



ОБРАБОТКА ИСКЛЮЧЕНИЙ И ЗАМЫКАНИЯ



АЛЕКСЕЙ СУДНИЧНИКОВ



АЛЕКСЕЙ СУДНИЧНИКОВ

Руководитель группы разработки
«Портал ПФДО»



[@avsudnichnikov](https://www.instagram.com/avsudnichnikov)

ПЛАН ЗАНЯТИЯ

1. Внутренние исключения выполнения
2. Создание собственных исключений
3. Перехват исключений
4. Консольные методы для удобной работы
5. Области видимости
6. Замыкания
7. Конструкция `delete`

ВСПОМНИМ ПРОШЛЫЙ МАТЕРИАЛ

Что будет выведено?

```
let person {  
  name: "Олег",  
  printName: () => this.name,  
}  
console.log(person.printName()); // ?
```

ВНУТРЕННИЕ ИСКЛЮЧЕНИЙ ВЫПОЛНЕНИЯ

Откуда появляются ошибки? Некоторые стандартные ситуации не должны происходить.

```
let age = null.age;  
// Uncaught TypeError: Cannot read property 'age' of null
```

Если интерпретатор не может выполнить команду, он выбрасывает ошибку.

СОЗДАНИЕ СОБСТВЕННЫХ ИСКЛЮЧЕНИЙ

СВОИ ИСКЛЮЧЕНИЯ

А можем ли мы сами выбрасывать исключения? Если да, то как и когда?

```
const divider = (a,b) => a / b;  
console.log(divider(1,0)); // Infinity;
```

Вопрос: как запретить деление на 0?

СВОИ ИСКЛЮЧЕНИЯ

Иногда возникают исключительные ситуации, которые, с точки зрения интерпретатора, ошибкой не считаются, но ошибка проявится позже - например, недополучены какие-то параметры с сервера. Или может произойти ошибка со стороны бизнес-логики - например, недопустимое значение исходных данных.

С точки зрения интерпретатора, такие случаи ошибками не считаются, однако, с точки зрения разработчика - они являются ошибками. В таких случаях требуется генерировать свои ошибки.

Для выбрасывания исключения используется конструкция `throw`.

```
1  const divider = (a,b) => {  
2      if(b == 0)  
3          throw "Ошибка деления на 0";  
4  
5      return a / b;  
6  };  
7  console.log(divider(1,2)); // 0.5;  
8  console.log(divider(1,0)); // "Ошибка деления на 0";
```

ОБЪЕКТ ОШИБКИ

Примитивы, вроде текста или чисел не используются как ошибки. Для ошибок существует класс `Error`.

```
1  const divider = (a,b) => {  
2      if(b !== 0){  
3          const divideError = new Error("Ошибка деления на 0");  
4          throw divideError;  
5          // или throw new Error("Ошибка деления на 0");  
6      }  
7  
8      return a / b;  
9  };  
10 console.log(divider(1,2)); // 0.5;  
11 console.log(divider(1,0)); // "Ошибка деления на 0";
```



ПЕРЕХВАТ ИСКЛЮЧЕНИЙ



ЗАЧЕМ ПЕРЕХВАТЫВАТЬ ИСКЛЮЧЕНИЯ?

Если Вы скажете, что ошибки надо решать, а не прятать, я полностью с Вами соглашусь. Однако, всегда ли наличие ошибки зависит от Вас?

Может ли быть ситуация, при которой может произойти ошибка, несмотря на то, что написанный Вами код полностью корректен?



Например:

- нельзя гарантировать, что необходимые данные получены с сервера;
- нельзя гарантировать, что сторонний сервис всегда доступен и корректно работает;
- нельзя гарантировать, что автор библиотеки (или тот, кто использует Вашу библиотеку) так же добросовестно пишет код, как и Вы;
- нельзя гарантировать, что пользователь не сможет ввести некорректные данные (хоть и надо к этому стремиться).

ЗАЧЕМ ПЕРЕХВАТЫВАТЬ ИСКЛЮЧЕНИЯ?

1. Например, при возникновении ошибки необходимо сообщить об этом пользователю как-то культурно

В случае, если пользователь встретится с ошибкой скрипта, лучше, наверное, ему сообщить об этом ошибкой "Что-то пошло не так..." зафиксировать ошибку и дать возможность продолжить работу, а не оставить его один на один с непонятным поведением страницы?

