**ИЗУЧИТЕ JAVASCRIPT В CONSTRUCT, ЧАСТЬ 1: НАЧАЛО РАБОТЫ**

Это руководство рекомендовано командой Construct! Это означает, что он содержит полезную и качественную информацию, которая поможет вам вырасти как разработчику игр.

[JavaScript](https://www.construct.net/out?u=https%3a%2f%2fen.wikipedia.org%2fwiki%2fJavaScript) — [один из самых популярных языков программирования в мире](https://www.construct.net/out?u=https%3a%2f%2finsights.stackoverflow.com%2fsurvey%2f2021%23technology-most-popular-technologies) , с широким спектром приложений, от браузеров до серверов, [множеством компаний, желающих нанять разработчиков JavaScript](https://www.construct.net/out?u=https%3a%2f%2fdevskiller.com%2fit-skills-report-2020%2f) , и многими другими преимуществами, такими как выдающаяся производительность. Все это делает его отличным выбором для начала обучения программированию. Эта серия руководств поможет вам изучить основы языка программирования JavaScript в [Construct](https://www.construct.net/) !

Этот урок предназначен для новичков. Если вы уже знаете JavaScript, ознакомьтесь с [кратким руководством для разработчиков Construct for JavaScript](https://www.construct.net/en/tutorials/construct-javascript-2866) .

## JAVASCRIPT — ЭТО НЕ JAVA

Одна из распространенных путаниц, которую следует сразу прояснить, заключается в том, что **Java** и **JavaScript** — это два совершенно разных языка программирования! Именование действительно несколько сбивает с толку, но постарайтесь не путать их. Например, если вы ищете в Интернете справку по JavaScript, обязательно ищите именно JavaScript , а не Java , иначе вы получите результаты для неправильного языка программирования!

## ОБ ЭТОМ РУКОВОДСТВЕ

Это руководство поможет вам **изучить язык программирования JavaScript,** используя функцию кодирования JavaScript Construct. Это хороший способ начать, поскольку вы можете сразу начать писать код без необходимости настраивать среду разработки (например, локальный HTTP-сервер). Вся серия руководств предназначена для работы в рамках бесплатной версии, поэтому вы можете изучать JavaScript с помощью этого руководства, не покупая ничего. Функция кодирования JavaScript в Construct также предоставляет множество способов сделать больше в ваших проектах, поэтому она будет полезна и для более продвинутых пользователей Construct.

Construct использует **стандартный язык программирования JavaScript** , поэтому вы изучаете то же самое, что используется повсюду в отрасли. Construct никоим образом не меняет язык — фактически браузер запускает ваш код напрямую.

Блочная система Construct в таблицах событий разработана таким образом, чтобы новичкам было проще начать работу. Если вы хотите попробовать начать работу с блоками или просто сначала немного познакомиться с Construct, рассмотрите возможность попробовать [руководство для начинающих по Construct](https://www.construct.net/en/tutorials/beginners-guide-construct-1) .

**Вам не обязательно использовать Construct раньше, чтобы следовать этому руководству.** Однако некоторое знакомство будет полезно. Если вы также знакомы с переменными, циклами и функциями в таблицах событий, это также будет полезно. Эти функции таблиц событий намеренно созданы так, чтобы быть похожими на работу эквивалентных функций языка программирования. Итак, если вы знаете, как они работают в таблицах событий, это поможет вам понять, как они работают в JavaScript. В этом руководстве будет указано, где таблицы событий имеют концепции, схожие с JavaScript. Однако эти ссылки не будут необходимы для следования руководству, это просто дополнительный контекст для пользователей Construct, которые уже знакомы с таблицами событий.

## ЧТО ВЫ УЗНАЕТЕ

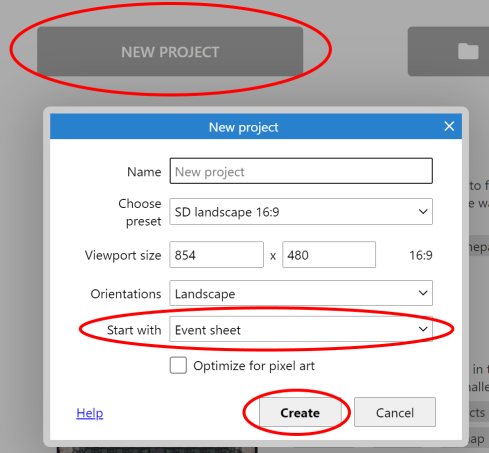
В этом руководстве будут рассмотрены основные возможности языков программирования JavaScript, включая такие темы, как переменные, операторы if, циклы и функции. JavaScript — это развитый язык с огромным набором функций, и данное руководство не пытается охватить его полностью. Цель состоит в том, чтобы научить вас достаточному количеству JavaScript, чтобы вы могли изучить его из других руководств в Интернете и научиться применять его в Construct. (Некоторые дополнительные руководства будут приведены в конце серии.)

JavaScript был изобретен в 1995 году, и, как и большинство старых языков программирования, JavaScript имеет немало особенностей и устаревших функций. Однако в этом руководстве основное внимание будет уделено наиболее полезным частям с использованием современного стиля и лучших практик, и мы не будем вдаваться в подробности об особенностях или старых функциях, которые больше не используются широко. Другими словами, это практическое руководство, ориентированное на написание реального кода.

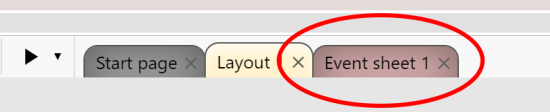
Готовы начать? Перейдите на следующую страницу, чтобы написать свою первую строку кода!

## ДОБАВЛЕНИЕ КОДА JAVASCRIPT В ЛИСТ СОБЫТИЙ

Для простого начала давайте напишем нашу первую строку кода в листе событий. Сначала [откройте редактор Construct](https://editor.construct.net/) и создайте новый пустой проект, убедившись, что для параметра «Начать с» установлено значение «Лист событий» .

[](https://construct-static.com/images/v1202/uploads/articleuploadobject/0/images/66137/create-new-project.png)

Затем щелкните вкладку «Лист событий 1» , чтобы переключиться на просмотр листа событий.

[](https://construct-static.com/images/v1202/uploads/articleuploadobject/0/images/66138/event-sheet-tab.png)

Теперь добавьте событие «В начале макета» :

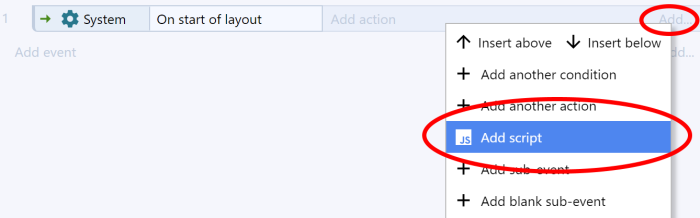
1. Нажмите **«Добавить событие»** на вкладке «Лист событий».
2. Дважды щелкните **Система**
3. Дважды щелкните **«В начале макета»** (в группе «Начало и конец »).

В итоге у вас должно получиться следующее:

[В начале макета](https://construct-static.com/images/v1202/uploads/articleuploadobject/0/images/65533/on-start-of-layout.png)

Это блок событий, который будет выполнять свои действия при запуске.

Мы можем добавить некоторый код JavaScript вместо действия в этом блоке событий, чтобы наш код запускался при запуске. Нажмите «Добавить...» справа от блока и выберите « Добавить скрипт» .

[](https://construct-static.com/images/v1202/uploads/articleuploadobject/0/images/65562/add-script-action.png)

Текстовое поле появляется как действие. Это небольшой редактор сценариев, в который вы можете ввести код JavaScript для запуска.

[Редактирование действия сценария](https://construct-static.com/images/v1202/uploads/articleuploadobject/0/images/65536/edit-script-action.png)

А пока давайте просто скопируем и вставим следующую строку кода в это текстовое поле:

**console.log("Hello world!");**

Обратите внимание, что если вы перейдете в другое окно, редактор сценариев может завершить работу и превратиться в пустой блок кода. В этом случае **дважды щелкните** пустой блок кода, чтобы отредактировать его еще раз, а затем вставьте или скопируйте приведенный выше пример кода. Затем щелкните за пределами редактора сценариев или нажмите Ctrl+ Enter, чтобы закрыть редактор сценариев.

Нажатие Enterдобавляет еще одну строку в редактор сценариев на случай, если вы захотите написать несколько строк кода, поэтому вместо этого для завершения редактирования используется Ctrl+ .Enter

Теперь вы должны увидеть свою первую строку кода JavaScript в таблице событий!

[Ваша первая строка кода JavaScript!](https://construct-static.com/images/v1202/uploads/articleuploadobject/0/images/65537/console-log-code.png)

Эта строка кода зарегистрирует сообщение Hello world! в консоль браузера при запуске. Вскоре мы найдем, как просмотреть это сообщение. Чтобы помочь вам быстро начать работу, мы пока не будем полностью объяснять каждую часть этого кода — это руководство будет подробно рассмотрено позже. А пока вот краткое изложение того, что означает эта строка кода:

* **console.log**означает «получить доступ к **log**функции объекта **console**»
* Круглые скобки **(**и **)**означают «вызов **log**функции»
* **"Hello world!"**это некоторый текст, заключенный в двойные кавычки и передаваемый функции **log**.
* Точка с запятой **;**в конце обозначает конец оператора

Конечный результат означает «добавление сообщения с текстом Hello world! на консоль». Давайте узнаем, как увидеть наше сообщение!

## ИСПОЛЬЗОВАНИЕ ИНСТРУМЕНТОВ РАЗРАБОТЧИКА

Консоль браузера является частью инструментов разработчика браузера , или для краткости «инструментов разработки». Когда вы только начинаете, знакомство с инструментами разработчика может показаться утомительным занятием, но оно того стоит по двум причинам:

1. Инструменты разработчика и такие вещи, как запись сообщений в консоль, являются важной частью повседневной работы по программированию. Определенно стоит узнать о них и ознакомиться с тем, как они работают.
2. Консоль браузера также является очень полезным инструментом, помогающим вам изучить JavaScript. Вы можете ввести фрагменты JavaScript, запустить их и сразу же увидеть результат, что является отличным способом поэкспериментировать и узнать больше. Мы попробуем это позже.

### ОТКРЫТИЕ ИНСТРУМЕНТОВ РАЗРАБОТЧИКА

Вы можете открыть инструменты разработки для любого окна или вкладки браузера. Однако нам нужны инструменты разработки только для окна предварительного просмотра, в котором запускается ваш проект, поскольку именно там запускается и ваш код JavaScript. Хотя вы можете открыть инструменты разработки для редактора Construct, здесь это бесполезно.

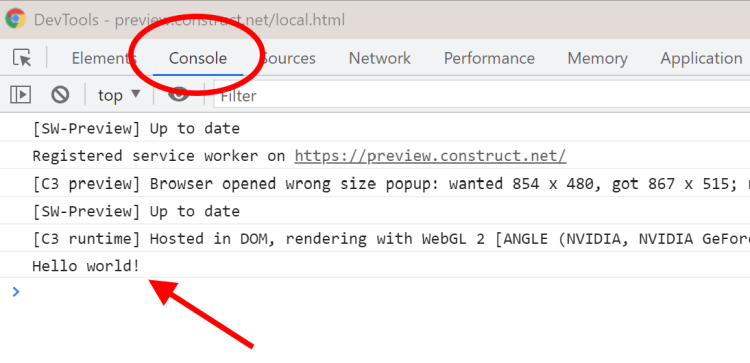
Поэтому прежде всего **просмотрите свой проект,** щелкнув значок «Воспроизвести» на панели инструментов рядом с кнопкой «Меню». (Вы также можете просмотреть, нажав F5.)

[Кнопка предварительного просмотра в Construct 3](https://construct-static.com/images/v1202/uploads/articleuploadobject/0/images/1096/preview-button.png)

Construct откроет всплывающее окно с пустым белым экраном, так как в макете ничего нет. Убедитесь, что окно предварительного просмотра является активным. Вы можете щелкнуть внутри окна предварительного просмотра, чтобы убедиться, что оно активно, а затем:

* В Safari нажмите Command+ Option+ I.
* В любом другом браузере нажмите F12.

Должно открыться новое окно для инструментов разработчика браузера. Перейдите на вкладку **«Консоль»** вверху, чтобы убедиться, что вы просматриваете список сообщений консоли. Вы должны увидеть Привет, мир! в списке сообщений! Могут быть и другие сообщения, которые Construct записал сам, но вы можете их игнорировать.

[](https://construct-static.com/images/v1202/uploads/articleuploadobject/0/images/65547/dev-tools-console.png)

Поздравляем! Вы запустили свою первую строку кода JavaScript и увидели записанное сообщение.

В этом руководстве мы будем использовать снимки экрана из инструментов разработчика Chrome. Инструменты разработчика других браузеров могут выглядеть немного иначе, но их все равно можно использовать таким же образом.

Если вы не можете открыть инструменты разработчика с помощью F12 в браузере Edge, перейдите в настройки и установите флажок « Использовать клавишу F12 для открытия инструментов разработчика» .

Если вы не видите свое сообщение в консоли или видите красную надпись «Стоп!» сообщение, возможно, вы открыли инструменты разработчика не в том окне. Закройте окно и повторите попытку, прежде чем нажимать сочетание клавиш, убедившись, что окно предварительного просмотра является активным.

## ИЗМЕНЕНИЕ СООБЩЕНИЯ

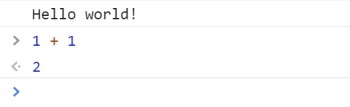
В Construct вы можете попробовать изменить зарегистрированное сообщение. (Не забудьте дважды щелкнуть блок сценария, чтобы отредактировать его, и щелкнуть снаружи, чтобы завершить.) Любой текст в двойных кавычках будет показан в консоли. Например, вы можете попробовать:

**console.log("My super cool message!");**

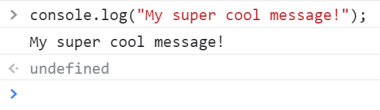
Просмотрите проект, откройте инструменты разработки, и вы увидите обновленное сообщение.

## ТЕСТИРОВАНИЕ КОДА В КОНСОЛИ

Как упоминалось ранее, замечательной особенностью консоли является то, что вы также можете ввести код и сразу же увидеть его запуск. Это очень удобный способ попробовать что-то. Внизу списка сообщений в консоли должно быть поле ввода, куда можно ввести какой-нибудь код. Попробуйте ввести **1 + 1**и нажать Enter. Он мгновенно запустит это как код JavaScript и зарегистрирует результат.

[](https://construct-static.com/images/v1202/uploads/articleuploadobject/0/images/65550/console-evaluation.png)

Вы также можете попробовать ввести код журнала консоли внутри консоли и увидеть, как он печатает там сообщение.

[](https://construct-static.com/images/v1202/uploads/articleuploadobject/0/images/65551/console-log-evaluation.png)

Вы можете задаться вопросом, почему **undefined**также появился. Это потому, что он регистрирует результат написанного вами кода. Когда писал **1 + 1**, результат был **2**. Однако **console.log()**не возвращает результат, поэтому результат **undefined**.

Вы можете ввести код в консоль таким образом в любой момент и сразу увидеть результат — или ошибку, если вы что-то ввели неправильно. Это отличный способ быстро поэкспериментировать с множеством различных вариантов кода и узнать больше о том, как работает JavaScript и какие результаты вы получаете от различных фрагментов кода. Не стесняйтесь опробовать его в любой момент в этом руководстве и поэкспериментировать с различными примерами кода — это отличный способ узнать больше.

## ЗАКЛЮЧЕНИЕ

В этой части мы рассмотрели:

* Об этом руководстве и о том, что вы узнаете из этой серии
* Как добавить код JavaScript в лист событий в Construct
* Использование **console.log()**для добавления сообщения в консоль браузера
* Как открыть инструменты разработчика браузера и просмотреть консоль
* Как ввести код в консоль, чтобы быстро опробовать

В этом руководстве для регистрации результатов выполнения кода часто используется консоль браузера, и она предоставляет отличное место для экспериментов с кодом, поэтому стоит потратить время на ее ознакомление.

# ИЗУЧИТЕ JAVASCRIPT В CONSTRUCT, ЧАСТЬ 2: ОСНОВЫ ЯЗЫКА

Это вторая часть серии руководств [«Изучите JavaScript в Construct»](https://www.construct.net/en/courses/learn-javascript-construct-79) . Это начинается с того места, где остановилась часть 1. Так что, если вы это пропустили, см. [«Изучаем JavaScript в Construct, часть 1: начало работы»](https://www.construct.net/en/tutorials/learn-javascript-construct-2800) .

Мы продолжим использовать тот же проект из предыдущей части. Вы можете отредактировать блок кода в событии «В начале макета», дважды щелкнув его, изменив код, а затем щелкнув за пределами или нажав Ctrl+ Enter, чтобы завершить редактирование. Затем вы можете просмотреть проект и открыть консоль браузера в инструментах разработчика, чтобы увидеть результаты. Стоит ознакомиться с этим процессом, поскольку мы будем в значительной степени полагаться на него. Попробуйте таким образом следующие фрагменты кода, чтобы увидеть, как они работают.

Иногда фрагменты кода предлагают вам попробовать их в консоли браузера. В этом случае не вводите их в свой блок кода в Construct — просмотрите проект, откройте консоль браузера и вместо этого попробуйте ввести туда каждую строку фрагмента кода.

## ЗАЯВЛЕНИЯ

Код JavaScript состоит из ряда операторов , разделенных точкой с запятой **;**. Например, этот код регистрирует два консольных сообщения, и каждая строка представляет собой оператор:

**console.log("First message!");**

**console.log("Second message!");**

Вы можете поместить два оператора в одну строку, поскольку на самом деле операторы разделяет точка с запятой. Однако традиционно в каждой строке содержится один оператор.

Точка с запятой в конце строки технически необязательна. Если вы забудете его туда положить, скорее всего, все будет работать. Однако это зависит от того, как JavaScript автоматически определяет, где ставить точки с запятой (функция под названием « Автоматическая вставка точки с запятой »), и это иногда может привести к неожиданным результатам. Поэтому лучше всего всегда использовать точки с запятой.

## КОММЕНТАРИИ

В программировании комментарий — это просто что-то, что вы можете написать в своем коде и что компьютер будет игнорировать. Другими словами, это примечание для людей, читающих код, а не инструкция для выполнения компьютером. Это аналогичная идея с комментариями в таблицах событий, которые также представляют собой просто заметки. В JavaScript есть два типа комментариев.

### ОДНОСТРОЧНЫЕ КОММЕНТАРИИ

Везде, где появляются два символа косой черты, **//**это означает однострочный комментарий. Все, начиная с косой черты и заканчивая концом строки, является комментарием. Комментарий может занимать отдельную строку или идти после некоторого кода.

**// This is a single-line comment!**

**console.log("Hello world!"); // And this!**

В этом учебном руководстве такие комментарии часто используются для аннотирования кода, указывая, что он делает или какой результат даст тот или иной код.

Вы заметите, что комментарии и другие фрагменты кода окрашены по-разному в этом руководстве, в Construct и в инструментах разработчика браузера. Это сделано для улучшения читаемости. (Цвет никак не влияет на работу кода.) В разных инструментах используются разные цветовые темы, но цель состоит в том, чтобы облегчить обнаружение таких вещей, как комментарии, поскольку они выделяются разным цветом.

### МНОГОСТРОЧНЫЕ КОММЕНТАРИИ

Однострочные комментарии естественным образом заканчиваются в конце строки, поэтому, если вам нужен длинный комментарий, вам нужно комментировать каждую строку:

**// First line of comment**

**// Second line of comment**

**// Third line of comment**

Многострочные комментарии могут быть более простым способом написать это. Многострочные комментарии начинаются с косой черты и звездочки **/\***и заканчиваются звездочкой и косой чертой **\*/**.

**/\* First line of comment**

**Second line of comment**

**Third line of comment \*/**

Несмотря на название, вы также можете написать такой комментарий в одну строку. Это может быть полезно, поскольку вы все равно можете писать код после окончания комментария.

**/\* comment before code \*/ console.log("Hello world!");**

**console.log(/\* comment inside code \*/ "Hello world!");**

В общем, вы можете оставлять комментарии где угодно, и это не повлияет на код. Как вы, возможно, заметили и в таблицах событий, добавление комментариев часто является хорошей идеей, чтобы помочь себе или другим людям понять код. Профессионально написанный код часто подробно комментируется, чтобы помочь любому, кто его читает, понять, как он работает. Лучший код ясен и прост для понимания, а не непонятен и сложен.

## ПЕРЕМЕННЫЕ

Переменная — это имя значения , хранящегося в памяти компьютера. Если вы использовали переменные в таблице событий Construct, например, глобальные или локальные переменные, концепция аналогична. Например, у вас может быть переменная с именем Score, в которой будет храниться число очков игрока. Значение может быть изменено или варьироваться , отсюда и название переменной .

### ОБЪЯВЛЕНИЕ И ПРИСВОЕНИЕ ПЕРЕМЕННЫХ

В JavaScript переменные объявляются с помощью **let**ключевого слова, за которым следует имя переменной. Код ниже создает переменную с именем message .

**let message;**

Значение, в данном случае строка (т.е. некоторый текст), может быть присвоено этой переменной с помощью оператора присваивания **=**.

**message = "Hello world!";**

Обратите внимание, что в данном случае **=**это не означает равенство . Это означает « присвоить правую часть левой». Вы также можете читать **=**как «установлено», например, для «сообщения» установлено значение «Привет, мир!» . Использование **=**этого способа — очень распространенный подход к присваиванию, используемый во многих языках программирования.

Теперь мы можем записать эту переменную в консоль. Вот полный фрагмент кода.

**let message;**

**message = "Hello world!";**

**console.log(message);**

Очень часто объявляют переменную и сразу же присваивают ей значение, поэтому и то, и другое можно объединить в один оператор, например:

**let message = "Hello world!";**

Несколько переменных также можно объявить с помощью запятых, с присвоением или без него. Например:

**let firstName, lastName, food = "pizza", drink = "cola";**

Эквивалентно:

**let firstName;**

**let lastName;**

**let food = "pizza";**

**let drink = "cola";**

Обычно мы рекомендуем использовать последний стиль с **let**объявлением каждой переменной, поскольку он более понятен.

Вы можете присваивать переменной разные значения. Следующий код устанавливает сообщение как Первое сообщение! а затем Второе сообщение! . Второе присвоение заменяет содержимое переменной, поэтому будет записано только второе сообщение! .

**let message;**

**message = "First message!";**

**message = "Second message!";**

**console.log(message);**

Что произойдет, если вы никогда не присвоите значение? Значением переменной по умолчанию в JavaScript является **undefined**специальное значение, означающее «ничего».

**let message;**

**console.log(message);**

Избегайте этого: использование **undefined**переменной обычно означает, что вы допустили ошибку. Чтобы избежать этого, рекомендуется всегда инициализировать переменную каким-либо значением.

### ПРАВИЛА ИСПОЛЬЗОВАНИЯ ПЕРЕМЕННЫХ

Вы должны объявить переменную перед ее использованием. Следующий код приведет к ошибке, поскольку он попытается использовать сообщение перед его объявлением.

**// ReferenceError: Cannot access 'message' before initialization**

**console.log(message);**

**let message = "Hello world!";**

Вы не можете объявить одно и то же имя переменной дважды.

**let message = "First message!";**

**// SyntaxError: Identifier 'message' has already been declared**

**let message = "Second message!";**

Имена переменных могут содержать только буквы, цифры или символы **$**и **\_**. Однако первый символ не может быть цифрой. Также имя переменной не может быть [зарезервированным словом](https://www.construct.net/out?u=https%3a%2f%2fdeveloper.mozilla.org%2fen-US%2fdocs%2fWeb%2fJavaScript%2fReference%2fLexical_grammar%23keywords) , то есть ключевым словом, используемым в языке JavaScript, например **let**.

**// Valid variable names:**

**let userName;**

**let user\_name;**

**let testing123;**

**// Invalid variable names:**

**let user-name; // dash '-' not allowed**

**let 9slice; // starting with digit not allowed**

**let let; // keyword not allowed**

Также обратите внимание, что JavaScript **чувствителен к регистру** , т. е. **message**рассматривается как имя переменной, отличное от имени **Message**или **MESSAGE**. В этом отличие от листов событий Construct, которые обычно не чувствительны к регистру.

Рекомендуется всегда использовать описательные имена переменных, чтобы облегчить понимание кода. Существуют также различные соглашения об именах, например, используете ли вы подчеркивание ( **first\_name**) или «верблюжий регистр» ( **firstName**). Не имеет большого значения, что вы выберете — просто будьте последовательны, и если вы работаете в команде, убедитесь, что все согласны с одним и тем же стилем именования.

## КОНСТАНТЫ

Вы также можете объявить константу, которая похожа на переменную, которую нельзя изменить. Они объявлены **const**вместо **let**. Некоторые значения никогда не должны меняться во время работы вашего кода, например математическая константа «пи» или максимальное разрешенное количество попыток входа в систему. Попытка изменить эти значения должна быть ошибкой и **const**требует этого.

**// Declare a constant**

**const PI = 3.141592653589793;**

**// TypeError: Assignment to constant variable.**

**PI = 4;**

Правила именования констант идентичны переменным, но общепринятым соглашением является присвоение константам имен в верхнем регистре, чтобы отличать их от переменных.

## ВАР

Вы можете встретить старый код JavaScript, используемый **var**для объявления переменных:

**var message = "Hello world!";**

Вы можете предположить, что это работает так же, как **let**. Однако избегайте его использования, поскольку это старая функция с некоторыми странными особенностями, в которые мы не будем здесь вдаваться. Современный код должен использовать только **let**.

## ТИПЫ ДАННЫХ

До сих пор мы работали только со строками (т.е. с текстом), такими как string **"Hello world!"**. JavaScript поддерживает несколько других типов данных. Здесь мы рассмотрим только два других: числа и логические значения.

### ЧИСЛА

Вы также можете использовать числовые значения вместо строк для переменных и констант. (На самом деле мы видели это ранее, когда использовали **1 + 1**в консоли и для константы **PI**.)

**let score = 100;**

**console.log(score);**

В JavaScript все числа являются числами с плавающей запятой , то есть они могут хранить как целые числа, например, **10**так и дробные значения, например **1.5**.

Вы можете использовать математические операторы с числами: **+**для сложения, **-**вычитания, **\***умножения и **/**деления. Вы можете попробовать это в консоли браузера — попробуйте ввести каждую строку ниже. В комментарии показан результат. Вы также можете попробовать свои собственные комбинации.

**// Try entering in to the console:**

**10 + 5 // 15**

**7 - 4 // 3**

**6 \* 3 // 18**

**20 / 4 // 5**

Эти расчеты можно включать в задания.

**let value = 10 + 5;**

**value = 6 \* 3; // replaces value**

**console.log(value + 1); // 19**

Вы также можете комбинировать вычисления, например **6 \* 3 + 1**, который соответствует стандартному порядку операций, при этом сначала выполняется умножение. Круглые скобки **(**и **)**могут использоваться для изменения порядка операций, например **6 \* (3 + 1)**.

Вы также можете использовать имя переменной в расчете:

**let a = 2 + 1.5;**

**let b = a \* 2;**

**console.log(b); // 7**

#### ТОЧНОСТЬ

Вы можете заметить странный результат с некоторыми дробями:

**// Try entering in to the console:**

**0.1 + 0.2 // 0.30000000000000004**

Это не точно! Это очень близко, но не совсем к правильному результату 0,3.

Это не ошибка. На самом деле именно так и происходят все дробные вычисления на компьютерах. Причина относительно сложна, но вкратце: компьютеры не имеют неограниченной памяти и поэтому вычисляют дроби только с ограниченной степенью точности. Рассмотрите возможность расчета **(2 / 3) \* 3**только до 6 знаков после запятой. **2 / 3**вычислил бы **0.666667**. Умножение этого числа на 3 даст **2.000001**- близко, но не совсем к предполагаемому результату - 2. В компьютерах этот тип округления происходит в двоичном формате (по основанию 2 или все 0 и 1), что означает, что он происходит в разных ситуациях, например как и **0.1 + 0.2**.

Нам, программистам, просто приходится с этим жить. Часто дробные вычисления будут немного неправильными, но иногда это не имеет значения, или программист следит за тем, чтобы значения в небольшом диапазоне считались приемлемыми, поэтому ошибки округления не влияют на результат.

#### СПЕЦИАЛЬНЫЕ НОМЕРА

Числа в JavaScript могут содержать три возможных специальных значения: положительную бесконечность ( **Infinity**), отрицательную бесконечность ( **-Infinity**) и «Не число» ( **NaN**).

Положительная и отрицательная бесконечность работают так, как и следовало ожидать:

**// Try entering in to the console:**

**1 / 0 // Infinity**

**-1 / 0 // -Infinity**

**Infinity + 1 // Infinity**

«Не число» или **NaN**, менее очевидно. Это результат, используемый, когда результат не может быть представлен математически, например, квадратный корень из отрицательного числа или неверный расчет.

**// Try entering in to the console:**

**0 / 0 // NaN**

В общем, если вы **NaN**это видите, значит, вы ошиблись в расчетах. Возможно, вам придется проверить правильность математических вычислений или добавить дополнительные проверки, чтобы убедиться, что вы не пытаетесь вычислить что-то вроде **0 / 0**. Это не всегда так очевидно, как кажется: вы можете использовать такие переменные, как **numberOfSweets / numberOfPeople**, и если обе переменные равны 0, результатом будет **NaN**.

### СТРУНЫ

Мы уже использовали строки, но давайте рассмотрим еще несколько деталей.

Почему мы называем текст в коде строкой ? Имя является сокращением от «строка символов», что на самом деле и представляет собой этот тип данных — последовательность отдельных символов, следующих один за другим. Строка имени обычно используется для обозначения типа данных текста в компьютерном программировании.

Для обозначения строки в строке могут использоваться символы **'**, **"**или **`**, при условии, что начальный и конечный символы совпадают. Все три примера ниже эквивалентны.

**"Hello world!"**

**'Hello world!'**

**`Hello world!`**

Не имеет значения, какой из них вы используете, но желательно соблюдать последовательность. В этом руководстве обычно для строк используются двойные кавычки.

Если вы хотите включить двойную кавычку внутри строки в двойных кавычках, вы можете использовать символ обратной косой черты \, чтобы экранировать ее. Это означает, что двойная кавычка является частью текста, а не отмечает конец строки.

**// Try entering in to the console:**

**"Hello \"world\"!" // Hello "world"!**

Это также означает, что сами обратные косые черты должны быть экранированы как \\. Вы можете использовать различные другие escape-последовательности, например \t для табуляции и \n для новой строки.

Альтернативой является использование для строки другой кавычки, например одинарных кавычек, например **'Hello "world"!'**.

Мы расскажем больше о строках в следующей части этого руководства.

### ЛОГИЧЕСКИЕ ЗНАЧЕНИЯ

Логическое значение может быть только либо **true**или **false**. Это полезно для значений включения/выключения, таких как настройки, а также для сравнения.

**let b = true;**

**console.log(b);**

Мы вернемся к логическим значениям позже в руководстве, поскольку они имеют некоторые ключевые применения в программировании на JavaScript, особенно при сравнениях и **if**операторах. Однако сейчас полезно знать, что логические значения — это еще один тип данных в JavaScript.

## ДИНАМИЧЕСКАЯ ТИПИЗАЦИЯ

JavaScript — это динамически типизированный язык, что означает, что вы можете присваивать переменным тип, отличный от того, с которого они начались. Код ниже действителен.

**let foo = "string";**

**foo = 10; // changing to number**

**foo = false; // changing to boolean**

JavaScript позволяет вам это сделать, но вам это не обязательно. Возможно, вам будет легче понять код, если переменные обычно сохраняют один и тот же тип. Используете ли вы эту функцию динамической типизации или нет, зависит от вашего предпочтительного стиля кодирования.

Также стоит упомянуть, что другие языки программирования имеют разные типы чисел, например, представление целых чисел и чисел с плавающей запятой (дробных) с отдельными типами данных. Однако JavaScript этого не делает, и существует только один тип чисел, который, как уже упоминалось, является типом чисел с плавающей запятой.

## ТИП

JavaScript также предоставляет **typeof**оператор, который может сказать вам, к какому типу относится что-либо. Он возвращает строку.

