**Основы JavaScript ES5. Вступление**

**Основной синтаксис**

Переменные — это контейнеры, внутри которых вы можете хранить значения. Вы начинаете с того, что объявляете переменную с помощью ключевого слова var, за которым следует любое имя, которым вы захотите её назвать. С помощью знака равно (“=”) присваиваете ей значение, которое хотите в ней хранить.

Например:  
 *var* num = 6;

Операторы — это математический символ или их группа, которая или который производит результат, основанный на двух значениях (или переменных).

* сложение “+”
* вычитание “-”
* умножение “\*”
* деление “/”
* целочисленное деление “\”
* остаток от деления “%”
* равенство “=”
* тождество с приведением типов “==” или отрицание “!=”
* тождество без приведения типов “===” или отрицание “!==”
* отрицание логического значения “!”
* битовое отрицание “~” ~x = - ( x + 1 )

Условные выражения — это кодовые структуры, которые позволяют вам проверять, истинно или ложно выражение, а затем выполнить другой код в зависимости от результата. Самая распространённая форма условия называется, if ... else.

Например:

*var* num = 6;  
 *if* (num ===6) {  
 console.log("Это 6");  
 } *else* {  
 console.log("Это не 6");  
 }

Короткие вариации условного выражения*:*

Если значение присваиваемое переменной зависит от какого-либо условия, вы можете воспользоваться условным выражением “условие ? если true : если false”.

Например:

var text = (num === 6) ? "Это 6" : "Это не 6";

В данном примере переменной text будет присвоено такое значение, что если num равна 6 то “Это 6”, в противном случае “Это не 6”. Скобки добавлены для наглядности, писать их не обязательно.

Если вам нужно запустить функцию при выполнении определенного условия, вы можете воспользоваться выражением “условие && функция()”.

Например:

num === 6 && console.log ("Это 6!");

Циклы — это кодовые структуры для многократного повторения одного участка кода. При написании скриптов зачастую встаёт задача сделать однотипное действие много раз. Например, вывести товары из списка один за другим. Или просто перебрать все числа от 1 до 10 и для каждого выполнить одинаковый код.

Цикл while выполняет действие, пока истинно условие:

var i = 10;  
while (i > 0) {  
 console.log(i--);  
}  
/\*  
10 9 8 7 6 5 4 3 2 1  
\*/

Цикл do while выполнит действие и будет повторять его, если условие истинно:

var i = 10;  
do{  
 console.log(i--);  
} while (i > 0)  
/\*  
10 9 8 7 6 5 4 3 2 1  
\*/

Цикл for выполняется, пока переменная, инициализированная для цикла, удовлетворяет условию:

//инициализируем переменную, ставим условие, при котором цикл будет выполняться, добавляем правило изменения переменной  
for (var i = 0; i <= 10; i += 2) {   
 console.log(i);  
}  
/\*  
0 2 4 6 8 10  
\*/

Функции и Процедуры — способ упаковки функциональности, которую вы хотите использовать повторно. Всякий раз, когда вам нужна определённая процедура, вы можете просто вызвать функцию по её имени, а не переписывать весь код каждый раз.

Вы начинаете с того, что объявляете функцию с помощью ключевого слова function за которым следует любое имя, которым вы захотите её назвать, в скобках указываете аргументы функции (переменные, которые вы будете передавать в свою функцию и пользоваться ими при ее выполнении). Так как переменные в JavaScript явно не типизированы, нужны только названия. Возвращаемое значение указывается, используя ключевое слово return.

Главное отличие процедуры от функции в том, что функция возвращает значение, а процедура только выполняет последовательность действий.

*//Функция  
 function multiplyFun* (num1, num2) {  
 *var* result = num1 \* num2;  
 *return* result;  
 }  
 *//Процедура  
 function multiplyProc* (num1, num2) {  
 *var* result = num1 \* num2;  
 console.log(result);  
 }

**Типы данных**

JavaScript использует два класса данных:

* Примитивы
* Number или число
* String или строка
* Boolean или логический тип
* Null или отсутствие значения
* undefined или отсутствие примитива (ошибочное обращение)
* Объекты (ссылки)
* Array или массив
* function или функция
* Regular Exp или регулярные выражения
* И многое другое

Для того чтобы оперировать информацией о типах данных или просто узнавать тип переменной, вы можете воспользоваться командой typeof().

Например:

var num = 6;  
var str = "JavaScript";  
function onlyNum (num) {  
 if (typeof(num) === "number") {  
 console.log("this is number!");  
 } else {  
 console.log("this is not number!");  
 }  
}  
onlyNum(num);  
onlyNum(str);

**Примитивы**

У примитивов так же, как и у объектов есть свои методы. Они вызываются с помощью “.”(точки) после вашей переменной и указания на метод.

