Почему повторное введение? Потому что JavaScript известен тем, что является самым неправильно понятым языком программирования в мире. Его часто называют игрушкой, но под слоем обманчивой простоты ожидают мощные языковые возможности. В настоящее время JavaScript используется невероятным количеством высококлассных приложений, показывая, что углублённое знание этой технологии является важным навыком для любого веб или мобильного разработчика.

Было бы полезно начать с истории языка. JavaScript был создан в 1995 Бренданом Айком, инженером в компании Netscape. Первый релиз состоялся вместе с выходом браузера Netscape 2 в начале 1996 года. Сначала язык назывался LiveScript, но затем был переименован в связи с маркетинговыми целями, чтобы сыграть на популярности языка Java компании Sun Microsystem — несмотря на это языки практически не имеют ничего общего друг с другом. Так было положено начало путаницы между этими языками.

Чуть позже Microsoft выпустила очень похожий и практически совместимый язык JScript, который шёл вместе с IE3. Через пару месяцев Netscape отправил язык в Ecma International, Европейскую организацию занимающуюся стандартами, которая выпустила первую версию стандарта ECMAScript в 1997. Стандарт получил значимое обновление в ECMAScript edition 3 в 1999, и остается самым стабильным до сегодняшнего дня. Четвертая версия была отклонена, из-за проблем с усложнениями в языке. Многие вещи из четвертого издания послужили основой для стандарта ECMAScript 5 (декабрь 2009) и ECMAScript 6 (июнь 2015).

На заметку: Далее по тексту мы будем называть язык ECMAScript как "JavaScript".

В отличие от большинства языков, JavaScript не следует концепции ввода (input) и вывода (output). Он спроектирован таким образом, чтобы запускаться как язык сценариев, встроенный в среду исполнения. Самая популярная среда исполнения это браузер, однако интерпретаторы JavaScript присутствуют и в Adobe Acrobat, Photoshop, Yahoo!'s Widget engine, и даже в серверном окружении, например node.js.

Описание

JavaScript является объектно-ориентированным языком, имеющий типы и операторы, встроенные объекты и методы. Его синтаксис происходит от языков Java и C, поэтому много конструкций из этих языков применимы и к JavaScript. Одним из ключевых отличий JavaScript является отсутствие классов, вместо этого функциональность классов осуществляется прототипами объектов (смотрите ES6 Classes) . Другое главное отличие в том, что функции это объекты, в которых содержится исполняемый код и которые могут быть переданы куда-либо, как и любой другой объект.

Начнём с основы любого языка: с типов данных. Программы на JavaScript оперируют значениями, и все эти значения принадлежат к определенному типу. Типы данных в JavaScript:

Числа
Строки
Логические типы
Функции
Объекты
Символы (новый тип из шестой редакции)
Да, еще Undefined и Null, которые немного обособлены. И Массивы, которые являются особым видом объектов. А также Даты и Регулярные выражения, тоже являющиеся объектами. И, если быть технически точным, функции это тоже особый вид объекта. Поэтому схема типов выглядит скорее так:

Числа
Строки
Логические типы
Символы (новый тип из шестой редакции)
Объекты
Функции
Массивы
Даты
Регулярные выражения
Null
undefined
Также есть несколько встроенных типов Ошибок. Чтобы было проще, рассмотрим подробнее первую схему.

Числа
Числа в JavaScript — это "64-битные значения двойной точности формата IEEE 754", согласно спецификации. Это имеет интересные последствия. В JavaScript нет такой вещи, как целое число, поэтому с арифметикой нужно быть начеку, если вы привыкли к вычислениям в языках C или Java. Взгляните на пример:

0.1 + 0.2 == 0.30000000000000004
На практике целые значения это 32-битные целые (и хранятся таким образом в некоторых браузерных реализациях), что может быть важно для побитовых операций.

Поддерживаются стандартные арифметические операторы, включая сложение, вычитание, остаток от деления и т.д. Есть ещё встроенный объект, который я забыл упомянуть, называемый Math, который содержит более продвинутые математические функции и константы:

```
Math.sin(3.5);
var circumference = Math.PI * (r + r);
```

Вы можете преобразовать строку в целое число, используя встроенную функцию parseInt(). Её необязательный второй параметр — основание системы счисления, которое следует всегда явно указывать:

```
parseInt("123", 10); // 123
parseInt("010", 10); // 10
```

Если вы не предоставите основание, то можете получить неожиданные результаты:

```
parseInt("010");  // 8
parseInt("0x10"); // 16
```

Это случилось потому, что функция parseInt() расценила строку как восьмеричную из-за начального 0, а шестнадцатеричную - из-за начального "0x".

