

Асинхронна обробка даних. Promise

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise

<https://javascript.info/async-await>

<https://uk.javascript.info/async>

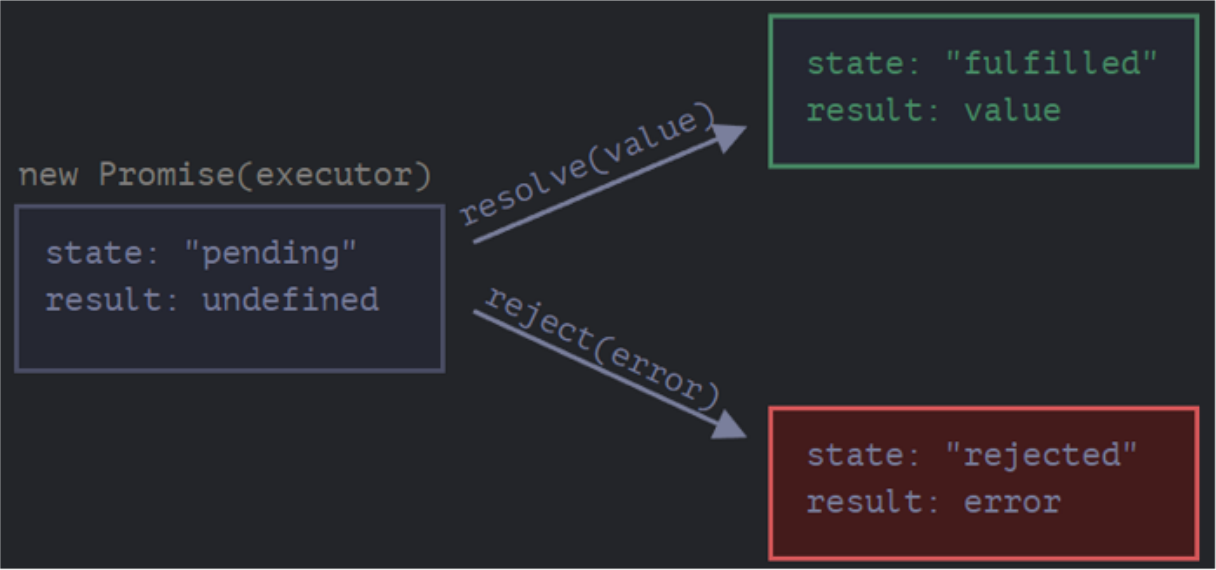
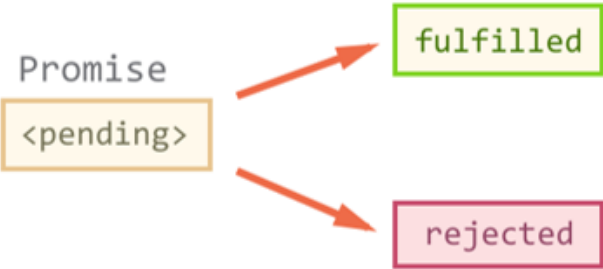
Асинхронне виконання коду. Promise

Promise (проміс) – інструмент для організації асинхронного коду (тобто коду, який не блокує виконання інших фрагментів коду)

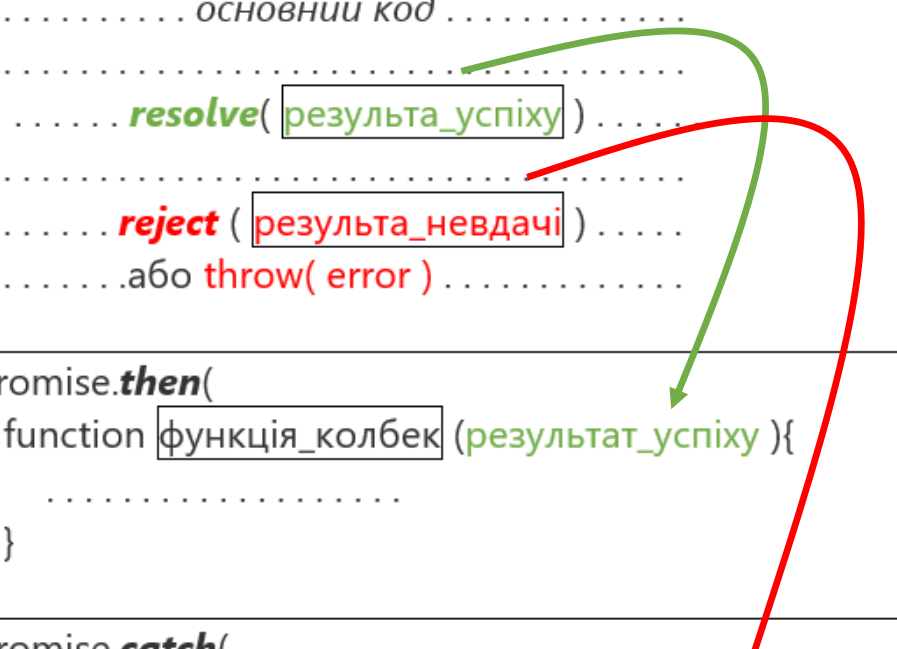
Promise – це об’єкт, який дозволяє асинхронно (без очікування/блокування) виконувати деякий фрагмент коду. Тобто, ми не очікуємо на його завершення. Натомість додаємо можливість аналізу результату цього фрагменту коду, коли він завершиться. Тобто задаємо функції, які будуть виконуватись після його виконання .

