THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

SECTION

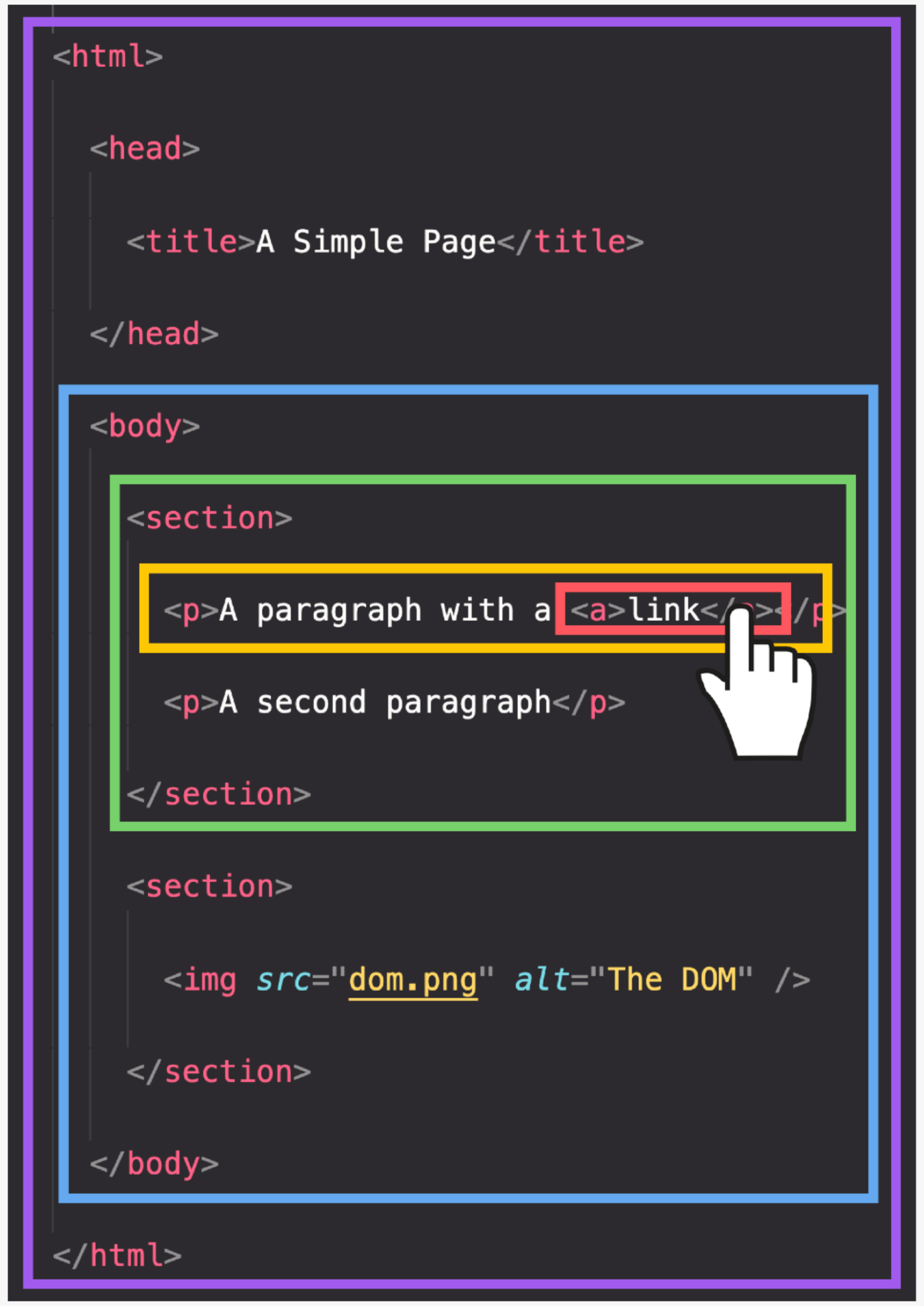
ADVANCED DOM AND EVENTS

LECTURE

EVENT PROPAGATION: BUBBLING AND CAPTURING

JS

BUBBLING AND CAPTURING



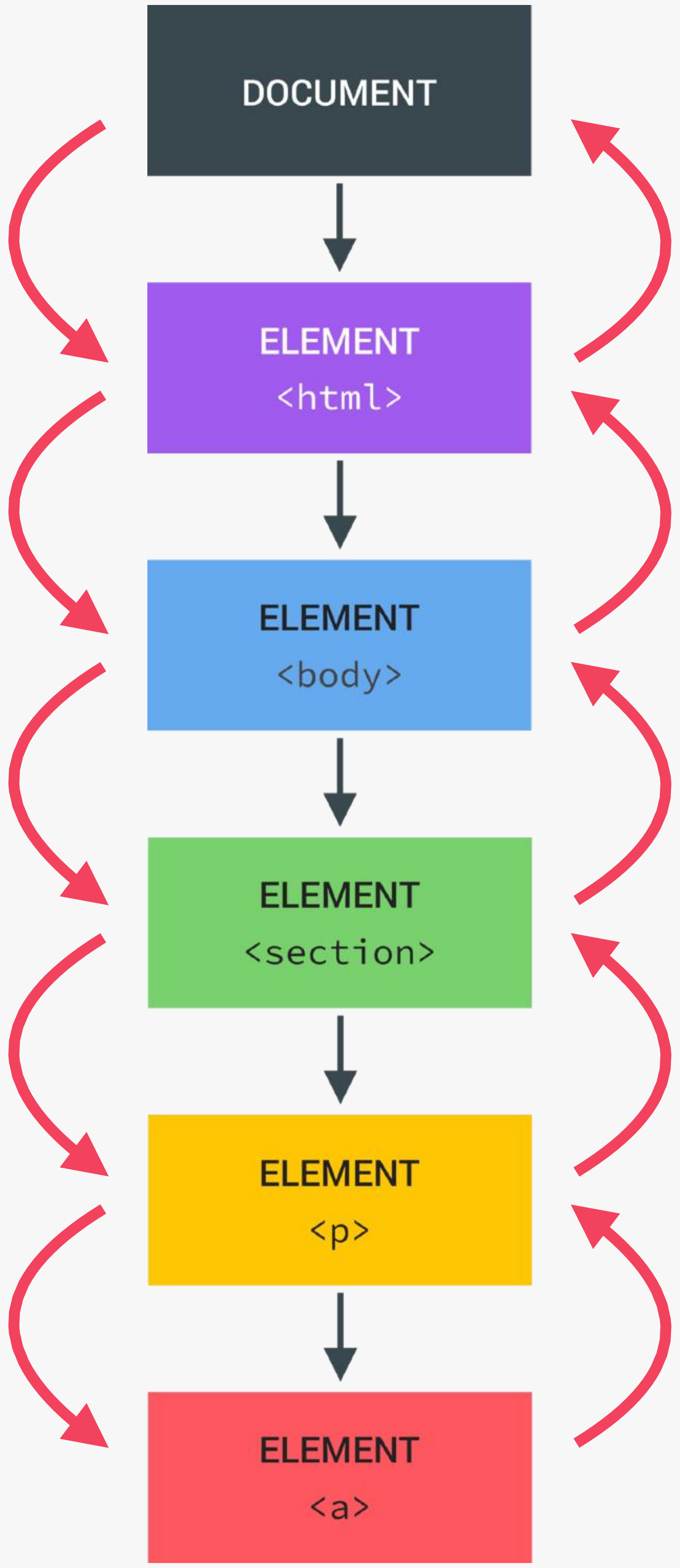
```
<html>

  <head>

    <title>A Simple Page</title>

  </head>

  <body>

    <section>

      <p>A paragraph with a <a>link</a></p>

      <p>A second paragraph</p>

    </section>

    <section>

      <img src="dom.png" alt="The DOM" />

    </section>

  </body>

</html>
```

