Что такое JavaScript

*Чтобы добавить сайту больше динамичности нужно использовать JavaScript.*

**Что такое JavaScript?**

Ранее мы изучили HTML и CSS. С их помощью мы научились верстать веб-страницы, стилизовать их, добавлять интересные CSS-анимации.

Однако современные веб-сайты куда более динамичные и интерактивные: когда мы нажимаем на кнопку, открываются модальные окна, появляются крутящиеся галереи с возможностью пролистать изображения. Все это можно добавить на веб-сайт с помощью **JavaScript.**

Сейчас с его помощью создают по-настоящему необычные сайты — например, [сайт студии Hello Monday](https://www.hellomonday.com/).

Эти необычные анимации с перетеканиями сделаны с помощью JavaScript.

JavaScript лежит в основе многих игр, вплоть до сложнейшей 3D-графики, для этого существует даже специальная библиотека [Three.js](https://threejs.org/). Вращающаяся планета Земля на [главной странице GitHub](https://github.com/home) была сделана с помощью **Three.js**:

С помощью JavaScript делают приложения для телефонов, программы для компьютеров. Он считается одним из самых популярных языков программирования, поэтому для него написано огромное множество библиотек, в том числе для анализа данных, для работы с нейронными сетями.

Большие платформы, такие как YouTube, VK, Facebook, на 90% состоят из JavaScript, команды из нескольких десятков frontend-разработчиков работают над этими проектами. И frontend-разработчик — это в первую очередь программист, который пишет на JavaScript.

JavaScript — интерпретируемый, объектно-ориентированный язык программирования.

Разберемся сначала с тем, что такое язык программирования.

**Язык программирования** — *это формальный язык, с помощью которого пишут программы. Термин «формальный» говорит о том, что этот язык состоит из набора правил, команд. Люди общаются на естественном языке, для общения с компьютером нужен язык программирования.*

JavaScript, как и любой другой язык программирования, имеет свою лексику, функции и операторы, из которых по правилам синтаксиса составляется программный код.

А что такое **объектно-ориентированный**? Это значит, что он поддерживает объектно-ориентированный подход написания кода (парадигму), подробнее о нем мы поговорим дальше.

Также мы упомянули, что JavaScript — **интерпретируемый** язык программирования. Код в таких языках выполняет специальная программа, интерпретатор (от англ.*interpreter — истолкователь*). Язык состоит из набора команд, которые затем выполняет интерпретатор.

Существуют также компилируемые языки, в которых программы сначала компилируются и только потом выполняются. В них нельзя написать одну команду и сразу же получить результат, как в интерпретируемых.

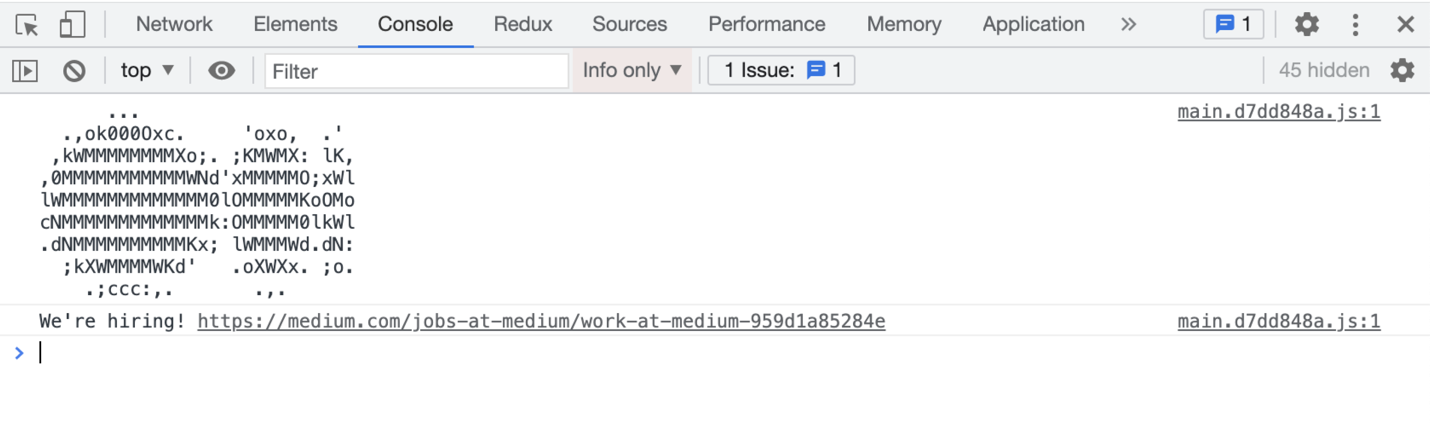
Итак, у интерпретируемых языков есть интерпретатор. Для JavaScript интерпретатор встроен в браузер, благодаря чему прямо в браузере можно выполнять разные команды. Во многих браузерах есть даже специальная **консоль разработчика**.

Консоль разработчика

**Саша, старший разработчик:** *«Чтобы ты смог запустить код на JavaScript, тебе необходим браузер. Давай я тебе расскажу об инструменте, который есть в каждом браузере, — он поможет выполнить твой первый JavaScript-код».*

**Консоль разработчика** — часть инструментов разработчика браузера, которая позволяет выполнять JavaScript-код, показывает ошибки при его выполнении, она даже показывает ошибки открытого в браузере сайта. Если на сайте не загрузится какая-то картинка или информация, об этом будет написано в консоли разработчика.

В эту консоль в основном заглядывают программисты, поэтому иногда там можно увидеть сообщение **We're hiring!** (*Мы нанимаем*), как, например, на платформе [Medium](https://medium.com/" \t "_blank):

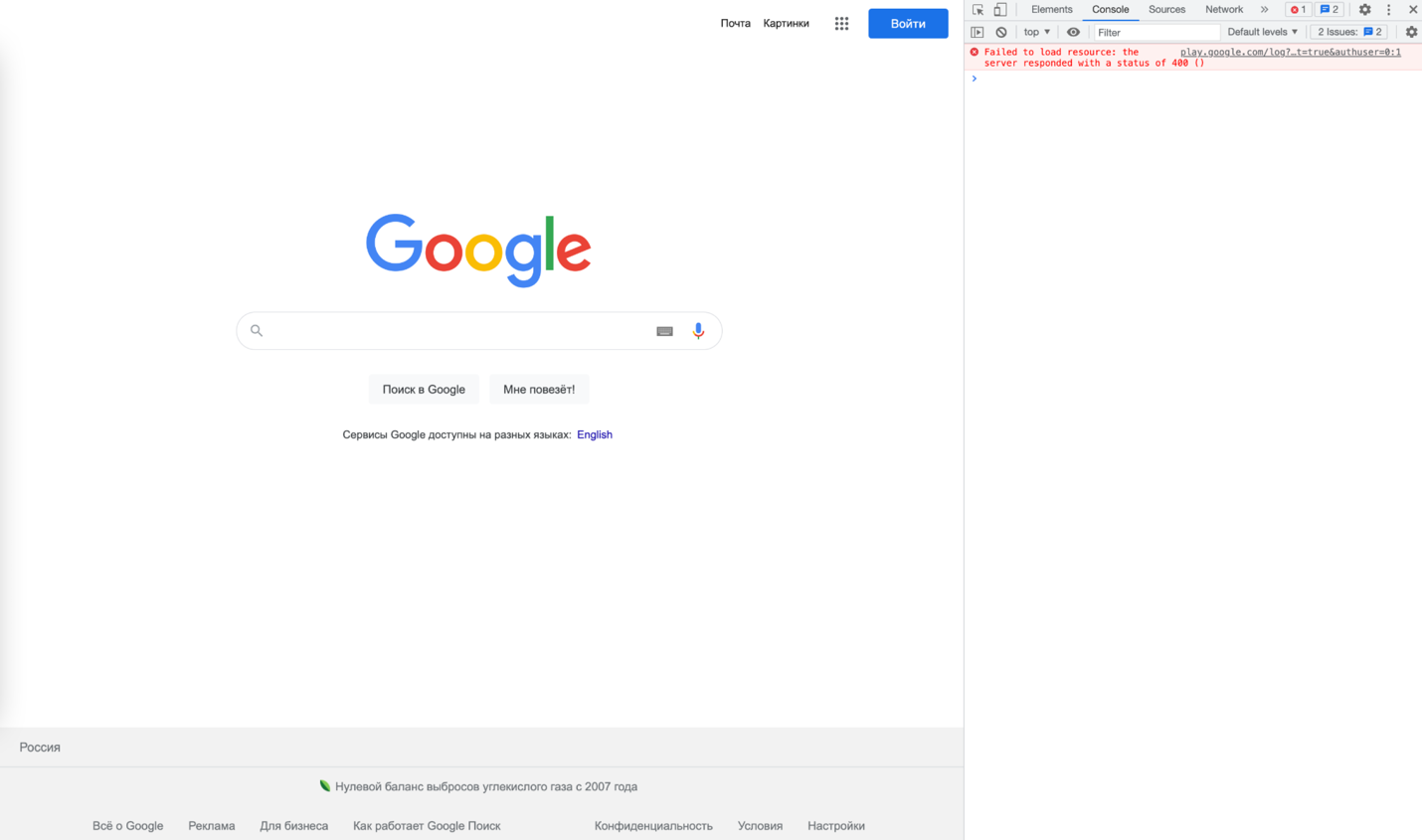
[](https://cs.sberuniversity.online/image/full/full/resize/0fbfc9d0-33fc-11ec-971d-0242ac160002)

Давайте подробнее изучим инструменты разработчика. Откроем браузер Google Chrome, а в нем страницу [Google](https://www.google.com/" \t "_blank). Теперь откроем инструменты разработчика, удобнее всего это сделать набором команд (шорткатом):

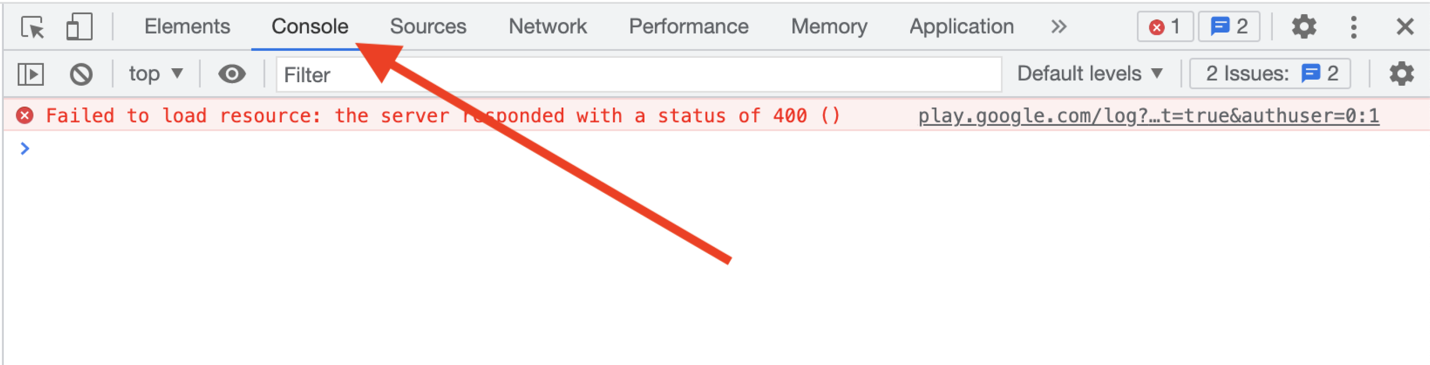
Для Mac OS: command + option + I,  
для Windows: F12,  
для Linux: control + shift + I.

Их также можно открыть, найдя **Инструменты разработчика** в меню Chrome.

Итак, после ввода шортката открывается меню, оно может быть сбоку или снизу:

[](https://cs.sberuniversity.online/image/full/full/resize/669ba4c2-33fc-11ec-9042-0242ac160003)

Консоль разработчика находится во вкладке **Console**, откройте ее, если она у вас еще не открыта:

[](https://cs.sberuniversity.online/image/full/full/resize/8a6c2232-33fc-11ec-8995-0242ac160002)

Как видно, в нашей консоли уже есть какой-то вывод красного цвета. Указан текст:

**Failed to load resource: the server responded with a status of 400 ()**

Это ошибка с сайта Google, говорящая о том, что не удалось загрузить какой-то ресурс, вероятно, файл. Так же будут выглядеть и другие ошибки, возникающие при выполнении кода.

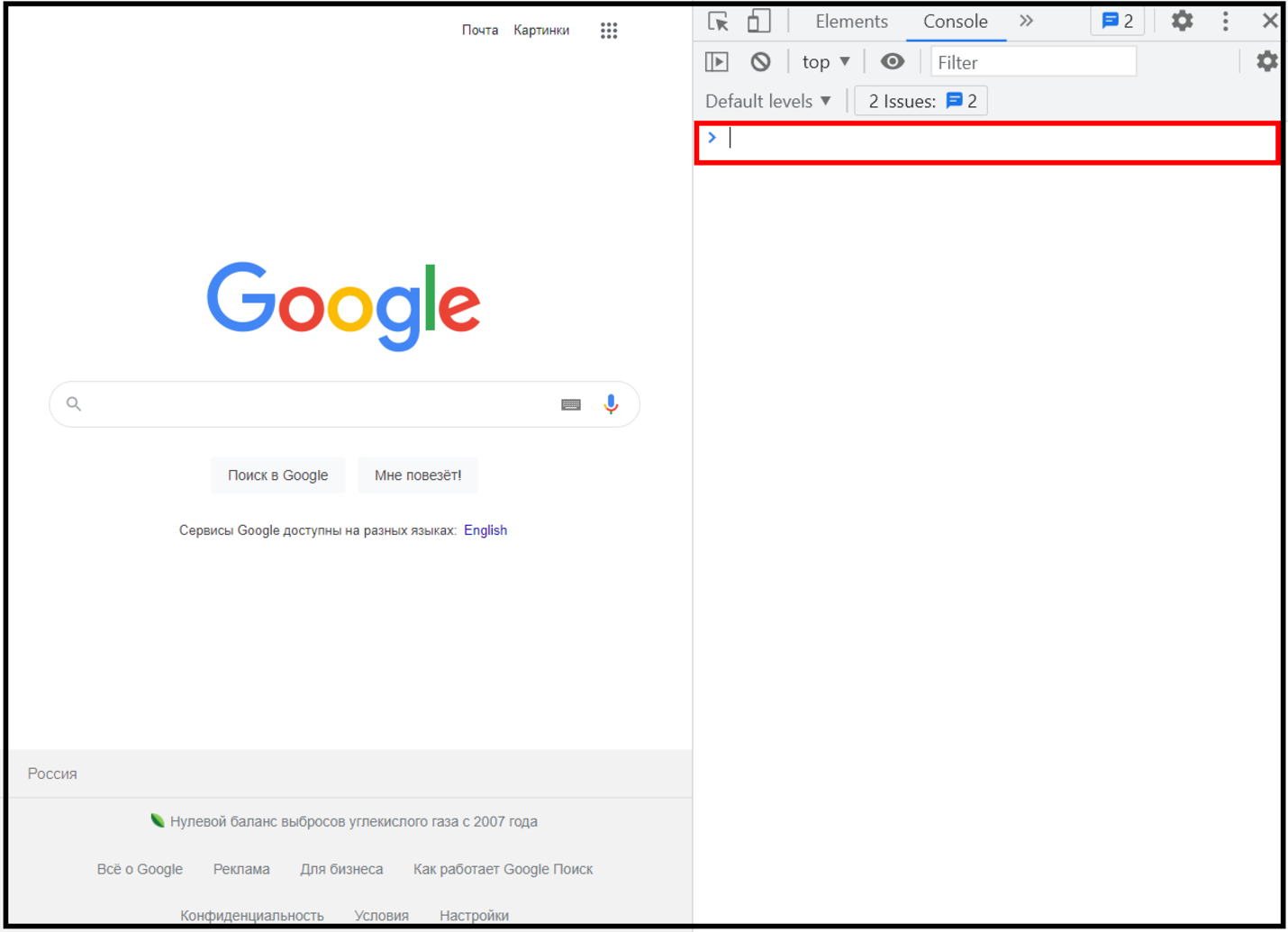
Почистим ошибки, чтобы они нам не мешали, для этого нужно нажать на кнопку очищения (**Clear console**):

[](https://cs.sberuniversity.online/image/full/full/resize/b8921dba-33fc-11ec-8ca6-0242ac160002)

Чаще всего консолью разработчика пользуются, чтобы искать ошибки в коде. Если на сайте есть ошибка, она выводится в консоль, как мы уже видели в примере выше. Консоль также позволяет найти точное место в коде, где произошла эта ошибка.

Помимо этого, консоль разработчика позволяет запускать JavaScript-код.

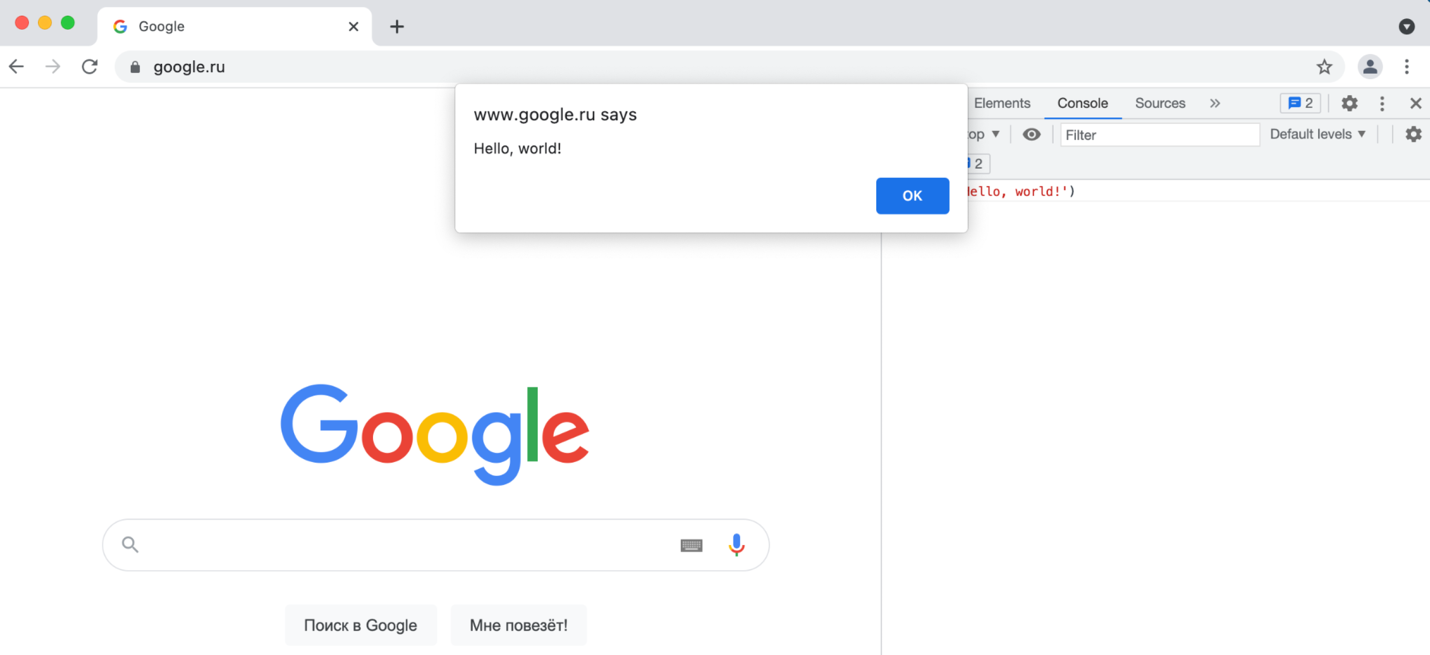
Выделенный на скриншоте символ **>** синего цвета означает командную строку: в ней запускаются и редактируются JavaScript-команды:

[](https://cs.sberuniversity.online/image/full/full/resize/dd380166-33fc-11ec-97be-0242ac160002)

Пришло время написать первую команду на JavaScript. Существует традиция начинать изучение языка программирования с вывода сообщения **Hello, world!.** Для этого воспользуемся специальной функцией.

**Функция** —*это мини-программа, с помощью которой можно что-то сделать. Например, мы хотим вывести сообщение, для этого есть функция****alert.***

Напишите **alert('Hello, world!')** в командной строке, а затем нажмите **Enter.** Откроется модальное окно с этим сообщением:

[](https://cs.sberuniversity.online/image/full/full/resize/19630802-33fd-11ec-86b2-0242ac160002)

Чтобы воспользоваться функцией, мы написали ее имя, **alert**, и далее круглые скобки **(),** тем самым мы вызвали функцию. Внутри круглых скобок мы передали сообщение, это называется аргументом функции. Подробнее о функциях мы поговорим в дальнейшем.

Функция **alert** позволяет вывести сообщение в окне браузера: как видно, она открывает модальное окно, которое не позволяет ничего сделать на сайте, пока не будет нажата кнопка **OK**. Код страницы тоже не будет выполняться до тех пор, пока пользователь не закроет это окно.

Сейчас такие окна используют сомнительные сайты, чтобы напугать пользователя вирусом на компьютере или чем-то подобным.

Ранее мы демонстрировали пример с сообщением **We're hiring!**, которое показывалось в консоли разработчика на сайте компании. Его можно вывести с помощью функции **log** у объекта **console**, таким образом команда будет следующей: **console.log**.

Функции, которые являются частью чего-то, например частью **console**, называют методами. К методу сначала надо обратиться через точку, а затем вызывать через круглые скобки, например **console.log()**. Мы обратились к методу **log** у объекта **console** через точку, а затем вызывали с помощью круглых скобок.

