

JavaScript, часть 1: основы и функции

Глава 1

JavaScript. Введение

1.1. Введение

Изначально JavaScript развивался для того, чтобы сделать интернет-страницы более отзывчивыми и интерактивными. Чтобы пользователь во время взаимодействия со страницей получал мгновенный feedback от своих действий.

До JavaScript для этих целей использовался язык Java. Создание динамической страницы требовало от Java-программиста много усилий. Написанный код нужно было сначала скомпилировать, затем упаковать результат компиляции в апплет и подключить его к странице. Такое положение вещей не соответствовало высоким темпам развития всемирной сети. Использование JavaScript позволяет исключить лишние действия, тем самым упрощая жизнь разработчику: достаточно написать код и подключить его на страницу.

Можно выделить следующие особенности языка JavaScript:

Синтаксис → **Java, C, C++**. Синтаксис языка был сделан похожим на синтаксис Java, C и C++. Это было сделано специально, чтобы разработчики, которые писали на этих языках, легко могли освоить и JavaScript.

Динамическая типизация Динамическая типизация была позаимствована из Perl, который был популярным в то время языком.

Ссылки на функции → **Lips**. Функции в JavaScript являются объектами первого класса, то есть их можно использовать и передавать по ссылке как аргументы.

Наследование через прототипы → **Self**. Наследование в JavaScript реализовано через прототипы. Эта идея была заимствована из языка Self.

JavaScript был разработан Брэндоном Айком для компании Netscape в 1995 году.



Создатель JavaScript, Брэндон Аик.

Вот как сам Брэндон пишет о том, как создавался язык:

... он должен был быть написан за 10 дней, а иначе мы бы имели что-то похуже JS...

... В то время мы должны были двигаться очень быстро, т.к. знали, что Microsoft идет за нами...

... JS был обязан «выглядеть как Java», только поменьше, быть эдаким младшим братом-простаком для Java...

Следующим важным этапом создания JavaScript было появление формата JSON (JavaScript Object Notation). JSON разработал Дуглас Крокфорд в 2001 году, чтобы заменить популярный в то время формат XML.

```

<?xml version="1.0" encoding="UTF-8" ?>
<coffee_shop>
  <name>Works</name>
  <cashlessPayment>true</cashlessPayment>
  <capacity>3</capacity>
  <barista>
    <persone>
      <name>Лёша</name>
      <favourite>cappuccino</favourite>
    </persone>
    <persone>
      <name>Лиза</name>
      <favourite>tea</favourite>
    </persone>
  </barista>
</coffee_shop>

```

Listing 1: Пример данных в формате XML.

Формат XML сам по себе является избыточным. Поэтому, чтобы ускорить передачу данных по сети, он был заменен на JSON.

```

{
  "name": "Works",
  "cashlessPayment": true,
  "capacity": 3,
  "barista": [
    {
      "name": "Лёша",
      "favourite": "cappuccino"
    },
    {
      "name": "Лиза",
      "favourite": "tea"
    }
  ]
}

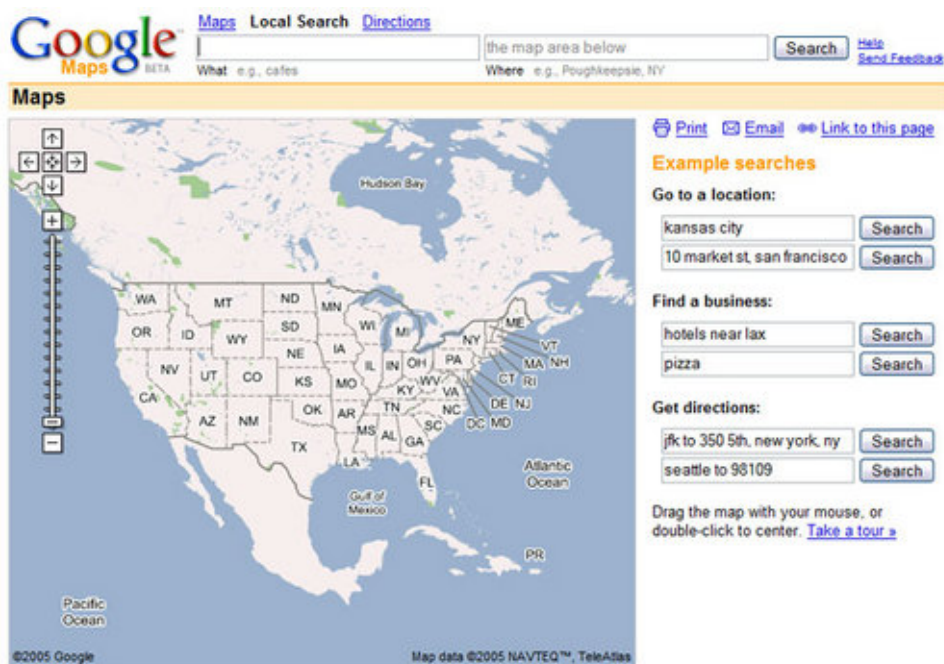
```

Listing 2: Пример данных в формате JSON

JSON также является более выразительным, чем XML, в дополнение к тому, что он не является избыточным. Кроме того, с JSON было легко

работать из JavaScript, поскольку данные в формате JSON были представлениями объектов JavaScript.

В 2005 году Джеймс Гаррет разрабатывает AJAX (Asynchronous JavaScript and XML). AJAX позволяет подгружать данные без обновления веб-страницы.



Google maps, 2005 год

С использованием этого подхода были реализованы карты Google. В результате этого пользователь мог подгружать нужный фрагмент карты без необходимости обновлять страницу браузера. В то время это было прорывом.

JavaScript распространялся с бешеной скоростью и вскоре был в браузере почти у каждого пользователя. Но кроме этого JavaScript стал пригоден для разработки серверной части. Появился NodeJS, позволяющий запускать на сервере скрипты на JavaScript. Он набрал популярность за счет следующего:

Кроссплатформенность node.js работал на любой операционной системе, в том числе Linux, Windows, MacOS и так далее ...

Единая архитектура Код для сервера и клиента написаны на одном языке. Это позволяет избежать ряда проблем.

Один поток В NodeJS код выполняется в одном потоке, за счет чего он становится проще и его легче разрабатывать. Исчезают многие трудности, которые появляются при работе с многопоточным кодом: не нужно переключаться между потоками, входить в зону блокировок и так далее.

Поддержка сообщества Большинство повседневных задач, как правило, уже решены и доступны в качестве пакета в Node Package Manager (NPM). На данный момент NPM является самым быстро-развивающимся пакетным менеджером.

Таким образом, JavaScript на данный момент можно использовать не только в браузере, но и на сервере. Более того, можно даже делать запросы в базы данных с использованием JavaScript.

1.2. Типы данных

В JavaScript существуют 6 типов данных:

- Числа
- Строки
- Булевы величины
- Неопределённые величины
- Объекты и массивы
- Функции

1.2.1. Числовой тип

Чтобы определить число, в JavaScript используется запись, состоящая из цифр.

```
123      // 123
12.3     // 12.3
```

По умолчанию используется десятичная система счисления, но также можно использовать шестнадцатичную (используется префикс `0x`), восьмеричную (используется префикс `0o`) и двоичную (используется префикс `0b`) системы.

```
0x11     // 17
0b11     // 3
0o11     // 9
```

В JavaScript на хранение числа отводится 64 бита. Казалось бы, максимальное целое число, представимое в JavaScript равно

$$2^{64} = 18446744073709552000.$$

На самом деле это не совсем так. Во-первых, 1 бит используется для хранения знака числа. Несколько бит используются для хранения позиции десятичной точки, поэтому максимальное целое число, представимое в JavaScript, равно:

```
Number.MAX_SAFE_INTEGER  
// 9007199254740991
```

Это число всегда доступно как свойство `MAX_SAFE_INTEGER` объекта `Number`.

Для работы с большими значениями можно использовать экспоненциальную запись числа. Слева от *e* расположено основание экспоненты (любое десятичное число), а справа — степень экспоненты.

```
2.998e8  
// 2.998 × 108  
// 299800000
```

Следует помнить, что вычисления не с целыми числами будут неточными. Также в JavaScript есть два особенных числа:

```
Infinity  
-Infinity
```

Если вычесть из бесконечности единицу, получится по-прежнему бесконечность:

```
Infinity - 1 === Infinity // true
```

`Infinity` ($+\infty$) больше любого другого представимого в JavaScript числа. Аналогичным образом ведет себя `-Infinity`

Кроме того, в JavaScript есть особенное число `NaN` (Not a Number)

```
NaN
```

Это число возникает как результат недопустимых арифметических операций:

```
0/0 // NaN  
Infinity - Infinity // NaN  
'один' / 'два' // NaN
```

1.2.2. Строковый тип

Следующий тип данных — строки. Для того, чтобы представить строку в JavaScript, произвольный текст обрамляется одинарными или двойными кавычками:

```
'строка текста'  
"строка текста"  
' español русский \' '
```

Оба варианта допустимы, однако рекомендуется заранее для себя выбрать предпочтительный тип представления строки и придерживаться ему. Внутри строки можно использовать Unicode-символы, а также символы одинарной или двойной кавычки (для этого их нужно экранировать с помощью обратного слеша).

1.2.3. Логический тип

Логический тип представляет булевы величины (принимают значения true и false).

```
true  
false
```

Самый простой способ получить булеву величину — выполнить операцию сравнения.

1.2.4. Неопределённые величины

Неопределённые величины — самый необычный тип данных в JavaScript.

```
undefined  
null
```

При определении переменной ее значение по умолчанию не определено (undefined):

```
var a;  
console.log(a); // undefined
```

1.2.5. Оператор typeof

Оператор typeof позволяет узнавать тип значения. Результат возвращается в виде строки.

```
typeof 0;    // 'number'  
typeof '0';  // 'string'
```

1.2.6. Преобразование к числу

Для того, чтобы преобразовать явно строку к числу, можно использовать `parseInt` и `parseFloat`. `parseInt` принимает два аргумента.

```
parseInt(string, radix);
```

Первый из них — это строка (если первый аргумент не строка, он неявно приводится к строке при помощи метода `toString`), которую нужно привести к числу, а второй — система счисления.

```
parseInt('17', 10); // 17
parseInt('123');    // 123
parseInt('11', 2);  // 3
```

Если второй аргумент не указан, `parseInt` самостоятельно подбирает систему счисления. Однако, в зависимости от реализации, он может это делать по-разному, поэтому настоятельно рекомендуется указывать систему счисления всегда.

Если функция `parseInt` не может разобрать первый символ строки, то он возвращает `NaN`.

```
parseInt('a1', 10); // NaN
parseInt('2b', 10); // 2
```

Если же строка начинается с числа, за которым идет не число, то `parseInt` возвращает число, которое он смог прочесть.

Функция `parseFloat` является двойственной к функции `parseInt` и позволяет приводить строки к вещественным числам.

```
parseFloat(string);
```

В отличие от `parseInt`, `parseFloat` принимает один аргумент, строку, и работает всегда в десятичной системе счисления:

```
parseFloat('3.14');    // 3.14
parseFloat('314e-2');  // 3.14
parseFloat('a1');      // NaN
```

`parseFloat` может принимать как строки, представляющие числа с использованием точки, так строки, представляющие числа в экспоненциальной записи. Если передать в `parseFloat` строку, которая начинается не с числа, результат будет `NaN`.

1.3. Переменные

1.3.1. Определение переменной

Для того, чтобы определить новую переменную, используется ключевое слово `var` и вслед за ним название переменной.

```
var studentsCount;  
studentsCount = 98;
```

После определения переменной, ее можно использовать, например присвоить значение, как в примере выше.

Можно использовать более короткую запись и присваивать значение в момент объявления переменной:

```
var studentsCount = 98;
```

Кроме того, с помощью оператора «запятая» можно объявить несколько переменных после одного ключевого слова `var`:

```
var studentsCount = 98,  
    language = 'JavaScript';
```

1.3.2. Допустимое имя переменной

В качестве первой буквы переменной можно использовать буквы, символы нижнего подчеркивания и доллара:

a-z _ \$

Однако рекомендуется для первого символа использовать буквы нижнего регистра латинского алфавита. Имя переменной не может начинаться с цифры. Остальные символы в имени переменной могут быть буквами, цифрами, символами подчеркивания и доллара:

a-z 0-9 _ \$

Использовать знак минус (-) в имени переменной нельзя.

В качестве названия переменной нельзя использовать зарезервированные слова:

<code>break</code>	<code>do</code>	<code>try</code>	<code>while</code>
<code>case</code>	<code>else</code>	<code>new</code>	<code>with</code>
<code>catch</code>	<code>finally</code>	<code>return</code>	
<code>continue</code>	<code>for</code>	<code>switch</code>	
<code>debugger</code>	<code>function</code>	<code>this</code>	
<code>default</code>	<code>if</code>	<code>throw</code>	
<code>delete</code>	<code>in</code>	<code>instanceof</code>	
<code>typeof</code>	<code>var</code>	<code>void</code>	

Кроме того, с выходом новой спецификации этот список пополнился:

```
class enum    extends super
const export import
```

1.3.3. Именованние констант

Иногда возникает необходимость в константах, то есть в переменных, значения которых не будут меняться по ходу программы. Чтобы отличать в коде константы от переменных, значение которых не предполагается постоянным, рекомендуется именовать константы буквами верхнего регистра, отделяя слова с помощью символа нижнего подчеркивания:

```
// Переменная
var currentTime;

// Константа
var MILLISECONDS_IN_DAY;
```

1.3.4. Именованние переменных

Рекомендуется придерживаться следующих правил при выборе имени переменной, чтобы сделать код более понятным и читаемым:

- Следует избегать транслита в именах переменных.

```
spisokDruzey; // X
tsena;        // X
```

Обычно такие переменные читаются хуже, а также при использовании внешних библиотек в коде будут присутствовать как переменные, имя которых написано транслитом, так и переменные, имя которых написано английскими словами:

```
friends;      // OK
price;        // OK
```

- Следует избегать слишком коротких и слишком длинных названий переменных:

```
h, w;                // X
friendsListWithNameAndAge; // X
height, width;       // OK
myFriends;           // OK
```

- Следует использовать для именования переменных так называемый camelCase:

```
my_friends;    // X
myFriends;     // OK
```

Если название переменной состоит из нескольких слов, начинайте каждое слово, кроме первого, с заглавной буквы.

