JONAS.IO
SCHMEDTMANN

THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

@JONASSCHMEDTMAN

SECTION

OBJECT ORIENTED PROGRAMMING (OOP) WITH JAVASCRIPT

LECTURE

WHAT IS OBJECT-ORIENTED PROGRAMMING?

JS

WHAT IS OBJECT-ORIENTED PROGRAMMING? (OOP)

OOP

Style of code, "how" we write and organize code

- Object-oriented programming (OOP) is a programming paradigm based on the concept of objects;
- We use objects to **model** (describe) real-world or abstract features;
- Objects may contain data (properties) and code (methods). By using objects, we pack **data and the corresponding behavior** into one block;
- In OOP, objects are **self-contained** pieces/blocks of code;
- Objects are **building blocks** of applications, and **interact** with one another;
- Interactions happen through a **public interface** (API): methods that the code **outside** of the object can access and use to communicate with the object;
- OOP was developed with the goal of **organizing** code, to make it **more flexible and easier to maintain** (avoid "spaghetti code").

E.g. user or todo list item

E.g. HTML component or data structure

Data

```
const user = {
  user: 'jonas',
  password: 'dk23s',

  login(password) {
    // Login logic
  },
  sendMessage(str) {
    // Sending logic
  }
}
```

Behaviour

CLASSES AND INSTANCES (TRADITIONAL OOP)

Like a blueprint from which we can create new objects

CLASS

```
User {
  user
  password
  email

  login(password) {
    // Login logic
  }
  sendMessage(str) {
    // Sending logic
  }
}
```

Just a representation, NOT actual JavaScript syntax!

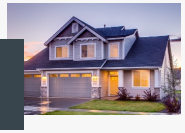
JavaScript does NOT support *real* classes like represented here

new User('jonas')

new User('mary')

new User('steven')

Instance

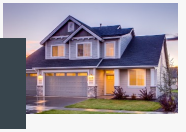


```
{
  user = 'jonas'
  password = 'dk23s'
  email = 'hello@jonas.io'

  login(password) {
    // Login logic
  }
  sendMessage(str) {
    // Sending logic
  }
}
```

New object created from the class. Like a *real* house created from an *abstract* blueprint

Instance



```
{
  user = 'mary'
  password = 'qwerty23'
  email = 'mary@test.com'

  login(password) {
    // Login logic
  }
  sendMessage(str) {
    // Sending logic
  }
}
```

Instance

```
{
  user = 'steven'
  password = '5p8dz32dd'
  email = 'steven@tes.co'

  login(password) {
    // Login logic
  }
  sendMessage(str) {
    // Sending logic
  }
}
```

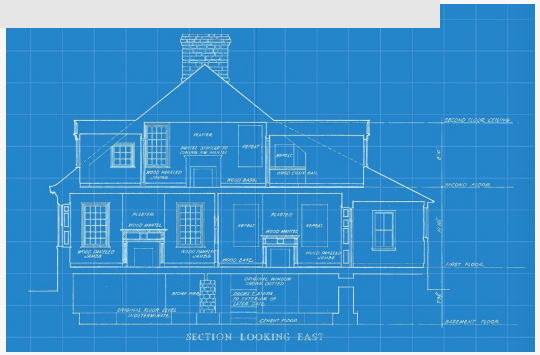
☝ Conceptual overview: it works a bit **differently** in JavaScript. Still important to understand!

THE 4 FUNDAMENTAL OOP PRINCIPLES

- Abstraction
- Encapsulation
- Inheritance
- Polymorphism

The 4 fundamental principles of Object-Oriented Programming

🤔 *"How do we actually design classes? How do we model real-world data into classes?"*



PRINCIPLE 1: ABSTRACTION

- **Abstraction**
- Encapsulation
- Inheritance
- Polymorphism

```
Phone {
  charge
  volume
  voltage
  temperature

  homeBtn() {}
  volumeBtn() {}
  screen() {}
  verifyVolt() {}
  verifyTemp() {}
  vibrate() {}
  soundSpeaker() {}
  soundEar() {}
  frontCamOn() {}
  frontCamOff() {}
  rearCamOn() {}
  rearCamOff() {}
}
```

Real phone

Abstracted phone

```
Phone {
  charge
  volume

  homeBtn() {}
  volumeBtn() {}
  screen() {}
}
```

Details have been abstracted away

Do we *really need* all these low-level details?

👉 **Abstraction:** Ignoring or hiding details that **don't matter**, allowing us to get an **overview** perspective of the *thing* we're implementing, instead of messing with details that don't really matter to our implementation.

PRINCIPLE 2: ENCAPSULATION

- Abstraction
- Encapsulation
- Inheritance
- Polymorphism

NOT accessible from outside the class!

STILL accessible from within the class!

STILL accessible from within the class!

NOT accessible from outside the class!

```
User {
  user
  private password
  private email

  login(word) {
    this.password === word
  }
  comment(text) {
    this.checkSPAM(text)
  }
  private checkSPAM(text) {
    // Verify logic
  }
}
```

Again, NOT actually JavaScript syntax (the `private` keyword doesn't exist)

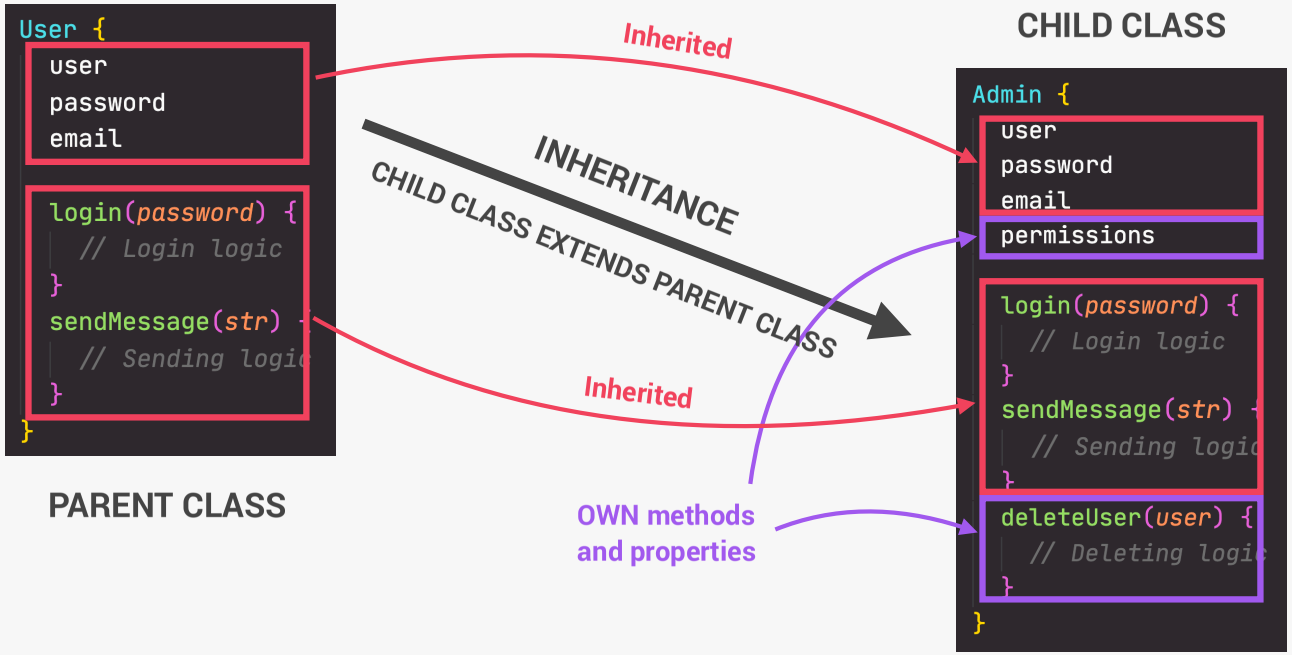
WHY?

