**Подготовка к собеседованию. Прототипы объектов. Введение**

Девятый спринт учебной программы почти позади. Кажется, вы твёрдо решили стать разработчиком раз дошли до него. Если так, рано или поздно вам предстоит найти работу в этой сфере. Процесс найма устроен по-разному. Где-то вам зададут вопросы о вашем опыте и дадут тестовое задание, а где-то в дополнение к этому проведут техническое интервью.

Первое вам предстоит сформулировать в треке по трудоустройству ближе к концу программы. Готовиться к техническому интервью начнём прямо сейчас.

Такое интервью обычно состоит из двух частей: вопросов по вёрстке и JavaScript, и практической части, на которой нужно писать код. Теперь в конце каждого спринта будет тема, посвящённая подготовке к техническому интервью. В каких-то из этих тем вы будете изучать глубокие концепции JavaScript и аспекты браузера. В каких-то тренироваться в написании кода и решении задач.

Перед вами первая из таких тем — «Прототипы объектов». Вопрос про прототип могут задать на собеседовании, но прототипы и без этого очень важная для JS-разработчика тема. Она открывает глаза на то, как устроен JS.

Обещаем, что после неё вы и JavaScript станете немного ближе.

# Прототип объекта. Свойство \_\_proto\_\_

В одной из прошлых тем мы рассматривали песни как объекты:

Скопировать кодJAVASCRIPT

const song1 = {

title: 'Футбольный мяч',

artist: 'Антоха MC',

isLiked: false,

like: function () {

song1.isLiked = !song1.isLiked;

}

};

const song2 = {

title: 'На заре',

artist: 'Альянс',

isLiked: false,

like: function () {

song2.isLiked = !song2.isLiked;

}

};

const song3 = {

title: 'Ай',

artist: 'Хаски',

isLiked: true,

like: function () {

song3.isLiked = !song3.isLiked;

}

};

Каждый объект хранит функцию like: это одна и та же функция, она ставит лайк «своей» песне, то есть меняет свойство isLiked объекта, в котором находится. Функция делает одно и то же, но хранится в виде множества копий — по одной для каждого объекта песни. Это нехорошо.

Вы уже умеете сокращать такой код с применением классов, но пока забудьте о них. Скоро вы узнаете, что же на самом деле стоит за классами. Но, чтобы это понять, нужно разобраться с тем, что такое прототип.

### Свойство \_\_proto\_\_

У каждого объекта в JS есть свойство \_\_proto\_\_. Оно позволяет одному объекту обращаться к свойствам другого.

Работает это так:

* Вы обращаетесь к свойству объекта, которого у того нет.
* Движок, не найдя нужного свойства, не прекращает поиски, а заглядывает в свойство \_\_proto\_\_: если здесь хранится ссылка на другой объект, движок будет искать нужное свойство у него.

Скопировать кодJAVASCRIPT

const obj1 = {

messageRu: 'Привет, мир!'

};

const obj2 = {

messageEng: 'Hello, world!'

};

obj2.\_\_proto\_\_ = obj1;

console.log(obj2.messageEng); *// "Hello, world!"*

console.log(obj2.messageRu); *// "Привет, мир!" — свойство messageRu тоже нашлось*

Объекты obj2 и obj1 связаны ссылкой \_\_proto\_\_. В последней строке мы обращаемся к свойству messageRu, которого нет в объекте obj2. Но движок находит его в объекте obj1, переходя по ссылке \_\_proto\_\_.

Так можно объединить несколько объектов в цепочку:

Скопировать кодJAVASCRIPT

const obj3 = {

messageRu: 'Привет, мир!'

};

const obj4 = {

messageEng: 'Hello, world!'

};

const obj5 = {

messageFrench: 'Bonjour le monde!'

};

obj4.\_\_proto\_\_ = obj3;

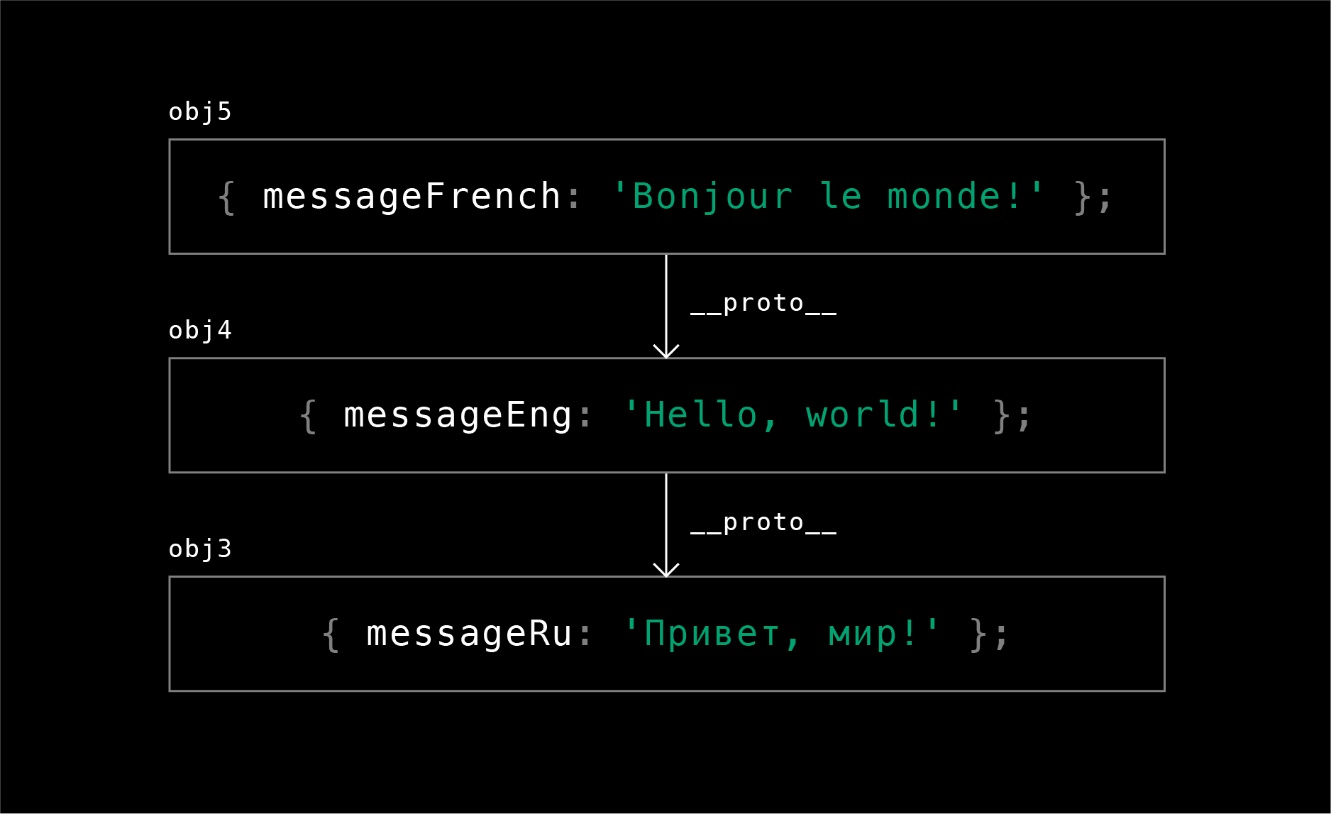
obj5.\_\_proto\_\_ = obj4;

console.log(obj5.messageFrench); *// "Bonjour le monde!"*

console.log(obj5.messageEng); *// "Hello, world!" — не нашлось в obj5, но нашлось по ссылке \_\_proto\_\_ в obj4*

console.log(obj5.messageRu); *// "Привет, мир!" — не нашлось в obj5, не нашлось по ссылке \_\_proto\_\_ в obj4, но нашлось по ссылке obj4.\_\_proto\_\_, то есть в obj3*

