FRONT-END

LECTURE VI/VI

SCHEDULE

Lecture IV (today)

I. HomeWork
   A. Anyone?
II. Prototypes
III. Hoisting
IV. Closures
V. Current state of affairs

Homework

- Continue working on Feature
- Document findings
  - prototypes
  - closure
  - hoisting
- Chapter 11. async programming

Homework

VI/VI

Prototypes

synopsis

A Prototype in JavaScript is a property on a function that points to an Object

Prototypes

SYNOPSIS

- const students = []; is actually:
  const students = new Array();

  Meaning we are creating a new instance of Array.

  Built in methods which live on the Array's prototype are now at our disposal. We can validate by typing the following in our browser console; Array.prototype

  This means that all of the Array's methods live on the Array prototype.

Prototypes

```
> Array.prototype
< ▼ [constructor: ƒ, concat: ƒ, copyWithin: ƒ, fill: ƒ, find: ƒ, …]
    ▶ concat: ƒ concat()
    ▶ constructor: ƒ Array()
    ▶ copyWithin: ƒ copyWithin()
    ▶ entries: ƒ entries()
    ▶ every: ƒ every()
    ▶ fill: ƒ fill()
    ▶ filter: ƒ filter()
    ▶ find: ƒ find()
    ▶ findIndex: ƒ findIndex()
    ▶ flat: ƒ flat()
    ▶ flatMap: ƒ flatMap()
    ▶ forEach: ƒ forEach()
    ▶ includes: ƒ includes()
    ▶ indexOf: ƒ indexOf()
    ▶ join: ƒ join()
    ▶ keys: ƒ keys()
    ▶ lastIndexOf: ƒ lastIndexOf()
      length: 0
    ▶ map: ƒ map()
```

Synopsis

Here we see all of the methods that live on the Arrays prototype

Prototypes

SYNOPSIS

So: prototypes are just objects that every function has - we, as developers, can add functions to a specific prototype (see next slide)

Prototypes

```
function Animal (name, species, power) {

  let animal = Object.create(Animal.prototype);

  animal.name = name;
  animal.species = species;
  animal.power = power;

  return animal;
}

Animal.prototype.eat = function (amount) {
  console.log(`${this.name} is eating`);
  this.power += amount;
}

Animal.prototype.sleep = function (length) {
  console.log(`${this.name} is sleeping`);
  this.power += length;
}

const Snoop = Animal('Snoop', 'Dog', 10);
```

Synopsis

> The function Animal is called a Constructor function, because it constructs an object for us.

> We can add methods to the constructors prototype

> This for the purpose of sharing methods across all instances of a particular constructor function.

This can be done easier.. Namely; by using the ‘new’ keyword.

Prototypes

synopsis

```
function Animal (name, species, power) {
  // the new keyword takes care of let animal = ...
  // and instead of having to create
  // a variable animal - JavaScript creates the keyword 'this';
  // let this = Object.create(Animal.prototype);

  // we can now set the values by referring to 'this'
  this.name = name;
  this.species = species;
  this.power = power;
  // the new keyword also takes care of 'return animal'
  // return this;
}
```

```
const snoop = new Animal('Snoop', 'Dog', 10);
```

Using the new keyword changes several things in our code.

We no longer have to specify Object.create(); JS creates an Object for us.

JS also sets the value of the ‘this’ keyword equal to the creation of this object. (in the previous example this would refer to the window object).

Last but not least - JS also implicitly returns the ‘this’ keyword.

Prototypes

SYNOPSIS

In essence - what we did, was creating a class.

Since the release of ES6, JavaScript has a built in way to create Classes.

We could reproduce the above mentioned code by creating the following class

```
class Animal {
  // the constructor function will create the object
  // which is similar to this.name and this.species and this.power
  constructor(name, species, power) {
    this.name = name;
    this.species = species;
    this.power = power;
  }

  // we can now write our functions inside the Animal class
  eat(amount) {
    console.log(`${this.name} is eating`);
    this.power += amount;
  }

  sleep(length) {
    console.log(`${this.name} is sleeping`);
    this.power += length;
  }
}

const Snoop = new Animal('Snoop', 'Dog', 10);
```

Prototypes

Classes are **syntactical** sugar over the existing way we described earlier.

So we first have to understand what classes get compiled to; namely - function invocations with the new keyword.

Hoisting

synopsis

In order to understand hoisting, closures and scope we need to have some more context of the way JS works.

When JS runs your code - it creates an execution context. Top level code refers to the **Global Execution Context**.

Whenever a function is called a **Function Execution Context** is created.

The execution context consists of two phases;

- creation phase
- execution phase

In the creation phase all variable declarations get ‘hoisted’ - meaning; memory allocation and assigned a value of undefined. Function declarations get placed directly into memory.

Sidestep: (Scope?)

