Материалы занятия

Курс: Разработка интерфейса на JavaScript

Дисциплина: Основы JavaScript

Тема занятия №32: Асинхронность. Промисы. Async-Await. Fetch

Промисы

Представьте, что вы известный певец, которого фанаты постоянно донимают расспросами о предстоящем сингле.

Чтобы получить передышку, вы обещаете разослать им сингл, когда он будет выпущен. Вы даёте фанатам список, в который они могут записаться. Они могут оставить там свой e-mail, чтобы получить песню, как только она выйдет. И даже больше: если что-то пойдёт не так, например, в студии будет пожар и песню выпустить не выйдет, они также получат уведомление об этом.

Все счастливы! Вы счастливы, потому что вас больше не донимают фанаты, а фанаты могут больше не беспокоиться, что пропустят новый сингл.

Это аналогия из реальной жизни для ситуаций, с которыми мы часто сталкиваемся в программировании:

1. Есть «создающий» код, который делает что-то, что занимает время. Например, загружает данные по сети. В нашей аналогии это – «певец».
2. Есть «потребляющий» код, который хочет получить результат «создающего» кода, когда он будет готов. Он может быть необходим более чем одной функции. Это – «фанаты».
3. Promise (по англ. promise, будем называть такой объект «промис») – это специальный объект в JavaScript, который связывает «создающий» и «потребляющий» коды вместе. В терминах нашей аналогии – это «список для подписки». «Создающий» код может выполняться сколько потребуется, чтобы получить результат, а *промис* делает результат доступным для кода, который подписан на него, когда результат готов.

Аналогия не совсем точна, потому что объект Promise в JavaScript гораздо сложнее простого списка подписок: он обладает дополнительными возможностями и ограничениями. Но для начала и такая аналогия хороша.

Синтаксис создания Promise:

```
let promise = new Promise(function(resolve, reject) {
  // функция-исполнитель (executor)
  // "певец"
});
```

Функция, переданная в конструкцию new Promise, называется *исполнитель* (executor). Когда Promise создаётся, она запускается автоматически. Она должна

содержать «создающий» код, который когда-нибудь создаст результат. В терминах нашей аналогии: *исполнитель* – это «певец».

Её аргументы resolve и reject – это колбэки, которые предоставляет сам JavaScript. Наш код – только внутри исполнителя.

Когда он получает результат, сейчас или позже – не важно, он должен вызвать один из этих колбэков:

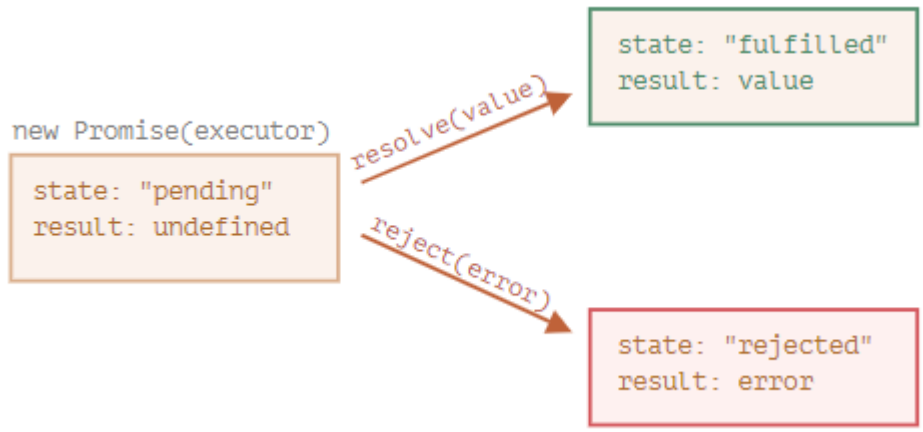
- resolve(value) — если работа завершилась успешно, с результатом value.
- reject(error) — если произошла ошибка, error – объект ошибки.

