

Лабораторная работа №1. Основы JavaScript

Введение в JavaScript. Отладка скриптов. Синтаксис JS. Автоматическое тестирование. Строгий режим — "use strict". Переменные. Типы данных. Преобразование типов. Операторы. Операторы сравнения. Взаимодействие с пользователем: alert, prompt, confirm. Условные операторы: if, '?'. Логические операторы. Циклы while, for. Конструкция switch.

Задание: изучите теорию и решите задачи. Форматирование кода, синтаксис, именование переменных и функций должны соответствовать рекомендациям, приведенным ниже.

Задачи:

1. Выполните форматирование кода, согласно рекомендациям (п.2).

```
1  function pow(x,n)
2  {
3      let result=1;
4      for(var i=0;i<n;i++) {result*=x;}
5      return result;
6  }
7
8  x=prompt("x?","")
9  n=prompt("n?","")
10 if (n<0)
11 {
12     alert(`Степень ${n} не поддерживается, введите целую степень, большую 0`);
13 }
14 else
15 {
16     alert(pow(x,n))
17 }
```

2. Объявите переменные и задайте им следующие значения: имя пользователя, название города, год рождения, красный цвет, ответ пользователя (да/нет), бесконечность, 532, периметр четырехугольника 120 мм, сообщение пользователю «Введенные данные неверны».
3. Определите тип переменных:

```
1  let a = 5;
2  let name = "Name";
3  let i = 0;
4  let double = 0.23;
5  let result = 1/0;
6  let answer = true;
7  let no = null;
```

4. Вычислите площадь четырехугольника А, если его стороны равны 45 мм и 21 мм.
5. Сколько квадратов В со сторонами 5 мм поместятся в четырехугольник А (45мм x 21мм).
6. Что больше а или b?

```
1  let i = 2;
2  let a = ++i;
3  let b = i++;
```

7. Что больше «Привет» или «привет»? «Привет» или «Пока»? 45 или «53»? false или 3? true или «3»? 3 или «5мм»? null или undefined?
8. Выведите сообщение о том, что введенные пользователем данные неверные.

9. Проверьте ответ пользователя на секретный вопрос. Ответ вводит пользователь в окно prompt.
10. Пользователь выполняет вход в личный кабинет (вводит логин и пароль в prompt). Выведите соответствующее сообщение об успешном/неуспешном входе в зависимости от правильности введенных данных.
11. У студента 3 экзамена: русский, математика, английский. Если он сдаст все экзамены, то его переведут на следующий курс. Если он не сдаст ни одного экзамена, то его отчислят. Если он не сдаст хотя бы один экзамен, то его ожидает пересдача.
Для решения задачи использовать логические операторы.
12. Пользователь вводит число *a* и число *b*. Вычислите сумму чисел.
13. Вычислите и поясните:

```
1  true + true
2  0 + "5"
3  5 + "мм"
4  8/Infinity
5  9 * "\n9"
6  null - 1
7  "5" - 2
8  "5px" - 3
9  true - 3
10 7 || 0
```

14. К каждому четному числу в диапазоне [1, 10] прибавьте 2, а каждое нечетное число преобразуйте к виду «Xмм», где *X* – нечетное число. Выведите результат.
15. Пользователь может вводить числа до тех пор, пока введенное число меньше 100.
16. По номеру дня недели определить какой это день: 1 – пн, 2 – вт, ... , 7 – вс. Номер дня вводит пользователь.

Теория:

1. Отладка скриптов

Все современные браузеры и многие среды разработки поддерживают средства отладки кода — специальный графический интерфейс для быстрого поиска и устранения ошибок. Он также позволяет по шагам отследить, что именно происходит в коде.


Рассмотрим браузер Chrome, так как у него достаточно возможностей, в большинстве других браузеров процесс будет схожим.

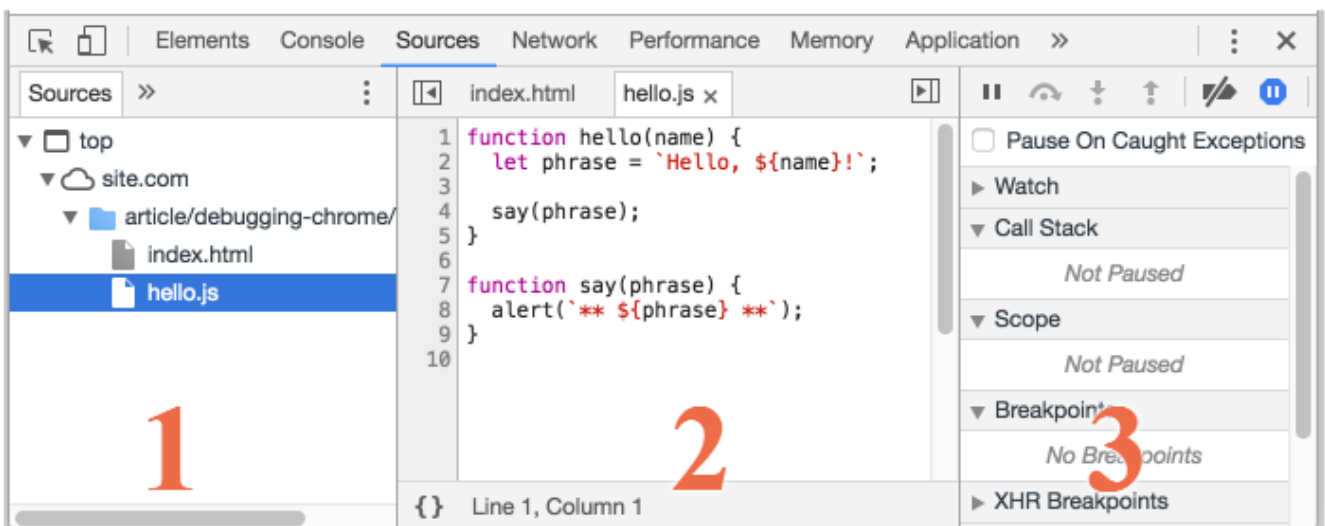
Панель «Исходный код» («Sources»)

- Откройте в Chrome свою страницу
- Включите инструменты разработчика, нажав F12 (Mac: Cmd+Opt+I).
- Щёлкните по панели sources («исходный код»).

При первом запуске увидите следующее:




Кнопка-переключатель  откроет вкладку со списком файлов. Кликните на неё и выберите свой файл *.js. Появится следующее:



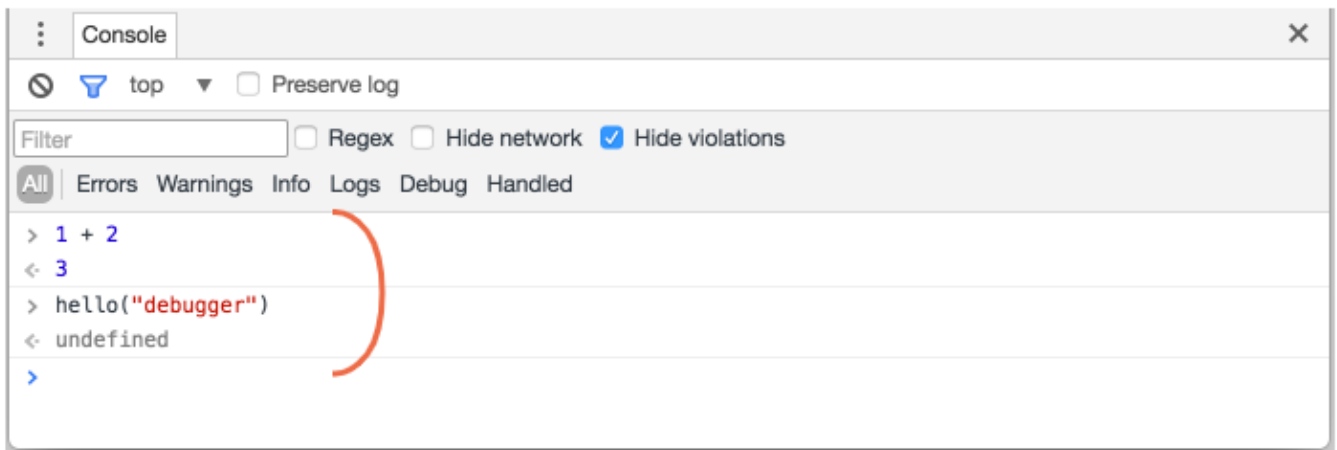
Интерфейс состоит из трёх зон:

1. В зоне Resources (Ресурсы) показаны файлы HTML, JavaScript, CSS, включая изображения, используемые на странице. Здесь также могут быть файлы различных расширений Chrome.
2. Зона Source показывает исходный код.
3. Зона Information and control (Сведения и контроль) отведена для отладки.