2. Например, при возникновении ошибки необходимо получить расширенную информацию

При выполнении скрипта возникает ошибка. Не всегда очевидно, какие значения данных к этому приводят. В этом случае в блоке `catch` достаточно будет дописать вывод необходимых нам данных, приводящих к ошибке.

3. Например, случившаяся ошибка не должна прерывать выполнение дальнейшего кода

Подключение виджета для отображение погоды на сайте может прекратить выполнение дальнейшего кода. Лучше заранее такое предусмотреть.

КАК ПЕРЕХВАТЫВАТЬ ИСКЛЮЧЕНИЯ?

Конструкция `try..catch` служит для того, чтобы браузер "попытался" интерпретировать код. Однако, если выполнить код не удастся, то можно "поймать" ошибку и/или промежуточные данные, обработать её и затем безопасно выполнять код дальше.

Конструкция `try..catch` состоит из блоков:

- `try`;
- `catch`;
- `finally`.

ОПИСАНИЕ БЛОКОВ КОНСТРУКЦИИ

`try..catch`

- `try`:

В блоке `try` описывается программный код, который браузер должен "попытаться" выполнить.

- `catch`:

В блоке `catch` описывается программный код, который браузер должен выполнить, если в результате выполнения кода в блоке `try` произошла ошибка.

- `finally`:

В блоке `finally` описывается программный код, выполнение которого произойдет независимо от того, произойдёт ли ошибка в результате выполнения кода в блоке `try` или нет.

ПЕРЕХВАТ ИСКЛЮЧЕНИЙ В ДЕЙСТВИИ (ДЕМО)

– `try..catch`

```
1  try {  
2      // .. код, который может выполняться неверно  
3  } catch(e) {  
4      // .. код, который в этом случае выполнится  
5  }
```

– `try..finally`

```
1  try {  
2      // .. код, который может выполняться неверно  
3  } finally {  
4      // .. код, который выполнится в любом случае  
5  }
```

– `try..catch..finally`

```
1  try {  
2      // .. код, который может выполняться неверно  
3  } catch(e) {  
4      // .. код, который в этом случае выполнится  
5  } finally {  
6      // .. код, который выполнится в любом случае  
7  }
```

ОГРАНИЧЕНИЯ ДЛЯ `try...catch`

Перехват ошибки НЕ СРАБОТАЕТ:

- если имеется синтаксическая ошибка;

```
1  try{
2      console.log(Ошибка не произошла!);
3  }catch(e){
4      console.log('Ошибка произошла!');
5  }
6  // Uncaught SyntaxError: missing ) after argument list
```

В этом случае `try...catch` не будет выполняться, интерпретатор сообщит о синтаксической ошибке.

ОГРАНИЧЕНИЯ ДЛЯ `try...catch`

Перехват ошибки НЕ СРАБОТАЕТ:

- если код, в котором произошла ошибка работает асинхронно по отношению к `try...catch`.

```
1 try {
2     setTimeout(()={
3         console.log(null.unknown_property);
4     },200)
5 }catch(e){
6     console.log('Ошибка произошла!');
7 }
8 // Uncaught ReferenceError: Invalid left-hand side in assignment
```

асинхронность будет изучаться в одной из следующих лекций



КОНСОЛЬНЫЕ МЕТОДЫ ДЛЯ УДОБНОЙ РАБОТЫ

ВЫВОД НА КОНСОЛЬ

Всем известен вывод на консоль с помощью `console.log` используемый для вывода, но можно выводить не только сообщение. Можно выводить:

- Сообщение;
- Предупреждение;
- Ошибку.

ОЧИСТКА КОНСОЛИ

После многих сообщений в выводе можно запутаться. Для очистки всего вывода можно использовать `console.clear()`. Либо использовать кнопку кружка.

ФОРМАТИРОВАННЫЙ ВЫВОД

Вы можете напечатать очень хорошую таблицу с объектами, которые вы выводите, используя `console.table()`.

ЗАМЕР ВРЕМЕНИ ВЫПОЛНЕНИЯ КОДА

Быстро ли выполняется ваша программа? Это сложно понять, если не производить замер времени выполнения отдельных функций.

Для замера времени выполнения используются методы

`console.time()` и `console.timeEnd()` (в метод передаётся `id` таймера). Таким образом можно получить время выполнения цикла из 10000 итераций.



