# WHAT IS OBJECT-ORIENTED PROGRAMMING? (OOP)

OOP

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

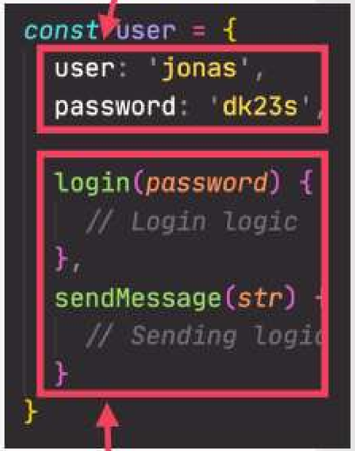
- Object-oriented programming (OOP) is a programming paradigm based on the concept of objects;

E.g. user or todo list item

E.g. HTML component or data structure

- 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”).

Data



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

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

Behaviour

# 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

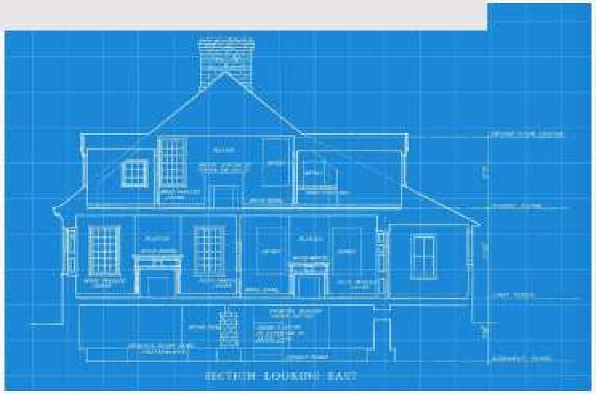
```
{
  user = 'steven'
  password = '5p8dz32dd'
  email = 'steven@tes.co'

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

# THE 4 FUNDAMENTAL OOP PRINCIPLES

The 4 fundamental principles of Object-Oriented Programming

- Abstraction
- Encapsulation
- Inheritance
- Polymorphism

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



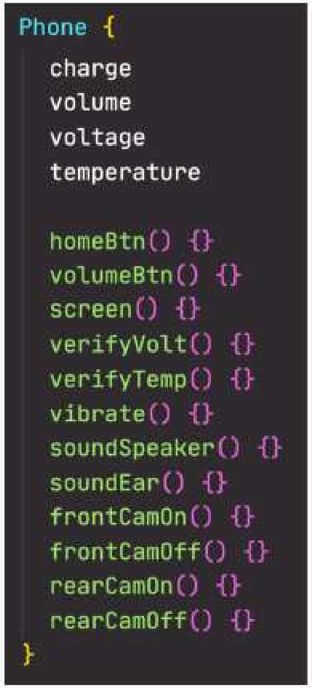
# PRINCIPLE 1: ABSTRACTION

**Abstraction**

Encapsulation

Inheritance

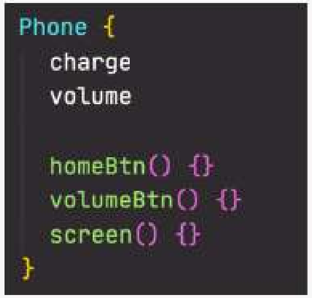
Polymorphism



```
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.