Процесс поиска свойства в объекте obj5 будет таким:



То, что хранится в свойстве \_\_proto\_\_ , называют прототип объекта. obj3 — прототип объекта obj4, а тот в свою очередь прототип объекта obj5.

# Прототип объекта — это другой объект, где движок будет искать свойство или метод, если их нет в исходном объекте.

Чтобы организовать объекты в цепочку, можно вручную назначить им ссылки \_\_proto\_\_, но обычно так не делают. Движок JavaScript не любит такие манёвры и из-за них может работать медленнее. Как правильно задавать объекту прототип, расскажем в следующем уроке.

## Резюме

* В JavaScript объекты можно организовать в цепочку, для этого есть специальное свойство \_\_proto\_\_.
* Если свойство или метод объекта не найдены в самом объекте, движок ищет их в объекте по ссылке в свойстве \_\_proto\_\_.
* Объект, записанный в \_\_proto\_\_ называют прототипом того объекта, в котором хранится эта ссылка.

# Метод Object.create

Рассмотрим ещё один пример, [который мы приводили](https://praktikum.yandex.ru/trainer/web/lesson/213a1fd1-cf8f-4806-97fb-34fa03351a15) до того, как познакомились с классами:

Скопировать кодJAVASCRIPT

*// функция createSong создаёт объекты*

function createSong(title, artist) {

const newSong = {

title,

artist,

isLiked: false,

*// каждый из этих объектов хранит функцию like*

like: function () {

newSong.isLiked = !newSong.isLiked;

}

};

return newSong;

}

*// можно создать несколько песен одной функцией*

const song1 = createSong('Футбольный мяч', 'Антоха MC');

const song2 = createSong('На заре', 'Альянс');

const song3 = createSong('Ай', 'Хаски');

Мы уже упоминали, что в этом примере есть проблема. Функция like делает для каждого объекта одно и то же, но хранится как свойство каждой отдельной песни. Бестолку множить сущности не стоит, поэтому функцию вынесем в отдельный объект — прототип. Затем привяжем все песни к этому прототипу.

Скопировать кодJAVASCRIPT

*// этот объект должен стать прототипом объектов песен*

const songPrototype = {

like: function () {

newSong.isLiked = !newSong.isLiked;

}

};

Правда теперь переменной newSong у нас нет. Чтобы функция like стала универсальной заменим newSong в её теле на this:

Скопировать кодJAVASCRIPT

const songPrototype = {

like: function () {

this.isLiked = !this.isLiked;

}

};

Осталось сделать этот объект прототипом песни. Один из способов сделать это — метод Object.create.

## Метод Object.create

Метод Object.create создаёт новый пустой объект. Первый аргумент метода — обязательный — объект, который должен стать прототипом нового объекта:

Скопировать кодJAVASCRIPT

*// это будет прототип объекта-песни*

const songPrototype = {

like: function () {

this.isLiked = !this.isLiked;

}

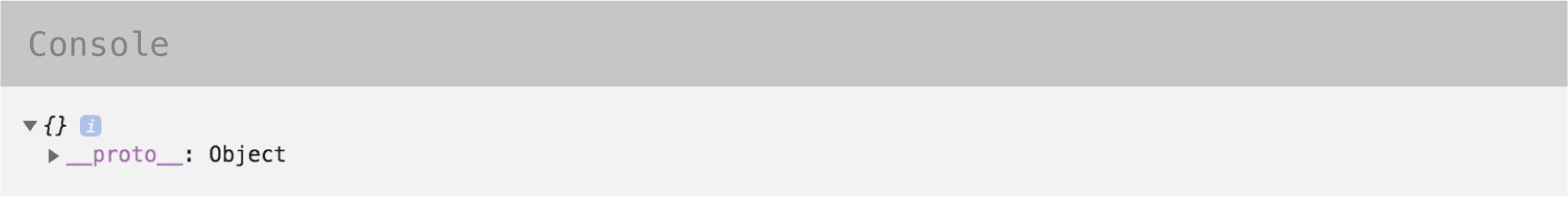
};

*// создаём пустой объект с указанным прототипом*

const newSong = Object.create(songPrototype);

console.log(newSong); *// пустой объект*

Скопируйте этот код в консоль — там появится пустой объект:



И откройте свойство \_\_proto\_\_, кликнув по нему:



Теперь функция like «живёт» в прототипе объекта newSong.

Добавим объекту newSong нужные свойства:

Скопировать кодJAVASCRIPT

*// прототип объекта newSong*

const songPrototype = {

like: function () {

this.isLiked = !this.isLiked;

}

};

*// создаём пустой объект с прототипом*

const newSong = Object.create(songPrototype);

*// добавляем нужные свойства в объект*

newSong.title = 'Футбольный мяч';

newSong.artist = 'Антоха MC';

newSong.isLiked = false;

console.log(newSong); *// теперь данные хранятся в объекте newSong, а функциональность в прототипе*

Логика понятна, теперь перенесём её в функцию createSong:

Скопировать кодJAVASCRIPT

*// прототип объекта newSong*

const songPrototype = {

like: function () {

this.isLiked = !this.isLiked;

}

};

function createSong(title, artist) {

*// создаём пустой объект с прототипом*

const newSong = Object.create(songPrototype);

*// добавляем нужные свойства в объект*

newSong.title = title;

newSong.artist = artist;

newSong.isLiked = false;

*// возвращаем объект песни*

return newSong;

}

const song1 = createSong('Футбольный мяч', 'Антоха MC');

const song2 = createSong('На заре', 'Альянс');

const song3 = createSong('Ай', 'Хаски');

console.log(song1); *// { title: 'Футбольный мяч' ... }*

console.log(song2); *// { title: 'На заре' ... }*

console.log(song1.like === song2.like); *// true*

Скопируйте этот код в консоль и исследуйте содержимое объектов song1 и song2. Как видите эти объекты содержат все нужные свойства, но функция like теперь в объекте по ссылке \_\_proto\_\_. Этой ссылкой все три песни ссылаются на songPrototype. Код больше не дублируется.

