hillel IT school

Lesson 22

План заняття

- Вступ до Promises;
- Стан promise;
- then, catch, finaly;

Promis

Синтаксис

```
const promise = new Promise(function(resolve, reject) {
  // return result of promise
});
```

Функція передана в new Promise називається виконавцем. Коли створюється new Promise вона виконується автоматично. В ній знаходиться код “виробник” котрий зрештою поверне результат.

resolve і reject – це колбеки які надає нам сам JavaScript. Наш код – тільки всередині виконавця.

Коли функція-виконавець завершить свою роботу, неважливо – зараз чи пізніше, вона повинна викликати один з цих колбеків:

- resolve(value) – якщо код успішно виконався, з результатом value.
- reject(error) – якщо виникла помилка, error – об’єкт помилки.

Promis

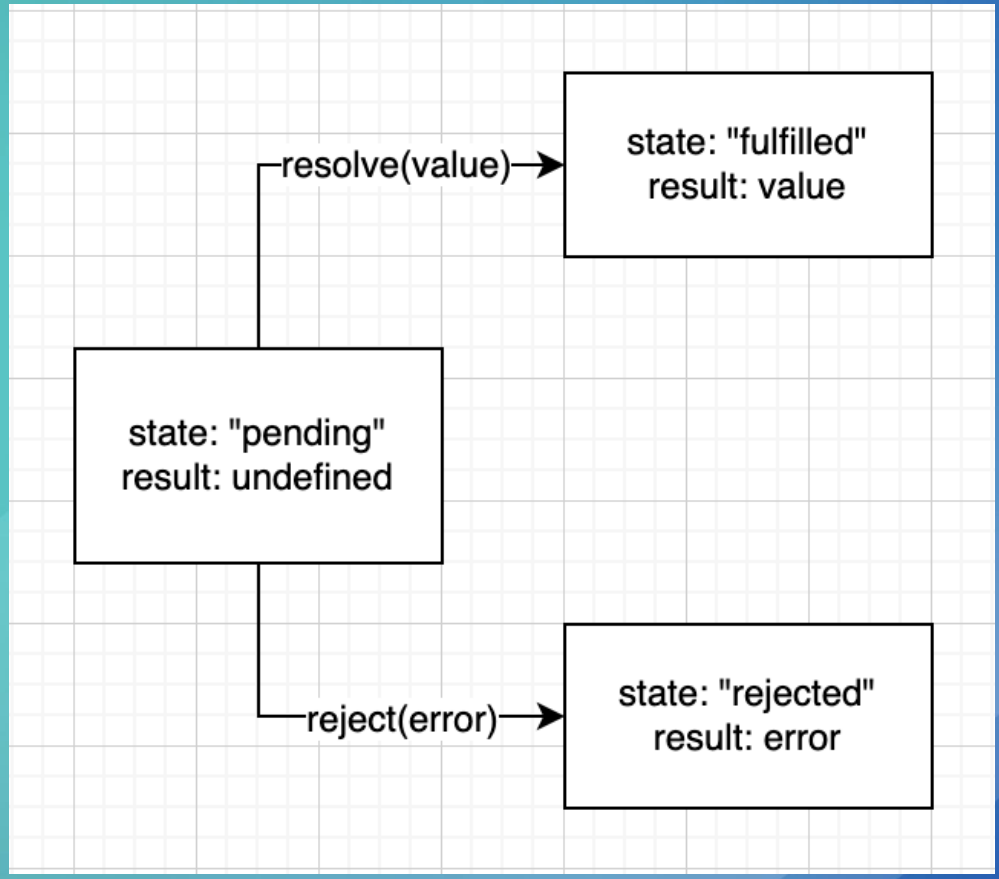
В об’єкта promise, що повертається конструктором new Promise є внутрішні властивості:

- state (стан) — спочатку "pending" (очікування), в результаті виконання функції він може змінюватися на: "fulfilled" коли викликається метод resolve і на "rejected" коли reject.
- result (результат) — спочатку undefined, далі змінюється на value коли викликається метод resolve(value) або error коли reject(error).

