#1

const parent = {

name: 'Stacey',

surname: 'Moore',

age: 54,

heritage: 'Irish'

};

// Пиши код ниже этой строки

const child = Object.create(parent);

// Пиши код выше этой строки

child.name = 'Jason';

child.age = 27;

**Прототип объекта и метод Object.create()**

Теория

Объекты можно организовать в цепочки так, чтобы свойство не найденное в одном объекте, автоматически искалось бы в другом. Связующим звеном выступает специальное скрытое свойство [[Prototype]], которое в консоли браузера отображается как \_\_proto\_\_.

const animal = {

legs: 4

};

const dog = Object.create(animal);

dog.name = 'Манго';

console.log(dog); // { name: 'Манго', \_\_proto\_\_: animal }

console.log(animal.isPrototypeOf(dog)); // true

Метод Object.create(obj) создаёт и возвращает новый объект, связывая его с объектом obj. Объект, на который указывает ссылка в \_\_proto\_\_, называется прототипом. В нашем примере объект animal это прототип для объекта dog. Метод isPrototypeOf() проверяет является ли объект animal прототипом для dog и возвращает true или false.

console.log(dog.hasOwnProperty('name')); // true

console.log(dog.name); // 'Манго'

console.log(dog.hasOwnProperty('legs')); // false

console.log(dog.legs); // 4

Обращение dog.name работает очевидным образом - возвращает собственное свойство name объекта dog. При обращении к dog.legs интерпретатор ищет свойство legs в объекте dog, не находит и продолжает поиск в объекте по ссылке из dog.\_\_proto\_\_, то есть, в данном случае, в объекте animal - его прототипе.

То есть прототип - это резервное хранилище свойств и методов объекта, автоматически используемое при их поиске. У объекта, который выступает прототипом может также быть свой прототип, у того свой, и так далее.

Поиск свойства выполняется до первого совпадения. Интерпретатор ищет свойство по имени в объекте, если не находит, то обращается к свойству \_\_proto\_\_, т. е. переходит по ссылке к объекту-прототипу, а затем и прототипу прототипа. Если интерпретатор доберется до конца цепочки и не найдет свойства с таким именем, то вернёт undefined.

**Задание**

Измени код так, чтобы объект parent стал прототипом для объекта в переменной сhild.

**Тесты**

* Объявлена переменная parent.
* Значение переменной parent это объект.
* Вызов parent.hasOwnProperty('surname') возвращает true.
* Вызов parent.hasOwnProperty('heritage') возвращает true.
* Объявлена переменная child.
* Значение переменной child это объект.
* Вызов child.hasOwnProperty('name') возвращает true.
* Обращение к child.name возвращает 'Jason'.
* Вызов child.hasOwnProperty('age') возвращает true.
* Обращение к child.age возвращает 27.
* Вызов child.hasOwnProperty('surname') возвращает false.
* Обращение к child.surname возвращает 'Moore'.
* Вызов child.hasOwnProperty('heritage') возвращает false.
* Обращение к child.heritage возвращает 'Irish'.
* Вызов parent.isPrototypeOf(child) возвращает true.
* Используется метод Object.create().

#2

const ancestor = {

name: 'Paul',

age: 83,

surname: 'Dawson',

heritage: 'Irish'

};

// Пиши код ниже этой строки

const parent = Object.create(ancestor);

parent.name = 'Stacey';

parent.surname = 'Moore';

parent.age = 54;

const child = Object.create(parent);

child.name = 'Jason';

child.age = 27;

// Пиши код выше этой строки

**Задача. Цепочка прототипов**

**Задание**

Измени код, построив цепочку прототипов так, чтобы объект ancestor был прототипом для parent, а тот в свою очередь был прототипом для child.

**Тесты**

* Объявлена переменная ancestor.
* Значение переменной ancestor это объект.
* Объявлена переменная parent.
* Значение переменной parent это объект.
* Объявлена переменная child.
* Значение переменной child это объект.
* Вызов parent.isPrototypeOf(child) возвращает true.
* Обращение к parent.surname возвращает 'Moore'.
* Обращение к parent.heritage возвращает 'Irish'.
* Вызов parent.hasOwnProperty('surname') возвращает true.
* Вызов parent.hasOwnProperty('heritage') возвращает false.
* Вызов ancestor.isPrototypeOf(parent) возвращает true.
* Вызов child.hasOwnProperty('surname') возвращает false.
* Обращение к child.surname возвращает 'Moore'.
* Обращение к child.heritage возвращает 'Irish'.
* Вызов child.hasOwnProperty('heritage') возвращает false.
* Обращение к ancestor.heritage возвращает 'Irish'.
* Вызов ancestor.hasOwnProperty('surname') возвращает true.
* Вызов ancestor.hasOwnProperty('heritage') возвращает true.
* Обращение к ancestor.surname возвращает 'Dawson'.
* Используется метод Object.create().

#3

function Car(brand, model, price){

this.brand = brand;

this.model = model;

this.price = price;

}

const myCar = new Car();

console.log(myCar);

**Функция-конструктор**

Теория

Синтаксис литерала объекта позволяет создать один объект. Но часто нужно создать много однотипных объектов с одинаковым набором свойств, но разными значениями, и методами для взаимодействия со свойствами. Всё это нужно сделать динамически, во время выполнения программы. Для этого используют функции-конструкторы, вызывая их при помощи специального оператора new.

function User() {

// Тело функции

}

const mango = new User();

console.log(mango); // {}

const poly = new User();

console.log(poly); // {}

Любая функция, кроме стрелочной, может быть использована как конструктор. Чтобы отличить конструктор от обычной функции, их принято называть с большой буквы, а в самом названии отражать тип создаваемого объекта (существительное).

