Типы данных

Тезисно:
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
```

https://replit.com/@hexlet/js-basics-data-types-primitive-data-types

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.

```
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 (strings). 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.

Определения

- Тип данных — множество данных в коде (разновидность информации). Тип определяет, что можно делать с элементами конкретного множества. Например, целые числа, рациональные числа, строки — это разные типы данных.
- Примитивные типы данных — простые типы, встроенные в сам язык программирования.
- Строка (string) — тип данных, описывающий набор символов (иными словами — текст); например, 'text' или "text".

Подробно:

Значение в JavaScript всегда относится к данным определённого типа. Например, это может быть строка или число.

Есть восемь основных типов данных в JavaScript. В этой главе мы рассмотрим их в общем, а в следующих главах поговорим подробнее о каждом.

Переменная в JavaScript может содержать любые данные. В один момент там может быть строка, а в другой – число:

```
// Не будет ошибкой
let message = "hello";
message = 123456;
```

Языки программирования, в которых такое возможно, называются «динамически типизированными». Это значит, что типы данных есть, но переменные не привязаны ни к одному из них.

Число

```
let n = 123;
n = 12.345;
```

Числовой тип данных (`number`) представляет как целочисленные значения, так и числа с плавающей точкой.

Существует множество операций для чисел, например, умножение `*`, деление `/`, сложение `+`, вычитание `-` и так далее.

Кроме обычных чисел, существуют так называемые «специальные числовые значения», которые относятся к этому типу данных: `Infinity`, `-Infinity` и `NaN`.

- `Infinity` представляет собой математическую бесконечность ∞. Это особое значение, которое больше любого числа.

  Мы можем получить его в результате деления на ноль:

  ```
  alert( 1 / 0 ); // Infinity
  ```

  Или задать его явно:

  ```
  alert( Infinity ); // Infinity
  ```

- `NaN` означает вычислительную ошибку. Это результат неправильной или неопределённой математической операции, например:

  ```
  alert( "не число" / 2 ); // NaN, такое деление является ошибкой
  ```

Значение NaN «прилипчиво». Любая математическая операция с NaN возвращает NaN:

```
alert( NaN + 1 ); // NaN
alert( 3 * NaN ); // NaN
alert( "не число" / 2 - 1 ); // NaN
```

Если где-то в математическом выражении есть NaN, то оно распространяется на весь результат (есть только одно исключение: NaN ** 0 равно 1).

Математические операции – безопасны

Математические операции в JavaScript «безопасны». Мы можем делать что угодно: делить на ноль, обращаться с нечисловыми строками как с числами и т.д.

Скрипт никогда не остановится с фатальной ошибкой (не «умрёт»). В худшем случае мы получим NaN как результат выполнения.

Специальные числовые значения относятся к типу «число». Конечно, это не числа в привычном значении этого слова.

Подробнее о работе с числами мы поговорим в главе Числа.

BigInt

В JavaScript тип number не может безопасно работать с числами, большими, чем $(2^{53}-1)$ (т. е. 9007199254740991) или меньшими, чем $-(2^{53}-1)$ для отрицательных чисел.

Если говорить совсем точно, то, технически, тип number *может* хранить большие целые числа (до $1.7976931348623157 * 10^{308}$), но за пределами безопасного диапазона целых чисел $\pm(2^{53}-1)$ будет ошибка точности, так как не все цифры помещаются в фиксированную 64-битную память. Поэтому можно хранить «приблизительное» значение.

Например, эти два числа (прямо за пределами безопасного диапазона) совпадают:

```
console.log(9007199254740991 + 1); // 9007199254740992
console.log(9007199254740991 + 2); // 9007199254740992
```

То есть все нечетные целые числа, большие чем $(2^{53}-1)$, вообще не могут храниться в типе number.

В большинстве случаев безопасного диапазона чисел от $-(2^{53}-1)$ до $(2^{53}-1)$ вполне достаточно, но иногда нам требуется весь диапазон действительно гигантских целых чисел без каких-либо ограничений или пропущенных значений внутри него. Например, в криптографии или при использовании метки времени («timestamp») с микросекундами.

Тип `BigInt` был добавлен в JavaScript, чтобы дать возможность работать с целыми числами произвольной длины.

Чтобы создать значение типа `BigInt`, необходимо добавить `n` в конец числового литерала:

```
// символ "n" в конце означает, что это BigInt
const bigInt = 1234567890123456789012345678901234567890n;
```

Так как необходимость в использовании `BigInt`–чисел появляется достаточно редко, мы рассмотрим их в отдельной главе BigInt. Ознакомьтесь с ней, когда вам понадобятся настолько большие числа.