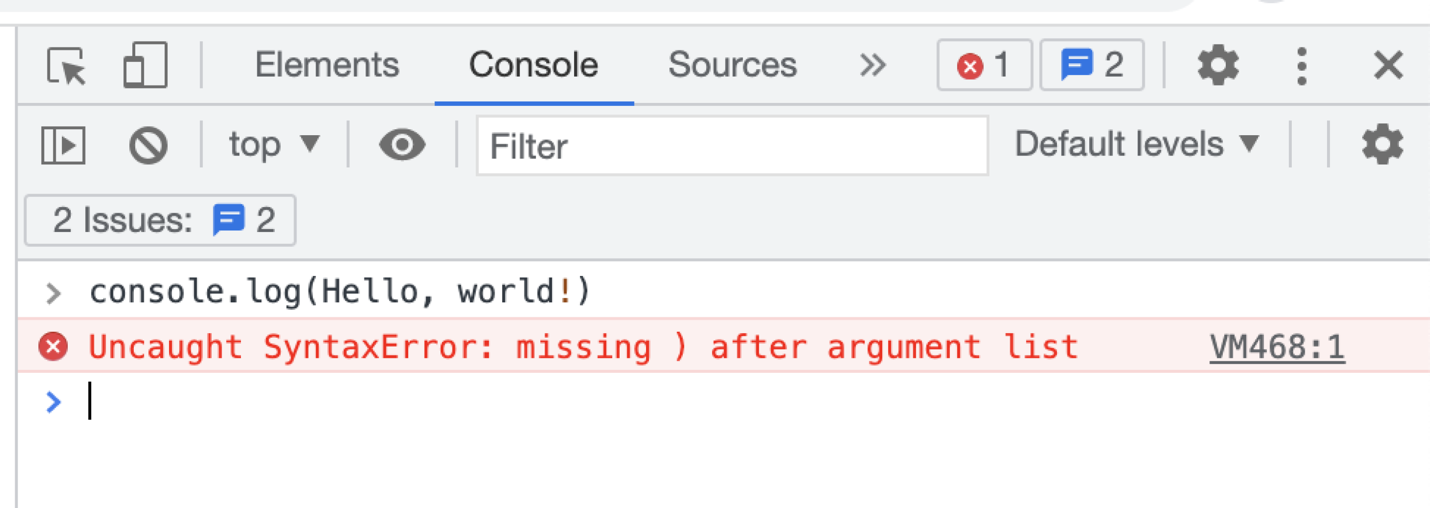
Команда **console.log** очень часто используется для вывода отладочной информации. Давайте попробуем вывести в консоль число **123**. Код будет выглядеть так:

console.log(123);

Сразу после выполнения команды видим ее вывод, число **123**:

[](https://cs.sberuniversity.online/image/full/full/resize/ae0717aa-33fd-11ec-a61a-0242ac170003)

С помощью этой команды можно также выводить разные сообщения. Ранее мы выводили строку**Hello, world!**, заключив ее в кавычки. Обязательно указывайте строку в кавычках, без них браузер бы вывел ошибку:

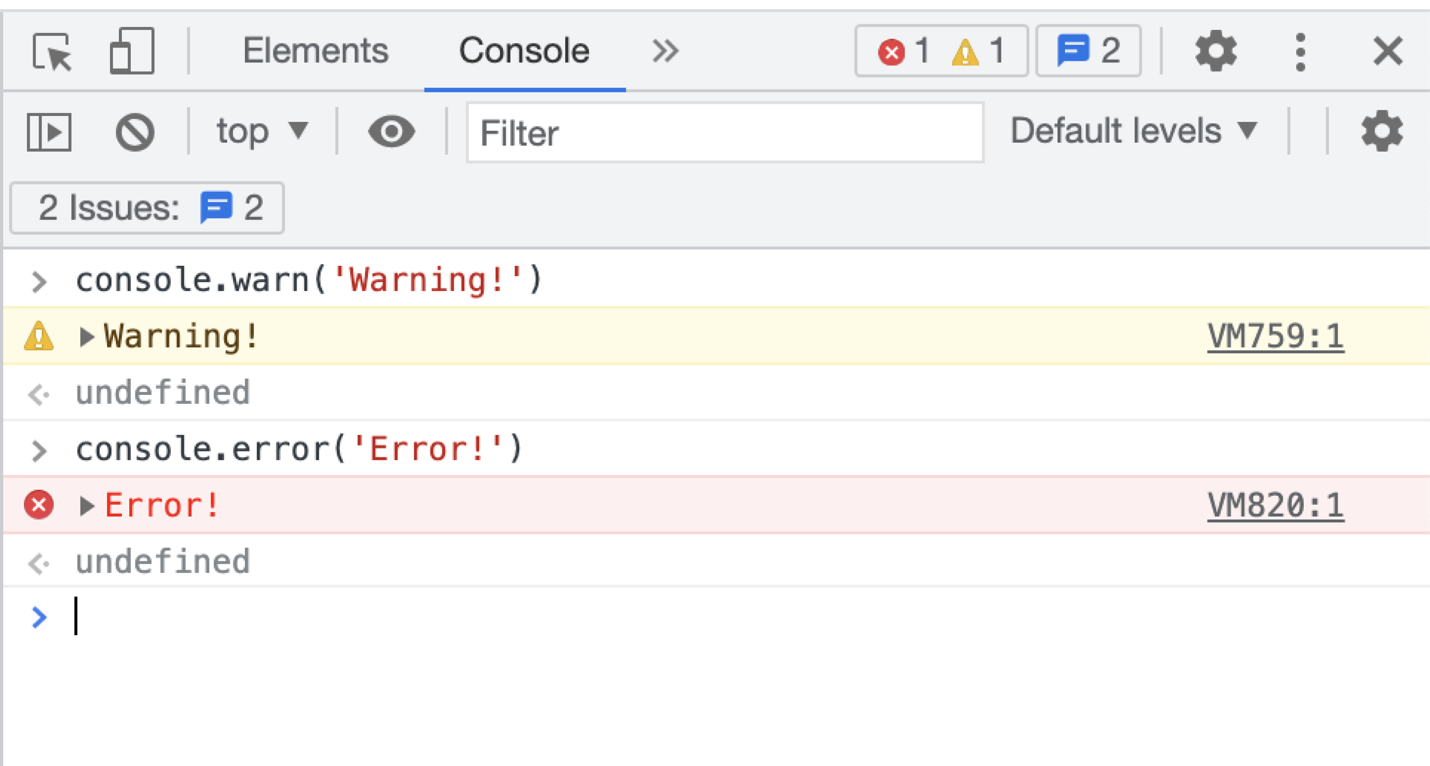
[](https://cs.sberuniversity.online/image/full/full/resize/cf615e6a-33fd-11ec-b2f9-0242ac170004)

Теперь напишем **Hello, world!** правильно:

[](https://cs.sberuniversity.online/image/full/full/resize/ed2587a0-33fd-11ec-a714-0242ac170004)

Как и с числами, мы сразу же видим ответ от браузера, который передал данные в новой строке.

С помощью **console** можно также выводить предупреждения и ошибки. Метод **console.warn** — для предупреждений, а **console.error** — для ошибок:

[](https://cs.sberuniversity.online/image/full/full/resize/0f54b4f4-33fe-11ec-a42e-0242ac160003)

**Практическое задание 1.1**

**Выберите код, который получился в результате проверки выполненного задания**

3Z@YO

YGCBZ

MX%44

F7031

#Q9PN

Поздравляю, Вы справились с заданием за 1 попытку.

 Ответить снова

НазадК следующему шагу

# JavaScript в HTML

**Саша, старший разработчик:** «Я думаю, неплохо для первого знакомства с JavaScript. Перейдем от теории к практике, ведь вы уже достаточно знаете, чтобы написать первую программу. В предыдущем разделе мы изучили, как писать HTML-разметки и CSS-стили, теперь ты попробуешь подключить JavaScript к своему проекту!»

Итак, мы умеем запускать JavaScript в консоли разработчика браузера. Это полезно для того, чтобы быстро запустить небольшой код и проверить его, однако нам нужно, чтобы JavaScript запускался на нашем сайте. Для этого в HTML есть тег**script**.

### Тег script

Тег **script** позволяет вставить JavaScript-код на сайт. Это парный тег, а значит, у него есть открывающий и закрывающий теги:

<script></script>

Есть два вида таких скриптов, определяющих способ добавления JavaScript-кода:

* внутренние скрипты;
* внешние скрипты.

### Внутренние скрипты

В случае внутреннего скрипта мы пишем JavaScript-код сразу внутри тега **script:**

<body>

*<!-- HTML код -->*

<h1>Первый заголовок сайта</h1>

<h2>Второй заголовок сайта</h2>

<script>

*// JavaScript-код*

console.log('Привет!');

</script>

</body>

Браузер читает HTML-код сверху вниз, поэтапно распознавая нашу разметку и стили. JavaScript в этом случае не исключение. Другими словами, пока тег **script** не загрузится, все, что после него, тоже. Поэтому тег **script** обычно размещают самым последним тегом в теге **body,** таким образом он не будет блокировать отрисовку HTML-элементов и стилей и при этом в JavaScript-коде будут доступны все элементы страницы.

### Внешние скрипты

Когда мы изучали CSS, мы научились подключать CSS-стили, находящиеся в отдельном файле. JavaScript-код также можно хранить в отдельном файле и подключить его на сайт, тогда это будут внешние скрипты.

Чтобы подключить внешний файл JavaScript-кода, надо использовать атрибут **src:**

<script src="js/script.js"></script>

Таким образом, внутри тега мы ничего не пишем, а весь код находится во внешнем файле. Если внутри тега **script** что-то написать, то его содержимое будет игнорироваться, так как указан атрибут **src.**

**js/script.js** — относительный путь от папки с HTML-файлом до файла с JavaScript-кодом. В нашем примере **script.js** находится в папке **js.**

Можно указать ссылку на файл (URL-адрес), например:

<script src="https://example.com/somescripts/script.js">

</script>

С помощью ссылок на сайты обычно подключаются дополнительные библиотеки.

Чтобы подключить несколько скриптов, надо использовать несколько тегов:

<script src="js/script1.js"></script>

<script src="js/script2.js"></script>

НазадК следующему шагу

# Что такое переменная

Ранее мы выводили сообщения, сразу передавая их в функции. На практике же очень часто нужно хранить какие-то значения, которые могут меняться. Например, нужно хранить телефон и адрес пользователя, он может их изменить, и тогда эти значения надо перезаписать.

Для хранения каких-либо данных в языках программирования используют **переменные**.

Переменная — это область памяти, где можно хранить и изменять данные. У переменной всегда есть имя, по нему мы можем обращаться к данным.

Важно понимать, что переменные это не сами значения, это как коробка, куда можно положить различные значения. Именно поэтому они называются переменными, ведь данные внутри можно менять. Например, у вас есть коробка, где вы храните конфеты, а потом вы решили их выложить и положить туда печенье. Печенье и конфеты — это значения, а сама коробка — переменная.

Чтобы использовать переменную, надо ее создать. Для этого можно написать оператор **let**, а затем имя переменной. Например, создадим переменную для хранения адреса с именем **address**:

let address;

Такой процесс создания переменной правильнее называть ее объявлением. Таким образом мы объявили переменную **address** с помощью оператора**let.**

**Оператор** в языках программирования — это какая-то команда или возможно набор команд. В нашем случае это команда объявить переменную с таким-то именем. По ходу изучения JavaScript мы будем знакомиться с все большим количеством разных операторов.

Итак, мы объявили переменную, давайте посмотрим, что находится в ней сейчас. Чтобы получить доступ к данным в переменной, надо воспользоваться ее именем. Выведем в консоль значение переменной:

let address;

console.log(address);

### Что выведется в консоль после выполнения блока кода выше?

Ошибка

address

undefined

empty

Поздравляю, Вы справились с заданием за 1 попытку.

 Ответить снова

Такое поведение будет у любой объявленной, «пустой» переменной.

Переменную можно сразу объявить с каким-то значением:

let address = "Москва";

В данном случае говорят, что мы инициализировали переменную значением **"Москва"**.

Итак, у нас есть переменная **address** с записанным в нее адресом. Предположим, что адрес изменился и теперь его нужно перезаписать. Для этого переменной **присваивается** новое значение с помощью **оператора присваивания, =**. Присвоим нашей переменной новое значение адреса:

address = "Санкт-Петербург";

Как видно из блока кода выше, чтобы присвоить новое значение переменной, надо написать имя переменной, далее символ равно (**=**) и новое значение.

Теперь в нашей переменной есть данные. Выведем в консоль значение переменной:

let address = "Санкт-Петербург";

console.log(address);

Также можно объявлять и инициализировать сразу несколько переменных:

let login, password;

let name = "Вася", address = "Санкт-Петербург";

Итак, **переменная** — это область памяти, где мы храним данные. Сперва мы объявляем переменную, можем ее инициализировать каким-то начальным значением. Далее, чтобы изменить данные в переменной, мы присваиваем ей новое значение. У каждой переменной должно быть имя, и все операции с переменными мы делаем через ее имя.

НазадК следующему шагу

# Как называть переменные

Названия переменных могут состоять из букв, цифр и знаков **\_** или **$**. При этом имя переменной не может начинаться с цифры:

let myVariable;

let 123; *// Ошибка!*

Также в JavaScript есть слова, которые используются в синтаксисе языка для разных команд, их нельзя использовать как имя переменной. Например, нельзя объявить переменные с именами **let, function, var, for** и т. д. Все эти команды мы будем изучать по ходу курса.

Правильно назвать переменную для программиста — очень важно. Нельзя к этому подходить легкомысленно, особенно если программа пишется не только вами, а в команде с коллегами. Почему? Потому что ваш код будут читать и пытаться понять другие люди, а правильно названная переменная поможет лучше понять код.

Представьте, если бы у ранее изученной функции**alert** было бы имя **a**. Конечно, писать его короче, но по букве **a** ничего не понятно о том, что делает эта функция. Также и с изученным объектом **console** и его методом **log** — а что если бы их назвали **b** и **с**? Мы бы тогда выводили сообщение так: **b.c(123).** Совершенно непонятно, что за **b** и **c**.

Мы сформулировали несколько правил, которых стоит придерживаться, когда придумываете имя для переменной:

**1.** Используйте понятные имена для переменных.

Старайтесь не использовать короткие имена для переменных, особенно состоящие из одной буквы: **a, b, c, d, e** и т. д.

Имя из одной буквы не несет совершенно никакого смысла, а значит, не поможет понять, что именно хранится в данной переменной. Можно давать такое имя переменной, если она используется в совсем небольшом фрагменте кода, но даже в этом случае рекомендуем все-таки придумать более подробное имя.

Так переменные называть плохо:

let a;

let n1;

let z\_x\_e;

Пример хороших имен переменных:

let address;

let firstName;

**2.** Когда вы будете придумывать понятные имена, появится необходимость составлять их из нескольких слов. В JavaScript есть два самых популярных стиля написания составных слов: **camelCase** и **snake\_case**. В стиле **camelCase** каждое следующее слово пишется с заглавной буквы:

let firstName; *// Имя*

let secondName; *// Фамилия*

let currentUserAddress; *// Адрес текущего пользователя*

В стиле **snake\_case** слова разделяются нижним подчеркиванием:

let first\_name;

let second\_name;

let current\_user\_address;

Чаще всего используют стиль **camelCase,** таким образом запись короче (нет символов подчеркивания), а также в таком стиле именуются встроенные функции в браузере.

**3.** Переменные надо называть по-английски, не рекомендуется использовать транслит русских слов. Транслит тяжело читать, а также вы можете работать в команде с нерусскоязычными разработчиками. В целом имена переменных на транслите выглядят как дурной тон.

Примеры плохих имен переменных:

let kolichestvo;

let opisanie;

let cena;

let magazin;

Примеры хороших имен переменных:

let quantity; *// Количество*

let description; *// Описание*

let price; *// Цена*

let shop; *// Магазин*

**4.** Неизменяемые значения лучше всего называть словами в верхнем регистре.

На практике при работе в команде над большим проектом вам будут встречаться неизменяемые значения, которые будут общими для всего проекта. Например, число дней в неделе или максимальная длина пароля. Такие значения, которые никогда не изменятся, еще называют **константами**. При написании кода их принято называть заглавными буквами, слова разделять через подчеркивание, например:

const PASSWORD\_MIN\_LENGTH = 8;

НазадК следующему шагу

# Способы объявления переменных: let, const и var

Ранее мы приводили примеры с переменными, которые объявляли с помощью оператора**let.** Для объявления переменных существуют также операторы **var** и **const.**

С помощью оператора **let** можно создать переменную, значение которой можно перезаписать:

let myFavouriteColor = "blue";

myFavouriteColor = "orange";

myFavouriteColor = "green";

Если вы уверены, что значение переменной меняться не будет, то стоит использовать оператор **const** (от англ. constant — постоянная). Таким переменным нельзя присваивать новое значение, это приведет к ошибке:

const myFavouriteColor = "blue";

myFavouriteColor = "orange"; *// Ошибка!*

Значит, у переменных, созданных с помощью **const,** значение менять нельзя. А что будет, если объявить переменную, но не инициализировать ее?

const myFavouriteColor;

console.log(myFavouriteColor);

### Что выведется в консоль после выполнения блока кода выше?

Ошибка

myFavouriteColor

undefined

empty

Поздравляю, Вы справились с заданием за 1 попытку.

 Ответить снова

На практике чаще всего переменным не присваиваются новые значения, а создаются новые переменные. Именно поэтому иногда в командах разработчиков есть правило: по умолчанию все переменные объявлять через **const**, лишь в крайних случаях использовать**let.** Таким образом сразу видно неизменяемую переменную в коде, следовательно, это улучшает понимание кода. Также есть мнение, что использование **const** вместо**let** улучшает скорость работы кода, однако оно не доказано.

В других командах разработчиков, наоборот, всегда используют**let**, а **const** используют только для констант всего проекта (как, например, константа для минимальной длины пароля). У этой точки зрения аргументы заключаются в том, что **let** — это три символа, а **const** — пять, а также весь код выглядит единообразно.

В целом рекомендуем вам выбрать для себя один из этих подходов и придерживаться его.

Оператор **var** — устаревший аналог оператора **let**. В современных программах его уже не используют. На первый взгляд этот оператор ничем не отличается от **let**:

var myFavouriteColor = "blue";

myFavouriteColor = "orange";

Как видно, с его помощью мы также можем объявить переменную и присвоить ей новое значение. Однако разница все-таки есть.

Посмотрим на такой код:

console.log(message);

var message = "Привет!";

В коде мы сначала обращаемся к переменной и только потом объявляем ее и присваиваем ей значение. При этом в консоль выведется **undefined.**

В случае с оператором**let** такой пример выводит ошибку **ReferenceError: message is not defined:**

console.log(message);

let message = "Привет!";

Как так? Получается, что мы пытаемся получить данные из переменной, которую еще не объявили, и при этом, в случае **let** по понятным причинам выводится ошибка, а в случае **var** ошибки нет, получаем на выходе **undefined.**

С **var** даже такой пример выполнится без ошибок:

message = "Привет, мир!";

console.log(message);

var message = "Привет!";

При этом в консоль выведется **Привет, мир!**.

Все потому, что объявления переменных **var** обрабатываются до выполнения кода. То есть, где бы ни находилось объявление переменной **var,** оно все равно будет как будто в самом начале кода. Первый пример можно переписать так, чтобы понять, что происходит в коде:

var message;

console.log(message);

message = "Привет!";

Как видно, наверх перенеслось именно объявление переменной, без присвоения значения. Второй же пример можно переписать так:

var message;

message = "Привет, мир!";

console.log(message);

message = "Привет!";

Такое поведение называется **hoisting** (или также «всплытием», «поднятием»). Это достаточно неочевидное поведение, которое плохо влияет на написание кода и может привести к ошибкам.

У оператора **var** есть и другие особенности, которые связаны с областью видимости переменных. Подробнее вы можете изучить их на [MDN](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Statements/var).

В целом оператор **var** считается устаревшим как раз из-за таких особенностей, которые вызывают неочевидное поведение и могут привести к ошибкам в коде.

### Практическое задание 2.1

### Выберите код, который получился в результате проверки выполненного задания

DL3BD

WW03K

44G71

SSTYE

1C590

Поздравляю, Вы справились с заданием за 1 попытку.

 Ответить снова

НазадК следующему шагу

# Как писать комментарии в программе

В JavaScript-коде можно оставлять комментарии так же, как мы это делали в HTML-коде. Комментарии помогают сделать код более читабельным, с их помощью можно пояснять неочевидные моменты в коде.

Также комментирование пригодится, если необходимо временно «отключить» часть JavaScript-кода. Ведь все, что находится в комментарии, никак не обрабатывается, а просто игнорируется.

Есть **однострочные** и **многострочные** комментарии.

### Однострочные комментарии

Такие комментарии начинаются со специальных символов **//**. Любой текст между **//** и концом строки будет игнорироваться JavaScript, то есть не будет выполняться.

Посмотрим на пример использования однострочных комментариев:

*// Процесс создания переменной*

let age = 36;

console.log(age); *// Вывод переменной в консоль*

Первая и часть третьей строки после**//** не видны JavaScript. Соответственно, браузер не учитывает их.

### Многострочные комментарии

Многострочные комментарии начинаются символами**/\*** и заканчиваются **\*/**. Любой текст между **/\*** и **\*/** JavaScript будет игнорироваться.

В этом примере для объяснения кода используется многострочный комментарий:

*/\**

*Процесс создания переменной, ее переопределения*

*и дальнейший вывод в консоль разработчика*

*\*/*

let age = 10;

age = 5;

console.log(age);

Как говорилось выше, помимо написания объяснений, может понадобиться закомментировать какую-то часть кода, если при разработке необходимо на время отключить ее, например:

1. let age = 10;

2.

3. /\*

4. age = 25;

5. ...

6. \*/

7. console.log(age);

В этом случае строки **3–6** включительно не будут задействованы в обработке кода и метод **console.log(age)** выведет в консоль разработчика значение **10.**

НазадК следующему шагу

Какие типы данных есть в JavaScript

В прошлых разделах мы познакомились с переменными. Теперь надо понять, что мы можем в них хранить.

В JavaScript любое значение имеет свой тип. Всего на данный момент есть восемь типов данных:

* **string**,
* **number**,
* **boolean**,
* **null**,
* **undefined**,
* **object**,
* **Symbol**,
* **BigInt**.

JavaScript является языком с **динамической типизацией**, то есть в переменной могут храниться значения разного типа: сначала проинициализировать переменную строкой, затем присвоить ей новое значение в виде числа и т. д.

Например:

let id = "10";

id = 10;

В языках со статической типизацией вы так сделать не сможете. В них, если в переменной хранится строка, то присвоить этой переменной можно только строку.

Разберем все типы данных подробнее.

НазадК следующему шагу

Строки

Итак, мы хотим создать строку. Для ее создания надо заключить текст в кавычки.

Существует три вида кавычек:

* Одинарные — **'Hello, world!'**,
* Двойные — **"Hello, world!"**,
* Обратные — **`Hello, world!`**.

Объявление через одинарные и двойные кавычки ничем не отличается, можно писать и так, и так:

const message1 = 'Hello, world!';

const message2 = "Hello, world!";

У обратных кавычек есть особенности, о них расскажем чуть позже. В строке можно хранить любые символы, в том числе и цифры:

const message = '123!\*\_{}';

Любой одинарный символ в кавычках тоже является строкой:

const oneChar = "П";

Строка также может быть пустой:

const empty = "";

* Вывод

Строкой является абсолютно все, что заключено в кавычки, вне зависимости от количества символов.

Перейдем к следующему примеру:

const company = "ООО "ГринКорп"";

Здесь переменной **company** на самом деле ничего не присвоено. Интерпретатор не поймет, что тут написано, поскольку наша строка закончилась на пробеле после**ООО**. После закрытия кавычек интерпретатор рассчитывает увидеть код, а видит непонятный ему **ГринКорп**. Что делать, когда надо использовать кавычки внутри строки?

Мы можем записать в одинарные кавычки:

const company = 'ООО "ГринКорп"';

И это сработает. А что делать, если мы хотим использовать в одном тексте оба вида кавычек? Мы ведь снова закроем строку и получим ошибку.

Тут поможет **экранирование символов**, т. е. замена управляющих символов на так называемые **управляющие последовательности** (их еще называют *escape-последовательностями*). С их помощью мы можем вставлять в текст не только кавычки, но и другие специальные символы (перенос строки, табуляцию и т. д.). Выглядят они как сочетание из **обратного слеша** и набора цифр или букв. Ниже представлена таблица со всеми последовательностями в JavaScript:

|  |  |
| --- | --- |
| **Код** | **Значение** |
| \0 | нулевой символ (символ NUL) |
| \' | одинарная кавычка |
| \" | двойная кавычка |
| \\ | обратный слеш |
| \n | новая строка |
| \r | возврат каретки |
| \v | вертикальная табуляция |
| \t | табуляция |
| \b | забой |
| \f | подача страницы |
| \uXXXX | кодовая точка Unicode |
| \xXX | символ из кодировки Latin-1 |

Из таблицы выше на практике чаще всего используются символы кавычек (**\"** и **\'**) и новой строки (**\n**).

* Важно!

Если мы написали несуществующую последовательность, допустим, **"\c"**, мы просто получим символ **"c"**.

Вернемся к примеру с **ООО «ГринКорп»**. Теперь мы можем записать название компании следующим образом:

const company = "ООО \"ГринКорп\"";

Рассмотрим пример записи текста с двумя видами кавычек:

console.log("Neighbor's child said \"hello\"!");

В примере выше мы экранируем только двойные кавычки (**\"**), поскольку с их помощью была создана строка. Значит, одинарные кавычки экранировать не нужно, ведь они не являются управляющим символом.

Рассмотрим еще несколько примеров:

**1.** Если нужен текст с переносами, можно использовать символ **\n**:

console.log("Первая строка\nВторая строка");

Выведется:

Первая строка

Вторая строка

**2.** Если нужен отдельный символ **\** и буква **n**, можно написать так:

console.log("Первая строка\\nВторая строка");

Мы экранировали обратный слеш, следовательно, получили не специальный символ переноса **\n**, а отдельный символ **\** и букву **n**.

Итого выведется:

Первая строка\nВторая строка

**3.** Таким же образом можно экранировать кавычки:

console.log("Строка с кавычкой \"");

Выведется:

Строка с кавычкой "

**4.** Экранирование слеша может дать ошибку:

console.log("Строка с кавычкой \\"");

Как и во втором случае, мы заэкранировали **слеш**, получили просто символ**\**, закрыли строку и вернулись к ошибке, как в примере с **"ООО "ГринКорп""**.

**Конкатенация строк**

На практике часто необходимо объединять строки. Допустим, надо поприветствовать пользователя, который зашел на сайт, фразой **Привет, <имя пользователя>**.

**Имя пользователя** может меняться, поэтому оно будет храниться в отдельной переменной, а текст **Привет** — в другой. Тогда код будет таким:

let name = "Петр";

const greeting = "Привет,";

Чтобы объединить эти строки, можно использовать обычное сложение:

let name = "Петр";

console.log("Привет," + name); *// Привет,Петр*

Этот процесс называется **конкатенацией.**

Посмотрите: мы объединили две строки, но между ними нет пробела. Его можно добавить таким способом:

console.log("Привет, " + "Петр") *// Привет, Петр*

Если указать пробел в конце первой строки и в начале второй, при конкатенации расстояние между словами будет слишком большим:

console.log("Привет, " + " Петр") *// Привет, Петр*

**Практическое задание 3.1**

**Выберите код, который получился в результате проверки выполненного задания**

7MT0

Z69%X

009T

RWRW

MG%0K

Подтвердить выбор

НазадК следующему шагу

# Шаблонные строки

Время поговорить о строках, созданных с помощью обратных кавычек. Выглядят они так:

const greeting = `Привет!`;

Такие строки также называют **шаблонными**. Это более продвинутые строки, у которых есть дополнительный функционал — например, с их помощью мы можем вставлять в текст различные выражения.

Посмотрим, как вставить переменную внутрь обычной (не шаблонной) строки:

const name = "Петр";

const greeting = "Привет " + name + "!"; *// Привет Петр!*

С помощью обратных кавычек получится короче и удобнее:

const name = "Петр";

const greeting = `Привет ${name}!`;

Тут мы при помощи конструкции в виде доллара и фигурных скобок **${name}** внутрь строки вставили переменную.

