# Занятие 2 - Контрольные структуры. Ошибки.

*"Самые лучшие уроки мы получаем, когда делаем ошибки,   
ошибка прошлого — мудрость будущего."*

-— Дейл Тернер

## Булевые значения

На прошлом занятии мы познакомились с некоторыми из используемых в JavaScript типов - целыми и дробными числами и строками. Еще один тип, с которым познакомимся сегодня и будем активно работать - это так называемый **булевый** тип (boolean). Назван он так в честь английского математика Джорджа Буля, который одним из первых занимался проблемами математической логики. Значения булевого типа могут быть только одним из двух вариантов - **правдой** или **ложью**. В логике эти значения являются основополагающими.

const programmingIsCool = true;

const numberIsOdd = false;

Как видите, в JavaScript булевый тип определяется с помощью специальных зарезервированных слов true и false, которые означают правду и ложь соответственно.

## Ветвления в программе

Мы подобрались вплотную к одной из основополагающих концепций в программировании - **ветвлениям**. Мы уже примерно знаем, что такое ветвления, из предыдущих занятий и упражнений по алгоритмам и псевдокоду. Ветвления помогают нам изменять течение программы в зависимости от некоторого **условия**.

Пример ветвления в алгоритме покупки в магазине, при наличии или отсутствии дисконтной карты:

* Рассчитать общую стоимость покупок клиента
* Если у клиента есть дисконтная карта:
  + Общая стоимость = общая стоимость - процент дисконтной карты
* Выдать клиенту общую стоимость для оплаты

Или, например, алгоритм похода в кинотеатр:

* Добраться до кинотеатра
* Войти в кинотеатр
* Подойти к кассе кинотеатра
* Если есть свободные места на ближайший сеанс:
  + Купить билеты
  + Дождаться сеанса
  + Посмотреть фильм
* Иначе:
  + Немного погрустить
* Поехать домой

В примере с дисконтной картой проверяется условие "у клиента есть дисконтная карта". Оно может быть **правдой** - карта есть, или **ложью** - карты нет. В случае правды мы производим дополнительное действие (вычитаем скидку из общей суммы). Если условие ложно, то мы не выполняем это действие и переходим сразу к выводу суммы к оплате.

В примере с кинотеатром проверяется условие "есть свободные места на сеанс". Это условие может быть также правдой- места действительно есть, или ложью - мест нет. В случае, когда выражение в условии получилось правдой, мы производим определенную последовательность действий (покупаем билет, смотрим фильм), а иначе - другие действия (грустим). Потом алгоритм снова сходится к единой линии поведения - независимо от того, смотрели ли мы фильм, в конце концов мы все равно возвращаемся домой.

В JavaScript мы тоже можем проверять на значения правда или ложь, и при этом выполнять разный код. Это достигается с помощью условной конструкции **if**:

const isWinner = true;

if (isWinner) {

alert('Congratulations, you win!');

}

В этом коде у нас появляется новый синтаксис, в виде фигурных скобок. Итак, разберем этот пример:

* Сначала идет ключевое слово if - которое указывает, что началось ветвление. Переводится как "если".
* Затем в скобках пишется условие. В нашем случае у нас там находится переменная, содержащая булевое значение true.
* Открывающая фигурная скобка обозначает начало блока кода, который должен выполниться, когда условие будет правдивым.
* Некоторый код следует после фигурной скобки. В нашем случае - это просто alert, выводящий сообщение на экран.
* Закрывающая фигурная скобка показывает конец блока кода.

Здесь следует отметить, что если условие, стоящее в скобках, будет ложью, то код в блоке if не выполнится вообще.

В блоке if может быть сколько угодно строк кода, и даже другие конструкции.

Такой блок if обеспечивает выполнение алгоритма, наподобие первого примера (если есть дисконтная карта, то выполняются дополнительные действия).

Если нам надо выполнить алгоритм второго примера, то у блока if есть еще специальное продолжение else, позволяющее указать блок кода, который должен выполниться в противном случае, если условие в скобках if не выполняется (или другими словами, является **ложью**):

const isWinner = true;

if (isWinner) {

alert('Congratulations, you win!');

} else {

alert('Sorry, you lost');

}

В этом случае, блок else идет сразу после закрывающейся фигурной скобки, а затем открывается новый блок кода. В конце этот блок кода также закрывается фигурной скобкой.

В настоящем примере блок кода в else никогда не выполнится, поскольку переменная isWinner, служащая условием, всегда равна true.

