**Задача 2 / 10**

**Колбэк-функции**

Теория

**Функция обратного вызова (callback, колбэк)** - это функция, которая передаётся другой функции в качестве аргумента и та, в свою очередь, вызывает переданную функцию.

**Функция высшего порядка(higher order function)** - функция, принимающая в качестве параметров другие функции или возвращающая функцию как результат.

// Колбэк-функция

function greet(name) {

consle.log(`Добро пожаловать ${name}.`);

}

// Функция высшего порядка

function registerGuest(name, callback) {

console.log(`Регистрируем гостя ${name}.`);

callback(name);

}

registerGuest('Манго', greet);

Мы передали ссылку на функцию greet как аргумент, поэтому она будет присвоена в параметр callback и вызвана внутри функции registerGuest через круглые скобки. Имя параметра для колбэка может быть произвольным, главное помнить, что значением будет функция.

**Задание**

Дополни функцию makeMessage так, чтобы она ожидала вторым параметром (параметр callback) колбэк-функцию и возвращала ее вызов. Функция deliverPizza или makePizza будет передаваться как колбэк и ожидать аргументом имя готовой доставляемой пиццы.

**Тесты**

* Объявлена функция deliverPizza.
* Объявлена функция makePizza.
* Объявлена функция makeMessage.
* Функция makeMessage принимает два параметра, названые согласно задания, pizzaName и callback.
* Вызов makeMessage('Роял гранд', makePizza) возвращает строку 'Пицца Роял гранд готовится, ожидайте...'.
* Вызов makeMessage('Ультрасыр', deliverPizza) возвращает строку 'Доставляем пиццу Ультрасыр.'.

function deliverPizza(pizzaName) {

return `Доставляем пиццу ${pizzaName}.`;

}

function makePizza(pizzaName) {

return `Пицца ${pizzaName} готовится, ожидайте...`;

}

// Пиши код ниже этой строки

function makeMessage(pizzaName, callback) {

return callback(pizzaName);

}

#4

**Несколько колбэков**

Теория

Функция может принимать произвольное количество колбэков. Например, представим что мы пишем логику принятия звонков для телефона. Программа должна включить автоответчик если абонент недоступен, или соединить звонок в противном случае. Доступность абонента будем имитировать генератором случайного числа, чтобы между разными вызовами функции можно было получить различные результаты.

function processCall(recipient) {

// Имитируем доступность абонента случайным числом

const isRecipientAvailable = Math.random() > 0.5;

if (!isRecipientAvailable) {

console.log(`Абонент ${recipient} недоступен, оставьте сообщение.`);

// Логика активации автоответчика

} else {

console.log(`Соединяем с ${recipient}, ожидайте...`);

// Логика принятия звонка

}

}

processCall('Манго');

Проблема такого подхода в том, что функция processCall делает слишком много и привязывает проверку доступности абонента к двум заранее определённым действиям. Что если в будущем вместо автоответчика нужно будет оставлять голограмму?

Мы могли бы написать функцию так, чтобы она возвращала какое-то значение и потом по результату её выполнения делать проверки и выполнять нужный код. Но проверки не относятся к внешнему коду и будут его засорять.

Выполним рефакторинг функции так, чтобы она принимала два колбэка onAvailable и onNotAvailable, и вызывала их по условию.

function processCall(recipient, onAvailable, onNotAvailable) {

// Имитируем доступеность абонента случайным числом

const isRecipientAvailable = Math.random() > 0.5;

if (!isRecipientAvailable) {

onNotAvailable(recipient);

return;

}

onAvailable(recipient);

}

function takeCall(name) {

console.log(`Соединяем с ${name}, ожидайте...`);

// Логика принятия звонка

}

function activateAnsweringMachine(name) {

console.log(`Абонент ${name} недоступен, оставьте сообщение.`);

// Логика активации автоответчика

}

function leaveHoloMessage(name) {

console.log(`Абонент ${name} недоступен, записываем голограмму.`);

// Логика записи голограммы

}

processCall('Манго', takeCall, activateAnsweringMachine);

processCall('Поли', takeCall, leaveHoloMessage);

Колбэки применяются для обработки действий пользователя на странице, при обработке запросов на сервер, выполнения заранее неизвестных функций и т. п. В этом и заключается их суть - это функции предназначенные для отложенного выполнения.