Все выражения, заключенные в эту конструкцию, немедленно анализируются и вычисляются, этот процесс называется интерполяцией.

Вставлять в эту конструкцию можно не только переменные, но и вызовы функций, математические и логические выражения, другие строки — обычные и шаблонные.

В шаблонные строки можно также сразу добавлять перенос на новую строку. Используя обычные строки, мы писали следующим образом:

console.log("Первая строка\nВторая строка");

В шаблонных строках можно и так:

console.log(`Первая строка

Вторая строка`);

### Длина строки

В коде бывает нужно узнать длину строки. Например, когда пользователь регистрируется на сайте, нужно проверить, что длина пароля не менее восьми символов, а значит, нам нужно узнать длину строки, которую ввел пользователь.

Для этого у строк в JavaScript есть специальное свойство **length**. Чтобы обратиться к свойству, мы ставим после нашей строки точку и пишем название этого свойства. У строк есть лишь одно свойство — **str.length**, которое показывает длину строки **str**:

console.log("ООО \"ГринКорп\"".length); *// 14*

console.log("".length); *// 0*

НазадК следующему шагу

# Доступ к символу

Допустим, нам надо получить какой-то символ, зная его позицию. Для начала надо знать, что каждый символ в строке имеет свой **индекс** (позицию). Индексация начинается с **0**:

[](https://cs.sberuniversity.online/image/full/full/resize/9edbbb3a-2fef-11ec-8045-0242ac170003)

В примере выше **11** символов, при этом первый символ с индексом **0**, а последний с индексом **10**.

Есть несколько способов получить символ по позиции:

**1.** Используя метод **str.charAt(index)**, где **index** — значение от **0** до длины строки минус **1**. Если **index** выходит за пределы длины, метод вернет пустую строку.

console.log("Привет".charAt(1)); *// р*

console.log("Привет".charAt(100)); *// пустая строка*

**2.** Используя квадратные скобки **[index]**:

console.log("Привет"[1]); *// р*

Этот способ немного отличается от первого. Если **index** выходит за пределы длины, вернется **undefined**:

console.log("Привет"[100]); *// undefined*

На практике чаще используется метод с квадратными скобками, так как он короче в написании. А еще этот способ возвращает **undefined**, а не пустую строку при выходе за пределы, что вызывает меньше ошибок в программах.

НазадК следующему шагу

Изменение регистра

Для изменения регистра есть четыре метода:

* **toLowerCase(),**
* **toUpperCase(),**
* **toLocaleLowerCase(),**
* **toLocaleUpperCase()**.

Метод **str.toLowerCase** приводит все символы к нижнему регистру:

console.log("Привет, Петр".toLowerCase()); *// привет, петр*

Метод **str.toUpperCase** приводит все символы к верхнему регистру:

console.log("Привет, Петр".toUpperCase()); *// ПРИВЕТ, ПЕТР*

Методы с припиской **Locale** принимают на вход опциональное значение**locale**, которое указывает, с учетом какой локали надо преобразовывать (по умолчанию берется из окружения браузера). В большинстве случаев результаты будут одинаковыми, но для некоторых локалей — например, турецкой — правила преобразования отличаются и могут выдать другой результат.

**Обрезание пробелов**

Иногда мы на вход получаем строку, у которой в начале и конце есть ненужные нам пробельные символы (пробел, табуляцию, перенос строки и пр.). Чтобы избавиться от них, есть метод **str.trim()**.

console.log(" Привет, Петр ".trim()) *// "Привет, Петр"*

**Практическое задание 3.2**

**Выберите код, который получился в результате проверки выполненного задания**

W770H

IJ=2

H0$B?

00NY@

DHWIC

Подтвердить выбор

**Практическое задание 3.3**

**Выберите код, который получился в результате проверки выполненного задания**

N08YR

4MVGJ

06SI@

G0DBG

48841

Подтвердить выбор

НазадК следующему шагу

Числа

Мы часто будем работать с числами. В JavaScript они могут быть как целыми, так и с плавающей точкой (дробными):

const integer = 42; *// Целое*

const float = 0.42; *// Число с плавающей точкой*

Среди чисел существует еще несколько специальных значений —**Infinity** и **NaN**.

**Infinity**

Специальное значение Infinity представляет собой бесконечность. Оно может быть как положительным, так и отрицательным (-Infinity).

Работает**Infinity**, как и бесконечность в математике: **любое число + бесконечность = бесконечность**.

Число, деленное на **бесконечность**, дает **0**. Ну а число, деленное на **0**, дает **бесконечность**:

console.log(10 / 0); *// Infinity*

console.log(1 + Infinity); *// Infinity*

console.log(10000 / Infinity); *// 0*

**NaN**

NaN — сокращение от Not A Number, т. е. это не число. NaN появляется при ошибках в вычислении.

Например, если попытаться слово поделить на число, в результате получится **NaN**:

console.log("Hello" / 2); *// NaN*

При делении бесконечности на бесконечность, взятии квадратного корня из **-1** и при других некорректных математических операциях также в результате будет **NaN**:

console.log(Infinity / Infinity); *// NaN*

console.log(0 / 0); *// NaN*

При этом у значения **NaN** есть несколько особенностей:

1. Оно ничему не равно, даже другому **NaN**.
2. Какие бы математические операции мы ни проводили с ним, всегда будет получаться **NaN**.

Можно проверить, является ли значение **NaN**, с помощью глобальной функции**isNaN**:

console.log(isNaN("hello")); *// true*

console.log(isNaN(42)); *// false*

console.log(isNaN("42")); *// false*

**Способы записи числа**

Существует несколько способов записи числа.

const a = 1000000;

const b = 0.00042;

или

const b = .00042;

Для записи очень больших или очень маленьких чисел используется экспоненциальная запись:

const a = 1e6;

const b = 42e-5;

Как правильно понимать данную запись? Мы умножаем число перед **e** на **10** в степени после **e**.

const a = 1e6; *// 1 \* 1000000*

const b = 42e-5; *// 42 \* 0.00001*

Помимо записи в десятичной системе счисления, число также можно записать в двоичной, восьмеричной и шестнадцатеричной системах:

const dec = 42;

const oct = 0o52;

const hex = 0x2a;

const bin = 0b101010;

console.log(dec === 42); *// true*

console.log(oct === 42); *// true*

console.log(hex === 42); *// true*

console.log(bin === 42); *// true*

НазадК следующему шагу

# Операторы

С числами часто нужно выполнять разные операции: складывать, вычитать, умножать и т. д. Каждую такую операцию определяет свой оператор. Например, для сложения есть оператор **+**, а для вычитания**-**.

Значения или переменные, которые участвуют в выражении с оператором, называются операндами.

Например, в выражении **1 + 2** числа **1** и **2** являются операндами, а **+** — это оператор. В результате вычисления выражения мы получим **3**, это будет новым значением этого выражения.

Операторы делятся на несколько категорий:

**1.** **Бинарные** — работают сразу с двумя операндами (слева и справа). Например, оператор плюс (**+**) в выражении **1 + 2** работает с левым операндом **1** и правым **2**.

**2.** **Унарные** — работают с **1** операндом. Например, оператор минус (**-**), который меняет знак:

let a = 1;

let b = -a;

В этом выражении оператор минус (**-**) применяется только к одному операнду **a** и возвращает значение **-1**.

**3. Тернарные** — работают сразу с тремя операндами. В JavaScript существует всего лишь один тернарный оператор. О нем поговорим в следующем разделе.

Каждый оператор имеет свой приоритет выполнения. Чем выше приоритет, тем раньше применится оператор. Таблицу приоритетов можно посмотреть на [MDN.](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Operators/Operator_Precedence)

С большинством операторов мы знакомы со школы: сложение (**+**), вычитание (**-**), умножение (**\***) и деление (**/**), на них не будем заострять внимание. Но есть еще несколько нестандартных:

#### **1. Взятие остатка от деления (%).**

#### **2. Возведение в степень (\*\*).**

#### **3. Унарный плюс (+).**

#### **4. Унарный минус (-).**

#### **5. Инкремент/декремент (++/--).**

#### **6. Оператор присваивания (=).**

#### **7. Оператор «запятая» (,).**

#### **8. Побитовые операторы.**

#### **9. Оператор присваивания, совмещенный с другим оператором.**

НазадК следующему шагу

# Функции

### parseFloat(string)

Часто, например, при получении размеров из стиля мы будем работать со значением, у которого есть единица измерения — например, **1200px**. Такие размеры нужно будет преобразовывать в числа, чтобы далее их обрабатывать. Для этого пригодится функция **parseFloat**.

Она принимает на вход строку, в результате выдает число или **NaN**. Сперва она в переданной строке обрезает пробелы слева, далее посимвольно считывает строку и формирует результат. Когда дойдет до ошибки, вернет сохраненный результат. Если же с самого начала не было цифр, вернет **NaN**.

console.log(parseFloat("42px")); *// 42*

console.log(parseFloat("some text 42")); *// NaN*

console.log(parseFloat("4.2px")); *// 4.2*

console.log(parseFloat("4.2.0")); *// 4.2*

console.log(parseFloat("hello")); *// NaN*

### parseInt(string [, radix])

По работе эта функция похожа на предыдущую **parseFloat**, только возвращает целое число. Вдобавок есть второй необязательный аргумент, который обозначает, из какой системы счисления происходит считывание. По умолчанию **radix** равен **10**.

console.log(parseInt("42px")); *// 42*

console.log(parseInt("some text, 42")); *// NaN*

console.log(parseInt("4.2px")); *// 4*

console.log(parseInt("4.2.0")); *// 4*

console.log(parseInt("hello")); *// NaN*

console.log(parseInt("101010", 2)); *// 42*

### Number.isFinite(testValue)

Этот метод проверяет конечность числа. Для этого есть глобальная функция**isFinite**, но она имеет одно отличие: глобальная функция пытается преобразовать **testValue** в число и только потом сравнивает, а**Number.isFinite** проверяет сразу. Подробнее про преобразования типов мы поговорим в дальнейшем.

console.log(Number.isFinite(Infinity)); *// false*

console.log(Number.isFinite(-Infinity)); *// false*

console.log(Number.isFinite(NaN)); *// false*

console.log(Number.isFinite(0)); *// true*

console.log(Number.isFinite(42)); *// true*

console.log(Number.isFinite("42")); *// false*

console.log(isFinite("42")); *// true*

### Number.isNaN(testValue)

Мы уже рассматривали глобальную функцию**isNaN**. Так же, как и с **Number.isFinite, Number.isNaN** не преобразует в число, а проверяет сразу значение.

console.log(Number.isNaN(NaN)); *// true*

console.log(Number.isNaN(0/0)); *// true*

console.log(Number.isNaN("hello")); *// false*

console.log(isNaN("hello")); *// true*

### numberVariable.toFixed([digits])

Этот метод округляет число до **digits** знаков после точки и преобразует в строку. Если **digits** больше, чем сейчас знаков после точки, дополняет нулями; если меньше — округляет до**digits**. Если **digits** опущен, он по умолчанию будет равен **0**. Аргумент **digits** должен находиться в диапазоне от **1** до **20** включительно. В некоторых реализациях диапазон может отличаться.

let x = 12345.6789;

console.log(x.toFixed()); *// "12346"*

console.log(x.toFixed(2)); *// "12345.68"*

console.log(x.toFixed(10)); *// "12345.6789000000"*

console.log(-12.34.toFixed(1)); *// -12.3*

console.log((-12.34).toFixed(1)); *// "-12.3"*

НазадК следующему шагу

# Глобальный объект Math

В JavaScript встроен глобальный объект **Math** с большим набором математических операций и констант. Здесь мы пройдемся лишь по основным, более подробно можно с ними ознакомиться на [MDN](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Math).

Начнем с округления. У нас есть четыре функции на все случаи жизни.

**Math.ceil(x)** — округление вверх. Вернет ближайшее большее целое число:

console.log(Math.ceil(1.01)); *// 2*

**Math.floor(x)** — округление вниз. Вернет ближайшее меньшее целое число:

console.log(Math.floor(1.01)); *// 1*

console.log(Math.floor(1.99)); *// 1*

**Math.round(x)** — округление до ближайшего целого числа:

console.log(Math.round(1.01)); *// 1*

console.log(Math.round(1.99)); *// 2*

**Math.trunc(x)** — вернет целую часть числа (не поддерживается в браузере Internet Explorer).

Поиск максимального и минимального значения:

**Math.max(x1 …[, xN]) / Math.min(x1, …[, xN])** — из всех полученных аргументов вернут максимальное и минимальное значение соответственно.

Работа со степенями:

**Math.pow(x, exponent)** — возведение **x** в степень **exponent**.

**Math.sqrt(x)** — возвращение квадратного корня **x**.

**Math.cbrt(x)** — возвращение кубического корня **x**.

**Math.exp(x)** — возвращение **e** в степени **x**.

Работа со знаками:

**Math.abs(x)** — возвращение абсолютного значения **x** (отбрасывает знак).

**Math.sign(x)** — возвращение знака числа **x**.

console.log(Math.sign(42)); *// 1*

console.log(Math.sign(-42)); *// -1*

console.log(Math.sign(0)); *// 0*

console.log(Math.sign("hello")); *// NaN*

### Получение случайного числа

В некоторых задачах нам надо работать со случайными числами. Для этого пригодится **Math.random():** он вернет псевдослучайное число от **0** (включая) до **1** (не включая). Далее мы можем масштабировать это значение под текущую задачу. К примеру, можем написать функцию, которая выдает случайное целое число меньше **max**:

function getRandomInt(max) {

return Math.floor(Math.random() \* max);

}

### Константы

В объекте **Math** также есть множество полезных констант.

**Math.E** — основание натурального логарифма.

**Math.LN10** — натуральный логарифм из **10.**

**Math.LN2** — натуральный логарифм из **2**.

**Math.PI** — число Пи.

Кроме того, объект включает множество других констант и функций для работы с логарифмами, тригонометрическими функциями и т. д.

### Практическое задание 3.4

### Выберите код, который получился в результате проверки выполненного задания

QUPB1

ZZ5M1

BE#91

7?G?1

V$@V1

Подтвердить выбор

### Потеря точности

В JavaScript выражения не всегда возвращают то, что мы ожидаем. Один из самых распространенных примеров потери точности:

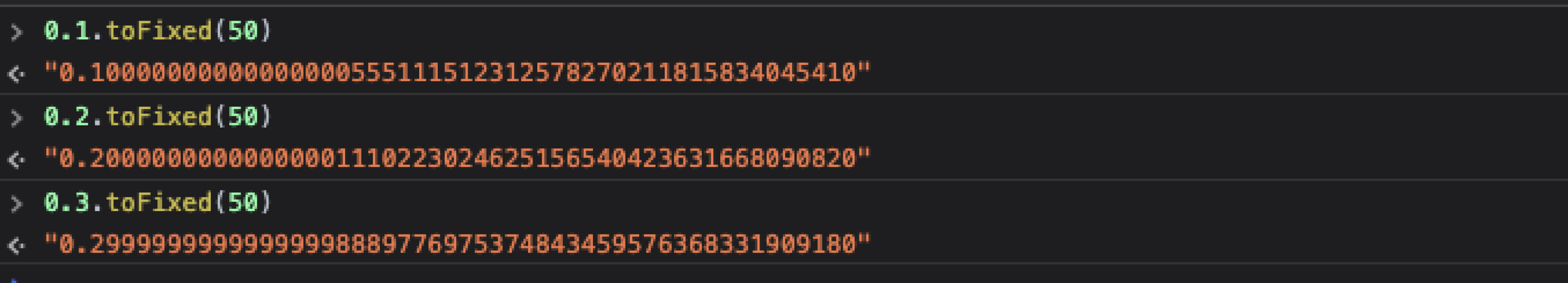
console.log(0.1 + 0.2 === 0.3); *// false*

Вроде бы все правильно написали, но оказывается, что сумма не равна **0.3**? А чему же? Давайте посмотрим:

console.log(0.1 + 0.2); *// 0.30000000000000004*

Что произошло: в JavaScript числа хранятся в формате **IEEE-754**. Для этого используются **64** бита: **1** — для хранения знака, **52** — для цифр и **11** — для положительной десятичной точки. Когда мы переполняем **64** бита, число автоматически становится равным **Infinity**.

Вернемся к нашей проблеме. Наши числа **0.1** и **0.2** в двоичной форме хранятся в виде бесконечной дроби. Их попросту невозможно записать, точно так же, как мы не можем записать число **1/3** в виде конечной десятичной дроби **0.333(3)**. Формат**IEEE-754** использует округление до ближайшего возможного числа. Если мы используем метод **toFixed**, видим, что на каком-то этапе начинаются неточности. И, когда мы складываем наши значения, эти неточности тоже складываются. Именно из-за этого получается, что **0.1 + 0.2** не равно **0.3**.

[](https://cs.sberuniversity.online/image/full/full/resize/9e7022fe-30a2-11ec-b8b7-0242ac160003)

Об этом не стоит забывать, потому что в какой-то момент пользователи могут сильно удивиться. Чтобы проверить, есть ли ошибки, используйте **toFixed**.

console.log((0.1 + 0.2).toFixed(1) === "0.3") *// true*

### Практическое задание 3.5

### Выберите код, который получился в результате проверки выполненного задания

G2IRW

IN3D4

T6V51

@0QVO

$0E95

Подтвердить выбор

### Практическое задание 3.6

### Выберите код, который получился в результате проверки выполненного задания

1D@2

D?1L1

#Y%C1

B49K1

@XB%1

Подтвердить выбор

НазадК следующему шагу

Типы Boolean, null, undefined

**Тип Boolean**

Тип **Boolean** также называют булевым, или логическим, типом. У него всего два значения: **true** и **false**: **true** — если значение истинно, **false** — если ложно. Он важен, и использовать вы его будете довольно часто. Он может использоваться для сохранения состояний чекбоксов в интерфейсе либо же в разных логических выражениях.

Объявить булевую переменную можно напрямую, указав **true**или **false**:

let a = true;

let b = false;

Также булевый тип получается в результате вычисления выражений с операторами сравнения. Рассмотрим, какими они бывают:

**1.**Меньше (**<**)**:**

console.log(1 < 5); *// true*

console.log(1 < '5'); *// true*

**2.** Меньше или равно (**<=**):

console.log(5 <= 5); *// true*

**3.** Больше (**>**):

console.log(5 > 5); *// false*

**4.** Больше или равно (**>=**):

console.log(5 >= 5); *// true*

**5.** Равенство (**==**). Этот оператор сравнивает два операнда и, если они равны, возвращает**true**, иначе — **false**:

console.log(10 == 10); *// true*

console.log(10 == 9); *// false*

console.log(5 == '5'); *// true*

**6.** Строгое равенство (**===**). Этот оператор также сравнивает два операнда и возвращает**true** (если они равны), и **false** (если нет). Однако если прошлый оператор при сравнении числа **5** и строки **'5'** давал **true**, в случае строгого равенства результат будет **false** т.к. у них разный тип:

console.log(10 === 10); *// true*

console.log(10 === 9); *// false*

console.log(5 === '5'); *// false*

**7.** Отрицание (**!**). Оператор отрицания еще называют **НЕ**. Он возвращает обратное булевое значение:

console.log(!true); *// false*

console.log(!false); *// true*

**8.** Не равно (**!=**) — комбинация оператора равенства и отрицания. Он проверяет, что два значения не равны друг другу, и возвращает **true**, если не равны, а иначе — **false**:

console.log('A' != 'B'); *// true*

console.log(5 != '5'); *// false*

**9.** Строгое неравенство (**!==**) — комбинация строгого неравенства и отрицания:

console.log('A' !== 'B'); *// true*

console.log(5 !== '5'); *// true*

**Тип null**

Тип **null** означает отсутствие чего-либо. Это специальный тип, в котором есть только одно значение. Его используют, чтобы указать, что в переменной «*ничего нет*» — например, пустую корзину покупок можно обозначить **null**. При приведении к булевому типу мы получим **false**, а при приведении к числу — **0**. Пример создания **null**:

const value = null;

**Тип undefined**

Когда значение не было задано, мы имеем дело с типом **undefined**:

let x;

console.log(x); *// undefined*

Некоторые путают **null**и**undefined**, но надо научиться их различать:

* **undefined** — значение не задано;
* **null** — значение задано, но оно пустое либо неизвестно.

Например:

let login;

console.log(login); *// undefined, логин ещё не задали*

login = null; *// null, значение задали и оно пустое, значит логина нет*

**undefined** также можно присвоить:

let x = undefined;

Однако обычно так не делают, ведь это противоречит его смыслу.

НазадК следующему шагу

# Объекты

Объекты являются главным и самым сложным типом в JavaScript. Остальные типы составляют отдельную категорию — **примитивы**.

**Примитивы** просты и неизменяемы. Тут важно понимать разницу: мы можем перезаписать то, что присвоено переменной, но само значение мы не можем изменить. Иначе говоря, можно переменной, в которой хранится **2**, присвоить значение **3**, но мы не можем изменить **2** и сказать, что она теперь равна **3**.

Вернемся к**объекту**: это некая обертка, внутри которой может находиться множество свойств. Свойством является пара **ключ: значение**, где ключ (также называемый имя свойства) — строка или символ, а значение может быть чем угодно. Ключи должны быть уникальны. Если в качестве ключа была передана не строка и не символ, интерпретатор преобразует ключ в строку.

Есть несколько способов создать объект:

let myObj = new Object();

и

let myObj = {};

Второй способ короче и быстрее: с его помощью можно сразу задать значение свойствам. Свойство задается так: объявляется ключ, после ставится двоеточие, затем идет значение. И все эти наборы (ключ, двоеточие, значение) разделяются запятыми:

let myObj = {

key1: 'first key',

key2: 222,

myThirdKey: true,

"key with spaces": {},

12: null

};

### Доступ к значениям

Доступ к значениям свойств осуществляется двумя способами:

**1. Через точку:**

console.log(myObj.key1); *// 'first key'*

**2. Через квадратные скобки.**

С их помощью мы можем получать доступ по ключам, у которых есть пробел. Кроме того, появляется возможность использовать переменную в качестве ключа.

console.log(myObj["key with spaces"]); *// {}*

let keyVariable = "12";

console.log(myObj[keyVariable]); *// null*

Если обратимся к несуществующему свойству, получим **undefined**:

console.log(myObj.test); *// undefined*

Квадратные скобки помогают задать значения свойствам при создании объекта, если имя свойства находится в другой переменной:

let keyVariable = "someKey";

let myObj2 = {

[keyVariable]: "test"

}

console.log(myObj2.someKey); *// "test"*

### Практическое задание 3.7

### Выберите код, который получился в результате проверки выполненного задания

4NN=1

ZB%C1

B7%F1

PV211

3W#D1

Подтвердить выбор

### Практическое задание 3.8

### Выберите код, который получился в результате проверки выполненного задания

9M501

21$K1

G5@@1

@2@:1

?UC01

Подтвердить выбор

### Добавление/изменение свойств

Чтобы **добавить**или **изменить** свойство, достаточно обратиться к объекту по ключу и записать в него значение:

myObj.test = true;

console.log(myObj.test); *// true*

myObj.test = 0;

console.log(myObj.test); *// 0*

### Удаление свойств

Для удаления свойства существует оператор **delete**:

delete myObj.test;

console.log(myObj.test); *// undefined*

Встречались ли мы уже в этом курсе с какими-то объектами? Да, к примеру, **Math** — объект, в котором в качестве значений есть много полезных функций и констант. Или **alert**.

Функции внутри тоже являются объектами, а, как уже известно, объекты — важная и сложная составляющая языка JavaScript. Классы, массивы, другие коллекции и даже функции наследуются от объектов. Но более подробно с объектами мы познакомимся позже.

НазадК следующему шагу

Типы BigInt и Symbol

**BigInt**

Тип **number** имеет ограниченный диапазон **[- (253 - 1); 253 - 1]. BigInt** позволяет записывать числа произвольной длины. Для объявления переменной типа **BigInt** необходимо в конец числа добавить **n** или же обернуть в **BigInt(value)**:

let bigInt = 2n;

let bigInt2 = BigInt("10");

* Из важных особенностей стоит отметить:

**1.** В арифметических выражениях нельзя смешивать **number** и **BigInt** (а в логических выражениях, например при сравнении, можно).