**// Try entering in to the console:**

**typeof 100 // number**

**typeof "Hello" // string**

**typeof true // boolean**

Это может помочь вам определить тип объекта, если вы не уверены, например:

**let myVar = "Hello"; // start off with string**

**myVar = 100; // change to number**

**// ... lots more code ...**

**// What type is myVar?**

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

Специальный тип **undefined**также возник ранее, и у него **"undefined"**также есть свой собственный тип.

**// Try entering in to the console:**

**typeof undefined // "undefined"**

Это также будет тип переменной, которой никогда ничего не присваивалось.

**let unassignedVariable;**

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

## ЗАКЛЮЧЕНИЕ

В этой части мы рассмотрели;

* Заявления
* Комментарии
* Переменные, присваивание и константы
* Числа, строки и логические типы данных
* Динамическая типизация
* Оператор**typeof**​

Надеюсь, теперь вы уже знакомы с вводом кода как в Construct, так и в консоли браузера. Руководство продолжит использовать оба варианта. В следующей части мы узнаем больше об операторах, логических значениях, сравнениях и типах преобразования.

## УЗНАТЬ БОЛЬШЕ

В конце каждой части этого руководства мы добавим ссылки, чтобы узнать больше о функциях JavaScript, упомянутых в [веб-документах MDN](https://www.construct.net/out?u=https%3a%2f%2fdeveloper.mozilla.org%2fen-US%2fdocs%2fWeb) , которые являются одним из лучших справочников по веб-разработке. Вы можете пропустить это и просто продолжить работу с руководством, если хотите, но если вас устраивает то, что вы узнали, и вы хотите копнуть глубже с более полной справкой или углубиться в более сложные детали, вы можете использовать эти ссылки, чтобы узнать больше.

# ИЗУЧИТЕ JAVASCRIPT В CONSTRUCT, ЧАСТЬ 3: ОПЕРАТОРЫ

Это третья часть серии руководств [«Изучите JavaScript в Construct»](https://www.construct.net/en/courses/learn-javascript-construct-79) . Эта часть является продолжением части 2. Если вы ее пропустили, см. раздел [«Изучаем JavaScript в Construct, часть 2: основы языка»](https://www.construct.net/en/tutorials/learn-javascript-construct-2802) .

В этой части мы продолжим использовать тот же проект, работая с фрагментами кода, введенными в Construct и в консоль браузера.

Если вы использовали таблицы событий Construct, вы можете заметить некоторое сходство между тем, как работают выражения Construct и выражения JavaScript. Однако имейте в виду, что выражения Construct не являются кодом JavaScript — это другая система, которая в некоторых отношениях работает аналогично.

## ОПЕРАТОРЫ

Во второй части мы использовали математические расчеты, такие как **10 + 5**. Вот **+**оператор сложения . Мы уже рассмотрели несколько операторов, таких как **-**вычитание, **\***умножение и **=**присваивание. В этой части мы рассмотрим ряд других операторов, которые часто используются в программировании.

### ДРУГИЕ МАТЕМАТИЧЕСКИЕ ОПЕРАТОРЫ

В JavaScript можно использовать еще два математических оператора:

* Остаток**%**
* Возведение в степень**\*\***

#### ОСТАТОК

Оператор **остатка** вычисляет остаток от деления, т.е. **a % b**вычисляет остаток от **a**деления на **b**. Например, **13 % 5**возвращается 3, потому что 13/5 равно 2 остатку 3. Другой способ подумать об этом: 13 — это 3 выше следующего наименьшего кратного 5 (то есть 10). Обратите внимание, что, несмотря на использование знака процента, это не связано с процентами.

**// Try entering in to the console:**

**13 % 5 // 3**

**5 % 2 // 1**

**27 % 10 // 7**

Это также работает с дробями в JavaScript:

**// Try entering in to the console:**

**5.5 % 2 // 1.5**

**9 % 3.5 // 2**

**3.25 % 1.25 // 0.75**

#### ВОЗВЕДЕНИЕ В СТЕПЕНЬ

Оператор возведения **в степеньa \*\* b** вычисляет **a**возведение в степень **b**. Например, **5 \*\* 2**возвращает **25**. Это также может работать с дробями — вы, возможно, помните, что возведение в степень половинной — это то же самое, что извлечение квадратного корня, поэтому **25 \*\* (1/2)**возвращается **5**.

**// Try entering in to the console:**

**6 \*\* 2 // 36**

**2 \*\* 3 // 8**

**100 \*\* (1/2) // 10**

**1.5 \*\* 2 // 2.25**

**6.25 \*\* 0.5 // 2.5**

Возможно, вы видели **^**символ, используемый для возведения в степень, но в JavaScript этот символ имеет другое значение, поэтому **\*\***используется вместо него.

В JavaScript также есть встроенные математические функции, такие как **Math.pow()**и **Math.sqrt()**которые вы также можете использовать вместо **\*\***.

## ОПЕРАТОРЫ ПРИСВАИВАНИЯ

Раньше мы **=**присваивали значение переменной. При работе с числами обычно присваивается переменная вычислению, которое включает в себя самого себя, например:

**let number = 7;**

**number = number \* 2;**

**console.log(number); // 14**

Посмотрите, как присваивание **number = number \* 2**присваивает номер вычислению, включающему его самого. В этом случае эта строка удвоит значение числа , поскольку она присваивает себе собственное значение, умноженное на 2.

В JavaScript для этого предусмотрены несколько дополнительных операторов присваивания, например **\*=**. Другими словами, **number \*= 2**это сокращение от **number = number \* 2**. Фрагмент кода ниже делает то же самое, что и раньше.

**let number = 7;**

**number \*= 2;**

**console.log(number); // 14**

Для каждого математического оператора, который мы пробовали до сих пор, существует специальный оператор присваивания:

* **a += b**такой же как**a = a + b**
* **a -= b**такой же как**a = a - b**
* **a \*= b**такой же как**a = a \* b**
* **a /= b**такой же как**a = a / b**
* **a %= b**такой же как**a = a % b**
* **a \*\*= b**такой же как**a = a \*\* b**

### ПРИСВАИВАНИЕ ВОЗВРАЩАЕТ ЗНАЧЕНИЕ

Интересно отметить, что присваивание фактически возвращает значение. Например, вы можете записать результат задания.

**let a = 2;**

**console.log(a += 2); // logs 4**

Присваивание возвращает значение переменной после завершения присваивания. В этом случае, поскольку **a**к нему добавлено 2, результатом присваивания является значение **a**после сложения, равное 4.

Это не имеет значения, если вы используете присваивание в отдельной строке, поскольку возвращаемое значение игнорируется. Однако это актуально для следующего раздела.

## УВЕЛИЧЕНИЕ И УМЕНЬШЕНИЕ

В программировании очень часто прибавляют или вычитают 1. Вы можете использовать это значение **a += 1**для увеличения **a**. Однако JavaScript предоставляет для этого дополнительное сокращение: **a++**.

**// These lines all do the same thing:**

**a = a + 1;**

**a += 1;**

**a++;**

Аналогично нужно **a--**вычесть 1.

Вы также можете поставить **++**или **--** перед именем переменной, например **++a**. Это делает почти то же самое, но возвращает другое значение:

* Постфиксная форма добавляет 1 к и возвращает старое значение до того , как было выполнено добавление.**a++a**
* Форма префикса**++a** добавляет 1 к **a**и возвращает новое значение после завершения добавления .

Вот пример кода, демонстрирующий разницу:

**let a = 1;**

**console.log(a++);**

**console.log(a);**

**1**Тогда это будет зарегистрировано **2**. В журнале **a++**будет отображено начальное значение **a**, а затем добавлено 1; в следующей строке снова отобразится значение, то есть 2. Вместо этого попробуйте изменить приращение на форму префикса:

**let a = 1;**

**console.log(++a);**

**console.log(a);**

**2**Тогда это будет зарегистрировано **2**. При ведении журнала **++a**сначала будет добавлено 1, а затем отображено результирующее значение 2. В следующей строке снова будет отображено значение, которое по-прежнему равно 2.

Поэтому, если вы используете возвращаемое значение присваивания **a += 1**и форму префикса, **++a**они одинаковы, поскольку они оба добавляют 1 и возвращают новое значение. Однако **a += 1**постфиксная форма **a++**не будет точно такой же, как **a++**при добавлении 1, но возвращает старое значение. Если вы не используете возвращаемое значение, например, если оно находится в отдельной строке, то **a += 1**, **++a**и **a++**все эквивалентны.

## СТРОКОВЫЕ ОПЕРАТОРЫ

До сих пор мы в основном рассматривали операторов, работающих с числами. Существуют также некоторые специальные применения операторов для строк.

Строки могут быть объединены , т.е. объединены вместе с помощью **+**. В этом случае, пока левая или правая часть **+**является строкой, строки объединяются. Только если и левая, и правая часть числа **+**являются числами, произойдет математическое сложение.

**// Try entering in to the console:**

**"Hello " + "world!" // Hello world!**

Таким образом вы также можете вставлять числа, включая переменные, при условии, что одна сторона **+**является строкой.

**let score = 100;**

**console.log("Your score is " + score + "!");**

**// Your score is 100!**

Строки, использующие обратный апостроф, **`**также имеют специальную функцию, позволяющую легко вставлять в текст другие значения с помощью **${value}**:

**let score = 100;**

**console.log(`Your score is ${score}!`);**

**// Your score is 100!**

Эти строки иногда называют литералами шаблона или строками шаблона , поскольку они похожи на текстовый шаблон, заполняемый значениями в определенных точках.

Эти функции являются полезными способами отображения значений в виде текста как для входа в консоль, так и для отображения информации пользователю. В этом руководстве также будут использоваться литералы шаблонов, чтобы упростить отображение сообщений, включая цифры.

### СМЕШИВАНИЕ ТИПОВ

Остерегайтесь распространенной ошибки — случайного смешивания типов. Обратите внимание **+**: если любая из сторон является строкой, выполняется конкатенация строк, включающая преобразование чисел в строку для их объединения. Иногда у вас может быть строка, содержащая последовательность цифр, поэтому она выглядит как число, например **"5"**. Однако это все еще строка, тип которой отличается от числа! Затем, если вы добавите что-нибудь к нему, вы получите конкатенацию строк.

**// Try entering in to the console:**

**"5" + "5" // "55" (string)**

**"5" + 5 // "55" (string)**

**5 + "5" // "55" (string)**

**5 + 5 // 10 (number)**

Это может произойти в менее очевидных обстоятельствах, например, если вы сохраняете строку, введенную пользователем, в переменную, а затем добавляете к этой переменной 1.

Поскольку такого рода ошибка может привести к запутанным результатам, обычно предпочтительнее использовать литералы шаблона вместо **+**создания строк и использовать их только **+**для математического сложения чисел.

## ПРЕОБРАЗОВАНИЕ МЕЖДУ СТРОКАМИ И ЧИСЛАМИ

JavaScript часто автоматически преобразует значения, например, строка + число автоматически преобразует число в строку, а затем добавляет его к первой строке. Однако иногда вам нужно намеренно преобразовать типы. Например, пользователь может ввести число в поле ввода текста, которое вернет строку, но вместо этого вы хотите использовать его как число. Также часто считается, что лучший стиль программирования — явное преобразование значений, а не полагаться на неявные функции языка, которые преобразуют значения за вас.

На самом деле существует множество способов явного преобразования значений в JavaScript, но сейчас мы сосредоточимся только на одном: использовании **Number()**и **String()**функциях. Использование **Number(string)**преобразует строку в число.

**// Try entering in to the console:**

**Number("5") // 5 (number)**

**Number("-6.5") // -6.5 (number)**

**Number(5) // 5 (still a number)**

**Number("Hello") // NaN**

Не забудьте обратить внимание на тип результатов — строки будут в кавычках, а числа — нет. В большинстве консолей браузеров для каждого типа также используются разные цвета, чтобы подчеркнуть разницу.

Обратите внимание, что в последнем числе мы снова получили специальный результат Not A Number (NaN). Это связано с тем, что строка «Привет» не представляет собой допустимое число, поэтому мы получаем специальный результат, указывающий на это.

Аналогично **String()**можно преобразовать число в строку.

**String(5) // "5" (string)**

**String(-6.5) // "-6.5" (string)**

**String("Hello") // "Hello" (still a string)**

**String(NaN) // "NaN" (string)**

Обратите внимание, что числа всегда можно преобразовать в строку. Если вы преобразуете специальное значение NaN в строку, вы получите строку с надписью «NaN». (Это преобразование прошло успешно; числовое значение было правильно преобразовано в соответствующую строку.)

Также обратите внимание, что в обоих случаях, если вы преобразуете тип в тот же тип, например **String("Hello")**, он просто возвращает одно и то же значение без изменений. Это полезно, поскольку если вы когда-либо не уверены, является ли значение строкой или числом, вы можете использовать **String(value)**, и вы знаете, что результат определенно является строкой.

## БУЛЕВЫ ОПЕРАТОРЫ

Помните, ранее мы рассмотрели логический тип, который представляет собой значения либо **true**или **false**. В этом разделе мы рассмотрим операторы, которые либо работают с логическими значениями, либо выдают логический результат.

### СРАВНЕНИЯ

JavaScript предоставляет несколько операторов сравнения, которые возвращают логическое значение. Например, **a < b**возвращается **true**, если **a**меньше **b**, в противном случае возвращается **false**. Операторы сравнения:

* Меньше **<**и больше чем**>**
* Меньше или равно **<=**и больше или равно**>=**
* Равно **==**и строгое равенство**===**
* Не равны **!=**и строги не равны**!==**

Обратите внимание: поскольку **=**используется для присваивания, нам нужно использовать что-то еще, чтобы проверить, действительно ли числа равны. Вот почему используются два или три равенства подряд, чтобы отличить сравнение от присваивания.

Почему существует два вида равенства? Стандартному методу равенства **==**разрешено преобразовывать типы, поэтому он может возвращать значения, **true**даже если типы различаются. Строгое равенство **===**не преобразует типы, поэтому всегда возвращает результат **false**, если типы различаются.

**// Try entering in to the console:**

**5 == 5 // true**

**5 == "5" // true (converts types)**

**5 === 5 // true**

**5 === "5" // false (types are different)**

Аналогичным образом **!=**можно преобразовывать типы, а **!==**при проверке на неравенство — нет.

**// Try entering in to the console:**

**5 != "5" // false (converts types)**

**5 !== "5" // true (types are different)**

Обычно считается лучшей практикой отдавать предпочтение сравнениям строгого равенства **===**и **!==**. Как отмечалось ранее, автоматическое преобразование типов может привести к запутанным результатам. Например, если вы сравниваете число со строкой, это часто является ошибкой, и вы хотели что-то преобразовать. Последовательное использование строгого равенства **===**и строгого не равно **!==**позволяет избежать путаницы, поскольку они включают сравнение типов. По этой причине в остальной части руководства будет предпочтительнее использовать строгие сравнения.

Распространенной ошибкой является путать присваивание **=**и равенство **==**или **===**. Например, если вы напишете **a = 5**вместо **a === 5**, вы фактически присвоите значение **a**вместо его сравнения. Постарайтесь убедиться, что вы используете правильные операторы!

Операторы меньше/больше не имеют строгих эквивалентов; к сожалению, они всегда могут конвертировать значения, **"3" < 5**например **true**. Нам просто приходится с этим жить, но, к счастью, проблема возникает реже, чем при проверке равенства.

Также стоит отметить, что строки можно сравнивать как меньше или больше. Строка считается меньшей, чем другая строка, если она предшествует ей в порядке сортировки. Например **"apple" < "banana"**, потому **true**, что в отсортированном списке яблоко будет стоять перед бананом (поскольку это сравнение основано на алфавитном порядке сортировки).

## ПРЕОБРАЗОВАНИЕ В ЛОГИЧЕСКОЕ ЗНАЧЕНИЕ

Строки и числа также можно преобразовать в логические значения, сведя их к значению **true**или **false**. Как и в случае с другими типами, это можно сделать с помощью **Boolean()**.

В JavaScript имеется множество правил преобразования различных типов данных и специальных значений в логические значения. Однако сейчас мы сосредоточимся только на следующих случаях.

* С **числами** и **0**специальным значением **NaN**преобразуется в **false**, а все остальное преобразуется в **true**.
* При использовании **строк** пустая строка **""**преобразуется в **false**, а все остальное преобразуется в **true**.
* Специальное значение **undefined**преобразуется в **false**.

См. следующие примеры.

**// Try entering in to the console:**

**Boolean(0) // false (is zero)**

**Boolean(NaN) // false (is NaN)**

**Boolean(1) // true (is not zero or NaN)**

**Boolean(42) // true (is not zero or NaN)**

**Boolean(Infinity) // true (is not zero or NaN)**

**Boolean("") // false (is empty string)**

**Boolean("Hello") // true (is not empty string)**

**Boolean("0") // true (is not empty string)**

**Boolean(undefined) // false**

**Boolean(true) // true (no change)**

**Boolean(false) // false (no change)**

В совокупности все значения, которые преобразуются в, **true**называются истинными значениями, а все значения, которые преобразуются в, **false**называются ложными значениями. Например, число 5 является правдивым, а пустая строка — ложной.

Обратите внимание, что правила истинности и ложности зависят от типа. Число **0**является ложным, поскольку оно равно нулю, но строка с содержимым **"0"**является правдивой, поскольку она не является пустой строкой.

## ЛОГИЧЕСКИЕ ОПЕРАТОРЫ

Есть некоторые операторы, которые особенно полезно использовать с логическими значениями.

### НЕТ

Вы можете инвертировать логическое значение – превратить его в **true**или **false**наоборот – с помощью оператора not **!**, который должен идти перед логическим значением, например **!b**.

**let b = true;**

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

### ИЛИ/И

Предположим, вы хотите узнать, истинно ли одно из двух условий или оба условия верны. Это можно определить с помощью логических операторов ИЛИ **||**и И.**&&**

Поскольку слова «или» и «и» регулярно встречаются в обычном письме, мы используем заглавные буквы OR и AND, чтобы было понятно, что мы говорим об операторах.

Например, **a < b || a < c**вернет **true**, если **a**меньше **b** или если **a**меньше **c**; в противном случае он возвращается **false**. Аналогичным образом **a < b && a < c**будет возвращено значение **true**if **a**is less then **b** и if **a**is less than **c**; в противном случае он возвращается **false**.

Вы можете использовать консоль для проверки всех комбинаций сравнений с логическими операторами:

**// Try entering in to the console:**

**false || false // false**

**false || true // true**

**true || false // true**

**true || true // true**

**false && false // false**

**false && true // false**

**true && false // false**

**true && true // true**

Эти операторы также работают с любыми истинными или ложными значениями, и результат также будет истинным или ложным, так же, как и с истинными и ложными. Например, **0 || 5**возвращает 5 (истинное значение) и **"Hello" && ""**возвращает «» (ложное значение). Это может показаться странным, но это может оказаться полезным при дальнейшем использовании таких вещей, как операторы if.

## УСЛОВНЫЙ ?: ОПЕРАТОР

Единственный тернарный оператор (оператор, который включает три значения) в Javascript — это условный оператор ?:. Это принимает форму **comparison ? valueIfTruthy : valueIfFalsy**. Например, следующий фрагмент кода будет регистрировать хороший результат! потому что переменная оценки 120 больше или равна 100.

**let score = 120;**

**console.log(score >= 100 ? "Nice score!" : "Try again!");**

Попробуйте изменить оценку для инициализации на **50**. Обратите внимание, что текст, записанный на консоль, меняется на « Повторите попытку!». , поскольку условие **score >= 100**стало ложным, поэтому возвращается последнее значение.

JavaScript часто принимает любое истинное или ложное значение при сравнении, и это применимо и здесь. Первое значение сравнения может быть другого типа, например числа или строки, и оно будет проверять, является ли значение правдивым или ложным.

**// Try entering in to the console:**

**0 ? "truthy" : "falsy" // "falsy"**

**1 ? "truthy" : "falsy" // "truthy"**

**"" ? "truthy" : "falsy" // "falsy"**

**"Hello!" ? "truthy" : "falsy" // "truthy"**

Еще раз, это пригодится, когда мы будем использовать операторы «если».

## ЗАКЛЮЧЕНИЕ

В этой части мы рассмотрели:

* Математические операторы **%**и**\*\***
* Дополнительные операторы присваивания, такие как **+=**и**\*=**
* Операторы увеличения и уменьшения **++**и**--**
* Создание строк с помощью **+**литералов шаблона и
* Преобразование между типами, в том числе между числами и строками, а также в логические значения и концепция истинных и ложных значений.
* Операторы сравнения, такие как **<**и**===**
* Логические операторы **!**, **||**,**&&**
* Условный **?:**оператор

В следующей части мы начнем использовать поток управления , такой как оператор if, о котором мы уже упоминали пару раз. Здесь вы можете начать писать программы, которые меняют свои действия в зависимости от условий — например, игру «Угадай число»!

# Изучите JavaScript в Construct, часть 4: поток управления

Это четвертая часть серии руководств [«Изучите JavaScript в Construct»](https://www-construct-net.translate.goog/en/courses/learn-javascript-construct-79?_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru) . Эта часть является продолжением части 3. Если вы ее пропустили, см. [«Изучаем JavaScript в Construct», часть 3: Операторы](https://www-construct-net.translate.goog/en/tutorials/learn-javascript-construct-2803?_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru) .

В этой части мы начнем с использования того же проекта, что и раньше. Позже мы представим новый проект, чтобы опробовать то, что мы узнали.

## Высказывания «если»

В программировании часто бывает так, что вы хотите делать некоторые вещи только при соблюдении определенных условий — например, если пароль правильный, то разрешите вход в систему. Это то, что ifделает заявление.

Блоки событий Construct работают немного похоже на оператор «if», поскольку, если условия выполнены, действия будут выполняться.

В JavaScript оператор if имеет следующую форму:

if (condition)

{

// Statements to run if condition is true

}

// The following statements always run

Обратите внимание, что операторы между оператором {и }после него ifвыполняются только в том случае, если условие истинно (или правдиво). Для условия полезно использовать различные операторы сравнения, которые мы рассмотрели в предыдущей части, например number < 10. Попробуйте следующий фрагмент кода в своем проекте Construct и посмотрите в консоли браузера результат.

let number = 5;

if (number < 10)

{

console.log("Number is less than 10!");

}

console.log("Logged after if statement");

При этом будут зарегистрированы оба числа меньше 10! а затем Зарегистрировано после оператора if . Поскольку number < 10это правда, код внутри оператора if запускается, а затем всегда запускается код после него.

Теперь попробуйте изменить numberинициализацию на 15. number < 10Теперь условие ложно, поэтому код внутри оператора if не запускается. Поэтому единственным зарегистрированным сообщением является «Записано после оператора if» , поскольку оно идет после оператора «if».

Мы работаем с предсказуемыми значениями, чтобы упростить задачу. Скоро мы перейдем к более интересному примеру!

Подобно ?:условному оператору, условие оператора if может иметь любое значение и будет интерпретироваться как истинное или ложное. Например, вы можете использовать just if (number), и это будет считаться ложным, если numberоно равно нулю или NaN, в противном случае — true.

### Еще

За оператором «If» также может сразу следовать оператор «else». Код внутри оператора else выполняется, если условие ложно (или ложно). Попробуйте следующий фрагмент кода.

let number = 15;

if (number < 10)

{

console.log("Number is less than 10!");

}

else

{

console.log("Number is greater or equal to 10!");

}

Поскольку numberинициализируется значением 15, условие number < 10ложно, поэтому код в операторе else запускается и регистрируется: Число больше или равно 10!

Когда есть операторы «if» и «else», один из двух разделов гарантированно будет выполнен, поскольку они соответствуют двум возможным результатам условия.

### Иначе, если

else ifИспользуя такой шаблон, можно объединить несколько условий .

let number = 15;

if (number < 10)

{

console.log("Number is less than 10!");

}

else if (number < 20)

{

console.log("Number is between 10 and 20!");

}

else

{

console.log("Number is greater or equal to 20!");

}

Попробуйте изменить начальное значение, numberчтобы каждое сообщение появлялось.

### Дополнительные брекеты

{Вы можете поместить в фигурные скобки столько операторов, сколько захотите }, включая дополнительные операторы «if». Однако если между фигурными скобками находится только один оператор, фигурные скобки можно опустить. Например это:

if (number < 10)

{

console.log("Number is less than 10!");

}

эквивалентно этому:

if (number < 10)

console.log("Number is less than 10!");

поскольку существует только одно утверждение. Это также относится к цепочкам «else» и «else if», например так:

let number = 15;

if (number < 10)

{

console.log("Number is less than 10!");

}

else if (number < 20)

{

console.log("Number is between 10 and 20!");

}

else

{

console.log("Number is greater or equal to 20!");

}

эквивалентно этому:

let number = 15;

if (number < 10)

console.log("Number is less than 10!");

else if (number < 20)

console.log("Number is between 10 and 20!");

else

console.log("Number is greater or equal to 20!");

### Пробелы

В общем, JavaScript не заботится о пробелах (т.е. пробелах и табуляции). Это обеспечивает большую гибкость в компоновке кода. В следующем фрагменте кода показано несколько способов написания оператора if, все из которых эквивалентны.

if (number < 10)

{

console.log("Number is less than 10!");

}

if (number < 10) {

console.log("Number is less than 10!");

}

if (number < 10)

console.log("Number is less than 10!");

if(number<10)console.log("Number is less than 10!");

if (number < 10) { console.log("Number is less than 10!"); }

if (number < 10)

{

console.log("Number is less than 10!");

}

На практике лучше всего придерживаться разумного стиля (избегая крайностей, связанных с добавлением большого количества пробелов или удалением как можно большего количества пробелов) и просто быть последовательными в использовании одного и того же стиля. Это руководство будет придерживаться первой формы выше.

### Предпочитаю брекеты

Хотя отсутствие фигурных скобок иногда является полезным сокращением, обычно предпочтительнее их включать. Это помогает сохранить код ясным и читаемым, а также избежать ошибок.

Одна из распространенных ошибок при исключении фигурных скобок связана с тем, что JavaScript игнорирует отступы — они нужны только для удобства чтения. Поэтому, если вы напишете такой код:

if (number < 10)

console.log("First message!");

console.log("Second message!");

Это может выглядеть так, будто оба сообщения регистрируются, если number < 10, но на самом деле это эквивалентно следующему:

if (number < 10)

{

console.log("First message!");

}

console.log("Second message!");

Это связано с тем, что если фигурные скобки опущены, только один следующий оператор считается внутренним, независимо от отступа. Предпочтение использованию брекетов помогает избежать этой потенциальной ошибки.

### Предпочитайте операторы «if» ?:

Иногда можно использовать оператор if или условный ?:оператор. Например, следующий оператор «if»:

let number = 5;

let message;

if (number < 5)

{

message = "Number is less than 5!";

}

else

{

message = "Number is greater or equal to 5!";

}

console.log(message);

эквивалентно этому:

let number = 5;

let message = number < 5 ? "Number is less than 5!" : "Number is greater or equal to 5!";

console.log(message);

Несмотря на то, что последний содержит меньше строк кода, первый обычно считается более читабельным. Поэтому, если у вас есть сомнения, лучше использовать операторы «если». Условный оператор имеет свое применение, но лучше всего, когда все три его части очень короткие и простые.

## Игра «Угадай число»

Чтобы продемонстрировать, как на практике использовать операторы «если», вот простая игра «Угадай число», реализованная на JavaScript. Компьютер генерирует случайное число от 0 до 100, и когда вы делаете предположение, он сообщает вам, является ли это число больше, меньше или равно вашему предположению. Это позволяет игроку сделать ряд предположений и в конечном итоге определить число. Откройте проект в Construct и убедитесь, что вы просматриваете вкладку «Лист событий 1» .

.C3P

#### угадай число-js.c3p

[Скачать сейчас49,84 КБ](https://translate.google.com/website?sl=auto&tl=ru&hl=ru&u=https://construct-static.com/tutorials/downloads/66850/guess-the-number-js.c3p)

Вы должны увидеть GuessButton: при нажатии триггера с блоком сценария, который будет запускаться в качестве действия. Это означает, что фрагмент кода JavaScript будет запускаться каждый раз, когда игрок нажимает кнопку «Угадать».

В других местах этого проекта есть больше кода JavaScript, но сейчас мы сосредоточимся только на этом одном блоке кода, поскольку он является хорошей демонстрацией операторов «если».

Попробуйте просмотреть проект и сделать ряд предположений, пока не найдете число. Код, который запускается при нажатии кнопки «Угадать», выглядит следующим образом:

// This is the number the player must guess.

let theNumber = getTheNumber();

// This is the number the player just guessed.

let guessedNumber = getEnteredNumber();

let message;

// Set the message to display.

if (guessedNumber < theNumber)

{

message = "Higher!";

}

else if (guessedNumber > theNumber)

{

message = "Lower!";

}

else

{

message = "That's it!";

}

// Show the message.

showResultMessage(message);

Этот блок кода использует некоторые функции, объявленные в другом месте проекта: getTheNumber() , getEnteredNumber() и showResultMessage() . Пока не беспокойтесь об этом (в этом руководстве функции будут рассмотрены позже). На данный момент все, что вам нужно знать, это три переменные:

1. theNumber, случайное число, которое игрок должен угадать.
2. guessedNumber, это число, которое игрок ввел, когда нажал «Угадай».
3. message, это строка, которая будет отображаться на экране

Блок кода использует операторы «if» для сравнения введенного игроком числа с числом, которое ему нужно угадать. Если угаданное число меньше, оно устанавливает сообщение «Выше!», Указывая, что нужно угадать большее число; если угаданное число больше, отображается сообщение «Ниже!», указывающее, что нужно угадать меньшее число; и если они все сделали правильно, сообщение устанавливается на «Вот и все!».

Это простой пример, но он демонстрирует, как можно использовать операторы «if» для отображения пользователю различных результатов в зависимости от входных данных и превратить это в небольшую игру!

## Пока циклы

Еще одна важная часть программирования — циклы . Это фрагменты кода, которые можно повторять снова и снова. В этом руководстве мы рассмотрим два типа циклов: циклы while и циклы for . (Есть и другие типы циклов JavaScript, но это два основных типа циклов.)

На первый взгляд циклы while имеют структуру, аналогичную оператору if: условие с серией операторов в фигурных скобках.

while (condition)

{

// statements to repeat

}

Однако в отличие от оператора «если» он может повторять содержащиеся в нем операторы. Это работает следующим образом:

1. Проверьте, истинно ли условие (или правдиво).
2. Если это правда, запустите инструкции и вернитесь к шагу 1.
3. Если это ложь, не запускайте утверждения. Он продолжится до следующего оператора после цикла.

Другими словами, он повторяет операторы цикла, пока условие истинно — отсюда и название цикла while .

Давайте вернемся к проекту, который мы использовали ранее, с блоком кода JavaScript в событии «При запуске макета», чтобы опробовать его. Используйте следующий код:

let i = 0;

while (i < 3)

{

console.log(`Variable 'i' is ${i}`);

i++;

}

Вот файл проекта, в который это уже введено, чтобы можно было легко вернуться к этой настройке.

.C3P

#### while-loop.c3p

[Скачать сейчас46,86 КБ](https://translate.google.com/website?sl=auto&tl=ru&hl=ru&u=https://construct-static.com/tutorials/downloads/66851/while-loop.c3p)

Просмотрите проект и проверьте консоль браузера. Он должен зарегистрировать три сообщения:

Variable 'i' is 0

Variable 'i' is 1

Variable 'i' is 2

Обратите внимание, что мы использовали шаблонную строку с символами обратной кавычки `, используя ${i}для вставки значения iin в строку.

Как это работает? Операторы внутри цикла, также известные как тело цикла, записывают значение iв консоль браузера, а затем увеличивают его i(т. е. добавляют к нему 1). Значение iначинается с 0, поэтому тело цикла будет повторяться три раза:

1. First iравен **0** , это i < 3правда. Тело цикла выполняется, печатая значение iи увеличивая его до 1.
2. Тогда iравно **1** , поэтому i < 3это правда, и тело цикла выполняется снова.
3. Тогда iравно **2** , что i < 3верно, и тело цикла выполняется снова.
4. Тогда iравно **3** и i < 3теперь ложно. Цикл заканчивается.

Конечным результатом является цикл, который повторяет некоторый код три раза.

## Для петель

Оказывается, что шаблон, который мы только что использовали, — использование переменной для подсчета и повторения некоторого кода определенное количество раз — очень распространен в программировании. Поэтому большинство языков программирования, включая JavaScript, предоставляют более краткий способ написания этого.

