13. [Неизменяемость и примитивные типы](https://ru.hexlet.io/courses/js-basics/lessons/immutability-of-primitive-types/theory_unit)

Узнаем, может ли JavaScript преобразовывать строки или числа

Что произойдет, если попытаться изменить символ в строке?

**let** firstName **=** 'Alexander';

*// Код выполнится без ошибок*

firstName[0] **=** 'B';

console.log(firstName); *// => Alexander*

Как это ни странно, но значение переменной firstName останется прежним, хотя код выполнится без ошибок. Так происходит из-за неизменяемости примитивных типов в JavaScript — язык не дает никакой физической возможности поменять строку. Неизменяемость примитивных типов важна по многим причинам, ключевая — производительность. Но что делать, если нам действительно нужно ее изменить? Для этого и существуют переменные:

**let** firstName **=** 'Alexander';

*// Код выполнится без ошибок*

firstName **=** 'Blexander';

console.log(firstName); *// => Blexander*

Есть большая разница между изменением значения переменной и изменением самого значения. Примитивные типы в JavaScript поменять нельзя, а заменить значение переменной — без проблем.

Слабая типизация

Нам известно про два разных типа данных: числа и строки. Мы, например, можем складывать числа, потому что операция сложения — это операция для типа «числа».

А что, если применить эту операцию не к двум числам, а к числу и строке?

console.log(1 **+** '7'); *// => 17*

Несмотря на то, что '7' — это строка, а не число, интерпретатор JavaScript выдал ответ 17, как если бы мы складывали две строки. Когда JavaScript видит несоответствие типов, он сам пытается преобразовать информацию. В данном случае он преобразовал число 1 в строку '1', а потом спокойно сделал конкатенацию '1' и '7'.

Не все языки так делают. JavaScript — это язык со **слабой типизацией**. Он знает о существовании разных типов (числа, строки и др.), но относится к их использованию не очень строго, пытаясь преобразовывать информацию, когда это кажется разумным. Иногда JavaScript даже доходит до крайностей. Большинство выражений, не работающих в других языках, прекрасно работают в JavaScript. Попробуйте выполнить любую арифметическую операцию (кроме сложения), подставив туда строки или любые другие типы данных (кроме ситуации, когда оба операнда – это числа или строки, содержащие только число) — вы увидите, что они всегда будут работать и возвращать NaN, что довольно логично.

**const** result **=** 'one' **\*** 'two';

console.log(result); *// => NaN*

<https://replit.com/@hexlet/js-basics-immutability-of-primitive-types>

В языках со **строгой типизацией** сложить число со строкой не получится.

JavaScript был создан для интернета, а в интернете вся информация — это строки. Даже когда вы вводите на сайте номер телефона или год рождения, на сервер эта информация поступает не как числа, а как строки. Поэтому авторы языка решили, что автоматически преобразовывать типы — правильно и удобно.

Такое автоматическое неявное преобразование типов с одной стороны и правда удобно. Но на практике это свойство языка создает множество ошибок и проблем, которые трудно найти. Код может иногда работать, а иногда не работать — в зависимости от того, «повезло» ли в конкретном случае с автоматическим преобразованием. Программист это заметит не сразу.

В дальнейших заданиях вы будете встречаться с таким поведением не раз. Часто будет возникать вопрос «почему мой код работает не так, как я ожидаю?».

