Лекция №17

ООР

Что такое ООР

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

- Объектно-ориентированное программирование (ООП) — это парадигма программирования, основанная на концепции объектов;
- Мы используем объекты для моделирования (описания) реальных или абстрактных функций;

E.g. HTML component or data structure

- Объекты могут содержать данные (свойства) и код (методы). Используя объекты, мы упаковываем данные и соответствующее поведение в один блок;
- Объекты могут содержать данные (свойства) и код (методы). Используя
- Объекты являются строительными блоками приложений и взаимодействуют друг с другом;
- Взаимодействия происходят через общедоступный интерфейс (API): методы, к которым код вне объекта может получить доступ и использовать для связи с объектом;
- ООП было разработано с целью организации кода, чтобы сделать его более гибким.
и проще в обслуживании (избегайте «спагетти-кода»).

CLASS

```
User {
  user
  password
  email

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

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
  }
}
```

Instance

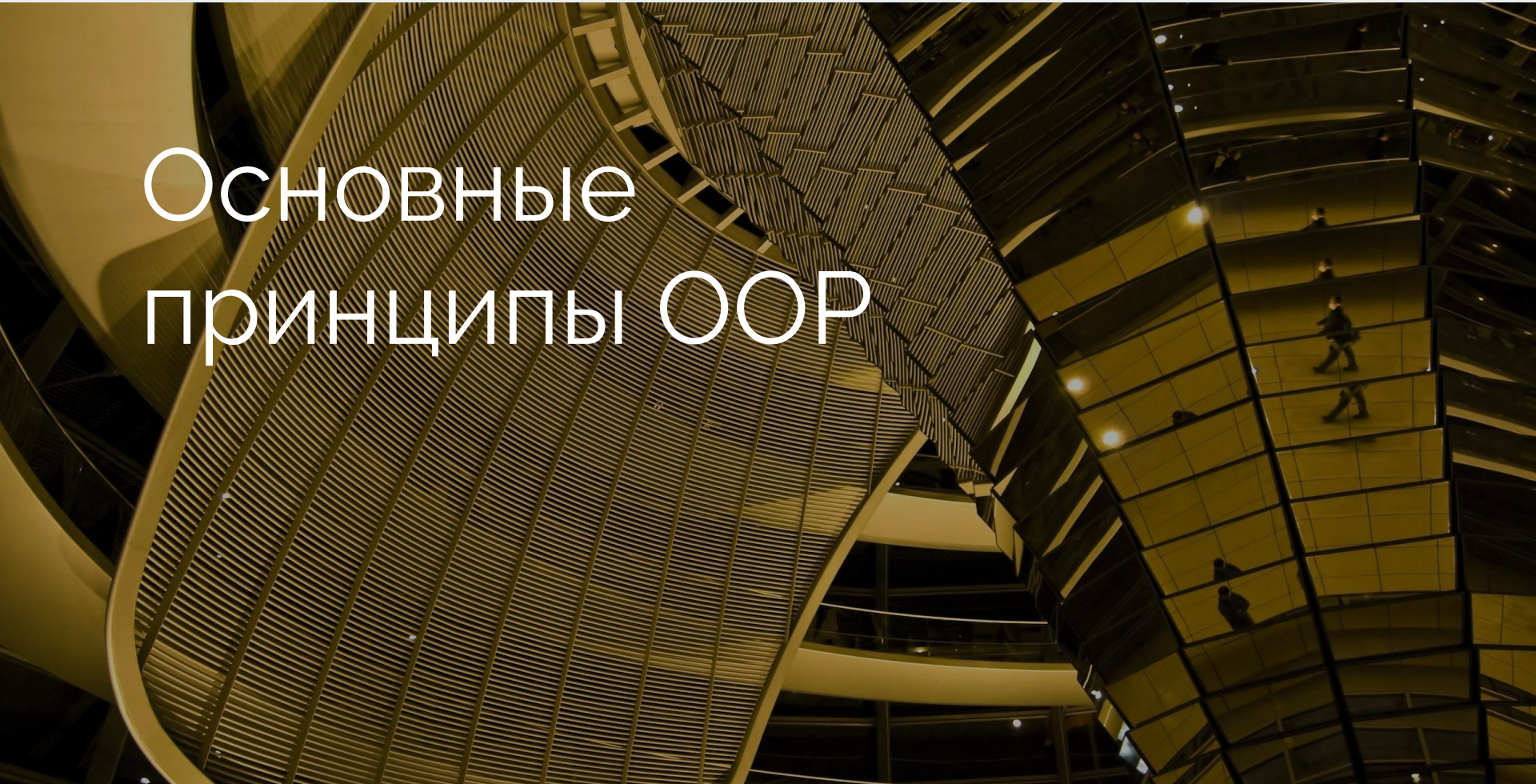
```
{
  user = 'mary'
  password = 'qwerty23'
  email = 'mary@test.com'

  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
  }
}
```


