ООП в Javascript

Оглавление

[Синтаксис языка 0](#_Toc444164282)

[Внешние скрипты и порядок их исполнения 1](#_Toc444164283)

[Структура кода 2](#_Toc444164284)

[Комментарии 2](#_Toc444164285)

[Переменные 3](#_Toc444164286)

[Константы 3](#_Toc444164287)

[Типы данных 4](#_Toc444164288)

[Оператор typeof 5](#_Toc444164289)

[Современный стандарт, "use strict" 6](#_Toc444164290)

[Домашнее задание 6](#_Toc444164291)

[Используемая литература 7](#_Toc444164292)

# ООП

ООП – это подход к программированию, который применим, в том числе и в языке JavaScript наряду с процедурным подходом.

ООП (Объектно-ориентированное программирование) – парадигма программирования, в котором основными концепциями являются понятия классы и объекты.

Класс - это тип, описывающий устройство объектов. Понятие «класс» подразумевает некоторое поведение и способ представления. Понятие «объект» подразумевает нечто, что обладает определённым поведением и способом представления. Говорят, что объект — это экземпляр класса. Класс можно сравнить с чертежом, согласно которому создаются объекты. Обычно классы разрабатывают таким образом, чтобы их объекты соответствовали объектам предметной области.

Объект — сущность в адресном пространстве вычислительной системы, появляющаяся при создании экземпляра класса (например, после запуска результатов компиляции (и линковки) исходного кода на выполнение).

# Жизненный цикл объекта

В JavaScript почти все является объектом. Все примитивные типы за исключением null и undefined обрабатываются как объекты. Они могут быть назначены свойствам, и у них есть все характеристики, присущие объектам.

**Создание объектов**

JavaScript содержит набор встроенных объектов. Также вы можете создавать свои объекты. Начиная с JavaScript 1.2, вы можете создавать объект с помощью инициализатора объекта. Другой способ -- создать функцию-конструктор и сделать экземпляр объекта с помощью этой функции и оператора new.

Помимо создания объектов с помощью функции конструктора, вы можете создавать объекты и другим, особым способом. Фактически вы можете записать объект синтаксически, и он будет создан интерпретатором автоматически во время выполнения. Эта синтаксическая схема приведена ниже:

|  |
| --- |
| var obj = { property\_1: value\_1, // property\_# may be an identifier...  2: value\_2, // or a number...  // ...,  "property n": value\_n }; // or a string |

# здесь obj это имя нового объекта, каждое property\_i это идентификатор (имя, число или строковый литерал), и каждый value\_i это значения, назначенные property\_i. Имя obj и ссылка объекта на него необязательна; если далее вам не надо будет ссылаться на данный объект, то вам не обязательно назначать объект переменной. (Обратите внимание, что вам потребуется обернуть литерал объекта в скобки, если объект находится в месте, где ожидается инструкция, чтобы интерпретатор не перепутал его с блоком.)

# Если объект создан при помощи инициализатора объектов на высшем уровне скрипта, то JavaScript интерпретирует объект каждый раз, когда анализирует выражение, содержащее объект записанный как литерал. Плюс, если пользоваться функцией инициализатором, то он будет создается каждый раз, когда функция вызывается.

# Следующая инструкция создает объект и назначает его переменной x, когда выражение cond истинно.

|  |
| --- |
| if (cond) var x = {hi: "there"}; |

След. пример создает объект myHonda с тремя свойствами. Заметьте, что свойство engine - это также объект со своими собственными свойствами.

|  |
| --- |
| var myHonda = {  color: "red",  wheels: 4,  engine: {  cylinders: 4,  size: 2.2  }  }; |

**Использование функции конструктора**

Другой способ создать объект в два шага описан ниже:

1. Определите объект, набрав функцию-конструктор. Название такой функции, как правило, начинается с заглавной буквы.
2. Создайте экземпляр объекта с помощью ключевого слова new.