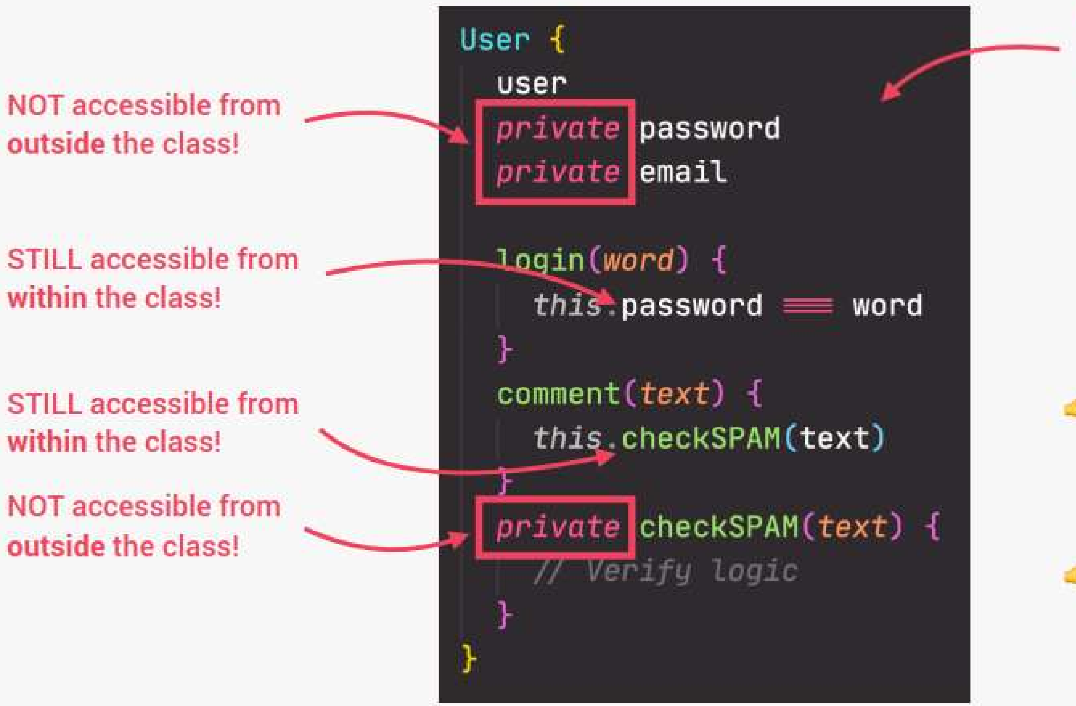
# PRINCIPLE 2: ENCAPSULATION

Abstraction

Encapsulation

Inheritance

Polymorphism



```
User {
  user
  private password
  private email

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

NOT accessible from outside the class!

STILL accessible from within the class!

STILL accessible from within the class!

NOT accessible from outside the class!

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

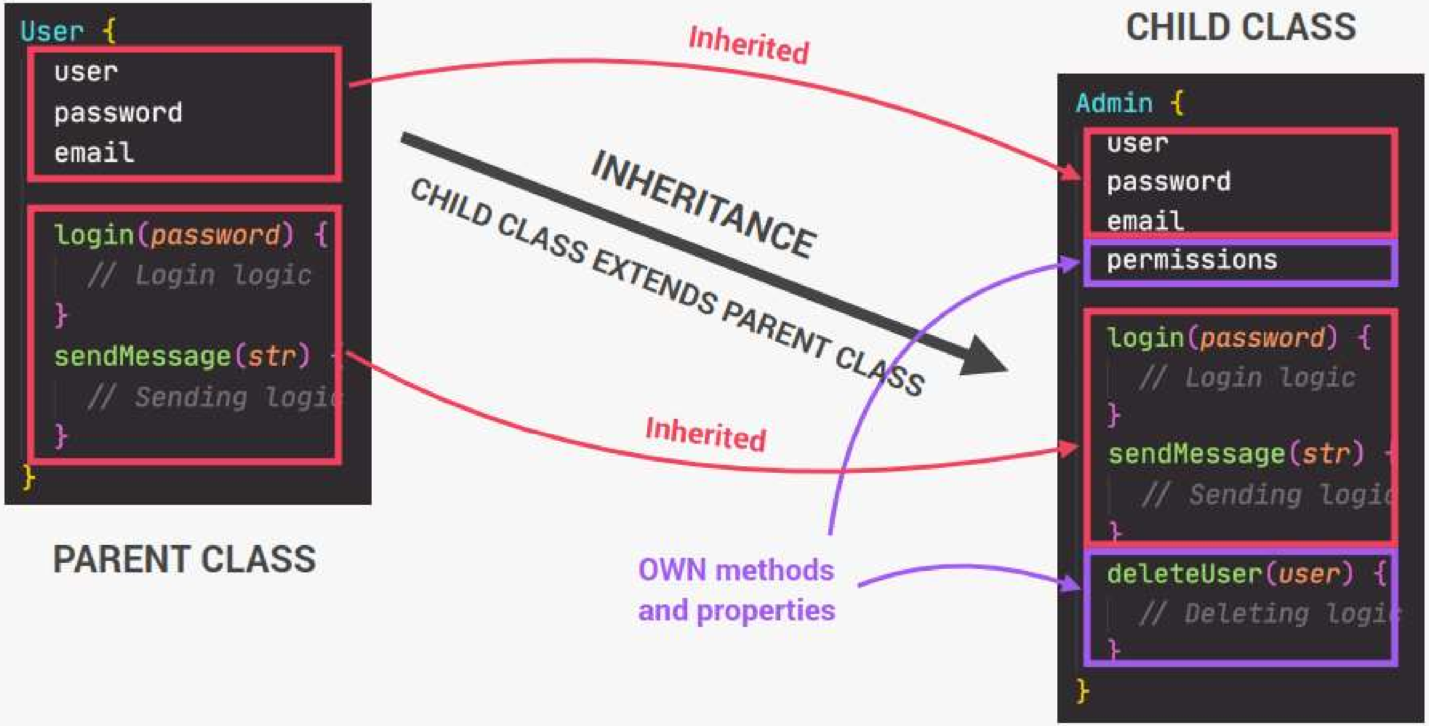
## WHY?

- 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

INHERITANCE

CHILD CLASS EXTENDS PARENT CLASS

Inherited

Inherited

CHILD CLASS

```
Admin {
  user
  password
  email
  permissions

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

OWN methods and properties

👉 **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.

