Сравнение значений

Существует два типа сравнения значений:

- проверка равенства;
- проверка неравенства.

Сравнивая значения, мы получаем булево значение — true, то есть «значения равны», или false — «значения не равны» — независимо от того, какие значения мы сравниваем. Причина в том, что JavaScript применяет приведение типов, когда они разные и напрямую их сравнивать нельзя.

Приведение типов

Приведение типов — это механизм, когда один тип данных изменяется на другой. Приведение типов может быть явным и неявным. Явное — когда мы сами меняем тип у значения, а неявное — когда движок JavaScript делает это в необходимых местах. Явно привести типы можно с помощью средств языка, например, вызвав конструктор нужного типа:

```
let numberFromString = Number('12');
console.log(numberFromString - 1); // 11 - строка "12" была переведена в число, и операция вычитания прошла успешно.

let numberToString = String(numberFromString);
console.log(numberToString.length); // 2 - numberToString — это строка, так как у числа нет свойства length
```

Неявное приведение типов может произойти в результате какой-то другой операции. Когда для её выполнения требуется операнд другого типа, чем полученный, движок JavaScript делает приведение несоответствующего типа к нужному.

Правила приведения типов разумны и довольно просты. Есть очень удобные таблицы, по которым можно быстро найти, как преобразовывается один тип в другой. Вот такая таблица на w3schools.

Пример неявного приведения типов:

```
let stringNumber = '12';
let realNumber = 1;
console.log(stringNumber + realNumber); // "121" - получили строку из соединения первой переменной и второй, вторая переменная была преобразована в строку.
console.log(stringNumber - realNumber); // 11 - получили число, так как при
```

```
вычитании строка была преобразована в число.
```

Арифметические операции (кроме сложения!) над не числами всегда вызывают приведение типов.

Значения, приводящиеся к true (Truthy) и false (Falsy)

Если мы приводим любые данные к булеву значению, мы получим `true` или `false`.

По стандарту JavaScript следующие значения и типы приводятся к `false`:

- `""` (пустая строка);
- `0`, `-0`, `NaN` (`0` и `NaN`);
- `null`, `undefined`;
- `false`.

А эти значения и типы приводятся к `true`:

- `"Hello"` (непустые строки);
- `42`;
- `true`;
- `[ ]`, `[ 1, "2", 3 ]` (массивы);
- `{ }`, `{ a: 42 }` (объекты);
- `function foo() { .. }` (функции).

Приведение к булеву типу в операторах условной логики часто происходит автоматически. Но мы можем явно привести любой тип к булеву значению, используя оператор отрицания `(!)`. Этот оператор отрицает или инвертирует текущее булево значение. Он ставится перед именем переменной, и если она не булева типа, то автоматически приводится к булеву и инвертируется. Чтобы получить правильный, а не инвертированный результат, этот оператор нужно использовать дважды:

```
let stringNumber = '12';
console.log(!true); // false -- оператор отрицания инвертировал значение нашей булевой переменной.

console.log(!!stringNumber); // true — получили логическое значение «правда», так как наша строка была не пустой. Обратите внимание, как используется оператор отрицания для приведения к булеву типу, тут он использован дважды: !!
```

Равенство

Есть четыре оператора проверки на равенство: `==` (нестрогое равенство), `===` (строгое равенство), `!=` (нестрогое неравенство) и `!==` (строгое неравенство). Снова мы видим оператор отрицания (восклицательный знак `!`), который в данном случае означает «не равно».

