OBJECT ORIENTED PROGRAMMING (OOP) WITH JAVASCRIPT

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')

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

new User('mary')

Instance

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

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

new User('steven')

Instance

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

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

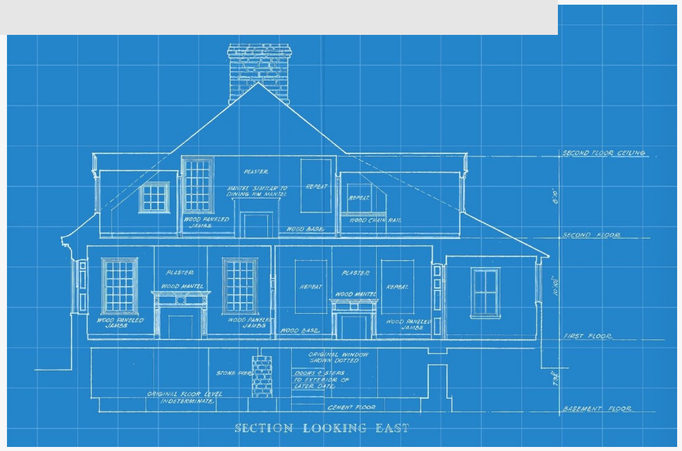
☝ Conceptual overview: it works a bit **differently** in JavaScript. Still important to understand!

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)

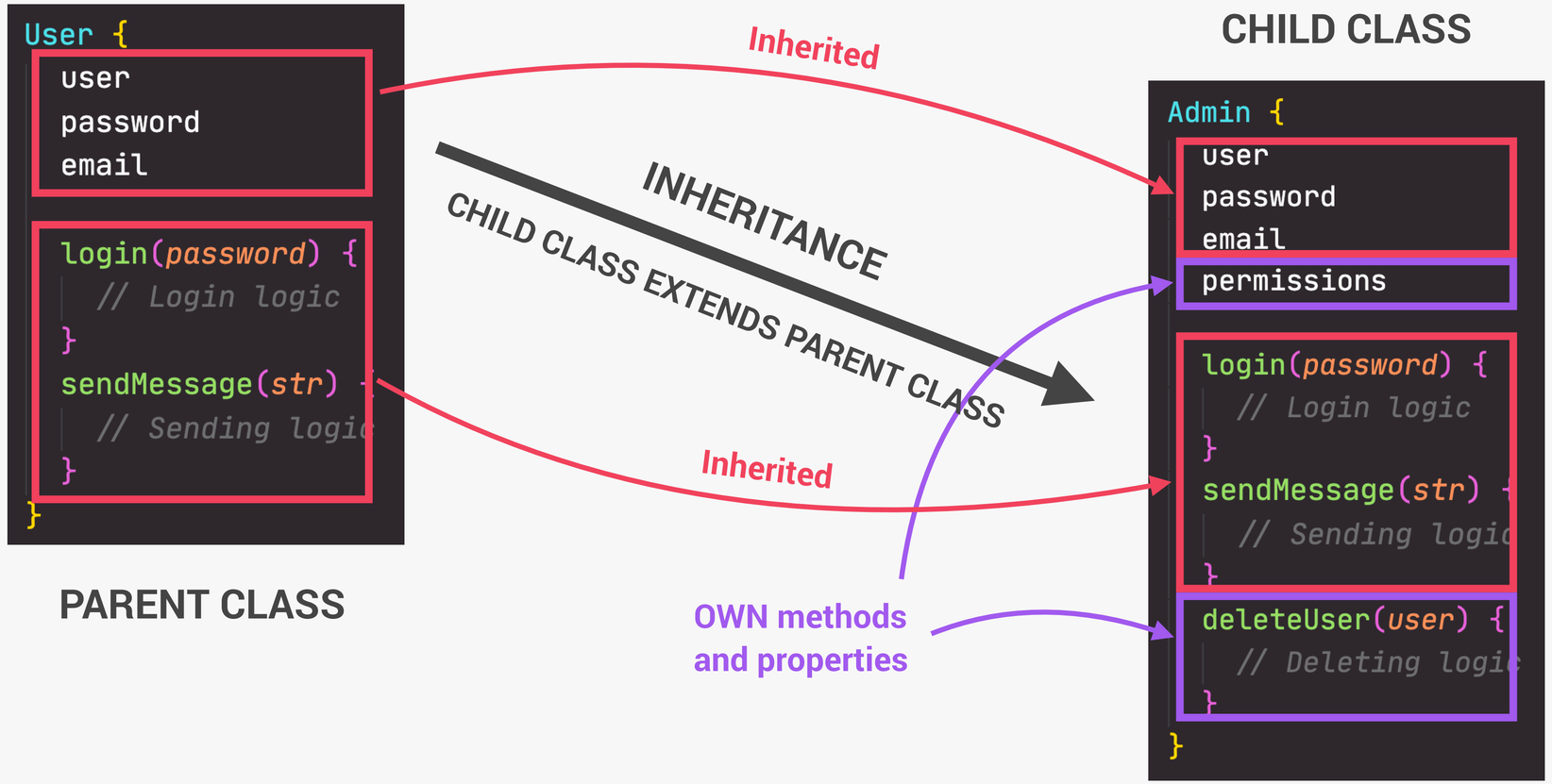
WHY?