- Следует учитывать тип данных, который будет у значения этой переменной, при выборе ее имени. Переменные, которые хранят булево значение, могут начинаться на `is`, а имена переменных, хранящих массивы, — оканчиваться на `s`.

```
isCorrect = true;
totalCount = 47;
friends = [];
```

1.4. Комментарии

Чтобы пояснить некоторые участки кода, можно использовать комментарии: строки, которые игнорируются интерпретатором. В JavaScript доступен строчный вид комментариев:

```
// это короткий комментарий
```

А также его блочный аналог:

```
/* а это длинный комментарий
   написанный в несколько строк */
```

Комментарий может располагаться на отдельной строке, а также заканчивать строку с кодом:

```
/* ах этот длинный комментарий ... */
var weather = 'cold';

console.log(weather); // cold
```

Рекомендуется использовать только строчные комментарии, потому как использование блочных комментариев может привести к следующей ситуации.

```
var weather = 'sunny';

/** ах этот длинный комментарий ... */
```

```
var weather = 'cold';*/  
  
console.log(weather);  
// SyntaxError: Unexpected token *
```

1.5. Операторы

Все операторы имеют приоритет. Это означает, что если несколько операторов используются в одной записи, то сначала выполняются операторы с высшим приоритетом. Далее операторы перечислены в порядке уменьшения приоритета.

1.5.1. Унарные операторы

Унарные операторы, то есть операторы, которые применяются к одному операнду, имеют наивысший приоритет.

++ (инкремент)
-- (декремент)
+ (унарный плюс)
- (унарный минус)
! (логическое НЕ)

Унарный минус меняет значение числовой переменной на противоположенное. Логическое отрицание меняет булево значение true на false и наоборот.

Инкремент (декремент) бывает двух видов:

- Постфиксный инкремент:

```
var a = 1;  
var b = a++; // b === 1, a === 2
```

Сначала производится присваивание, а после этого увеличивается значение переменной на 1.

- Префиксный инкремент:

```
var a = 1;  
var b = ++a; // b === 2, a === 2
```

Сначала увеличивается значение переменной на 1, а после этого производится присваивание.

1.5.2. Бинарные

Бинарные операторы работают с двумя операндами. Наибольший приоритет среди них имеют бинарные арифметические операторы:

- * (умножение)
- / (деление)
- % (остаток от деления)
- + (сложение)
- (вычитание)
- + (сложение строк)

Сложение числовых значений:

$2 + 3 = 5$

Сложение (конкатенация) строк:

```
'«JavaScript - это простой, но ' +  
'изящный язык, который является ' +  
'невероятно мощным для решения ' +  
'многих задач» © Джон Резиг'
```

Дальше, по уменьшению приоритета, идут (бинарные) операторы сравнения:

- < (меньше)
- <= (меньше или равно)
- > (больше)
- >= (больше или равно)
- == (проверка на равенство)
- != (проверка на неравенство)
- === (проверка на идентичность)
- !== (проверка на неидентичность)

Операторы сравнения возвращают булевы величины. Разница между сравнением на равенство и сравнением на идентичность будет обсуждаться позже в рамках курса.

Далее идут логические операторы:

- && (И)
- || (ИЛИ)

Оператор логического И имеет больший приоритет среди логических операторов.

Наименьший приоритет среди бинарных операторов имеют операторы присваивания и присваивания с операцией:

= (присваивание)

*, /=, +=, -=, &=, ^=, |= (присваивание с операцией)

Следующий код демонстрирует смысл присваивания с операцией:

```
var a = 1;
a += 1;
a = a + 1;
```

1.5.3. Условные операторы

Самый простой способ записать условный оператор — использовать ключевое слово `if`, после которого в круглых скобках идет логическое выражение.

```
if (language === 'JavaScript') {
    likes = likes + 1;
} else {
    likes = likes - 1;
}
```

Если это логическое выражение истинно, выполняется код в первых фигурных скобках, иначе будет выполнен код, который записан в фигурных скобках после ключевого слова `else`. При этом `else` и последующая часть являются необязательными.

Другой вид условного оператора — тернарный оператор. Записывается он следующим образом: логическое выражение, после которого идет знак вопроса. Если это выражение истинно, то выполняется код до двоеточия, иначе выполняется выражение, которое записано после двоеточия.

```
likes = language === 'JavaScript' ?
    likes + 1 :
    likes - 1;
```

Оператор `switch-case` также является условным оператором:

```
switch (language) {
    case 'JavaScript':
        likes++;
        break;
    case 'C++':
    case 'Java':
        break;
    default:
        likes--;
}
```

После ключевого слова `switch` в круглых скобках следует выражение, вычисляя которое можно получить некоторое значение. Если это значение совпадает с одним из значений, записанных после ключевого слова `case`, выполняется код этого `case`. Иначе выполняется код, записанный в `default`. Если в `case` встречается ключевое слово `break`, то работа `case`'а прекращается, иначе — происходит проваливание в следующий `case`.

1.6. Точка с запятой

В примерах выше каждая строка заканчивается символом «точка с запятой». Это очень важный символ, который обязан присутствовать в конце каждой строки.

Однако, если его пропустить, ошибки не будет. Это связано с тем, что интерпретатор неявным образом поставит точку запятой. Такое неявное добавление точки с запятой может привести к ошибкам в работе программы:

```
function getTrue() {  
    return true;  
}  
  
getTrue();    // true
```

Теперь, если добавить перенос строки после ключевого слова `return`, казалось бы функция должна работать также.

```
function getTrue() {  
    return  
    true;  
}  
  
getTrue();    // undefined
```

Интерпретатор неявным образом поставил символ «точка с запятой» после слова `return`. В результате функция возвращает `undefined`.

1.7. Строгий режим

Строгий режим появился вместе со спецификацией 5.1. Дело в том, что кроме расширения возможностей языка, в этой спецификации были внесены коррективы, которые нарушают обратную совместимость с написанным ранее кодом.

Чтобы такого не произошло, по умолчанию интерпретатор работает в режиме совместимости с предыдущими спецификациями. Для того, чтобы писать код в соответствии с последней спецификацией, нужно включить строгий режим. Для этого в начале файла или функции следует добавить директиву:

```
'use strict';  
  
// этот код будет работать  
// по современному стандарту ES5
```

После включения строгого режима код начинает вести себя иначе. Без строгого режима можно объявить переменную без использования ключевого слова `var`:

```
text = 'hello';  
  
text; // 'hello'
```

В строгом режиме это приведет к ошибке:

```
'use strict';  
  
text = 'hello'; // ReferenceError:  
                // text is not defined
```

Полный список изменений, которые включаются вместе со строгим режимом, доступен по [ссылке](#).

Следует отметить, что строгий режим нельзя выключить. Поэтому, если строгий режим был включен в начале файла и в коде используются внешние библиотеки, которые не готовы к работе в строгом режиме, это может привести к ошибкам.

Рекомендуется в таких случаях не включать строгий режим глобально, а вместо этого включить его в рамках своих функций.

1.8. Пример запуска

Для начала приведем пример кода для запуска в браузере:

```
console.log('Hello, world!');
```

Данный код выводит в консоль браузера приветственное сообщение. Чтобы запустить программу, нужно перейти в консоль браузера (`Ctrl+Alt+J`) и скопировать строчку кода в строку ввода.

В результате в консоль выведено сообщение, а после этого (так как код ничего не возвращает) — `undefined`.

Для того, чтобы запустить код на сервере, нужно сперва установить [NodeJS](#). После это с использованием любого текстового редактора или IDE нужно создать js-файл со следующим содержимым:

```
// index.js
console.log('Hello, world!');
```

Чтобы запустить этот файл, в консоли терминала используем команду `node`:

```
$ node index.js
```

JavaScript, часть 1: основы и функции

Глава 2

Типы данных (часть 1)

2.1. Строки

Строки полезны для хранения данных, которые можно представить в текстовой форме.

Mozilla Developer Network

2.1.1. Создание строки

Чтобы создать строку, достаточно поместить текст между парой одинаковых кавычек.

```
var emptyString = '';
```

В данном случае была создана пустая строка. Чтобы убедиться в этом, можно проверить свойство `length`, которое всегда содержит длину строки:

```
emptyString.length; // 0
```

Для создания строк можно использовать одинарные кавычки:

```
var russianString = 'строка текста';  
russianString.length; // 13
```

А также — двойные:

```
var russianString = "строка текста";  
russianString.length; // 13
```

2.1.2. Экранирование. Escape-последовательности.

Некоторые символы строки должны быть заэкранированы, то есть записаны как escape-последовательность, которая начинается с обратного слеша:

- Если строка создана при помощи одинарных кавычек, все символы одинарных кавычек в ней должны быть экранированы:

```
var escapeCodesString = 'a\'b' // a'b
escapeCodesString.length; // 3
```

- Символ обратного следа экранируется всегда, чтобы можно было отличить его от экранирования какого-то другого символа:

```
var escapeCodesString = 'a\\b' // a\b
escapeCodesString.length; // 3
```

- Специальные символы записываются при помощи соответствующих им escape-последовательностей. Например, перевод строки записывается при помощи `\n`, а символ табуляции — при помощи `\t`:

```
var escapeCodesString = 'a\n\tb' // a
                                //      b
escapeCodesString.length; // 4
```

2.1.3. Unicode-символы

Строки представляют собой последовательности unicode символов. Важно отметить, что это могут быть любые символы на любом языке:

```
var utf8String = ' español English русский ';
utf8String.length; // 53
```

При этом длина строки высчитывается корректно, какие бы символы из каких языков не использовались бы.

2.1.4. Неизменяемость строк

В JavaScript возможно обращение к символу по индексу:

```
var russianString = 'кот';
russianString[1]; // 'о'
```

При этом перезаписать по индексу символ другим нельзя, так как в JavaScript строки неизменяемые. Причем следующий код выполнится корректно, то есть не вызовет ошибку:

```
russianString[1] = 'и';
russianString; // 'кот'
```

Однако строка в результате не поменяется.

2.1.5. Практический пример работы со строками

Как известно, в Twitter существует ограничение размера сообщений в 140 символов. Пусть требуется реализовать аналогичное ограничение, а в случае, если строка имеет длину больше заданной, обрезать ее.

Например, задана следующая строка:

```
var longString = 'Очевидно проверяется, что математический анализ \
существенно масштабирует интеграл по поверхности, что неудивительно. \
Первая производная, очевидно, позиционирует ортогональный определитель.'
```

Метод `slice(start [, end])` строки возвращает ее часть от позиции `start` до позиции `end` (но не включая ее). Если параметр `end` не указан, извлечение идет до конца строки.

Так можно решить поставленную задачу:

```
var shortString = longString;
if (longString.length > 140) {
    shortString = longString.slice(0, 139) + '...';
}
```

```
shortString; // 'Очевидно проверяется, что математический анализ
// существенно масштабирует интеграл по поверхности, что неудивительно.
// Первая производная, оч...'
shortString.length; // 140
```

Пусть теперь требуется определить, содержит ли строка определенный хеш-тег.

```
var tweet = 'PWA. Что это такое? Третий доклад на WSD в Питере Сергея ' +
    'Густуна #wstdays';
```

Для поиска хеш-тега в строке можно воспользоваться методом `indexOf`.

Метод `indexOf(searchValue[, fromIndex])` возвращает индекс **первого** вхождения указанной в качестве аргумента подстроки или `-1`, если подстрока отсутствует. Параметр `fromIndex` необязательный и задает местоположение внутри строки, откуда начинать поиск.

В частности, так можно найти индекс первого вхождения подстроки с хеш-тегом в строке:

```
tweet.indexOf('#wstdays'); // 65
tweet.indexOf('#fronttalks'); // -1
```

В данном случае получается, что в сообщении есть хеш-тег `#wstdays` и нет `#fronttalks`.

2.1.6. Основные операции со строками

Полный список операций со строками доступен по следующей [ссылке](#).

splice извлекает часть строки и возвращает новую строку.

indexOf возвращает индекс первого вхождения указанного значения.

toLowerCase Приведение строки к нижнему регистру.

trim Удалить пробельные символы с начала и с конца строки.

startsWith Проверяет, начинается ли строка с заданной подстроки.

2.2. Массивы

Массивы являются спископодобными объектами, чьи прототипы содержат методы для операций обхода и изменения массива. Ни размер JavaScript-массива, ни типы его элементов не являются фиксированными.