Например:

var text ="это будут большие буквы";  
 var newText = text.toUpperCase(); //метод меняет всем буквам регистр на заглавный

Чтобы пользоваться методами конкретного типа данных, при необходимости можно использовать функции приведения к типу:

* Number()
* String()
* Boolean()
* Primitive
* JSON.stringify()

Например:

var text = "это будут большие буквы";  
 var newText = String(text).toUpperCase();

**Объекты (ссылочные типы)**

Создавать объекты можно с помощью оператора new или соответствующего ему литерала:

К ним относится, например, Object (объект), Array (Массив), function (функция), regular expression (регулярное выражение).

var object = new Object();  
 var object\_with\_literal = {};  
 var array = new Array(1, 2, 3, 4, 5);  
 var array\_with\_literal = [1, 2, 3, 4, 5];  
 var func = new Function('a,b', 'return a + b');  
 function func\_with\_literal(a, b) {  
 return a + b;  
 }  
 var reg = new RegExp('\\w', 'g');  
 var reg\_with\_literal = /\w/g;

Большинство встроенных объектов можно создать, не дописывая оператор new, он пригодится только при считывании даты, т.к. без оператора вы получите просто string Data, а с оператором object Data.

typeof new Date() //Объект  
 typeof Date() //Строка

У ссылочных типов также есть встроенные методы, доступ к ним производится также, как и у примитивов. Например, тип данных Object имеет возможность не только хранить в себе статичную информацию в формате “ключ => значение”, но и добавлять методы (функции).

Например:

var forest = {  
 count\_tree: 284,  
 trees\_name: ["Birch", "Spruce", "Maple"],  
 show\_about: function (count) {  
 console.log("There are " + count + " trees in this forest. Birches, spruces, maples")  
 }  
 };  
 var count = forest.count\_tree; // 284  
 forest.show\_about(count); // There are 284 trees in this forest. Birches, spruces, maples

**Подробнее о массивах**

Массив, как объект, представляет собой набор других объектов. Они не обязательно должны быть одного типа данных.

Например:

//Массив чисел  
var numberArray = [1, 2, 3, 4, 5];  
//Массив строк  
var stringArray = ["поешь", "еще", "этих", "мягких", "французских", "булок"];  
//Массив из разных объектов  
var allArray = [273, "деревьев",{ numberArray, stringArray }, true];

**Типы преобразований**

1. Boolean()

Данное преобразование вызывает встроенный или переопределенный метод toBoolean(), работает по законам:

* 1. Object всегда превращается в true.
  2. “” (пустая строка) – false.
  3. Null – false.
  4. undefined – false.
  5. false (очевидно) тоже даст false.
  6. 0 – false.
  7. -0 – false.
  8. NaN – false.
  9. Все остальное ВСЕГДА true.

1. String()

Данное преобразование вызывает встроенный или переопределенный метод toString(). Добавляет кавычки к числу или логическому значению. Если попробовать Объект, например массив, перевести в строку, вызовется оператор преобразования Primitive (о нем дальше).

1. Number()

Данное преобразование вызывает встроенный или переопределенный метод toNumber() и работает по законам.

* 1. true / false превращает в 1 / 0.
  2. Cтроку попробует конвертировать в число.
     1. Если это пустая строка “” - это всегда 0.
     2. Если в строке есть что-то похожее на число, попробует отбросить пробелы и конвертировать в него. Например: “ 43 ” => 43
     3. Если в строке будет хоть один лишний символ, это будет NaN. Например: “43px” => NaN

В таком случае можно воспользоваться функциями parseInt() или parseFloat(). Например: parseInt(“42 человека и 1 ребенок”) => 42 (выхватит первое целое число), parseFloat(“0.34 кг и 3 литра”) => 0.34 (выхватит первое число с плавающей запятой).

* 1. Для объекта вызовет Primitive.

1. Алгоритм Primitive

Алгоритм уже вшитый в JavaScript, который вызывает встроенные или переопределенные методы valueOf() и String().

При попытке изменить тип объекта с помощью классического преобразователя, JavaScript запускает алгоритм:

* 1. Если первоначальный метод вернул примитив, то подставляет его в выражение.
  2. Если надо, то преобразовать примитив к другому типу.
  3. Если метод вернул объект (один из двух вышесказанных), вызывай другой метод.
  4. Второй метод тоже вернул объект? Ошибка!