Чтобы определить тип объекта, создайте функцию для которая определяет тип объекта, его имя, свойства и методы. Наример положим вы хотите создать тип объект для cars.Вы хотите чтобы объект этого типа назывался car, и вы хотите чтобы у него были свойства make, model, и year. Чтобы сделать это, напишите след. функцию:

|  |
| --- |
| function Car(make, model, year) {  this.make = make;  this.model = model;  this.year = year;  } |

Заметьте, что используется this чтобы назначить значения (переданные как аргументы функции) свойствам объекта.

Теперь вы можете создать объект нызваемый mycar след. образом:

|  |
| --- |
| var mycar = new Car("Eagle", "Talon TSi", 1993); |

Эта инструкция создает объект типа Car с ссылкой mycar и назначает определенные значения его свойствам. Значение mycar.make строка "Eagle", mycar.year это целое число 1993, и так далее.

Вы можете создать столько объектов car сколько нужно просто вызывая new. Например,

|  |
| --- |
| var kenscar = new Car("Nissan", "300ZX", 1992);  var vpgscar = new Car("Mazda", "Miata", 1990); |

Объект может иметь свойтво которое будет другим объектом. Например, далее определяет объект типа Person след. образом:

|  |
| --- |
| function Person(name, age, sex) {  this.name = name;  this.age = age;  this.sex = sex;  } |

и затем создать два новых экземпляря объектов person как показано далее:

|  |
| --- |
| var rand = new Person("Rand McKinnon", 33, "M");  var ken = new Person("Ken Jones", 39, "M"); |

Затем, вы можете переписать определение car и включить в него свойство owner которому назначить объект person след. образом:

|  |
| --- |
| function Car(make, model, year, owner) {  this.make = make;  this.model = model;  this.year = year;  this.owner = owner;  } |

Затем, чтобы создать экземпляры новых объектов, выполните след. инструкции:

|  |
| --- |
| var car1 = new Car("Eagle", "Talon TSi", 1993, rand);  var car2 = new Car("Nissan", "300ZX", 1992, ken); |

Замечание. Вместо того чтобы передавать строку литерал или целое число когда создаются новые объекты, в выражениях выше передаются объекты rand и ken как аргумент функции. Затем если вам нужно узнать имя владельца car2, это можно узнать след. образом:

|  |
| --- |
| car2.owner.name |

Заметьте что вы всегда можете добавить свойство после создания объекта. Например, выражение

|  |
| --- |
| car1.color = "black"; |

добавляет свойство color к car1, и устанавливаего его значение равным "black." Как бы там ни было, это не влияет на любые другие объекты. Чтобы добавить новое свойство всем объектам одного типа, вы должны добавить свойство в определения типа объекта car.

**Использование метода Object.create**

Объекты также можно создавать с помощью метода Object.create. Этот метод очень удобный, так как позволяет вам указывать объект прототип для нового вашего объекта, без определения функции конструктора.

|  |
| --- |
| // Shape - superclass  function Shape() {  this.x = 0;  this.y = 0;  }  // superclass method  Shape.prototype.move = function(x, y) {  this.x += x;  this.y += y;  console.info('Shape moved.');  };  // Rectangle - subclass  function Rectangle() {  Shape.call(this); // call super constructor.  }  // subclass extends superclass  Rectangle.prototype = Object.create(Shape.prototype);  Rectangle.prototype.constructor = Rectangle;  var rect = new Rectangle();  console.log('Is rect an instance of Rectangle?', rect instanceof Rectangle);// true  console.log('Is rect an instance of Shape?', rect instanceof Shape);// true  rect.move(1, 1); // Outputs, 'Shape moved.' |

# Свойство prototype

Вы можете добавить свойство к ранее определенному типу объекта воспользовавшись свойством prototype. Это определяет свойство которое единое для всего объектов определенного типа, а не одного экземпляря этого типа объекта. След. код демонстрирует это добавляя свойство color ко всем объектам типа car,и замет присваивая значение значение свойству color объекта car1.

|  |
| --- |
| Car.prototype.color = null;  car1.color = "black"; |

При этом неважно, когда был создан объект данного типа, до добавления нового свойства или после – свойство появится у всех объектов.

