НЕТОЛОГИЯ

ОБРАБОТКА ИСКЛЮЧЕНИЙ И ЗАМЫКАНИЯ

АЛЕКСЕЙ СУДНИЧНИКОВ

АЛЕКСЕЙ СУДНИЧНИКОВ

Руководитель группы разработки
«Портал ПФДО»

@avsudnichnikov

ПЛАН ЗАНЯТИЯ

1. Внутренние исключений 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",
    secondName: "Петров",
    getName: () => this.name,
    getSecondName: function() {
      return this.secondName;
    }
}
console.log(person.getName()); // ?
console.log(person.getSecondName()); // ?
```

ВНУТРЕННИЕ ИСКЛЮЧЕНИЙ JS

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`.

```
const divider = (a,b) => {
    if(b == 0)
        throw "Ошибка деления на 0";

    return a / b;
};
console.log(divider(1,2)); // 0.5;
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`.

```
const divider = (a,b) => {
    if(b !== 0){
        const divideError = new Error("Ошибка деления на 0");
        throw divideError;
        // или throw new Error("Ошибка деления на 0");
    }

    return a / b;
};
console.log(divider(1,2)); // 0.5;
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 встретиться с при выполнении скрипта, то объект ошибки может зафиксировать не вычисляемое действие. Объект ошибки позволит подробно описать, какая произошла проблема и почему она возникла.

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`

```
try {
    // .. код, который может выполниться неверно
} catch(e) {
    // .. код, который в этом случае выполнится
}
```

- `try..finally`

```
try {
    // .. код, который может выполниться неверно
} finally {
    // .. код, который выполнится в любом случае
}
```

– `try..catch..finally`

```
try {
    // .. код, который может выполниться неверно
} catch(e) {
    // .. код, который в этом случае выполнится
} finally {
    // .. код, который выполнится в любом случае
}
```

ОГРАНИЧЕНИЯ ДЛЯ `try...catch`

Перехват ошибки НЕ СРАБОТАЕТ:

– если имеется синтаксическая ошибка;

```
try{
    console.log(Ошибка не произошла!);
}catch(e){
    console.log('Ошибка произошла!');
}
// 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`.

```
try {
    setTimeout(()={
        console.log(null.unknown_property);
    },200)
}catch(e){
    console.log('Ошибка произошла!');
}
// 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.

```
let firstName = 'Олег';
function showGreeting( person ) {
  console.log( `С днём рождения, ${person}!` );
}

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.

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

```
let firstName = 'Олег';
function showGreeting( person ) {
  console.log( `С днём рождения, ${person}!` );
}

showGreeting( firstName );
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!

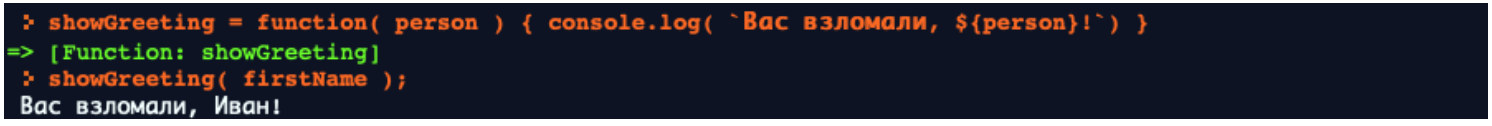
ПОШАЛИМ?

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 );
```



