**ОБЪЕКТНО ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ**

Основные понятия ООП

Объектно-ориентированное программирование (ООП) – это методология, которая позволяет представить приложение, в виде совокупности объектов, взаимодействующих друг с другом. В большинстве объектно-ориентированных языках программирования такие объекты создаются с помощью специальных конструкций, называемых Классами (Classes).  
Класс – это программный код, который представляет из себя шаблон или заготовку, на основе которой в последствии и будет создан объект. Класс не имеет состояния и не предполагает вызов методов, описанных в классе. Это только заготовка/схема/чертеж.  
Объект – структура, которая была создана из класса. Объект часто называют экземпляром класса. Работа программы происходит именно с объектами.  
В JavaScript классов не существует, поэтому все объекты создаются с использованием функций конструкторов. Объекты, созданные из класса (функции-конструктора) называются экземплярами класса. Таким образом, объекты используются как составные блоки для приложения. Да и все приложение, по сути, является набором объектов со своими свойствами и методами, которые взаимодействуют друг с другом.  
Следует добавить, что понятие класса появилось в синтаксисе ES6, однако на самом деле, эти классы, являются лишь удобной синтаксической конструкцией. На самом деле, все еще используется механизм прототипов, который будет рассмотрен далее.  
При использовании ООП, следует придерживаться следующих принципов:

* • инкапсуляция (encapsulation) – каждый объект отвечает за конкретную функциональность;
* • наследование (inheritance) – объекты могут наследовать функциональность других объектов;
* • полиморфизм (polymorphism) – объекты могут предоставлять одинаковый интерфейс и его использование, но внутренняя реализация этого интерфейса будет разной.

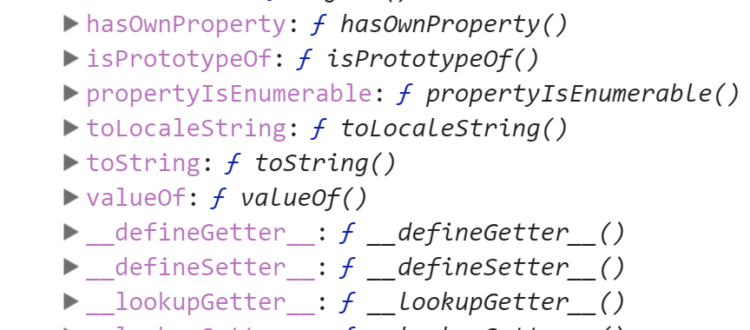
В JavaScript используют два принципа: инкапсуляция и наследование.  
Инкапсуляция позволяет изолировать всю необходимую функциональность внутри объекта, таким образом, внутренние свойства и методы будут спрятаны от остальной части приложения.  
Наследование позволяет объекту унаследовать свойства и методы из другого, родительского объекта.  
То есть, объект может инкапсулировать в себе функциональность и наследовать свойства и методы из других объектов.

Прототип готового объекта \_\_proto\_\_

У любого созданного объекта всегда присутствует ссылка на другой объект, который называется прототипом. Не имеет значения каким образом создается объект, с помощью литерала объекта или конструктора new Object(), все они наследуются от Object. Прототипом всех объектов является глобальный объект Object.

const newObject = {};

console.log(newObject);

В консоли увидим пустой объект, со свойством, которое явно в нем не определялось – \_\_proto\_\_.  
  
\_\_proto\_\_ – является ссылкой на прототип объекта.  
С помощью этой ссылки, объекту можно назначить, как прототип, любой другой объект.  
Для чего вообще нужны прототипы. Для, пока что, поверхностного понимания, можно объяснить так. Существует объект, который хранит в себе множество полезных для работы, например, с дверью, методов. И есть множество объектов, например, множество дверей, которым необходимо предоставить какую-то функциональность. Без наличия прототипов, необходимо было бы в каждом объекте-дверь, создавать методы для работы с ней. Но имея прототипы, можно, имея один объект со всеми необходимыми методами, просто унаследовать методы из него.

let functionality = {

open: function() {

console.log(`${this.room} is open.`);

}

}

let bathRoom = {

room: 'Bath room'

}

let kitchen = {

room: 'Kitchen'

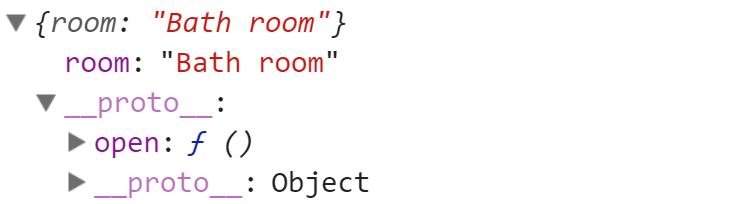
}

bathRoom.\_\_proto\_\_ = functionality;

kitchen.\_\_proto\_\_ = functionality;

bathRoom.open(); // Bath room is open.

kitchen.open(); // Kitchen is open.