Чтобы скрыть список ресурсов и освободить экранное место для исходного кода, щёлкните по тому же переключателю .

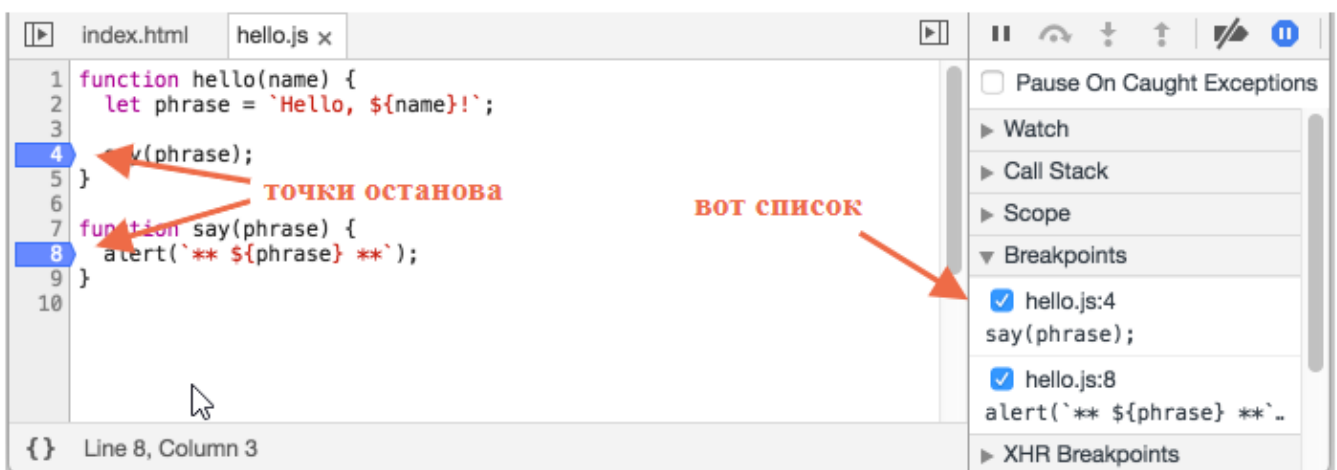
Консоль

При нажатии на клавишу Esc в нижней части экрана вызывается консоль, где можно вводить команды и выполнять их клавишей Enter. Результат выполнения инструкций сразу же отображается в консоли. Например, результатом `1+2` будет `3`, а инструкция `hello("debugger")` ничего не возвращает, так что получаем `undefined`:



Точки останова (breakpoints)

В вашем файле *.js щёлкните по строке (по самой цифре, не по коду), например, номер 4. Вы поставили точку останова. А теперь щёлкните по цифре 8 на восьмой линии. Номер строки будет окрашен в синий цвет. Вот что в итоге должно получиться:



Точка останова – это участок кода, где отладчик автоматически приостановит исполнение JavaScript.

Пока исполнение поставлено «на паузу», можно просмотреть текущие значения переменных, выполнить команды в консоли, т.е. выполнить отладку кода.

В правой части графического интерфейса – список точек останова. Когда таких точек выставлено много, да ещё и в разных файлах, этот список поможет эффективно ими управлять:

- Быстро переместиться к любой точке останова в коде – нужно щёлкнуть по точке в правой части экрана.
- Временно деактивировать точку – в общем списке снимите галочку напротив ненужной в данный момент точки.
- Удалить точку – щёлкните по ней правой кнопкой мыши и выберите Remove (Удалить).
- ...и так далее.

Условные точки останова

Можно задать и так называемую условную точку останова – щёлкните правой кнопкой мыши по номеру строки в коде. Если задать выражение, то именно при его истинности выполнение кода будет приостановлено.

Этот метод используется, когда выполнение кода нужно остановить при присвоении определённого выражения какой-либо переменной или при определённых параметрах функции.

Команда Debugger

Выполнение кода можно также приостановить с помощью команды debugger прямо изнутри самого кода:

```
function hello(name) {
  let phrase = `Привет, ${name}!`;

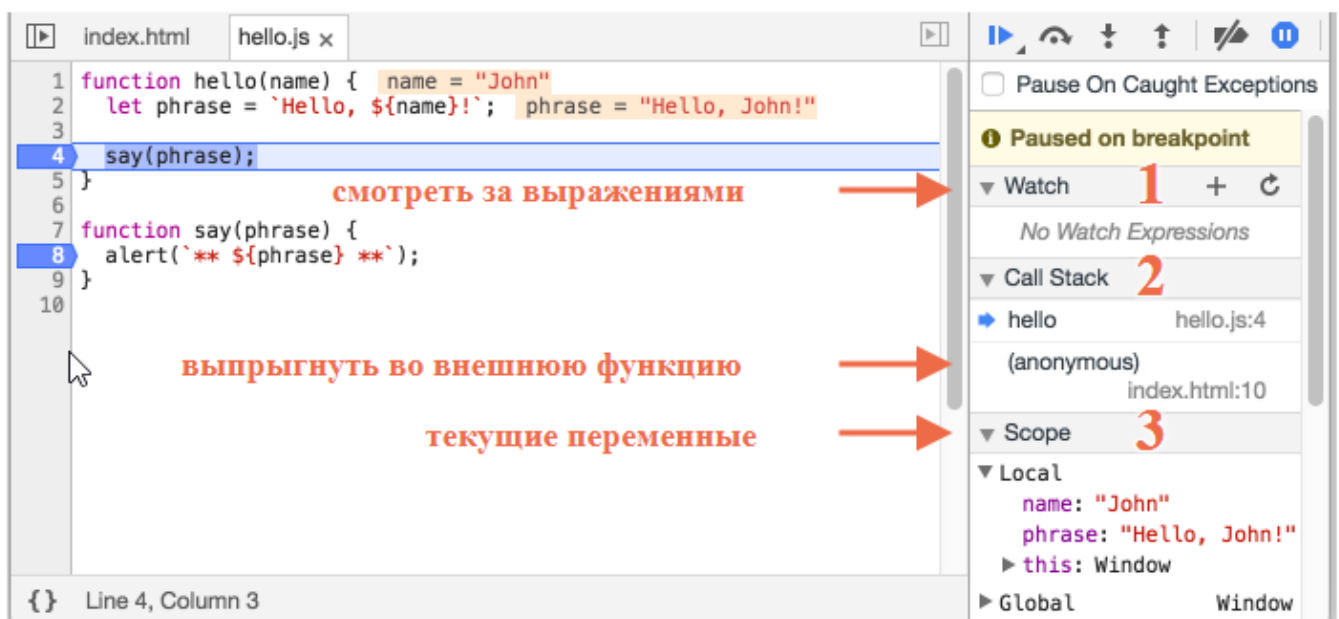
  debugger; // <-- здесь выполнение прерывается

  say(phrase);
}
```

Способ удобен тем, что можно продолжить работать в редакторе кода без необходимости переключения в браузер для выставления точки останова.

Текущее значение

После того, как вы поставили точки останова перезагрузите страницу (F5 (Windows, Linux) или Cmd+R (Mac)). Выполнение прервётся на четвёртой строчке:



Чтобы понять, что происходит в коде, щёлкните по стрелочкам справа:

1. Watch показывает текущие значения выражений.

Нажмите на + и введите выражение. В процессе выполнения отладчик автоматически пересчитывает и выводит его значение.