Если хотите преобразовать двоичное число в десятичное целое, просто смените основание:

```
parseInt("11", 2); // 3
```

Вы можете аналогично парсить дробные числа, используя встроенную функцию parseFloat(), которая использует всегда основание 10 в отличие от родственной parseInt().

Также можно использовать унарный оператор + для преобразования значения в число:

```
+ "42";   // 42
+ "0x10"; // 16
```

Специальное значение NaN (сокращение от "Not a Number") возвращается, если строка не является числом:

```
parseInt("hello", 10); // NaN
```

NaN "заразителен": любая математическая операция над NaN возвращает NaN:

```
NaN + 5; // NaN
```

Проверить значение на NaN можно встроенной функцией isNaN():

```
isNaN(NaN); // true
```

JavaScript также имеет специальные значения Infinity (бесконечность) и -Infinity:

```
1 / 0; // Infinity
-1 / 0; // -Infinity
```
Проверить значение на Infinity, -Infinity и NaN можно с помощью встроенной функции isFinite():

```
isFinite(1/0); // false
isFinite(-Infinity); // false
isFinite(NaN); // false
```
Примечание: функции parseInt() и parseFloat() обрабатывают строку до тех пор, пока не будет встречен символ, не являющийся корректным для заданного числового формата, затем эти функции возвращают число, полученное в результате обработки вплоть до этого символа. А оператор "+" просто возвращает NaN, если в строке есть хоть один некорректный символ. Попробуйте сами в консоли преобразовать строку "10.2abc" каждым из методов, и различие станет ясным.

Строки

Строки в JavaScript - это последовательности символов Unicode (в кодировке UTF-16). Для тех, кто имеет дело с интернационализацией, это должно стать хорошей новостью. Если быть более точным, то строка - это последовательность кодовых единиц, каждая из которых представлена 16-битовым числом, а каждый символ Unicode состоит из 1 или 2 кодовых единиц.

Чтобы представить единственный символ, используйте строку, содержащую только этот символ.

Чтобы выяснить длину строки (в кодовых единицах), используйте свойство length:

```
"hello".length; // 5
```
Это уже первый шаг для работы с объектами! Мы уже говорили, что и строки можно использовать как объекты? У них тоже есть методы:

```
"hello".charAt(0); // h
"hello, world".replace("hello", "goodbye"); // goodbye, world
"hello".toUpperCase(); // HELLO
```
Другие типы

JavaScript дополнительно различает такие типы, как null, который указывает на преднамеренное отсутствующее значение, и undefined, указывающий на неинициализированное значение — то есть, значение, которое даже не было назначено. Мы поговорим о переменных позже, но в JavaScript можно объявить переменную без присвоения ей значения. В этом случае тип переменной будет "undefined".

Ещё в JavaScript есть логический (булевый) тип данных, который может принимать два возможных значения true или false (оба являются ключевыми словами). Любое значение может быть преобразовано в логическое значение в соответствии со следующими правилами:

false, 0, пустая строка (""), NaN, null и undefined преобразуются в false.
Все остальные значения преобразуются в true.
Преобразование значений можно осуществить явно, используя функцию Boolean():

```
Boolean(""); // false
Boolean(234); // true
```

Этот метод используется редко, так как JavaScript может автоматически преобразовывать типы в тех случаях, когда ожидается булевое значение, например в операторе if. Из-за того, что любой тип данных может быть преобразован в булевое значение, иногда говорят, что данные "истинные" или "ложные".

Для операций с логическими данными используются логические операторы: && (логическое И), || (логическое ИЛИ), ! (логическое НЕ).

Переменные
Для объявления новых переменных в JavaScript используются ключевые слова let, const или var.

```
let a;
let name = "Simon";
```

let позволяет объявлять переменные, которые доступны только в блоке, в котором они объявлены:

```
// myLetVariable недоступна здесь

for (let myLetVariable = 0; myLetVariable < 5; myLetVariable++) {
  // myLetVariable доступна только здесь
}

// myLetVariable недоступна здесь
```

const позволяет создавать переменные, чьи значения не предполагают изменений. Переменная доступна из блока, в котором она объявлена.

```
const Pi = 3.14; // в переменную Pi записано значение.
Pi = 1; // вызовет исключение, так как значение константы нельзя изменить.
```

var наиболее общее средство объявления переменной. Оно не имеет ограничений, которые имеют два вышеописанных способа. Это потому, что это был изначально единственный способ объявления переменной в JavaScript. Переменная, объявленная с помощью var, доступна в пределах функции, в которой она объявлена.

```
var a;
var name = 'Simon';
```

Пример кода с переменной, объявленной с помощью var:

```
// myVarVariable доступна здесь
```

```
for (var myVarVariable = 0; myVarVariable < 5; myVarVariable++) {
  // myVarVariable доступна для всей функции
}

// myVarVariable доступна и здесь
```
Если вы объявляете переменную без присвоения ей какого-либо значения, то её тип будет определён как undefined.

