

Раздел

[Обработка ошибок](#)

Навигация по уроку

Синтаксис «try...catch»

Объект ошибки

Блок «catch» без переменной

Использование «try...catch»

Генерация собственных ошибок

Проброс исключения

try...catch...finally

Глобальный catch

Итого

Задачи (1)

Комментарии

Поделиться

[Редактировать на GitHub](#)[→ Язык программирования JavaScript](#)
[→ Обработка ошибок](#)

7-го июля 2020

Обработка ошибок, "try..catch"

Неважно, насколько мы хороши в программировании, иногда наши скрипты содержат ошибки. Они могут возникать из-за наших промахов, неожиданного ввода пользователя, неправильного ответа сервера и по тысяче других причин.

Обычно скрипт в случае ошибки «падает» (сразу же останавливается), с выводом ошибки в консоль.

Но есть синтаксическая конструкция `try..catch`, которая позволяет «ловить» ошибки и вместо падения делать что-то более осмысленное.

Синтаксис «try...catch»

Конструкция `try..catch` состоит из двух основных блоков: `try`, и затем `catch`:

```
1 try {  
2  
3   // код...  
4  
5 } catch (err) {  
6  
7   // обработка ошибки  
8  
9 }
```

Работает она так:

1. Сначала выполняется код внутри блока `try {...}`.
2. Если в нём нет ошибок, то блок `catch(err)` игнорируется: выполнение доходит до конца `try` и потом далее, полностью пропуская `catch`.
3. Если же в нём возникает ошибка, то выполнение `try` прерывается, и поток управления переходит в начало `catch(err)`. Переменная `err` (можно использовать любое имя) содержит объект ошибки с подробной информацией о произошедшем.



Таким образом, при ошибке в блоке `try {...}` скрипт не «падает», и мы получаем возможность обработать ошибку внутри `catch`.

Давайте рассмотрим примеры.

- Пример без ошибок: выведет `alert (1)` и `(2)`:

Раздел

[Обработка ошибок](#)

Навигация по уроку

Синтаксис «try...catch»

Объект ошибки

Блок «catch» без переменной

Использование «try...catch»

Генерация собственных ошибок

Проброс исключения

try...catch...finally

Глобальный catch

Итого

Задачи (1)

Комментарии

Поделиться



Редактировать на GitHub



```
1 try {
2
3   alert('Начало блока try'); // (1) <--
4
5   // ...код без ошибок
6
7   alert('Конец блока try'); // (2) <--
8
9 } catch(err) {
10
11   alert('Catch игнорируется, так как нет ошибок'); //
12
13 }
```

- Пример с ошибками: выведет (1) и (3):

```
1 try {
2
3   alert('Начало блока try'); // (1) <--
4
5   lalala; // ошибка, переменная не определена!
6
7   alert('Конец блока try (никогда не выполнится)');
8
9 } catch(err) {
10
11   alert('Возникла ошибка!'); // (3) <--
12
13 }
```



⚠ try...catch работает только для ошибок, возникающих во время выполнения кода

Чтобы try...catch работал, код должен быть выполнимым. Другими словами, это должен быть корректный JavaScript-код.

Он не сработает, если код синтаксически неверен, например, содержит несовпадающее количество фигурных скобок:

```
1 try {
2   {}{}{}{}{}{}{}{}
3 } catch(e) {
4   alert("Движок не может понять этот код, он некорректен");
5 }
```

JavaScript-движок сначала читает код, а затем исполняет его.

Ошибки, которые возникают во время фазы чтения, называются ошибками парсинга. Их нельзя обработать (изнутри этого кода), потому что движок не понимает код.

Таким образом, try...catch может обрабатывать только ошибки, которые возникают в корректном коде. Такие ошибки называют «ошибками во время выполнения», а иногда «исключениями».



Раздел

Обработка ошибок

Навигация по уроку

Синтаксис «try...catch»

Объект ошибки

Блок «catch» без переменной

Использование «try...catch»

Генерация собственных ошибок

Проброс исключения

try...catch...finally

Глобальный catch

Итого

Задачи (1)

Комментарии

Поделиться



Редактировать на GitHub



⚠ try..catch работает синхронно

Исключение, которое произойдёт в коде, запланированном «на будущее», например в `setTimeout`, `try..catch` не поймает:

```
1 try {
2   setTimeout(function() {
3     noSuchVariable; // скрипт упадёт тут
4   }, 1000);
5 } catch (e) {
6   alert( "не сработает" );
7 }
```

Это потому, что функция выполняется позже, когда движок уже покинул конструкцию `try..catch`.