- 👉 Prevents external code from accidentally manipulating internal properties/state
- 👉 Allows to change internal implementation without the risk of breaking external code

👉 **Encapsulation:** Keeping properties and methods **private** inside the class, so they are **not accessible from outside the class**. Some methods can be **exposed** as a public interface (API).

PRINCIPLE 3: INHERITANCE

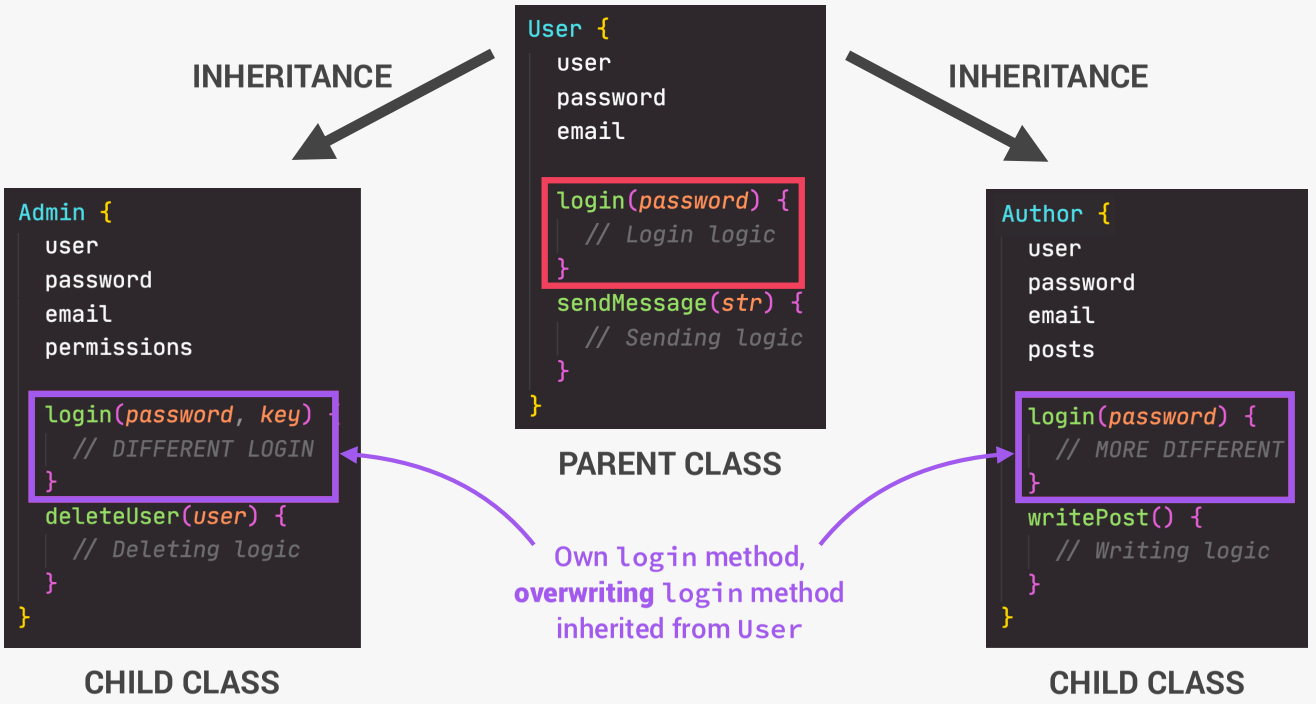
- Abstraction
- Encapsulation
- Inheritance
- Polymorphism



👉 **Inheritance:** Making all properties and methods of a certain class **available to a child class**, forming a hierarchical relationship between classes. This allows us to **reuse common logic** and to model real-world relationships.

PRINCIPLE 4: POLYMORPHISM

- Abstraction
- Encapsulation
- Inheritance
- Polymorphism



👉 **Polymorphism:** A child class can **overwrite** a method it inherited from a parent class [it's more complex that that, but enough for our purposes].

JONAS.IO
SCHMEDTMANN

THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

@JONASSCHMEDTMAN

SECTION

OBJECT ORIENTED PROGRAMMING (OOP) WITH JAVASCRIPT

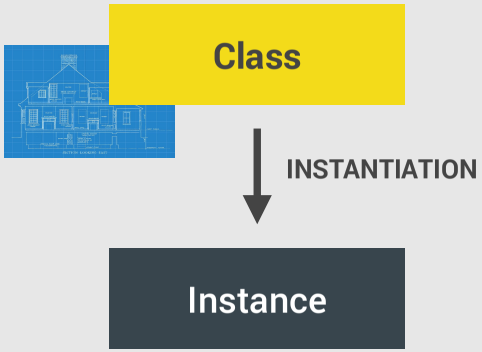
LECTURE

OOP IN JAVASCRIPT

JS

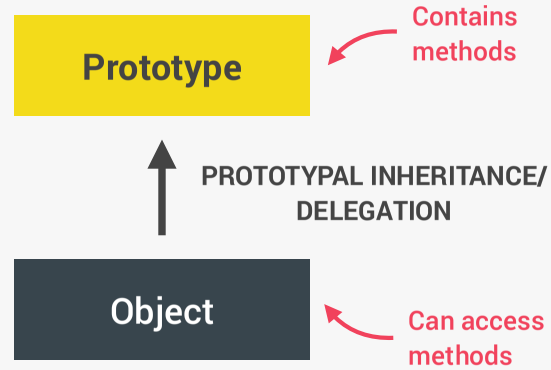
OOP IN JAVASCRIPT: PROTOTYPES

"CLASSICAL OOP": CLASSES



- Objects (instances) are **instantiated** from a class, which functions like a blueprint;
- Behavior (methods) is **copied** from class to all instances.

OOP IN JS: PROTOTYPES



- Objects are **linked** to a prototype object;
- **Prototypal inheritance:** The prototype contains methods (behavior) that are **accessible to all objects linked to that prototype;**
- Behavior is **delegated** to the linked prototype object.

