Асинхронность в JS

Frontend. Урок 23

Вопросы на повторение

1. Что такое DOM?
2. Какие есть методы поиска нужного элемента?
3. Какие есть свойства у найденного элемента?
4. Как обработать нажатие на элемент?

Постановка проблемы

Как уйти за значением выражения и вернуться, когда оно будет доступно?

Асинхронность в JS

Чтобы выполнить код, нам нужен JavaScript Engine (движок) — программа, которая «читает и выполняет» то, что мы написали. Самый распространенный движок среди всех — это V8, он используется в Google Chrome и Node.js.

Выполнение JS-кода — однопоточное. Это значит, что в конкретный момент времени движок может выполнять не более одной строки кода. То есть вторая строка не будет выполнена, пока не выполнится первая.

Такое выполнение кода (строка за строкой) называется синхронным. Синхронный код понятный, его удобно читать, потому что он выполняется ровно так, как написан. Никаких сюрпризов: в каком порядке команды указаны — в таком они и выполняются.

Асинхронность в JS

Однако с ним могут возникать некоторые проблемы. Представим, что нам нужно выполнить какую-то операцию, требующую некоторого времени — например, напечатать в консоли приветствие, но не сразу, а через 5 секунд. Ниже псевдокод — синхронная функция задержки delay() вымышленная:

```
function greet() {
  console.log('Hello!')
}
delay(5000)
greet()
```

Мы помним, что выполнение синхронного кода — строка за строкой. То есть пока delay() не выполнится до конца, к следующей строке интерпретатор не перейдёт. Соответственно, пока не пройдет 5 секунд и delay() не выполнится, мы ничего делать не можем.

SetTimeout

Теперь попробуем решить эту же задачу, но так, чтобы наш код не блокировал выполнение. Для этого мы воспользуемся функцией setTimeout():

```
setTimeout(function greet() {
console.log('Hello!')
}, 5000)
```

В этот раз, однако, в эти «5 секунд молчания» мы можем выполнять другие действия.

Стек вызовов

При вызове какой-то функции она попадает в так называемый стек вызовов. Стек — это структура данных, в которой элементы упорядочены так, что последний элемент, который попадает в стек, выходит из него первым (LIFO: last in, first out). Стек похож на стопку книг: та книга, которую мы кладем последней, находится сверху.

В стеке вызовов хранятся функции, до которых дошел интерпретатор, и которые надо выполнить.

В синхронном коде в стеке хранится вся цепочка вызовов.

Пример:

http://latentflip.com/loupe/?code=ZnVuY3Rpb24gbWFpbigpIHsKICBzZXRUaW1lb3V0KGZ1bmN0aW9uIGdyZWV0KCkgewogICAgY29uc29sZS5sb2coJ0hlbGxvIScpCiAgfSwgMjAwMCkKCiAgY29uc29sZS5sb2coJ0J5ZSEnKQp9CgptYWluKCk%3D!!!PGJ1dHRvbj5DbGljayBtZSE8L2J1dHRvbj4%3D

Функции колбэки

Callback (колбэк, функция обратного вызова) — функция, которая вызывается в ответ на совершение некоторого события.
В нашем случае таким событием было срабатывание таймера через 2 секунды, а колбэком — функция greet(). В целом, событием может быть что угодно:
ответ от сервера;
завершение какой-то длительной вычислительной задачи;
получение доступа к каким-то API устройства, на котором выполняется код.
Таким образом колбэк — это первый способ обработать какое-либо асинхронное действие.

Ад колбэков

Однако у колбэков есть неприятный минус, так называемый ад колбэков
Нагляднее всего его можно показать на примере.
Допустим, у нас есть ряд асинхронных задач, которые зависят друг от друга: то есть первая задача запускает по завершении вторую, вторая — третью и т. д.

```
setTimeout(() => {
  setTimeout(() => {
    setTimeout(() => {
      setTimeout(() => {
        console.log('Hello!')
      }, 5000)
    }, 5000)
  }, 5000)
}, 5000)
```

Промисы

Промис (Promise) — специальный объект JavaScript, который используется для написания и обработки асинхронного кода.

Асинхронные функции возвращают объект Promise в качестве значения. Внутри промиса хранится результат вычисления, которое может быть уже выполнено или выполнится в будущем.