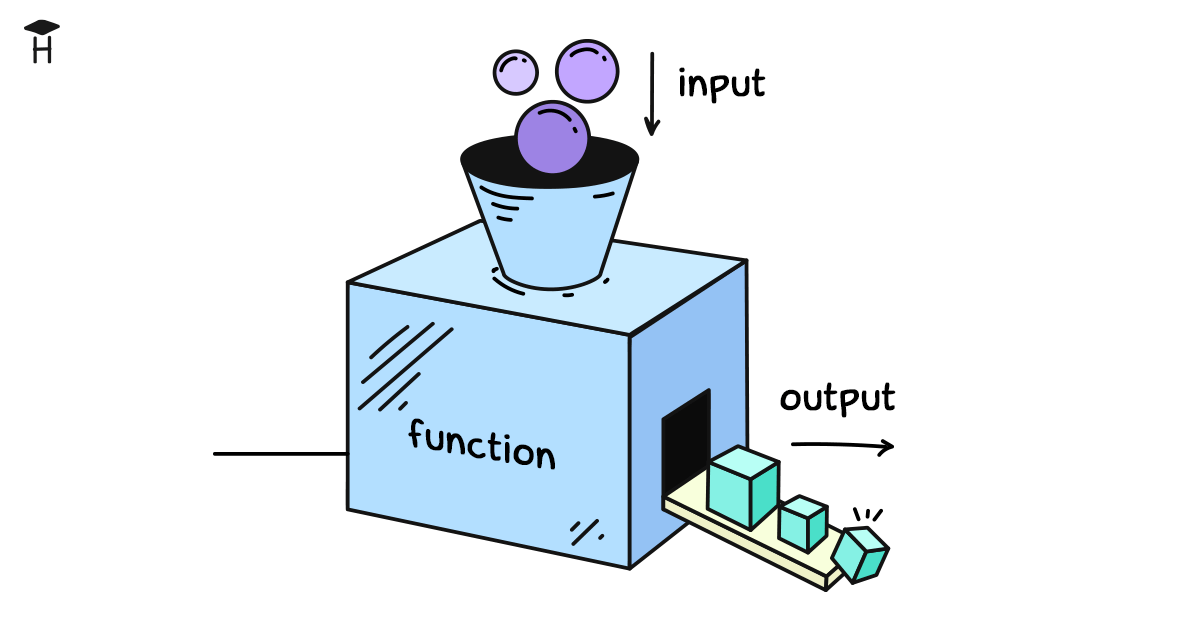
Слабая типизация красной нитью проходит сквозь всю разработку на Javascript.

14. Функции и их вызов—Основы JavaScript

Сложение, конкатенация, нахождение остатка от деления и остальные рассмотренные операции – все это довольно базовые возможности языков программирования. Математика не ограничена арифметикой, кроме нее есть и множество других разделов со своими операциями, например, геометрия. То же самое касается и строк: их можно переворачивать, менять регистр букв, удалять лишние символы — и это только самое простое. И, наконец, на более высоком уровне есть прикладная логика конкретного приложения. Программы списывают деньги, считают налоги, формируют отчеты. Количество подобных операций бесконечно и индивидуально для каждой программы. И все они должны быть как-то выражены в коде.

Для выражения любой произвольной операции в программировании существует понятие *функция*. Функции бывают как встроенные, так и добавленные программистом. С одной встроенной функцией мы уже знакомы, это console.log().

Функции — одна из ключевых конструкций в программировании, без них невозможно сделать практически ничего. Знакомство с ними мы начинаем как можно раньше, так как весь дальнейший материал оперирует функциями по максимуму. Сначала мы научимся пользоваться уже созданными функциями, а уже потом — создавать свои собственные.



Начнем с простых функций для работы над строками. Ниже пример вызова функции length(), которая считает количество символов в строке:

*// length это функция*

**import** { length } **from** './hexlet-basics/string.js';

*// Вызов функции length с параметром 'Hello!'*

**const** result **=** length('Hello!');

console.log(result); *// => 6*

Лирическое отступление. Первая строчка в этом коде – импорт функции из другого модуля. Импорты и модули изучаются в других курсах, здесь же они будут присутствовать в задании «как есть», так как без них невозможно использовать функции, определенные в других файлах. Не заморачивайтесь, если вам не понятен смысл происходящего, мы разберем эту тему позже [в этом уроке](https://ru.hexlet.io/courses/js-basics/lessons/modules/theory_unit).