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:

```
// эта информация доступна любой функции в программе
let secret = 'Ленин - гриб!';

function showSecret() {
  // любая функция имеет доступ к глобальной области видимости
  console.log( secret ); // выдаст содержимое secret
}

console.log( secret ); // тоже выдаст содержимое uglyNews
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.

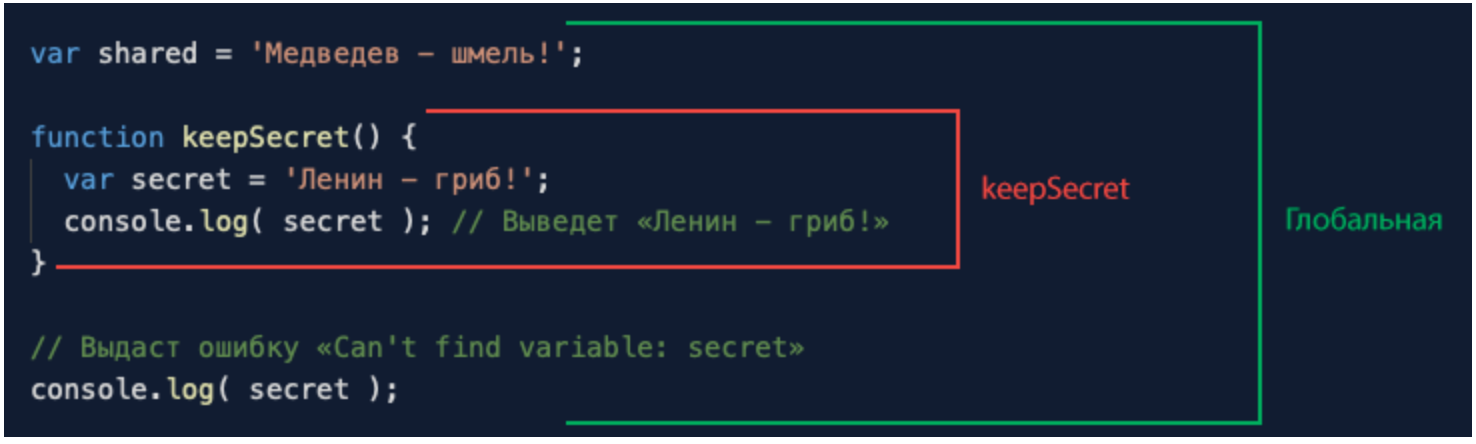
Функция ограничивает доступ к своим переменным:

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

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

ГРАНИЦЫ ОБЛАСТЕЙ ВИДИМОСТИ

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



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`

```
function init() {
    let firstName = 'Олег';
    function showGreeting( person ) {
        console.log( `С днём рождения, ${person}!` );
    }

    showGreeting( firstName );
}

init();

console.log( firstName );
```

Результат вызова:

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):

```
function tick() {
    // какой-то код, который мы ещё напишем
}

tick(); // 1
tick(); // 2
tick(); // 3
```

Задача: напишите функцию `tick`.

УНИЧТОЖЕНИЕ

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

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

tick(); // 1
tick(); // 1
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!

```
let start = 1;

function tick() {
    console.log( start++ );
}

tick(); // 1
tick(); // 2
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?

```
let start = 1;

function tick() {
    console.log( start++ );
}
```

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:

```
let start = 1;

function tick() {
    console.log( start++ );
}

tick(); // 1
tick(); // 2
tick(); // 3

let start10 = 10;

function tick10() {
    console.log( start10++ );
}

tick10(); // 10
tick10(); // 11
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:

```
function createCounter() {
    // какой-то код
}

let tick = createCounter( 1 ), // начинаем считать с 1
    tick10 = createCounter( 10 ); // начинаем считать с 10

tick(); // 1
tick(); // 2
tick(); // 3

tick10(); // 10
tick10(); // 11
tick10(); // 12
```

АНАЛИЗИРУЕМ

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

```
function createCounter() {
    /*
      Именно эта функция сработает, когда мы будем вызывать tick или tick10.
      Иными словами, именно эта функция поместится в tick и tick10.
    */
    return function() {
      // какой-то код
    }
}
```

В `createCounter` мы передаём начальное значение счётчика:

```
function createCounter( start = 0 ) {
    return function() {
      // какой-то код
    }
}
```

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

```
function createCounter( start = 0 ) {
    return function() {
      console.log( start++ );
    }
}
```

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?

```
function createCounter( start = 0 ) {
    return function() {
      console.log( start++ );
    }
}

let tick = createCounter( 1 ), // начинаем считать с 1
    tick10 = createCounter( 10 ); // начинаем считать с 10

tick(); // 1
tick(); // 2
tick(); // 3

tick10(); // 10
tick10(); // 11
tick10(); // 12
```

ПОИСК В ОБЛАСТЯХ ВИДИМОСТИ

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

```
let x = 8;
function findX() {
  function stillFindX() {
    function whereIsX() {
      console.log(`X = ${x}`); // X = 8
    }
    whereIsX();
  }
  stillFindX();
}
findX();
```

ОСТАНОВКА ПОИСКА

Поиск останавливается при нахождении переменной.