Чтобы поймать исключение внутри запланированной функции, `try..catch` должен находиться внутри самой этой функции:

```
1 setTimeout(function() {
2   try {
3     noSuchVariable; // try..catch обрабатывает ош
4   } catch {
5     alert( "ошибка поймана!" );
6   }
7 }, 1000);
```

Объект ошибки

Когда возникает ошибка, JavaScript генерирует объект, содержащий её детали. Затем этот объект передаётся как аргумент в блок `catch`:

```
1 try {
2   // ...
3 } catch(err) { // <-- объект ошибки, можно использовать
4   // ...
5 }
```

Для всех встроенных ошибок этот объект имеет два основных свойства:

name

Имя ошибки. Например, для неопределённой переменной это `"ReferenceError"`.

message

Текстовое сообщение о деталях ошибки.

В большинстве окружений доступны и другие, нестандартные свойства. Одно из самых широко используемых и поддерживаемых – это:

stack

Текущий стек вызова: строка, содержащая информацию о последовательности вложенных вызовов, которые привели к ошибке. Используется в целях отладки.

Например:

```
1 try {
2   lalala; // ошибка, переменная не определена!
3 } catch(err) {
4   alert(err.name); // ReferenceError
5   alert(err.message); // lalala is not defined
6   alert(err.stack); // ReferenceError: lalala is not de
```

Раздел

Обработка ошибок

Навигация по уроку

Синтаксис «try...catch»

Объект ошибки

Блок «catch» без переменной

Использование «try...catch»

Генерация собственных ошибок

Проброс исключения

try...catch...finally

Глобальный catch

Итого

Задачи (1)

Комментарии

Поделиться



Редактировать на GitHub



Блок «catch» без переменной

⚠ Новая возможность

Эта возможность была добавлена в язык недавно. В старых браузерах может понадобиться полифил.

Если нам не нужны детали ошибки, в `catch` можно её пропустить:

```
1 try {
2   // ...
3 } catch { // <-- без (err)
4   // ...
5 }
```

Использование «try...catch»

Давайте рассмотрим реальные случаи использования `try...catch`.

Как мы уже знаем, JavaScript поддерживает метод `JSON.parse(str)` для чтения JSON.

Обычно он используется для декодирования данных, полученных по сети, от сервера или из другого источника.

Мы получаем их и вызываем `JSON.parse` вот так:

```
1 let json = '{"name":"John", "age": 30}'; // данные с сервера
2
3 let user = JSON.parse(json); // преобразовали текстовое
4
5 // теперь user - объект со свойствами из строки
6 alert( user.name ); // John
7 alert( user.age ); // 30
```

Вы можете найти более детальную информацию о JSON в главе [Формат JSON, метод toJSON](#).

Если `json` некорректен, `JSON.parse` генерирует ошибку, то есть скрипт «падает».

Устроит ли нас такое поведение? Конечно нет!

Получается, что если вдруг что-то не так с данными, то посетитель никогда (если, конечно, не откроет консоль) об этом не узнает. А люди очень не любят, когда что-то «просто падает» без всякого сообщения об ошибке.

Давайте используем `try...catch` для обработки ошибки:

```
1 let json = "{ некорректный JSON }";
2
3 try {
4
5   let user = JSON.parse(json); // <-- тут возникает оши
6   alert( user.name ); // не работает
7
8 } catch (e) {
9
10
11
12
```

Раздел

[Обработка ошибок](#)

Навигация по уроку

Синтаксис «try...catch»

Объект ошибки

Блок «catch» без переменной

Использование «try...catch»

Генерация собственных ошибок

Проброс исключения

try...catch...finally

Глобальный catch

Итого

Задачи (1)

Комментарии

Поделиться



[Редактировать на GitHub](#)



```
13 // ...выполнение прыгает сюда
    alert( "Извините, в данных ошибка, мы попробуем получ
    alert( e.name );
    alert( e.message );
}
```

Здесь мы используем блок `catch` только для вывода сообщения, но мы также можем сделать гораздо больше: отправить новый сетевой запрос, предложить посетителю альтернативный способ, отослать информацию об ошибке на сервер для логирования, ... Всё лучше, чем просто «падение».

Генерация собственных ошибок

Что если `json` синтаксически корректен, но не содержит необходимого свойства `name` ?