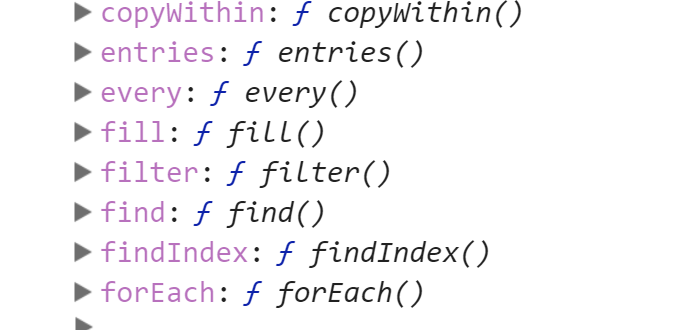
Таким образом, прототипы помогают оптимизировать использование кода и избегать его дублирования. В прототип можно помещать свойства и методы, которые могут быть многократно использованы для работы с множеством других объектов. Мы выделяем место в памяти для хранения методов и вместо повторного написания этих методов в каждом из объектов, обращаемся к месту в памяти и вызываем оттуда необходимый метод.  
Если вывести объект bathRoom в консоль, то можно увидеть, что его непосредственным прототипом является не Object, а объект, который содержит метод open и уже далее, этот объект-прототип имеет прототип Object.  
  
То есть непосредственно в самом объекте bathRoom метода open нет. Но благодаря механизму прототипов, когда мы вызовем bathRoom.open(), произойдут следующие действия:

1. поиск метода open внутри объекта bathRoom – там он не найден;
2. поиск метода open у прототипа объекта bathRoom – найден и запущен.

Рассмотрим, как прототипы используются самим JavaScript на примере встроенного объекта Array.  
Если создать новый массив с помощью конструктора new Array(), то прототипом будет не объект Object, а объект Array. Но прототипом самого объекта Array уже будет все тот же объект Object. Такая структура называется цепочкой прототипов.

const newArray = new Array();

console.log(newArray);

  
Именно в прототипе Array хранятся все методы для работы с массивами. Таким образом прототип – это место где хранятся все общедоступные свойства и методы, которые будут унаследованы при создании нового экземпляра объекта. Такие свойства и методы не копируются в каждый экземпляр объекта, они постоянно хранятся в прототипе.  
Цепочка прототипов работает следующим образом: создав экземпляр объекта и вызвав какой-либо метод на нем, интерпретатор ищет сначала этот метод внутри текущего экземпляра объекта, если его там не обнаружилось, поиск продолжается в прототипе экземпляра объекта, и этот прототип доступен по ссылке \_\_proto\_\_.  
Например, создав массив с числами, обратившись к свойству length, оно не будет найдено в самом массиве и будет искаться в прототипе массива, которым является Array. Так же, можно на массиве вызвать метод valueOf. Поиск метода сначала будет происходить в прототипе Array, но не найдя его там, поиск продолжится в прототипе Object, откуда и будет вызван.

const array = [1, 2, 3];

console.log(array.length); // 3

console.log(array.valueOf()); // [1, 2, 3]

Таким образом, если какое-то свойство или метод будут сразу найдены в самом объекте, то их поиск по цепочке прототипов не происходит. Например, у объекта Array есть метод toString и у объекта Object так же есть метод toString. При вызове этого метода на массиве, вызовется метод именно из объекта Array, а не объекта Object. Метод toString в Array переопределяет одноименный метод в Object.  
Таким образом можно организовать цепочку поиска свойств и методов. Если их нет в оригинальном объекте, то они ищутся в объекте прототипа. Это и есть суть наследования в JavaScript.

