1. Введение

<https://www.youtube.com/channel/UCdnFX7mzgup9moXG2fULOog>

2. Инструкции

В этих уроках мы будем сначала говорить об абстрактных вещах, и поэтому если поначалу не всё для вас будет полностью понятно, просто попытайтесь запомнить то, что говорится и в будущем, когда мы приступим к кодингу вы всё это без труда поймете. Мы будем идти от общего к частному, и поэтому мы начнем с ***инструкций***. Все программы на языке Javascript состоят из инструкций, а когда я буду объяснять вам всякие понятия, я буду использовать кириллицу, то есть я буду так и писать на русском ***инструкция*** вместо какого-то кода, и вы всё поймёте. На самом деле мне не очень-то нравится слово ***инструкция*** для описания этого понятия, но, похоже, что ни каких других вариантов у нас нет. ***Инструкции*** *также можно называть* ***командами***, но такое название мне нравится ещё меньше. Чуть больше мне нравится слово ***оператор***, но оно уже занято для другого понятия, с которым мы тоже вскоре познакомимся. Поэтому будем использовать слово ***инструкция***, и также, возможно, иногда мы будем называть их просто ***Statement***-тами [ ˈsteɪtmənt ]. Просто потому, что так это называется на английском. И на самом деле нам удалось бы избежать всяких сложностей с названиями, если бы мы просто называли вещи так же как они называются на английском. Например в некоторых русскоязычных источниках, когда пишут про марджины (***margin***[ˈmɑːdʒɪn ]) в CSS их называют ***полями***, но CSS –код мы всё равно пишем на английском и мы часто пишем слово ***margin*** и все отлично понимаем что оно означает. Так зачем называть это как-то иначе если можно просто говорить «***margin***» и все поймут. Так вот все программы на Javascript состоят из ***инструкций*** и ***инструкции*** выполняются по очереди, и все ***инструкции***, кроме ***составных инструкций***, разделяются точками с запятой (**;**), а Javascript, в общем случае игнорирует пустые символы и переносы строк, поэтому обычно ***инструкции*** пишут на разных строчках. Когда вы пишите инструкцию на разных строчках, вы можете опускать точки с запятой в конце, потому что ***Parser*** [ ˈpɑːsər ] ***(****синтаксический* ***анализатор)*** будет вставлять их за вас. Но когда говорят, что вы можете что-то делать, это не всегда значит что вы должны это делать. В этом случаем можно сказать, что вы всегда должны ставить точки с запятой в конце ***Statement***, потому что это позволит избежать вам трудно уловимых ошибок в будущем. В конце ***составных инструкций***, как говорилось ранее, точку с запятой ставить не нужно, и для объединения нескольких ***инструкций*** в ***составную инструкцию*** используется ***блок инструкций*** (***Block*** [ blɑːk ] ***Statement***) и пишется он так :

|  |
| --- |
| 1. { 2. *инструкция;* 3. *инструкция;* 4. *инструкция;* 5. } |

### Блок инструкций можно использовать в тех местах, где *Parser* увидит только одну *инструкцию* , а вы хотите использовать несколько *инструкций*. Так же в скрипте могут быть *пустые инструкции (Empty* [ˈempti] *Statement)*. Если ввести в скрипте насколько точек с запятой, то в *Parser* мы увидим, что мы создали несколько *пустых инструкций*. Пустые инструкции мы обычно будем использовать там где *Parser* ожидает увидеть какую-либо инструкцию, но мы не хотим, что бы что-то выполнялось. Это один из случаев, когда они применяются. И второй случай, чуть более интересный, в некоторых скриптах вы можете увидеть что-то вроде этого:

|  |
| --- |
| ;(function(){}()); |

### Это *самовызывающаяся функция* (*Self-invoking function* [self ɪnˈvəʊkɪŋ ˈfʌŋkʃn]). Зачем это нужно, мы будем говорить в соответствующем уроке, но сейчас нас волнует первый символ (;). Дело в том, что перед тем, как выкладывать сайт в *production* [prəˈdʌkʃn] *(производство)* считается хорошей привычкой сжимать и конкатенировать *(concatenate* [ kənˈkæt.ə.neɪt ]*(сцеплять, объединять))* все ваши скрипты в один файл. После того как вы это сделаете и в одном файле будут идти, возможно сначала чей-то чужой скрипт, потом ваш скрипт, потом ещё чей-нибудь скрипт. И если в последней *инструкции* перед вашим скриптом будет пропущена точка с запятой (;), то это, вероятно, создаст какую-нибудь ошибку в коде, поскольку *Parser* будет пытаться прочитать *начало вашего скрипта* как *продолжение последнего* *Statement* в этом скрипте. Поэтому, если вы поставите точку с запятой в начале скрипта, вы таким образом обезопасите себя от потенциальных проблем, при этом, если в конце предыдущего скрипта стоит точка с запятой, то здесь будет просто *Empty Statement* который ни на что не повлияет. Существует какое-то конечное количество типов инструкций о каждом из которых мы будем подробно говорить в соответствующем уроке. Но само понятие *инструкции* является, скажем так, корневым как в языке *Javascript* так и в остальных высокоуровневых языках, и поэтому мы будем использовать его часто. Самым простым типом инструкций является *инструкция выражения*, и о *выражениях* мы поговорим в следующем уроке.

### 3.Выражения и операторы

### В предыдущем уроки мы говорили об *инструкцях*. Если говорить русским языком, то *инструкции* можно назвать предложениями языка. В русском языке предложения разделяются точками, а в предложениях *Javascript* они разделяются точками с запятой. И если *инструкции* сравнимы с предложениями, то *выражения* в *Javascript* сравнимы с отдельными фразами языка. И если *инструкции* в программе просто выполняются, то *выражения* всегда возвращают какие-то значения. Когда интерпретатор видит какое-то *выражение*, он вычисляет его значение заменяет *выражение* его значением. *Выражения*, бывают простыми и сложными, или первичными и составными. Простыми называют те выражения, которые не включают в себя другие выражения. К простым выражениям относятся идентификаторы(sorax;) литералы (456321;) и некоторые ключевые слова (например this;). Про идентификаторы мы поговорим, когда будем говорить про переменные. Литералы – это данные которые появляются непосредственно в программе, то есть когда *Parser* видит идентификаторы, ему сначала нужно вычислить его значение, в случае с литералом его значение будет таким же каким мы его видим в коде программы. Выше указан числовой литерал, о других типах литералов мы будем говорить тогда, когда будем говорить о типах данных. Если внести такое значение в интерпретатор, то мы можем убедиться, что вернётся такое-же значение. Все три *инструкции* которые мы написали называются *инструкциями* *выражения (Expression* [ ɪkˈspreʃn ] *Statement)*. И для того, чтобы объединить несколько простых выражения в одно сложное можно использовать оператора. Операторы бывают унарными(+1), бинарными(1+1) и один тернарный оператор (?). И различаются они количеством операндов. Для того, чтобы инструкция была хоть как-нибудь нам полезна, в ней должно быть хотя бы одно выражение с так называемым побочным эффектом (*Side Effect*). Если мы напишем просто такое выражение (*4586+8965*), то это абсолютно правильное во всех отношениях выражение и это правильная инструкция и это правильны скрипт, если мы его запусти то мы не получим какой-нибудь ошибки , но он не делает ничего полезного . Интерпретатор просто вычислит значение этого выражения и сразу же его забудет. Но если мы, например, присвоим значение этого выражения какой-нибудь переменной (*num = 4586 +8965* ), то это выражение уже изменяет что-то за пределами нашего скрипта , поскольку это значение записывается в оперативную память компьютера , и поэтому, можно сказать, что оператор присваивания (=) – это оператор с побочным эффектом. Чтобы этот скрипт работал, нам сначала нужно объявить переменную, с именем *num* (*var num*). Про переменные мы будем говорить в отдельном уроке. Если в инструкции нет ни одного выражения с побочным эффектом, то в такой инструкции нет никакого смысла.