Например, так:

```
1 let json = '{ "age": 30 }'; // данные неполны
2
3 try {
4
5     let user = JSON.parse(json); // <-- выполнится без ош
6     alert( user.name ); // нет свойства name!
7
8 } catch (e) {
9     alert( "не выполнится" );
10 }
```

Здесь `JSON.parse` выполнится без ошибок, но на самом деле отсутствие свойства `name` для нас ошибка.

Для того, чтобы унифицировать обработку ошибок, мы воспользуемся оператором `throw`.

Оператор «throw»

Оператор `throw` генерирует ошибку.

Синтаксис:

```
1 throw <объект ошибки>
```

Технически в качестве объекта ошибки можно передать что угодно. Это может быть даже примитив, число или строка, но всё же лучше, чтобы это был объект, желательно со свойствами `name` и `message` (для совместимости со встроенными ошибками).

В JavaScript есть множество встроенных конструкторов для стандартных ошибок: `Error`, `SyntaxError`, `ReferenceError`, `TypeError` и другие. Можно использовать и их для создания объектов ошибки.

Их синтаксис:

```
1 let error = new Error(message);
2 // или
3 let error = new SyntaxError(message);
4 let error = new ReferenceError(message);
5 // ...
```

Для встроенных ошибок (не для любых объектов, только для ошибок), свойство `name` — это в точности имя конструктора. А свойство `message` берётся из аргумента.

Например:

```
1 let error = new Error(" Ого, ошибка! o_0");
```

Раздел

Обработка ошибок

Навигация по уроку

Синтаксис «try...catch»

Объект ошибки

Блок «catch» без переменной

Использование «try...catch»

Генерация собственных ошибок

Проброс исключения

try...catch...finally

Глобальный catch

Итого

Задачи (1)

Комментарии

Поделиться



Редактировать на GitHub



Давайте посмотрим, какую ошибку генерирует `JSON.parse` :

```
1 try {
2   JSON.parse("{ bad json o_0 }");
3 } catch(e) {
4   alert(e.name); // SyntaxError
5   alert(e.message); // Unexpected token b in JSON at po
6 }
```

Как мы видим, это `SyntaxError` .

В нашем случае отсутствие свойства `name` – это ошибка, ведь пользователи должны иметь имена.

Сгенерируем её:

```
1 let json = '{ "age": 30 }'; // данные неполны
2
3 try {
4
5   let user = JSON.parse(json); // <-- выполнится без ош
6
7   if (!user.name) {
8     throw new SyntaxError("Данные неполны: нет имени");
9   }
10
11   alert( user.name );
12
13 } catch(e) {
14   alert( "JSON Error: " + e.message ); // JSON Error: Д
15 }
```

В строке (*) оператор `throw` генерирует ошибку `SyntaxError` с сообщением `message` . Точно такого же вида, как генерирует сам JavaScript. Выполнение блока `try` немедленно останавливается, и поток управления прыгает в `catch` .

Теперь блок `catch` становится единственным местом для обработки всех ошибок: и для `JSON.parse` и для других случаев.

Проброс исключения

В примере выше мы использовали `try...catch` для обработки некорректных данных. А что, если в блоке `try {...}` возникнет *другая неожиданная ошибка*? Например, программная (неопределённая переменная) или какая-то ещё, а не ошибка, связанная с некорректными данными.

Пример:

```
1 let json = '{ "age": 30 }'; // данные неполны
2
3 try {
4   user = JSON.parse(json); // <-- забыл добавить "let"
5
6   // ...
7 } catch(err) {
8   alert("JSON Error: " + err); // JSON Error: Reference
9   // (не JSON ошибка на самом деле)
10 }
```

Конечно, возможно все! Программисты совершают ошибки. Даже в утилитах с открытым исходным кодом, используемых миллионами людей

Раздел

[Обработка ошибок](#)

Навигация по уроку

Синтаксис «try...catch»

Объект ошибки

Блок «catch» без переменной

Использование «try...catch»

Генерация собственных ошибок

Проброс исключения

try...catch...finally

Глобальный catch

Итого

Задачи (1)

Комментарии

Поделиться



Редактировать на GitHub



на протяжении десятилетий – вдруг может быть обнаружена ошибка, которая приводит к ужасным взломам.