**2.** При строгом сравнении с **number** всегда будет **false**:

console.log(2 == 2n); *// true*

console.log(2 === 2n); *// false*

**3.** При делении всегда происходит округление в нижнюю сторону (отбрасывается дробная часть).

**Symbol**

Символ является уникальным идентификатором. Он используется в качестве ключей для свойств объекта. Чтобы создать символ, надо использовать обертку **Symbol(description)**. Причем каждый раз будет создаваться уникальное значение, вне зависимости от того, что придет на вход.

let key1 = Symbol("test");

let key2 = Symbol("test");

let key3 = Symbol("test");

console.log(key1 == key2); *// false*

console.log(key2 == key3); *// false*

Есть возможность также получить описание символа из свойства **description**:

console.log(key1.description); *// test*

Но иногда нам нужно, чтобы по одинаковому **description** всегда был один и тот же символ. Это можно сделать с помощью глобального реестра символов. Используя **Symbol.for(description)**:

let key1 = Symbol.for("test"); *// test нет в глобальном реестре - создает и добавляет*

let key2 = Symbol.for("test"); *// test есть в глобальном реестре - достает из него*

let key3 = Symbol.for("test");

console.log(key1 === key2); *// true*

console.log(key2 === key3); *// true*

НазадК следующему шагу

Оператор typeof

Теперь мы знаем обо всех типах. Иногда непонятно, с каким типом мы сейчас работаем: допустим, мы получили что-то с сервера или получили в функции аргумент с неопределенным типом. Тип переменной распознает **typeof**. Его можно использовать двумя способами: как унарный оператор **typeof** **x** и как функцию **typeof(x)**. Результатом будет строка, в которой будет указан тип:

console.log(typeof 12); *// number*

console.log(typeof 1n) *// bigint*

console.log(typeof "12"); *// string*

console.log(typeof true); *// boolean*

console.log(typeof Symbol()); *// symbol*

console.log(typeof undefined); *// undefined*

console.log(typeof {a: 1}); *// object*

console.log(typeof alert);*// function*

console.log(typeof null); *// object*

* Тут может возникнуть два вопроса:
  1. У нас же всего восемь типов, и типа **function** нет, почему же тогда **typeof alert** вернул**function**? Удивление закономерно: функции не являются отдельным типом. И, как мы узнали ранее, они унаследованы от объекта, однако для них в **typeof** сделано исключение — он выводит**function**. Просто для удобства.
  2. У нас есть тип **null**, тогда почему **typeof** **null** выводит**object**? Это официально признанная ошибка, допущенная при создании JavaScript. Было предложение ее исправить, но его отклонили в пользу совместимости. Поэтому, когда вы хотите узнать, является ли переменная объектом, следует использовать следующее условие:
  3. if (typeof myObject === "object" && myObject !== null) {
  4. ...

}

* **Помните об этих отступлениях от правил, они пригодятся в работе.**

**Практическое задание 3.9**

**Выберите код, который получился в результате проверки выполненного задания**

5X4X1

@PSW1

QR6Z1

84XV1

FQD41

Подтвердить выбор

НазадК следующему шагу

# Преобразования типов

Преобразование значений из одного типа данных в другой также называют приведением типов. Ранее мы упоминали про различные приведения одних типов к другим, теперь разберем их подробнее.

### Преобразования в строку

Есть несколько способов создать строку из переменной другого типа:

**1.** Использовать конструкцию **String(variable)**:

let str = String(null); *// "null"*

str = String(12); *// "12"*

**2.** Вызвать метод **toString()**. Практически у каждого типа есть метод **toString**, который приведет значение к строке. Исключениями являются **null** и **undefined**:

let num = 100;

str = num.toString(); *// "100"*

null.toString() *// Ошибка!*

#### **Правила преобразования**

Практически из всех типов преобразования происходят очевидным образом: **100** будет преобразовано в **"100", 100n** в **"100"**, **false**в**"false"**, **undefined** в **"undefined", null** в **"null"**.

Исключениями являются **Object**и**Symbol**. Простой объект вне зависимости от содержания возвращает **"[object Object]"**, а **Symbol** возвращает **"Symbol(description)"**:

console.log(String({})); *// "[object Object]"*

console.log(String(Symbol("some text"))); *// "Symbol(some text)"*

#### **Примечание**

### Неявные преобразования

Иногда преобразования в коде совершаем не мы, а сам язык JavaScript — например, при конкатенации. Такие преобразования принято называть неявными:

let position = 2;

console.log("Ваша позиция " + position); *// Ваша позиция 2*

При сложении со строкой всегда неявно происходит следующее преобразование:

console.log("Ваша позиция " + String(position));

### Что выведется в консоль после выполнения console.log("Ваша позиция: " + { position: 1 })?

Ваша позиция: { position: 1 }

Ваша позиция: [object Object]

Ваша позиция: { }

Ваша позиция: NaN

Подтвердить выбор

По ходу обучения мы будем показывать, где могут происходить неявные преобразования.

НазадК следующему шагу

# Преобразования в числа

Мы уже рассматривали несколько примеров преобразования строки в число: **parseInt, parseFloat**. Помимо этого, есть еще способы, но уже актуальные для всех типов данных, а не только для строк:

**1.** Конструкция **Number(variable)** позволяет получить число из строк и других типов:

let num = Number("100"); *// 100*

let boolNum = Number(false); *// 0*

Если значение, передаваемое в **Number**, не может быть преобразовано в число, в результате получается **NaN**:

let a = Number({}); *// NaN*

let b = Number("Hello"); *// NaN*

let c = Number(undefined); *// NaN*

**2.** Выражения с операторами сравнения. Ранее мы рассказывали об операторах сравнения **==, !=, >, >=, <, <=**. Если в таких операциях участвует число, второй операнд также приводится к числу:

console.log(1 < '5'); *// true*

console.log(100 > '50'); *// true*

console.log('42' == 42); *// true*

Мы также рассказывали про операторы строгого равенства (**===**) и неравенства (**!==**). Они не используют приведение типов, поэтому дают другой результат:

console.log('42' === 42); *// false*

**3.** В разделе с операторами также упоминали унарный плюс (**+**). Эта запись эквивалентна **Number(variable**), только короче:

let num = +"100"; *// 100*

Унарный плюс имеет один из самых высоких приоритетов. Рассмотрим пример:

let a = '4';

let b = '2';

console.log(+a + +b); *// 6*

В примере сначала происходит преобразование значений **a** и **b** в число, и только потом они складываются. Эту запись можно немного изменить, чтобы она легче читалась:

console.log(Number(a) + Number(b));

На практике унарный плюс почти не используют и предпочитают ему преобразование с помощью **Number(...)**, так как оно более понятно.

* Важно!

Унарный плюс нельзя применять к **BigInt**. Чтобы преобразовать **BigInt**в **Number**, необходимо использовать **Number(value)**.

**4.** Математические операции. Если у математического оператора какой-либо из операндов не является числом, он практически всегда неявно преобразуется в число с помощью **Number(variable)**:

console.log('4' / 2); *// 2*

console.log(4 / '2'); *// 2*

console.log(4 \* '2'); *// 8*

console.log('4px' / 2); *// NaN*

Есть одно исключение: бинарный оператор плюс (**+**), когда один из операндов является строкой. В этом случае будет произведена конкатенация строк:

console.log(4 + '2'); *// "42"*

console.log('4' + 2); *// "42"*

console.log(4 + 2 + ' - some text'); *// "6 - some text"*

console.log(4 + (2 + ' - some text')); *// "42 - some text"*

Мы также ранее рассказывали про различия функции **isFinite** и **Number.isFinite, isNaN** и **Number.isNaN**. Глобальные функции **isFinite** и **isNaN** не используют приведение типов, а **Number.isFinite** и **Number.isNaN** используют:

console.log(Number.isFinite("42")); *// false*

console.log(isFinite("42")); *// true*

console.log(Number.isNaN("hello")); *// false*

console.log(isNaN("hello")); *// true*

### Правила преобразования других типов данных в число

#### **Строки**

#### **undefined при преобразовании в число в результате дает NaN**

#### **null при преобразовании в число даёт 0**

#### **Булевые значения**

#### **Символы**

#### **BigInt**

#### **Объекты преобразуются в NaN**

### Практическое задание 3.10

### Выберите код, который получился в результате проверки выполненного задания

EQ4Z2

FKW<2

O6WF2

NUTV2

5D?D2

Подтвердить выбор

### Практическое задание 3.11

### Выберите код, который получился в результате проверки выполненного задания

U8ZA2

7M4P2

1W3A2

X5D#2

A#U02

Подтвердить выбор

### Практическое задание 3.12

### Выберите код, который получился в результате проверки выполненного задания

$U4D1

KD731

WRVG1

@VA2

#VQV

Подтвердить выбор

НазадК следующему шагу

# Преобразования в булевое значение

Преобразовать какое-либо значение в булевое можно с помощью оборачивания в **Boolean([value])**. Если **value** опущено или равно **0, null, false, NaN, undefined** или пустой строке (**""**), результатом будет **false**. В остальных случаях результатом будет **true**:

console.log(Boolean()); *// false*

console.log(Boolean(10)); *// true*

console.log(Boolean(0)); *// false*

console.log(Boolean("0")); *// true*

console.log(Boolean("")); *// false*

console.log(Boolean(" ")); *// true*

console.log(Boolean("false")); *// true*

console.log(Boolean({})); *// true*

Есть еще один способ преобразования. Мы вскользь упомянули, что оператор отрицания приводит к булевому значению и после возвращает обратный результат. С помощью двойного отрицания можно добиться более короткой записи преобразования:

console.log(!!10); *// true*

console.log(!!0); *// false*

console.log(!!"0"); *// true*

console.log(!!""); *// false*

console.log(!!" "); *// true*

console.log(!!"false"); *// true*

console.log(!!{}); *// true*

НазадК следующему шагу

ЗАДАЧИ

ЗАДАЧИ

ЗАДАЧИ

ЗАДАЧИ

ЗАДАЧИ

ЗАДАЧИ

# Как добавить условия в код

Иногда надо выполнить код по какому-то условию. Для этого существует оператор **if**. Он может быть записан двумя способами.

**1.** Без скобок будет выполнена одна команда:

if (condition) alert("hello");

**2.** С фигурными скобками (**{}**) — целый блок:

if (condition) {

let text = "hello";

alert(text);

}

Чаще всего используют второй вариант, так как очень редко выполняется всего одна команда.

Команда или блок будут выполнены, если **condition** после приведения к булевому значению будет равен**true**. О преобразованиях разных типов в булевое значение мы подробно говорили в прошлом разделе.

Примеры преобразований:

if (0) {

*// после преобразования в булевый тип, получится false,*

*// следовательно этот блок никогда не будет выполнен*

}

if ("hello") {

*// после преобразования строки "hello" в булевый тип, получится true,*

*// следовательно этот блок всегда будет выполнен*

}

Иногда надо сделать разветвление и выполнить один код, если выполняется условие, и другой — если не выполняется. Например, если у пользователя есть деньги на счете, оплатить заказ, а если их нет — предложить пополнить счет. Для этого можно написать два противоположных **if**, однако короче и читабельнее будет использовать оператор **else**.

Этот оператор идет сразу после блока **if**:

if (age >= 18) {

console.log('Вы совершеннолетний!')

} else {

console.log('Вы несовершеннолетний')

}

Можно также комбинировать условия с помощью **else if**:

if (age < 14) {

console.log('Еще нет паспорта')

} else if (age < 18) {

console.log('Уже есть паспорт, но все еще несовершеннолетний')

} else {

console.log('Уже совершеннолетний')

}

### Практическое задание 4.1

### Выберите код, который получился в результате проверки выполненного задания

%VZV1

T<TA1

I?EX1

DUA11

DM1W1

Подтвердить выбор

НазадК следующему шагу

Сложные условия с помощью логических операторов

Часто условия становятся более сложными — например, возраст меньше **18** и больше **14**. Можно, конечно, написать один блок **if** внутри другого блока **if**, например:

if (age < 18) {

if (age > 14) {

console.log('Мне 15-17 лет')

}

}

Однако, если добавится еще одно условие, вложенных блоков станет три и снизится их читабельность: чем больше вложенностей, тем сложнее понимать, что к чему относится.

Чтобы решить подобные проблемы, используйте более короткую запись для сложных условий — с помощью логических операторов.

С одним логическим оператором мы уже знакомы, это отрицание (**!**). Разберем еще два: И (**&&**) и ИЛИ (**||**).

Оператор И (**&&**) по очереди получает значения операндов и приводит их к булевому типу. Если встречает **false**, он останавливает проверку и возвращает это значение:

console.log(true && false); *// false*

console.log(0 && 2); *// 0*

console.log(null && "hello" && true); *// null*

Если же ни один операнд не равен **false**, оператор вернет последний:

console.log(1 && 2); *// 2*

console.log(true && true); *// true*

console.log("hello" && true); *// true*

Ранее показанный пример с помощью оператора **&&** можно переписать так:

if (age < 18 && age > 14) {

console.log('Мне 15-17 лет')

}

С помощью этого оператора можно также выполнять блоки кода:

age >= 18 && console.log('Вы совершеннолетний!');

Минус в том, что код в блоке выше плохо читается, поэтому лучше использовать**if (…) {}**.

Оператор ИЛИ (**||**) также по очереди получает значения операндов и приводит их к булевому типу. Однако, если встречается**true**, он останавливает проверку и возвращает это значение:

console.log(false || true); *// true*

console.log(1 || 2); *// 1*

console.log(0 || 2); *// 2*

console.log(0 || null || "hello"); *// "hello"*

console.log(null || "hello" || true); *// "hello"*

Если же ни один операнд не будет равен **true**, оператор вернет последний:

console.log(false || false); *// false*

console.log(0 || null); *// null*

console.log(null || false || 0); *// 0*

Все логические операторы можно комбинировать, а еще важно знать приоритет (порядок) их выполнения:

1. **!** (Отрицание).
2. **&&** (И).
3. **||** (ИЛИ).

Допустим, нам надо вывести сообщение о том, закрыто ли сейчас заведение. Заведение открыто по будням с **10** до **18**, а по выходным с **10** до **24**. Как это можно записать:

if (hour >= 10 && (hour < 18 || !isWeekday)) {

console.log('Мы открыты!')

} else {

console.log('Мы закрыты!')

}

Из-за разного приоритета у операторов мы заключили второе условие в скобки.

Если бы мы это не сделали, код условия был бы таким:

hour >= 10 && hour < 18 || !isWeekday

А значит, получился бы следующий порядок:

1. Выполняем отрицание**isWeekday,**
2. Получаем **hour >= 10 && hour < 18,**
3. Далее выполнится оператор **ИЛИ**с **!isWeekday.**

В этом случае условие некорректно, т. к. проверка **|| !isWeekday** стала относиться к **hour >= 10 && hour < 18**, а не только к **hour < 18**. Такой код выдаст сообщение **Мы открыты в любое время в выходной день**.

**Практическое задание 4.2**

**Выберите код, который получился в результате проверки выполненного задания**

WDND

MO#?1

FOJ2

FGQQ

?0%@1

Подтвердить выбор

**Практическое задание 4.3**

**Выберите код, который получился в результате проверки выполненного задания**

I4Q71

Z?SK1

1@HA1

H4QV1

?4DE1

Подтвердить выбор

**Практическое задание 4.4**

**Выберите код, который получился в результате проверки выполненного задания**

@D1D1

BK8B1

U@8J1

1@HP1

0@?W1

Подтвердить выбор

НазадК следующему шагу

# Значения по умолчанию

Иногда мы не уверены, есть на данный момент что-то в переменной или в свойстве объекта, но при этом должны уже обрабатывать значение. В таких случаях используют значения по умолчанию:

if (!username) {

username = "Гость";

}

console.log(`Привет, ${username}!`);

if (myObj.count === undefined) {

myObj.count = 0;

}

myObj.count++;

Можно также сократить запись:

username = username || "Guest";

И еще сократить:

username || "Guest";

НазадК следующему шагу

# Тернарный оператор

Часто приходится присваивать значение переменной по условию:

let status;

if (hour < 10 || hour > 18) {

status = 'closed';

} else {

status = 'opened';

}

Это довольно громоздко. В JavaScript есть тернарный оператор, позволяющий сократить код, синтаксис у него такой:

**Переменная = условие ? значение1 : значение2**

Переменной будет присвоено**значение1**, если условие после приведения типа будет равно **true**, иначе будет присвоено **значение2**. Вот так с помощью тернарного оператора будет выглядеть пример со статусом:

let status = (hour < 10 || hour > 18) ? 'closed' : 'opened';

Так же, как и в **if-else**, в тернарном операторе можно создавать вложенность:

let allowedText = age < 14

? 'Нет доступа'

: age < 18

? 'Ограниченный доступ'

: 'Полный доступ'

Это равносильно следующему:

let allowedText;

if (age < 14) {

allowedText = 'Нет доступа';

} else if (age < 18) {

allowedText = 'Ограниченный доступ'

} else {

allowedText = 'Полный доступ';

}

Отметим, что комбинация тернарных операторов — неудачный вариант (антипаттерн), поскольку с каждым уровнем вложенности становится сложнее читать такие выражения.

### Практическое задание 4.5

### Выберите код, который получился в результате проверки выполненного задания

L@NI1

1QZR1

?GWP1

WUBD1

Q14I1

Подтвердить выбор

НазадК следующему шагу

Конструкция switch

Оператор **switch** используется в случаях, когда необходимо выполнить разные блоки кода в зависимости от значения выражения.

Рассмотрим его структуру:

switch (expression) {

case value1:

*// Если результат выражения равен value1, то выполнится этот код*

[break;]

case value2:

*// Если результат выражения равен value2, то выполнится этот код*

[break;]

...

case valueN:

*// Код, соответствующий значению valueN*

[break;]

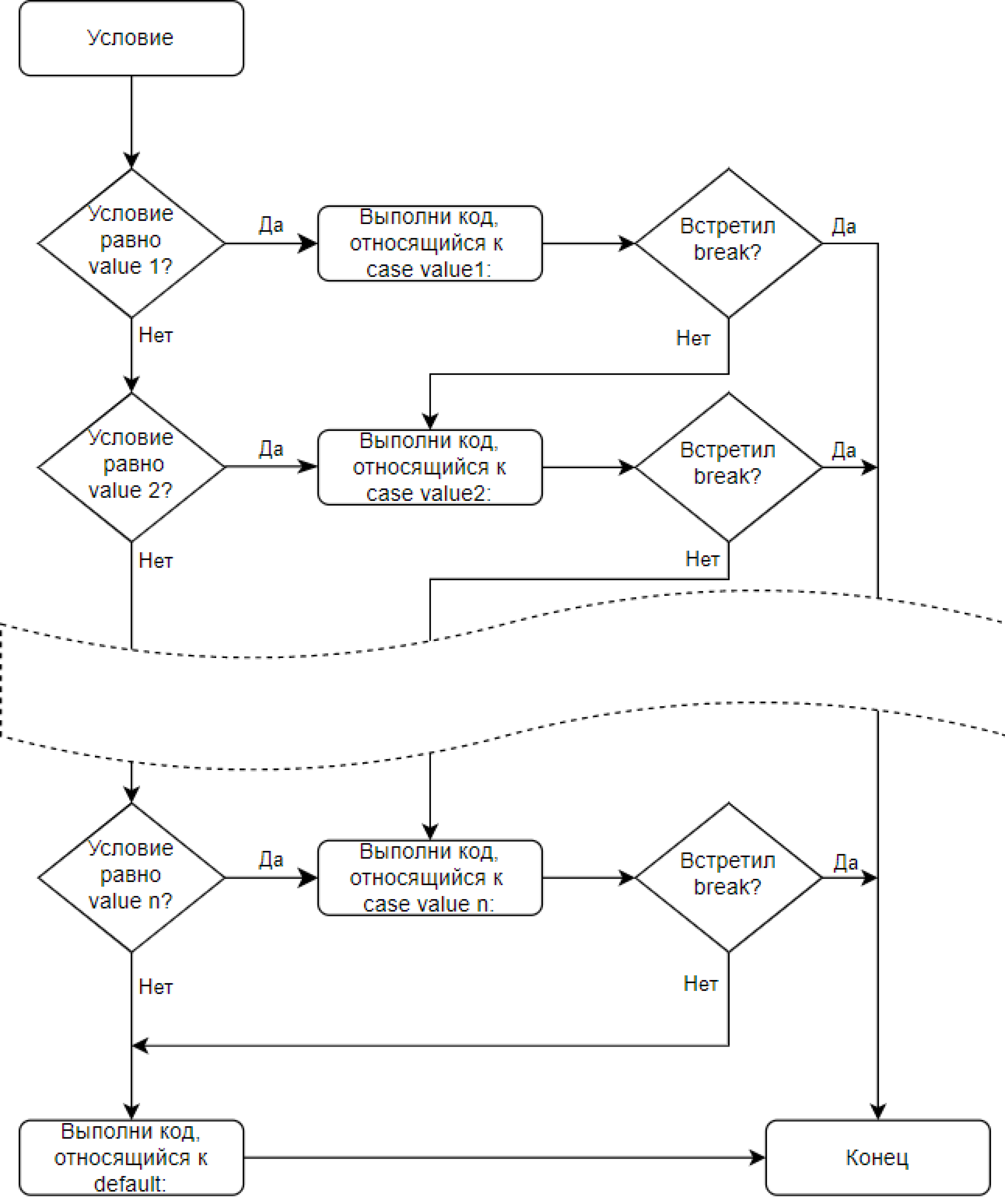
default:

*// Если не будет ни одного совпадения, то выполнится этот код*

[break;]

}

Блок-схема **switch** выглядит так:

[](https://cs.sberuniversity.online/image/full/full/resize/4862574c-3660-11ec-9d85-0242ac160002)

У **switch** есть несколько важных особенностей:

**1.** Используется строгое сравнение, поэтому в коде ниже мы всегда будем попадать в секцию **default**:

switch ('1') {

case 1:

console.log('Вы ввели 1'); *// этот код не выполнится*

break;

default:

console.log('Вы ввели не 1')

}

**2.** Если код попал в тело **case**, он будет выполнять все следующие инструкции **case-default** до тех пор, пока он не встретит **break**.

switch (1) {

case 0:

console.log('Вы ввели 0');

case 1:

console.log('Вы ввели 1');

case 2:

console.log('Вы ввели 2');

default:

console.log('Вы ввели больше 2')

}

Результатом этого кода будут три сообщения: **Вы ввели 1**, **Вы ввели 2**, **Вы ввели больше 2**. Если бы мы перед**default** поставили **break**, последнее сообщение не вывелось бы.

**3.** Возникает как следствие из предыдущего пункта возможность объединить несколько **case**:

switch (expression) {

case value1:

case value2:

*// Если результат выражения равен value1 или value2, то выполнится этот код*

break;

default:

*// Если не будет ни одного совпадения, то выполнится этот код*

}

* Сравнение **switch** и **if-else**:
  1. **switch** быстрее находит место, откуда надо начать выполнять код.
  2. Помимо того, что **switch** быстрее находит, откуда надо начать выполнять код, он лишь один раз вычисляет выражение, переданное на вход. Если мы в**if** не сохраним значение выражения, при каждом сравнении будут происходить вычисления. И чем сложнее вычисления, тем более медленным будет **if**.
  3. В **switch** легче объединять блоки.
  4. В **switch** можно легко дополнять одни блоки другими.
  5. Условия в **switch** легче читаются.
  6. В **if** мы можем делать сложные условия на основе нескольких значений, в то время как **switch** лишь строго сравнивает с одним значением.
  7. **if-else** лучше подходит для логических выражений, в то время как **switch** больше подходит, когда необходимо сравнить результат выражения с фиксированными значениями.

**Практическое задание 4.6**

**Выберите код, который получился в результате проверки выполненного задания**

E:<71

BE021

26Y01

OWLN1

DD2J1

Подтвердить выбор

**Практическое задание 4.7**

**Выберите код, который получился в результате проверки выполненного задания**

B4R01

S2DL1

2DWI1

AN0G1

FSP2

Подтвердить выбор

**Практическое задание 4.8**

**Выберите код, который получился в результате проверки выполненного задания**

XRCW1

IW54

QEWW

UL7;1

T%@N

Подтвердить выбор

**Практическое задание 4.9**

**Выберите код, который получился в результате проверки выполненного задания**

VDCT1

4%NH1

16Q@1

927%1

H@G$1

Подтвердить выбор

НазадК следующему шагу

# Тестирование по разделу

В шаге имеется обязательный к прохождению элемент

Пройдите тестирование по изученным ранее разделам **Первая программа на JavaScript, Переменные, комментарии, Типы данных и Условный оператор.**

[Начать тест](https://sberuniversity.online/programs/16083/item/671963/launch?is_iframe_mode=1?registration=fff94b1d-b58d-483c-aa0b-95bc003fd02c)

НазадК следующему шагу

ЗАДАЧИ

ЗАДАЧИ

ЗАДАЧИ

ЗАДАЧИ

ЗАДАЧИ

# Циклы

***Сергей, руководитель группы:*** «В твоем проекте частенько будут ситуации, когда один и тот же по сути код потребуется дублировать несколько раз. Хороший специалист постарается упростить и сократить свой код. Посмотрим, как это делается!»

### Зачем нужны циклы?

Рано или поздно вам придется столкнуться с задачей написать однотипный код несколько раз. Например, написать метод **console.log** пять раз, записывая в аргументе значение от **1** до **5**. Реализация без использования циклов выглядела бы так:

console.log(1);

console.log(2);

console.log(3);

console.log(4);

console.log(5);

На первый взгляд, решение выглядит актуальным, но что делать, когда необходимо вывести не от **1** до **5**, а, например, от**1** до **500**? Вряд ли у вас хватит сил писать для каждого значения метод **console.log()**. Да и выглядеть это будет странно. Как раз для таких целей используют циклы.