ОБЛАСТИ ВИДИМОСТИ

ГЛОБАЛЬНАЯ ОБЛАСТЬ ВИДИМОСТИ

Сделаем программу, которая выводит приветственное сообщение пользователю. Реализуем полноценный пример. Для этого нам понадобится HTML и JavaScript-код.

```
1  let firstName = 'Олег';
2  function showGreeting( person ) {
3      console.log( `С днём рождения, ${person}!` );
4  }
5
6  showGreeting( firstName );
```

МАНИПУЛЯЦИИ С КОНСОЛЬЮ

Теперь откроем страницу и включим консоль.

Введём в консоль:

```
firstName
```

И увидим значение, которое хранится на настоящий момент в нашей переменной.

```
>  
С днём рождения, Олег!  
> firstName  
=> 'Олег'
```

JavaScript ведёт себя так, что для него обращение к переменной `firstName` равносильно:

```
1  let firstName = 'Олег';
2  function showGreeting( person ) {
3      console.log( `С днём рождения, ${person}!` );
4  }
5
6  showGreeting( firstName );
7  console.log( firstName ); // !!!
```

Более того, если в консоли же ввести друг за другом команды:

```
firstName = 'Иван';  
  
showGreeting( firstName );
```

То получим текст поздравления с тем именем, которое мы установили к консоли!

```
> firstName = 'Иван';  
=> 'Иван'  
> showGreeting( firstName );  
С днём рождения, Иван!
```

ПОШАЛИМ?

Проделаем ещё одну шалость и попробуем поменять код функции `showGreeting`. Запишем в консоль:

```
showGreeting = function( person ) { console.log( `Вас взломали, ${person}!` ) }
```

И следом запустим:

```
showGreeting( firstName );
```

```
> showGreeting = function( person ) { console.log( `Вас взломали, ${person}!` ) }  
=> [Function: showGreeting]  
> showGreeting( firstName );  
Вас взломали, Иван!
```

Шалость удалась! Мы смогли подменить и функцию и переменную.

P.S. Содержимое файлов `main.js` и `index.html` при таком способе не меняется. После обновления страницы значение `name` будет снова Олег, а функция `showGreeting` будет снова работать как надо.



ПРОБЛЕМЫ БЕЗОПАСНОСТИ

Очевидно, что если бы наша программа была бы чуть серьёзнее и хранила бы, например, данные о кредитных картах или пароли пользователей, возможность изменения нашего кода делала бы владельца любого проекта беспомощным против хакеров.

Для решения этой проблемы нам необходимо ограничить доступ к переменной и функции.

ГЛОБАЛЬНАЯ ОБЛАСТЬ ВИДИМОСТИ

До настоящего времени мы писали код в *глобальной области видимости* - месте, в котором можно изменить любую функцию или переменную и на них не накладывается никаких ограничений.

Все функции имеют доступ к переменным и функциям глобальной области видимости:

```
1 // эта информация доступна любой функции в программе
2 let secret = 'Ленин - гриб!';
3
4 function showSecret() {
5     // любая функция имеет доступ к глобальной области видимости
6     console.log( secret ); // выдаст содержимое secret
7 }
8
9 console.log( secret ); // тоже выдаст содержимое uglyNews
10 showSecret();
```

ФУНКЦИОНАЛЬНАЯ ОБЛАСТЬ ВИДИМОСТИ

Функциональная область видимости ограничивает доступность переменных и функций *телом функции*. Иными словами, всё, что находится в функции, должно остаться только в ней.

Функция ограничивает доступ к своим переменным:

```
1 function keepSecret() {  
2   let secret = 'Ленин - гриб!';  
3   console.log( secret ); // Выведет «Ленин - гриб!»  
4 }  
5  
6 // Выдаст ошибку «Can't find variable: secret»  
7 console.log( secret );
```

ГРАНИЦЫ ОБЛАСТЕЙ ВИДИМОСТИ

Таким образом, переменные и функции, созданные в *функциональной области видимости* `keepSecret`, доступны только в рамках созданной `keepSecret` :

```
var shared = 'Медведев - шмель!';

function keepSecret() {
  var secret = 'Ленин - гриб!';
  console.log( secret ); // Выведет «Ленин - гриб!»
}

// Выдаст ошибку «Can't find variable: secret»
console.log( secret );
```

keepSecret

Глобальная

Переменные и функции, созданные в *глобальной области видимости*, доступны везде.

ИСПОЛЬЗОВАНИЕ ФУНКЦИЙ

Один из способов решения проблемы с безопасностью - использование *функциональной области видимости*.