2. Call Stack показывает последовательность вызовов функций.

При нажатии на элемент списка (например, на «anonymous») отладчик переходит к соответствующему коду, и нам представляется возможность его проанализировать.


3. Scope показывает текущие переменные.

В Local отображаются локальные переменные функций, а их значения подсвечены в исходном коде. В Global перечисляются глобальные переменные (т.е. объявленные за пределами функций).

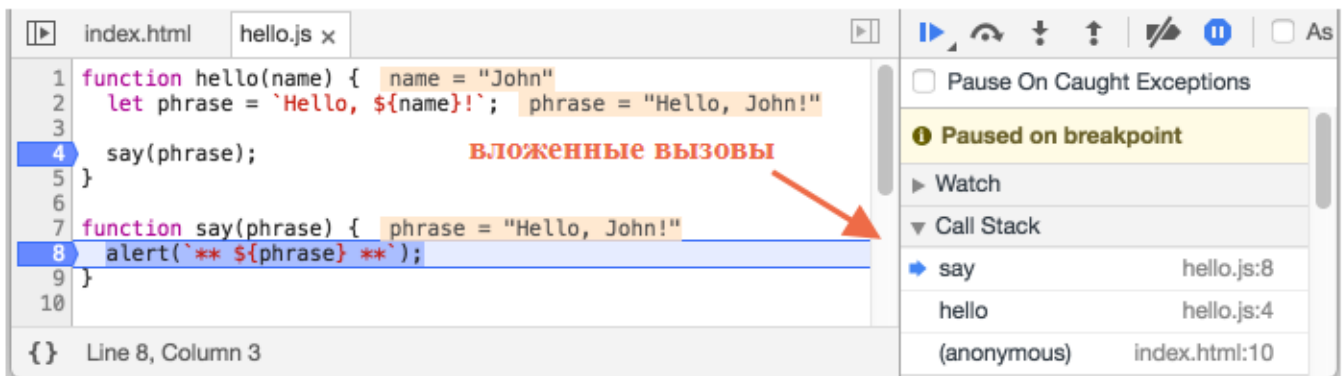
Пошаговое выполнение скрипта

Выполним отладку построчно.


В правой части панели для этого есть несколько кнопок:

-  *продолжить выполнение. Быстрая клавиша – F8.*

Возобновляет выполнение кода. Если больше нет точек останова, отладчик прекращает работу и позволяет приложению работать дальше.




Выполнение кода возобновилось, дошло до другой точки останова внутри say(), и отладчик снова приостановил выполнение. Обратите внимание на пункт «Call stack» справа: в списке появился ещё один вызов. Мы теперь внутри функции say().

-  *сделать шаг (выполнить следующую команду), не заходя в функцию. Быстрая клавиша – F10.*

Если мы нажмём на неё – будет вызван alert. Важно: на месте alert может быть любая другая функция, выполнение просто *перешагнёт* через неё, полностью игнорируя её содержимое.

-  *сделать шаг. Быстрая клавиша – F11.*

В отличие от предыдущего примера, здесь мы «заходим» во вложенные функции и шаг за шагом проходим по скрипту.

-  *продолжить выполнение до завершения текущей функции. Быстрая клавиша – Shift+F11.*

Выполнение кода остановится на самой последней строчке текущей функции. Этот метод применяется, когда мы случайно нажали и зашли в функцию, но нам она неинтересна и мы как можно скорее хотим из неё выбраться.

-  *активировать/деактивировать все точки останова.*

Эта кнопка не влияет на выполнение кода, она лишь позволяет массово включить/отключить точки останова.

- **II** *разрешить/запретить остановку выполнения в случае возникновения ошибки.*
Если опция включена и инструменты разработчика открыты, любая ошибка в скрипте приостанавливает выполнение кода, что позволяет его проанализировать. Поэтому если скрипт завершается с ошибкой, открываем отладчик, включаем эту опцию, перезагружаем страницу и локализуем проблему.

Continue to here

Если щёлкнуть правой кнопкой мыши по строчке кода, в контекстном меню можно выбрать опцию «Continue to here» («продолжить до этого места»).

Этот метод используется, когда нам нужно продвинуться на несколько шагов вперёд до нужной строки, но нет желания выставлять точки останова.

Логирование

Если нужно что-то вывести в консоль из кода, применяется функция `console.log`. К примеру, выведем в консоль значения от нуля до четырёх:

```
// чтобы увидеть результат, сначала откройте консоль
for (let i = 0; i < 5; i++) {
  console.log("значение", i);
}
```

Обычный пользователь сайта не увидит такой вывод, так как он в консоли. Консоль можно открыть через инструменты разработчика, выбрав вкладку «Консоль» или нажав Esc, находясь в другой вкладке – консоль откроется в нижней части интерфейса.

Если правильно выстроить логирование в приложении, то можно и без отладчика разобраться, что происходит в коде.

2. Синтаксис JavaScript

Код должен быть максимально читаемым и понятным. Для этого нужен *хороший стиль* написания кода.

Команды

Как правило, каждая команда пишется на отдельной строке – так код лучше читается:

```
alert('Привет');
alert('Мир');
```

Точка с запятой

Точку с запятой *во многих случаях* можно не ставить, если есть переход на новую строку. Так тоже будет работать:

```
alert('Привет')
alert('Мир')
```

В этом случае JavaScript интерпретирует переход на новую строку как разделитель команд и автоматически вставляет «виртуальную» точку с запятой между ними.

Однако, важно то, что «во многих случаях» не означает «всегда»!

Например, этот код:

```
alert(3 +
```



```
1  
+ 2);
```

Выведет 6. То есть, точка с запятой не ставится. Почему? Интуитивно понятно, что здесь дело в «незавершённом выражении», конца которого JavaScript ждёт с первой строки и поэтому не ставит точку с запятой. И здесь это, пожалуй, хорошо и приятно.

Но в некоторых важных ситуациях JavaScript «забывает» вставить точку с запятой там, где она нужна. Таких ситуаций не так много, но ошибки, которые при этом появляются, достаточно сложно обнаруживать и исправлять.

Например, такой код работает:

```
[1, 2].forEach(alert)
```

Он выводит по очереди 1, 2.

А вот такой код уже работать не будет:

```
alert("Сейчас будет ошибка")  
[1, 2].forEach(alert)
```

Выведется только первый alert, а дальше – ошибка. Потому что перед квадратной скобкой JavaScript точку с запятой не ставит, а как раз здесь она нужна.

Если её поставить, то всё будет в порядке:

```
alert("Сейчас будет ошибка");  
[1, 2].forEach(alert)
```

Поэтому в JavaScript рекомендуется точки с запятой ставить. Сейчас это, фактически, стандарт, которому следуют все большие проекты.

Комментарии

Со временем программа становится большой и сложной. Появляется необходимость добавить *комментарии*, которые объясняют, что происходит и почему.

Должен быть минимум комментариев, которые отвечают на вопрос "что происходит в коде?". Что интересно, в коде начинающих разработчиков обычно комментариев либо нет, либо они как раз такого типа: «что делается в этих строках». Хороший код и так понятен. Если вам кажется, что нужно добавить комментарий для улучшения понимания, это значит, что ваш код недостаточно прост, и, может, стоит переписать его

А какие комментарии полезны и приветствуются?

- *Архитектурный комментарий* – «как оно, вообще, устроено».

Какие компоненты есть, какие технологии использованы, поток взаимодействия. О чём и зачем этот скрипт. Эти комментарии особенно нужны, если вы не один, а проект большой.

- *Справочный комментарий перед функцией* – о том, что именно она делает, какие параметры принимает и что возвращает.

