

Javascript OOP

Programming Approaches

Процедурный - с использованием обычных переменных и функций

```
let person1Name = "Mike"
let person1Age = 25
let person1IsMale = true

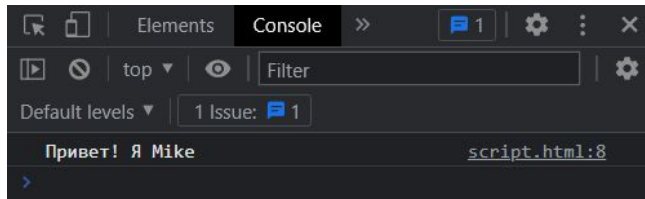
function sayHello(name) {
  console.log("Привет! Я " + name)
}

sayHello(person1Name)
```

Объектно-ориентированный - с использованием объектов и методов.

```
let person1 = {
  name: "Mike",
  age: 25,
  isMale: true,
  sayHello: function() {
    console.log("Привет! Я " + this.name)
  }
};

person1.sayHello()
```



Creating the same kind of objects: Problem

При выборе объектно-ориентированного подхода часто будет возникать необходимость создавать множество однотипных объектов. Что не очень удобно делать вручную.

```
let person1 = {  
  name: "Mike",  
  age: 25,  
  isMale: true,  
  sayHello: function() {  
    console.log("Привет! Я " + this.name)  
  }  
};
```

```
let person2 = {  
  name: "Artem",  
  age: 18,  
  isMale: true,  
  sayHello: function() {  
    console.log("Привет! Я " + this.name)  
  }  
};
```

```
let person3 = {  
  name: "Mary",  
  age: 34,  
  isMale: false,  
  sayHello: function() {  
    console.log("Привет! Я " + this.name)  
  }  
};
```

```
let person4 = {  
  name: "Sam",  
  age: 14,  
  isMale: true,  
  sayHello: function() {  
    console.log("Привет! Я " + this.name)  
  }  
};
```

OOP concepts

Инкапсуляция - сокрытие сложной реализации внутри объекта.
(Т.е. вызывающий код не знает, что происходит в объекте, он взаимодействует с объектом через его поля и методы.)

Наследование - создание новой сущности на базе уже существующей.
(Т.е. новый объект получает все свойства и методы другого объекта.)

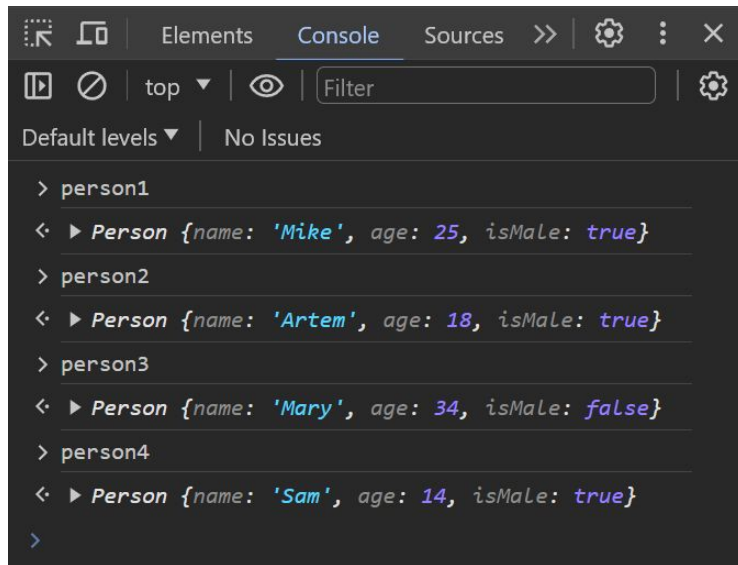
Полиморфизм - возможность иметь разные формы для одной и той же сущности.
(Т.е. один фрагмент кода может работать с разными типами данных.)