Итак, исполнитель запускается автоматически, он должен выполнить работу, а затем вызвать resolve или reject.

У объекта promise, возвращаемого конструктором new Promise, есть внутренние свойства:

- state («состояние») — вначале "pending" («ожидание»), потом меняется на "fulfilled" («выполнено успешно») при вызове resolve или на "rejected" («выполнено с ошибкой») при вызове reject.
- result («результат») — вначале undefined, далее изменяется на value при вызове resolve(value) или на error при вызове reject(error).

Так что исполнитель по итогу переводит promise в одно из двух состояний:



Позже мы рассмотрим, как «фанаты» узнают об этих изменениях.

Ниже пример конструктора Promise и простого исполнителя с кодом, дающим результат с задержкой (через setTimeout):

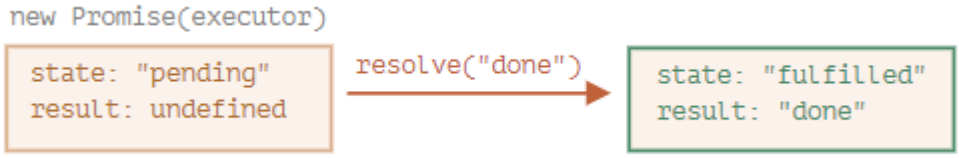
```
let promise = new Promise(function(resolve, reject) {
  // эта функция выполнится автоматически, при вызове new Promise

  // через 1 секунду сигнализировать, что задача выполнена с результатом
  setTimeout(() => resolve("done"), 1000);
});
```

Мы можем наблюдать две вещи, запустив код выше:

1. Функция-исполнитель запускается сразу же при вызове new Promise.

2. Исполнитель получает два аргумента: resolve и reject — это функции, встроенные в JavaScript, поэтому нам не нужно их писать. Нам нужно лишь позаботиться, чтобы исполнитель вызвал одну из них по готовности.

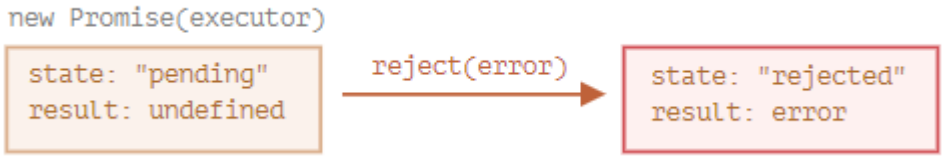
Спустя одну секунду «обработки» исполнитель вызовет resolve("done"), чтобы передать результат:



Это был пример успешно выполненной задачи, в результате мы получили «успешно выполненный» промис.

А теперь пример, в котором исполнитель сообщит, что задача выполнена с ошибкой:

```
let promise = new Promise(function(resolve, reject) {
  // спустя одну секунду будет сообщено, что задача выполнена с ошибкой
  setTimeout(() => reject(new Error("Whoops!")), 1000);
});
```



Подведём промежуточные итоги: исполнитель выполняет задачу (что-то, что обычно требует времени), затем вызывает resolve или reject, чтобы изменить состояние соответствующего Promise.

Промис – и успешный, и отклонённый будем называть «завершённым», в отличие от изначального промиса «в ожидании».

Может быть что-то одно: либо результат, либо ошибка.

Исполнитель должен вызвать что-то одно: resolve или reject. Состояние промиса может быть изменено только один раз.

Все последующие вызовы resolve и reject будут проигнорированы:

```
let promise = new Promise(function(resolve, reject) {
  resolve("done");

  reject(new Error("…")); // игнорируется
  setTimeout(() => resolve("…")); // игнорируется
});
```

Идея в том, что задача, выполняемая исполнителем, может иметь только один итог: результат или ошибку.

Также заметим, что функция resolve/reject ожидает только один аргумент (или ни одного). Все дополнительные аргументы будут проигнорированы.

Вызывайте reject с объектом Error.