Для таких комментариев существует синтаксис JSDoc.


```

1  /**
2   * Возвращает x в степени n, только для натуральных n
3   *
4   * @param {number} x Число для возведения в степень.
5   * @param {number} n Показатель степени, натуральное число.
6   * @return {number} x в степени n.
7   */
8  function pow(x, n) {
9      ...
10 }

```

Такие комментарии позволяют сразу понять, что принимает и что делает функция, не вникая в код.

Кстати, они автоматически обрабатываются многими редакторами, например Aptana и редакторами от JetBrains, которые учитывают их при автодополнении, а также выводят их в автоподсказках при наборе кода. Кроме того, есть инструменты, например JSDoc 3, которые умеют генерировать по таким комментариям документацию в формате HTML.

Более важными могут быть комментарии, которые объясняют не *что*, а *почему* в коде происходит именно это! Как правило, из кода можно понять, что он делает. *Почему* это сделано именно так? На это сам код ответа не даёт.

Например, есть несколько способов решения задачи. Вы выбрали именно это решение. Вы пробовали решить задачу по-другому, но не получилось – напишите об этом. Особенно это важно в тех случаях, когда используется не первый приходящий в голову способ, а какой-то другой. Вы открываете код, который был написан какое-то время назад, и видите, что он «неоптимален», захотите его переписать. Только этот вариант вы уже обдумали раньше. И отказались, а почему – забыли. Комментарии, которые объясняют выбор решения, очень важны. Они помогают понять происходящее и предпринять правильные шаги при развитии кода.

Один из показателей хорошего разработчика – качество комментариев, которые позволяют эффективно поддерживать код, возвращаться к нему после любой паузы и легко вносить изменения.

Комментарии могут находиться в любом месте программы и никак не влияют на её выполнение. Интерпретатор JavaScript попросту игнорирует их.

Однострочные комментарии начинаются с двойного слэша //. Текст считается комментарием до конца строки:

```

// Команда ниже говорит "Привет"
alert( 'Привет' );

alert( 'Мир' ); // Второе сообщение выводим отдельно

```

Многострочные комментарии начинаются слешем-звездочкой «/*» и заканчиваются звездочкой-слешем «*/», вот так:

```

/* Пример с двумя сообщениями.
Это – многострочный комментарий.
*/
alert( 'Привет' );
alert( 'Мир' );

```

Всё содержимое комментария игнорируется. Если поместить код внутрь /* ... */ или после // – он не выполнится.

```
/* Закомментировали код
alert( 'Привет' );
*/
alert( 'Мир' );
```

В большинстве редакторов комментариев можно поставить горячей клавишей, обычно это Ctrl+/ для однострочных и что-то вроде Ctrl+Shift+/ – для многострочных комментариев (нужно выделить блок и нажать сочетание клавиш).

Вложенные комментарии не поддерживаются! В этом коде будет ошибка:

```
/*
  /* вложенный комментарий ?!? */
*/
alert( 'Мир' );
```

Не бойтесь комментариев. Чем больше кода в проекте – тем они важнее. Что же касается увеличения размера кода – это не страшно, т.к. существуют инструменты сжатия JavaScript, которые при публикации кода легко их удалят.

Функции в качестве комментариев

Иногда выгодно заменить часть кода функцией, например, в таком случае:

```
function showPrimes(n) {
  nextPrime:
  for (let i = 2; i < n; i++) {

    // проверяем, является ли i простым числом
    for (let j = 2; j < i; j++) {
      if (i % j == 0) continue nextPrime;
    }

    alert(i);
  }
}
```

Лучший вариант – использовать отдельную функцию isPrime:

```
function showPrimes(n) {

  for (let i = 2; i < n; i++) {
    if (!isPrime(i)) continue;

    alert(i);
  }
}

function isPrime(n) {
  for (let i = 2; i < n; i++) {
    if (n % i == 0) return false;
  }

  return true;
}
```

Теперь код легче понять. Функция сама становится комментарием. Такой код называется самодокументированным.

И если мы имеем такой длинный кусок кода:

```
// здесь мы добавляем воду
for(let i = 0; i < 10; i++) {
  let drop = getWater();
  smell(drop);
  add(drop, glass);
}

// здесь мы добавляем сок
for(let t = 0; t < 3; t++) {
  let tomato = getTomato();
  examine(tomato);
  let juice = press(tomato);
  add(juice, glass);
}

// ...
```

То будет лучше отрефакторить его с использованием функций:

```
addWater(glass);
addJuice(glass);

function addWater(container) {
  for(let i = 0; i < 10; i++) {
    let drop = getWater();
    //...
  }
}

function addJuice(container) {
  for(let t = 0; t < 3; t++) {
    let tomato = getTomato();
    //...
  }
}
```

Здесь комментарии тоже не нужны: функции сами говорят, что делают (если вы понимаете английский язык). И ещё, структура кода лучше, когда он разделён на части. Понятно, что делает каждая функция, что она принимает и что возвращает.

В реальности мы не можем полностью избежать «объясняющих» комментариев. Существуют сложные алгоритмы. И есть хитрые уловки для оптимизации. Но в целом мы должны стараться писать простой и самодокументированный код.

Хорошие комментарии

Описывайте архитектуру

Сделайте высокоуровневый обзор компонентов, того, как они взаимодействуют, каков поток управления в различных ситуациях, т.е. обзор кода «с высоты птичьего полёта». Существует специальный язык UML для создания диаграмм, разъясняющих архитектуру кода. Его стоит изучить.

Документируйте параметры и использование функций

Есть специальный синтаксис JSDoc для документирования функций: использование, параметры, возвращаемое значение.

Например:

```
/**
 * Возвращает x, возведённое в n-ную степень.
 *
 * @param {number} x Возводимое в степень число.
 * @param {number} n Степень, должна быть натуральным числом.
 * @return {number} x, возведённое в n-ную степень.
 */
function pow(x, n) {
    ...
}
```

Подобные комментарии позволяют понимать назначение функции и правильно её использовать без необходимости заглядывать в код. Многие редакторы, такие как WebStorm, распознают их для того, чтобы выполнить автодополнение ввода и различные автоматические проверки кода.

Также существуют инструменты, например, JSDoc 3, которые умеют генерировать HTML-документацию из комментариев.

Почему задача решена именно таким способом?

Важно то, что написано. Но то, что не написано, может быть даже более важным, чтобы понимать происходящее. Почему задача решена именно этим способом? Код не даёт ответа.

Если есть несколько способов решить задачу, то почему вы выбрали именно этот? Особенно если ваш способ – не самый очевидный.

Без подобных комментариев возможна следующая ситуация:

1. Вы (или ваш коллега) открываете написанный некоторое время назад код и видите, что в нём есть что улучшить.
2. Вы думаете: «Каким глупым я раньше был и насколько умнее стал сейчас», и переписываете его на «более правильный и оптимальный» вариант.
3. Желание переписать код – это хорошо. Но в процессе вы понимаете, что «оптимальное» решение на самом деле не такое уж и оптимальное. Вы даже смутно припоминаете почему, так как в прошлый раз вы уже его пробовали. Вы возвращаетесь к правильному варианту, потратив время зря.

Комментарии, объясняющие решение, очень важны. Они помогают продолжать разработку в правильном направлении.

В коде есть какие-то тонкости? Где они используются?

Если в коде есть какие-то тонкости и неочевидные вещи, его определённо нужно комментировать.

Фигурные скобки

Пишутся на той же строке, так называемый «египетский» стиль. Перед скобкой – пробел.

```

1  /* Так писать плохо! */
2  if (n < 0) {alert("Число отрицательное.");}
3
4  /* Так допустимо! */
5  if (n < 0) alert("Число отрицательное.");
6
7  /* Лучший вариант! */
8  if (n < 0) {
9      alert("Число отрицательное.");
10 }

```

В большинстве JavaScript-фреймворков стиль именно такой.

Длина строки

Максимальную длину строки согласовывают в команде. Как правило, это либо 80, либо 120 символов, в зависимости от того, какие мониторы у разработчиков.

Более длинные строки необходимо разбивать для улучшения читаемости.

Отступы

Отступы нужны двух типов:

Горизонтальный отступ, при вложенности – два(или четыре) пробела.