В общих чертах картина следующая:

begin

while (condition)

{

// loop body

step

}

Цикл for объединяет все это в один оператор в форме:

for (begin; condition; step)

{

// loop body

}

Таким образом, следующий цикл for идентичен циклу while, который мы использовали ранее, только использует более короткую форму:

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

{

console.log(`Variable 'i' is ${i}`);

}

Это обеспечивает краткий способ повторить некоторый код определенное количество раз — попробуйте изменить 3условие на другое число, и вы увидите, что сообщение записывается соответствующее количество раз.

Обратите внимание на использование циклов, начинающихся с 0 вместо 1. Большинство вещей в программировании работают с индексом, начинающимся с нуля, где первый элемент имеет номер 0. Также имя переменной в цикле for может быть любым, но использование iявляется общепринятым соглашением. .

Каждая из трех частей цикла for не является обязательной. Например, часть «начало» может быть опущена, например, если переменная уже объявлена.

let i = 0;

for ( ; i < 3; i++) // note first part left empty

{

console.log(`Variable 'i' is ${i}`);

}

Пустое условие также считается true- поэтому цикл будет выполняться вечно! Вскоре это снова появится.

Если вы использовали циклы с блоками событий Construct, обратите внимание, что циклы while и for в JavaScript аналогичны условиям цикла While , Repeat и For в объекте System. While повторяется, пока выполняются все следующие условия. Повтор и For похожи на цикл JavaScript for, позволяющий повторять действия определенное количество раз.

## Перерыв

Этот breakоператор можно использовать для досрочного завершения цикла. Попробуйте следующий цикл — он настроен на выполнение 10 раз, но breakостановится, когда iдостигнет 5.

for (let i = 0; i < 10; i++)

{

console.log(`Variable 'i' is ${i}`);

if (i === 5)

break; // end loop early

}

Это приведет к регистрации переменной 'i' равно 5 , а затем досрочному завершению цикла при выполнении breakоператора. Вы можете использовать breakкак циклы for, так и while.

### Бесконечные циклы

Можно создать цикл с условием, которое всегда истинно, поэтому тело цикла будет повторяться вечно. Это можно сделать с помощью while (true)или for (;;)(где все три части цикла for остаются пустыми: ни одна переменная не объявлена, ни условие, ни шаг).

Тогда единственный способ завершить цикл — это оператор break. В какой-то момент это необходимо запустить, иначе цикл никогда не закончится. Это одна из возможных причин «зависания» программного обеспечения, то есть зависания и прекращения ответа — оно застряло в бесконечном цикле и никогда не получает возможности ответить на дальнейшие вводимые данные.

Будьте осторожны, если вы экспериментируете с бесконечными циклами в Construct. Если вы допустили ошибку и breakоператор никогда не запускается, вы можете зависнуть окно предварительного просмотра, что, в свою очередь, может привести к зависанию Construct или браузера! Один из способов смягчить это — установить для параметра «Использовать работника» значение «Да» в разделе «Дополнительно» свойств проекта, но закрыть замороженное окно все равно может быть сложно даже в этом случае.

Вот пример предположительно бесконечного цикла, который используется breakдля его завершения.

let i = 0;

while (true)

{

console.log(`Variable 'i' is ${i}`);

i++;

if (i === 5)

break; // end infinite loop

}

Это эквивалентно использованию цикла for для повторения определенного количества раз. Однако этот подход к написанию бесконечного цикла, который заканчивается, breakможет быть полезен в более сложных ситуациях, когда вы на самом деле заранее не знаете, сколько итераций необходимо выполнить, поэтому полезно знать об этом подходе.

## Продолжать

Оператор continueдосрочно завершает тело цикла, но продолжает выполнение цикла. Каждый запуск цикла называется итерацией , иначе говоря, он отменяет текущую итерацию и переходит к следующей итерации. Попробуйте следующий пример кода.

for (let i = 0; i < 10; i++)

{

if (i === 5)

continue; // skip logging message

console.log(`Variable 'i' is ${i}`);

}

При этом регистрируются все значения iот 0 до 9, но пропускаются 5. Это связано с тем, что при iзначении 5 выполняется continueоператор, который переходит к следующей итерации, где iравно 6. Вы можете использовать continueкак циклы for, так и циклы while. .

## Корпус переключателя

Этот switchоператор похож на серию операторов if/else if. Он принимает значение и переходит к соответствующей caseметке или defaultметке, если ни одна не соответствует. Вот образец switchзаявления.

let number = 5;

switch (number) {

case 0:

console.log("Number is 0!");

break;

case 5:

console.log("Number is 5!");

break;

case 10:

console.log("Number is 10!");

break;

default:

console.log("Number is something else!");

break;

}

Это эквивалентно следующей серии операторов if:

let number = 5;

if (number === 0)

{

console.log("Number is 0!");

}

else if (number === 5)

{

console.log("Number is 5!");

}

else if (number === 10)

{

console.log("Number is 10!");

}

else

{

console.log("Number is something else!");

}

Обратите внимание, что с switchутверждением:

* Каждое возможное значение указывается в caseметке, за которой следует двоеточие :(а не точка с запятой), и оно соответствует строгому равенству (как если бы оно сравнивалось с ===).
* Если ни один из заданных случаев не соответствует, происходит переход к defaultметке
* Каждый раздел caseand defaultзаканчивается утверждением break. Это выходит из switchутверждения. (Это то же ключевое слово, которое используется и для выхода из циклов.)

Этот defaultраздел является необязательным, и в этом случае, если ни один регистр не соответствует, он просто пропускает весь switchоператор.

### Дополнительный перерыв

Оператор breakпосле каждого случая на самом деле необязателен. Если он опущен, то он «проваливается» и продолжает выполнять следующие строки кода из следующего случая! Попробуйте следующий фрагмент кода, breakубрав все операторы, чтобы увидеть, как это работает.

let number = 5;

switch (number) {

case 0:

console.log("Number is 0!");

case 5:

console.log("Number is 5!");

case 10:

console.log("Number is 10!");

default:

console.log("Number is something else!");

}

Это зарегистрирует номер 5! , Номер 10! а номер - это нечто другое! . Причина в том, что, поскольку numberзначение равно 5, он перейдет к нему case 5:и начнет выполнять операторы оттуда. Поскольку операторов нет break, он продолжит выполнение всех операторов журнала консоли до конца. В некоторых случаях это ошибка, но она имеет некоторые применения, например, для сопоставления нескольких случаев с одной и той же строкой кода, как показано ниже.

let number = 5;

switch (number) {

case 0:

console.log("Number is 0!");

break;

case 5:

case 10:

console.log("Number is 5 or 10!");

break;

default:

console.log("Number is something else!");

break;

}

Теперь, если numberчисло равно 5 или 10, будет записано число 5 или 10! . Причина в том, что случай 5 может перейти в случай 10 и выполнить ту же строку кода. Это похоже на оператор if с условием number === 5 || number === 10.

Зачем использовать switchзаявление? Это может быть кратким способом сопоставления длинного списка возможных значений. Однако во многих случаях ту же самую работу выполняет серия операторов if/else if. Это еще один вариант выбора стиля. Стоит знать, как switchработают операторы, поскольку они широко используются, чтобы вы, скорее всего, с ними столкнулись.

## Заключение

В этой части мы рассмотрели различные функции, управляющие выполнением кода, также известные как поток управления , в том числе:

* утверждения «если»
* «иначе» и «иначе, если»
* Форматирование (например, пробелы и необязательные фигурные скобки)
* Простая игра «Угадай число».
* циклы «пока» и «для»
* breakиcontinue
* switchзаявления с caseи defaultметками

В следующей части мы рассмотрим еще один основной строительный блок программирования: функции! Мы также перейдем к использованию файлов сценариев вместо фрагмента кода в таблице событий.

# Изучите JavaScript в Construct, часть 5: Функции

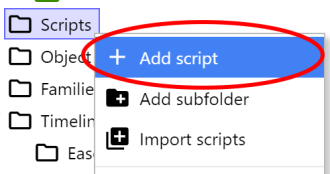
Это пятая часть серии руководств [«Изучите JavaScript в Construct»](https://www-construct-net.translate.goog/en/courses/learn-javascript-construct-79?_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru) . Эта часть является продолжением части 4. Если вы ее пропустили, см. [«Изучение JavaScript в Construct», часть 4: Поток управления](https://www-construct-net.translate.goog/en/tutorials/learn-javascript-construct-2811?_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru) .

В этой части мы перейдем к использованию файлов сценариев, а не блоков сценариев в листе событий. Это продемонстрирует, как использовать файлы сценариев в Construct, а файлы сценариев также предоставляют более удобное место для написания более длинных фрагментов кода.

## Использование файла сценария

Давайте начнем эту часть заново. Закройте существующий проект, если он у вас открыт, и снова создайте новый пустой проект в Construct.

На панели проекта **щелкните правой кнопкой мыши папку** «Сценарии» и выберите « **Добавить сценарий»** .

[](https://translate.google.com/website?sl=auto&tl=ru&hl=ru&u=https://construct-static.com/images/v1202/uploads/articleuploadobject/0/images/67226/add-script.png)

Это добавит новый файл сценария в ваш проект. Откроется редактор кода, который использует полный средний вид, поскольку мы больше не используем лист событий.

Первый файл сценария, добавляемый Construct, содержит некоторый код по умолчанию, который призван облегчить интеграцию вашего кода с Construct. Однако нам это не понадобится. Поэтому **выберите весь код** и **удалите** его. Теперь вы должны увидеть пустой редактор кода.

Раньше мы использовали код JavaScript вместо действия в листе событий. Это означало, что код запускался при каждом запуске блока событий, например, при запуске макета или при нажатии кнопки. Теперь мы используем файлы сценариев, и весь код верхнего уровня будет запускаться автоматически сразу после загрузки страницы. На самом деле это произошло еще до того, как Construct начал работу над проектом, но для целей данного руководства это не имеет значения.

Вы можете проверить, работает ли это, снова введя нашу первую строку кода в файл сценария:

console.log("Hello world!");

Просмотрите проект и проверьте консоль браузера, и вы увидите сообщение.

Этот код теперь запускается еще до того, как Construct запускает проект, и Construct регистрирует несколько сообщений при запуске проекта. Таким образом, вы можете увидеть, что сообщение консоли теперь появляется между другими сообщениями, зарегистрированными Construct. Однако это не имеет значения — как и раньше, вы можете просто игнорировать другие сообщения консоли.

Теперь мы перешли к использованию файла сценария. Поэтому с этого момента примеры кода из этого руководства следует вводить в редактор кода для файла сценария.

## Функции

Функции — еще одна важная часть всего компьютерного программирования. Функция — это, по сути, группа операторов, которую вы можете повторно использовать в нескольких местах. Они также являются хорошим способом убедиться, что код хорошо организован и содержит различные функции для каждой из ключевых задач вашей программы.

Если вы использовали функциональные блоки в таблицах событий Construct, принцип аналогичен.

### Объявление функции

В JavaScript функцию можно объявить следующим образом:

function logMessage()

{

console.log("Hello world!");

}

При этом создается функция с именем logMessage , которая будет регистрировать Hello world! на консоль. Сам по себе этот код фактически ничего не делает. Только что объявлено, что функция существует, но на самом деле ее еще никто не использовал.

### Вызов функции

Чтобы выполнить инструкции внутри функции, нам нужно вызвать функцию. Это делается с использованием его имени, за которым следуют круглые скобки (). Попробуйте этот фрагмент кода, который добавляет вызов функции.

// Declare function

function logMessage()

{

console.log("Hello world!");

}

// Call the function

logMessage();

Теперь это будет журнал Привет, мир! на консоль, поскольку мы как объявили функцию (описав, что она делает), так и вызвали ее (которая фактически ее запускает).

Эту функцию можно вызывать в нескольких местах, что избавляет от необходимости повсюду повторять весь код внутри функции. В данном случае это может показаться небольшой экономией, но реальные программы могут иметь длинные и сложные функции, что делает экономию гораздо большей.

Обратите внимание, что если вы неоднократно регистрируете одно и то же сообщение консоли, многие консоли браузера избегают повторения и просто добавляют число рядом с сообщением, указывающее, сколько раз оно было повторено. Вы заметите это, если позвоните logMessage()дважды. Чтобы избежать путаницы, в этом руководстве мы позаботимся о том, чтобы все примеры кода всегда регистрировали разные сообщения.

## Параметры

Функциям также могут передаваться параметры , которые являются дополнительными значениями, которые могут использоваться телом функции. Они заключаются в круглые скобки ()и разделяются запятыми. Использование нашего предыдущего кода ()на самом деле означает «без параметров». Следующий код объявляет функцию, которая использует два параметра для двух чисел для отображения в сообщении журнала, а затем дважды вызывает эту функцию с разными значениями параметров.

// Declare function with two parameters

function logNumbers(firstNumber, secondNumber)

{

console.log(`The first number is ${firstNumber} and the second number is ${secondNumber}`);

}

// Call the function twice with different parameters

logNumbers(3, 4);

logNumbers(6, 7);

Это зарегистрирует два сообщения:

* Первое число 3, второе число 4.
* Первое число 6, второе число 7.

Обратите внимание, как значения, передаваемые при вызове, например 3 и 4 в logNumbers(3, 4), передаются в соответствующие параметры firstNumberи secondNumberв функцию logNumbers . Затем он запускает код внутри функции, используя предоставленные значения.

Внутри функции logNumbers параметры firstNumberи secondNumberпо сути являются переменными, но их не нужно объявлять с помощью let— только имя — это все, что необходимо при объявлении параметров функции.

## Объем

Пришло время рассмотреть еще один аспект переменных: область видимости . Это относится к тому месту в вашем коде JavaScript, где вы можете ссылаться на переменную. Это зависит от того, где объявлена ​​переменная.

Функцию logNumbers можно настроить для использования переменной внутри функции для сообщения:

function logNumbers(firstNumber, secondNumber)

{

let message = `The first number is ${firstNumber} and the second number is ${secondNumber}`;

console.log(message);

}

Переменная message ограничена функцией logNumbers .​ Другими словами, его можно использовать только внутри этой функции. Его нельзя использовать вне этой функции, поскольку он находится за пределами ее области действия — фактически он больше нигде не существует.

Один из способов продемонстрировать это — попытаться получить доступ к сообщению вне функции. Обратите внимание, что в ошибке указано, что сообщение не определено.

function logNumbers(firstNumber, secondNumber)

{

let message = `The first number is ${firstNumber} and the second number is ${secondNumber}`;

console.log(message);

}

// ReferenceError: message is not defined

console.log(message);

Если вы предварительно просмотрите свой проект с этим кодом, Construct отобразит сообщение, уведомляющее вас о том, что при выполнении основного сценария произошла ошибка. Этого и следовало ожидать: мы ввели неверный код. Отклоните запрос, а затем откройте консоль браузера как обычно, чтобы увидеть сообщение об ошибке.

То же самое относится и к параметрам функции firstNumber и SecondNumber : они доступны только внутри тела функции.

То же самое относится и к переменным, объявленным в операторах if и циклах while и for. По сути, везде, где вы видите фигурные скобки {и }, любые переменные, объявленные внутри них, доступны только внутри этого диапазона, поскольку фигурные скобки определяют новую область видимости.

### Область видимости в циклах

Одним небольшим исключением из этого правила является переменная, объявленная как часть цикла for, которая также доступна как внутри, for (...)так и между фигурными скобками, но не где-либо за ее пределами:

// 'i' cannot be used here

// 'i' can be used inside for (...)

for (let i = 0; i < 10; i++)

{

// 'i' can be used here

}

// 'i' cannot be used here

Это дает переменной ограниченную область действия по сравнению с эквивалентным циклом while:

// 'i' cannot be used here (prior to its declaration)

let i = 0;

// 'i' can be used here

while (i < 10)

{

// 'i' can be used here

i++

}

// 'i' can be used here

На самом деле это полезный аспект циклов for, поскольку он гарантирует, что переменная цикла уникальна для цикла.

### Внутренняя область действия

Хотя переменные нельзя использовать вне их области действия, их можно использовать где угодно внутри своей области действия. Один из способов продемонстрировать это состоит в том, что функции могут ссылаться на переменные, объявленные вне функции.

let message = "Hello world!";

function logMessage()

{

// OK: 'message' is still in scope

console.log(message);

}

logMessage();

Переменная по-прежнему находится в области видимости внутри любых вложенных фигурных скобок в своей области видимости. Он пытается получить доступ к переменной за пределами своей области действия, что приводит к ошибке.

### Область верхнего уровня

Переменная, объявленная на «верхнем уровне» файла сценария, т. е. не внутри каких-либо фигурных скобок, как в случае с сообщением в предыдущем примере, распространяется на весь этот файл. Если вы добавите еще один файл сценария, переменная не будет там доступна, поскольку другие файлы находятся за пределами области действия переменной верхнего уровня.

Обратите внимание, что область верхнего уровня может работать по-другому в других средах JavaScript. Construct использует современные «модульные» скрипты, в которых область верхнего уровня работает таким образом. Однако если другие инструменты или среды используют старые «классические» сценарии, область верхнего уровня фактически является глобальной областью действия, и поэтому переменные верхнего уровня можно использовать в разных файлах. Несмотря на то, что этого не происходит в Construct, это все еще распространено в остальной части отрасли, поэтому стоит понимать, что в других местах это может работать по-другому.

### Функции тоже имеют область видимости

Функции также ограничены местом их объявления. Снова обращаясь к предыдущему примеру, функция logMessage объявлена ​​на верхнем уровне и поэтому может использоваться где угодно внутри этого файла сценария, но не в других файлах сценария.

Чтобы продемонстрировать возможности функций, попробуйте этот пример кода:

function logMessageOuter()

{

function logMessageInner()

{

console.log("Hello world!");

}

logMessageInner();

}

// 'logMessageInner' is not accessible here

logMessageOuter();

Это объявляет вложенную функцию или функцию внутри другой функции. logMessageInner можно использовать только в функции logMessageOuter , а не за ее пределами. Область действия функции аналогична переменной, объявленной внутри logMessageOuter .

### Краткое содержание

Область действия — это важный аспект объявлений переменных, объявлений функций и параметров функций, который определяет, где их можно использовать. Короче говоря, каждая пара фигурных скобок {определяет }новую область видимости, и все, что не находится внутри фигурных скобок, находится на верхнем уровне. Что-то, объявленное в определенной области, может использоваться где угодно внутри этой области, включая внутренние области, но не за ее пределами. Область видимости становится важной особенностью при работе с функциями, поэтому мы сделали небольшое отступление, чтобы рассмотреть ее сейчас.

Эти правила, касающиеся области действия, помогают поддерживать порядок в коде, поскольку предотвращают случайное использование элементов за пределами того места, где они предназначены. Это также обычно избавляет от необходимости беспокоиться о том, используется ли имя где-то еще в большой программе, поскольку вам нужно только думать о том, что входит в область действия конкретного фрагмента кода.

Можно использовать функции и переменные верхнего уровня в других файлах сценариев, если они явно помещены в глобальную область видимости или экспортированы и импортированы, но это более сложные темы, которые мы рассмотрим позже в этом руководстве.

## Возвращаемые значения

Другая полезная часть функций заключается в том, что они могут возвращать значение с помощью оператора return, как в следующем примере.

function add(a, b)

{

return a + b;

}

console.log(add(4, 5)); // 9

Функция add добавляет два своих параметра, а затем возвращает их. Это становится значением, возвращаемым вызовом функции по адресу add(4, 5), поэтому число 9 записывается в консоль.

Функции могут возвращать значения любого типа, например числа, строки или логические значения. Поскольку JavaScript является динамически типизированным, функции могут даже возвращать разные типы в разных обстоятельствах, например, строку в операторе if и число в операторе else, но обычно рекомендуется убедиться, что функции всегда возвращают один и тот же тип. убедитесь, что они работают предсказуемо.

Аналогично breakциклам in или операторам переключения, returnтакже происходит досрочный выход из функции, поэтому никакой код после него в функции выполняться не будет.

function add(a, b)

{

return a + b;

// no code here will run, as 'return'

// also exits the function

}

Вы также можете условно выйти из функции раньше, например, вернувшись в операторе if.

function divide(a, b)

{

if (b === 0)

{

// Log a warning message

console.log("Dividing by zero!");

// Return 0 instead of Infinity or NaN

return 0;

}

// This only runs if it did not already return

return a / b;

}

console.log(divide(1, 0));

Функции по-прежнему могут использоваться returnдля досрочного выхода, даже если им не нужно возвращать значение - в этом случае return;само по себе без значения просто выйдет из функции. (Если что-то все равно попытается использовать возвращаемое значение, оно получит undefined.)

Если вы использовали функциональные блоки, которые возвращают значения в таблицах событий Construct, обратите внимание, что действие Установить возвращаемое значение не завершает функцию раньше, как это происходит в JavaScript.

### Не добавляйте новую строку после возврата

При использовании return. Еще во второй части мы отмечали, что часто вы можете опускать точки с запятой, и JavaScript сам догадается, куда они должны идти. В части 4 мы также рассказали, что JavaScript обычно не заботится о том, где идут пробелы или разрывы строк. Однако если после , следует разрыв строки return, JavaScript всегда предполагает, что точка с запятой стоит сразу после return. Часто это ошибочное предположение, и его обычно рассматривают как ошибку проектирования, но изменить его уже слишком поздно, и нам просто приходится с этим жить.

Следующий фрагмент кода демонстрирует код, который **не будет работать должным образом** .

function add(firstValue, secondValue, thirdValue)

{

// OOPS: line break after 'return'

return

firstValue + secondValue + thirdValue;

}

В этом случае JavaScript неправильно предполагает, что после return. Тогда после выполнения оператора возврата ничего не выполняется. Итак, это эквивалентно написанию этого:

function add(firstValue, secondValue, thirdValue)

{

return;

}

Функция, которая не возвращает возвращаемое значение, возвращает undefined. Возможно, вы думали, что функция вернет число, содержащее сумму трех значений, но вместо этого она возвращает undefined, просто потому, что после return. Это одна из причин, по которой автоматическая вставка точки с запятой считается ошибкой.

Итак, урок здесь заключается в том, чтобы никогда не делать переноса строки после returnоператора. Вместо этого используйте код, подобный следующему.

function add(firstValue, secondValue, thirdValue)

{

// OK: no line break after 'return'

return firstValue + secondValue + thirdValue;

}

## Некоторые примеры

Поздравляем! Теперь вы достаточно изучили JavaScript, чтобы мы могли писать несколько полезных мини-программ. Чтобы примеры были простыми, давайте напишем код, который сможет выводить определенные числа.

### Перечислите четные числа

Вот код, который выводит на консоль все четные числа до 50.

function isEven(n)

{

return n % 2 === 0;

}

for (let i = 1; i < 50; i++)

{

if (isEven(i))

{

console.log(`Number ${i} is even`);

}

}

Функция isEven возвращает логическое значение (истина или ложь), указывающее, является ли параметр nчетным числом. Он осуществляет эту проверку с помощью оператора остатка %: если число, разделенное на 2, имеет нулевой остаток, то оно кратно двум и, следовательно, четное число.

Затем эта функция вызывается для каждого числа от 1 до 50 с использованием цикла for. Обратите внимание, что он вызывается внутри оператора «if», поэтому, если функция возвращает true, код внутри оператора «if» будет запущен, и этот код записывает число на консоль, указывая, что это четное число.

### Перечислить простые числа

Простое [число](https://www-construct-net.translate.goog/out?u=https://en.wikipedia.org/wiki/Prime_number&_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru) — это число, делителями которого являются только оно само и 1, или, другими словами, не существует двух целых чисел, которые можно умножить, чтобы получить это число, кроме него самого и 1.

Простой способ проверить, nявляется ли число простым, — это проверить, кратно ли оно любому числу от 2 до квадратного корня из n. Это также можно сделать с помощью цикла for. Вот код, который будет регистрировать каждое простое число от 1 до 100.

function isPrime(n)

{

// 1 is not a prime number

if (n === 1)

return false;

// Raise to power of 1/2 takes square root

let sqrtN = n \*\* 0.5;

// Check for factors from 2 to square root of n

for (let f = 2; f <= sqrtN; f++)

{

if (n % f === 0)

{

// Found a factor: not a prime

return false;

}

}

// Did not find any factors: is a prime

return true;

}

for (let i = 1; i < 100; i++)

{

if (isPrime(i))

{

console.log(`Number ${i} is prime`);

}

}

Это объединяет в себе ряд вещей, которые мы узнали до сих пор. Вот несколько замечаний о том, как работает функция isPrime :

* 1 не является простым числом, но в противном случае функция вернула бы для него значение true, поэтому нам придется сделать для него исключение. Таким образом, функция начинается с проверки того, равно ли число 1, и завершает работу досрочно, возвращая false, если это так.
* При возведении в степень половинной вычисляется квадратный корень числа. Можно также использовать Math.sqrt(n)вместо n \*\* 0.5(но в этом руководстве еще не рассматриваются ни одна из встроенных математических функций JavaScript).
* Затем цикл for проверяет каждое число от 2 до квадратного корня числа включительно. Обратите внимание, что цикл начинает переменную fсо значения 2 и увеличивает ее до тех пор, пока она меньше или равна sqrtN, что гарантирует, что он также проверяет, sqrtNявляется ли она множителем, когда это целое число.
* Подобно проверке в isEven , мы можем проверить, fявляется ли множителем, nиспользуя оператор остатка и проверяя, равен ли остаток нулю. В этом случае nкратно f, и, следовательно, nне является простым числом, поэтому функция возвращает false.
* Если весь цикл for выполняется без возврата, это означает, что он не нашел никаких факторов n. Следовательно, nявляется простым, поэтому функция возвращает true.
* Затем, как и в предыдущем примере, цикл for используется для проверки каждого числа от 1 до 100, чтобы определить, является ли оно простым числом, и выводит сообщение на консоль, если это так.

Если вы хотите еще немного попрактиковаться в том, что вы уже узнали, попробуйте написать еще несколько функций для проверки различных типов чисел, например квадратных чисел, и перечислите все числа от 1 до 100, которые соответствуют тесту. Совет: вы можете проверить, является ли число целым, используя n % 1 === 0, поскольку в JavaScript оператор остатка работает с дробями, поэтому, если остаток, разделенный на 1, равен 0, то число не имеет дробной части и поэтому является целым числом. Затем вы можете проверить такие вещи, как, например, является ли квадратный корень числа целым числом, что скажет вам, является ли число квадратным.

## Заключение

В этой части мы рассмотрели:

* Использование файлов сценариев в Construct
* Объявление функций
* Вызов функций
* Использование параметров функции
* Область применения переменных, параметров и функций
* Возврат значений из функций
* Почему следует избегать разрыва строки послеreturn
* Некоторые примеры объединения всего рассмотренного до сих пор для перечисления чисел с определенными математическими свойствами.

Функции в JavaScript включают в себя гораздо больше функций, поэтому в следующей части этого руководства будут рассмотрены еще некоторые аспекты функций.

# Изучите JavaScript в Construct, часть 6: больше о функциях

Это шестая часть серии руководств [«Изучите JavaScript в Construct»](https://www-construct-net.translate.goog/en/courses/learn-javascript-construct-79?_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru) . Эта часть является продолжением части 5. Если вы ее пропустили, см. [«Изучаем JavaScript в Construct», часть 5: Функции](https://www-construct-net.translate.goog/en/tutorials/learn-javascript-construct-2813?_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru) .

В этой части мы продолжим использовать файл сценария в новом проекте Construct. Вы можете продолжить работу с тем же проектом, что и раньше, или создать новый проект и снова добавить новый файл сценария; в любом случае удалите весь код в файле сценария, чтобы начать с пустого файла. В этой части мы расскажем больше о функциях. Поскольку функции являются важной частью программирования, стоит более подробно остановиться на том, как они работают и что с ними можно делать.

## Функциональные выражения

Функцию также можно использовать как выражение . Другими словами, это означает, что вы можете поместить объявления функций в то же место, что и число или строку, например, присвоив их переменной, как показано ниже.

// Assign a function to a variable

let myFunc = function logMessage()

{

console.log("Hello world!");

};

// Call the function held in the variable

myFunc();

При таком использовании функции обратите внимание, что она не объявляет функцию с именем logMessage, поэтому вызов logMessage()не будет работать. Единственный способ вызвать функцию — из переменной, которой она была присвоена, myFunc()т.е. Другими словами, имя функции здесь фактически не используется. Фактически, в этом случае имя функции не является обязательным и поэтому может быть опущено.

// Function has name omitted

let myFunc = function()

{

console.log("Hello world!");

};

myFunc();

Функции имеют тип "function"при использовании typeof.

let myFunc = function()

{

console.log("Hello world!");

};

console.log(typeof myFunc); // function

Помните, что JavaScript является динамически типизированным , то есть переменные могут иметь числа, строки или функции, назначенные в любое время, или не иметь ничего назначенного (неопределенного). Если вы попытаетесь вызвать переменную так, как будто в ней есть функция, но это не так, это вызовет ошибку при попытке вызвать функцию.

let myFunc = 5; // not a function

myFunc(); // try to call it anyway

// TypeError: myFunc is not a function

Зачем использовать функции в переменных? Это обеспечивает гибкость в том, что вы делаете с функциями, например, имея переменную, которой переназначаются различные функции в зависимости от того, что вы хотите сделать. Это также демонстрирует принцип функционального выражения, который, как мы вскоре увидим, может оказаться полезным и в других случаях.

## Обратные вызовы

Точно так же, как функции можно присваивать переменным, функции также можно передавать в качестве параметров другим функциям.

Поначалу это может сбить с толку, но это важная часть программирования. В частности, он позволяет использовать шаблон с именем callbacks , где функция может выполнить некоторую работу, а затем вызвать другую функцию с результатом работы.

Чтобы представить идею, приведем пример функции, вызывающей другую функцию.

// Declare a function that calls the function passed as its parameter

function logReturnValueOf(func)

{

console.log(`The function provided returned: ${func()}`);

}

// Call the above function, and also give it a function to call

logReturnValueOf(function ()

{

return "Hello world!";

});

На консоль выводится сообщение: Предоставленная функция вернула: Привет, мир!

Обратите внимание, что у нас есть вызов функции logReturnValueOf(...), а параметр между (и )является функциональным выражением для функции, которая просто возвращает строку «Hello world!». Затем внутри logReturnValueOfфункции она вызывает свой параметр с помощью func()и добавляет возвращаемое значение в сообщение журнала.

Этот код достаточно прост, и его можно переписать, чтобы просто передавать строку непосредственно в logReturnValueOf. Однако мы используем простые примеры, чтобы продемонстрировать принцип, поскольку эти методы полезны на практике.

Чтобы привыкнуть к синтаксису с таким использованием функций, может потребоваться некоторое время. В частности, вы можете получить множество пар ( ), {находящихся }в разных строках. Однако вы должны заметить, что редактор кода Construct, как и большинство других редакторов кода, может выделять пары скобок, чтобы помочь вам сопоставить их.

## Возвращающие функции

Функции также могут возвращать другие функции.

Возможно, вы начинаете замечать, что везде, где вы можете использовать строку или число, вы также можете использовать функцию. Это полезная функция JavaScript, которую иногда называют функциями первого класса , поскольку с функциями можно обращаться так же, как с любым другим типом данных.

Вот пример функции, возвращающей функцию.

// Declare a function that returns another function

function getFunction()

{

return function()

{

console.log("Hello world!");

};

}

// Call getFunction(), which returns a function

let returnedValue = getFunction();

// Call the returned function, logging "Hello world!"

returnedValue();

## Рекурсия

Функции также могут вызывать сами себя. Это называется рекурсией . Чтобы функция могла вызывать сама себя, она должна иметь имя, по которому можно ссылаться на себя.

Обратите внимание: если функция всегда вызывает саму себя, это имеет эффект, аналогичный бесконечному циклу: процесс никогда не будет выполняться, и программа зависнет или выйдет из строя. Поэтому важно убедиться, что рекурсивная функция в какой-то момент завершается.

Чтобы проиллюстрировать рекурсию, давайте напишем функцию для вычисления [факториала](https://www-construct-net.translate.goog/out?u=https://en.wikipedia.org/wiki/Factorial&_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru) числа. Например, факториал 5, который также пишется как 5!, равен 5 х 4 х 3 х 2 х 1 = 120.

Это также можно сделать с помощью цикла. Однако мы будем использовать рекурсивную функцию, чтобы продемонстрировать принцип рекурсии.

function factorial(n)

{

if (n === 1)

{

return 1;

}

else

{

// Recursion happens here

return n \* factorial(n - 1);

}

}

console.log(`5! = ${factorial(5)}`);

Это зарегистрирует 5! = 120 на консоль.