В случае, если что-то пошло не так, мы должны вызвать reject. Это можно сделать с аргументом любого типа (как и resolve), но рекомендуется использовать объект Error (или унаследованный от него). Почему так? Скоро нам станет понятно.

Вызов resolve/reject сразу.

Обычно исполнитель делает что-то асинхронное и после этого вызывает resolve/reject, то есть через какое-то время. Но это не обязательно, resolve или reject могут быть вызваны сразу:

```
let promise = new Promise(function(resolve, reject) {
  // задача, не требующая времени
  resolve(123); // мгновенно выдаст результат: 123
});
```

Это может случиться, например, когда мы начали выполнять какую-то задачу, но тут же увидели, что ранее её уже выполняли, и результат закеширован.

Такая ситуация нормальна. Мы сразу получим успешно завершённый Promise.

Потребители: then, catch

Объект Promise служит связующим звеном между исполнителем («создающим» кодом или «певцом») и функциями-потребителями («фанатами»), которые получат либо результат, либо ошибку. Функции-потребители могут быть зарегистрированы (подписаны) с помощью методов .then и .catch.

then

Наиболее важный и фундаментальный метод – .then.

Синтаксис:

```
promise.then(
  function(result) { /* обработает успешное выполнение */ },
  function(error) { /* обработает ошибку */ }
);
```

Первый аргумент метода .then – функция, которая выполняется, когда промис переходит в состояние «выполнен успешно», и получает результат.

Второй аргумент .then – функция, которая выполняется, когда промис переходит в состояние «выполнен с ошибкой», и получает ошибку.

Например, вот реакция на успешно выполненный промис:

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

// resolve запустит первую функцию, переданную в .then
promise.then(
  result => alert(result), // выведет "done!" через одну секунду
  error => alert(error) // не будет запущена
);
```

Выполнилась первая функция.

А в случае ошибки в промисе – выполнится вторая:

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

// reject запустит вторую функцию, переданную в .then
promise.then(
  result => alert(result), // не будет запущена
  error => alert(error) // выведет "Error: Whoops!" спустя одну секунду
);
```

Если мы заинтересованы только в результате успешного выполнения задачи, то в then можно передать только одну функцию:

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

promise.then(alert); // выведет "done!" спустя одну секунду
```

catch

Если мы хотели бы только обработать ошибку, то можно использовать null в качестве первого аргумента: .then(null, errorHandlingFunction). Или можно воспользоваться методом .catch(errorHandlingFunction), который сделает то же самое:

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

// .catch(f) это то же самое, что promise.then(null, f)
promise.catch(alert); // выведет "Error: Ошибка!" спустя одну секунду
```

Вызов .catch(f) – это сокращённый, «укороченный» вариант .then(null, f).

Очистка: finally

По аналогии с блоком finally из обычного try {...} catch {...}, у промисов также есть метод finally.

Вызов .finally(f) похож на .then(f, f), в том смысле, что f выполнится в любом случае, когда промис завершится: успешно или с ошибкой.

Идея finally состоит в том, чтобы настроить обработчик для выполнения очистки/доведения после завершения предыдущих операций.

Например, остановка индикаторов загрузки, закрытие больше не нужных соединений и т.д.

Думайте об этом как о завершении вечеринки. Независимо от того, была ли вечеринка хорошей или плохой, сколько на ней было друзей, нам все равно нужно (или, по крайней мере, мы должны) сделать уборку после нее.

Обратите внимание, что finally(f) – это не совсем псевдоним then(f,f), как можно было подумать.

Есть важные различия:

1. Обработчик, вызываемый из finally, не имеет аргументов. В finally мы не знаем, как был завершён промис. И это нормально, потому что обычно наша задача – выполнить «общие» завершающие процедуры.

   Пожалуйста, взгляните на приведенный выше пример: как вы можете видеть, обработчик finally не имеет аргументов, а результат promise обрабатывается в следующем обработчике.