Как правило, используются именно пробелы, т.к. они позволяют сделать более гибкие «конфигурации отступов», чем символ «Tab».

Например, выровнять аргументы относительно открывающей скобки:

```

1  show("Строки" +
2      " выровнены" +
3      " строго" +
4      " одна под другой");
5

```

Вертикальный отступ, для лучшей разбивки кода – перевод строки.

Используется, чтобы разделить логические блоки внутри одной функции. В примере разделены инициализация переменных, главный цикл и возвращение результата:

```

1  function pow(x, n) {
2      let result = x;
3      //      <--
4      for (let i = 1; i < n; i++) {
5          result *= x;
6      }
7      //      <--
8      return result;
9  }

```

Вставляйте дополнительный перевод строки туда, где это сделает код более читаемым. Не должно быть более 9 строк кода подряд без вертикального отступа.

Именованье

Общее правило:

Имя переменной – существительное.

Имя функции – глагол или начинается с глагола. Бывает, что имена для краткости делают существительными, но глаголы понятнее.

Для имён используется английский язык (не транслит) и верблюжья нотация.

Уровни вложенности

Уровней вложенности должно быть немного.

Например, проверки в циклах можно делать через «continue», чтобы не было дополнительного уровня if(..) { ... }:

Вместо:

```
1  for (let i = 0; i < 10; i++) {
2      if (i подходит) {
3          ... // уровень вложенности 2
4      }
5  }
```

Используйте:

```
1  for (let i = 0; i < 10; i++) {
2      if (i не подходит) continue;
3      ... // уровень вложенности 1
4  }
```

Аналогичная ситуация – с if/else и return. Следующие две конструкции идентичны.

Первая:

```
1  function isEven(n) { // проверка чётности
2      if (n % 2 == 0) {
3          return true;
4      } else {
5          return false;
6      }
7  }
```

Вторая:

```
1  function isEven(n) { // проверка чётности
2      if (n % 2 == 0) {
3          return true;
4      }
5
6      return false;
7  }
```

Если в блоке if идёт return, то else за ним не нужен.

Лучше быстро обработать простые случаи, вернуть результат, а дальше разбираться со сложным, без дополнительного уровня вложенности.

Главное – не краткость кода, а его простота и читаемость. Совсем не всегда более короткий код проще для понимания, чем более развёрнутый.

Функции

Функции должны быть небольшими. Если функция большая – желательно разбить её на несколько.

Сравните, например, две функции showPrimes(n) для вывода простых чисел до n.

Первый вариант использует метку:

```
1  function showPrimes(n) {
2      nextPrime: for (let i = 2; i < n; i++) {
3
4          for (let j = 2; j < i; j++) {
5              if (i % j == 0) continue nextPrime;
6          }
7
8          alert( i ); // простое
9      }
10 }
```

Второй вариант – дополнительную функцию isPrime(n) для проверки на простоту:

```

2
3     for (let i = 2; i < n; i++) {
4         if (!isPrime(i)) continue;
5
6         alert(i); // простое
7     }
8 }
9
10 function isPrime(n) {
11     for (let i = 2; i < n; i++) {
12         if (n % i == 0) return false;
13     }
14     return true;
15 }

```

Второй вариант проще и понятнее. Вместо участка кода мы видим описание действия, которое там совершается (проверка isPrime).

Как правило, лучше располагать функции под кодом, который их использует.

Дело в том, что при чтении такого кода мы хотим знать в первую очередь, *что он делает*, а уже затем *какие функции ему помогают*. Если первым идёт код, то это как раз даёт необходимую информацию. Что же касается функций, то вполне возможно нам и не понадобится их читать, особенно если они названы адекватно и то, что они делают, понятно из названия.

```

1 //код, использующий функции
2 let elem = createElement();
3 setHandler(elem);
4 walkAround();
5
6 // --- функции ---
7
8 function createElement() {
9     ...
10 }
11
12 function setHandler(elem) {
13     ...
14 }
15
16 function walkAround() {
17     ...
18 }

```

Руководства по стилю

Когда написанием проекта занимается целая команда, то должен существовать один стандарт кода, описывающий где и когда ставить пробелы, запятые, переносы строк и т.п.

Сейчас, когда есть столько готовых проектов, нет смысла придумывать целиком своё руководство по стилю. Можно взять уже готовое, к которому, по желанию, всегда можно что-то добавить.

Большинство есть на английском:

Google JavaScript Style Guide

jQuery JavaScript Style Guide

Airbnb JavaScript Style Guide (есть перевод)

Idiomatic.JS (есть перевод)

Dojo Style Guide

Автоматизированные средства проверки

Существуют средства, проверяющие стиль кода. Самые известные – это:

JSLint – проверяет код на соответствие стилю JSLint, в онлайн-интерфейсе можно ввести код и различные настройки проверки, чтобы сделать её более мягкой.

JSHint – вариант JSLint с большим количеством настроек.

В частности, JSLint и JSHint интегрированы с большинством редакторов, они гибко настраиваются под нужный стиль и совершенно незаметно улучшают разработку, подсказывая, где и что поправить.

Полифилы

JavaScript – динамично развивающийся язык программирования. Регулярно появляются новые предложения, они анализируются и, если предложения одобряются, их переносят в черновик <https://tc39.github.io/esma262/>, а затем публикуют в спецификации.

Разработчики JavaScript-движков сами принимают решение, какие предложения реализовать в первую очередь. Они могут заранее реализовать функции, которые находятся в черновике и отложить разработку функций, которые уже перенесены в спецификацию, потому что они менее интересны разработчикам, а может их сложнее реализовать.

Таким образом, довольно часто реализуется только часть стандарта.

Можно проверить текущее состояние поддержки различных возможностей JavaScript на странице <https://kangax.github.io/compat-table/es6/>.

Babel

Когда мы используем современные возможности JavaScript, некоторые движки могут не поддерживать их. Как и было сказано выше, не везде реализованы все функции. И тут приходит на помощь Babel.

Babel – это транспILER. Он переписывает современный JavaScript-код в предыдущий стандарт. На самом деле, есть две части Babel:

Во-первых, транспILER, который переписывает код. Разработчик запускает Babel на своём компьютере. Он переписывает код в старый стандарт. И после, код отправляется на сайт. Современные сборщики проектов, такие как webpack или brunch, предоставляют возможность запускать транспILER автоматически, после каждого изменения кода, что позволяет экономить время.

Во-вторых, полифил. Новые возможности языка могут включать встроенные функции и синтаксические конструкции. ТранспILER переписывает код, преобразовывая синтаксические конструкции в старые. Но что касается новых встроенных функций, нам нужно их реализовать. JavaScript является высокодинамичным языком, скрипты могут добавлять/изменять любые функции, чтобы они вели себя в соответствии с современным стандартом.

Термин «полифил» означает, что скрипт «заполняет» пробелы и добавляет современные функции.

Два интересных полифила:

- core.js поддерживает много функций, можно подключать только нужные.
- polyfill.io – сервис, который автоматически создаёт скрипт с полифилом в зависимости от необходимых функций и браузера пользователя.

Таким образом, чтобы современные функции поддерживались в старых движках, нам надо установить транспILER и добавить полифил.

Google Chrome обычно поддерживает современные функции, можно запускать новейшие примеры без каких-либо транспILеров, но и другие современные браузеры также хорошо работают.

3. Автоматическое тестирование с использованием фреймворка Mocha

Обычно, когда мы пишем функцию, мы легко можем представить, что она должна делать и как она будет вести себя в зависимости от переданных параметров. Во время разработки мы можем проверить правильность работы функции, просто вызвав её, например из консоли, и сравнив полученный результат с ожидаемым. Если функция работает не так, как мы ожидаем, то можно внести исправления в код и запустить её ещё раз. Так можно повторять до тех пор, пока функция не станет работать так, как нам нужно. Однако, такие «ручные перезапуски» – не лучшее решение.

При тестировании кода ручными перезапусками легко упустить что-нибудь важное.