(THIS DOES NOT HAPPEN
ON *ALL* EVENTS)

Click event

1

CAPTURING
PHASE



DOCUMENT

ELEMENT
`<html>`

ELEMENT
`<body>`

ELEMENT
`<section>`

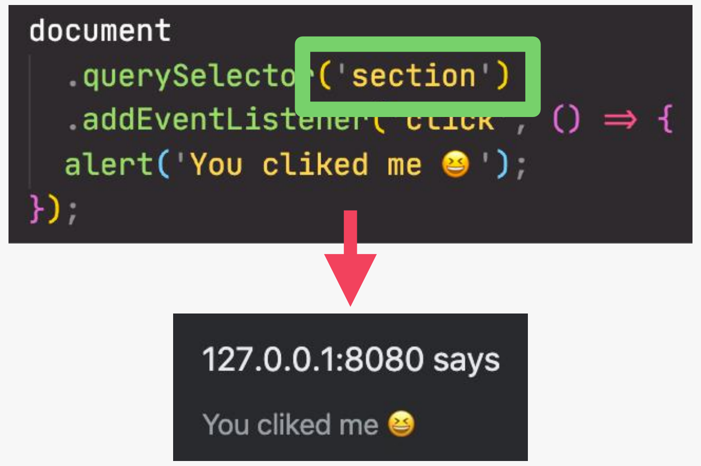
ELEMENT
`<p>`

ELEMENT
`<a>`

2 TARGET PHASE

3

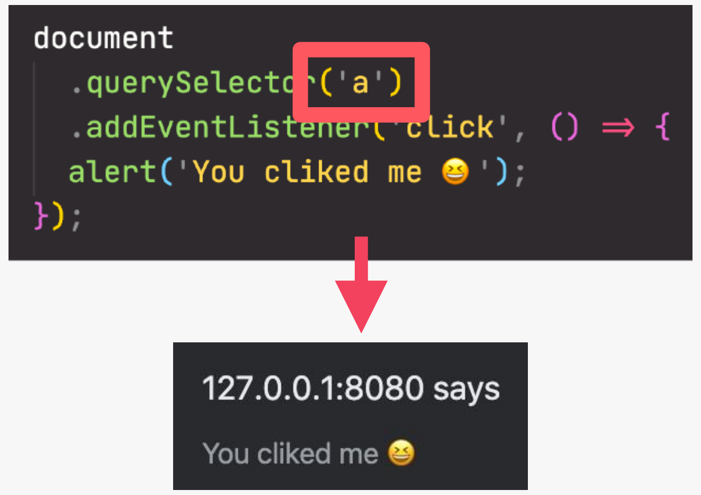
BUBBLING
PHASE



```
document
  .querySelector('section')
  .addEventListener('click', () => {
  alert('You cliked me 😆 ');
});
```

127.0.0.1:8080 says

You cliked me 😆



```
document
  .querySelector('a')
  .addEventListener('click', () => {
  alert('You cliked me 😆 ');
});
```