Classes ES6

Classes

По сути класс - это шаблон, по которому будут создаваться объекты

```
class Person {  
  constructor(name, age, isMale) {  
    this.name = name  
    this.age = age  
    this.isMale = isMale  
  }  
  sayHello() {  
    console.log("Привет! Я " + this.name)  
  }  
}  
  
let person1 = new Person("Mike", 25, true)  
let person2 = new Person("Artem", 18, true)  
let person3 = new Person("Mary", 34, false)  
let person4 = new Person("Sam", 14, true)
```



Classes

Имя класса

Конструктор

Входные аргументы для конструктора

```
class Person {  
  constructor(name, age, isMale) {  
    this.name = name  
    this.age = age  
    this.isMale = isMale  
  }  
  sayHello() {  
    console.log("Привет! Я " + this.name)  
  }  
}
```

Инициализация свойств объекта

Метод объекта

```
let person1 = new Person("Mike", 25, true)  
let person2 = new Person("Artem", 18, true)  
let person3 = new Person("Mary", 34, false)  
let person4 = new Person("Sam", 14, true)
```

Создание объектов (экземпляров класса)
с помощью класса

Class syntax

(Общий синтаксис класса)

```
class MyClass {  
  
    //конструктор  
    constructor() { ... }  
  
    // методы класса  
    method1() { ... }  
    method2() { ... }  
    method3() { ... }  
    ...  
}
```

(Минимальный синтаксис класса)

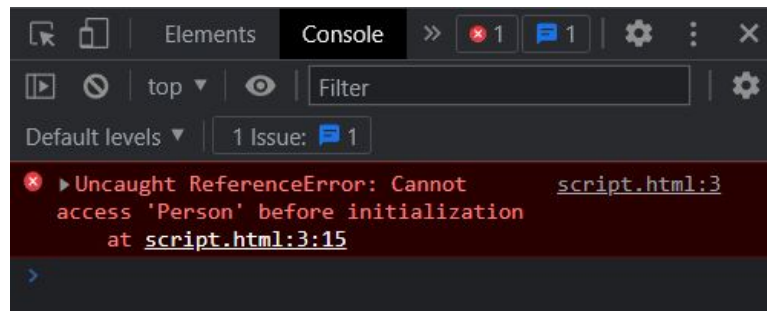
```
class EmptyClass {  
  
}
```


Using the class before declaring

```
let person1 = new Person("Mike", 25, true)
let person2 = new Person("Artem", 18, true)
let person3 = new Person("Mary", 34, false)
let person4 = new Person("Sam", 14, true)

class Person {
  constructor(name, age, isMale) {
    this.name = name
    this.age = age
    this.isMale = isMale
  }
  sayHello() {
    console.log("Привет! Я " + this.name)
  }
}
```

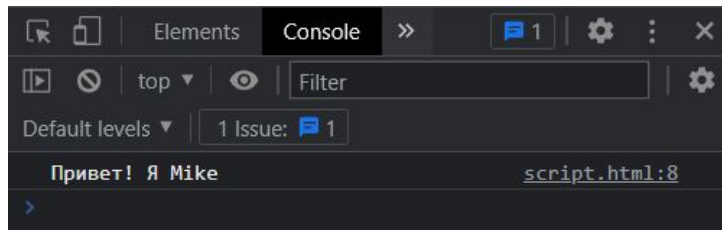
Нельзя пользоваться классом до инициализации



Properties outside the constructor

Свойства можно объявлять (аналогично методам) вне конструктора

```
class Person {  
  name = "Mike"  
  age = 25  
  isMale = true  
  
  sayHello() {  
    console.log("Привет! Я " + this.name)  
  }  
}  
  
let person = new Person()  
person.sayHello()
```



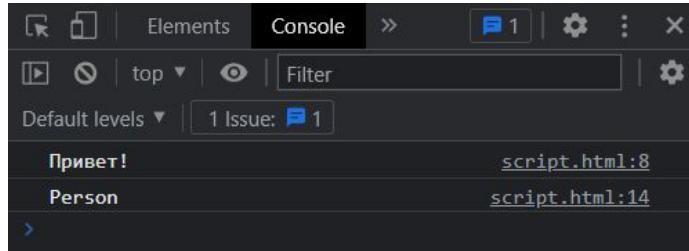
Static properties and methods

```
class Person {  
    static name = "Person"  
    static sayHello() {  
        console.log("Привет!")  
    }  
}
```

Статические свойства и методы относятся к самому классу, а не объектам, которые он создает. (Поэтому они не будут добавлены в объекты которые создаются по этому классу)

```
Person.sayHello()  
console.log(Person.name)
```

