Основы JavaScript. Hello, World!

По традиции начнем с написания программы «Hello, World!». Эта программа будет выводить на экран текст. Чтобы вывести что-то на экран, нужно дать компьютеру специальную команду. В языке JavaScript такая команда — console.log():

console.log('Hello, World!');

*// => Hello, World!*

Иногда для удобства мы будем показывать в комментариях результат запуска строчек кода, вот так: => РЕЗУЛЬТАТ. Например, // => 4.

Комментарии

Кроме собственного кода в файлах с исходным кодом могут находиться комментарии. Это текст, который не является частью программы и нужен программистам для пометок. С их помощью добавляют пояснения, как работает код, какие здесь ошибки нужно поправить или не забыть что-то добавить позже.

*// Удалить строку ниже после реализации задачи по регистрации*

console.log(10);

Комментарии в JavaScript бывают двух видов.

*Однострочные комментарии* начинаются с //. После этих двух символов может следовать любой текст, вся строка не будет анализироваться и исполняться. Комментарий может занимать всю строку. Если одной строки мало, то создаются несколько комментариев:

*// For Winterfell!*

*// For Lanisters!*

Комментарий может находиться на строке после какого-нибудь кода:

console.log('I am the King'); *// For Lannisters!*

*Многострочные комментарии* начинаются с /\* и заканчиваются \*/.

*/\**

*The night is dark and*

*full of terrors.*

*\*/*

console.log('I am the King');

<https://replit.com/@hexlet/helloworld>

Такие комментарии обычно используют для документирования кода, например, функций.

# 2. Инструкции—Основы JavaScript

Инструкция — это команда для компьютера выполнить что-то. Код на JavaScript — это набор инструкций, которые, как правило, отделяются друг от друга символом ;.



Вот пример кода с двумя инструкциями.

console.log('Mother of Dragons.');

console.log('Dracarys!');

*// => Mother of Dragons.*

*// => Dracarys!*

<https://replit.com/@hexlet/js-basics-instructions>

Теоретически инструкции можно написать друг за другом без переноса на новую строку:

console.log('Mother of Dragons.'); console.log('Drakarys!');

Результат на экране будет таким же, но такой код неудобен для чтения, поэтому инструкции располагают друг под другом.

Почему это важно знать? Инструкция — это единица исполнения. Интерпретатор (программа, которая запускает код на JavaScript), выполняет инструкции строго по очереди. И мы, как разработчики, должны понимать этот порядок и уметь мысленно разделять программу на независимые части, удобные для анализа.

# 3. Арифметические операции—Основы JavaScript

На базовом уровне компьютеры оперируют только числами. Даже в прикладных программах на высокоуровневых языках внутри много чисел и операций над ними. К счастью, для старта достаточно знать обычную арифметику — с нее и начнем.

Для сложения двух чисел в математике мы пишем, например, 3 + 4. В программировании — то же самое. Вот программа, складывающая два числа:

*// Не забываем точку с запятой в конце,*

*// так как каждая строчка в коде – инструкция*

3 **+** 4;

Инструкция 3 + 4; заставит компьютер сложить числа и узнать результат. Если запустить эту программу, то ничего не произойдет. А если быть точными, то компьютер вычислит сумму, но на этом все. Результат сложения никак не используется, и такая программа не представляет никакого интереса. Нам нужно попросить компьютер сложить 3 + 4 **и** дать команду сделать что-то с результатом. Например, вывести его на экран:

*// Сначала вычисляется сумма,*

*// затем она передается в функцию печати*

console.log(3 **+** 4); *// => '7'*

Всегда отбивайте арифметические операторы пробелами от самих чисел (операндов) – это хороший стиль программирования. Поэтому в наших примерах console.log(3 + 4), а не console.log(3+4).

Кроме сложения, доступны следующие операции:

* \* — умножение
* / — деление
* - — вычитание
* % — [остаток от деления](https://ru.wikipedia.org/wiki/%D0%94%D0%B5%D0%BB%D0%B5%D0%BD%D0%B8%D0%B5_%D1%81_%D0%BE%D1%81%D1%82%D0%B0%D1%82%D0%BA%D0%BE%D0%BC)
* \*\* — возведение в степень

Теперь давайте выведем на экран результат деления, а потом результат возведения в степень:

console.log(8 **/** 2); *// => 4*

console.log(3 **\*\*** 2); *// => 9*

Первая инструкция выведет на экран 4 (потому что 8 / 2 это 4), а вторая инструкция выведет на экран 9 (потому что 32 это 9).

Операторы

Перед тем, как двигаться дальше, разберем базовую терминологию. Знак операции, такой как +, называют **оператором**. Операторы выполняют операции над определенными значениями, которые называются **операндами**. Сами операторы обычно представлены одним или несколькими символами. Реже словом. Подавляющее большинство операторов соответствуют математическим операциям.

console.log(8 **+** 2);

В этом примере + — это **оператор**, а числа 8 и 2 — это **операнды**.

В случае сложения у нас есть два операнда: один слева, другой справа от знака +. Операции, которые требуют наличия двух операндов, называются **бинарными**. Если пропустить хотя бы один операнд, например, 3 + ;, то программа завершится с синтаксической ошибкой.

Операции (не операторы) бывают не только бинарными, но и унарными (с одним операндом), и даже тернарными (с тремя операндами)! Причем операторы могут выглядеть одинаково, но обозначать разные операции.

console.log(**-**3); *// => -3*

Выше пример применения унарной операции к числу 3. Оператор минус перед тройкой говорит интерпретатору взять число 3 и найти противоположное, то есть -3.

Это немного может сбить с толку, потому что -3 — это одновременно и число само по себе, и оператор с операндом, но у языков программирования такая структура.

Коммутативная операция

Мы все помним со школы: «от перемены мест слагаемых сумма не меняется». Это один из базовых и интуитивно понятных законов арифметики, он называется **коммутативным законом**.

Бинарная операция считается коммутативной, если поменяв местами операнды, вы получаете тот же самый результат. Очевидно, что сложение — коммутативная операция: *3 + 2 = 2 + 3*.

А вот является ли коммутативной операция вычитания? Конечно, нет: *2 - 3 ≠ 3 - 2*. В программировании этот закон работает точно так же, как в арифметике.

Более того, большинство операций, с которыми мы будем сталкиваться в реальной жизни, не являются коммутативными. Отсюда вывод: всегда обращайте внимание на порядок того, с чем работаете.

Композиция операций

А что, если понадобится вычислить такое выражение: 3 \* 5 - 2? Именно так мы и запишем:

console.log(3 **\*** 5 **-** 2); *// => 13*

Обратите внимание, что интерпретатор производит арифметические вычисления в правильном порядке: сначала деление и умножение, потом сложение и вычитание. Иногда этот порядок нужно изменить — об этом поговорим дальше. Или другой пример:

console.log(2 **\*** 4 **\*** 5 **\*** 10);

Как видно, операции можно соединять друг с другом, получая возможность вычислять все более сложные составные выражения. Чтобы представить себе то, как происходят вычисления внутри интерпретатора, давайте разберем пример: 2 \* 4 \* 5 \* 10.