Mozilla Developer Network

2.2.1. Создание массива

Для создания массива используются квадратные скобки.

```
var emptyArray = []; // Пустой массив
```

Как и у строк, у массивов есть свойство `length` — количество элементов в массиве.

```
emptyArray.length; // 0
```

Можно сразу создать массив из необходимых элементов:

```
var arrayOfNumbers = [1, 2, 3, 4]; // Массив чисел
arrayOfNumbers.length; // 4
var arrayOfStrings = ['a', 'b', 'c']; // Массив строк
arrayOfStrings.length; // 3
```

2.2.2. Основные операции с массивами

- Обращение к элементу массива по индексу производится при помощи квадратных скобок.
- Итерирование по массиву производится с помощью цикла `for`:

```
for (var i = 0; i < tweets.length; i++) {
    var tweet = tweets[i];
    // Что-то делаем с конкретным твитом
}
```

- Добавление элементов (в конец массива) производится при помощи метода `push`:

```
var emptyArray = [];
// Добавляем элементы
emptyArray.push('a');
emptyArray.push('b');
emptyArray; // ['a', 'b']
emptyArray.length; // 2
```

- Метод `pop` «выталкивает» последний элемент массива и возвращает его.

```
// Удаляем последний элемент
emptyArray.pop(); // 'b'
emptyArray; // ['a']

emptyArray.length; // 1
```

Длина массива при этом уменьшается на 1.

- Метод `concat` позволяет объединять массивы. Важно отметить, что этот метод не изменяет исходные массивы, а вместо этого возвращает новый массив.

```
var concatenatedArray =
  ↪ arrayOfNumbers.concat(arrayOfStrings);

concatenatedArray; // [1, 2, 3, 4, 'a', 'b', 'c']

arrayOfNumbers; // [1, 2, 3, 4]
arrayOfStrings; // ['a', 'b', 'c']
```

- Метод `splice(start, deleteCount, ...items)` изменяет содержимое массива, удаляя существующие элементы и/или добавляя новые. Подробнее о нем пойдет речь позже.
- Метод `slice([begin[, end]])` возвращает новый массив, содержащий копию части исходного массива. Если оба параметра не указаны, возвращается копия массива:

2.2.3. Практический пример работы с массивом

На примере массива из 12 строк (твитов) будут рассмотрены основные операции, которые можно производить с массивами. Пусть дан следующий массив:

```
var tweets = [
  'Я и IoT, пятый доклад на WSD в Питере Вадима Макеев
  ↪ #wstdays',
  'Вёрстка писем. Развенчиваем мифы. Четвёртый доклад на
  ↪ WSD в Питере Артура Коха #wstdays',
  'РWA. Что это такое? Третий доклад на WSD в Питере Сергея
  ↪ Густуна #wstdays',
```

```

'Pokémon GO на веб-технологиях, второй доклад на WSD в
→ Питере Егора Коновалова #wstdays',
'Ого сколько фронтендеров. #wstdays',
'<head> - всему голова, первый доклад на WSD в Питере
→ Романа Ганина #wstdays',
'Доброе утро! WSD в Питере начинается через 30 минут:
→ программа, трансляция и хештег #wstdays',
'Наглядная таблица доступности возможностей веб-платформы
→ Пола Айриша: Can I use + StatCounter, от CSS до JS',
'Node.js, TC-39 и модули, Джеймс Снел о проблемах Node.js
→ с асинхронными модулями ES и вариантах выхода из
→ ситуации',
'Всегда используйте <label>, перевод статьи Адама
→ Сильвера в блоге Академии HTML',
'JSX: антипаттерн или нет? Заметка Бориса Сердюка на
→ Хабре',
'Как прятать инлайновые SVG-иконки от читалок, Роджер
→ Йохансен объясняет, зачем это нужно'
];

```

```

tweets.length; // 12

```

Поиск в массиве твитов Чтобы найти в массиве все твиты с тегом #wstdays можно воспользоваться циклом for и проверять, содержится ли хеш тег в очередной строке:

```

var result = [];

for (var i = 0; i < tweets.length; i++) {
    var tweet = tweets[i];

    if (tweet.indexOf('#wstdays') !== -1) {
        result.push(tweet);
    }
}

```

Добавление элементов в копию массива Чтобы создать копию массива, можно использовать метод slice без параметров:

```

var tweetsWithAdv = tweets.slice();

```

Для изменения созданного массива (добавление рекламных твитов) используется метод splice:


```
tweetsWithAdv.splice(4, 0, 'Покупайте наших слонов!');
tweetsWithAdv.splice(9, 0, 'И натяжные потолки тоже у
  ↪ нас отличные!');

tweets.length; // 12
tweetsWithAdv.length; // 14
```

Постраничная навигация по твитам Если в методе `slice` указаны параметры, в результате получается «срез» массива. Это позволяет организовать постраничную навигацию по твитам:

```
var tweetsByPage = tweetsWithAdv.slice(5, 10);
```

2.2.4. Полезные функции для работы с массивами

Полный список функций для работы с массивами можно найти по следующей [ссылке](#).

push Добавляет один или более элементов в конец массива и возвращает новую длину массива.

pop Удаляет последний элемент из массива и возвращает его.

concat Возвращает новый массив, состоящий из данного массива, соединённого с другим массивом.

splice Добавляет и/или удаляет элементы из массива.

slice Извлекает диапазон значений и возвращает его в виде нового массива.

sort на месте сортирует элементы массива и возвращает отсортированный массив.

every проверяет, удовлетворяют ли все элементы массива условию, заданному в передаваемой функции.

some проверяет, удовлетворяет ли хоть какой-нибудь элемент массива условию, заданному в передаваемой функции.

shift удаляет первый элемент из массива и возвращает его значение.

unshift добавляет один или более элементов в начало массива и возвращает новую длину массива.

2.3. Объекты

Список, состоящий из пар с именем свойства и связанного с ним значения, которое может быть произвольного типа.

Mozilla Developer Network

2.3.1. Создание объекта

Объект можно создать с помощью пары фигурных скобок.

```
var emptyObject = {};
```

Так будет создан пустой объект. Также можно создать объект с требуемым набором свойств:

```
var tweet = {
  createdAt: 'Sat Oct 01 12:01:08 +0000 2016',
  id: 782188596690350100,
  text: 'Я и IoT, пятый доклад на WSD в Питере Вадима Макеева
    ↪ #wstdays',
  user: {
    id: 42081171,
    name: 'Веб-стандарты',
    screenName: 'webstandards_ru',
    followersCount: 6443
  },
  hashtags: ['wstdays']
};
```

2.3.2. Основные операции с объектами

Обращение к свойствам объекта Получение значения свойства и присваивание ему значения производится через точечную нотацию:

```
emptyObject.propertyName = 'foo'; // { propertyName: 'foo' }
emptyObject.propertyName; // 'foo'
```

Удаление свойства объекта производится при помощи оператора delete:

```
delete emptyObject.propertyName; // {}
```

Важно понимать, что удаляется один конкретный ключ, а не весь объект.

Обращение к свойствам вложенных объектов также производится при помощи точки:

```
tweet.id; // 782188596690350100
tweet.user.screenName; // 'webstandards_ru'
```

Обращение к свойствам объекта при помощи квадратных скобок:

```
tweet['i' + 'd']; // 782188596690350100
```

Такой способ обращения к значению часто используется, если значение ключа нужно сначала динамически вычислить.

Итерирование по ключам объекта Специальный метод `Object.keys` объекта `Object` возвращает для переданного в качестве единственного аргумента объекта массив его ключей. Это позволяет итерировать по ключам объекта:

```
var keys = Object.keys(tweet);
keys; // ['createdAt', 'id', 'text', 'user', 'hashtags']

for (var i = 0; i < keys.length; i++) {
  var key = keys[i];
  var value = tweet[key];
  // Что-то делаем с ключом и со значением
}
```

Проверка наличия свойства у объекта Для проверки наличия свойства у объекта используется метод `hasOwnProperty`:

```
tweet.hasOwnProperty('text'); // true
tweet.hasOwnProperty('nonExistantProperty'); // false
```

2.4. Функции

Именованный блок кода, который позволяет переиспользовать существующий код.

Может иметь входные параметры и возвращать значение.

Является объектом высшего порядка.

Mozilla Developer Network

2.4.1. Декларация функции. Возвращаемое значение

Пример декларации функции, которая возвращает значение:

```
function getFollowersCount() {  
    return 6443;  
}
```

Функции также могут не возвращать значения:

```
function noop() {  
}
```

В этом случае движок JavaScript неявно добавит возвращение значения `undefined` в конец тела функции:

```
function noop() {  
    return undefined;  
}
```

Все функции в JavaScript, таким образом, возвращают значение: либо то, которое было возвращено явно, либо `undefined`.

2.4.2. Декларация функции. Передача параметров

Функция может также принимать аргументы и работать с ними.

```
function getAuthor(tweet) {  
    return tweet.user.screen_name;  
}
```

Данная функция принимает в качестве параметра объект и возвращает вложенное свойство этого объекта.

Аргументы могут передаваться по значению и по ссылке в зависимости от того, какого они типа:

- Если в качестве аргумента передан примитив, то есть строка (`string`), логическое значение (`boolean`) или число (`number`), аргумент передается по значению.

```

tweet.user.followersCount; // 6443

function incrementFollowersCount(count) {
    count++; // 6444
}

incrementFollowersCount(tweet.user.followersCount);
tweet.user.followersCount; // 6443

```

Как видно из последнего примера, изменения, произведенные с примитивом внутри тела функции, не меняют значение исходной переменной.

- Все сложные типы данных: массивы, объекты, функции и так далее передаются по ссылке.

```

tweet.user; // { id: 42081171, name: 'Веб-стандарты',
// screenName: 'webstandards_ru', followersCount: 6443 }

function incrementFollowersCount(user) {
    user.followersCount++;
}

incrementFollowersCount(tweet.user);
tweet.user; // { id: 42081171, name: 'Веб-стандарты',
// screenName: 'webstandards_ru', followersCount: 6444 }

```

В этом случае изменения, произведенные с аргументом функции, приводят к изменению исходной (переданной) переменной и видны вне тела функции.

2.5. Функций в качестве аргументов

Функции — объекты высшего порядка. Они могут быть переданы в другие функции в качестве аргумента, а так же могут иметь личные свойства, как и другие объекты.

Во всех дальнейших примерах будет использоваться следующий массив твитов:

```
var tweets = [
  { hashtags: ['wstdays'], likes: 16, text: 'Я и IoT, пятый...' },
  { hashtags: ['wstdays', 'mails'], likes: 33, text: 'Вёрстка писем...' },
  { hashtags: ['wstdays'], likes: 7, text: 'PWA. Что это...' },
  { hashtags: ['wstdays', 'pokemongo'], likes: 12, text: 'Pokémon GO на...'
    ↪ },
  { hashtags: ['wstdays'], likes: 15, text: 'Ого сколько фронт...' },
  { hashtags: ['wstdays', 'html'], likes: 22, text: '<head> - всему...' },
  { hashtags: ['wstdays'], likes: 8, text: 'Доброе утро! WSD...' },
  { likes: 9, text: 'Наглядная таблица доступности...' },
  { hashtags: ['nodejs'], likes: 7, text: 'Node.js, TC-39 и модули...' },
  { hashtags: ['html'], likes: 28, text: 'Всегда используйте <label>...' },
  { likes: 18, text: 'JSX: антипаттерн или нет?' },
  { hashtags: ['svg'], likes: 19, text: 'Как прятать инлайновые...' }
];
```

Следует отметить, что хеш-теги твитов содержатся в качестве массива в виде отдельного поля объекта твита.

2.5.1. Итерация по массиву при помощи метода `forEach`

Метод `forEach` применяет передаваемую в качестве его аргумента функцию к массиву. Функция, передаваемая методу `forEach`, сама принимает два аргумента: элемент на текущем шаге итерации и текущий индекс.

Например, задача фильтрации твитов, которые содержат определенный хеш-тег, решается следующим образом:

```
var result;
// Теперь в result лежат отфильтрованные твиты
tweets.forEach(filterWithWstdaysHashtag);
// Выбираем только твиты с хештегом #wstdays
function filterWithWstdaysHashtag(tweet, index) {
  var hashtags = tweet.hashtags;
  if (Array.isArray(hashtags) &&
    ↪ hashtags.indexOf('wstdays') !== -1) {
    result.push(tweet);
  }
}
```

В последнем примере, чтобы проверить, содержится ли хеш тег в твите, сначала с помощью метода `Array.isArray` проверяется, что свойство `tweet.hashtags` является массивом.

```
Array.isArray(hashtags)
```

Поскольку логические операции в JavaScript ленивые, оставшаяся часть условия не выполняется, если `hashtags` не является массивом. Затем (уже известно, что `hashtags` — это массив) вызывается метод массива `indexOf`, который возвращает индекс первого вхождения элемента в массив (или `-1`, если такого элемента не нашлось):

```
Array.isArray(hashtags) && hashtags.indexOf('wstdays')
```

2.5.2. Фильтрация (`filter`) и отображение (`map`)

Метод **`filter`** создаёт новый массив со всеми элементами исходного массива, для которых функция фильтрации возвращает `true`.

Метод **`map`** Создаёт новый массив с результатами вызова указанной функции на каждом элементе данного массива.

Метод **`join`** объединяет все элементы массива в строку. В качестве аргумента передается разделитель.

С помощью методов `filter` и `map` операции над массивами могут обрабатываться в виде цепочки.

Например, в следующем примере сначала выделяются твиты, содержащие определенный хеш-тег (это реализуется с помощью метода `filter` и функции `filterWithWstdaysHashtag`), а затем каждый твит преобразуется в соответствующую ему HTML-строку (для этого используется метод `map` и функция `render`):

```
// Теперь в result лежит HTML с деревом твитов
var result = '<dl>' +
  ↪ tweets.filter(filterWithWstdaysHashtag)
                                     .map(render)
                                     .join('\n') + '</dl>';

// Выбираем только твиты с хештегом #wstdays
function filterWithWstdaysHashtag(tweet, index) {
  var hashtags = tweet.hashtags;
  return Array.isArray(hashtags) &&
    ↪ hashtags.indexOf('wstdays') !== -1;
}
```

```

// Превращаем массив объектов твитов в HTML-строки
function render(tweet, index) {
  return '<dt>' + tweet.text + '</dt>' +
    '<dd>' + tweet.user + '</dd>' +
    '<dd>' + tweet.hashtags.join(', ') + '</dd>';
}

```

После того, как получен массив из HTML-строк, он объединяется в одну строку при помощи метода `join` (в качестве разделителя выбран символ новой строки). Получившаяся строка «оборачивается» в `<dl>...</dl>`.