Вызов функции с оператором new приводит к созданию нового объекта и вызову функции в контексте этого объекта. То есть this внутри функции будет ссылаться на новосозданный объект. Это позволяет добавлять каждому объекту свойства с одинаковыми именами, но разными значениями.

function User(name, email, age) {

this.name = name;

this.email = email;

}

const mango = new User('Манго', 'mango@mail.com');

console.log(mango); // { name: 'Манго', email: 'mango@mail.com' }

const poly = new User('Поли', 'poly@mail.com');

console.log(poly); // { name: 'Поли', email: 'poly@mail.com' }

**Задание**

Объяви функцию-конструктор Car которая принимает три параметра:

* brand - марка автомобиля.
* model - модель автомобиля.
* price - цена автомобиля.

Функция Car должна создавать объект с одноимёнными свойствами brand, model и price, значениями которых должны быть переданные аргументы во время её вызова с оператором new.

**Тесты**

* Объявлена функция Car(brand, model, price).
* В результате вызова new Car('Audi', 'Q3', 36000) получится объект { brand: 'Audi', model: 'Q3', price: 36000 }.
* В результате вызова new Car('BMW', 'X5', 58900) получится объект { brand: 'BMW', model: 'X5', price: 58900 }.
* В результате вызова new Car('Nissan', 'Murano', 31700) получится объект { brand: 'Nissan', model: 'Murano', price: 31700 }.

#4

function Car({brand, model, price}) {

this.brand = brand;

this.model = model;

this.price = price;

}

**Объект настроек**

Теория

Функции-конструкторы всегда принимают большое количество входных данных для свойств будущего объекта. Поэтому, к ним также можно применить паттерн «Объект настроек», передавая один объект с логично именованными свойствами, вместо несвязанного набора аргументов.

function User({ name, email }) {

this.name = name;

this.email = email;

}

const mango = new User({ name: 'Манго', email: 'mango@mail.com' });

const poly = new User({ name: 'Поли', email: 'poly@mail.com' });

**Задание**

Выполни рефакторинг функции-конструктора Car так, чтобы она принимала один параметр - объект со свойсвами brand, model и price. Деструктуризируй объект в сигнатуре (подписи) функции.

**Тесты**

* Объявлена функция Car({ brand, model, price }).
* В результате вызова new Car({ brand: 'Audi', model: 'Q3', price: 36000 }) получится объект { brand: 'Audi', model: 'Q3', price: 36000 }.
* В результате вызова new Car({ brand: 'BMW', model: 'X5', price: 58900 }) получится объект { brand: 'BMW', model: 'X5', price: 58900 }.
* В результате вызова new Car({ brand: 'Nissan', model: 'Murano', price: 31700 }) получится объект { brand: 'Nissan', model: 'Murano', price: 31700 }.

#5

function Car({ brand, model, price }) {

this.brand = brand;

this.model = model;

this.price = price;

}

Car.prototype.getPrice = function () {

return this.price;

}

Car.prototype.changePrice = function (newPrice) {

this.price = newPrice;

};

const myCar = new Car({ brand: 'Audi', model: 'Q3', price: 36000 });

**Свойство prototype**

Теория

Мы уже знаем что такое прототип объекта, свойство \_\_proto\_\_ и как происходит поиск отсутствующих свойств объекта по цепочке прототипов. Во время вызова функции-конструктора через new и создания нового объекта со свойствами, ему также устанавливается прототип.

У каждой функции, кроме стрелочных, есть свойство prototype, значение которого это объект с единственным свойством constructor, указывающим на саму функцию-конструктор.

function User() {}

console.log(User.prototype); // { constructor: User }

При вызове функции-конструктора и создании объекта через new, объект в свойстве prototype функции-конструктора будет прототипом создаваемого объекта.

const mango = new User();

console.log(User.prototype.isPrototypeOf(mango)); // true

В свойство prototype можно записывать свойства и методы, которые будут доступны всем объектам созданным этой функцией-конструктором. Методы в prototype будут вызываться объектами созданными функцией-конструктором, поэтому для доступа к свойствам вызываемого объекта в методах используется ключевое слово this.

function User({ name, email }) {

this.name = name;

this.email = email;

}

User.prototype.getEmail = function () {

return this.email;

};

User.prototype.changeEmail = function (newEmail) {

this.email = newEmail;

};

const mango = new User({ name: 'Манго', email: 'mango@mail.com' });

console.log(mango.getEmail()); // mango@mail.com

mango.changeEmail('mango@supermail.com');

console.log(mango.getEmail()); // mango@supermail.com

**Задание**

Добавь в свойство prototype функции-конструктора Car два метода:

* getPrice() - возвращает значение свойства price из объекта который его будет вызывать.
* changePrice(newPrice) - обновляет значение свойства price у объекта который его будет вызывать на newPrice.

**Тесты**

* Объявлена функция Car({ brand, model, price }).
* Вызов Car.prototype.hasOwnProperty('getPrice') возвращает true.
* Значение Car.prototype.getPrice это функция.
* Вызов Car.prototype.hasOwnProperty('changePrice') возвращает true.
* Значение Car.prototype.changePrice это функция.
* У объекта, созданного вызовом new Car({ brand: 'Audi', model: 'Q3', price: 36000 }), вызов метода getPrice() вернет число 36000.
* У объекта, созданного вызовом new Car({ brand: 'Audi', model: 'Q3', price: 36000 }), вызов метода changePrice(35000) и последующем вызове getPrice() вернет число 35000.