**Циклы** — повтор определенных команд, пока заданное условие принимает значение **true**. Каждый такой повтор называется **итерацией** цикла.

НазадК следующему шагу

Цикл while

Цикл **while** имеет следующую конструкцию:

while (условие) {

*// Тело цикла*

}

Тело цикла будет повторяться, пока условие не примет значение **false**. Если же условие сразу равно **false**, тело цикла ни разу не выполнится.

* Важно отметить

Условие тут работает, как и в условных операторах. Другими словами, нам необязательно передавать туда булевое значение. Интерпретатор неявно за нас преобразует условие в следующую конструкцию: **Boolean(условие)**.

Реализация примера, где мы получаем значения от **1** до **500**, используя **while**, выглядит так:

1. let i = 1;

2. while (i <= 500) {

3. console.log(i++);

4. }

В первой строке задается переменная **i**, которая выполняет техническую роль, являясь неким счетчиком в цикле.

Во второй объявляется цикл **while**, где в скобках указывается условие цикла. В нашем случае, пока значение переменной **i** меньше или равно **500**, весь описанный код внутри блока **{ }** будет повторяться. Традиционно код внутри цикла называют **телом цикла**.

На третьей строке выполняется тело цикла, а именно: метод **console.log(i++)**, который с каждой итерацией цикла увеличивает значение переменной **i** на **1**, меняя ее значение для проверки условия и выполнения следующей итерации.

Аналогично инкременту мы можем использовать декремент. Например, составим цикл, который будет выполняться, пока переменная **i** не будет равна численному значению **0**. Тут вспомним, что необязательно передавать булевое значение в условие, поэтому просто запишем в нем переменную. Когда она станет равной **0, Boolean(0)** даст **false**:

let i = 500;

while (i) {

console.log(i--);

}

Если у нас в теле цикла лишь одна команда, мы также можем опустить фигурные скобки:

let i = 500;

while (i) console.log(i--);

* Важно!

Если бы в этих примерах мы не указали инкремент или декремент, цикл выполнялся бы бесконечное количество раз. Это чревато зависанием скрипта, поэтому при написании кода всегда удостоверьтесь, что условие рано или поздно станет равным **false**.

НазадК следующему шагу

# Цикл do-while

Если цикл **while** сначала проверяет условие и только потом выполняет тело цикла, конструкция **do-while**, наоборот, сначала выполняет тело цикла и только потом проверяет условие. Таким образом, мы гарантируем, что тело выполнится хотя бы один раз.

Выглядит цикл так:

do {

*// тело цикла*

} while

Для примера рассмотрим ту же задачу на вывод от **1** до **500**:

let i = 1;

do {

console.log(i++);

} while (i <= 500)

Тут также надо помнить, что условие само может преобразовываться в **Boolean**, а скобки можно опустить, если в теле цикла всего одна строка.

НазадК следующему шагу

# Цикл for

Пожалуй, из всех циклов именно **for** является наиболее часто встречающимся в JavaScript. Его чаще всего используют, когда нужен счетчик. Конструкция данного цикла выглядит так:

for ([Инициализация]; [Условие]; [Действие после итерации]) {

*// Тело цикла*

}

Каждый элемент конструкции в кавычках является необязательным. Если мы опускаем какой-то элемент внутри скобок **for(...)**, разделитель (**;**) должен остаться в любом случае. Если же мы опускаем тело цикла, также ставим символ **;** сразу же после **for(...)**.

Рассмотрим тот же пример, но уже используя цикл **for**:

for (let i = 1; i <= 500; i++) {

console.log(i)

}

Можно убрать некоторые элементы цикла:

let i = 1;

for (; i <= 500;) {

console.log(i++)

}

Полученный цикл аналогичен **while (i <= 500)**.

Не забывайте про оператор «запятая» (**,**), который позволяет выполнить несколько действий:

for (let i = 1, factorial = 1; i <= 10; i++, factorial\*=i) {

console.log(i, factorial);

}

В этом примере были выведены значения факториала от **1** до **10**.

НазадК следующему шагу

# Оператор break

Мы выяснили, что цикл итерирует наш код, пока заданное условие внутри аргумента равно **true**. В тот момент, когда условие возвращает булевый тип **false**, цикл завершает итерацию команд. Помимо заданного условия, мы также можем принудительно остановить цикл, используя оператор **break** в теле цикла.

Рассмотрим задачу, когда переменная **word** будет менять и выводить текущее значение в консоль каждую итерацию до того момента, когда она примет нужное нам значение:

1. let word = "string";

2.

3. for (let i = 0; i < 5; i++) {

4. word += "\_1" ;

5. if (word == "string\_1\_1\_1") break;

6. console.log(word);

7. }

В первой строке мы задали для переменной **word** значение **string**, а далее в цикле **for** изменяем и выводим ее в консоль.

В первой итерации цикла мы получим от метода **console.log(word)** значение **string\_1**, во второй — **string\_1\_1** и т. д   
В пятой строке описано условие **if**, при котором мы принудительно завершим цикл, используя директиву **break**, если переменная **word** примет значение **string \_1\_1\_1**. Результатом выполнения будут две полные итерации и частичная третья.

Если условие **if** в теле цикла никогда не выполнится, тогда сработает условие, описанное в аргументе **for**. В этом случае мы получим пять итераций.

НазадК следующему шагу

# Оператор continue

Оператор **continue** — подобие **break**, правда, он не принудительно останавливает цикл, а помогает перейти к следующей итерации.

Рассмотрим аналогичный пример с переменной**word**:

1. let word = "string";

2.

3. for (let i = 0; i < 5; i++) {

4. word += "\_1" ;

5. if (word == "string\_1\_1\_1") {

6. continue;

7. }

8. console.log(word);

9. }

Оператор **continue** описывается там же, где и **break** в прошлом примере. В одной из итераций, когда переменная **word** примет значение **string\_1\_1\_1**, цикл принудительно пропустит метод **console.log(word)** и перейдет к следующей итерации. Результатом данного примера будут полные первая, вторая, четвертая и пятая итерации и третья до **continue**.

Стоит понимать, что будут пропущены только те команды, которые были объявлены после оператора **continue**, например:

1. let word = "string";

2.

3. for (let i = 0; i < 5; i++) {

4. word += "\_1" ;

5. console.log(word);

6. if (word == "string\_1\_1\_1") {

7. continue;

8. }

9. }

Тут мы поменяли местами восьмую и пятую-седьмую строчки кода. Сначала будет вызван метод **console.log(word)** и только потом условный оператор **if**. Несмотря на то, что мы описали условие с директивой **continue**, итерация с сообщением **string\_1\_1\_1** появится в консоли.

### Практическое задание 5.1

### Выберите код, который получился в результате проверки выполненного задания

MKT71

FSPL1

4N1F1

P$D31

S#@K1

Подтвердить выбор

### Практическое задание 5.2

### Выберите код, который получился в результате проверки выполненного задания

3X?11

KM@41

4KO?1

WAXF1

UNXO1

Подтвердить выбор

### Практическое задание 5.3

### Выберите код, который получился в результате проверки выполненного задания

MOOD2

TH1A2

V9B%2

VC%M2

KNDV2

Подтвердить выбор

НазадК следующему шагу

# ПРИМЕРЫ РЕШЕНИЯ ЗАДАНИЙ

Для получения доступа вам необходимо пройти материал

[Выберите код, который получился в результате проверки выполненного задания](https://sberuniversity.online/programs/16083/item/671751?is_iframe_mode=1)

НазадК следующему шагу

ЗАДАЧИ

ЗАДАЧИ

ЗАДАЧИ

ЗАДАЧИ

ЗАДАЧИ

# Зачем нужны функции и как их создать

Часто в разных местах кода необходимо выполнять одно и то же действие. Чтобы не дублировать код, были созданы**функции**.

Функция — набор инструкций. Достаточно единожды описать алгоритм действий, а дальше, используя их, мы можем вызывать эту функцию в разных местах кода.

Функции решают сразу несколько проблем:

1. Избавляют от дублирования кода.
2. Позволяют исправить код лишь в одном месте (теле функции), а не везде, где был использован алгоритм, если он поменяется.
3. Разбивают одну большую задачу на подзадачи, тем самым упрощая восприятие кода.

### Способы объявления функции

#### **Function declaration**

Первый способ объявления функции называется **function declaration**. В общем виде выглядит так:

1. Ключевое слово **function**.
2. Пробел и название функции.
3. Скобки, внутри которых через запятую перечислены принимаемые параметры (если параметров нет, просто пустые скобки). Эти параметры еще называют **аргументами** функции.
4. Тело функции в скобках **{…}**.

Например:

function printHello(name) {

console.log(`Привет, ${name}!`);

}

printHello("Саша");

У такого способа объявления функций есть одна важная особенность: вы можете вызывать функцию еще до ее объявления.

Дело в том, что интерпретатор незаметно для нас в первую очередь собирает все функции, объявленные таким способом, и инициализирует (создает) их. Этот процесс называется **hoisting** (всплытие).

printHello('Саша'); *// все еще выведется "Привет, Саша!"*

function printHello(name) {

console.log(`Привет, ${name}!`);

}

#### **Function expression**

Второй способ объявления функции называется**function expression**. Такими функциями считаются все созданные внутри выражения функции:

let printHello = function printHello(name) {

console.log(`Привет, ${name}!`);

}

printHello('Саша');

Помимо способа объявления, у функций есть еще несколько особенностей:

1. Они могут называться как угодно, но обращаться к ним можно только через имя переменной:

let printHello = function someFunctionName(name) {

console.log(`Привет, ${name}!`);

}

printHello('Саша'); *// "Привет, Саша!"*

someFunctionName('Саша'); *// Ошибка! someFunctionName не определен*

2. Они могут вообще не иметь имени, то есть быть анонимными:

let printHello = function(name) {

console.log(`Привет, ${name}!`);

}

printHello('Саша'); *// "Привет, Саша!"*

В следующих разделах разберем подробнее разницу в именованных и анонимных функциях и рассмотрим примеры, когда стоит использовать тот или иной вид.

3. Они не обладают эффектом всплытия:

printHello('Саша'); *// Ошибка! printHello не определен*

let printHello = function someFunctionName(name) {

console.log(`Привет, ${name}!`);

}

4. Они могут вызываться на месте. Такие функции называются самовызывающиеся:

(function() {

console.log("Привет!");

})();

*// Привет!*

Как было сказано выше, **function expression** — все функции, которые созданы внутри выражения. Поэтому, начав со скобок, отрицания или любого другого оператора, мы превращаем нашу запись в **function expression**. После объявления такой функции мы можем моментально ее вызвать:

!function(name) {

console.log(`Привет, ${name}!`);

}("Саша");

*// Привет, Саша!*

### Практическое задание 6.1

### Выберите код, который получился в результате проверки выполненного задания

PQ7Y2

0QIE2

F1W92

9WXW2

ZSJW2

Подтвердить выбор

НазадК следующему шагу

Локальные и внешние переменные

Внутри функций можно использовать все инициализированные внешние переменные:

const name = "Саша"

printHello(); *// "Привет, Саша!"*

function printHello() {

console.log(`Привет, ${name}!`)

}

Однако нельзя использовать переменные, которые были созданы после вызова функции:

printHello(); *// Ошибка из-за того, что name еще не определен*

const name = "Саша"

function printHello() {

console.log(`Привет, ${name}!`)

}

При этом мы можем объявлять переменные внутри самой функции, но за ее пределами они не будут видны!

function printHello(name) {

const hello = "Привет, ";

console.log(hello + name);

}

printHello("Саша"); *// "Привет, Саша"*

console.log(hello); *// Ошибка из-за того, что hello не определен*

Иногда названия аргументов или переменных внутри функции могут также пересекаться с названиями внешних переменных.

* Запомните

В функциях **внутренние** переменные имеют приоритет над **внешними**. Говоря иначе, внутри функции внешняя переменная с таким же названием попросту блокируется. Мы теперь будем всегда ссылаться на внутреннюю.

Подробнее об этой механике мы поговорим в следующем разделе, а пока рассмотрим пример:

function printHello(name) {

const hello = "Привет, ";

console.log(hello + name);

}

let hello = "Hello, ";

printHello("Саша"); *// "Привет, Саша"*

console.log(hello); *// "Hello, "*

В этом примере мы внутри функции с помощью ключевого слова **const** создали новую переменную, поэтому внешняя переменная **hello** для нас заблокировалась и никак не поменялась.

Теперь уберем **const**:

function printHello(name) {

hello = "Привет, ";

console.log(hello + name);

}

let hello = "Hello, ";

printHello("Саша"); *// "Привет, Саша"*

console.log(hello); *// "Привет, "*

В этом случае мы перезаписали переменную. Внутри функции не создавалась переменная с именем **hello**, поэтому обращение шло к внешней переменной.

НазадК следующему шагу

# Аргументы функции

Аргументы — то, что функция получает на вход. В примерах выше мы их уже использовали. Их может быть либо сколько угодно, либо вообще не быть. Перечисляются аргументы через запятую.

function sumOfThree(a, b, c) {

console.log(a + b + c)

}

В данном примере **a, b, c** являются аргументами.

### Количество аргументов

Мы можем узнать количество предполагаемых аргументов функции с помощью свойства **length**, примененного к самой функции:

function sumOfThree(a, b, c) {

console.log(a + b + c)

}

console.log(sumOfThree.length); *// 3. !!! Обратите внимание, тут не вызывается функция!*

### Объект arguments

В функциях мы можем обращаться к аргументам не напрямую, а через специальный объект **arguments**, который доступен лишь внутри функции. Он содержит в себе все аргументы, но доступ к ним открывается по их номеру (начиная с **0**). Особенность объекта в том, что через него мы можем получить все аргументы, даже если их передали больше, чем мы описали в функции:

function printArgs(a, b) {

console.log(arguments[0]);

console.log(arguments[1]);

console.log(arguments[2]);

}

console.log(printArgs.length); *// 2*

printArgs(1, 2, 3); *// 1, 2, 3*

Как видите, мы описали у функции лишь два аргумента. Свойство **length** функции тоже показывает два. Но через **arguments** мы смогли получить доступ к третьему аргументу функции

У **arguments** также есть свойство**length**, которое показывает количество переданных аргументов.

function printArgsLength(a) {

console.log(arguments.length);

}

console.log(printArgsLength.length); *// 1*

printArgsLength(); *// 0*

printArgsLength(1); *// 1*

printArgsLength(1, 1); *// 2*

### Значения по умолчанию

Иногда бывают ситуации, когда мы хотим сделать какой-то аргумент необязательным, при этом присвоить ему значение по умолчанию. Как мы можем это сделать:

function printHello(username) {

if (username === undefined) {

username = 'гость';

}

console.log(`Привет, ${username}!`);

}

Вышло довольно громоздко. Попробуем сократить.

function printHello(username) {

username = (username === undefined) ? 'гость' : username;

console.log(`Привет, ${username}!`);

}

Уже лучше. Но вдруг у нас много аргументов со значениями по умолчанию? Для каждого расписывать такую строчку? К счастью, есть другой способ:

function printHello(username = 'гость') {

console.log(`Привет, ${username}!`);

}

Такая запись практически идентична предыдущей. Почему практически? Меняется логика получения количества предполагаемых аргументов функции через **printHello.length**. Дело в том, что данное свойство показывает количество необходимых аргументов. При этом, если встречается какой-то необязательный параметр, подсчет останавливается:

function firstTest(a, b, c = 1) {}

console.log(firstTest.length); *// 2*

function secondTest(a, b = 1, c) {}

console.log(secondTest.length); *// 1*

В этих примерах у нас всего три аргумента, и только один необязательный, но значение **length** отличается из-за позиции, на которой располагается аргумент со значением по умолчанию.

НазадК следующему шагу

Возвращаемое значение функции

По умолчанию функции в результате своей работы возвращают **undefined**. Но мы можем возвращать свои значения. Делается это через оператор **return**. Функция вернет значение, которое написано после него:

function square(num) {

return num \* num;

}

let four = square(2);

У оператора**return** есть несколько особенностей:

1. Когда интерпретатор дойдет до него, функция вернет значение и моментально завершит свое выполнение, даже если после**return** есть какой-то код.
2. Можно писать **return** без значения — тогда функция просто моментально завершится и вернет **undefined**.
3. В одной функции может быть неограниченное количество **return**.

Зная об этом, мы можем написать функцию с выходом по условию:

function square(num) {

if (isNaN(num)) {

return;

}

*// Если num не число, то интерпретатор до этой строки уже не дойдет*

return num \* num;

}

square("test"); *// undefined*

square(2); *// 4*

В первом случае **isNaN("test")** показал**true**, и мы попали в пустой**return**. Во втором случае **isNaN(2)** показал **false**, и мы попали в **return num \* num**. Как видите, в функции может быть сколько угодно **return**, но, дойдя до любого из них, обработка сразу же завершается.

**Практическое задание 6.2**

**Выберите код, который получился в результате проверки выполненного задания**

1HDK1

%D9@1

620C1

?LNJ1

MT351

Подтвердить выбор

НазадК следующему шагу

# Стрелочные функции

Есть еще один способ создать функцию с помощью Function expression — стрелочных функций.

Их несомненный плюс — они короче. У стрелочных функций отсутствует ключевое слово **function**, имя функции (т. е. они всегда анонимные), а после объявления аргументов ставится стрелка (**=>**):

const square = (num) => {

return num \* num;

}

Запись у стрелочных функций может быть еще короче: к примеру, если функция умещается в одно выражение:

const someVariable = () => expression

Такая запись всегда возвращает результат выражения. Другими словами, эта запись эквивалентна:

const someVariable = () => {

return expression;

}

Переписав **square** короче, мы получим следующее:

const square = (num) => num \* num;

Как мы видим, функция стала намного короче.

НазадК следующему шагу

# Колбеки

Колбек — функция, переданная в качестве аргумента в другую функцию. Они часто используются в асинхронных функциях (о них речь чуть позже) и в функциях, которые обрабатывают результат и должны вызвать одну функцию при успехе и другую при неудаче.

function stringToNumber(str, onOk, onError) {

if (typeof str !== 'string') {

onError("Введенное значение не строка!");

return;

}

if (!isNaN(str) && !isNaN(parseFloat(str))) {

onOk(Number(str));

} else {

onError("Введенная строка не является корректным числом")

}

}

function printNums(number) {

for (let i = 0; i <= number; i++) {

console.log(i);

}

}

stringToNumber("3", printNums, function handleError(message) {

console.error(message);

})

В данном примере мы написали функцию, которая пытается преобразовать строку в число и при успехе вызывает **onOk**, а при ошибках **onError**.

Вызвав функцию, в качестве **onOk** мы передали существующую функцию **printNums**. Обратите внимание: мы не вызываем функцию, а передаем ее. А в качестве **onError** мы прямо на месте написали функцию **handleError**, которая принимает и выводит ошибки.

### Практическое задание 6.3

### Выберите код, который получился в результате проверки выполненного задания

JWM%1

WO3Y1

Y82W1

ZC001

BQ3G1

Подтвердить выбор

### Практическое задание 6.4

### Выберите код, который получился в результате проверки выполненного задания

VVWT2

W16;2

YWF92

K@UE2

XO402

Подтвердить выбор

### Практическое задание 6.5

### Выберите код, который получился в результате проверки выполненного задания

VWVH2

NFJ12

LG0D2

OKM@2

I06:2

Подтвердить выбор

### Практическое задание 6.6

### Выберите код, который получился в результате проверки выполненного задания

U2<;2

WP#@2

2M9O2

%4IJ2

$YJQ2

Подтвердить выбор

НазадК следующему шагу

# ПРИМЕРЫ РЕШЕНИЯ ЗАДАНИЙ

Для получения доступа вам необходимо пройти материал

[Выберите код, который получился в результате проверки выполненного задания](https://sberuniversity.online/programs/16083/item/671779?is_iframe_mode=1)

НазадК следующему шагу

ЗАДАЧИ

ЗАДАЧИ

ЗАДАЧИ

ЗАДАЧИ

# Создание, чтение и изменение объектов

Как уже говорилось ранее, объекты — самый главный тип. Именно на их основе строятся все структуры данных в JavaScript. Давайте сначала вспомним основы.

const myObj = {

['my-name']: "Ваня"

};

console.log(myObj["my-name"]); *// Ваня*

console.log(myObj.my-name); *// Ошибка!*

У объектов есть свойства.

Свойство — это пара **ключ: значение**. Через ключ мы получаем доступ к значению.

Создать объект можно двумя способами:

const myObj1 = new Object();

const myObj2 = {};

Чаще всего объекты создаются вторым способом, через фигурные скобки.

Получить доступ к свойствам объектов также можно несколькими способами — через точку и квадратные скобки:

const myObj = {

greetings: "Hello, World!"

};

console.log(myObj.greetings);

console.log(myObj["greetings"]);

Удобнее пользоваться доступом к свойству объекта через точку, так как это меньше символов, не нужно писать кавычки. Однако если свойство объекта содержит символы пробела, тире, то через точку его уже нельзя получить:

const myObj = {

['my-name']: "Ваня"

};

console.log(myObj["my-name"]); *// Ваня*

console.log(myObj.my-name); *// Ошибка!*

При создании объектов можно использовать значения переменных:

function createUser(firstname, lastname) {

return {

name: firstname + lastname

};

}

Если название переменной соответствует названию ключа, мы можем сократить запись:

function createUser(firstname, lastname) {

return {

firstname, // эквивалентно firstname: firstname,

lastname // lastname: lastname

}

}

console.log(createUser("Вася", "Пупкин"));

// {firstname: "Вася", lastname: "Пупкин"}

Объекты могут иметь вложенность. Операторы доступа (точка и квадратные скобки) имеют самый высокий приоритет, они сразу же возвращают значение. Поэтому, если мы хотим обратиться к свойству вложенного объекта, нам достаточно просто продолжить цепочку обращений:

const myObj = {

count: 1,

texts: {

greetings: "Hello, World!"

}

};

console.log(myObj.texts.greetings);

console.log(myObj["texts"].greetings);

Обратите внимание, что можно комбинировать разные способы обращения к объекту.

Если свойства не существует, при попытке чтения JavaScript вернет **undefined**:

const config = {}

console.log(config.images\_links); *// undefined*

console.log(config.images\_links.logo); *// ошибка! Попытка обращения к undefined.logo*

Используя квадратные скобки, можно обратиться к свойству с помощью переменной. В качестве ключа будет использовано значение переменной:

const myObj = {

greetings: "Hello, World!"

};

const variable = "greetings";

console.log(myObj.greetings); *// "Hello, World!"*

console.log(myObj["greetings"]); *// "Hello, World!"*

console.log(myObj[variable]); *// "Hello, World!"*

console.log(myObj.variable); *// undefined. Свойства variable нет*

С помощью переменной мы можем еще и создать объект с динамическим ключом:

const variable = "greetings";

const myObj = {

[variable]: "Hello, World!"

};

console.log(myObj.greetings); *// "Hello, World!"*

console.log(myObj["greetings"]); *// "Hello, World!"*

console.log(myObj[variable]); *// "Hello, World!"*

console.log(myObj.variable); *// undefined. Свойства variable нет*

Если надо создать объект, в котором ключи будут иметь пробелы, данные ключи надо заключить в кавычки. Обращаться через точку к такому свойству уже не получится, только квадратные скобки:

const config = {

"project version": "5.0.0"

};

console.log(config["project version"]); *// "5.0.0"*

console.log(config.project version); *// ошибка*

Еще одна важная особенность: если ключ является не символом и не строкой, он преобразуется в строку:

const order = {

1: "user1",

2: "user2",

3: "user3"

};

console.log(order[1]); *// "user1"*

console.log(order["1"]); *// "user1"*

### Изменение объектов

Чтобы изменить объект, надо обратиться к свойству таким же образом, как и считывали значение, и использовать с ним оператор, который изменяет значение (инкремент, декремент, присваивание). Если такого свойства не существует, он будет добавлен в объект:

const config = {};

config.app\_title = "Hello, World!";

config["project version"] = "5.0.0";

config.version++; *// config = {version: NaN}*

console.log(config);

*/\* {*

*app\_title: "Hello, World!",*

*"project version": "5.0.0",*

*version: NaN*

*} \*/*

Удалить свойство можно с помощью оператора **delete**:

const banlist = {

user1: {

until: "01.01.2020",

reason: ""

},

};

delete banlist.user1;

console.log(banlist); *// {}*

НазадК следующему шагу

Чтение ключей объекта

Иногда нам надо провести какие-либо вычисления на основе всего объекта. И, если объекты будут динамическими, мы не будем знать все ключи. Да и количество записей может быть огромным. Что же тогда делать? Для решения этой проблемы существует цикл **for-in**. Он имеет следующую конструкцию:

for (let key in object) {

*// тело цикла*

}

Каждую итерацию цикла в переменную **key** будет попадать ключ объекта.

* Важно!

JavaScript не гарантирует порядок вывода ключей. В разных браузерах он может отличаться.