Важной особенностью языка JavaScript является то, что блоки данных не имеют своей области видимости, она есть только у функций. Поэтому, если объявить переменную через var в блоке данных (например, внутри контролирующей структуры if), то она будет доступна всей функции. Следует отметить, что в новом стандарте ECMAScript Edition 6 появились инструкции let и const, позволяющие объявлять переменные с областью видимости, ограниченной пределами блока.

Операторы

JavaScript поддерживает такие операторы, как +, -, *, / и %, который возвращает остаток от деления (не путать с модулем). Значения присваиваются с помощью оператора =, или с помощью составных операторов += и -=. Это сокращённая запись выражения x = x оператор y.

```
x += 5
x = x + 5
```
Так же используются операторы инкремента (++) и декремента (--). Которые имеют префиксную и постфиксную форму записи.

Оператор + так же выполняет конкатенацию (объединение) строк:

```
"hello" + " world"; // "hello world"
```
При сложении строкового и числового значений происходит автоматическое преобразование в строку. Поначалу такое может запутать:

```
"3" + 4 + 5; // "345"
3 + 4 + "5"; // "75"
```
Для приведения значения к строке просто прибавьте к нему пустую строку.

Для сравнения в JavaScript используются следующие операторы: <, >, <= и >=. Сравнивать можно не только числа, но и строки. Проверка на равенство немного сложнее. Для проверки используют двойной (==) или тройной (===) оператор присваивания. Двойной оператор == осуществляет автоматическое преобразование типов, что может приводить к интересным результатам:

```
123 == "123"; // true
1 == true;    // true
```
Если преобразование нежелательно, то используют оператор строгого равенства:

```
1 === true; // false
123 === "123"; // false
true === true; // true
```
Для проверки на неравенство используют операторы != и !==.

Отдельного внимания стоят побитовые операторы, с которыми вы можете ознакомиться в соответствующем разделе.

Управляющие структуры
Управляющие структуры в JavaScript очень похожи на таковые в языках семейства C. Условные операторы выражены ключевыми словами if и else, которые можно составлять в цепочки:

```
var name = "kittens";
if (name == "puppies") {
  name += "!";
} else if (name == "kittens") {
  name += "!!";
} else {
  name = "!" + name;
}
name == "kittens!!"
```
В JavaScript есть три типа циклов: while, do-while и for. While используется для задания обычного цикла, а do-while целесообразно применить в том случае, если вы хотите, чтобы цикл был выполнен хотя бы один раз:

```
while (true) {
  // бесконечный цикл!
}

var input;
do {
  input = get_input();
} while (inputIsNotValid(input))
```
Цикл for похож на такой же в языках C и Java: он позволяет задавать данные для контроля за выполнением цикла:

```
for (var i = 0; i < 5; i++) {
  // Выполнится 5 раз
}
```
JavaScript также содержит две других известных конструкции: for...of

```
for (let value of array) {
  // операции с value
}
```
и for...in:

```
for (let property in object) {
  // операции над свойствами объекта
}
```
Логические операторы && и || используют "короткий цикл вычисления", это значит, что вычисление каждого последующего оператора зависит от предыдущего. Например, полезно проверить существует ли объект или нет, прежде чем пытаться получить доступ к его свойствам:

```
var name = o && o.getName();
```
Таким способом удобно задавать значения по умолчанию:

```
var name = otherName || "default";
```
К условным операторам в JavaScript принадлежит также тернарный оператор "?" :

```
var allowed = (age > 18) ? "yes" : "no";
```
Оператор switch используется при необходимости множественного сравнения:

```
switch(action) {
    case 'draw':
        drawit();
        break;
    case 'eat':
        eatit();
        break;
    default:
        donothing();
}
```
Если в конце инструкции case не добавить останавливающую инструкцию break, то выполнение перейдёт к следующей инструкции case. Как правило, такое поведение нежелательно, но если вдруг вы решили его использовать, настоятельно рекомендуем писать соответствующий комментарий для облегчения поиска ошибок:

```
switch(a) {
    case 1: // fallthrough
    case 2:
        eatit();
        break;
    default:
        donothing();
}
```
Вариант default опциональный. Допускается использование выражений как в условии switch, так и в cases. При проверке на равенство используется оператор строгого равенства ===:

```
switch(1 + 3) {
```

```
    case 2 + 2:
        yay();
        break;
    default:
        neverhappens();
}
```