#6

function Storage(items){

this.items = items;

}

Storage.prototype.getItems = function(){

return this.items;

}

Storage.prototype.addItem = function(newItem){

return this.items.push(newItem);

}

Storage.prototype.removeItem = function(item){

const removeItem = this.items.indexOf(item)

return this.items.splice(removeItem, 1);

}

// Пиши код выше этой строки

const storage = new Storage(['Нанитоиды', 'Пролонгер', 'Антигравитатор']);

console.log(storage.getItems()); // ["Нанитоиды", "Пролонгер", "Антигравитатор"]

storage.addItem('Дроид');

console.log(storage.getItems()); // ["Нанитоиды", "Пролонгер", "Антигравитатор", "Дроид"]

storage.removeItem('Пролонгер');

console.log(storage.getItems()); // ["Нанитоиды", "Антигравитатор", "Дроид"]

**Задача. Хранилище**

**Задание**

С помощью Function Declaration напиши функцию-конструктор Storage, которая будет создавать объекты для управления складом товаров. Функция ожидает только один аргумент - начальный массив товаров, который записывается на создаваемый объект в свойство items.

Добавь методы на прототип:

* getItems() - возвращает массив текущих товаров в свойстве items объекта, который вызывает этот метод.
* addItem(newItem) - принимает новый товар newItem и добавляет его в массив товаров в свойстве items объекта, который вызывает этот метод.
* removeItem(item) - принимает товар item и удаляет его из массива товаров в свойстве items объекта, который вызывает этот метод.

Под комментарием мы добавили инициализацию экземпляра и вызовы методов в той последовательности, в которой твой код будут проверять тесты. Пожалуйста ничего там не меняй.

**Тесты**

* Объявлена функция Storage(items).
* Вызов Storage.prototype.hasOwnProperty('getItems') возвращает true.
* Вызов Storage.prototype.hasOwnProperty('addItem') возвращает true.
* Вызов Storage.prototype.hasOwnProperty('removeItem') возвращает true.
* В результате вызова new Storage([ 'Нанитоиды', 'Пролонгер', 'Антигравитатор' ]) значение переменной storage это объект.
* Вызов Storage.prototype.isPrototypeOf(storage) возвращает true.
* У объекта storage есть свойство items.
* Первый вызов storage.getItems(), сразу после инциализации экземпляра, возвращает массив ["Нанитоиды", "Пролонгер", "Антигравитатор"].
* Второй вызов, storage.getItems(), после вызова storage.addItem('Дроид'), возвращает массив ["Нанитоиды", "Пролонгер", "Антигравитатор", "Дроид"].
* Третий вызов storage.getItems(), после вызова storage.removeItem('Пролонгер'), возвращает массив ["Нанитоиды", "Антигравитатор", "Дроид"].

#7

function StringBuilder(baseValue) {

this.value = baseValue

}

StringBuilder.prototype.getValue = function () {

return this.value;

}

StringBuilder.prototype.padEnd = function (str) {

this.value += str;

}

StringBuilder.prototype.padStart = function (str) {

this.value = str + this.value;

}

StringBuilder.prototype.padBoth = function (str) {

this.value = str + this.value + str;

}

// Пиши код выше этой строки

const builder = new StringBuilder('.');

console.log(builder.getValue()); // '.'

builder.padStart('^');

console.log(builder.getValue()); // '^.'

builder.padEnd('^');

console.log(builder.getValue()); // '^.^'

builder.padBoth('=');

console.log(builder.getValue()); // '=^.^='

**Конструктор строк**

**Задание**

С помощью Function Declaration напиши функцию-конструктор StringBuilder, которая принимает один параметр baseValue - произвольную строку, которая записывается на создаваемый объект в свойство value.

Добавь методы на прототип:

* getValue() - возвращает текущее значение свойства value.
* padEnd(str) - получает парметр str (строку) и добавляет её в конец значения свойства value объекта, который вызывает этот метод.
* padStart(str) - получает парметр str (строку) и добавляет её в начало значения свойства value объекта, который вызывает этот метод.
* padBoth(str) - получает парметр str (строку) и добавляет её в начало и в конец значения свойства value объекта, который вызывает этот метод.

Под комментарием мы добавили инициализацию экземпляра и вызовы методов в той последовательности, в которой твой код будут проверять тесты. Пожалуйста ничего там не меняй.

**Тесты**

* Объявлена функция StringBuilder(baseValue).
* Вызов StringBuilder.prototype.hasOwnProperty('getValue') возвращает true.
* Вызов StringBuilder.prototype.hasOwnProperty('padEnd') возвращает true.
* Вызов StringBuilder.prototype.hasOwnProperty('padStart') возвращает true.
* Вызов StringBuilder.prototype.hasOwnProperty('padBoth') возвращает true.
* В результате вызова new StringBuilder('.') значение переменной builder это объект.
* Вызов StringBuilder.prototype.isPrototypeOf(builder) возвращает true.
* У объекта builder есть свойство value.
* Первый вызов builder.getValue(), сразу после инциализации экземпляра, возвращает строку '.'.
* Второй вызов builder.getValue(), после вызова builder.padStart('^'), возвращает строку '^.'.
* Третий вызов builder.getValue(), после вызова builder.padEnd('^'), возвращает строку '^.^'.
* Четвёртый вызов builder.getValue(), после вызова builder.padBoth('='), возвращает строку '=^.^='.

#8

class Car {

// Тело класса

}

**Объявление класса**

Теория

Классы это синтаксический сахар (надстройка) над функциями-конструкторами - новый удобный способ для задания конструктора вместе с прототипом.

function User() {

// Тело функции-конструктора

}

const mango = new User();

Объявление класса начинается с ключевого слова class, после которого идёт имя класса и фигурные скобки - его тело.

class User {

// Тело класса

}

const mango = new User();

Результат вызова new User() это объект, как и в функциях-конструкторах, он называется **экземпляр класса**, потому что содержит данные и поведение, описываемые классом.