# ООП

На протяжении долгого времени в программировании применялся процедурный подход. При этом программа состоит из функций, вызывающих друг друга.

Гораздо позже появилось объектно-ориентированное программирование (ООП), которое позволяет группировать функции и данные в единой сущности – «объекте».

При объектно-ориентированной разработке мы описываем происходящее на уровне объектов, которые создаются, меняют свои свойства, взаимодействуют друг с другом и (в случае браузера) со страницей, в общем, живут.

Например, «пользователь», «меню», «компонент интерфейса»… При объектно-ориентированном подходе каждый объект должен представлять собой интуитивно понятную сущность, у которой есть методы и данные.

function User(name) {

this.sayHi = function() {

alert( "Привет, я " + name );

};

}

var vasya = new User("Вася"); // создали пользователя

vasya.sayHi(); // пользователь умеет говорить "Привет"

Здесь мы видим ярко выраженную сущность – User (посетитель). Используя терминологию ООП, такие конструкторы часто называют классами, то есть можно сказать "класс User".

ООП – это наука о том, как делать правильную архитектуру. У неё есть свои принципы, например SOLID.

Один из важнейших принципов ООП – отделение внутреннего интерфейса от внешнего.

Это – обязательная практика в разработке чего угодно сложнее hello world.

Чтобы это понять, отвлечемся от разработки и переведем взгляд на объекты реального мира.

Как правило, устройства, с которыми мы имеем дело, весьма сложны. Но разделение интерфейса на внешний и внутренний позволяет использовать их без малейших проблем.

В программировании мы будем разделять методы и свойства объекта на две группы:

* Внутренний интерфейс – это свойства и методы, доступ к которым может быть осуществлен только из других методов объекта, их также называют «приватными» (есть и другие термины, встретим их далее).
* Внешний интерфейс – это свойства и методы, доступные снаружи объекта, их называют «публичными».

Приведем аналогию с кофеваркой – то, что спрятано внутри кофеварки: трубка кипятильника, нагревательный элемент, тепловой предохранитель и так далее – это её внутренний интерфейс.

Внутренний интерфейс используется для обеспечения работоспособности объекта, его детали используют друг друга. Например, трубка кипятильника подключена к нагревательному элементу.

Но снаружи кофеварка закрыта специальным кожухом, чтобы никто к ним не подобрался. Детали скрыты и недоступны. Виден лишь внешний интерфейс.

Получив объект, всё, что нужно для пользования им – это знать внешний интерфейс. О внутреннем же знать вообще не обязательно.

Это были общие слова по теории программирования.

Далее мы реализуем кофеварку на JavaScript с приватными и публичными свойствами. В кофеварке много деталей, мы конечно, не будем моделировать каждый винтик, а сосредоточимся на основных приёмах разработки.

**Шаг 1: публичное и приватное свойство**

Конструктор кофеварок будет называться CoffeeMachine.

function CoffeeMachine(power) {

this.waterAmount = 0; // количество воды в кофеварке

alert( 'Создана кофеварка мощностью: ' + power + ' ватт' );

}

// создать кофеварку

var coffeeMachine = new CoffeeMachine(100);

// залить воды

coffeeMachine.waterAmount = 200;

Локальные переменные, включая параметры конструктора, можно считать приватными свойствами.

В примере выше это power – мощность кофеварки, которая указывается при создании и далее будет использована для расчёта времени кипячения.

К локальным переменным конструктора нельзя обратиться снаружи, но они доступны внутри самого конструктора.

Свойства, записанные в this, можно считать публичными.

Здесь свойство waterAmount записано в объект, а значит – доступно для модификации снаружи. Можно доливать и выливать воду в любом количестве.