## Логические выражения

Как еще мы можем получать булевые значения? Одно из наиболее частых способов - использование логических выражений. Вы уже сталкивались с подобными в математике - когда нужно сравнить два числа:

> 6 < 9

< true

> 10 > 100

< false

Как видите, мы можем сравнивать числа, используя знакомые нам логические выражения: <, >, >=, <=.

Для сравнения на равенство мы можем использовать символ == - двойное равно, поскольку одинарное равно - это, как мы уже знаем, оператор присваивания.

> 10 == 10

< true

Также существует оператор "не равно", который возвращает false только в том случае, когда значения равны, и наоборот true, когда значения не равны:

> 10 != 10

< false

> 3 != 2

< true

Мы можем сравнивать строки на равенство:

> 'Hello' == 'Hello'

< true

> 'Hello' == 'HELLO'

< false

Строки равны друг другу только в случае полного совпадения всех символов в них.

Однако, если мы сравним строку, содержащую число с числом, мы получим неожиданный результат:

> '123' == 123

< true

> '' == 0

< true

Дело в том, что оператор "двойное равно" сравнивает без учета типа, и автоматически приводит строку к числу для сравнения, если один из операндов является числом. В связи с этим, общая рекомендация - никогда не сравнивать на равенство с помощью двойного равно, так как могут возникать проблемы при сравнении данных разных типов. Вместо этого, мы всегда будем пользоваться оператором "тройное равно", который сравнивает с учетом типа:

> '123' === 123

< false

> '' === 0

< false

Если сравнивать строки операторами > (больше) или < (меньше), то результат будет зависеть от лексикографического порядка данных строк, то есть, например:

> 'Hello' < 'Greek'

< false

Так как заглавная буква "G" стоит раньше, чем "H" по логике алфавита, а также по их кодам таблицы кодировки (ASCII, UTF-8), то в словаре слово "Greek" будет находиться раньше чем слово "Hello", и поэтому мы получаем false. Это и означает "лексикографический порядок".

В результате всех этих логических операций мы получаем обычное булевое значение, true или false, которое можем далее применять в блоках if-else для ветвления нашей программы.

const ageFromUser = prompt('How old are you?');

const age = parseInt(ageFromUser);

if (age >= 13) {

alert('You may access this content');

} else {

alert('You are not 13 yet, please leave this site!');

}

## Логические операторы И, ИЛИ, НЕ

Также, как и в логике, мы можем комбинировать логические выражения, чтобы получать сложные условия. Посмотрим пример алгоритма из реальной жизни, чтобы лучше понять, зачем это может понадобиться:

Алгоритм покупки в супермаркете:

* Покупаем нужные товары.
* Приходим на кассу.
* Если у нас есть с собой дисконтная карта:
  + Даем карту кассиру для расчета скидки
* Получаем общую сумму к оплате от кассира
* Если у нас есть (наличные И их достаточно) ИЛИ (банковская карта И на ней достаточный баланс):
  + Оплачиваем и упаковываем покупки
* Уходим домой.

Первое условие простое: если у нас есть с собой дисконтная карта, то мы можем получить скидку. Второе условие уже более сложное - мы можем расплатиться наличными или банковской картой. Поэтому, если даже у нас нет достаточно наличных, мы можем расплатиться картой, но при этом на ней тоже должен быть достаточный положительный баланс. Здесь учитываются четыре базовых условия:

1. Есть наличные в кошельке
2. Наличных достаточно для покупки
3. Есть банковская карта
4. На банковской карте достаточный баланс.

1 и 2, а также 3 и 4 условия комбинируются с помощью логического оператора И, затем две получившиеся группы комбинируются с помощью логического оператора ИЛИ.

Разберем этот пример.

Представим, что при прохождении алгоритма мы получили сумму к оплате, равную 500 долларов. У нас с собой есть наличными 15 долларов, но у нас также есть банковская карта с 500 долларами на счету.

Есть наличные в кошельке И наличных достаточно для покупки

правда И ложь = ложь

Есть банковская карта И на банковской карте достаточный баланс

правда И правда = правда

Получается, что первая группа условий дают нам ложь. Вторая группа условий нам дает правду.

Первая группа условий ИЛИ вторая группа условий

ложь ИЛИ правда = правда

Значит, при заданных условиях наше сложное логическое выражение определяется как правда, и мы сможем произвести покупку.