Ключевой частью, вызывающей рекурсию, является n \* factorial(n - 1), что означает, factorialчто функция снова вызывает себя, но с другим параметром. Это означает, что при передаче с 5 он вычислит 5 \* factorial(4); вызов factorial(4)вычислит 4 \* factorial(3), и так далее, пока nне станет 1.

Обратите внимание, что существует оператор if, который специально проверяет if (n === 1)и возвращает 1 (поскольку 1! = 1). Обратите внимание, что в этом случае он не вызывает себя снова. Это важно, чтобы убедиться, что рекурсия завершится! В противном случае это продолжалось бы вечно, сводясь nк отрицательным числам, даже не останавливаясь.

Вот эквивалентная factorialфункция, написанная с использованием цикла for. Обратите внимание, что в этом случае цикл for использует параметр в качестве переменной цикла (без объявления собственной переменной) и уменьшает его до тех пор, пока nон не станет равным 1 (в этот момент n > 1значение false). Это означает, что цикл будет повторяться с nобратным отсчетом, например, 5, 4, 3, 2, каждый раз умножая product. Он пропускает умножение на 1, но это не имеет значения, поскольку умножение на 1 не меняет число.

function factorial(n)

{

let product = 1;

for ( ; n > 1; n--)

{

product \*= n;

}

return product;

}

console.log(`5! = ${factorial(5)}`);

Как уже отмечалось, это также вполне разумный способ реализации функции, но мы использовали его как способ продемонстрировать принцип рекурсии с функциями. Бывают случаи, когда рекурсия удобнее циклов, например при обработке определенных структур данных.

## Стрелочные функции

JavaScript предоставляет сокращенный синтаксис для функций. Это основано на символах равенства и больше: =>. В данном контексте это не имеет ничего общего со сравнениями: два символа должны выглядеть как маленькая стрелка, отсюда и название стрелочных функций . Когда вы видите эту стрелку, вы знаете, что смотрите на функцию. (Постарайтесь не путать его с оператором сравнения «меньше или равно» <=— стрелка стрелочных функций указывает вправо .)

Стрелочные функции не могут иметь имени и поэтому всегда работают как функциональное выражение. Чтобы продемонстрировать синтаксис, приведем стандартное функциональное выражение для сложения двух чисел.

function (a, b)

{

return a + b;

}

Это можно записать в виде стрелочной функции следующим образом:

(a, b) =>

{

return a + b;

}

Обратите внимание, что ключевое слово не используется: вместо этого после параметров идет function«стрелка» .=>

Синтаксис стрелок можно еще больше сократить. Если returnвнутри тела функции есть только один оператор, фигурные скобки и returnключевое слово можно опустить. Обратите внимание, что их нельзя пропустить при использовании стандартного синтаксиса функции.

(a, b) => a + b

Обратите внимание, насколько это короче того, с чего мы начали, но все три приведенных выше примера работают одинаково. Это просто разные способы написать одно и то же.

Есть еще две вещи, которые следует отметить относительно стрелочных функций. Во-первых, если имеется только один параметр, скобки вокруг параметров необязательны. Например, стрелочную функцию для добавления 1 к числу можно записать одним из двух способов, описанных ниже.

(a) => a + 1

a => a + 1

// Equivalent to:

function (a)

{

return a + 1;

}

Если функция не имеет параметров, она может использовать пустой список параметров с ().

() => "Hello world!"

// Equivalent to:

function ()

{

return "Hello world!";

}

Один из распространенных методов, который вы можете увидеть, — это использовать один неиспользуемый параметр вместо (). Часто неиспользуемый параметр называется просто \_для краткости, поскольку единственное подчеркивание является допустимым именем параметра в JavaScript. Например, обе следующие стрелочные функции не используют никаких параметров, а объявляют один неиспользуемый параметр просто для сокращения.

unusedParam => "Hello world!"

\_ => "Hello world!"

Поэтому, если вы видите \_ =>в чужом коде, помните, что это не какой-то особый синтаксис, это то же самое, что сказать: « unusedParam =>Пытаться сделать код как можно короче».

Чтобы проиллюстрировать, почему синтаксис стрелочной функции полезен, давайте вернемся к logReturnValueOfпредыдущему примеру. Первоначально это выглядело так.

// Declare a function that calls the function passed as its parameter

function logReturnValueOf(func)

{

console.log(`The function provided returned: ${func()}`);

}

// Call the above function, and also give it a function to call

logReturnValueOf(function ()

{

return "Hello world!";

});

Переданная функция logReturnValueOf(...)не принимает параметров и просто возвращает значение. Поэтому его можно переписать вот так.

// Declare a function that calls the function passed as its parameter

function logReturnValueOf(func)

{

console.log(`The function provided returned: ${func()}`);

}

// Call the above function, and also give it a function to call

logReturnValueOf(() => "Hello world!");

Это делает вызов logReturnValueOfнамного короче и понятнее.

Стрелочные функции имеют ряд других особенностей, в частности, thisпо-другому обрабатывают ключевое слово. Однако мы пока не собираемся их рассматривать. А пока просто имейте в виду, что если вы видите =>, это просто еще один способ написать функцию. В этом руководстве иногда будут использоваться стрелочные функции, где более короткое написание функции делает код более понятным.

## Коротко о замыканиях

Замыкания — довольно сложная тема, но здесь мы рассмотрим ее лишь вкратце. Короче говоря, функции могут быть объединены с вещами, выходящими за рамки их собственной области действия.

Хороший способ продемонстрировать это — функции, возвращающие функции. Вот функция, которая возвращает другую функцию, которая возвращает messageпеременную из внешней функции .

function getFunction()

{

let message = "Hello world!";

return function()

{

return message;

}

}

// Get a function returning 'message'

let returnedFunc = getFunction();

// Call it and log the result to the console

console.log(returnedFunc());

При этом по-прежнему регистрируется сообщение «Привет, мир!» на консоль. Однако обратите внимание, что возвращаемая функция ссылается на messageпеременную внутри getFunction. Обычно messageпеременная существует только внутри тела getFunctionи перестает существовать после getFunctionвозврата. Однако, поскольку другая функция ссылается на message, ее время жизни увеличивается: она будет существовать до тех пор, пока существует возвращаемая функция, поскольку она фактически связана с этой функцией. Вот что делают замыкания: они, по сути, связывают функции с вещами, находящимися за пределами их области действия.

Вы можете задаться вопросом, откуда JavaScript сохраняет messageпеременную в памяти. Ключевым моментом здесь является то, что JavaScript — это язык со сборкой мусора , который автоматически управляет такими деталями, поэтому вам не нужно о них беспокоиться. В приведенном выше примере JavaScript знает, что возвращаемая функция ссылается на messageпеременную, и пока возвращаемая функция существует, переменная также будет храниться в памяти. Когда на возвращаемую функцию больше не ссылаются, сборщик мусора определит, что ее можно освободить из памяти вместе с переменной, на которую она ссылается.

Замыкания также могут включать параметры функции. В приведенном ниже примере показано использование функции для создания другой функции, добавляющей число.

function makeAddFunction(addNumber)

{

return function(n)

{

return n + addNumber;

}

}

let add2 = makeAddFunction(2);

let add10 = makeAddFunction(10);

console.log(add2(5)); // 7

console.log(add10(5)); // 15

makeAddFunctionвозвращает функцию, которая принимает параметр nи добавляет addNumberк нему значение. Итак, makeAddFunction(2)возвращает функцию, которая принимает один параметр и добавляет к нему 2, названный add2в приведенном выше примере. Эта функция запоминает addNumberизначально переданный параметр аналогично тому, как это messageзапоминалось ранее. Он также запоминает исходное переданное значение отдельно для каждой возвращаемой функции, поэтому функции add2и add10добавляют разные числа.

Вот еще один пример, демонстрирующий, что замыкания также являются постоянными. Он использует функцию, которая возвращает другую функцию, увеличивающую переменную counter.

function makeCounterFunction()

{

let counter = 0;

return function()

{

return counter++;

};

}

let counterFuncA = makeCounterFunction();

let counterFuncB = makeCounterFunction();

console.log(counterFuncA()); // 0

console.log(counterFuncA()); // 1

console.log(counterFuncA()); // 2

console.log(counterFuncB()); // 0

Это работает аналогично предыдущему примеру, где messageбыло запомнено. Однако этот пример показывает, что функции также могут изменять запоминаемую переменную. При counterFuncAвызове трижды значение counterзапомненной переменной увеличивается, заставляя ее возвращать увеличивающееся значение.

Запоминаемая переменная также уникальна для каждой возвращаемой функции. Обратите внимание, что вызов counterFuncBснова возвращает 0. Это потому, что у него есть собственная копия переменной counter, которая увеличивается независимо. Другими словами, каждая возвращаемая функция makeCounterFunctionзапоминает свою отдельную копию переменной counterи может изменять ее независимо.

Также стоит отметить, что внешний код makeCounterFunctionне имеет возможности изменить counterпеременную, поскольку он находится за пределами области действия переменной, поэтому вы не можете написать код, который ссылается на нее там. Однако возвращаемые функции связаны с переменной и могут по-прежнему использовать ее в телах своих функций. Это обеспечивает форму инкапсуляции : counterпеременная является частной для возвращаемой функции, поскольку ее невозможно изменить каким-либо иным способом, кроме вызова возвращаемой функции. Например, невозможно написать код, который сбрасывается counterобратно в 0: единственный способ поддержать это — makeCounterFunctionвернуть что-то другое, что специально позволяет это сделать.

Полные технические подробности о том, как работают замыкания, — это довольно сложная тема, поэтому мы не будем здесь больше рассказывать о замыканиях. Однако это важная особенность функций JavaScript, о которой полезно знать. Подводя итог, можно сказать, что функции JavaScript могут удерживать объекты, находящиеся за пределами их области действия, и продолжать использовать их даже после того, как в противном случае они перестали бы существовать. Попробуйте поэкспериментировать с другими примерами, чтобы улучшить свое понимание. Если это кажется сложным, не волнуйтесь слишком сильно — в остальной части руководства об этом больше не упоминается, и вы всегда можете вернуться к нему позже. Цель этого раздела — просто познакомить вас с этой концепцией, чтобы вы хотя бы знали, что это возможно.

## Заключение

В этой части мы рассказали больше о функциях:

* Выражения функций и способы использования функций, как и любой другой тип данных, например числа и строки.
* Передача функций в качестве параметров другим функциям (так называемым «обратным вызовам»)
* Возврат функций из функций
* Рекурсия
* Стрелочные функции как более короткий способ написания функциональных выражений
* Введение в замыкания

Функции в JavaScript — это еще не все — мы могли бы написать целое руководство только по функциям! Однако мы рассмотрели большинство ключевых особенностей функций, которые важны для новичков, и поэтому пришло время перейти к следующей теме. В следующей части этого руководства мы перейдем к объектам .

# Изучите JavaScript в Construct, часть 7: Объекты

Это седьмая часть серии руководств [«Изучите JavaScript в Construct»](https://www-construct-net.translate.goog/en/courses/learn-javascript-construct-79?_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru) . Эта часть является продолжением части 6. Если вы ее пропустили, см. [«Изучаем JavaScript в Construct», часть 6: Больше о функциях](https://www-construct-net.translate.goog/en/tutorials/learn-javascript-construct-2815?_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru) .

В этой части мы продолжим использовать файл сценария в новом проекте Construct. Вы можете продолжить работу с тем же проектом, что и раньше, или создать новый проект и снова добавить новый файл сценария; в любом случае удалите весь код в файле сценария, чтобы начать с пустого файла. В этой части мы рассмотрим еще один ключевой строительный блок программирования на JavaScript: объекты.

## Объекты

Объект по сути представляет собой группу переменных, но в контексте объектов мы используем свойство name вместо переменной . Объекты можно создавать с помощью фигурных скобок {и }. Каждое свойство объекта может быть объявлено с помощью property: valueсинтаксиса, разделенного запятыми. Ниже показан пример синтаксиса объекта с двумя свойствами.

{

firstProp: "Hello world!",

secondProp: 123

}

Свойства аналогичны переменным в том, что у них есть имя и им может быть присвоено значение любого типа. Однако они не объявлены с помощью let, и для разделения имени и значения используется двоеточие. Свойства также не имеют области действия, как переменные: свойства являются частью объекта и поэтому существуют до тех пор, пока существует объект.

Объекты — это еще один тип данных в JavaScript. Это означает, что вы также можете присваивать объекты переменным, как показано ниже. Фактически, как и в случае с функциями, вы можете использовать объекты везде, где можно использовать любое другое значение, например строку или число. Объект typeofявляется "object".

let myObject = {

firstProp: "Hello world!",

secondProp: 123

};

console.log(typeof myObject); // object

Обратите внимание, что ранее мы использовали фигурные скобки {и }для обхода групп операторов для таких вещей, как операторы if и циклы for. Однако здесь они имеют другое значение: вместо этого они создают объект. JavaScript может отличить контекст от контекста.

Если объект присвоен переменной, к каждому из его свойств можно получить доступ через точку и имя свойства, например myObject.firstProp.

let myObject = {

firstProp: "Hello world!",

secondProp: 123

};

console.log(myObject.firstProp); // Hello world!

То же самое происходит и в console.logкоде, который мы использовали все время: это означает доступ к logсвойству объекта с именем console. ( logСвойство также является функцией, а функции объектов будут рассмотрены позже.)

JavaScript позволяет вам читать несуществующие свойства. Это просто возвращает undefined, аналогично чтению переменной, которой никогда ничего не было присвоено.

let myObject = {

firstProp: "Hello world!",

secondProp: 123

};

console.log(myObject.thirdProp); // undefined

### Добавление и удаление свойств

Будучи высокодинамичным языком, JavaScript также позволяет добавлять и удалять свойства объектов в любое время. Присвоение несуществующему свойству приведет к его созданию. Пример ниже демонстрирует это, а также регистрирует весь объект в консоли браузера, что является удобным способом проверить, какие свойства и значения имеет объект. Обратите внимание, что вы можете просто увидеть объект в консоли, но рядом с ним должна быть кнопка, позволяющая развернуть его и просмотреть содержимое объекта.

let myObject = {

firstProp: "Hello world!",

secondProp: 123

};

// Adds thirdProp to the object

myObject.thirdProp = 456;

// Log entire object to console

console.log(myObject);

Развернув объект в консоли, вы должны увидеть, что он отображает три свойства объекта, включая thirdProp: 456те, которые были добавлены к объекту.

Вы также можете использовать пустой объект без свойств, то есть просто {}. Это полезно, если вам нужно добавить свойства условно, например, добавив свойство только в операторе if.

// Create an empty object

let myObject = {};

// Add a property to it (which could be done conditionally)

myObject.firstProp = "Hello world!";

console.log(myObject);

Свойства можно удалить с помощью специального deleteключевого слова.

let myObject = {

firstProp: "Hello world!",

secondProp: 123

};

// Delete secondProp from the object

delete myObject.secondProp;

// Log entire object to console

console.log(myObject);

В приведенном выше примере объект будет зарегистрирован только с помощью firstProp, поскольку другое свойство было удалено.

## Свойства строки

Свойства объекта также могут использовать строку в качестве имени. Это невозможно с переменными, поэтому есть специальные возможности для свойств. Вот пример использования строки в качестве имени свойства.

let myObject = {

// Use a string for a property name

"myProperty": 5

};

// This is equivalent to:

let myObject = {

myProperty: 5

};

Обычно имена свойств имеют те же правила именования, что и переменные: никаких пробелов, специальных символов и т. д. Однако использование строки позволяет использовать любое имя. Например, вы не можете использовать пробел или восклицательный знак в имени свойства, если в качестве имени свойства не используется строка.

// String properties allow spaces and special characters

let myObject = {

"my property!": 5

};

// Normal properties don't allow this

myObject = {

// SyntaxError

my property!: 5

};

Вам также необходимо использовать строку при назначении или получении свойства с таким именем, поскольку обычный точечный синтаксис также нельзя использовать по той же причине — он не будет допустимым синтаксисом для кода JavaScript. Для доступа к свойству строки вместо точки используются квадратные скобки [и со строкой между ними.]

let myObject = {

"my property!": 5

};

// Assign to a string property

myObject["my property!"] = 7;

// Read a string property

console.log(myObject["my property!"]); // 7

Как и раньше, вы также можете добавлять и удалять свойства, используя этот синтаксис — это просто еще один способ записи доступа к свойству, который в принципе работает так же, как и использование точки.

Если свойство имеет допустимое имя, вы можете получить к нему доступ обоими способами: с помощью точки или строки.

let myObject = {

myProperty: 5

};

// Read with dot property

console.log(myObject.myProperty); // 5

// Read with string property

console.log(myObject["myProperty"]); // 5

Одной из причин использования строковых свойств является то, что вы можете использовать строковую переменную или, по сути, любое выражение, возвращающее строку, для доступа к имени свойства. Это невозможно при использовании точки для доступа к свойствам, поскольку вы можете использовать только фиксированное имя.

let myObject = {

myProperty: 5

};

// Access a property via a string variable

let propName = "myProperty";

console.log(myObject[propName]); // 5

Это может быть полезно для изменения свойств, к которым осуществляется доступ, в зависимости от условий. Например, у вас может быть оператор «if», который изменяет имя свойства, к которому осуществляется доступ.

### Тестирование свойств

Ранее мы видели, как возвращается доступ к отсутствующему свойству undefined. Однако также возможно хранить undefinedна территории. Это означает, что вы не можете использовать его, undefinedчтобы определить, существует ли свойство объекта или нет, поскольку вы не можете отличить «отсутствует» от «существует, но имеет неопределенное значение».

let myObject = {

myProperty: undefined

};

// Reading a missing property returns undefined

console.log(myObject.otherProperty); // undefined

// But reading myProperty also returns undefined

console.log(myObject.myProperty); // undefined

Итак, чтобы определить, действительно ли свойство существует у объекта, JavaScript предоставляет inоператор. Однако из-за используемого синтаксиса имя свойства должно быть строкой. Таким образом, using "propertyName" in myObjectвернет логическое значение true, если имя свойства существует в данном объекте или falseне существует. Помните, что, поскольку строковые и обычные имена взаимозаменяемы, это по-прежнему работает и для обычных свойств.

let myObject = {

myProperty: undefined

};

console.log("otherProperty" in myObject); // false

console.log("myProperty" in myObject); // true

## Вложенные объекты

Помните, что объекты можно использовать везде, где могут использоваться другие значения, такие как числа и строки. Это также относится к свойствам объекта, которые также могут иметь значение, являющееся другим объектом. Затем к этим свойствам можно получить доступ с помощью другой точки или цепочки квадратных скобок при использовании строкового синтаксиса.

let person = {

// The "name" property is another object

name: {

first: "Joe",

last: "Bloggs"

},

age: 30

};

// Access the first name with dots

console.log(person.name.first); // Joe

// Access the first name with strings

console.log(person["name"]["first"]); // Joe

Это позволяет нам создавать более сложные структуры данных. Все может быть организовано по свойствам и подсвойствам, вместо того, чтобы иметь множество свойств в одном объекте.

## нулевой

Ранее мы рассмотрели undefinedспециальное значение, которое JavaScript предоставляет, когда что-то пусто или отсутствует. Несколько сбивает с толку то, что JavaScript предоставляет еще одно специальное пустое значение: null.

Почему существуют два специальных пустых значения? Обычно это считается ошибкой дизайна в JavaScript. Большинство других языков программирования имеют только одно специальное пустое значение, обычно также называемое nullили что-то подобное, например nil. Хорошим соглашением является всегда использовать его nullвсякий раз, когда вы, как программист, намеренно хотите иметь пустое или неопределенное значение, и использовать его только undefinedтам, где это значение дает вам язык JavaScript. Другими словами, вы можете думать о undefinedчем-то совершенно отсутствующем, например о несуществующем свойстве, и nullкак о чем-то, что существует, но содержит пустое значение, например о свойстве, которое существует, но которому присвоено пустое значение.

let myObject = {

// Prefer to use null for empty properties

myProperty: null

};

Like undefinedтакже nullявляется ложным , то есть в операторе if или при преобразовании в логическое значение считается как false. Язык JavaScript называет оба значенияundefined и nullнулевыми значениями, поскольку они оба являются пустыми значениями.

Попытка получить доступ к свойству чего-то, имеющего нулевое значение, приведет к ошибке.

let obj = null;

// TypeError: Cannot read properties of null (reading 'name')

console.log(obj.name);

Обратите внимание, что это также может произойти через вложенные ссылки, как показано ниже.

let myObject = {

myProperty: null

};

// TypeError: Cannot read properties of null (reading 'anotherProperty')

console.log(myObject.myProperty.anotherProperty);

Поскольку myObject.myPropertyесть null, следующая попытка доступа к свойству не удалась — в данном случае при попытке доступа anotherProperty.

Будьте осторожны с одной запутанной ловушкой null: typeof undefinedесть "undefined", но typeof nullесть "object". Широко распространено мнение, что это ошибка проектирования, nullимеющая тот же тип, что и объект. К сожалению, это слишком сложно изменить по соображениям совместимости. Вам просто придется следить за этим.

## Дополнительное связывание

Вместо точек вы можете использовать необязательный оператор цепочки?. — это позволяет вам считать свойство чем-то нулевым, и это не вызовет ошибки. Это просто даст вам undefinedв конце. Конечно, если свойство существует, оно все равно возвращает свое значение. Например, следующий код регистрирует, undefinedа не вызывает ошибку.

let myObject = {

myProperty: null

};

console.log(myObject?.myProperty?.anotherProperty); // undefined

Это работает с целыми цепочками нулевых значений, отсюда и «цепочка» в названии. Следующий код по-прежнему регистрируется, undefinedа не вызывает ошибку, хотя объекта вообще нет и ни одно из свойств не существует.

let obj = null;

console.log(obj?.firstProperty?.secondProperty?.thirdProperty);

Чтобы продемонстрировать, насколько это полезный ярлык, что, если мы захотим запустить приведенный выше код без использования дополнительной цепочки и без возникновения ошибки? Нам пришлось бы использовать большой оператор if, который по пути проверяет каждое существующее свойство, а это требует гораздо больше кода для написания.

let result;

let obj = null;

// Check the object and every single expected property exists

if (obj &&

obj.firstProperty &&

obj.firstProperty.secondProperty &&

obj.firstProperty.secondProperty.thirdProperty)

{

// All properties exist: read the result

result = obj.firstProperty.secondProperty.thirdProperty;

}

console.log(result);

Необязательный синтаксис цепочки также работает со строковыми свойствами, хотя он использует немного странный ?.["property"]синтаксис.

let myObject = {

myProperty: null

};

console.log(myObject?.["myProperty"]?.["anotherProperty"]);

Наконец, это также распространяется на вызов функций с помощью func?.(). Это можно использовать для вызова функции для чего-то нулевого, и вместо того, чтобы вызывать ошибку при попытке вызова функции, она просто вернет undefined.

let someFunction = null;

// Not an error even though someFunction is null,

// because optional ?.() syntax used

console.log(someFunction?.());

Это полезные способы попытаться получить доступ к свойствам, если вы не возражаете, если какое-либо из них существует, и вместо этого можете принять нулевое значение. В общем, вам следует знать, ожидает ли ваша программа существования свойств или нет, и использовать стандартный синтаксис, если они должны существовать. Так что не используйте его ?.повсюду — это не сделает ваш код лучше. Просто используйте его в конкретных случаях, когда отсутствие свойств не является проблемой, а удобство полезно.

## Рекомендации

Объекты можно передавать функциям так же, как строки и числа. В примере ниже показано два способа, которыми это можно сделать.

function logFood(obj)

{

// Log the "food" property of the object

console.log(obj.food);

}

// An object can be passed via a variable storing an object.

let myObject = {

food: "banana"

};

logFood(myObject); // logs "banana"

// An object can also be passed directly. This logs "pizza".

logFood({

food: "pizza"

});

Однако если функция каким-либо образом изменяет объект, она работает иначе, чем передача строки или числа. Это связано с ключевым различием между объектами и строками/числами: объекты передаются по ссылке , поэтому изменения влияют на оригинал. С другой стороны, строки и числа копируются (это также называется передачей по значению ), поэтому изменения не влияют на оригинал.

Простой способ продемонстрировать это — использовать переменные. Обычно, если у вас есть строковая переменная, присвоение ее другой переменной создает копию, как показано в примере ниже.

// Create a variable with a string.

let food1 = "pizza";

// Create another variable and assign it the same string.

// This creates a second copy of the string for this variable.

let food2 = food1;

// Change the food2 variable.

food2 = "noodles";

// Log both variables. They are different.

console.log(`food1 is: ${food1}`);

console.log(`food2 is: ${food2}`);

Однако если у вас есть две переменные, назначенные одному и тому же объекту, они обе ссылаются на один и тот же объект. Таким образом, если свойство объекта изменяется, оно возвращает одно и то же значение через обе переменные, поскольку они обе ссылаются на один и тот же объект.

// Create a variable with an object.

let ref1 = {

food: "pizza"

};

// Create a variable also assigned to the same object.

let ref2 = ref1;

// Change the 'food' property through the second variable.

ref2.food = "noodles";

// Log the 'food' property of both variables. They are the same,

// as they both reference the same object.

console.log(`ref1.food is: ${ref1.food}`);

console.log(`ref2.food is: ${ref2.food}`);

Аналогичное происходит при вызове функций с параметрами. Если вы передадите строку или число, функция получит копию и не сможет изменить оригинал.

function changeFood(food)

{

// Change the parameter. This does not affect

// the originally passed variable.

food = "noodles";

}

// The variable starts off with the string "pizza"

let food = "pizza";

console.log(`First, the food is: ${food}`);

// Calling the function does not change the variable

changeFood(food);

// Log the variable again. It's still "pizza".

console.log(`Second, the food is: ${food}`);

С другой стороны, если функция изменяет какие-либо свойства переданного объекта, она изменяет оригинал, как и в случае с переменными. В приведенном ниже примере показана функция, изменяющая переданный объект: на этот раз изменяется оригинал.

function changeFoodProperty(obj)

{

// Change the property. This affects

// the originally passed object.

obj.food = "noodles";

}

// The 'food' property starts off as "pizza"

let myObject = {

food: "pizza"

};

console.log(`First, the food is: ${myObject.food}`);

// Calling the function changes the 'food' property

changeFoodProperty(myObject);

// Log the 'food' property again - it's now "noodles"

console.log(`Second, the food is: ${myObject.food}`);

Помните, что в целом:

* **Строки** и **числа** передаются **по значению** — их назначение или передача функциям приведет к созданию **копии.**
* **Объекты** передаются **по ссылке** — их назначение или передача функциям будет ссылаться на один и тот же объект.

Это ключевые правила, которые следует знать при написании кода JavaScript. На первый взгляд это может показаться непоследовательным, но на самом деле это полезная функция, на которую вы можете положиться. Также зачастую более эффективно передавать объекты по ссылке: если вам нужно передать в функцию большой и сложный объект, его копирование может оказаться относительно трудоемкой операцией, которая значительно замедляет работу программы. Передача его по ссылке позволяет избежать необходимости что-либо копировать, обеспечивая эффективную работу программы. Копирование строк и чисел, как правило, является быстрой и простой задачей для программ, поэтому разница в эффективности здесь не так важна.

Что, если вы действительно хотите сделать копию объекта? В JavaScript это удивительно сложная область, поэтому для простоты руководства мы не будем ее здесь рассматривать.

### Равенство

Вы также можете использовать операторы равенства ===и !==проверить, ссылаются ли ссылки на один и тот же объект. Обратите внимание, что при этом проверяется только ссылка , а не какие-либо свойства объекта или их значения, только если эти две вещи относятся к одному и тому же объекту. Пример ниже демонстрирует, как это работает.

// Create an object

let objA = {

food: "pizza"

};

// objB refers to the same object as objA

let objB = objA;

// objC refers to a different object, even though

// it has the same property and value

let objC = {

food: "pizza"

};

// Display equality

console.log("objA === objB: ", objA === objB); // true

console.log("objA === objC: ", objA === objC); // false

## Глобальный объект

В части 5 мы отметили, что переменные на «верхнем уровне» (не внутри каких-либо фигурных скобок) ограничены файлом и поэтому на них не могут ссылаться другие файлы сценариев. (Вот как область видимости работает в современных модулях JavaScript; старые сценарии «классического» режима работают иначе.)

Что, если вы хотите, чтобы значение хранилось глобально, и его можно было бы использовать даже в других файлах сценариев? JavaScript предоставляет глобальный объект, к которому вы можете обратиться с помощью globalThis, который всегда доступен везде. Затем, если вы добавите свойство глобального объекта, оно будет доступно всем сценариям. Код ниже демонстрирует добавление глобального свойства.

// Add a property on the global object.

globalThis.myGlobal = "Hello world!";

// Logs "Hello world!".

// This will also work in other script files,

// as the global object is available everywhere.

console.log(globalThis.myGlobal);

Использование globalThis— относительно недавнее дополнение к JavaScript. Вы можете увидеть старый код, использующий windowили selfссылающийся на глобальный объект. Это просто другие названия одного и того же.

Название globalThisнемного странное: почему у него есть «Это»? Короче говоря, до того, как было добавлено это имя, JavaScript существовал уже много лет globalThis, и по соображениям совместимости многие наиболее очевидные возможные имена нельзя было использовать. В JavaScript есть еще одно ключевое слово this(описанное ниже в этом руководстве), которое в некотором роде связано с ним, поэтому в итоге мы получили имя globalThis.

В JavaScript у глобального объекта также есть особенность: если вы ссылаетесь на имя переменной, которая не существует нигде в области видимости, он также проверит глобальный объект на наличие этого имени. Если он найден, он использует это свойство.

// Add a global property named myGlobal

globalThis.myGlobal = "Initial string";

// Assign myGlobal as if it's a variable

myGlobal = "Hello world!";

// Log myGlobal as if it's a variable

console.log(myGlobal); // Hello world!

Однако если имя не существует нигде в области видимости, а также не существует в глобальном объекте, это вызывает ошибку. (Это поможет вам обнаружить опечатки, поскольку неправильно введенное имя может привести к ошибке.)

// ReferenceError: doesNotExist is not defined

console.log(doesNotExist);

Обратите внимание, что вы не можете создать глобальное свойство без globalThis. Вы должны создать глобальные свойства, присвоив их свойству globalThis, и только тогда вы сможете получить к ним доступ как к глобальной переменной. Следующие действия вызовут ошибку, если myGlobalони еще не были созданы как свойства globalThis.

// ReferenceError: myGlobal is not defined

myGlobal = "Hello world!";

Обычно рекомендуется иметь как можно меньше глобальных переменных. Позже в этом руководстве будут рассмотрены модули, которые являются лучшим способом организации кода между файлами сценариев.

### Встроенные глобальные переменные

JavaScript предоставляет довольно много встроенных функций, к которым можно получить доступ через глобальный объект. На самом деле мы всегда использовали один из них: consoleобъект является свойством глобального объекта. Доступ к нему также можно получить через globalThis, например:

globalThis.console.log("Hello world!");

Однако, поскольку consoleобъект гарантированно всегда существует в глобальном объекте, эту globalThisчасть можно опустить, что мы и делали в наших примерах кода.

Еще в части 3 мы использовали String(...)и Number(...)для преобразования между строками и числами — они также являются глобальными переменными, и к ним также можно получить доступ с помощью globalThis.String(...). Но опять же, в этом нет необходимости, поскольку они встроены в JavaScript.

В язык JavaScript встроено множество глобальных переменных, от которых вы можете зависеть везде, где пишете код JavaScript. Браузеры добавляют множество других функций, предоставляющих широкий спектр сложных функций, от работы в сети до доступа к периферийным устройствам, таким как камеры. Другие среды, такие как [node.js,](https://www-construct-net.translate.goog/out?u=https://nodejs.org/&_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru) также добавляют различные наборы функций. Такие инструменты, как Construct, также предоставляют собственный набор функций. Эти наборы функций вместе называются интерфейсами прикладного программирования или API. На данный момент мы рассмотрели очень небольшое количество. Позже в этом руководстве будут рассмотрены еще несколько функций, встроенных в JavaScript, которые будут работать везде, где только доступен JavaScript, а также несколько API-интерфейсов браузера и некоторые API-интерфейсы, специфичные для Construct.