Пример работы:

var ObjectAnswer = {  
 valueOf() {  
 return 43;  
 },  
 toString() {  
 return {};  
 }  
 }  
 console.log("Answer - " + ObjectAnswer); // answer - 43  
 console.log(String(ObjectAnswer));// "43"

5. JSON.stringify()

Сигнатура:

JSON.stringify(value,raplacer,space);

где:

* 1. value – значение
  2. replacer – функция, принимающая 2 параметра key и value, и производящая какие-либо действия с входящими переменными. Должна возвращать value.
  3. space – красиво форматирование, можно использовать символ ‘ ‘(пробела) или табуляции “\t”

Пример использования:

var object = {  
 "name": "Masha",  
 "years": 6,  
 "eyes\_color": "green"  
}  
function replacer(key, value) {  
 value = value === 6 ? 12 : value;  
 return value  
}  
console.log(JSON.stringify(object, replacer, '\t'))  
// {  
// "name": "Masha",  
// "years": 12,  
// "eyes\_color": "green"  
// }

Для обратного перевода из JSON формата в объект можно воспользоваться функцией JSON.parse(value).

**Явное и неявное преобразование типов**

Будем называть явным любое преобразование, понятное большинству программистов в команде (группе в университете), а неявным все остальные (не очевидные и не понятные из контекста).

Например, оператор “+” в JavaScript отвечает не только за сложение и присоединение строк друг к другу, но и может являться оператором для преобразования типов в Number.

Например:

var num = "3434";  
 console.log(typeof +num); // number

Чтобы преобразовать переменную к типу Boolean, можно вызвать двойное отрицание “!!” .

Операнды && и || также не явно преобразуют левую часть в Boolean.

Если прибавить к переменной пустые кавычки с помощью оператора “+”, переменная неявно преобразуется в String.

**Область видимости и объявление**

В JavaScript ES5 достаточно неочевидное поведение переменных. Они, объявленные как var, игнорируют блочную систему видимости.

Разберем на примере:

for (var i = 0; i <= 6; i++) {  
 //тут что-то происходит  
 }  
 console.log(i); //Выведет 7

Но их область видимости не выходит за пределы функции, в которой они были объявлены.

Прежде чем запустить код, интерпретатор JS читает все команды, функции и переменные, и если переменная объявлена, хоть и в самом конце, то с ней можно работать, просто переменная будет равна undefined, но, если вы вызовете необъявленную нигде переменную, очевидно, что это будет ошибка, и прекратится выполнение программы.

console.log(test); //undefined

var test = "test";

console.log(test); //test

var test; //повторное объявление просто проигнорируется без ошибки

console.log(foo); //error foo is not defined

Функцию можно вызывать в любой части программы, даже раньше ее объявления, но можно это сделать только с теми функциями, которые были объявлены не как переменные (хранящие функцию), а как функции или процедуры.

Если вам нужно изменить действие функции, вы можете создать функцию с таким же названием, и функция автоматически будет переопределена. “Победит та функция, что находится ниже всех в коде”.

**Подробнее об Object**

**Контекст и this**

Контекст и this это способ связать функцию или свойство с объектом.

Способы связывания this:

* Связывание контекста по ходу выполнения программы
* Связывание контекста при создании переменной
* Явное связывание. Операторы call /bind

Добавление контекста по ходу выполнения программы:  
 var object = {  
 show: function () {  
 return this.value  
 }  
 }  
 object.value = 12;  
 console.log(object) //{ show: [Function: show], value: 12 }

Добавление контекста при создании переменной:

var object = {  
 value: 12,  
 show: function () {  
 return this.value  
 }  
 }  
 console.log(object.show()) //12

Явное связывание call/bind:

var forest = {  
 count\_tree: 284,  
 trees\_name: ["Birch", "Spruce", "Maple"],  
 show\_about: function (count) {  
 console.log("There are " + count + " trees in this forest. Birches, spruces, maples")  
 },  
 show\_about\_two: function () {  
 console.log("There are " + this.count\_tree + " trees in this forest. Birches, spruces, maples")  
 }  
 };  
 var firstForestMethod = forest.show\_about;  
 //Теперь переменная firstForestMethod будет хранить в себе метод  
 var secondForestMethod = forest.show\_about\_two;  
 //Теперь переменная secondForestMethod будет хранить в себе метод, взаимодействующий с контекстом  
 //Для вызова каждому из вновь созданных методов ОБЯЗАТЕЛЬНО иметь 2 аргумента  
 firstForestMethod.call(forest, '284'); //Вызовет метод относительно forest, используя аргумент 284  
 firstForestMethod.bind(forest)('284'); //Привяжет по очереди 2 аргумента  
 secondForestMethod.call(forest); //Вызовет метод, получит доступ к контексту  
 secondForestMethod.bind(forest)( this); //Сработает только если мы дополнительно привяжем this

Не совсем очевидно, зачем же нужны эти методы. Примеров множество, и разобрать их всех практически невозможно, но самые частые из них это “проброс this в функцию более низкого уровня” и “использование одного метода на нескольких переменных”.