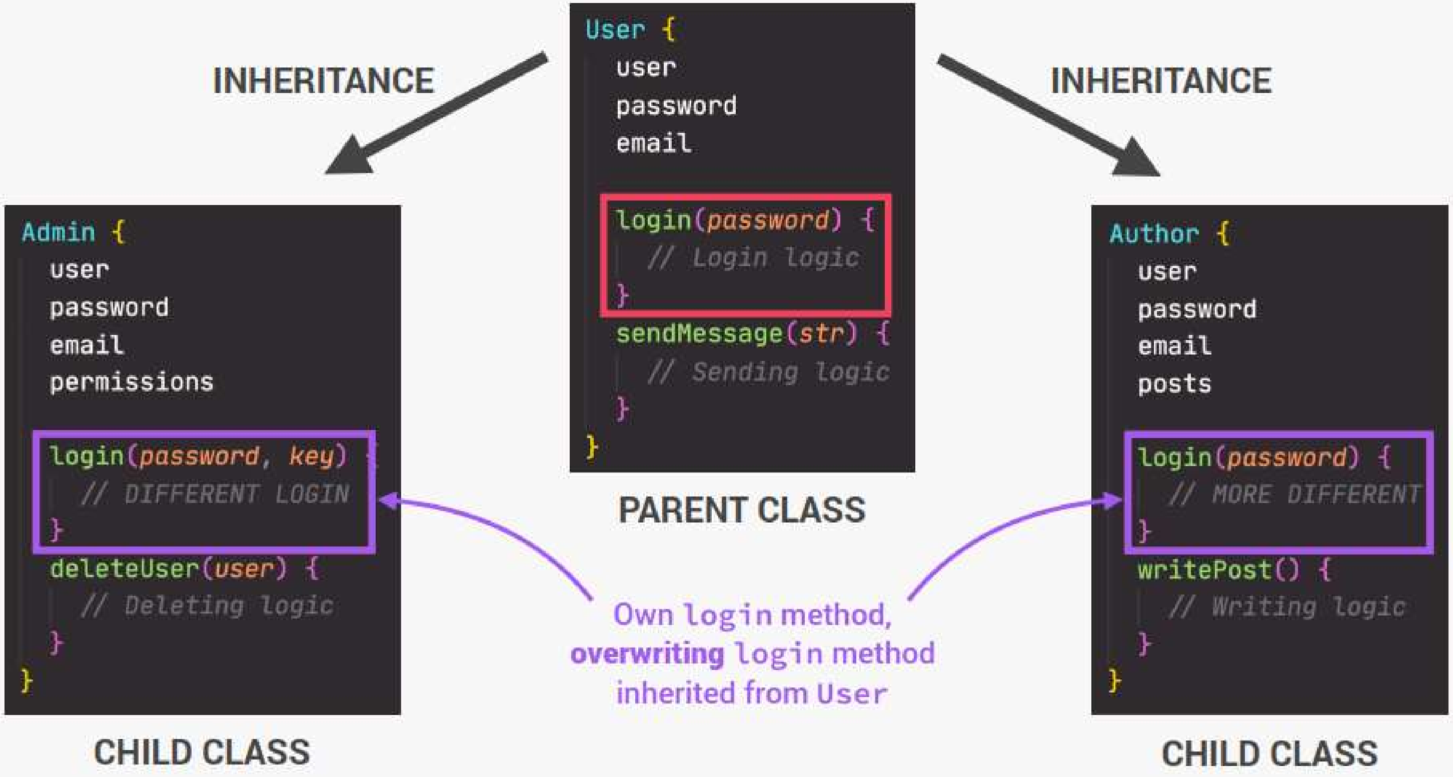
# PRINCIPLE 4: POLYMORPHISM

Abstraction

Encapsulation

Inheritance

Polymorphism



```
User {
  user
  password
  email

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

PARENT CLASS

INHERITANCE

```
Admin {
  user
  password
  email
  permissions

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

CHILD CLASS

INHERITANCE

```
Author {
  user
  password
  email
  posts

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

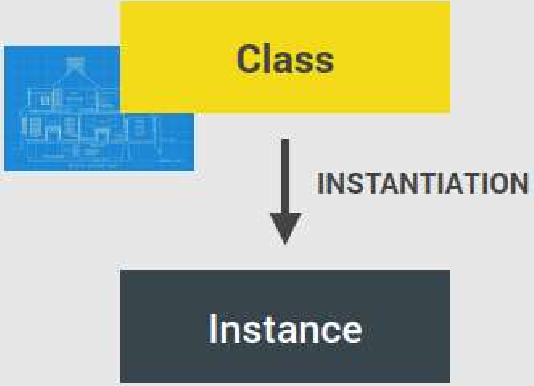
CHILD CLASS

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].

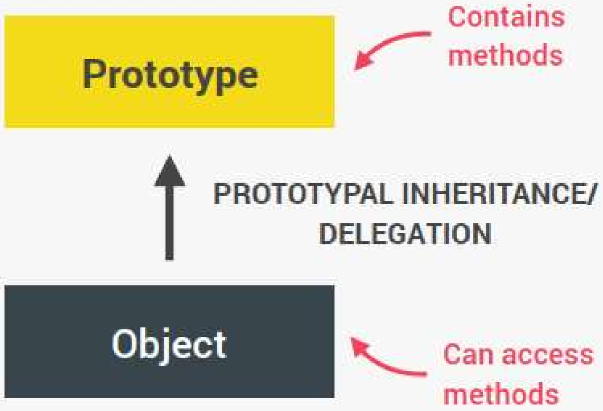
# OOP IN JAVASCRIPT: PROTOTYPES

## "CLASSICAL OOP": CLASSES



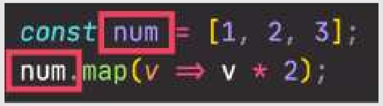
- Objects (instances) are **instantiated** from a class, which functions like a blueprint;
- Behavior (methods) is **copied** from class to all instances.

## OOP IN JS: PROTOTYPES

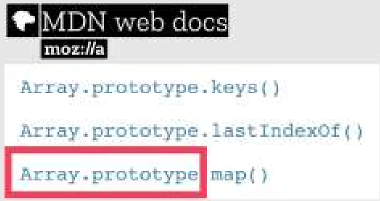


- Objects are **linked** to a prototype object;
- **Prototypal inheritance:** The prototype contains methods (behavior) that are **accessible to all objects linked to that prototype;**
- Behavior is **delegated** to the linked prototype object.

- Example: Array



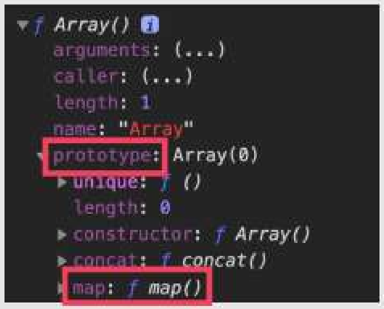
```
const num = [1, 2, 3];
num.map(v => v * 2);
```