- **Example: Array**

```
const num = [1, 2, 3];
num.map(v => v * 2);
```

MDN web docs
moz://a

```
Array.prototype.keys()
Array.prototype.lastIndexOf()
Array.prototype.map()
```

`Array.prototype` is the **prototype** of all array objects we create in JavaScript

Therefore, **all** arrays have access to the `map` method!

```
▼ƒ Array()
  arguments: (...)
  caller: (...)
  length: 1
  name: "Array"
▼ prototype: Array(0)
  ▶ unique: ƒ ()
    length: 0
  ▶ constructor: ƒ Array()
  ▶ concat: ƒ concat()
  ▶ map: ƒ map()
```

3 WAYS OF IMPLEMENTING PROTOTYPAL INHERITANCE IN JAVASCRIPT

🤔 *"How do we actually create prototypes? And how do we link objects to prototypes? How can we create new objects, without having classes?"*

☝️ **The 4 pillars of OOP are still valid!**

- 👉 Abstraction
- 👉 Encapsulation
- 👉 Inheritance
- 👉 Polymorphism

1 Constructor functions

- 👉 Technique to create objects from a function;
- 👉 This is how built-in objects like Arrays, Maps or Sets are actually implemented.

2 ES6 Classes

- 👉 Modern alternative to constructor function syntax;
- 👉 "Syntactic sugar": behind the scenes, ES6 classes work **exactly** like constructor functions;
- 👉 ES6 classes do **NOT** behave like classes in "classical OOP" (last lecture).

3 `Object.create()`

- 👉 The easiest and most straightforward way of linking an object to a prototype object.