В нашем случае `try...catch` предназначен для выявления ошибок, связанных с некорректными данными. Но по своей природе `catch` получает все свои ошибки из `try`. Здесь он получает неожиданную ошибку, но всё также показывает то же самое сообщение `"JSON Error"`. Это неправильно и затрудняет отладку кода.

К счастью, мы можем выяснить, какую ошибку мы получили, например, по её свойству `name`:

```
1 try {
2   user = { /*...*/ };
3 } catch(e) {
4   alert(e.name); // "ReferenceError" из-за неопределённ
5 }
```

Есть простое правило:

Блок `catch` должен обрабатывать только те ошибки, которые ему известны, и «пробрасывать» все остальные.

Техника «проброс исключения» выглядит так:

1. Блок `catch` получает все ошибки.
2. В блоке `catch(err) {...}` мы анализируем объект ошибки `err`.
3. Если мы не знаем как её обработать, тогда делаем `throw err`.

В коде ниже мы используем проброс исключения, `catch` обрабатывает только `SyntaxError`:

```
1 let json = '{ "age": 30 }'; // данные неполны
2 try {
3
4   let user = JSON.parse(json);
5
6   if (!user.name) {
7     throw new SyntaxError("Данные неполны: нет имени");
8   }
9
10  blabla(); // неожиданная ошибка
11
12  alert( user.name );
13
14 } catch(e) {
15
16   if (e.name == "SyntaxError") {
17     alert( "JSON Error: " + e.message );
18   } else {
19     throw e; // проброс (*)
20   }
21
22 }
```

Ошибка в строке `(*)` из блока `catch` «выпадает наружу» и может быть поймана другой внешней конструкцией `try...catch` (если есть), или «убьёт» скрипт.

Таким образом, блок `catch` фактически обрабатывает только те ошибки, с которыми он знает, как справляться, и пропускает остальные.

Пример ниже демонстрирует, как такие ошибки могут быть пойманы с помощью ещё одного уровня `try...catch`:

```
1 function readData() {
2   let json = '{ "age": 30 }';
3
4   try {
5     // ...
```

Раздел

[Обработка ошибок](#)

Навигация по уроку

Синтаксис «try...catch»

Объект ошибки

Блок «catch» без переменной

Использование «try...catch»

Генерация собственных ошибок

Проброс исключения

try...catch...finally

Глобальный catch

Итого

Задачи (1)

Комментарии

Поделиться



[Редактировать на GitHub](#)



```
6     blabla(); // ошибка!
7   } catch (e) {
8     // ...
9     if (e.name !== 'SyntaxError') {
10      throw e; // проброс исключения (не знаю как это о
11    }
12  }
13 }
14
15 try {
16   readData();
17 } catch (e) {
18   alert( "Внешний catch поймал: " + e ); // поймал!
19 }
```

Здесь `readData` знает только, как обработать `SyntaxError`, тогда как внешний блок `try..catch` знает, как обработать всё.

try...catch...finally

Подождите, это ещё не всё.

Конструкция `try..catch` может содержать ещё одну секцию: `finally`.

Если секция есть, то она выполняется в любом случае:

- после `try`, если не было ошибок,
- после `catch`, если ошибки были.

Расширенный синтаксис выглядит следующим образом:

```
1 try {
2   ... пробуем выполнить код...
3 } catch(e) {
4   ... обрабатываем ошибки ...
5 } finally {
6   ... выполняем всегда ...
7 }
```

Попробуйте запустить такой код:

```
1 try {
2   alert( 'try' );
3   if (confirm( 'Сгенерировать ошибку?' )) BAD_CODE();
4 } catch (e) {
5   alert( 'catch' );
6 } finally {
7   alert( 'finally' );
8 }
```

У кода есть два пути выполнения:

1. Если вы ответите на вопрос «Сгенерировать ошибку?» утвердительно, то `try -> catch -> finally`.
2. Если ответите отрицательно, то `try -> finally`.

Секцию `finally` часто используют, когда мы начали что-то делать и хотим завершить это вне зависимости от того, будет ошибка или нет.

Например, мы хотим измерить время, которое занимает функция чисел Фибоначчи `fib(n)`. Естественно, мы можем начать измерения до того, как функция начнёт выполняться и закончить после. Но что делать, если при вызове функции возникла ошибка? В частности, реализация `fib(n)` в коде ниже возвращает ошибку для отрицательных и для нецелых чисел.

Секция `finally` отлично подходит для завершения измерений несмотря ни на что.