**Задание**

Используя ключевое слово class объяви класс Car с пустым телом.

**Тесты**

* Объявлен класс Car с пустым телом.
* Результат вызова new Car() это пустой объект.

#9

class Car{

brand;

model;

price;

constructor({ brand, model, price }) {

// Инициализация объявленных свойств

this.brand = brand;

this.model = model;

this.price = price;

}

}

**Конструктор класса**

Теория

При вызове с опертором new, функция-конструктор добавляет свойства на создаваемый объект.

function User({ name, email }) {

this.name = name;

this.email = email;

}

Для инициализации экземпляра в классе есть метод constructor. Если он не объявлен, создаётся конструктор по умолчанию - пустая функция, которая не изменяет экземпляр.

class User {

// Необязательное объявление публичных свойств

name;

email;

// Аналог функции-конструктора

constructor({ name, email }) {

// Инициализация объявленных свойств

this.name = name;

this.email = email;

}

}

const mango = new User({ name: 'Манго', email: 'mango@mail.com' });

Свойства name и email называются **публичные свойства**, потому что они будут собственными свойствами объекта-экземпляра и к ним можно будет получить доступ обратившись через точку.

Объявлять публичные свойства в теле класса до конструктора необязательно, но это может повысить читабельность, так как код внутри конструктора может быть довольно громоздким и визуально выделить набор свойств класса будет затруднительно.

**Задание**

Выполни рефакторинг кода, заменив функцию-конструктор Car на класс с методом-конструктором, принимающим объект.

**Тесты**

* Объявлен класс Car.
* Конструктор класса принимает объект со свойствами brand, model и price.
* В результате вызова new Car({ brand: 'Audi', model: 'Q3', price: 36000 }) получится объект { brand: 'Audi', model: 'Q3', price: 36000 }.
* В результате вызова new Car({ brand: 'BMW', model: 'X5', price: 58900 }) получится объект { brand: 'BMW', model: 'X5', price: 58900 }.
* В результате вызова new Car({ brand: 'Nissan', model: 'Murano', price: 31700 }) получится объект { brand: 'Nissan', model: 'Murano', price: 31700 }.

#10

class Car {

constructor({ brand, model, price }) {

this.brand = brand;

this.model = model;

this.price = price;

}

getPrice(){

return this.price;

}

changePrice(newPrice) {

this.price = newPrice;

}

}

**Методы класса**

Теория

В функции-конструкторе объявление методов для работы со свойствами экземпляра делается явно, путём обращения к свойству prototype и добавления на него методов.

function User({ name, email }) {

this.name = name;

this.email = email;

}

User.prototype.getEmail = function () {

return this.email;

};

User.prototype.changeEmail = function (newEmail) {

this.email = newEmail;

};

В классах используется более удобный синтаксис методов класса, который за ширмой делает тоже самое - добавляет методы на свойство User.prototype. Поэтому в самом начале мы говорили что классы это просто синтаксический сахар - удобная надстройка над функциями-конструкторами.

class User {

constructor({ name, breed }) {

this.name = name;

this.breed = breed;

}

// Аналог User.prototype.getEmail

getEmail() {

return this.email;

}

// Аналог User.prototype.changeEmail

changeEmail(newEmail) {

this.email = newEmail;

}

}

**Задание**

Добавь классу Car две метода.

* getPrice() - возвращает значение свойства price из объекта который его будет вызывать.
* changePrice(newPrice) - обновляет значение свойства price у объекта который его будет вызывать на newPrice.

**Тесты**

* Объявлен класс Car.
* Конструктор класса принимает объект со свойствами brand, model и price.
* Вызов Car.prototype.hasOwnProperty('getPrice') возвращает true.
* Значение Car.prototype.getPrice это функция.
* Вызов Car.prototype.hasOwnProperty('changePrice') возвращает true.
* Значение Car.prototype.changePrice это функция.
* У объекта, созданного вызовом new Car({ brand: 'Audi', model: 'Q3', price: 36000 }), вызов метода getPrice() вернет число 36000.
* У объекта, созданного вызовом new Car({ brand: 'Audi', model: 'Q3', price: 36000 }), вызов метода changePrice(35000) и последующем вызове getPrice() вернет число 35000.

#11

class Car {

#brand;

constructor({ brand, model, price }) {

this.#brand = brand;

this.model = model;

this.price = price;

}

getBrand() {

return this.#brand;

}

changeBrand(newBrand) {

this.#brand = newBrand;

}

}

**Приватные свойства**

Теория

Инкапсуляция - это концепция позволяющая скрыть внутренние детали класса. Пользователь класса должен получать доступ только к публичному интерфейсу - набору публичных свойств и методов класса.

В классах инкапсуляция реализуется приватными свойствами, доступ к которым можно получить только внутри класса. Это одно из отличий классов и функций-конструкторов - в классах легко объявить приватные свойства.

Допустим почта пользователя должна быть недоступна из вне, то есть приватна. Добавляя к имени свойства символ # мы делаем его приватным. Объявление приватного свойства до инциализации в конструкторе - обязательно.

class User {

// Необязательное объявление публичных свойств

name;

// Обязательное объявление приватных свойств

#email;

constructor({ name, email }) {

this.name = name;

this.#email = email;

}

getEmail() {

return this.#email;

}

changeEmail(newEmail) {

this.#email = newEmail;

}

}

const mango = new User({ name: 'Манго', email: 'mango@mail.com' });

mango.changeEmail('mango@supermail.com');

console.log(mango.getEmail()); // mango@supermail.com

console.log(mango.#email); // Будет ошибка, это приватное свойство

Методы класса также могут быть приватными, то есть доступны только в теле класса. Для этого перед их именем необходимо поставить символ #.