2.5.3. Метод `reduce` для последовательной обработки массива

Для работы с массивами есть еще один полезный метод.

Метод `reduce` Применяет функцию к аккумулятору и каждому значению массива (слева-направо), сводя его к одному значению.

Результатом работы метода `reduce` является так называемый аккумулятор. На каждом шаге цикла функция обратного вызова (которая передается в качестве первого аргумента) должна возвращать обновленный аккумулятор.

В следующем примере показано вычисление полного количества лайков у всех твитов в массиве. Аккумулятором в данном случае является число лайков.

```

var likesCount = tweets.reduce(getTotalLikes, 0)

function getTotalLikes(acc, item) {
  return acc + item.likes;
}

likesCount; // 194

```

Несколько более сложный пример использования метода `reduce` для получения статистики хеш-тегов:

```

var hashtagsStat = tweets.reduce(flattenHashtags, [])
// ['wstdays', 'wstdays', 'mails', 'wstdays',
  ↪ 'wstdays', 'pokemongo',
// 'wstdays', 'wstdays', 'html', 'wstdays', 'nodejs',
  ↪ 'html', 'svg']

```



```

        .reduce(getHashtagsStats, {});

function flattenHashtags(acc, item) {
    return acc.concat(item.hashtags || []);
}

function getHashtagsStats(acc, item) {
    if (!acc.hasOwnProperty(item)) {
        acc[item] = 0;
    }

    acc[item]++;

    return acc;
}

hashtagsStats; // { html: 2, mails: 1, nodejs: 1,
    ↪ pokemongo: 1, svg: 1, wstdays: 7 }

```

В данном случае используется цепочка из двух `reduce`. В первом `reduce` строится плоский массив со всеми хеш-тегами всех твитов. Этот массив используется во втором `reduce` для получения объекта, ключами которого будут имена хеш-тегов, а значениями — статистика по хеш-тегам.

JavaScript, часть 1: основы и функции

Глава 3

Типы данных (часть 2)

3.1. Методы объекта

3.1.1. Методы объекта

У объекта можно определить произвольное свойство и установить в качестве значения этого свойства функцию. Такое свойство будет называться методом объекта.

Внутри этой функции будет доступно ключевое слово `this`, о котором подробнее речь пойдет позже в данном курсе. Ключевое слово `this` внутри метода объекта ссылается на сам этот объект, а значит с его помощью можно обращаться к вложенным свойствам этого объекта.

```
// Объект с предопределенным набором свойств
var tweet = {
  ...
  likes: 16,
  getLikes: function() {
    return this.likes;
  },
  setLikes: function(value) {
    this.likes = parseInt(value) || 0;
    return this;
  },
  getAuthor: function() {
    return this.user.screenName;
  }
};
```

3.1.2. Цепочка вызовов методов

Метод, в том числе, может возвращать значение этого ключевого слова. Это часто используется на практике в методах, меняющих внутреннее состояние объекта. В примере выше таким методом является метод `setLikes` (задает количество лайков), а метод `getLikes` позволяет получить количество лайков:

```
tweet.getLikes(); // 16
```

Если функция, меняющая состояние объекта, после своего вызова возвращает сам объект, появляется возможность выстраивать цепочку вызовов других методов этого объекта:

```
tweet.setLikes(17) // { ... }  
  .getLikes();    // 17
```

В данном простом примере таким образом получается проверить, что внутреннее состояние объекта действительно изменилось.

3.1.3. Геттеры/сеттеры свойств объекта

Существует альтернативный способ создания методов объектов с помощью так называемых геттеров/сеттеров. Для некоторого свойства могут быть определены:

- функция-геттер вызывается каждый раз при попытке прочитать свойство. Возвращаемое в ходе исполнения этой функции значение как раз и будет считанным значением свойства.
- функция-сеттер вызывается каждый раз при попытке изменить значение свойства, получает в качестве параметра устанавливаемое значение и может изменить внутренние переменные объекта.

Геттер определяется как функция, не принимающая параметров, определяемая с помощью ключевого слова `get`, имя которой является именем свойства, для которого определяется геттер. Сеттер же определяется с помощью ключевого слова `set` и принимает один параметр. Внутри сеттеров и геттеров свойств объекта можно использовать ключевое слово `this` для доступа к внутренним переменным.

В следующем примере определяются геттер и сеттер для свойства `likes`, которые соответственно возвращают и устанавливают значение внутренней переменной `_likes`:

```
// Объект с предопределенным набором свойств  
var tweet = {
```

```

...
_likes: 16,
get likes() {
    return this._likes;
},
set likes(value) {
    this._likes = parseInt(value) || 0;
},
getAuthor: function() {
    return this.user.screenName;
}
};

```

При обращении к свойству объекта, для которого определен геттер, вызывается соответствующая функция, результат выполнения которой и будет считанным значением:

```
tweet.likes; // 16
```

При этом обращение к свойству объекта происходит аналогично тому, как происходило обращение к статическому свойству.

При попытке присвоить значение свойству, для которого определен сеттер, вызывается функция-обработчик, первым аргументом которой передается значение справа от оператора присваивания и которая уже устанавливает внутренние свойства объекта.

```

tweet.likes = 17;
tweet.likes; // 17

```

3.2. Обработка исключений

3.2.1. Основные подходы к обработке ошибок

Есть несколько шаблонов для работы с ошибками:

- Явная обработка нештатных ситуаций и граничных условий.
- Работать по так называемому контракту.

3.2.2. Обработка ошибок по контракту. Исключения

При работе по контракту устанавливаются ограничения на принимаемые функцией параметры. Если они не выполнены, функция «кидает исключение» какого-то типа. В свою очередь, код, который использует функцию, должен будет обработать эту исключительную ситуацию.

Можно переписать следующим образом пример выше:

```

// Объект с predefined набором свойств
var tweet = {
  ...
  _likes: 16,
  get likes() {
    return this._likes;
  },
  set likes(value) {
    var likes = parseInt(value);

    if (isNaN(likes) || likes < 0) {
      throw new TypeError('Передано неверное значение');
    }

    this._likes = likes;
  },
  getAuthor: function() {
    return this.user.screenName;
  }
};

```

Здесь внутри сеттера явно производится проверка, что устанавливаемое значение является неотрицательным числом. В случае, если это не так, срабатывает исключение.

3.2.3. Обработка ошибок по контракту. Оператор try-catch.

Код, который вызвал функцию, может обрабатывать возможные исключительные ситуации с помощью оператора try-catch:

```

try {
  tweet.likes = 'foo';
} catch (e) {
  if (e instanceof TypeError) {
    tweet.likes = 0;
  }
  console.error(e);
}

tweet.likes; // 0

```

Оператор try-catch состоит из двух частей:

- Блок `try`, в котором расположен код, который может бросить исключение.
- Блок `catch`, который принимает на вход исключение, если оно будет выброшено, и позволяет его обработать.

Внутри блока `catch` удобно использовать оператор `instanceof` для проверки типа исключения, которое было выброшено. Таким образом, появляется возможность отделить известные ошибки, которые были задокументированы разработчиком, от остальных.

3.2.4. Объект исключения. `TypeError`

Объект исключения содержит несколько полезных свойств:

- Свойство `name`, которое содержит имя типа ошибки:

```
e.name; // 'TypeError'
```

- Свойство `message`, которое содержит сообщение об ошибке:

```
e.message; // 'Передано неверное значение'
```

- Свойство `stack`, которое содержит стек вызовов функций, которые привели к ошибке:

```
e.stack;
// TypeError: Передано неверное значение
//      at Object.set likes [as likes]
//      ↪ (<anonymous>:10:15)
//      at <anonymous>:18:15
```

В `stack` также содержится имя и сообщение об ошибке.

3.3. Сравнение и приведение объектов

JavaScript — язык с динамической типизацией. Иногда возникают ситуации, когда функция возвращает значение произвольного типа, причем тип возвращаемого значения заранее не известен.

В этом разделе речь пойдет о том, как можно адаптировать различные типы данных так, чтобы их можно было друг с другом сравнивать.

3.3.1. Сравнение объектов

Пусть есть объект `panda`, который обладает методами `valueOf` и `toString`. Метод `valueOf` возвращает код `unicode`-символа с пандой, а метод `toString` возвращает HTML-entity с кодом для вставки панды в HTML страницу.

```
var panda = {
  valueOf: function() {
    return 128060;
  },
  toString: function() {
    return '\\&\\#x1F43C;';
  }
}
```

Также есть код «поросенка» в `unicode`-таблице, заданный примитивом (целым числом):

```
var pigCode = 128055;
```

Ставится вопрос, каков будет результат сравнения объекта с пандой и числа:

```
panda == pigCode; // ???
```

Сравнение будет происходить следующим образом:

- К левому операнду применяется внутренний метод `isPrimitive`:

```
isPrimitive(panda); // false
```

Метод `isPrimitive` проверяет, является ли операнд примитивом. В данном случае `panda` — объект и не является примитивом.

- К правому операнду применяется внутренний метод `isPrimitive`:

```
isPrimitive(pigCode); // true
```

В данном случае `pigCode` является примитивом (числом).

- Проверяется, имеет ли операнд, который не является примитивом, метод `valueOf`:

```
typeof panda.valueOf === 'function'; // true
```

В данном случае этот метод действительно есть.

- После этого у не-примитива вызывается метод `valueOf` и результат сравнивается со значением примитива:

```
panda.valueOf() === pigCode; // false
```

По сути будут сравниваться два целых числа:

```
128060 === 128055; // false
```

Таким образом, результат сравнения объекта panda и кода pigCode:

```
panda == pigCode; // false
```

Полный алгоритм можно найти по ссылке: [Абстрактный Алгоритм Эквивалентного Сравнения](#).

Важно понимать, что при сравнении двух переменных сложных типов (два объекта или два массива), которые передаются по ссылке, реально сравниваются всегда ссылки на области памяти. Таким образом, операция сравнения двух сложных типов вернет истину только в том случае, если внутренние ссылки обоих объектов ссылаются на один и тот же объект в памяти.

В частности, два идентичных (например, пустых), но разных объекта при сравнении друг с другом будут давать false:

```
{ } == { }; // false
```

Если же сравнить объект panda с кодом панды:

```
var pandaCode = 128060
```

То получится, как в этом не сложно убедиться, true:

```
panda == pandaCode; // true
```

3.3.2. Приведение объекта к числу

```
var panda = {  
  valueOf: function() { return 128060; },  
  toString: function() { return '&#x1F43C;'; }  
}
```

Способы приведения переменной к числу:

- Передача этой переменной внутрь конструктора Number:

```
Number(panda); // 128060
```

Это самый простой способ преобразования переменной к числу. В результате будет вызван метод valueOf объекта и получено целое число.

- Использование унарного сложения. Если поставить знак плюс перед переменной, будет вызвана принудительная конвертация переменной в целое число, а именно вызывается метод `valueOf`:

```
+panda; // 128060
```

Не рекомендуется пользоваться таким методом на практике, так как такая конвертация очень неявна и очень легко спутать операцию конкатенацию или сложение нескольких переменных с операцией унарного сложения для преобразования типов данных.

- Двойной бинарный сдвиг — еще один способ преобразования переменной к числу:

```
~~panda; // 128060
```

- Операция сравнения с целым числом. При этом вызывается метод `valueOf`.

```
panda == 128060; // true
```

Следует отметить, что при операции строгого сравнения (`===`), в отличие от нестрогого сравнения (`==`), переменные сравниваются всегда только в рамках одного типа данных.

```
panda === 128060; // false
```

- Функция `parseInt`/`parseFloat` пытается привести объект сначала к строке, а потом распарсить ее к целому числу/числу с плавающей точкой.

```
parseInt(panda); // NaN  
parseFloat(panda); // NaN
```

В обоих случаях получается результат `Not A Number`. Это связано с тем, что сначала происходит преобразование к строке, а в данном случае нет строчного представления, которое могло бы быть преобразовано к строке.

- Явный вызов метода `valueOf`:

```
panda.valueOf(); // 128060
```

3.3.3. Приведение объекта к строке

Способы приведения объекта к строке:

- Передача этой переменной внутрь конструктора String:

```
String(panda); // '🐼'
```

В результате будет вызван метод toString объекта.

- Конкатенация переменной со строкой:

```
' ' + panda; // '128060'
```

В результате вызывается метод valueOf, но преобразуется в строку.

- Преобразование к строке также происходит при нестрогом сравнении переменной со строкой.

```
panda == '128060'; // true
```

При этом также вызывается метод valueOf и его значение преобразуется к строке. Строгое сравнение объекта со строкой всегда вернет false:

```
panda === '128060'; // false
```

- Явный вызов метода toString:

```
panda.toString(); // '🐼'
```

Следует отметить, что только два из представленных метода сконвертировали объект в строку согласно желаемой логике (используя toString). Чтобы избежать путаницы, рекомендуется не определять методы valueOf и toString одновременно.