## Заключение

В этой части мы рассмотрели:

* Создание объекта
* Добавление и удаление свойств объекта
* Использование строк для ссылки на свойства объекта
* Использование inдля проверки существования свойства
* Вложение объектов
* Значениеnull​
* Опциональная цепочка с?.
* Как объекты передаются по ссылке, тогда как числа и строки передаются по значению
* Использование глобального объекта черезglobalThis

Об объектах можно рассказать еще больше, поскольку они представляют собой фундаментальный аспект программирования на JavaScript. В частности, использование свойств функций на объектах имеет некоторые дополнительные уникальные особенности. В следующей части мы рассмотрим это и некоторые другие подробности об объектах в JavaScript.

# Изучите JavaScript в Construct, часть 8: методы объекта

Это восьмая часть серии руководств [«Изучите JavaScript в Construct»](https://www-construct-net.translate.goog/en/courses/learn-javascript-construct-79?_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru) . Эта часть является продолжением части 7. Если вы ее пропустили, см. [«Изучаем JavaScript в Construct», часть 7: Объекты](https://www-construct-net.translate.goog/en/tutorials/learn-javascript-construct-2819?_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru) .

В этой части мы продолжим использовать пустой файл сценария в новом проекте Construct, как мы это делали в предыдущих нескольких уроках. Ранее мы рассматривали функции и объекты по отдельности, но в сочетании они обладают некоторыми интересными и полезными особенностями, которые рассматриваются в этой части.

## Методы

Объект может иметь функцию для одного из своих свойств. В приведенном ниже примере показан объект со свойством, logMessageкоторое является функцией.

let person = {

name: "Joe",

logMessage: function()

{

console.log("Hello world!");

}

};

// Logs "Hello world!" to the console

person.logMessage();

Обратите внимание, как мы объединяем несколько вещей, которые мы уже узнали:

* Есть объект со свойством с именемlogMessage
* Это свойство имеет значение, которое является функцией (с использованием функционального выражения).
* Доступ к свойству осуществляется с помощью точки, а функция вызывается с помощью (), поэтому строка person.logMessage()вызывает функцию, хранящуюся в logMessageсвойстве объекта.person

Функции, которые используются таким образом, также называются **методами** . Это просто другое имя функции, связанной с объектом.

Мы достигли точки, когда теперь можем видеть, что console.log()код, который мы использовали все это время, на самом деле вызывает logметод глобального consoleобъекта.

### Сокращенный синтаксис

Предыдущий пример также можно записать с использованием сокращенного синтаксиса метода logMessage, опустив двоеточие и functionключевое слово. Это работает точно так же, как и раньше, просто обеспечивает более короткий способ написания.

let person = {

name: "Joe",

// Shorthand syntax for method

logMessage()

{

console.log("Hello world!");

}

};

person.logMessage();

## этот

Предположим, мы хотим изменить logMessageметод logNameи включить в него nameсвойство объекта в зарегистрированном сообщении. Один из способов сделать это — обратиться к person.nameметоду. Это сработает, но метод будет привязан к конкретной personпеременной, в которой хранится объект. Объекты можно назначать различным переменным, передавать как параметры функции и т. д. Итак, нам действительно нужен способ сказать « nameсвойство объекта, для которого был вызван метод».

JavaScript предоставляет thisключевое слово для этого. Он ссылается на объект, для которого был вызван метод. В приведенном ниже примере демонстрируется logNameметод с использованием this.

let person = {

name: "Joe",

logName()

{

console.log(`The person's name is ${this.name}`);

}

};

// Logs "The person's name is Joe"

person.logName();

Другими словами, вызов obj.method()вызовет функцию method()со thisссылкой на obj.

Теперь у нас есть способ создавать объекты как со свойствами данных, так и с методами, использующими эти свойства данных. Это называется [объектно-ориентированным программированием](https://www-construct-net.translate.goog/out?u=https://en.wikipedia.org/wiki/Object-oriented_programming&_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru) или ООП.

thisфактически может использоваться в любой функции, и при вызове он по-прежнему ссылается на объект слева от точки. Следующий пример демонстрирует, как thisиспользуется в обычной функции, и когда он добавляется как свойство объекта и вызывается, он все равно ссылается на объект, для которого был вызван.

// Note the logName function uses 'this'

function logName()

{

console.log(`The person's name is ${this.name}`);

}

let person1 = {

name: "Joe"

};

let person2 = {

name: "Tom"

};

// Add the logName function as properties of both objects

person1.logName = logName;

person2.logName = logName;

// Logs: "The person's name is Joe"

person1.logName();

// Logs: "The person's name is Tom"

person2.logName();

Обратите внимание, что в приведенном выше примере оба logNameсвойства относятся к одной и той же функции. Однако каждый раз, когда он вызывается, thisон ссылается на другой объект, потому что слева от точки, когда он вызывается, находится другой объект.

### «это» без объекта

Что, если вы вызовете функцию, используя thisбез объекта? Это будет undefined, поскольку связанного объекта нет.

function logThis()

{

console.log(`'this' is ${this}`);

}

// Logs: 'this' is undefined

logThis();

Обратите внимание, что в нашем предыдущем примере с использованием logName, это может привести к ошибке, поскольку вы не можете прочитать свойства ничего, имеющего значение NULL ( nullили undefined).

function logName()

{

console.log(`The person's name is ${this.name}`);

}

// TypeError: Cannot read properties of undefined (reading 'name')

logName();

Точно так же, хотя вы можете писать код, используя thisвне функции, это также будет undefined.

console.log(this); // undefined

Другими словами, thisдействительно полезно обращаться к объекту, к которому был вызван метод, только внутри методов объекта. В остальном это вообще undefined. В JavaScript можно использовать и другие способы this, например, другие способы вызова функций с указанием того, thisна что следует ссылаться, но это более сложные случаи, которые мы здесь не будем рассматривать.

Вы можете встретить более старый код, основанный на особенностях JavaScript, где thisон фактически ссылается на глобальный объект (т. е. так же, как globalThis), а не undefinedпри использовании без объекта. Это происходит только в старых сценариях «классического» режима и может привести к ошибочным ошибкам. Construct всегда использует современные сценарии «строгого» режима, которые правильно реализуют правило, thisиспользуемое undefinedбез объекта.

## Стрелочные функции и «это»

В части 6 мы рассмотрели функции стрелок и кратко упомянули, как thisони по-разному работают. Давайте рассмотрим разницу сейчас.

Стрелочные функции не имеют собственного thisзначения. Вместо этого они берут значение thisиз окружающей области видимости. Пример ниже демонстрирует это.

let person = {

name: "Joe"

};

// Outside of a function, 'this' is undefined.

// The arrow function takes 'this' from the surrounding scope

// - which is here - so it is undefined.

person.arrowFunc = () => console.log(`'this' is ${this}`);

// Logs: 'this' is undefined

person.arrowFunc();

Обычно вы ожидаете, person.arrowFunc()что вызов arrowFunc()будет thisустановлен в person. Однако стрелочные функции переопределяют это правило и используют thisсвою внешнюю область видимости, в данном случае устанавливая thisзначение undefined.

На первый взгляд это может показаться странным исключением из правил, но на практике оно оказывается полезным. Это полезно для использования функций внутри методов, когда вы хотите повторно использовать одно и то же значение this, в частности, при использовании шаблона обратного вызова.

Чтобы проиллюстрировать это, давайте быстро пробежимся по событию щелчка . Браузеры предоставляют глобальный documentобъект, представляющий HTML-документ страницы. Многие объекты браузера также имеют addEventListenerметод обработки событий. Этот метод принимает функцию, которая будет вызываться браузером при возникновении данного события, также известную как обратный вызов . Таким образом, мы можем использовать следующий код, чтобы определять, когда пользователь щелкает мышью в любом месте страницы.

document.addEventListener("click", function ()

{

console.log("Click event!");

});

При предварительном просмотре щелкните внутри окна предварительного просмотра, а затем проверьте консоль браузера. Вы должны увидеть это в журнале событий Click! поскольку браузер вызывает эту функцию каждый раз, когда событие «щелчок» происходит в любом месте страницы.

Обратите внимание, что в консоли браузера, если одно и то же сообщение регистрируется несколько раз, обычно оно отображается с номером рядом с ним, а не повторяется.

Теперь попробуйте переместить этот код в метод и добавить ссылку на this. Это не будет работать должным образом, поскольку thisне относится к объекту person .

let person = {

name: "Joe",

initialize()

{

document.addEventListener("click", function ()

{

console.log(`Click event! The person's name is ${this.name}`);

});

}

}

// Call initialize() to add the click event listener

person.initialize();

Теперь, когда вы щелкаете мышью в окне предварительного просмотра, оно регистрирует событие Click! Имя человека не определено , а это не то, что нам нужно. Причина в том, что браузер вызывает функцию события «click» без ссылки на объект person .

В этом случае браузер на самом деле устанавливает thisзначение documentв надежде, что это поможет функции щелчка. Это все еще не то, что мы хотели, но также объясняет, почему это регистрируется, undefinedа не вызывает ошибку: thisссылается на объект, но у него нет nameсвойства.

Один из способов решить эту проблему — воспользоваться замыканием, которое мы рассмотрели в части 6. В initialize()функцию можно добавить переменную, которая запоминает значение this. Тогда функция «click» может вместо этого ссылаться на эту переменную.

let person = {

name: "Joe",

initialize()

{

// Save the value of 'this' when it refers to the right object

let savedThis = this;

document.addEventListener("click", function ()

{

// Refer to 'savedThis' instead of 'this'

console.log(`Click event! The person's name is ${savedThis.name}`);

});

}

}

person.initialize();

Теперь он работает как положено: когда мы нажимаем на страницу, регистрируется событие Click! Этого человека зовут Джо .

Однако мы можем воспользоваться тем фактом, что стрелочные функции принимают значение thisиз своей внешней области видимости, как savedThisв примере выше. Тогда мы можем обращаться к нему thisкак к нормальному, не имея другой переменной.

let person = {

name: "Joe",

initialize()

{

// Use an arrow function, which remembers 'this'

document.addEventListener("click", () =>

{

console.log(`Click event! The person's name is ${this.name}`);

});

}

}

person.initialize();

Это работает так же, как и раньше, правильно регистрируя имя Joe . Он демонстрирует, насколько полезны стрелочные функции как из-за их более короткого синтаксиса, так и для использования значения thisиз внешней области. Этот тип шаблона обратного вызова очень распространен на практике, поэтому эта особенность запоминания стрелочных функций thisна самом деле является полезной вещью, которой можно воспользоваться.

## Новый

Если вы хотите создать несколько объектов с одинаковыми свойствами, это может оказаться довольно повторяющимся. В приведенном ниже примере обратите внимание, как нам приходится повторять одни и те же свойства объекта снова и снова.

let person1 = {

name: "Joe",

age: 25,

preferredFood: "pizza"

};

let person2 = {

name: "Jenny",

age: 30,

preferredFood: "noodles"

};

let person3 = {

name: "Maya",

age: 20,

preferredFood: "burrito"

};

// Show each person in the console

console.log("person1:", person1);

console.log("person2:", person2);

console.log("person3:", person3);

В функцию можно передать несколько параметров console.log(), что и сделано в приведенном выше примере. Это полезно для отображения нескольких значений, в данном случае строкового сообщения и объекта.

Это можно упростить, вызвав функцию с new, например new Person(). Это делает следующее:

1. Создает новый пустой объект
2. Вызывает данную функцию с thisустановкой нового объекта
3. Когда функция завершается, возвращает объект

По соглашению, имя функции, используемой с, newначинается с заглавной буквы. Пример ниже демонстрирует более простой способ достижения предыдущего примера.

// This function is called via 'new', so 'this'

// will be set to a new empty object

function Person(name, age, preferredFood)

{

// Add the parameters as properties on the new object

this.name = name;

this.age = age;

this.preferredFood = preferredFood;

}

// Now each person object can be created with 'new'

let person1 = new Person("Joe", 25, "pizza");

let person2 = new Person("Jenny", 30, "noodles");

let person3 = new Person("Maya", 20, "burrito");

// Show each person in the console

console.log("person1:", person1);

console.log("person2:", person2);

console.log("person3:", person3);

Это исключает повторение, значительно упрощает изменение свойств, используемых объектом Person (поскольку теперь все они указаны в одном месте), и позволяет создавать более сложные объекты с гораздо меньшим объемом кода. Это ключевой аспект объектно-ориентированного программирования, поскольку теперь мы можем удобно создавать объекты определенного типа для представления определенных типов вещей.

JavaScript довольно спокойно относится к использованию функций и new. Функцию можно вызвать Personкак обычную функцию без new. Точно так же вы можете вызвать любую другую функцию, newдаже если она для этого не предназначена. Однако ни один из этих случаев на самом деле не является особенно полезным. Поэтому лучше всего использовать только newспециальную функцию, предназначенную для использования с ним.

Функции, используемые с, newназываются функциями- конструкторами , поскольку они конструируют (т. е. создают) новый объект определенным образом. (В названии программного обеспечения Construct , которое мы использовали, нет каламбура !)

### Методы

Также возможно добавлять методы в функцию-конструктор, например:

function Person(name, age, preferredFood)

{

this.name = name;

this.age = age;

this.preferredFood = preferredFood;

// Add a function

this.logName = function ()

{

console.log(`The person's name is ${this.name}`);

}

}

// Create a person

let person = new Person("Joe", 25, "pizza");

// Call the logName function on it

// Logs: The person's name is Joe

person.logName();

Обратите внимание: поскольку метод вызывается с помощью person.logName(), то внутри logNameфункция thisссылается на person, и поэтому она работает как положено.

Однако этот подход к добавлению методов на практике не получил широкого распространения. Проблема в том, что logNameдля каждой Personсозданной функции создается отдельная копия. Это необходимо для свойств данных, но не обязательно для функций — более эффективно использовать одну и ту же logNameфункцию для всех Personобъектов.

Этого можно достичь, назначая функции Person.prototype, что включает в себя цепочку прототипов JavaScript — довольно сложную тему. Однако в современном JavaScript этот подход также был заменен классами . Поскольку большая часть кода сейчас пишется именно так и решает одну и ту же задачу, мы пропустим детали цепочек прототипов и перейдем сразу к классам.

## Классы

Классы — общая черта многих объектно-ориентированных языков программирования, и JavaScript тоже их предлагает. Классы имеют широкий спектр сложных функций. Однако здесь мы рассмотрим только основы, продемонстрировав, как написать предыдущий пример с синтаксисом классов.

Предыдущий Personобъект можно объявить как класс следующим образом:

class Person {

constructor(name, age, preferredFood)

{

this.name = name;

this.age = age;

this.preferredFood = preferredFood;

}

}

// Create a person

let person = new Person("Joe", 25, "pizza");

// Show the object in the console

console.log("person: ", person);

Обратите внимание на сравнение с предыдущим примером:

* Теперь мы используем classключевое слово, за которым следует имя, а также фигурные скобки {и }, внутри которых находится содержимое класса.
* Внутри класса теперь есть функция с именем constructor, которая является специальным именем функции в JavaScript. Это выполняет ту же работу, что и Personфункция в предыдущем примере: она вызывается при использовании new Person, добавляя свойства к новому пустому объекту.
* Обратите внимание, что функции внутри класса используют сокращенный синтаксис методов.

Теперь мы можем добавить logNameфункцию в класс, снова используя сокращенный синтаксис:

class Person {

constructor(name, age, preferredFood)

{

this.name = name;

this.age = age;

this.preferredFood = preferredFood;

}

logName()

{

console.log(`The person's name is ${this.name}`);

}

}

// Create a person

let person = new Person("Joe", 25, "pizza");

// Call logName()

person.logName();

Это работает аналогично предыдущему примеру, но классы разделяют свои методы класса между всеми созданными объектами. Другими словами, одна и та же logNameфункция будет повторно использоваться для всех созданных Personобъектов. Это решает проблему предыдущего подхода, когда функция создавалась заново для каждого объекта. Он также использует более удобный синтаксис, чем альтернативный вариант с прототипами (который мы не рассматривали).

Классы имеют гораздо больше возможностей. Однако мы не будем здесь вдаваться в подробности. Если вы хотите углубиться в них, взгляните на [Классы на MDN](https://www-construct-net.translate.goog/out?u=https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes&_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru) . Однако, чтобы сосредоточить внимание этого руководства на быстром изучении основ JavaScript, мы продолжим. На данный момент мы только что показали, что если вы хотите создать множество объектов с одинаковыми свойствами, лучший способ сделать это — использовать classсинтаксис. Это работает аналогично использованию newс функцией, но классы предпочтительнее, и именно так пишется большая часть современного JavaScript.

## Заключение

В этой части мы рассмотрели;

* Использование функций в свойствах объекта, также известных как методы.
* Сокращенный синтаксис методов (также используется в классах)
* Как thisотносится к объекту, для которого был вызван метод
* Как функции стрелок обрабатываются thisпо-разному, что часто бывает полезно
* Добавление обработчика событий «щелчка», который запускает функцию всякий раз, когда что-то происходит.
* Использование newс функцией для создания объекта с определенными свойствами
* classсинтаксис как лучшая альтернатива для использования сnew

Это все, что нам нужно рассказать об объектах и ​​функциях, которые вместе были рассмотрены в последних четырех частях этого руководства. Однако стоит потратить время на их ознакомление, поскольку объекты, функции и методы являются фундаментальными частями программирования. Без них сложно сделать что-то серьёзное! В следующей части мы перейдем к некоторым другим структурам данных, полезным для программирования, например, к массивам.

# Изучите JavaScript в Construct, часть 9: Структуры данных

Это девятая часть серии руководств [«Изучите JavaScript в Construct»](https://www-construct-net.translate.goog/en/courses/learn-javascript-construct-79?_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru) . Эта часть является продолжением части 8. Если вы ее пропустили, см. [«Изучаем JavaScript в Construct», часть 8: Методы объектов](https://www-construct-net.translate.goog/en/tutorials/learn-javascript-construct-2823?_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru) .

В этой части мы продолжим использовать пустой файл сценария в новом проекте Construct, как мы это делали в предыдущих нескольких уроках. Пришло время узнать еще о паре структур данных в JavaScript. Эти структуры данных используются в большинстве других языков программирования, но мы научимся использовать их конкретно в JavaScript.

## Массивы

Массив представляет собой список значений. Значения в массиве также называются элементами . Элементы могут быть любого типа — строки, числа, объекты, даже функции и другие массивы. Для простоты мы сначала сосредоточимся на использовании строк.

Если вы использовали объект Array из Construct, концепция аналогична.

Подобно тому, как объекты могут быть созданы с помощью {и }, массивы могут быть созданы с помощью [и ]со списком значений, разделенных запятыми. Обратите внимание, что здесь нет имен свойств, а только значения.

let myArray = ["pizza", "noodles", "burrito"];

// Show this array in the console

console.log(myArray);

В приведенном выше примере создается массив с тремя строковыми элементами: "pizza", "noodles"и "burrito". Он записывает массив на консоль, чтобы вы могли проверить его аналогично просмотру объектов. Обратите внимание, что вам может потребоваться нажать кнопку «Развернуть», чтобы увидеть элементы внутри массива.

Массивы также имеют некоторые встроенные свойства, доступ к которым можно получить с помощью свойств объекта, подобных точке. Свойство lengthимеет количество элементов, которое в данном случае равно 3.

let myArray = ["pizza", "noodles", "burrito"];

console.log(`The array length is ${myArray.length}`);

Мы также использовали этот синтаксис для объектов. Фактически в JavaScript массивы представляют собой особый тип объектов, поэтому мы можем использовать с ними функции объектного стиля, такие как чтение свойств и вызов методов, в дополнение к другим функциям массивов.

Каждый элемент массива нумеруется индексом . Это **начинается с нуля** — большинство вещей в программировании на самом деле используют индекс, отсчитываемый от нуля, поэтому первый элемент имеет номер 0, а не 1. Это может показаться странным, но имеет смысл, учитывая, как все работает внутри (что в интересах простоты мы выиграли). не захожу сюда).

В предыдущем примере массив состоит из трех элементов со следующими индексами:

* Элемент **0**​"pizza"
* Элемент **1**​"noodles"
* Элемент **2**​"burrito"

Доступ к отдельным элементам можно получить, используя квадратные скобки после массива, например, myArray[0]для доступа к первому элементу с индексом 0. Обратите внимание, что это использование отличается от создания массива - в этом контексте квадратные скобки означают чтение элемента.

let myArray = ["pizza", "noodles", "burrito"];

console.log(`Element 0 is ${myArray[0]}`);

console.log(`Element 1 is ${myArray[1]}`);

console.log(`Element 2 is ${myArray[2]}`);

Обратите внимание на одну важную деталь: последний элемент имеет индекс 2, что на единицу меньше длины 3. Это связано с тем, что он использует индекс, отсчитываемый от нуля.

Элементы массива также могут быть назначены для их изменения, аналогично тому, как работают переменные.

let myArray = ["pizza", "noodles", "burrito"];

// Change the second element (at index 1)

myArray[1] = "chocolate";

console.log(`Element 0 is ${myArray[0]}`);

console.log(`Element 1 is ${myArray[1]}`);

console.log(`Element 2 is ${myArray[2]}`);

### Итерация массива

Массивы могут быть любого размера. Итак, если вы хотите сделать что-то вроде регистрации каждого элемента на консоли, нам понадобится forцикл, который повторяется один раз для каждого элемента, вплоть до длины массива. Пример ниже демонстрирует это.

let myArray = ["pizza", "noodles", "burrito"];

// Repeat once for each element in the array

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

{

console.log(`Element ${i} is ${myArray[i]}`);

}

Это повторит журнал консоли со iзначениями 0, 1 и 2, что соответствует индексам элементов. Обратите внимание на проверку условий цикла, длина которых iменьше длины массива — не меньше или равна — поскольку, как мы упоминали ранее, последний элемент массива имеет индекс на единицу меньше длины.

Другой способ сделать это — использовать другой тип цикла, предоставляемый JavaScript, который называется for..ofциклом «for-of». Вместо трех операторов для управления переменной в цикле используется ofключевое слово. Пример ниже демонстрирует это.

let myArray = ["pizza", "noodles", "burrito"];

// Repeat once for each element of myArray

for (let elem of myArray)

{

console.log(`The array element is ${elem}`);

}

В этой форме цикл объявляет переменную с именем elem. Затем цикл автоматически повторяется один раз для каждого элемента myArray, при этом переменная цикла elemустанавливает значение текущего элемента. Таким образом, этот цикл также будет регистрировать сообщение с каждым элементом на консоли.

Этот тип цикла немного удобнее писать, так как вам не нужно включать части цикла для сравнения или увеличения. Однако обратите внимание, что он не обеспечивает прямого определения текущего индекса — он дает только фактические значения. Поэтому в некоторых случаях, если вам все равно нужно знать индекс массива в цикле, например, для одновременного чтения из двух массивов одинакового размера, может быть удобнее использовать предыдущий тип цикла for.

Циклы For-of также работают с другими вещами в JavaScript, и на самом деле детали того, как они работают, относятся к довольно сложной области JavaScript, известной как итераторы. Мы не будем здесь вдаваться в подробности, но стоит знать, что циклы for-of — это еще один способ сделать что-то для каждого элемента массива.

### Добавление и удаление элементов

JavaScript предоставляет множество способов изменения массивов, включая добавление, удаление и замену элементов. Для простоты мы рассмотрим здесь только два.

Вы можете использовать встроенный push()метод массива, чтобы добавить новый элемент в конец массива. Давайте также используем смайлики для строк!

let myArray = ["😀", "👽", "👾"];

// Add a new element at the end of the array

myArray.push("🤖")

// Show the array in the console

// It now has: 😀, 👽, 👾, 🤖

console.log(myArray);

После добавления элемента lengthзначение свойства автоматически увеличивается до 4, что отражает новый размер массива.

Метод pop()также удаляет последний элемент из конца массива, а также возвращает значение удаленного им элемента.

let myArray = ["😀", "👽", "👾"];

// Remove element from end of array

let removedElem = myArray.pop();

// Show the array in the console

// It now has: 😀, 👽

console.log("The array has: ", myArray);

// Show the removed element in the console (👾)

console.log(`The removed element is: ${removedElem}`);

После удаления элемента lengthсвойство массива автоматически уменьшается до 2.

### Еще несколько деталей

Еще в части 7 мы рассмотрели, как строки и числа передаются по значению, а объекты передаются по ссылке. Массивы также передаются по ссылке в JavaScript. (На самом деле, как отмечалось ранее, массивы в JavaScript представляют собой объекты особого типа.)

Массивы — это тип коллекции , то есть имя, данное структурам данных, хранящим несколько значений. Массивы также являются упорядоченной коллекцией, поскольку элементы имеют четкий порядок, и вы можете определить, идет ли один элемент до другого или после него, и порядок можно изменить (хотя мы не рассматривали это здесь). В этом отличие от других структур данных, которые могут не иметь определенного порядка или не могут быть переупорядочены.

Массивы имеют гораздо больше возможностей в JavaScript. Они являются еще одной ключевой фундаментальной частью программирования и поэтому обладают широким спектром возможностей, которые делают их гибкими и полезными для решения множества задач. Мы вернемся к еще нескольким из этих функций позже в этом руководстве. Однако мы рассмотрели основы массивов и для простоты продолжим.

## Набор

В JavaScript a Set— это коллекция уникальных значений. Это означает, что если вы попытаетесь добавить одно и то же значение дважды, оно будет добавлено в коллекцию только один раз. Это полезная функция, если вы хотите иметь уникальные значения, но еще одна причина ее использования заключается в том, что для этой цели она более эффективна, чем массивы.

Например, представьте, что вы хотите, чтобы у каждого, кто использует вашу программу, было уникальное имя пользователя, и что 1 миллион человек уже зарегистрировались. Когда следующий человек придет и скажет вам, какое имя пользователя он хочет, вы должны убедиться, что это не одно из 1 миллиона имен пользователей, которые уже используются. При использовании массива это означало бы необходимость проверки каждого имени пользователя в списке, т. е. 1 миллион проверок. Однако Setиспользуются специальные алгоритмы поиска, позволяющие сделать эту проверку гораздо более эффективной.

С другой стороны, это улучшение поиска означает, что теряется возможность доступа к элементам по их индексу. Таким образом, мы можем проверить, есть ли элементы в наборе, но не имеем прямого доступа, скажем, к пятому элементу. Если все, что вам нужно сделать, это проверить, используется ли уже имя пользователя, это не проблема. Это типичный пример множества компромиссов, связанных с программированием, и того, почему так важно выбрать правильную структуру данных для работы.

В JavaScript наборы не имеют специального синтаксиса, как массивы — они больше похожи на объекты. Основное использование показано ниже.

// Create a new empty set

let mySet = new Set();

// Add a string to the set

mySet.add("😀");

// Display the set in the console

console.log("mySet is: ", mySet);

Обратите внимание, как с помощью метода создается новый пустой набор new Set()и добавляется значение add().

Набор можно создать с некоторыми начальными значениями, передав массив. Несмотря на использование здесь массива, набор прочитает все эти значения и добавит их в свою собственную структуру данных — массив — это всего лишь способ предоставить начальные значения.

// Create a set with two strings

let mySet = new Set(["😀", "👽"]);

// Add another string to the set

mySet.add("👾");

// Display the set in the console

// It now has: 😀, 👽, 👾

console.log("mySet is: ", mySet);

Пока это работает во многом как массив, но мы сможем увидеть разницу, если попытаемся добавить одну и ту же строку дважды:

// Create a set with two strings

let mySet = new Set(["😀", "👽"]);

// Add another string to the set

mySet.add("👾");

// Add the same string again - this does nothing

// as it's already in the set

mySet.add("👾");

// Display the set in the console

// It still has: 😀, 👽, 👾

console.log("mySet is: ", mySet);

Добавление значения, уже находящегося в наборе, ничего не дает, поскольку в наборе хранятся только уникальные значения.

У наборов есть sizeсвойство, указывающее количество элементов в наборе. Это похоже на lengthсвойство массивов, но помните, что наборы хранят только уникальные значения, поэтому добавление значения не приведет к увеличению размера, если оно уже есть в наборе.

let mySet = new Set(["😀", "👽"]);

console.log(`The set size is ${mySet.size}`);

// This value is already in the set, so nothing is added

mySet.add("👽");

// The set size is still the same

console.log(`The set size is still ${mySet.size}`);

Этот delete()метод можно использовать для удаления элемента.

let mySet = new Set(["😀", "👽", "👾"]);

mySet.delete("👽");

// Set has 😀 and 👾

console.log("mySet is: ", mySet);

Значение, которого нет в наборе, все равно можно удалить — оно просто ничего не делает. Существует также clear()метод, который удаляет все элементы.

Метод has()возвращает логическое значение, указывающее, находится ли данное значение в наборе.

let mySet = new Set(["😀", "👽"]);

console.log("mySet is: ", mySet);

// true

console.log("mySet has 😀: ", mySet.has("😀"));

// false

console.log("mySet has 👾: ", mySet.has("👾"));

### Итерация набора

Поскольку наборы не могут получить доступ к элементам по их индексу, единственный способ перебора значений — это цикл стиля «for-of», как в примере ниже.

let mySet = new Set(["😀", "👽", "👾"]);

for (let item of mySet)

{

console.log(`mySet has item ${item}`);

}

JavaScript определяет порядок наборов, поэтому говорить, что они неупорядочены, не совсем корректно. Цикл будет перебирать элементы в порядке вставки , то есть в том порядке, в котором они были добавлены в набор. Однако порядок элементов изменить нельзя, включая тот факт, что если вы добавите элемент, уже входящий в набор, порядок существующего элемента не изменится. В целом это означает, что порядок набора не особенно полезен — основная причина, по которой JavaScript определяет порядок, состоит в том, чтобы обеспечить согласованную работу программ на разных платформах. Поэтому лучше всего рассматривать наборы как коллекцию, где порядок не важен. Наш предыдущий пример хранения списка из 1 миллиона имен пользователей для проверки того, какие из них используются, является хорошим примером того, где порядок не имеет значения — вы просто хотите знать, есть ли данное имя пользователя в списке или нет, и вы не делаете этого. Мне не нужно знать, где именно оно находится в списке.

### Преобразование в/из массива

Мы уже рассмотрели, как преобразовать массив в набор: если вы передаете массив при создании набора, он добавляет в набор все элементы массива.

let mySet = new Set(["😀", "👽"]);

Обратите внимание: как и при использовании add(), если в массиве есть дубликаты, будут добавлены только уникальные значения. Передача массива — это, по сути, ярлык для вызова add()каждого элемента.

let mySet = new Set(["😀", "👽", "👽"]);

// Set has just 2 elements: 😀 and 👽

console.log(`mySet size is ${mySet.size}`);

Набор также можно преобразовать обратно в массив с помощью оператора расширения... в форме [...mySet]:

let mySet = new Set(["😀", "👽"]);

let asArray = [...mySet];

console.log("asArray: ", asArray);

Вы можете думать о [и ]как о создании нового массива, ...mySetимея в виду, что содержимым массива является каждый элемент в mySet.

Это также обеспечивает хороший способ удаления повторяющихся элементов из массива. Преобразование массива в набор удаляет дубликаты, а затем преобразование набора обратно в массив снова дает уникальные элементы в виде массива. В примере кода ниже показана функция, которая делает это.

function removeDuplicates(arr)

{

// Convert array to set to remove duplicates

let asSet = new Set(arr);

// Return set converted back to an array

return [...asSet];

}

console.log("Example 1: ", removeDuplicates(["😀", "👽", "👾", "😀"]));

console.log("Example 2: ", removeDuplicates(["👽", "👽", "👽"

## карта

В JavaScript a Mapочень похоже на набор, но вставленные значения на самом деле являются ключами со связанным значением. Это похоже на свойства объекта, которые мы рассмотрели в части 7: каждое свойство имеет имя и связанное с ним значение. Действительно, объекты можно использовать аналогично карте, поскольку свойства можно добавлять, удалять, извлекать и так далее. Однако карты являются лучшим решением, если вам нужен динамический набор ключей. Во-первых, вы можете использовать любой тип значения для ключей, включая числа и даже объекты, что невозможно для свойств объекта, которые должны быть строками. Во-вторых, в объектах JavaScript есть встроенные имена свойств, и их перезапись может вызвать проблемы. Карты не имеют таких встроенных имен ключей, поэтому с картами эта проблема никогда не возникает.

Если вы использовали объект Dictionary из Construct, концепция аналогична. (Название «Словарь» — еще один распространенный термин в программировании, обозначающий ту же структуру данных, что и в JavaScript Map.)

Как и в случае с наборами, новую пустую карту можно создать с помощью new Map(). Однако метода нет add()— ключи и значения вставляются с помощью set(key, value). Затем вы можете использовать его get(key)для возврата значения, связанного с ключом. В приведенном ниже примере показано использование карты для хранения описания смайлика.