Чтобы ими пользоваться, нужно обращаться к классу напрямую



Inheritance ES6

Inheritance

При создании нового класса мы можем указать, что он наследуется от какого-то другого уже существующего класса. В таком случае наш новый класс будет обладать всеми свойствами и методами родительского класса, а в теле нового класса мы уже можем добавить его собственные свойства и методы, которых не будет у родителя.

```
class Enemy {  
    constructor hp, damage {  
        this.hp = hp  
        this.damage = damage  
    }  
  
    attack() {  
        console.log("-" + this.damage + "hp")  
    }  
}  
  
class Orc extends Enemy{  
}
```

При наследовании используется ключевое слово "extends"

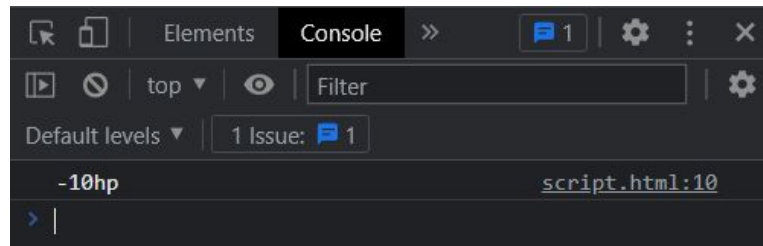
Класс Orc будет иметь все методы которые есть у Enemy.
Поля от Enemy тоже унаследуются, но будут undefined, так как мы пока не проинициализировали их в конструкторе.

Calling the parent constructor

```
class Enemy {  
  constructor hp, damage {  
    this.hp = hp  
    this.damage = damage  
  }  
  
  attack() {  
    console.log("-" + this.damage + "hp")  
  }  
}
```

```
class Orc extends Enemy {  
  constructor hp, damage {  
    super hp, damage  
  }  
}  
  
let orc = new Orc(100, 10)  
orc.attack()
```

В каждом конструкторе наследника необходимо вызывать конструктор родителя для инициализации его свойств.
Делается это через кл. слово **super**

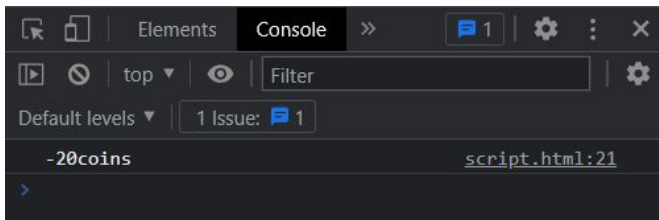


Inheritance

Мы можем добавлять любые другие дополнительные свойства и методы дочернему классу. На родительском это никак не отразится.

```
class Enemy {  
  constructor hp, damage {  
    this.hp = hp  
    this.damage = damage  
  }  
  
  attack() {  
    console.log("-" + this.damage + "hp")  
  }  
}
```

```
class Orc extends Enemy {  
  constructor hp, damage, boost {  
    super hp, damage  
    this.boost = boost  
  }  
  
  rob() {  
    console.log("-" + this.boost + "coins")  
  }  
}  
  
let orc = new Orc(100, 10, 20)  
orc.rob()
```

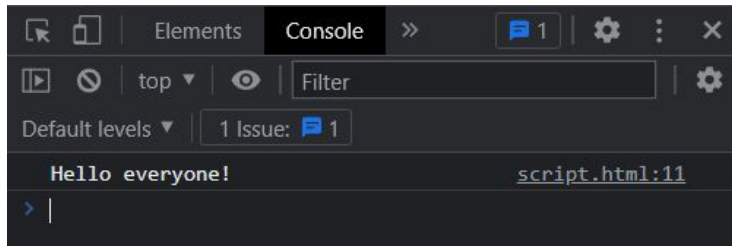


Overriding methods

При переопределении родительский метод затирается

```
class Parent {  
    sayHello() {  
        console.log("Hello!")  
    }  
}
```

```
class Child extends Parent{  
    sayHello() {  
        console.log("Hello everyone!")  
    }  
}  
  
let child = new Child()  
child.sayHello()
```

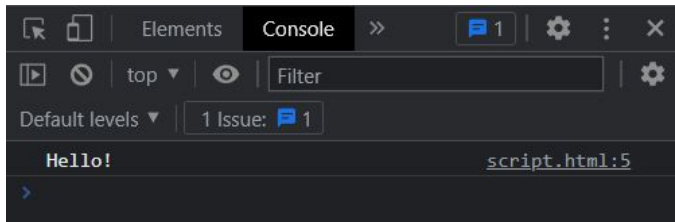


Overriding methods

```
class Parent {  
  sayHello() {  
    console.log("Hello!")  
  }  
}
```