## Резюме

Метод Object.create создаёт новый пустой объект с указанным прототипом:

Скопировать кодJAVASCRIPT

const objPrototype = {

foo: function() {

console.log('foo');

}

};

const obj = Object.create(objPrototype);

console.log(obj); *// {} — пустой объект*

console.log(obj.\_\_proto\_\_); *// { foo: ƒ } — прототип*

После создания объекта таким образом, ему можно добавить свойства или методы:

Скопировать кодJAVASCRIPT

const objPrototype = {

foo: function() {

console.log('foo');

}

};

const obj = Object.create(objPrototype);

obj.bar = 'bar';

console.log(obj); *// { bar: "bar" } — появилось свойство bar*

console.log(obj.\_\_proto\_\_); *// { foo: ƒ }*

# Оператор new

Вернёмся к функции createSong. Она выполняет три действия:

1. создаёт новый пустой объект с привязанным прототипом;
2. добавляет этому объекту необходимые свойства;
3. возвращает этот объект.

Скопировать кодJAVASCRIPT

function createSong(title, artist) {

*// 1. Создаём пустой объект с прототипом*

const newSong = Object.create(songPrototype);

*// 2. Добавляем нужные свойства в объект*

newSong.title = title;

newSong.artist = artist;

newSong.isLiked = false;

*// 3. Возвращаем объект*

return newSong;

}

Два шага из трёх можно автоматизировать. Мы намеренно этого не делали, чтобы вам было проще понять работу функций-конструкторов.

# Функция-конструктор

Любую функцию в JavaScript можно вызвать с ключевым словом new перед вызовом:

Скопировать кодJAVASCRIPT

const song = new createSong('На заре', 'Альянс');

Если движок натыкается на такой вызов, он начинает исполнять код внутри этой функции, но дополняет стандартный процесс исполнения кода двумя шагами (1 и 3 в списке):

1. Перед тем, как исполнять код, создаёт новый пустой объект и записывает его в значение this внутри этой функции.
2. Исполняет код функции.
3. Возвращает this.

Получается, если функция createSong вызвана с оператором new от шага создания объекта и его возвращения можно избавиться — эти действия всё равно выполнятся благодаря new:

Скопировать кодJAVASCRIPT

function createSong(title, artist) {

*// this = {}; — движок сделает это сам*

*// Добавим в this необходимые свойства*

this.title = title;

this.artist = artist;

this.isLiked = false;

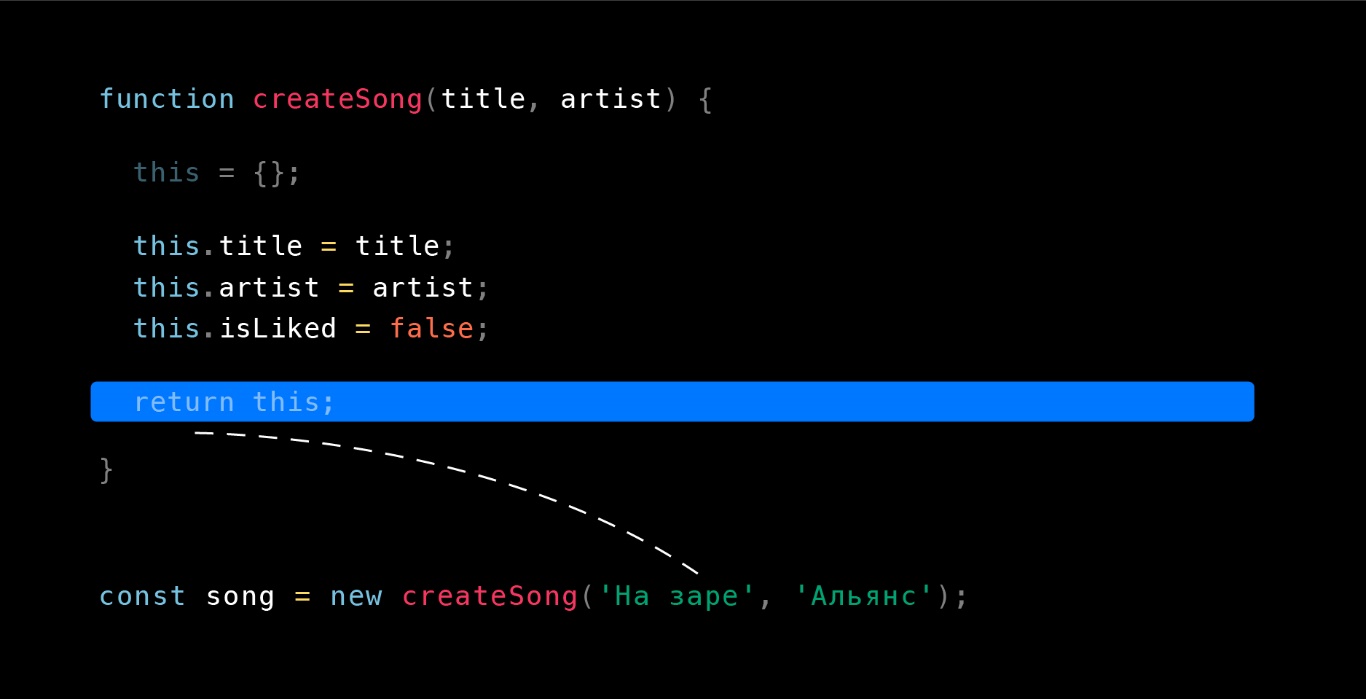
*// return this; — это движок тоже берёт на себя*

}

const song = new createSong('На заре', 'Альянс');

console.log(song);

Запустите этот код в консоли, чтобы убедиться, что функция вернула объект со свойствами title, artist и isLiked, которые мы добавили в this (хотя вручную мы this не возвращали).



Функции, предназначенные для того, чтобы их вызывали с оператором new, называют функциями-конструкторами. Их основная задача — «сконструировать» какой-то новый объект, в нашем случае объект песни. Отсюда и название.

# Именование функций-конструкторов

Получившийся пример работает, но есть проблема. По имени функции невозможно понять, что её следует вызывать с оператором new. Что, если её код написал один разработчик, а пользуется этим кодом другой. Конечно, можно изучить код функции и понять, что это конструктор. Но это долго и сложно, поэтому разработчики придумали способ проще. Функции-конструкторы называют с прописной буквы. Также, функцию-конструктор обычно называют существительным. Отличное имя для нашей функции — Song, потому что функция-конструктор создаёт объекты песен:

Скопировать кодJAVASCRIPT

function Song(title, artist) {

this.title = title;

this.artist = artist;

this.isLiked = false;

}

const song = new Song('На заре', 'Альянс');

console.log(song);

Для движка имя функции роли не играет: ему неважно, назовём мы её с заглавной буквы или со строчной. Именование функции с прописной не делает её конструктором. Но вы говорите другим разработчикам, что эту функцию следует вызывать как конструктор, то есть с оператором new.

