



СИНТАКСИЧЕСКИЕ КОНСТРУКЦИИ



АЛЕКСАНДР ШЛЕЙКО / ЯНДЕКС





АЛЕКСАНДР ШЛЕЙКО

разработчик в Яндекс

3

ПЛАН ЗАНЯТИЯ

- [Легенда](#)
- [Перехват ошибки](#)
- Генерация ошибки

- Оператор delete
- Стрелочные функции
- Замыкания
- Интересное чтение



ЛЕГЕНДА

После обучения Вы устроились на стажировку в небольшую it-компанию. До Вас стажировку проходил на этом месте другой стажер, который не смог завершить небольшой проект. Ваш куратор поручил Вам разобраться с его кодом.

5

ПРИСТУПИМ!





ЗАЧЕМ ПЕРЕХВАТЫВАТЬ ОШИБКИ?

Как бы хорошо не был написан код, он не застрахован от ошибок. Обычное поведение программы - в случае ошибки сообщить о ней и окончить выполнение дальнейшего кода. Однако, бывают ситуации, когда требуется иное поведение.

ЗАЧЕМ ПЕРЕХВАТЫВАТЬ ОШИБКИ?

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 ОШИБКИ?

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?



ЗАЧЕМ ПЕРЕХВАТЫВАТЬ ОШИБКИ?

Например, при возникновении ошибки необходимо получить большую 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.



ЗАЧЕМ ПЕРЕХВАТЫВАТЬ ОШИБКИ?

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.



TRY..CATCH

ПЕРЕХВАТ ОШИБКИ

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.

ПЕРЕХВАТ ОШИБКИ

Конструкция `try..catch` состоит из блоков:

- `try`
- `catch`
- `finally`



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.

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

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

20

TRY..CATCH..FINALLY

```
try {
```

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

21

ПЕРЕХВАТ ОШИБКИ НЕ СРАБОТАЕТ:

- если имеется **синтаксическая** ошибка;

try {

```
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

22

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
```

23

js/game.src.js:102

```
try {  
  this.gameOverDiv.classList.remove('fade');  
} catch (e) {  
  console.info('Not found #game_over!');  
}
```

js/game.src.js:191

```
try {  
  this.gameOverDiv.classList.add('fade');  
} catch (e) {  
  console.info('Not found #game_over!');  
}
```


25

Если произойдёт ошибка при выполнении кода в блоке `try`, в консоли будет выведено сообщение:

```
Not found #game_over!
```

Это сообщение будет выведено независимо от того, какая ошибка в `try` произойдёт на самом деле.

ЗАЧЕМ ГЕНЕРИРОВАТЬ СВОИ ОШИБКИ?

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.

THROW

Оператор `throw` создаёт ошибку.

Например,

```
throw('Я ошибка!');  
// -> Uncaught Я ошибка!
```

THROW

или

```
const a = 1;
const b = 1;
const c = 1;
const d = b ** 2 - 4 * a * c;
if(d < 0) throw ('При дискриминанте меньше нуля\
уравнение не имеет вещественных корней!')
console.log(x1 = (-b - Math.sqrt(d))/2/a);
// -> Uncaught При дискриминанте меньше нуля
// уравнение не имеет вещественных корней!
```