13. Функции

***Функции*** – это какое-то количество инструкций или кусок кода, который объявляется один раз и потом может быть вызван какое-либо угодно количество раз. Если в вашем скрипте вы повторяете какие-то строчки кода больше одного раза то это уже хороший повод, чтобы вынести их в отдельную функцию. Объявление функции выглядит таким образом:

Ключевое слово ***function*** [ ˈfʌŋkʃn ] потом идёт ***выражение*** которое будет являться именем нашей ***функции***, после него идёт список (***аргументов***) в круглых скобках через запятую после чего в фигурных скобках мы пишем {***инструкции***}. Каждый из ***аргументов***, строго говоря, тоже представляет из себя ***выражение***, но мы абстрагируемся до ***аргументов***.

***function******выражение*** *(****аргументы****){*

***инструкции***

***return выражение***

}

***аргументы*** - это переменные, которые мы передаём в нашу ***функцию***, внутри функции мы можем что-то делать с этими переменными и после этого мы можем вернуть из ***функции*** какое-то значение. Для этого предусмотрено ключевое слово ***return*** [rɪˈtɜːn].

Сразу посмотрим на какой-нибудь банальный пример, объявим ***функцию*** которая просто будет *конкатенировать (****concatenate*** [kənˈkæt.ə.neɪt]***(сцеплять, объединять))*** две строки:

***function******greet*** *(****name****){*

***return “Hello” + name***

}

***(greet****(****Sorax****));*

То, что мы написали здесь – называется ***Function declaration statement*** [ˈfʌŋkʃn dek.ləˈreɪ.ʃən ˈsteɪtmənt] ***(инструкция объявления функции)***

Для того, чтобы теперь вызвать эту ***функцию***, мы используем выражение вызова, которое имеет такой синтаксис: ***выражение*** *(****аргументы****).* При этом в точку вызова, то есть там где мы прописали это выражение, вернётся значение которое мы возвращаем из нашей ***функции***. В том случае если функция ни чего не возвращает, то она возвращает ***undefined*** [ ʌndɪˈfaɪnd ]***(не определено).***В том случае что если мы просто вызовем ***функцию*** таким образом, то мы, разумеется, не сможем увидеть результаты её выполнения, поскольку интерпретатор просто вычисляет значение этого выражения и сразу его забывает, но мы можем вынести значение этого выражения в консоль:

***function******greet*** *(****name****){*

***return “Hello” + name***

}

***console.log(greet****(****Sorax****));*

Поскольку мы знаем, что ***функция*** возвращает строку мы можем, например, сразу вызывать какие-нибудь методы объекта *string*:

***function******greet*** *(****name****){*

***return “Hello” + name***

}

***console.log(greet****(****Sorax****).toUpperCase());*

***Function definition expression*** [ˈfʌŋkʃn defɪˈnɪʃn ɪkˈspreʃn ] ***(ыражение определения функции)***

***Функцию*** можно также определить, если присвоить какой-то переменной определение ***функции***. В выражении определения ***функции*** можно пропускать имя ***функции*** (***~~greet~~***), такая ***функция*** называется **анонимной.**

***function******greet*** *(****name****){*

***return “Hello” + name***

}

***var greet function******~~greet~~*** *(****name****){*

***return “Hello” + name***

}

***console.log(greet****(****Sorax****).toUpperCase());*

Выражение определения ***функции*** (***Function definition expression***) и инструкция объявления ***функции*** (***Function declaration ststement***) – это разные вещи и интерпретатор понимает их по разному.

***Function definition expression***

**≠**

***Function declaration ststement***

И поскольку в этом примере это обычная инструкция объявления с инициализацией то здесь после закрывающейся фигурной скобки мы должны поставить иочку с запятой(;) в первом примере точка с запятой была не нужна.