# Свойство prototype

В прошлом уроке вы узнали о функциях-конструкторах. Такие функции вызывают с оператором new: он говорит движку, что перед тем как исполнять код функции, нужно создать объект, а после выполнения — созданный объект вернуть.

Но вернёмся к тому, как наша функция выглядела до превращения в конструктор. В её теле был такой шаг:

Скопировать кодJAVASCRIPT

const newSong = Object.create(songPrototype);

Этим шагом мы говорили движку, что нужно создавать не просто пустой объект, а пустой объект с songPrototype в качестве прототипа. Но когда мы переделали функцию в конструктор, этот шаг потеряли. Так что теперь движок не знает, какой объект брать за прототип новых объектов, что приводит к ошибке:

Скопировать кодJAVASCRIPT

const songPrototype = {

like: function () {

this.isLiked = !this.isLiked;

}

};

function Song(title, artist) {

this.title = title;

this.artist = artist;

this.isLiked = false;

}

const song1 = new Song('Футбольный мяч', 'Антоха MC');

console.log(song1.like); *// undefined — значит такого свойство нет ни в объекте, ни в прототипе*

song1.like(); *// ошибка — undefined не функция*

Выход из этой ситуации прост. Помните, что функция — это объект? Так вот, у любой функции-конструктора есть свойство prototype. То, что хранится в этом свойстве, становится прототипом объекта, созданного этой функцией.

То есть, когда функция Song будет вызвана с оператором new, за кулисами произойдёт следующее:

Скопировать кодJAVASCRIPT

function Song(title, artist) {

*// Произойдёт за кулисами:*

*// this = Object.create(Song.prototype);*

this.title = title;

this.artist = artist;

this.isLiked = false;

}

const song1 = new Song('Футбольный мяч', 'Антоха MC');

Объект Song.prototype будет автоматически добавлен как прототип для нового объекта, то есть в его свойство \_\_proto\_\_.

Поэтому нужно записать в свойство prototype объект songPrototype — тогда всё снова заработает.

Скопировать кодJAVASCRIPT

function Song(title, artist) {

this.title = title;

this.artist = artist;

this.isLiked = false;

}

*// добавим функцию like в объект Song.prototype*

Song.prototype.like = function () {

this.isLiked = !this.isLiked;

};

const song1 = new Song('Футбольный мяч', 'Антоха MC');

*/\* теперь функция like снова есть*

*в прототипе и доступна в новых объектах \*/*

console.log(song1.isLiked); *// false*

song1.like();

console.log(song1.isLiked); *// true*

Не путайте свойства \_\_proto\_\_ и prototype — это не одно и то же. Вот в чём отличия:

* свойство \_\_proto\_\_ принадлежит объекту, свойство prototype — функции-конструктору;
* эти два свойства служат разным целям. \_\_proto\_\_ нужно движку для того, чтобы знать, где искать свойство, если в объекте оно не найдено. А свойство prototype — чтобы движок понял, куда будет вести ссылка \_\_proto\_\_ у объекта, который создаёт функция-конструктор.

# Функции-конструкторы: резюме

Функции, конструирующие объекты, называют функциями-конструкторами.

Функции-конструкторы называют с заглавной буквы. Это подразумевает, что такую функцию нужно вызывать с оператором new.

Скопировать кодJAVASCRIPT

const song1 = new Song('Футбольный мяч', 'Антоха MC');

const song2 = new Song('На заре', 'Альянс');

const song3 = new Song('Ай', 'Хаски');

Вызов c new автоматически создаёт пустой объект, записывает его в this, и после того, как функция отработала, возвращает this. Мы же в своём коде модифицируем this в теле функции:

Скопировать кодJAVASCRIPT

function Song(title, artist) {

this.title = title;

this.artist = artist;

this.isLiked = false;

}

В свойство prototype функции-конструктора можно положить методы и свойства, которые будут добавлены в прототип, то есть в свойство \_\_proto\_\_ сконструированного объекта:

Скопировать кодJAVASCRIPT

*// like будет добавлен в прототип объектов песен*

Song.prototype.like = function () {

this.isLiked = !this.isLiked;

};

# Что стоит за классами в JavaScript?

Сейчас наша функция-конструктор выглядит так:

Скопировать кодJAVASCRIPT

*// создаём функцию-конструктор*

function Song(title, artist) {

*// в её теле модифицируем this*

this.title = title;

this.artist = artist;

this.isLiked = false;

}

*// добавляем метод в прототип*

Song.prototype.like = function () {

this.isLiked = !this.isLiked;

};

*// создаём объект*

const song1 = new Song('Футбольный мяч', 'Антоха MC');

Вам ничего не напоминает это код?

Да. Приведённая функция-конструктор делает то же самое, что и такой класс:

Скопировать кодJAVASCRIPT

*// создаём класс Song*

class Song {

constructor(title, artist) {

*// в конструкторе класса модифицируем this*

this.title = title;

this.artist = artist;

this.isLiked = false;

}

*// добавляем классу методы, они попадут в прототип объекта*

*// который будет создан таким классом*

like() {

this.isLiked = !this.isLiked;

}

}

*// создаём объект классом Song*

const song1 = new Song('Футбольный мяч', 'Антоха MC');

Вот и раскрыт главный секрет классов в JS.

Класс — это не какая-то отдельная сущность или тип данных. Это просто более удобная запись функции-конструктора.

До 2015 года в JS не было классов и все пользовались только функциями-конструкторами. Но разработчики спецификации решили, что неудобно каждый раз писать такие функции и записывать методы в их свойство prototype. Так был придуман синтаксис классов.

Именно синтаксис, так как классы не добавляют новых возможностей в язык. Когда мы создаём класс, на самом деле, мы:

* создаём функцию-конструктор;
* добавляем методы в её свойство prototype.



Просто синтаксис классов позволяет сделать это в более удобной форме. Так действительно удобнее, но не забывайте, как это работает внутри. На собеседовании обязательно спросят.

# Встроенные конструкторы и их прототипы

Вы узнали, что такое прототип. Если запросить свойство, которого у объекта нет, движок пойдёт искать его в другой объект, который хранится в свойстве \_\_proto\_\_. Так можно связывать несколько объектов в цепочку:

Скопировать кодJAVASCRIPT

const needle = {

contentDeath: 'смерть Кощея'

};

const egg = {

contentNeedle: 'Игла',

\_\_proto\_\_: needle *// Как вы помните, напрямую перезаписывать свойство \_\_proto\_\_ не следует. Здесь мы так сделали только для наглядности.*

};

const duck = {

contentEgg: 'Яйцо',

\_\_proto\_\_: egg

};

const rabbit = {

contentDuck: 'Утка',

\_\_proto\_\_: duck

};

const chest = {

contentRabbit: 'Заяц',

\_\_proto\_\_: rabbit

};

const koscheyDeath = {

\_\_proto\_\_: chest

};

console.log(koscheyDeath.contentDeath); *// "смерть Кощея"*

В этом уроке мы будем говорить о встроенных методах объектов.