Установка прототипа для функции-конструктора

Прототипное программирование – это модель ООП которая не использует классы, а вместо классов используются прототипы.  
На практике при программировании в ООП стиле, в JavaScript, для создания объектов используются функции-конструкторы. Для установки прототипа в данном случае так же можно использовать ссылку \_\_proto\_\_, но у функции-конструктора (и у любой другой функции) есть специальное свойство prototype, с помощью которого можно установить прототип объекту.  
Кроме того, способ установки прототипа с помощью свойства prototype является полностью кроссбраузерным и поддерживается в старых версиях браузеров. Использование данного свойства стало уже обычной практикой, практически стандартом.  
В данном случае, значение ссылки \_\_proto\_\_, которая указывает на прототип объекта, берется из свойства prototype.  
Итак, каждая функция в JavaScript имеет свойство prototype, в которое присваиваются свойства и методы, которые необходимо сделать доступными для наследования. Данное свойство используется, прежде всего, для реализации наследования.

let room = {

area : 12

};

function BusinessRoom() {

this.isMeetingAvailable = true;

};

BusinessRoom.prototype = room;

const businessRoom = new BusinessRoom();

console.log(businessRoom.area); // 12

Из свойства prototype в ссылку \_\_proto\_\_ был установлен объект room как прототип для BusinessRoom.

Реализация «класса» с помощью прототипа

Как таковых классов в JavaScript нету, они реализуются с помощью функций-конструкторов и чаще называются просто прототипом или родительским объектом, но для удобства, иногда, будем использовать именно слово «класс».  
Итак, существует возможность создавать «классы» и без помощи прототипов.

function Printer(doc) {

this.document = doc;

this.print = function () {

console.log(this.document);

};

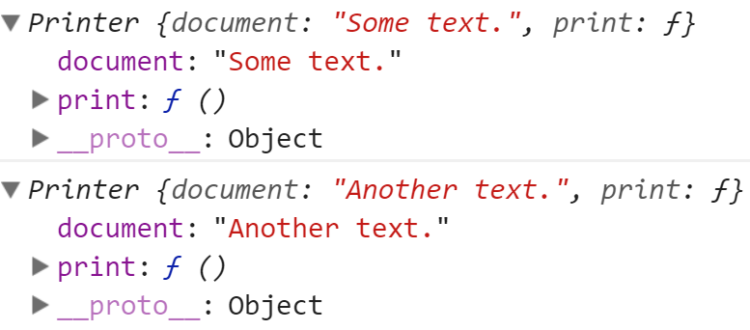
}

const newPrinter = new Printer('Some text.');

const somePrinter = new Printer('Another text.');

newPrinter.print(); // Some text.

somePrinter.print(); // Another text.

Данный код отработает без ошибок и на первый взгляд является весьма приемлемым.  
Выведем созданные на основе Printer объекты в консоль.  
  
Как видно, оба объекта содержат в себе метод print.  
Теперь реализуем то же, но с помощью prototype.

function Printer(doc) {

this.document = doc;

}

Printer.prototype.print = function () {

console.log(this.document);

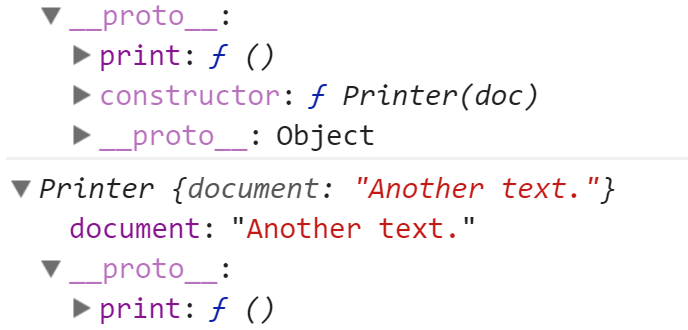
}

const newPrinter = new Printer('Some text.');

const somePrinter = new Printer('Another text.');

newPrinter.print(); // Some text.

somePrinter.print(); // Another text.

И выведем новые объекты в консоль.  
  
Как видно, теперь метод print не находится в каждом созданном объекте, а лишь в их прототипах. Прототип один, общий для обеих объектов. То есть, находясь в прототипе, как уже было указано выше, метод записывается в памяти и другие объекты просто ссылаются на это место в памяти для вызова метода.  
Создание новых экземпляров newPrinter и somePrinter, с помощью функции-конструктора, позволило унаследовать свойства и методы (в приведенном примере только один метод print) из родительского Printer благодаря тому, что метод print был добавлен в свойство prototype родительской функции-конструктора.  
Далее, на созданном экземпляре, благодаря прототипному наследованию, можно вызывать метод print, не создавая такой метод в самом экземпляре.