1. Сначала вычисляется 2 \* 4 и получается выражение 8 \* 5 \* 10.
2. Затем 8 \* 5. В итоге имеем 40 \* 10.
3. В конце концов происходит последнее умножение, и получается результат 400.

Таким образом, интерпретатор соединяет сложные составные выражения, последовательно выполняя заложенные в них арифметические действия, по умолчанию соблюдая правильный порядок: сначала умножение и деление, затем — сложение и вычитание.

Приоритет операций

Посмотрите внимательно на выражение 2 + 2 \* 2 и посчитайте в уме ответ.

Правильный ответ: 6. Если у вас получилось 8, то этот раздел для вас. В школьной математике мы изучали понятие «приоритет операции». Приоритет определяет то, в какой последовательности должны выполняться операции. Например, умножение и деление имеют больший приоритет, чем сложение и вычитание, а приоритет возведения в степень выше всех остальных арифметических операций: 2 \*\* 3 \* 2 вычислится в 16.

Но нередко вычисления должны происходить в порядке, отличном от стандартного приоритета. В сложных ситуациях приоритет можно (и нужно) задавать круглыми скобками, точно так же, как в школе, например: (2 + 2) \* 2.

Скобки можно ставить вокруг любой операции. Они могут вкладываться друг в друга сколько угодно раз. Вот пара примеров:

console.log(3 **\*\*** (4 **-** 2)); *// => 9*

console.log(7 **\*** 3 **+** (4 **/** 2) **-** (8 **+** (2 **-** 1))); *// => 14*

Иногда выражение сложно воспринимать визуально. Тогда можно расставить скобки, не повлияв на приоритет. Например:

Было:

console.log(8 **/** 2 **+** 5 **-** **-**3 **/** 2); *// => 10.5*

Стало:

console.log(((8 **/** 2) **+** 5) **-** (**-**3 **/** 2)); *// => 10.5*

<https://replit.com/@hexlet/js-basics-arithmetics>

Запомните: код пишется для людей, потому что код будут читать люди, а машины будут только исполнять его. Для машин код — или корректный, или не корректный, для них нет «более» понятного или «менее» понятного кода.

# 4. Ошибки оформления (синтаксиса и линтера)—Основы JavaScript

Если программа на JavaScript написана синтаксически некорректно, то интерпретатор выводит на экран соответствующее сообщение, а также указание на файл и строчку в нем, где, по его мнению, произошла ошибка. *Синтаксическая ошибка* возникает в том случае, когда код был записан с нарушением грамматических правил. В человеческих языках грамматика важна, но текст с ошибками чаще всего можно понять и прочитать. В программировании все строго. Любое мельчайшее нарушение, и программа даже не запустится. Примером может быть забытая ;, неправильно расставленные скобки и другие детали.

Вот пример кода с синтаксической ошибкой:

console.log('Hodor'

Если запустить код выше, то мы увидим следующее сообщение: SyntaxError: missing ) after argument list, а также указание на строку и файл, где возникла эта ошибка. Подобные синтаксические ошибки в JavaScript относятся к разряду SyntaxError.

С одной стороны, ошибки SyntaxError — самые простые, потому что они связаны исключительно с грамматическими правилами написания кода, а не с самим смыслом кода. Их легко исправить: нужно лишь найти нарушение в записи.

С другой стороны, интерпретатор не всегда может четко указать на это нарушение. Поэтому бывает, что забытую скобку нужно поставить не туда, куда указывает сообщение об ошибке.

Ошибки линтера

Теперь, когда мы уже научились писать простые программы, можно немного поговорить о том, как их писать.

Код программы следует оформлять определенным образом, чтобы он был достаточно понятным и простым в поддержке. Специальные наборы правил — стандарты — описывают различные аспекты написания кода. Таких стандартов несколько, самые известные в JavaScript: [AirBnb](https://github.com/airbnb/javascript" \t "_blank), [Standard](https://github.com/standard/eslint-config-standard), [Google](https://github.com/google/eslint-config-google). В уроках мы будем придерживаться AirBnb.

В любом языке программирования существуют утилиты — так называемые **линтеры**. Они проверяют код на соответствие стандартам. В JavaScript это [eslint](https://eslint.org/" \t "_blank).

Взгляните на пример из предыдущего урока:

console.log(8**/**2**+**5 **-** **-**3 **/** 2); *// => 10.5*

Линтер будет «ругаться» на нарушение сразу нескольких правил:

* [space-infix-ops](https://eslint.org/docs/rules/space-infix-ops) – Отсутствие пробелов между оператором и операндами.
* [no-mixed-operators](https://eslint.org/docs/rules/no-mixed-operators) – По стандарту нельзя писать код, в котором разные операции используются в одном выражении без явного разделения скобками.

В прошлом уроке мы сами признали, что такое обилие цифр и символов запутывает, и решили добавить скобки исключительно для удобства чтения:

console.log(((8 **/** 2) **+** 5) **-** (**-**3 **/** 2)); *// => 10.5*

Этот вариант уже не нарушает правил, и линтер будет «молчать».

Рассмотрим еще один пример:

console.log(70 **\*** (3 **+** 4) **/** (8 **+** 2));

Есть ли здесь нарушение стандарта?

К сожалению, да. На этот раз операции \* и / находятся в одном выражении без разделения скобками. Вы можете решить эту проблему, добавив дополнительные скобки. Но в какой-то момент количество скобок может быть уже настолько большим, что код снова станет неудобным и непонятным. В этот момент разумнее будет разделить выражение на отдельные части. Мы научимся это делать в следующих уроках.

[no-mixed-operators](https://eslint.org/docs/rules/no-mixed-operators) — лишь одно из большого количества правил. Другие правила описывают отступы, названия создаваемых сущностей, скобки, математические операции, длину строк и множество иных аспектов. Каждое отдельное правило кажется довольно мелким, не очень важным. Но вместе они составляют основу хорошего кода.

На Хекслете линтер начинает проверять код и сообщать о нарушениях после оформления подписки.

# 5. Строки—Основы JavaScript

Какие из этих пяти вариантов — строки?

'Hello'

'Goodbye'

'G'

' '

''

Кавычки

С первыми двумя все понятно, это точно строки, мы уже работали с подобными конструкциями и говорили, что строки – это наборы символов.

Любой одиночный символ в кавычках — это строка. Пустая строка '' — это тоже строка. То есть строкой мы считаем все, что находится внутри кавычек, даже если это пробел, один символ или вообще отсутствие символов.

Ранее в уроках мы записывали строки в одинарных кавычках, но это не единственный способ. Можно использовать и двойные:

*// Стандарт кодирования airbnb, рекомендует*

*// использовать, по возможности, одинарные*

console.log('Dracarys!');

Представьте, что вы хотите напечатать строчку *Dragon's mother*. Апостроф перед буквой **s** — это такой же символ, как одинарная кавычка. Попробуем:

console.log('Dragon's mother');

// Uncaught SyntaxError: missing ) after argument list

Такая программа не будет работать. С точки зрения JavaScript, строка началась с одинарной кавычки, а потом закончилась после буквы **n**. Дальше были символы s mother без кавычек — значит, это не строка. А потом была одна открывающая строку кавычка, которая так и не закрылась: ');. Этот код синтаксически некорректен (это видно даже по тому, как подсвечен код).