Давайте рассмотрим использование этого цикла на примере. Допустим, наш объект представляет из себя корзину. В ней лежит несколько товаров, и нам надо посчитать сумму:

const basket = {

12: {

id: 12,

title: "HDMI-кабель",

cost: 1500,

count: 2

},

10: {

id: 10,

title: "Телевизор",

cost: 50000,

count: 1

}

}

let sum = 0;

for (let key in basket) {

sum += basket[key].cost \* basket[key].count;

}

console.log(sum); *// 53000*

НазадК следующему шагу

# Копирование объектов

### Этот раздел очень важен!

Объект отличается от примитивов тем, что он хранится и копируется по ссылке. Что это значит? Представьте, что объект — это коробка. Когда вы создаете объект, вы создаете новую коробку и кладете ее куда-то в память, а в переменную вы помещаете не саму коробку, а указание, где именно она находится в памяти. Поэтому, когда вы в другую переменную копируете объект, на самом деле вы копируете ссылку на место в памяти. Рассмотрим подробнее.

Допустим, мы создали объект:

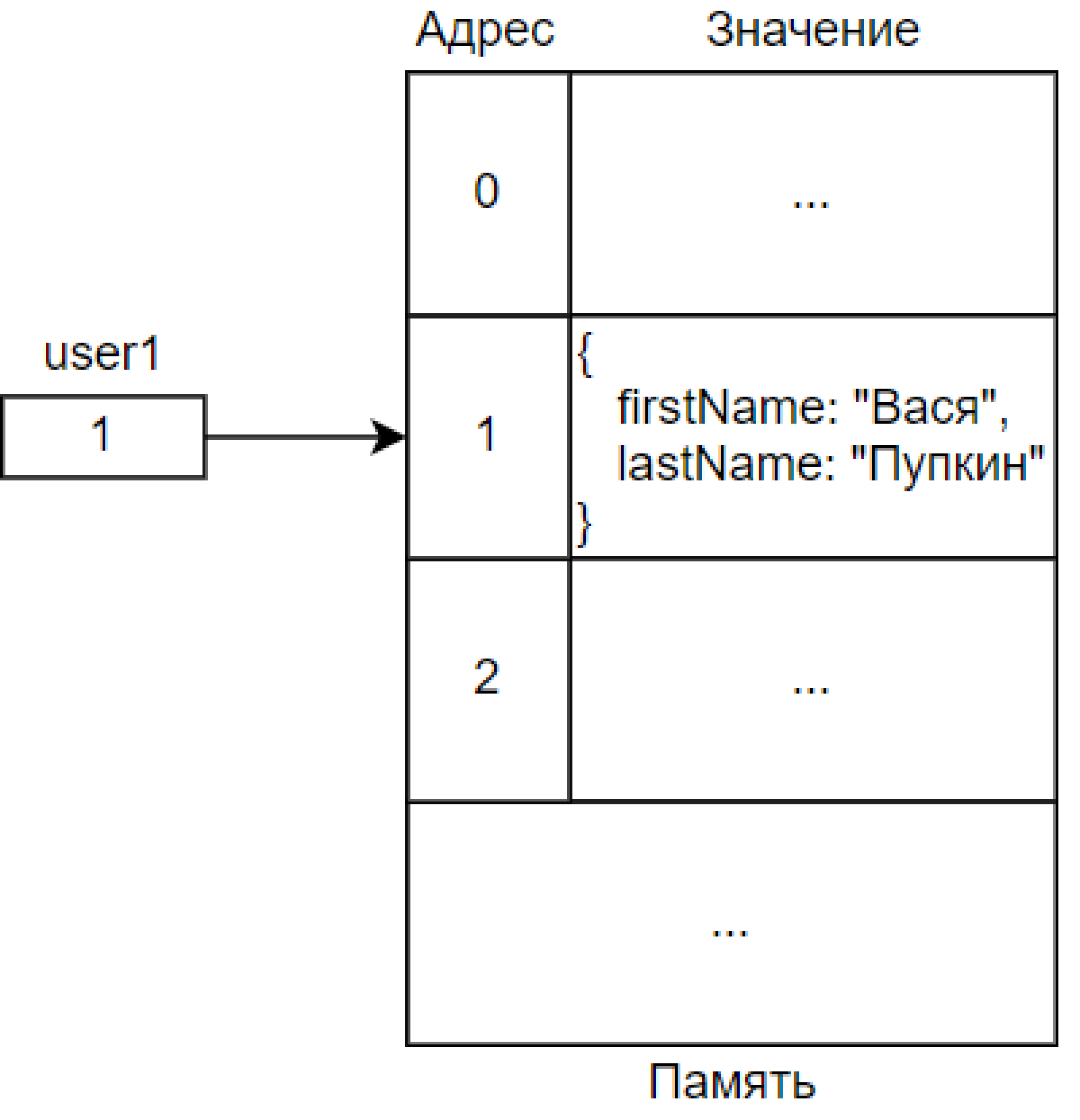
const user1 = {

firstName: "Вася",

lastName: "Пупкин"

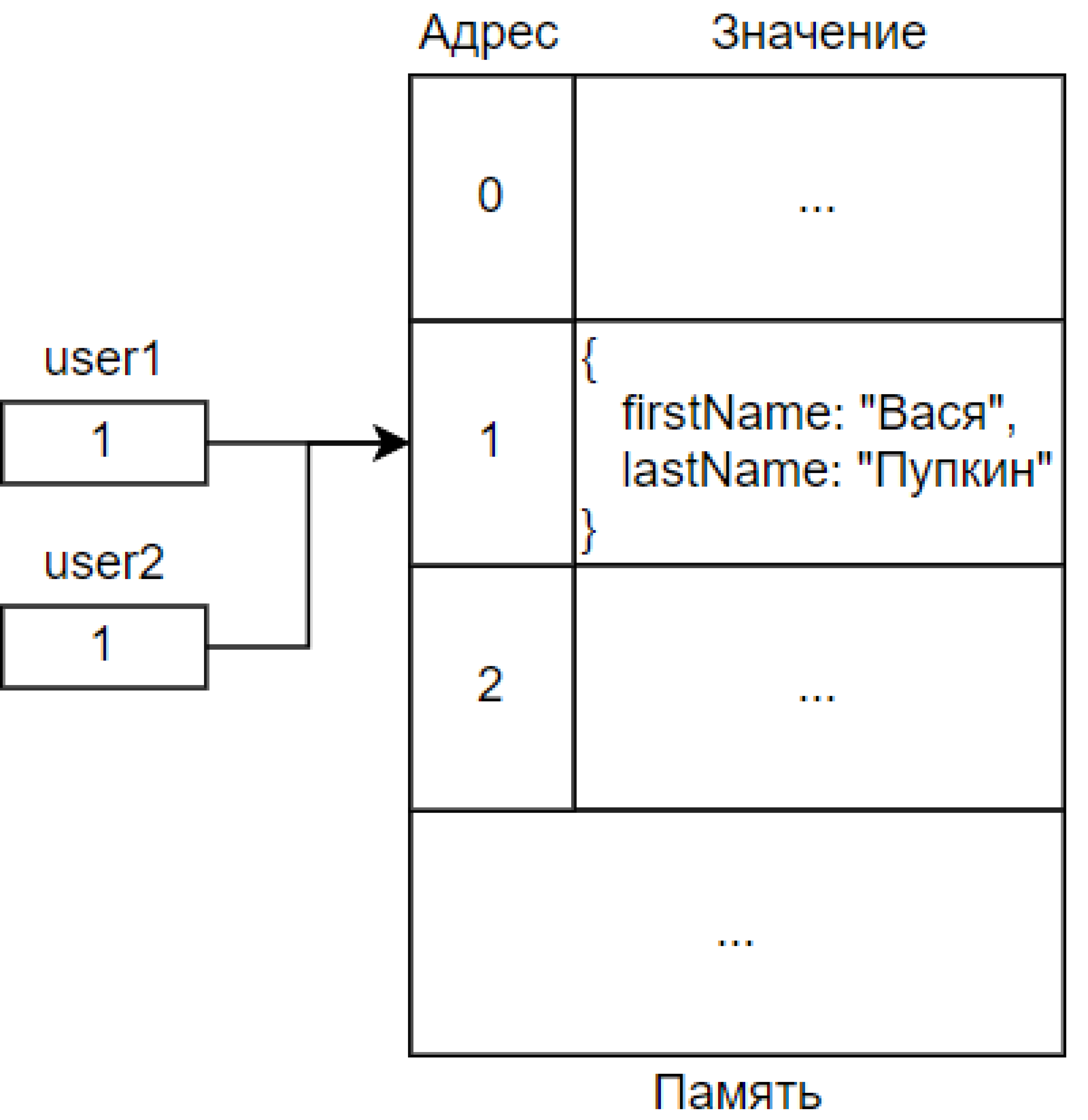
};

В этот момент где-то в случайном месте в памяти записался объект, а переменной присвоили адрес, по которому он записался. При обращении к переменной JavaScript смотрит на записанный адрес и достает по нему значение из памяти.

[](https://cs.sberuniversity.online/image/full/full/resize/cc8de9ec-367d-11ec-9157-0242ac170004)

Когда мы копируем объект в другую переменную, мы не создаем новый объект и располагаем по новому адресу, а просто копируем текущий адрес.

const user2 = user1;

[](https://cs.sberuniversity.online/image/full/full/resize/d4fedef6-367d-11ec-80aa-0242ac170003)

Поэтому, если мы изменим что-то в **user1**, это же изменение будет видно и в **user2**:

const user1 = {

firstName: "Вася",

lastName: "Пупкин"

};

const user2 = user1;

user2.age = 18;

console.log(user1);

*// {firstName: "Вася", lastName: "Пупкин", age: 18}*

При любых действиях с объектами мы работаем со ссылками (изменение, сравнение, передача в функцию и т. д.). Поэтому, если мы создадим новый объект с такими же полями и сравним их, из-за того, что сравнивается не содержимое объектов, а ссылки на них, мы получим **false**:

const user1 = {

firstName: "Вася",

lastName: "Пупкин"

};

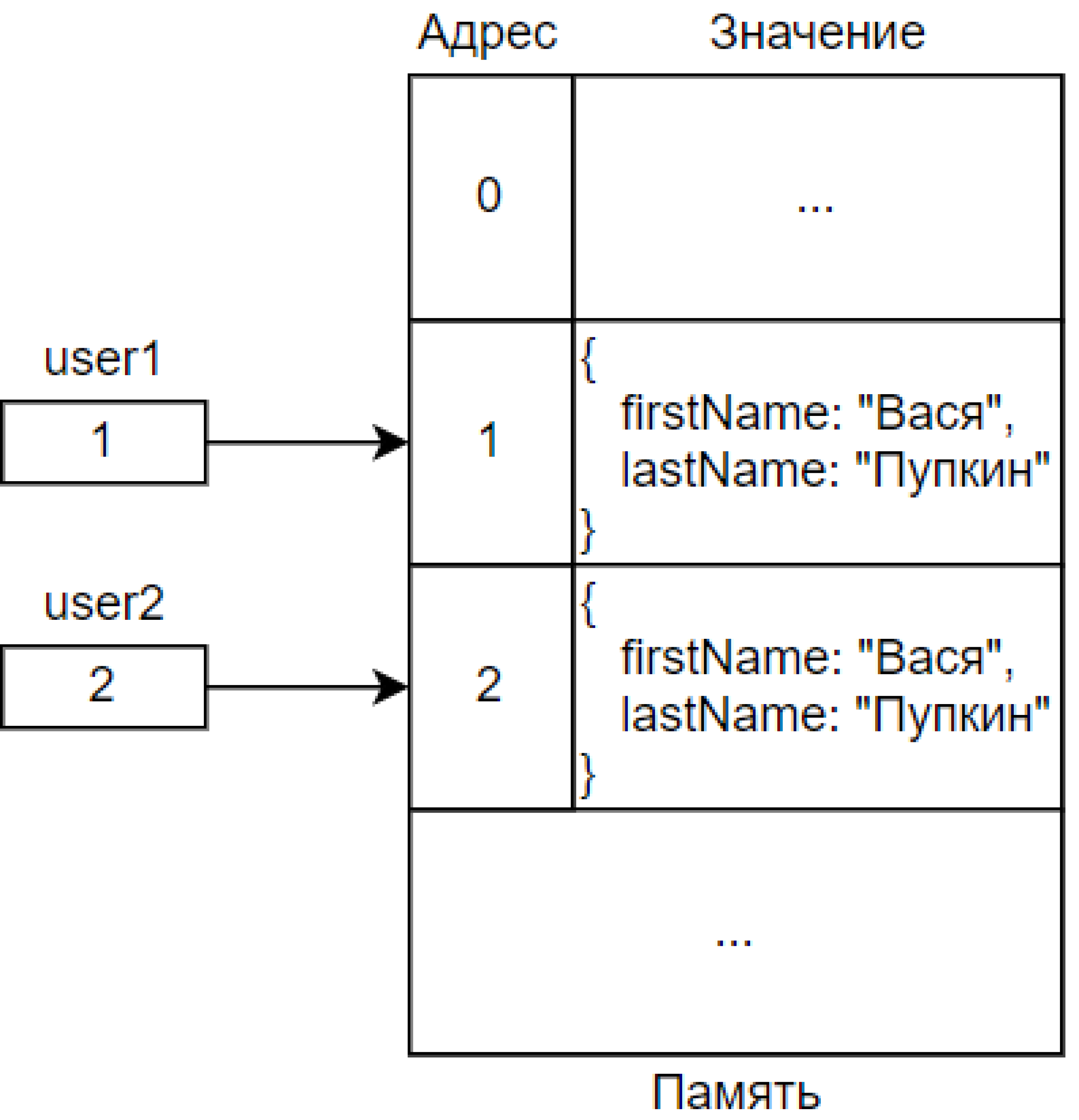
const user2 = {

firstName: "Вася",

lastName: "Пупкин"

};

console.log(user1 == user2); *// false*

[](https://cs.sberuniversity.online/image/full/full/resize/05a2db20-367e-11ec-bd72-0242ac170003)

Одна из частых ошибок новичков в том, что они боятся изменять объект, который записан в переменную, созданную через **const**. Этого не стоит бояться. В переменной хранится адрес. Изменяя содержимое объекта, мы никак не меняем адрес. Поэтому во всех примерах объекты были созданы с **const** и прекрасно работали, когда мы удаляли, изменяли или создавали свойства. Если же легче представлять на коробках, запомните следующее: «Мы можем как угодно менять содержимое коробки, но из-за этого коробка не перестанет быть собой».

* Важно!

Все вышеописанное не относится к примитивам. Это относится только к объектам и другим сущностям на основе объектов (классы, функции и т. д.).

Давайте под конец разберем два примера, где потребуется проверить самих себя и ответить на вопрос, что выводится в консоль.

**Пример 1:**

var obj = { a: 1 };

function f1(o) {

o.a = 5;

}

f1(obj);

console.log(obj); *// ???*

#### **Спойлер!**

**Пример 2:**

var obj = { a: 1 };

function f2(o) {

o = { hello: 1 };

}

f2(obj);

console.log(obj); *// ???*

#### **Спойлер!**

НазадК следующему шагу

# Что такое лексическое окружение

Мы уже знаем: если создать переменную внутри функции, извне она не будет видна. Давайте разберемся, как JavaScript работает с переменными.

Когда интерпретатор попадает в функцию, он создает скрытый объект **LexicalEnvironment**, в котором будут храниться все переменные.

Как заполняется этот объект: при вызове функции создается новый объект, в него сразу же записываются аргументы и инициализируются переменные, созданные с помощью **var**, а далее по мере выполнения записываются и переменные, объявленные через **let/const**:

getHelloMessage("Петр");

function getHelloMessage(name) {

*// LexicalEnvironment = {name: "Петр", hello: undefined}*

*// Именно из-за того, что в LexicalEnvironment переменные let/const попадают только после того, как до них дойдет интерпретатор, мы получим ошибку, если попробуем тут обратиться к result. Этого свойства просто еще нет в LexicalEnvironment*

var hello = "Привет, ";

*// LexicalEnvironment = {name: "Петр", hello: "Привет, "}*

const result = hello + name;

*// LexicalEnvironment = {name: "Петр", hello: "Привет, ", result: "Привет, Петр"}*

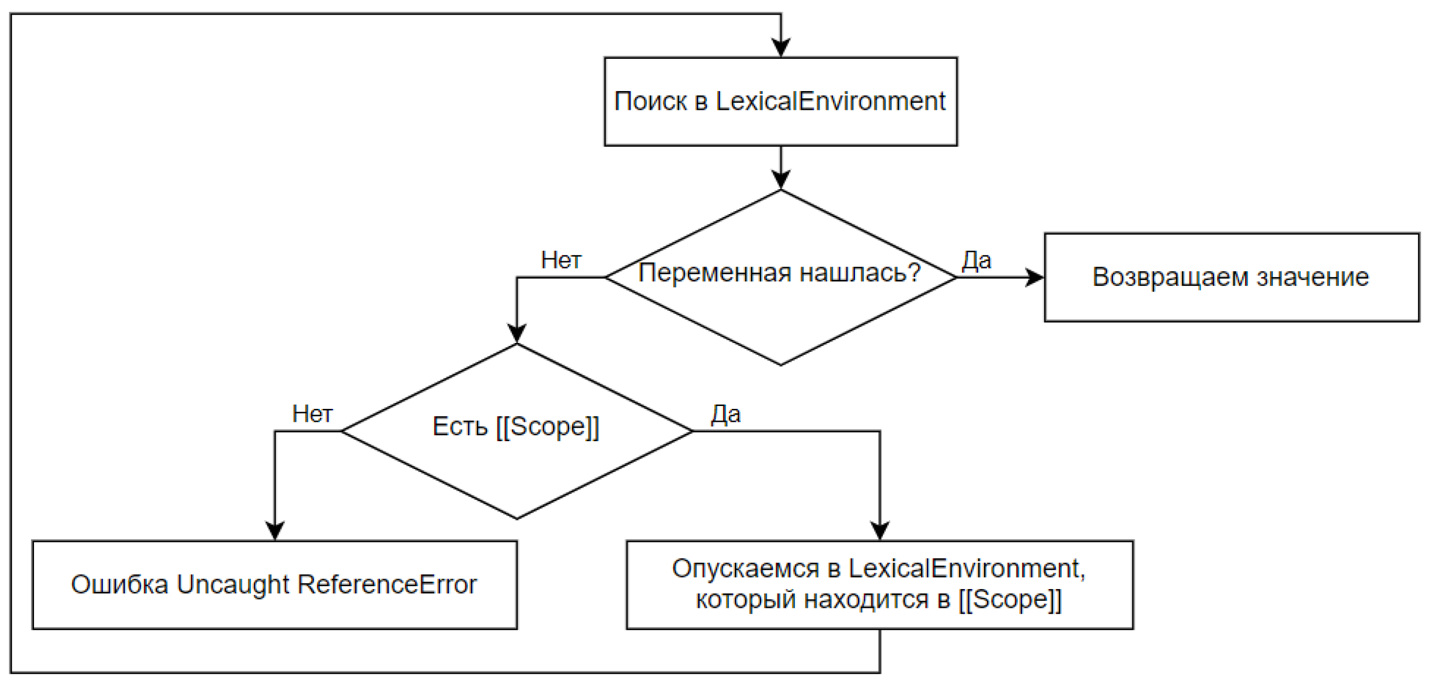
return result;

}

С заполнением разобрались. Теперь надо понять, как работает чтение. Тут тоже не все так просто.

Если переменная создана внутри функции, мы получаем ее следующим образом: **LexicalEnvironment.variableName**. Но из функций мы можем получать доступ и к внешним переменным.

Это происходит потому, что **LexicalEnvironment** содержит в себе еще одно свойство — **[[Scope]]**. Это свойство ссылается на **LexicalEnvironment**, в котором была создана функция. Именно на то, где была создана, а не вызвана. Алгоритм поиска переменной следующий:

[](https://cs.sberuniversity.online/image/full/full/resize/5771cc7c-367e-11ec-b3ea-0242ac170003)

Давайте рассмотрим пример:

let prefix = "Привет, ";

let name = "Вася";

*// \*LexicalEnvironment = {prefix: "Привет, "}*

function getMessage() {

*// \*\*LexicalEnvironment = {[[Scope]]: \*LexicalEnvironment}*

let name = "Петр";

*// \*\*LexicalEnvironment = {name: "Петр", [[Scope]]: \*LexicalEnvironment}*

return prefix + name;

}

console.log(getMessage()); *// "Привет, Петр!"*

function start() {

let prefix = "Пока, ";

*// \*\*\*LexicalEnvironment = {prefix: "Пока, "}*

console.log(getMessage());

}

start(); *// "Привет, Петр!"*

Для удобства внешнее лексическое окружение обозначено одной звездочкой, **getMessage** — двумя, а **start** — тремя.

В**start** мы объявили переменную **prefix** с другим текстом, но из-за того, что функция ссылается на лексическое окружение, в котором она была создана, **getMessage** имеет **[[Scope]] = \*LexicalEnvironment**, и, соответственно, функция видит **prefix**из **\*LexicalEnvironment**. О лексическом окружении **start** функция **getMessage** ничего не знает.

Вторым важным пунктом является то, что в **[[Scope]]** хранится не копия лексического окружения, а ссылка на него.

Другими словами, если после объявления функции во внешнем лексическом окружении изменятся значения переменных или добавятся новые, функция об этом узнает и возьмет актуальное значение.

Рассмотрим пример:

let prefix = "Привет, ";

*// \*LexicalEnvironment = {prefix: "Привет, "}*

function getMessage(name) {

*// \*\*LexicalEnvironment = {name: "Петр", [[Scope]]: \*LexicalEnvironment}*

return prefix + name + postfix;

}

let postfix = "!";

*// \*LexicalEnvironment = {prefix: "Привет, ", postfix: "!"}*

console.log(getMessage("Петр")); *// "Привет, Петр!"*

prefix = "Пока, ";

*// \*LexicalEnvironment = {prefix: "Пока, ", postfix: "!"}*

console.log(getMessage("Петр")); *// "Пока, Петр!"*

В примере переменная **postfix** была объявлена позже самой функции, но из-за того, что в **[[Scope]]** хранится ссылка, а не копия лексического окружения, и на момент вызова, там уже есть переменная **postfix**, функция отрабатывает корректно. В примере также был изменен **prefix**, и функция при втором вызове взяла актуальное значение.

НазадК следующему шагу

# Лексическое окружение вне функций

На самом деле лексическое окружение существует не только в функциях, но и в условиях, циклах — и, вообще, в любых блоках **{...}**. Но с небольшим отличием: переменная **var** попадает либо в ближайшее лексическое окружение функции, либо в глобальное.

Рассмотрим пример:

*// \*LexicalEnvironment = {globalVariable: undefined}*

if (true) {

*// \*\*LexicalEnvironment = {[[Scope]]: \*LexicalEnvironment}*

let innerVariable = 1;

*// \*\*LexicalEnvironment = {innerVariable: 1, [[Scope]]: \*LexicalEnvironment}*

var globalVariable = 2;

}

*// \*LexicalEnvironment = {globalVariable: 2}*

console.log(globalVariable); *// 2*

console.log(innerVariable); *// Ошибка*

В этом примере видно, что переменные, созданные через **let** (и **const**), ведут себя ровно так же, как обсуждалось выше. А вот переменная, созданная через **var**, попала в глобальное лексическое окружение.

Рассмотрим второй пример:

*// \*LexicalEnvironment = {}*

function start() {

*// \*\*LexicalEnvironment = {globalVariable: undefined}*

if (true) {

*// \*\*\*LexicalEnvironment = {[[Scope]]: \*\*LexicalEnvironment}*

let innerVariable = 1;

*// \*\*\*LexicalEnvironment = {innerVariable: 1, [[Scope]]: \*\*LexicalEnvironment}*

var globalVariable = 2;

}

*// \*\*LexicalEnvironment = {globalVariable: 2}*

}

start();

console.log(globalVariable); *// ошибка*

На этом примере видно, что **globalVariable** дошла до лексического окружения ближайшей функции и осталась там.

НазадК следующему шагу

# Лексическое окружение в for

На прошлых двух примерах мы увидели, как работает лексическое окружение в **if**. Поведение в циклах такое же, но стоит отметить одну особенность, связанную с **for**.

Лексическое окружение получает любой блок **{...}**, и из-за этого может сложиться обманчивое впечатление. Если мы внутри **for(...)** объявим переменные с помощью **let/const**, они тоже попадут во внутреннее лексическое окружение и не будут доступны во внешнем.

Другими словами, часть **for(...)** будто находится внутри блока**{...}**:

for (let i = 0; i < 3; i++) {

console.log(i); *// 0, 1, 2*

}

console.log(i); *// Ошибка!*

### Блоки {...}

В JavaScript мы можем использовать просто блоки **{...}** без каких-либо конструкций! И эти блоки будут обладать своей областью видимости.