Параметры (или аргументы) — это информация, которую функция получает при вызове. Именно на основе этой информации функция, как правило, вычисляет что-то и выдает результат.

Мы создали константу result и указали интерпретатору записать в нее результат, **возвращаемый** функцией length() при ее вызове. В этом смысле функции подобны операциям – они всегда возвращают результат своей работы.

*// Вызов length возвращает результат (длину строки)*

*// который записывается в константу с именем result*

**const** result **=** length('Hello!');

Запись length('Hello!') означает, что вызывается функция с именем *length*, в которую был передан параметр 'Hello!'. Функция length() считает длину именно той строки, которая ей была передана.

Вызов функции всегда обозначается скобками (), идущими сразу за именем функции. В скобках может быть любое количество параметров, а иногда — вообще ни одного. Количество зависит от используемой функции. Возьмем для примера функцию pow(), которая возводит указанное число в нужную степень. Она принимает на вход два параметра и возводит число, переданное первым параметром, в степень, переданную вторым параметром.

**import** { pow } **from** './math.js';

*// Вызов pow(2, 3) возвращает значение 2 в 3 степени*

**const** result **=** pow(2, 3); *// 2 \* 2 \* 2*

console.log(result); *// => 8*

По большому счету, операторы и функции — это одно и то же. Ключевая разница только в том, как они записываются. Если представить (гипотетически) сложение как функцию, то она будет выглядеть так:

*// Обычное сложение*

3 **+** 5; *// 8*

*// Сложение, представленное как функция*

*// Выглядит странновато, но передает смысл функций*

**+**(3, 5);

Резюме

Функции вызываются и возвращают результат, который затем может быть использован в дальнейших вычислениях или, например, выведен на экран.

Вопрос на самопроверку. Как узнать, что возвращает вызов функции console.log()? Проверьте.

Математические функции JavaScript

Объяснение функций в JavaScript немного осложняется структурой языка. Изначально он появился в браузерах и имел сильно ограниченные возможности по сравнению с языками общего назначения. Со временем все изменилось — JavaScript стал мощным языком, захватившим клиентскую разработку и активно использующимся на сервере. Однако наследие осталось, так как нужно поддерживать обратную совместимость. Поэтому в некоторых местах есть несостыковки, которые нельзя объяснить системой: на них можно только махнуть рукой и сказать: «Так исторически сложилось».

К подобным «местам» относятся математические функции. В предыдущем задании мы использовали самописную функцию pow() (мы, как создатели курса, добавили ее в практику), а теперь давайте рассмотрим ее версию, встроенную в сам язык.

Math.pow(2, 3); *// 8*

Что такое Math? Технически — это объект, доступный из любого места программы, но перед тем, как говорить об объектах, нужно проделать очень большой путь. Сейчас достаточно запомнить, что функции для математических операций вызываются через Math.. Наличие этой приставки никак не влияет на понятие функции, которое мы рассмотрели ранее и будем рассматривать позже.

# 15. Сигнатура функции—Основы JavaScript

Функция Math.pow(), возводящая число в какую-нибудь степень, принимает два параметра: **какое число возводить** и **в какую степень возводить**. Если вызывать pow() без параметров, то вернется NaN. Функция честно пытается выполнить возведение в степень, но если значение не передано, то интерпретатор автоматически передает ей undefined. JavaScript заставляет программистов быть более аккуратными. В большинстве языков, если передать в функцию меньше параметров, чем она ожидает, то возникнет ошибка, — но только не в JavaScript. NaN вернется при передаче любых нечисловых значений:

**const** result **=** Math.pow(2, 'boom');

console.log(result); *// => NaN*

Другая функция может иметь другое число параметров и другие типы параметров. Например, может существовать функция, которая принимает три параметра: число, строку и еще одно число.