Здесь нам помогут двойные кавычки. Такой вариант программы отработает корректно:

console.log("Dragon's mother");

Теперь интерпретатор знает, что строка началась с двойной кавычки — значит, и закончиться должна на двойной кавычке. А одинарная кавычка внутри стала частью строки.

Верно и обратное. Если внутри строки мы хотим использовать двойные кавычки, то саму строку надо делать в одинарных. Причем количество кавычек внутри самой строки не важно.

А что, если мы хотим создать такую строку:

Dragon's mother said "No"

В ней есть и одинарные и двойные кавычки. Как быть в этой ситуации? Нужно каким-то образом сказать интерпретатору считать каждую кавычку частью строки, а не началом или концом строки.

Для этого **экранируют** специальные символы. В нашем случае тот символ, который является признаком конца и начала строки, это либо одинарная кавычка, либо двойная, в зависимости от ситуации. Для экранирования используется обратный слеш \.

*// Экранируется только ", так как в этой ситуации*

*// двойные кавычки имеют специальное значение*

console.log("Dragon's mother said \"No\"");

*// => Dragon's mother said "No"*

Посмотрите внимательно: нам нужно было добавить \ для двойных кавычек, но не для одинарной (апостроф), потому что сама строка создана с двойными кавычками. Если бы строка создавалась с одинарными кавычками, то символ экранирования нужен был бы перед апострофом, но не перед двойными кавычками.

*// \ не выводится, если после него идет обычный,*

*// а не специальный символ*

console.log("Death is \so terribly final");

*// => Death is so terribly final*

А что, если нужно вывести сам обратный слеш? Точно так же, как и любой другой специальный символ, его надо экранировать самим собой.

console.log("\\");

*// => \*

Вопрос на самопроверку, что выведет этот код?

console.log("\\ \\ \\\\ \\\ \'\"");

Экранирующие последовательности

Мы хотим показать диалог Матери Драконов со своим ребенком:

- Are you hungry?

- Aaaarrrgh!

Если вывести на экран строку с таким текстом:

console.log('- Are you hungry?- Aaaarrrgh!');

то получится так:

- Are you hungry?- Aaaarrrgh!

Не то, что мы хотели. Строки расположены друг за другом, а не одна ниже другой. Нам нужно как-то сказать интерпретатору «нажать на энтер» — сделать перевод строки после вопросительного знака. Это можно сделать, используя символ перевода строки: \n.

console.log('- Are you hungry?\n- Aaaarrrgh!');

результат:

- Are you hungry?

- Aaaarrrgh!

\n — это специальный символ. В литературе его часто обозначают как *LF* (Line Feed). Возможно вы сейчас подумали, что это опечатка, ведь здесь мы видим два символа \ и n, но это не так. С точки зрения компьютера — это один невидимый символ перевода строки. Доказательство:

*// Мы это не изучали, но вы должны знать правду*

*// Ниже код, который возвращает длину строки*

'a'.length; *// 1*

'\n'.length; *// 1 !!!*

'\n\n'.length; *// 2 !!!*

Почему так сделано? \n — всего лишь способ записать символ перевода строки, но сам перевод строки по своему смыслу – это один символ, правда, невидимый. Именно поэтому и возникла такая задача. Нужно было как-то представить его на клавиатуре. А поскольку количество знаков на клавиатуре ограничено и отдано под самые важные, то все специальные символы реализуются в виде таких обозначений.

Символ перевода строки не является чем-то специфичным для программирования. Все, кто хоть раз печатал на компьютере, использовал перевод строки, нажимая на Enter. Во многих редакторах есть опция, позволяющая включить отображение невидимых символов — с ее помощью можно понять, где они находятся (хотя это всего лишь схематичное отображение, у этих символов нет графического представления, они невидимые):

- Привет!¶

- О, привет!¶

- Как дела?

Устройство, которое выводит соответствующий текст, учитывает этот символ. Например, принтер при встрече с LF протаскивает бумагу вверх на одну строку, а текстовый редактор переносит весь последующий текст ниже, также на одну строку.

\n — это пример **экранирующей последовательности** (escape sequence). Их еще называют управляющими конструкциями. Хотя таких символов не один десяток, в программировании часто встречаются всего несколько. Кроме перевода строки, к таким символам относятся табуляция (разрыв, получаемый при нажатии на кнопку Tab) и возврат каретки (только в Windows). Нам, программистам, часто нужно использовать перевод строки \n для правильного форматирования текста.

console.log('Gregor Clegane\nDunsen\nPolliver\nChiswyck');

На экран выведется:

Gregor Clegane

Dunsen

Polliver

Chiswyck

Обратите внимание на следующие моменты:

1. Не имеет значения, что стоит перед или после \n: символ или пустая строка. Перевод будет обнаружен и выполнен в любом случае.
2. Помните, что строка может содержать один символ или вообще ноль символов. А еще строка может содержать только \n. Проанализируйте следующий пример:
3. console.log('\n');
4. console.log('Dunsen');
5. *// =>*
6. *// =>*
7. *// => Dunsen*

Здесь мы сначала выводим строку «перевод строки», а потом делаем вывод обыкновенной строки.

Почему перед строкой *Dunsen* появилось две пустые строки, а не одна? Дело в том, что console.log() при выводе значения автоматически добавляет в конец символ перевода строки. Таким образом, один перевод строки мы указали явно, передав этот символ экранирующей последовательности аргументом в функцию, а второй перевод строки добавлен самой функцией автоматически.

Еще пример кода:

console.log('Polliver');

console.log('Gregor Clegane');

console.log();

console.log('Chiswyck');

console.log('\n');

console.log('Dunsen');

*// => Polliver*

*// => Gregor Clegane*

*// =>*

*// => Chiswyck*

*// =>*

*// =>*

*// => Dunsen*

Сейчас у вас достаточно знаний, чтобы самостоятельно разобраться и понять, почему вывод сформировался именно таким образом.

1. Если нам понадобится вывести \n именно как текст (два отдельных печатных символа), то можно воспользоваться уже известным нам способом экранирования, добавив еще один \ в начале. То есть последовательность \\n отобразится как символы \ и n, идущие друг за другом.
2. console.log('Joffrey loves using \\n');
3. *// => Joffrey loves using \n*

Небольшое, но важное замечание про Windows. В Windows для перевода строк по умолчанию используется \r\n. Такая комбинация хорошо работает только в Windows, но создает проблемы при переносе в другие системы (например, когда в команде разработчиков есть пользователи как Windows, так и Linux). Дело в том, что последовательность \r\n имеет разную трактовку в зависимости от выбранной кодировки (рассматривается позже). По этой причине, в среде разработчиков принято всегда использовать \n без \r, так как LF всегда трактуется одинаково и отлично работает в любой системе. Не забудьте настроить ваш редактор на использование \n.

Конкатенация

В веб-разработке программы постоянно оперируют строками. Все, что мы видим на сайтах, так или иначе представлено в виде текста. Этот текст чаще всего динамический, то есть полученный из разных частей, которые соединяются вместе. Операция соединения строк в программировании называется **конкатенацией**.