- 👉 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

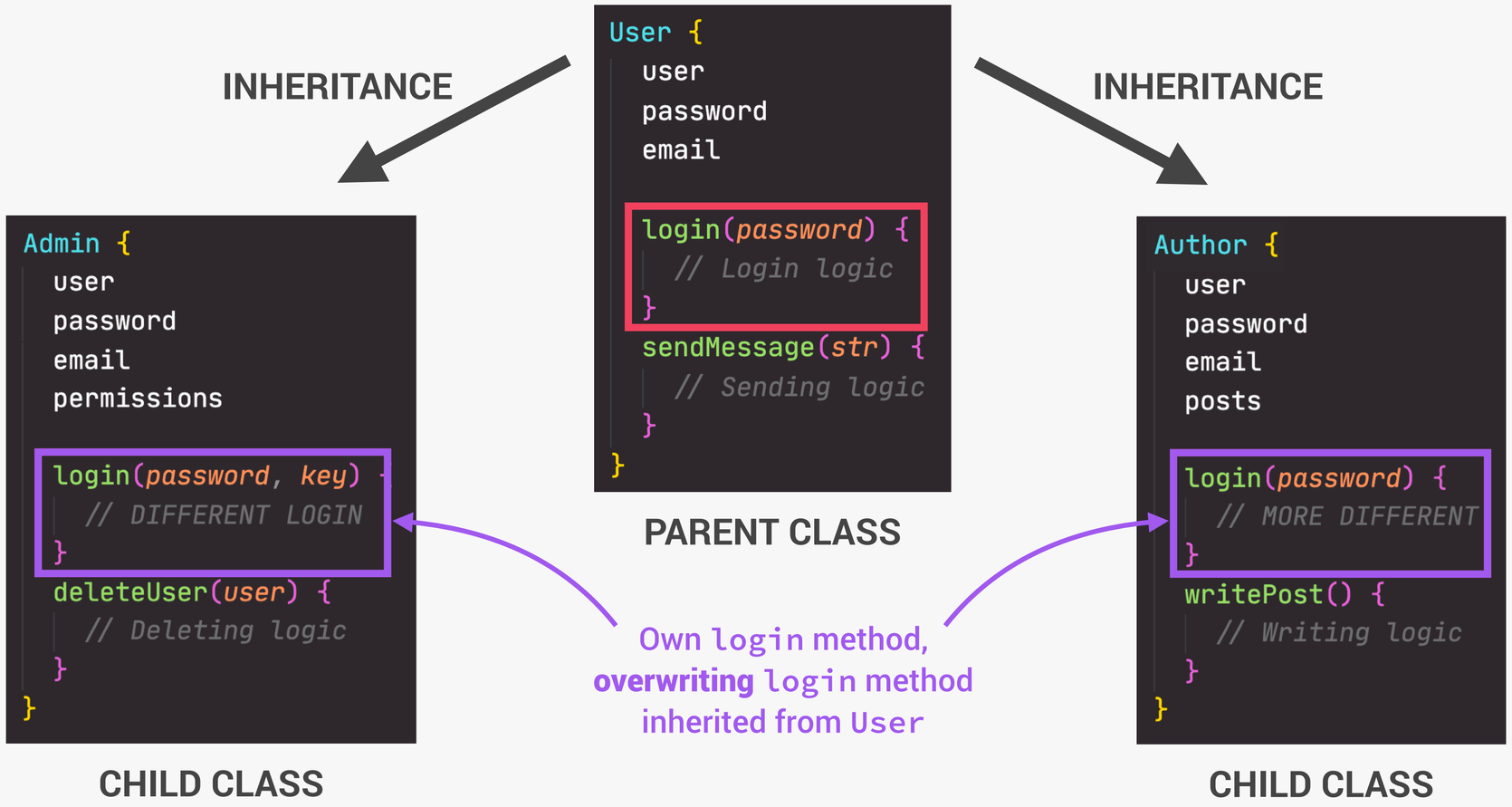
- Abstraction
- Encapsulation
- Inheritance
- Polymorphism



👉 **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].