Например, пусть дан объект (без метода valueOf):

```
var panda = {  
  toString: function() { return '🐼'; }  
}
```

При использовании описанных выше способов приведения к строке:

- Передача этой переменной внутрь конструктора String:

```
String(panda); // '🐼'
```

- Конкатенация переменной со строкой:

```
' ' + panda; // '🐼'
```

- Преобразование при нестрогом сравнении со строкой.

```
panda == '🐼'; // true
```

- Преобразование при строгом сравнении со строкой.

```
panda === '🐼'; // false
```

- Явный вызов метода toString:

```
panda.toString(); // '🐼'
```

Как видно из данного примера, по возможности следует определять только метод toString, так как его поведение более предсказуемо.

3.4. Скрытые методы

3.4.1. Понятие скрытого метода

В данном разделе будет рассмотрен еще один способ создания свойств у объектов, который позволяет более глубоко настраивать их поведение.

Объект «панда» из предыдущего примера:

```
var panda = {  
  valueOf: function() { return 128060; },  
  toString: function() { return '🐼'; }  
}
```

Если применить метод Object.keys к объекту «панда», будет выведен массив из имен ключей созданного объекта.

```
Object.keys(panda); // ['valueOf', 'toString']
```

Этот массив состоит из двух строк, как и ожидалось.

Если же создать пустой массив и точно также передать его в метод Object.keys, результатом будет пустой массив:

```
var emptyObject = {};  
Object.keys(emptyObject); // []
```

Это вполне предсказуемо, ведь объект был создан пустой.

С помощью оператора typeof можно определить тип свойства panda.valueOf, чтобы лишний раз убедиться, что это метод и тип значения данного свойства — функция:

```
typeof panda.valueOf === 'function'; // true
```

Если же сделать такую же проверку для пустого объекта:

```
typeof emptyObject.valueOf === 'function'; // true
```

то окажется, что метод `valueOf` определен на пустом объекте, не видим в списке ключей объекта, но может быть вызван.

3.4.2. Метод `Object.defineProperty`

Метод `Object.defineProperty` позволяет объявлять свойства объекта и принимает три аргумента:

1. Объект, в котором нужно определить новое свойство.
2. Название свойства, которое будет создано.
3. Объект конфигурирования, который может содержать следующие ключи:

value Значение, которое будет установлено в качестве свойства объекта.

атрибут writable определяет, является ли свойство записываемым.

атрибут enumerable определяет, видно ли свойство в списке ключей объекта, получаемом при помощи `Object.keys`.

атрибут configurable определяет,
Значения параметров `writable`, `enumerable` и `configurable` по умолчанию — `false`.

Одним из ключей объекта конфигурирования является `value`, по которому содержится значение, которое будет установлено в качестве свойства объекта.

3.4.3. Атрибут `enumerable`

В рассматриваемом примере с объектом «panda» создаются два метода, а значит в качестве `value` должна быть указана функция. Чтобы скрыть свойство из списка ключей объекта нужно указать значение атрибута `enumerable` равным `false`.

```
var panda = {};
```

```
Object.defineProperty(panda, 'valueOf', {
```

```

    value: function() {
        return 128060;
    },
    writable: true,
    enumerable: false,
    configurable: true
});

Object.defineProperty(panda, 'toString', {
    value: function() {
        return '&#x1F43C;';
    },
    writable: true,
    enumerable: false,
    configurable: true
});

```

3.4.4. Атрибут writable

Пусть дан пустой объект, в котором предполагается хранить твит:

```
var tweet = {};
```

Новое свойство text этого объекта можно определить при помощи метода defineProperty с значением writable равным false:

```

Object.defineProperty(tweet, 'text', {
    value: 'Я и IoT, пятый доклад на #wstdays в Питере',
    writable: false
})

```

Убедиться, что свойство создано можно при помощи метода getOwnPropertyDescriptor

```

Object.getOwnPropertyDescriptor(tweet, 'text');
// { value: 'Я и IoT, пятый доклад на #wstdays в Питере',
//   writable: false,
//   enumerable: false,
//   configurable: false }

```

В результате свойство может быть прочитано, но не может быть изменено:

```

tweet.text; // 'Я и IoT, пятый доклад на #wstdays в Питере'
tweet.text = 'Вёрстка писем. Развенчиваем мифы. ...
↪ #wstdays';
tweet.text; // 'Я и IoT, пятый доклад на #wstdays в Питере'

```

При этом (не в строгом режиме) такой код ошибку не вызывает, но и не меняет значение свойства также.

3.4.5. Атрибут configurable

```
var tweet = {};
```

Теперь можно определить свойство со значением configurable

→ равным `false`:

```
Object.defineProperty(tweet, 'text', {  
  value: 'Я и IoT, пятый доклад на #wstdays в Питере',  
  configurable: false  
});
```

При помощи метода `getOwnPropertyDescriptor` можно убедиться,

→ что свойство создано:

```
Object.getOwnPropertyDescriptor(tweet, 'text');  
// { value: 'Я и IoT, пятый доклад на #wstdays в Питере',  
//   writable: false,  
//   enumerable: false,  
//   configurable: false }
```

```
tweet.text; // 'Я и IoT, пятый доклад на #wstdays в  
→ Питере'
```

Можно проверить, что свойство существует с помощью `hasOwnProperty`:

```
tweet.hasOwnProperty('text'); // true
```

Однако удалить свойство не получается:

```
delete tweet.text; // false
```

Оператор `delete` всегда возвращает результат, успешно ли произведено удаление. Удаление является частным случаем конфигурирования свойства, поэтому свойство не получилось удалить.

Более того, можно непосредственно проверить, что свойство все еще есть:

```
tweet.text; // 'Я и IoT, пятый доклад на #wstdays в  
→ Питере'
```

```
tweet.hasOwnProperty('text'); // true
```

3.4.6. Заморозка объекта

Пусть из внешнего API был получен объект:

```
var tweet = {
  ...
  likes: 16,
  getLikes: function() {
    return this.likes;
  }
};
```

Этот объект может быть изменен в ходе исполнения кода. На практике же часто требуется обеспечить его неизменяемость. Именно для этой цели и служит заморозка.

Проверить, является ли объект замороженным, можно с помощью метода `Object.isFrozen`:

```
Object.isFrozen(tweet); // false
```

В данном случае объект не заморожен и может быть изменен.

В частности, если посмотреть дескриптор любого из его свойств:

```
Object.getOwnPropertyDescriptor(tweet, 'likes')
// { value: 16,
//   writable: true,
//   enumerable: true,
//   configurable: true }
```

окажется, что оно может быть изменено (`writable` равно `true`) и удалено (`configurable` равно `true`).

Теперь можно применить заморозку при помощи `Object.freeze`:

```
Object.freeze(tweet);
```

Теперь объект является замороженным:

```
Object.isFrozen(tweet); // true
Object.getOwnPropertyDescriptor(tweet, 'likes')
// { value: 16,
//   writable: false,
//   enumerable: true,
//   configurable: false }
```

А его свойства стали неизменяемыми и неконфигурируемыми. Заморозка никак не повлияла на видимость свойств (не меняет атрибут `enumerable`).

В результате свойство `likes` не получается ни удалить, ни изменить:

```
tweet.likes = 17;
tweet.likes; // 16

delete tweet.likes; // false
```

3.5. Объект Даты

Часто на практике приходится работать с датами. В JavaScript существует специальный объект Date для работы с датами.

Создание объекта-даты с помощью конструктора Date:

- Вызов конструктора Date без параметров создает объект с текущей датой в системном часовом поясе:

```
new Date(); // Mon Oct 17 2016 09:37:20 GMT+0500  
→ (YEKT)
```

- Создание даты по строке с датой:

```
tweet.createdAt; // 'Sat Oct 01 12:01:08 +0000 2016'  
// Пытаемся сконвертировать строку в дату  
new Date(tweet.createdAt); // Sat Oct 01 2016  
→ 17:01:08 GMT+0500 (YEKT)
```

- Создание даты из UNIX Timestamp:

```
new Date(1475323268000); // Sat Oct 01 2016 17:01:08  
→ GMT+0500 (YEKT)
```

- Создание даты из набора параметров:

```
new Date(2016, 9, 1, 17, 1, 8); // Sat Oct 01 2016  
→ 17:01:08 GMT+0500 (YEKT)
```

Количество параметров должно быть больше или равно двум и они должны быть целыми числами.

Объект Date в JavaScript был импортирован из языка Java образца 1995 года, когда JavaScript был создан, и с тех пор практически не менялся. Поэтому он содержит много странностей, которые следует просто запомнить. Например, второй параметр при создании даты из набора параметров, месяц, начинается с нуля, а третий (день) — с единицы.

Чтобы получить значение UNIX Timestamp из даты можно воспользоваться методом valueOf:

```
(new Date(2016, 9, 1, 17, 1, 8)).valueOf(); //  
→ 1475323268000
```

Текущее значение UNIX Timestamp можно получить с помощью статического метода Date.now():

```
Date.now(); // 1476680054602
```

Полный список методов доступен по ссылке: [Date](#)

3.6. Библиотека математических функций и констант Math

Библиотека Math представляет собой множество математических функций и констант. В частности в Math определены следующие функции:

Math.random Генерирует случайное число от 0 до 1

```
Math.random(); // 0.4468546273336771
```

Math.min Принимает на вход неограниченное количество чисел и определяет меньшее из них:

```
Math.min(1, 5); // 1
```

Math.max Определяет большее из чисел:

```
Math.max(1, 5, 10); // 10
```

Math.round Округляет число до ближайшего целого:

```
Math.round(2.7); // 3
```

```
Math.round(2.3); // 2
```

Math.floor Округляет число до целого в меньшую сторону

```
Math.floor(2.7); // 2
```

```
Math.floor(2.3); // 2
```

Math.ceil Округляет число до целого в большую сторону

```
Math.ceil(2.7); // 3
```

```
Math.ceil(2.3); // 3
```

Math.log Возвращает натуральный (по основанию e) логарифм числа

```
Math.log(10); // 2.302585092994046
```

Math.pow Возвращает основание, возведённое в степень

```
Math.pow(2, 5); // 32
```

Math.sin Возвращает синус угла в радианах

```
Math.sin(1); // 0.8414709848078965
```

Math.tan Возвращает тангенс угла в радианах

```
Math.tan(1); // 1.5574077246549023
```

Полный список доступных математических функций и констант можно посмотреть по ссылке: [Math](#).

3.7. Регулярные выражения

Регулярные выражения в JavaScript имеют стандартный PCRE-синтаксис. Подробнее прочитать про PCRE (Perl Compatible Regular Expressions) можно по [ссылке](#). Также полезно прочесть [руководство по регулярным выражениям](#) применительно именно к JavaScript.

Рекомендуется прибегать к использованию регулярных выражений только в крайних случаях, поскольку написание регулярных выражений представляет собой достаточно трудную задачу, а также сами регулярные выражения очень тяжело читать, если не иметь в этом опыта.

С помощью регулярного выражения можно выделить все хеш-теги в твитах:

```
tweet.text; // 'Node.js, и модули, Джеймс о проблемах  
→ Node.js #nodejs #modules'
```

Хеш-теги могут быть как на русском языке, так и на английском. В любом случае хеш-тег начинается с символа `#`. Регулярное выражение будет иметь вид:

```
/#[a-z0-9]+/
```

Оно начинается с символа решетки, за которым следует более чем один символ, который может быть от `a` до `z` или числом.

Проверить, содержится ли указанное регулярное выражение в строке с помощью метода `test` регулярного выражения:

```
/#[a-z0-9]+/gi.test(tweet.text); // true
```

Флаг `g` обозначает глобальное сопоставление, а `i` — игнорирование регистра при сопоставлении.

В случае, если в твите нет хеш-тегов:

```
var tweetWithoutHashtag; // 'Я и IoT, пятый доклад на WSD  
→ в Питере'
```

Будет иметь место следующий результат:

```
/#[a-z0-9]+/gi.test(tweetWithoutHashtag); // false
```

Пусть дан объект со свойством `text`, которое все также содержит текст твита:

```
var tweet = {  
  text: 'Node.js, и модули, Джеймс о проблемах Node.js  
    ↪ #nodejs #modules #модули'  
};
```

Требуется написать метод, который бы на лету оборачивал бы все хештеги твита в ссылки.

Для этого можно воспользоваться методом `Object.defineProperty`, чтобы определить геттер для свойства `linkify`:

```
Object.defineProperty(tweet, 'linkify', {  
  get: function() {  
    return this.text.replace(/#[a-z0-9]+/gi, '<a  
    ↪ href="$1">$1</a>');  
  }  
});
```

Функция обработчик заменяет все вхождения регулярного выражения с помощью метода `text.replace`. Первым аргументом метода `replace` является регулярное выражение, а вторым — шаблон, согласно которому должна производиться замена.

Создание геттеров при помощи `defineProperty` более явно, нежели, чем с помощью ключевых слов `get` и `set`, а также может быть более тонко сконфигурировано.

При помощи метода `Object.getOwnPropertyDescriptor` можно получить текущие параметры свойства `linkify`:

```
Object.getOwnPropertyDescriptor(tweet, 'linkify');  
// { get: [Function: get],  
//   set: undefined,  
//   enumerable: false,  
//   configurable: false }
```

Геттер определен, но не сеттер — поэтому свойство `linkify` неизменяемо.