При переопределении родительский метод всё ещё доступен внутри объекта через ключевое слово `super`

```
class Child extends Parent {  
  sayHello() {  
    console.log("Hello everyone!")  
  }  
  sayHelloOriginal() {  
    super.sayHello()  
  }  
}  
  
let child = new Child()  
child.sayHelloOriginal()
```




instanceof

Оператор **instanceof** позволяет проверить, к какому классу принадлежит объект, с учетом наследования.


```
class Person {  
    static name = "Person"  
    static sayHello() {  
        console.log("Привет!")  
    }  
}  
  
class Student extends Person {  
}  
  
let student = new Student()
```

instanceof сверяет прототип по цепочке.
Поэтому объект будет принадлежать ко всем родительским классам.




```
> student instanceof Student  
< true  
> student instanceof Person  
< true  
> student instanceof Object  
< true  
>
```

Но объект родительского класса *не принадлежит* к дочернему



```
> (new Person()) instanceof Student  
< false
```

Дополнительный пример:
Объект не является массивом.
Но массив является объектом.*



```
> ([]) instanceof Array  
< true  
> ({}) instanceof Array  
< false  
> ([]) instanceof Object  
< true
```

*Так как Array наследуется от Object

Extending built-in classes

Мы можем наследоваться также и от стандартных классов, таких как Array, Map, Number и т.п.

Наследуемся от Array

```
class ArrayWithPrint extends Array {  
  print() {  
    for(let i=0; i<this.length; i++) {  
      console.log(this[i])  
    }  
  }  
}
```

```
> let arr;  
   arr = new ArrayWithPrint(100, 200, 300, 400)  
< ▶ ArrayWithPrint(4) [100, 200, 300, 400]  
  
> arr.print()  
100 script.html:6  
200 script.html:6  
300 script.html:6  
400 script.html:6  
  
< undefined
```

Getters and Setters

Getters and setters

Контроль над свойствами можно установить с помощью геттеров и сеттеров.

```
class Student {  
  _age = 0;  
  
  set age(value) {  
    this._age = value  
  }  
  
  get age() {  
    return this._age  
  }  
}
```

Реальное свойство. По соглашению начинается с символа “_”. Геттер и сеттер будут работать с ним. (Но технически имя этого свойства могло быть абсолютно любым.)

```
let student = new Student()
```

Сеттер будет вызываться при записи

```
student.age = 25
```

Геттер будет вызываться при чтении

```
console.log(student.age)
```

Getters and setters

При получении и установке свойств через геттеры и сеттеры можно проводить различные проверки и манипуляции:

```
class Student {  
  _age = 0;  
  
  set age(value) {  
    if(value>0 && value<100) {  
      this._age = value  
    } else {  
      alert("Указан неправильный возраст")  
      return  
    }  
  }  
  
  get age() {  
    let digit = this._age % 10  
    if(digit == 1)  
      return this._age + " год"  
    if(digit == 2 || digit == 3 || digit == 4)  
      return this._age + " года"  
  
    return this._age + " лет"  
  }  
}
```

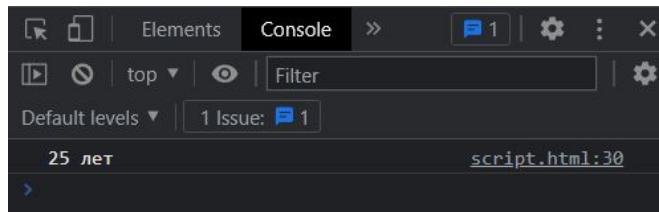
```
student.age = -6  
console.log(student.age)
```

This page says

Указан неправильный возраст

OK

```
let student = new Student()  
  
student.age = 25  
console.log(student.age)
```



Private and protected class members

Приватные и защищенные поля нужны, чтобы ограничить доступ к свойствам и методам классов из внешнего кода.

```
class Counter {  
  constructor() {  
    this._count = 0  
  }  
  _increment(num) {  
    return num + 1  
  }  
  count() {  
    this._count = this._increment(this._count)  
    return this._count  
  }  
}
```

Защищенные свойства и методы начинаются с символа “_”

Они используются только внутри класса. Внешний код не должен к ним обращаться.