***var greet function****(****name****){*

***return “Hello” + name***

}**;**

***console.log(greet****(****Sorax****).toUpperCase());*

Поскольку ***функции*** в JS являются объектами, лучше объявлять ***функции*** таким образом, поскольку такое объявление лучше отражает суть происходящего. Перед тем как разобрать пару примеров того, что ***функция*** может вести себя как объекты, узнаем ещё кое-что об ***аргументах***. Если мы передаём в ***функцию*** больше ***аргумент***ов, чем ожидаем получить, то мы не получаем никакой ошибки, просто внутри ***функции*** этому ***аргумент***у не будет присвоено ни какое имя. Это, тем не менее не значит, что мы не сможем использовать этот ***аргумент*** внутри ***функции***, но об это позже. Если мы передаём в ***функцию*** меньше ***аргумент***ов чем ожидаем получить, то опять-таки мы не получаем ни какой ошибки, и всем не инициализированным ***аргумент***ам присваивается значение ***undefined***.

***var greet function****(****name****){*

***return “Hello” + name***

}**;**

***console.log(greet****(****Sorax,34,23,25****).toUpperCase());*

*//* ***undefined***.

Для того, чтобы получить все переданные ***аргумент***ы внутри ***функции*** мы можем использовать объект ***arguments*** [ ˈɑːɡ.jʊ.mənts ] .

***var greet function****(****name****){*

***console.log(arguments****);*

***return “Hello” + name***

}**;**

***console.log(greet****(****Sorax,34,23,25****).toUpperCase());*

*//**[****Sorax,34,23,25****]*

Это объект ведет себя как массив, он выводится в консоль как массив *[****Sorax,34,23,25****]* и так же мы можем обращаться к отдельным ***аргумент***ам при помощи индексов, но тем не менее мы знаем что это объект. Одно полезное свойство этого объекта - это длина; при помощи его мы можем, например, перебрать все переданные ***аргумент***ы в цикле и что-нибудь с ними сделать.

***var greet function****(****name****){*

***console.log(arguments.length****);*

***return “Hello” + name***

}**;**

***console.log(greet****(****Sorax,34,23,25****).toUpperCase());*

*//* ***4***

Поскольку ***функции*** в JS являются объектами мы можем передавать ***функцию*** в ***функцию*** в качестве ***аргумент***ов, а также возвращать ***функцию*** из ***функции***. Передача ***функции*** в качестве ***аргумент***ов может использоваться для реализации так называемых call back-ов, то есть для передачи в ***функцию***, которая будет выполнена поле выполнении ***функции***, в которую она передаётся.

***var func = function****(****callback****){*

***var name = ”Sorax”****;*

***callback(name);***

};

***func = (function****(****n****){*

***console.log(greet****(****”Hello” + n****);*

*});*

*//* ***Hello Sorax***

И также ***функция*** может быть возвращаемым значением

***var func = function****(){*

***return function****()*

***console.log(”Hi”****);*

*});*

Теперь, когда мы вызываем эту ***функцию***, в точку вызова возвращается анонимная ***функция***, и если мы вызовем эту ***функцию func()();*** , то увидим результат её выполнения в консоли.

***var func = function****(){*

***return function****()*

***console.log(”Hi”****);*

*});*

***func()();***

*//* ***Hi***

***Функция*** может быть вызвана сразу после определения в том случае если мы используем ***выражение*** определения

***var greeting = function****(name){*

***return ”Hello” +*** *name****;***

*}*

После этого выражения определения мы можем прописать круглые скобки с ***аргумент***ами и, таким образом, мы получим ***выражение*** вызова.