```
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?"*

## 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).

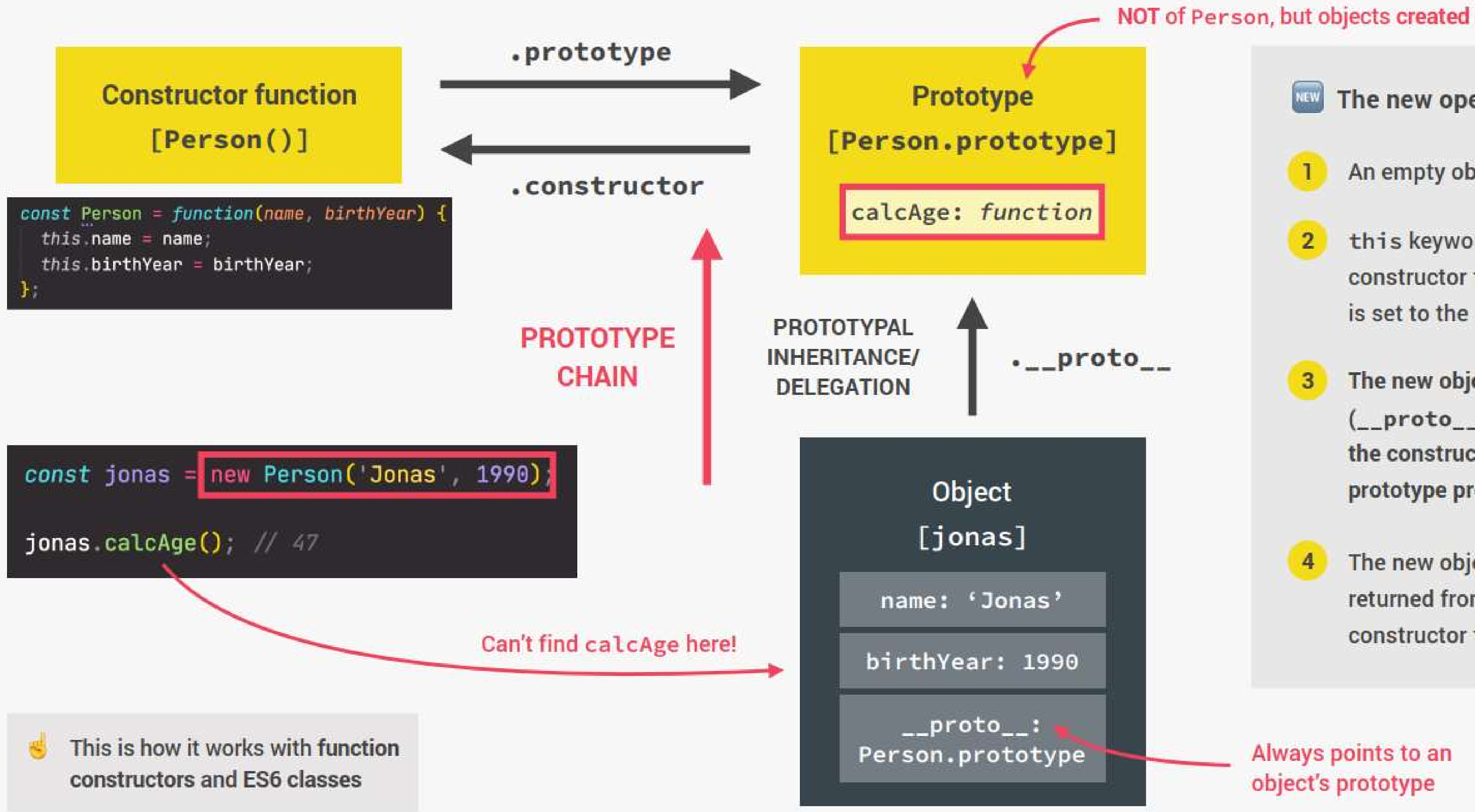
## 3 `Object.create()`

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

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

- Abstraction
- Encapsulation
- Inheritance
- Polymorphism

# HOW PROTOTYPAL INHERITANCE / DELEGATION WORKS



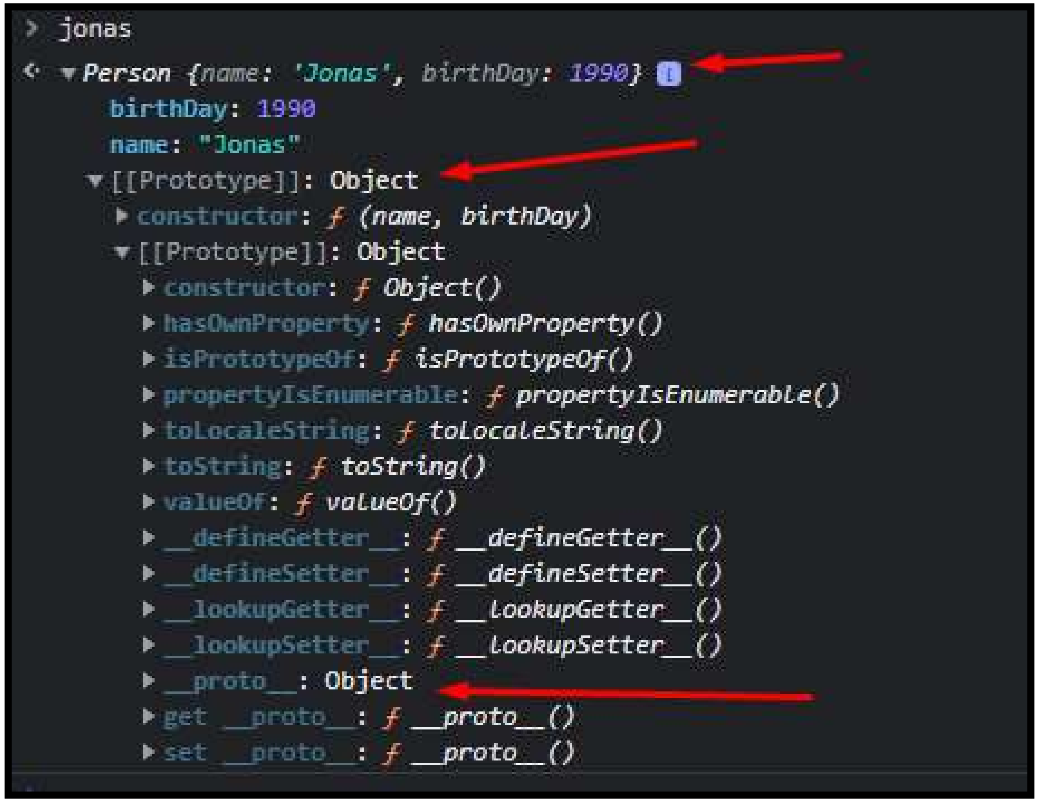
```
const Person = function(name, birthYear) {
  this.name = name;
  this.birthYear = birthYear;
};
```

```
const jonas = new Person('Jonas', 1990);

jonas.calcAge(); // 47
```