Основные
принципы ООР

```
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?

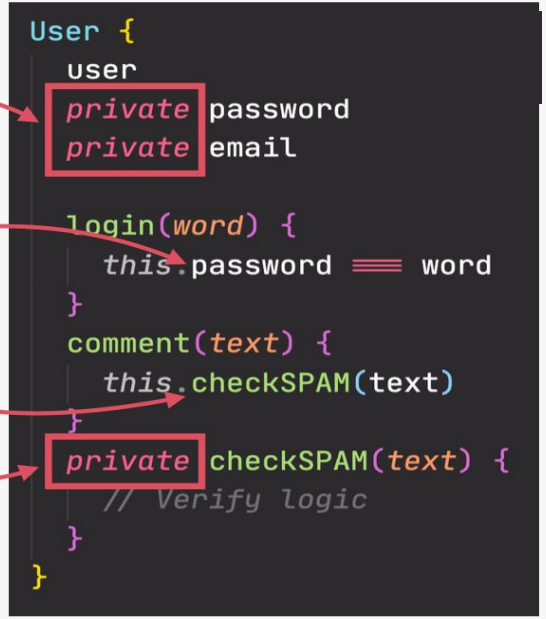
Абстракция: игнорирование или сокрытие деталей, которые не имеют значения, что позволяет нам получить общее представление о том, что мы реализуем, вместо того, чтобы возиться с деталями, которые на самом деле не имеют значения для нашей реализации.

NOT accessible from outside the class!

STILL accessible from within the class!

STILL accessible from within the class!

NOT accessible from outside the class!

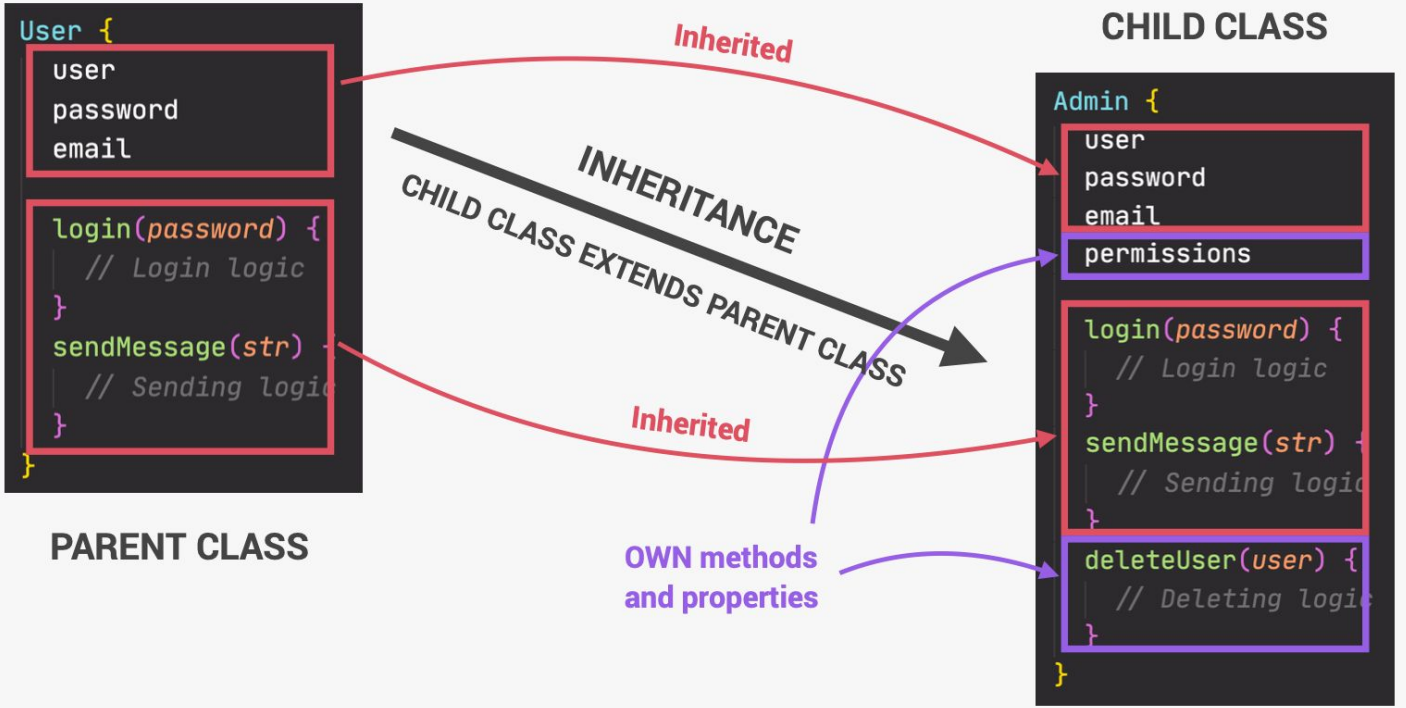


WHY?

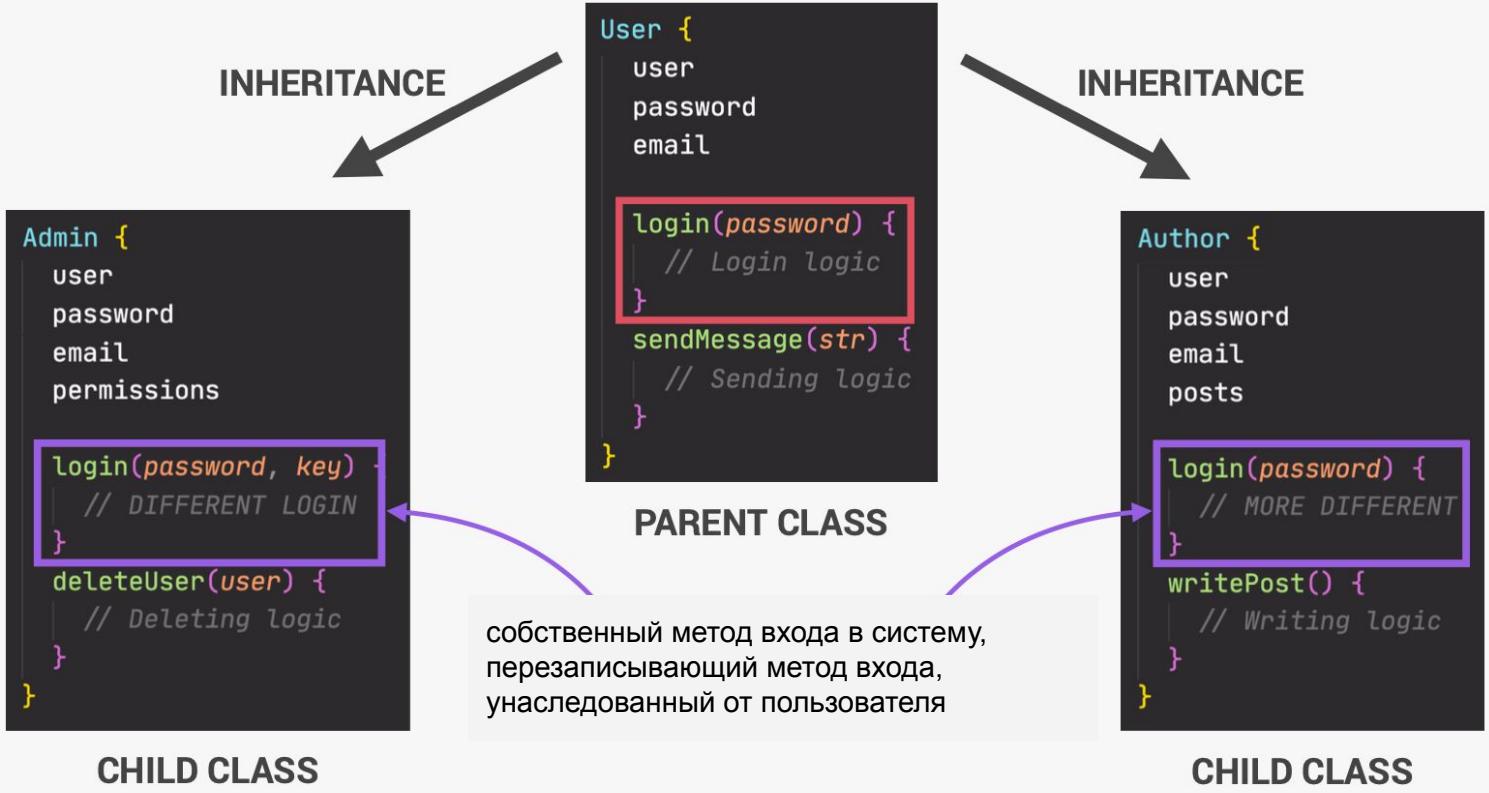
Предотвращает случайное манипулирование внутренними свойствами/состоянием внешнего кода.

Позволяет изменять внутреннюю реализацию без риска взлома внешнего кода