Поддержка
В данный момент `BigInt` поддерживается только в браузерах Firefox, Chrome, Edge и Safari, но не поддерживается в IE.

Строка

Строка (`string`) в JavaScript должна быть заключена в кавычки.

```
let str = "Привет";
let str2 = 'Одинарные кавычки тоже подойдут';
let phrase = `Обратные кавычки позволяют встраивать переменные ${str}`;
```

В JavaScript существует три типа кавычек.

1. Двойные кавычки: `"Привет"`.
2. Одинарные кавычки: `'Привет'`.
3. Обратные кавычки: `` `Привет` ``.

Двойные или одинарные кавычки являются «простыми», между ними нет разницы в JavaScript.

Обратные же кавычки имеют расширенную функциональность. Они позволяют нам встраивать выражения в строку, заключая их в `${…}`. Например:

```
let name = "Иван";

// Вставим переменную
alert( `Привет, ${name}!` ); // Привет, Иван!

// Вставим выражение
alert( `результат: ${1 + 2}` ); // результат: 3
```

Выражение внутри `${…}` вычисляется, и его результат становится частью строки. Мы можем положить туда всё, что угодно: переменную `name`, или выражение `1 + 2`, или что-то более сложное.

Обратите внимание, что это можно делать только в обратных кавычках. Другие кавычки не имеют такой функциональности встраивания!

```
alert( "результат: ${1 + 2}" ); // результат: ${1 + 2} (двойные кавычки ничего не делают)
```

Мы рассмотрим строки более подробно в главе Строки.

Нет отдельного типа данных для одного символа.
В некоторых языках, например C и Java, для хранения одного символа, например `"a"` или `"%"`, существует отдельный тип. В языках C и Java это `char`.

В JavaScript подобного типа нет, есть только тип `string`. Строка может содержать ноль символов (быть пустой), один символ или множество.

Булевый (логический) тип

Булевый тип (`boolean`) может принимать только два значения: `true` (истина) и `false` (ложь).

Такой тип, как правило, используется для хранения значений да/нет: `true` значит «да, правильно», а `false` значит «нет, не правильно».

Например:

```
let nameFieldChecked = true; // да, поле отмечено
let ageFieldChecked = false; // нет, поле не отмечено
```

Булевые значения также могут быть результатом сравнений:

```
let isGreater = 4 > 1;

alert( isGreater ); // true (результатом сравнения будет "да")
```

Значение «null»

Специальное значение `null` не относится ни к одному из типов, описанных выше.

Оно формирует отдельный тип, который содержит только значение `null`:

```
let age = null;
```

В JavaScript `null` не является «ссылкой на несуществующий объект» или «нулевым указателем», как в некоторых других языках.

Это просто специальное значение, которое представляет собой «ничего», «пусто» или «значение неизвестно».

В приведённом выше коде указано, что значение переменной age неизвестно.

Значение «undefined»

Специальное значение `undefined` также стоит особняком. Оно формирует тип из самого себя так же, как и `null`.

Оно означает, что «значение не было присвоено».

Если переменная объявлена, но ей не присвоено никакого значения, то её значением будет `undefined`:

```
let age;

alert(age); // выведет "undefined"
```

Технически мы можем присвоить значение `undefined` любой переменной:

```
let age = 123;

// изменяем значение на undefined
```

```
age = undefined;

alert(age); // "undefined"
```

...Но так делать не рекомендуется. Обычно `null` используется для присвоения переменной «пустого» или «неизвестного» значения, а `undefined` – для проверок, была ли переменная назначена.

JS: undefined

Объявление переменных возможно и без указания конкретного значения. Что будет выведено на экран если её 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]);

https://replit.com/@hexlet/js-basics-data-types-undefined

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 роль выполняет значение null, которое, кстати, тоже есть в JavaScript.

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

Определения

- undefined — аналог отсутствия значения; указывает, что переменной не присвоено значение или она вообще не объявлена.

JS: Неизменяемость примитивных типов

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 её 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 вот составные можно), а заменить значение переменной — без проблем.

Определения

- Неизменяемость — состояние, при котором объект, переменная не могут быть изменены после создания

Объекты и символы

Тип `object` (объект) – особенный.

Все остальные типы называются «примитивными», потому что их значениями могут быть только простые значения (будь то строка, или число, или что-то ещё). В объектах же хранят коллекции данных или более сложные структуры.

Объекты занимают важное место в языке и требуют особого внимания. Мы разберёмся с ними в главе Объекты после того, как узнаем больше о примитивах.

Тип `symbol` (символ) используется для создания уникальных идентификаторов в объектах. Мы упоминаем здесь о нём для полноты картины, изучим этот тип после объектов.