☝ This is how it works with function constructors and ES6 classes

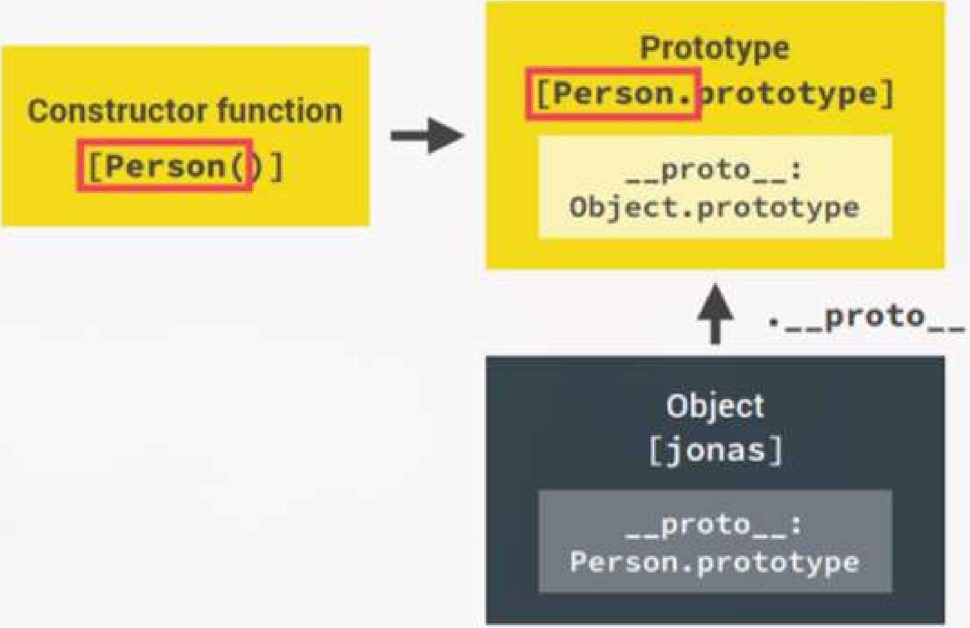
🆕 **The new operator:**

1. An empty object is created
2. `this` keyword in constructor function call is set to the new object
3. The new object is linked (`__proto__` property) to the constructor function's prototype property
4. The new object is returned from the constructor function call



```
> jonas
<· ▼Person {name: 'Jonas', birthDay: 1990}
      birthDay: 1990
      name: "Jonas"
    ▼[[Prototype]]: Object
      ▶constructor: ƒ (name, birthDay)
      ▼[[Prototype]]: Object
        ▶constructor: ƒ Object()
        ▶hasOwnProperty: ƒ hasOwnProperty()
        ▶isPrototypeOf: ƒ isPrototypeOf()
        ▶propertyIsEnumerable: ƒ propertyIsEnumerable()
        ▶toLocaleString: ƒ toLocaleString()
        ▶toString: ƒ toString()
        ▶valueOf: ƒ valueOf()
        ▶__defineGetter__: ƒ __defineGetter__()
        ▶__defineSetter__: ƒ __defineSetter__()
        ▶__lookupGetter__: ƒ __lookupGetter__()
        ▶__lookupSetter__: ƒ __lookupSetter__()
        ▶__proto__: Object
        ▶get __proto__: ƒ __proto__()
        ▶set __proto__: ƒ __proto__()
```

# THE PROTOTYPE CHAIN



Constructor function
[Person()]

Prototype
[Person.prototype]