# Свойство \_\_proto\_\_ встроенных объектов

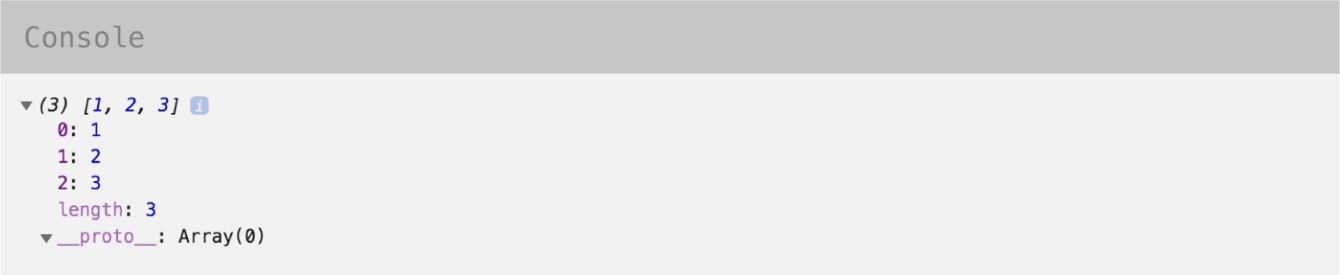
Массивы тоже объекты. При этом у массивов есть собственные методы: push, filter, map и другие. Создадим простой массив и выведем его в консоль:

Скопировать кодJAVASCRIPT

const arr = [1, 2, 3];

console.log(arr);

Вот, что мы увидим в консоли:



Это записанные в массив данные. Но где его методы? Откройте свойство \_\_proto\_\_, кликнув по нему:



Массив — это объект, потому у него тоже есть свойство \_\_proto\_\_. Это ссылка на прототип массива — там хранятся его встроенные методы.

Мы не добавляли метода push в массив, и этого метода у массива нет. Но когда мы вызываем метод push, движок идёт по ссылке \_\_proto\_\_ и находит нужный метод там.

# Встроенные функции-конструкторы

Несколько раз мы уже обращались к встроенному объекту Array: пользовались его методами Array.isArray и Array.from.

Заглавная буква в названии объекта Array подсказывает, что это особый объект. На самом деле, это встроенная функция-конструктор массива. Её можно вызвать с оператором new, чтобы создать массив:

Скопировать кодJAVASCRIPT

const arr = new Array(1, 2, 3);

console.log(arr); *// [1, 2, 3]*

Обычно массивы так не создают — проще и удобнее поставить квадратные скобки. Но это важно знать для понимания иерархии объектов внутри JavaScript.

Кроме Array есть встроенные функции конструкторы и для других объектов: Object, Function. Функции-конструкторы есть даже у некоторых примитивов: Number, String, Boolean, но ими никогда не пользуются напрямую.

# Свойство prototype встроенных конструкторов

У встроенных функций-конструкторов есть и свойство prototype. Когда мы создаём новый массив, в его свойство \_\_proto\_\_ попадает объект, хранящийся в Array.prototype:

Скопировать кодJAVASCRIPT

const arr2 = [1, 2, 3];

console.log(arr2.\_\_proto\_\_ === Array.prototype); *// true, потому что свойство \_\_proto\_\_ ссылается на Array.prototype*

Свойство \_\_proto\_\_ каждого типа данных указывает на prototype соответствующей функции-конструктора:

Скопировать кодJAVASCRIPT

const obj = {};

const num = 4;

const str = 'Привет';

const bool = true;

const func = function () {

console.log('Hello world!');

};

console.log(obj.\_\_proto\_\_ === Object.prototype); *// true*

console.log(num.\_\_proto\_\_ === Number.prototype); *// true*

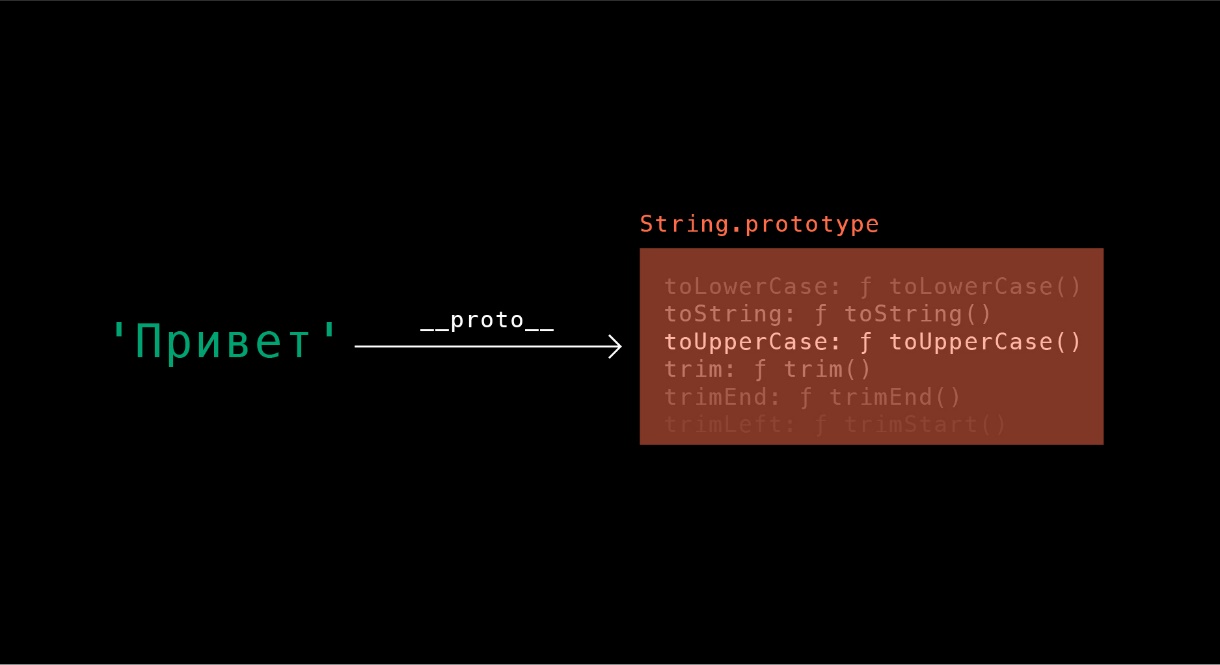
console.log(str.\_\_proto\_\_ === String.prototype); *// true*

console.log(bool.\_\_proto\_\_ === Boolean.prototype); *// true*

console.log(func.\_\_proto\_\_ === Function.prototype); *// true*

Внимание, вопрос. Где хранятся встроенные методы строк? Например, toUpperCase или toLowerCase?