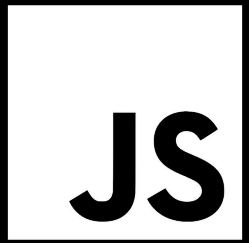
The current context of execution mdn

That sounds familiar :D

HOISTING

A strict definition of hoisting suggests that variable and function declarations are **physically moved to the top of your code.**

Eloquent Javascript by marijn haverbeke

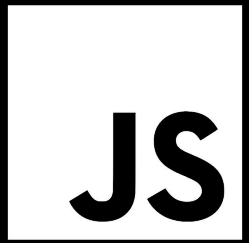


HOISTING

JavaScript only **hoists declarations, not assignments.** If a variable is declared and assigned after using it, the value will be undefined.

```
> let name;
  console.log(`Hi ${name}`);

  name = 'Jan';
  Hi undefined
```

MDN - Hoisting



Closures

declaration

Say we have a function, and inside this function we have several variables declared. For the sake of this example, let's call this function the outer function.

However, inside of this function, we also have another function declared, let's call this function the 'inner function'.

Closures allow the 'inner function' to access the variables inside of its 'outer function'. Even if the execution context of the outer function already happened.

The inner function creates a closure over the outer functions execution context

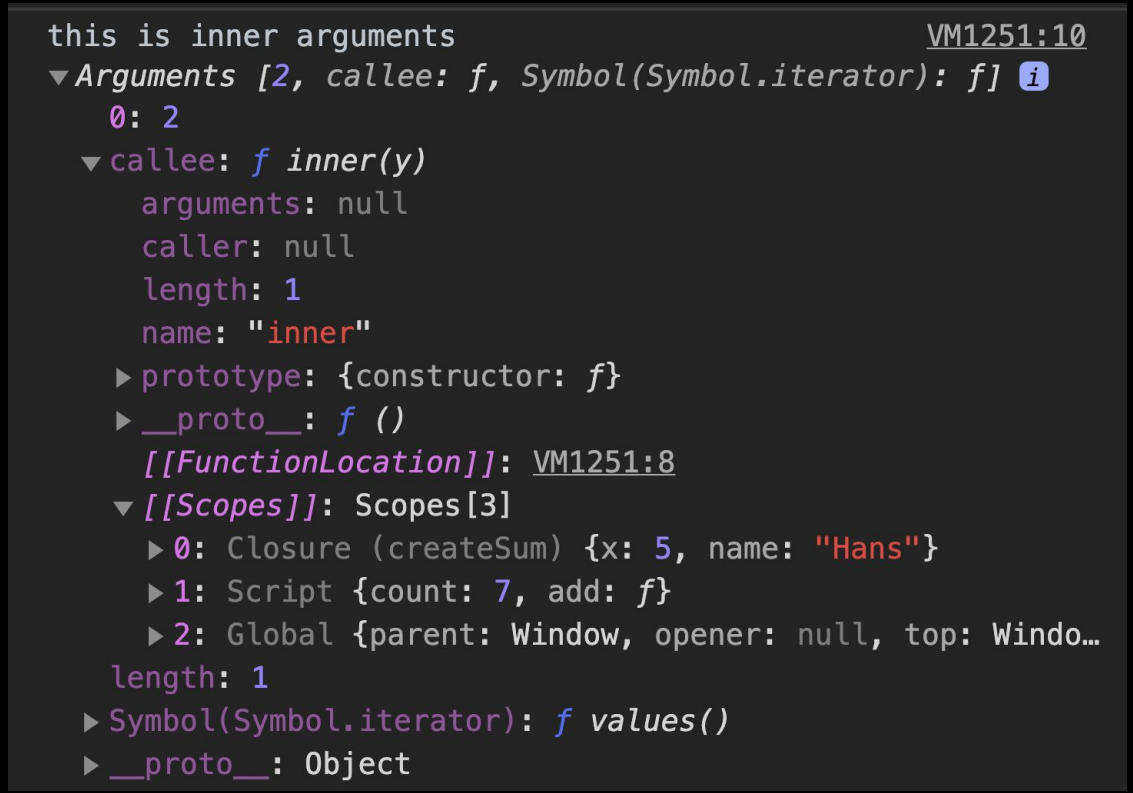
Closures

Closure in action

```
> let count = 0;

  function createSum(x) {
    console.log('this is x', x);
    console.log('this is arguments', arguments);
    const name = 'Hans';

    return function inner(y) {
      // creates a closure around the outer function execution
      // context and has access to the outer scope
      console.log('this is inner arguments', arguments);
      console.log('this is y', y);
      console.log('this is name', name);
      return x + y;
    };
  }

  let add = createSum(5);
  count += add(2);
```

Closures

Closure ->



SYNOPSIS

Homework

- Document your findings of:
  - Prototypes in JS (for your own good)
  - Hoisting (a1 criteria)
  - Closures (a1 criteria)
- Read h11 (async)
- Continue working on FE feature

EXIT;

see you Next();