***var greeting = function****(name){*

***return ”Hello” +*** *name****;***

*}****(Sorax)****;*

***console.log(greeting****);*

***//Hello Sorax***

В последнем примере мы присваиваем результат выполнения ***функции*** переменной. И если в этой ***функции*** будет много инструкций, то, посмотрев на первую строчку можно будет подумать что мы присваиваем переменной ***функцию***. Хотя, на самом деле мы присваиваем ей значение ***функции***. Для большей ясности в таких случаях выражение вызова берут в круглые скобки. *Функция, которая вызывается сразу после определения*, называется ***анонимной самовызывающейся функцией(self-invoking anonimous function).*** Разумеется она может быть не анонимной и мы можем дать ей имя после ключевого слова ***function***, но самовызывающимся функциям как правило нет ни какого смысла давать какие-то имена. На самом деле *единственный пример* который приходит в голову , чтобы на пришлось *дать имя* самовызывающейся функции, это *рекурсия*, то есть *если мы хотим вызвать эту функцию внутри самой себя* то нам нужно дать ей имя, *иначе мы ни как не сможем её вызвать*.

14. Цепочки областей видимости (*scope chain*).

Из этого урока мы узнаем что такое область видимости переменной (*scope*), что такое цепочки областей видимости (*scope chain*). и таким образом подготовимся к следующему уроку, который будет посвящен (*closures*) замыканиям. Область видимости переменной – это часть программы где эта переменная определена и доступна. Переменный по области видимости делятся на глобальные и локальные. Глобальными называются все переменные объявленные вне каких-либо функций. Переменный объявленные внутри функции являются локальными. Локальная переменная с таким же именем как и глобальная имеет больший приоритет, поэтому если мы попробуем вывести в консоль эту переменную, то мы увидим, что выведется значение локальной переменной.

***var i = 5****;*

***var func = function****(){*

***var i = 10****;*

***console.log(i****);*

*};*

***func();***

***//10***

В языке JS только функции создают локальную область видимости*(function scope)* в отличи, например, от языка С, где локальную область видимости создают любые блоки (*block scop*) c фигурными скобками. Функции в JS могут быть вложенными:

***var i = 5****;*

***var func = function****(){*

***var i = 10****;*

***console.log(i****);*

***var innerFunc = function****(){*

*};*

*};*

***func();***

***//10***

Внутри этой функции может быть ещё одна локальная переменная (***var i = 15****;*)

***var i = 5****;*

***var func = function****(){*

***var i = 10****;*

***console.log(i****);*

***var innerFunc = function****(){*

***var i = 15****;*

*};*

*};*

***func();***

***//10***

И если мы выведем (i), то, очевидно, мы выведем значение этой локальной переменной.

***var i = 5****;*

***var func = function****(){*

***var i = 10****;*

***console.log(i****);*

***var innerFunc = function****(){*

***var i = 15****;*

***console.log(i****);*

*};*

***innerFunc();***

*};*

***func();***

***//10***

Вложенные функции создают так называемые цепочки областей видимости, когда мы обращаемся здесь к переменной (***i***), интерпретатор в первую очередь первую область видимости в цепочке, то есть вложенную функцию (***innerFunc***). Если в этой функции не окажется переменной (), то интерпретатор постарается найти её в следующей области видимости цепочки, то есть переменной (***func***). Если при объявлении глобальных переменных мы, строго говоря, можем не писать ключевое слово (***var***), поскольку при этом ни чего не изменится, то при объявлении локальных переменных мы всегда должны писать здесь (***var***), иначе мы будем менять переменную, которая находится в следующей области видимости в цепочке.

Есть ещё одна интересная особенность объявления переменных, которую назвали подъём (hoisting). Мы уже разобрались с тем, что если бы в нашей внутренней функции переменная () не была бы объявлена, то интерпретатор просто взял бы значение переменной () из внешней функции. Но что будет если мы обратимся к переменной до её объявления?