Промис может находиться в одном из трёх состояний:

pending — стартовое состояние, операция стартовала;

fulfilled — получен результат;

rejected — ошибка.

У промиса есть методы then() и catch(), которые позволяют использовать результат работы промиса.

Промисы

Промис создаётся с помощью конструктора.
В конструктор передаётся функция-исполнитель асинхронной операции (англ. executor). Она вызывается сразу после создания промиса. Задача этой функции — выполнить асинхронную операцию и перевести состояние промиса в fulfilled (успех) или rejected (ошибка).
Изменить состояние промиса можно, вызвав колбэки, переданные аргументами в функцию:

```
const promise = new Promise(function
(resolve, reject) {
  const data = getData() // делаем
асинхронную операцию: запрос в БД, API,
etc.
  resolve(data) // переводим промис в
состояние fulfilled. Результатом
выполнения будет объект data
})
const errorPromise = new Promise(function
(resolve, reject) {
  reject(new Error('ошибка')) //
переводим промис в состояние rejected.
Результатом выполнения будет объект Error
})
```

Методы промисов

С помощью методов then(), catch() и finally() мы можем реагировать на изменение состояния промиса и использовать результат его выполнения.
Пример ниже показывает состояние промиса, который создаётся при нажатии на кнопку купить. Промис случайным образом завершается успехом или ошибкой:
https://doka.guide/js/promise/demos/Lopinopulos-QWNLMwR/

В работе мы чаще используем промисы, чем создаём. Использовать промис — значит выполнять код при изменении состояния промиса.
Существует три метода, которые позволяют работать с результатом выполнения вычисления внутри промиса:

then()
catch()
finally()

Метод then

Метод then() используют, чтобы выполнить код после успешного выполнения асинхронной операции.
Например, мы запросили у сервера список фильмов и хотим отобразить их на экране, когда сервер получит результат. В этом случае:
асинхронная операция — запрос данных у сервера;
код, который мы хотим выполнить после её завершения, — отрисовка списка.
Метод then() принимает в качестве аргумента две функции-колбэка. Если промис в состоянии fulfilled то выполнится первая функция. Если в состоянии rejected — вторая.

```
fetch(`https://swapi.dev/api/films/${id}/`).then(function (movies) {
  renderList(movies)
})
```

Метод catch

Метод catch() используют, чтобы выполнить код в случае ошибки при выполнении асинхронной операции.
Например, мы запросили у сервера список фильмов и хотим показать экран обрыва соединения, если произошла ошибка. В этом случае:
асинхронная операция — запрос данных у сервера;
код, который мы хотим выполнить при ошибке — экран обрыва соединения.
Метод catch() принимает в качестве аргумента функцию-колбэк, которая выполняется сразу после того, как промис поменял состояние на rejected.
Параметр колбэка содержит экземпляр ошибки:

```
fetch(`https://swapi.dev/api/films/${id}/`).catch(function (error) {
  renderErrorMessage(error)
})
```

Метод finally

Метод finally() используют, чтобы выполнить код при завершении асинхронной операции. Он будет выполнен вне зависимости от того, была ли операция успешной или завершилась ошибкой.
Самый частый сценарий использования finally() — работа с индикаторами загрузки. Перед началом асинхронной операции разработчик включает индикатор загрузки. Индикатор нужно убрать вне зависимости от того, как завершилась операция. Если этого не сделать, то пользователь не сможет взаимодействовать с интерфейсом.
Метод finally() принимает в качестве аргумента функцию-колбэк, которая выполняется сразу после того, как промис поменял состояние на rejected или fulfilled:

```
let isLoading = true
fetch(`https://swapi.dev/api/films/${id}/`).finally(function () {
  isLoading = false
```

Цепочки вызовов

Методы then(), catch() и finally() часто объединяют в цепочки вызовов, чтобы обработать и успешный, и ошибочный сценарии:

```
let isLoading = true

fetch(`https://swapi.dev/api/films/${id}/`)
  .then(function (movies) {
    renderList(movies)
  })
  .catch(function (err) {
    renderErrorMessage(err)
  })
  .finally(function () {
    isLoading = false
  })
```

Async/await

Добавленное перед определением функции ключевое слово async делает функцию асинхронной. Возвращаемое значение такой функции автоматически оборачивается в Promise:

```
async function getStarWarsMovies() {
  return 1
}
console.log(getStarWarsMovies()) // Promise { <state>: "fulfilled", <value>: 1
}
```

Асинхронные функции нужны для выполнения асинхронных операций: работой с API, базами данных, чтения файлов и т.д.