Объекты

Объекты в JavaScript представляют собой коллекции пар имя-значение (ключ-значение). Они похожи на:

Словари в Python.
Хеши в Perl и Ruby.
Таблицы хешей в C и C++.
HashMaps в Java.
Ассоциативные массивы в PHP.

Именем свойства объекта в JavaScript выступает строка, а значением может быть любой тип данных JavaScript, даже другие объекты. Это позволяет создавать структуры данных любой сложности.

Существует два основных способа создать объект:

```
var obj = new Object();
```

А также:

```
var obj = {};
```

Обе эти записи делают одно и то же. Вторая запись называется литералом объекта и более удобная. Такой способ является основой формата JSON, и при написании кода лучше использовать именно его.

С помощью литерала объекта можно создавать не только пустые объекты, но и объекты с данными:

```
var obj = {
  name: "Carrot",
  "for": "Max",
  details: {
    color: "orange",
    size: 12
  }
}
```

Доступ к свойствам объекта можно получить следующими способами:

```
obj.details.color; // orange
obj['details']['size']; // 12
```

Эти два метода равнозначны. Первый метод используется, если мы точно знаем к какому методу нам нужно обратиться. Второй метод принимает в качестве имени

свойства строку, и позволяет вычислять имя в процессе вычислений. Следует отметить, что последний метод мешает некоторым движкам и минимизаторам оптимизировать код. Если появится необходимость назначить в качестве имён свойств объекта зарезервированные слова, то данный метод тоже может пригодиться:

```
// Вызовет Syntax error, ведь 'for' это зарезервированное слово
obj.for = "Simon";

// А тут всё нормально
obj["for"] = "Simon";
```

На заметку: Стандарт EcmaScript 5 позволяет использовать зарезервированные слова в качестве имён свойств объекта без "оборачивания" их в кавычки. Подробнее в спецификации ES5.

Больше информации об объектах и прототипах: Object.prototype.

Для получения информации по прототипам объектов и цепям прототипов объектов смотрите Inheritance and the prototype chain.

Начиная с ECMAScript 2015, ключи объектов могут быть определены переменными с использованием квадратных скобок при создании: {[phoneType]: 12345} допустимо вместо var userPhone = {}; userPhone[phoneType] = 12345.

Массивы
Массивы в JavaScript всего лишь частный случай объектов. Работают они практически одинаково (если именем свойства является число, то доступ к нему можно получить только через вызов в скобках []), только у массивов есть одно удивительное свойство 'length' (длина). Оно возвращает число, равное самому большому индексу массива + 1.

Создать массив можно по старинке:

```
var a = new Array();
a[0] = "dog";
a[1] = "cat";
a[2] = "hen";
a.length; // 3
```
Но гораздо удобнее использовать литерал массива:

```
var a = ["dog", "cat", "hen"];
a.length; // 3
```
Запомните, свойство array.length не обязательно будет показывать количество элементов в массиве. Посмотрите пример:

```
var a = ["dog", "cat", "hen"];
a[100] = "fox";
```

a.length; // 101
Запомните — длина массива это его самый большой индекс плюс один.

Если попытаться получить доступ к несуществующему элементу массива, то получите undefined:

typeof a[90]; // undefined
Для перебора элементов массива используйте такой способ:

```
for (var i = 0; i < a.length; i++) {
    // Сделать что-нибудь с элементом a[i]
}
```
ES2015 представляет более краткий for...of способ обхода по итерируемым объектам, в т.ч. массивам:

```
for (const currentValue of a) {
  // Сделать что-нибудь с currentValue
}
```
Перебрать элементы массива также можно с помощью цикла for...in. Но, если вдруг будет изменено какое-либо свойство Array.prototype, то оно тоже будет участвовать в выборке. Не используйте данный метод.

И самый новый способ перебора свойств массива был добавлен в ECMAScript 5 — это метод forEach():

```
["dog", "cat", "hen"].forEach(function(currentValue, index, array) {
// Сделать что-нибудь с currentValue или array[index]
});
```
Для добавления данных в массив используйте метод push():

```
a.push(item);
```
У массивов есть ещё множество полезных методов. С их полным списком вы можете ознакомиться по ссылке.