**Шаг 2: публичный и приватный методы**

Добавим публичный метод run, запускающий кофеварку, а также вспомогательные внутренние методы getBoilTime и onReady:

function CoffeeMachine(power) {

this.waterAmount = 0;

// расчёт времени для кипячения

function getBoilTime() {

return 1000; // точная формула расчета будет позже

}

// что делать по окончании процесса

function onReady() {

alert( 'Кофе готово!' );

}

this.run = function() {

// setTimeout - встроенная функция,

// она запустит onReady через getBoilTime() миллисекунд

setTimeout(onReady, getBoilTime());

};

}

var coffeeMachine = new CoffeeMachine(100);

coffeeMachine.waterAmount = 200;

coffeeMachine.run();

Приватные методы, такие как onReady, getBoilTime могут быть объявлены как вложенные функции.

В результате естественным образом получается, что доступ к ним (через замыкание) имеют только другие функции, объявленные в том же конструкторе.

**Шаг 3: константа**

Для расчёта времени на кипячение воды используется формула c\*m\*ΔT / power, где:

* c – коэффициент теплоёмкости воды, физическая константа равная 4200.
* m – масса воды, которую нужно нагреть.
* ΔT – температура, на которую нужно подогреть, будем считать, что изначально вода – комнатной температуры 20°С, то есть до 100° нужно греть на ΔT=80.
* power – мощность.

Используем её в более реалистичном варианте getBoilTime(), включающем использование приватных свойств и константу:

"use strict"

function CoffeeMachine(power) {

this.waterAmount = 0;

// физическая константа - удельная теплоёмкость воды для getBoilTime

var WATER\_HEAT\_CAPACITY = 4200;

// расчёт времени для кипячения

function getBoilTime() {

return this.waterAmount \* WATER\_HEAT\_CAPACITY \* 80 / power; // ошибка!

}

// что делать по окончании процесса

function onReady() {

alert( 'Кофе готово!' );

}

this.run = function() {

setTimeout(onReady, getBoilTime());

};

}

var coffeeMachine = new CoffeeMachine(1000);

coffeeMachine.waterAmount = 200;

coffeeMachine.run();

Удельная теплоёмкость WATER\_HEAT\_CAPACITY выделена большими буквами, так как это константа.

Внимание, при запуске кода выше в методе getBoilTime будет ошибка. Как вы думаете, почему?

**Шаг 4: доступ к объекту из внутреннего метода**

Внутренний метод вызывается так: getBoilTime(). А чему при этом равен this?… Как вы наверняка помните, в современном стандарте он будет undefined (в старом – window), из-за этого при чтении this.waterAmount возникнет ошибка!

Её можно решить, если вызвать getBoilTime с явным указанием контекста: getBoilTime.call(this):

function CoffeeMachine(power) {

this.waterAmount = 0;

var WATER\_HEAT\_CAPACITY = 4200;

function getBoilTime() {

return this.waterAmount \* WATER\_HEAT\_CAPACITY \* 80 / power;

}

function onReady() {

alert( 'Кофе готово!' );

}

this.run = function() {

setTimeout(onReady, getBoilTime.call(this));

};

}

// создаю кофеварку, мощностью 100000W чтобы кипятила быстро

var coffeeMachine = new CoffeeMachine(100000);

coffeeMachine.waterAmount = 200;

coffeeMachine.run();

Такой подход будет работать, но он не очень-то удобен. Ведь получается, что теперь везде, где мы хотим вызвать getBoilTime, нужно явно указывать контекст, т.е. писать getBoilTime.call(this).

К счастью существуют более элегантные решения.

Можно при объявлении привязать getBoilTime к объекту через bind, тогда вопрос контекста отпадёт сам собой:

function CoffeeMachine(power) {

this.waterAmount = 0;

var WATER\_HEAT\_CAPACITY = 4200;

var getBoilTime = function() {

return this.waterAmount \* WATER\_HEAT\_CAPACITY \* 80 / power;

}.bind(this);

function onReady() {

alert( 'Кофе готово!' );

}

this.run = function() {

setTimeout(onReady, getBoilTime());

};

}

var coffeeMachine = new CoffeeMachine(100000);

coffeeMachine.waterAmount = 200;

coffeeMachine.run();

Это решение будет работать, теперь функцию можно просто вызывать без call. Но объявление функции стало менее красивым.

Пожалуй, самый удобный и часто применяемый путь решения состоит в том, чтобы предварительно скопировать this во вспомогательную переменную и обращаться из внутренних функций уже к ней.