Проміс – в стані resolve чи reject будемо називати “завершеним (settled)”, на відміну від початкового стану проміса “в очікуванні (pending)”.

Ідея в тому, що функція виконавець може мати тільки один результат чи помилку.

Методи resolve/reject можуть прийняти тільки один аргумент (або жодного), а всі додаткові аргументи будуть проігноровані.



then

Синтаксис

```
promise.then(onFulfilled[,  onRejected]);

promise.then(value  => {
  // success
  }, reason  => {
  // reject
});
```

Перший аргумент метода .then – функція що викликається коли проміс успішно виконується, тобто переходить зі стану "pending" в "resolved" і отримує результат.

Другим аргументом метод .then приймає функцію, що викликається коли проміс переходить в стан "rejected" і отримує помилку.

catch

Синтаксис

```
promise.catch(onRejected);

promise.catch( (reason) => {
  // error
 });
```

Якби ми хотіли лише обробити помилку, тоді ми могли б використати null як перший аргумент .then(null, errorHandlingFunction). Або можемо скористатись методом .catch(errorHandlingFunction), котрий зробить те ж саме

Виклик .catch(f) – це скорочений варіант .then(null, f).

finally

Синтаксис

promise.finally(onFinally)

Виклик .finally(f) подібний до .then(f, f), в тому сенсі, що f виконається в будь-якому випадку, коли проміс перейде в стан "виконано (settled)" не залежно від того став він resolved чи rejected.

finally добре підходить для чистки, або робити який небудь діспатч до стану application.

Важливо!

- Обробник finally не приймає аргументів. В finally ми не знаємо як був завершений проміс, успішно чи ні. І це нормально, тому що зазвичай наше завдання заключається в тому щоб виконати “загальні” процедури доопрацювання.
- Обробник finally пропускає результат чи помилку до наступних обробників.

Порівняння

Promises	Callbacks
Проміси дозволяють нам виконувати речі в природному порядку. Спочатку ми запускаємо loadScript(script), і потім ми записуємо в .then що робити з результатом.	У нас повинна бути функція callback на момент виклику loadScript(script, callback). Іншими словами нам потрібно знати що робити з результатом до того як викличеться loadScript.
Ми можемо викликати .then у проміса стільки раз, скільки захочемо.	Колбек може бути тільки один.

Ланцюжок промісів

Ідея полягає в тому, що результат передається через ланцюжок .then обробників.

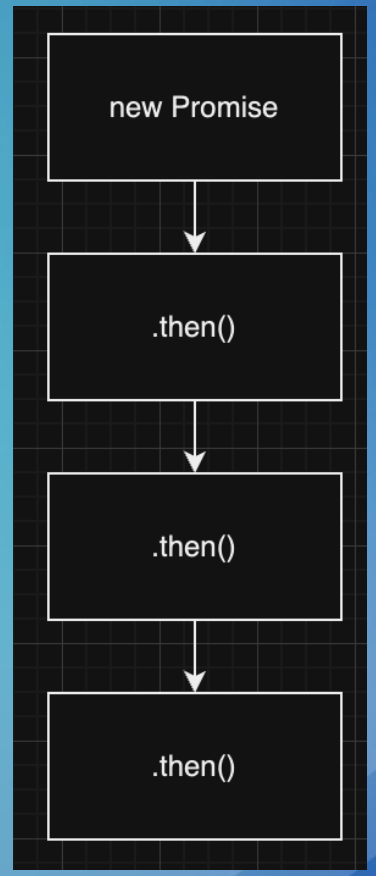
Ось потік виконання:

- Початковий проміс успішно виконується повертає 1,
- Далі на викликається обробник .then, який у свою чергу, створює новий проміс (вирішується зі значенням 1 + 1 = 2).
- Наступний then приймає результат попереднього, оброблює його та передає до наступного обробника (2 + 1 = 3).
- …і так далі.