Тем не менее это не правила языка, а *договоренность разработчиков*.

Внешний код технически может их вызывать, как и любые обычные свойства.

```
> let counter;  
  counter = new Counter()  
< ▶ Counter { _count: 0 }
```

```
> counter.count()  
< 1
```

```
> counter.count()  
< 2
```

```
> counter.count()  
< 3
```

```
>
```

```
> counter._count  
< 3
```

Private and protected class members

Приватные и защищенные поля нужны, чтобы ограничить доступ к свойствам и методам классов из внешнего кода.

```
class Counter {  
  #count = 0  
  constructor() {  
  }  
  #increment(num) {  
    return num + 1  
  }  
  count() {  
    this.#count = this.#increment(this.#count)  
    return this.#count  
  }  
}
```

Приватные свойства и методы начинаются с символа "#". Они используются только внутри класса. Внешний код не может к ним обращаться.

В отличие от защищённых полей, **приватные** реализованы на уровне языка.

Обратиться к ним из внешнего кода невозможно в принципе.

```
> let counter;  
  counter = new Counter()  
< ▶ Counter {#increment: f, #count: 0}  
  
> counter.count()  
< 1  
  
> counter.count()  
< 2  
  
> counter.count()  
< 3  
  
> counter.#count
```

✖ Uncaught SyntaxError: Private field '#count' must be declared in an enclosing class [VM1988:1](#)

(Кроме того приватные свойства нужно задавать вне конструктора)

Private and protected class members

Несмотря на поддержку языком, **приватные поля** имеют значительный недостаток - они **не передаются при наследовании**.

Приватные поля не наследуются!

```
class Counter {
    #count = 0
    #increment(num) {
        return num + 1
    }
}

class SuperCounter extends Counter {
    constructor() {
        super()
        console.log(this.#count)
        console.log(this.#increment(1))
    }
}
```

Вызовет ошибку

Защищенные поля наследуются

```
class Counter {
    _count = 0
    _increment(num) {
        return num + 1
    }
}

class SuperCounter extends Counter {
    constructor() {
        super()
        console.log(this._count)
        console.log(this._increment(1))
    }
}
```

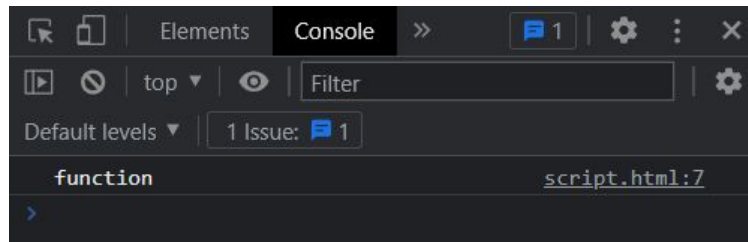
Из-за такого поведения эти поля и были названы *приватными*(private) и *защищенными*(protected)

Classes as functions,
Prototype inheritance

A class is a function

Если передать класс оператору `typeof`, то он вернёт тип `function`.

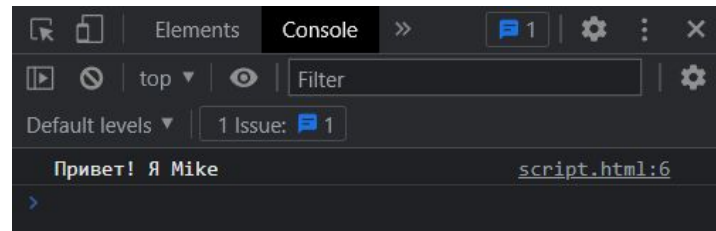
```
class Person {  
  
}  
  
console.log(typeof Person)
```



Creating objects with the function

Любую функцию можно использовать как конструктор объектов, вызвав через ключевое слово `new`.

```
function Person(name) {  
  this.name = name  
  this.sayHello = function() {  
    console.log("Привет! Я " + this.name)  
  }  
}  
  
let person1 = new Person("Mike")  
person1.sayHello()
```



Этим методом активно пользовались до введения в JS синтаксиса классов.

Creating objects with the function

Синтаксис функций

```
function Person(name) {  
    this.name = name  
  
    this.sayHello = function() {  
        console.log("Привет! Я " + this.name)  
    }  
}
```

```
let person1 = new Person("Mike")  
person1.sayHello()
```

Синтаксис класса

```
class Person {  
    constructor(name) {  
        this.name = name  
    }  
    sayHello() {  
        console.log("Привет! Я " + this.name)  
    }  
}
```

```
let person1 = new Person("Mike")  
person1.sayHello()
```

Class expression

Так как классы являются функциями, их тоже можно объявлять, присваивая переменным (**classes expression**).