127.0.0.1:8080 says

You cliked me 😆

THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

SECTION

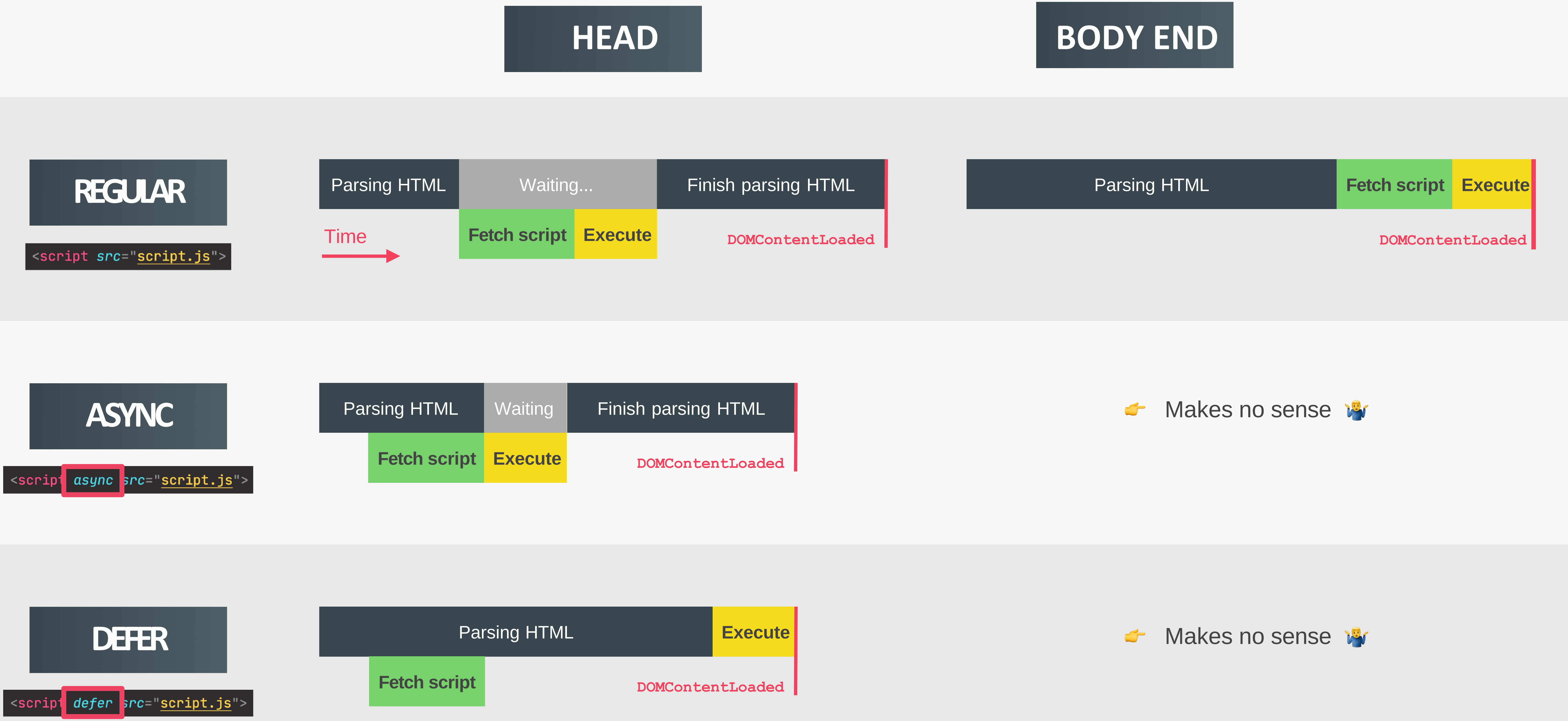
ADVANCED DOM AND EVENTS

LECTURE

EFFICIENT SCRIPT LOADING: DEFER AND ASYNC

JS

DEFER AND ASYNC SCRIPT LOADING



HEAD

BODY END

REGULAR

```
<script src="script.js">
```

Parsing HTML | Waiting... | Finish parsing HTML

Fetch script | Execute

Time →

DOMContentLoaded

Parsing HTML | Fetch script | Execute

DOMContentLoaded

ASYNC

```
<script async src="script.js">
```

Parsing HTML | Waiting | Finish parsing HTML

Fetch script | Execute

DOMContentLoaded

👉 Makes no sense 🤷

DEFER

```
<script defer src="script.js">
```

Parsing HTML | Execute

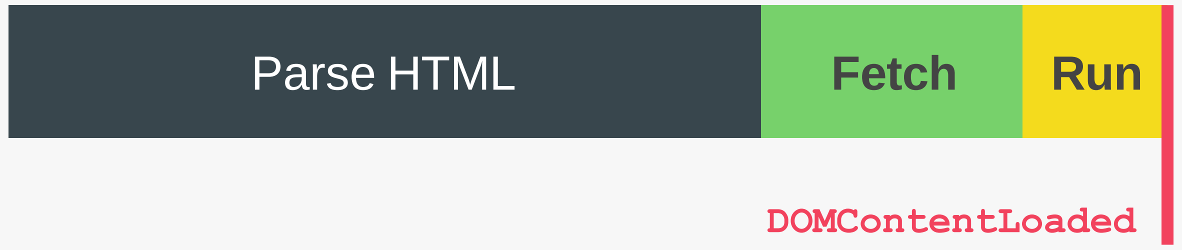
Fetch script

DOMContentLoaded

👉 Makes no sense 🤷

REGULAR VS. ASYNC VS. DEFER

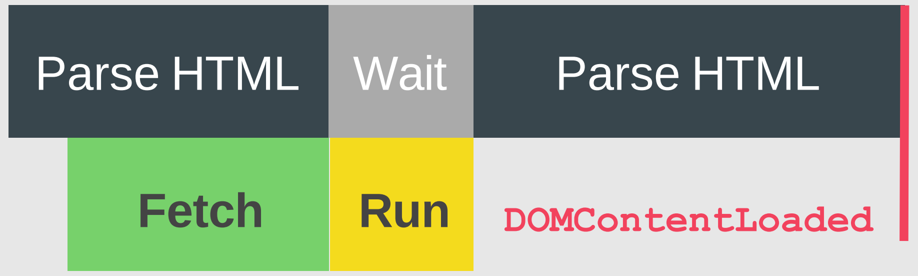
END OF BODY


Parse HTML
Fetch
Run
DOMContentLoaded

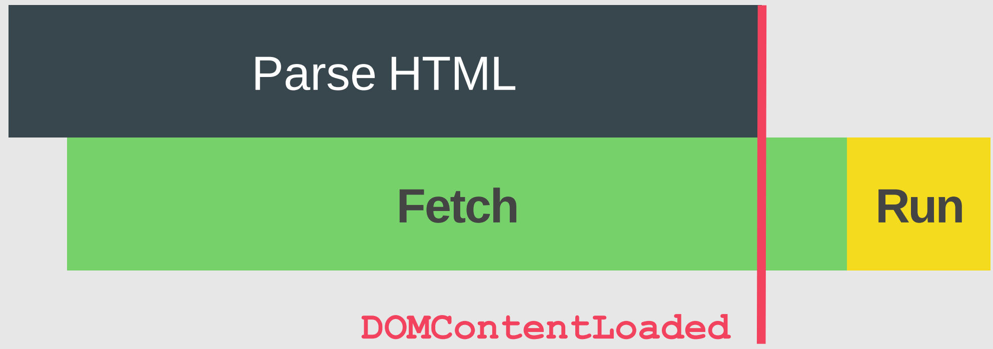
- Scripts are fetched and executed ***after the HTML is completely parsed***
- **Use if you need to support old browsers**

You can, of course, use **different strategies for different scripts**. Usually a complete web applications includes more than just one script