Promise може знаходитись у одному з станів

- pending – очікування (основний код виконується)
- fulfilled – основний код виконано з статусом «успішно» (ми самі вирішуємо, що це означає).
- rejected – основний код виконано з статусом «відхилено» (ми самі вирішуємо зміст цього статусу)



На *Promise* можна призначати функції обробки (функції-колбеки)

| | | |
|--|--|---|
| Опис | Загальна форма | Приклад. Генеруємо через 3 секунди число. Якщо число від 1 до 12 – то генерування успішне |
| Створення проміса | <pre>let promise = new Promise(function(<i>resolve</i>, <i>reject</i>) { основний код <i>resolve</i>(результат_успіху) <i>reject</i> (результат_невдачі) або <i>throw</i>(<i>error</i>) }</pre>  | <pre>let promise = new Promise(function (<i>resolve</i>, <i>reject</i>) { setTimeout(() => { let month = 1 + Math.floor(Math.random() * 100) if (month <= 12) <i>resolve</i>(month) else <i>reject</i>(new Error('Month is incorrect')) }, 3000) })</pre> |
| Додаємо опис обробки успішного завершення проміса (якщо викликано <i>resolve</i>) | <pre>promise.<i>then</i>(function функція_колбек (<i>результат_успіху</i>){ })</pre> | <pre>promise.<i>then</i>((<i>generatedMonth</i>) => { console.log(`Month = \${generatedMonth}`) })</pre> |
| Додаємо опис обробки відхиленого завершення проміса (якщо викликано <i>reject</i>) | <pre>promise.<i>catch</i>(function функція_колбек (<i>результат_невдачі</i>){ })</pre> | <pre>promise.<i>catch</i>((err) => { console.log(err.message) })</pre> |
| Додаємо завершальний метод, який буде виконувати обов'язково після закінчення проміса | <pre>promise.<i>finally</i>(function функція_колбек (){ })</pre> | <pre>promise.<i>finally</i>() => { console.log('Completed') })</pre> |

З окремим призначенням функцій-колбеків на раніше створений об'єкт-проміс.

```
let promise = new Promise(function (resolve, reject) {
  setTimeout(() => {
    let month = Math.floor(Math.random() * 100)
    if (month < 12) resolve(month)
    else reject(new Error('Month is incorrect'))
  }, 3000)
})
promise.then((generatedMonth) => generatedMonth+1)
promise.then((generatedMonth) => {
  console.log(`Month = ${generatedMonth}`)
})
promise.catch((err) => {
  console.log(err.message)
})
promise.catch((err) => {
  console.log(err.message)
})
```

Так робити небезпечно

console.log('Next operation')

З використанням ланцюжка промісів
new Promise(...). then(...). then(...).catch(...).finally(...)

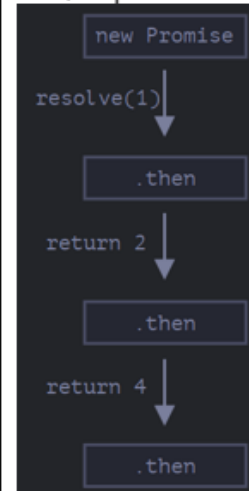
```
new Promise(function (resolve, reject) {
  setTimeout(() => {
    let month = Math.floor(Math.random() * 100)
    if (month < 12) resolve(month)
    else reject(new Error('Month is incorrect'))
  }, 3000)
})
  .then((generatedMonth) => generatedMonth+1)
  .then((generatedMonth) => {
    console.log(`Month = ${generatedMonth}`)
  })
  .catch((err) => {
    console.log(err.message)
  })
  .finally(() => {
    console.log('Completed')
  })
```

console.log('Next operation')

Якщо призначаємо декілька **then**
Виконуються незалежно.