2. Обработчик finally «пропускает» результат или ошибку дальше, к последующим обработчикам.

   Например, здесь результат проходит через finally к then:

   ```
   new Promise((resolve, reject) => {
     setTimeout(() => resolve("value"), 2000);
   })
     .finally(() => alert("Промис завершён")) // срабатывает первым
     .then(result => alert(result)); // <-- .then показывает "value"
   ```

   Как вы можете видеть, значение возвращаемое первым промисом, передается через finally к следующему then.

   Это очень удобно, потому что finally не предназначен для обработки результата промиса. Как уже было сказано, это место для проведения общей очистки, независимо от того, каков был результат.

   А здесь ошибка из промиса проходит через finally к catch:

```
new Promise((resolve, reject) => {
  throw new Error("error");
})
  .finally(() => alert("Промис завершён")) // срабатывает первым
  .catch(err => alert(err));  // <-- .catch показывает ошибку
```

3. Обработчик finally также не должен ничего возвращать. Если это так, то возвращаемое значение молча игнорируется.

   Единственным исключением из этого правила является случай, когда обработчик finally выдает ошибку. Затем эта ошибка передается следующему обработчику вместо любого предыдущего результата.

Подведем итог:

- Обработчик finally не получает результат предыдущего обработчика (у него нет аргументов). Вместо этого этот результат передается следующему подходящему обработчику.
- Если обработчик finally возвращает что-то, это игнорируется.
- Когда finally выдает ошибку, выполнение переходит к ближайшему обработчику ошибок.

Эти функции полезны и заставляют все работать правильно, если мы используем finally так, как предполагается: для общих процедур очистки.

Пример: loadScript

Теперь рассмотрим несколько практических примеров того, как промисы могут облегчить нам написание асинхронного кода.

У нас есть функция loadScript для загрузки скрипта из предыдущей главы.

Давайте вспомним, как выглядел вариант с колбэками:

```
function loadScript(src, callback) {
  let script = document.createElement('script');
  script.src = src;

  script.onload = () => callback(null, script);
  script.onerror = () => callback(new Error(`Ошибка загрузки скрипта ${src}`));

  document.head.append(script);
}
```

Теперь перепишем её, используя Promise.

Новой функции loadScript более не нужен аргумент callback. Вместо этого она будет создавать и возвращать объект Promise, который перейдет в состояние «успешно завершён», когда загрузка закончится. Внешний код может добавлять обработчики («подписчиков»), используя .then:

```
function loadScript(src) {
  return new Promise(function(resolve, reject) {
    let script = document.createElement('script');
    script.src = src;

    script.onload = () => resolve(script);
    script.onerror = () => reject(new Error(`Ошибка загрузки скрипта ${src}`));

    document.head.append(script);
  });
}
```

Промисы	Колбэки
Промисы позволяют делать вещи в естественном порядке. Сперва мы запускаем `loadScript(script)`, и затем (`.then`) мы пишем, что делать с результатом.	У нас должна быть функция `callback` на момент вызова `loadScript(script, callback)`. Другими словами, нам нужно знать что делать с результатом *до того*, как вызовется `loadScript`.
Мы можем вызывать `.then` у `Promise` столько раз, сколько захотим. Каждый раз мы добавляем нового «фаната», новую функцию-подписчика в «список подписок». Больше об этом в следующей главе: Цепочка промисов.	Колбэк может быть только один.

Async/await

Существует специальный синтаксис для работы с промисами, который называется «async/await». Он удивительно прост для понимания и использования.

Асинхронные функции

Начнём с ключевого слова async. Оно ставится перед функцией, вот так:

```
async function f() {
  return 1;
}
```

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

Например, эта функция возвратит выполненный промис с результатом 1:

```
async function f() {
  return 1;
}

f().then(alert); // 1
```

Можно и явно вернуть промис, результат будет одинаковым:

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

f().then(alert); // 1
```

Так что ключевое слово async перед функцией гарантирует, что эта функция в любом случае вернёт промис. Согласитесь, достаточно просто? Но это ещё не всё. Есть другое ключевое слово – await, которое можно использовать только внутри async-функций.

Await

Синтаксис:

```
// 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.