// Create a new empty map

let myMap = new Map();

// Add three keys for emoji and their associated description

myMap.set("😀", "smiley face");

myMap.set("👽", "alien");

myMap.set("🦄", "unicorn");

// Show the map in the console

console.log("myMap is: ", myMap);

// Retrieve the description for each emoji

console.log("Value for 😀: ", myMap.get("😀"));

console.log("Value for 👽: ", myMap.get("👽"));

console.log("Value for 🦄: ", myMap.get("🦄"));

Как показано выше, если вы позвоните set(key, value)с ключом, которого нет в карте, он добавит его. Однако если вы вызываете set(key, value)ключ, который уже существует, он просто заменяет значение ключа.

Если вы вызовете get(key)ключ, которого нет на карте, он вернет undefined, аналогично доступу к несуществующему свойству объекта.

Карта также может быть создана с некоторыми исходными ключами и значениями. Это немного сложнее, чем с набором: каждая запись на карте на самом деле представляет собой ключ и значение, то есть два значения. Таким образом, вы можете создать карту, используя массив из двухэлементных массивов, где первый элемент является ключом, а второй элемент — значением. В следующем примере показано добавление тех же ключей и значений, что и в предыдущем примере, но путем создания карты с начальными ключами и значениями.

let myMap = new Map([

["😀", "smiley face"],

["👽", "alien"],

["🦄", "unicorn"]

]);

console.log("Value for 😀: ", myMap.get("😀"));

console.log("Value for 👽: ", myMap.get("👽"));

console.log("Value for 🦄: ", myMap.get("🦄"));

Как и в случае с наборами, здесь также есть sizeсвойство с количеством ключей, has(key)метод проверки наличия ключа на карте, delete(key)метод удаления ключа и связанного с ним значения с карты и clear()метод удаления всего.

let myMap = new Map([

["😀", "smiley face"],

["👽", "alien"],

["🦄", "unicorn"]

]);

// Delete a key

myMap.delete("👽");

// The size is 2, as one of the keys was deleted

console.log(`myMap size is ${myMap.size}`);

// true

console.log("myMap has 😀: ", myMap.has("😀"));

// false (it was deleted)

console.log("myMap has 👽: ", myMap.has("👽"));

### Итерация карты

Записи в картах можно повторять с помощью цикла for-of, аналогично наборам. Однако вы можете перебирать его по-разному: вы можете перебирать только ключи, только значения или и то, и другое одновременно. Чтобы перебирать только ключи, используйте цикл for-of с myMap.keys(). Чтобы перебирать только значения, используйте цикл for-of с myMap.values(). Следующий пример демонстрирует оба варианта.

let myMap = new Map([

["😀", "smiley face"],

["👽", "alien"],

["🦄", "unicorn"]

]);

// Iterate map keys

for (let key of myMap.keys())

{

console.log(`Map key: ${key}`);

}

// Iterate map values

for (let value of myMap.values())

{

console.log(`Map value: ${value}`);

}

Обратите внимание, что, как и в случае с наборами, это соответствует порядку вставки.

Можно одновременно перебирать как ключи, так и значения. Однако это требует нового синтаксиса. Мы хотим использовать цикл for-of с двумя значениями на итерацию, что мы можем сделать, используя квадратные скобки для объявления двух переменных в цикле for-of. Объект для итерации также был изменен на myMap.entries(), который перебирает как ключ, так и значение.

let myMap = new Map([

["😀", "smiley face"],

["👽", "alien"],

["🦄", "unicorn"]

]);

// Iterate both keys and values

for (let [key, value] of myMap.entries())

{

console.log(`Map key: ${key}, value: ${value}`);

}

Цикл for-of также можно написать for (let [key, value] of myMap), т.е. без вызова .entries(), поскольку по умолчанию, если вы не укажете что-то еще, выполняется итерация записей.

Синтаксис let [key, value]является частью JavaScript, называемой деструктуризацией . Это еще одна довольно сложная область, в которую мы сейчас не будем углубляться, но мы кратко рассмотрели ее здесь, просто чтобы продемонстрировать, как можно одновременно перебирать как ключи, так и значения на карте. А пока стоит отметить сходство с тем, как создаются карты с двухэлементными массивами для каждого ключа и значения, например ["😀", "smiley face"]: цикл for-of поместит первый элемент в первую переменную, а второй элемент во вторую переменную. .

### Преобразование в массив

Подобно наборам, карты также можно преобразовать в массив с помощью [...myMap]. При этом создается массив в том же формате, который использовался для инициализации карты: массив из двухэлементных массивов с ключом и значением.

let myMap = new Map([

["😀", "smiley face"],

["👽", "alien"],

["🦄", "unicorn"]

]);

// Convert map back to array

let asArray = [...myMap];

// Log array to console.

// This is the same format as the map was

// initialised with, i.e.:

// [

// ["😀", "smiley face"],

// ["👽", "alien"],

// ["🦄", "unicorn"]

// ]

console.log("Map converted to array: ", asArray);

## Заключение

В этой части мы рассмотрели:

* Создание массивов, доступ к элементам массива и чтение длины массива
* Итерация массивов с помощью циклов for и for-of
* Добавление и удаление элементов массива
* Создание наборов, добавление и удаление значений наборов, а также чтение размера набора.
* Итерация наборов и преобразование наборов обратно в массивы
* Создание карт, добавление и удаление ключей и значений, а также получение значений из ключей.
* Различные способы перебора карт и преобразования карт обратно в массивы.

Эти структуры данных пригодятся при повседневном программировании на JavaScript, поэтому полезно ознакомиться с тем, как они работают. В частности, массивы очень распространены, поэтому, если вы хотите сосредоточиться на одной области, сосредоточьтесь на них.

В этом руководстве мы затронули несколько встроенных функций JavaScript. В следующей части мы сосредоточимся на дополнительных встроенных модулях, включая некоторые из многих функций, предоставляемых браузерами.

# Изучите JavaScript в Construct, часть 10: Стандартная библиотека

Это 10-я часть серии руководств [«Изучите JavaScript в Construct»](https://www-construct-net.translate.goog/en/courses/learn-javascript-construct-79?_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru) . Эта часть является продолжением части 9. Если вы ее пропустили, см. [«Изучаем JavaScript в Construct», часть 9: Структуры данных](https://www-construct-net.translate.goog/en/tutorials/learn-javascript-construct-2828?_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru) .

В этом руководстве мы уже рассмотрели некоторые стандартные встроенные функции JavaScript, такие как методы push()и pop()для массивов. В этой части мы рассмотрим еще кое-что. JavaScript предоставляет полный набор стандартных встроенных функций, часто называемых стандартной библиотекой, и мы рассмотрим лишь некоторые из наиболее полезных функций. Как только вы освоитесь с использованием функций стандартной библиотеки, вам будет несложно найти дополнительную информацию в справочнике по веб-документам MDN и опробовать их.

В этой части мы также вернемся к опробованию фрагментов кода в консоли в разное время, поскольку это хороший способ быстро опробовать некоторые из более простых встроенных функций.

## Консоль

Для полноты картины стоит отметить, что console.log()код, который мы использовали в этом руководстве, использует log()метод встроенного consoleобъекта для добавления сообщений в консоль браузера. Существует также console.warn()и , console.error()которые работают очень похоже, но отображают сообщения в виде предупреждений или ошибок соответственно в консоли. Это может быть полезно для диагностики — например, вместо того, чтобы молча допустить сбой, полезно зарегистрировать ошибку на консоли, чтобы вы могли проверить, не пошло ли что-то не так в консоли.

Вы можете использовать гораздо больше консольных методов — узнайте больше в [документации консоли MDN](https://www-construct-net.translate.goog/out?u=https://developer.mozilla.org/en-US/docs/Web/API/console&_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru) .

## Математика

Язык JavaScript предоставляет встроенный Mathобъект, предоставляющий множество методов для математических вычислений. Несколько полезных из них приведены ниже. Попробуйте их в консоли браузера.

// Try entering in to the browser console:

// Math.PI: convenient constant for PI

Math.PI // 3.141592653589793

// Math.round(): round number to nearest whole number

Math.round(5.2) // 5

Math.round(5.6) // 6

Math.round(-1.2) // -1

// Math.floor(): round down to nearest whole number

// (towards negative infinity)

Math.floor(5.2) // 5

Math.floor(5.6) // 5

Math.floor(-1.2) // -2

// Math.ceil(): round up to nearest whole number

// (towards infinity)

Math.ceil(5.2) // 6

Math.ceil(5.6) // 6

Math.ceil(-1.2) // -1

// Math.max(): get highest of a range of numbers

// (you can pass as many parameters as you like)

Math.max(3, 5) // 5

Math.max(3, 7, 1, 4) // 7

// Math.min(): get the lowest of a range of numbers

Math.min(3, 5) // 3

Math.min(3, 7, 1, 4) // 1

// Math.abs(): get absolute value (turns a number positive)

Math.abs(-3) // 3

Math.abs(3) // 3

// Math.sqrt(): calculate square root

Math.sqrt(25) // 5

// Math.pow(): raise to power, like \*\* operator

Math.pow(5, 2) // 25

// Math.random(): generate random number in range 0-1

// (excluding 1) - each call returns a different number

Math.random()

// Trigonometric calculations. Note these work with angles

// in radians - where a full rotation is 2 \* pi -

// instead of using degrees. Remember fractional calculations

// are not always perfectly precise, so answers may be very

// close to but not exactly equal to the true answer.

Math.sin(Math.PI / 2) // 1

Math.cos(0) // 1

Math.tan(Math.PI / 4) // 0.9999999999999999

Math.asin(1) // 1.5707963267948966

Math.acos(1) // 0

Math.atan(1) // 0.7853981633974483

// Math.atan2(y, x) is another special method which does some extra

// work to make it useful for calculating angles. For example a position

// at (10, 10) is 45 degrees from the origin, so Math.atan2(10, 10)

// returns Math.PI / 4 (which is 45 degrees in radians).

Math.atan2(10, 10) // 0.7853981633974483

Еще больше вы можете найти в [документации Math MDN](https://www-construct-net.translate.goog/out?u=https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math&_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru) .

### Пример кода

В следующем примере кода показано использование функции для генерации случайного целого числа от 0 до предела — в данном случае 10 (поэтому она генерирует случайные числа от 0 до 9 включительно, поскольку Math.random()никогда не возвращает 1). Затем он генерирует 20 случайных чисел, добавляет их в массив и отображает этот массив в консоли. Попробуйте добавить это в скрипт в проекте Construct.

// Get a random whole number up to, but not including, the limit

function getRandomNumber(limit)

{

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

}

// Generate 20 random numbers from 0-9 in an array

let arr = [];

for (let i = 0; i < 20; i++)

{

arr.push(getRandomNumber(10));

}

// Display array of random numbers in console

console.log("Random numbers: ", arr);

## Строковые методы

Ранее мы рассмотрели такие строки, как "Hello world". В JavaScript строки также являются объектами особого типа, поэтому у них есть свойства и методы, которые вы можете вызывать. Например, "Hello".length5, так как строка состоит из 5 символов.

Строки в вычислениях внутренне гораздо сложнее, чем может показаться на первый взгляд. То, что JavaScript считает отдельными символами в строке, может отличаться от того, что отображается на самом деле. Например "😀".length, равно 2, хотя в строке только один символ (смайлик). Причинами этого являются сложная смесь дизайнерских решений JavaScript, принятых много лет назад, и то, как работает Unicode (который является международным стандартом для текста, охватывающим все языки мира). В целях простоты мы не будем вдаваться в подробности. Лучший подход — преобразовать строку в массив, что позволяет избежать этих проблем. Это описано ниже.

Возможно, вы заметили, что массивы также являются особым типом объектов в JavaScript. На самом деле в JavaScript почти всё является объектом! Вот несколько полезных строковых методов, которые вы можете опробовать в консоли.

// Try entering in to the browser console:

// toLowerCase() returns a new string with lower case

"Hello World".toLowerCase() // "hello world"

// toUpperCase() returns a new string with upper case

"Hello World".toUpperCase(); // "HELLO WORLD"

// includes() returns a boolean indicating if a given

// string appears in the string it is called on

"Hello world".includes("world") // true

"Hello world".includes("pizza") // false

// startsWith() returns a boolean indicating if the

// string it is called on starts with a given string

"Hello world".startsWith("Hello") // true

"Hello world".startsWith("Pizza") // false

// endsWith() does the same but for the end of the string

"Hello world".endsWith("world") // true

"Hello world".endsWith("pizza") // false

// repeat() returns a new string which repeats the

// string it is called on a number of times

"😀".repeat(5) // 😀😀😀😀😀

// split() returns an array of strings separated

// by a given character

"pizza,chocolate,burrito".split(",")

// returns an array:

// ["pizza", "chocolate", "burrito"]

Строки также можно преобразовать в массив с одним символом на элемент с помощью оператора расширения .... В приведенном ниже примере показано, как затем можно использовать массив для доступа к количеству символов и символу по определенному индексу.

// A string with an emoji

let str = "Hi😀";

// Convert the string to an array

let arr = [...str];

// The array now has:

// [ "H", "i", "😀" ]

// Log details about the string from the array

console.log(`The array length is ${arr.length}`);

console.log(`The 3rd character is ${arr[2]}`);

Обратите внимание: если вы напрямую обращаетесь к длине и индексу строки, вы получите потенциально неожиданные ответы, как отмечалось ранее. Преобразование строки в массив позволяет избежать этих проблем, поэтому это предпочтительный способ доступа к строкам по символам.

Вы также можете использовать цикл for-of для прямого перебора символов строки.

// A string with an emoji

let str = "Hi😀";

// Use a for-of loop to repeat for each character

for (let ch of str)

{

console.log(`String character: ${ch}`);

}

Использование цикла for со строкой также работает правильно, поскольку оно работает аналогично преобразованию строки в массив с последующим перебором массива.

Со строками в JavaScript можно сделать гораздо больше. Дополнительную информацию см. [в документации по String MDN .](https://www-construct-net.translate.goog/out?u=https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String&_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru)

## Методы массива

В части 9 мы рассмотрели массивы и два метода push()для добавления элемента в конец и pop()удаления элемента из конца. Массивы предоставляют гораздо больше полезных функций, позволяющих удобно и эффективно обрабатывать списки элементов.

Обратите внимание, что некоторые методы массива изменяют массив, к которому они вызваны, в то время как другие возвращают совершенно новый массив, не изменяя исходный.

join()это как противоположность split()строкам. Он автоматически объединяет все элементы массива в строку, при необходимости добавляя дополнительные строки между элементами. Мы также будем использовать это для регистрации снимка массива на консоли.

// Try entering in to the browser console:

["😀", "👽", "🤖"].join(",") // "😀,👽,🤖"

Метод at()работает аналогично доступу к элементам массива с помощью квадратных скобок. Однако он также поддерживает использование отрицательных чисел для индексации с конца массива, что удобно для доступа к последнему элементу.

// Try entering in to the browser console:

["😀", "👽", "🤖"].at(-1) // 🤖

Метод includes()возвращает логическое значение, указывающее, есть ли в массиве данный элемент или нет.

Помните, что если вы хотите эффективно проверять наличие элементов, возможно, лучше использовать файл Set.

// Try entering in to the browser console:

["😀", "👽", "🤖"].includes("👽") // true

["😀", "👽", "🤖"].includes("👻") // false

Аналогично indexOfметод возвращает индекс элемента, если он был найден, или -1, если он не был найден.

// Try entering in to the browser console:

["😀", "👽", "🤖"].indexOf("👽") // 1

["😀", "👽", "🤖"].indexOf("👻") // -1

В следующих примерах используется несколько строк кода, поэтому попробуйте использовать их в файле сценария Construct, а не в консоли браузера.

В то время как push()и pop()добавлять и удалять из конца массива, unshift()а также shift()добавлять и удалять из начала массива.

let arr = ["😀", "👽"];

console.log(`Array at start: ${arr.join(",")}`);

// Add at start of array

arr.unshift("🤖");

// Array is now:

// ["🤖", "😀", "👽"]

console.log(`Array after unshift: ${arr.join(",")}`);

// Remove from start of array

let removedElem = arr.shift();

// Array is now:

// ["😀", "👽"]

console.log(`Array after shift: ${arr.join(",")}`);

console.log(`Removed element: ${removedElem}`);

Метод fill()заменяет каждый элемент массива одним и тем же.

let arr = ["😀", "👽", "🤖"];

arr.fill("👾");

// Array is now ["👾", "👾", "👾"]

console.log(`Array after fill: ${arr.join(",")}`);

Метод reverse()меняет порядок элементов в массиве.

let arr = ["😀", "👽", "🤖"];

arr.reverse();

// Array is now ["🤖", "👽", "😀"]

console.log(`Array after reverse: ${arr.join(",")}`);

Метод slice()возвращает новый массив с копией диапазона элементов из массива, из которого он был вызван. Вы можете указать либо только начальный индекс (который копируется из этого индекса до конца), либо как начальный, так и конечный индекс (в этом случае диапазон доходит до конечного индекса, но не включает его).

let arr1 = ["😀", "👽", "🤖", "👻"];

// Copy from index 1 to the end

// This returns ["👽", "🤖", "👻"]

let arr2 = arr1.slice(1);

// Copy from index 1 to index 3 (not including 3)

// This returns ["👽", "🤖"]

let arr3 = arr1.slice(1, 3);

console.log("arr1: ", arr1);

console.log("arr2: ", arr2);

console.log("arr3: ", arr3);

Этот splice()метод может как добавлять, так и удалять элементы из любого места массива. Требуется начальный индекс, количество элементов для удаления, а затем, при необходимости, дополнительные параметры элементов для вставки в индекс.

Постарайтесь не путать методы срез и сращивание , так как их названия похожи.

let arr = ["😀", "👽", "🤖", "👻"];

console.log(`Starting array: ${arr.join(",")}`);

// splice(1, 2) means deletes 2 elements from index 1

arr.splice(1, 2);

// Array is now: ["😀", "👻"]

console.log(`After deleting 2 elements: ${arr.join(",")}`);

// splice(1, 0, "👽", "🤖") means from index 1, don't delete

// anything (as we provide 0 for the number of elements to

// delete), and insert elements "👽" and "🤖".

// So this adds back the elements that were deleted.

arr.splice(1, 0, "👽", "🤖");

// Array is now: ["😀", "👽", "🤖", "👻"]

console.log(`After adding 2 elements: ${arr.join(",")}`);

// We can also simultaneously delete and insert elements.

// This time we delete 2 elements and insert 2 elements,

// replacing the middle two elements with different emoji.

arr.splice(1, 2, "👾", "🎉");

// Array is now: ["😀", "👾", "🎉", "👻"]

console.log(`After replacing 2 elements: ${arr.join(",")}`);

Метод sort()сортирует элементы массива в порядке возрастания, используя по умолчанию текстовый алфавитный порядок.

let arr = ["beta", "alpha", "gamma"];

arr.sort();

// Array is now: ["alpha", "beta", "gamma"]

console.log(`Sorted array: ${arr.join(",")}`);

Обратите внимание, что текстовый алфавитный порядок по умолчанию не подходит для чисел: при таком порядке сортировка «1», «2» и «10» приведет к получению «1», «10», «2», что правильно в алфавитном порядке, но не численно. Чтобы выполнить числовую сортировку, метод опционально принимает функцию, которая сравнивает два элемента. Метод должен возвращать отрицательное число, если первый параметр упорядочивается первым, положительное число, если первым упорядочивается второй параметр, или 0, если они имеют одинаковый порядок. Самый простой способ сделать это — использовать стрелочную функцию, которая просто вычитает параметры.

let arr = [10, 2, 1, 7, 12, 8];

arr.sort((a, b) => a - b);

// Array is now: [1, 2, 7, 8, 10, 12]

console.log(`Sorted array: ${arr.join(",")}`);

Другим случаем использования функции для изменения массивов является mapметод. Это вызывает функцию для каждого элемента массива и создает новый массив с возвращаемым значением функции.

let arr1 = ["😀", "👽", "🤖", "👻"];

console.log(`Starting array: ${arr1.join(",")}`);

// map() will call the function on every element in the array.

// The function adds "🎉" to the end of the string.

// It then returns a new array with the return values of the

// function - so every element has "🎉" added on the end.

let arr2 = arr1.map(elem => elem + "🎉");

// arr2 is now: ["😀🎉", "👽🎉", "🤖🎉", "👻🎉"]

console.log(`Mapped array: ${arr2.join(",")}`);

Наконец reduce()использует метод для объединения всех элементов массива. Это полезно для таких вещей, как суммирование всех чисел в массиве. Он принимает функцию, которая вызывается для каждого элемента для его объединения — в данном случае просто прибавления к итогу. Он также принимает начальное значение, что полезно в случае, если массив пуст (так как тогда объединять будет нечего).

let arr = [1, 2, 3, 4];

// Sum every element in the array.

// Note 0 is also passed as the starting value.

let sum = arr.reduce((total, current) => total + current, 0);

console.log(`Sum of array: ${sum}`);

С массивами в JavaScript можно сделать гораздо больше. Дополнительную информацию см. [в документации по Array MDN .](https://www-construct-net.translate.goog/out?u=https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array&_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru)

## Дата

Объект JavaScript Dateпредоставляет методы для работы с датами и временем.

Имейте в виду, что Dateэто очень старая функция JavaScript, имеющая ряд особенностей. Например, синтаксический анализ строк по датам может работать непредсказуемо и использует сочетание индексов, основанных на 0 и 1. TemporalВ будущем он может быть заменен новым, более спроектированным объектом.

В вычислениях даты и время часто представляются как количество миллисекунд, прошедших с 1 января 1970 года 00:00:00 UTC. Это несколько произвольный выбор, но многие компьютерные системы используют этот метод, поскольку он обеспечивает точность до миллисекунд и совместим на международном уровне, избегая необходимости учитывать географические часовые пояса. Он происходит от старых компьютерных систем на базе UNIX, поэтому дату 1 января 1970 года часто называют «эпохой UNIX». Например, на момент написания статьи 26 ноября 2021 года текущее время составляет 1637930545179 миллисекунд с эпохи UNIX.

При отображении дат в виде строк будут использоваться местный язык и стиль форматирования, а также местный часовой пояс в зависимости от настроек вашей системы. Обратите внимание, что конкретный формат, показанный в примерах ниже, может отличаться от того, что вы видите.

Попробуйте следующие строки кода в консоли браузера.

// Try entering in to the browser console:

// Using just Date() returns a string of the current date and time

Date() // Fri Nov 26 2021 12:38:13 GMT+0000 (Greenwich Mean Time)

// Date.now() provides the current time in milliseconds since the UNIX epoch

Date.now() // 1637930545179

// new Date() creates a new object representing the

// current date. Note while this creates an object,

// the console will show it as a date and time.

new Date() // Fri Nov 26 2021 12:38:13 GMT+0000 (Greenwich Mean Time)

// new Date(time) creates a Date object using the

// time in milliseconds since the UNIX epoch.

new Date(1637930545179) // Fri Nov 26 2021 12:38:13 GMT+0000 (Greenwich Mean Time)

// new Date(year, monthIndex), with optional additional parameters up to

// new Date(year, monthIndex, day, hours, minutes, seconds, milliseconds)

// creates a Date object with the given date and time.

// NOTE: 'monthIndex' is 0-based (0 being January), but 'day' is 1-based.

// Mon Nov 01 2021 00:00:00 GMT+0000 (Greenwich Mean Time)

new Date(2021, 10)

// Tue Nov 23 2021 00:00:00 GMT+0000 (Greenwich Mean Time)

new Date(2021, 10, 23)

// Tue Nov 23 2021 11:53:12 GMT+0000 (Greenwich Mean Time)

new Date(2021, 10, 23, 11, 53, 12)

Вот пример кода, который вы можете попробовать в Construct, который демонстрирует еще несколько методов объектов Dateдля получения дня, месяца и года текущей даты в виде чисел (но обратите внимание, что месяц начинается с нуля, поэтому мы добавляем к нему 1, чтобы начать с январём как 1 вместо 0).

// Get a Date object representing the current time

let currentDate = new Date();

// Log the day, month and year

// NOTE: the month is zero-based (0 is January) so we add 1 to it

console.log(`The day is ${currentDate.getDate()}`);

console.log(`The month is ${currentDate.getMonth() + 1}`);

console.log(`The year is ${currentDate.getFullYear()}`);

В JavaScript вы можете сделать гораздо больше Date. Дополнительную информацию см. [в документации Date MDN .](https://www-construct-net.translate.goog/out?u=https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Date&_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru)

## API браузера

До сих пор все, что мы рассмотрели, является частью самого языка JavaScript, поэтому будет доступно везде, где вы можете написать код JavaScript. Браузеры также предоставляют некоторые дополнительные встроенные функции. Это работает для веб-контента, работающего в браузере, но эти функции не всегда доступны в других средах JavaScript, таких как node.js. Однако, поскольку Интернет сам по себе является важной платформой, стоит упомянуть, какие функции предоставляют браузеры.

Браузеры предоставляют огромный набор встроенных функций, от сетевых технологий до 3D-графики. Это руководство было бы очень длинным, если бы мы попытались охватить их все! Мы кратко рассмотрим лишь очень небольшой набор, просто чтобы познакомить вас с этой областью.

## документ

Объект documentпредставляет собой весь HTML-документ на странице. Это обеспечивает доступ к каждому элементу HTML на странице через JavaScript и позволяет индивидуально манипулировать каждым элементом, например добавлять и удалять элементы, перемещать их или изменять стили CSS. Однако это охватывает широкий набор API, которые мы не будем здесь вдаваться в подробности. На данный момент достаточно подчеркнуть его существование.

Одним из интересных свойств, которые следует изменить, является titleатрибут. Это отражено в заголовке вкладки, поэтому вы можете использовать JavaScript, чтобы установить его по своему усмотрению!

// Change page title to "Hello world!"

document.title = "Hello world!"

Дополнительную информацию см. [на странице «Документ MDN» .](https://www-construct-net.translate.goog/out?u=https://developer.mozilla.org/en-US/docs/Web/API/Document&_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru)

## окно

В браузерах windowна самом деле это просто другое имя для globalThis, то есть глобального объекта. Однако он предоставляет некоторые методы и свойства, которые относятся к окну браузера. При доступе к вещам, связанным с окном браузера, принято использовать имя, windowчтобы прояснить ваши намерения.

Два интересных свойства — это innerWidthи innerHeight, которые возвращают размер страницы в окне браузера в пикселях.

console.log(`Page size is ${window.innerWidth} x ${window.innerHeight}`);

Дополнительную информацию смотрите [на странице Window MDN .](https://www-construct-net.translate.goog/out?u=https://developer.mozilla.org/en-US/docs/Web/API/Window&_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru)

## Прослушивание событий

Мы уже рассказывали об этом ранее, но стоит упомянуть об этом еще раз, поскольку это важная часть API браузера. Этот addEventListenerметод позволяет вам зарегистрировать функцию, которая будет вызываться всякий раз, когда что-то происходит. Например, добавление обработчика события «щелчок» documentбудет вызывать данную функцию всякий раз, когда пользователь нажимает на страницу.

document.addEventListener("click", () =>

{

console.log("Click event!");

});

В общем, шаблон obj.addEventListener(event, func)означает «каждый раз, когда событие происходит в obj , вызывайте func ». Разные объекты имеют разные наборы событий, которые вы можете прослушивать.

Если вы использовали триггеры в таблицах событий Construct, то концепция здесь аналогична. В случае с событиями, когда на объекте срабатывает триггер, выполняются действия. В JavaScript, когда на объекте возникает событие, вызывается предоставленная функция.

Например, windowу объекта есть событие «изменения размера», которое срабатывает каждый раз, когда изменяется размер окна. Попробуйте использовать следующий фрагмент кода в файле сценария, а затем измените размер окна предварительного просмотра, а затем проверьте консоль браузера на наличие полученных сообщений.

window.addEventListener("resize", () =>

{

console.log(`Resize event! New page size is ${window.innerWidth} x ${window.innerHeight}`);

});

Этот шаблон также используется во всех встроенных функциях Construct, о которых мы поговорим позже. Как в документации MDN, так и в документации Construct по JavaScript ссылка на каждый тип объекта также будет включать список событий, которые вы можете прослушивать.

## Таймеры

Метод setIntervalпериодически запускает функцию в интервале времени, заданном в миллисекундах. В следующем примере сообщение будет записываться на консоль каждые 1 секунду с увеличивающимся счетчиком.

let counter = 0;

setInterval(() =>

{

console.log(`setInterval callback, counter = ${counter++}`);

}, 1000);

Метод setTimeoutожидает время, указанное в миллисекундах, а затем запускает функцию. В следующем примере будет выполнен вход в консоль по событию «щелчок», подождите 1 секунду, а затем будет записано другое сообщение. Попробуйте щелкнуть в окне предварительного просмотра и подождите, пока появится второе сообщение.

document.addEventListener("click", () =>

{

console.log("Click event!");

setTimeout(() =>

{

console.log("Callback 1 second after click")

}, 1000);

});

## расположение

Глобальный locationобъект предоставляет информацию об URL-адресе текущей страницы. Вы можете использовать его location.hrefдля получения текущего URL-адреса в виде строки и location.reload()обновления страницы. Попробуйте войти locationв консоль браузера, и вы сможете изучить все предоставляемые им свойства.

## Заключение

В этой части мы рассмотрели:

* Математические методы
* Строковые методы
* Методы массива
* Краткое изложениеDate
* Краткое введение в API браузера, включая documentтаймеры , и прослушивание событий window.location

Эти функции регулярно оказываются полезными в повседневной работе по программированию, поэтому с ними стоит ознакомиться. JavaScript предоставляет гораздо больше встроенных функций, а браузеры — еще больше. Мы затронули лишь несколько наиболее распространенных и полезных встроенных модулей, и в целях простоты перейдем к следующей части, где рассмотрим некоторые API, специфичные для Construct.

# Изучите JavaScript в Construct, часть 11: API Construct

Это 11-я часть серии руководств [«Изучите JavaScript в Construct»](https://www-construct-net.translate.goog/en/courses/learn-javascript-construct-79?_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru) . Эта часть продолжается с части 10. Если вы ее пропустили, см. [«Изучаем JavaScript в Construct», часть 10: Стандартная библиотека](https://www-construct-net.translate.goog/en/tutorials/learn-javascript-construct-2830?_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru) .

В этом руководстве мы рассмотрели некоторые встроенные функции языка программирования JavaScript, также известные как API, а также коснулись некоторых API браузера. Теперь мы рассмотрим некоторые API, которые предоставляет Construct. Эти части специфичны для Construct и поэтому будут либо недоступны, либо будут работать по-другому на других платформах или инструментах, даже если они используют JavaScript. Однако по-прежнему полезно знать, как использовать собственные API Construct, поскольку мы работаем в Construct, и это позволяет нам писать код, использующий функции Construct, такие как спрайты. Многие платформы также предоставляют свои собственные API, поэтому здесь также показано, как начать работу с другим набором API.

## Перемещение спрайта

Одна из самых простых задач, которые мы можем выполнить с помощью JavaScript-кодирования в Construct, — это просто переместить спрайт при нажатии кнопки. Мы вернемся к использованию фрагментов кода в таблицах событий, чтобы это работало. Загрузите проект ниже и откройте его в Construct. Он предоставляет спрайт и кнопку, а также событие нажатия кнопки .

.C3P

#### перемещение-sprite.c3p

[Скачать сейчас62,55 КБ](https://translate.google.com/website?sl=auto&tl=ru&hl=ru&u=https://construct-static.com/tutorials/downloads/69946/move-sprite.c3p)

Откройте вкладку «Лист событий 1» , чтобы увидеть кнопку: «При нажатии триггера». Здесь мы добавим кусок JavaScript вместо действия по перемещению спрайта. Но перед этим давайте объясним некоторые подробности API Construct.

### Время выполнения

При использовании сценариев в таблицах событий Construct автоматически предоставляет две дополнительные встроенные переменные. Это:

* runtime- представляет среду выполнения Construct, то есть общий игровой движок, на котором работает ваш проект. Обычно это основной способ доступа к функциям Construct в JavaScript.
* localVars- представление любых локальных переменных в листе событий, находящихся в области действия. Это позволяет вам получать и устанавливать значения этих переменных из JavaScript, что полезно для передачи значений между листами событий и JavaScript.

Нам пока это не понадобится localVars— мы сосредоточимся на runtime.