Например, мы работаем над функцией `f`. Написали часть кода и решили протестировать. Выясняется, что `f(1)` работает правильно, в то время как `f(2)` – нет. Мы вносим в код исправления, и теперь `f(2)` работает правильно. Вроде бы, всё хорошо. Однако, мы забыли заново протестировать `f(1)`. Возможно, после внесения правок `f(1)` стала работать неправильно.

Это типичная ситуация. Во время разработки мы учитываем множество различных сценариев использования. Но сложно ожидать, что программист станет вручную проверять каждый из них после любого изменения кода. Поэтому легко исправить что-то одно и при этом сломать что-то другое.

Автоматическое тестирование означает, что тесты пишутся отдельно, в дополнение к коду. Они по-разному запускают наши функции и сравнивают результат с ожидаемым.

Behavior Driven Development (BDD)

Техника под названием Behavior Driven Development или, коротко, BDD. BDD – это три в одном: и тесты, и документация, и примеры использования. Чтобы понять BDD – рассмотрим практический пример разработки.

Разработка функции возведения в степень — «pow»: спецификация

Допустим, мы хотим написать функцию `pow(x, n)`, которая возводит `x` в целочисленную степень `n`. Мы предполагаем, что `n ≥ 0`.

Эта задача взята в качестве примера. В JavaScript есть оператор `**`, который служит для возведения в степень. Мы сосредоточимся на процессе разработки, который также можно применять и для более сложных задач.

Перед тем, как начать писать код функции `pow`, мы можем представить себе, что она должна делать, и описать её. Такое описание называется спецификацией (specification) и содержит описания различных способов использования и тесты для них, например, вот такое:

```
describe("pow", function() {  
  
  it("возводит в степень n", function() {  
    assert.equal(pow(2, 3), 8);  
  });  
  
});
```

Спецификация состоит из трёх основных блоков:

```
describe("заголовок", function() { ... })
```

Какой функционал мы описываем. В нашем случае мы описываем функцию `pow`.

Используется для группировки рабочих лошадок – блоков `it`.

```
it("описание", function() { ... })
```

В первом аргументе блока `it` мы русским/английским языком описываем конкретный способ использования функции, а во втором – пишем функцию, которая тестирует данный случай.

`assert.equal(value1, value2)`

Код внутри блока `it`, если функция работает верно, должен выполняться без ошибок. Функции вида `assert.*` используются для проверки того, что функция `row` работает так, как мы ожидаем. В этом примере мы используем одну из них – `assert.equal`, которая сравнивает переданные значения и выбрасывает ошибку, если они не равны друг другу. Существуют и другие типы сравнений и проверок, которые мы добавим позже.

Спецификация может быть запущена, и при этом будет выполнена проверка, указанная в блоке `it`, мы увидим это позднее.

Процесс разработки

Процесс разработки обычно выглядит следующим образом:

1. Пишется начальная спецификация с тестами, проверяющими основную функциональность.

2. Создаётся начальная реализация.

3. Для запуска тестов мы используем фреймворк `Mocha`. Пока функция не готова, будут ошибки. Вносим изменения до тех пор, пока всё не начнёт работать так, как нам нужно.

4. Теперь у нас есть правильно работающая начальная реализация и тесты.

5. Мы добавляем новые способы использования в спецификацию, возможно, ещё не реализованные в тестируемом коде. Тесты начинают «падать» (выдавать ошибки).

6. Возвращаемся на шаг 3, дописываем реализацию до тех пор, пока тесты не начнут завершаться без ошибок.

7. Повторяем шаги 3-6, пока требуемый функционал не будет готов.

Таким образом, разработка проходит итеративно. Мы пишем спецификацию, реализуем её, проверяем, что тесты выполняются без ошибок, пишем ещё тесты, снова проверяем, что они проходят и т.д.

Давайте посмотрим этот поток разработки на нашем примере.

Первый шаг уже завершён. У нас есть спецификация для функции `row`. Теперь, перед тем как писать реализацию, давайте подключим библиотеки для пробного запуска тестов, просто чтобы убедиться, что тесты работают (разумеется, они завершатся ошибками).

Спецификация в действии

JavaScript-библиотеками для тестов:

- `Mocha` – основной фреймворк. Он предоставляет общие функции тестирования, такие как `describe` и `it`, а также функцию запуска тестов.

- `Chai` – библиотека, предоставляющая множество функций проверки утверждений.

- `Sinon` – библиотека, позволяющая наблюдать за функциями, эмулировать встроенные функции и многое другое.

Эти библиотеки подходят как для тестирования внутри браузера, так и на стороне сервера. Рассмотрим вариант с браузером.

Полная HTML-страница с этими библиотеками и спецификацией функции `row`:

```
<!DOCTYPE html>
```

```
<html>
```

```
<head>
```

```
  <!-- добавим стили mocha для отображения результатов -->
```

```

<link rel="stylesheet"
href="https://cdnjs.cloudflare.com/ajax/libs/mocha/3.2.0/mocha.css">
<!-- добавляем сам фреймворк mocha -->
<script
src="https://cdnjs.cloudflare.com/ajax/libs/mocha/3.2.0/mocha.js"></sc
ript>
<script>
    // включаем режим тестирования в стиле BDD
    mocha.setup('bdd');
</script>
<!-- добавим chai -->
<script
src="https://cdnjs.cloudflare.com/ajax/libs/chai/3.5.0/chai.js"></scri
pt>
<script>
    // chai предоставляет большое количество функций. Объявим assert
глобально
    let assert = chai.assert;
</script>
</head>

<body>

    <script>
        function pow(x, n) {
            /* Здесь будет реализация функции, пока пусто */
        }
    </script>

    <!-- скрипт со спецификацией (describe, it...) -->
    <script src="test.js"></script>

    <!-- элемент с id="mocha" будет содержать результаты тестов -->
    <div id="mocha"></div>

    <!-- запускаем тесты! -->
    <script>
        mocha.run();
    </script>
</body>

</html>

```

Условно страницу можно разделить на пять частей:

1. Тег <head> содержит сторонние библиотеки и стили для тестов.
2. Тег <script> содержит тестируемую функцию, в нашем случае – pow.
3. Тесты – в нашем случае внешний скрипт test.js, который содержит спецификацию describe("pow", ...), представленную выше.
4. HTML-элемент <div id="mocha"> будет использован фреймворком Mocha для вывода результатов тестирования.

Запуск тестов производится командой mocha.run().

Результаты:



Пока что тест завершается ошибкой. Это логично, потому что у нас пустая функция `pow`, так что `pow(2,3)` возвращает `undefined` вместо 8.

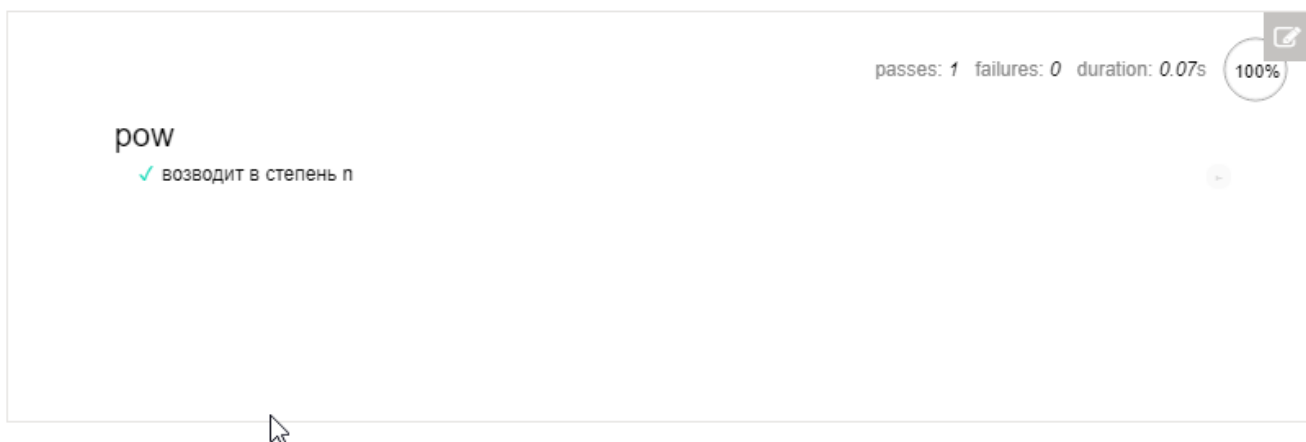
На будущее отметим, что существуют более высокоуровневые фреймворки для тестирования, такие как `karma` и другие. С их помощью легко сделать автозапуск множества тестов.

