СОДЕРЖАНИЕ

[1. Ошибка и исключение. Разница 3](#_Toc159155843)

[1.1. Ошибка. 3](#_Toc159155844)

[1.2. Исключение. 4](#_Toc159155845)

[1.3. Отличие return от throw 5](#_Toc159155846)

[2. new Error 7](#_Toc159155847)

[3. Подклассы Error 8](#_Toc159155848)

[4. Создание своих типов ошибок 9](#_Toc159155849)

[5. Finally 10](#_Toc159155850)

[6. Асинхронность и обработка ошибок 12](#_Toc159155851)

[7. Альтернативы try catch 13](#_Toc159155852)

[8. Обработка непойманных async исключений, через события 15](#_Toc159155853)

[9. Обработка ошибок через события 16](#_Toc159155854)

# Ошибка и исключение. Разница

## ­Ошибка.

Сначала необходимо конкретизировать, что под ошибкой подразумевается не ошибка логики (например, вместо +, сделали -), а ошибка именно с точки зрения ЯП.

Например:

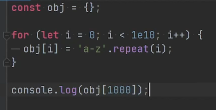


1. Пример ошибки несоблюдения синтаксических контрактов.

В итоге, если мы попытаемся запустить, возникнет именно не отлавливаемая ошибка! То есть мы не можем ее обработать. У нас падает весь процесс. То есть ошибка — это когда по каким-то причинам программа не может быть выполнена, не может дальше продолжаться. Они ловятся именно во время компиляции кода, а не во время рантайма.

Также есть второй тип ошибок - программа требует больше памяти чем есть. Процесс упадет с ошибкой. Out of memory

Например:



1. Пример ошибки Out of memory

## Исключение.

Часто на работе разработчик может сказать "ошибка", имея ввиду исключение.

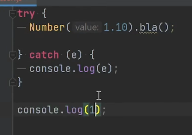
Исключение — это то, что можно обработать. То есть мы можем программно из нашего кода ее обработать.

Например:



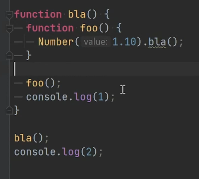
1. Пример исключения.

Но мы можем ее обработать:



1. Обработка исключений.

Исключение — это чисто runtime история. (95%)



1. Пример второй.

Здесь ни один лог не будет вызван, то есть программа остановится.



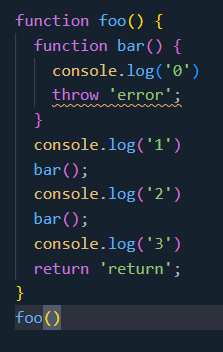
1. Обработка.

Но если обработать вызов этой функции, то лог 1 и 2 все же будет вызыван.

Но если в catch мы опять вызовем foo(). Да, ошибка была перехвачена, но в блоке обработки ошибок опять произошло исключение, но его никто не перехватывал.

## Отличие return от throw

Throw себя ведет более радикально, нежели return. Return завершает выполнение текущей ф-ции каким-то значением. А throw завершает выполнение текущего блока, дальше распространяется на родительский блок итд, летит наверх, пока где-то не встретиться try catch. Throw также может использовать как оператор go to.



1. Пример завершения ф-ций через throw.

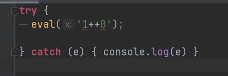
# new Error

* message — это то, что мы передали в конструктор Error,
* name - название самого конструктора,
* stack - вывод инфы зависит от браузера. Смысл источник исключения, строка, какая позиция.

# Подклассы Error

* SyntaxError - синтаксическая ошибка. Например когда пишем парсер, во-вторых, есть конструкция динамического выполнения кода, например строку преобразовывать в js код и там тоже могут быть синтаксические ошибки.

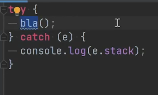
Пример:



1. Пример ошибки SyntaxError

* ReferenceError - ошибки невозможности разрешения некоторой ссылки.

Пример:



1. ReferenceError пример ошибки.

ReferenceError: bla is not defined.

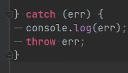
* TypeError - такие ошибки возникают, когда мы ф-ции передаем какое-то значение, а оно с этим типом работать не может.
* EvalError - блокировка выполнения eval() (см. Рисунок 8).
* RangeError - когда мы работаем с числами и происходит переполнение числа. Оно больше не может расширяться.
* InternalError - внутренняя ошибка виртуальной машины, возникает при переполнении стека, и различных таких ситуаций.

# Создание своих типов ошибок

См. код

Исходя из кода и недостатков, работа с исключениями обычно рекомендуется на уровне интеграции с различными API. Например веб сервер и он принимает запросы, и он сигнализирует об ошибке, в этом случае используем исключения.

Однако если код предметного уровня, там значение не найдено, то работать надо без исключений.