JONAS.IO
SCHMEDTMANN

THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

@JONASSCHMEDTMAN

SECTION

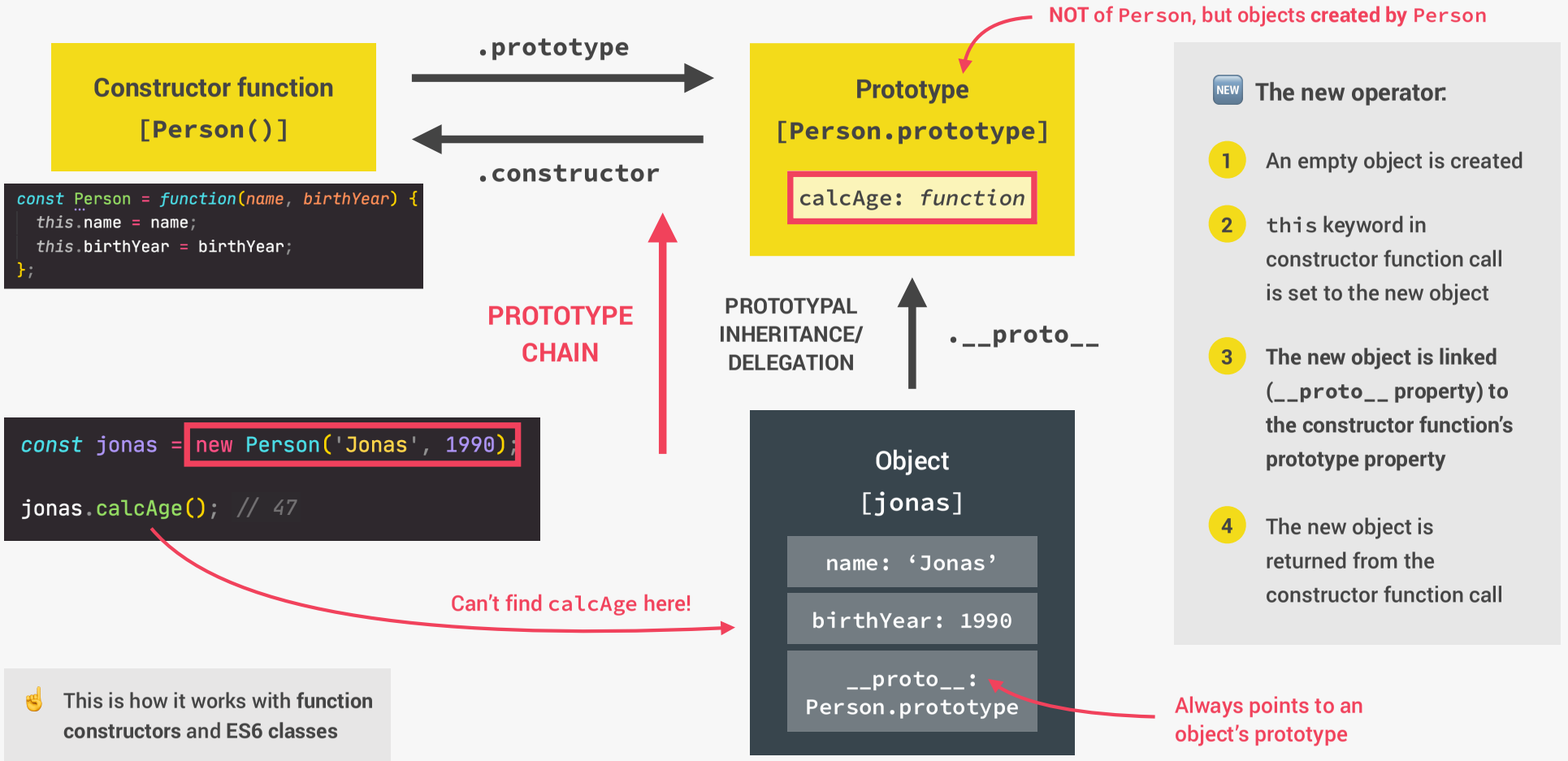
OBJECT ORIENTED PROGRAMMING (OOP) WITH JAVASCRIPT

LECTURE

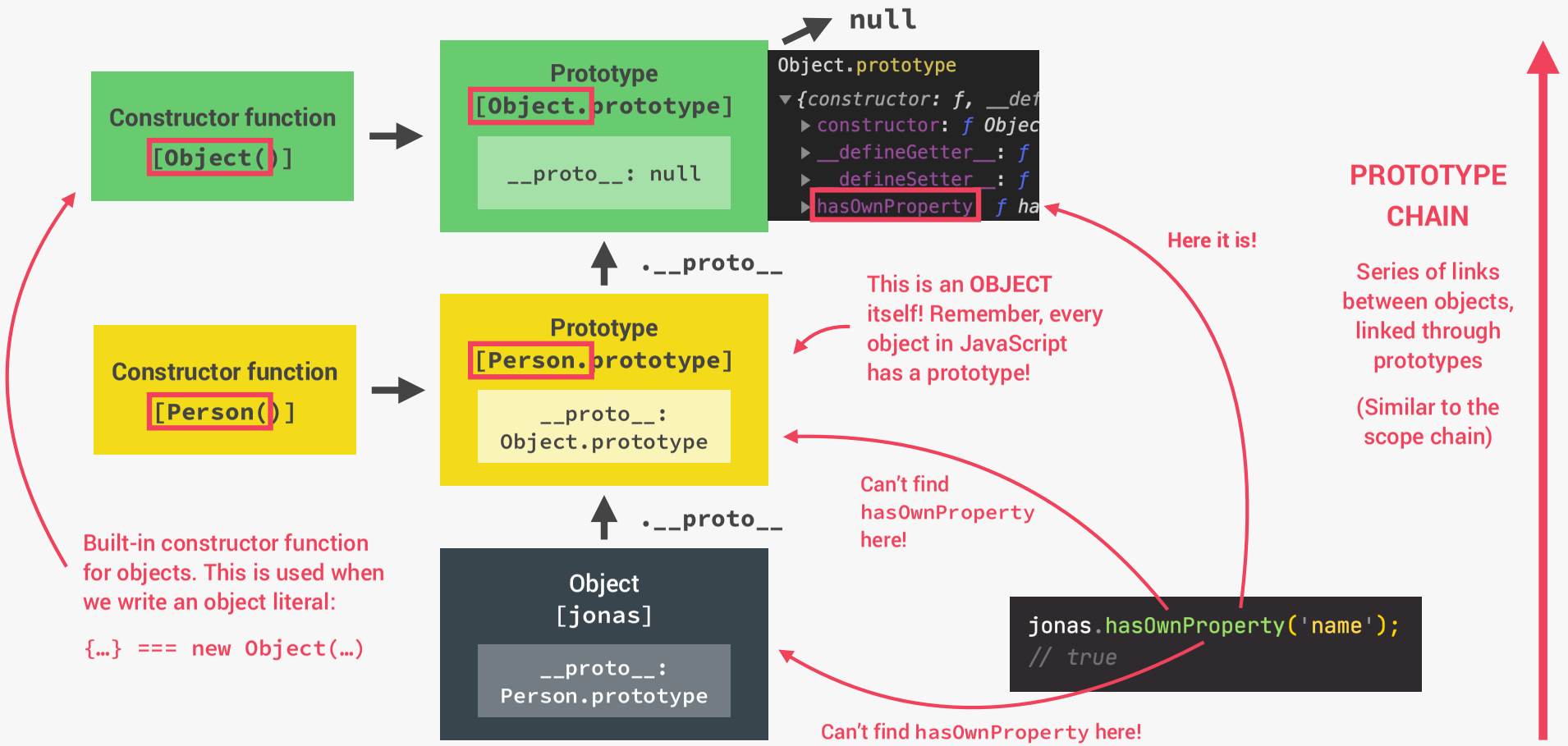
PROTOTYPAL INHERITANCE AND THE PROTOTYPE CHAIN

JS

HOW PROTOTYPAL INHERITANCE / DELEGATION WORKS



THE PROTOTYPE CHAIN



JONAS.IO
SCHMEDTMANN

THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

@JONASSCHMEDTMAN

SECTION

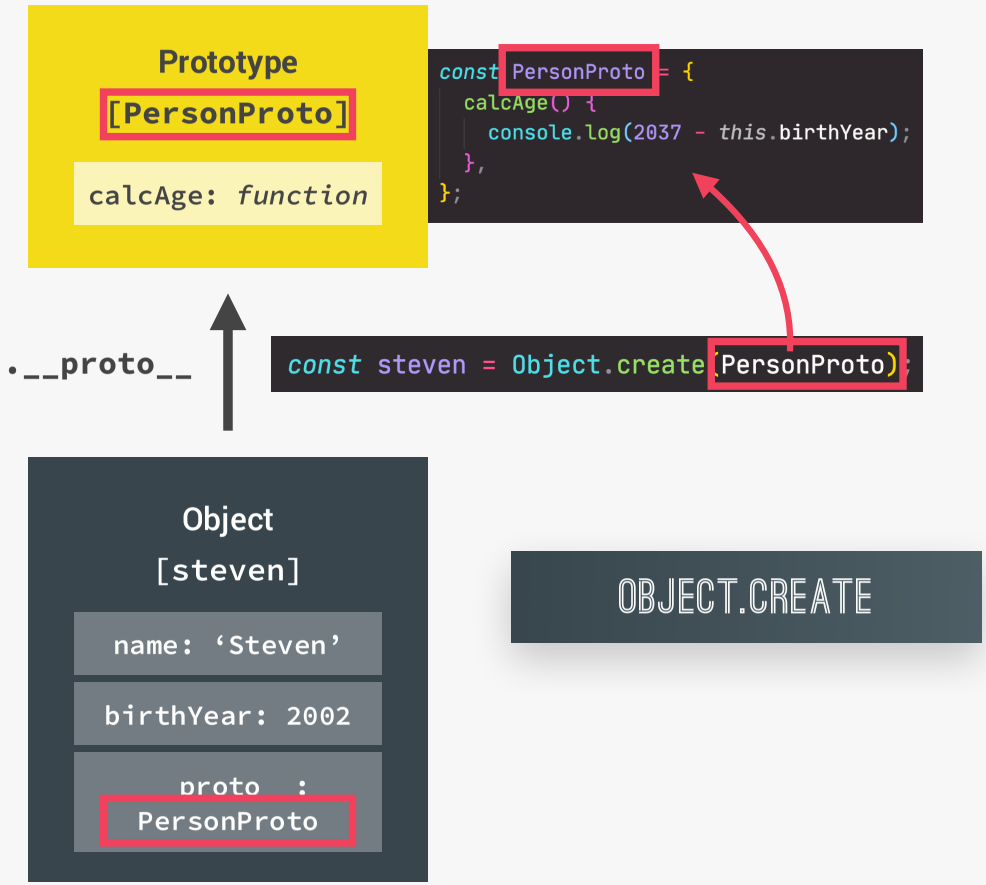