Начальная реализация

Напишем простую реализацию функции `pow`, чтобы пройти тесты.

```
function pow(x, n) {  
  return 8; // :) сжульничаем!  
}
```

Теперь всё работает:



Улучшаем спецификацию

Однако, это не так. Функция не работает. Попытка посчитать `pow(3,4)` даст некорректный результат, однако тесты проходят.

Такая ситуация вполне типична, она случается на практике. Тесты проходят, но функция работает неправильно. Наша спецификация не идеальна. Нужно дополнить её тестами. Добавим ещё один тест, чтобы посмотреть, что `pow(3, 4) = 81`.

У нас есть два пути организации тестов:

1. Первый – добавить ещё один `assert` в существующий `it`:

```
describe("pow", function() {  
  
  it("возводит число в степень n", function() {  
    assert.equal(pow(2, 3), 8);  
    assert.equal(pow(3, 4), 81);  
  });  
});
```

```
});
```

2. Второй – написать два теста:

```
describe("pow", function() {  
  
  it("2 в степени 3 будет 8", function() {  
    assert.equal(pow(2, 3), 8);  
  });  
  
  it("3 в степени 3 будет 27", function() {  
    assert.equal(pow(3, 3), 27);  
  });  
  
});
```

Принципиальная разница в том, что когда один из assert выбрасывает ошибку, то выполнение it блока тут же прекращается. Таким образом, если первый assert выбросит ошибку, результат работы второго assert мы уже не узнаем. Разделять тесты предпочтительнее, так как мы получаем больше информации о том, что конкретно пошло не так. Помимо этого есть одно хорошее правило, которому стоит следовать: **один тест проверяет одну вещь**.

Если вы посмотрите на тест и увидите в нём две независимые проверки, то такой тест лучше разделить на два более простых. Продолжим со вторым вариантом.

Результаты:



Как мы и ожидали, второй тест провалился. Естественно, наша функция всегда возвращает 8, в то время как assert ожидает 27.

Улучшаем реализацию

Давайте напишем что-то более похожее на функцию возведения в степень, чтобы заставить тесты проходить.

```
function pow(x, n) {  
  let result = 1;  
  
  for (let i = 0; i < n; i++) {  
    result *= x;  
  }  
  
  return result;  
}
```

Чтобы убедиться, что эта реализация работает нормально, давайте протестируем её на большем количестве значений. Чтобы не писать вручную каждый блок it, мы можем генерировать их в цикле for:

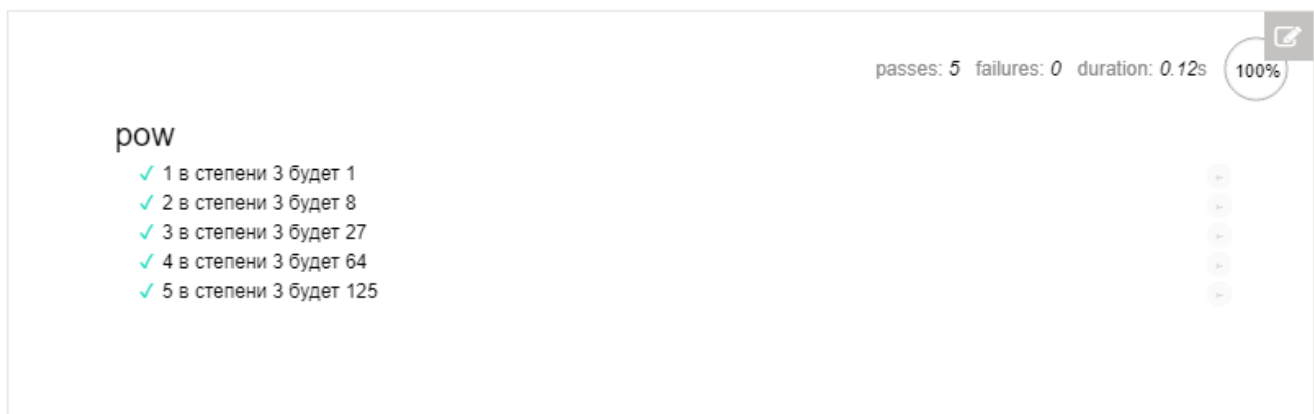
```
describe("pow", function() {

  function makeTest(x) {
    let expected = x * x * x;
    it(`${x} в степени 3 будет ${expected}`, function() {
      assert.equal(pow(x, 3), expected);
    });
  }

  for (let x = 1; x <= 5; x++) {
    makeTest(x);
  }

});
```

Результат:



Вложенные блоки describe

Мы собираемся добавить больше тестов. Однако, перед этим стоит сгруппировать вспомогательную функцию makeTest в цикл for. Нам не нужна функция makeTest в других тестах, она нужна только в цикле for. Её предназначение – проверить, что pow правильно возводит число в заданную степень.

Группировка производится вложенными блоками describe:

```
describe("pow", function() {

  describe("возводит x в степень 3", function() {

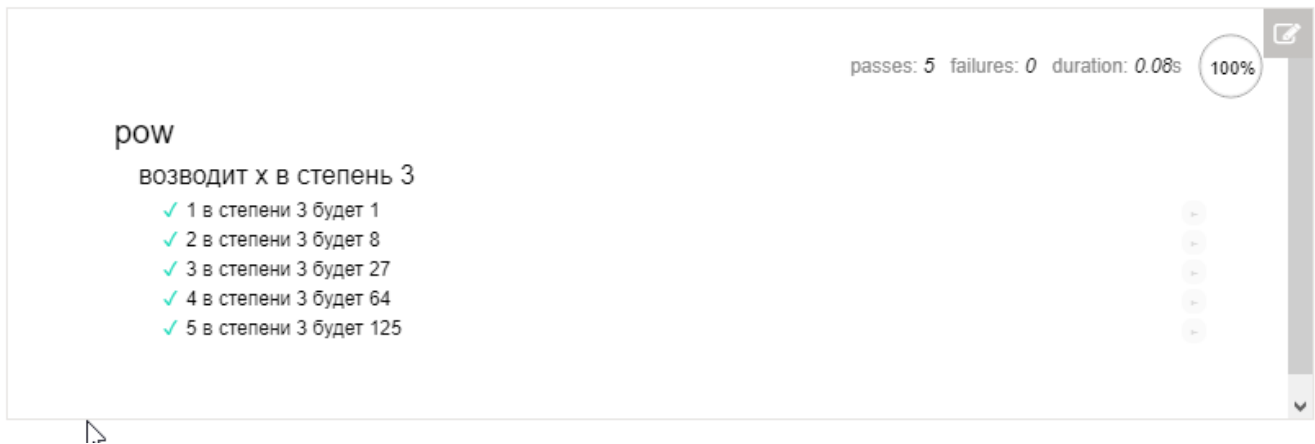
    function makeTest(x) {
      let expected = x * x * x;
      it(`${x} в степени 3 будет ${expected}`, function() {
        assert.equal(pow(x, 3), expected);
      });
    }

    for (let x = 1; x <= 5; x++) {
      makeTest(x);
    }

  });

  // ... другие тесты. Можно писать и describe, и it блоки.
});
```


Вложенные describe образуют новую подгруппу тестов. В результатах мы можем видеть дополнительные отступы в названиях.