Classes expression

```
const Person = class {  
  constructor(name) {  
    this.name = name  
  }  
  sayHello() {  
    console.log("Привет! Я " + this.name)  
  }  
}  
  
let person1 = new Person("Mike")
```

Classes declaration

```
class Person {  
  constructor(name) {  
    this.name = name  
  }  
  sayHello() {  
    console.log("Привет! Я " + this.name)  
  }  
}  
  
let person1 = new Person("Mike")
```

В принципе переменную можно объявить и через let

Prototype inheritance

В JS наследование реализуется через прототипы. У каждого объекта есть объект-прототип (лежащий в скрытом свойстве `[[prototype]]`), от которого он наследует (получает) свойства и методы. (прототип может быть null)

У каждой функции-конструктора есть свойство **prototype**, в котором и находится объект-прототип для созданных этой функцией объектов. По умолчанию **prototype** конструктора равен `{constructor: <функция-конструктор>}`

```
function PersonConstructor(name) {  
  this.name = name  
  this.sayHello = function() {  
    console.log("Привет! Я " + this.name)  
  }  
}  
  
PersonConstructor.prototype = {  
  type: "human",  
  printType() {  
    console.log(this.type)  
  }  
}  
  
let person = new PersonConstructor("Mike")  
console.log(person)
```

Устанавливаем прототип, который будет установлен на все объекты, созданные этим конструктором

Видим, что наш прототип установился на созданный функцией объект

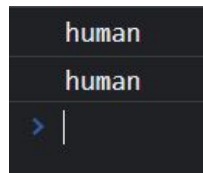
```
▼ PersonConstructor {name: 'Mike', sayHello: f} ⓘ  
  name: "Mike"  
  ▶ sayHello: f ()  
  ▼ [[Prototype]]: Object  
    ▶ printType: f printType()  
    type: "human"  
    ▶ [[Prototype]]: Object
```

У прототипа тоже может быть свой прототип.
У любого объекта по умолчанию прототипом является Object.prototype

Prototype inheritance

Если происходит обращение к свойству или методу, **которых нет у самого объекта**, то JS будет **искать их у прототипа**. (Если и у прототипа их нет, то у прототипа прототипа и так по цепочке пока последний прототип не окажется null)

```
function PersonConstructor(name) {  
  this.name = name  
  this.sayHello = function() {  
    console.log("Привет! Я " + this.name)  
  }  
}  
  
PersonConstructor.prototype = {  
  type: "human",  
  printType() {  
    console.log(this.type)  
  }  
}  
  
let person = new PersonConstructor("Mike")  
console.log(person.type)  
person.printType()
```



При обращении к методу или свойству, которых нету у объекта, будет производиться их поиск у прототипа

Prototype inheritance

Чтобы наследовать свойства от другого конструктора, **необходимо вызвать его в своём конструкторе с подменой контекста**

```
function Enemy hp, speed {  
  this.hp = hp  
  this.speed = speed  
}  
  
function Orc hp, speed, damage {  
  Enemy.call(this, hp, speed)  
  this.damage = damage  
}  
  
let orc = new Orc(100, 5, 10)  
console.log(orc)
```

Вызываем родительский конструктор в дочернем конструкторе со своим контентом.

Это аналогично вызову **super()** в синтаксисе классов.

Таким образом Orc унаследовал свойства от конструктора Enemy

► Orc {hp: 100, speed: 5, damage: 10}

>

Prototype inheritance

Методы принято задавать в прототипе, а не конструкторе (чтобы не плодить копии определений методов). Таким образом, чтобы наследовать методы от родительского конструктора, нужно просто скопировать его прототип.

```
function Enemy hp, speed {  
  this.hp = hp  
  this.speed = speed  
}  
Enemy.prototype.showHp = function() {  
  console.log("My hp: " + this.hp)  
}  
Enemy.prototype.showSpeed = function() {  
  console.log("My speed: " + this.speed)  
}  
  
function Orc hp, speed, damage {  
  Enemy.call(this, hp, speed)  
  this.damage = damage  
}  
Orc.prototype = Object.create(Enemy.prototype)  
Orc.prototype.showDamage = function() {  
  console.log("My damage: " + this.damage)  
}
```