Обычно говорят, что оператор `===` проверяет равенство значений и типов, а оператор `==` проверяет только равенство значений. Правильнее сказать, что оператор `==` совершает приведение типов значений, а затем сравнивает их.

```
let a = "42";
let b = 42;

a == b; // true
a === b; // false
```

Разберёмся, что здесь происходит. Интерпретатор привёл сравнение к `42 == 42` или `"42" == "42"`? Ответ: `42 == 42`, типы были приведены **к числу**. Движок JavaScript при проверке на равенство всегда приводит данные к числу. Это очень важный момент, который позволит вам в дальнейшем распутывать такие хитрые примеры:

```
[] == ![] // true
Пустой массив при преобразовании в число даёт 0.
+[] // 0
Отрицания массива — это преобразование массива в булев тип, что даёт true, а
затем отрицание булева типа, что даёт false.
![] // false
И в итоге преобразование полученного булева значения false в число даёт нам 0.
А 0 == 0 - true.
```

Строгое сравнение проверяет значения переменных и не пытается при этом привести типы. Профессиональным стандартом считается строгое сравнение, так как его легче читать в коде и не нужно задумываться, к какому типу будет приведено то или иное значение при сравнении. Да и в абсолютном большинстве случаев нет необходимости сравнивать данные разных типов, но понимать нестрогое сравнение очень полезно. Иногда всё же приходится к нему прибегать.

Механизм описан в спецификации, и, если однажды потратить время на этот раздел, можно удивиться, насколько прямолинейный механизм использует нестрогое сравнение. И вот ещё маленькая статья, как приводятся типы при нестрогом равенстве.

Для неравенств `!=` и `!==` работают те же правила.

Сравнение объектов и массивов подчиняется особым правилам. Они сравниваются по ссылке, и операторы `==` и `===` просто проверяют, совпадают ли ссылки, никак не сравнивая значения объектов. Однако если мы сравниваем объект с примитивом, включается механизм приведения типов, JavaScript пытается привести объект к примитиву и проверяет его равенство со вторым значением. Как осуществляется приведение к примитиву — это отдельная тема.

Сравнения

Рассмотрим операторы сравнения `<`, `>`, `<=` и `>=`. Обычно их используют для сравнения чисел или строк. Например, для перебора значений и сортировки их по алфавиту или для реализации условной логики.

Правила приведения типов похожи на правила нестрогого сравнения `==`.

```
let a = 41;
let b = "42";
let c = "43";

a < b; // true
b < c; // true
```

Поясним, что тут происходит: когда оба значения при сравнении являются строками (b < c) то они сравниваются посимвольно, как строки, и получается, что строка `"42"` меньше `"43"`. Сначала сравниваются первые символы `"4"` и `"4"`, они равны, потом сравниваются вторые символы `"2"` и `"3"`: символ `"2"` находится в таблице символов раньше символа `"3"` поэтому он считается меньше и вся строка считается меньше. Такое сравнение называют лексикографическим.

Когда одно из значений — число, JavaScript приводит оба значения к числовому типу, поэтому a < b сравниваются как числа, и 41 меньше, чем 42.

В JavaScript нет формы строгого сравнения, которая есть для равенства, поэтому при сравнениях мы всегда сталкиваемся с приведением типов, и надо быть с этим осторожнее. Лучше не сравнивать разные типы данных. Вот пример, который кажется странным, но работает по чёткой логике приведения типов:

```
let a = 43;
let b = "foo";

a < b; // false
b > c; // false
```

```
a == b; // false
```

Так происходит потому, что значение переменной b не может быть преобразовано в число. Мы получаем тут `NaN`, а тип `NaN` по спецификации JavaScript не может быть больше или меньше любого значения. Поэтому при сравнениях и при проверке равенства мы получаем false.

Подробно работа операторов сравнения описана в стандарте.

Унарные и бинарные операторы

Как и у любого языка программирования, в JavaScript также есть операторы. Сам по себе оператор – это наименьшая автономная часть языка программирования, т.е. команда. У операторов есть операнды. Операнд (или аргумент оператора) – это сущность, к которой применяется оператор. К примеру, при сложении двух чисел (3 + 2) работает оператор сложения с двумя операндами.