Оператор typeof

Оператор `typeof` возвращает тип аргумента. Это полезно, когда мы хотим обрабатывать значения различных типов по-разному или просто хотим сделать проверку.

У него есть две синтаксические формы:

```
// Обычный синтаксис
typeof 5 // Выведет "number"
// Синтаксис, напоминающий вызов функции (встречается реже)
typeof(5) // Также выведет "number"
```

Если передается выражение, то нужно заключать его в скобки, т.к. typeof имеет более высокий приоритет, чем бинарные операторы:

```
typeof 50 + " Квартир"; // Выведет "number Квартир"
typeof (50 + " Квартир"); // Выведет "string"
```

Другими словами, скобки необходимы для определения типа значения, которое получилось в результате выполнения выражения в них.

Вызов `typeof x` возвращает строку с именем типа:

```
typeof undefined // "undefined"

typeof 0 // "number"
```

```
typeof 10n // "bigint"

typeof true // "boolean"

typeof "foo" // "string"

typeof Symbol("id") // "symbol"

typeof Math // "object"  (1)

typeof null // "object"  (2)

typeof alert // "function"  (3)
```

Последние три строки нуждаются в пояснении:

1. `Math` — это встроенный объект, который предоставляет математические операции и константы. Мы рассмотрим его подробнее в главе Числа. Здесь он служит лишь примером объекта.
2. Результатом вызова `typeof null` является `"object"`. Это официально признанная ошибка в `typeof`, ведущая начало с времён создания JavaScript и сохранённая для совместимости. Конечно, `null` не является объектом. Это специальное значение с отдельным типом.
3. Вызов `typeof alert` возвращает `"function"`, потому что `alert` является функцией. Мы изучим функции в следующих главах, где заодно увидим, что в JavaScript нет специального типа «функция». Функции относятся к объектному типу. Но `typeof` обрабатывает их особым образом, возвращая `"function"`. Так тоже повелось от создания JavaScript. Формально это неверно, но может быть удобным на практике.

JS: 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.

Определения

- Слабая типизация — это типизация, при которой язык программирования выполняет множество неявных преобразований типов автоматически, даже если может произойти потеря точности или преобразование неоднозначно.

Итого

В JavaScript есть 8 основных типов данных.

- Семь из них называют «примитивными» типами данных:
  - `number` для любых чисел: целочисленных или чисел с плавающей точкой; целочисленные значения ограничены диапазоном $\pm(2^{53}-1)$.
  - `bigint` для целых чисел произвольной длины.
  - `string` для строк. Строка может содержать ноль или больше символов, нет отдельного символьного типа.
  - `boolean` для `true`/`false`.
  - `null` для неизвестных значений – отдельный тип, имеющий одно значение `null`.
  - `undefined` для неприсвоенных значений – отдельный тип, имеющий одно значение `undefined`.
  - `symbol` для уникальных идентификаторов.
- И один не является «примитивным» и стоит особняком:
  - `object` для более сложных структур данных.

Оператор `typeof` позволяет нам увидеть, какой тип данных сохранён в переменной.

- Имеет две формы: `typeof x` или `typeof(x)`.

- Возвращает строку с именем типа. Например, `"string"`.
- Для `null` возвращается `"object"` – это ошибка в языке, на самом деле это не объект.

Задачи

1) Шаблонные строки

Что выведет этот скрипт и почему?

```
let name = "Ilya";

alert( `hello ${1}` ); // ?

alert( `hello ${"name"}` ); // ?

alert( `hello ${name}` ); // ?
```

Решение:

```
Обратные кавычки позволяют вставить выражение внутри ${...} в строку.

let name = "Ilya";

// выражение - число 1
alert( `hello ${1}` ); // hello 1

// выражение - строка "name"
alert( `hello ${"name"}` ); // hello name

// выражение - переменная, вставим её в строку
alert( `hello ${name}` ); // hello Ilya
```

2) Выведите на экран число -0.304.

Решение:

```
console.log(-0.304);
```

3) **Выведите на экран значение undefined, не указывая его явно (через присваивание или передав напрямую в console.log()).**

Решение:

```
let name;
console.log(name);
```

4) **Вам даны три константы с фамилиями разных людей. Составьте и выведите на экран в одну строку слово из символов в таком порядке:**

1. Третий символ из первой строки
2. Второй символ из второй строки
3. Четвертый символ из третьей строки
4. Пятый символ из второй строки
5. Третий символ из второй строки

Попробуйте использовать интерполяцию: внутри фигурных скобок можно помещать не только целые переменные, но и отдельные символы с помощью квадратных скобок.

```
const one = 'Naharis';
const two = 'Mormont';
const three = 'Sand';

// BEGIN (write your solution here)

// END
```

Решение:

```
const one = 'Naharis';
const two = 'Mormont';
const three = 'Sand';

// BEGIN
console.log(`${one[2]}${two[1]}${three[3]}${two[4]}${two[2]}`);
// END
```

5) **Выведите на экран результат выражения: 7 - (-8 - -2).** Попробуйте сделать число 7 не числом, а строкой. Поэкспериментируйте с другими числами тоже.