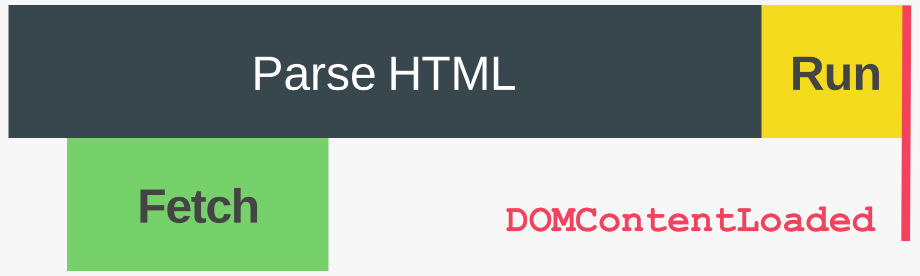
ASYNC IN HEAD


Parse HTML
Wait
Parse HTML
Fetch
Run
DOMContentLoaded

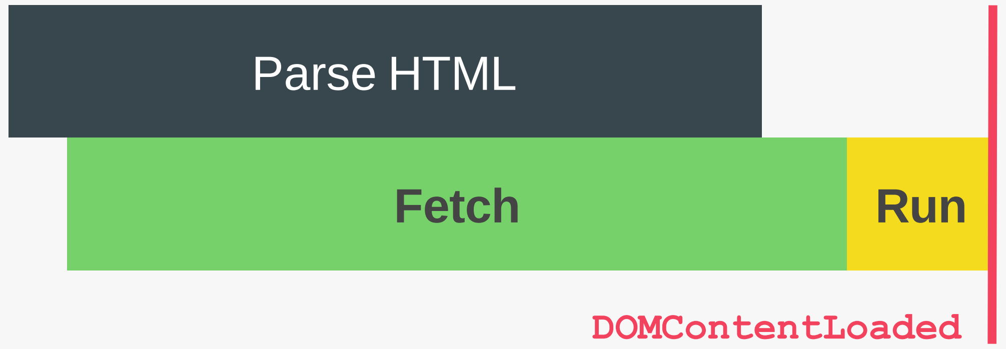
- Scripts are fetched ***asynchronously*** and executed ***immediately***
- Usually the `DOMContentLoaded` event waits for ***all*** scripts to execute, except for `async` scripts. So, `DOMContentLoaded` does ***not*** wait for an `async` script
- Scripts ***not*** guaranteed to execute in order
- **Use for 3rd-party scripts where order doesn't matter (e.g. Google Analytics)**


Parse HTML
Fetch
Run
DOMContentLoaded

DEFER IN HEAD


Parse HTML
Run
Fetch
DOMContentLoaded

- Scripts are fetched ***asynchronously*** and executed ***after the HTML is completely parsed***
- `DOMContentLoaded` event fires ***after*** `defer` script is executed
- Scripts are executed ***in order***
- **This is overall the best solution! Use for your own scripts, and when order matters (e.g. including a library)**


Parse HTML
Fetch
Run
DOMContentLoaded

OBJECT ORIENTED PROGRAMMING (OOP) WITH JAVASCRIPT

THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

SECTION

OBJECT ORIENTED PROGRAMMING (OOP) WITH JAVASCRIPT

LECTURE

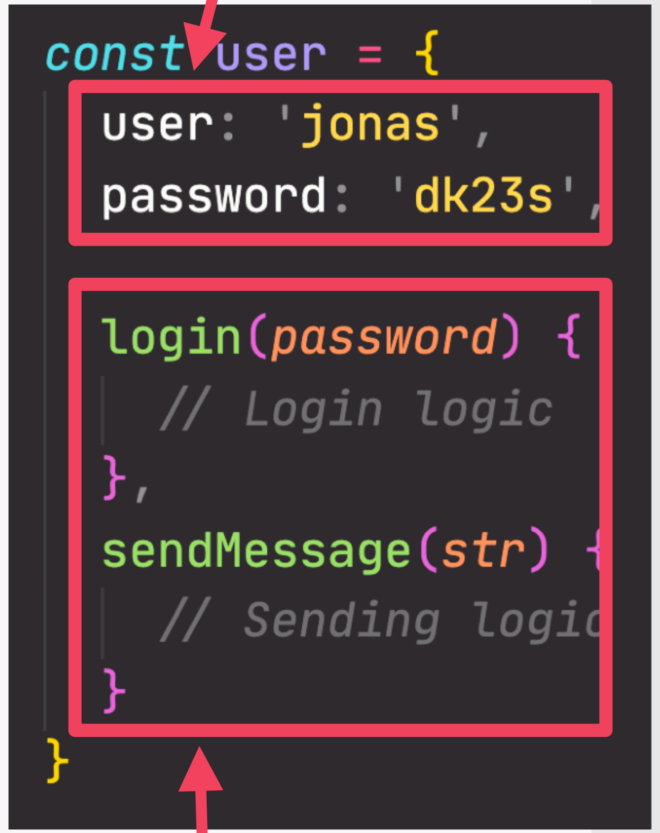
WHAT IS OBJECT-ORIENTED PROGRAMMING?