При попытке вызвать метод `linkify` можно заметить, что русские теги не обрабатываются:

```
tweet.linkify;  
// 'Node.js, и модули, Джеймс о проблемах Node.js  
// <a href="$1">$1</a> <a href="$1">$1</a> #модули'
```

Чтобы исправить это, можно дополнить регулярное выражение:

```
return this.text.replace(
  /#[a-z0-9a-я]+/gi,
  '<a href="$1">$1</a>'
);
```

Теперь все хеш-теги обрабатываются, но вместо ссылок на места хеш-тегов попадают ссылки с placeholder'ами \$1 и так далее:

```
tweet.linkify;
// 'Node.js, и модули, Джеймс о проблемах Node.js
// <a href="$1">$1</a> <a href="$1">$1</a> <a
↪ href="$1">$1</a>'
```

Для того, чтобы решить эту проблему, регулярное выражение следует обернуть в круглые скобки.

```
return this.text.replace(
  /([a-z0-9a-я]+)/gi,
  '<a href="$1">$1</a>'
);
```

Таким образом определяется захватывающая группа, которая захватывает все символы, которые попали в скобки. После этого \$1 в шаблоне заменяется на содержимое первой захватывающей группы:

```
tweet.linkify;
// 'Node.js, и модули, Джеймс о проблемах Node.js
// <a href="#nodejs">#nodejs</a> <a
↪ href="#modules">#modules</a> <a
↪ href="#модули">#модули</a>'
```

Если определены две захватывающие группы, вторая захватывающая группа доступна в шаблоне по \$2:

```
return this.text.replace(
  /([a-z0-9a-я]+)([a-z0-9a-я]+)/gi,
  '<a href="$2">$1</a>'
);
```

Получается:

```
tweet.linkify;
// 'Node.js, и модули, Джеймс о проблемах Node.js
// <a href="nodejs">#nodejs</a> <a
↪ href="modules">#modules</a> <a
↪ href="модули">#модули</a>'
```

Во многих справочниках по регулярным выражениям можно увидеть применение классификатора `\w`, который соответствует любому цифробуквенному символу, включая нижнее подчеркивание. Эквивалентен `[A-Za-z0-9_]`. Судя по всему, буквами считаются только символы латинского алфавита.

```
return this.text.replace(
    /(#([\w]+))/gi,
    '<a href="$2">$1</a>'
);

tweet.linkify;
// 'Node.js, и модули, Джеймс о проблемах Node.js
// <a href="nodejs">#nodejs</a> <a
  ↪ href="modules">#modules</a> #модули'
```

JavaScript, часть 1: основы и функции

Глава 4

Функции

В программировании, согласно принципу DRY (Don't Repeat Yourself), принято выделять повторяющиеся блоки кода в функции и давать им осмысленные имена. Это позволяет поддерживать читаемость кода, а также упростить жизнь программиста: при изменении логики программы или исправлении бага, достаточно внести исправление только в одном месте кода.

Кроме этого, функции позволяют организовать рекурсивный вызов. Также в JavaScript функции выполняют особую роль — они создают области видимости переменных.

4.1. Работа с аргументами функции

4.1.1. Аргументы функции

Покажем, как работать с аргументами функций в JavaScript, на примере функции для нахождения минимума двух чисел:

```
function min(a, b) {  
    return a < b ? a : b;  
}
```

Эта функция принимает два аргумента и работает следующим образом:

```
min(2, 7);    // 2  
min(3, 4, 2); // 3  
min(13);     // undefined
```

Если вызвать эту функцию от трех аргументов, ошибки не произойдет, в отличие от многих других языков программирования. При этом последний аргумент в этом случае будет просто проигнорирован.

Если же вызвать эту функцию с одним аргументом, происходит следующее. Значение `b` оказывается не определено (`undefined`). При сравнении

любого числа и `undefined` результат всегда будет `false`, поэтому в результате выполнения функции будет возвращено значение `b`, то есть `undefined`.

Сделать так, чтобы можно было вызывать функцию от одного аргумента (и функция возвращала этот аргумент), можно банально инвертируя условие в условном операторе:

```
function min(a, b) {  
    return a > b ? b : a;  
}
```

В таком случае функция по прежнему будет возвращать минимум, если ей переданы два аргумента, и возвращать значение единственного аргумента, если передан только один:

```
min(2, 7);    // 2  
min(13);     // 13
```

Однако так сделать получается далеко не всегда. В общем случае можно проверить явно, передан аргумент или нет. Для этого достаточно сравнить этот аргумент и `undefined`.

```
function min(a, b) {  
    if (b === undefined) {  
        return a;  
    }  
  
    return a < b ? a : b;  
}
```

В данном случае, если при явной проверке оказывается, что аргумент `b` не был передан, функция возвращает значение `a`. Иначе — используется старая логика.

```
min(2, 7);    // 2  
min(13);     // 13
```

Такой подход также работает.

4.1.2. Значения по умолчанию

При помощи оператора `ИЛИ` можно задать значение по умолчанию для аргумента. Такой подход позволяет компактнее реализовать функцию из предыдущего примера:

```
function min(a, b) {
    b = b || Infinity;

    return a < b ? a : b;
}
```

Если **b** передан, то в качестве **b** будет использоваться переданное значение. Если же нет, то **b** будет равен `undefined` и значение `b || Infinity` вернет бесконечность.

```
min(2, 7);    // 2
min(13);      // 13
```

Такое использование оператора ИЛИ опасно. Например, в следующем примере рассматривается функция для расчета стоимости товара, которая принимает два параметра — цену и количество единиц товара. Если второй параметр не задан, следует считать, что количество единиц товара равно одному.

```
function getCartSum(price, count) {
    count = count || 1;

    return price * count;
}
```

Функция работает вполне ожидаемым образом:

```
getCartSum(27.70, 10); // 277
getCartSum(49.90);     // 49.9
```

Но в случае, когда количество товара 0, функция ведет себя не так, как надо:

```
getCartSum(99999, 0); // 99999 ???
```

Дело в том, что 0 неявно приводится к `false`, поэтому значение `0 || 1` будет 1.

Чтобы пример работал правильно, значение по умолчанию для параметра следует устанавливать после явного сравнения с `undefined`:

```
function getCartSum(price, count) {
    if (count === undefined) {
        count = 1;
    }

    return price * count;
}
```


Теперь функция работает корректно во всех случаях:

```
getCartSum(27.70, 10); // 277
getCartSum(49.90);     // 49.9
getCartSum(99999, 0);  // 0
```

4.1.3. Именованные аргументы

Еще один способ работы с аргументами — использование именованных аргументов. Пусть дана функция для вычисления индекса массы тела:

```
function BMI(params) {
    var height = params.height;

    return params.weight / (height * height);
}
```

Эта функция, строго говоря, принимает один аргумент, а нужные значения извлекаются из переданного в качестве этого аргумента объекта. При вычислении индекса массы тела необходимо знать рост человека и его вес, поэтому вызов этой функции будет выглядеть следующим образом:

```
BMI({ weight: 60, height: 1.7 }) // 20.7
```

Такой подход обладает рядом преимуществ:

- Подходит для случаев, когда есть несколько необязательных аргументов
- Не важен порядок аргументов
- Неограниченное число аргументов
- Легко рефакторить код: можно легко добавить или удалить аргумент из цепочки вызовов.

Также есть несколько недостатков:

- Неявный интерфейс: не читая код функции, невозможно понять, какие аргументы нужно передать, чтобы она отработала правильно.
- Неудобно работать с аргументами внутри самой функции, потому что к ним приходится получать доступ через передаваемый объект.

4.1.4. arguments

Еще один способ работы с аргументами функции заключается в использовании объекта `arguments`.

Согласно [статье Arguments object с MDN](#), объект `arguments` - это подобный массиву объект, который содержит аргументы, переданные в функцию. У этого объекта есть свойство `length`, которое содержит количество переданных аргументов, а также можно обратиться по индексу к каждому конкретному аргументу:

```
function example() {
    arguments[1]; // 12
    arguments.length; // 2
}

example(3, 12);
```

В следующем примере определена функция `sum`, которая складывает два числа. Для работы с аргументами используется объект `arguments` и оператор ИЛИ.

```
function sum() {
    var a = arguments[0] || 0;
    var b = arguments[1] || 0;

    return a + b;
}
```

Если значение аргумента не было передано, для него, в результате использования оператора ИЛИ, устанавливается значение по умолчанию, равное нулю.

```
sum(3, 12);      // 15
sum(45);         // 45
sum(2, 4, 8);    // 6
```

Если же функция была вызвана от трех аргументов, последний аргумент игнорируется и возвращается сумма первых двух.

Чтобы подсчитать сумму всех переданных чисел, следует использовать свойство `length` объекта `arguments` для реализации следующего цикла:

```
function sum() {
    var sum = 0;

    for(var i = 0; i < arguments.length; i++) {
```

```

        sum += arguments[i];
    }

    return sum;
}

```

В результате функция будет работать в случае произвольного числа аргументов:

```
sum(2, 4, 8);    // 14
```

Объект `arguments` — не массив, но может быть приведен к массиву с помощью метода `slice`, заимствованного у массива. Метод `call` позволяет вызвать заимствованный метод от лица объекта `arguments`.

```

function sum() {
    var args = [].slice.call(arguments);

    return args.reduce(function (sum, item) {
        return sum + item;
    });
}

```

Переменная `args`, таким образом, будет содержать массив, у которого будет доступен метод `reduce`. Функция также будет работать правильно:

```
sum(2, 4, 8);    // 14
```

Метод Call

Разберем подробнее работу метода `call`. Например, если вызвать метод `slice` от массива, будет создана копия исходного массива. Кроме прямого вызова, для вызова метода `slice` можно использовать метод `call`, передавая исходный массив в качестве единственного аргумента. В этом случае также будет создана копия исходного массива.

```

function example() {
    [1, 2].slice();           // [1, 2]
    [].slice.call([3, 4]);    // [3, 4]

    [].slice.call(arguments); // [5, 6]
}

example(5, 6);

```

Кроме массивов метод `slice` может принимать массивоподобные объекты, каким и является объект `arguments`.

4.2. Объявление функции

4.2.1. function declaration

Существует несколько способов объявления функции. Первый способ уже встречался ранее и называется function declaration.

```
// function declaration
function add(a, b) {
  return a + b;
}
```

В этом случае после ключевого слова function указывается ее имя.

4.2.2. function expression

Существует и альтернативный способ, который называется function expression:

```
// function expression
var add = function (a, b) {
  return a + b;
}
```

В этом способе значение функции присваивается некоторой переменной.

4.2.3. Отличия

Оба этих способа можно использовать, но следует учитывать, что их поведение несколько отличается. В случае использования function declaration вызов функции может быть до момента ее объявления:

```
add(2, 3); // 5

function add(a, b) {
  return a + b;
}
```

В случае же function expression это приводит к ошибке:

```
add(2, 3); // TypeError

var add = function (a, b) {
  return a + b;
}
```

4.2.4. Named function expression

Можно объединить эти два способа и получить третий способ объявления функции — named function expression.

```
var factorial = function inner(n) {  
    return n === 1 ?  
        1 : n * inner(n - 1);  
}
```

В этом случае имя функции, которое указано после ключевого слова `function`, будет доступно только внутри функции, а по имени переменной, в которую присваивается значение функции, функция будет доступна только снаружи:

```
typeof factorial; // 'function'  
typeof inner;    // ReferenceError
```

Имя функции `inner` не доступно снаружи функции и попытка определения ее типа приводит к ошибке интерпретатора. Но она доступна внутри самой функции, в чем можно убедиться непосредственно вызывая ее:

```
factorial(3);    // 6
```

4.2.5. Конструктор Function

Существует еще один, достаточно экзотический, способ объявления функции с помощью конструктора `Function`:

```
var add = new Function('a', 'b', 'return a + b');
```

Сначала перечисляются через запятую как строки имена аргументов функции, а последним аргументом — тело функции, также в виде строки.

Созданную таким образом функции также без проблем можно использовать:

```
add(2, 3);    // 5
```

Применяется такой способ редко, обычно когда код функции генерируется «на лету», то есть когда код функции генерируется другим кодом. Использовать такой способ объявления, вообще говоря, нежелательно. Во-первых, становится сложно ориентироваться в коде. Во-вторых, интерпретатор не сможет оптимизировать код созданной так функции, а значит функция может работать значительно медленнее, чем в случае, если бы она была объявлена иным способом.

4.3. Область видимости

4.3.1. Глобальный объект

Любая переменная или функция, которая была объявлена не в теле другой функции, объявлена в глобальной области видимости.

```
var text = 'Привет'; // { text, greet }
                    //
function greet() {    //
}                    //
```

Переменная `text` и функция `greet` оказываются объявленными в глобальной области видимости и доступны на протяжении всего кода. Получить значение переменной `text` можно через свойство `text` объекта `global`:

```
global.text; // 'Привет'
```

4.3.2. Создание области видимости

Новую область видимости можно создать при помощи функции. При объявлении функции `greet` создается новая область видимости, в которую помещаются все аргументы функции и объявленные в ней переменные.

```
function greet() {      // { greet }
  var text = 'Привет';  // { text }
  text; // 'Привет'     //
}                       //
                        //
text; // ReferenceError: //
      // text is not defined
```

В этом примере в глобальной области видимости оказывается функция `greet`, а в области видимости функции — переменная `text`. Переменная `text` перестает быть доступной после того, как функция завершает свое выполнение.

4.3.3. Нет блочной области видимости

Согласно ECMAScript 5.1, область видимости создается только функцией. То есть переменные, которые были объявлены в блоке кода, например внутри условного оператора, будут доступны за пределами этого блока.

```
function greet() {
  if (true) {
    var text = 'Привет';
  }

  text; // 'Привет'
}
```

```
// { greet }
//   { text }
//
//
//
//
//
```

4.3.4. Вложенные функции

Функции, объявленные внутри других функций, называются вложенными функциями.

```
function greet() {
  var text = 'Привет';

  function nested() {
    text; // 'Привет'
  }
}
```

```
// { greet }
//   { text, nested }
//
//     { }
//
//
//
//
```

При этом переменные из области видимости родительской функции становятся доступными дочерней.