Решение:

```
console.log('7' - (-8 - -2));
```

Взаимодействие: alert, prompt, confirm

Так как мы будем использовать браузер как демо-среду, нам нужно познакомиться с несколькими функциями его интерфейса, а именно: `alert`, `prompt` и `confirm`.

alert

С этой функцией мы уже знакомы. Она показывает сообщение и ждёт, пока пользователь нажмёт кнопку «OK».

Например:

```
alert("Hello");
```

Это небольшое окно с сообщением называется *модальным окном*. Понятие *модальное* означает, что пользователь не может взаимодействовать с интерфейсом остальной части страницы, нажимать на другие кнопки и т.д. до тех пор, пока взаимодействует с окном. В данном случае – пока не будет нажата кнопка «OK».

prompt

Функция `prompt` принимает два аргумента:

```
result = prompt(title, [default]);
```

Этот код отобразит модальное окно с текстом, полем для ввода текста и кнопками OK/Отмена.

`title`

Текст для отображения в окне.

`default`

Необязательный второй параметр, который устанавливает начальное значение в поле для текста в окне.

Квадратные скобки в синтаксисе `[...]`

Квадратные скобки вокруг `default` в описанном выше синтаксисе означают, что параметр факультативный, необязательный.

Пользователь может напечатать что-либо в поле ввода и нажать OK. Введённый текст будет присвоен переменной `result`. Пользователь также может отменить ввод нажатием на кнопку «Отмена» или нажав на клавишу `Esc`. В этом случае значением `result` станет `null`.

Вызов `prompt` возвращает текст, указанный в поле для ввода, или `null`, если ввод отменён пользователем.

Например:

```
let age = prompt('Сколько тебе лет?', 100);

alert(`Тебе ${age} лет!`); // Тебе 100 лет!
```

Для IE: всегда устанавливайте значение по умолчанию

Второй параметр является необязательным, но если не указать его, то Internet Explorer вставит строку `"undefined"` в поле для ввода.

Запустите код в Internet Explorer и посмотрите на результат:

```
let test = prompt("Test");
```

Чтобы `prompt` хорошо выглядел в IE, рекомендуется всегда указывать второй параметр:

```
let test = prompt("Test", ''); // <-- для IE
```

confirm

Синтаксис:

```
result = confirm(question);
```

Функция `confirm` отображает модальное окно с текстом вопроса `question` и двумя кнопками: OK и Отмена.

Результат – `true`, если нажата кнопка OK. В других случаях – `false`.

Например:

```
let isBoss = confirm("Ты здесь главный?");

alert( isBoss ); // true, если нажата OK
```

Итого

Мы рассмотрели 3 функции браузера для взаимодействия с пользователем:

`alert`

показывает сообщение.

`prompt`

показывает сообщение и запрашивает ввод текста от пользователя. Возвращает напечатанный в поле ввода текст или `null`, если была нажата кнопка «Отмена» или `Esc` с клавиатуры.

`confirm`

показывает сообщение и ждёт, пока пользователь нажмёт OK или Отмена. Возвращает `true`, если нажата OK, и `false`, если нажата кнопка «Отмена» или `Esc` с клавиатуры.

Все эти методы являются модальными: останавливают выполнение скриптов и не позволяют пользователю взаимодействовать с остальной частью страницы до тех пор, пока окно не будет закрыто.

На все указанные методы распространяются два ограничения:

1. Расположение окон определяется браузером. Обычно окна находятся в центре.
2. Визуальное отображение окон зависит от браузера, и мы не можем изменить их вид.

Такова цена простоты. Есть другие способы показать более приятные глазу окна с богатой функциональностью для взаимодействия с пользователем, но если «навороты» не имеют значения, то данные методы работают отлично.

Задачи

Простая страница

Создайте страницу, которая спрашивает имя у пользователя и выводит его.

Решение:

```
<!DOCTYPE html>

<html>

<body>


 <script>

   'use strict';

   let name = prompt("Ваше имя?", "");

   alert(name);

 </script>


</body>

</html>
```

Преобразование типов

Чаще всего операторы и функции автоматически приводят переданные им значения к нужному типу.