Метод Описание
a.toString() Возвращает строковое представление массива, где все элементы разделены запятыми.
a.toLocaleString() Возвращает строковое представление массива в соответствии с выбранной локалью.
a.concat(item1[, item2[, ...[, itemN]]]) Возвращает новый массив с добавлением указанных элементов.
a.join(sep) Преобразует массив в строку, где в качестве разделителя используется параметр sep
a.pop() Удаляет последний элемент массива и возвращает его.
a.push(item1, ..., itemN) Добавляет один или более элементов в конец массива.
a.reverse() Меняет порядок элементов массива на обратный.

a.shift() Удаляет первый элемент массива и возвращает его.
a.slice(start[, end]) Возвращает новый массив.
a.sort([cmpfn]) Сортирует данные в массиве.
a.splice(start, delcount[, item1[, ...[, itemN]]]) Позволяет вырезать из массива его часть и добавлять на это место новые элементы.
a.unshift(item1[, item2[, ...[, itemN]]]) Добавляет элементы в начало массива.
Функции
Наряду с объектами функции также являются ключевыми компонентами языка JavaScript. Базовые функции очень просты:

```
function add(x, y) {
   var total = x + y;
   return total;
}
```

В этом примере показано практически всё, что нужно знать о функциях. Функции в JavaScript могут принимать ноль или более параметров. Тело функции может содержать любые выражения и определять свои собственные переменные, которые будут для этой функции локальными. Инструкция return используется для возврата значения и остановки выполнения функции. Если инструкции return в функции нет (или есть, но не указано возвращаемое значение), то JavaScript возвратит undefined.

Можно вызвать функцию, вообще не передавая ей параметры. В таком случае будет считаться, что их значения равны undefined:

```
add(); // NaN
// Нельзя проводить сложение undefined и undefined
```

Можно передать больше аргументов, чем ожидает функция:

```
add(2, 3, 4); // 5
// используются только первые два аргумента, "4" игнорируется
```

Это может показаться бессмысленным, но на самом деле функции могут получить доступ к "лишним" аргументам с помощью псевдомассива arguments, в нём содержатся значения всех аргументов, переданных функции. Давайте напишем функцию, которая принимает неограниченное количество аргументов:

```
function add() {
   var sum = 0;
   for (var i = 0, j = arguments.length; i < j; i++) {
      sum += arguments[i];
   }
   return sum;
}

add(2, 3, 4, 5); // 14
```

Или создадим функцию для вычисления среднего значения:

```
function avg() {
    var sum = 0;
    for (var i = 0, j = arguments.length; i < j; i++) {
        sum += arguments[i];
    }
    return sum / arguments.length;
}
avg(2, 3, 4, 5); // 3.5
```

Это довольно полезно, но при этом кажется излишне подробным. Для уменьшения количества кода взглянем на замену использования массива аргументов способом Rest parameter syntax. В этом случае мы можем передавать в функцию любое количество аргументов, сохраняя код минималистичным. Rest parameter operator используется в списке параметров функции в формате: ...variable и включает в себя целый список аргументов, с которыми функция будет вызвана. Мы будем также использовать замену цикла for циклом for...of для получения значений, которые будет содержать наша переменная.

```
function avg(...args) {
  var sum = 0;
  for (let value of args) {
    sum += value;
  }
  return sum / args.length;
}

avg(2, 3, 4, 5); // 3.5
```

В вышенаписанном коде переменная args содержит все значения, которые были переданы в функцию.

Важно отметить, что где бы ни был размещен rest parameter operator в объявлении функции, он будет содержать все аргументы после его объявления, не раньше. например: function avg(firstValue, ...args) будет хранить первое переданное значение в переменной firstValue и оставшиеся в args. Это еще одно полезное свойство языка, однако оно ведет нас к новой проблеме. avg() функция принимает список аргументов, разделенный запятыми. Но что если вы хотите найти среднее значение в массиве? Вы можете переписать функцию следующим образом:

```
function avgArray(arr) {
  var sum = 0;
  for (var i = 0, j = arr.length; i < j; i++) {
    sum += arr[i];
  }
  return sum / arr.length;
}

avgArray([2, 3, 4, 5]); // 3.5
```

На тот случай, если вы хотите использовать первый вариант функции, а не переписывать её заново, то в JavaScript есть возможность вызывать функцию с произвольным массивом аргументов. Для этого используется метод apply():

```
avg.apply(null, [2, 3, 4, 5]); // 3.5
```

Вторым аргументом метода apply() передаётся массив, который будет передан функции в качестве аргументов. О первом аргументе мы поговорим позже. Наличие у функций методов также говорит о том, что на самом деле они являются объектами.