Откуда мы знаем, сколько каких параметров нужно функции Math.pow() и какого типа будет «возврат»? Мы заглянули в **сигнатуру** этой функции. Сигнатура определяет входные параметры и их типы, а также выходной параметр и его тип. Про функцию Math.pow() можно почитать в [документации](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Math/pow). В разделе «Синтаксис» есть такой текст:

Math.pow(base, exponent)

Параметры

base

Основание степени.

exponent

Показатель степени, в которую возводится основание base.

Это сигнатура функции и короткое пояснение на русском языке. Документация позволяет понять, сколько аргументов у функции и какого они типа, возвращает ли что-то функция и если да, то какого типа возвращаемое значение.

Параметры по умолчанию

Рассмотрим функцию round(), которая округляет число с плавающей точкой:

**const** result **=** round(10.25, 0); *// 10*

Мы передали в нее два параметра: число, которое нужно округлить, и точность округления. 0 означает, что округление будет до ближайшего целого значения.

Чаще всего нужно округлять именно до целого числа (а не до десятых, например), поэтому создатели функции round() сделали второй параметр **необязательным** и задали ему внутри функции **значение по умолчанию 0**. Значит, можно не указывать второй параметр, а результат будет тем же:

**const** result **=** round(10.25); *// 10*

Если нужна другая точность, то можно передать параметр:

*// округление до одного знака после запятой*

**const** result **=** round(10.25, 1); *// 10.3*

Если функция в JavaScript принимает необязательные параметры, то они всегда стоят после обязательных. Их количество может быть любым (это зависит от самой функции), но они всегда идут рядом и в конце списка аргументов.

# 16. Вызов функции — выражение

В программировании выражение – нечто, возвращающее результат, который можно использовать. Мы уже знаем достаточно много о выражениях и о принципах их построения. Математические операции (сложение, вычитание), строковые операции (конкатенация) – все это выражения:

1 **+** 5 **\*** 3;

'He' **+** 'Let';

*// Переменные могут быть частью выражения*

rate **\*** 5;

Особенность выражений в том, что они возвращают результат, который можно, например, присвоить константе или вывести на экран. Например:

*// Тут выражение это 1 + 5*

**const** sum **=** 1 **+** 5;

console.log(1 **+** 5);

Но не все в программировании является выражением. Определение переменной – это инструкция, она не может быть частью выражения. То есть такой код выдаст ошибку:

*// Бессмысленный код, который не сработает*

10 **+** **const** sum **=** 1 **+** 5;

Почему об этом важно знать? Как вы увидите дальше, выражения можно комбинировать, получая все более сложное поведение в самых неожиданных местах и самым неожиданным образом. Вы будете лучше понимать, как можно соединять части кода, чтобы получить нужный результат.

Поговорим о функциях. Вызов функции – это выражение или нет? Мы знаем, что функции возвращают результат, поэтому они считаются выражениями. Из этого автоматически следует много интересного. Например, мы можем использовать вызов функции прямо в математических операциях. Вот как можно получить индекс последнего символа в слове:

**import** { length } **from** './hexlet-basics/string.js';

**const** name **=** 'JavaScript';

*// Индексы начинаются с нуля*

*// Вызов функции и вычитание вместе!*

**const** lastIndex **=** length(name) **-** 1;

console.log(lastIndex); *// 9*

В этом коде нет нового синтаксиса. Мы всего лишь соединили уже известные части, опираясь на их природу. Можно пойти еще дальше:

console.log(length(name) **-** 1); *// 9*

Все это справедливо для любых функций, например, строковых:

**import** { length } **from** './hexlet-basics/string.js';

**const** name **=** 'JavaScript';

*// Используется интерполяция*

console.log(`Последний символ: ${name[length(name) **-** 1]}`);

*// 'Последний символ: t'*

# 17. Функции с переменным числом параметров

Интересная особенность некоторых функций — принимать переменное число параметров. Речь не идет о значениях по умолчанию. Посмотрите на этот пример:

Math.max(1, 10, 3); *// 10*

Функция Math.max() находит максимальное значение среди переданных параметров. Как вы думаете, сколько параметров она ожидает на вход? Если открыть документацию этой функции, то мы увидим странную конструкцию:

Math.max([value1[, value2[, ...]]])