Например, `alert` автоматически преобразует любое значение к строке. Математические операторы преобразуют значения к числам.

Есть также случаи, когда нам нужно явно преобразовать значение в ожидаемый тип.

Пока что мы не говорим об объектах
В этой главе мы не касаемся объектов. Сначала мы разберём преобразование примитивных значений.

Мы разберём преобразование объектов позже, в главе Преобразование объектов в примитивы.

Строковое преобразование

Строковое преобразование происходит, когда требуется представление чего-либо в виде строки.

Например, `alert(value)` преобразует значение к строке.

Также мы можем использовать функцию `String(value)`, чтобы преобразовать значение к строке:

```
let value = true;
alert(typeof value); // boolean

value = String(value); // теперь value это строка "true"
alert(typeof value); // string
```

Преобразование происходит очевидным образом. `false` становится `"false"`, `null` становится `"null"` и т.п.

Численное преобразование

Численное преобразование происходит в математических функциях и выражениях.

Например, когда операция деления `/` применяется не к числу:

```
alert( "6" / "2" ); // 3, строки преобразуются в числа
```

Мы можем использовать функцию `Number(value)`, чтобы явно преобразовать `value` к числу:

```
let str = "123";
alert(typeof str); // string

let num = Number(str); // становится числом 123

alert(typeof num); // number
```

Явное преобразование часто применяется, когда мы ожидаем получить число из строкового контекста, например из текстовых полей форм.

Если строка не может быть явно приведена к числу, то результатом преобразования будет NaN. Например:

```
let age = Number("Любая строка вместо числа");

alert(age); // NaN, преобразование не удалось
```

Правила численного преобразования:

Значение	Преобразуется в…
undefined	NaN
null	0
true / false	1 / 0
string	Пробельные символы (пробелы, знаки табуляции \t, знаки новой строки \n и т. п.) по краям обрезаются. Далее, если остаётся пустая строка, то получаем 0, иначе из непустой строки «считывается» число. При ошибке результат NaN.

Примеры:

```
alert( Number("   123   ") ); // 123
alert( Number("123z") );      // NaN (ошибка чтения числа на месте символа "z")
alert( Number(true) );        // 1
alert( Number(false) );       // 0
```

Учтите, что null и undefined ведут себя по-разному.
Так, null становится нулём, тогда как undefined приводится к NaN.

Большинство математических операторов также производит данное преобразование, как мы увидим в следующей главе.

Логическое преобразование

Логическое преобразование самое простое.

Происходит в логических операциях (позже мы познакомимся с условными проверками и подобными конструкциями), но также может быть выполнено явно с помощью функции Boolean(value).

Правило преобразования:

- Значения, которые интуитивно «пустые», вроде 0, пустой строки, null, undefined и NaN, становятся false.

- Все остальные значения становятся `true`.

  Например:

  ```
  alert( Boolean(1) ); // true
  alert( Boolean(0) ); // false

  alert( Boolean("Привет!") ); // true
  alert( Boolean("") ); // false
  ```

  Заметим, что строка с нулём `"0"` — это `true`
  Некоторые языки (к примеру, PHP) воспринимают
  строку `"0"` как `false`. Но в JavaScript, если строка не пустая, то она всегда `true`.

  ```
  alert( Boolean("0") ); // true
  alert( Boolean(" ") ); // пробел это тоже true (любая непустая строка это true)
  ```

  ## Итого

  Существует 3 наиболее широко используемых преобразования: строковое, численное и логическое.

  Строковое – Происходит, когда нам нужно что-то вывести. Может быть вызвано с помощью `String(value)`. Для примитивных значений работает очевидным образом.

  Численное – Происходит в математических операциях. Может быть вызвано с помощью `Number(value)`.

  Преобразование подчиняется правилам:

Значение	Становится…
`undefined`	`NaN`
`null`	`0`
`true / false`	`1 / 0`
`string`	Пробельные символы по краям обрезаются. Далее, если остаётся пустая строка, то получаем `0`, иначе из непустой строки «считывается» число. При ошибке результат `NaN`.

  Логическое – Происходит в логических операциях. Может быть вызвано с помощью `Boolean(value)`.

Подчиняется правилам:

Значение	Становится…
`0`, `null`, `undefined`, `NaN`, `""`	`false`
любое другое значение	`true`

Большую часть из этих правил легко понять и запомнить. Особые случаи, в которых часто допускаются ошибки:

- `undefined` при численном преобразовании становится `NaN`, не `0`.
- `"0"` и строки из одних пробелов типа `" "` при логическом преобразовании всегда `true`.