СОДЕРЖАНИЕ

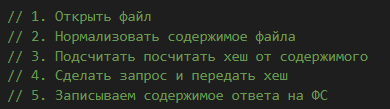
[1. Проблема 3](#_Toc158912094)

[2. Promise 5](#_Toc158912095)

[3. Статические методы 7](#_Toc158912096)

# Проблема

(см. файл promises.js)



1. Задача



1. Ф-ция doWork

Она запускает пять асинхронных событий друг за другом. Асинхронные действия не гарантируют, когда они должны быть выполнены, но нам важен порядок их выполнения.

Выглядит это не очень, спагетти код, сложно читается, сложно поддерживается, бешенные вложенности, синтаксический шум.

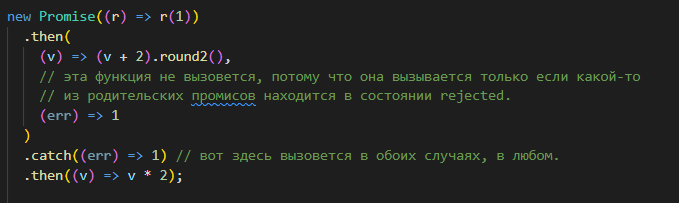
# Promise

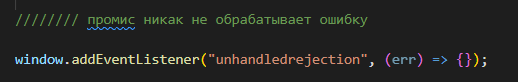
(см. файл promises.js)

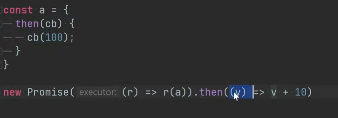
Promise — это некоторая структура данных, которая позволяет поковать "сырые" данные в специальный контейнер, который подразумевает, что этих данных может и не быть, что они еще не загрузились.

* Есть три статуса pending, fulfilled, rejected. У pending нет каких-то специальных обработчиков. Можно вызвать только на fulfilled -> then finally или rejected -> then catch finally.
* У then есть второй аргумент, можно передать функцию, которая вызовется, если родитель rejected, но не сам then.
* Также важно понимать, что все эти методы тоже вернут promise, new Promise всегда возвращает промис, async function всегда возвращает промис.
* Если промис по каким то причинам перешел в состояние rejected, то будет автоматически создано исключение, след промис rejected. След then не будут вызваны.

Разница между catch и вторым параметром then.







1. Thenебел объект

Finally

Не важно что из finally возвращаем, все что угодно, но finally сохранит состояние родительского промиса.

Также если родительский промис rejected, finally не переведет его в состояние fulfilled.

# Статические методы

* Promise.resolve() - создает новый промис из какого-то значения, по сути это аналог Number('12') попытается преобразовать в number. То есть если то, что мы ему передали, не явл промисом, то он из этого что-то создает промис. Но если мы передали thneble объект, то создастся новый промис, который будет учитывать этот объект.

Нет логики с авто реджект.

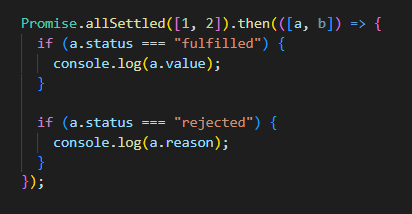
* Promise.all([]) - принимает iterable значение, например массив или new Set. Это могут быть обычные значение или какие-то промисы.

Возвращает новый промис, который будет resolve, если все внутри будет resolve.

А если какой то промис reject, то он тоже reject. Короче аналог Array.every.

Значение, которое не промис, будет неявным образом сделано промисом.

* Promise.allSettled - всегда fulfilled, но он получает не массив чистых распакованных значений, а массив специальных объектов. Например [1, 2] будет не 1 и 2, а объект:



Then выполнится тогда, когда все значения либо fulfilled, либо rejected.

* Promise.race() - получает iterable, например массив. И resolve или reject с тем значением, которое пришло самое первое. Например: запустим 5 запросов к серверу и самый первый будет результатом нового промиса. А reject, только в том, случае если хотя бы один reject.

Самая популярная задача: выполнить какую-нибудь задачу и если она выполняется слишком долго, сделать reject:



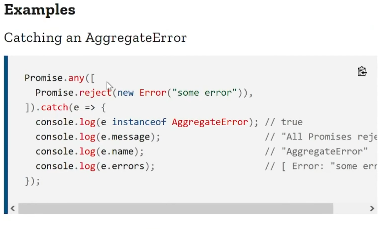
То есть если doAction выполнится раньше, будет race будет fulfilled, в противном случае всегда reject через 200мс.

* Promise.any() - принимает interable, но возвращаемый промис, который resolve, он будет fulfilled, когда хотя бы один из промисов. А reject, только в том, случае если ВСЕ reject.

Задача: проверить, если ли подключение к интернету. Может произойти ситуация, что доступ к интернету есть, но сам сервер лежит (ddos) например.

Поэтому можно сделать так: можно сделать несколько запросов к разным сервисам, например Yandex, google, bingo за фавиконками и когда первый прибежал, то все у нас есть интернет.

ОЧЕНЬ ВАЖНО!!!



1. Когда у нас reject все промисы внутри all,

То ошибка имеет специальный под класс AggregateError. e содердит внутри себя message, name и errors, которая содержит все ошибки reject промисов.