WHAT IS OBJECT-ORIENTED PROGRAMMING? (OOP)

OOP

Style of code, “how” we write and organize code

Data



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

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

Behaviour

- 👉 Object-oriented programming (OOP) is a programming paradigm based on the concept of objects;

E.g. user or todo list item

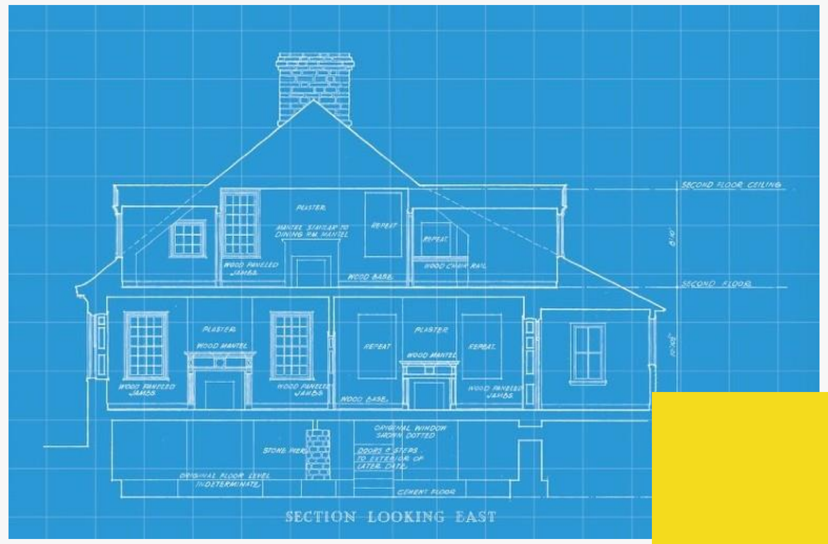
- 👉 We use objects to **model** (describe) real-world or abstract features;

E.g. HTML component or data structure

- 👉 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”).

CLASSES AND INSTANCES (TRADITIONAL OOP)

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



Like a blueprint from which we can create **new objects**

CLASS

```
User {
  user
  password
  email

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

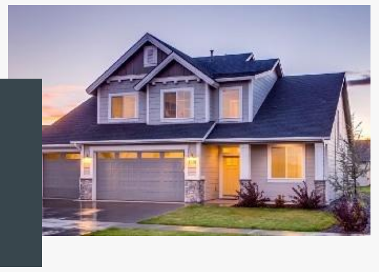
Just a representation, **NOT** actual JavaScript syntax!

JavaScript does **NOT** support *real* classes like represented here

`new User('jonas')`

`new User('mary')`

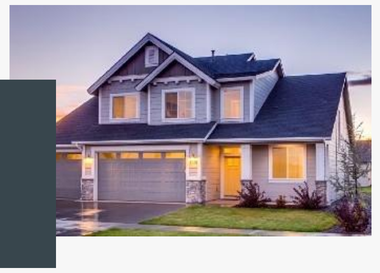
`new User('steven')`



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

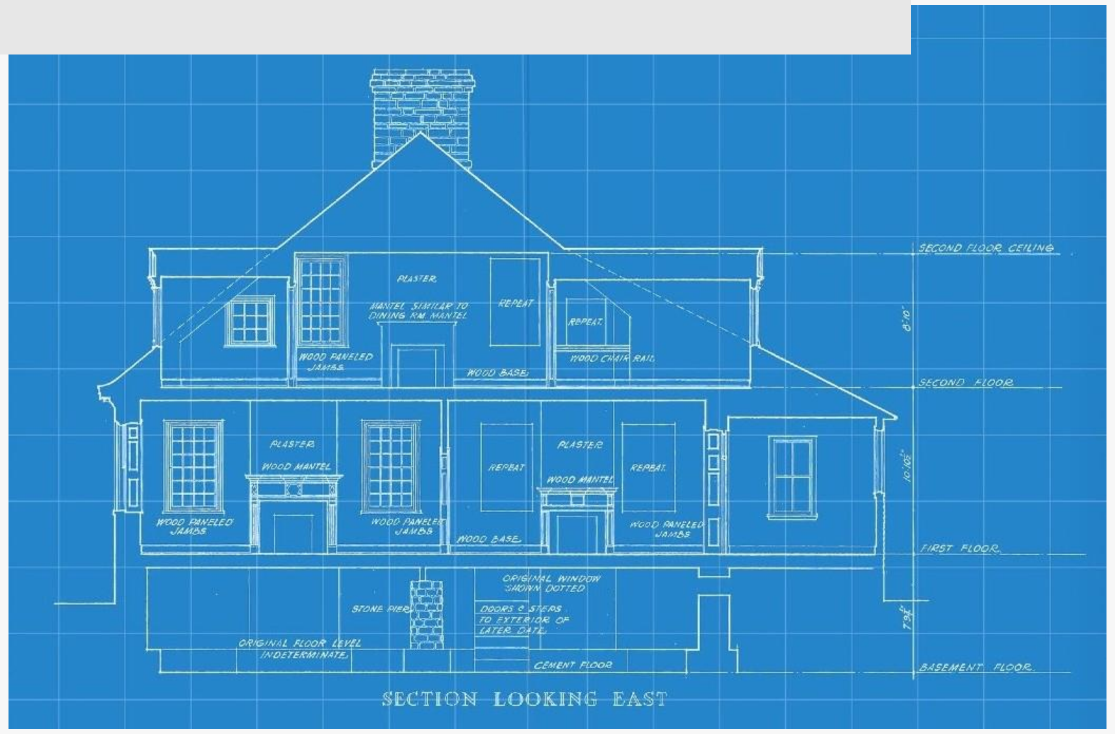
```
{
  user = 'steven'
  password = '5p8dz32dd'
  email = 'steven@tes.co'