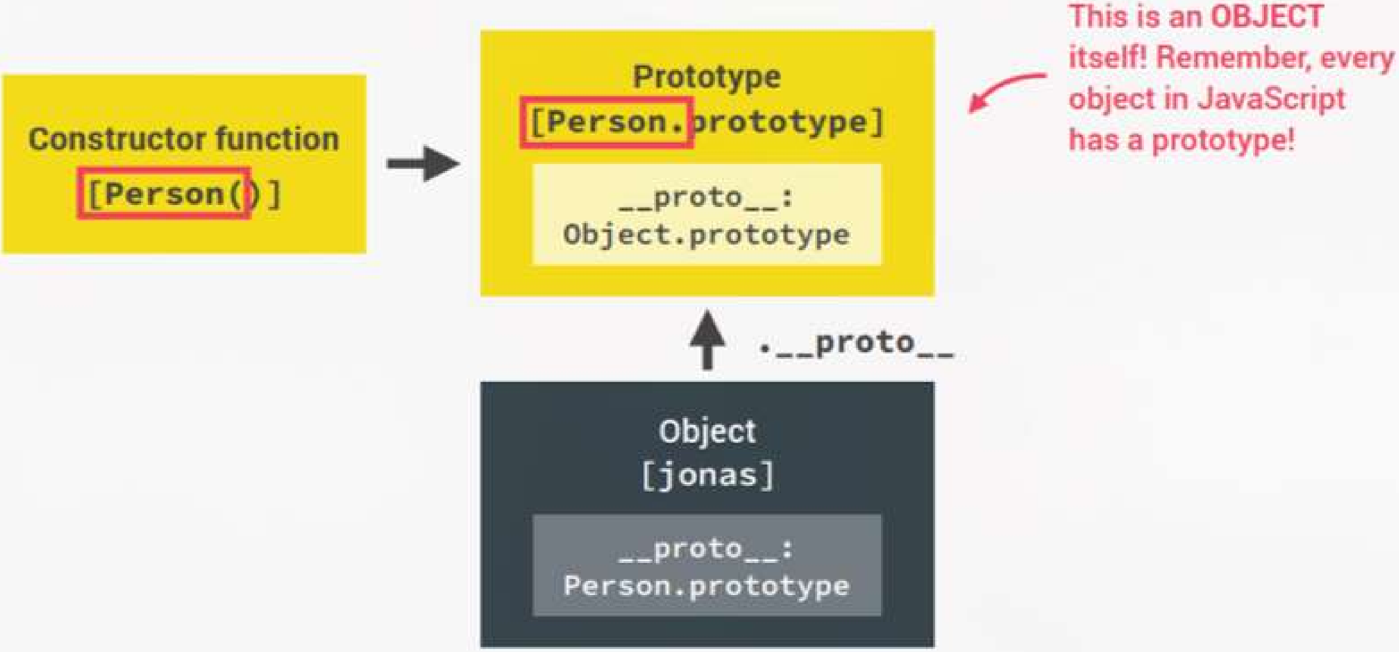
```
__proto__:
Object.prototype
```

.__proto__

Object
[jonas]

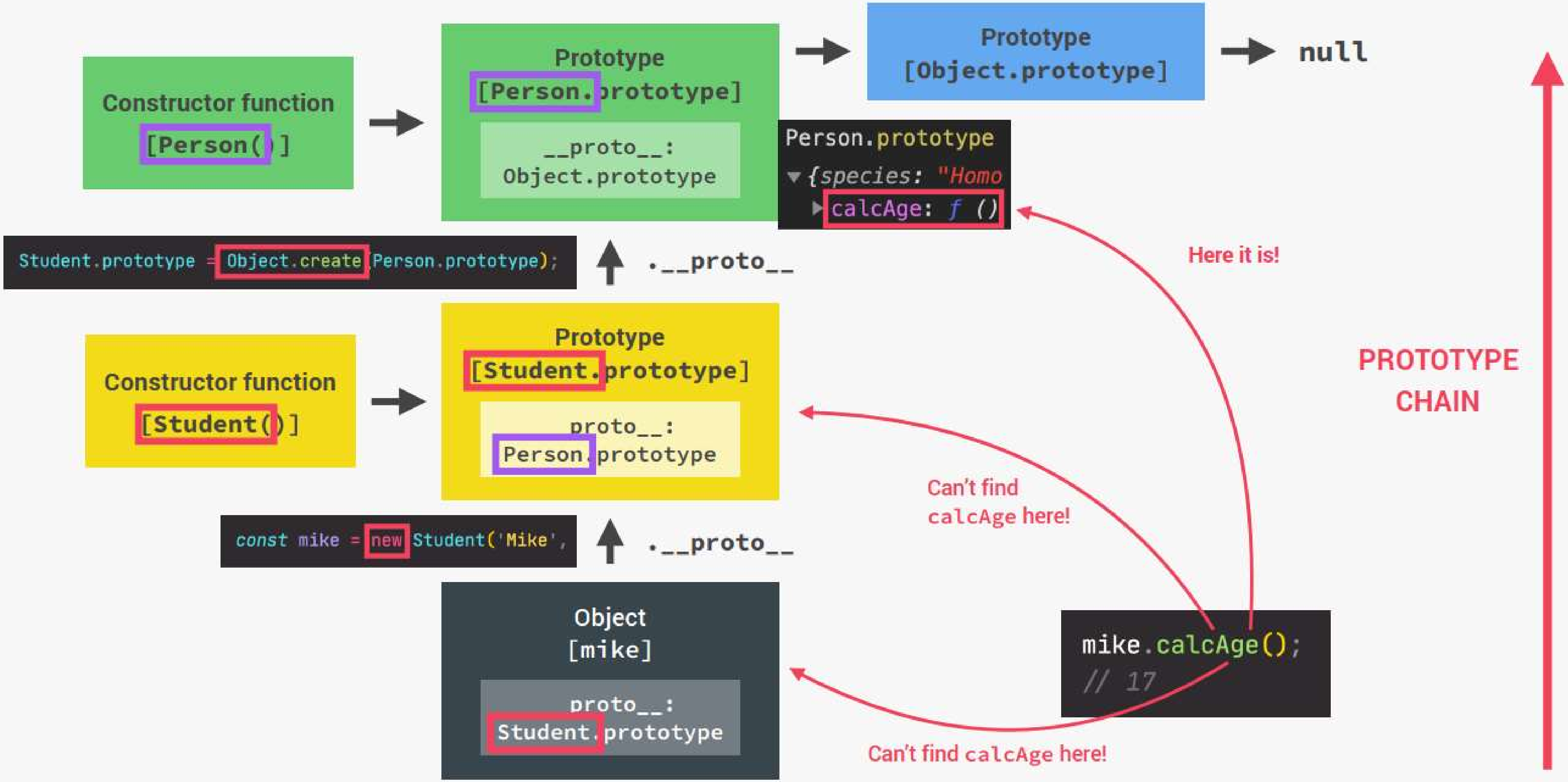
```
__proto__:
Person.prototype
```

# THE PROTOTYPE CHAIN



This is an **OBJECT** itself! Remember, every object in JavaScript has a prototype!

