**Функции под микроскопом. Введение**

Вы уже много знаете о функциях из предыдущих курсов. Например, [здесь целая тема про них](https://praktikum.yandex.ru/trainer/web/lesson/11848dbd-22f3-4521-b378-960ccbae34be). В проектных работах вы написали уже не одну функцию и, кажется, поняли, что это и как они работают. Но в программировании всегда можно взглянуть на что-то глубже.

Функции — не исключение. В этой теме мы расскажем о неочевидных особенностях функций. Именно на таких особенностях часто строятся задачи технических собеседований.

Перед тем, как начать тему, восстановите в памяти такие понятия: область видимости, затенение, всплытие, функциональные выражения. Дальше они могут раскрыться необычным образом.

# Области видимости изнутри

Во введении мы просили вспомнить, что такое область видимости. Теперь разберёмся, как это работает изнутри — то есть как движок JavaScript создаёт области видимости.

Представьте такой файл script.js:

Скопировать кодJAVASCRIPT

*/\* script.js \*/*

const message = 'Hello, world, one more time!'; *// англ. «Привет, мир, ещё раз!»*

console.log(message);

Пока мы не создали функций в этом файле, в нём только одна область видимости — глобальная. Переменная message — глобальная переменная.

Если открыть script.js в браузере, движок создаст специальный объект LexicalEnvironment (англ. «лексическая среда»). Для краткости назовём его global. Все переменные (в нашем случае только message) будут добавлены в этот объект. Если обратиться к переменной message, движок будет искать её в объекте global.

Добавим в файл script.js функцию:

Скопировать кодJAVASCRIPT

const message = 'Hello, world, one more time!';

function viewAlert(statement) {

const someVariable = 'Какая-то переменная';

alert(someVariable);

alert(statement);

alert(message);

}

Мы объявили функцию viewAlert. Теперь к ней можно обращаться по имени. Это работает, потому что движок добавил её в объект global.

Разберёмся, где хранятся переменные, объявленные внутри viewAlert. Пока нигде, потому что код функции не исполняется, пока она не вызвана. Вызовем её:

Скопировать кодJAVASCRIPT

const message = 'Hello, world, one more time!';

function viewAlert(statement) {

const someVariable = 'Какая-то переменная';

alert(someVariable);

alert(statement);

alert(message);

}

viewAlert('Hello, world!');

В момент вызова функции viewAlert движок начнёт исполнять её код. Локальные переменные этой функции нужно где-то хранить, поэтому движок создаст новый объект LexicalEnvironment для неё. Назовём его LexicalEnvironmentViewAlert, сокращённо — LEVA. Параметр statement и переменная someVariable окажутся в объекте LEVA.

Если обратиться к переменным внутри функции, движок будет искать их в LEVA. Перейдём к вызовам alert:

Скопировать кодJAVASCRIPT

const message = 'Hello, world, one more time!';

function viewAlert(statement) { *// англ. statement, «высказывание»*

const someVariable = 'Какая-то переменная';

alert(someVariable); *// someVariable найдена в LEVA*

alert(statement); *// statement найдена в LEVA*

alert(message); *// хм, message в LEVA нет*

}

viewAlert('Hello, world!');

someVariable и statement найдены в объекте LEVA, но message там нет — она находится в global, который был создан раньше.

Если бы движок прекращал поиск переменных в этот момент, мы бы увидели сообщение об ошибке: «переменная message не найдена». Но, как вы знаете из прошлых курсов, у движка на этот случай есть план — если переменная не найдена внутри, он будет искать её снаружи, то есть в объекте global.

Но как движок понял, что продолжить поиск нужно именно в объекте global? Мы подобрались к главному секрету области видимости. Когда мы объявили функцию viewAlert, ей было присвоено секретное свойство [[Environment]] (англ. «окружение»):

Скопировать кодJSX

*/\* при объявлении viewAlert движок делает так \*/*

viewAlert.[[Environment]] = global

Это свойство никогда не меняется и, благодаря ему, функция viewAlert навсегда запоминает область видимости, в которой она была создана.

