Front-end

JavaScript

JS. try catch, async await, callback, event loop

- async await
- try catch
- callback
- event loop

JS. try catch, async await, callback, event loop

Async/await — это специальный синтаксис, который предназначен для более простого и удобного написания асинхронного кода. Появился он в языке, начиная с ES2017 (ES8).

Синтаксис **«async/await»** упрощает работу с промисами (позволяет асинхронный код записывать синхронным способом).

JS. try catch, async await, callback, event loop

Асинхронные функции

Асинхронные функции — это такие, которые объявлены с использованием ключевого слова async.

```
// асинхронная функция delay
async function delay() {
  // ...
}
// асинхронная функция как часть выражения
const wait = async function() {
  // ...
}
// стрелочная запись асинхронной функции sleep
const sleep = async() => {
  // ...
}
```

JS. try catch, async await, callback, event loop

Асинхронные функции

Движок JavaScript при этом не блокируется и может выполнять другой код. Как только ответ получен, выполнение кода продолжается.

Ключевые слова **async/await** не привносят в JavaScript что-то новое. Они только упрощают работу с Promise.

JS. try catch, async await, callback, event loop

Асинхронные функции

Await

await – это ключевое слово, которое используется в асинхронных функциях для того, чтобы указать, что здесь необходимо дождаться завершения промиса. Таким образом await, указанный перед промисом запрещает интерпретатору перейти к следующей строчке кода, пока он не выполнится.

Ключевое слово await работает только внутри асинхронных функций. Вызов его вне функции будет синтаксической ошибкой

JS. try catch, async await, callback, event loop

Асинхронные функции

Не смотря на то, что await заставляет интерпретатор остановиться и дождаться завершения промиса, движок JavaScript в это время может выполнять другие задачи. Это достигается благодаря тому, что асинхронный код выполняется в фоне и не блокирует основной поток. Таким образом, await не приводит к «зависанию» страницы и не мешает пользователю взаимодействовать с ней.

По сути, «async/await» – это просто удобный способ работать с промисами.

JS. try catch, async await, callback, event loop

Обработка ошибок

Обработать потенциальные ошибки в асинхронной функции можно с помощью **try..catch**. Для этого этот блок кода (в котором используется await) необходимо заключить в эту конструкцию.

JS. try catch, async await, callback, event loop

Обработка ошибок

Чтобы использовать try...catch, необходимо в блоке try написать код, который нужно исполнить, а в блоке catch написать, что делать в случае ошибки.

```
try {

  someFunction()

  anotherFunction()

} catch (err) {

  console.log('Поймали ошибку! Вот она: ', err.message)

}
```

Если в блоке try не произошло ошибок, то код в блоке catch не выполнится.

Важно помнить, что код в try должен быть синтаксически верным. Если написать невалидный код (например, не закрыть фигурные скобки), то скрипт не запустится, потому что JavaScript не поймёт код. Ошибки, которые обработает блок catch, будут ошибками во время выполнения программы.

JS. try catch, async await, callback, event loop

Обработка ошибок

В случае ошибки выполнение в блоке try прерывается и сразу же переходит в блок catch . После него скрипт продолжит своё выполнение, как и прежде.

JS. try catch, async await, callback, event loop

Обработка ошибок

finally

Рассмотрим ситуацию, когда в случае успеха или неудачи выполнения какого-то участка кода нам необходимо проводить какие-то действия, чтобы корректно завершить работу скрипта.

JS. try catch, async await, callback, event loop

Обработка ошибок

```
try {
  // подключаемся к вебсокету, но в конце нужно обязательно отключиться
  webSocket.connect('ws://....')
  callMayThrowError()
} catch (err) {
  ...
}

// Пробуем отключаться после try...catch
webSocket.disconnect('ws://....')
```

Казалось бы никаких проблем с этим кодом быть не должно, ведь неважно выполнится код в блоке try правильно или попадёт в catch, следующая строчка должна выполниться. Однако возможна ситуация, что в блоке catch тоже возникнет ошибка, и тогда выполнение следующей строчки уже не случится.

JS. try catch, async await, callback, event loop

Обработка ошибок

В конструкцию try...catch можно добавить блок finally, который выполнится после блоков try и catch. Неважно какой код выполнился в предыдущих блоках, после их завершения (даже если из catch была выброшена новая ошибка) исполнится код в блоке finally.

```
try {
  webSocket.connect('ws://....')
  callMayThrowError()
} catch (err) {
doSomeWithError(err) // Здесь тоже может возникнуть ошибка
} finally {
webSocket.disconnect('ws://....') // Выполнится всегда
}
```

JS. try catch, async await, callback, event loop

Обработка ошибок

Наличие блока finally необязательно. finally можно использовать и без блока catch.

```
try {
  // Отправить данные на сервер, здесь нам неважна обработка ошибки
  sendData()
} finally {
  // Закрыть соединение при любом результате
  closeConnection()
}
```

JS. try catch, async await, callback, event loop

Обработка ошибок: пример

```
async function getUser() {
 try {
   const response = await fetch(url);
   const data = await response.json();
 } catch(e) {
   // если что-то пойдёт не так на каком-то этапе в блоке try, то мы автоматически попадём в метод catch()
   console.error(e);
 }
}
```

