **any** function or block
- Variables declared in global scope are accessible **everywhere**

## FUNCTION SCOPE

```
function calcAge(birthYear) {
  const now = 2037;
  const age = now - birthYear;
  return age;
}

console.log(now); // ReferenceError
```

- Variables are accessible only **inside function**, NOT outside
- Also called local scope

## BLOCK SCOPE (ES6)

```
if (year >= 1981 && year <= 1996) {
  const millenial = true;
  const food = 'Avocado toast';
}
```

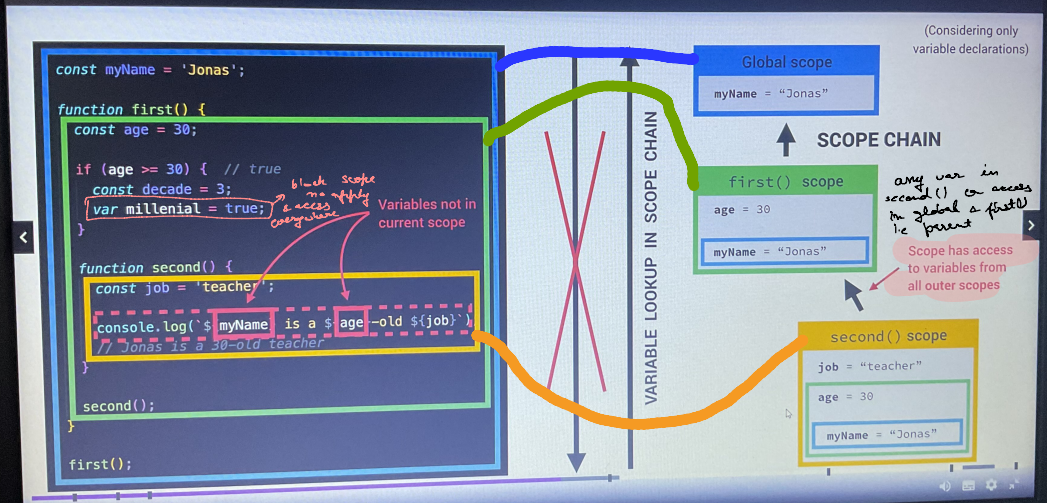
Example: `if` block, `for` loop block, etc.

```
console.log(millenial); // ReferenceError
```

- Variables are accessible only **inside block** (block scoped)
- ⚠ **HOWEVER**, this only applies to `let` and `const` variables!
  var would still be accessible
- Functions are **also block scoped** (only in strict mode)

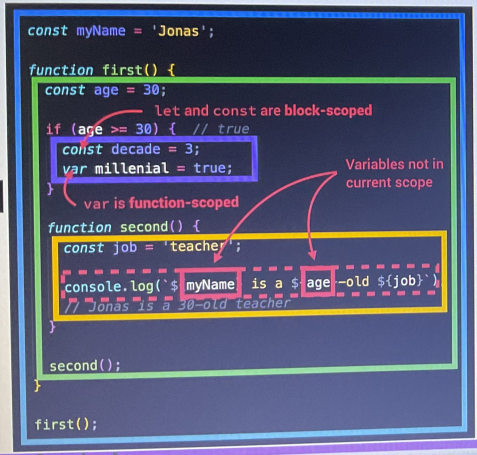
2x 8:36 / 25:37

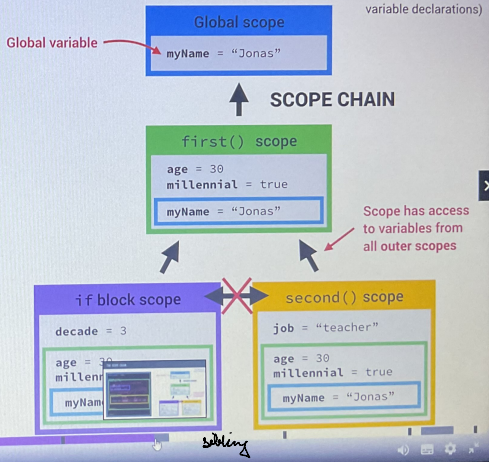
92. Scope and The Scope Chain




2x 13:04 / 25:37

# 92. Scope and The Scope Chain




2x 16:44 / 25:37

92. Scope and The Scope Chain

# SCOPE CHAIN VS. CALL STACK

```
const a = 'Jonas';
first();

function first() {
  const b = 'Hello!';
  second();

  function second() {
    const c = 'Hi!';
    third();
  }
}

function third() {
  const d = 'Hey!';
  console.log(d + c + b + a);
  // ReferenceError
}
```

c and b can NOT be found in third() scope!

Variable environment (VE)

**third() EC**
d = "Hey!"

**second() EC**
c = "Hi!"

**first() EC**
b = "Hello!"
second = <function>

**Global EC**
a = "Jonas"
first = <function>
third = <function>

**CALL STACK**
*Order in which functions were called*

third() VE

Global VE

**Global scope**
a = "Jonas"
first = <function>
third = <function>

**first() scope**
b = "Hello!"
second = <function>
a = "Jonas"
first = <function>
third = <function>

**third() scope**
d = "Hey!"
a = "Jonas"
first = <function>
third = <function>

**second() scope**
c = "Hi!"
b = "Hello!"
second = <function>
a = "Jonas"
first = <function>
third = <function>

**SCOPE CHAIN**
*Order in which functions are **written in the code***

Has **nothing** to do with order in which functions were **called**!

2x 22:56 / 25:37

# SUMMARY

- Scoping asks the question *"Where do variables live?"* or *"Where can we access a certain variable, and where not?"*;
- There are 3 types of scope in JavaScript: the global scope, scopes defined by functions, and scopes defined by blocks;
- Only `let` and `const` variables are block-scoped. Variables declared with `var` end up in the closest function scope;
- In JavaScript, we have lexical scoping, so the rules of where we can access variables are based on exactly where in the code functions and blocks are written;
- Every scope always has access to all the variables from all its outer scopes. This is the scope chain!
- When a variable is not in the current scope, the engine looks up in the scope chain until it finds the variable it's looking for. This is called variable lookup;
- The scope chain is a one-way street: a scope will never, ever have access to the variables of an inner scope;
- The scope chain in a certain scope is equal to adding together all the variable environments of the all parent scopes;
- The scope chain has nothing to do with the order in which functions were called. It does not affect the scope chain at all!