Async/await

Обработчики промисов .then/.catch/.finally всегда асинхронны. Даже когда промис сразу же выполнен, код ниже обработчиков .then/.catch/.finally будет запущен до этих обработчиков. Асинхронные задачи требуют правильного управления. Для этого стандарт предусматривает внутреннюю очередь PromiseJobs, более известную как «очередь микрозадач (microtask queue)» (термин V8). Как сказано в спецификации:

- Очередь определяется как первым-пришёл-первым-ушёл (FIFO): задачи, попавшие в очередь первыми, выполняются тоже первыми.
- Выполнение задачи происходит только в том случае, если ничего больше не запущено.

Или, проще говоря, когда промис выполнен, его обработчики .then/catch/finally попадают в очередь. Они пока не выполняются. Движок JavaScript берёт задачу из очереди и выполняет её, когда он освободится от выполнения текущего кода.

Если нам нужно гарантировать выполнение какого-то кода после .then/catch/finally, то лучше всего добавить его вызов в цепочку .then.

"Необработанная ошибка" возникает в случае, если ошибка промиса не обрабатывается в конце очереди микрозадач. Обычно, если мы ожидаем ошибку, мы добавляем .**catch** в конец цепочки промисов, чтобы обработать её:

```
let promise = Promise.reject(new Error("Ошибка в промисе!"));
promise.catch(err => alert('поймана!'));

// не запустится, ошибка обработана
window.addEventListener('unhandledrejection', event => {
  alert(event.reason);
});
```

…Но если мы забудем добавить .catch, то, когда очередь микрозадач опустеет, движок сгенерирует событие: **'unhandledrejection'**

```
async function f() {
  return 1;
}
f().then(alert); // 1
```

У слова **async** один простой смысл: эта функция всегда возвращает промис. Значения других типов оборачиваются в завершившийся успешно промис автоматически.

Можно и явно вернуть промис, результат будет одинаковым:

```
async function f() {
  return Promise.resolve(1);
}
f().then(alert); // 1

// работает только внутри async–функций
let value = await promise;
```

Ключевое слово **await** заставит интерпретатор JavaScript ждать до тех пор, пока промис справа от await не выполнится. После чего оно вернёт его результат, и выполнение кода продолжится. Пока промис не выполнится, JS-движок может заниматься другими задачами: выполнять прочие скрипты, обрабатывать события и т.п.

await нельзя использовать в обычных функциях!

На верхнем уровне вложенности можно обернуть этот код в анонимную async–функцию:

```
(async () => {
  let response = await fetch('/article/promise-chaining/user.json');
  let user = await response.json();
  ...
})();
```

Асинхронные методы классов

Для объявления асинхронного метода достаточно написать async перед именем:

```
class Waiter {
  async wait() {
    return await Promise.resolve(1);
  }
}
```

```
new Waiter()
  .wait()
  .then(alert); // 1
```
Как и в случае с асинхронными функциями, такой метод гарантированно возвращает промис, и в его теле можно использовать await.

Обработка ошибок

Когда промис завершается успешно, await promise возвращает результат. Когда завершается с ошибкой – будет выброшено исключение. Как если бы на этом месте находилось выражение **throw**. Такой код:

```
async function f() {
  await Promise.reject(new Error("Упс!"));
}
```
Делает то же самое, что и такой:

```
async function f() {
  throw new Error("Упс!");
}
```
Но есть отличие: на практике промис может завершиться с ошибкой не сразу, а через некоторое время. В этом случае будет задержка, а затем await выбросит исключение.Такие ошибки можно ловить, используя try..catch, как с обычным throw:

```
async function f() {

  try {
    let     fetch('http://no-such-url');
  } catch(err) {
    alert(err); // TypeError: failed to fetch
  }
}
f();
```
Если у нас нет try..catch, асинхронная функция будет возвращать **завершившийся с ошибкой промис** (в состоянии **rejected**). В этом случае мы можем использовать метод **.catch** промиса, чтобы обработать ошибку:

```
async function f() {
  let response = await fetch('http://no-such-url');
}
// f() вернёт промис в состоянии rejected
f().catch(alert); // TypeError: failed to fetch // (*)
```
Если забыть добавить .catch, то будет сгенерирована ошибка «**Uncaught promise error**» и информация об этом будет выведена в консоль.

async/await отлично работает с Promise.all Когда необходимо подождать несколько промисов одновременно, можно обернуть их в Promise.all, и затем await:

```
// await будет ждать массив с результатами выполнения всех промисов
let results = await Promise.all([
  fetch(url1),
  fetch(url2),
  ...
]);
```