Создадим функцию `init` и сразу же её вызовем:

`main.js`

```
1 function init() {  
2     let firstName = 'Олег';  
3     function showGreeting( person ) {  
4         console.log( `С днём рождения, ${person}!` );  
5     }  
6  
7     showGreeting( firstName );  
8 }  
9  
10 init();  
11  
12 console.log( firstName );
```

Результат вызова:

```
>  
ReferenceError: Can't find variable: firstName  
global code@https://replbox.repl.it/data/web_hosting_1/neizerth/ElderlyContentDos/script.js:12:23  
С днём рождения, Олег!  
> █
```

Итого: мы получили сообщение с поздравлением и ошибку, которая возникла на последней строке из-за попытки обращения к переменной `firstName`, которая находится в *функциональной области видимости* и доступна только функции `init`, но не глобальной переменной.

ЗАМЫКАНИЯ. ОКРУЖЕНИЕ

Предположим, что у нас есть некая, нами написанная функция `tick`, которая должна каждый раз при вызове выводить в консоль число на единицу больше (своеобразный таймер):

```
1  function tick() {  
2      // какой-то код, который мы ещё напишем  
3  }  
4  
5  tick(); // 1  
6  tick(); // 2  
7  tick(); // 3
```

Задача: напишите функцию `tick`.

УНИЧТОЖЕНИЕ

Попробуем создать в `tick` переменную и сразу же увеличить её на 1:

```
1 function tick() {  
2     let start = 1;  
3     console.log( start++ );  
4 }  
5  
6 tick(); // 1  
7 tick(); // 1  
8 tick(); // 1
```

Почему это не работает? Дело в том, что `start` создается и уничтожается с каждым новым вызовом `tick`.

ПЕРЕМЕННЫЕ В ГЛОБАЛЬНОЙ ОБЛАСТИ ВИДИМОСТИ

Но мы можем использовать глобальную область видимости!

```
1 let start = 1;
2
3 function tick() {
4     console.log( start++ );
5 }
6
7 tick(); // 1
8 tick(); // 2
9 tick(); // 3
```

В таком случае `start` уничтожится только после того как посетитель **закроет вкладку**. Иными словами, переменные в *глобальной области видимости* **не уничтожаются из памяти**. Переменные *функциональной области видимости* **доступны только на время выполнения функции**.



ЗАМЫКАНИЯ

ЧТО ТАКОЕ ЗАМЫКАНИЕ?

```
1 let start = 1;
2
3 function tick() {
4     console.log( start++ );
5 }
```

Замыкания—это функция вместе со всеми внешними переменными, которые ей доступны. Другими словами, функция, определённая в *замыкании*, «запоминает» окружение, в котором она была создана.

В нашем случае замыканием является функция `tick`.

ДОБАВЛЯЕМ УСЛОВИЕ

Если нам потребуется создать функцию `tick10`, которая бы считала с 10, нам потребуется доработать программу:

```
1  let start = 1;
2
3  function tick() {
4      console.log( start++ );
5  }
6
7  tick(); // 1
8  tick(); // 2
9  tick(); // 3
10
11 let start10 = 10;
12
13 function tick10() {
14     console.log( start10++ );
15 }
16
17 tick10(); // 10
18 tick10(); // 11
19 tick10(); // 12
```

Можно ли объединить этот код?

ИЗБАВЛЯЕМСЯ ОТ ПЕРЕМЕННЫХ В ГЛОБАЛЬНОЙ ОБЛАСТИ ВИДИМОСТИ

Как и говорилось ранее, `return` может возвращать любое выражение, в том числе и функцию.

Предположим, что у нас есть мифическая функция `createCounter`, которая должна работать так:

```
1 function createCounter() {  
2     // какой-то код  
3 }  
4  
5 let tick = createCounter( 1 ), // начинаем считать с 1  
6     tick10 = createCounter( 10 ); // начинаем считать с 10  
7  
8 tick(); // 1  
9 tick(); // 2  
10 tick(); // 3  
11  
12 tick10(); // 10  
13 tick10(); // 11  
14 tick10(); // 12
```

АНАЛИЗИРУЕМ

Раз `tick` и `tick10` - функции, значит `createCounter` должна возвращать функцию:

```
1 function createCounter() {  
2     /*  
3     Именно эта функция сработает, когда мы будем вызывать tick или tick10.  
4     Иными словами, именно эта функция поместится в tick и tick10.  
5     */  
6     return function() {  
7         // какой-то код  
8     }  
9 }
```

В `createCounter` мы передаём начальное значение счётчика:

```
1 function createCounter( start = 0 ) {  
2   return function() {  
3     // какой-то код  
4   }  
5 }
```

Как и в базовой версии `tick`, нам нужно использовать консоль и увеличение значения на один для работы функции:

```
1 function createCounter( start = 0 ) {  
2   return function() {  
3     console.log( start++ );  
4   }  
5 }
```

Готово!