OBJECT ORIENTED PROGRAMMING (OOP) WITH JAVASCRIPT

LECTURE

OBJECT.CREATE

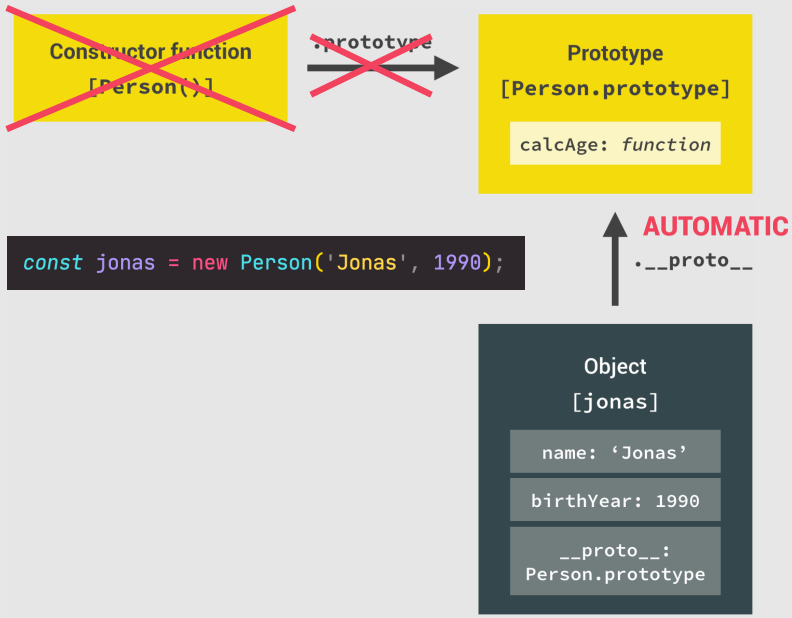
JS

HOW OBJECT.CREATE WORKS



OBJECT.CREATE

CONSTRUCTOR FUNCTIONS



JONAS.IO
SCHMEDTMANN

THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

@JONASSCHMEDTMAN

SECTION

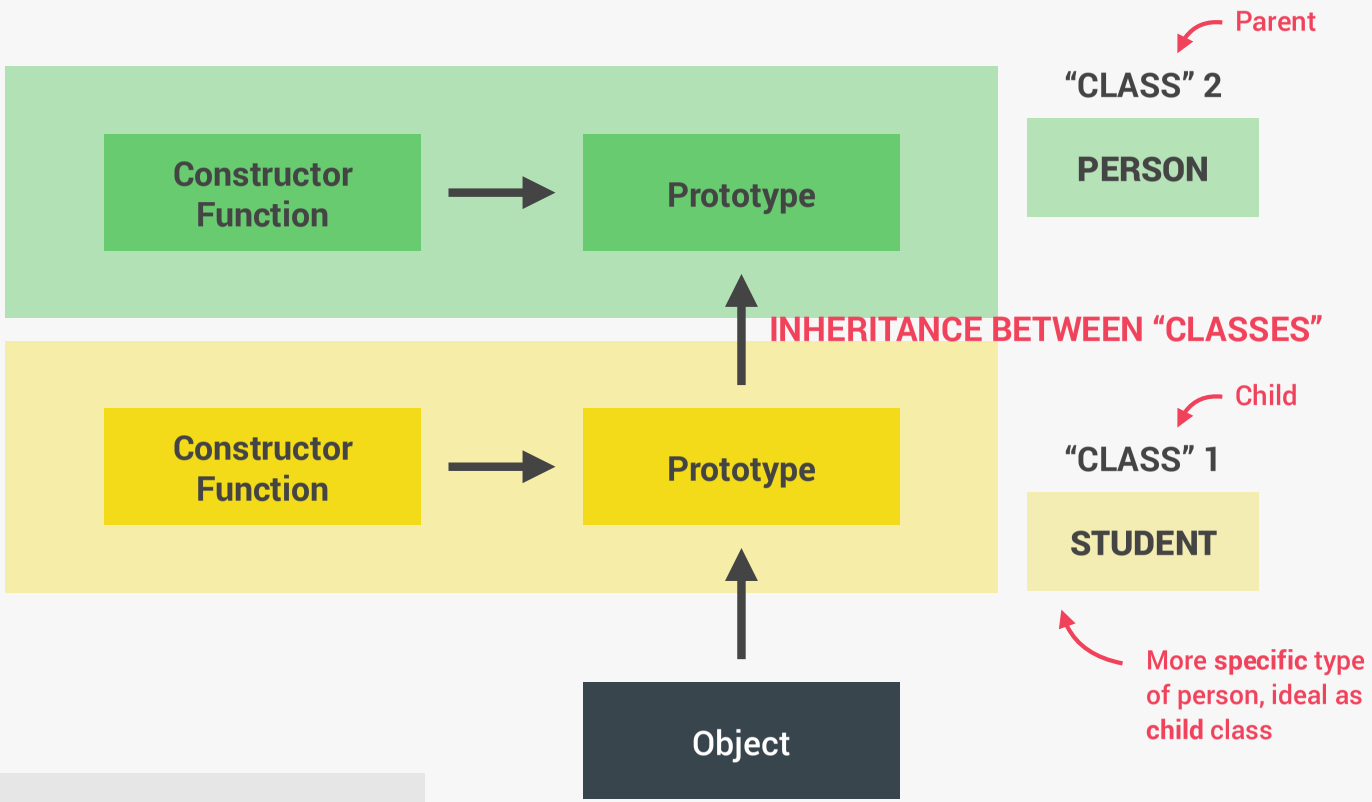
OBJECT ORIENTED PROGRAMMING (OOP) WITH JAVASCRIPT

LECTURE

INHERITANCE BETWEEN "CLASSES": CONSTRUCTOR FUNCTIONS

JS

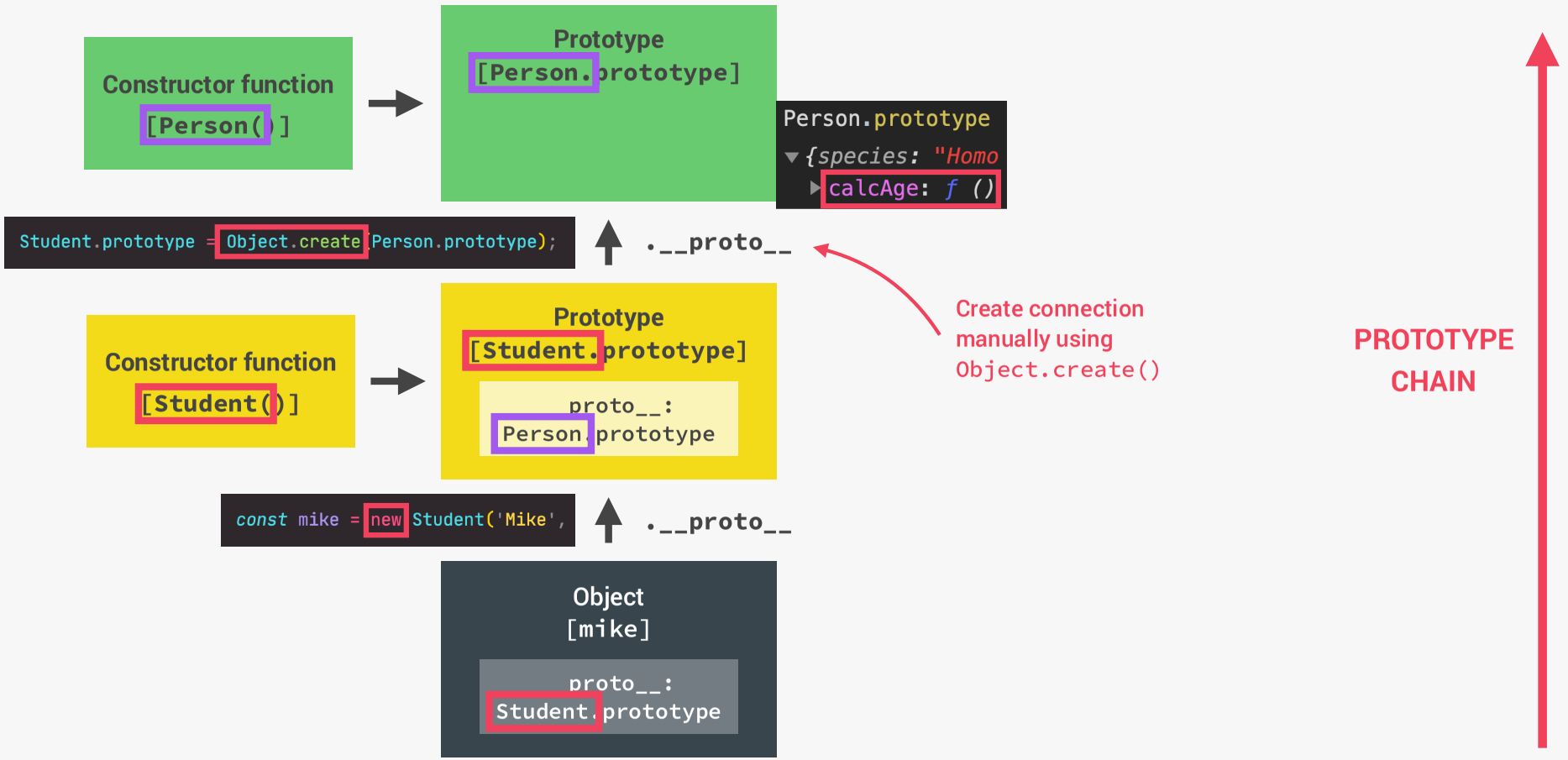
INHERITANCE BETWEEN "CLASSES"