Если внутри viewAlert объявить функцию insideViewAlert, она тоже навсегда запомнит место, где была создана:

Скопировать кодJAVASCRIPT

const message = 'Hello, world, one more time!';

function viewAlert(statement) {

const someVariable = 'Какая-то переменная';

alert(someVariable);

alert(statement);

alert(message);

function insideViewAlert() {

*// ...*

}

}

viewAlert('Hello, world!');

Потому что для функции insideViewAlert движок тоже создаст свойство [[Environment]]:

Скопировать кодJAVASCRIPT

*/\* при объявлении insideViewAlert запомнит место её создания \*/*

insideViewAlert.[[Environment]] = LEVA

Даже если в файле 1000 вложенных функций, и из самой глубокой функции мы обращаемся к переменной из глобальной области видимости, движок найдёт её. Ведь каждая из функций хранит место, где она была создана, и движок смотрит в каждое из этих мест по очереди.

**Важно**

К объекту LexicalEnvironment и свойству [[Environment]] нельзя обратиться из кода. Движок делает это сам, а от разработчика они скрыты.

## Резюме

При объявлении функции она навсегда запоминает место, где была создана благодаря секретному свойству [[Environment]]. С момента своего создания функция никогда не расстаётся с этим свойством — она помнит место, где родилась.

# Замыкание

Один из самых распространённых вопросов на собеседовании на должность JavaScript-разработчика: «Что такое замыкание?».

О замыкании ходят легенды, а понимание замыканий сравнивают с прозрением. В этом уроке мы попытаемся обрести это прозрение. На самом деле вы уже много раз пользовались замыканиями, просто не знали об этом.

## Вспомним про область видимости

Мы уже про неё рассказывали. Напомним: если обратиться к идентификатору внутри функции, движок сначала будет искать его внутри этой функции. Если не найдёт, пойдёт искать снаружи. При этом если одна область видимости вложена в другую, движок будет переходить от самой вложенной до глобальной, как по ступенькам.

Скопировать кодJAVASCRIPT

const a = 1;

function callMe() {

const b = 2;

function callMeToo() {

console.log(a); *// a найдётся в глобальной области*

console.log(b); *// b найдётся в области видимости функции callMe*

}

callMeToo();

}

callMe();

*/\* В консоль выведется:*

*1*

*2*

*\*/*

Движок руководствуется правилом: «Если переменной нет внутри, поищу снаружи».

## Функцию можно вызвать не там, где она была создана

Раньше вы могли вызвать функцию только в той же области видимости, в какой она объявлена, или «более внутренней»:

Скопировать кодJAVASCRIPT

function abc() {

console.log('Hello');

function xyz() {

*/\* Это область видимости функции xyz.*

*Отсюда мы можем обратиться к функции abc. \*/*

abc();

}

*/\* Это область видимости функции abc.*

*Отсюда можем обратиться к xyz —*

*она в области видимости функции abc. \*/*

xyz();

}

*/\* А дальше будет ошибка — в глобальной*

*области видимости нет функции с именем xyz. \*/*

xyz();

Но на самом деле функцию можно вызвать и из другой области видимости. Вот как это делается.

Функция может вернуть другую функцию:

Скопировать кодJAVASCRIPT

function callMe() {

const internet = 'Internet';

function callMeToo() {

console.log(internet);

}

return callMeToo;

}

Теперь если вызвать callMe и записать её результат в переменную, получим функцию callMeToo во внешней области видимости.

Скопировать кодJAVASCRIPT

function callMe() {

const internet = 'Internet';

function callMeToo() {

console.log(internet);

}

return callMeToo;

}

const newCallMeToo = callMe();

console.log(newCallMeToo);

*/\**

*ƒ callMeToo() {*

*console.log(internet);*

*}*

*\*/*

Теперь у нас есть функция callMeToo в глобальной области видимости — она записана в переменную newCallMeToo. Посмотрите на код этой функции: она обращается к переменной internet, объявленной внутри callMe.