Мы можем посмотреть, как это runtimeвыглядит в консоли браузера. Добавьте скрипт к событию On clicked с помощью этого кода:

console.log(runtime);

Это просто записывает встроенную runtimeпеременную в консоль браузера. Просмотрите проект, нажмите кнопку и проверьте консоль. У вас должна быть возможность развернуть его и увидеть всевозможные свойства и методы.

Встроенная runtimeпеременная — это класс с именем IRuntime. Каждое его свойство и метод описано в [разделе «Сценарии» руководства](https://www-construct-net.translate.goog/en/make-games/manuals/construct-3/scripting/overview?_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru) . [Здесь](https://www-construct-net.translate.goog/en/make-games/manuals/construct-3/scripting/scripting-reference/iruntime?_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru) вы можете найти справочник по IRuntime , который охватывает все, что вы видите в консоли. Рекомендуется обратиться к этой документации, чтобы узнать, какие свойства и методы доступны и как их использовать. Однако просмотр вещей в консоли — это также быстрый способ узнать, что это такое и что оно может делать.

### Объекты и экземпляры

Все объекты вашего проекта, то есть все, что находится в разделе « Типы объектов» на панели проекта, представлены в разделе runtime.objects. Например, runtime.objects.Spriteпредставляет тип объекта Sprite на панели проекта.

Важно отметить, что это runtime.objects.Spriteотносится к типу объекта . Это отличается от экземпляра объекта . Если у вас есть 10 экземпляров Sprite, то по-прежнему существует только один тип объекта Sprite, а экземпляров — 10. Тип объекта имеет такие свойства, как имя объекта, а экземпляры имеют такие свойства, как положение и размер. Распространенная ошибка — путать их и пытаться получить доступ к позиции типа объекта. Но у типов объектов нет позиции — она есть только у экземпляров.

Поэтому, если мы хотим получить или изменить положение экземпляра спрайта, мы должны сначала получить экземпляр типа объекта Sprite, который уже находится в макете. Если он только один, самый простой способ сделать это — вызвать getFirstInstance()тип объекта, т. е runtime.objects.Sprite.getFirstInstance(). . Затем возвращается экземпляр с позицией, размером и т. д.

Измените свой код JavaScript в таблице событий на следующее:

let inst = runtime.objects.Sprite.getFirstInstance();

inst.x += 10;

Это позволит получить первый экземпляр типа объекта Sprite (тот, который уже размещен в макете) и сохранить его в переменной inst. Затем он добавляет 10 к координате X экземпляра, что перемещает его вправо. Попробуйте просмотреть проект и нажать кнопку — экземпляр спрайта переместится вправо!

Помните, что такой код runtime.objects.Sprite.x += 10не будет работать: типы объектов не имеют позиции и, следовательно, не имеют свойств xили y. Подходят только экземпляры, поэтому сначала необходимо получить экземпляр, а затем изменить его.

### Документация

Еще раз полезно обратиться к документации и посмотреть, что мы использовали.

runtime.objects.Sprite— это класс с именем [IObjectClass](https://www-construct-net.translate.goog/en/make-games/manuals/construct-3/scripting/scripting-reference/object-interfaces/iobjectclass?_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru) (названный так потому, что он также используется для семейств, т. е. «класс объекта» означает «либо тип объекта, либо семейство»). В связанной документации вы можете найти, что этот метод getFirstInstance()указан и задокументирован, а также различные другие свойства и методы.

Возвращенный экземпляр включает в себя свойства и методы [IWorldInstance](https://www-construct-net.translate.goog/en/make-games/manuals/construct-3/scripting/scripting-reference/object-interfaces/iworldinstance?_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru) (названные так потому, что объекты, появляющиеся в макете, являются частью игрового «мира», тогда как такие вещи, как словарь или объект Array, таковыми не являются). В связанной документации вы можете найти свойства x, yкоторые перечислены и задокументированы, а также различные другие свойства и методы.

Фактически экземпляры Sprite представляют собой классы с именем [ISpriteInstance](https://www-construct-net.translate.goog/en/make-games/manuals/construct-3/scripting/scripting-reference/plugin-interfaces/sprite?_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru) , которые также включают в себя свойства и методы [IWorldInstance](https://www-construct-net.translate.goog/en/make-games/manuals/construct-3/scripting/scripting-reference/object-interfaces/iworldinstance?_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru) и [IInstance](https://www-construct-net.translate.goog/en/make-games/manuals/construct-3/scripting/scripting-reference/object-interfaces/iinstance?_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru) . Это делается с помощью функции JavaScript, известной как наследование. Однако мы еще не рассмотрели это в этом руководстве. На данный момент вы можете думать об этом как о объединении свойств и методов нескольких классов в один. Это полезный способ разделить функции между разными классами — например, все, что появляется в игровом мире, имеет IWorldInstanceсвою позицию.

Мы выделяем документацию, поскольку это важный справочник, который можно использовать при выяснении того, какие функции Construct можно использовать из JavaScript. Редактор часто пытается автоматически заполнять имена по мере ввода, но поскольку JavaScript — динамический язык, он обычно не может определить, какие свойства и методы действительно доступны, и поэтому склонен предлагать все. Чтобы узнать, что действительно доступно и что вы действительно можете использовать, вам необходимо обратиться к документации. После регулярного использования вы начнете помнить, что такое общие свойства и методы, и вам не придется постоянно их искать.

## Использование файла сценария

Предыдущий пример основан на том факте, что runtimeон автоматически предоставляется в сценариях в таблицах событий. В файлах сценариев эта переменная не предоставляется автоматически. Вместо этого есть другой способ доступа к API среды выполнения. Давайте вместо этого воссоздадим то же самое с помощью файла сценария, чтобы продемонстрировать разницу.

Загрузите проект ниже и откройте его в Construct. На этот раз он предоставляет спрайт и кнопку, никаких событий и только шаблон скрипта main.js.

.C3P

#### использование-script-file.c3p

[Скачать сейчас62,64 КБ](https://translate.google.com/website?sl=auto&tl=ru&hl=ru&u=https://construct-static.com/tutorials/downloads/69952/using-script-file.c3p)

Скрипт main.js выглядит следующим образом:

runOnStartup(runtime =>

{

});

runOnStartup— это специальная глобальная функция, которую Construct предоставляет вашему скрипту для доступа к runtimeинтерфейсу. Вы передаете ему функцию, и он вызовет ее, как только интерфейс runtimeстанет доступен, передав ее в качестве первого параметра.

Обычно код в файле сценария запускается сразу после загрузки страницы (что может произойти еще до того, как Construct даже начал загрузку проекта), поэтому runtimeк нему нельзя получить немедленный доступ. Этот метод дает вам возможность получить к нему доступ, как только он станет доступен.

Однако проект все еще загружается при runOnStartupвыполнении обратного вызова. Так что на этом этапе вы еще мало что можете сделать. Лучше всего дождаться события «beforeprojectstart» , которое срабатывает непосредственно перед триггером «При запуске макета» на листе событий, то есть непосредственно перед тем, как первый макет фактически начнет выполняться.

### События времени выполнения

Помните, в предыдущих частях мы использовали следующий шаблон:

obj.addEventListener(event, func)

Это означает: «Когда событие происходит в obj , вызовите func ». API Construct также используют этот шаблон с событиями, специфичными для Construct. Среда выполнения предоставляет несколько событий, которые также описаны в [документации IRuntime](https://www-construct-net.translate.goog/en/make-games/manuals/construct-3/scripting/scripting-reference/iruntime?_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru) , включая «beforeprojectstart» . Таким образом, мы можем вызвать функцию, когда это событие происходит, используя:

runtime.addEventListener("beforeprojectstart", someFunction);

Используйте следующий код для main.js, чтобы прослушивать это событие.

runOnStartup(runtime =>

{

runtime.addEventListener("beforeprojectstart", () => OnBeforeProjectStart(runtime));

});

function OnBeforeProjectStart(runtime)

{

console.log("On before project start");

}

Просмотрите проект, и в консоли должно появиться сообщение «Включить перед запуском проекта».

Обратите внимание, как:

* Мы добавили функцию с именем OnBeforeProjectStart, которая принимает runtimeв качестве параметра.
* Мы используем стрелочную функцию для функции в addEventListener. Это, в свою очередь, просто вызывает OnBeforeProjectStart(runtime), который проходит дальше runtime.

Короче говоря, это означает, что наша OnBeforeProjectStartфункция вызывается, runtimeкогда происходит событие «beforeprojectstart» . Также обратите внимание, что у нас теперь есть короткая последовательность событий: при загрузке страницы runOnStartupбудет выполняться ее обратный вызов; затем он ожидает события «beforeprojectstart» ; когда это срабатывает, он звонит OnBeforeProjectStart. Этот тип шаблона часто встречается при обработке событий в JavaScript, поэтому с ним стоит ознакомиться.

### Событие нажатия кнопки

В OnBeforeProjectStartфункции среда выполнения завершила загрузку, и теперь можно получить доступ к таким вещам, как типы и экземпляры объектов.

Первое, что мы хотим сделать, это определить момент нажатия кнопки. Как и раньше, мы можем получить первый экземпляр объекта Button в макете с помощью:

let buttonInst = runtime.objects.Button.getFirstInstance();

И снова экземпляры используют addEventListenerметод для обработки событий. В [документации IButtonInstance](https://www-construct-net.translate.goog/en/make-games/manuals/construct-3/scripting/scripting-reference/plugin-interfaces/button?_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru) отмечается, что событие «click» срабатывает при нажатии кнопки. Итак, мы можем прослушивать события кликов, как показано ниже (показан весь код main.js):

runOnStartup(runtime =>

{

runtime.addEventListener("beforeprojectstart", () => OnBeforeProjectStart(runtime));

});

function OnBeforeProjectStart(runtime)

{

let buttonInst = runtime.objects.Button.getFirstInstance();

buttonInst.addEventListener("click", () => OnButtonClick(runtime));

}

function OnButtonClick(runtime)

{

console.log("Button clicked!");

}

Просмотрите этот проект, и при нажатии кнопки вы должны увидеть сообщение консоли. На этот раз мы сделали все только с помощью кода JavaScript — лист событий не задействован!

### Перемещение экземпляра спрайта

Последняя часть головоломки — это перемещение экземпляра спрайта в макете. Это можно сделать так же, как и раньше, только используя наш код в OnButtonClickфункции:

function OnButtonClick(runtime)

{

let inst = runtime.objects.Sprite.getFirstInstance();

inst.x += 10;

}

Просмотрите проект, и теперь спрайт будет перемещаться вправо при нажатии кнопки. Мы перестроили то, что было раньше, но с использованием чистого кода.

## Управление с клавиатуры

Давайте реализуем простое движение в четырех направлениях с помощью клавиш со стрелками с помощью кода JavaScript. Загрузите проект ниже и откройте его в Construct. На этот раз он предоставляет только экземпляр спрайта (без кнопки), в проект добавлен объект Keyboard, а в main.js имеется достаточно кода для вызова OnBeforeProjectStart.

.C3P

#### клавиатура-контроли-template.c3p

[Скачать сейчас62,62 КБ](https://translate.google.com/website?sl=auto&tl=ru&hl=ru&u=https://construct-static.com/tutorials/downloads/69955/keyboard-controls-template.c3p)

Реализация управления с клавиатуры означает проверку того, находится ли каждая из четырех клавиш со стрелками вниз, и перемещение спрайта в соответствующем направлении. Это нужно делать неоднократно в каждом кадре, чтобы движение продолжалось. В листах событий Construct действия можно запускать в каждом кадре (также известном как каждый тик , например тиканье часов) с системным условием «Каждый тик» . В коде JavaScript это вместо этого обрабатывается с помощью события времени выполнения «tick» . Это можно сделать так (с полным кодом main.js):

runOnStartup(runtime =>

{

runtime.addEventListener("beforeprojectstart", () => OnBeforeProjectStart(runtime));

});

function OnBeforeProjectStart(runtime)

{

runtime.addEventListener("tick", () => OnTick(runtime));

}

function OnTick(runtime)

{

// this function is now called every tick

}

Обратите внимание, что мы используем тот же шаблон, что и раньше, для обработки событий другого типа — на этот раз вызовы событий «тик»OnTick снова передаются runtimeв качестве параметра.

Теперь любой добавляемый нами код OnTickбудет выполняться каждый такт. Мы можем добавить сюда код, чтобы проверять, какие клавиши со стрелками нажаты, и соответствующим образом перемещать спрайт.

## Обработка ввода с клавиатуры

Обратите внимание, что объект «Клавиатура» добавлен в проект. Это необходимо для приема ввода с клавиатуры.

Доступ к свойствам и методам объекта «Клавиатура» можно получить с помощью runtime.objects.Keyboard, но, поскольку обычно используется ввод с клавиатуры, runtime.keyboardон также предоставляется в виде ярлыка.

Объект «Клавиатура» является глобальным и не имеет экземпляров. Это означает, что доступ к его свойствам и методам осуществляется непосредственно в runtime.keyboard. Нет необходимости звонить getFirstInstance().

Интерфейс [сценария клавиатуры](https://www-construct-net.translate.goog/en/make-games/manuals/construct-3/scripting/scripting-reference/plugin-interfaces/keyboard?_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru) предоставляет метод isKeyDown(), который мы можем использовать для определения того, нажата ли в данный момент клавиша во время вызова. Ключ указывается строкой. Полный список можно найти [на этой странице MDN](https://www-construct-net.translate.goog/out?u=https://developer.mozilla.org/en-US/docs/Web/API/KeyboardEvent/code/code_values&_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru) , но нам нужны клавиши со стрелками, представленные строками "ArrowLeft", "ArrowRight"и ."ArrowUp""ArrowDown"

Таким образом, вызов runtime.keyboard.isKeyDown("ArrowLeft")вернет логическое значение, указывающее, удерживается ли в данный момент клавиша со стрелкой влево. Если это так, то мы можем переместить спрайт влево.

Добавьте следующий код в OnTickфункцию.

function OnTick(runtime)

{

let inst = runtime.objects.Sprite.getFirstInstance();

if (runtime.keyboard.isKeyDown("ArrowLeft"))

inst.x -= 10;

if (runtime.keyboard.isKeyDown("ArrowRight"))

inst.x += 10;

if (runtime.keyboard.isKeyDown("ArrowUp"))

inst.y -= 10;

if (runtime.keyboard.isKeyDown("ArrowDown"))

inst.y += 10;

}

Обратите внимание на четыре оператора «if», которые проверяют, нажата ли каждая из четырех клавиш со стрелками. Если клавиша нажата, позиция экземпляра Sprite перемещается в этом направлении. Просмотрите проект и попробуйте нажать клавиши со стрелками. Спрайт будет двигаться!

### Независимость от частоты кадров

Одна из проблем с перемещением объектов на фиксированное количество пикселей за такт заключается в том, что скорость зависит от частоты кадров. Например, в системе с дисплеем 120 Гц спрайт будет двигаться в два раза быстрее, чем в системе с дисплеем 60 Гц, поскольку на дисплее с частотой 120 Гц тики происходят в два раза быстрее. Более подробно эта проблема объяснена в туториале [Delta-time and Framerate Independent](https://www-construct-net.translate.goog/en/tutorials/delta-time-framerate-2?_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru) .

Решение состоит в том, чтобы вместо этого переместить позицию на дельту-время (время, которое занимает этот кадр в секундах), умноженное на скорость в пикселях в секунду. В JavaScript runtime.dtвозвращается текущая разница времени, которая совпадает с dtвыражением в таблицах событий Construct. Попробуйте следующий код для OnTick.

function OnTick(runtime)

{

let inst = runtime.objects.Sprite.getFirstInstance();

if (runtime.keyboard.isKeyDown("ArrowLeft"))

inst.x -= 400 \* runtime.dt;

if (runtime.keyboard.isKeyDown("ArrowRight"))

inst.x += 400 \* runtime.dt;

if (runtime.keyboard.isKeyDown("ArrowUp"))

inst.y -= 400 \* runtime.dt;

if (runtime.keyboard.isKeyDown("ArrowDown"))

inst.y += 400 \* runtime.dt;

}

Теперь спрайт движется с обычной скоростью 400 пикселей в секунду независимо от частоты кадров, поскольку он перемещается на расстояние 400 \* runtime.dt.

Как отмечается в ранее связанном учебнике, встроенные поведения Construct, такие как 8 Direction, автоматически обрабатывают дельту-время. Однако при выполнении движений с использованием кода JavaScript или блоков событий вам необходимо учитывать разницу во времени, чтобы движение работало одинаково на разных устройствах. Использование дельта-времени также позволяет использовать полезные эффекты, основанные на времени, такие как пауза или замедленное движение, поскольку они работают путем изменения значения dt.

Вы также можете взглянуть на [пример простого движения клавиатуры](https://translate.google.com/website?sl=auto&tl=ru&hl=ru&u=https://editor.construct.net/%23open%3Dsimple-keyboard-movement) , который поставляется с Construct, который также обрабатывает изменение угла спрайта.

## Еще несколько примеров

Давайте кратко рассмотрим еще несколько вещей, которые вы можете сделать с помощью API Construct. Загрузите проект ниже и откройте его в Construct. Это предоставляет шаблон для изменения спрайтов с помощью кода JavaScript — имеется 12 спрайтов, включен объект Mouse, а файл main.js предварительно заполнен кодом для обработки щелчка в любом месте страницы, но он еще ничего не делает, когда происходит щелчок.

.C3P

#### изменить-спрайты-template.c3p

[Скачать сейчас62,92 КБ](https://translate.google.com/website?sl=auto&tl=ru&hl=ru&u=https://construct-static.com/tutorials/downloads/69962/modify-sprites-template.c3p)

Обратите внимание, что этот проект использует событие «click»document для обнаружения щелчка в любом месте страницы.

Давайте попробуем ввести в OnClickфункцию какой-нибудь другой код.

### Установить случайные углы

Метод [IObjectClass](https://www-construct-net.translate.goog/en/make-games/manuals/construct-3/scripting/scripting-reference/object-interfaces/iobjectclass?_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru) можно instances()использовать для перебора всех экземпляров типа объекта. Короче говоря, это означает, что вы можете использовать его с циклом for-of для повторения один раз для каждого экземпляра.

Этот метод является функцией-генератором. Мы не рассмотрели их в этом руководстве, но обратите внимание, что эти типы методов обозначаются \*в документации значком, например \*instances(). По сути, это означает, что метод можно использовать с циклами for-of или с расширенным ...синтаксисом.

Если задать для каждого экземпляра angleDegreesслучайное число от 0 до 360, каждый экземпляр будет указывать в случайном направлении. Попробуйте следующий код для OnClick:

function OnClick(runtime)

{

for (let inst of runtime.objects.Sprite.instances())

{

inst.angleDegrees = Math.random() \* 360;

}

}

Свойство angleработает в радианах. Вместо этого в angleDegreesсобственности используются градусы, что иногда более удобно, и мы использовали здесь.

Предварительный просмотр проекта. Теперь каждый раз, когда вы нажимаете, все свиньи меняют свой ракурс на случайный!

Стоит отметить, что ...оператор распространения также можно использовать для получения массива всего, что будет повторяться с помощью оператора for-of. В приведенном ниже примере показано получение массива из всех 12 экземпляров Sprite и его запись в консоль. Затем он также меняет только последний спрайт на случайный угол, демонстрируя доступ к массиву.

function OnClick(runtime)

{

let arr = [...runtime.objects.Sprite.instances()];

console.log("Array of all instances: ", arr);

// Set last sprite to random angle

arr.at(-1).angleDegrees = Math.random() \* 360;

}

### Выравнивание спрайтов

Следующий код увеличивает позицию на 30 для каждого экземпляра, устанавливая их координаты X и Y в текущее значение, что выстраивает их в диагональную линию.

function OnClick(runtime)

{

let pos = 30;

for (let inst of runtime.objects.Sprite.instances())

{

inst.x = pos;

inst.y = pos;

pos += 30;

}

}

### Случайное смещение

Следующий код смещает положение каждого экземпляра спрайта на случайное число от -10 до 10, заставляя их покачиваться, если вы нажимаете несколько раз.

function OnClick(runtime)

{

for (let inst of runtime.objects.Sprite.instances())

{

inst.x += Math.random() \* 20 - 10;

inst.y += Math.random() \* 20 - 10;

}

}

### Переместить случайный экземпляр на мышь

Следующий код выберет случайный экземпляр одним щелчком мыши и переместит его в позицию мыши.

function OnClick(runtime)

{

// Get array of all Sprite instances

let arr = [...runtime.objects.Sprite.instances()];

// Pick a random instance

let inst = arr[Math.floor(Math.random() \* arr.length)];

// Move that instance to the mouse position (on layer 0)

inst.x = runtime.mouse.getMouseX(0);

inst.y = runtime.mouse.getMouseY(0);

// Also move it to the top of Z order

inst.moveToTop();

}

Обратите внимание, что в этом примере:

* Оператор распространения ...используется для получения массива всех экземпляров.
* Math.random() \* arr.lengthполучает случайное число до длины массива, включая дроби. Нам нужно целое число, поэтому мы используем его Math.floor(Math.random() \* arr.length)для получения случайного индекса массива и выбора случайного экземпляра.
* Аналогично использованию клавиатуры, runtime.mouseобеспечивает доступ [к интерфейсу сценария мыши](https://www-construct-net.translate.goog/en/make-games/manuals/construct-3/scripting/scripting-reference/plugin-interfaces/mouse?_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru) при условии, что объект «Мышь» добавлен в проект. Это предоставляет методы getMouseXи getMouseYметоды для определения положения мыши на слое. В этом случае слой задается как индекс и 0передается как первый (и единственный) слой.

Попробуйте придумать свои собственные способы изменения экземпляров спрайтов!

## Заключение

В этой части мы рассмотрели:

* Использование встроенной runtimeпеременной в скриптах в листах событий
* Доступ к типам и экземплярам объектов черезruntime.objects
* Обращение к ссылкам Construct на доступные API в документации.
* Использование runOnStartupв файле сценария для получения доступа кruntime
* Обработка событий времени выполнения с помощью addEventListener, включая события «beforeprojectstart» и «tick».
* Обработка событий экземпляра, таких как нажатие кнопок
* Использование ввода с клавиатуры для управления спрайтом
* Различные примеры изменения нескольких спрайтов с помощью кода JavaScript.

# Изучите JavaScript в Construct, часть 12: Модули

Это 12-я часть серии руководств [«Изучите JavaScript в Construct»](https://www-construct-net.translate.goog/en/courses/learn-javascript-construct-79?_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru) . Эта часть продолжается с части 11. Если вы ее пропустили, см. раздел [«Изучение JavaScript в Construct», часть 11: API-интерфейсы Construct](https://www-construct-net.translate.goog/en/tutorials/learn-javascript-construct-2833?_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru) .

В этом руководстве мы рассмотрели многие основы языка JavaScript, а также ряд встроенных функций языка, браузера и Construct. Мы подходим к концу этого руководства, и осталось затронуть только одну важную тему: модули. Опять же, мы потенциально могли бы рассказать здесь многое, но мы представим концепцию, рассмотрим основы, а также опишем некоторые аспекты, уникальные для Construct. В частности, поскольку сценарии в таблицах событий — это концепция, уникальная для Construct, существует особый способ справиться с этим.

В более широкой экосистеме JavaScript существует множество различных подходов к работе с модулями. Однако в этом руководстве мы будем иметь дело с системой модулей, встроенной в язык JavaScript, иногда также известной как ES-модули. Большинство других модульных систем появились раньше, и более широкая экосистема JavaScript, вероятно, в долгосрочной перспективе будет двигаться в сторону использования ES-модулей повсеместно.

## Модули

Помните, что переменные и функции верхнего уровня привязаны к файлу. Это означает, что вы можете использовать их в любом месте этого файла сценария, но не в другом файле сценария.

Один из способов использования объектов в файлах — сделать их глобальными, добавив их как свойства globalThis. Однако это имеет ряд проблем. Он быстро становится дезорганизованным, поскольку в одном месте могут оказаться сотни или даже тысячи функций. Кроме того, это может вызвать конфликты имен: если вы хотите использовать две разные библиотеки, и они обе пытаются определить глобальную функцию с одинаковым именем, одна перезапишет другую, и это, вероятно, что-то сломает.

Модули решают эти проблемы, предоставляя понятный и организованный интерфейс для обмена данными между файлами JavaScript. Каждый файл представляет собой автономный модуль — доступ к переменным и функциям верхнего уровня за пределами файла невозможен. Другой способ сказать, что его содержимое является частным для модуля. Если вы хотите предоставить что-то для использования за пределами файла, это необходимо экспортировать с exportключевым словом. Затем другой файл должен импортировать его с importключевым словом.

На данном этапе это может показаться несколько надуманным, поскольку все наши примеры до сих пор включали небольшое количество переменных и функций. Зачем вам нужно распределять их по разным файлам? Для простого кода действительно легко хранить все в одном файле. Однако, когда вы начинаете писать больше кода, его организация в нескольких файлах быстро становится очень важной. В противном случае вы можете получить беспорядочный код, который будет сложно добавлять, изменять или исправлять. Понимание модулей поможет вам поддерживать порядок в коде, когда вы начнете писать большие объемы кода. Это также полезно для повторного использования кода и интеграции библиотек, которые могут работать как модули.

### Простой экспорт

В этом примере давайте воспользуемся новым пустым проектом с двумя файлами сценариев: main.js и mymodule.js . Мы экспортируем что-нибудь из mymodule.js и используем это в main.js. Загрузите и откройте файл проекта ниже для этого примера.

.C3P

#### модули-template.c3p

[Скачать сейчас47,07 КБ](https://translate.google.com/website?sl=auto&tl=ru&hl=ru&u=https://construct-static.com/tutorials/downloads/70360/modules-template.c3p)

Откройте файл **mymodule.js** . Добавьте функцию ниже, которая просто возвращает строку.

function GetMessage()

{

return "Hello world from mymodule.js!";

}

В настоящее время эту функцию нельзя использовать вне этого файла. Однако если мы добавим exportключевое слово, оно станет доступно для импорта другим файлам. Это происходит в начале функции, как показано ниже.

export function GetMessage()

{

return "Hello world from mymodule.js!";

}

Файлы сценариев могут экспортировать столько вещей, сколько захотят. В данном случае мы просто экспортируем одну функцию.

Теперь откройте **main.js.** Мы хотим позвонить GetMessage(), но он пока недоступен. Чтобы импортировать все из mymodule.js , добавьте следующую строку:

import \* as MyModule from "./mymodule.js";

Означает \*«все», as MyModuleозначает помещение всего в объект с именем MyModuleи from "./mymodule.js"указывает, какой файл загружать. В целом это означает «импортировать все в объект с именем MyModuleиз mymodule.js ».

Обратите внимание, что путь к файлу "./mymodule.js"начинается с ./. На самом деле это необходимо и указывает, что это файл (поскольку все можно импортировать из других мест или другими способами).

Теперь все, экспортированное из mymodule.js, находится в объекте с именем MyModule. Таким образом, нашу GetMessage()функцию можно вызвать с помощью MyModule.GetMessage(). Попробуйте следующий код ниже для **main.js** , который записывает сообщение на консоль.

import \* as MyModule from "./mymodule.js";

console.log(MyModule.GetMessage());

Просмотрите проект и проверьте консоль браузера, и вы должны увидеть сообщение!

Поздравляем – у вас первый импорт работает.

## Несколько дополнительных примечаний

Вот еще несколько вещей о том, как работают модули.

### Основной сценарий

Сам Construct будет только загружать и запускать main.js. Если вы выберете файл сценария на панели проектов, вы увидите, что его свойства появятся на панели свойств. К этим свойствам относится Purpose , для которого установлено значение Main script для main.js.

В проекте может быть только один основной скрипт. Все остальные скрипты должны быть импортированы, если они должны быть загружены и использованы, поскольку Construct не будет загружать их автоматически. (Обратите внимание, что цель mymodule.js — « не установлена».)

Основной сценарий также выделен жирным шрифтом на панели проекта, поэтому вы можете легко увидеть, какой из сценариев установлен в качестве основного.

### Повторный импорт одного и того же файла

Если у вас есть несколько файлов, каждый из которых импортирует mymodule.js , он все равно загружает и запускает скрипт только один раз. В первый раз он фактически загрузится, а все последующие разы будут просто повторно использовать первый. Это обеспечивает эффективную работу модулей.

### Импортированное имя

Обратите внимание, что элементы, экспортированные из mymodule.js, были импортированы в объект, который мы назвали MyModule . Это позволяет избежать того, чтобы все они были глобальными функциями или функциями верхнего уровня, что могло бы привести к различным ошибкам, упомянутым ранее. Он сохраняет все организованным и означает, что вы можете легко использовать две функции с одинаковым именем из двух разных модулей.

В программировании термин «пространство имен» относится к месту, где используются имена, например, к верхнему уровню файла сценария или внутри такого объекта, как MyModule . Хранение объектов в разных пространствах имен позволяет избежать таких проблем, как перезапись двух функций с одинаковым именем.

Есть способы импортировать объекты в область верхнего уровня, если хотите, или даже импортировать их под другим именем. Но поскольку вы можете писать операторы импорта как угодно, вы всегда можете контролировать, какие имена будут использоваться для всего, что вы импортируете.

### Импорт и экспорт статичны.

Экспорт может отображаться только на верхнем уровне скрипта. Вы не можете делать такие вещи, как экспортировать что-то внутри оператора «if». Это означает, что модули всегда должны экспортировать один и тот же набор элементов.

Операторы импорта должны располагаться в верхней части сценария перед любыми другими строками кода. Это также означает, что вы не можете использовать оператор импорта внутри оператора if (хотя для этого существует отдельная функция динамического импорта ).

Эти правила предназначены для того, чтобы позволить инструментам выполнять такие действия, как объединение и оптимизация кода. Это также помогает сохранять последовательность и предсказуемость.

## Импорт и экспорт

Вот еще несколько вещей, которые вы можете сделать с ключевыми словами importи export. Это не всеобъемлющее, но полезное резюме.

### Импортировать определенную функцию

Повторно используя наш предыдущий пример, GetMessage()функцию также можно импортировать и использовать следующим образом:

import { GetMessage } from "./mymodule.js";

console.log(GetMessage());

В этом примере import { GetMessage } from "./mymodule.js";это означает импортировать только GetMessageфункцию, экспортированную из mymodule.js . Она также сделана функцией верхнего уровня, поэтому вызывается только в GetMessage()одиночку; нет объекта с именем MyModule.

С помощью формы можно импортировать несколько вещей:import { FuncA, FuncB, FuncC } from "./file.js";

Исходный пример импортировал все и дал этому имя. Этот подход полезен, если вы знаете, что вам нужно импортировать только некоторые конкретные вещи, а не абсолютно все.

### Импортируйте скрипт, чтобы запустить его.

Вы также можете import "./file.js";просто загрузить и запустить файл сценария. Он не будет ничего импортировать и каким-либо образом использовать свой экспорт.

Это может быть полезно для старых библиотек, которые ничего не экспортируют, а вместо этого просто добавляют глобальные переменные. В этом случае после импорта скрипта вы сможете получить доступ ко всему, что он добавил в globalThis. В качестве альтернативы сценарий может сделать что-то полезное, просто запустив свой код верхнего уровня, и в этом случае вам не нужно ничего импортировать, поскольку достаточно просто загрузить и запустить сценарий.

### Экспорт по умолчанию

Если сценарий имеет только один экспорт или имеет основной объект, который вы захотите импортировать большую часть времени, его можно сделать экспортом по умолчанию . Это означает, что он экспортируется export defaultне просто с расширением export. Попробуйте приведенный ниже код для **mymodule.js** .

export default function GetMessage()

{

return "Hello world from mymodule.js!";

}

Экспорт по умолчанию, в свою очередь, импортируется несколько иначе. Попробуйте приведенный ниже код для **main.js.**

import GetMessage from "./mymodule.js";

console.log(GetMessage());

Теперь просмотрите проект, и сообщение будет зарегистрировано.

В данном случае import Thing from "./file.js"это означает импорт экспорта по умолчанию из file.js и присвоение ему имени Thing . Обратите внимание, что здесь не используются фигурные скобки {и }в предыдущем примере (который импортировал экспорт с определенными именами) или части \*или as(которые импортируют все).

Не забывайте, что импортированный таким образом скрипт должен указывать экспорт по умолчанию, иначе вы получите ошибку.