Операторы бывают унарными и бинарными. Унарный оператор применяется к одному операнду. Например

```
let a = 1;

a = -a; // унарный минус
```

Бинарный же оператор применяется к двум операндам:

```
let a = 1;
let b = 2;

a + b; // бинарный плюс
```

У некоторых операторов есть свои особые названия.

- Инкремент – увеличение операнда на установленный фиксированный шаг (как правило – единицу). Он же a++ или a+1.
- Декремент – обратная инкременту операция. a-- или a-1.
- Конкатенация – сложение строк. Обратной операции нет.

Частые примеры приведения типов и проверок

Очень часто в коде делаются проверки, приводящие к неявному приведению типов, просто потому, что так проще и быстрее писать код. Чаще всего происходит проверка, есть ли значение у переменной или не пустой ли массив, которые неявно приводятся к булеву значению. Посмотрим на примерах.

Проверка, есть ли значение

Мы можем в условии просто проверить нашу переменную. Её значение будет преобразовано в числовой тип, а затем в булев. Будьте осторожны: если переменная может быть числом и в ней есть значение 0, это будет преобразовано в `false`. Проверка, возможно, пройдёт не так, как вы ожидаете. Лучше всего такая проверка подходит для переменных, тип которых вы знаете. Там не может быть пустой строки “” или 0, а, скорее всего, должен содержаться объект или массив.

```
let a;
if (a) {
  console.log(a);
}
```

Приведение строки к числу

Часто бывает, что вы получаете данные из другого источника, например из поля ввода, и хотите сложить полученное значение с каким-то числом. Но вот незадача: порой к нам приходит не число, а строка, и тогда сложение может превратиться в конкатенацию строк. Чтобы явно привести строку в число, можно воспользоваться оператором +, который, будучи поставленным перед переменной, приводит её значение в числовой тип.

```
let a = "42";
const result = +a + 3; // 45
```

Также для преобразования строки в число можно воспользоваться конструктором типа `Number` или встроенной функцией `parseInt`.

Приведение числа в строку

Для приведения числа к строке часто вообще ничего не приходится делать, так как при сложении с другой строкой число автоматически будет приведено к строке. При необходимости

явно привести число к строке можно воспользоваться конструктором типа строки `String()` или сложить наше число с пустой строкой:

```
let a = 42;
const result = a + ''; // "42" - сложение с пустой строкой приводит число в строку.
```

Приведение к булеву типу

Многие значения автоматически приводятся к булеву значению, и часто с ними ничего не нужно делать. Например, проверка, пустой ли у нас массив, делается проверкой свойства **`length`**. Когда массив пустой, в этом свойстве мы получаем значение 0, которое преобразуется при проверке условия в `false`:

```
let a = [];
if (a.length) {
  console.log(a) // не отработает, так как a.length равно 0, и это falsy значение.
}
```

Все операторы в наглядной таблице

Оператор	Описание
. [ ] ( )	Доступ к полям, индексация массивов, вызовы функций и группировка выражений.
++ -- - ~ ! delete new typeof void	Унарные операторы, тип возвращаемых данных, создание объектов, неопределённые значения.
* / %	Умножение, деление, деление по модулю..
+ - +	Сложение, вычитание, объединение строк.
<< >> >>>	Сдвиг бит.
< <= > >= instanceof	Меньше, меньше или равно, больше, больше или равно, instanceof.
== != === !==	Равенство, неравенство, строгое равенство, строгое неравенство.
&	Побитовое И.
^	Побитовое исключающее ИЛИ.

|	Побитовое ИЛИ.
&&	Логическое И.
||	Логическое ИЛИ.
?:	Условный оператор.
= *OP*=	Присваивание, присваивание с операцией (например += и &=).
,	Вычисление нескольких выражений.