Такая запись говорит о том, что эта функция принимает на вход любое число параметров (и даже может быть вызвана без них). Необязательность передаваемых параметров описывается скобками *[ ]*, точно так же описываются и опциональные параметры, у которых есть значения по умолчанию. Возможность передачи любого числа параметров зашита в этой части *[, ...]*.

Math.max(1, **-**3, 2, 3, 2); *// 3*

18. Детерминированность—Основы JavaScript

Независимо от того, какой язык программирования используется, функции внутри него обладают некоторыми фундаментальными свойствами. Зная эти свойства, легче прогнозировать поведение функций, способы их тестирования и место их использования. К таким свойствам относится детерминированность. Функция называется детерминированной тогда, когда для одних и тех же входных параметров она возвращает один и тот же результат. Например, функция, считающая количество символов, детерминированная:

**import** { length } **from** './hexlet-basics/string.js';

length('hexlet'); *// 6*

length('hexlet'); *// 6*

length('wow'); *// 3*

length('wow'); *// 3*

Сколько бы раз мы ни вызывали эту функцию, передавая туда значение 'hexlet', она всегда вернет 6. В свою очередь функция, возвращающая случайное число, не является детерминированной, так как у одного и того же входа (даже если он пустой, то есть параметры не принимаются) мы получим всегда разный результат. Насколько он разный – не важно, даже если хотя бы один из миллиона вызовов вернет что-то другое, эта функция автоматически считается недетерминированной.

*// Функция, возвращающая случайное число*

Math.random(); *// 0.0985661311319767*

Math.random(); *// 0.8839904367241888*

Зачем это нужно знать? Детерминированность серьезно влияет на многие аспекты. Детерминированные функции удобны в работе, их легко оптимизировать, легко тестировать. Если есть возможность сделать функцию детерминированной, то лучше ее такой и сделать.

Побочные эффекты

Вы, скорее всего, уже заметили (может, подсознательно), что console.log() — это тоже функция. Она принимает на вход данные любого типа и выводит их на экран.

Внимание, вопрос: **что возвращает функция console.log()?** Ответ: что бы она ни возвращала, это значение никак не используется.

console.log() выводит что-то на экран, но это не возврат значения, это просто какое-то действие, которое выполняет функция.

Вывод на экран и возврат значения из функции — разные и независимые операции. Технически вывод на экран равносилен записи в файл (немного особый, но все-таки файл). Для понимания этой темы необходимо немного разобраться в устройстве операционных систем, что крайне важно для программистов.

С точки зрения программы вывод на экран — это так называемый **побочный эффект**. Побочным эффектом называют действия, которые изменяют внешнее окружение (среду выполнения). К таким действиям относятся любые сетевые взаимодействия, взаимодействие с файловой системой (чтение и запись файлов), вывод информации на экран или печать на принтере и так далее.

Побочные эффекты — один из основных источников проблем и ошибок в программных системах. Код с побочными эффектами сложен в тестировании, ненадежен. При этом без побочных эффектов программирование не имеет смысла. Без них было бы невозможно получить результат работы программы (записать в базу, вывести на экран, отправить по сети и так далее).

Понимание принципов работы с побочными эффектами очень сильно влияет на стиль программирования и способность строить качественные программы. Эта тема полностью раскроется в следующих курсах на Хекслете.

Вопрос для самопроверки. Можно ли определить наличие побочных эффектов внутри функции, опираясь только на ее возврат?