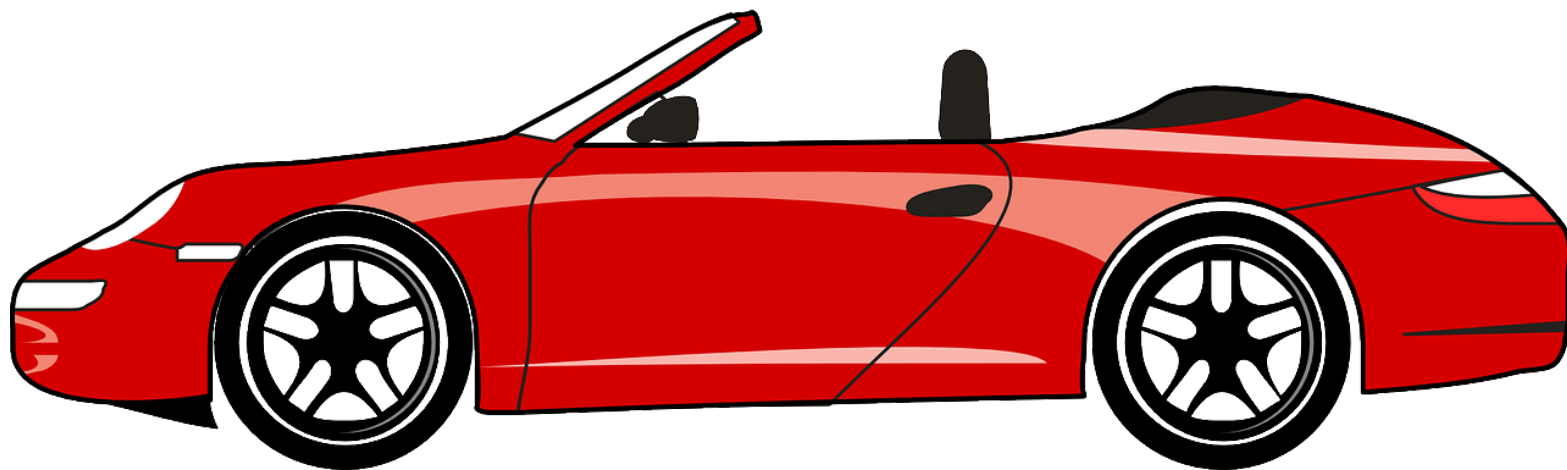
THROW И TRY..CATCH

Удобно, не правда ли?

Если `try...catch` служит для перехвата ошибки и позволяет дальнейшее выполнение кода, `throw` прекращает выполнение кода и создает ошибку.

Используя эти операторы разработчик может контролировать поведение скрипта в отношении ошибок.

ДВИГАЕМСЯ ДАЛЬШЕ



ОПЕРАТОР DELETE

КАК ПРОИСХОДИТ УДАЛЕНИЕ ОБЪЕКТОВ В JAVASCRIPT

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.

КАК ПРОИСХОДИТ УДАЛЕНИЕ ОБЪЕКТОВ В JAVASCRIPT

Если мы объявим переменную:

```
x = 'red_car';
```

в памяти будет записано значение *red_car*, на которое ссылается указатель

x

```
console.log(x);  
// -> red_car
```

КАК ПРОИСХОДИТ УДАЛЕНИЕ ОБЪЕКТОВ В JAVASCRIPT

Можем присвоить еще одному указателю это значение. И этот указатель тоже будет ссылаться на значение *red_car*

```
y = x;  
console.log(y);  
// -> red_car
```

ЗАЧЕМ ИСПОЛЬЗОВАТЬ `delete` ?

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

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'];
```

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

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

DELETE ИМЕЕТ СВОИ ОСОБЕННОСТИ

2. С помощью `delete` можно удалить только свойство объекта, а значит

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

```
var x = "you can't delete me";  
console.log(delete x);  
// -> false  
console.log(x);  
// -> you can't delete me
```

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

```
(empty), then the element is completely removed from the array.  
  
let array = ["first", "second", "third"];  
console.log(delete array[2]);  
// -> true  
console.log(array);  
// -> ["first", "second", empty]
```

40

DELETE ИМЕЕТ СВОИ ОСОБЕННОСТИ

4. delete не изменяет прототип объекта

5. существуют свойства, которые нельзя удалить. Например:

```
f = [1,2,'third'];  
console.log(delete f.length);  
// -> false;  
console.log(f.length);  
// -> 3;
```

41

ДАВАЙТЕ УДАЛИМ НАШ КОНУС

```
delete this.cones[key];
```





СТРЕЛОЧНЫЕ ФУНКЦИИ

43

СТРЕЛОЧНЫЕ ФУНКЦИИ

В ES6 появилась возможность задания функций через «стрелку» =>

Например,

```
let sum = function(a,b){  
  return a + b;  
}
```

МОЖНО ЗАПИСАТЬ ТАК:

```
let sum(a,b)=>{  
  return a + b;  
}
```

44

СТРЕЛОЧНЫЕ ФУНКЦИИ

Если в теле функции не более одной операции, то не обязательно использовать `{}`, при этом для возврата значения не требуется писать `return`.

Например,

```
let sum = function(a,b){  
  return a + b  
}
```

можно записать так:

```
let sum(a,b) => a + b;
```

45

СТРЕЛОЧНЫЕ ФУНКЦИИ

Зачем нужны стрелочные функции?

Стрелочные функции - НЕ “просто короткая запись” обычных функций. В отличие от стрелочных, обычные функции имеют свой контекст. (свой *this*). Стрелочные функции не имеют своего контекста (своего *this*), а берут его из своего окружения.

СТРЕЛОЧНЫЕ ФУНКЦИИ

Например,

если функцию

```
this.car_pos = this.POS_UNDEFINED;  
    setTimeout(function(){  
        this.car_pos = pos;  
    }, 500 / this.speed);
```

заменить на стрелочную:

```
this.car_pos = this.POS_UNDEFINED;
```

```
this.car_pos = this.POS_UNDEFINED;  
setTimeout(()=>  
  this.car_pos = pos  
  , 500 / this.speed);
```

то контекстом `setTimeout` станет нужный нам объект *game*.

47






**А ДАВАЙТЕ ПОСЧИТАЕМ. КАК МОЖНО
ПОСЧИТАТЬ КОЛИЧЕСТВО ПРОЕХАННЫХ
КОНУСОВ?**

ЗАМЫКАНИЯ

НЕМНОГО ПОДУМАЕМ ЛОГИЧЕСКИ

```
26
27 w.onkeypress = function () {
28     game.changeLane();
29 };
30
31 function removeCone(key) {
32     game.cones[key].remove();
33     delete game.cones[key];
34 }
```



В переменной `key` в функции `removeCone` у нас уже хранится значение количества конусов, которые мы проехали

ОСВЕЖИМ ТЕОРИЮ

в JavaScript есть 7 базовых типов данных:

- `undefined`;
- `null`;
- `boolean`;
- `number`;
- `string`;
- `symbol`;
- `object`.

ОСВЕЖИМ ТЕОРИЮ

При этом все, кроме `object` считаются примитивными.

Условно у типа `object` можно выделить «подтипы»:

- массив (`Array`),
- функция (`Function`),
- регулярное выражение (`RegExp`)
- и другие.

ОСВЕЖИМ ТЕОРИЮ

Каким образом можно скопировать объект в javascript?

ОСВЕЖИМ ТЕОРИЮ

в JavaScript есть только один способ создать новый объект - создать новый объект.

При попытке скопировать объект, например, таким образом:

```
func2 = func1;
```

, Вы получите новую ссылку на объект.

ОСВЕЖИМ ТЕОРИЮ

При запуске функции создается объект *LexicalEnvironment* - дополнительный параметр, который включает в себя аргументы, функции и локальные переменные, которые определяются при вызове функции.

Областью видимости таких параметров является тело функции.

Теория теперь достаточно свежая

ТАК ЧТО ЖЕ ТАКОЕ ЗАМЫКАНИЕ?

Например:

```
powerOfTwo = function() {  
  var power = 1;  
  return function() {  
    power *= 2;  
    p = power  
    return p;  
  }  
}()
```

57

```
console.log(powerOfTwo());  
console.log(powerOfTwo());  
console.log(powerOfTwo());  
console.log(powerOfTwo());
```

```
console.log(powerOfTwo());
```

```
// -> 2  
// -> 4  
// -> 8  
// -> 16  
// -> 32
```

58

НАШЕ ЗАМЫКАНИЕ

```
removeConeWrapper() {  
  let last = 0;
```



```
    last = key;
  }
  return last;
};
},
```

59

ЗАЧЕМ НУЖНЫ ЗАМЫКАНИЯ?

- Замыкания позволяют сохранять данные функции между ее вызовами;
- Замыкания позволяют создавать функции, генерирующие функции;

- Замыкания позволяют ограничить область видимости переменных.

60

LINTING

```
.js\user.settings.js  
11:17  warning  Unexpected unnamed function func-names  
28:21  warning  Unexpected unnamed function func-names
```

Видно, что есть ещё места, над которыми нам нужно поработать.

61

ИТАК, ПОДВЕДЕМ ИТОГИ

Сегодня нами были освоены новые инструменты:

1. перехват ошибок
2. удаление объекта
3. стрелочная функция

ИНТЕРЕСНОЕ ЧТИВО:

Перехват ошибки:

- [MDN - try..catch](#)
- [learn.javascript - исключения](#)

Удаление переменной:

- [MDN - delete](#)

Defunct: Kill the last variable before

- [Perfection Kills - Understanding delete](#)

Стрелочные функции:

- [MDN - arrow functions](#)
- [Habr - Введение в стрелочные функции](#)

Замыкания:

- [MDN - closures](#)
- [Habr - Замыкания в JavaScript](#)

63

COPYRIGHT

Все изображения (за исключением конуса и взрыва из репозитория и эмблемы JS) взяты с ресурса [pixabay](#) и распространяются по лицензии [CC0](#).

**СПАСИБО ЗА ВНИМАНИЕ!!!
ЖДУ ВАШИХ ВОПРОСОВ**





АЛЕКСАНДР ШЛЕЙКО



https://telegram.me/dustyo_0



<https://vk.com/shleiko>