В String.prototype. Один ноль в пользу знатоков.



Когда мы создаём новую строку, ссылка на String.prototype попадает в свойство \_\_proto\_\_ созданной строки.

# Во главе всего — объект

Number.prototype, String.prototype, Boolean.prototype — это тоже объекты. Стало быть, у них тоже есть свойство \_\_proto\_\_. Оно есть и ссылается на Object.prototype:

Скопировать кодJAVASCRIPT

console.log(String.prototype.\_\_proto\_\_ === Object.prototype); *// true*

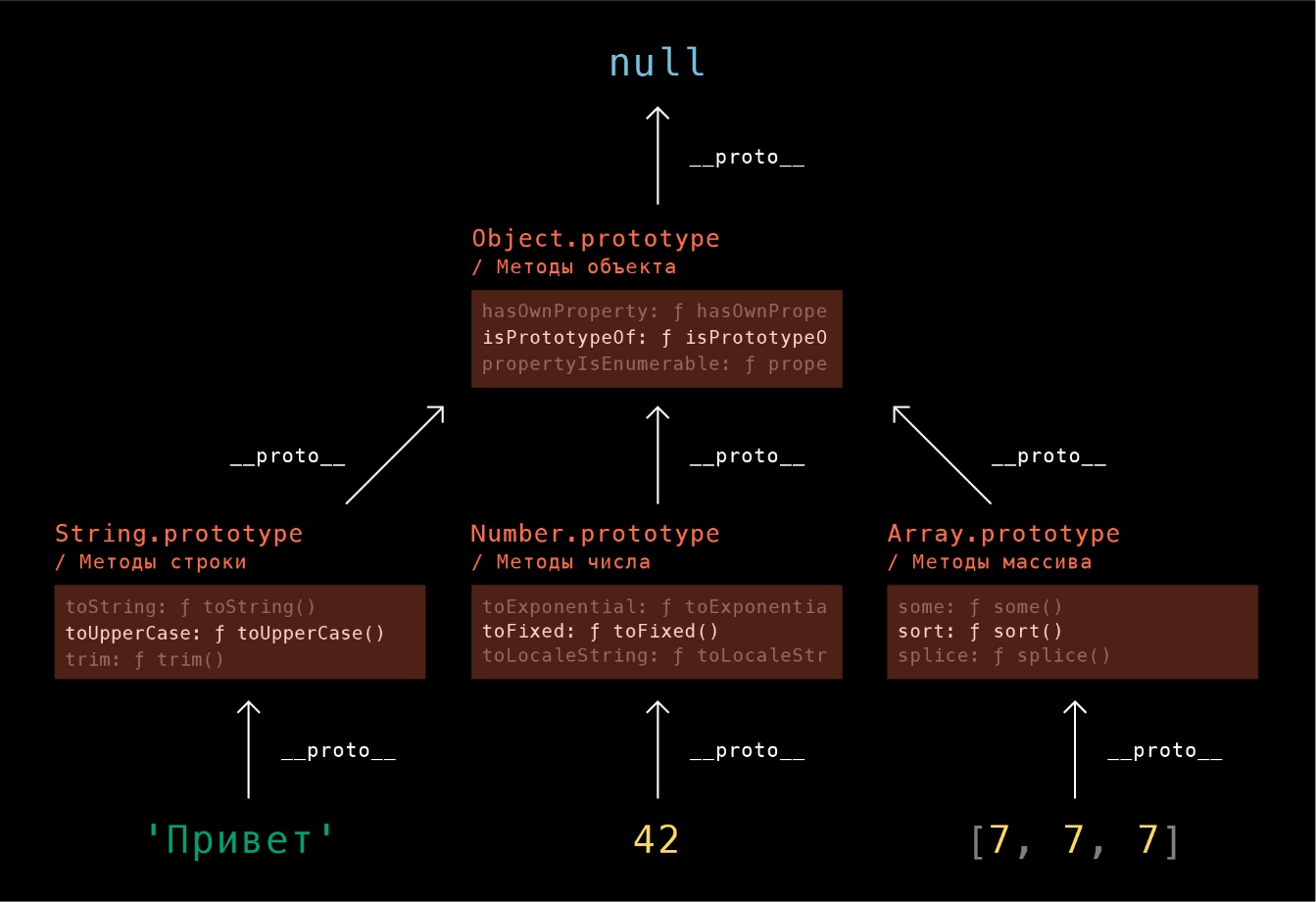
console.log(Number.prototype.\_\_proto\_\_ === Object.prototype); *// true*

console.log(Boolean.prototype.\_\_proto\_\_ === Object.prototype); *// true*

console.log(Function.prototype.\_\_proto\_\_ === Object.prototype); *// true*

console.log(Array.prototype.\_\_proto\_\_ === Object.prototype); *// true*

Встроенные объекты тоже связаны друг с другом, образуя такую иерархию:



Если метод массива не найден в Array.prototype, движок продолжит поиски в Object.prototype. То есть у массивов есть все методы, которые есть у объектов. А благодаря звену Array.prototype, у массива есть дополнительные методы. Массив имеет доступ ко всем методам объекта и дополняет его своими. Принято говорить, что Array.prototype «наследует» от Object.prototype.

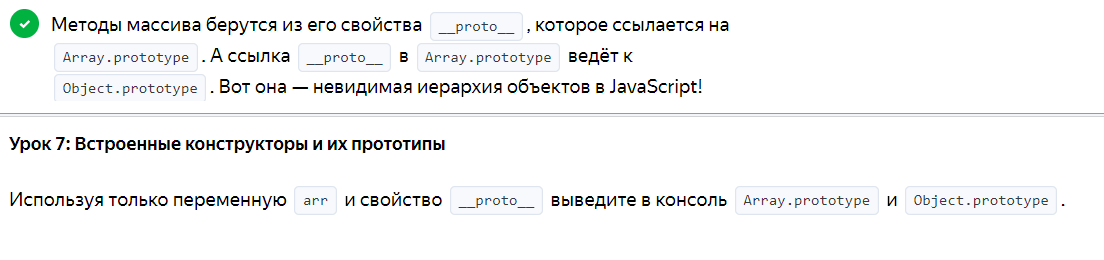
С наследованием вы уже знакомы. Скоро узнаете что скрывается за наследованием в JavaScript.

Пример:

*const arr = [1, 2, 3];*

*console.log(arr.\_\_proto\_\_); // выведите Array.prototype*

*console.log(arr.\_\_proto\_\_.\_\_proto\_\_); // выведите Object.prototype*



**Прототипное наследование**

В этом уроке мы разберём наследование в JavaScript на примере конструктора Song:

Скопировать кодJAVASCRIPT

class Song {

constructor(title, artist) {

this.title = title;

this.artist = artist;

this.isLiked = false;

}

like() {

this.isLiked = !this.isLiked;

}

}

Теперь нужно добавить на сайт новую сущность — кавер-версии песен. Кроме данных и функциональности обычных песен у них должно быть свойство originalAuthor, которое хранит ссылку на оригинальную версию трека, и метод report, чтобы можно было сообщить о нарушении авторских прав.