Усе це працює тому, що кожний виклик .then повертає новий проміс, тому ми можемо викликати наступний .then на ньому.

Коли обробник повертає значення, воно стає результатом того промісу, тому наступний .then викликається з цим значенням.

Класична помилка новачка: технічно ми також можемо додати багато .then до одного промісу. Та це не ланцюжок.



Повернення промісів

Обробник, використанний в .then(handler) може створити й повернути проміс.

У цьому випадку інші обробники чекають, поки він виконається, а потім отримають його результат.

Повернення промісів дозволяє нам будувати ланцюжки асинхронних дій.

Promise.all

Синтаксис:

let promise = Promise.all(iterable);

Promise.all приймає ітеративний об’єкт (зазвичай масив промісів) і повертає новий проміс.

Новий проміс завершиться тоді, коли всі перераховані проміси завершаться, а його результатом стане масив їхніх результатів.

Якщо будь-який з промісів завершується з помилкою, то проміс, що поверне Promise.all, негайно завершиться з цією ж помилкою.

Якщо один проміс завершується з помилкою, то весь Promise.all негайно завершується з нею ж, повністю забувши про інші проміси у списку. Їх результати ігноруються.

Promise.allSettled

Синтаксис:

```
let promise = Promise.allSettled(iterable);
```

Promise.allSettled просто чекає, коли всі проміси завершаться, незалежно від результату. В отриманому масиві буде:

- {status:"fulfilled", value:result} для успішних відповідей,
- {status:"rejected", reason:error} для помилок.

Promise.race

Синтаксис:

```
let promise = Promise.race(iterable);
```

Подібний до Promise.all, але чекає лише на перший виконаний проміс та отримує його результат (або помилку).

Promise.any

Синтаксис:

let promise = Promise.any(iterable);

Схожий на Promise.race, але чекає лише на перший успішно виконаний проміс і отримує його результат. Якщо ж всі надані проміси завершуються з помилкою, то повертається проміс, що завершується з помилкою за допомогою AggregateError – спеціального об’єкта помилки, який зберігає всі помилки промісів у своїй властивості errors.

Promise.resolve/reject

Рідко використовуються

Promise.resolve(value) створює вирішений проміс із результатом value.

Те ж саме, що: let promise = new Promise(resolve => resolve(value));

Цей метод використовується для сумісності, коли очікується, що функція поверне проміс.

Promise.reject(error) створює проміс, що завершується помилкою error.

Те ж саме, що: let promise = new Promise((resolve, reject) => reject(error));

Async/await

hillel IT school

Існує спеціальний синтаксис для більш зручної роботи з промісами, який називається “async/await”. Його напрочуд легко зрозуміти та використовувати.

Слово **async** перед функцією означає одну просту річ: функція завжди повертає проміс. Інші значення автоматично загортаються в успішно виконаний проміс.

Ключове слово **await** змушує JavaScript чекати, поки проміс не виконається, та повертає його результат.

await буквально призупиняє виконання функції до тих пір, поки проміс не виконається, а потім відновлює її з результатом проміса. Це не вимагає жодних ресурсів ЦП, тому що рушій JavaScript може тим часом робити інші завдання: виконувати інші скрипти, обробляти події тощо.

Якщо ми спробуємо використати await у не-асинхронній функції, виникне синтаксична помилка

Try … catch

Якщо виникають помилки, то скрипти, зазвичай, “помирають” (раптово припиняють роботу) та виводять інформацію про помилку в консоль. Але існує синтаксична конструкція try...catch, що дозволяє нам “перехоплювати” помилки, що дає змогу скриптам виконати потрібні дії, а не раптово припинити роботу.

Конструкція try...catch містить два головних блоки: try, а потім catch. Це працює наступним чином:

1. В першу чергу виконується код в блоці try {...}.
2. Якщо не виникає помилок, то блок catch (err) ігнорується: виконання досягає кінця блоку try та продовжується поза блоком catch.
3. Якщо виникає помилка, тоді виконання в try припиняється і виконання коду продовжується з початку блоку catch (err). Змінна err (можна обрати будь-яке ім’я) буде містити об’єкт помилки з додатковою інформацією.