Получается такая ситуация: мы получили функцию callMeToo во внешней области видимости. Эта функция обращается к переменной internet, которой в этой области видимости нет. Вызовем эту функцию и посмотрим, что будет:

Скопировать кодJAVASCRIPT

function callMe() {

const internet = 'Internet';

function callMeToo() {

console.log(internet);

}

return callMeToo;

}

const newCallMeToo = callMe();

newCallMeToo(); *// "Internet"*

У функции newCallMeToo всё равно есть доступ к переменной internet. И он у неё будет, даже если удалить функцию callMe вовсе:

Скопировать кодJAVASCRIPT

let callMe = function () {

const internet = 'Internet';

function callMeToo() {

console.log(internet);

}

return callMeToo;

}

const newCallMeToo = callMe();

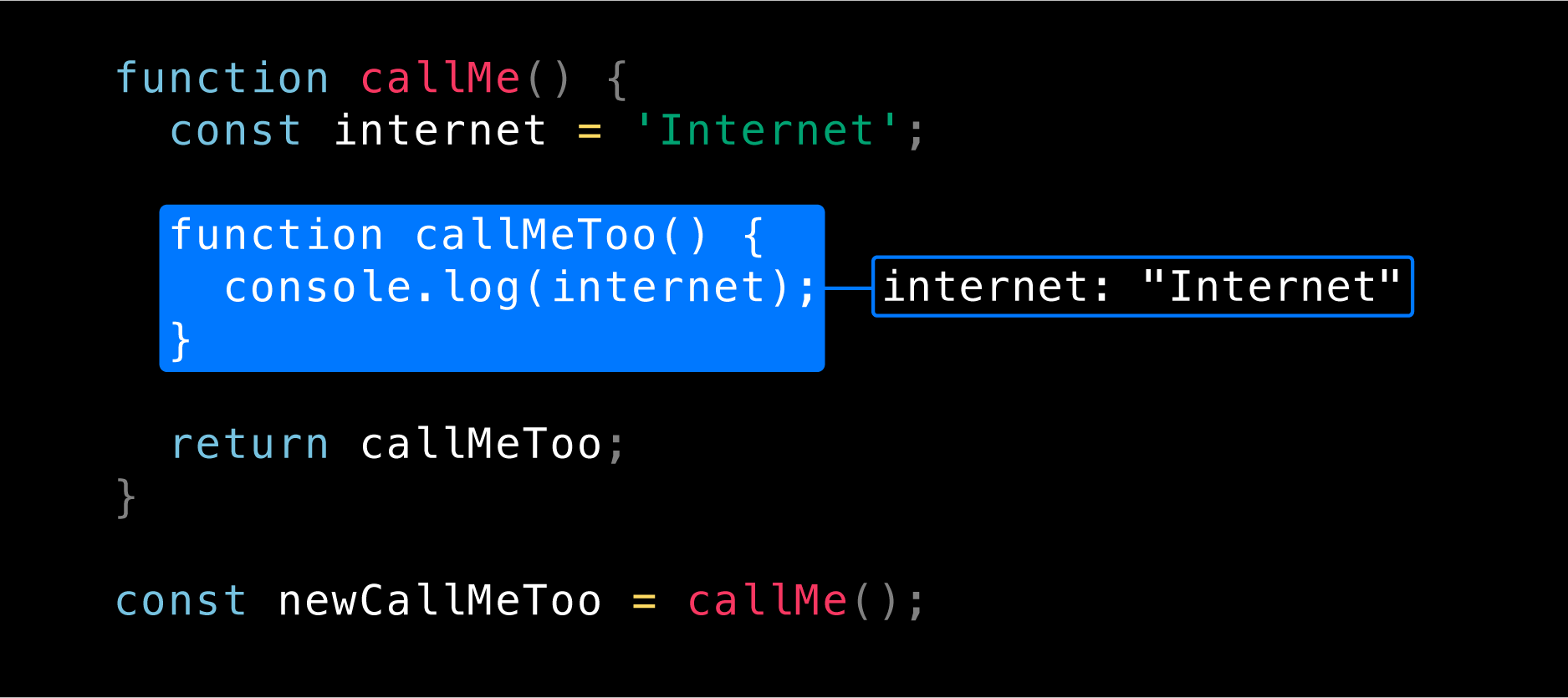
callMe = null; *// убиваем функцию callMe, её больше нет*

newCallMeToo(); *// "Internet"*

Дело в том, что функция callMeToo запомнила свою область видимости, когда мы её объявили — то есть момент вызова callMe. И даже если вызвать её из другого места, она всё равно будет помнить, какие идентификаторы существовали с ней в одной области видимости при объявлении. Это и есть замыкание.