*// Оператор такой же, как и при сложении чисел*

*// но здесь он имеет другой смысл (семантику)*

console.log('Dragon' **+** 'stone');

*// => Dragonstone*

Склеивание строк всегда происходит в том же порядке, в котором записаны операнды. Левый операнд становится левой частью строки, а правый — правой.

Вот еще несколько примеров:

console.log('Kings' **+** 'wood'); *// => Kingswood*

*// Обратный порядок слов*

console.log('road' **+** 'Kings'); *// => roadKings*

*// Конкатенировать можно абсолютно любые строки*

console.log("King's" **+** 'Landing'); *// => King'sLanding*

Как видите, строки можно склеивать, даже если они записаны с разными кавычками.

В последнем примере название города получилось с ошибкой: *King's Landing* нужно писать через пробел. Но в наших начальных строках не было пробелов, а пробелы в самом коде слева и справа от символа + не имеют значения, потому что они не являются частью строк.

Выхода из этой ситуации два:*// Оба способа равнозначны*

*// Ставим пробел в левой части*

console.log("King's " **+** 'Landing'); *// => King's Landing*

*// Ставим пробел в правой части*

console.log("King's" **+** ' Landing'); *// => King's Landing*

Пробел — такой же символ, как и другие. Чем больше пробелов, тем шире отступы:

console.log("King's " **+** ' Landing'); *// => King's Landing*

console.log("King's " **+** ' Landing'); *// => King's Landing*

# 6. Переменные—Основы JavaScript

Представьте себе задачу: нам нужно напечатать на экран фразу *Father!* два раза или даже пять раз. Эту задачу можно решить в лоб:

console.log('Father!'); *// => Father!*

console.log('Father!'); *// => Father!*

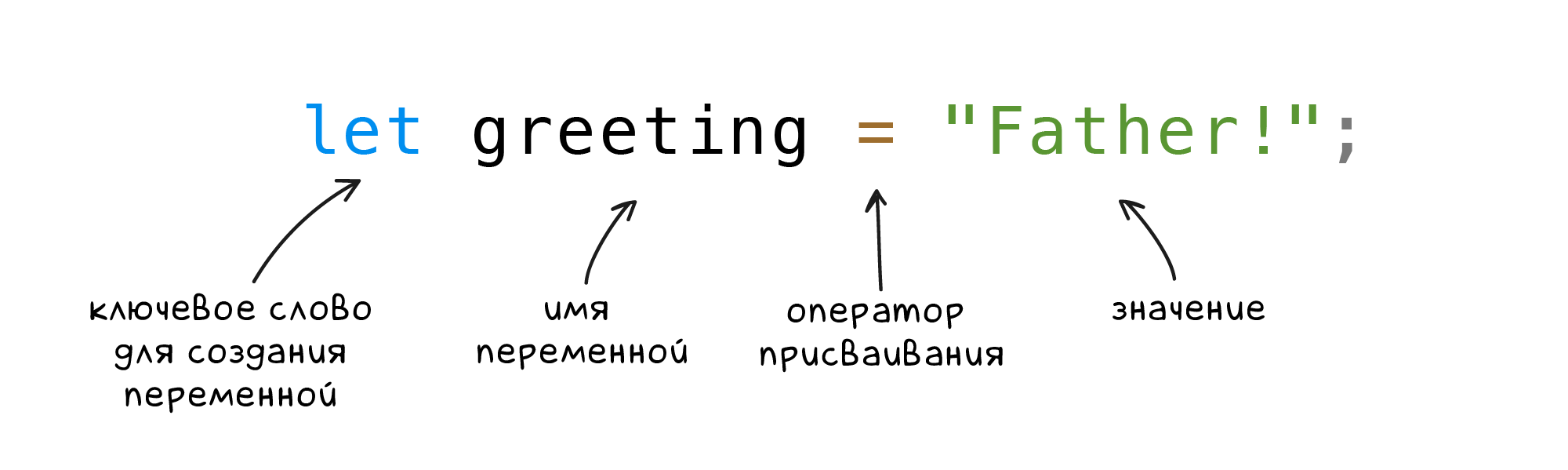
В простейшем случае так и стоит поступить, но если фраза *Father!* начнет использоваться чаще, да еще и в разных частях программы, то придется ее везде повторять. Проблемы с таким подходом начнутся тогда, когда понадобится изменить нашу фразу, а такое происходит довольно часто. Нам придется найти все места, где использовалась фраза *Father!*, и выполнить необходимую замену. А можно поступить по-другому. Вместо копирования нашего выражения достаточно создать (объявить) переменную с этой фразой.

*// greeting – переводится как приветствие*

**let** greeting **=** 'Father!';

console.log(greeting); *// => Father!*

console.log(greeting); *// => Father!*



Переменная указывает на данные, которые были в нее записаны. Благодаря этому, данные можно использовать многократно без необходимости их постоянно дублировать. Сама переменная создается и наполняется данными (инициализируется) с помощью инструкции let greeting = 'Father!'.

Для имени переменной используется любой набор допустимых символов, к которым относятся буквы английского алфавита, цифры, а также знаки *\_* и *$*. При этом цифру нельзя ставить в начале. Имена переменных регистрозависимы, то есть имя hello и имя heLLo – это два разных имени, а значит и две переменные. Регистр в JavaScript имеет важное значение, никогда не забывайте про него.

Переменную не обязательно инициализировать данными во время объявления. Иногда бывает нужно ее создать, а наполняться она будет потом:

**let** greeting;

*// Использование*

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

*// Изменение переменной в следующем уроке*

Объявленная, но не инициализированная переменная, содержит внутри себя значение undefined. Это специальное значение, используемое тогда, когда ничего не определено.

Количество создаваемых переменных ничем не ограничено, большие программы содержат десятки и сотни тысяч имен переменных:

**let** greeting1 **=** 'Father!';

console.log(greeting1); *// => Father!*

console.log(greeting1); *// => Father!*

**let** greeting2 **=** 'Mother!';

console.log(greeting2); *// => Mother!*

console.log(greeting2); *// => Mother!*

Для удобства анализа программы, переменные принято создавать как можно ближе к тому месту, где они используются.

Изменение переменной

Само слово «переменная» говорит, что ее можно менять. И действительно, с течением времени внутри программы значения переменных могут изменяться.

**let** greeting **=** 'Father!';

console.log(greeting); *// => Father!*

console.log(greeting); *// => Father!*

greeting **=** 'Mother!';

console.log(greeting); *// => Mother!*

console.log(greeting); *// => Mother!*

Имя осталось тем же, но внутри другие данные. Обратите внимание на ключевое различие между объявлением переменной и ее изменением. Ключевое слово let ставится только при создании переменной, но при изменении оно уже не используется.

Ошибки при работе с переменными

Порядок следования инструкций в коде с переменными имеет огромное значение. Переменная должна быть определена до того, как будет использована. Ниже пример ошибки, которую очень часто допускают новички:

*// Uncaught ReferenceError: greeting is not defined*

console.log(greeting);

**let** greeting **=** 'Father!';

Запуск программы с примера выше завершается ошибкой *ReferenceError: greeting is not defined*. *ReferenceError* – это ошибка обращения, она означает, что в коде используется имя (говорят идентификатор), которое не определено. Причем в самой ошибке об этом говорят прямо: *greeting is not defined*, что переводится как *greeting не определен*. Кроме неправильного порядка определения, в JavaScript встречаются банальные опечатки — как при использовании переменной, так и при ее объявлении.