***var i = 5****;*

***var func = function****(){*

***var i = 10****;*

***console.log(i****);*

***var innerFunc = function****(){*

***console.log(i****);*

***var i = 15****;*

*};*

***innerFunc();***

*};*

***func();***

***// undefined***.

Логично было бы предположить, что когда интерпретатор начнет выполнение этой функции он увидит обращение к необъявленной переменной () и возьмет её из внешней функции. Подобным образом это произошло бы в других популярных языках, но в языке JS интерпретатор уже заранее знает об объявлении всех переменных внутри функции. Поэтому при выполнении этого скрипта в консоль выведется (***undefined***). И здесь возникает логичный вопрос почему в консоль выводится (***undefined***) а не (***15***)? Дело в том, что такое поведение назвали подъём, потому что интерпретатор по сути поднимает все объявления переменных в начало функции и то, что мы записали здесь, аналогично такому коду

***i = 5****;*

***var func = function****(){*

***var i = 10****;*

***console.log(i****);*

***var innerFunc = function****(){*

***var i****;*

***console.log(i****);*

***i = 15****;*

*};*

***innerFunc();***

*};*

***func();***

***// undefined***

Поскольку в начало функции поднимаются только объявления переменных без инициализации, то переменная (***var i***) еще не была инициализирована, потому в консоль выводится (***undefined***). В других языках программирования, например в том же С считается хорошей привычкой объявлять переменные как можно ближе к тому месту где они будут использованы. В языке же Джава скрипт многие программисты всегда объявляют переменные в начале функции, поскольку это лучше отражает суть происходящего. Это, в конце концов то, что интерпретатор сделает за вас в любом случае.

15 Замыкания(***closures***)

При выполнении функция в языке JS используется та область видимости переменных, которая существовала при объявлении этой функции, Это называется лексическая область видимости (***lexical scoping***) и на примере это выглядит таким образом: Объявим функцию (***func***) внутри которой объявим переменную (***i***) которой дадим какое-то значение (***10***) и вернём из этой функции в функцию() которая будет возвращать значение переменной (***i***)

***var func = function****(){*

***var 10 = 10****;*

***return = function****(){*

***return i****;*

*}*

*};*

Теперь объявим ещё одну функцию (***anotherFunc***) внутри которой объявим переменную (***i***) с каким-нибудь другим значением (***20***) и попробуем здесь вызвать функцию которая возвращается из функции (***func***) и выведем возвращаемое значение в консоль (***console.log (func*** *()());*).

***1.var func = function****(){*

***2. var i = 10****;*

***3. return = function****(){*

***4. return i****;*

*5. }*

*6.};*

***7. var anotherFunc = function****(){*

***8. var i = 20****;*

***9. console.log (func*** *()());*

*10. }*

*11.};*

***12.anotherFunc();***

Из-за лексической области видимости в консоль выводится переменная (***i***) объявленная на второй строчке (***10***), а не переменная (***i***) которая объявлена на восьмой строчке (***10***), для ещё большей наглядности можно вначале присвоить какой-то переменной функцию(***myFunc***) возвращаемую из функции (***func***) и затем вызвать эту функцию (***console.log (myFunc*** *()());*)

***1.var func = function****(){*

***2. var i = 10****;*

***3. return = function****(){*

***4. return i****;*

*5. }*

*6.};*

*7.* ***var myFunc = func*** *();*

***7. var anotherFunc = function****(){*

***8. var i = 20****;*

***9. console.log (myFunc*** *()());*

*10. }*

*11.};*

***12.anotherFunc();***

Результат при этом не изменится. С каждой функцией связана цепочка областей видимости переменных и функция вместе с этой цепочкой называется ***замыканием.*** Поэтому, строго говоря, ***каждую функцию можно назвать замыканием***. Всё это возможно благодаря тому, что когда интерпретатор выполняет какую-либо функцию, например, когда он выполняет нашу первую функцию, он создаёт объект, который будет содержать все локальные переменные для этого вызова функции. В частности нашу переменную (***i***). И любые функции, которые мы определяем внутри этой функции (***func***), например наша возвращаемая функция (***return = function****()*), сохраняют ссылку на этот объект с локальными переменными. И этот объект существует только в том случае, если есть фуекции, которые на него ссылаются. Если бы в этой функции не было ни каких вложенных функция, то у нас не было больше бы ни какого способа получить доступ к этой переменной () и у интерпретатора не было бы больше ни какого повода, чтобы хранить эту переменную после выполнения функции и поэтому объект, содержащий эту переменную был бы уничтожен (garbage collection) сразу после выполнения функции.