**Задание**

Выполни рефакторинг класса Car так, чтобы свойство brand было приватным и добавь два метода для публичного интерфейса, для чтения и изменения этого свойства.

* getBrand() - возвращает значение приватного свойства brand.
* changeBrand(newBrand) - изменяет значение приватного свойства brand на newBrand.

**Тесты**

* Объявлен класс Car.
* Свойство brand в классе Car объявлено приватным.
* Конструктор класса принимает объект со свойствами brand, model и price.
* В результате вызова new Car({ brand: 'Audi', model: 'Q3', price: 36000 }) получится объект { model: 'Q3', price: 36000 }.
* В результате вызова new Car({ brand: 'BMW', model: 'X5', price: 58900 }) получится объект { model: 'X5', price: 58900 }.
* В результате вызова new Car({ brand: 'Nissan', model: 'Murano', price: 31700 }) получится объект { model: 'Murano', price: 31700 }.
* У экземпляра нет публичного свойства brand.
* Метод getBrand() возвращает значение приватного свойства brand.
* Метод changeBrand('Honda') изменяет значение приватного свойства brand на 'Honda'.

#12

class Storage {

#items;

constructor(items){

this.#items = items;

}

getItems(){

return this.#items;

}

addItem(newItem){

return this.#items.push(newItem);

}

removeItem(item){

const itemIndex = this.#items.indexOf(item);

this.#items.splice(itemIndex, 1);

}

}

// Пиши код выше этой строки

const storage = new Storage(["Нанитоиды", "Пролонгер", "Антигравитатор"]);

console.log(storage.getItems()); // ["Нанитоиды", "Пролонгер", "Антигравитатор"]

storage.addItem("Дроид");

console.log(storage.getItems()); // ["Нанитоиды", "Пролонгер", "Антигравитатор", "Дроид"]

storage.removeItem("Пролонгер");

console.log(storage.getItems()); // ["Нанитоиды", "Антигравитатор", "Дроид"]

**Задача. Хранилище 2.0**

**Задание**

Выполни рефакторинг заменив функцию-конструктор Storage на класс с методами. Сделай так, чтобы свойство items было приватным.

Под комментарием мы добавили инициализацию экземпляра и вызовы методов в той последовательности, в которой твой код будут проверять тесты. Пожалуйста ничего там не меняй.

**Тесты**

* Объявлен класс Storage.
* Свойство items в классе Storage объявлено приватным.
* Конструктор класса принимает свойство items.
* Вызов Storage.prototype.hasOwnProperty('getItems') возвращает true.
* Вызов Storage.prototype.hasOwnProperty('addItem') возвращает true.
* Вызов Storage.prototype.hasOwnProperty('removeItem') возвращает true.
* В результате вызова new Storage([ 'Нанитоиды', 'Пролонгер', 'Антигравитатор' ]) значение переменной storage это объект.
* Вызов Storage.prototype.isPrototypeOf(storage) возвращает true.
* У объекта storage нет свойства items.
* Первый вызов storage.getItems(), сразу после инциализации экземпляра, возвращает массив ["Нанитоиды", "Пролонгер", "Антигравитатор"].
* Второй вызов, storage.getItems(), после вызова storage.addItem('Дроид'), возвращает массив ["Нанитоиды", "Пролонгер", "Антигравитатор", "Дроид"].
* Третий вызов storage.getItems(), после вызова storage.removeItem('Пролонгер'), возвращает массив ["Нанитоиды", "Антигравитатор", "Дроид"].

#13

class StringBuilder {

#value

constructor(value){

this.#value = value;

}

getValue(){

return this.#value;

}

padEnd(str) {

this.#value += str;

}

padStart(str) {

this.#value = str + this.#value;

};

padBoth(str) {

this.padStart(str);

this.padEnd(str);

};

}

// Пиши код выше этой строки

const builder = new StringBuilder('.');

console.log(builder.getValue()); // '.'

builder.padStart('^');

console.log(builder.getValue()); // '^.'

builder.padEnd('^');

console.log(builder.getValue()); // '^.^'

builder.padBoth('=');

console.log(builder.getValue()); // '=^.^='

**Задача. Конструктор строк 2.0**

**Задание**

Выполни рефакторинг заменив функцию-конструктор StringBuilder на класс с методами. Сделай так, чтобы свойство value было приватным.

Под комментарием мы добавили инициализацию экземпляра и вызовы методов в той последовательности, в которой твой код будут проверять тесты. Пожалуйста ничего там не меняй.

**Тесты**

* Объявлен класс StringBuilder.
* Свойство value в классе StringBuilder объявлено приватным.
* Вызов StringBuilder.prototype.hasOwnProperty('getValue') возвращает true.
* Вызов StringBuilder.prototype.hasOwnProperty('padEnd') возвращает true.
* Вызов StringBuilder.prototype.hasOwnProperty('padStart') возвращает true.
* Вызов StringBuilder.prototype.hasOwnProperty('padBoth') возвращает true.
* В результате вызова new StringBuilder('.') значение переменной builder это объект.
* Вызов StringBuilder.prototype.isPrototypeOf(builder) возвращает true.
* У объекта builder нет свойства value.
* Первый вызов builder.getValue(), сразу после инциализации экземпляра, возвращает строку '.'.
* Второй вызов builder.getValue(), после вызова builder.padStart('^'), возвращает строку '^.'.
* Третий вызов builder.getValue(), после вызова builder.padEnd('^'), возвращает строку '^.^'.
* Четвёртый вызов builder.getValue(), после вызова builder.padBoth('='), возвращает строку '=^.^='.