4.3.5. Затенение

Если в родительской функции и в дочерней объявить переменные с одинаковыми именами, будет иметь место затенение.

```
function greet() {
  var text = 'Привет';

  function nested() {
    var text = 'Пока';
    text; // 'Пока'
  }

  text; // 'Привет'
}
```

```
// { greet }
//   { text: Привет, nested }
//
//     { text: Пока }
//
//
//
//
//
//
```

В области видимости дочерней функции по имени `text` доступна переменная со значением «Пока», а в области видимости родительской — переменная со значением «Привет».

4.4. Всплытие

При обращении к переменной до момента ее объявления не возникает ошибки. Механизм, позволяющий это, называется всплытие.

Выполнение кода можно условно разделить на две части:

- Инициализация: Интерпретатор просматривает весь код на предмет объявления функций и переменных:
 - function declaration
 - var

Это и называется всплытием.

- Собственно выполнение

Функции и переменные «всплывают» немного по-разному.

```
add(2, 3);                                // { add: function }

function add(a, b) {
    return a + b;
}
```

Если функция add была объявлена через function declaration, то после инициализации в add будет лежать функция, которая может быть вызвана до момента своего объявления.

```
add(2, 3);    // 5

function add(a, b) {
    return a + b;
}
```

Если же функция была объявлена через function expression, до момента объявления в add будет находиться значение undefined.

```
add(2, 3);                                // { add: undefined }

var add = function (a, b) {
    return a + b;
}
```

Использовать функцию до момента ее объявления не получится, поскольку в этом случае операция «круглые скобки» применяется к undefined, что приводит к TypeError:


```
add(2, 3); // TypeError
```

```
var add = function (a, b) {  
    return a + b;  
}
```

В переменной add функция окажется только в момент присваивания.

```
add(2, 3); // TypeError
```

```
var add = function (a, b) {  
    return a + b;           // { add: function }  
}
```

Всплытие переменных работает в пределах области видимости. Это проиллюстрировано на следующем примере:

```
function greet(){           // { greet: function }  
    ↪ undefined }           // { greet: function, text:  
    var text = "Привет";     // { greet: function, text:  
    ↪ "Привет" }  
}                             // { greet: function }  
                             // { greet: function }
```

В глобальной области видимости доступна только функция greet. Внутри функции всплывает переменная text, которая до момента присваивания имеет значение undefined. После завершения функции переменная text перестает быть доступной.

4.4. Замыкание

Замыкание — это функция со всеми ее внешними переменными, к которым она имеет доступ.

При объявлении новых переменных выделяется новый участок памяти. Все переменные, на которые никто не ссылается, сборщик мусора JavaScript может вычистить с помощью счетчика ссылок.

Внутри функции `makeCounter` мы имеем доступ к переменной `currentCount`, поэтому счетчик ссылок на эту переменную не ноль. Однако и за пределами этой функции мы по-прежнему продолжаем ссылаться на эту переменную. Мы делаем это неявно через функцию `Counter`, которая имеет доступ к `currentCount`. В этом и есть смысл замыкания.

```
1  function makeCounter() {  
2      var currentCount = 0; // { currentCount: 1 }  
3      return function () {  
4          return currentCount++;  
5      };  
6  }  
7  
8  var counter = makeCounter(); // { currentCount: 1 }
```

`makeCounter` в качестве результата возвращает новую функцию. В JavaScript область видимости создается функциями. При этом внутри этой функции мы обращаемся к переменной `currentCount`.

Ищем `currentCount` в области видимости функции, которую возвращаем. Не находим ничего и идем в родительскую функцию. Там объявлена переменная `currentCount`, к ней мы имеем доступ. Таким образом, вызывая функцию `Counter`, мы будем увеличивать счетчик внешней переменной на 1.

```
1  function makeCounter() {  
2      var currentCount = 0;  
3      return function () {  
4          return currentCount++;  
5      };  
6  }  
7  
8  var counter = makeCounter();
```

```
1  // { makeCounter } // 1  
2  // { currentCount } // 2  
3  //  
4  //  
5  // { } // 3  
6  //  
7  //  
8  //
```

Однако если мы позволим функцию `makeCounter` еще раз, мы получим новую функцию, которая ссылается уже на новую переменную `currentCount` вне зависимости от первой переменной.

```
1  var counter = makeCounter();
2  counter(); // 0
3  counter(); // 1
4  counter(); // 2
5
6  var yetAnother = makeCounter();
7  yetAnother(); // 0
```

Объявим функцию `greet`, которая принимает переменную `name` и возвращает новую функцию. Поскольку переменная `name` является внешней по отношению к этой функции, имеет место замыкание. Вызывая функцию `greet` с некоторой строкой, например, строкой ('мир!'), мы получаем новую функцию `helloWorld`, которая при вызове возвращает строку "Привет, мир!"

```
1  function greet(name) {
2      return function () {
3          return 'Привет,' + name;
4      }
5  }
6
7  var helloWorld = greet('мир!');
8  helloWorld(); // "Привет, мир!"
```

4.4.1. Модуль

Определим в нашем коде две функции. Функцию `format`, которая преобразует переданную дату в строку. И функцию `getStringDate`, которая определяет `Date`, если не определено текущим временем, и возвращает время в виде строки. Если мы вызовем `getStringDate`, не передав туда аргументы, мы получим текущую дату в виде строки.

```

1  function format(date) {
2      return date.ToGMSstring()
3  }
4
5  function getDateString(date){
6      date = date || new Date();
7      return format(date);
8  }

```

Если кто-то случайно переопределит функцию `format`, функция `getDateString` будет испорчена. Вместо того, чтобы возвращать текущую дату в виде строки, она будет возвращать новую строку, определенную в новой функции `format`. Чтобы защитить поведение функции `getDateString`, мы сможем воспользоваться `pattern module`. Для того чтобы определить модуль, мы воспользуемся самовызывающейся функцией. Функцию `format` и `getDateString` мы обернем в новую функцию, которую сразу же и вызовем. Из этой функции мы будем возвращать функцию `getDateString` и складывать в переменную с одноименным названием. Для того чтобы показать, что это самовызывающаяся функция, до ключевого слова `function` ставится открывающаяся скобка.

```

1  var getDateString = (function () {
2      function format(date) {
3          return date.toGMTString()
4      }
5
6      return function getDateString(date) {
7          date = date || new Date();
8          return format(date);
9      }
10 }());

```

Функции `format` и `getDateString` находятся внутри новой самовызывающейся функции: портить их значения нельзя.

Для того чтобы написать самовызывающуюся функцию, мы должны определить ее в режим `function expression`: чтобы интерпретатор отличил это от `function declaration`, мы ставим открывающуюся круглую скобочку до ключевого слова `function`. Закрывающуюся круглую скобочку мы можем поставить как после вызова самовызывающейся функции, так и до ее вызова.

```
1  (function() {  
2  }());  
3  
4  (function() {  
5  }());
```

С использованием самовывзывающихся функций мы можем реализовать pattern module.

- Область видимости в JavaScript ...
- Замыкания в JavaScript
- Замыкания, область видимости
- Лексическая область видимости

Глава 5

Контекст исполнения. Объект `this`

5.1. Ключевое слово `this`

5.1.1. Ключевое слово `this` в различных языках программирования

Ключевое слово `this` встречается в различных языках программирования. Например, в языке C# его можно использовать следующим образом. Объявляется класс `User` (Пользователь), который состоит из приватного поля `age` и публичного метода `ShowAge`:

```
class User
{
    private int age = 24;

    public void ShowAge()
    {
        Console.WriteLine(this.age);
    }
}

static void Main()
{
    User mike = new User();

    mike.ShowAge(); // 24
}
```

Метод `ShowAge` выводит на консоль приватное поле `age`. Для этого он

обращается к нему через ключевое слово `this`.

Аналогичный пример можно рассмотреть в языке программирования Java:

```
public class User {
    private int age = 24;

    public void showAge() {
        System.out.println(this.age);
    }
}

public static void main(String []args){
    User mike = new User();

    mike.showAge();
}
```

Также, как и в C#, обращение к приватному полю `age` происходит через ключевое слово `this`.

5.1.2. Свойства `this`

Во всех рассмотренных языках программирования `this` обладает рядом свойств:

- `this` является ключевым словом языка.
- Следовательно, `this` нельзя перезаписать, то есть в него нельзя положить другое значение каким-либо способом.
- `this` всегда ссылается на текущий объект.

5.1.3. Ключевое слово `this` в JavaScript

Аналогичный пример можно написать на JavaScript. Для этого объявим функцию `User`, которая будет возвращать объект. Этот объект состоит из двух полей: поля `age` (аналог свойства) и поля `showAge` (аналог метода).

```
function User () {
    return {
        age: 24,

        showAge: function () {
            console.log(this.age);
        }
    }
}
```



```

    }
  }
}

var mike = new User();

mike.showAge(); // 24

```

В методе `showAge` обращение к свойству `age` происходит также с помощью ключевого слова `this`.

Такое поведение полностью аналогично другим языкам программирования. Однако, в отличие от них, в JavaScript ключевое слово `this` обладает уникальным свойством. А именно `this` в JavaScript можно использовать и за пределом объекта.

Если в консоли браузера вызвать свойство `innerWidth` у ключевого слова `this`, находясь в глобальной области видимости, будет возвращена ширина текущего окна браузера:

```
this.innerWidth; // 1280
```

В интерпретаторе NodeJS, если вызвать свойство `process`, у которого, в свою очередь, вызвать свойство `version`, будет возвращена текущая версия NodeJS:

```
this.process.version; // v7.0.0
```

Таким образом, ключевое слово `this` может быть использовано в глобальной области видимости за пределом какого-либо объекта и это работает корректно.

5.2. Контекст исполнения

5.2.1. Область видимости (повторение)

Ранее была рассмотрена такая тема, как лексическая область видимости. Для того, чтобы понять, как работает ключевое слово `this`, следует углубить эти знания и рассмотреть, что такое контекст исполнения.

Напомним, что любая функция или переменная, которые описаны не в рамках другой функции, попадают в глобальную область видимости:

```

function sum(a, b) {
  return a + b;
}
// Область видимости:
// { sum }
// { a, b }

```

```

}
sum(1, 2);

```



А переменные и функции, которые описаны в рамках другой функции, попадают в область видимости последней функции.

```

function sum(a, b) {
  return a + b;
}

sum(1, 2);

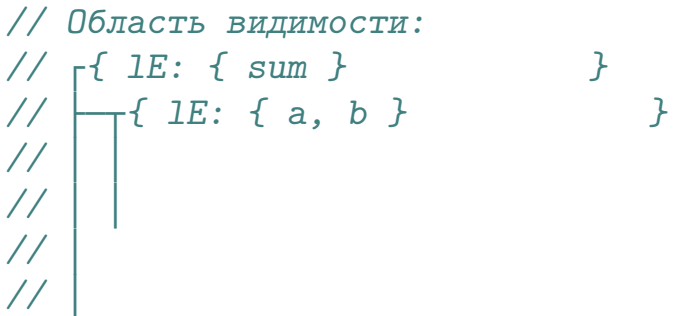
```

// Область видимости:

```

// { lE: { sum } }
// { lE: { a, b } }

```



5.2.2. Контекст исполнения

Контекст исполнения содержит область видимости. Далее область видимости кратко будет обозначаться с помощью аббревиатуры lE (lexical environment). Кроме области видимости, контекст исполнения содержит ключевое слово `this`, которое определяется в момент интерпретации участка кода.

```

function sum(a, b) {
  return a + b;
}

sum(1, 2);

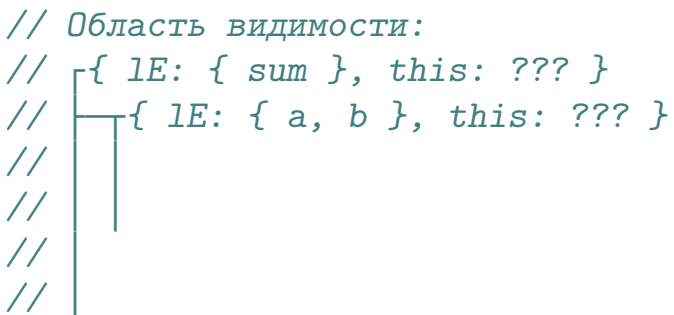
```

// Область видимости:

```

// { lE: { sum }, this: ??? }
// { lE: { a, b }, this: ??? }

```



Значение `this` зависит от следующих факторов:

1. Типа участка кода
2. Как мы попали на этот участок кода
3. Режимы работы интерпретатора

5.2.3. Значение `this` в глобальном участке кода

Как уже было сказано, значение `this` в контексте исполнения зависит от типа участка кода.

Например, если в браузере обратиться к полю `innerWidth` ключевого слова `this`, будет получено некоторое значение, для определенности 1280.

```
this.innerWidth;    // 1280
window.innerWidth;  // 1280
```

Это значение равно ширине окна браузера. Точно также это значение можно получить, если обратиться к полю `innerWidth` глобального объекта `window`.

В интерпретаторе NodeJS обращение к полю `process.version` позволяет получить строку с текущей версией NodeJS:

```
this.process.version;    // "v7.0.0"
global.process.version;  // "v7.0.0"
```

То же самое значение можно получить, если обратиться к полю `process.version` объекта `global`.

Многие, наверняка, использовали `console.log` для вывода на консоль нужные значения.

```
console.log('Hello!');
```

В этом случае на самом деле вызывается свойство `console.log` глобального объекта:

```
global.console.log('Hello!');
this.console.log('Hello!');
```

Тот же самый эффект можно получить, вызвав метод `console.log` у ключевого слова `this`.