Этот же результат можно получить, используя spread operator в вызове функции.

For instance: avg(...numbers)

В JavaScript можно создавать анонимные функции:

```
var avg = function() {
   var sum = 0;
   for (var i = 0, j = arguments.length; i < j; i++) {
      sum += arguments[i];
   }
   return sum / arguments.length;
}
```

Данная запись семантически равнозначна записи function avg(). Это даёт возможность использовать разные интересные трюки. Вот посмотрите, как можно "спрятать" локальные переменные в функции:

```
var a = 1;
var b = 2;
(function() {
   var b = 3;
   a += b;
})();
a; // 4
b; // 2
```

В JavaScript есть возможность рекурсивного вызова функции. Это может оказаться полезным при работе с иерархическими (древовидными) структурами данных (например такие, которые встречаются при работе с DOM).

```
function countChars(elm) {
   if (elm.nodeType == 3) { // TEXT_NODE
      return elm.nodeValue.length;
   }
   var count = 0;
   for (var i = 0, child; child = elm.childNodes[i]; i++) {
      count += countChars(child);
   }
```

```
    return count;
}
```
Тут мы сталкиваемся с проблемой: как вызвать функцию рекурсивно, если у неё нет имени? Для этого в JavaScript есть именованные функциональные выражения IIFEs (Immediately Invoked Function Expressions). Вот пример использования именованной самовызывающейся функции:

```
var charsInBody = (function counter(elm) {
    if (elm.nodeType == 3) { // TEXT_NODE
        return elm.nodeValue.length;
    }
    var count = 0;
    for (var i = 0, child; child = elm.childNodes[i]; i++) {
        count += counter(child);
    }
    return count;
})(document.body);
```
Имя функции в примере доступно только внутри самой функции. Это улучшает оптимизацию и читаемость кода.

Собственные объекты

На заметку: Для более подробной информации по объектно-ориентированному программированию в JavaScript смотрит Введение в объектно-ориентированный JavaScript.

В классическом Объектно-Ориентированном Программировании (ООП) объекты — это коллекции данных и методов, которые этими данными оперируют. JavaScript - это язык, основанный на прототипах, и в его определении нет понятия классов, таких, как в языках C++ или Java. (Иногда это может запутать программистов, знакомых с языками, в которых есть классы.) Вместо классов JavaScript использует функции. Давайте представим объект с личными данными, содержащий поля с именем и фамилией. Есть два типа отображения имён: "Имя Фамилия" или "Фамилия, Имя". С помощью объектов и функций можно сделать следующее:

```
function makePerson(first, last) {
    return {
        first: first,
        last: last
    }
}

function personFullName(person) {
    return person.first + ' ' + person.last;
}

function personFullNameReversed(person) {
    return person.last + ', ' + person.first
```

```
}

s = makePerson("Simon", "Willison");
personFullName(s); // Simon Willison
personFullNameReversed(s); // Willison, Simon
```

Работает, но сам код никуда не годится. С таким подходом у вас будут десятки функций, засоряющих глобальный объект. Это можно исправить, прикрепив функцию к объекту. Это просто, ведь все функции и есть объекты:

```
function makePerson(first, last) {
    return {
        first: first,
        last: last,
        fullName: function() {
            return this.first + ' ' + this.last;
        },
        fullNameReversed: function() {
            return this.last + ', ' + this.first;
        }
    }
}
s = makePerson("Simon", "Willison")
s.fullName(); // Simon Willison
s.fullNameReversed(); // Willison, Simon
```

А вот кое-что новенькое: ключевое слово 'this'. Когда 'this' используется внутри функции, оно ссылается на текущий объект. Значение ключевого слова зависит от способа вызова функции. Если вызвать функцию с обращением к объекту через точку или квадратные скобки, то 'this' получится равным данному объекту. В ином случае 'this' будет ссылаться на глобальный объект. Это часто приводит к ошибкам. Например:

```
s = makePerson("Simon", "Willison")
var fullName = s.fullName;
fullName(); // undefined undefined
```

При вызове fullName(), 'this' получает ссылку на глобальный объект. А так как в глобальном объекте не определены переменные first и last, то имеем два undefined.

