

СИНТАКСИЧЕСКИЕ КОНСТРУКЦИИ



АЛЕКСАНДР ШЛЕЙКО / ЯНДЕКС



АЛЕКСАНДР ШЛЕЙКО

разработчик в Яндекс



ПЛАН ЗАНЯТИЯ

- [Легенда](#)
- [Перехват ошибки](#)
- [Генерация ошибки](#)
- [Стрелочные функции](#)
- [Оператор delete](#)
- [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.

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..CATCH

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

TRY..FINALLY

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

TRY..CATCH..FINALLY

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

ПЕРЕХВАТ ОШИБКИ:

> js/game.src.js:86

```
try {  
  this.distanceBox.innerText = Math.ceil(this.distance / 30);  
} catch (e) {  
  console.log(e);  
}
```

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

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 TypeError: Cannot read property  
// 'unknown_property' of null at setTimeout
```

ОШИБКУ ЛУЧШЕ ИСПРАВИТЬ

> добавить в index.html:16

```
<div>Distance: <span id="distance_box">0</span>m</div>
```




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!
```

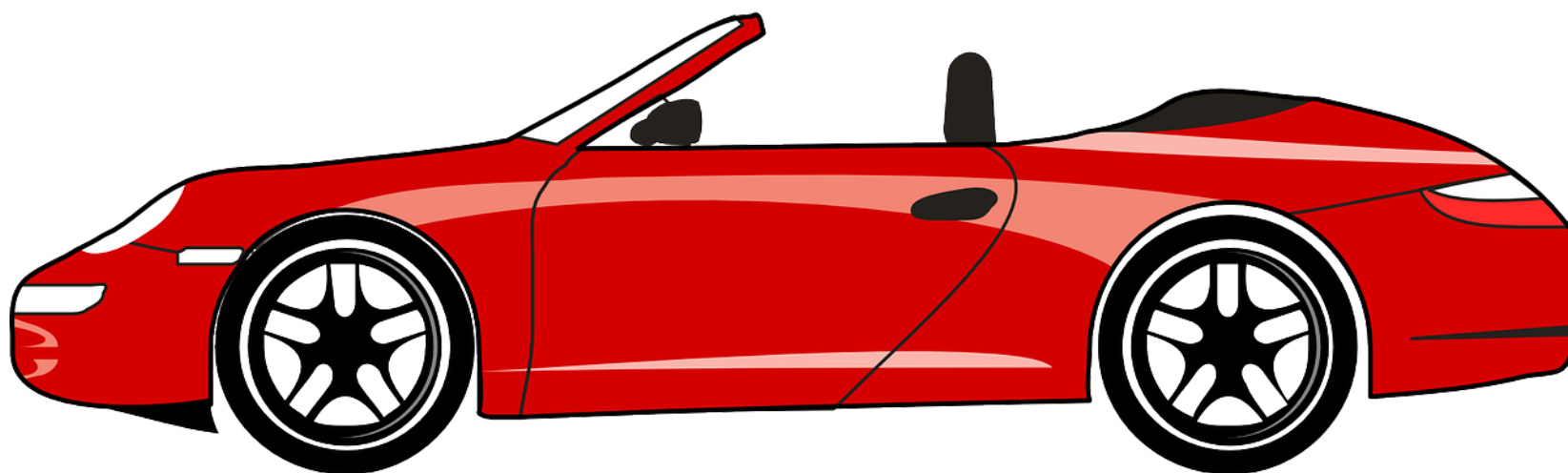
THROW И TRY..CATCH

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





СТРЕЛОЧНЫЕ ФУНКЦИИ

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;  
}
```

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;
```




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

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





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:

```
let 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*

```
let 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.