Количество подобных ошибок уменьшается за счет использования правильно настроенного редактора. Такой редактор подсвечивает имена, которые используются без объявления, и предупреждает о возможных проблемах.

Еще одна распространенная ошибка — попытаться объявить уже объявленную переменную:

**let** greeting **=** 'Father!';

**let** greeting **=** 'Father!';

Так делать нельзя. Придется создать новую переменную.

Константы

Во всем модуле подавляющее большинство примеров кода использовало переменные в качестве имен (псевдонимы) конкретных значений, а не как переменные, которые меняют свое значение со временем.

**let** dollarsInEuro **=** 1.25;

**let** rublesInDollar **=** 60;

**let** dollarsCount **=** 50 **\*** dollarsInEuro; *// 62.5*

**let** rublesCount **=** dollarsCount **\*** rublesInDollar; *// 3750*

console.log(rublesCount); *// => 3750*

В программировании принято называть такие имена константами, и многие языки поддерживают константы как конструкцию. JavaScript, как раз, относится к таким языкам, и его стандарты кодирования [говорят прямо](https://eslint.org/docs/rules/prefer-const) — если значение не меняется, то мы имеем дело с константой. Перепишем пример выше с использованием констант:

**const** dollarsInEuro **=** 1.25;

**const** rublesInDollar **=** 60;

**const** euros **=** 1000;

**const** dollars **=** euros **\*** dollarsInEuro; *// 1250*

**const** rubles **=** dollars **\*** rublesInDollar; *// 75000*

console.log(rubles); *// => 75000*

<https://replit.com/@hexlet/js-basics-variables>

Единственное изменение заключается в том, что ключевое слово let заменилось на const, но это только синтаксис. Теперь, если попытаться изменить любую константу, то мы получим сообщение об ошибке. В остальном они используются точно так же, как и переменные.

**const** pi **=** 3.14;

pi **=** 5; *// TypeError: Assignment to constant variable.*

Зачем такие сложности? Почему бы не оставить только переменные? Даже если бы мы оставили только переменные, то это не отменяет того факта, что они часто использовались бы как константы, более того, код на JavaScript можно и идиоматично писать без использования переменных вообще. Посмотрите на пример из [реального кода Хекслета](https://github.com/Hexlet/hexlet-exercise-kit/blob/main/import-documentation/index.js). На текущем этапе вы его вряд ли поймете, но попробуйте посчитать количество констант и переменных внутри него, вы увидите, что здесь ровно одна переменная, и целая куча констант.

Константы значительно проще для анализа, когда мы видим константу в коде, то нам сразу понятно, что ее значение всегда остается прежним. При использовании констант отсутствует понятие времени. С переменными все не так, мы не можем быть уверены в их значении, приходится анализировать весь код, чтобы понять, как они могли измениться. Переменные жизненно необходимы только в одном случае (во всех остальных гарантированно можно обойтись без них) – при работе с циклами, до которых мы еще дойдем. В дальнейшем мы будем предпочитать константы и использовать переменные только тогда, когда без них никак.

# 7. Выражения в определениях—Основы JavaScript

Переменные полезны не только для хранения и переиспользования информации, но и для упрощения сложных вычислений. Давайте рассмотрим пример: нужно перевести евро в рубли через доллары. Подобные конвертации через промежуточную валюту часто делают банки при покупках за рубежом.

Для начала переведем 50 евро в доллары. Допустим, что один евро — 1.25 долларов:

**let** dollarsCount **=** 50 **\*** 1.25;

console.log(dollarsCount); *// => 62.5*

В предыдущем уроке мы записывали в переменную конкретное значение. А здесь let dollarsCount = 50 \* 1.25; справа от знака равно находится **выражение**. Интерпретатор вычислит результат — 62.5 — и запишет его в переменную. С точки зрения интерпретатора не важно, что перед ним: 62.5 или 50 \* 1.25, для него оба варианта — выражения, которые надо вычислить. И они вычисляются в одно и то же— 62.5.

Любая строка — выражение. Конкатенация строк — тоже выражение. Когда интерпретатор видит выражение, он обрабатывает его и генерирует результат — **значение выражения**. Вот несколько примеров выражений, а в комментариях справа от каждого выражения — итоговое значение:

62.5 *// 62.5*

50 **\*** 1.25 *// 62.5*

120 **/** 10 **\*** 2 *// 24*

'hello' *// hello*

'Good' **+** 'will' *// Goodwill*

Правила построения кода (грамматика языка) таковы, что в тех местах, где ожидается выражение, можно поставить любое вычисление (не только математическое, но и, например, строковое — как конкатенация), и программа останется работоспособной. По этой причине невозможно описать и показать все случаи использования всех операций.

Программы состоят из множества комбинаций выражений, и понимание этой концепции — один из ключевых шагов на вашем пути.

Основываясь на сказанном выше, подумайте, сработает ли такой код?

**let** who **=** "dragon's" **+** 'mother';

console.log(who);

Запустите его на [repl.it](https://repl.it/languages/javascript) и поэкспериментируйте.

Вернемся к нашей валютной программе. Запишем стоимость доллара в рублях как отдельную переменную. Вычислим цену 50 евро в долларах, умножив их на 1.25. Допустим, что 1 доллар — 60 рублей:

**let** rublesPerDollar **=** 60;

**let** dollarsCount **=** 50 **\*** 1.25; *// 62.5*

**let** rublesCount **=** dollarsCount **\*** rublesPerDollar; *// 3750*

console.log(rublesCount);

А теперь давайте добавим к выводу текст с помощью конкатенации:

**let** rublesPerDollar **=** 60;

**let** dollarsCount **=** 50 **\*** 1.25; *// 62.5*

**let** rublesCount **=** dollarsCount **\*** rublesPerDollar; *// 3750*

console.log('The price is ' **+** rublesCount **+** ' rubles');

*// => The price is 3750 rubles*

Любая переменная может быть частью любого выражения. В момент вычисления, вместо имени переменной подставляется ее значение.

Интерпретатор вычисляет значение rublesCount до того, как эта переменная начнет использоваться в других выражениях. Когда подходит момент использования переменной, Javascript «знает» значение, потому что уже вычислил его.

Переменные и конкатенация

Для закрепления предыдущей темы, попробуем использовать переменные с конкатенацией. Синтаксически ничего не меняется: мы умеем конкатенировать (склеивать) две строки:

**let** what **=** 'Kings' **+** 'road';

console.log(what); *// => Kingsroad*

… а значит сумеем конкатенировать строку и одну переменную, в которой записана строка:

**let** first **=** 'Kings';

**let** what **=** first **+** 'road';

console.log(what); *// => Kingsroad*

… и даже конкатенировать две переменные, в которых записаны строки:

**let** first **=** 'Kings';

**let** last **=** 'road';

**let** what **=** first **+** last;

console.log(what); *// => Kingsroad*

<https://replit.com/@hexlet/variables-expression>

# 8. Именование—Основы JavaScript

Представим себе, что программа из прошлого урока выглядит так:

**let** x **=** 'Father!';

console.log(x);

console.log(x);

Она по-прежнему работает, но в ней изменилось имя переменной на x. Компьютеру без разницы, как мы называем переменные, это бездушная машина, но вот программистам — нет. Мы гораздо чаще читаем код, чем пишем. Причем не свой, а написанный другими людьми. От качества и понятности имен переменных зависит половина успеха в анализе кода.