☝ Using class terminology here to make it easier to understand.

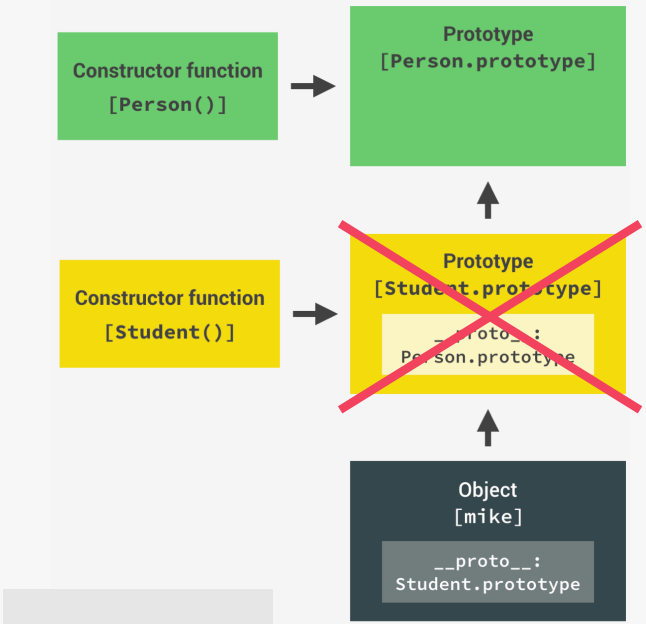
1. Constructor functions
2. ES6 Classes
3. `Object.create()`

INHERITANCE BETWEEN "CLASSES"



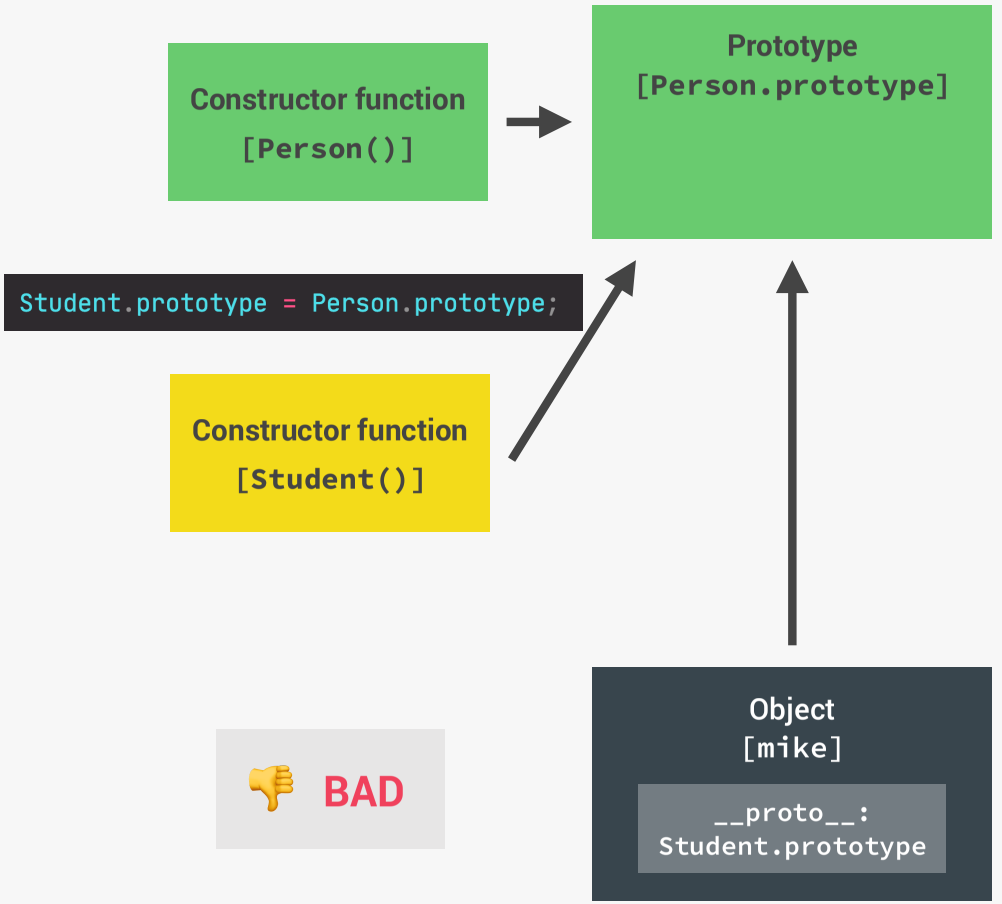
INHERITANCE BETWEEN "CLASSES"

```
Student.prototype = Object.create(Person.prototype);
```