ЗАМЫКАНИЯ. ПОЯСНЕНИЕ (ДЕМО)

В нашем случае *замыканием* является вложенная в `createCounter` функция, которая использует аргумент `start`, находящийся в области видимости вне `createCounter`.

Как происходит поиск переменных и функций по областям видимости?

```
1 function createCounter( start = 0 ) {  
2     return function() {  
3         console.log( start++ );  
4     }  
5 }  
6  
7 let tick = createCounter( 1 ), // начинаем считать с 1  
8     tick10 = createCounter( 10 ); // начинаем считать с 10  
9  
10 tick(); // 1  
11 tick(); // 2  
12 tick(); // 3  
13  
14 tick10(); // 10  
15 tick10(); // 11  
16 tick10(); // 12
```

ПОИСК В ОБЛАСТЯХ ВИДИМОСТИ

Если в текущей области видимости какая-то переменная не была найдена, она ищется в **области видимости выше** (включая глобальную).

```
1  let x = 8;
2  function findX() {
3      function stillFindX() {
4          function whereIsX() {
5              console.log(`X = ${x}`); // X = 8
6          }
7          whereIsX();
8      }
9      stillFindX();
10 }
11 findX();
```


ОСТАНОВКА ПОИСКА

Поиск останавливается при нахождении переменной.

```
1 console.log(`X = ${x}`); // X = 3;
2 function findX() {
3   let x = 8;
4   function stillFindX() {
5     let x = 4; // тут поиск переменной x остановится
6     function whereIsX() {
7       console.log(`X = ${x}`); // X = 4
8     }
9   }
10  stillFindX();
11
12  console.log(`X = ${x}`); // X = 8;
13 }
14 findX();
15
16 console.log(`X = ${x}`); // X = 3;
```

Если переменная не найдена ни в одной области видимости, мы получим ошибку.

ПОИСК В ОБЛАСТЯХ ВИДИМОСТИ

```
1 function leninHistory() {
2   let leninName = 'Владимир';
3   console.log(`Меня зовут ${leninName}, у меня нет предшественников`)
4   function stalinHistory() {
5     let stalinName = 'Иосиф';
6
7     console.log(`Меня зовут ${stalinName}, Имя моего предшественника: ${leninName}`);
8
9     function medvedevHistory() {
10      let medvedevName = 'Дмитрий';
11
12      console.log(`Меня зовут ${medvedevName}, Имена моих предшественников: ${stalinName}, ${leninName}`);
13    }
14    medvedevHistory();
15  }
16  // не забываем выполнить объявленную функцию
17  stalinHistory();
18 }
19
20 leninHistory()
```

```

1 function leninHistory() {
2   var leninName = 'Владимир';
3   console.log(`Меня зовут ${leninName}, у меня нет предшественников`)
4   function stalinHistory() {
5     var stalinName = 'Иосиф';
6
7     console.log(`Меня зовут ${stalinName}, Имя моего предшественника: ${leninName}`);
8
9     function medvedevHistory() {
10      var medvedevName = 'Дмитрий';
11
12      console.log(`Меня зовут ${medvedevName}, Имена моих предшественников: ${stalinName}, ${leninName}`);
13    }
14    medvedevHistory();
15  }
16  // не забываем выполнить объявленную функцию
17  stalinHistory();
18 }
19
20 leninHistory()

```

leninHistory



У меня не было
предшественников

stalinHistory



Я знаю Ленина

medvedevHistory



Я знаю Ленина
и Сталина

Иллюстрация областей видимости:

```

1 function leninHistory() {
2   var leninName = 'Владимир';
3   console.log(`Меня зовут ${leninName}, у меня нет предшественников`)
4   function stalinHistory() {
5     var stalinName = 'Иосиф';
6
7     console.log(`Меня зовут ${stalinName}, Имя моего предшественника: ${leninName}`);
8
9     function medvedevHistory() {
10      var medvedevName = 'Дмитрий';
11
12      console.log(`Меня зовут ${medvedevName}, Имена моих предшественников: ${stalinName}, ${leninName}`);
13    }
14    medvedevHistory();
15  }
16  // не забываем выполнить объявленную функцию
17  stalinHistory();
18 }
19
20 leninHistory()