Лучше посидеть и придумать название, которое описывает суть, смысл переменной, чем назвать ее как попало, а в будущем переделывать. Постарайтесь давать им такие имена, чтобы они были максимально понятны без контекста, без изучения окружающего кода.

Существует общепринятое правило: не используйте транслит для имен, только английский язык. Если вы испытываете сложности с английским, то пользуйтесь переводчиком. Со временем, копаясь в чужом коде, вы сформируете правильные понятия для именования.

Среди разработчиков есть шутка: «самое сложное в программировании — названия переменных и инвалидация кеша». Придумывать названия и правда сложно. Как бы вы назвали переменную, в которой хранится *количество неоплаченных заказов от клиентов, имеющих задолженность в предыдущем квартале?*

Самопроверка. Придумайте название для переменной, в которой будет храниться *«количество братьев и сестёр короля»*. Запишите его в блокноте или отправьте себе на почту. Не указывайте там ничего, кроме названия переменной. А через несколько уроков мы вернёмся к этой теме ;-)

Стили именования

greeting — пример простого имени, но не все имена так просты. Довольно часто они составные, то есть включают в себя несколько слов. Например, «имя пользователя». В разных языках применяются разные стили кодирования, и имя переменной будет отличаться.

В именовании переменных можно выделить четыре основных подхода, которые иногда комбинируют друг с другом. Все эти подходы проявляют себя, когда имя переменной состоит из нескольких слов:

* kebab-case — составные части переменной разделяются дефисом. Например: my-super-var.
* snake\_case — для разделения используется подчеркивание. Например: my\_super\_var.
* CamelCase — каждое слово в переменной пишется с заглавной буквы. Например: MySuperVar.
* lowerCamelCase — каждое слово в переменной пишется с заглавной буквы, кроме первого. Например: mySuperVar.

В Javascript используется CamelCase и его вариация lowerCamelCase, при котором первая буква первого слова — строчная. Именно lowerCamelCase применяется для переменных. Это значит, что имена соединяются друг с другом, при этом все имена кроме первого становятся с заглавной буквы: userName. С тремя словами это выглядит так: mySuperVariable.

Магические числа

Вспомним один из прошлых уроков:

**let** dollarsCount **=** 50 **\*** 1.25; *// 62.5*

**let** rublesCount **=** dollarsCount **\*** 60; *// 3750*

console.log(rublesCount);

С точки зрения профессиональной разработки, такой код «пахнет». Так описывают код, который сложен для понимания. И причина здесь вот в чем: уже сейчас, глядя на число 60 и 1.25, можно задаться вопросом: «что это за числа?». А представьте, что будет через месяц! А как его поймет новый программист, не видевший код ранее? В нашем примере контекст восстанавливается благодаря грамотному именованию, но в реальной жизни код значительно сложнее, и догадаться до смысла чисел зачастую невозможно.

Этот «запах» называют Magic Numbers (магические числа). Числа, происхождение которых невозможно понять без глубокого знания происходящего внутри данного участка кода.

Выход из ситуации прост: достаточно создать переменные с правильными именами, как все встанет на свои места.

**let** dollarsPerEuro **=** 1.25;

**let** rublesPerDollar **=** 60;

**let** dollarsCount **=** 50 **\*** dollarsPerEuro; *// 62.5*

**let** rublesCount **=** dollarsCount **\*** rublesPerDollar; *// 3750*

console.log(rublesCount);

Обратите внимание на следующие детали:

* Именование lowerCamelCase.
* Две новые переменные отделены от последующих вычислений пустой строчкой. Эти переменные имеют смысл и без вычислений, поэтому такое отделение уместно, оно повышает читаемость.
* Получился хорошо именованный и структурированный код, но он длиннее прошлой версии. Так часто бывает, и это нормально. Код должен быть читабельным.

# 9. Интерполяция—Основы JavaScript

Представим, что перед нами стоит задача создать заголовок письма из двух констант и знаков препинания. Скорее всего, мы бы решили задачу так:

**const** firstName **=** 'Joffrey';

**const** greeting **=** 'Hello';

console.log(greeting **+** ', ' **+** firstName **+** '!');

*// => Hello, Joffrey!*

Это довольно простой случай, но даже здесь нужно приложить усилия, чтобы увидеть, какая в итоге получится строка. Нужно следить за несколькими кавычками и пробелами, и без вглядывания не понять, где что начинается и кончается.

Есть другой, более удобный и изящный способ решения той же задачи — **интерполяция**. Вот, как это выглядит:

**const** firstName **=** 'Joffrey';

**const** greeting **=** 'Hello';

*// Обратите внимание на ограничители строки, это бектики*

*// Интерполяция не работает с одинарными и двойными кавычками*

console.log(`${greeting}, ${firstName}!`);

*// => Hello, Joffrey!*

<https://replit.com/@hexlet/js-basics-interpolation>

Мы просто создали одну строку и «вставили» в нее в нужные места константы с помощью знака доллара и фигурных скобок ${ }. Получился как будто бланк, куда внесены нужные значения. И не нужно заботиться об отдельных строках для знаков препинания и пробелов — все эти символы просто записаны в этой строке-шаблоне. В одной строке можно делать сколько угодно подобных блоков.Интерполяция работает только со строками в [бектиках](https://ru.wikipedia.org/wiki/%D0%9C%D0%B0%D1%88%D0%B8%D0%BD%D0%BE%D0%BF%D0%B8%D1%81%D0%BD%D1%8B%D0%B9_%D0%BE%D0%B1%D1%80%D0%B0%D1%82%D0%BD%D1%8B%D0%B9_%D0%B0%D0%BF%D0%BE%D1%81%D1%82%D1%80%D0%BE%D1%84" \t "_blank). Это символ`. (Машинописный обратный апостроф)

Почти во всех языках интерполяция предпочтительнее конкатенации для объединения строк. Строка при этом получается склеенная, и внутри нее хорошо просматриваются пробелы и другие символы. Во-первых, интерполяция позволяет не путать строки с числами (из-за знака +), а во-вторых, так гораздо проще (после некоторой практики) понимать строку целиком.-------------------------

**Шаблонные строки**

Шаблонными литералами называются строковые литералы, допускающие использование выражений внутри. С ними вы можете использовать многострочные литералы и строковую интерполяцию. В спецификациях до ES2015 они назывались "шаблонными строками".