В будущем мы можем написать новые it и describe блоки на верхнем уровне со своими собственными вспомогательными функциями. Им не будет доступна функция makeTest из примера выше.

before/after и beforeEach/afterEach

Мы можем задать before/after функции, которые будут выполняться до/после тестов, а также функции beforeEach/afterEach, выполняемые до/после каждого it.

Например:

```
describe("тест", function() {

  before(() => alert("Тестирование началось - перед тестами"));
  after(() => alert("Тестирование закончилось - после всех тестов"));

  beforeEach(() => alert("Перед тестом - начинаем выполнять тест"));
  afterEach(() => alert("После теста - зааничиваем выполнение теста"));

  it('тест 1', () => alert(1));
  it('тест 2', () => alert(2));

});
```

Порядок выполнения будет таким:

```
Тестирование началось - перед тестами (before)
Перед тестом - начинаем выполнять тест (beforeEach)
1
После теста - зааничиваем выполнение теста (afterEach)
Перед тестом - начинаем выполнять тест (beforeEach)
2
После теста - зааничиваем выполнение теста (afterEach)
Тестирование закончилось - после всех тестов (after)
```

Обычно beforeEach/afterEach и before/after используются для инициализации, обнуления счетчиков или чего-нибудь ещё между тестами (или группами тестов).

Расширение спецификации

Основной функционал pow реализован. Первая итерация разработки завершена. Улучшим pow. Как было сказано, функция pow(x, n) предназначена для работы с целыми положительными значениями n. Для обозначения математических ошибок функции JavaScript обычно возвращают NaN. Давайте делать также для некорректных значений n.

Сначала опишем это поведение в спецификации.

```
describe("pow", function() {

  // ...

  it("для отрицательных n возвращает NaN", function() {
    assert.isNaN(pow(2, -1));
  });

  it("для дробных n возвращает NaN", function() {
    assert.isNaN(pow(2, 1.5));
  });

});
```

Результаты с новыми тестами:

passes: 5 failures: 2 duration: 0.08s 100%

pow

✗ если n - отрицательное число, результат будет NaN

```
AssertionError: expected 1 to be NaN
    at Function.assert.isNaN (https://cdnjs.cloudflare.com/ajax/libs/chai/3.5.0/chai.js:2594:35)
    at Context.<anonymous> (test.js:19:12)
```

✗ если n не число, результат будет NaN

```
AssertionError: expected 4 to be NaN
    at Function.assert.isNaN (https://cdnjs.cloudflare.com/ajax/libs/chai/3.5.0/chai.js:2594:35)
    at Context.<anonymous> (test.js:23:12)
```

возводит x в степень 3

- ✓ 1 в степени 3 будет 1
- ✓ 2 в степени 3 будет 8
- ✓ 3 в степени 3 будет 27
- ✓ 4 в степени 3 будет 64
- ✓ 5 в степени 3 будет 125

Новые тесты падают, потому что наша реализация не поддерживает их. Так работает BDD. Сначала мы добавляем тесты, которые падают, а уже потом пишем под них реализацию.

Другие функции сравнения

Обратите внимание на `assert.isNaN`. Это проверка того, что переданное значение равно NaN.

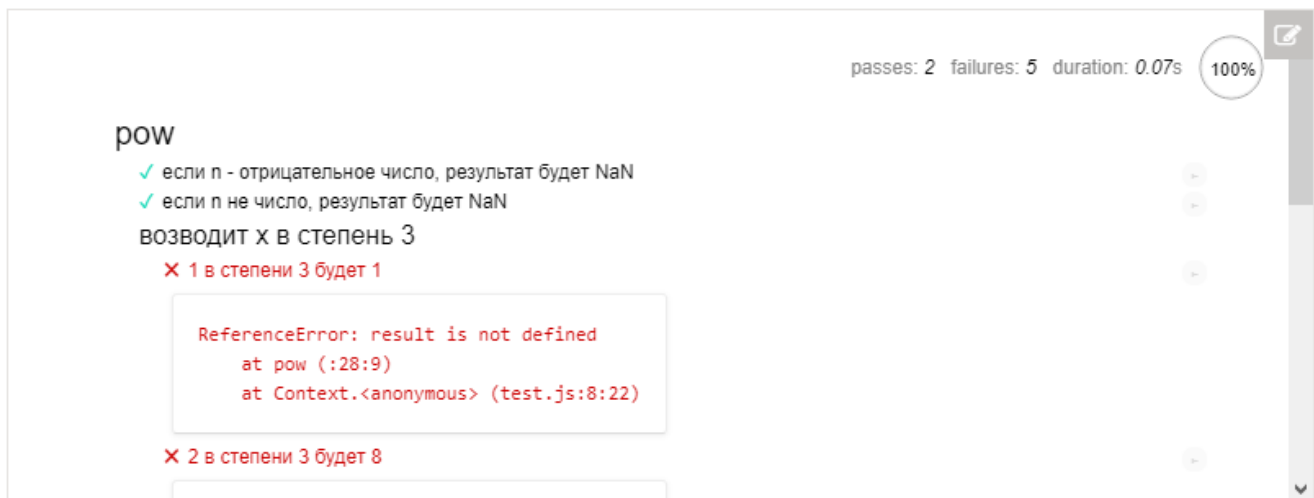
Библиотека Chai содержит множество других подобных функций, например:

- `assert.equal(value1, value2)` – проверяет равенство `value1 == value2`.
- `assert.strictEqual(value1, value2)` – проверяет строгое равенство `value1 === value2`.
- `assert.notEqual`, `assert.notStrictEqual` – проверяет неравенство и строгое неравенство соответственно.
- `assert.isTrue(value)` – проверяет, что `value === true`
- `assert.isFalse(value)` – проверяет, что `value === false`

Итак, нам нужно добавить пару строчек в функцию pow:

```
function pow(x, n) {  
  if (n < 0) return NaN;  
  if (Math.round(n) !== n) return NaN;  
  
  let result = 1;  
  
  for (let i = 0; i < n; i++) {  
    result *= x;  
  }  
  
  return result;  
}
```

Теперь работает, все тесты проходят:



4.Взаимодействие с пользователем: alert, prompt, confirm

В этом разделе мы рассмотрим базовые UI операции: alert, prompt и confirm, которые позволяют работать с данными, полученными от пользователя.

alert

Синтаксис:

```
alert(сообщение)
```

alert выводит на экран окно с сообщением и приостанавливает выполнение скрипта, пока пользователь не нажмёт «ОК».

```
alert( "Привет" );
```

Окно сообщения, которое выводится, является *модальным окном*. Слово «модальное» означает, что посетитель не может взаимодействовать со страницей, нажимать другие кнопки и т.п., пока не разберётся с окном. В данном случае – пока не нажмёт на «ОК».

prompt

Функция prompt принимает два аргумента:

```
result = prompt(title, default);
```

Она выводит модальное окно с заголовком title, полем для ввода текста, заполненным строкой по умолчанию default и кнопками OK/CANCEL. Пользователь должен либо что-то ввести и нажать ОК, либо отменить ввод кликом на CANCEL или нажатием Esc на клавиатуре.

Вызов prompt возвращает то, что ввёл посетитель – строку или специальное значение null, если ввод отменён.

Как и в случае с alert, окно prompt модальное.

```
var years = prompt('Сколько вам лет?', 100);  
  
alert('Вам ' + years + ' лет!')
```

Всегда указывайте default

Второй параметр может отсутствовать. Однако при этом IE вставит в диалог значение по умолчанию "undefined".

Запустите этот код в IE, чтобы понять о чём речь:

```
var test = prompt("Тест");
```

Поэтому рекомендуется *всегда* указывать второй аргумент:

```
var test = prompt("Тест", ''); // <-- так лучше
```

confirm

Синтаксис:

```
result = confirm(question);
```

confirm выводит окно с вопросом question с двумя кнопками: ОК и CANCEL.

Результатом будет true при нажатии ОК и false – при CANCEL(Esc).

Например:

```
var isAdmin = confirm("Вы - администратор?");  
  
alert( isAdmin );
```