Замыкание — это способность функции запоминать область видимости, в которой она была создана, и иметь доступ к ней даже при вызове вне этой области видимости.

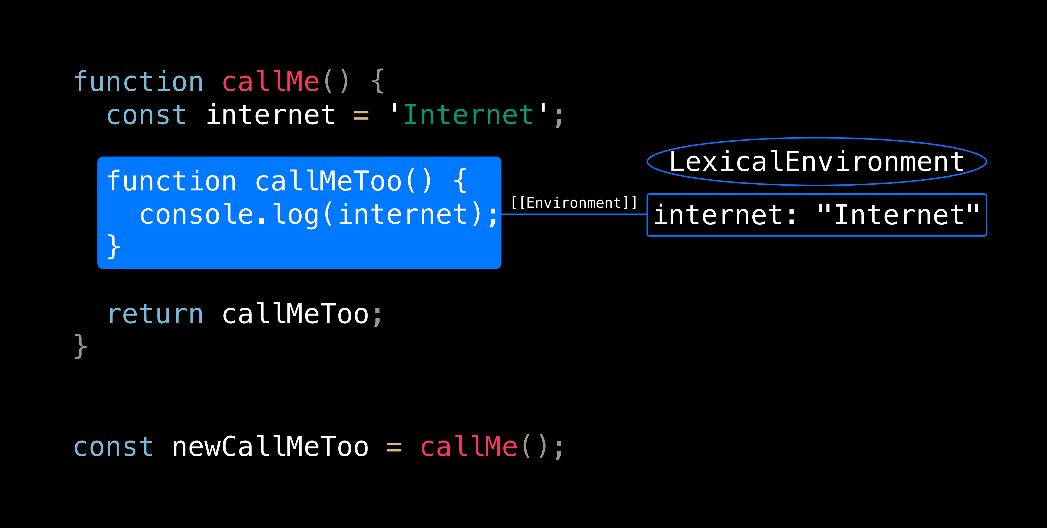
Можно сравнить замыкание с портфелем, который собирает функция при объявлении, а затем берёт его с собой в любое место, где бы она ни была. Функция кладёт в этот портфель все переменные области видимости, в которой она была объявлена.