#14

class Car {

#model;

#price;

#brand;

constructor({ brand, model, price }) {

this.#brand = brand;

this.#model = model;

this.#price = price;

}

get brand() {

return this.#brand;

}

set brand(newBrand) {

this.#brand = newBrand;

}

get model() {

return this.#model;

}

set model(newModel) {

this.#model = newModel;

}

get price() {

return this.#price;

}

set price(newPrice) {

this.#price = newPrice;

}

}

**Геттеры и сеттеры**

Теория

Геттеры и сеттеры - это более краткий синтаксис объявления методов для взаимодействия со свойствами. Геттер и сеттер имитируют обычное публичное свойство класса, но позволяют изменять другие свойства более удобным способом. Геттер выполняется при попытке получить значение свойства, а сеттер - при попытке его изменить.

Геттеры и сеттеры хорошо использовать для простых операций чтения и изменения значения свойств, особенно приватных, как их публичный интерфейс. Для работы со свойством которое хранит массив или объект они не подойдут.

class User {

name;

#email;

constructor({ name, email }) {

this.name = name;

this.#email = email;

}

// Геттер email

get email() {

return this.#email;

}

// Сеттер email

set email(newEmail) {

this.#email = newEmail;

}

}

Мы объявили геттер и сеттер email поставив перед именем свойства ключевые слова get и set. Внутри этих методов мы или возвращаем значение приватного свойства #email или изменяем его значение. Геттер и сеттер идут в паре и должны называться одинаково.

const mango = new User({ name: 'Манго', email: 'mango@mail.com' });

console.log(mango.email); // mango@mail.com

mango.email = 'mango@supermail.com';

console.log(mango.email); // mango@supermail.com

При обращении к mango.email вызывается геттер get email() {...} и выполняется его код. При попытке записи mango.email = 'mango@supermail.com' вызывается сеттер set email(newEmail) {...} и строка 'mango@supermail.com' будет значением параметра newEmail.

Плюс в том, что это методы, а значит при записи можно выполнить дополнительный код, например с какими-то проверками, в отличии от выполнениях этой же операции напрямую со свойством.

set email(newEmail) {

if(newEmail === '') {

console.log('Ошибка! Почта не может быть пустой строкой!');

return;

}

this.#email = newEmail;

}

**Задание**

Выполни рефакторинг класса Car. Сделай свойства model и price приватными, также как #brand. Стандартизируй публичный интерфейс класса заменив уже объявленные методы на геттеры и сеттеры brand, model и price для взаимодействия с приватными свойствами.

**Тесты**

* Объявлен класс Car.
* В классе Car объявлено приватное свойство brand.
* В классе Car объявлено приватное свойство model.
* В классе Car объявлено приватное свойство price.
* Конструктор класса принимает объект со свойствами brand, model и price.
* В классе Car объявлен геттер brand.
* В классе Car объявлен сеттер brand.
* В классе Car объявлен геттер model.
* В классе Car объявлен сеттер model.
* В классе Car объявлен геттер price.
* В классе Car объявлен сеттер price.

#15  
class Car {

// Пиши код ниже этой строки

static MAX\_PRICE = 50000;

#price;

constructor({ price }) {

this.#price = price;

}

get price() {

return this.#price;

}

set price(newPrice) {

if (newPrice <= Car.MAX\_PRICE){

this.#price = newPrice;

}

}

// Пиши код выше этой строки

}

const audi = new Car({price: 35000});

console.log(audi.price); // 35000

audi.price = 49000;

console.log(audi.price); // 49000

audi.price = 51000;

console.log(audi.price); // 49000

**Статические свойства**

Теория

Кроме публичных и приватных свойств будущего экземпляра, в классе можно объявить его собственные свойства, доступные только классу, но не его экземплярам - статические свойства (static). Они полезны для хранения информации относящейся к самому классу.

Добавим классу пользователя приватное свойство type - его тип, определяющий набор прав, например администратор, редактор, просто пользователь и т п. Возможные типы пользователей будем хранить как статическое свойство TYPES - объект со свойствами.

Статические свойства объявляются в теле класса. Перед именем свойства добавляется ключевое слово static.

class User {

// Объявление и инициализация статического свойства

static TYPES = {

ADMIN: 'admin',

EDITOR: 'editor',

};

#email;

#type;

constructor({ email, type }) {

this.#email = email;

this.#type = type;

}

get type() {

return this.#type;

}

set type(newType) {

if (User.TYPES[newType] === undefined) {

console.log('Ошибка! Такого типа пользователя не существет');

return;

}

this.#type = newType;

}

}

const mango = new User({

email: 'mango@mail.com',

type: User.TYPES.ADMIN,

});

console.log(mango.TYPES); // undefined

console.log(User.TYPES); // { ADMIN: 'admin', EDITOR: 'editor' }

console.log(mango.type); // admin

mango.type = User.TYPES.EDITOR;

console.log(mango.type); // editor

Статические свойства также могут быть приватные, то есть доступные только внутри класса. Для этого имя свойства должно начинаться с символа #, также как приватные свойства. Обращение к приватному статическому свойству вне тела класса вызовет ошибку.

**Задание**

Выполни рефакторинг класса Car. Добавь публичное статическое свойство MAX\_PRICE со значением 50000 - максимально допустимая цена автомобиля.

Добавь сеттеру price проверку передаваемого значения параметра newPrice. Если оно больше чем MAX\_PRICE, сеттер ничего не делает, а если меньше или равно, то перезаписывает цену автомобиля.