Теперь посмотрим на какой-то более практический применимый пример.

Напишем простой счетчик.

***var counter = (function****(){*

***var count = 0****;*

***return = function****(){*

***return count++****;*

*}*

*}());*

***console.log (counter*** *());*

***console.log (counter*** *());*

***console.log (counter*** *());*

*//0*

*//1*

*//2*

Замыкания очень удобно создавать при помощи анонимных самовызывающихся функций, поэтому во многих скриптах вы сможете встретить такой код, где какой-либо переменной присваивается функция, которая возвращается из анонимной самовызывающейся функции. В написанной функции мы возвращаем значение переменной (***count***) увеличенной на единицу(***count++***). И в этом замыкании инициализируем нашу переменную (***var count = 0***). Теперь попробуем вызвать эту функцию и вывести результат её выполнения в консоль (***console.log*** *(****counter*** *();*). Когда мы первый раз вызываем нашу функцию то в консоли выводится (0). Второй раз в консоль выведется (1). В третий (2) и так далее. Таким образом мы довольно просто создали функцию, которая не принимает ни каких аргументов и при этом возвращает разные значения при разных вызовах. И в этом и заключается вся прелесть замыканий, они позволяют нам использовать функции уже не просто как подпрограммы, они позволяют нам сделать функции немного более умными благодаря, как в нашем примере, возможности сохранения каких-то данных и также нужно отметить, что замыкание позволяет нам реализовать что-то вроде инкапсуляции данных. В нашем примере мы можем получить переменную (***counter***) или изменить переменную (***counter***) только при помощи вложенных функций, и поэтому переменную (***counter***) можно условно считать приватной. Мы можем немного улучшить нашу функцию добавив возможность изменения значения счетчика, Эта функция будет принимать какое-то число (), и если этот аргумент был передан, то есть, если переменная () не равна () мы присваиваем счетчику это число, иначе значение остаются тем же.

***var counter = (function****(){*

***var count = 0****;*

***return = function****(****num****){*

***count = num !== undefined ? num : count;***

***return count++****;*

*}*

*}());*

***console.log (counter*** *());*

***console.log (counter*** *(500));*

***console.log (counter*** *());*

*//0*

*//501*

*//2*

Когда мы первый раз вызываем нашу функцию то в консоли выводится (0). Второй раз в консоль выведется уже (501). В третий (502) и так далее. Раз мы взяли счетчик в качестве примера, здесь нужно также сказать, что замыкание – это не единственный способ чтобы добиться такого поведения. Поскольку функции являются объектами, мы можем просто добавить нашу переменную () как свойство объекта () и внутри функции мы можем изменять это свойство.

***var counter = (function****(****num****){*

***return counter.count= num !== undefined ? num : counter.count;***

*};*

***counter.count =0;***

***console.log (counter*** *());*

***console.log (counter*** *(500));*

***console.log (counter*** *());*

*//0*

*//501*

*//2*

24. Массивы

Массивом называется упорядоченный набор элементов, каждый из которых имеет свой порядковый номер, который называется индекс(). Литерал массива выглядит так ()

***var array = [1, 3, 4]****;*

К отдельным элементам нашего массива можно обращаться таким образом:

***var array = [1, 3, 4]****;*

***console.log (array[2]);***

Здесь эти квадратные скобки это на самом деле оператор поэтому после выражения и перед скобками может быть какое угодно количество пробелов. Индексация как и в большинстве других языков начинается с 0 (0=based indexing)