Мы будем переиспользовать функциональность уже готового конструктора Song. Синтаксис классов позволяет сделать это просто. Вам уже знакомо ключевое слово extends:

Скопировать кодJAVASCRIPT

*/\* объявляем конструктор SongCover,*

*который должен расширять Song \*/*

class SongCover extends Song {

}

Конструктор SongCover должен делать всё то же самое, что и конструктор Song, а также, добавлять в this новое свойство originalAuthor. Это можно сделать так:

Скопировать кодJAVASCRIPT

class SongCover extends Song {

constructor(title, artist, originalAuthor) {

this.title = title; *// эта строчка дублирует код из конструктора Song*

this.artist = artist; *// и эта дублирует*

this.isLiked = false; *// и эта тоже*

this.originalAuthor = originalAuthor;

this.isReported = false;

}

}

Первые три строчки полностью дублируют конструктор Song. Как вы уже знаете, этого дублирования помогает избежать ключевое слово super. Так в конструкторе SongCover можно вызвать конструктор Song:

Скопировать кодJAVASCRIPT

class SongCover extends Song {

constructor(title, artist, originalAuthor) {

super(title, artist);

this.originalAuthor = originalAuthor;

this.isReported = false;

}

}

Осталось добавить недостающий метод report. Это делается внутри объявления класса:

Скопировать кодJAVASCRIPT

class SongCover extends Song {

constructor(title, artist, originalAuthor) {

super(title, artist);

this.originalAuthor = originalAuthor;

this.isReported = false;

}

report() {

this.isReported = true;

}

}

Теперь у песни, созданной классом SongCover есть методы report и like:

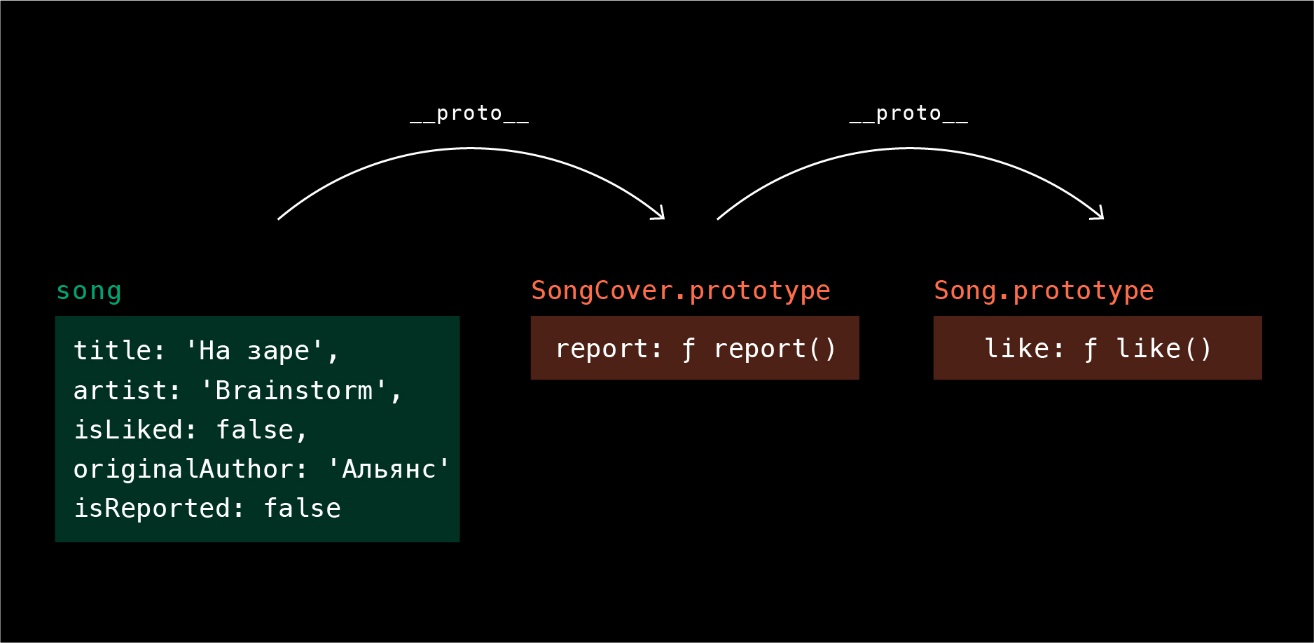
Скопировать кодJAVASCRIPT

const song = new SongCover('The Man Who Sold the World', 'Nirvana', 'David Bowie');

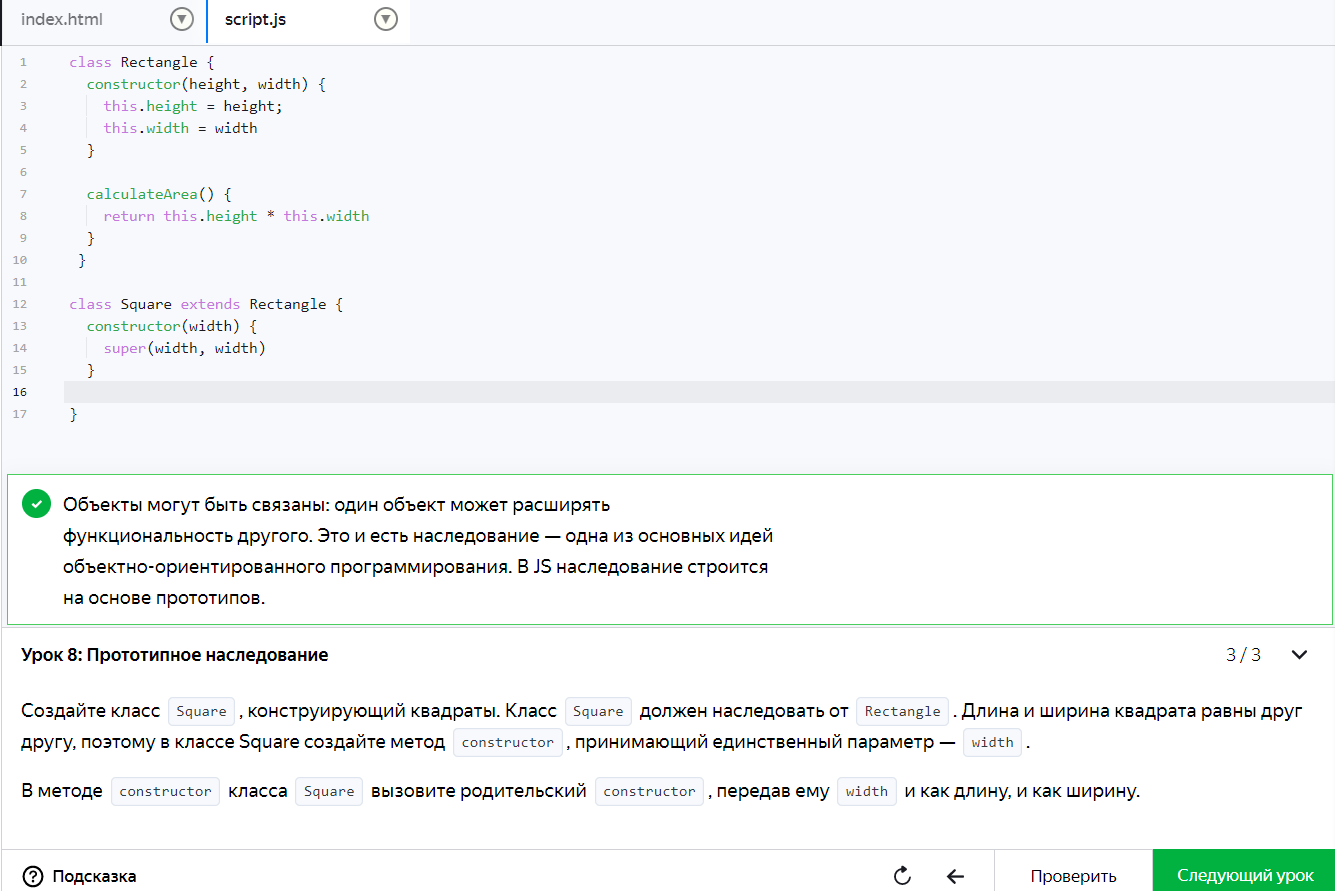
song.like();