# 19. Стандартная библиотека—Основы JavaScript

JavaScript, как и любой другой язык, поставляется с набором полезных функций. Все вместе они составляют так называемую **стандартную библиотеку**. В нее обычно входят тысячи функций, которые невозможно выучить — этого и не нужно делать. Подразумевается, что любой программист знает, где искать документацию по ним и примерно представляет себе, чего он хочет достичь. А дальше — дело техники. Если отнять у программистов интернет, то большинство не сможет ничего запрограммировать.

Для новичков эта информация часто выглядит так: «Сходи туда, не знаю куда, принеси то, не знаю что». То есть непонятно, как узнавать про эти функции, когда ты ничего не знаешь вообще. Как ни странно, не существует способа раз и навсегда познать все, что нужно познать. Любой разработчик в процессе своего профессионального взросления знакомится со все более интересными функциями, решающими его задачи более элегантно, и таким образом пополняет свой арсенал.

Вот некоторые советы, как узнавать о новых функциях:

* Всегда четко отслеживайте, с чем вы сейчас работаете (какой тип данных). Почти всегда вы найдете необходимую функцию в соответствующем разделе документации — например, для работы со строками нужно изучать строковые функции
* Периодически открывайте раздел со стандартными функциями по изучаемой тематике и просто пробегайтесь по ним, изучая сигнатуры и способы использования
* Чаще читайте чужой код, особенно код библиотек, которые вы используете. Он весь доступен на GitHub

У JavaScript есть свои особенности по структуре стандартной библиотеки. Так как его код может исполняться в разных средах, таких как серверное окружение или браузер, то возможности стандартной библиотеки сильно зависят от варианта использования. Например, из браузера невозможно выполнять некоторые задачи, которые необходимо уметь выполнять на сервере. Документацию по серверной части необходимо смотреть на сайте [https://nodejs.org](https://nodejs.org/). Серверные части стандартной библиотеки организованы в модули, у каждого модуля есть своя страница с описанием всех функций, находящихся внутри него. Например, модуль [fs](https://nodejs.org/api/fs.html" \t "_blank) необходим для работы с файловой системой, через его функции происходит запись и чтение файлов.

Если говорить про браузер, то, кроме специфических вещей самого браузера, он включает какие-то базовые функции, встроенные в сам язык — например те же [функции](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Math) для работы с математикой. Остальные возможности добавляются через использование сторонних библиотек.

# 20. Свойства и методы—Основы JavaScript

Данные, которыми мы оперируем в своих программах, могут обладать важными свойствами — например, у строк есть длина. Как вы увидите далее, это свойство очень важно для реализации алгоритмов, связанных с преобразованием строки (как пример — переворот строки). Как узнать длину строки? Во многих языках длина строки вычисляется с помощью специальной функции и выглядит это примерно так:

**import** { length } **from** './hexlet-basics/string.js';

**const** name **=** 'Robb';

console.log(length(name)); *// => 4*

В JavaScript свойства встроены прямо в язык. Они указываются через точку сразу после переменной (или константы):

**const** name **=** 'Robb';

**const** len **=** name.length;

console.log(len); *// => 4*

Свойства связаны с данными, у которых они берутся. Для примитивных типов все свойства описаны в документации, как например, у [строк](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/String). При этом у чисел вообще нет свойств.