Здесь `finally` гарантирует, что время будет измерено корректно в обеих ситуациях – и в случае успешного завершения `fib` и в случае ошибки:

Раздел

Обработка ошибок

Навигация по уроку

Синтаксис «try...catch»

Объект ошибки

Блок «catch» без переменной

Использование «try...catch»

Генерация собственных ошибок

Проброс исключения

try...catch...finally

Глобальный catch

Итого

Задачи (1)

Комментарии

Поделиться



Редактировать на GitHub



```
1 let num = +prompt("Введите положительное целое число?"),
2
3 let diff, result;
4
5 function fib(n) {
6   if (n < 0 || Math.trunc(n) !== n) {
7     throw new Error("Должно быть целое неотрицательное
8   }
9   return n <= 1 ? n : fib(n - 1) + fib(n - 2);
10 }
11
12 let start = Date.now();
13
14 try {
15   result = fib(num);
16 } catch (e) {
17   result = 0;
18 } finally {
19   diff = Date.now() - start;
20 }
21
22 alert(result || "возникла ошибка");
23
24 alert(`Выполнение заняло ${diff}ms`);
```

Вы можете это проверить, запустив этот код и введя 35 в prompt – код завершится нормально, finally выполнится после try. А затем введите -1 – незамедлительно произойдёт ошибка, выполнение займёт 0ms. Оба измерения выполняются корректно.

Другими словами, неважно как завершилась функция: через return или throw. Секция finally срабатывает в обоих случаях.



i Переменные внутри try..catch..finally локальны

Обратите внимание, что переменные result и diff в коде выше объявлены до try..catch.

Если переменную объявить в блоке, например, в try, то она не будет доступна после него.



i finally и return

Блок finally срабатывает при *любом* выходе из try..catch, в том числе и return.

В примере ниже из try происходит return, но finally получает управление до того, как контроль возвращается во внешний код.

```
1 function func() {
2
3   try {
4     return 1;
5
6   } catch (e) {
7     /* ... */
8   } finally {
9     alert( 'finally' );
10   }
11 }
12
13 alert( func() ); // сначала срабатывает alert из 1
```

Раздел

Обработка ошибок

Навигация по уроку

Синтаксис «try...catch»

Объект ошибки

Блок «catch» без переменной

Использование «try...catch»

Генерация собственных ошибок

Проброс исключения

try...catch...finally

Глобальный catch

Итого

Задачи (1)

Комментарии

Поделиться



Редактировать на GitHub



try..finally

Конструкция try..finally без секции catch также полезна. Мы применяем её, когда не хотим здесь обрабатывать ошибки (пусть выпадут), но хотим быть уверены, что начатые процессы завершились.

```
1 function func() {
2     // начать делать что-то, что требует завершения
3     try {
4         // ...
5     } finally {
6         // завершить это, даже если все упадёт
7     }
8 }
```

В приведённом выше коде ошибка всегда выпадает наружу, потому что тут нет блока catch. Но finally отработает до того, как поток управления выйдет из функции.

Глобальный catch

Зависит от окружения

Информация из данной секции не является частью языка JavaScript.

Давайте представим, что произошла фатальная ошибка (программная или что-то ещё ужасное) снаружи try..catch, и скрипт упал.

Существует ли способ отреагировать на такие ситуации? Мы можем захотеть залогировать ошибку, показать что-то пользователю (обычно они не видят сообщение об ошибке) и т.д.

Такого способа нет в спецификации, но обычно окружения предоставляют его, потому что это весьма полезно. Например, в Node.js для этого есть `process.on("uncaughtException")`. А в браузере мы можем присвоить функцию специальному свойству `window.onerror`, которая будет вызвана в случае необработанной ошибки.

Синтаксис:

```
1 window.onerror = function(message, url, line, col, erro
2     // ...
3 };
```

message

Сообщение об ошибке.

url

URL скрипта, в котором произошла ошибка.

line, col

Номера строки и столбца, в которых произошла ошибка.

error

Объект ошибки.

Пример:

```
1 <script>
2     window.onerror = function(message, url, line, col, er
3         alert(`${message}\n B ${line}:${col} на ${url}`);
4     };
```

Раздел

Обработка ошибок

Навигация по уроку

Синтаксис «try...catch»

Объект ошибки

Блок «catch» без переменной

Использование «try...catch»

Генерация собственных ошибок

Проброс исключения

try...catch...finally

Глобальный catch

Итого

Задачи (1)

Комментарии

Поделиться



Редактировать на GitHub



```
5
6   function readData() {
7       badFunc(); // Ой, что-то пошло не так!
8   }
9
10  readData();
11 </script>
```

Роль глобального обработчика `window.onerror` обычно заключается не в восстановлении выполнения скрипта – это скорее всего невозможно в случае программной ошибки, а в отправке сообщения об ошибке разработчикам.