**Задание**

Необходимо написать логику обработки заказа пиццы. Выполни рефакторинг метода order так, чтобы он принимал вторым и третим параметрами два колбэка onSuccess и onError.

* Если в свойстве pizzas нет пиццы с названием из параметра pizzaName, метод order должен возвращать результат вызова колбэка onError, передавая ему аргументом строку 'В ассортименте нет пиццы с названием <имя пиццы>.'
* Если в свойстве pizzas есть пицца с названием из параметра pizzaName, метод order должен возвращать результат вызова колбэка onSuccess, передавая ему аргументом имя заказанной пиццы.

После объявления объекта pizzaPalace мы добавили колбэки и вызовы методов. Пожалуйста ничего там не меняй.

**Тесты**

* Метод order объявляет три параметра.
* Вызов pizzaPalace.order('Аль Копчино', makePizza, onOrderError) возвращает 'Ваш заказ принят. Готовим пиццу Аль Копчино.'.
* Вызов pizzaPalace.order('Четыре нарезона', makePizza, onOrderError) возвращает 'Ваш заказ принят. Готовим пиццу Четыре нарезона.'.
* Вызов pizzaPalace.order('Биг майк', makePizza, onOrderError) возвращает 'Ошибка! В ассортименте нет пиццы с названием Биг майк.'.
* Вызов pizzaPalace.order('Венская', makePizza, onOrderError) возвращает 'Ошибка! В ассортименте нет пиццы с названием Венская.'.

const pizzaPalace = {

pizzas: ['Ультрасыр', 'Аль Копчино', 'Четыре нарезона'],

order(pizzaName, onSuccess,onError) {

if(this.pizzas.includes(pizzaName)){

return onSuccess(pizzaName);

}

return onError(`В ассортименте нет пиццы с названием ${pizzaName}.`);

},

};

// Пиши код выше этой строки

// Колбэк для onSuccess

function makePizza(pizzaName) {

return `Ваш заказ принят. Готовим пиццу ${pizzaName}.`;

}

// Колбэк для onError

function onOrderError(error) {

return `Ошибка! ${error}`;

}

// Вызовы метода с колбэками

pizzaPalace.order('Аль Копчино', makePizza, onOrderError);

pizzaPalace.order('Четыре нарезона', makePizza, onOrderError);

pizzaPalace.order('Биг майк', makePizza, onOrderError);

pizzaPalace.order('Венская', makePizza, onOrderError);

#5

**Контекст вызова функции**

Теория

Внутри функций есть зарезервированное ключевое слово this. Во время исполнения функции, в this записывается ссылка на объект, в контексте которого она была вызвана. Таким образом, в теле функции мы можем получить доступ к свойствам и методам этого объекта.

const bookShelf = {

authors: ['Бернард Корнуэлл', 'Роберт Шекли'],

getAuthors() {

return this.authors;

},

addAuthor(authorName) {

this.authors.push(authorName);

},

};

console.log(bookShelf.getAuthors()); // ['Бернард Корнуэлл', 'Роберт Шекли']

bookShelf.addAuthor('Ли Танит');

console.log(bookShelf.getAuthors()); // ['Бернард Корнуэлл', 'Роберт Шекли', 'Ли Танит']

Методы getAuthors и addAuthor это функции, которые вызываются в контексте объекта bookShelf. Во время их выполнения в this записывается ссылка на объект bookShelf и мы можем обратиться к его свойствам и методам.

**Задание**

Выполни рефакторинг методов объекта pizzaPalace, расставив отсутствующие this в местах обращения к свойствам и методам объекта.

**Тесты**

* Вызов pizzaPalace.order('Аль Копчино') возвращает строку 'Заказ принят, готовим пиццу «Аль Копчино».'.
* Вызов pizzaPalace.order('Четыре нарезона') возвращает строку 'Заказ принят, готовим пиццу «Четыре нарезона».'.
* Вызов pizzaPalace.order('Биг майк') возвращает строку 'В ассортименте нет пиццы с названием «Биг майк».'.
* Вызов pizzaPalace.order('Венская') возвращает строку 'В ассортименте нет пиццы с названием «Венская».'.
* Метод checkPizza объекта pizzaPalace использует this.
* Метод order объекта pizzaPalace использует this.

const pizzaPalace = {

pizzas: ['Ультрасыр', 'Аль Копчино', 'Четыре нарезона'],

// Пиши код ниже этой строки

checkPizza(pizzaName) {

return this.pizzas.includes(pizzaName);