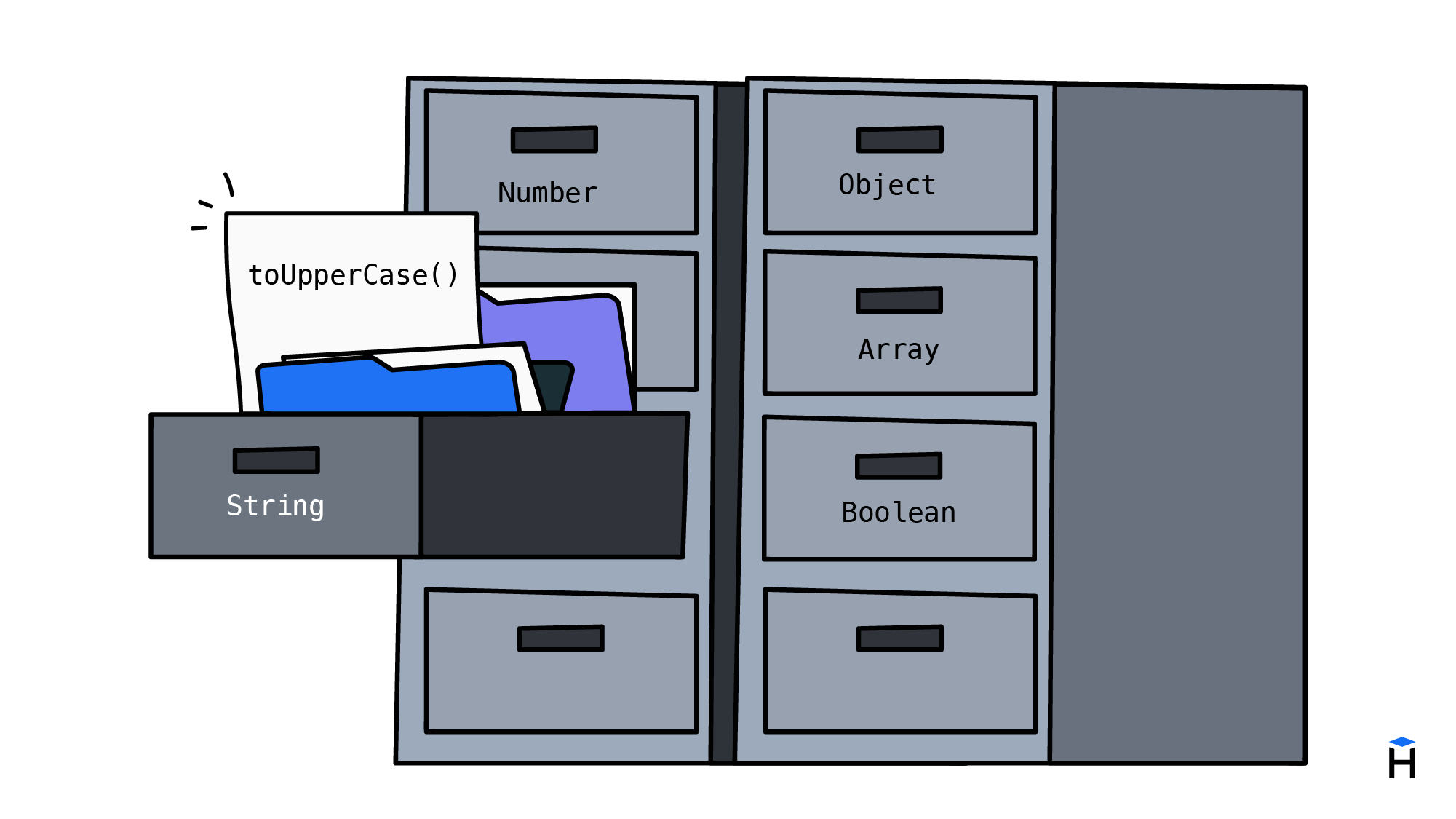
JavaScript позволяет обращаться к свойствам, которые не существуют (например, при опечатках). В таком случае их значением является undefined:

**const** name **=** 'Robb';

console.log(name.whatIsThat); *// => undefined*

*Вопрос для самопроверки. Что распечатает код console.log(name[name.length]) для name, определенного выше? Почему ответ такой?*

Методы



Кроме свойств, у данных существуют методы — функции, находящиеся внутри свойств. С практической точки зрения это значит, что метод работает и вызывается как функция, но делает это как свойство, через точку.