Методы alert, prompt, confirm

Как только мы научились создавать переменные возникает вопрос, а как мы можем увидеть значение данной переменной или может как-то с ней взаимодействовать, тут к нам на помощь приходят **alert**, **prompt**, **confirm** эти методы позволяют в браузере увидеть значение, присвоить или отреагировать, но обо всё по порядку

1. Первое что нужно помнить про эти методы
2. Расположение определяется браузером
3. Внешний вид этих окон мы не можем поменять

alert

```
alert(message);
```

Всплывающая подсказка, внешний вид зависит от вашего браузера, достаточно старый способ отображения ошибок или подсказок на странице.

Этот диалог следует использовать для сообщений, которые не требуют никакого ответа от пользователя, кроме подтверждения самого сообщения.

Окна сообщений - модальные, они препятствуют получению пользователем доступа к другим частям страницы до тех пор, пока окно не будет закрыто. По этой причине, вам не следует злоупотреблять этой функцией.

Аргумент является опциональным и необязательным согласно спецификации.

prompt()

Синтаксис: prompt(message, default);

message — это строка текста, которая показывается пользователю. Этот параметр является необязательным и может быть пропущен если в окне prompt ничего не показывать.

default — это строка, содержащая значение по умолчанию, отображаемое в поле ввода текста. Это необязательный параметр. Обратите внимание, что в Internet Explorer 7 и 8, если вы не укажете этот параметр, строка "underfined" будет значением по умолчанию.

confirm()

Синтаксис: confirm(message);

message опциональная (необязательная) строка, которая будет отображена в диалоговом окне.

Работа с переменными и методами

Мы уже умеем создавать переменные. С помощью метода prompt() мы можем запоминать значения переменных, но есть закономерный вопрос, как правильно выводить значения этих переменных на экран или в консоль?

Шаблонные литералы

Звучит сложно, но на практике это популярные и достаточно простые в использовании “косые кавычки”. Контент, который вы хотите отобразить внутри метода или в консоли вы можете поместить в косые кавычки.

Пример:

```
alert(`Любой текст внутри косых кавычек`);
```

В чем же преимущество данного метода и почему он является таким популярным?

Ответ на этот вопрос прост: внутри этих кавычек могут быть заключены переменные.

```
let personName = prompt('Как вас зовут?');

alert(`Добро пожаловать на сайт ${personName}`);
```

что мы можем увидеть на примере данного кода, что когда мы создаём переменную и вводим текст внутри prompt() мы используем простые одинарные кавычки, но уже внутри метода alert() мы можем заметить использование косых кавычек, с помощью которых мы внутри может разместить конструкцию ${} которая позволяет нам добавлять переменную и использовать ее для вывода

Возникает вопрос, а как же вывести переменную, если нам нужно использовать одинарные кавычки?

```
alert('Добро пожаловать на сайт' + personName);
```

Мы можем увидеть конкатенацию, для пользователя результат будет одинаковым, но чем больше переменных мы используем, тем сложнее будет выглядеть код, поэтому первый способ приоритетнее.

Принципы ветвления, визуализация, блок-схемы

Давайте вспомним одну из самых популярных частей программирования, конечно же **ветвления**

В программном коде, как и в жизни, множество решений зависят от внешних факторов. И зависимость эта выражается в вербальном виде «Если случится событие А, то я выполню действие Б». Именно по такому принципу начинает строиться ветвление во всех языках программирования.

Как в русском языке для ветвления используется слово «если», в программировании применяются специальные операторы, обеспечивающие выполнение определённой команды или набора команд только при условии истинности логического выражения или группы выражений. Ветвление — одна из трёх (наряду с последовательным выполнением команд и циклом) базовых конструкций структурного программирования.

Операторы if, if-else

Для реализации ветвления в JS используется оператор if

```
if (Условие) {
```

```
    // действие
}
```

Условие - это любое выражение, возвращающее булевское значение (true, false), т.е. такой вопрос, на который ответить можно только двумя способами: либо да, либо нет. Если выражение возвращает значение, отличное от типа boolean, то возвращаемое значение будет автоматически приведено к типу boolean: 0, null undefined, “” и NaN будут транслировать в false, остальные значения - в true. Действие выполняется тогда, когда условие истинно (true). Обычно условием является операция сравнения, либо несколько таких операций, объединённых логическими связками (И, ИЛИ).