Существуют также веб-сервисы, которые предоставляют логирование ошибок для таких случаев, такие как <https://errorception.com> или <http://www.muscula.com>.

Они работают так:

1. Мы регистрируемся в сервисе и получаем небольшой JS-скрипт (или URL скрипта) от них для вставки на страницы.
2. Этот JS-скрипт ставит свою функцию `window.onerror`.
3. Когда возникает ошибка, она выполняется и отправляет сетевой запрос с информацией о ней в сервис.
4. Мы можем войти в веб-интерфейс сервиса и увидеть ошибки.

Итого

Конструкция `try...catch` позволяет обрабатывать ошибки во время исполнения кода. Она позволяет запустить код и перехватить ошибки, которые могут в нём возникнуть.

Синтаксис:

```
1 try {
2     // исполняем код
3 } catch(err) {
4     // если случилась ошибка, прыгаем сюда
5     // err - это объект ошибки
6 } finally {
7     // выполняется всегда после try/catch
8 }
```

Секций `catch` или `finally` может не быть, то есть более короткие конструкции `try...catch` и `try...finally` также корректны.

Объекты ошибок содержат следующие свойства:

- `message` – понятное человеку сообщение.
- `name` – строка с именем ошибки (имя конструктора ошибки).
- `stack` (нестандартное, но хорошо поддерживается) – стек на момент ошибки.

Если объект ошибки не нужен, мы можем пропустить его, используя `catch { вместо catch(err) {`.

Мы можем также генерировать собственные ошибки, используя оператор `throw`. Аргументом `throw` может быть что угодно, но обычно это объект ошибки, наследуемый от встроенного класса `Error`. Подробнее о расширении ошибок см. в следующей главе.

Проброс исключения – это очень важный приём обработки ошибок: блок `catch` обычно ожидает и знает, как обработать определённый тип ошибок, поэтому он должен пробрасывать дальше ошибки, о которых он не знает.

Даже если у нас нет `try...catch`, большинство сред позволяют настроить «глобальный» обработчик ошибок, чтобы ловить ошибки, которые «выпадают наружу». В браузере это `window.onerror`.

✓ Задачи

Finally или просто код?

важность: 5

Сравните два фрагмента кода.

1.

Первый использует `finally` для выполнения кода после `try..catch`:

```
1 try {
2   начать работу
3   работать
4 } catch (e) {
5   обработать ошибку
6 } finally {
7   очистить рабочее пространство
8 }
```

2.

Второй фрагмент просто ставит очистку после `try..catch`:

```
1 try {
2   начать работу
3   работать
4 } catch (e) {
5   обработать ошибку
6 }
7
8 очистить рабочее пространство
```

Нам определённо нужна очистка после работы, неважно возникли ошибки или нет.

Есть ли здесь преимущество в использовании `finally` или оба фрагмента кода одинаковы? Если такое преимущество есть, то дайте пример, когда оно проявляется.

решение

Проводим [курсы по JavaScript и фреймворкам](#). 

Комментарии

перед тем как писать...

Раздел

[Обработка ошибок](#)

Навигация по уроку

Синтаксис «try...catch»

Объект ошибки

Блок «catch» без переменной

Использование «try...catch»

Генерация собственных ошибок

Проброс исключения

try...catch...finally

Глобальный catch

Итого

Задачи (1)

Комментарии

Поделиться



Редактировать на GitHub

 Рекомендовать 5 Твитнуть Поделиться

Лучшее в начале ▾



Присоединиться к обсуждению...

войти с помощью

или через DISQUS [Загрузить ещё комментарии](#) Подписаться [Добавь Disqus на свой сайт](#) [Добавить Disqus](#) [Добавить](#) Do Not Sell My Data

Раздел

[Обработка ошибок](#)

Навигация по уроку

Синтаксис «try...catch»

Объект ошибки

Блок «catch» без переменной

Использование «try...catch»

Генерация собственных
ошибок

Проброс исключения

try...catch...finally

Глобальный catch

Итого

Задачи (1)

Комментарии

Поделиться

[Редактировать на GitHub](#)