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

THE 4 FUNDAMENTALOOP PRINCIPLES

- Abstraction
- Encapsulation
- Inheritance
- Polymorphism

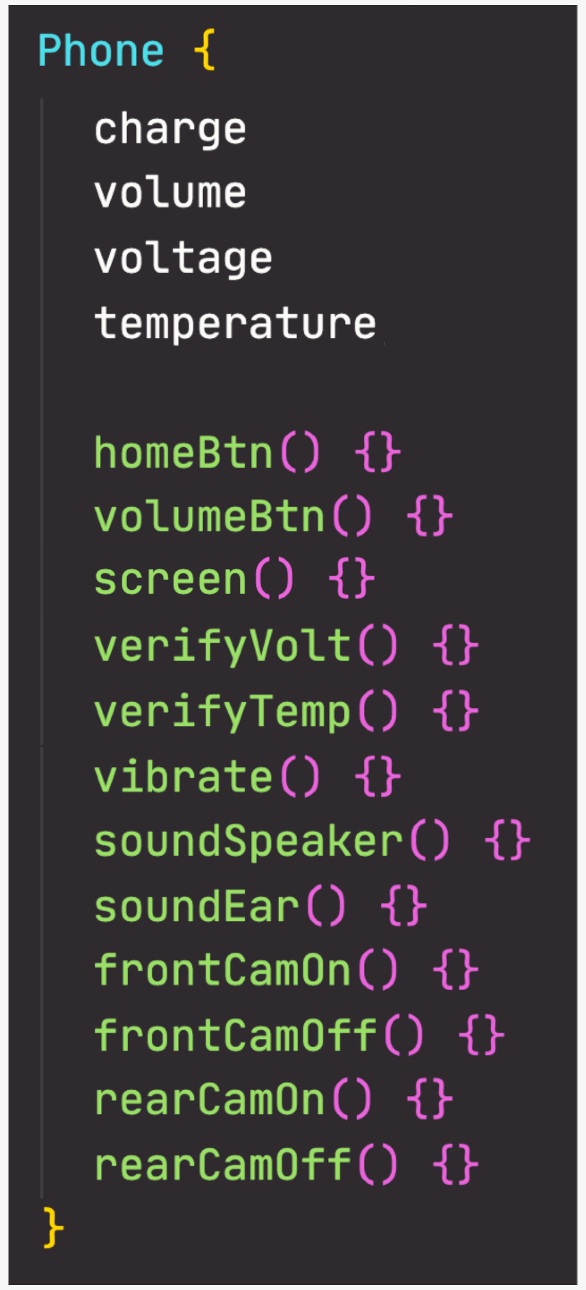
The 4 fundamental principles of Object-Oriented Programming

> 🤔 *“How do we actually design classes? How do we model real-world data into classes?”*