Но что, если одного условия недостаточно? Рассмотрим пример ветвления, когда в случае истины мы выполним одно действие, а иначе – другое.

```
if (Условие) {

    // Действие1;

} else {

    // Действие2;

}
```

Давайте попробуем реализовать простой пример:

```
let x = 5;

let y = 42;

if (x > y) {

    alert(x + y); // сложить значения переменных если условие верно

} else {

     alert(x * y); // умножить значения переменных если условие
ложно

}
```

JS позволяет разделять нашу программу на сколько угодно вариантов с помощью конструкции else if, которая позволяет анализировать дополнительное условие. При этом выполняться будет первое условие, вернувшее true.

Тернарный оператор

Тернарный оператор – это операция, возвращающая либо второй, либо третий операнд в зависимости от условия (первого операнда). Звучит страшно, однако выглядит он достаточно просто:

```
(Условие) ? (Оператор по истине) : (Оператор по лжи);
```

Например, мы хотим сохранить максимальное из двух произвольных чисел в какую-то переменную. В этом случае, вместо громоздких строк ветвления, можно написать:

```
let x = 10;
let y = 15;
let max = (x > y) ? x : y;
alert(max);
```

Тернарный оператор – красивая возможность, делающая код лаконичнее. Но, как и любым инструментом, не стоит злоупотреблять данной возможностью, наоборот усложняя код.

По своей сути тернарный оператор отличается от оператора if. Во-первых, недопустимо множественное использование тернарного оператора, как в случае if - else if. Это засоряет код. Во-вторых, тернарный оператор нужен для встраивания небольших условных веток прямо в выражение, т.е. он не заменяет собой стандартный if-else. В случае, если Вам необходимо описать условия непосредственно в выражении, то следует использовать тернарный оператор. Но если Вы хотите создать более сложное условие с телом, состоящим из более, чем одной инструкции, то Вы используете if и else.

Консоль в chrome

Начало работы с консоль браузера chrome

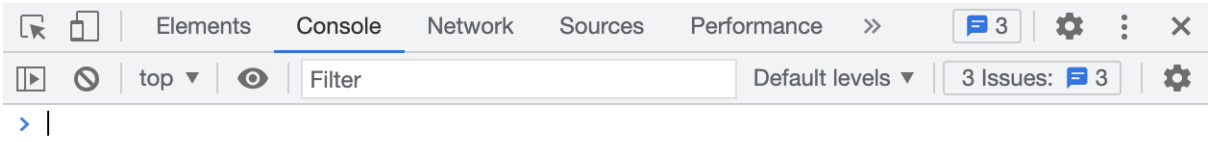
Чтобы открыть консоль в Chrome, используйте сочетание клавиш:

— на Windows `Ctrl` + `Shift` + `J`

— на macOS `Option` + `Command` + `J`

Как вы уже знаете это тот же отладчик браузера, что вы использовали на курсе по html/css

Какие основные моменты нужно отметить, javascript это язык программирования и те ошибки которые мы допускаем должны отображаться, именно для этой задачи нам потребуется вкладка console.