Вот так:

function CoffeeMachine(power) {

this.waterAmount = 0;

var WATER\_HEAT\_CAPACITY = 4200;

var self = this;

function getBoilTime() {

return self.waterAmount \* WATER\_HEAT\_CAPACITY \* 80 / power;

}

function onReady() {

alert( 'Кофе готово!' );

}

this.run = function() {

setTimeout(onReady, getBoilTime());

};

}

var coffeeMachine = new CoffeeMachine(100000);

coffeeMachine.waterAmount = 200;

coffeeMachine.run();

Теперь getBoilTime получает self из замыкания.

Конечно, чтобы это работало, мы не должны изменять self, а все приватные методы, которые хотят иметь доступ к текущему объекту, должны использовать внутри себя self вместо this.

Вместо self можно использовать любое другое имя переменной, например var me = this.

# Вывод

Итак, мы сделали кофеварку с публичными и приватными методами и заставили их корректно работать.

В терминологии ООП отделение и защита внутреннего интерфейса называется инкапсуляция.

Кратко перечислим бонусы, которые она даёт:

**Защита пользователей от выстрела себе в ногу**

Представьте, команда разработчиков пользуется кофеваркой. Кофеварка создана фирмой «Лучшие Кофеварки» и, в общем, работает хорошо, но с неё сняли защитный кожух и, таким образом, внутренний интерфейс стал доступен.

Все разработчики цивилизованны – и пользуются кофеваркой как обычно. Но хитрый Вася решил, что он самый умный, и подкрутил кое-что внутри кофеварки, чтобы кофе заваривался покрепче. Вася не знал, что те изменения, которые он произвёл, приведут к тому, что кофеварка испортится через два дня.

Виноват, разумеется, не только Вася, но и тот, кто снял защитный кожух с кофеварки, и тем самым позволил Васе проводить манипуляции.

В программировании – то же самое. Если пользователь объекта будет менять то, что не рассчитано на изменение снаружи – последствия могут быть непредсказуемыми.

**Удобство в поддержке**

Ситуация в программировании сложнее, чем с кофеваркой, т.к. кофеварку один раз купили и всё, а программа может улучшаться и дорабатываться.

При наличии чётко выделенного внешнего интерфейса, разработчик может свободно менять внутренние свойства и методы, без оглядки на коллег.

Гораздо легче разрабатывать, если знаешь, что ряд методов (все внутренние) можно переименовывать, менять их параметры, и вообще, переписать как угодно, так как внешний код к ним абсолютно точно не обращается.

Ближайшая аналогия в реальной жизни – это когда выходит «новая версия» кофеварки, которая работает гораздо лучше. Разработчик мог переделать всё внутри, но пользоваться ей по-прежнему просто, так как внешний интерфейс сохранён.

**Управление сложностью**

Люди обожают пользоваться вещами, которые просты с виду. А что внутри – дело десятое.

Программисты здесь не исключение.

Всегда удобно, когда детали реализации скрыты, и доступен простой, понятно документированный внешний интерфейс.

# Наследование

Наследование – это создание новых «классов» на основе существующих.

В JavaScript его можно реализовать несколькими путями, один из которых – с использованием наложения конструкторов, мы рассмотрим в этой главе.

Ранее мы обсуждали различные реализации кофеварки. Продолжим эту тему далее.

Хватит ли нам только кофеварки для удобной жизни? Вряд ли… Скорее всего, ещё понадобятся как минимум холодильник, микроволновка, а возможно и другие машины.

В реальной жизни у этих машин есть базовые правила пользования. Например, большая кнопка – включение, шнур с розеткой нужно воткнуть в питание и т.п.

Можно сказать, что «у всех машин есть общие свойства, а конкретные машины могут их дополнять».

Именно поэтому, увидев новую технику, мы уже можем что-то с ней сделать, даже не читая инструкцию.

Механизм наследования позволяет определить базовый класс Машина, в нём описать то, что свойственно всем машинам, а затем на его основе построить другие, более конкретные: Кофеварка, Холодильник и т.п.

