skillUp

Promise. Ajax

План

- try … catch
- json
- setTimeout
- promise
- async/await
- ajax
- storage

Перехват помилок try..catch

- коли потрібно щоб скрипт при помилці не припинив роботу, а обробив помилку

```
try {
  // код ...
} catch (err) {
  // обробка помилки
}
```

Виконується код всередині блоку **try**.

Якщо в ньому помилок немає, то блок **catch(err)** ігнорується, тобто виконання доходить до кінця **try** і потім стрибає через **catch**.

Якщо в ньому виникне помилка, то виконання **try** на ній переривається, і управління стрибає в початок блоку **catch(err).**

При цьому змінна **err** (можна вибрати й іншу назву) буде містити об'єкт помилки з докладною інформацією про те, що сталося.

Приклад

```
try {
  test; // помилка, змінна не визначена!
} catch(e) {
  alert('ПОмилка ' + e.name + ":" + e.message + "\n" + e.stack); // (3) <--
}
```

name - тип помилки. Наприклад, при зверненні до неіснуючої змінної: "ReferenceError".

message - текстове повідомлення про деталі помилки.

stack - скрізь, крім IE8-, є також властивість stack, яке містить рядок з інформацією про послідовність викликів, яка привела до помилки.

Генерація своїх помилок

- Як конструктор помилок можна використовувати вбудований конструктор: new Error (message) або будь-який інший.
- стандартні помилки: SyntaxError, ReferenceError, RangeError

```
try {
  throw new SyntaxError("повідомлення");
} catch (err) {
  // обробка помилки
}
```

блок finally

- виконається завжди

```
try {
    .. код ..
} catch(e) {
    .. перехват  ..
} finally {
    .. виконається завжди ..
}
```

json

- Це один з найбільш зручних форматів даних при взаємодії з JavaScript. Якщо потрібно з сервера взяти об'єкт з даними і передати його клієнту, то в якості проміжного формату - для передачі по мережі, майже завжди використовують саме його

Методи для роботи з JSON в JavaScript - це:

JSON.parse - читає об'єкти з рядка в форматі JSON.

JSON.stringify - перетворює об'єкти в рядок у форматі JSON, використовується, коли потрібно з JavaScript передати дані по мережі.

Приклади

```
var user = '{ "name": "Вася", "age": 35, "isAdmin": false, "friends": [0,1,2,3] }';
user = JSON.parse(user);
alert( user.friends[1] ); // 1
```

```
var user = {
  name: "Вася",
  age: 25,
  window: window
};
alert( JSON.stringify(user, ["name", "age"]) );
// {"name":"Вася","age":25}
```

setTimeout

- метод викликає функцію або виражає вираз після заданої кількості мілісекунд

```
setTimeout(function() {
  alert( 'стоп' );
}, 5000);
```

clearTimeout

- Функція setTimeout повертає числовий ідентифікатор таймера timerId, який можна використовувати для скасування дії

```
var timerId = setTimeout(...);
clearTimeout(timerId);
```

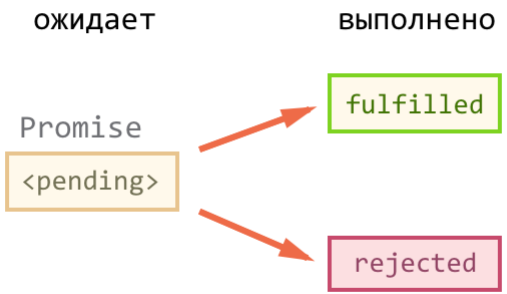
setInterval

- запускає код через рівні проміжки часу

```
setInterval(function() {
  alert( "тик" );
}, 2000);
```

Promise

- надають зручний спосіб організації асинхронного коду
- спеціальний об'єкт, який містить свій стан. Спочатку pending («очікування»), потім - одне з: fulfilled («виконано успішно») або rejected («виконано з помилкою»).



https://learn.javascript.ru/promise

```
var promise = new Promise(function(resolve, reject) {
  // Ця функція буде викликана автоматично

  // В ній можна робити будь-які асинхронні операції,
  // А коли вони завершаться - потрібно викликати одне з:
  // resolve (результат) при успішному виконанні
  // reject (помилка) при помилку
})
promise.then(onFulfilled, onRejected)
```

onFulfilled - функція, яка буде викликана з результатом при resolve.

onRejected - функція, яка буде викликана з помилкою при reject.

https://jsfiddle.net/blog_code/u68qvhy4/1/

Приклад

```
let promise = new Promise(function(resolve, reject) {
  setTimeout(() => resolve("done!"), 1000);
});
// resolve запустить першу функцію в .then
promise.then(
  result => alert(result), // виведе "done!" через одну секунду
  error => alert(error) // не буде запущена
);
```

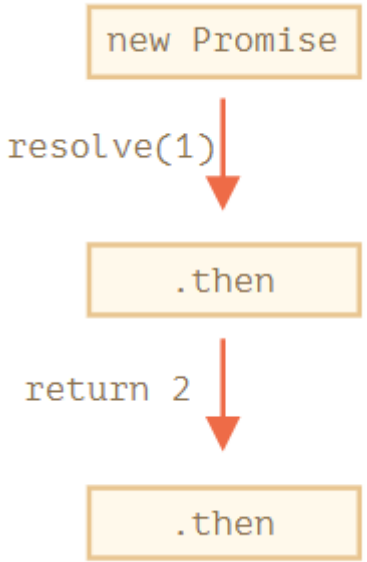
Приклад

```
let promise = new Promise(function(resolve, reject) {
  setTimeout(() => reject(new Error("Whoops!")), 1000);
});

// reject запустить другу функцію в .then
promise.then(
  result => alert(result), // не будет запущена
  error => alert(error) // виведе "Error: Whoops!" через одну секунду
);
```