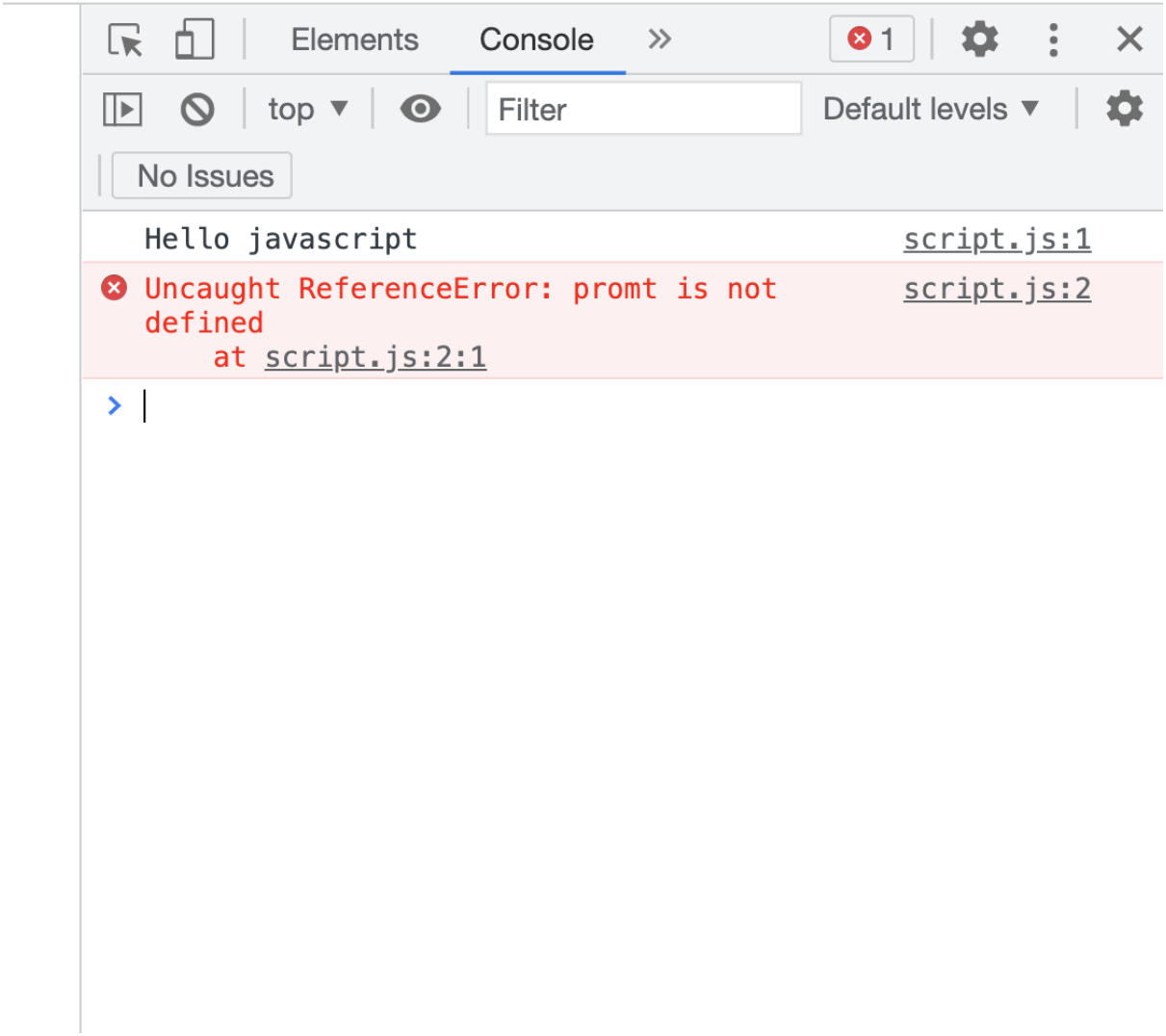


Работа в с консолью браузера

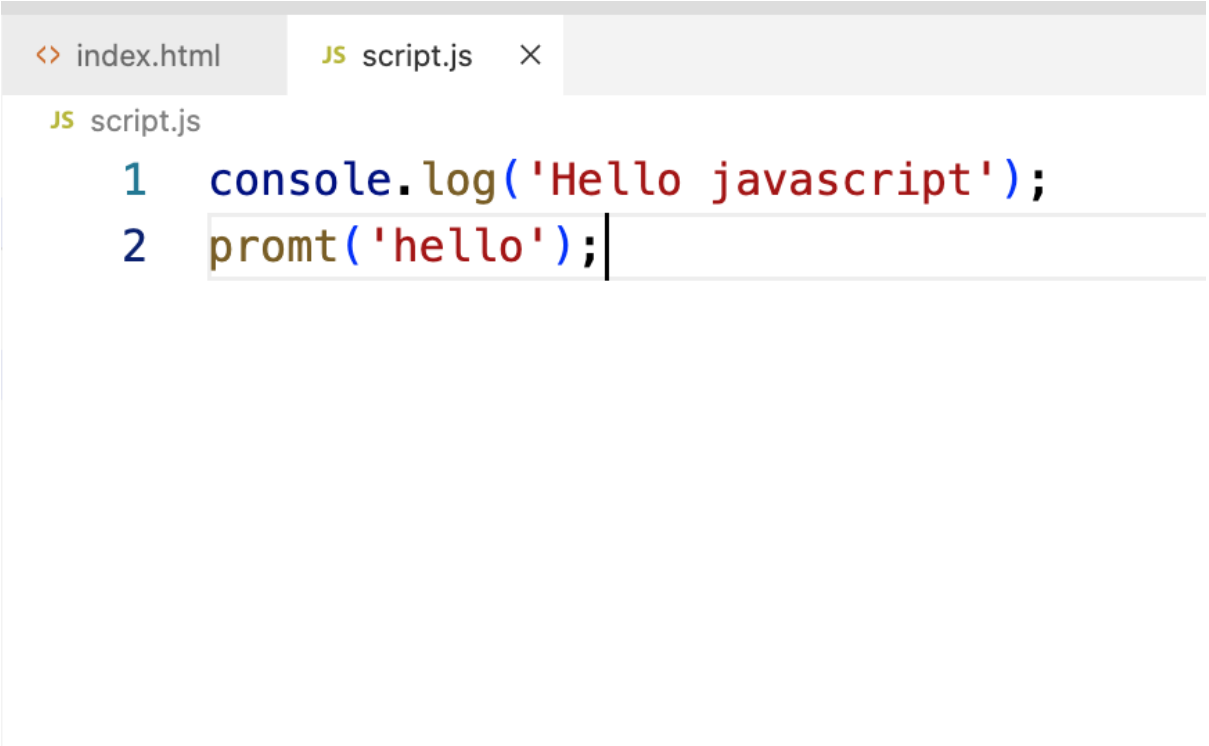
В консоли вы можете:

- Увидеть журнал сообщений, где выводятся сообщения и ошибки из кода.
- Запускать JS-код.

Например, так выглядит журнал сообщений:



На первой строчке мы видимо сообщение и код отработал корректно, справа подписывается script.js:1 это название файла и номер строки 1, а дальше подсказывает ошибку и в какой строчке