**Наследование от Machine**

Базовый класс «машина» Machine будет реализовывать общего вида методы «включить» enable() и «выключить» disable():

function Machine() {

var enabled = false;

this.enable = function() {

enabled = true;

};

this.disable = function() {

enabled = false;

};

}

Унаследуем от него кофеварку. При этом она получит эти методы автоматически:

function CoffeeMachine(power) {

Machine.call(this); // отнаследовать

var waterAmount = 0;

this.setWaterAmount = function(amount) {

waterAmount = amount;

};

}

var coffeeMachine = new CoffeeMachine(10000);

coffeeMachine.enable();

coffeeMachine.setWaterAmount(100);

coffeeMachine.disable();

Наследование реализовано вызовом Machine.call(this) в начале конструктора CoffeeMachine.

Он вызывает функцию Machine, передавая ей в качестве контекста this текущий объект. Machine, в процессе выполнения, записывает в this различные полезные свойства и методы, в нашем случае this.enable и this.disable.

Далее конструктор CoffeeMachine продолжает выполнение и может добавить свои свойства и методы.

В результате мы получаем объект coffeeMachine, который включает в себя методы из Machine и CoffeeMachine.

В коде выше есть одна проблема.

Наследник не имеет доступа к приватным свойствам родителя.

Иначе говоря, если кофеварка захочет обратиться к enabled, то её ждёт разочарование:

function Machine() {

var enabled = false;

this.enable = function() {

enabled = true;

};

this.disable = function() {

enabled = false;

};

}

function CoffeeMachine(power) {

Machine.call(this);

this.enable();

// ошибка, переменная не определена!

alert( enabled );

}

var coffeeMachine = new CoffeeMachine(10000);

Это естественно, ведь enabled – локальная переменная функции Machine. Она находится в другой области видимости.

Чтобы наследник имел доступ к свойству, оно должно быть записано в this.

При этом, чтобы обозначить, что свойство является внутренним, его имя начинают с подчёркивания \_.

function Machine() {

this.\_enabled = false; // вместо var enabled

this.enable = function() {

this.\_enabled = true;

};

this.disable = function() {

this.\_enabled = false;

};

}

function CoffeeMachine(power) {

Machine.call(this);

this.enable();

alert( this.\_enabled ); // true

}

var coffeeMachine = new CoffeeMachine(10000);

Подчёркивание в начале свойства – общепринятый знак, что свойство является внутренним, предназначенным лишь для доступа из самого объекта и его наследников. Такие свойства называют защищёнными.

Технически, залезть в него из внешнего кода, конечно, возможно, но приличный программист так делать не будет.

**Перенос свойства в защищённые**

У CoffeeMachine есть приватное свойство power. Сейчас мы его тоже сделаем защищённым и перенесём в Machine, поскольку «мощность» свойственна всем машинам, а не только кофеварке.

function Machine(power) {

this.\_power = power; // (1)

this.\_enabled = false;

this.enable = function() {

this.\_enabled = true;

};

this.disable = function() {

this.\_enabled = false;

};

}

function CoffeeMachine(power) {

Machine.apply(this, arguments); // (2)

alert( this.\_enabled ); // false

alert( this.\_power ); // 10000

}

var coffeeMachine = new CoffeeMachine(10000);

Теперь все машины Machine имеют мощность power. Обратим внимание, что мы из параметра конструктора сразу скопировали её в объект в строке (1). Иначе она была бы недоступна из наследников.

В строке (2) мы теперь вызываем не просто Machine.call(this), а расширенный вариант: Machine.apply(this, arguments), который вызывает Machine в текущем контексте вместе с передачей текущих аргументов.

Можно было бы использовать и более простой вызов Machine.call(this, power), но использование apply гарантирует передачу всех аргументов, вдруг их количество увеличится – не надо будет переписывать.

**Переопределение методов**

Итак, мы получили класс CoffeeMachine, который наследует от Machine.