В этом примере промис успешно выполнится через 1 секунду:

```
async function f() {

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

  let result = await promise; // будет ждать, пока промис не выполнится (*)

  alert(result); // "готово!"
}

f();
```

В данном примере выполнение функции остановится на строке (*) до тех пор, пока промис не выполнится. Это произойдёт через секунду после запуска функции. После чего в переменную result будет записан результат выполнения промиса, и браузер отобразит alert-окно «готово!».

Обратите внимание, хотя await и заставляет JavaScript дожидаться выполнения промиса, это не отнимает ресурсов процессора. Пока промис не выполнится, JS-движок может заниматься другими задачами: выполнять прочие скрипты, обрабатывать события и т.п.

По сути, это просто «синтаксический сахар» для получения результата промиса, более наглядный, чем promise.then.

await нельзя использовать в обычных функциях.

Если мы попробуем использовать await внутри функции, объявленной без async, получим синтаксическую ошибку:

```
function f() {
  let promise = Promise.resolve(1);
  let result = await promise; // SyntaxError
}
```

Ошибки не будет, если мы укажем ключевое слово async перед объявлением функции. Как было сказано раньше, await можно использовать только внутри async–функций.

Давайте перепишем пример showAvatar() из раздела Цепочка промисов с помощью async/await:

1. Нам нужно заменить вызовы .then на await.
2. И добавить ключевое слово async перед объявлением функции.

```
async function showAvatar() {

  // запрашиваем JSON с данными пользователя
  let response = await fetch('/article/promise-chaining/user.json');
  let user = await response.json();

  // запрашиваем информацию об этом пользователе из github
  let githubResponse = await fetch(`https://api.github.com/users/${user.name}`);
  let githubUser = await githubResponse.json();

  // отображаем аватар пользователя
  let img = document.createElement('img');
  img.src = githubUser.avatar_url;
  img.className = "promise-avatar-example";
  document.body.append(img);

  // ждём 3 секунды и затем скрываем аватар
  await new Promise((resolve, reject) => setTimeout(resolve, 3000));

  img.remove();

  return githubUser;
}

showAvatar();
```

Получилось очень просто и читаемо, правда? Гораздо лучше, чем раньше.

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 response = await fetch('http://no-such-url');
  } catch(err) {
    alert(err); // TypeError: failed to fetch
  }
}

f();
```

В случае ошибки выполнение try прерывается и управление прыгает в начало блока catch. Блоком try можно обернуть несколько строк:

```
async function f() {

  try {
    let response = await fetch('/no-user-here');
    let user = await response.json();
  } catch(err) {
    // перехватит любую ошибку в блоке try: и в fetch, и в response.json
    alert(err);
  }
}

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. Такие ошибки можно поймать глобальным обработчиком, о чём подробно написано в разделе Промисы: обработка ошибок.

Итого

Ключевое слово async перед объявлением функции:

1. Обязывает её всегда возвращать промис.
2. Позволяет использовать await в теле этой функции.

Ключевое слово await перед промисом заставит JavaScript дождаться его выполнения, после чего:

1. Если промис завершается с ошибкой, будет сгенерировано исключение, как если бы на этом месте находилось throw.
2. Иначе вернётся результат промиса.

Вместе они предоставляют отличный каркас для написания асинхронного кода. Такой код легко и писать, и читать.

Хотя при работе с async/await можно обходиться без promise.then/catch, иногда всё-таки приходится использовать эти методы (на верхнем уровне вложенности, например). Также await отлично работает в сочетании с Promise.all, если необходимо выполнить несколько задач параллельно.