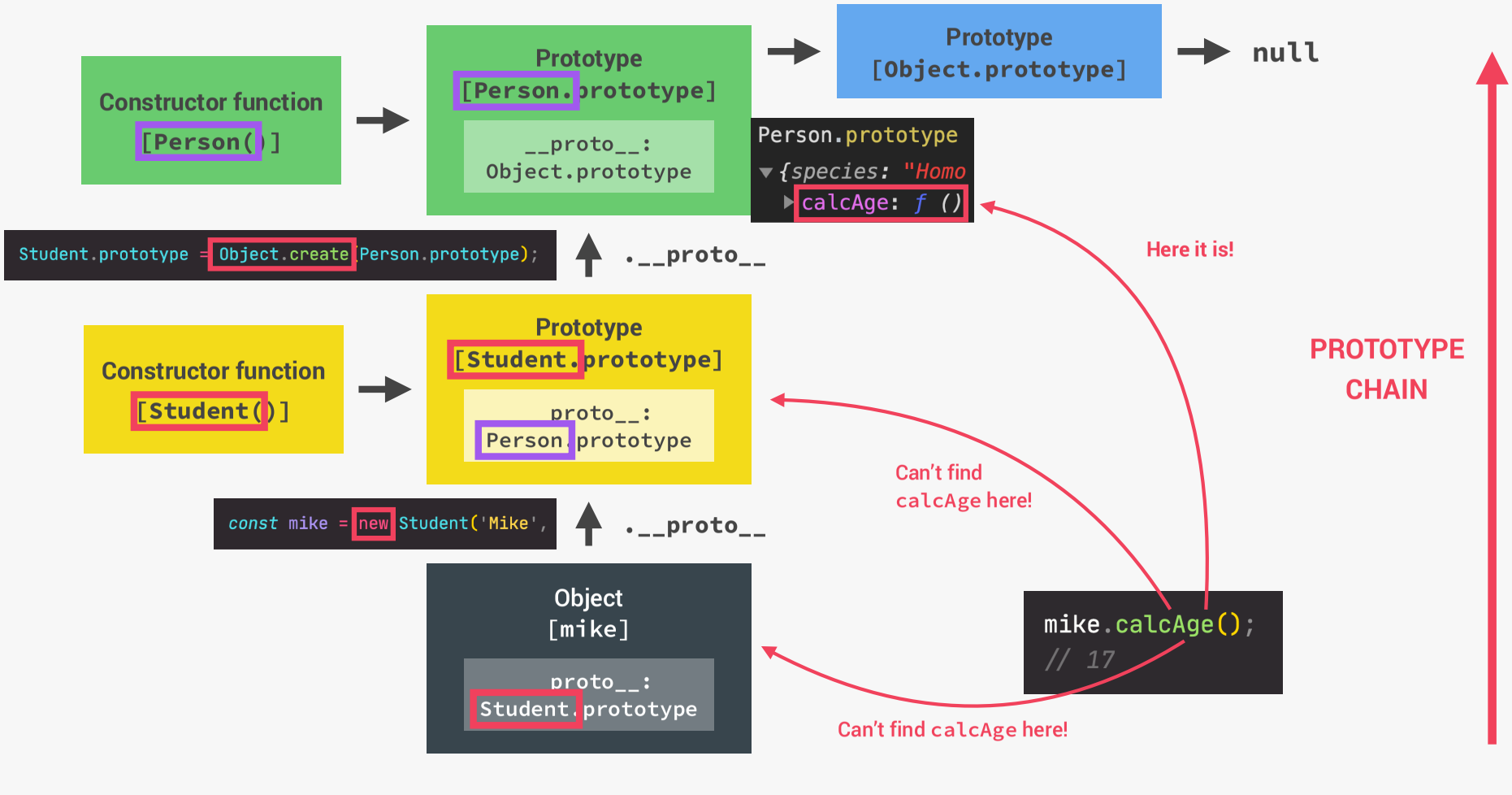
👍 GOOD

```
Student.prototype = Person.prototype;
```



👎 BAD

INHERITANCE BETWEEN "CLASSES"