Щоб блок try...catch спрацював, код повинен запускатися. Іншими словами, це повинен бути валідний JavaScript.

try...catch працює синхронно!!!

Try … catch

Коли виникає помилка, JavaScript генерує об’єкт, що містить інформацію про неї. Потім цей об’єкт передається як аргумент в catch

Для всіх вбудованих помилок об’єкт помилки має дві головні властивості:

- name - Назва помилки. Наприклад, для невизначеної змінної назва буде "ReferenceError".
- message - Текстове повідомлення з додатковою інформацією про помилку.

Існують інші властивості, що доступні в більшості оточень. Одна з найуживаніших та часто підтримується:

- stack - Поточний стек викликів: рядок з інформацією про послідовність вкладених викликів, що призвели до помилки. Використовується для налагодження.

Пам’ятайте

Блок catch не обов’язково повинен перехоплювати інформацію про об’єкт помилки.

Try … catch … finaly

Інструкція finally дозволяє виконувати код після try та catch незалежно від результату. Блок finally використовується, якщо ми почали виконувати якусь роботу і хочемо завершити її в будь-якому разі.

Змінні визначені всередині try...catch...finally є локальними. Частина finally виконається в будь-якому разі при виході з try...catch. Навіть якщо явно викликати return.

Конструкція try...finally може не мати catch частини, що також може стати у пригоді. Така конфігурація може бути використана, коли ми не хочемо перехоплювати помилку, але потрібно завершити розпочаті задачі.

Об’єкт Error

JavaScript має вбудований об’єкт error, що надає інформацію про помилку при її виникненні.

Об’єкт error надає дві корисних властивості: ім’я та повідомлення.

Типи:

1. Error - Конструктор Error створює об'єкт помилки. Екземпляри об'єкта Error викидаються при помилках під час виконання. Приймає message помилки.
2. EvalError - Відбулась помилка в eval() функції;
3. RangeError - Відбулось число "out of range" (поза діапазоном);
4. ReferenceError - Відбулось неприпустиме посилання;
5. SyntaxError - Відбулась синтаксична помилка;
6. TypeError - Відбулась помилка типу;
7. URIError - Відбулась помилка в encodeURI();

throw

Оператор throw використовується для викидання помилки.

Рушії дозволяє використовувати будь-які значення як об’єкти помилки. Це може бути навіть примітивне значення, як число чи рядок, але краще використовувати об’єкти, що мають властивості name та message (для сумісності з вбудованим типом помилок).

catch

Блок catch повинен оброблювати тільки відомі помилки та повторно генерувати всі інші типи помилок.

Розгляньмо підхід “повторного викидання” покроково:

1. Конструкція catch перехоплює всі помилки.
2. В блоці catch (err) {...} ми аналізуємо об’єкт помилки err.
3. Якщо ми не знаємо як правильно обробити помилку, ми робимо throw err.

Зазвичай, тип помилки можна перевірити за допомогою оператора instanceof

Глобальний catch

Специфікація не згадує таку можливість, але оточення, зазвичай, надають таку функцію для зручності. Наприклад, Node.js дозволяє викликати process.on("uncaughtException") для цього. В браузері можна присвоїти функцію спеціальній властивості window.onerror, що виконається, коли виникне помилка.

Синтаксис:

```
window.onerror = function(message, url, line, col, error) {
  // ...
};
```

- message - Повідомлення помилки.
- url - URL скрипту, де трапилась помилка.
- line, col - Номер рядку та колонки, де трапилась помилка.
- error - Об’єкт помилки.

Розширення Error

JavaScript дозволяє використовувати throw з будь-яким аргументом, тому технічно наші спеціальні класи помилок не повинні успадковуватись від Error. Але якщо ми успадкуємо, то стає можливим використовувати obj instanceof Error для ідентифікації об’єктів помилки. Тому краще успадкувати від нього.

Дякую за увагу