Инкапсуляция: Сохранение свойств и методов закрытыми внутри класса, поэтому они недоступны извне класса. Некоторые методы могут быть представлены как общедоступный интерфейс (API).



Наследование: Делает все свойства и методы определенного класса доступными для дочернего класса, формируя иерархические отношения между классами. Это позволяет нам повторно использовать общую логику и моделировать реальные отношения.



Полиморфизм: дочерний класс может перезаписать метод, унаследованный от родительского класса [это сложнее, но для наших целей достаточно].

🤔 *“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

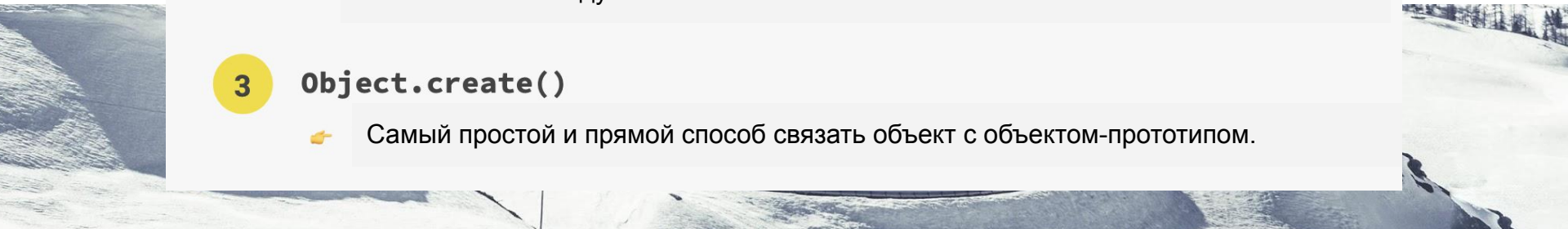
- Техника создания объектов из функции;
- Вот как на самом деле реализованы встроенные объекты, такие как массивы, Maps или Sets.

2 ES6 Classes

- Современная альтернатива синтаксису функций-конструкторов;
- «Синтаксический сахар»: за кулисами классы ES6 работают точно так же, как конструкторы;