Также как и в реальной жизни в JavaScript есть логические операторы:

|  |  |  |  |
| --- | --- | --- | --- |
| И (AND) | && | > true && true  < true  > true && false  < false  > false && false  < false | Равно true только если оба операнда равны true. В остальных случаях равно false. |
| ИЛИ (OR) | || | > true || true  < true  > true || false  > true  > false || false  > false | Равно false только если оба операнда равны false. В остальных случаях равно true. |
| НЕ (NOT) | ! | > !true  < false  > !false  < true | Является инверсией. Возвращает противоположное значение. |

Пример:

const number = parseInt(prompt('Enter a number from 1 to 100:'));

if (number < 1 || number > 100) {

alert('You entered invalid number!');

} else {

alert('Thanks!');

}

Мы можем комбинировать условия с помощью скобок, что изменяет порядок действий на нужный нам. Сымитируем условие из алгоритма, который мы разбирали ранее:

const hasCash = true;

const cashAmount = 15;

const hasCard = true;

const cardAmount = 500;

const total = 20; // сколько надо заплатить

if ((hasCash && cashAmount >= total) || (hasCard && cardAmount >= total)) {

alert('Hooray! Can buy goods!');

} else {

alert('Sorry, you cannot buy goods....');

}

## Confirm

Есть еще одна встроенная в браузер функция, которая позволяет получать нам именно булевые значения: confirm.

const userWantsCoffee = confirm('Do you want coffee?');

if (userWantsCoffee) {

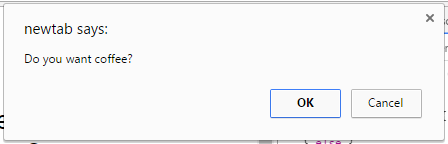
alert('Here, have some coffee!');

} else {

alert("Well, it's your choice");

}

Функция confirm выведет на экран такую табличку с текстом, который вы "передаете" этой функции аргументом.



Если вы нажмете OK, то функция "вернет" вам значение true, если Cancel (Отмена) - то false.

Значит, в зависимости от нажатия на эту кнопку, переменная userWantsCoffee будет содержать либо false, либо true. Ну а потом - дело техники, с помощью конструкции if-else мы можем обработать такой ввод и вывести разные сообщения. Ну, либо сделать что-то еще.

## Else-if

Рассмотрим еще один вариант конструкции if-else в случае, если необходимо делать разветвление программы не по 2-м веткам, а более. Например:

const number = parseInt(prompt('Please, enter an integer:'));

if (number > 0) {

alert('Your number is positive');

} else if (number < 0) {

alert('Your number is negative');

} else {

alert('Your number is zero');

}

В этом случае у нас есть три варианта - когда число положительное, отрицательное, или ноль. Конструкция else if позволяет проверить на дополнительное условие, когда первое условие было ложным. Таким образом можно "соединять" сколько угодно дополнительных условий, в случае, если вам нужно сделать несколько ветвлений:

const grade = parseInt(prompt('Please enter your grade (1-100)');

if (grade < 1 || grade > 100) {

alert('Please enter correct grade');

} else if (grade >= 90) {

alert('This is A grade');

} else if (grade >= 80) {

alert('This is B grade');

} else if (grade >= 70) {

alert('This is C grade');

} else if (grade >= 60) {

alert('This is D grade');

} else if (grade >= 50) {

alert('This is E grade');

} else {

alert('This is F grade. You failed');

}

Если запустить данный код и попробовать ввести несколько чисел, можно заметить, что в случае, если срабатывает один из блоков кода, то остальные проверки уже не производятся. Дело в том, что else означает "иначе", и если одно из условий становится правдой, то все остальные "иначе" уже не сработают.

## Switch

Если у нас есть переменная, которая может принимать много значений, и в зависимости от этого значения должны происходить разные действия, есть специальная структура Switch, которая поможет нам в этом:

const day = parseInt(prompt('Enter week day number (from 1 to 7)'));

switch (day) {

case 1:

alert('Monday');

break;

case 2:

alert('Tuesday');

break;

case 3:

alert('Wednesday');

break;

case 4:

alert('Thursday');

break;

case 5:

alert('Friday');

break;

case 6:

alert('Saturday');

break;

case 7:

alert('Sunday');

break;

default:

alert('Invalid day number');

}

Здесь мы указываем в скобках переменную, значение которой будет сравниваться со всеми строками case. Как только произошло совпадение, выполняется блок кода после него и до оператора break, который указывает на конец блока кода. Если из примера убрать все break-и, то выполнятся все строки кода после совпадения case-а.

Блок default как правило находится в конце и туда попадает выполнение в том случае, если ни один case не совпал со значением переменной.