Ланцюжок промісів

```
new Promise(function(resolve, reject) {
  setTimeout(() => resolve(1), 1000);
}).then(function(result) {
  alert(result); // 1
  return new Promise((resolve, reject) => { // (*)
    setTimeout(() => resolve(result * 2), 1000);
  });
}).then(function(result) {
  alert(result); // 2
});
```



catch

якщо потрібно обробити тільки помилку

```
let promise = new Promise((resolve, reject) => {
  setTimeout(() => reject(new Error("Ошибка!")), 1000);
});

promise.catch(alert); // виведе "Error: Ошибка!" через одну секунду
```

finally

виконається в будь-якому випадку, коли промис завершиться: успішно чи з помилкою

```
let promise = new Promise((resolve, reject) => {
  setTimeout(() => resolve("result"), 2000)
})
promise.finally(() => alert("Промис завершений"))
```

Функції у promise

Promise.**all** (promises) - очікує виконання всіх Проміс і повертає масив з результатами. Якщо будь-який із зазначених Проміс поверне помилку, то результатом роботи Promise.all буде ця помилка, результати інших Проміс будуть ігноруватися.

Promise.**allSettled** (promises) (доданий недавно) - чекає, поки всі Проміс завершаться і повертає їх результати у вигляді масиву з об'єктами

Promise.**race** (promises) - очікує перший виконаний промис, який стає його результатом, інші ігноруються.

Async/await

спеціальний синтаксис для роботи з Проміс, який називається «async / await». Обгортка на промісами

```
async function f() {

  let promise = new Promise((resolve, reject) => {
    setTimeout(() => resolve("готово!"), 1000)
  });

  let result = await promise; // будет чекати, поки промис не виконається (*)

  alert(result); // "готово!"
}

f(); // виклик функції
```

Перехоплення помилок

```
async function f() {
  try {
    let promise = new Promise((resolve, reject) => {
                  setTimeout(() => reject("готово!"), 1000)
         });
         let result = await promise; ;
  } catch(err) {
    // перехопить будь-яку помилку в блоці try: і в fetch, і в response.json
    alert(err);
  }
}
f();
```

ajax

Asynchronous **J**avaScript **A**nd **X**ML підхід до побудови користувацьких інтерфейсів веб-застосунків, за яких веб-сторінка, не перезавантажуючись, у фоновому режимі надсилає запити на сервер і сама звідти довантажує потрібні користувачу дані

методи http запитів

GET використовується для отримання інформації від сервера

HEAD працює точно так само, як GET, але у відповідь сервер посилає тільки заголовки і статусну строку без тіла HTTP повідомлення

POST використовується для відправлення даних на сервер, наприклад, з HTML форм

PUT використовується для завантаження вмісту запиту на вказане в цьому ж запиті URI

DELETE удаляет указанный в URI ресурс

OPTIONS використовується для отримання параметрів поточного HTTP соединения

https://developer.mozilla.org/ru/docs/Web/HTTP/Methods

XMLHttpRequest

```
var xhr = new XMLHttpRequest();

    xhr.open('GET', 'phones.json', false);
    xhr.send();

    if (xhr.status != 200) {
      // обробити помилку
      alert('Помилка ' + xhr.status + ': ' + xhr.statusText);
    } else {
      // вивести результат
      alert(xhr.responseText);
    }
```

https://learn.javascript.ru/ajax-xmlhttprequest

fetch

використовується щоб отримати інформацію з віддаленого сервера.

```
let promise = fetch(url, [options]);
```
- повертає проміс

Приклад

```
fetch('https://learn.javascript.ru/article/promise-chaining/user.json')
  .then(response => response.json())
  .then(user => alert(user.name));
```

- запит на віддалений файлик на сервері

- якщо проміс успішний то повертається об’єкт **response** і читає дані у форматі json і теж повертає проміс

- далі ми беремо властивість **name** з об’єкта **user**

Приклад

```
async function f() {
  try {
    let response = await fetch('https://learn.javascript.ru/article/promise-chaining/user.json');
        let user = await response.json();
  } catch(err) {
    console.log(err);
  }
}
f();
```

CRUD

CRUD (скор. Від англ. Create, read, update, delete - «створити, прочитати, оновити, видалити») - визначає чотири базові функції, які використовуються при роботі з сховищами даних: створення; читання; редагування; видалення.

Практика

Вивести на сторінку аватар свого профіля github
https://api.github.com/users/${user.name}

Практика

виконати ajax запити через api https://jsonplaceholder.typicode.com/

документація по fetch https://learn.javascript.ru/fetch-api

Посилання

https://learn.javascript.ru/error-handling
https://learn.javascript.ru/async

https://learn.javascript.ru/network

Відео

https://www.youtube.com/watch?v=vNEDPtVchfw

https://www.youtube.com/watch?v=1idOY3C1gYU

https://www.youtube.com/watch?v=SHiUyM_fFME