},

order(pizzaName) {

const isPizzaAvailable = this.checkPizza(pizzaName);

if (!isPizzaAvailable) {

return `В ассортименте нет пиццы с названием «${pizzaName}».`;

}

return `Заказ принят, готовим пиццу «${pizzaName}».`;

},

// Пиши код выше этой строки

};

#6

**Задача. Аккаунт пользователя**

**Задание**

Перед увольнением разработчик сломал исходный код управления аккаунтами пользователей нашего сервиса доставки еды. Выполни рефакторинг методов объекта customer, расставив отсутствующие this при обращении к свойствам объекта.

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

**Тесты**

* Объявлена переменная customer.
* Значение переменной customer это объект со свойствами и методами.
* Вызов customer.getDiscount() возвращает текущее значение свойства discount.
* Вызов customer.setDiscount(0.15) обновляет значение свойства discount.
* Вызов customer.getBalance() возвращает текущее значение свойства balance.
* Вызов customer.getOrders() возвращает текущее значение свойства orders.
* Вызов customer.addOrder(5000, 'Steak') добавляет 'Steak' в массив значений свойства orders и обновляет баланс.
* Метод getBalance объекта customer использует this.
* Метод getDiscount объекта customer использует this.
* Метод setDiscount объекта customer использует this.
* Метод getOrders объекта customer использует this.
* Метод addOrder объекта customer использует this.

const customer = {

username: 'Mango',

balance: 24000,

discount: 0.1,

orders: ['Burger', 'Pizza', 'Salad'],

// Пиши код ниже этой строки

getBalance() {

return this.balance;

},

getDiscount() {

return this.discount;

},

setDiscount(value) {

this.discount = value;

},

getOrders() {

return this.orders;

},

addOrder(cost, order) {

this.balance -= cost - cost \* this.discount;

this.orders.push(order);

},

// Пиши код выше этой строки

};

customer.setDiscount(0.15);

console.log(customer.getDiscount()); // 0.15

customer.addOrder(5000, 'Steak');

console.log(customer.getBalance()); // 19750

console.log(customer.getOrders()); // ['Burger', 'Pizza', 'Salad', 'Steak']

#7

**Метод call**

Теория

Бывают ситуации когда функцию нужно вызвать в контексте какого-то объекта, при этом функция не является его методом. Для этого у функций есть методы call и apply.

foo.call(obj, arg1, arg2, ...)

Метод call вызовет функцию foo так, что в this будет ссылка на объект obj, а также передаст аргументы arg1, arg2 и т. д.

function greetGuest(greeting) {

console.log(`${greeting}, ${this.username}.`);

}

const mango = { username: "Манго" };

const poly = { username: "Поли" };

greetGuest.call(mango, "Добро пожаловать");

// Добро пожаловать, Манго.

greetGuest.call(poly, "С приездом");

// С приездом, Поли.

**Задание**

Сервису приготовления и доставки еды требуется функция генерации сообщений о статусе заказа.

Дополни функцию composeMessage(position) так, чтобы она возвращала строку в формате 'Готовим <блюдо> для <почта>. Ваш заказ <позиция>-й в очереди.' Позиция это значение параметра position - позиция элемента в массиве (на единицу больше чем индекс).

* Не объявляй дополнительные параметры функции composeMessage(position).
* Используй call для вызова функции в контексте одного объекта-заказа.
* Используй this в теле функции для доступа к свойствам объекта-заказа в контексте которого она была вызывана.

Дополни код так, чтобы в переменной messages получился массив сообщений о статусе заказов из массива orders с помощью цикла for.

**Тесты**

* Объявлена переменная orders.
* Значение переменной orders это исходный массив объектов-заказов.
* Объявлена функция composeMessage(position).
* У функции composeMessage объявлен один параметр position.
* В теле функции composeMessage используется this.
* Функция composeMessage вызывается методом call внутри цикла for.
* Объявлена переменная messages.
* Значение переменной messages это массив ['Готовим Burger для solomon@topmail.ua. Ваш заказ 1-й в очереди.', 'Готовим Pizza для artemis@coldmail.net. Ваш заказ 2-й в очереди.', 'Готовим Taco для jacob@mail.com. Ваш заказ 3-й в очереди.'].