Создаем методы для Enemy

Object.create создает пустой объект, с указанным прототипом*

Наследуем методы Enemy

Создаем метод для Orc

```
let orc = new Orc(100, 5, 10)  
orc.showHp()  
orc.showSpeed()  
orc.showDamage()
```

My hp: 100

My speed: 5

My damage: 10

*Мы могли бы просто присвоить прототипу Orc прототип Enemy, но тогда при добавлении собственных методов для Orc это меняло бы прототип Enemy)

Syntax of functions vs Syntax of classes

Синтаксис функций

```
function Enemy hp, speed {
  this.hp = hp
  this.speed = speed
}
Enemy.prototype.showHp = function() {
  console.log("My hp: " + this.hp)
}
Enemy.prototype.showSpeed = function() {
  console.log("My speed: " + this.speed)
}

function Orc hp, speed, damage {
  Enemy.call(this, hp, speed)
  this.damage = damage
}
Orc.prototype = Object.create(Enemy.prototype)
Object.defineProperty(Orc.prototype, 'constructor', {value: Orc, enumerable: false});
Orc.prototype.showDamage = function() {
  console.log("My damage: " + this.damage)
}
```



Также чтобы полностью копировать поведение классов нам **нужно задать свойство 'constructor'** для Orc.prototype

Object.defineProperty добавляет свойство переданному объекту и устанавливает для этого свойства различные параметры

Синтаксис класса

```
class Enemy {
  constructor hp, speed {
    this.hp = hp
    this.speed = speed
  }
  showHp() {
    console.log("My hp: " + this.hp)
  }
  showSpeed() {
    console.log("My speed: " + this.speed)
  }
}

class Orc extends Enemy{
  constructor hp, speed, damage {
    super hp, speed
    this.damage = damage
  }
  showDamage() {
    console.log("My damage: " + this.damage)
  }
}
```

Syntax of functions vs Syntax of classes

Синтаксис функций

```
> new Orc(100, 5, 20)
< ▼ Orc {hp: 100, speed: 5, damage: 20} ⓘ
  damage: 20
  hp: 100
  speed: 5
  ▼ [[Prototype]]: Enemy
    ▶ showDamage: f ()
    ▶ constructor: f Orc(hp, speed, damage)
  ▼ [[Prototype]]: Object
    ▶ showHp: f ()
    ▶ showSpeed: f ()
    ▶ constructor: f Enemy(hp, speed)
  ▼ [[Prototype]]: object
```

Хоть мы и создали
аналогичный
функционал без классов,
но классы всё равно
помечаются в консоли
именно как классы

Object.prototype

Синтаксис класса

```
> new Orc(100, 5, 20)
< ▼ Orc {hp: 100, speed: 5, damage: 20} ⓘ
  damage: 20
  hp: 100
  speed: 5
  ▼ [[Prototype]]: Enemy
    ▶ constructor: class Orc
    ▶ showDamage: f showDamage()
  ▼ [[Prototype]]: Object
    ▶ constructor: class Enemy
    ▶ showHp: f showHp()
    ▶ showSpeed: f showSpeed()
  ▼ [[Prototype]]: object
```

Поэтому классы не на 100% являются синтаксическим сахаром

Syntax of functions vs Syntax of classes

Синтаксис функций

```
> new Orc(100, 5, 20)
< ▼Orc {hp: 100, speed: 5, damage: 20} ⓘ
  damage: 20
  hp: 100
  speed: 5
  ▼[[Prototype]]: Enemy
    ▶ showDamage: f ()
    ▶ constructor: f Orc(hp, speed, damage)
  ▼[[Prototype]]: Object
    ▶ showHp: f ()
    ▶ showSpeed: f ()
    ▶ constructor: f Enemy(hp, speed)
  ▼[[Prototype]]: Object ←
    ▶ constructor: f Object()
    ▶ hasOwnProperty: f hasOwnProperty()
    ▶ isPrototypeOf: f isPrototypeOf()
    ▶ propertyIsEnumerable: f propertyIsEnumerable()
    ▶ toLocaleString: f toLocaleString()
    ▶ toString: f toString()
    ▶ valueOf: f valueOf()
    ▶ __defineGetter__: f __defineGetter__()
    ▶ __defineSetter__: f __defineSetter__()
    ▶ __lookupGetter__: f __lookupGetter__()
    ▶ __lookupSetter__: f __lookupSetter__()
    ▶ __proto__: (...)
    ▶ get __proto__: f __proto__()
    ▶ set __proto__: f __proto__()
```