- Классы ES6 НЕ ведут себя как классы в «классическом ООП».

3 `Object.create()`

- Самый простой и прямой способ связать объект с объектом-прототипом.



NOT of Person, but objects created by Person

Constructor function
[Person()]

.prototype

.constructor

Prototype
[Person.prototype]
calcAge: *function*

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

PROTOTYPE CHAIN

PROTOTYPAL INHERITANCE/ DELEGATION

.__proto__

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

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

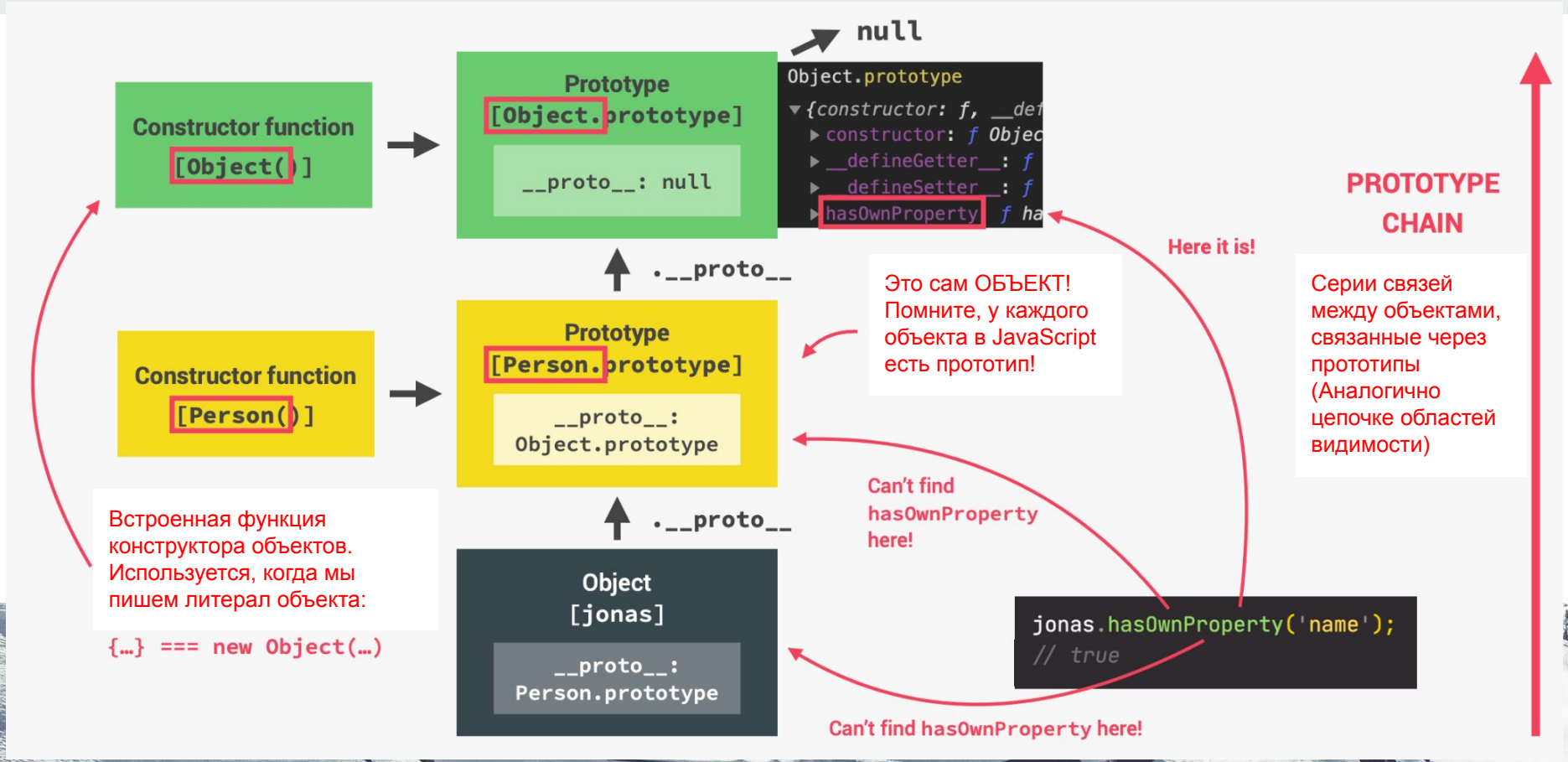
Can't find calcAge here!

Object
[jonas]
name: 'Jonas'
birthYear: 1990
__proto__: Person.prototype

Always points to an object's prototype

The new operator:

1. Создается пустой объект
2. это ключевое слово в вызове функции конструктора установлено для нового объекта
3. Новый объект связан (свойство __proto__) со свойством прототипа функции-конструктора.
4. Новый объект возвращается из вызова функции конструктора


null
Constructor function
[Object()]
Prototype
[Object.prototype]
__proto__: null
Object.prototype
{constructor: f, __def
constructor: f Objec
__defineGetter__: f
__defineSetter__: f
hasOwnProperty: f ha
.__proto__
Constructor function
[Person()]
Prototype
[Person.prototype]
__proto__:
Object.prototype
.__proto__
Object
[jonas]
__proto__:
Person.prototype
Here it is!
Это сам ОБЪЕКТ! Помните, у каждого объекта в JavaScript есть прототип!
PROTOTYPE CHAIN
Серии связей между объектами, связанные через прототипы (Аналогично цепочке областей видимости)
Can't find hasOwnProperty here!
Встроенная функция конструктора объектов. Используется, когда мы пишем литерал объекта:
{…} === new Object(…)
jonas.hasOwnProperty('name');
// true
Can't find hasOwnProperty here!

Спасибо!