**Тесты**

* Объявлен класс Car.
* У класса Car есть статическое свойство MAX\_PRICE.
* Значение статического свойства MAX\_PRICE это число 50000.
* У экземпляра нет свойства MAX\_PRICE.
* В классе Car объявлен геттер price.
* В классе Car объявлен сеттер price.
* У экземпляра класса Car вызов сеттера price, со значением аргумента меньше чем значение MAX\_PRICE, изменяет свойство #price.
* У экземпляра класса Car вызов сеттера price, со значением аргумента больше чем значение MAX\_PRICE, не изменяет свойство #price.

#16

class Car {

static #MAX\_PRICE = 50000;

// Пиши код ниже этой строки

static checkPrice(price) {

if (price > Car.#MAX\_PRICE){

return "Внимание! Цена превышает допустимую."

}

return "Всё хорошо, цена в порядке."

}

// Пиши код выше этой строки

constructor({ price }) {

this.price = price;

}

}

const audi = new Car({ price: 36000 });

const bmw = new Car({ price: 64000 });

console.log(Car.checkPrice(audi.price)); // Всё хорошо, цена в порядке.

console.log(Car.checkPrice(bmw.price)); // Внимание! Цена превышает допустимую.

**Статические методы**

Теория

В классе можно объявить не только методы будущего экземпляра, но и методы доступные только классу - статические методы, которые могут быть как публичные, так и приватные. Синтаксис объявления аналогичен статическим свойствам, за исключением того, что значением будет метод.

class User {

static #takenEmails = [];

static isEmailTaken(email) {

return User.#takenEmails.includes(email);

}

#email;

constructor({ email }) {

this.#email = email;

User.#takenEmails.push(email);

}

}

const mango = new User({ email: 'mango@mail.com' });

console.log(

User.isEmailTaken('poly@mail.com')

); // false

console.log(

User.isEmailTaken('mango@mail.com')

); // true

Особенность статических методов в том, что во время их вызова ключевое слово this ссылается на сам класс. Это значит, что статический метод может получить доступ к статическим свойствам класса, но не к свойствам экземпляра. Логично, потому что статические методы вызывает сам класс, а не его экземпляры.

**Задание**

Добавь классу Car публичный статический метод checkPrice(price), принимающий цену автомобиля. Метод должен сравнить значения параметра price и приватного статического свойства MAX\_PRICE.

* Если цена автомобиля превышает максимальную, метод должен вернуть строку 'Внимание! Цена превышает допустимую.'.
* В противном случае метод должен вернуть строку 'Всё хорошо, цена в порядке.'.

Под объявлением класса мы добавили инициализацию экземпляра и вызовы методов, чтобы показать как будет использоваться метод checkPrice(price).

**Тесты**

* Объявлен класс Car.
* У класса Car есть статический метод checkPrice(price).
* Вызов Car.checkPrice(36000) возвращает строку 'Всё хорошо, цена в порядке.'.
* Вызов Car.checkPrice(18000) возвращает строку 'Всё хорошо, цена в порядке.'.
* Вызов Car.checkPrice(64000) возвращает строку 'Внимание! Цена превышает допустимую.'.
* Вызов Car.checkPrice(57000) возвращает строку 'Внимание! Цена превышает допустимую.'.

#17

class User {

email;

constructor(email) {

this.email = email;

}

get email() {

return this.email;

}

set email(newEmail) {

this.email = newEmail;

}

}

// Пиши код ниже этой строки

class Admin extends User {

static AccessLevel = { BASIC: 'basic', SUPERUSER: 'superuser' };

}

**Наследование классов**

Теория

Ключевое слово extends позволяет реализовать наследование классов, когда один класс (дочерний, производный) наследует свойства и методы другого класса (родителя). В выражении class Child extends Parent дочерний класс Child наследует (расширяет) от родительского класса Parent.

Это означает, что мы можем объявить базовый класс, который хранит общие характеристики и методы для группы производных классов, которые наследуют свойства и методы родителя, но также добавляют свои уникальные.

Например, в приложении есть пользователи разных типов - администратор, писатель статей, контент менеджер и т. п. У каждого типа пользователя есть набор общих характеристик, например почта и пароль, но также есть и уникальные.

Сделав независимые классы для каждого типа пользователя мы получим дублирование общих свойств и методов, и при необходимости изменить, например, название свойства, придётся проходить по всем классам, это неудобно и трудозатратно.

Вместо этого можно сделать общий класс User, который будет хранить набор общих свойств и методов, после чего сделать классы для каждого типа пользователя, которые наследуют этот набор от класса User. При необходимости изменить что-то общее, достаточно будет поменять только код класса User.

class User {

email;

constructor(email) {

this.email = email;

}

get email() {

return this.email;

}

set email(newEmail) {

this.email = newEmail;

}

}

class ContentWriter extends User {

// Тело класса ContentWriter

}

const writer = new ContentWriter('mango@mail.com');

console.log(writer); // { email: 'mango@mail.com' }

console.log(writer.email); // 'mango@mail.com'

Класс ContentWriter наследует от класса User его конструктор, геттер и сеттер email, а также одноимённое публичное свойство. Важно помнить, что приватные свойства и методы класса-родителя не наследуются классом-ребёнком.

**Задание**

В приложении нужен администратор с возможностью добавлять почты пользователей в чёрный список.

* Объяви класс Admin, который наследует от класса User.
* Добавь классу Admin публичное статическое свойство AccessLevel (уровень доступа), значение которого это объект { BASIC: 'basic', SUPERUSER: 'superuser' }.