# INHERITANCE BETWEEN "CLASSES"



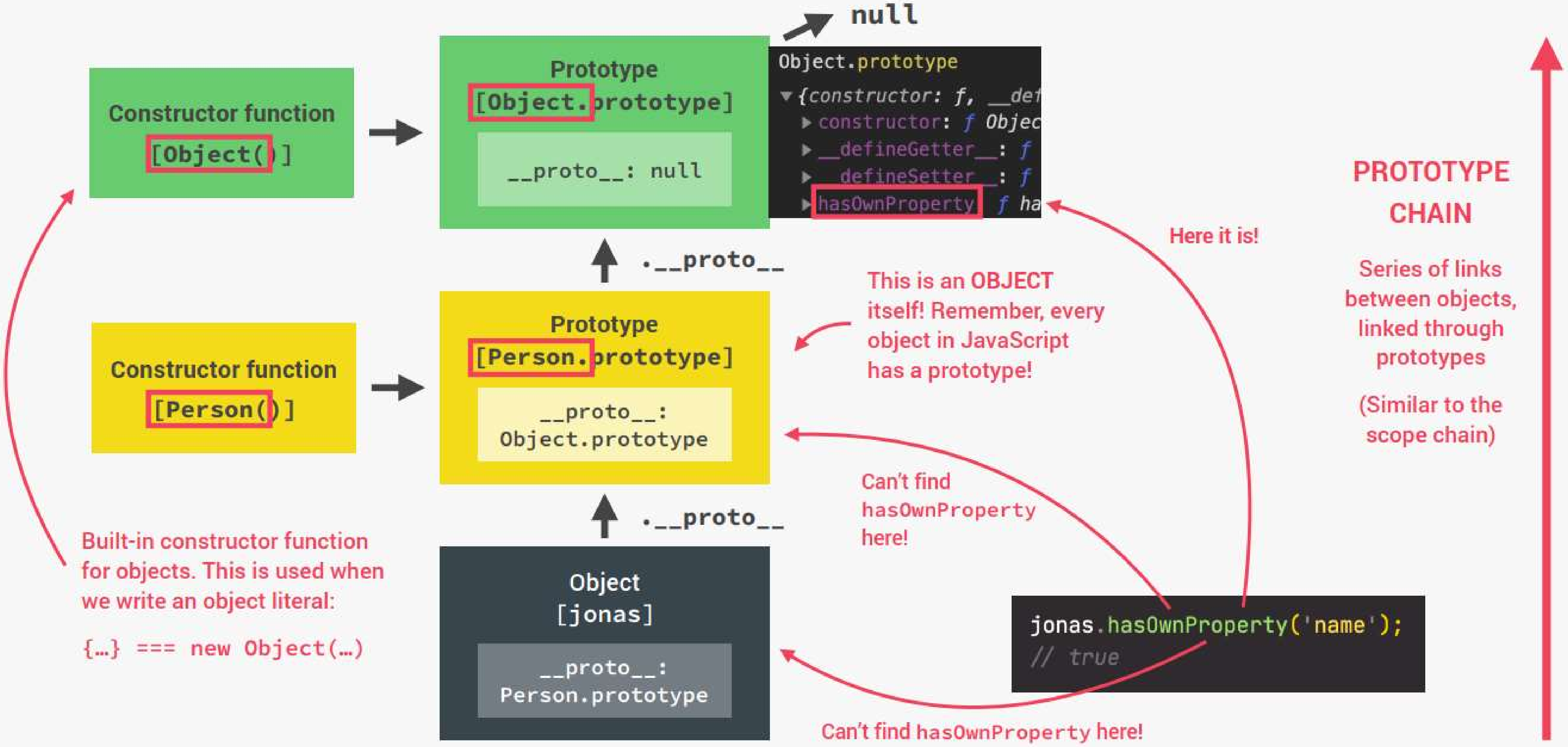
```
Person.prototype
▼{species: "Homo
 ▶calcAge: ƒ ()
```

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

```
const mike = new Student('Mike',
```

```
mike.calcAge();
// 17
```

# THE PROTOTYPE CHAIN



Constructor function
[Object()]

Prototype
[Object.prototype]

__proto__: null

null

```
Object.prototype
▼{constructor: ƒ, __def
  ▶constructor: ƒ Objec
  ▶__defineGetter__: ƒ
  ▶__defineSetter__: ƒ
  ▶hasOwnProperty: ƒ ha
```

Here it is!

.__proto__

Constructor function
[Person()]

Prototype
[Person.prototype]

__proto__:
Object.prototype

This is an OBJECT itself! Remember, every object in JavaScript has a prototype!

Can't find `hasOwnProperty` here!

.__proto__

Object
[jonas]

__proto__:
Person.prototype

```
jonas.hasOwnProperty('name');
// true
```

Can't find `hasOwnProperty` here!

Built-in constructor function for objects. This is used when we write an object literal:

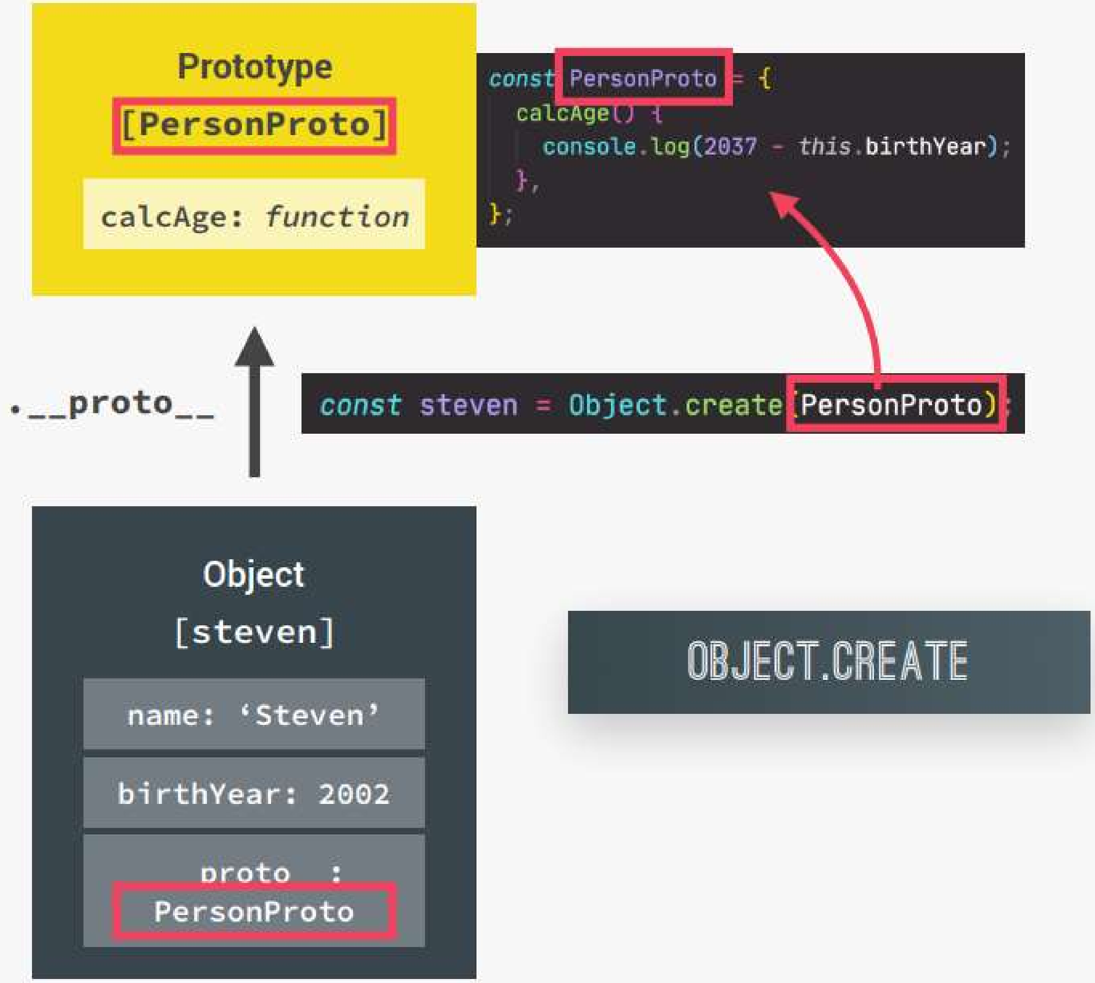
`{…} === new Object(…)`

**PROTOTYPE CHAIN**

Series of links between objects, linked through prototypes

(Similar to the scope chain)

# HOW OBJECT.CREATE WORKS



Prototype

[PersonProto]

calcAge: *function*

```
const PersonProto = {
  calcAge() {
    console.log(2037 - this.birthYear);
  },
};
```

.__proto__

```
const steven = Object.create(PersonProto);
```

Object

[steven]

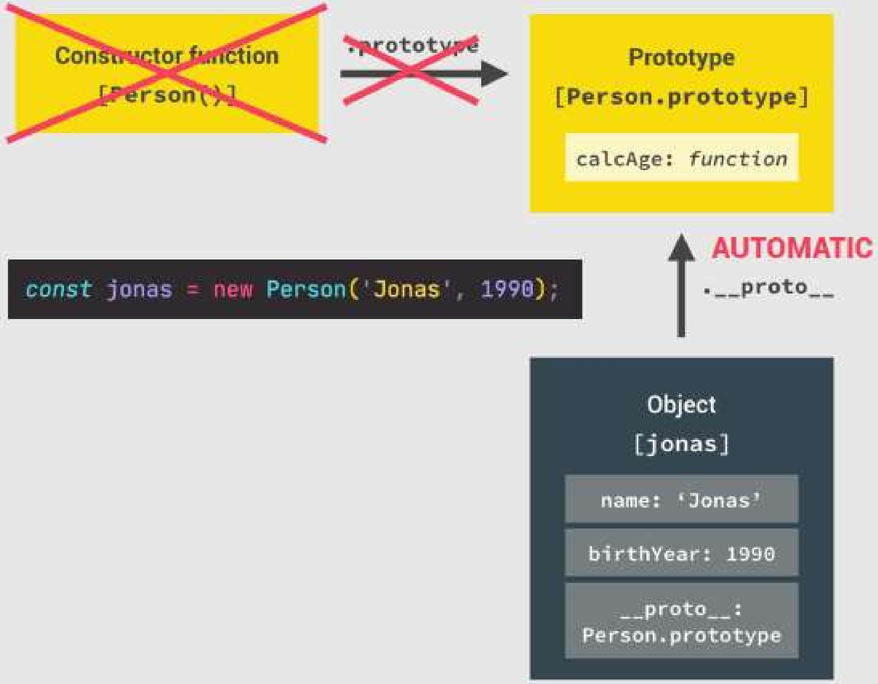
name: ‘Steven’

birthYear: 2002

__proto__: PersonProto

OBJECT.CREATE

## CONSTRUCTOR FUNCTIONS



~~Constructor function~~

~~[Person()]~~

~~.prototype~~

Prototype

[Person.prototype]

calcAge: *function*

```
const jonas = new Person('Jonas', 1990);
```

AUTOMATIC

.__proto__

Object

[jonas]

name: ‘Jonas’

birthYear: 1990

__proto__:
Person.prototype