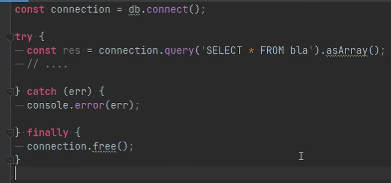


1. Допустимый паттерн

# Finally

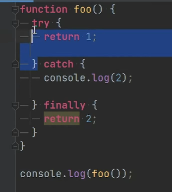
Бывает необходимость освобождения ресурсов после и try и catch, то есть в обоих случаях.

Пример:



1. Пример блока finally.

Но есть нюансы:



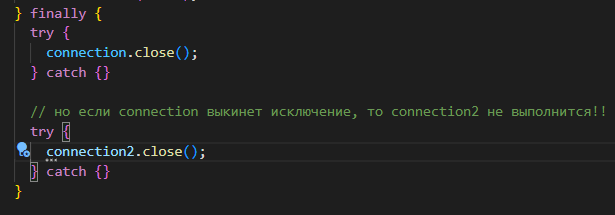
1. Return в try и finally

Что будет в итоге? В данном случае finally блок все равно выполняется и победит return из finally. Но такой код не надо писать.

Также мы можем не писать catch блок:



1. Без catch, вывод: 2



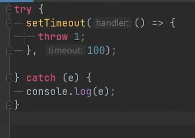
1. Проблема reject в finally

# Асинхронность и обработка ошибок

JS встраиваемый язык и он не предполагает какой-либо концепции асинхронности. В стандарте языка нет возможности создать поток (можно в NodeJS и web workers).

И именно СРЕДА определяет, что там асинхронно и нет. В NodeJS libuv, в браузере свой движок.

Пример:



1. setTimeout внутри try catch.

В таком случае мы получим не отловленное исключение:



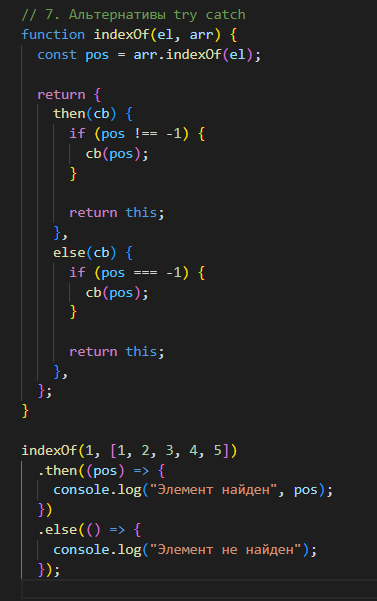
1. Не отловленное исключение.

Как его отловить?



1. Отлов ошибок

# Альтернативы try catch

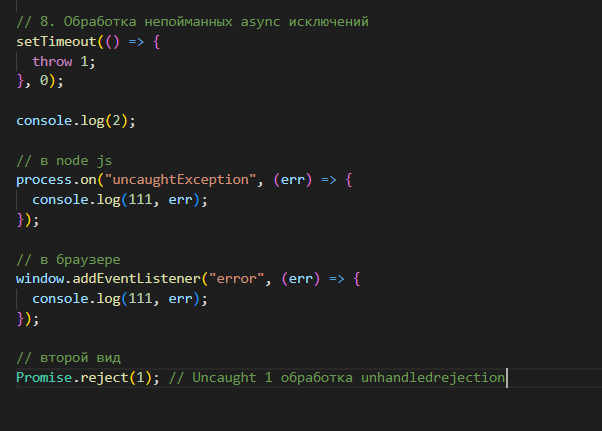


1. Then else

Этот способ лучше всех предыдущих, во-первых, мы на 100% знаем, что ф-ция может вернуть некоторую исключительную ситуацию. Потому что возвращаемое значение ф-ции не содержит в себе чистые данные, оно содержит данные, упакованные в специальную структуру, которая этим данным добавляет некоторый контекст, например, что их может не быть. В данным случае - найден / не найден.

При проектировании try catch ключевая ошибка заключается в том, что у нас действительно есть ошибки, которые должны обязаны останавливать программу (syntax or oom и прочие) и мы не должны пытаться их обработать, а вот исключение, оно и не должно работать как ошибки. В этом проблема. Оберточные типы решают эту проблему.

# Обработка непойманных async исключений, через события



1. Решение проблемы

# Обработка ошибок через события

Такой подход иногда лучше промисов. Чем он лучше? Лучше в том плане, что если обработка ошибок не какой то определенной ф-ции, а, например есть компонент, у которого множество эл. Могут привести к какой то нештатной ситуации, и мы хотим единую точку обработки ошибок.

Вот здесь правильно, чтобы ошибки генерировали события и эти события на каком-то базовом уровне компонента обрабатывались.

EventEmitter - это механизм, который позволяет реализовывать шаблон проектирования "Observer" (наблюдатель) в JavaScript.