Если надо объединить некоторые условия, то мы можем просто не писать блок кода, а также ключевое слово break в нужных из них. Пример:

const answer = prompt('Are you sure you want to continue? y/n:');

switch (answer) {

case 'y':

case 'Y':

alert('You chose to continue.);

break;

case 'n':

case 'N':

alert('You chose to not continue');

break;

default:

alert('Error');

}

Теперь, если ввести y или Y, будет выполнен блок кода под ними, а если n или N, то блок кода под этими буквами.

## Ошибки. Логические, синтаксические.

Вы уже некоторое время пишете код на JavaScript, делали несколько домашних заданий, и возможно, допускали ошибки. Самое главное - не пугаться. Ошибки бывают нескольких видов. Самая простая - это синтаксическая. Когда вы допускаете ошибку в синтаксисе вашей программы, то есть делаете опечатку, не закрываете скобки или кавычки и ваш код при этом не запускается вовсе.

Понять, что вы допустили синтаксическую ошибку просто - код не запускается, когда вы пробуете его запустить. Понять, где проблема, можно по сообщению, которое выводится в консоль. Если из сообщения непонятно, в чем дело, проверьте все скобки и кавычки. Умные редакторы, как WebStorm способны показать вам синтаксические ошибки даже до того, как вы запустите код. Если вы все перепробовали, попробуйте поставить его, и открыть файл в нем.

С логическими ошибками сложнее - это ошибки в основном по логике приложения. Например, вы принимаете в каком-то месте программы число от 0 до 10, но при этом вы написали условие number > 0 && number < 10, при этом вы "выкинули" граничные значения - 0 и 10 никогда не пройдут в это условие. Такие ошибки находить сложнее, но их можно выявить с помощью контрольных примеров. Это работает в случае, когда вы заранее планируете, рассчитываете некий набор данных на входе и соответственные выходные значения этим входным. Потом проверяете программу, и если значения соответствуют вашим контрольным примерам, то все в порядке. Также, в дальнейших занятиях мы посмотрим, как с помощью тестов автоматизировать процесс проверки программы через контрольные примеры.

Основной совет при встрече с ошибкой - не паникуйте. Внимательно прочитайте сообщение. Обычно там содержится строка и символ исходного кода, на котором произошла ошибка. Если в сообщении ничего не понятно, попробуйте вбить частично это сообщение в Google. Экспериментируйте - передайте другое значение в функцию, проверьте, как она работает, посмотрите ее исходный код. Спросите у технической поддержки курса, или у преподавателя, если совсем ничего не получается.

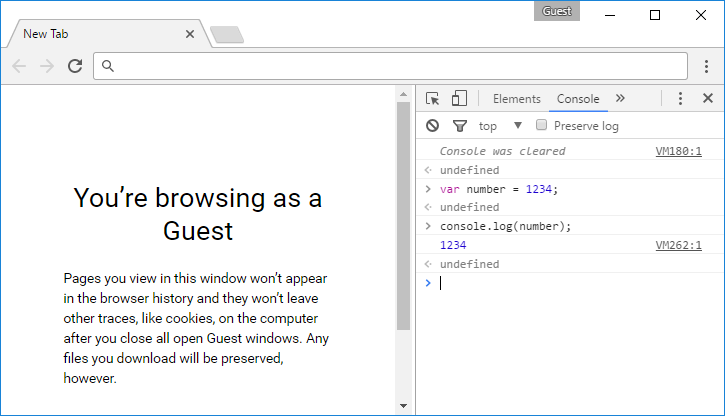
И помните - только ошибаясь, и исправляя свои ошибки, вы сможете научиться.

## console.log

Очень полезным инструментом в отлове ошибок будет console.log. Он позволяет вам выводить значение переменной в консоль, при этом не пользуясь окном alert. Это бывает значительно удобнее в случаях, когда вам нужно вывести несколько значений сразу, и при этом не изменять поведение приложения. Работает это так:

const number = 1234;

console.log(number);



Вы увидите все значения, которые выводите с помощью console.log в консоли по очереди. Иногда бывает полезно перед значением написать какой-то текст. В этом случае можно произвести обычную конкатенацию:

console.log('The number is ' + number);

Или написать текст и значение через запятую (передать вторым аргументом):

console.log('The number is', number);

Во втором случае тип переменной не будет приведен к строковому (т.к. нет конкатенации), что иногда бывает полезно.