const orders = [

{ email: 'solomon@topmail.ua', dish: 'Burger' },

{ email: 'artemis@coldmail.net', dish: 'Pizza' },

{ email: 'jacob@mail.com', dish: 'Taco' },

];

// Пиши код ниже этой строки

function composeMessage(position) {

return `Готовим ${this.dish} для ${this.email}. Ваш заказ ${position}-й в очереди.`;

}

const messages = [];

for (let i = 0; i < orders.length; i += 1){

let result = composeMessage.call(orders[i], i + 1);

messages.push(result)}

console.log(messages)

#8

**Метод apply**

Теория

Метод apply это аналог метода call за исключением того, что синтаксис передачи аргументов требует не перечисление, а массив, даже если аргумент всего один.

foo.call(obj, arg1, arg2, ...)

foo.apply(obj, [arg1, arg2, ...])

Метод apply вызовет функцию foo так, что в this будет ссылка на объект obj, а также передаст элементы массива как отдельные аргументы arg1, arg2 и т. д. На практике, в основном используется метод call.

function greetGuest(greeting) {

console.log(`${greeting}, ${this.username}.`);

}

const mango = { username: "Манго" };

const poly = { username: "Поли" };

greetGuest.apply(mango, ["Добро пожаловать"]); // Добро пожаловать, Манго.

greetGuest.apply(poly, ["С приездом"]); // С приездом, Поли.

**Задание**

Выполни рефакторинг кода так, чтобы функция composeMessage(position) вызывалась методом apply.

**Тесты**

* Объявлена переменная orders.
* Значение переменной orders это исходный массив объектов-заказов.
* Объявлена функция composeMessage(position).
* У функции composeMessage объявлен один параметр position.
* В теле функции composeMessage используется this.
* Функция composeMessage вызывается методом apply.
* Объявлена переменная messages.
* Значение переменной messages это массив ['Готовим Burger для solomon@topmail.ua. Ваш заказ 1-й в очереди.', 'Готовим Pizza для artemis@coldmail.net. Ваш заказ 2-й в очереди.', 'Готовим Taco для jacob@mail.com. Ваш заказ 3-й в очереди.'].

const orders = [

{ email: "solomon@topmail.ua", dish: "Burger" },

{ email: "artemis@coldmail.net", dish: "Pizza" },

{ email: "jacob@mail.com", dish: "Taco" },

];

// Пиши код ниже этой строки

function composeMessage(position) {

return `Готовим ${this.dish} для ${this.email}. Ваш заказ ${position}-й в очереди.`;

}

const messages = [];

for (let i = 0; i < orders.length; i++) {

const msg = composeMessage.apply(orders[i], [i + 1]);

messages.push(msg);

}

#9

**Метод bind**

Теория

Методы call и apply вызывают функцию «на месте», то есть сразу. Но в случае колбэк-функций, когда необходимо не сразу вызвать функцию, а передать ссылку на неё, причём с привязанным контекстом, используется метод bind.

foo.bind(obj, arg1, arg2, ...)

Метод bind создаёт и возвращает копию функции foo с привязанным контекстом obj и аргументами arg1, arg2 и т. д. Получается копия функции которую можно передать куда угодно и вызвать когда угодно.

function greet(clientName) {

return `${clientName}, добро пожаловать в «${this.service}».`;

}

const steam = { service: 'Steam' };

const steamGreeter = greet.bind(steam);

steamGreeter('Манго'); // "Манго, добро пожаловать в «Steam»."

const gmail = { service: 'Gmail' };

const gmailGreeter = greet.bind(gmail);

gmailGreeter('Поли'); // "Поли, добро пожаловать в «Gmail»."

**Задание**

Функция composeMessage(customerName) создаёт приветственные сообщения для ресторанов. Дополни код так, чтобы в переменных pizzaPalaceComposer и burgerShackComposer были её копии с привязанным контекстом к соответствующим объектам.