Например:

var forest = {  
 count\_tree: 284,  
 show\_about: function () {  
 var callback = function () {  
 console.log("There are " + this.count\_tree + " trees in this forest. Birches, spruces, maples")  
 };  
 callback.bind(this)();//Пробросили this и добавили пустой аргумент для переполнения и срабатывания bind  
 //напомню, что bind срабатывает тогда, когда получает полный  
 // набор нужных для работы аргументов (каждый в новой скобке)  
 }  
 };  
 forest.show\_about()

Еще один пример:  
var forest = {  
 show\_about: function () {  
 console.log("There are " + this.count\_tree + " trees in this forest. Birches, spruces, maples")  
 }  
}  
var forestOne = {  
 count\_tree: 129  
};  
var forestTwo = {  
 count\_tree: 376  
}  
var method = forest.show\_about //Привязали метод к переменной  
method.call(forestOne) //There are 129 trees in this forest. Birches, spruces, maples  
method.call(forestTwo) //There are 376 trees in this forest. Birches, spruces, maples

**Свойства-аксессоры (Properties)**

Свойства-аксессоры – это уже встроенные методы get property и set property, которые можно определять. В разговоре их часто называют геттеры и сеттеры. Они используются обычно в тех случаях, когда нужно усложнить логику объекта для упрощения кода. Например, есть объект со статичными свойствами и какими-либо методами. При этом по мере выполнения кода будет изменяться только одно поле, мы можем определить метод set property. Благодаря этому, при использовании метода property и знака “=” не нужно будет полностью прописывать объект, а только значение, которое мы хотим присвоить данному полю.

Например:

var forest = {  
 count\_tree: 129,  
 set property(count) {  
 this.count\_tree = count;  
 },   
 get property() {  
 return this.count\_tree;  
 }  
 };  
 forest.property = 547;  
 console.log(forest.property); //547 вместо { count\_tree: 547, property: [Getter/Setter] }

**Свойства-данные (Extensible)**

Свойства-данные — это полностью настроенное свойство (поле) объекта. То есть если вы хотите внедрить какое-либо свойство в ваш объект и не хотите, чтобы кто-нибудь из взломщиков воспользовался вашим приложением для получения данных, вы можете добавить правила работы с этим полем, тем самым сделать взаимодействие с ним более строгим.

Для добавления свойства используется метод Object.defineProperty. Метод принимает в качестве аргументов переменную, в которую требуется добавить свойство, название будущего свойства и список атрибутов (флагов) данного свойства.

К флагам относятся:

1. value – значение.
2. writable – если true, свойство можно изменить, иначе оно только для чтения.
3. enumerable – если true, свойство перечисляется в циклах, в противном случае циклы его игнорируют. Так же побочным эффектом является видимость свойства при выводе объекта в консоль.
4. configurable – если true, свойство можно удалить, а эти атрибуты можно изменять, иначе этого делать нельзя.

Также мы имеем возможность запретить любые изменения в нашем объекте, используя метод Object.preventExtensions(). Как аргумент используется объект, изменение которого требуется запретить.

Рассмотрим подробный пример:

var forest = {} //создадим пустой объект  
//Добавим новое свойство  
Object.defineProperty(forest, 'count\_trees', {  
 value: 145, //значение - 145  
 writable: true, // Можно изменять значение по ходу программы  
 enumerable: false, // сделаем наше свойство невидимым при выводе  
 configurable: false // его нельзя дополнить или удалить  
});  
forest.count\_trees = 325; //значение будет изменено, т.к мы разрешили writable  
console.log(forest.count\_trees); //325  
Object.preventExtensions(forest); //запретим изменение объекта  
forest.foo = "bar";//попробуем создать новое свойство  
console.log(forest.foo);   
//undefined, т.к создание нового свойства было проигнорировано из-за preventExtensions  
console.log(forest);  
//{} пустые скобки т.к foo был проигнорирован, а наше свойство count\_trees мы сделали невидимым для взломщика.

**Служебные методы и use strict**

**Строгий режим**

JavaScript – ваш верный друг и товарищ, ни на секунду не сомневающийся в вас, как в программисте. Поэтому 80% ошибок будут пропущены, и ваш верный пёс продолжит выполнять программу, бегая за невидимыми палками и принося их вам. Получая от вас сухой корм и слушая, что это для питья, ваш верный друг будет приводить типы данных таким образом, чтобы это действительно утоляло его жажду.