Тот самый
Object.prototype, которым
кончаются все цепочки
прототипов в JS. (Всё
наследуется от объекта)

Синтаксис класса

```
> new Orc(100, 5, 20)
< ▼Orc {hp: 100, speed: 5, damage: 20} ⓘ
  damage: 20
  hp: 100
  speed: 5
  ▼[[Prototype]]: Enemy
    ▶ constructor: class Orc
    ▶ showDamage: f showDamage()
  ▼[[Prototype]]: Object
    ▶ constructor: class Enemy
    ▶ showHp: f showHp()
    ▶ showSpeed: f showSpeed()
  ▼[[Prototype]]: Object
    ▶ constructor: f Object()
    ▶ hasOwnProperty: f hasOwnProperty()
    ▶ isPrototypeOf: f isPrototypeOf()
    ▶ propertyIsEnumerable: f propertyIsEnumerable()
    ▶ toLocaleString: f toLocaleString()
    ▶ toString: f toString()
    ▶ valueOf: f valueOf()
    ▶ __defineGetter__: f __defineGetter__()
    ▶ __defineSetter__: f __defineSetter__()
    ▶ __lookupGetter__: f __lookupGetter__()
    ▶ __lookupSetter__: f __lookupSetter__()
    ▶ __proto__: (...)
    ▶ get __proto__: f __proto__()
    ▶ set __proto__: f __proto__()
```

Other differences between class and function syntax

- Статические свойства и методы не наследуются с помощью прототипов, но наследуются с помощью синтаксиса классов и ключевого слова **extends**.

```
function Person() {}  
Person.name = "Person"  
Person.sayHello = function() {  
  console.log("Привет!")  
}  
  
function Student() {}  
  
Object.assign(Student, Person)
```

```
> Student.name  
< 'Student'  
> Student.sayHello()  
Привет!  
< undefined  
> Student  
< f Student() {}
```

Чтобы унаследовать статические свойства и методы, нужно их копировать с помощью метода **Object.assign**

```
class Person {  
  static name = "Person"  
  static sayHello() {  
    console.log("Привет!")  
  }  
}  
  
class Student extends Person {  
}
```

```
> Student.name  
< 'Student'  
> Student.sayHello()  
Привет!  
< undefined  
> Student  
< class Student extends Person {  
  }  
}
```

extend без лишних напрягов копирует статические методы

- Классы JavaScript должны быть объявлены перед доступом.** В противном случае выдается ошибка. Функции же, объявленные через function declaration, могут использоваться в коде выше их объявления. Это свойство функций называется совершением подъёма.
- В остальном классы используют “под капотом” функции и прототипы

Extending built-in classes

С помощью изменения прототипов встроенных классов можно расширить или изменить их функциональность.

```
Array.prototype.print = function() {  
  for(let i=0; i<this.length; i++) {  
    console.log(this[i])  
  }  
}
```

Добавляем метод
print для массивов

```
> [1,2,3,4,5].print()  
1  
2  
3  
4  
5  
◀ undefined
```

Меняем метод push для массивов

```
Array.prototype.push = function (value) {  
  return this.unshift(value);  
}
```

Теперь push
добавляет элемент
не в конец, а в
начало массива!

```
> let arr;  
  arr = [1, 2, 3]  
◀ ▶ (3) [1, 2, 3]  
>> arr.push(4)  
◀ 4  
>> arr  
◀ ▶ (4) [4, 1, 2, 3]
```

когда вас будут увольнять, можете спрятать такой код, в каком-нибудь большом файле :)

Темы для докладов

- 1) WebWorkers. Service workers vs WebWorkers.

<https://bitsofco.de/web-workers-vs-service-workers-vs-worklets/>

<https://www.dhiwise.com/post/web-workers-vs-service-workers-in-javascript>

- 2) CommonJS vs EsmaScript modules.

<https://blog.logrocket.com/commonjs-vs-es-modules-node-js/>

<https://medium.com/globant/exploring-node-js-modules-commonjs-vs-es6-modules-2766e838bea9>