## Дебаггер в Google Chrome

В ваш браузер встроен полезный инструмент разработчика - дебаггер JavaScript, который позволяет смотреть, как именно работает программа, какие переменные имеют какие значения в различный момент времени. Напишем такой код в HTML-файл и откроем его в браузере:

let value = 1;

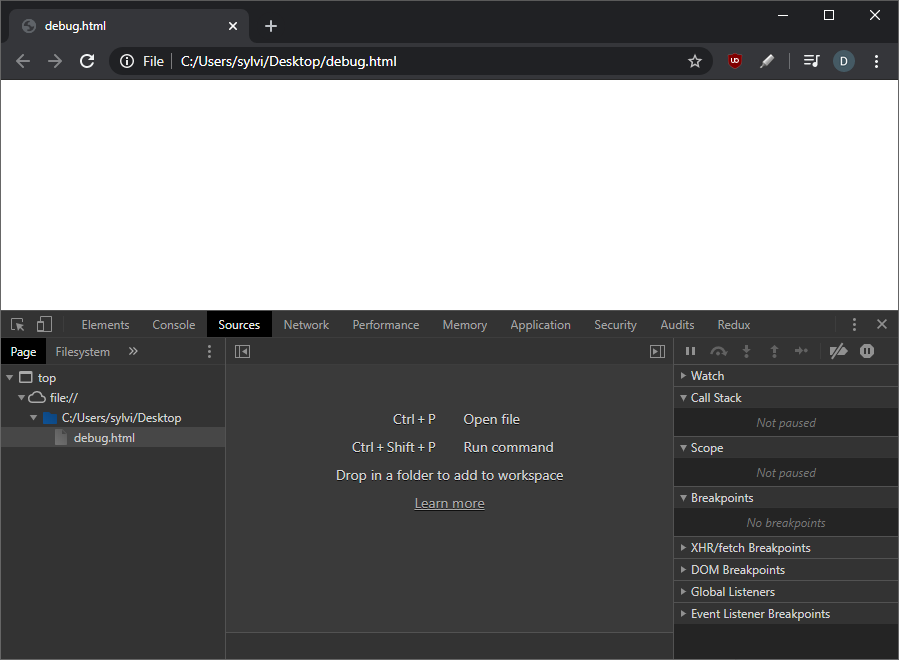
value = value + 100;

value = value - 50;

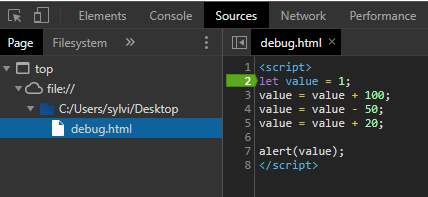
value = value + 20;

alert(value);

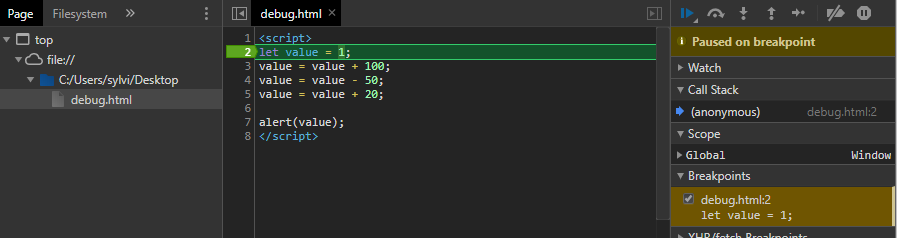
Затем откроем вкладку Sources:



Откроем в левой панели файл, который мы открыли (здесь - это debug.html). Кликнем на номер строки, в которой содержится первая строка программы:



Тем самым мы создали breakpoint - это точка программы, на котором выполнение должно остановиться и дебаггер покажет состояние программы на этот момент. Проверим это - перезагрузим страницу.



Действительно, мы видим сообщение "Paused on breakpoint".

Далее мы можем управлять, как программа должна выполняться. Например, мы можем нажать Resume (первая кнопка сверху) (F8) и программа продолжит выполняться как обычно.

Следующая кнопка (Step over) (F10) - перемещает выполнение программы на одну строку в пределах текущей функции. Мы будем ей пользоваться в основном на данном этапе.

Другие кнопки позволяют посмотреть детали исполнения той или иной функции в случае, если текущая строка кода вызывает эту функцию. Но так как мы пока что не дошли до таких продвинутых вещей, мы их рассматривать не будем.

Нажмем на кнопку Step over несколько раз и убедимся, что в панели справа переменная value отображается и ее значение изменяется в соответствии с тем кодом, который выполняется (можно увидеть значение, наведя мышью на название переменной, либо добавив переменную в Watch).

Так вы сможете просматривать, куда производятся переходы по условиям и чему равны те или иные переменные в нужный вам момент времени.

Убрать breakpoint можно кликнув на него же, либо пунктом меню Remove all breakpoints.