Более того, во всех предыдущих примерах `this` равен глобальному объекту:

```
this === global; // true
```

5.2.4. Значение `this` в модуле Node.js

Участок кода может быть не только глобальным, но и быть написанным в рамках модуля NodeJS. Например, пусть дан следующий файл:

```
// year-2016.js

module.exports.days = 366;
```

Это простейший модуль, в рамках которого будут описываться свойства 2016 года. Для того, чтобы свойства стали доступными извне, они должны быть экспортированы: необходимо положить свойства в объект `module.exports`.

Оказывается, что сократить запись можно, если записывать свойства в ключевое слово `this`:

```
this.isLeapYear = true;
```

Это аналог `module.exports` и обе записи работают одинаково.

Для того, чтобы использовать свойство извне модуля, например в файле `index.js`, нужно сперва импортировать модуль:

```
// index.js

var year2016 = require('./year-2016');

year2016.days; // 366;
year2016.isLeapYear; // true;
```

Импортирование модуля производится при помощи вызова функции `require`. На вход функции передается путь до нашего модуля, а в результате будет возвращен как раз тот самый объект, который был экспортирован из модуля.

После этого через обращение к свойствам этого объекта можно получить нужные значения.

5.2.5. Значение `this` при простом вызове функции

Еще одним фактором, который влияет на значение ключевого слова `this`, является то, как именно произошел переход к участку кода.

Например, пусть объявлена функция `getSelf`, которая возвращает текущее значение `this`:

```
function getSelf() {
    return this;
}
```

```
getSelf(); // global
```

Возвращаемое этой функцией значение всегда будет равно значению `this` на том участке кода, на котором она была вызвана. То есть, если функция была вызвана в глобальном контексте, она вернет глобальный объект: `global` (в случае NodeJS) или `window` (в случае работы в браузере).

Если описать эту функцию в контексте NodeJS модуля и вызвать внутри модуля:

```
// year-2016.js

module.exports.days = 366;

function getSelf() {
```

```
        return this;
    }

    getSelf(); // { days: 366 }
```

функция вернет экспортируемое этим модулем значение.

5.2.6. Значение `this` при вызове метода объекта

Еще один способ вызвать функцию — вызвать ее как метод объекта.

```
var block = {
    innerHeight: 300,

    getHeight: function () {
        return this.innerHeight;
    }
}
```

Здесь объект `block` состоит из двух полей: поля `innerHeight` и поля `getHeight`, которое содержит метод, возвращающий значение высоты. Внутри метода значение высоты берется из ключевого слова `this`.

Важно понимать, что значение ключевого слова `this` определяется не в момент определения функции, а в момент ее вызова. Если функция `getHeight` вызывается от объекта `block`, значение `this` будет равно объекту `block`:

```
var block = {
    innerHeight: 300,

    getHeight: function () {
        return this.innerHeight;
    }
}

block.getHeight(); // 300
```

В данном случае будет получено значение 300.

5.2.7. Заимствование метода

Если же сперва функция `getHeight` будет положена в некоторую переменную, то при вызове через эту переменную, значение ключевого слова `this` будет равно глобальному объекту:

```

var block = {
  innerHeight: 300,

  getHeight: function () {
    return this.innerHeight;
  }
}

var getHeight = block.getHeight;
getHeight(); // 1280

```

При работе в браузере значение `this` будет `window` и значение поля `innerHeight` будет взято из объекта `window`. Такое поведение может поначалу сбивать с толку, однако оно получило широкое распространение и называется «заимствование методов».

5.2.8. Заимствование метода. Метод `call`

Чтобы вызвать метод одного объекта в контексте другого объекта, можно использовать метод `call` функции.

Согласно [Function.prototype.call\(\) - JavaScript | MDN](#), метод `call()` вызывает функцию с указанным значением `this` и индивидуально предоставленными аргументами.

Этот метод функции имеет следующую сигнатуру:

```
fun.call(thisArg, arg1, arg2, ...)
```

Первым аргументом передается контекст, а остальные аргументы являются аргументами, с которыми функция будет вызвана.

Например, пусть даны два объекта `mike` (содержит свойство `age` и метод `getAge`) и `anna` (содержит только свойство `age`):

```

var mike = {
  age: 24,

  getAge: function () {
    return this.age;
  }
}

var anna = {
  age: 21
}

```

Значение ключевого слова `this` определяется в момент вызова функции, а не в момент объявления. Поэтому можно воспользоваться методом `getAge` Михаила, чтобы узнать возраст Анны, через `call`:

```
mike.getAge.call(anna); // 21
```

Таким образом, в качестве значения ключевого слова `this` в методе `getAge` будет объект `anna` и, следовательно, будет получено значение свойства `age` объекта `anna`.

На практике заимствование методов используется для того, чтобы превратить массивоподобный объект `arguments` в массив. Для этого заимствуется метод `slice` массива и вызывается в контексте `arguments`:

```
function func() {  
    var args = Array.prototype.slice.call(arguments);  
}
```

Таким образом, в переменной `args` окажется массив.

5.2.9. Заимствование метода. Метод `apply`

Другой метод для работы с контекстом исполнения — это метод `apply`:

Согласно [Function.prototype.apply\(\) - JavaScript | MDN](#), метод `apply()` вызывает функцию с указанным значением `this` и аргументами, представленными в виде массива.

Он ведет себя аналогичным образом, но имеет другую сигнатуру:

```
fun.apply(thisArg, [arg1, arg2]);
```

Метод `apply` всегда принимает два аргумента. Первый аргумент будет использован в качестве ключевого слова `this`. Второй аргумент содержит в себе массив аргументов, с которыми будет вызвана функция `fun`.

Можно продемонстрировать работу метода `apply` на примере функции `min` из библиотеки `Math` (математическая библиотека):

```
Math.min(4, 7, 2, 9); // 2
```

Пусть нужно найти минимум среди элементов некоторого массива. Если передать массив в качестве единственного аргумента, желаемый результат получен не будет:

```
var arr = [4, 7, 2, 9];  
Math.min(arr); // NaN
```

Функция `Math.min`, на самом деле, приводит каждый аргумент к числу, в том числе и переданный в качестве единственного аргумента массив будет приведен к значению `NaN`. Соответственно, результатом выполнения также будет `NaN`.

Чтобы переписать пример правильно, можно воспользоваться методом `apply`:

```
Math.min.apply(Math, arr); // 2
```

Здесь `min` вызвана в контексте `Math` с аргументами из `arr`. Но, поскольку `min` в своей реализации не использует ключевое слово `this`, в качестве `this` можно передать `null`:

```
Math.min.apply(null, arr); // 2
```

5.2.10. Callback

Еще один способ вызова функции — вызов функции как коллбэк:

```
var person = {
  name: 'Sergey',
  items: ['keys', 'phone', 'banana'],

  showItems: function () {
    this.items.map(function (item) {
      return this.name + ' has ' + item;
    });
  }
}
```

В данном случае объект `person` состоит из трех полей: свойств `name` и `items`, а также метода `showItems`.

Как уже много раз упоминалось, значение `this` определяется в момент вызова функции. Пусть метод `showItems` вызывается как метод объекта `person`:

```
person.showItems();
```

В этом случае значение первого `this` будет `person`. В качестве второго ключевого слова `this` будет подставлен объект `global`, поскольку контекст никак явным образом не задан. А значит поле `name` будет браться не у объекта `person`, а у объекта `global`.

Если `name` в `global` не определено, результат будет следующий:

```
'undefined has keys'
'undefined has phone'
'undefined has banana'
```


Такой результат не совсем тот, который требуется.

Существует несколько способов исправить эту проблему. Самый простой из них — сохранить контекст исполнения в некоторую переменную:

```
var person = {                                     // { person }
  name: 'Sergey',                                  //
  items: ['keys', 'phone', 'banana'],              //
                                                    //
  showItems: function () {                         // { _this }
    var _this = this;                              //
                                                    //
    this.items.map(function (item) {               // { item }
      return _this.name+' has '+item;              //
    });                                              //
  }                                                  //
}                                                    //

person.showItems();                                //
```

Callback переписывается таким образом, чтобы обращение происходило не к ключевому слову `this`, а к сохраненному контексту `_this`.

Каждый раз при вызове функции-callback'a переменная `_this` сперва ищется в области видимости callback'a. После того, как найти `_this` там не удалось, она ищется в области видимости родительской функции, то есть в области видимости `showItems`. Значение переменной `_this` как раз такое, какое нужно — объект `person`, а значит `_this.name` вернет правильный результат.

Это пример использования замыкания. В итоге получается желаемый результат:

```
'Sergey has keys'
'Sergey has phone'
'Sergey has banana'
```

Описанная проблема встречается каждый раз, когда нужно обратиться к ключевому слову `this` внутри callback'a, который передан в метод `map`. В JavaScript есть способ сохранить контекст исполнения без создания дополнительных конструкций: контекст исполнения callback'a можно передать в качестве второго аргумента метода `map`.

То есть можно переписать код так:

```
var person = {
  name: 'Sergey',
  items: ['keys', 'phone', 'banana'],
```

```

    showItems: function () {
        this.items.map(function (item) {
            return this.name + ' has ' + item;
        }, this);
    }
}

person.showItems();

```

В этом случае будет получен желаемый результат, поскольку `this` внутри `showItems` и внутри `callback`'а будет равен `person`.

5.2.11. Метод `bind`

Однако не все функции, которые работают с `callback`'ами, принимают в качестве одного из аргументов контекст выполнения `callback`'а. В таком случае можно воспользоваться методом `bind()`.

Метод `bind()` создаёт новую функцию, которая при вызове устанавливает в качестве контекста выполнения `this` предоставленное значение.

<...>

[Function.prototype.bind\(\) - JavaScript | MDN](#)

Метод `bind`, в отличие от методов `call` и `apply`, не вызывает функцию, а возвращает новую. Сигнатура метода `bind` следующая:

```
fun.bind(thisArg, arg1, arg2, ...);
```

Первым аргументом передается контекст исполнения новой функции, а остальные — задают аргументы, с которыми она будет вызвана.

Таким образом, пример можно переписать так:

```

var person = {
    name: 'Sergey',
    items: ['keys', 'phone', 'banana'],

    showItems: function () {
        this.items.map(function (item) {
            return this.name + ' has ' + item;
        }).bind(this));
    }
}

person.showItems();

```

5.3. myBind

Чтобы лучше понять, как работает функция `bind`, рассмотрим ее возможную реализацию:

```
Function.prototype.myBind = function(_this) {  
    var fn = this;  
    var args = [].slice.call(arguments, 1);  
  
    return function () {  
        var curArgs = [].slice.call(arguments);  
  
        return fn.apply(_this, args.concat(curArgs));  
    };  
};
```

Первым аргументом функция `bind` принимает контекст выполнения новой функции и сохраняется в переменной `_this`. В переменной `fn` сохраняется исходная функция. В переменной `args` будет лежать массив аргументов, с которыми вызвали функцию `bind`. Для этого массивоподобный объект `arguments` превращается при помощи заимствования метода `slice` в массив.

В результате возвращается новая функция. Следует обратить внимание, что вызвать новую функцию также можно с аргументами, поэтому исходная функция будет вызвана с объединенным набором аргументов.

5.4. Частичное применение

При помощи метода `bind` можно реализовать так называемое частичное применение.

Например, функция `pow` из библиотеки `Math` возвращает первое переданное как аргумент число, возведенное в степень второго числа.

```
Math.pow(2, 3); // 8  
Math.pow(2, 10); // 1024
```

Если требуется возводить в различную степень только число 2, можно написать функцию `binPow`, которая получается из функции `Math.pow` частичным применением:

```
var binPow = Math.pow.bind(null, 2);
```

Теперь в результате вызова функции `binPow` число 2 будет возводиться в переданную в качестве аргумента степень:

```
binPow(3); // 8  
binPow(10); // 1024
```

5.5. Режим работы интерпретатора

Последний фактор, который влияет на значение ключевого слова `this` — это режим работы интерпретатора. По умолчанию включен режим совместимости, а строгий режим можно включить, указав специальную директиву.

В режиме обратной совместимости, если вызывается функция, внутри которой используется `this`, значение `this` совпадает со значением контекста, внутри которого была вызвана функция.

```
function getSelf(){  
    return this;  
}
```

```
getSelf(); //global
```

В данном примере функция вызывается в глобальном контексте, поэтому внутри этой функции значение `this` равняется `global`.

Для того, чтобы вызвать функцию в строгом режиме, в начале файла или в начале функции нужно добавить директиву `'use strict'`;

```
function getSelf(){  
    'use strict';  
  
    return this;  
}
```

```
getSelf(); //undefined
```

В строгом режиме значение ключевого слова `this` будет `undefined`.

5.6. eval

Функция `eval` используется в JavaScript для того, чтобы интерпретировать код, который написан в виде строки.

```
var temperature = 12;  
  
eval('temperature + 5'); // 17
```

В строке можно использовать один или несколько операторов, перечисленных через точку с запятой, а также обращаться к внешним переменным. В том числе, можно использовать ключевое слово `this`.

```

var person = {
  name: 'Sergey',

  showName: function () {
    return eval('this.name');
  }
}

```

Внутри метода showName происходит обращение к ключевому слову this внутри строки, которая передается в eval. Значение this в данном случае совпадает с контекстом исполнения, в котором был вызван eval.

Это означает, что если showName будет вызван в контексте person, то this будет равен person.

```

person.showName(); // Sergey

```

Если же функцию eval сначала положить в некоторую переменную, например evil, поведение будет совершенно другим: в качестве this всегда будет глобальный объект.

```

var person = {
  name: 'Sergey',

  showName: function () {
    var evil = eval;

    return evil('this.name');
  }
}

person.showName(); // ''

```

В данном случае результат работы showName будет пустая строка.