При запуске массивной программы не очевидно, где именно и каким образом интерпретатор JavaScript помогает вам, а также в каких местах игнорируется отсутствие значений в переменных и т.д.

В таком случае программисты используют строгий режим, прописывая в начале документа ‘use strict’ (в одинарных кавычках).

Строгий режим не позволяет интерпретатору JavaScript что-то менять по ходу выполнения вашей программы. При использовании прозрачной палки ваш пёс подаст голос в знак того, что палка отсутствует, и остановит выполнение программы. Строгий режим сильно упрощает отладку кода и поиск ошибок в логике программы.

**Создание служебных методов**

Служебные методы — это все методы, которые уже встроены в JavaScript, они вызываются с помощью глобального объекта Object, который есть в JavaScript с момента проектировки языка. Вы уже сталкивались с некоторыми из них при работе со свойствами объекта. Служебные методы можно вызывать из любой части программы, не задумываясь о ее области видимости. Если добавить свой метод в список служебных, он также станет доступным в любой части программы, то есть станет глобальным.

Для того чтобы добавить свой метод в список служебных методов, воспользуйтесь командой Object.\_\_proto\_\_. В основном взаимодействие в JavaScript происходит с объектами (так принято, и язык к этому располагает). Предполагается, что мы создаем объект с определенным методом и помещаем этот объект в глобальную область видимости, тем самым оставляем себе возможность вызывать его методы глобально.

Таким же образом можно добавлять методы и к созданным вами объектам, нужно только заменить Object на имя вашего объекта, Например, forest.\_\_proto\_\_.

Например:

//Создадим 3 леса  
var forestOne = {  
 count\_trees: 123,  
};  
var forestTwo = {  
 count\_trees: 421,  
};  
var forestThree = {  
 count\_trees: 276,  
};  
//Создадим объект с массивом лесов  
var forests = {  
 array: [forestOne, forestTwo, forestThree]  
};  
//создадим метод для подсчета деревьев во всех лесах  
var method = {  
 method\_forests: function (forests) {  
 var sumTrees = 0;  
 forests.forEach(function (element) { sumTrees += element.count\_trees});  
 console.log(sumTrees);  
 }  
}  
Object.\_\_proto\_\_ = method; //Добавим метод в список служебных методов  
Object.method\_forests(forests.array); // Выведет 820

**CallBack функция**

В JavaScript многие методы требуют от вас информации, что же им делать с теми или иными элементами, попадающими в этот метод. В примере выше был использован метод для переборки массива forEach. Как аргумент он принимает функцию. Такие функции называются CallBack, также они используются при асинхронном программировании, которое в этом курсе мы затрагивать не будем. В каждом методе они могут принимать разные аргументы (эти аргументы выбираете не вы, они уже прописаны в JavaScript), поэтому тут вопрос только вышей будущей эрудиции, но большую часть мы пройдем в дальнейших уроках.

Давайте разберем forEach поподробнее:

var array = [2, 5, 3, 11];  
//Выведем информацию о каждом элементе массива используя максимум аргументов callback  
array.forEach(function (element, index, array) {  
 console.log("Value = " + element + " index = " + index + " for array = " + array)  
});  
/\*   
Value = 2 index = 0 for array = 2,5,3,11  
Value = 5 index = 1 for array = 2,5,3,11  
Value = 3 index = 2 for array = 2,5,3,11  
Value = 11 index = 3 for array = 2,5,3,11  
\*/

**Задержка**

В JavaScript, если вы хотите вызвать какую-либо функцию с задержкой, воспользуйтесь функцией setTimeout или setInterval. Отличие в том, что первая выполняет одно действие с задержкой, а вторая выполняет действие постоянно с интервалом, пока не выполнится условие прерывания. Чтобы прервать функцию setInterval, воспользуйтесь clearInterval.

Например, для setTimeout:

function sayHi(phrase, who) {  
 console.log( phrase + ', ' + who );  
}  
// Аргументы функции: название вызываемой функции, время в миллисекундах, аргументы  
setTimeout(sayHi, 1000, "Привет", "Джон"); // Привет, Джон

Например, для setInterval:

function forSetInterval() {  
 console.log('tick');  
}  
// вывести tick с интервалом в секунду  
var timerId = setInterval(forSetInterval, 1000);  
  
function forSetTimeout() {  
 clearInterval(timerId); //Функция для прерывания setInterval  
 console.log('stop');  
}  
// остановить вывод через 5 секунд и вывести stop  
setTimeout(forSetTimeout, 5000);