Используя особенность ключевого слова 'this', можно улучшить код функции makePerson:

```
function Person(first, last) {
    this.first = first;
    this.last = last;
    this.fullName = function() {
        return this.first + ' ' + this.last;
    };
```

```
    this.fullNameReversed = function() {
        return this.last + ', ' + this.first;
    };
}
var s = new Person("Simon", "Willison");
```

В примере мы использовали новое ключевое слово: 'new'. Оно тесно связано с 'this'. Данное ключевое слово создаёт новый пустой объект, а потом вызывает указанную функцию, а this получает ссылку на этот новый объект. Функции, которые предназначены для вызова с 'new' называются конструкторами. Существует соглашение, согласно которому все функции-конструкторы записываются с заглавной буквы.

Мы доработали наш код в предыдущем примере, но всё равно остался один неприятный момент с самостоятельным вызовом fullName().

Каждый раз, когда с помощью конструктора создаётся новый объект, мы заново создаём и две новые функции. Гораздо удобнее создать эти функции отдельно и дать доступ к ним конструктору:

```
function personFullName() {
    return this.first + ' ' + this.last;
}
function personFullNameReversed() {
    return this.last + ', ' + this.first;
}
function Person(first, last) {
    this.first = first;
    this.last = last;
    this.fullName = personFullName;
    this.fullNameReversed = personFullNameReversed;
}
```

Уже лучше: мы создали функции-методы только один раз, а при новом вызове функции-конструктора просто ссылаемся на них. Можно сделать ещё лучше? Конечно:

```
function Person(first, last) {
    this.first = first;
    this.last = last;
}
Person.prototype.fullName = function fullName() {
    return this.first + ' ' + this.last;
}
Person.prototype.fullNameReversed = function fullNameReversed() {
    return this.last + ', ' + this.first;
}
```

Person.prototype это объект, доступ к которому есть у всех экземпляров класса Person. Он создаёт особую цепочку прототипов. Каждый раз, когда вы пытаетесь получить

доступ к несуществующему свойству объекта Person, JavaScript проверяет, существует ли свойство в Person.prototype. В результате все, что передано в Person.prototype, становится доступным и всем экземплярам этого конструктора через this объект.

Это очень мощный инструмент. JavaScript позволяет изменять прототипы в любое время, это значит, что можно добавлять новые методы к существующим объектам во время выполнения программы:

```
s = new Person("Simon", "Willison");
s.firstNameCaps();
// TypeError on line 1: s.firstNameCaps is not a function

Person.prototype.firstNameCaps = function() {
    return this.first.toUpperCase()
}
s.firstNameCaps(); // "SIMON"
```

Занимательно то, что добавлять свойства в прототип можно и для встроенных объектов JavaScript. Давайте добавим новый метод reversed классу String, этот метод будет возвращать строку задом наперёд:

```
var s = "Simon";
s.reversed(); // TypeError on line 1: s.reversed is not a function

String.prototype.reversed = function reversed() {
    var r = "";
    for (var i = this.length - 1; i >= 0; i--) {
        r += this[i];
    }
    return r;
}
s.reversed(); // "nomiS"
```

Данный метод будет работать даже на литералах строки!

```
"This can now be reversed".reversed();
// desrever eb won nac sihT
```

Как уже упоминалось, prototype формирует часть цепочки. Конечным объектом этой цепочки прототипов является Object.prototype, методы которого включают и toString() — тот метод, который вызывается тогда, когда надо получить строковое отображение объекта. Вот что можно сделать с нашими объектами Person:

```
var s = new Person("Simon", "Willison");
s.toString(); // [object Object]

Person.prototype.toString = function() {
    return '<Person: ' + this.fullName() + '>';
```

```
}

s.toString(); // "<Person: Simon Willison>"
```
Помните, мы вызывали avg.apply() с первым аргументом равным null? Теперь мы можем сделать так: первым аргументом, переданным методу apply() будет объект, который примет значение 'this'. Вот к примеру упрощённая реализация 'new':

```
function trivialNew(constructor, ...args) {
   var o = {}; // Создаём новый объект
   constructor.apply(o, args);
   return o;
}
```
Это не точная копия new, так как она не устанавливает цепочку прототипов (это сложно). Метод apply() применяется не очень часто, но знать его важно. В примере выше, запись ...args (включая многоточие) называется "rest arguments"— она включает в себя все оставшиеся аргументы.