## Замыкания изнутри

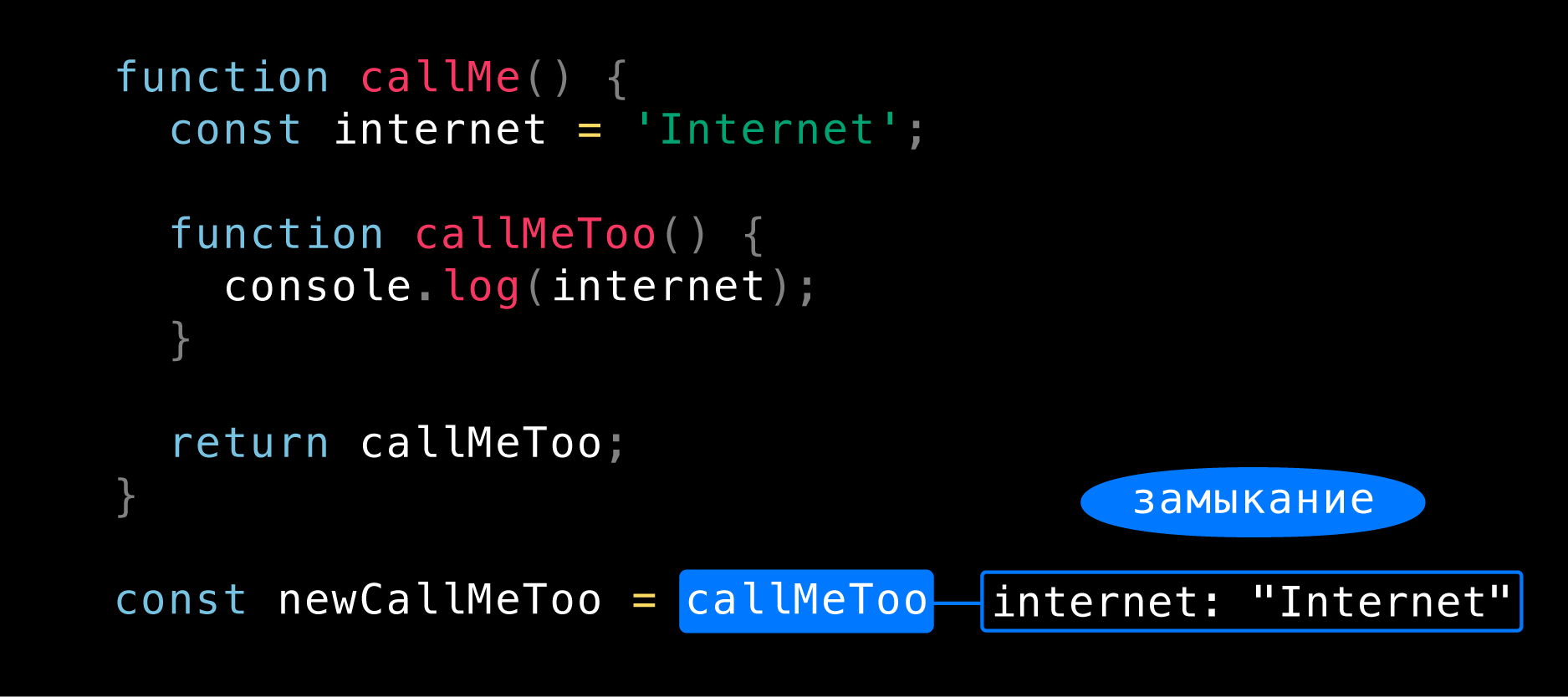
Что такое замыкание, разобрались. Теперь расскажем, как движок их создаёт.

Вы уже знаете об этом — вспомните прошлый урок этой темы. При создании функции у неё появляется внутреннее свойство [[Environment]]. Оно указывает на внешний LexicalEnvironment — объект переменных области видимости, в которой функция была создана. Здесь это объект переменных функции callMe.



Когда мы вернули callMeToo наружу, в свойстве [[Environment]] она унесла с собой объект переменных области видимости, в которой была объявлена.

Теперь откуда ни вызови функцию, при обращении к переменным движок сначала ищет их внутри самой функции, а затем во внешнем объекте переменных, находя его в свойстве [[Environment]]. Если в этом объекте есть нужная переменная (в нашем случае internet), говорят, что функция берёт её из замыкания.



Если бы переменной internet не было в свойстве [[Environment]] функции callMeToo, движок стал бы искать её в глобальной области видимости, заглянув в свойство [[Environment]] функции callMe. Если бы нашёл её там, мы всё равно сказали бы, что переменная взята из замыкания.

# Важно

Замыкание создаётся тогда же, когда и внешний объект LexicalEnvironment. То есть, при вызове функции.

Скопировать кодJAVASCRIPT

function callMe() {

const internet = 'Internet';

function callMeToo() {

console.log(internet);

}

return callMeToo;

}

const newCallMeToo = callMe(); *// здесь создаётся замыкание функции callMeToo*

newCallMeToo(); *// "Internet"*

Пока функция callMe не вызвана, код внутри неё движком не выполняется. Соответственно и callMeToo не объявляется, и замыкание не создаётся.

## Резюме

Тема замыканий не проста. Поэтому не переживайте, если не вникли сразу. Мы и сами поняли её не с первого раза.

В следующем уроке расскажем, чем замыкания полезны на практике. Это тоже поможет понять их суть.