Синтаксис:

```
delete nameOfGlobalObjectProperty;  
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};  
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, 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

```
let array = ["first", "second", "third"];
console.log(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'];  
console.log(delete f.length);  
// -> false;  
console.log(f.length);  
// -> 3;
```

ВАЖНО ПОМНИТЬ, ЧТО DELETE ДОСТАТОЧНО МЕДЛЕННЫЙ ОПЕРАТОР

Delete - очень медленный инструмент.

Его использование не лишено смысла, если действительно требуется удалить свойство:

- вы работаете с большим количеством больших объектов;
- существование свойства ставит под угрозу корректность выполнения дальнейшего кода.

ДАВАЙТЕ УДАЛИМ НАШ КОНУС

> добавить в js/game.src.js:140

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




**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.

Да-да, функция - это тоже объект.

Как мы уже сегодня говорили, любой объект существует в оперативной памяти, пока есть какой либо указатель, указывающий на него.



ТАК ЧТО ЖЕ ТАКОЕ ЗАМЫКАНИЕ?

Это сохранение лексического окружения функции в оперативной памяти после её выполнения с помощью ссылки из объекта, создаваемого при её выполнении.

ТАК ЧТО ЖЕ ТАКОЕ ЗАМЫКАНИЕ?

Например:

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

Для большей наглядности создадим два объекта:

```
powerOfTwo1 = powerOfTwo();  
powerOfTwo2 = powerOfTwo();
```

И несколько раз их вызовем:

```
console.log(powerOfTwo1());  
console.log(powerOfTwo1());  
console.log(powerOfTwo2());  
console.log(powerOfTwo1());  
console.log(powerOfTwo2());
```

```
// -> 2  
// -> 4  
// -> 2  
// -> 8  
// -> 4
```

НЕТ НИКАКОЙ МАГИИ!

`powerOfTwo` - функция, которая возвращает функцию:

```
return function() {  
    power *= 2;  
    return power;  
}
```

При выполнении команды

```
powerOfTwo1 = powerOfTwo();
```

произошло выполнение `powerOfTwo()` с сохранением её результата в `powerOfTwo1`.

В `powerOfTwo1` записана функция.

При выполнении `powerOfTwo()` в переменную `power` было записано значение 1.

Логично, что `power` должна быть удалена после выполнения `powerOfTwo()`.

Но в результате `powerOfTwo()`:

```
power *= 2;  
return power;
```

используется `power`.

А значит, раз есть ссылка на `power` из `powerOfTwo1`, `power` удалена не будет, пока существует `powerOfTwo1`.

ЗАМЫКАНИЕ

Таким образом, функция `powerOfTwo1` *замыкает* на себя переменную `power` из лексического окружения отработавшей функции `powerOfTwo`

НАШЕ ЗАМЫКАНИЕ

```
removeConeWrapper() {  
  let last = 0;  
  this.removeCone = (key) => {  
    this.cones[key].remove();  
    delete this.cones[key];  
    last = key;  
  };  
  this.getLastConeNumber => last;  
},
```



ЗАЧЕМ НУЖНЫ ЗАМЫКАНИЯ?

- Замыкания позволяют сохранять данные функции между ее вызовами;
- Замыкания позволяют создавать функции, генерирующие функции;
- Замыкания позволяют ограничить область видимости переменных.



ИТАК, ПОДВЕДЕМ ИТОГИ

Сегодня нами были освоены новые инструменты:

1. перехват ошибок
2. стрелочная функция
3. удаление свойства объекта
4. замыкание

ИНТЕРЕСНОЕ ЧТИВО:

Перехват ошибки:

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

Удаление переменной:

- [MDN - delete](#)
- [Perfection Kills - Understanding delete](#)

Стрелочные функции:

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

Замыкания:

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



COPYRIGHT

Все изображения (за исключением конуса и взрыва из репозитория и эмблемы JS) взяты с ресурса [pixabay](https://pixabay.com/) и распространяются по лицензии [CC0](https://creativecommons.org/licenses/by/4.0/).

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





АЛЕКСАНДР ШЛЕЙКО



https://telegram.me/dustyo_0



<https://vk.com/shleiko>