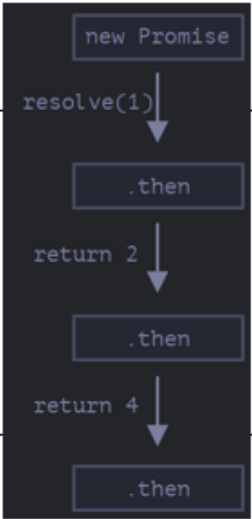


Якщо призначаємо декілька **then**. Виконуються поступово



Ланцюжок промісів. Передача даних між функціями-коблєками (результат, що ретурниться у функції-коблєку then є вхідним параметром у наступній функції-коблєку then)

| | |
|---|---|
| Загальна форма | Приклад. Генеруємо через 3 секунди число. Якщо число від 1 до 12 – то генерування успішне. Потім визначити пору року, потім визначити тип одягу |
| <pre>let promise = new Promise(function(resolve, reject) { основний код resolve(результат_успіху) reject(результат_невдачі) або throw(error) })</pre> | <pre>let promise = new Promise(function (resolve, reject) { setTimeout(() => { let month = 1 + Math.floor(Math.random() * 100) if (month <= 12) resolve(month) else reject(new Error('Month is incorrect')) }, 3000) })</pre> |
| <pre>.then(function функція_коблєк1 (параметр1){ return result2 })</pre> | <pre>.then((generatedMonth) => { let result2 switch (generatedMonth) { case 1: case 2: case 12: result2 = 'winter' break case 3: case 4: case 5: result2 = 'spring' break case 6: case 7: case 8: result2 = 'summer' break case 9: case 10: case 11: result2 = 'fall' break } return result2 })</pre> |
| <pre>.then(function функція_коблєк2 (параметр2){ return result2 })</pre> | <pre>.then((season) => { let result3 switch (season) { case 'winter': result3 = 'coat' break case 'spring': result3 = 'jacket' break case 'summer': result3 = 'shorts' break case 'fall': result3 = 'jacket' break } return result3 })</pre> |
| <pre>.then(function функція_коблєк3 (параметр3){ return result3 })</pre> | <pre>.then((dress) => { console.log(dress) })</pre> |



Спрощений спосіб створення успішного проміса з значенням *value*

| Загальна форма | Приклад |
|---|--|
| Promise.resolve(value) | Promise.resolve (25) .then(alert) //25 |
| ----- Аналог ----- <i>new Promise((resolve) => resolve(value))</i> | <i>new Promise((resolve) => resolve(25))</i> .then(alert) //25 |

Спрощений спосіб створення «помилкового» проміса з значенням *value*

| Загальна форма | Приклад |
|---|---|
| Promise.reject(error) | Promise.reject (new Error("error")) .catch(alert) // Error: ... |
| ----- Аналог ----- <i>new Promise((resolve, reject) => reject (value))</i> або <i>new Promise(() => throw value)</i> | <i>new Promise((resolve, reject) => reject(new Error("error"))</i> <i>.catch(alert) // Error: ...</i> |

Паралельне виконання промісів. *Promise.all*

Promise.all

- дозволяє виконувались паралельно декільком промісам
- очікує поки всі вони виконаються (результатом є масив результатів кожного з промісів)
- якщо якийсь з промісів завершується невдачею, то весь проміс припиняє роботу

| Загальна форма | Приклад |
|--|---|
| <pre>Promise.all(iterable) .then(function(results_array){ }) .catch(function(one_error){ })</pre> <p>iterable – ітерована колекція промісів results_array – масив з результатами виконання переданих промісів one_error – помилка, першого проблемного промісу (виконання всього промісу зупиняється)</p> | <pre>Promise.all([new Promise(resolve => setTimeout(() => resolve(1), 3000)), // 1 new Promise(resolve => setTimeout(() => resolve(2), 2000)), // 2 new Promise(resolve => setTimeout(() => resolve(3), 1000)) // 3]) .then(alert) // [1, 2, 3] .catch((err)=>{ console.log(err) })</pre> |