PRINCIPLE 1: ABSTRACTION

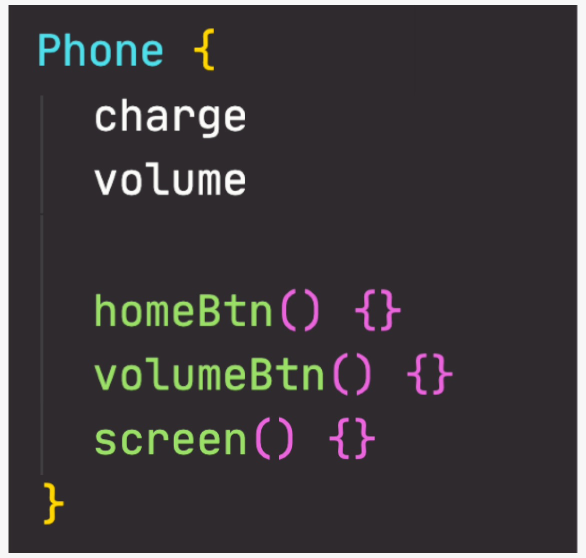
- **Abstraction**
- Encapsulation
- Inheritance
- Polymorphism



```
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

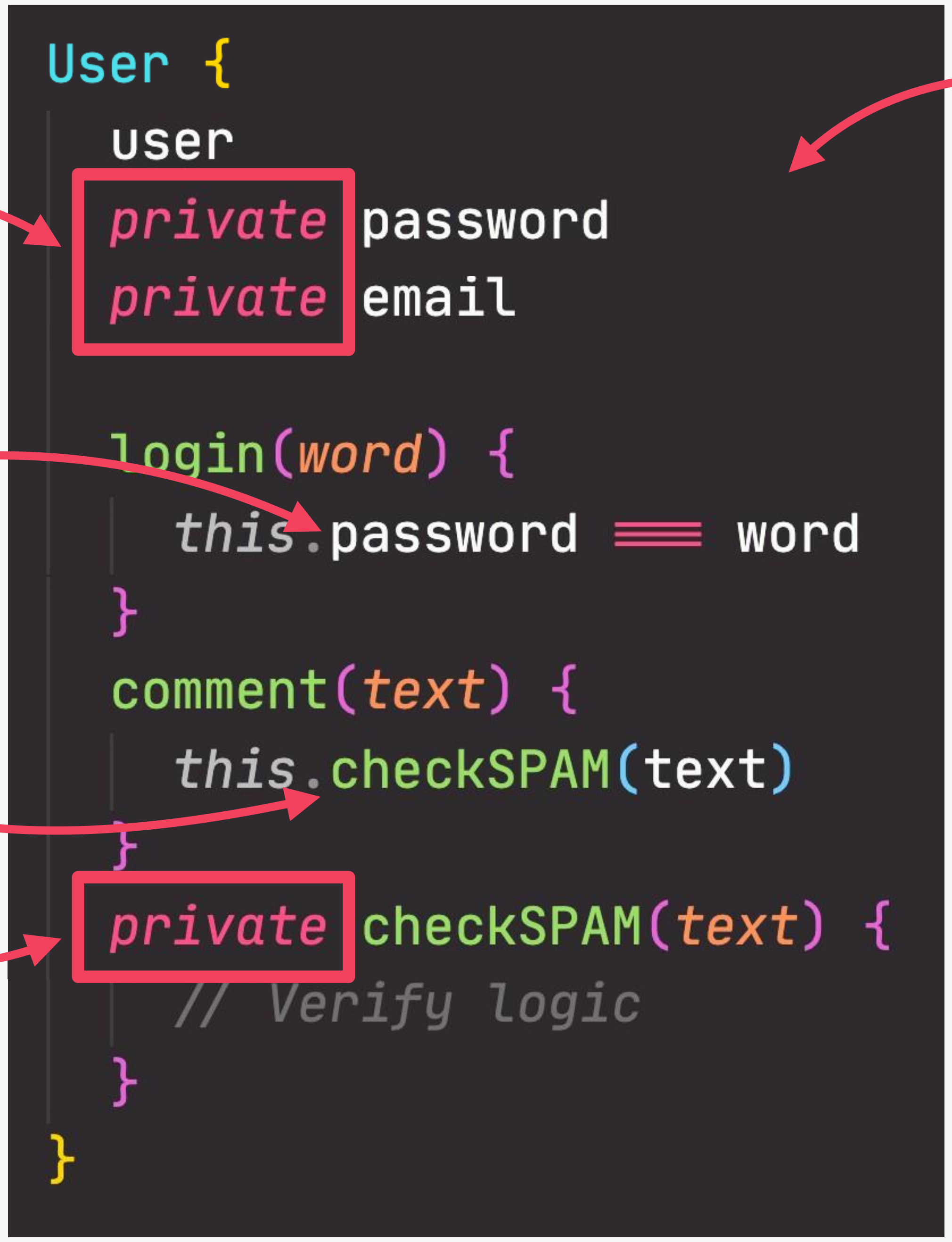
Polymorphism

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)

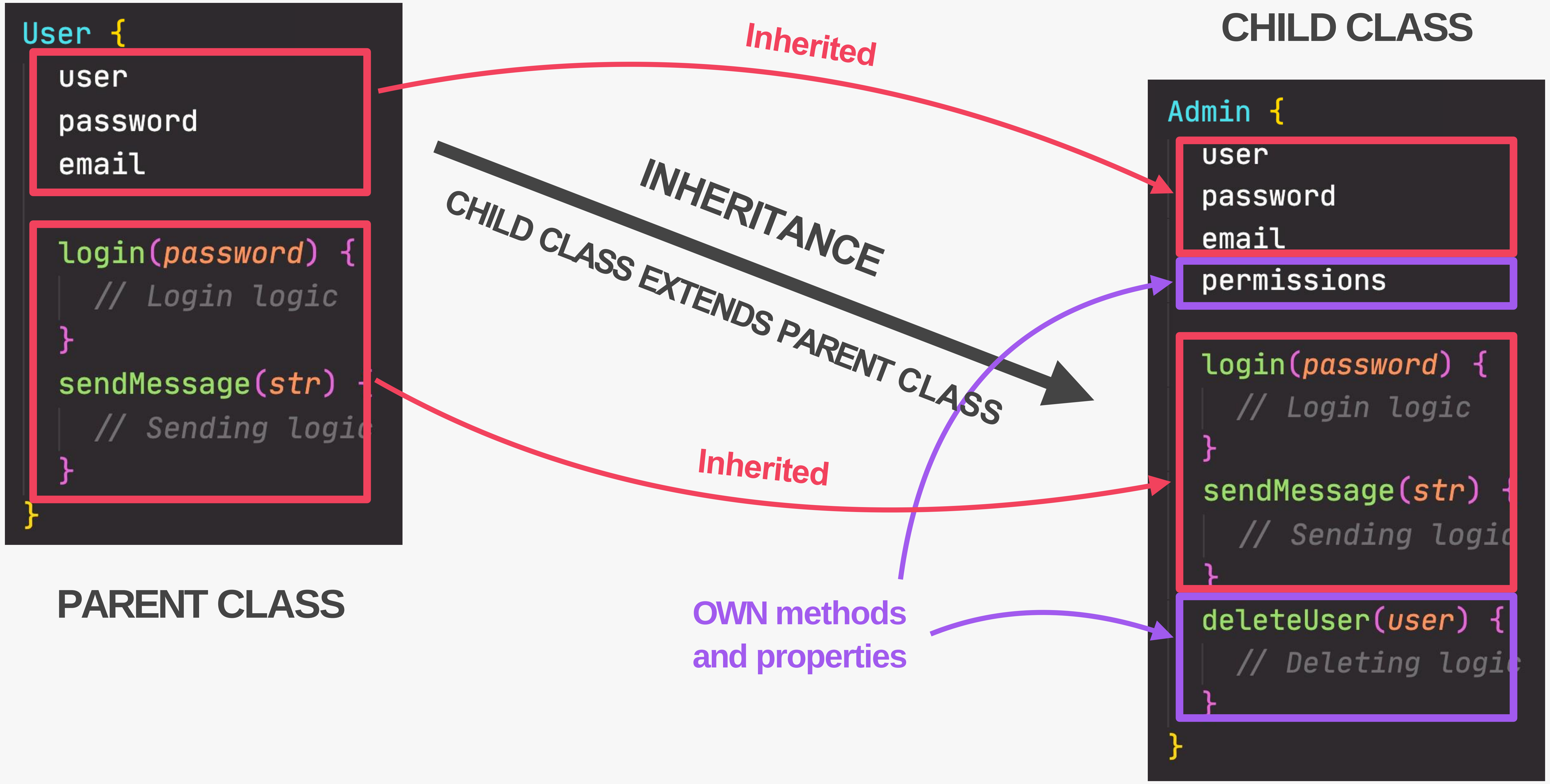
WHY?

- 👉 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