Пример

Напишите функцию makeConsoleMessage. Она должна создавать другие функции вот так:

Скопировать кодJAVASCRIPT

function makeConsoleMessage(text) {

*// напишите код здесь*

}

const consoleHello = makeConsoleMessage('Привет');

const consoleGoodBye = makeConsoleMessage('Пока');

consoleHello(); *// "Привет"*

consoleGoodBye(); *// "Пока"*

При вызове makeConsoleMessage ей передаётся строка, которую должна выводить в консоль функция-результат.

*function makeConsoleMessage(text) {*

*function consoleMessage() {*

*console.log(text)*

*}*

*return consoleMessage;*

*}*

*const consoleHello = makeConsoleMessage('Привет');*

*const consoleGoodBye = makeConsoleMessage('Пока');*

*consoleHello(); // "Привет"*

*consoleGoodBye(); // "Пока"*

Усовершенствуем функцию makeConsoleMessage*.* Пусть функция-результат тоже принимает аргумент. Работать должно так:

Скопировать кодJAVASCRIPT

function makeConsoleMessage(text) {

function consoleMessage() { *// добавьте функции параметр*

console.log(text); *// и используйте его в вызове console.log*

}

return consoleMessage;

}

const consoleHello = makeConsoleMessage('Привет, ');

const consoleGoodBye = makeConsoleMessage('Пока, ');

const consoleHemingway = makeConsoleMessage('Прощай, ')

consoleHello('Андрей'); *// "Привет, Андрей"*

consoleGoodBye('Ирина'); *// "Пока, Ирина"*

consoleHemingway('оружие!'); *// "Прощай, оружие!"*

Результат

*function makeConsoleMessage(text) {*

*function consoleMessage(newText) { // добавьте функции параметр*

*console.log(`${text}${newText}`); // и используйте его в вызове console.log*

*}*

*return consoleMessage;*

*}*

*const consoleHello = makeConsoleMessage('Привет, ');*

*const consoleGoodBye = makeConsoleMessage('Пока, ');*

*const consoleHemingway = makeConsoleMessage('Прощай, ');*

*consoleHello('Андрей'); // "Привет, Андрей"*

*consoleGoodBye('Ирина'); // "Пока, Ирина"*

*consoleHemingway('оружие!'); // "Прощай, оружие!"*

# Замыкания на практике

В прошлом уроке вы узнали о замыкании: способности функции запоминать область видимости, в которой она была объявлена. В этом уроке будем применять замыкания на практике.

## Функция обратного вызова

Вы уже использовали замыкания. Этот пример вам раньше встречался:

Скопировать кодJAVASCRIPT

const button = document.querySelector('button');

const hint = document.querySelector('#hint');

*// сделаем элемент #hint невидимым*

hint.style.display = 'none';

function handler(event) {

hint.style.display = 'block';

}

*// делаем hint видимым по двойному клику*

button.addEventListener('dblclick', handler);

Это пример использования замыкания. Методу addEventListener мы передаём функцию обратного вызова, но не вызываем её. Мы говорим методу addEventListener: «вызови эту функцию, когда произойдёт двойной клик». addEventListener — это тоже функция, просто заранее описанная создателями браузера. Когда произойдёт двойной клик, addEventListener вызовет переданную ему функцию handler в своём коде. То есть handler будет вызвана вне области видимости, в которой была объявлена, но у неё по-прежнему будет доступ к переменной hint. При объявлении функция предусмотрительно собрала портфель с переменными своей области видимости.



## Не рассчитывайте на глобальные переменные

Глобальными переменными лучше не пользоваться, потому что у всего кода есть к ним доступ. Например, на сайте есть счётчик, чьё значение при загрузке равно 0. При клике значение счётчика растёт:

Скопировать кодJAVASCRIPT

*/\* наш код \*/*

const button = document.querySelector('button');

const div = document.querySelector('div');

let counter = 0; *// изначально counter равен 0*

function increaseCounter(event) {

counter += 1; *// увеличим счётчик на единицу*

div.textContent = counter; *// вставим на страницу обновлённое значение*

}

*// увеличиваем счётчик по клику на кнопку*

button.addEventListener('click', increaseCounter);