Async/await

Асинхронные операции выполняются не сразу: код отправил запрос к API и ждёт, пока сервер пришлёт ответ. Ключевое слово await используется, чтобы дождаться выполнения асинхронной операции:

```
async function getStarWarsMovie(id) {
  const response = await fetch(`https://swapi.dev/api/films/${id}/`)
  console.log("ответ получен", response) // *1
  return response.json()
}

const movies = getStarWarsMovie(1).then((movie) => {
  console.log(movie.title)
}) // *2
console.log("результат вызова функции", movies) // *3
```

fetch()

С помощью функции fetch() можно отправлять сетевые запросы на сервер — как получать, так и отправлять данные. Метод возвращает промис с объектом ответа, где находится дополнительная информация (статус ответа, заголовки) и ответ на запрос.
Браузер предоставляет глобальный API для работы с запросами и ответами HTTP. Раньше для подобной работы использовался XMLHttpRequest, однако fetch() более гибкая и мощная альтернатива, он понятнее и проще в использовании из-за того, что использует Promise.

Функция fetch() принимает два параметра:
url — адрес, по которому нужно сделать запрос;
options (необязательный) — объект конфигурации, в котором можно настроить метод запроса, тело запроса, заголовки и многое другое.

fetch()

По умолчанию вызов fetch() делает GET-запрос по указанному адресу. Базовый вызов для получения данных можно записать таким образом:

fetch('http://jsonplaceholder.typicode.com/posts')

Результатом вызова fetch() будет Promise, в котором будет содержаться специальный объект ответа Response. У этого объекта есть два важных для нас поля:

ok — принимает состояние true или false и сообщает об успешности запроса;

json — метод, вызов которого, возвращает результат запроса в виде json.

В следующем примере используем .then() - обработчик результата, полученного от асинхронной операции.

```
fetch('http://jsonplaceholder.typicode.com/posts')
  // функция then вернет другой промис (их можно чейнить). Когда отрезолвится промис (r.json()), который вернула функция then, будет вызван следующий колбек в цепочке
  .then((response) => response.json())
  // Получим ответ в виде массива из объектов [{...}, {...}, {...}, ...]
```

Практика A

```
// В каком порядке будут выведены консоли и какие именно?
const p = new Promise((resolve, reject) => {
  reject(Error('Всё сломалось :('));
})
  .catch((error) => console.log('1-я', error.message))
  .catch((error) => console.log('2-я', error.message));
```

```
const p2 = new Promise((resolve, reject) => {
  reject(Error('Всё сломалось :('));
});
p2.catch((error) => console.log('3-я', error.message));
p2.catch((error) => console.log('4-я', error.message));
```

```
const p3 = new Promise((resolve, reject) => {
  reject(Error('Всё сломалось :('));
})
  .then((error) => console.log('5-я', error.message))
  .catch((error) => console.log('6-я', error.message));
```

Практика B

```
// в каком порядке будут выведены консоли и что в них будет?
setTimeout(() => {
  console.log('timeout')
}, 0);

const p = new Promise((resolve, reject) => {
  console.log('Promise creation');
  resolve()
})

const p2 = new Promise((resolve, reject) => {
  console.log(123)
})

p.then(() => {
  console.log('Promise resolving');
})
console.log('End')
console.log('p2 =>>', p2)
```

Практика С

```
// todo в каком порядке будут выведены консоли и что в них будет?
console.log('script start')

setTimeout(function() {
  console.log('setTimeout');
}, 0);

Promise
  .resolve()
  .then(function() {
  console.log('promise1');
})
  .then(function() {
  console.log('promise2');
});

console.log('script end');
```

Вопросы на закрепление

1. Что такое асинхронность?
2. Что такое Promis?
3. Для чего существует async/await?
4. Для чего используется fetch?