```
console.log(`X = ${x}`); // X = 3;
function findX() {
  let x = 8;
  function stillFindX() {
    let x = 4; // тут поиск переменной x остановится
    function whereIsX() {
      console.log(`X = ${x}`); // X = 4
    }
  }
  stillFindX();

  console.log(`X = ${x}`); // X = 8;
}
findX();

console.log(`X = ${x}`); // X = 3;
```

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

ПОИСК В ОБЛАСТЯХ ВИДИМОСТИ

```
function leninHistory() {
  let leninName = 'Владимир';
  console.log(`Меня зовут ${leninName}, у меня нет предшественников`)
  function stalinHistory() {
    let stalinName = 'Иосиф';

    console.log(`Меня зовут ${stalinName}, Имя моего предшественника: ${leninName}`);

    function medvedevHistory() {
      let medvedevName = 'Дмитрий';

      console.log(`Меня зовут ${medvedevName}, Имена моих предшественников: ${stalinName}, ${leninName}`);
    }
    medvedevHistory();
  }
  // не забываем выполнить объявленную функцию
  stalinHistory();
}

leninHistory()
```

```
function leninHistory() {
  var leninName = 'Владимир';
  console.log(`Меня зовут ${leninName}, у меня нет предшественников`)
  function stalinHistory() {
    var stalinName = 'Иосиф';

    console.log(`Меня зовут ${stalinName}, Имя моего предшественника: ${leninName}`);

    function medvedevHistory() {
      var medvedevName = 'Дмитрий';

      console.log(`Меня зовут ${medvedevName}, Имена моих предшественников: ${stalinName}, ${leninName}`);
    }
    medvedevHistory();
  }
  // не забываем выполнить объявленную функцию
  stalinHistory();
}

leninHistory()
```

leninHistory

У меня не было предшественников

stalinHistory

Я знаю Ленина

medvedevHistory

Я знаю Ленина и Сталина

Иллюстрация областей видимости:

```
function leninHistory() {
  var leninName = 'Владимир';
  console.log(`Меня зовут ${leninName}, у меня нет предшественников`)
  function stalinHistory() {
    var stalinName = 'Иосиф';

    console.log(`Меня зовут ${stalinName}, Имя моего предшественника: ${leninName}`);

    function medvedevHistory() {
      var medvedevName = 'Дмитрий';

      console.log(`Меня зовут ${medvedevName}, Имена моих предшественников: ${stalinName}, ${leninName}`);
    }
    medvedevHistory();
  }
  // не забываем выполнить объявленную функцию
  stalinHistory();
}

leninHistory()
```

global

ИЗМЕНЕНИЯ В ОБЛАСТЯХ ВИДИМОСТИ

Будьте аккуратны, используя замыкания:

```
let firstName = 'Олег';
function changeName() {
  firstName = 'Иван';
}

console.log( firstName ); // Олег
changeName();
console.log( firstName ); // Иван
```

Для этого не забывайте ставить `let`:

```
let firstName = 'Олег';
function changeName() {
  // !!!
  let firstName = 'Иван';
}

console.log( firstName ); // Олег
changeName();
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:

```
x = 'any_value';
/*
  в памяти будет записано значение 'any_value',
  на которое ссылается указатель x
*/
console.log(x); // any_value

/*
  можем присвоить еще одному указателю это значение.
  И этот указатель тоже будет ссылаться
  на значение 'any_value'
*/
y = x;
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:

```
delete nameOfVariable;
delete object.property;
delete object['property'];
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.

```
let anybodyObject = {"first": 1};
delete anybodyObject.second; // true
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`);

```
let x = "you can`t delete me";
delete x; // false
console.log(x); // "you can`t delete me"
// ----- ***** ----- //
function g() {
  let x = 5;
  delete x;
  return x;
}
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;

```
let array = ["first","second","third"];
delete array[2]; //true
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 **все задачи**.

НЕТОЛОГИЯ

Спасибо за внимание! Время задавать вопросы

АЛЕКСЕЙ СУДНИЧНИКОВ

@avsudnichnikov