### Импорт доступен только для чтения.

Все, что вы импортируете, доступно только для чтения, поскольку его нельзя переназначить. Это помогает гарантировать, что модули всегда предоставляют согласованный интерфейс.

Однако иногда это может создать некоторую проблему. Например, если вы хотите экспортировать такую ​​переменную:

export let myVariable = 2;

И импортируйте его вот так:

import { myVariable } from "./mymodule.js";

// TypeError: Assignment to constant variable.

myVariable = 3;

...тогда после import, вы не сможете изменить значение myVariable. Несмотря на то, что мы экспортировали переменную, объявленную с помощью let, она все равно действует так, как будто это действительно const.

Одним из способов решения этой проблемы является использование объекта. Сам объект переназначить нельзя, но можно переназначить его свойства. Поэтому мы можем использовать свойства объекта вместо переменных. Вот пример, показывающий, как это можно использовать для хранения групп связанных переменных в модуле. В **mymodule.js** используйте:

export default {

score: 100,

lives: 3,

ammo: 50

};

При этом используется экспорт объекта по умолчанию со свойствами очков, жизней и боеприпасов.

Затем в **main.js** используйте:

import GameVariables from "./mymodule.js";

GameVariables.score += 100;

console.log(`Score: ${GameVariables.score}`);

При этом будет импортирован экспорт по умолчанию с именем GameVariables. Этот объект сам по себе не подлежит переназначению, но это не имеет значения: его свойства можно изменить, поэтому линия GameVariables.score += 100работает так, как ожидалось, в результате чего общий балл равен 200.

Это полезный метод управления наборами связанных переменных как отдельными модулями. Например, вы можете использовать это для «глобальных» переменных, которые на самом деле не будут находиться в глобальной области видимости (поскольку они на самом деле не находятся в глобальном объекте), но обеспечивают более организованный способ совместного использования набора переменных по всему вашему коду. .

### Что можно экспортировать

Из файла сценария можно экспортировать практически все, что угодно. Однако, поскольку все импортируемое доступно только для чтения (т.е. не подлежит переназначению), экспортировать переменные не очень полезно, поскольку они всегда будут работать так, как если бы они были объявлены с помощью const. Обычно модули экспортируют объекты, функции и классы, поскольку это наиболее полезные вещи для совместного использования.

## Импорт для событий

Раньше мы использовали скрипты в таблицах событий. Использование импорта здесь требует еще одного шага. Проблема в том, что код сценариев в таблицах событий фактически запускается внутри функции, а операторы importand exportвнутри функции не допускаются. Поэтому должен быть другой способ использовать импорт сценариев в таблицах событий.

Экспорт из сценариев в листы событий не поддерживается — для этого необходимо использовать файл сценария. Однако поддерживается импорт с использованием специального файла с целью импорта событий . Короче говоря, все, что импортировано в этот файл, доступно всем сценариям в таблицах событий.

Загрузите и откройте проект ниже, чтобы увидеть пример этого.

.C3P

#### импорт-для-событий.c3p

[Скачать сейчас47,58 КБ](https://translate.google.com/website?sl=auto&tl=ru&hl=ru&u=https://construct-static.com/tutorials/downloads/70368/imports-for-events.c3p)

В этом примере мы имеем следующее:

* В main.jsGetMessage экспортируется функция с именем , которая просто возвращает строку с сообщением. (Основной сценарий по-прежнему может экспортировать данные и импортировать их в другое место.)
* Существует еще один файл сценария с именем importsForEvents.js , для свойства которого Purpose установлено значение Imports for Events . Все, что импортировано в этот специальный файл сценария, можно использовать в сценариях в таблицах событий. Этот файл импортируется GetMessageиз main.js. Поэтому GetMessageтеперь его можно использовать во всех сценариях в таблицах событий.
* На листе событий 1 в начале макета запускается сценарий , который регистрируется GetMessage()на консоли. Просмотрите проект, и вы увидите сообщение в консоли.

Вместо вызова функции для получения значения вы также можете использовать функцию для выполнения некоторой работы. Например, если у вас есть длинная функция с большим количеством кода, зачастую удобнее поместить ее в файл сценария, а затем вызывать эту функцию из сценария в листе событий. Это делается точно так же, как и в приведенном выше примере, только функция GetMessageбудет делать что-то другое (и называться как-то еще).

Использование глобальных переменных — еще один способ сделать это, поскольку globalThisони доступны как в файлах сценариев, так и в сценариях в таблицах событий. Но, как отмечалось ранее, использование глобальных переменных имеет несколько подводных камней, поэтому использование модулей таким образом предпочтительнее.

Другой полезный подход — поместить это в importsForEvents.js :

import \* as Main from "./main.js";

Тогда ваши сценарии в листах событий смогут вызывать любую функцию, экспортированную из main.js , например Main.GetMessage(). В более крупных проектах вы можете повторить это для других файлов сценариев, чтобы упростить вызов функций в разных файлах сценариев из сценариев в таблицах событий.

Скрипты в таблицах событий — это уникальная концепция Construct, поэтому такой подход к использованию модулей для предоставления функций является частью собственной разработки Construct. При работе со скриптами в листах событий не всегда достаточно места для ввода (особенно в разделе действий, так как левая часть занята условиями), да и событий зачастую тоже много. Поэтому хороший стиль — писать любые длинные фрагменты кода в виде функций в файлах сценариев и вызывать эти функции в сценариях в таблицах событий посредством импорта событий. Это делает код JavaScript в таблицах событий коротким и простым, а также упрощает вызов других функций и импорт в файле сценария.

## Заключение

В этой части мы рассмотрели:

* Каждый файл сценария представляет собой отдельный модуль, который по умолчанию сохраняет свое содержимое конфиденциальным.
* Использование exportдля обмена чем-либо за пределами файла сценария
* Использование importдля экспорта другого файла сценария.
* Как Construct загружает только основной скрипт, а другие скрипты необходимо импортировать
* Другие способы использования importи export, включая экспорт по умолчанию.
* Как экспорт доступен только для чтения и чем может быть полезно использование свойств объекта вместо переменных для экспорта
* Использование сценария «Импорт для событий» для вызова функции в файле сценария из сценария в листе событий.

## Узнать больше

Мы рассмотрели лишь некоторые из способов import, которые exportможно использовать. Их можно использовать гораздо больше: для импорта и экспорта объектов как со значениями по умолчанию, так и с именованными элементами, используя подмножество элементов, переименовывая элементы и многое другое. Для получения более подробной информации проверьте эти ссылки в веб-документах MDN.

* [экспортное](https://www-construct-net.translate.goog/out?u=https://developer.mozilla.org/en-US/docs/web/javascript/reference/statements/export&_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru) заявление
* [оператор импорта](https://www-construct-net.translate.goog/out?u=https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/import&_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru) (который также охватывает динамический импорт )
* В разделе MDN, посвященном [модулям JavaScript,](https://www-construct-net.translate.goog/out?u=https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Modules&_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru) содержится более подробная информация об использовании модулей.
* В примере кода Ghost [Shooter](https://translate.google.com/website?sl=auto&tl=ru&hl=ru&u=https://editor.construct.net/%23open%3Dghost-shooter-code) показана полностью написанная на JavaScript версия примера Ghost Shooter, в которой используются модули для обмена данными между файлами сценариев.

# Изучите JavaScript в Construct, часть 13: далее

Это заключительная часть 13 серии руководств [«Изучите JavaScript в Construct»](https://www-construct-net.translate.goog/en/courses/learn-javascript-construct-79?_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru) . Эта часть является продолжением части 12. Если вы ее пропустили, см. [«Изучаем JavaScript в Construct», часть 12: Модули](https://www-construct-net.translate.goog/en/tutorials/learn-javascript-construct-2834?_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru) .

В этом руководстве мы рассмотрели большую часть основ языка JavaScript. Однако, как мы отмечали в начале, это руководство не охватывает весь язык полностью. Если бы это было так, руководство было бы гораздо длиннее! JavaScript — это зрелый язык с широким спектром сложных функций, который постоянно совершенствуется с течением времени, а также за счет новых дополнений, улучшений производительности и многого другого.

Очень редко изменения что-либо ломают — производители браузеров не хотят ломать веб-сайты! Таким образом, даже если язык изменится, вам вряд ли когда-нибудь придется возвращаться и менять код JavaScript.

В этой заключительной части руководства есть несколько заключительных замечаний, небольшой совет, некоторые заключительные подробности о JavaScript, которые не были рассмотрены ранее, а также указатели того, куда двигаться дальше. Цель этого руководства — научить вас достаточно, чтобы вы могли комфортно использовать широкий спектр других ресурсов JavaScript в Интернете, узнать больше и научиться применять их в Construct. Мы включили ряд ссылок, которые вы можете использовать больше и посоветовать, на чем сосредоточиться дальше. Однако вы можете многое сделать, так что это всего лишь предложения — если вы знаете, над чем хотите работать, возможно, вы захотите сосредоточиться на других конкретных областях, имеющих к этому отношение.

## Больше руководств

Это руководство познакомило вас с основами языка JavaScript. В Интернете есть и другие руководства, в которых подробно описываются все аспекты JavaScript. Вы можете использовать их, чтобы освежить свои знания, узнать более подробно о темах, которые мы рассмотрели в этом руководстве, а также узнать о новых частях JavaScript, которые не были затронуты в этом руководстве.

Вот несколько руководств, с которыми вы можете ознакомиться:

* [javascript.info](https://www-construct-net.translate.goog/out?u=https://javascript.info/&_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru) предоставляет бесплатное руководство, состоящее из трех основных частей, в которых подробно описывается язык JavaScript, API-интерфейсы браузера, такие как DOM, и дополнительная часть, посвященная дополнительным деталям, таким как работа в сети и регулярные выражения.
* Руководство [MDN Web Docs по JavaScript](https://www-construct-net.translate.goog/out?u=https://developer.mozilla.org/en-US/docs/Learn/JavaScript&_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru) — еще один бесплатный справочник, посвященный веб-разработке. На этом сайте также представлен [справочник по JavaScript](https://www-construct-net.translate.goog/out?u=https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference&_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru) , на который мы ссылались в этом руководстве. Веб-документы MDN являются де-факто официальным справочником по веб-технологиям, таким как JavaScript.
* [«Изучение JavaScript на web.dev»](https://www-construct-net.translate.goog/out?u=https://web.dev/learn/javascript&_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru) — это еще один подробный и бесплатный для чтения курс, выходящий за рамки этого руководства.
* [Learnjavascript.online](https://www-construct-net.translate.goog/out?u=https://learnjavascript.online/&_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru) — это интерактивный онлайн-ресурс для изучения JavaScript, который предоставляет первые несколько глав бесплатно, но обратите внимание, что дальнейшее продвижение требует оплаты.

Поищите в Интернете, и вы найдете множество других ресурсов, как бесплатных, так и платных, таких как [Codecademy.com](https://www-construct-net.translate.goog/out?u=https://www.codecademy.com/&_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru) , [code.org](https://www-construct-net.translate.goog/out?u=https://code.org/&_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru) , [freeCodeCamp.org](https://www-construct-net.translate.goog/out?u=https://www.freecodecamp.org/&_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru) , [Edabit](https://www-construct-net.translate.goog/out?u=https://edabit.com/challenges/javascript&_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru) и другие.

Чтобы узнать больше о том, как JavaScript работает в Construct, также обратитесь к [разделу документации по сценариям](https://www-construct-net.translate.goog/en/make-games/manuals/construct-3/scripting/overview?_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru) . Руководства по [Javascript в Construct](https://www-construct-net.translate.goog/en/make-games/manuals/construct-3/scripting/using-scripting/javascript-construct?_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru) , [сценариям в таблицах событий](https://www-construct-net.translate.goog/en/make-games/manuals/construct-3/scripting/using-scripting/scripts-in-event-sheets?_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru) и [файлам сценариев](https://www-construct-net.translate.goog/en/make-games/manuals/construct-3/scripting/using-scripting/script-files?_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru) содержат важную информацию о том, как конкретно используется JavaScript в Construct. Также имеется руководство по [сценариям отладки](https://www-construct-net.translate.goog/en/make-games/manuals/construct-3/scripting/using-scripting/debugging-script?_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru) , которое поможет вам начать работу с некоторыми более продвинутыми инструментами отладки, предоставляемыми инструментами разработчика браузера.

## Что изучать дальше

Вот несколько идей о том, что следует изучить в дальнейшем, если вы хотите расширить свои знания JavaScript. Они примерно соответствуют тому, что мы предлагаем изучить дальше, но вы, конечно, можете делать все по-своему или следовать порядку, предложенному другими руководствами.

* [Классы](https://www-construct-net.translate.goog/out?u=https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes&_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru) являются фундаментальной частью программирования на JavaScript и предоставляют широкий спектр возможностей объектно-ориентированного программирования (ООП). Если вы не уверены, куда идти дальше, лучше всего узнать больше о классах и обо всем, что вы можете с ними делать. **Наследование** также является ключевой частью ООП, которую вы можете использовать в классах с помощью ключевого слова [Extensions](https://www-construct-net.translate.goog/out?u=https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes/extends&_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru) . Если вы хотите углубиться во все технические подробности того, как работает наследование в JavaScript, см. [Наследование и цепочка прототипов](https://www-construct-net.translate.goog/out?u=https://developer.mozilla.org/en-US/docs/Web/JavaScript/Inheritance_and_the_prototype_chain&_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru) .
* Есть и другие **операторы** , которые вы можете использовать, например побитовые [~](https://www-construct-net.translate.goog/out?u=https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Bitwise_NOT&_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru) , [&](https://www-construct-net.translate.goog/out?u=https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Bitwise_AND&_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru) , [|](https://www-construct-net.translate.goog/out?u=https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Bitwise_OR&_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru) , [^](https://www-construct-net.translate.goog/out?u=https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Bitwise_XOR&_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru) ; битовые сдвиги [<<](https://www-construct-net.translate.goog/out?u=https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Left_shift&_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru) и [>>](https://www-construct-net.translate.goog/out?u=https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Right_shift&_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru) ; и нулевое слияние [??](https://www-construct-net.translate.goog/out?u=https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Nullish_coalescing_operator&_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru) .
* [JSON](https://www-construct-net.translate.goog/out?u=https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/JSON&_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru) — это полезный формат данных, основанный на том, как литералы объектов пишутся в JavaScript. Это также полезно в Construct с [плагином JSON](https://www-construct-net.translate.goog/en/make-games/manuals/construct-3/plugin-reference/json?_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru) . См. [JSON.parse()](https://www-construct-net.translate.goog/out?u=https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/JSON/parse&_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru) и [JSON.stringify()](https://www-construct-net.translate.goog/out?u=https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/JSON/stringify&_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru) .
* **Исключения** — полезный способ обработки ошибок. См. [try...catch](https://www-construct-net.translate.goog/out?u=https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/try...catch&_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru) и [поток управления и обработка ошибок](https://www-construct-net.translate.goog/out?u=https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Control_flow_and_error_handling&_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru) .
* [асинхронные функции](https://www-construct-net.translate.goog/out?u=https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function&_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru) , оператор [await](https://www-construct-net.translate.goog/out?u=https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/await&_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru) и [промисы](https://www-construct-net.translate.goog/out?u=https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise&_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru) полезны для решения асинхронных (т. е. длительных) задач, таких как загрузка ресурсов из сети.
* [Регулярные выражения](https://www-construct-net.translate.goog/out?u=https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions&_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru) предоставляют полезные и расширенные инструменты для обработки текста. Construct также позволяет использовать их в таблицах событий с некоторыми системными выражениями.
* Браузеры предоставляют множество [веб-API](https://www-construct-net.translate.goog/out?u=https://developer.mozilla.org/en-US/docs/Web/API&_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru) , таких как [выборка](https://www-construct-net.translate.goog/out?u=https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API&_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru) для выполнения сетевых запросов, [WebSockets](https://www-construct-net.translate.goog/out?u=https://developer.mozilla.org/en-US/docs/Web/API/WebSocket&_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru) для связи в реальном времени, [Web Workers](https://www-construct-net.translate.goog/out?u=https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API&_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru) для запуска кода в фоновом потоке, [Web Crypto](https://www-construct-net.translate.goog/out?u=https://developer.mozilla.org/en-US/docs/Web/API/Web_Crypto_API&_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru) для различных криптографических инструментов, [Web Audio](https://www-construct-net.translate.goog/out?u=https://developer.mozilla.org/en-US/docs/Web/API/Web_Audio_API&_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru) для воспроизведения звука и многое другое. .
* Существует множество полезных **шаблонов кодирования,** о которых стоит узнать, например, [выражения с немедленным вызовом функций](https://www-construct-net.translate.goog/out?u=https://developer.mozilla.org/en-US/docs/Glossary/IIFE&_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru) (IIFE), [каррирование](https://www-construct-net.translate.goog/out?u=https://javascript.info/currying-partials&_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru) , расширенные методы функций, такие как [bind()](https://www-construct-net.translate.goog/out?u=https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_objects/Function/bind&_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru) , [call()](https://www-construct-net.translate.goog/out?u=https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/call&_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru) и [apply()](https://www-construct-net.translate.goog/out?u=https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/apply&_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru) , и многое другое.
* [TypeScript](https://www-construct-net.translate.goog/out?u=https://www.typescriptlang.org/&_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru) расширяет JavaScript за счет статической типизации, предоставляя полезные инструменты, такие как улучшенное автозаполнение. Дополнительную информацию см. [в разделе «Использование TypeScript в Construct» .](https://www-construct-net.translate.goog/en/tutorials/using-typescript-construct-3003?_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru)

Отладка кода также полезна, чтобы научиться диагностировать код, который работает неправильно. Он позволяет вам выполнять такие действия, как пошаговое выполнение кода, показывая значения всех переменных, чтобы вы могли точно видеть, что он делает и где что-то идет не так. Руководство [по сценариям отладки](https://www-construct-net.translate.goog/en/make-games/manuals/construct-3/scripting/using-scripting/debugging-script?_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru) в руководстве Construct показывает, как с этим начать.

## Примеры стартовой страницы

В разделе «Сценарии» на стартовой странице вы найдете несколько примеров, основанных на коде JavaScript: от базовых примеров до полностью написанной на JavaScript версии примера Ghost Shooter. Их изучение поможет вам узнать больше о JavaScript и о том, как его использовать в Construct.

## Получать помощь

Что касается общих вопросов по программированию, [StackOverflow.com](https://www-construct-net.translate.goog/out?u=https://stackoverflow.com/&_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru) — это известный ресурс, где можно задавать вопросы и получать помощь, а также бесценный ресурс прошлых вопросов и ответов, где вы часто можете найти кого-то еще, у кого была такая же проблема и ответ, который решил их. проблема.

Если у вас возникла проблема, связанная с кодированием в Construct, например использование сценариев в таблицах событий или использование собственных API Construct, попробуйте задать вопрос на [форуме сценариев](https://www-construct-net.translate.goog/en/forum/construct-3/scripting-51?_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru) .

## Шансы и концы

Прежде чем мы закончим, следует упомянуть еще несколько подробностей о JavaScript, которые не вошли ни в один предыдущий раздел или были пропущены для простоты.

### Предпочитаю const вместо let

В этом руководстве мы обычно letобъявляли переменные. Однако рекомендуется использовать его constдля любой переменной, которая не переназначается. В приведенном ниже коде показан пример того, где вы можете constпредпочесть let.

function getSomething()

{

// 'x' is never reassigned, so prefer 'const'

// over 'let' for it.

const x = getValue();

return x \* 2;

}

Обратите внимание: это не означает, xчто значение всегда одно и то же — это просто означает, что оно никогда не меняется после инициализации. Это может быть полезно знать при чтении более сложных функций.

Примеры JavaScript на стартовой странице Construct написаны с использованием предпочтения const, а редактор скриптов в Construct предложит изменить объявления переменных с letна const, если они никогда не переназначаются.

### делать пока

Оператор [do... while](https://www-construct-net.translate.goog/out?u=https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/do...while&_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru) — это довольно редко используемый вариант стандартного whileцикла, который всегда выполняет свои операторы хотя бы один раз. Иногда это полезный способ написания цикла, и о нем стоит знать.

### фор-ин

В этом руководстве рассматриваются циклы for-of, но есть и другой вид, использующий это inключевое слово, который чаще всего используется в старом коде. Его можно использовать для итерации свойств объекта. Дополнительную информацию см [. в](https://www-construct-net.translate.goog/out?u=https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for...in&_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru) разделе «Веб-документы MDN». Однако в современном коде предпочтительнее использовать форму «для-из» с [Object.keys()](https://www-construct-net.translate.goog/out?u=https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/keys&_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru) , [Object.values()](https://www-construct-net.translate.goog/out?u=https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_objects/Object/values&_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru) или [Object.entries(),](https://www-construct-net.translate.goog/out?u=https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/entries&_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru) если вы хотите перебирать свойства объекта, значения или и то, и другое.

### Устаревшие функции

В JavaScript есть различные старые функции, которые, по сути, больше не следует использовать, но вы можете столкнуться с ними, если работаете с существующими частями старого кода. К ним относятся:

* [var](https://www-construct-net.translate.goog/out?u=https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/var&_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru) для объявления переменных, у которого есть некоторые необычные и довольно сложные особенности, поэтому letсейчас он предпочтительнее.
* До появления классов написание объектно-ориентированного кода на JavaScript обычно включало изменение прототипа функций. Это также может оказаться полезным для занятий в некоторых более редких или сложных случаях. Подробнее об этом можно узнать в разделе [«Наследование и цепочка прототипов»](https://www-construct-net.translate.goog/out?u=https://developer.mozilla.org/en-US/docs/Web/JavaScript/Inheritance_and_the_prototype_chain&_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru) .
* Старый код JavaScript иногда также работает в «небрежном режиме», то есть в нестрогом режиме. В Construct весь код всегда выполняется в строгом режиме, поэтому принцип работы «небрежного режима» не имеет отношения к коду, написанному в Construct. Но еще раз, если вам нужно работать со старым кодом и поэтому нужно знать, как он работает, различия отмечены в руководстве MDN по [строгому режиму](https://www-construct-net.translate.goog/out?u=https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode&_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru) .
* Construct всегда использует модули. Немодульные скрипты, также известные как «классические» скрипты, работают немного по-другому. Опять же, это не влияет на Construct, поскольку он всегда использует модульные скрипты, но классические скрипты имеют некоторые небольшие различия, в частности, переменные верхнего уровня являются глобальными, а не привязанными к файлу, а также невозможность использования importи exportоператоров (но вы можете использовать динамический импорт). ).

### Причуды

Как и большинство языков программирования, JavaScript имеет некоторые особенности — необычные вещи, которые работают странно или неожиданным образом. Их лучше избегать. Обычно это всего лишь неясности, случайности или остатки плохих проектных решений многолетней давности, которые сейчас слишком сложно изменить. Не пишите код, который на них полагается. Однако иногда о них нужно знать, хотя бы для того, чтобы держаться подальше.

#### Преобразования типов

При преобразовании типов можно обнаружить множество странных результатов. Например:

// Convert empty array to number

Number([]) // 0

// Convert empty object to number

Number({}) // NaN

// Many operators automatically convert

// to numbers, producing weird results like:

1 / [] // Infinity (1 / 0)

1 + "1" // "11" (string)

true + true // 2 (1 + 1)

// Operators can also automatically convert

// to strings, with weird results like:

[1] + 1 // "11" (string)

[1, 2] + [3, 4] // "1,23,4" ("1,2" + "3,4")

Избегайте написания кода, который использует это! Убедитесь, что любые преобразования типов являются явными и, следовательно, преднамеренными.

#### Нестрогое равенство

Оператор нестрогого равенства ==разрешен для преобразования типов. Это может привести к странным результатам, что является веской причиной предпочесть ===вместо этого строгое равенство. Ниже приведены некоторые примеры неожиданных преобразований типов.

0 == "0" // true

0 == false // true

0 == [] // true

0 == [0] // true

"1" == [1] // true

"" == false // true

null == undefined // true

В каждом приведенном выше случае использование строгого равенства ===возвращает falseвместо true, как вам, вероятно, хотелось бы, поскольку в каждом случае типы разные. Поэтому избегайте использования нестрогого равенства и не полагайтесь на преобразования типов, поскольку они могут делать неожиданные вещи.

#### Индексы массива

Если вы получите доступ к массиву с дробным индексом, он вернет undefined.

["😀", "👾"][0] // "😀"

["😀", "👾"][0.5] // undefined

Причиной этого является очень странная часть JavaScript, где технически элементы массива представляют собой строковые свойства, представляющие собой строку числа. При доступе arr[0.5]фактически осуществляется доступ к свойству с именем «0.5», т. е. к свойству arr["0.5"], которое не существует в массиве; доступ к несуществующему свойству возвращает undefined.

Многие другие языки программирования избегают этого, используя для индексов массива числа, которые автоматически округляются. Однако, поскольку в JavaScript этого не происходит, мы должны убедиться, что индексы массива всегда являются целыми числами. Использование Math.floor()индекса массива — хороший способ сделать это. Например, чтобы получить доступ к случайному элементу массива, используйте arr[Math.floor(Math.random() \* arr.length)], поскольку Math.random()возвращает дробное число, поэтому его необходимо округлить до целого числа.

#### Больше особенностей

Вероятно, вы столкнетесь с другими особенностями или удивительными вещами в JavaScript, когда будете больше с ним работать. Не забывайте стараться избегать неожиданных частей JavaScript — лучше всего писать понятный код, а использование непонятных или случайных функций JavaScript усложняет понимание вашего кода.

## Чем блоки событий отличаются от кода

Прежде чем мы закончим, стоит упомянуть, как код JavaScript сравнивается с блоками событий для тех из вас, кто раньше работал с листами событий Construct.

Блоки событий — это принципиально другая парадигма по сравнению с текстовыми языками, такими как JavaScript. Это означает, что зачастую не существует короткого и удобного кода JavaScript, эквивалентного блоку событий. Это связано с тем, что блоки событий представляют собой концепцию гораздо более высокого уровня, чем код. Казалось бы, простой блок событий может на самом деле включать в себя выполнение большого количества сложного кода для выполнения чего-то, что в таблице событий кажется простым. В этом вся цель блоков событий: заботясь о стольких деталях, они делают создание игр намного быстрее, проще и доступнее для людей, вообще не имеющих опыта программирования.

Если вы только начинаете работать с таблицами событий, вам часто придется использовать другой подход, если вы хотите сделать то же самое в коде JavaScript, к которому может потребоваться некоторое привыкание. В качестве альтернативы вы можете применить смешанный подход, при котором для одних целей вы используете блоки событий, а для других — код JavaScript. Возможность добавлять код JavaScript в листы событий специально разработана, чтобы упростить использование этого подхода «смешивай и сопоставляй».

Вот два примера, демонстрирующие концептуальную разницу между блоками событий и кодом.

### Сбор

Условия в блоках событий «выбирают» экземпляры, соответствующие условию. Дальнейшие условия отфильтровывают выбранные экземпляры, оставляя только те, которые соответствуют всем условиям. Затем действия выполняются на выбранных экземплярах.

Эта концепция «выбора» уникальна для листов событий — в JavaScript нет встроенной концепции выбора. Та же концепция может быть применена в JavaScript, например, путем создания массива экземпляров и постепенной его фильтрации, а затем итерации массива, выполняющей некоторые действия над оставшимися экземплярами. Однако код JavaScript обычно не пишется таким образом.

Более типичным шаблоном для использования в JavaScript было бы перебор всех экземпляров с помощью цикла for-of. Например, рассмотрим блок событий, который проверяет Sprite: X < 500 с действием Sprite: Set X to Self.X + 100 \* dt . Это переместит все спрайты вправо, пока они не преодолеют линию X = 500. Эквивалент JavaScript будет выглядеть примерно так:

function Tick(runtime)

{

for (const inst of runtime.objects.Sprite.instances())

{

if (inst.x < 500)

{

inst.x += 100 \* runtime.dt;

}

}

}

Обратите внимание, что здесь не используется процесс выбора — код повторяет каждый экземпляр Sprite, а внутри цикла он использует оператор if, чтобы проверить, следует ли перемещать спрайт.

### При столкновении

Событие On Collision на самом деле выполняет довольно большую работу. Перекрытие всегда истинно, пока экземпляры пересекаются. Однако при столкновении запоминается, пересекались ли экземпляры ранее, и это значение истинно только при первом их пересечении, что действует как триггер при первом соприкосновении двух экземпляров.

В JavaScript API Construct есть только метод [testOverlap](https://www-construct-net.translate.goog/en/make-games/manuals/construct-3/scripting/scripting-reference/object-interfaces/iworldinstance?_x_tr_sl=auto&_x_tr_tl=ru&_x_tr_hl=ru) , который по сути является эквивалентом Is overlapping . Эквивалента «При столкновении» нет : часть, запоминающая предыдущее состояние, является частью системы событий.

Иногда достаточно перекрытия при тестировании, например, если объекты разрушаются при касании, поскольку они не будут продолжать соприкасаться. Однако при кодировании на JavaScript ваш код должен помнить, перекрывались ли ранее два экземпляра, и делать что-то другое, когда это происходит в первый раз. Это показывает, как листы событий выполняют за вас много дополнительной работы, делая вашу жизнь проще, чем эквивалент на языке программирования. Часто задачи, простые в таблицах событий, требуют гораздо больше работы над кодом.

## Несколько советов

Вот несколько дополнительных советов по программированию, которые помогут вам в этом.

* Программирование часто требует терпения. Когда что-то не получается, постарайтесь проявить любопытство и узнать, что происходит. Хотя это вполне объяснимая реакция, постарайтесь не злиться и не расстраиваться — это только усложнит решение проблемы. Если вам нужно, выделите немного времени. Часто для решения проблемы достаточно сделать перерыв или вернуться на следующий день.
* Если вы серьезно настроены научиться хорошо программировать, постарайтесь много практиковаться. Написание большого количества разного кода для решения разных задач — отличный способ обучения.
* JavaScript — старый язык, первоначально созданный в 1995 году. Некоторые старые его части были заменены более новыми и лучшими функциями до такой степени, что вообще больше нет причин использовать старые функции. Хорошим примером этого является letзамена varобъявления переменных. В сети полно информации о JavaScript, но при поиске ресурсов проверяйте дату публикации — чем старше он, тем больше вероятность, что теперь он будет написан по-другому. Также нет необходимости изучать, как работают старые устаревшие функции, если только вам не придется работать со старым кодом, который их использует.
* JavaScript также является постоянно совершенствующимся языком, регулярно получающим новые функции. Однако новые функции не всегда появляются во всех браузерах или платформах одновременно. Проверьте поддержку новых функций браузером, чтобы случайно не написать код, который не работает в других браузерах. Тестирование вашего кода работает во всех основных браузерных движках (Chrome, Firefox и Safari) также является хорошим способом проверить это.
* Приятно ощущать чувство выполненного долга, когда пишешь сложный и работающий код. Однако помните, что для любой конкретной задачи лучший код должен быть настолько простым и легким для чтения, насколько это возможно, и при этом выполнять свою работу. Хорошие программисты по возможности избегают непонятных трюков или ненужной сложности.

## Заключительные замечания

Мы надеемся, что это руководство дало вам прочные знания языка программирования JavaScript, а также указало на дополнительные ресурсы для углубления вашего понимания JavaScript. Construct предоставляет отличное место для обучения кодированию, позволяя смешивать и сопоставлять блоки событий и код JavaScript, чтобы облегчить переход к кодированию, а также предоставляет редактор и API для решения многих утомительных задач программирования игр, таких как дизайн уровней. и рендеринг.

Язык программирования JavaScript и все API, доступные в браузерах и таких инструментах, как Construct, представляют собой огромный объем знаний, о котором стоит узнать. Программирование также часто является трудным и может занять много времени, чтобы сделать ваш код правильным. Завершение этого руководства — это только самое начало вашего путешествия в мир программирования. Если вы серьезно относитесь к программированию, признайте, что, как и во многих других областях работы, вероятно, потребуется год или больше регулярной практики, чтобы стать профессионалом, и еще много лет, чтобы стать экспертом. Однако даже обладая базовыми навыками, вы можете делать полезные и интересные вещи — и вы будете изучать навыки программирования на стандартном языке программирования, что позволяет легко перенести полученные знания в другие инструменты и платформы на основе JavaScript. А если пойти дальше, JavaScript — это тот язык, с которым можно получить работу!

Поздравляем с завершением этого руководства, удачи в дальнейших усилиях и удачного программирования! 🎉