Аналогичным образом мы можем унаследовать от Machine холодильник Fridge, микроволновку MicroOven и другие классы, которые разделяют общий «машинный» функционал, то есть имеют мощность и их можно включать/выключать.

Для этого достаточно вызвать Machine в текущем контексте, а затем добавить свои методы.

// Fridge может добавить и свои аргументы,

// которые в Machine не будут использованы

function Fridge(power, temperature) {

Machine.apply(this, arguments);

// ...

}

Бывает так, что реализация конкретного метода машины в наследнике имеет свои особенности.

Можно, конечно, объявить в CoffeeMachine свой enable:

function CoffeeMachine(power, capacity) {

Machine.apply(this, arguments);

// переопределить this.enable

this.enable = function() {

/\* enable для кофеварки \*/

};

}

…Однако, как правило, мы хотим не заменить, а расширить метод родителя, добавить к нему что-то. Например, сделать так, чтобы при включении кофеварка тут же запускалась.

Для этого метод родителя предварительно копируют в переменную, и затем вызывают внутри нового enable – там, где считают нужным:

function CoffeeMachine(power) {

Machine.apply(this, arguments);

var parentEnable = this.enable; // (1)

this.enable = function() { // (2)

parentEnable.call(this); // (3)

this.run(); // (4)

}

...

}

Общая схема переопределения метода (по строкам выделенного фрагмента кода):

1. Копируем доставшийся от родителя метод this.enable в переменную, например parentEnable.
2. Заменяем this.enable на свою функцию…
3. …Которая по-прежнему реализует старый функционал через вызов parentEnable.
4. …И в дополнение к нему делает что-то своё, например запускает приготовление кофе.

Обратим внимание на строку (3).

В ней родительский метод вызывается так: parentEnable.call(this). Если бы вызов был таким: parentEnable(), то ему бы не передался текущий this и возникла бы ошибка.

Технически, можно сделать возможность вызывать его и как parentEnable(), но тогда надо гарантировать, что контекст будет правильным, например привязать его при помощи bind или при объявлении, в родителе, вообще не использовать this, а получать контекст через замыкание, вот так:

function Machine(power) {

this.\_enabled = false;

var self = this;

this.enable = function() {

// используем внешнюю переменную вместо this

self.\_enabled = true;

};

this.disable = function() {

self.\_enabled = false;

};

}

function CoffeeMachine(power) {

Machine.apply(this, arguments);

var waterAmount = 0;

this.setWaterAmount = function(amount) {

waterAmount = amount;

};

var parentEnable = this.enable;

this.enable = function() {

parentEnable(); // теперь можно вызывать как угодно, this не важен

this.run();

}

function onReady() {

alert( 'Кофе готово!' );

}

this.run = function() {

setTimeout(onReady, 1000);

};

}

var coffeeMachine = new CoffeeMachine(10000);

coffeeMachine.setWaterAmount(50);

coffeeMachine.enable();

В коде выше родительский метод parentEnable = this.enable успешно продолжает работать даже при вызове без контекста. А всё потому, что использует self внутри.

# Домашнее задание

* Разработать класс, генерирующий шахматную доску на странице. Конструктор в качестве параметра должен принимать селектор элемента в котором должна создаться доска, либо DOM Node. При этом должна быть возможность подписаться на события доски через созданный объект (не напрямую к DOM Node, а именно извне используя только объект доски), получения и установки координаты активной ячейки (шахматной координаты вида “A1”). Для генерации доски можно использовать произвольные html-тэги. Подумать какие свойства должны быть скрыты, а какие нет решение аргументировать в комментариях к коду.
* Добавить базовый класс, который мог бы генерировать таблицы (доски) любого размера и унаследовать от него разработанный класс шахматной доски.
* \* Все то же самое с помощью ООП в прототипном стиле (разобраться самостоятельно).

# Используемая литература

* [Mozilla Developer Network](https://developer.mozilla.org/)
* [MSDN](http://msdn.microsoft.com/)
* [Safari Developer Library](https://developer.apple.com/library/safari/navigation/index.html)
* [Современный учебник JavaScript](http://learn.javascript.ru)