[**Синтаксис**](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Template_literals#%D1%81%D0%B8%D0%BD%D1%82%D0%B0%D0%BA%D1%81%D0%B8%D1%81)

`строка текста`

`строка текста 1

строка текста 2`

`строка текста ${выражение} строка текста`

tag `строка текста ${выражение} строка текста`

[**Описание**](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Template_literals#%D0%BE%D0%BF%D0%B8%D1%81%D0%B0%D0%BD%D0%B8%D0%B5)

Шаблонные литералы заключены в обратные кавычки (` `) вместо двойных или одинарных. Они могут содержать подстановки, обозначаемые знаком доллара и фигурными скобками (${выражение}). Выражения в подстановках и текст между ними передаются в функцию. По умолчанию функция просто объединяет все части в строку. Если перед строкой есть выражение (здесь это tag), то шаблонная строка называется "теговым шаблоном". В этом случае, теговое выражение (обычно функция) вызывается с обработанным шаблонным литералом, который вы можете изменить перед выводом. Для экранирования обратной кавычки в шаблонных литералах указывается обратный слеш **\**.

`\`` === '`' // --> true

[**Многострочные литералы**](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Template_literals#%D0%BC%D0%BD%D0%BE%D0%B3%D0%BE%D1%81%D1%82%D1%80%D0%BE%D1%87%D0%BD%D1%8B%D0%B5_%D0%BB%D0%B8%D1%82%D0%B5%D1%80%D0%B0%D0%BB%D1%8B)

Символы новой строки являются частью шаблонных литералов. Используя обычные строки, вставка переноса потребовала бы следующего синтаксиса:

console.log('string text line 1\n' +

'string text line 2');

// "string text line 1

// string text line 2"

То же с использованием шаблонных литералов:

console.log(`string text line 1

string text line 2`);

// "string text line 1

// string text line 2"

[**Интерполяция выражений**](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Template_literals#%D0%B8%D0%BD%D1%82%D0%B5%D1%80%D0%BF%D0%BE%D0%BB%D1%8F%D1%86%D0%B8%D1%8F_%D0%B2%D1%8B%D1%80%D0%B0%D0%B6%D0%B5%D0%BD%D0%B8%D0%B9)

Для вставки выражений в обычные строки вам пришлось бы использовать следующий синтаксис:

var a = 5;

var b = 10;

console.log('Fifteen is ' + (a + b) + ' and not ' + (2 \* a + b) + '.');

// "Fifteen is 15 and not 20."

Теперь, при помощи шаблонных литералов, вам доступен "синтаксический сахар", делающий подстановки вроде той более читабельными:

var a = 5;

var b = 10;

console.log(`Fifteen is ${a + b} and not ${2 \* a + b}.`);

// "Fifteen is 15 and not 20."

[**Вложенные шаблоны**](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Template_literals#%D0%B2%D0%BB%D0%BE%D0%B6%D0%B5%D0%BD%D0%BD%D1%8B%D0%B5_%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD%D1%8B)

Временами, вложить шаблон — это кратчайший и, возможно, более читабельный способ составить строку. Просто поместите внутрь шаблона с обратными кавычками ещё одни, обернув их в подстановку ${ }. Например, если выражение истинно, можно вернуть шаблонный литерал.

В ES5:

var classes = 'header'

classes += (isLargeScreen() ?

'' : item.isCollapsed ?

' icon-expander' : ' icon-collapser');

В ES2015 с шаблонными литералами без вложения:

const classes = `header ${ isLargeScreen() ? '' :

(item.isCollapsed ? 'icon-expander' : 'icon-collapser') }`;

В ES2015 с вложенными шаблонными литералами:

const classes = `header ${ isLargeScreen() ? '' :

`icon-${item.isCollapsed ? 'expander' : 'collapser'}` }`;

[**Теговые шаблоны**](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Template_literals#%D1%82%D0%B5%D0%B3%D0%BE%D0%B2%D1%8B%D0%B5_%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD%D1%8B)

Расширенной формой шаблонных литералов являются *теговые* шаблоны. Они позволяют разбирать шаблонные литералы с помощью функции. Первый аргумент такой функции содержит массив строковых значений, а остальные содержат выражения из подстановок. В итоге, функция должна вернуть собранную строку (или что-либо совсем иное, как будет показано далее). Имя функции может быть любым.

var person = 'Mike';

var age = 28;

function myTag(strings, personExp, ageExp) {

var str0 = strings[0]; // "That "

var str1 = strings[1]; // " is a "

// Технически, в конце итогового выражения

// (в нашем примере) есть ещё одна строка,

// но она пустая (""), так что пропустим её.

// var str2 = strings[2];

var ageStr;

if (ageExp > 99){

ageStr = 'centenarian';

} else { ageStr = 'youngster'; } // Мы даже можем вернуть строку, построенную другим шаблонным литералом

return `${str0}${personExp}${str1}${ageStr}`;

}

var output = myTag`That ${ person } is a ${ age }`;

console.log(output);

// That Mike is a youngster

Copy to Clipboard

Функция тега не обязана возвращать строку, как показано в примере ниже:

function template(strings, ...keys) {

return (function(...values) {

var dict = values[values.length - 1] || {};

var result = [strings[0]];

keys.forEach(function(key, i) {

var value = Number.isInteger(key) ? values[key] : dict[key];

result.push(value, strings[i + 1]); });

return result.join(''); });}

var t1Closure = template`${0}${1}${0}!`;

t1Closure('Y', 'A'); // "YAY!"

var t2Closure = template`${0} ${'foo'}!`;

t2Closure('Hello', {foo: 'World'}); // "Hello World!"

Copy to Clipboard

[**Сырые строки**](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Template_literals#%D1%81%D1%8B%D1%80%D1%8B%D0%B5_%D1%81%D1%82%D1%80%D0%BE%D0%BA%D0%B8)

Специальное свойство raw, доступное для первого аргумента тегового шаблона, позволяет получить строку в том виде, в каком она была введена, без [экранирования](https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide/Grammar_and_types#%D0%B8%D1%81%D0%BF%D0%BE%D0%BB%D1%8C%D0%B7%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5_%D1%81%D0%BF%D0%B5%D1%86%D0%B8%D0%B0%D0%BB%D1%8C%D0%BD%D1%8B%D1%85_%D1%81%D0%B8%D0%BC%D0%B2%D0%BE%D0%BB%D0%BE%D0%B2_%D0%B2_%D1%81%D1%82%D1%80%D0%BE%D0%BA%D0%B0%D1%85).

function tag(strings) { return strings.raw[0];}

tag`string text line 1 \\n string text line 2`;

// выводит "string text line 1 \\n string text line 2",

// включая 'n' и два символа '\'

Вдобавок, существует метод [String.raw()](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/String/raw), возвращающий точно такую же исходную строку, какую вернула бы функция шаблона по умолчанию и строковая конкатенация вместе.

var str = String.raw`Hi\n${2+3}!`;// "Hi\n5!"

str.length;// 6

str.split('').join(',');

// "H,i,\,n,5,!"

----------------------

# 10. Извлечение символов из строки—

Иногда нужно получить один символ из строки. Например, если сайт знает имя и фамилию пользователя, и в какой-то момент требуется вывести эту информацию в формате *A. Ivanov*, то нужно взять первый символ из имени.