{

let hello = "Hello, World!";

console.log(hello); *// "Hello, World!"*

}

console.log(hello); *// Ошибка!*

НазадК следующему шагу

# Замыкания в функциях

Давайте рассмотрим пример функции, которая создает счетчик:

function makeCounter() {

let count = 0;

return function() {

return ++count;

}

}

const counter = makeCounter();

console.log(counter()); *// 1*

console.log(counter()); *// 2*

Довольно простой счетчик. Причем мы можем создать их несколько!

Лексическое окружение создается в момент вызова функции, поэтому каждый **makeCounter** будет создавать новое лексическое окружение с **count = 0** и возвращать функцию, которая увеличивает именно этот **count**:

function makeCounter() {

let count = 0;

return function() {

return ++count;

}

}

const counter1 = makeCounter();

const counter2 = makeCounter();

console.log(counter1()); *// 1*

console.log(counter1()); *// 2*

console.log(counter2()); *// 1*

console.log(counter2()); *// 2*

Это называется замыканием. Замыкание, другими словами, это комбинация вложенных функций и внешнего лексического окружения.

НазадК следующему шагу

# Тестирование по разделу

В шаге имеется обязательный к прохождению элемент

Вам нужно пройти тестирование по изученным ранее разделам **Циклы, Функции** и **Замыкания: основы**.

[Начать тест](https://sberuniversity.online/programs/16083/item/671964/launch?is_iframe_mode=1?registration=872a356b-c6a9-4041-b75d-0eb9404afa1d)

НазадК следующему шагу

ЗАДАЧИ

ЗАДАЧИ

ЗАДАЧИ

ЗАДАЧИ

# Создание массива

### Зачем нужны массивы?

Сейчас мы умеем хранить и изменять строки, числа, другие примитивы и объекты. Зачастую на сайтах встречается много однотипных объектов — например, продукты в магазине, логины зарегистрированных пользователей. Для хранения таких сущностей можно использовать объекты:

let products = {

'product-1': {

name: 'Зарядка для телефона',

price: 1000

},

'product-2': {

name: 'Наушники Apple Pro',

price: 25000

},

'product-3': {

name: 'Чехол для телефона',

price: 1000

}

};

Однако, если нам захочется изменить порядок продуктов в объекте — например, отсортировать по цене, начнутся сложности. В JavaScript, как и во многих языках программирования, для хранения списка любых элементов используют**массивы**.

### Создание массива

Массив можно создать с помощью квадратных скобок **[**…**]**. Давайте создадим массив из примитивных типов данных в JavaScript:

const primitivesInJs = ['number', 'string', 'boolean', 'undefined', 'null', 'bigint', 'symbol'];

В примере выше массив состоит только из строк. Однако в JavaScript в массиве можно хранить разные типы данных, например:

const multipleTypes = [0, '1', true, {}, undefined];

Пустой массив можно создать так:

const emptyArray = [];

Есть еще один способ создания массива в JavaScript — с помощью глобального объекта **Array**. Создадим пустой массив с его помощью:

const emptyArray = new Array();

У него есть свои особенности, и подробнее о нем мы поговорим в будущем. На практике чаще всего используют создание массивов через квадратные скобки.

Давайте также переделаем самый первый пример из объекта в массив. Вот что получится:

let products = [

{

id: 'product-1',

name: 'Зарядка для телефона',

price: 1000

},

{

id: 'product-2',

name: 'Наушники Apple Pro',

price: 25000

},

{

id: 'product-3',

name: 'Чехол для телефона',

price: 1000

}

];

Сравнивая с первым примером, можно заметить, что в объекте продукта появилось новое свойство **id**. Раньше оно было ключом объекта **products**, а теперь его пришлось внести в объекты в массиве.

Массив, как и любой непримитивный тип в JavaScript, является объектом. Следовательно, вызов**typeof** для него вернет тип **object**:

let array = [1, 2, 3];

console.log(typeof array); *// object*

НазадК следующему шагу

# Как получить и изменить элементы в массиве

Посмотрим еще раз пример с **products**. Если бы нам надо было получить **product-1** из объекта, мы бы написали так:

const product1 = products['product-1'];

*// {name: "Зарядка для телефона", price: 1000}*

В случае с массивом удобнее всего получать объекты по их порядку. То есть **product-1** у нас первый в массиве, значит, нужно получить первый элемент массива. Это делается так:

const product1 = products[0];

*// {id: 'product-1', name: "Зарядка для телефона", price: 1000}*

То есть в случае объекта элемент удобнее получать по его имени, а в случае массива — по его порядку. Именно поэтому объекты называют именованными коллекциями, а массивы — упорядоченными.

Заметили, что мы написали**[0]**, хотя хотели получить первый элемент массива?

Это потому, что элементы в массиве нумеруются с **0: 0, 1, 2, 3** и т. д. То есть, если бы мы пытались получить пятый элемент, написали бы **products[4]**. Эта особенность нумерации элементов в массиве есть во многих языках программирования, в Интернете шутят, что программисты считают не с **1**, а с **0**.

Порядковый номер элемента в массиве также называют индексом элемента. То есть у первого элемента индекс**0**, у второго — **1** и т. д. Получается, что в JavaScript массивы индексируются, начиная с **0**.

Если нужно изменить первый элемент в массиве, используют также обращение по индексу через квадратные скобки:

let array2 = [1, 2, 3];

array2[0] = 'Hello!';

console.log(array2); *// ['Hello!', 2, 3]*

Часто в коде индекс элемента массива и его новое значение — это какие-то переменные, поэтому в общем виде изменение элемента в массиве выглядит так:

arr[index] = value;

Индекс в массиве, следовательно, может принимать значение от **0** до **длины массива - 1**. То есть, если у нас список из восьми типов языка JavaScript, индекс может принимать значения от **0** до **7**.

А что будет, если попытаться получить элемент с индексом меньше **0** или больше **длины массива - 1**?

Попробуйте выяснить это самостоятельно, выполнив код в консоли разработчика:

let array = [1, 2, 3];

console.log(array[-1]);

### Что выведется в консоль после выполнения блока кода выше?

Ошибка

3

undefined

1

Подтвердить выбор

Это интересная особенность JavaScript.

Такой же результат получится и для кода ниже:

let array = [1, 2, 3];

console.log(array[9999]);

И даже для такого кода:

let array = [1, 2, 3];

console.log(array["Hello"]);

Почему так? Все просто: мы говорили ранее, что массивы — это объекты. Значит, у них сохраняется часть поведения объектов. Попробуйте выполнить код ниже:

let array = [1, 2, 3];

console.log(array[-1]);

console.log(array[9999]);

console.log(array['Hello']);

let object = { 1: '1' };

console.log(object[-1]);

console.log(object[9999]);

console.log(object['Hello']);

Вывод в консоль будет одинаковый. И хотя массивы позволяют так делать, на практике лучше не пользоваться этой особенностью, ведь она замедляет обработку массивов. Кроме того, такой код будет неочевиден для других разработчиков, из-за чего он может привести к ошибкам в программе (их еще называют багами).

От объектов массивы получили и особенность с копированием по ссылке. Например, в блоке кода ниже массивы будут вести себя так же, как и объекты:

let array1 = [1, 2, 3];

let array2 = array1;

array2[0] = 2;

console.log(array1); *// ???*

### Что выведется в консоль после выполнения блока кода выше?

[1, 2, 3]

Ошибка

[2, 2, 3]

undefined

Подтвердить выбор

При передаче массива в качестве аргумента функции, массив передается по ссылке:

let array = [4, 5, 6];

function f1(arr) {

arr[0] = 'Oops!';

}

f1(array);

console.log(array); *// ???*

### Что выведется в консоль после выполнения блока кода выше?

['Oops', 5, 6]

Ошибка

[4, 5, 6]

undefined

Подтвердить выбор

НазадК следующему шагу

# Длина массива

Получить длину массива можно с помощью специального свойства **length**:

let array = [1, 2, 3];

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

Если помните, ровно так же мы узнавали длину строки.

Длина массива полезна для перебора всех значений массива, когда мы заранее не знаем, какой длины он будет. Про перебор значений мы еще поговорим в будущем.

Также с помощью этого свойства можно получить последний элемент в массиве:

let array = [4, 5, 6];

console.log(array[array.length - 1]); *// 6*

### Как добавить и удалить элементы в массиве?

Часто массивы нужно создавать с нуля, добавляя по одному элементу в массив. Для добавления элемента у массивов в JavaScript есть два метода: **push** и **unshift**. Метод **push** добавляет элемент в конец массива, а **unshift** — в начало.

Например:

const result = [];

result.push('World!');

result.unshift('Hello');

console.log(result); *// ['Hello', 'World!']*

Если хочется добавить сразу несколько элементов, можно перечислить их через запятую:

const result = [];

result.push('Hello', 'World!');

console.log(result); *// ['Hello', 'World!']*

Аналогично работает **unshift**.

На практике чаще используется метод **push**, так как логичнее заполнять массив, добавляя по элементу в конец.

Кроме того, **push** работает быстрее, потому что операция добавления элемента в конец быстрее, чем операция добавления в начало. Это потому, что при добавлении в начало массива надо сместить все следующие элементы, чтобы освободить место.

Для удаления используют методы**pop** и **shift.** Когда удаляешь какой-то элемент, обычно его в дальнейшем нужно как-то использовать, поэтому эти методы возвращают удаленный элемент.

**pop** удаляет последний элемент массива и возвращает его:

let array = ["Hello", "World!"];

console.log(array.pop()) *// "World!"*

console.log(array); *// ["Hello"]*

**shift** удаляет первый элемент массива и возвращает его:

let array = ["Hello", "World!"];

console.log(array.shift()) *// "Hello"*

console.log(array); *// ["World!"]*

Как и в случае с **push**и**unshift** удаление элемента из конца массива — операция более быстрая, чем удаление из начала массива.

### Практическое задание 9.1

### Выберите код, который получился в результате проверки выполненного задания

??QV1

9OPS1

K4OL1

9H8F1

3M$D1

Подтвердить выбор

НазадК следующему шагу

# Как пройтись по массиву

Мы ранее упоминали о переборе элементов в массиве, теперь поговорим об этом подробнее.

Чаще всего массивы не имеют какой-то определенной, заранее заданной длины — она все время изменяется. Массивы, в которых заранее неизвестна их длина, называют динамическими. Для таких массивов вывести все элементы через доступ по индексу уже становится тяжело. Например, если у нас массив из двух значений, тут вывести его значения легко:

let array = ["Hello", "World!"];

console.log(array[0]);

console.log(array[1]);

А если мы заранее не знаем, сколько будет элементов, начинаются проблемы:

function printArray(array) {

console.log(array[0]);

console.log(array[1]);

*// ... А когда остановиться?*

}

Еще можно отметить, что вывод значений массива — это одна и та же строчка **console.log,** которая будет повторяться столько раз, сколько элементов в массиве. А ранее мы говорили, что для подобных операций очень хорошо подходят циклы. Так вот, циклы и массивы — лучшие друзья, одно без другого представить сложно.

Вот как можно переписать функцию**printArray** с использованием цикла:

function printArray(array) {

for (let index = 0; index < array.length; index++) {

console.log(array[index]);

}

}

printArray(['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday'])

Тут очень пригодился цикл**for**. Разберем, как он работает. Мы начинаем с индекса, равного нулю. Проверяем значение индекса: оно должно быть меньше длины массива. Далее выполняем код цикла, а именно: операцию с**console.log**. А после увеличиваем**index** на **1**. И повторяем это снова и снова, пока не выведем все элементы.

В блоке о циклах мы рассказывали, что один проход в цикле называется итерацией. А вот когда мы перебираем с помощью цикла значения в массиве — мы итерируемся по массиву.

В примере нам не нужен индекс элемента, а только значение элемента в массиве. Для такого в JavaScript сделали специальный цикл **for-of**. Давайте проитерируемся по массиву с помощью нового цикла:

function printArray(array) {

for (let value of array) {

console.log(value);

}

}

printArray(['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday'])

Видно, что он короче и понятнее предыдущего.

По массиву можно пройтись и в обратном порядке (реверсивном — от англ. reverse). Здесь уже не получится использовать **for-of**, нужен вариант цикла с индексом. Код тогда будет таким:

function printReverseArray(array) {

for (let index = array.length - 1; index >= 0; index--) {

console.log(array[index]);

}

}

printReverseArray(['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday'])

Отсчет индекса начинаем с **длины массива - 1**, ведь это будет последний элемент массива. Затем после каждой итерации уменьшаем индекс на **1**, пока он не станет равным **0**.

Давайте попробуем с помощью изученных знаний написать небольшую функцию-хелпер (от англ. helper — помощник). Итак, необходимо написать функцию **compactArray**.

Она принимает на вход массив, на выходе также выдает массив, но с отфильтрованными **falsy-значениями**.

Falsy-значения — все, что выдает false при вызове Boolean(value). То есть это undefined, null, 0, пустая строка. Такие значения обычно не несут никакой смысловой нагрузки, поэтому функция, которая их фильтрует, часто бывает полезна в разных проектах.

Шаблон и примеры вызова будут такие:

function compactArray(arr) {

*// Здесь будет код*

}

compactArray(['Вася', 'Петя', undefined, 'Иван']) *// -> ['Вася', 'Петя', 'Иван']*

Сперва нам нужно создать новый массив, куда будем записывать нужные элементы массива. Его же мы в конце программы и вернем из функции:

function compactArray(arr) {

const result = [];

*// Здесь будет код*

return result;

}

Теперь пройдемся по массиву **arr** и добавим в новый массив только те элементы, для которых **Boolean(value)** равен **true**:

function compactArray(arr) {

const result = [];

for (let item of arr) {

if (Boolean(item)) {

result.push(item);

}

}

return result;

}

В нашем случае индекс элемента не нужен, поэтому мы воспользовались циклом **for-of**. И методом массива **push** для добавления новых элементов, так как он быстрее и понятнее, чем **unshift**.

Может напрашиваться еще другое решение: создать копию массива и удалить из нее **falsy-значения**. Это решение более сложное, так как для создания копии массива нужно написать один цикл. Далее, чтобы удалить элементы, нужно написать еще один цикл, который пройдется по всем элементам и, если значение **falsy**, удалит его. Получается какая-то двойная работа, которую еще и выполнять сложнее, ведь нужно писать больше кода.

При работе с массивами лучше всегда думать в сторону создания нового массива, а не попытки исправления старого, удаляя или изменяя в старом массиве какие-то элементы.

### Практическое задание 9.2

### Выберите код, который получился в результате проверки выполненного задания

SK?R1

L4<L1

R3MV1

9L9F1

GD1Y1

Подтвердить выбор

### Практическое задание 9.3

### Выберите код, который получился в результате проверки выполненного задания

T6:I1

8W9C1

Z%NI1

7AZ?1

0KCD1

Подтвердить выбор

### Практическое задание 9.4

### Выберите код, который получился в результате проверки выполненного задания

EZP11

4SVU1

NCUB1

$%0S1

@IOD1

Подтвердить выбор

### Практическое задание 9.5

### Выберите код, который получился в результате проверки выполненного задания

BEPN1

D%0S1

D@%31

1K@P1

QL2B1

Подтвердить выбор

### Практическое задание 9.6

### Выберите код, который получился в результате проверки выполненного задания

D0XW1

?0W#1

N90A1

N1VL1

T@W01

Подтвердить выбор

### Практическое задание 9.7

### Выберите код, который получился в результате проверки выполненного задания

VIR21

BEHR1

2YLV1

Q5VA1

XI?V1

Подтвердить выбор

### Практическое задание 9.8

### Выберите код, который получился в результате проверки выполненного задания

#Q#G1

E2VT1

@1KH1

W#331

@BY12

Подтвердить выбор

### Практическое задание 9.9

### Выберите код, который получился в результате проверки выполненного задания

Z6DL1

XR0M1

14EW1

FS@H1

4#JP1

Подтвердить выбор

НазадК следующему шагу

# Преобразование массива в строку и в Boolean. Задачи на Массивы

Массивы можно превращать в строку, чтобы можно было посмотреть их содержимое.

В JavaScript для этого существует метод **toString()**, о котором мы рассказывали ранее. Он выведет элементы массива через запятую, приводя каждый элемент к строке:

let stringArray = ['A', 'B', 'C'];

console.log(stringArray.toString()); *// 'A,B,C'*

let numberArray = [1, 2, 3];

console.log(numberArray.toString()); *// '1,2,3'*

Методу **toString()** аналогична функция **String**, об этом мы также рассказывали:

let stringArray = ['A', 'B', 'C'];

console.log(String(stringArray)); *// 'A,B,C'*

Давайте повторим особенность этого метода.

let objectArray = [{ name: 'Иван' }, { name: 'Петя' }, { name: 'Саша' }];

console.log(objectArray.toString());

### Что выведется в консоль после выполнения блока кода выше?

Object,Object,Object

{ name: 'Иван' },{ name: 'Петя' },{ name: 'Саша' }

[object Object],[object Object],[object Object]

Иван,Петя,Саша

Подтвердить выбор

Примеры показали, что элементы массива хотя и выводятся через запятую, но после запятой, например, нет пробела. Это типографическая ошибка, которую хотелось бы исправить. В JavaScript у массивов есть специальный метод **join(separator)**, который позволяет превратить массив в строку, соединив элементы через **separator**.

Если в**join** не указывать**separator**, он делает то же самое, что и метод **toString** или функция **String**:

console.log([1, 2, 3].join()) *// '1,2,3'*

Если же мы хотим выводить элементы через запятую с пробелом, можно написать так:

console.log([1, 2, 3].join(', ')) *// '1, 2, 3'*

Есть и обратный метод **split(separator)**, когда из строки мы можем получить массив, разделив строку по какому-то разделителю:

console.log('1,2,3'.split(',')) *// ['1', '2', '3']*

Если вызвать **split** без указания разделителя, он всегда будет возвращать массив из одного элемента — исходной строки:

console.log('1,2,3'.split()) *// ['1,2,3']*

Отметим, что **split** на выходе всегда будет давать массив из строк. То есть из строки можно получить только массив из строк, а вот обратно не так: из массива любых значений можно получить строку.

Отметим также, что преобразование массива в **Boolean** всегда дает **true**, неважно, есть элементы в массиве или нет:

console.log(Boolean([])) *// true*

console.log(Boolean([1, 2])) *// true*

Поэтому для проверки не пустой ли массив лучше пользоваться свойством **length**:

console.log(Boolean([].length)) *// false*

console.log(Boolean([1, 2].length)) *// true*

### Практическое задание 9.10

### Выберите код, который получился в результате проверки выполненного задания

01151

IHLS1

ESP71

@X2Y1

DGY61

Подтвердить выбор

### Практическое задание 9.11

### Выберите код, который получился в результате проверки выполненного задания

0P@#1

UD0X

F3YA1

XH?L1

QG6L1

Подтвердить выбор

### Практическое задание 9.12

### Выберите код, который получился в результате проверки выполненного задания

VW352

L70K2

H%RV2

LBE<2

QGAO2

Подтвердить выбор

### Практическое задание 9.13

### Выберите код, который получился в результате проверки выполненного задания

HVTP1

8AGB1

FM6P1

0V6G1

@4OS1

Подтвердить выбор

### Практическое задание 9.14

### Выберите код, который получился в результате проверки выполненного задания

L?DE2

HAH%2

NBQ%2

RD482

WI682

Подтвердить выбор

### Практическое задание 9.15

### Выберите код, который получился в результате проверки выполненного задания

GEE02

U6N;2

GVJQ2

0ZWL2

YWD$2

Подтвердить выбор

### Практическое задание 9.16

### Выберите код, который получился в результате проверки выполненного задания

T02Q1

?#U?1

C63D1

NF#B1

D6OP1

Подтвердить выбор

НазадК следующему шагу

# Многомерные массивы

Массивы могут хранить в себе любые элементы, в том числе и другие массивы:

let array = [

[1, 2, 3],

[4, 5, 6]

];

Такого рода массивы называют **многомерными**, а в примере выше — **двумерный**.

Чтобы получить из него элементы, мы два раза используем квадратные скобки. Сперва мы получаем элемент из первого массива:

let array = [

[1, 2, 3],

[4, 5, 6]

];

console.log(array[0]); *// [1, 2, 3]*

А затем вытаскиваем необходимый элемент из второго массива:

let array = [

[1, 2, 3],

[4, 5, 6]

];

console.log(array[0][0]); *// 1*

console.log(array[0][1]); *// 2*

console.log(array[0][2]); *// 3*

console.log(array[1][0]); *// 4*

Многомерные массивы используются редко, чаще всего они нужны для представления математических структур, таких как матрицы.

Чтобы пройтись по всем элементам многомерного массива, нужно два цикла. Первый будет идти по первому массиву, а второй — по второму:

let array = [

[1, 2, 3],

[4, 5, 6]

];

for (let itemFromArr1 of array) {

for (let itemFromArr2 of itemFromArr1) {

console.log(itemFromArr2);

}

}

### Практическое задание 9.17

### Выберите код, который получился в результате проверки выполненного задания

O6QH1

M3OS1

E54B1

9SGA1

W1:Q1

Подтвердить выбор

### Практическое задание 9.18

### Выберите код, который получился в результате проверки выполненного задания

I$VI2

L4L22

QD712

E%SH2

$CD62

Подтвердить выбор

НазадК следующему шагу

# ПРИМЕРЫ РЕШЕНИЯ ЗАДАНИЙ

Для получения доступа вам необходимо пройти материал

[Многомерные массивы](https://sberuniversity.online/programs/16083/item/671856?is_iframe_mode=1)

НазадК следующему шагу

ЗАДАЧИ

ЗАДАЧИ

ЗАДАЧИ

ЗАДАЧИ

ПОВТОРЕНИЕ ХТМЛ

Создание блока с преимуществами продукта

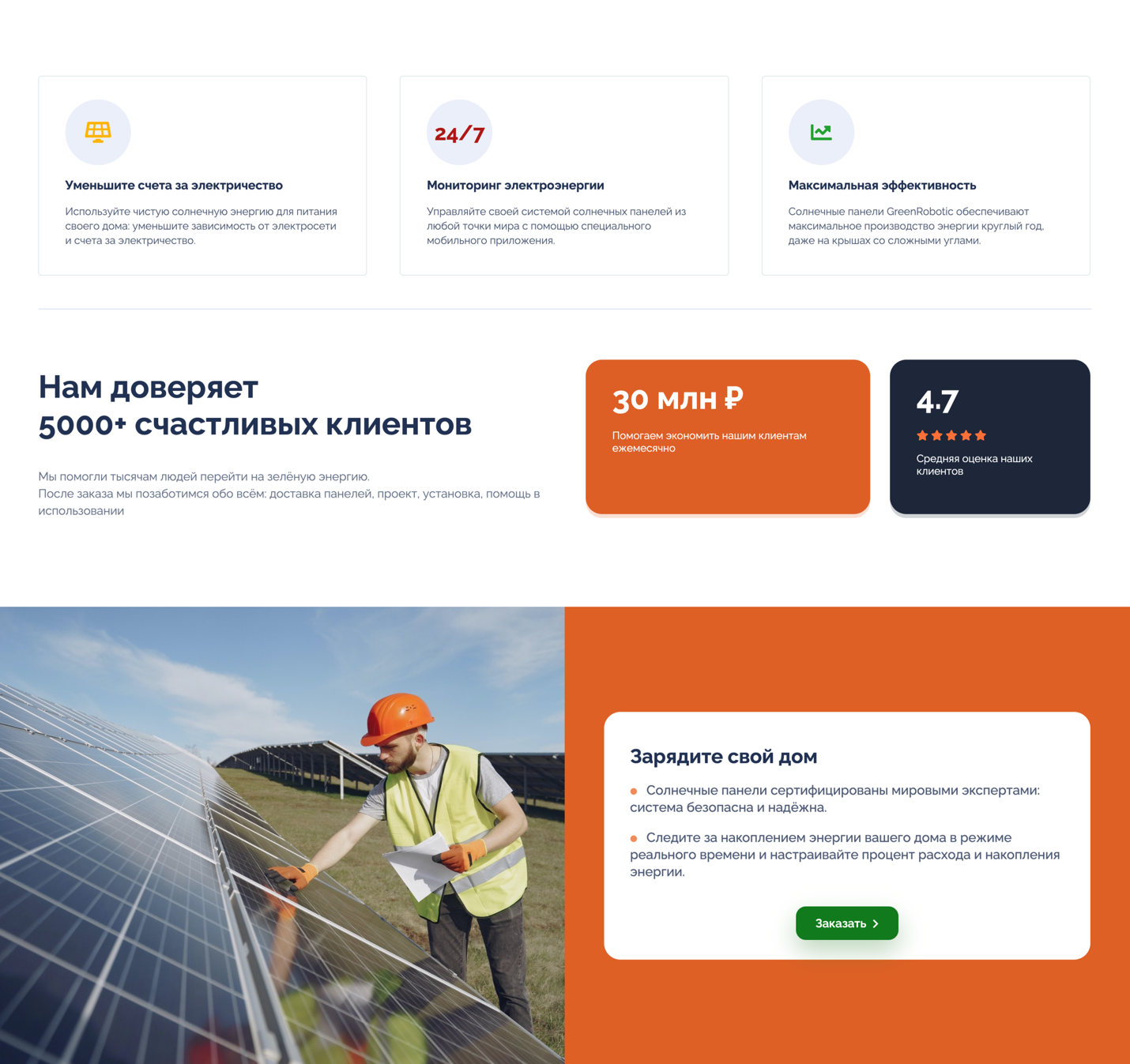
В лендинге о солнечных панелях появились новые блоки, более подробно раскрывающие преимущества продукта. Макет всего лендинга теперь выглядит так:

[Макет лендингаpng, 5,8 МБ](https://sberuniversity.online/programs/16083/item/671862/launch?is_iframe_mode=1" \t "_blank)

Также изменился макет для мобильных устройств.