**const** name **=** 'Robb';

**const** upperName **=** name.toUpperCase();

console.log(upperName); *// => ROBB*

Встроенные методы всегда оперируют теми данными, с которыми они связаны. Метод .toUpperCase() возвращает такую же строку, но преобразуя все символы в верхний регистр. Методов у данных обычно значительно больше, чем свойств, например, для строк их [несколько десятков](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/String). В документации, на первый взгляд, они описаны немного странно: *String.prototype.toLowerCase()*. Это описание раскрывает некоторые внутренние детали реализации, которые сейчас не важны, да и мы не изучили всей необходимой базы для разговора о прототипах.

Методы есть и у [чисел](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Number):

**const** temperature **=** 22.93;

*// Округление до одного знака после запятой*

**const** roundedTemperature **=** temperature.toFixed(1);

*// Метод возвращает строку, которая содержит преобразованное число*

console.log(roundedTemperature); *// => 22.9*

*// Напрямую можно вызывать так*

(22.93).toFixed(1); *// 22.9*

*Технически всё несколько сложнее. Методы есть не у самих чисел, а у данных (объектов) типа Number. Числа, записанные в переменные или константы, автоматически преобразуются к данному типу во время обращения к ним, в это время происходит так называемый boxing.*

Возникает закономерный вопрос: зачем нужны методы, почему не просто функции? С числами ситуация еще сложнее. Часть операций реализована в виде методов самих чисел, например, .toFixed(), а еще большая часть — в виде методов, доступных через Math.

Есть две причины, почему так сделано:

1. Исторически так сложилось. JavaScript разрабатывался слишком быстро и поэтому не все было продумано хорошо.
2. Далеко не все функции имеют отношение к конкретному значению. Возьмем для примера Math.min(). Эта функция находит минимальное число среди всех, которые ему были переданы. Эту функцию нелогично делать методом конкретного числа, например, так — (1).min(). Она не имеет никакой связи с конкретным числом

С другой стороны, функции, работающие с конкретным числом, для единообразия должны быть реализованы как методы. К таким функциям относится получение модуля числа. То есть вместо такого вызова Math.abs(-10), логично иметь такой: (-10).abs().

Что касается методов в целом, то не все так однозначно. Есть языки, в которых методов нет и там все прекрасно, есть языки, где методы — это основной способ работы с функциями. JavaScript — язык, в котором прижились оба подхода, в нем активно используются как обычные функции, так и методы. О плюсах и минусах подобных подходов подробно рассказывается в курсах, [посвященных ООП](https://ru.hexlet.io/programs/js-oop).

Неизменяемость

Что напечатает на экран последний вызов?

**const** name **=** 'Tirion';

console.log(name.toUpperCase()); *// => TIRION*

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

Ответ на этот вопрос зависит от того, как вы поняли урок про неизменяемость примитивных типов данных. Вызов метода .toUpperCase() возвращает новое значение, в котором все буквы преобразованы в верхний регистр, но он не меняет (и не может этого сделать) исходную строку. Поэтому внутри константы (или переменной — это не важно) окажется старое значение: 'Tirion'. Эта логика справедлива для методов всех примитивных типов. Более того, попытка изменить значение свойства этих данных ни к чему не приведет:

**const** name **=** 'Tirion';

console.log(name.length); *// => 6*

name.length **=** 100;

console.log(name.length); *// => 6*

Вместо изменения значения можно *заменить* значение. Для этого понадобятся переменные:

**let** name **=** 'Tirion';

name **=** name.toUpperCase();

console.log(name); *// => TIRION*

Свойства и методы как выражения

Свойства и методы — такие же выражения, как переменные, константы или вызовы функции, а значит, их можно всячески комбинировать.

Использование в операциях:

**const** name **=** 'Shaya';

name.length **+** 5; *// 10*

`hi, ${name.toUpperCase()}!`; *// hi, SHAYA!*

Использование в параметрах функций:

**const** name1 **=** 'Robb';

**const** name2 **=** 'Shaya';

console.log(name2.length); *// => 5*

console.log(name2.toLowerCase()); *// => shaya*

console.log(Math.min(name1.length, name2.length)); *// => 4*

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