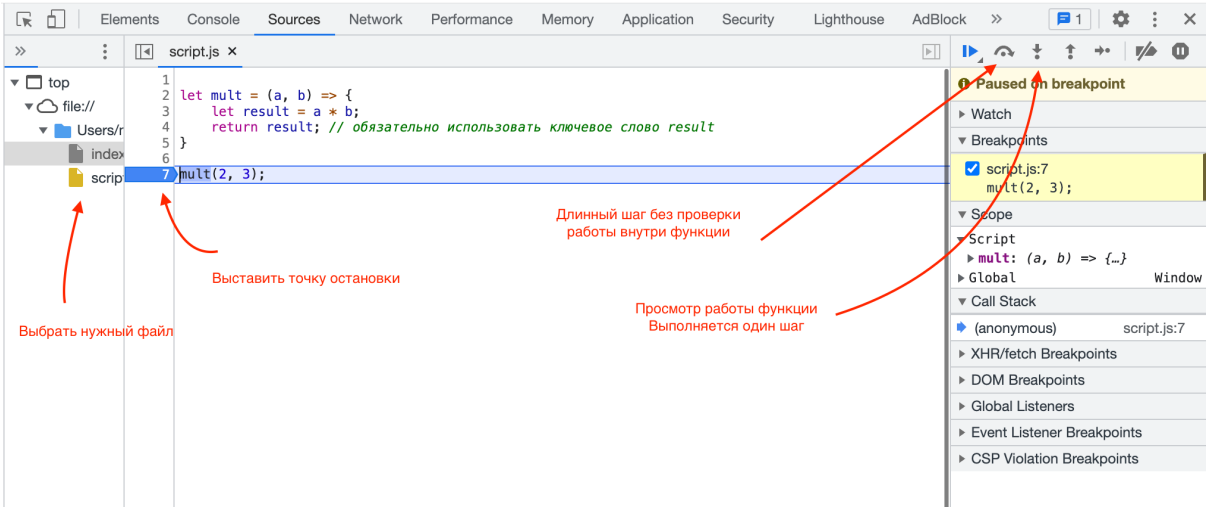


так что если мы заглянем в редактор кода, то сразу будет понятно где искать ошибку. Чаще всего это опечатка в синтасисе, так что слово promt написано с ошибкой, необходимо исправить и сделать prompt();

Так будет намного проще проверять работу кода и разобраться в какой строке у вас ошибка

Пошаговая отладка кода

Как же быть в ситуации, когда программа выводит неверный результат, но никаких ошибок нет, вы не можете найти ошибку в коде. Для этого есть возможность работать по шагам.



1. В данном примере мы видим, что выбрана вкладка Sources
2. Выбираем необходимый файл
3. Выставляем точку остановки
4. Обновляем страницу в браузере
5. Двигаемся по шагам
6. При каждом шаге программа показывает какие значения принимает
7. Так же вы всегда можете выделить переменную или операцию в коде, чтобы узнать, чему она равняется на данный момент времени.

Введение в браузерные события

Давайте для начала разберёмся что такое браузерные события, мы уже знаем что html это контент сайта, а css это стили для данного контента, но когда мы заходим на сайт мы очень часто выполняем какие-то действия на странице, например, открываем или закрываем окна, вводим значения в поле ввода, нажимаем на любые элементы именно эти события и называются браузерными событиями.

На последующих уроках мы последовательно будем разбираться с браузерными событиями и узнавать про них последовательно, не нужно делать вывод что мы сейчас изучаем всё и этого мало.

Первое с чего мы начнем это Использование атрибута HTML

Обработчик может быть назначен прямо в разметке, в атрибуте, который называется **on<событие>**.

Например, чтобы назначить обработчик события click на элементе button, можно использовать атрибут onclick, вот так:

```
<button onclick="alert('Вы нажали на кнопку купить')">Купить</button>
```

При клике мышкой на кнопке выполнится код, указанный в атрибуте onclick.

Обратите внимание, для содержимого атрибута onclick используются одинарные кавычки, так как сам атрибут находится в двойных. Если мы забудем об этом и поставим двойные кавычки внутри атрибута, код не будет работать.

Атрибут HTML-тега – не самое удобное место для написания большого количества кода, поэтому лучше создать отдельную JavaScript-функцию и вызвать её там.

Что такое функции и как с ними работать, мы узнаем на следующем уроке

Дополнительные материалы

1. Разбираемся с поднятием (hoisting) в JavaScript — статья на medium.com.
2. Вы не знаете JS: типы данных и значения — статья на medium.com.
3. Преобразования типов.
4. Строка.
5. Число.
6. Что за чёрт, Javascript — статья на Хабре, раскрывающая множество так называемых странностей JavaScript. Может быть полезно прочитать перед собеседованием.

Используемые источники

1. Вы не знаете JS: типы данных и значения — статья на medium.com.
2. Документация MDN.