[Макет для мобильных устройствpng, 2,4 МБ](https://sberuniversity.online/programs/16083/item/671864/launch?is_iframe_mode=1" \t "_blank)

Новые блоки из макета, соответственно, выглядят так:

[](https://cs.sberuniversity.online/image/full/full/resize/0436e36e-327b-11ec-8b16-0242ac170003)

Их можно поделить на два отдельных блока, добавим в проект сперва верхний блок. Как и для всех прочих блоков в лендинге, создадим ему тег **section**, класс у него будет **features**, так как в блоке описываются преимущества продукта. HTML-код блока:

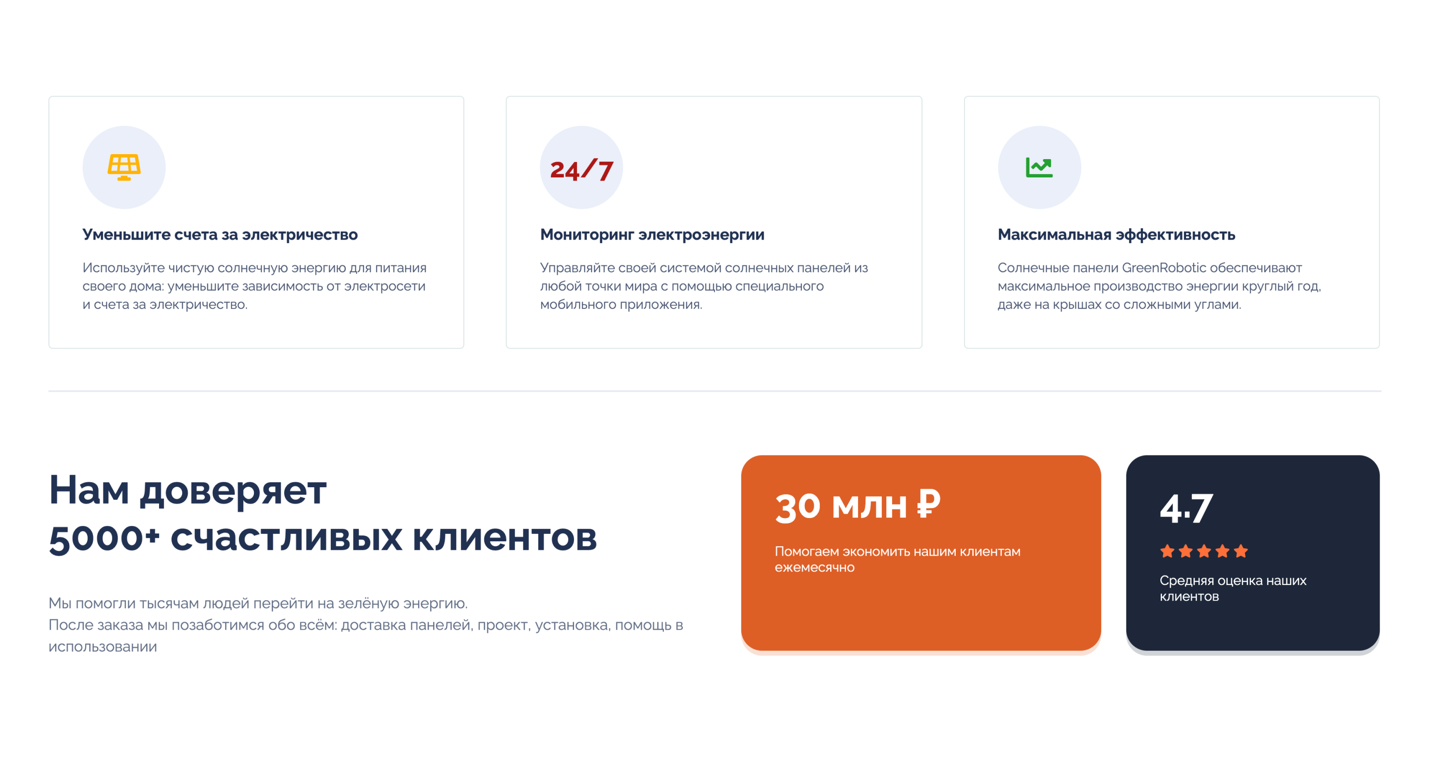
<section class="features">

</section>

Сразу добавим ему стилей. Нужно задать ему фон и внутренние отступы.

* Проектное задание 10.1
  1. Для тега с классом **features** задайте белый фон и внутренние отступы: **115px** сверху и снизу, **60px** справа и слева.
  2. В мобильной версии нужно изменить отступы: **50px** сверху и снизу, **10px** справа и слева.

Этот блок состоит из двух частей, разделенных серой полосой:

[](https://cs.sberuniversity.online/image/full/full/resize/ec9dda58-327c-11ec-afea-0242ac160003)

Начнем сверху. Верхняя часть состоит из трех карточек, в каждой есть иконка, заголовок и абзац текста. Карточки равномерно распределены в одну линию. Давайте добавим их в проект. Сперва нужно создать для них контейнер, чтобы можно было расположить их в ряд. Контейнером будет тег **div** с классом **cards**, добавим его сразу в тег **section**:

<section class="features" id="features">

<div class="cards">

</div>

</section>

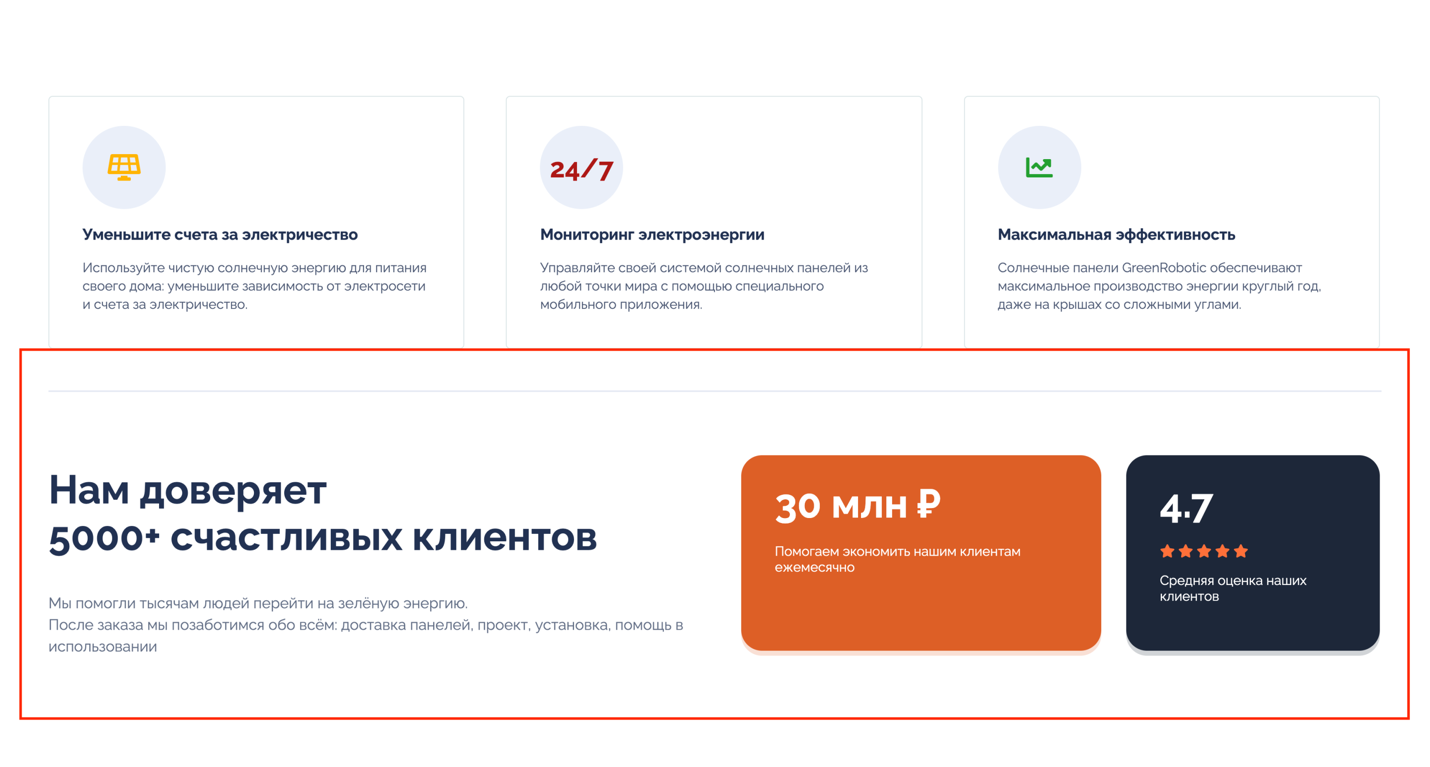
Теперь необходимо добавить сами карточки.

* Проектное задание 10.2
  1. Каждая карточка должна быть отдельным тегом **div** с классом **card**. Внутри должен быть тег для логотипа, **div** с классом **card\_\_logo**, тег **h3** для заголовка и тег **p** для параграфа текста. Иконки для карточек взяты из **fontawesome**.
  2. Иконка первой карточки должна быть **fa-solar-panel**. Заголовок: **Уменьшите счета за электричество**.  
     Абзац текста: **Используйте чистую солнечную энергию для питания своего дома: уменьшите зависимость от электросети и счета за электричество**.
  3. Иконка у второй карточки это обычный текст 24/7. Заголовок: **Мониторинг электроэнергии**. Абзац текста: **Управляйте своей системой солнечных панелей из любой точки мира с помощью специального мобильного приложения**.
  4. Иконка третьей карточки должна быть **fa-chart-line**.  
     Заголовок: **Максимальная эффективность**.  
     Абзац текста: **Солнечные панели GreenRobotic обеспечивают максимальное производство энергии круглый год, даже на крышах со сложными углами.**

Далее необходимо добавить им стили. Чтобы расположить карточки в три колонки, удобно использовать **display: grid**.

* Проектное задание 10.3
  1. Стилизуйте карточки согласно [макету](https://sberuniversity.online/programs/16083/item/671867/launch), [макет мобильной версии карточек](https://sberuniversity.online/programs/16083/item/671868/launch).
  2. Карточки должны быть выстроены тремя колонками, расстояние между карточками **50px**. В мобильной версии они должны идти друг под другом (для этого можно сбросить свойство задания колонок в изначальное, например **grid-template-columns: auto**).
  3. У карточки должны быть внутренние отступы: **35px** сверху, **40px** справа, **25px** снизу, **40px** слева. У карточки должна быть граница толщиной **1px** цвета **#D9E4E6**. У границы должны быть закругления радиусом **5px**.
  4. У текста абзаца карточки должен быть размер **16px**, размер строчки **22px**, а также текст должен быть непрозрачным на **0.8** (**opacity: 0.8**).
  5. Контейнер логотипа карточки должен быть размером **100x100px**, текст внутри должен быть размера **32px**, толщина текста **700**, цвет фона **#eaeff9**, он должен быть круглым, это можно сделать с помощью свойства **border-radius: 50%**. Внутри контейнера логотипа лежат иконки из**fontawesome** и текст, их нужно отцентровать по вертикали и горизонтали, это можно сделать с помощью **flexbox**.
  6. У каждой карточки свой цвет логотипа, это можно настроить, изменяя цвет текста. Цвет текста первой карточки **#ffb406**, второй **#ad1714**, третьей **#22a030**.

Отлично, верхние карточки закончены. Теперь можно приступать ко второй части блока. В ней есть серая линия, которая отделяет части блока, и подблок из заголовка, описания слева и двух карточек справа:

[](https://cs.sberuniversity.online/image/full/full/resize/26ac468c-3294-11ec-8f63-0242ac170003)

Для отделяющей серой линии сделаем отдельный тег.

* Проектное задание 10.4
  1. Добавьте тег **div** с классом **features\_\_border.**
  2. У этого блока должны быть внешние отступы сверху и снизу в **50px**. Чтобы сделать его линией, можно добавить ему границу толщиной **1px**, цвет должен быть **#E6EAF4**.

Теперь добавим блок с текстом и двумя карточками. Как и для прошлых карточек, чтобы правильно все расположить, нужен отдельный контейнер. Сделаем его тегом **div** с классом **features\_\_feedback**.

В контейнере есть левая и правая часть, поэтому внутрь поместим два тега **div** с классами **features\_\_feedback-left**и **features\_\_feedback-right** соответственно:

<div class="features\_\_feedback">

<div class="features\_\_feedback-left">

</div>

<div class="features\_\_feedback-right">

</div>

</div>

Далее в левую и правую его части надо добавить необходимые элементы.

* Проектное задание 10.5
  1. В левой части должен быть заголовок **h2**, текст заголовка: **Нам доверяет 5000+ счастливых клиентов**. Часть с текстом **5000+ счастливых клиентов** должна быть на следующей строчке.
  2. Под заголовком должен быть абзац текста (тег **p**).  
     Текст абзаца, первая строчка: **Мы помогли тысячам людей перейти на зеленую энергию**.  
     Вторая строчка: **После заказа мы позаботимся обо всем: доставка панелей, проект, установка, помощь в использовании**.
  3. В правой части (тег с классом **features\_\_feedback-right**) должно быть внутри два тега для каждой из карточек. Карточка является тегом **div** с классом **features\_\_box**.
  4. У первой карточки также должен быть второй класс **features\_\_savings-box**.  
     Внутри нее есть заголовок **h3** с текстом: **30 млн ₽**.  
     Далее идет абзац (тег **p**) с текстом: **Помогаем экономить нашим клиентам ежемесячно**.
  5. У второй карточки также должен быть второй класс **features\_\_rate-box**. Внутри нее есть заголовок **h3** с текстом: **4.7**.  
     Далее идет абзац (тег **p**) с классом **stars**, содержащий внутри себя пять иконок **fa-star**.  
     После абзаца с иконками должен быть еще один абзац с текстом **Средняя оценка наших клиентов**.

Теперь необходимо стилизовать полученные тексты и карточки.

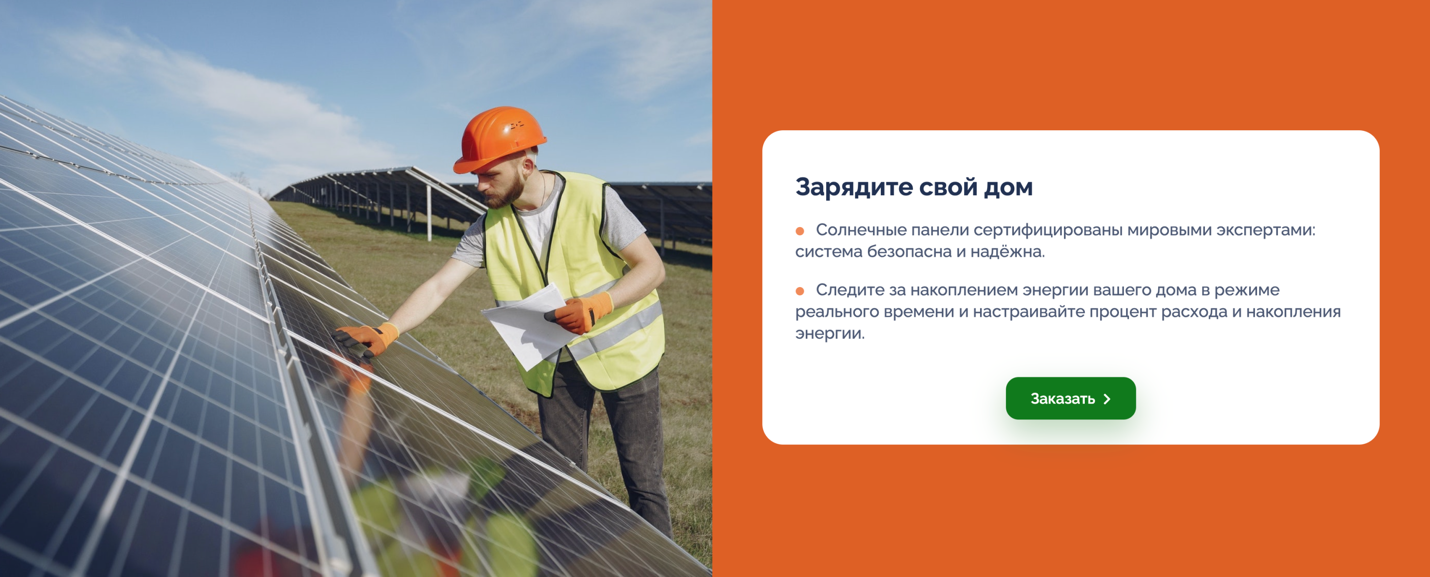
* Проектное задание 10.6
  1. Нужно стилизовать блок согласно макету. [Макет мобильной версии](https://sberuniversity.online/programs/16083/item/671869/launch).
  2. Нужно расположить левую и правую часть на одной линии. Для этого можно использовать**flexbox** для **features\_\_feedback**. В мобильной версии блоки должны идти друг под другом, для этого нужно изменить направление главной оси flex-контейнера.
  3. У левой части (**features\_\_feedback-left**) должен быть отступ справа **10px**. В мобильной версии этого отступа не должно быть.
  4. Размер текста заголовка (**h2**) левой части **48px**. В мобильной версии он должен быть равен **36px**.
  5. Внутри абзаца в левой части (тег **p**) у текста должен быть размер **18px**, высота строчки **26px**, а также полупрозрачность **70% (opacity: 0.7)**.
  6. Далее необходимо стилизовать правую часть (**features\_\_feedback-right**). В правой части находятся две цветные карточки, они расположены в одну строчку и отцентрованы по вертикали. Это можно сделать с помощью **flexbox**. В мобильной версии они должны идти друг под другом и не должно быть центрирования по вертикали.
  7. У каждой карточки (**features\_\_box**) должны быть внутренние отступы: **30px** сверху, **40px** справа, **25px** снизу, **40px** слева. Цвет текста внутри них должен быть белый, а также должны быть закругления радиусом **25px**.
  8. Заголовок**h3** внутри карточки должен быть без внешнего отступа сверху, внешний отступ снизу **20px**, размер текста **48px**. В мобильной версии размер текста заголовка должен быть **36px**.
  9. У левой карточки (**features\_\_savings-box**) цвет фона **#dc5e26**, цвет тени **rgb(221 95 38 / 20%)**. Тень должна на **6px** отступать вниз, на **0px** горизонтально и иметь **0** размытие. Также у левой карточки должен быть внешний отступ справа **30px.** В мобильной версии внешнего отступа справа быть не должно, но должен быть внешний отступ снизу **30px.**
  10. У правой карточки (**features\_\_rate-box**) цвет фона **#1D263A**, цвет тени **rgb(29 38 58 / 20%)**. Тень должна на **6px** отступать вниз, на **0px** горизонтально и иметь **0** размытие.
  11. У правой карточки цвет звезд равен**#ff7039**.

Отлично, с первым блоком справились. Теперь можно приступать к следующему.

НазадК следующему шагу

Создание блока с изображением и кнопкой заказа продукта

Во втором блоке сравнительно меньше элементов, есть картинка и карточка с текстом справа:

[](https://cs.sberuniversity.online/image/full/full/resize/80531e88-3297-11ec-9e8a-0242ac170004)

На [макете мобильной версии](https://sberuniversity.online/programs/16083/item/671872/launch) они выстраиваются друг под другом, причем изменяется их порядок: сначала карточка, потом изображение. Также меняются отступы.

Так как это отдельный блок, ему также нужен свой тег **section**. Дадим ему класс **promo**. Блок содержит изображение слева и контейнер с карточкой справа. Добавим эти теги:

<section class="promo">

<div class="promo\_\_image"></div>

<div class="promo\_\_right">

</div>

</section>

Добавим им стили и настроим картинку. Для картинки мы создали тег **div** с классом **promo\_\_image**. Сохраните файл картинки в папку проекта**img**, назовите ее **promo**:

[promojpg, 268 КБ](https://sberuniversity.online/programs/16083/item/671873/launch?is_iframe_mode=1" \t "_blank)

Как мы знаем, картинку можно добавить двумя способами: создать тег **img** или изменить **background-image** у тега. В данном случае хочется воспользоваться выставлением фона у тега**div**, так как у фона есть свойство **background-size: cover**, позволяющее картинке подстраиваться под размеры своего контейнера.

* Проектное задание 10.7
  1. Cтилизуем тег для картинки (**promo\_\_image**). Нужно выставить ему фон в виде картинки, картинка должна располагаться по центру и растягиваться по контейнеру. Также тег с картинкой должен быть шириной в половину от карточки, тег может сплющивать по ширине, это можно исправить свойством**flex-shrink: 0**. В мобильной версии картинка должна растягиваться по всей ширине, иметь высоту **500px**.
  2. В теге **promo** нужно расположить картинку и правую карточку на одной линии. Для этого можно использовать **flexbox**. В мобильной версии блоки должны идти друг под другом в обратном порядке.
  3. Цвет фона правой части (**promo\_\_right**) **#dd5f26**. Внутренние отступы сверху и снизу **160px**, справа и слева **60px**. В мобильной версии внутренние отступы сверху и снизу **50px**, справа и слева **10px**.

Далее добавим недостающие элементы из правой части блока.

* Проектное задание 10.8
  1. Внутрь правой части нужно добавить тег для карточки, ему нужно задать класс.
  2. Внутри тега для карточки должен быть заголовок **h2** с текстом **Зарядите свой дом**.
  3. После заголовка должен быть абзац с текстом: **Солнечные панели сертифицированы мировыми экспертами: система безопасна и надежна**.
  4. Далее должен быть еще один абзац с текстом: **Следите за накоплением энергии вашего дома в режиме реального времени и настраивайте процент расхода и накопления энергии**.
  5. Далее после абзацев нужно добавить тег контейнера для кнопки, а внутри него тег кнопки с текстом **Заказать** и иконкой **fa-chevron-right.** Контейнер нужен, чтобы в дальнейшем отцентровать кнопку по горизонтали.

И теперь добавленные элементы из правой части надо стилизовать.

* Проектное задание 10.9
  1. Стилизуем тег карточки. У этого тега должен быть белый фон, внутренние отступы сверху и снизу **30px**, справа и слева **40px**. Размер текста **20px**, высота строчки **22px**, толщина текста **500**. Также должны быть скругления радиусом **25px**.
  2. Текст абзацев в карточке должен быть с полупрозрачностью **80%**.
  3. Перед каждым абзацем должен быть круг радиусом **10px** цвета**#ec6b29** и с внешним отступом справа в **10px.**
  4. Добавим стили для кнопки **Заказать**. Цвет фона кнопки **#107b1c**, у нее должен быть отступ сверху **20px**, размер шрифта **18px** и внутренние отступы сверху и снизу **15px**, справа и слева **30px**. Размер текста кнопки **18px**, толщина **600**. У иконки внутри кнопки размер текста **15px**, а также имеется внешний отступ слева **5px**.

Отлично, мы закончили с новыми блоками! Не забудьте обновить проект на GitHub Pages, закоммитив и отправив изменения в репозиторий.

Таким образом вы обезопасите себя от потери данных при поломке компьютера и обновите лендинг.

НазадК следующему шагу

# Тестирование по разделу

В шаге имеется обязательный к прохождению элемент

Вам нужно пройти тестирование по изученным ранее разделам **Объекты: основы, Массивы: основы и Повторение основ HTML и CSS**.

[Начать тест](https://sberuniversity.online/programs/16083/item/671965/launch?is_iframe_mode=1?registration=0e0e6c91-11d8-4f81-a0bf-7c893422cd34)

НазадК следующему шагу

# ПРИМЕРЫ РЕШЕНИЯ ЗАДАНИЙ

Для получения доступа вам необходимо пройти материал

[Тестирование по разделу](https://sberuniversity.online/programs/16083/item/671875?is_iframe_mode=1)

НазадК следующему шагу

# Анкета обратной связи

В шаге имеется обязательный к прохождению элемент

Просим вас уделить несколько минут заполнению анкеты обратной связи. Ваши ответы помогут нам сделать этот курс лучше.

Анкета обратной связи

[exam.sberbank-school.ru](https://sberuniversity.online/programs/16083/item/672016/launch?is_iframe_mode=1)

[Перейти](https://sberuniversity.online/programs/16083/item/672016/launch?is_iframe_mode=1)

НазадК следующему шагу