**const** firstName **=** 'Tirion';

console.log(firstName[0]); *// => T*

Квадратные скобки с цифрой — это специальный синтаксис извлечения символа из строки. Цифра называется **индексом** — позицией символа внутри строки. Индексы начинаются с 0 почти во всех языках программирования — поэтому, чтобы получить первый символ, нужно указать индекс 0. Индекс последнего элемента равен длине строки минус единица:

*// Длина строки 6, поэтому последний индекс — это 5*

**const** firstName **=** 'Tirion';

console.log(firstName[5]); *// => n*

*// Вопрос на самопроверку. Что выведет этот код?*

**const** magic **=** '\nyou';

console.log(magic[1]); *// => ?*

Индексом может быть не только конкретное число, но и значение переменной. Вот пример, который приведет к тому же результату — выводу на экран символа *T*, но индекс внутри квадратных скобок записан не числом, а константой:

**const** firstName **=** 'Tirion';

**const** index **=** 0;

console.log(firstName[index]); *// => T*

Технически можно указать индекс и за пределами слова. Для нашего примера — это числа от 6. JavaScript не считает такое поведение ошибкой. Обращение по несуществующему индексу вернет значение undefined.

**const** firstName **=** 'Tirion';

console.log(firstName[10]); *// => undefined*

# 11. Типы данных—Основы JavaScript

Что произойдет, если мы попробуем умножить число на строку? JavaScript вернет NaN (не число) — то самое значение. Оно возникает там, где вместе используются несовместимые значения. В данном случае число и строка:

3 **\*** 'Dracarys'; *// NaN*

Внутри высокоуровневых языков программирования данные разделяются по типам. Любая строка относится к типу String, а числа — к типу Number и BigInt (очень большие числа). Зачем нужны типы? Для защиты программы от трудноотловимых ошибок. Типы определяют две вещи:

* Возможные (допустимые) значения. Например, числа в JavaScript делятся на два типа: Number и BigInt. Первые — это все числа ниже определенного порога (его можно посмотреть), вторые — выше. Такое разделение связано с техническими особенностями работы аппаратуры.
* Набор операций, которые можно выполнять над этим типом. Например, операция умножения имеет смысл для типа «целые числа». Но не имеет смысла для типа «строки»: умножать слово «мама» на слово «блокнот» — бессмыслица.

JavaScript ведет себя двояко, когда встречается с нарушениями. В некоторых ситуациях он ругается на недопустимость операции и завершается с ошибкой. В других — программа продолжает работать. В этом случае недопустимая операция возвращает что-то похожее на NaN, как в примере выше.

Каким образом JavaScript понимает, что за тип данных перед ним? Достаточно просто. Любое значение где-то инициализируется и, в зависимости от способа инициализации, становится понятно, что перед нами. Например, числа — это просто числа без дополнительных символов, кроме точки для рациональных чисел. А вот строки всегда ограничены специальными символами (в JavaScript три разных варианта). Например, такое значение '234' – строка, несмотря на то, что внутри нее записаны цифры.

JavaScript позволяет узнать тип данных с помощью оператора typeof:

**typeof** 3; *// number*

**typeof** 'Game'; *// string*

Типы данных Number, BigInt и String — это *примитивные* типы. Но есть и другие. В JavaScript встроен составной тип Object (а на его базе массивы, даты и другие). С его помощью можно объединять данные разных типов в одно значение, например, мы можем создать пользователя, добавив к нему имя и возраст:

*// Этот синтаксис изучается далее на Хекслете*

**const** user **=** { name: 'Toto', age: 33 };

По-английски строки в программировании называются "strings", а строчки текстовых файлов — "lines". Например, в коде выше есть две строчки (lines), но только одна строка (string). В русском иногда может быть путаница, поэтому во всех уроках мы будем говорить **строка** для обозначения типа данных «строка», и **строчка** для обозначения строчек (lines) в файлах.

undefined

Объявление переменных возможно и без указания конкретного значения. Что будет выведено на экран, если ее распечатать:

**let** name;

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

На экране появится undefined, специальное значение особого типа, которое означает отсутствие значения. Undefined активно используется самим JavaScript в самых разных ситуациях, например, при обращении к несуществующему символу строки:

**const** name **=** 'Arya';

console.log(name[8]);

Смысл (семантика) значения undefined именно в том, что значения нет. Однако, ничто не мешает написать такой код:

**let** key **=** **undefined**;

И хотя интерпретатор позволяет такое сделать, это нарушение семантики значения undefined, ведь в этом коде выполняется присваивание, а значит — подставляется значение.

JavaScript — один из немногих языков, в которых в явном виде присутствует понятие undefined. В остальных языках его функцию выполняет значение null, которое, кстати, тоже есть в JavaScript.

*Вопрос на самопроверку. Почему нельзя объявить константу без указания значения?*

Числа с плавающей точкой

В математике существуют разные виды чисел, например, натуральные – это целые числа от одного и больше, или рациональные – это числа с точкой, например 0.5. С точки зрения устройства компьютеров, между этими видами чисел – пропасть. Попробуйте ответить на простой вопрос, сколько будет *0.2 + 0.1*? А теперь посмотрим, что на это скажет JavaScript:

0.2 **+** 0.1 *// 0.30000000000000004*

Операция сложения двух рациональных чисел внезапно привела к неточному вычислению результата. Тот же самый результат выдадут и другие языки программирования. Такое поведение обуславливается ограничениями вычислительных мощностей. Объем памяти, в отличие от чисел, конечен (бесконечное количество чисел требует бесконечного количества памяти для своего хранения).

Рациональные числа не выстроены в непрерывную цепочку, между *0.1* и *0.2* бесконечное множество чисел. Соответственно возникает серьезная проблема, а как хранить рациональные числа? Это интересный вопрос сам по себе. В интернете множество статей, посвященных организации памяти в таких случаях. Более того, существует стандарт, в котором описано, как это делать правильно, и подавляющее число языков на него опирается.

Для нас, как для разработчиков, важно понимать, что операции с плавающими числами неточны (эту точность можно регулировать), а значит при решении задач, связанных с подобными числами, необходимо прибегать к специальным трюкам, которые позволяют добиться необходимой точности.

Явное преобразование типов

В программировании регулярно встречаются задачи, когда один тип данных нужно преобразовать в другой — например, при работе с формами на сайтах. Данные формы всегда приходят в текстовом виде, даже если значение — число. Вот как его можно преобразовать:

**const** number **=** parseInt('345');

console.log(number); *// => 345*

parseInt() — это функция, в которую передается значение, чтобы его преобразовать. Функция ведет себя подобно арифметическим операциям, но делает особые действия. Вот еще несколько примеров:

**const** value **=** '0';

*// Внутри скобок можно указывать переменную*

**const** number1 **=** parseInt(value);

console.log(number1); *// => 0*

*// Или конкретное значение*

**const** number2 **=** parseInt('10');

console.log(number2); *// => 10*

*// Если преобразуется число с плавающей точкой*

*// то отбрасывается вся дробная часть*

**const** number5 **=** parseInt(3.5);

console.log(number5); *// => 3*

Точно так же можно преобразовать строку в число с плавающей точкой с помощью parseFloat():

**const** value3 **=** parseFloat('0.5');

console.log(value3); *// 0.5*

12.