```

ИЗМЕНЕНИЯ В ОБЛАСТЯХ ВИДИМОСТИ

Будьте аккуратны, используя замыкания:

```
1 let firstName = 'Олег';  
2 function changeName() {  
3     firstName = 'Иван';  
4 }  
5  
6 console.log( firstName ); // Олег  
7 changeName();  
8 console.log( firstName ); // Иван
```

Для этого не забывайте ставить `let`:

```
1 let firstName = 'Олег';
2 function changeName() {
3     // !!!
4     let firstName = 'Иван';
5 }
6
7 console.log( firstName ); // Олег
8 changeName();
9 console.log( firstName ); // Олег
```



КОНСТРУКЦИЯ
delete



КАК ПРОИСХОДИТ УДАЛЕНИЕ ОБЪЕКТОВ В JAVASCRIPT

Во время выполнения скрипта для каждого примитива или объекта выделяется определенный участок памяти.

Память не бесконечна, поэтому ее требуется периодически очищать от "мусора" - неиспользуемых значений переменных, объектов и их свойств. За этим следит "сборщик мусора" - алгоритм, очищающий память.

Как понять, можно ли удалить какое-то значение? Это просто. Значение считается неиспользуемым, если на него не ведет никакая ссылка.

Если мы объявим переменную:

```
1  x = 'any_value';
2  /*
3   в памяти будет записано значение 'any_value',
4   на которое ссылается указатель x
5  */
6  console.log(x); // any_value
7
8  /*
9   можем присвоить еще одному указателю это значение.
10  И этот указатель тоже будет ссылаться
11  на значение 'any_value'
12  */
13  y = x;
14  console.log(y); // any_value
```


ЗАЧЕМ ИСПОЛЬЗОВАТЬ `delete` ?

Сборщик мусора удаляет те значения, на которые не ссылается ни одна ссылка-указатель.

Если на значение ведёт ссылка-указатель `x`, значение не будет удалено. Оператор `delete` удаляет ссылку на значение и позволяет сборщику мусора высвободить память компьютера (если нет других ссылок на значение).

delete

Оператор `delete` позволяет удалять свойства объектов. Оператор `delete` служит для удаления элемента массива, имени или свойства объекта.

Синтаксис:

```
1 delete nameOfVariable;  
2 delete object.property;  
3 delete object['property'];  
4 delete array['index'];
```

delete ИМЕЕТ СВОИ ОСОБЕННОСТИ

1. `delete` возвращает `false` только если свойство существует, но не может быть удалено, и `true` - в любых других случаях.

```
1 | let anybodyObject = {"first": 1};  
2 | delete anybodyObject.second; // true  
3 | console.log(anybodyObject) // {first: 1}
```

delete ИМЕЕТ СВОИ ОСОБЕННОСТИ

2. С помощью `delete` можно удалить только свойство объекта, а значит, нельзя удалить переменные (объявленные через `var` и `let`);

```
1  let x = "you can't delete me";
2  delete x; // false
3  console.log(x); // "you can't delete me"
4  // ----- ***** ----- //
5  function g() {
6      let x = 5;
7      delete x;
8      return x;
9  }
10 console.log(g()); // 5
```

`delete` ИМЕЕТ СВОИ ОСОБЕННОСТИ

3. при удалении элемента массива, в массиве сохраняется "пустое место" (`empty`) от этого элемента, то есть длина массива при этом не изменится;

```
1 | let array = ["first", "second", "third"];
2 | delete array[2]; //true
3 | console.log(array); // ["first", "second", empty]
```

`delete` ИМЕЕТ СВОИ ОСОБЕННОСТИ

4. `delete` не изменяет прототип объекта;
5. существуют свойства, которые нельзя удалить. Например:

```
f = [1, 2, 'third'];  
delete f.length; // false
```

ЧЕМУ МЫ НАУЧИЛИСЬ?

1. Познакомились с исключениями: узнали как генерировать и перехватывать исключения
2. Изучили новые интересные методы для работы с консолью
3. Узнали про области видимости
4. Познакомились с замыканиями и научились замыкать переменные
5. Познакомились с конструкцией `delete`, которая служит для очистки памяти



ДОМАШНЕЕ ЗАДАНИЕ

Давайте посмотрим ваше [домашнее задание](#).

- Вопросы по домашней работе задаем в чате Slack!
- Задачи можно сдавать по частям.
- Зачет по домашней работе проставляется после того, как приняты **все задачи**.



Спасибо за внимание! Время задавать вопросы

АЛЕКСЕЙ СУДНИЧНИКОВ



[@avsudnichnikov](#)