Конкурентне виконання промісів. *Promise.race*

- дозволяє виконувались паралельно декільком промісам
- чекає лише на перший виконаний проміс (результати інших ігноруються)
- результатом є результат першого виконаного промісу або помилка

| Загальна форма | Приклад |
|--|---|
| <pre>Promise.race (iterable) .then(function(one_result){ }) .catch(function(one_error){ })</pre> <p>iterable – ітерована колекція промісів one_result – результат виконання першого успішного проміса one_error – помилка, першого проблемного промісу (виконання всього промісу зупиняється)</p> | <pre>Promise.race ([new Promise(resolve => setTimeout(() => resolve(1), 3000)), // 1 new Promise(resolve => setTimeout(() => resolve(2), 2000)), // 2 new Promise(resolve => setTimeout(() => resolve(3), 1000)) // 3) .then(alert) .catch((err)=>{ console.log(err) })</pre> |

Спеціалізований синтаксис для роботи з промісами. `async/await`

1) Опис функції, що повертає проміс (або явно створюємо і повертаємо проміс, або він автоматично створюється)

| Загальна форма (використати <code>async</code>) | Приклад |
|--|---|
| <pre><code><u>async</u> function <u>назва</u> () { return <u>результат</u> }</code></pre> <p>результат – не обов’язково проміс (якщо не так, то він буде «загорнутий» у проміс)</p> | <pre><code>async function f() { return 1; //тут автоматично створюється проміс } f().then(alert); // 1 ----- аналог ----- async function f() { return Promise.resolve(1) //явно створений проміс } f().then(alert); // 1</code></pre> |

- 2) всередині асинхронної функції можна використовувати ***await***, що призводить до
- очікування виконання позначеної таким чином асинхронної операції.
 - обробка помилок здійснюється з використанням звичайного ***try..catch..finally***

| Загальна форма (використати <i>async</i>) | Приклад |
|---|--|
| <pre><i>async</i> function назва () { try{ const res1 = <i>await</i> asyncFunc1() const res2 = <i>await</i> asyncFunc2() return результат } catch(err){ } finally{ } }</pre> <p>результат – не обов’язково проміс (якщо не так, то він буде «загорнутий» у проміс)</p> | <pre>//--- асинхронні функції <i>async</i> function f1() { return new Promise((<i>resolve</i>) => { setTimeout(() => { <i>resolve</i>(1) }, 2000) }) } <i>async</i> function f2() { return new Promise((<i>resolve</i>) => { setTimeout(() => { <i>resolve</i>(10) }, 2000) }) } //----виклик асинхронних функцій з await----- <i>async</i> function resFunc2() { try{ const res1 = <i>await</i> f1() // очікуємо результат const res2 = <i>await</i> f2() // очікуємо результат console.log(res1 + res2) // 11 } catch(err){ console.log(err) } } //===== resFunc2()</pre> |

Приклад. Завантажити список усіх порід собак. Та вивести випадковим чином вибране зображення собаки якоїсь породи

```
<script>
  //--- масив з шляхами до API ----
  const apiEndpoints = {
    allBreedsList: 'https://dog.ceo/api/breeds/list/all',
    getReadByBreedNameLink: (breedName) =>
      `https://dog.ceo/api/breed/${breedName}/images/random`,
  }
  //---- Функція завантаження списку усіх порід ----
  async function loadBreedsList() {
    const url = apiEndpoints.allBreedsList
    return new Promise((resolve, reject) => {
      fetch(url)
        .then((response) => response.json())
        .then((data) => data.message)
        .then((listObject) => {
          resolve(Object.keys(listObject))
        })
        .catch((err) => {
          reject(err)
        })
    })
  }
  //--- Функція вибору випадкового елемента з масиву -----
  function getRandomListIten(list) {
    const randomIndex = Math.floor(Math.random() * list.length)
    return list[randomIndex]
  }
```

```
  //--- функція для завантаження випадкового зображення вказаної породи ---
  async function loadBreedImage(breedName) {
    const url = apiEndpoints.getReadByBreedNameLink(breedName)
    return new Promise((resolve, reject) => {
      fetch(url)
        .then((response) => response.json())
        .then((data) => data.message)
        .then((imageLink) => {
          resolve(imageLink)
        })
        .catch((err) => {
          reject(err)
        })
    })
  }
  //--- функція створення елемента зображення ----
  function createImage(imgSrc) {
    const img = document.createElement('img')
    img.src = imgSrc
    return img
  }
  //=== головна функція виконання усіх кроків =====
  async function go(params) {
    //-- отримання списку порід ---
    let breedsList = await loadBreedsList()
    //-- вибір випадкової породи ---
    const randomBreed = getRandomListIten(breedsList)
    //-- отримання випадкового зображення собаки вказаної породи ---
    const imageLink = await loadBreedImage(randomBreed)
    //-- створення і додавання елемента зображення собаки ---
    document.body.append(createImage(imageLink))
  }
  //===== Виклик головної функції =====
  go()
</script>
```