**Тесты**

* Объявлен класс Admin.
* Класс Admin наследует от класса User.
* У класса Admin есть публичное статическое свойство AccessLevel.
* Обращение к Admin.AccessLevel.BASIC возвращает строку 'basic'.
* Обращение к Admin.AccessLevel.SUPERUSER возвращает строку 'superuser'.

#18

class User {

email;

constructor(email) {

this.email = email;

}

get email() {

return this.email;

}

set email(newEmail) {

this.email = newEmail;

}

}

class Admin extends User {

// Пиши код ниже этой строки

accessLevel;

constructor({ email, accessLevel }) {

super(email);

this.accessLevel = accessLevel;

};

static AccessLevel = {

BASIC: 'basic',

SUPERUSER: 'superuser'

};

// Пиши код выше этой строки

}

const mango = new Admin({

email: 'mango@mail.com',

accessLevel: Admin.AccessLevel.SUPERUSER

});

console.log(mango.email); // mango@mail.com

console.log(mango.accessLevel); // superuser

**Конструктор дочернего класса**

Теория

Первым делом в конструкторе дочернего класса необходимо вызвать специальную функцию super(аргументы) - это псевдоним конструктора родительского класса. В противном случае, при попытке обратиться к this в конструкторе дочернего клаcса, будет ошибка. При вызове конструктора класса родителя передаём необходимые ему аргументы для инициализации свойств.

class User {

email;

constructor(email) {

this.email = email;

}

get email() {

return this.email;

}

set email(newEmail) {

this.email = newEmail;

}

}

class ContentWriter extends User {

posts;

constructor({ email, posts }) {

// Вызов конструктора родительского класса User

super(email);

this.posts = posts;

}

}

const writer = new ContentWriter({ email: 'mango@mail.com', posts: [] });

console.log(writer); // { email: 'mango@mail.com', posts: [] }

console.log(writer.email); // 'mango@mail.com'

**Задание**

Добавь классу Admin метод constructor, который принимает один параметр - объект настроек с двумя свойствами email и accessLevel. Добавь классу Admin публичное свойство accessLevel, значение которого будет передаваться при вызове конструктора.

Чтобы показать как будет использоваться класс Admin, мы добавили инициализацию экземпляра под объявлением класса.

**Тесты**

* Объявлен класс Admin.
* Класс Admin наследует от класса User.
* У класса Admin есть публичное статическое свойство AccessLevel.
* У класса Admin есть метод constructor с параметром в виде объекта {email, accessLevel}.
* У класса Admin в конструкторе для свойства email используется обращение к конструктору родительского класса.
* Обращение к Admin.AccessLevel.BASIC возвращает строку 'basic'.
* Обращение к Admin.AccessLevel.SUPERUSER возвращает строку 'superuser'.
* У класса Admin есть публичное свойство accessLevel.

#19

class User {

email;

constructor(email) {

this.email = email;

}

get email() {

return this.email;

}

set email(newEmail) {

this.email = newEmail;

}

}

class Admin extends User {

// Пиши код ниже этой строки

blacklistedEmails = [];

static AccessLevel = {

BASIC: 'basic',

SUPERUSER: 'superuser'

};

accessLevel;

constructor({ email, accessLevel }) {

super(email);

this.accessLevel = accessLevel;

}

blacklist(email){

this.blacklistedEmails.push(email);

}

isBlacklisted(email){

if (this.blacklistedEmails.includes(email)){

return true;

}

return false;

}

// Пиши код выше этой строки

}

const mango = new Admin({

email: 'mango@mail.com',

accessLevel: Admin.AccessLevel.SUPERUSER

});

console.log(mango.email); // mango@mail.com

console.log(mango.accessLevel); // superuser

mango.blacklist('poly@mail.com');

console.log(mango.blacklistedEmails); // 'poly@mail.com'

console.log(mango.isBlacklisted('mango@mail.com')); // false

console.log(mango.isBlacklisted('poly@mail.com')); // true

**Методы дочернего класса**

Теория

В дочернем классе можно объявлять методы которые будут доступны только его экземплярам.

// Представим что выше есть объявление класса User

class ContentWriter extends User {

posts;

constructor({ email, posts }) {

super(email);

this.posts = posts;

}

addPost(post) {

this.posts.push(post);

}

}

const writer = new ContentWriter({ email: 'mango@mail.com', posts: [] });

console.log(writer); // { email: 'mango@mail.com', posts: [] }

console.log(writer.email); // 'mango@mail.com'

writer.addPost('post-1');

console.log(writer.posts); // ['post-1']

**Задание**

Добавь классу Admin следующие свойства и методы.

* Публичное свойство blacklistedEmails для хранения чёрного списка почтовых адресов пользователей. Значение по умолчанию это пустой массив.
* Публичный метод blacklist(email) для добавления почты в чёрный список. Метод должен добавлять значение параметра email в массив хранящийся в свойстве blacklistedEmails.
* Публичный метод isBlacklisted(email) для проверки почты в чёрном списке. Метод должен проверять наличие значения параметра email в массиве хранящемся в свойстве blacklistedEmails и возвращать true или false.

После объявления класса мы добавили инициализацию экземпляра и вызовы методов в той последовательности, в которой твой код будут проверять тесты. Пожалуйста ничего там не меняй.

**Тесты**

* Объявлен класс Admin.
* Класс Admin наследует от класса User.
* У класса Admin есть публичное свойство blacklistedEmails.
* У класса Admin есть публичный метод blacklist.
* У класса Admin есть публичный метод isBlacklisted.
* После вызова mango.blacklist('poly@mail.com') значение свойства blacklistedEmails это массив [ 'poly@mail.com' ].