Вызов

```
var bill = trivialNew(Person, 'William', 'Orange');
```
практически полностью эквивалентен этому:

```
var bill = new Person('William', 'Orange');
```
В JavaScript метод apply() имеет похожий метод call(), который тоже позволяет устанавливать 'this', но принимает список, а не массив аргументов.

```
function lastNameCaps() {
   return this.last.toUpperCase();
}
var s = new Person("Simon", "Willison");
lastNameCaps.call(s);
// Аналогично записи:
s.lastNameCaps = lastNameCaps;
s.lastNameCaps(); // WILLISON
```
Вложенные функции

Объявлять новые функции можно и внутри других функций. Мы использовали этот приём чуть выше, создавая функцию makePerson(). Главная особенность вложенных функций в том, что они получают доступ к переменным, объявленным в их функции-родителе:

```
function parentFunc() {
  var a = 1;

  function nestedFunc() {
    var b = 4; // parentFunc can't use this
    return a + b;
```

```
  }
  return nestedFunc(); // 5
}
```
Это очень полезное свойство, которое делает сопровождение кода более удобным. Если ваша функция в своей работе использует другие функции, которые больше нигде не используются, то можно просто вложить вспомогательные функции в основную. Это сократит количество функций в глобальном объекте, что довольно неплохо.

Ещё это отличный способ сократить количество глобальных переменных. Так при написании кода у нас часто будет возникать искушение понасоздавать глобальных переменных, которые будут доступны разным функциям. Всё это усложняет код, делает его менее читаемым. Вложенные функции имеют доступ к переменным своей функции-родителя, и мы может использовать это для группировки множества функций вместе (естественно в разумных пределах), что позволит держать наш глобальный объект в чистоте и порядке.

Замыкания (Closures)
Мы подошли к одному из самых мощных и непонятных инструментов JavaScript. Давайте разберёмся.

```
function makeAdder(a) {
    return function(b) {
        return a + b;
    };
}

var x = makeAdder(5);
var y = makeAdder(20);
x(6); // ?
y(7); // ?
```
Функция makeAdder создаёт новую функцию, которая прибавляет полученное значение к значению, которые было получено при создании функции.

Такой же фокус мы наблюдали в предыдущем примере, когда внутренние функции получали доступ к переменным той функции, в которой были объявлены. Только в нашем примере основная функция возвращает вложенную. Поначалу может показаться, что локальные переменные при этом перестанут существовать. Но они продолжают существовать — иначе код попросту не сработал бы. Вдобавок ко всему у нас есть две разные "копии" функции makeAdder, присвоенные разным переменным (одна копия, в которой a - это 5, а во второй a - это 20). Вот что имеем в результате вызова:

```
x(6); // возвратит 11
y(7); // возвратит 27
```
И вот что произошло: когда JavaScript выполняет функцию, создаётся объект 'scope', который содержит в себе все локальные переменные, объявленные внутри этой

функции. Он инициализируется любым значением, переданным функции в качестве параметра. 'Scope' подобен глобальному объекту, который содержит все глобальные переменные и функции, кроме нескольких важных отличий: при каждом вызове функции создаётся новый объект 'scope' и, в отличие от глобального, к объекту 'scope' нельзя получить прямой доступ из вашего кода. И нет способа пройтись по свойствам данного объекта.

Так что при вызове функции makeAdder создаётся новый объект 'scope' с единственным свойством: a, которому присваивается значение, переданное функции в качестве аргумента. Потом makeAdder возвращает новую анонимную функцию. В любом другом случае 'сборщик мусора' удалил бы объект scope, но возвращаемая функция ссылается на этот объект. В итоге объект scope не удаляется до тех пор, пока существует хотя бы одна ссылка на него.

Все объекты scope соединяются в цепочку областей видимости, которая похожа на цепочку прототипов в объектной системе JavaScript'a.

Замыкание это связка из функции и объекта scope, созданного при её вызове.
Подробнее о замыканиях здесь.

Metadata
Последнее изменение: 7 окт. 2019 г., помощниками MDN
Связанные темы
JavaScript
Уроки:
Базовые
Руководство по JavaScript
Средние
Продвинутые
Справочная информация:
Встроенные объекты
Выражения и операторы
Инструкции и объявления
Функции
Классы
Ошибки
Ещё

Скрыть подписку на новостную рассылку
Веб-документация MDN
Веб-технологии
Изучение Веб-разработки
О MDN
Оставить отзыв
О нас
MDN Web Docs Store
Обратная связь
Firefox
MDN
Mozilla

Условия использования
Конфиденциальнсть
Куки