**Тесты**

* Объявлена переменная pizzaPalace.
* Значение переменной pizzaPalace это исходный объект.
* Объявлена переменная burgerShack.
* Значение переменной burgerShack это исходный объект.
* Объявлена функция composeMessage(customerName).
* Объявлена переменная pizzaPalaceComposer.
* Значение переменной pizzaPalaceComposer это копия функции composeMessage с контекстом привязанным к объекту pizzaPalace.
* Объявлена переменная pizzaPalaceMessage.
* Значение переменной pizzaPalaceMessage это строка 'Манго, всегда рады вас видеть в «Pizza Palace».'.
* Объявлена переменная burgerShackComposer.
* Значение переменной burgerShackComposer это копия функции composeMessage с контекстом привязанным к объекту burgerShack.
* Объявлена переменная burgerShackMessage.
* Значение переменной burgerShackMessage это строка 'Поли, всегда рады вас видеть в «Burger Shack».'.

const pizzaPalace = {

company: 'Pizza Palace',

};

const burgerShack = {

company: 'Burger Shack',

};

function composeMessage(customerName) {

return `${customerName}, всегда рады вас видеть в «${this.company}».`;

}

// Пиши код ниже этой строки

const pizzaPalaceComposer = composeMessage.bind(pizzaPalace);

const pizzaPalaceMessage = pizzaPalaceComposer(`Манго`);

const burgerShackComposer = composeMessage.bind(burgerShack);;

const burgerShackMessage = burgerShackComposer('Поли');

#10

**Метод bind и методы объекта**

Теория

При передаче методов объекта как колбэк-функций, контекст не сохраняется. Колбэк это ссылка на метод, которая присваивается как значение параметра, вызываемого без объекта.

const customer = {

firstName: 'Jacob',

lastName: 'Mercer',

getFullName() {

return `${this.firstName} ${this.lastName}`;

},

};

function makeMessage(callback) {

// callback() это вызов метода getFullName без объекта

console.log(`Обрабатываем заявку от ${callback()}.`);

}

makeMessage(customer.getFullName); // Будет ошибка при вызове функции

В строгом режиме, значение this в методе getFullName, при вызове как колбэк-функции callback(), будет undefined. При обращении к свойствам firstName и lastName будет ошибка, так как undefined это не объект.

Метод bind используется для привязки контекста при передаче методов объекта как колбэк-функций. Передадим колбэком не оригинальный метод getFullName, а его копию с привязанным контекстом к объекту customer.

// ❌ Было

makeMessage(customer.getFullName); // Будет ошибка при вызове функции

// ✅ Стало

makeMessage(customer.getFullName.bind(customer)); // Обрабатываем заявку от Jacob Mercer.

**Задание**

Сервису рассылки электронной почты необходимо добавить логирование действий для сбора статистики. Функция logAndInvokeAction(email, action) ожидает почту и действие которое нужно выполнить - ссылку на метод объекта service. Сбор статистики симулируется логированием строки. Разберись и дополни код так, чтобы он работал верно.

**Тесты**

* Объявлена переменная service.
* Значение переменной service это оригинальный объект.
* Объявлена функция logAndInvokeAction(email, action).
* Переменной firstInvoke присвоена строка 'Почта kiwi@mail.uk добавлена в рассылку.'.
* Первый вызов logAndInvokeAction с почтой kiwi@mail.uk и методом service.subscribe привязан к объекту service.
* Переменной secondInvoke присвоена строка 'Почта poly@hotmail.de удалена из рассылки.'.
* Второй вызов logAndInvokeAction с почтой poly@hotmail.de и методом service.unsubscribe привязан к объекту service.

const service = {

mailingList: ['mango@mail.com', 'poly@hotmail.de', 'ajax@jmail.net'],

subscribe(email) {

this.mailingList.push(email);

return `Почта ${email} добавлена в рассылку.`;

},

unsubscribe(email) {

this.mailingList = this.mailingList.filter((e) => e !== email);

return `Почта ${email} удалена из рассылки.`;

},

};

function logAndInvokeAction(email, action) {

console.log(`Выполняем действие с ${email}.`);

return action(email);

}

const firstInvoke = logAndInvokeAction('kiwi@mail.uk', service.subscribe.bind(service));

console.log(firstInvoke);

// Почта kiwi@mail.uk добавлена в рассылку.

//console.log(service.mailingList.bind(service));

/\* ['mango@mail.com',

'poly@hotmail.de',

'ajax@jmail.net',

'kiwi@mail.uk']\*/

const secondInvoke = logAndInvokeAction('poly@hotmail.de', service.unsubscribe.bind(service));

console.log(secondInvoke);

// Почта poly@hotmail.de удалена из рассылки.

//console.log(service); // ['mango@mail.com', 'ajax@jmail.net', 'kiwi@mail.uk']