song.report();

Всё потому, что наследование за кулисами привязало к объекту song прототип. Образовалась цепочка прототипов:



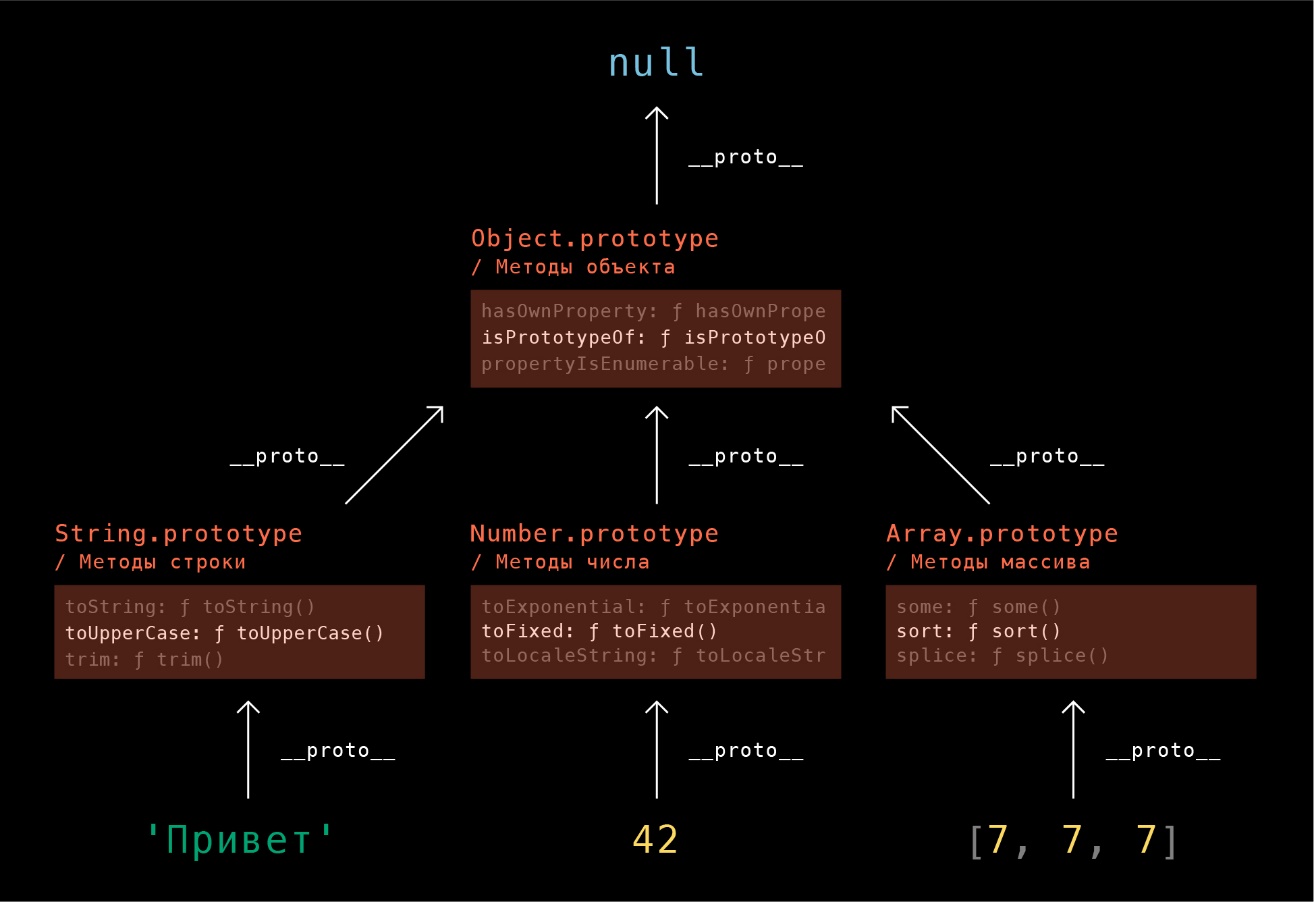
Именно поэтому наследование в JavaScript называют прототипным наследованием. При наследовании методы не копируются из одного объекта в другой, а берутся из прототипа объекта.



**Подготовка к собеседованию. Прототипы объектов. Заключение**

Одна из завес тайн JavaScript приоткрыта. Теперь вы знаете откуда у массива все эти методы, которых мы в него не добавляли.

Ещё вы узнали, что Array и Object — встроенные функции-конструкторы. Кроме них в JS есть встроенные конструкторы почти для каждого типа данных и для функций: Number, String, Boolean и Function. Все эти конструкторы связаны друг с другом:



Ну и, наконец, вы открыли для себя главный секрет классов в JS. На самом деле, класс — это удобная запись функции-конструктора с методами в прототипе. А наследование — просто ещё одна установка ссылки \_\_proto\_\_.

Может быть, все эти знания не удастся применить на практике прямо сейчас. Ведь писать код можно и не обладая ими. Но, узнавая эти концепции, вы проявляете вселенную JavaScript и веб-разработки. Это позволит вывести понимание JS на новый уровень и, как следствие, совершать меньше ошибок.

И вопросы на техническом интервью пока никто не отменял. Поэтому до встречи в следующем спринте. В нём мы снова уделим время подготовке к собеседованию.