```
User {
  user
  password
  email

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

PARENT CLASS

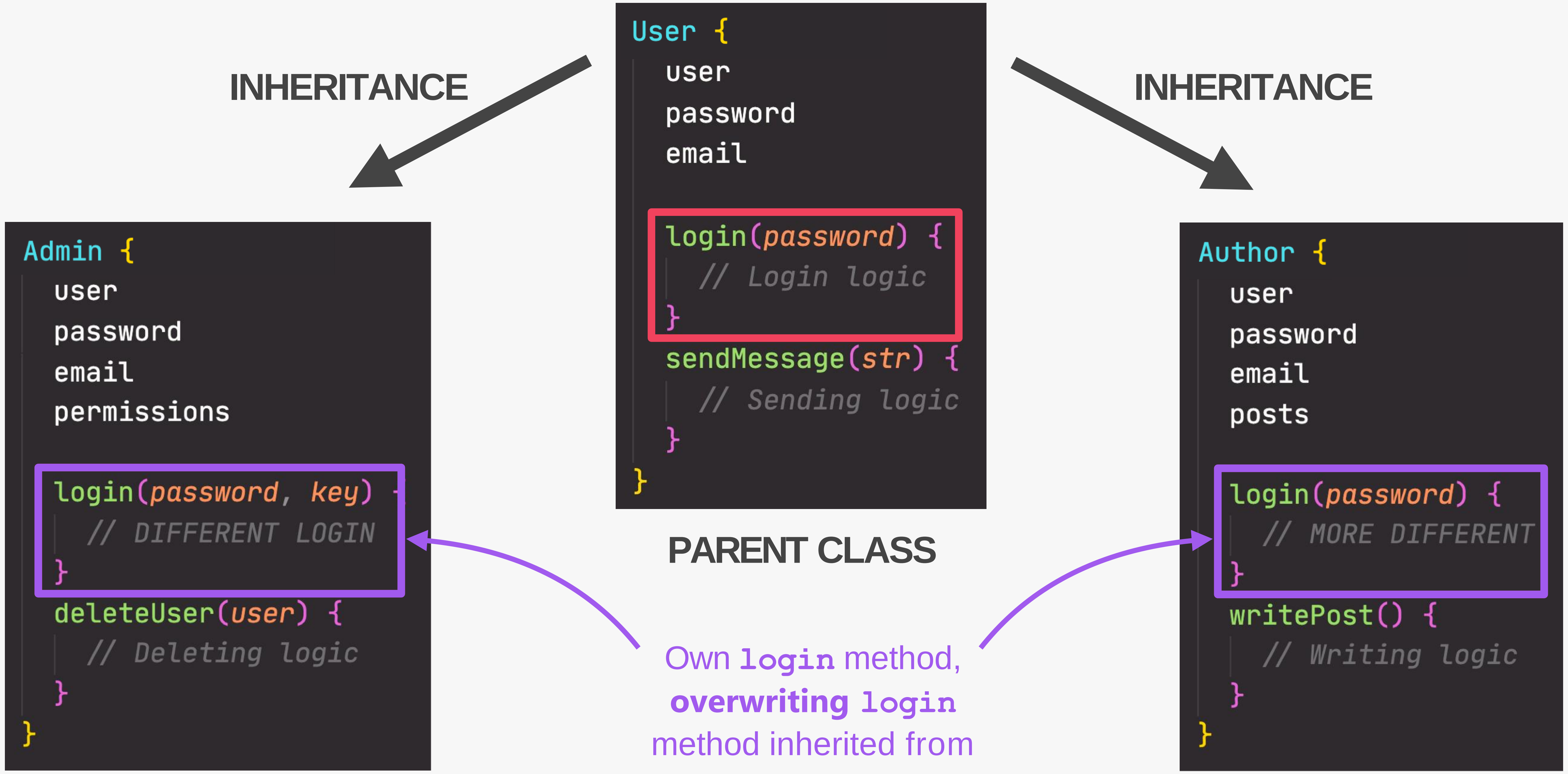
```
Admin {
  user
  password
  email
  permissions

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

CHILD CLASS

👉 **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



INHERITANCE

INHERITANCE

```
User {
  user
  password
  email

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

PARENT CLASS

```
Admin {
  user
  password
  email
  permissions

  login(password, key) {
    // DIFFERENT LOGIN
  }
  deleteUser(user) {
    // Deleting logic
  }
}
```

CHILD CLASS

```
Author {
  user
  password
  email
  posts

  login(password) {
    // MORE DIFFERENT
  }
  writePost() {
    // Writing logic
  }
}
```

CHILD CLASS

Own `login` method, **overwriting `login`** method inherited from **`User`**

👉 **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].