JONAS.IO
SCHMEDTMANN

THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

@JONASSCHMEDTMAN

SECTION

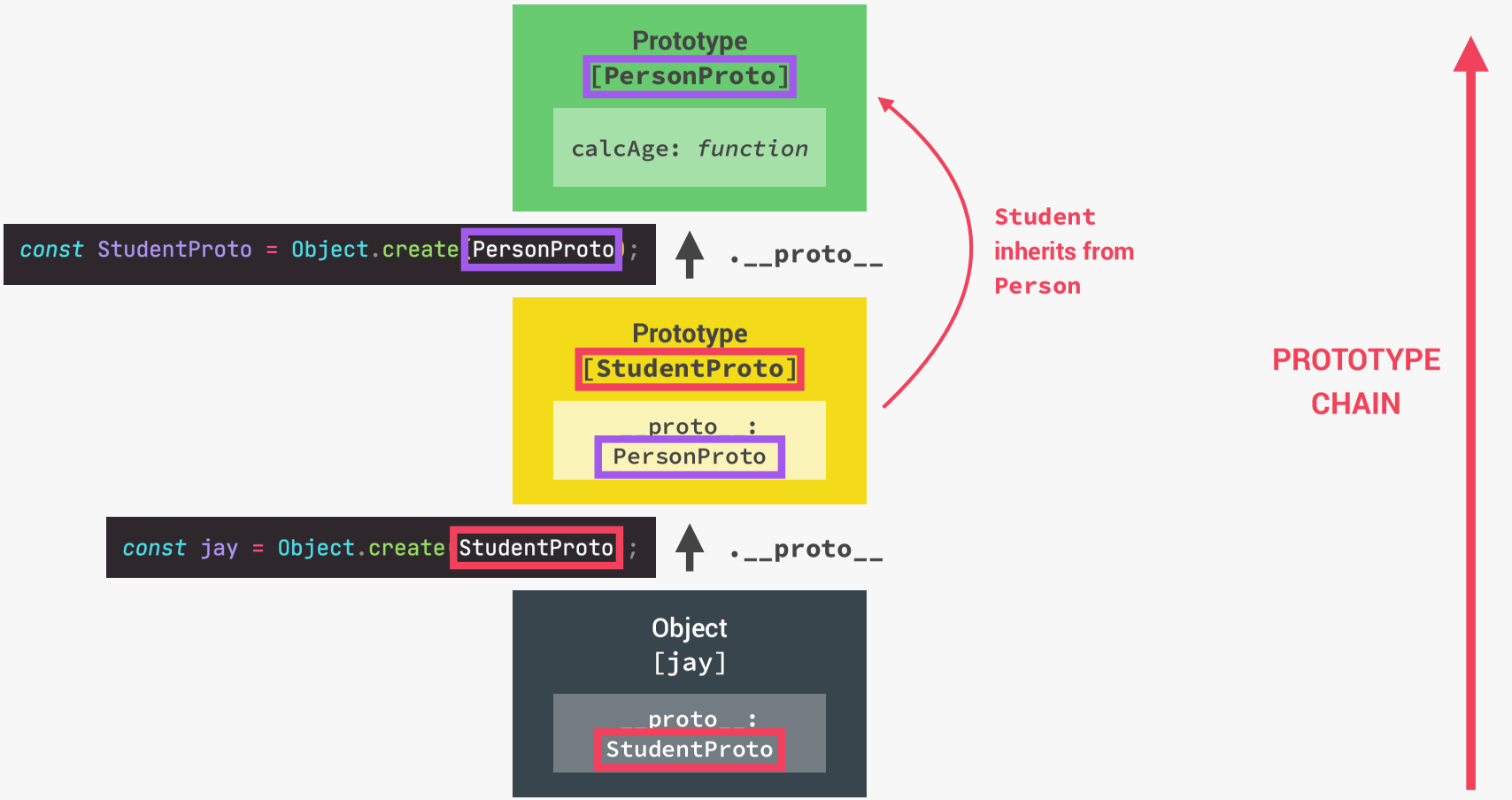
OBJECT ORIENTED PROGRAMMING (OOP) WITH JAVASCRIPT

LECTURE

INHERITANCE BETWEEN "CLASSES": OBJECT.CREATE

JS

INHERITANCE BETWEEN "CLASSES": OBJECT.CREATE



JONAS.IO
SCHMEDTMANN

THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

@JONASSCHMEDTMAN

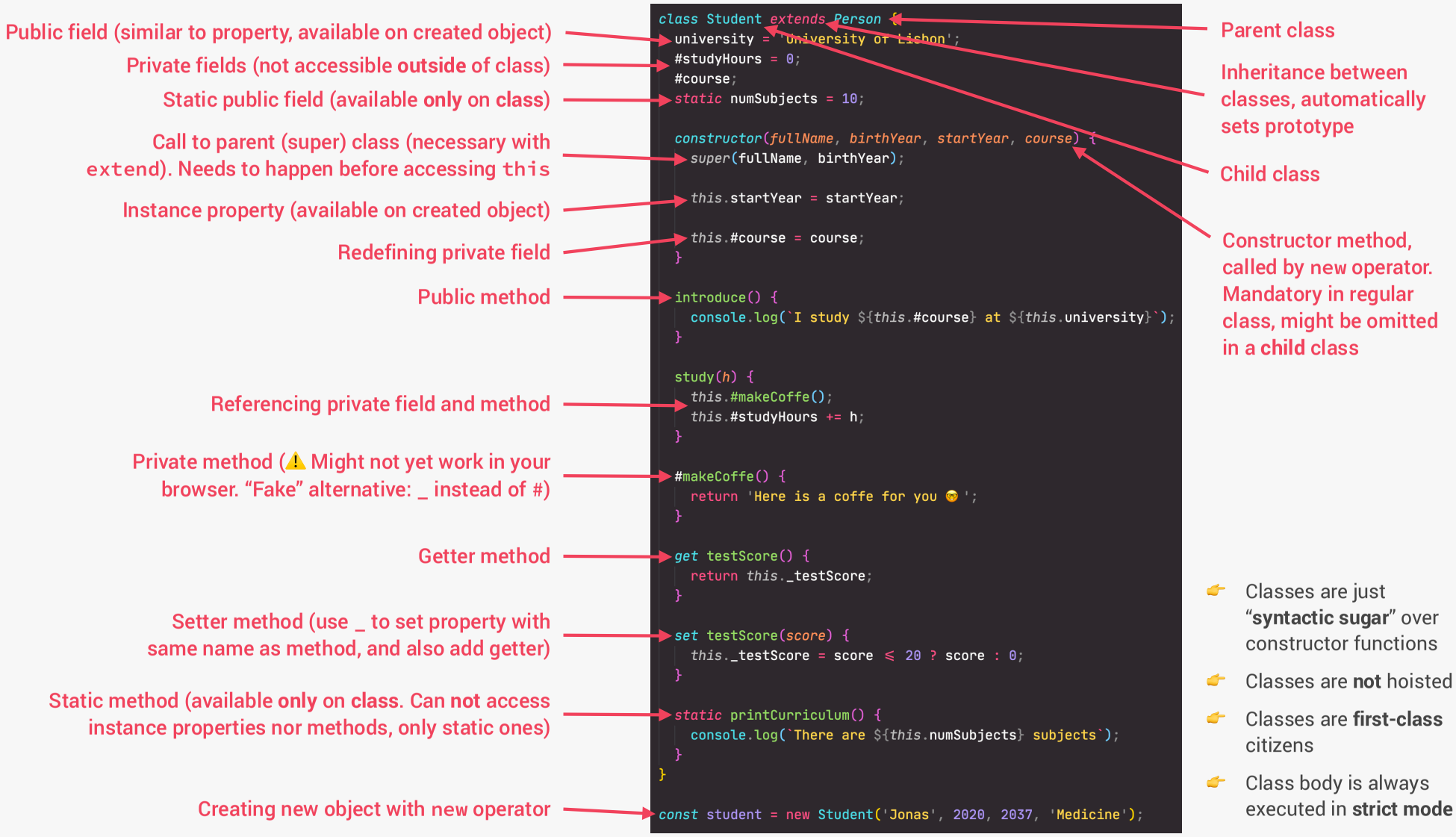
SECTION

OBJECT ORIENTED PROGRAMMING (OOP) WITH JAVASCRIPT

LECTURE

ES6 CLASSES SUMMARY

JS


Public field (similar to property, available on created object)
Private fields (not accessible outside of class)
Static public field (available only on class)
Call to parent (super) class (necessary with extend). Needs to happen before accessing this
Instance property (available on created object)
Redefining private field
Public method
Referencing private field and method
Private method (⚠️ Might not yet work in your browser. "Fake" alternative: _ instead of #)
Getter method
Setter method (use _ to set property with same name as method, and also add getter)
Static method (available only on class. Can not access instance properties nor methods, only static ones)
Creating new object with new operator
class Student extends Person {
university = 'University of Lisbon';
#studyHours = 0;
#course;
static numSubjects = 10;
constructor(fullName, birthYear, startYear, course) {
super(fullName, birthYear);
this.startYear = startYear;
this.#course = course;
}
introduce() {
console.log(`I study ${this.#course} at ${this.university}`);
}
study(h) {
this.#makeCoffe();
this.#studyHours += h;
}
#makeCoffe() {
return 'Here is a coffe for you 👦';
}
get testScore() {
return this._testScore;
}
set testScore(score) {
this._testScore = score ⩽ 20 ? score : 0;
}
static printCurriculum() {
console.log(`There are ${this.numSubjects} subjects`);
}
}
const student = new Student('Jonas', 2020, 2037, 'Medicine');
Parent class
Inheritance between classes, automatically sets prototype
Child class
Constructor method, called by new operator. Mandatory in regular class, might be omitted in a child class
👉 Classes are just "syntactic sugar" over constructor functions
👉 Classes are not hoisted
👉 Classes are first-class citizens
👉 Class body is always executed in strict mode