Всё работает. Но вы не единственный разработчик в проекте и кто-то другой решил тоже воспользоваться глобальной переменной counter для собственных целей:

Скопировать кодJAVASCRIPT

*/\* код другого разработчика \*/*

const someOtherButton = document.querySelector('.some-other-button');

*// функция, удваивающая счётчик*

function doubleCounter(event) {

counter = counter \* 2;

}

*// сбросил счётчик по клику на какую-то другую кнопку*

someOtherButton.addEventListener('click', doubleCounter);

Оба разработчика обращаются к одной переменной, и потому ни один из них не может быть уверенным в её значении. В результате не работает ни код одного, ни код другого.

Замыкания позволяют не допустить таких поломок:

Скопировать кодJAVASCRIPT

*/\* наш код \*/*

const button = document.querySelector('button');

function createCounter(event) {

const div = document.querySelector('div');

let counter = 0; *// эта переменная недоступна из окружающего кода*

*// но у функции increaseCounter есть доступ к ней*

function increaseCounter() {

counter += 1;

div.textContent = counter;

}

*/\* вернём increaseCounter, она сохранит доступ*

*к переменной counter через замыкание \*/*

return increaseCounter;

}

*// создадим increaseCounter с замыканием*

const increaseCounter = createCounter();

button.addEventListener('click', increaseCounter);

Теперь counter надёжно защищён. У других частей кода нет доступа к счётчику, а у функции increaseCounter есть — она берёт его из замыкания.

## Модуль

В примере со счётчиком мы возвращаем функцию increaseCounter. Только у неё есть доступ к переменной counter, а у внешнего кода этого доступа нет. Мы вернули наружу способ увеличить счётчик на единицу, но другие разработчики теперь не могут сбросить счётчик или записать в него своё значение. Теперь логика нашей программы разделена на приватную (counter нельзя обнулить или переписать из окружающего кода) и публичную (increaseCounter может увеличивать counter на 1). Такой приём проектирования называется модуль.

Теперь нам нужно расширить функционал счётчика: необходимо создать кнопку для его сброса. Чтобы интегрировать эту возможность в код, функция createCounter должна возвращать не функцию, а объект с функциями:

Скопировать кодJAVASCRIPT

const increaseButton = document.querySelector('button.increase');

const resetButton = document.querySelector('button.reset');

function createCounter(event) {

const div = document.querySelector('div');

let counter = 0;

function increaseCounter() {

counter += 1;

div.textContent = counter;

}

*// добавим resetCounter, у неё тоже есть доступ к counter*

function resetCounter() {

counter = 0;

div.textContent = counter;

}

*/\* вернём объект с двумя функциями,*

*это публичные методы нашего модуля \*/*

return {

increaseCounter,

resetCounter

};

}

*// создадим счётчик*

const counter = createCounter();

*// добавим обработчики*

increaseButton.addEventListener('click', counter.increaseCounter);

resetButton.addEventListener('click', counter.resetCounter);

Теперь createCounter возвращает объект. Мы можем добавлять в него новые методы, например setCounter, который будет устанавливать определённое значение счётчика или randomizeCounter, выбирающий число случайно.

А ещё мы можем создавать новые счётчики, которые будут независимы друг от друга:

Скопировать кодJAVASCRIPT

*// Это всё будут независимые счётчики*

const counter1 = createCounter();

const counter2 = createCounter();

const counter3 = createCounter();

Код внутри createCounter исполняется движком в момент вызова. И каждый раз создаётся новая переменная counter и объявляются новые функции increaseCounter и resetCounter.

Замыкания — лучшие друзья разработчика. Они позволяют использовать колбэки и разделять код на публичный и приватный. И не беспокоиться, что одни части программы сломают другие.

**Заключение**

Вы уже привыкли к сложной теме в конце спринта? Её цель — держать вас в тонусе перед будущими собеседованиями.

В этот раз мы разобрали не самые простые концепции, связанные с функциями. Если было сложно, перечитайте материалы прямо перед собеседованиями. Вопрос про замыкание точно однажды всплывёт.