JS. try catch, async await, callback, event loop

Обработка ошибок: пример

Если нужно с finally, то так:

```
async function getUser() {
 try {
   const response = await fetch(url);
   const data = await response.json();
 } catch(e) {
   // если что-то пойдёт не так на каком-то этапе в блоке try, то мы автоматически попадём в метод catch()
   console.error(e);
 } finally {
   // выполнится в любом случае, в независимости от того произошла ошибка или нет
 }
}
```

JS. try catch, async await, callback, event loop

Обратный вызов (или callback) — это функция передаваемая в качестве аргумента другой функции. Функция, получающая обратный вызов, решает, выполнять ли обратный вызов и когда:

```
function myFunction(callback) {
  // 1. Что-то делает
  // 2. Затем выполняет обратный вызов
  callback()
}

function myCallback() {
  // Делает что-то другое
}

myFunction(myCallback);
```

JS. try catch, async await, callback, event loop

Вы часто будет слышать, что JavaScrip является однопоточным. Это означает, что он может делать одну вещь за раз. При выполнении медленной операции, такой как получение данных из удалённого API, это может быть проблематичным. Было бы не очень приятно, если бы ваша программа зависла до тех пор, пока данные не будут возвращены.

Один из способов, которым JavaScript позволяет избежать этого узкого места, — использование обратного вызова. Мы можем передать вторую функцию в качестве аргумента функции, отвечающей за выборку данных. Затем запускается запрос на получение данных, но вместо ожидания интерпретатор JavaScript продолжает выполнение остальной части программы. Когда приходит ответ от API, функция обратного вызова выполняется и может выполнять действия с результатом

JS. try catch, async await, callback, event loop

```
function fetchData(url, cb) {
  // 1. Выполняет запрос к API по URL
  // 2. Если ответ успешный, выполнить обратный вызов
  cb(res);
}

function callback(res) {
  // Сделать что-то с результатом
}

// Сделать что-то
fetchData('https://sitepoint.com', callback);
// Сделать что-то ещё
```

JS. try catch, async await, callback, event loop

Асинхронные функции обратного вызова

В отличие от синхронного кода, асинхронный код JavaScript не будет выполняться сверху вниз, строка за строкой. Вместо этого асинхронная операция зарегистрирует функцию обратного вызова, которая будет выполнена после её завершения. Это означает, что интерпретатору JavaScript не нужно завершения асинхронной операции, а вместо этого он может выполнять другие задачи во время её выполнения.

Одним из основных примеров асинхронного кода является получение данных из удалённого API. Давайте посмотрим на пример и разберём как он использует обратные вызовы.

JS. try catch, async await, callback, event loop

Асинхронные функции обратного вызова

```
<div class="mb-3 users">
  <button type="submit" class="btn btn-primary">Fetch Users</button>
  <ul id="result"></ul>
</div>

.users {
  padding: 15px;
}

ul {
  margin-top: 15px;
}
```

JS. try catch, async await, callback, event loop

Асинхронные функции обратного вызова

```
const button = document.querySelector('button');
const ul = document.querySelector('ul');

button.addEventListener('click', (e) => {
 e.preventDefault();
 fetch('https://jsonplaceholder.typicode.com/users')
  .then(response => response.json())
  .then(json => {
   const names = json.map(user => user.name);
   names.forEach(name => {
    const li = document.createElement('li');
    li.textContent = name;
    ul.appendChild(li);
   });
  })
});
```

JS. try catch, async await, callback, event loop

Асинхронные функции обратного вызова

Изначально колбэки были единственным способом работать с асинхронным кодом в JavaScript.

Однако у колбэков есть неприятный минус, так называемый ад колбэков (callback hell).

```
function request(url, onSuccess) {
 /*...*/
}
request('/api/users/1', function (user) {
  request(`/api/photos/${user.id}/`, function (photo) {
    request(`/api/crop/${photo.id}/`, function (response) {
      console.log(response)
    })
  })
})
```

JS. try catch, async await, callback, event loop

Асинхронные функции обратного вызова

Читать такое сложно, не говоря уже о тестировании, которое здесь становится очень накладным.

Решить эту проблему были призваны Промисы (Promise).

JS. try catch, async await, callback, event loop

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 кладём последней, находится сверху.

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

```
function outer() {

  function inner() {

console.log('Hello!') // Функция 3

  }

inner() // Функция 2

}

outer() // Функция 1
```

JS. try catch, async await, callback, event loop

Стек вызовов

После выполнения всего блока стек станет пустым.

В синхронном коде в стеке хранится вся цепочка вызовов. Поэтому, например, рекурсия без базового случая может приводить к переполнению стека — в нём скапливается слишком большое количество вызовов.

Теперь посмотрим, как ведёт себя стек вызовов при работе с асинхронным кодом:

```
function main() {
  setTimeout(function greet() {
    console.log('Hello!')
  }, 2000)

  console.log('Bye!')
}
main()
```

JS. try catch, async await, callback, event loop

Цикл событий

setTimeout() — это не JavaScript!

Функция setTimeout() не является частью JavaScript-движка, это по сути Web API, включённое в среду браузера как дополнительная функциональность.

Эта дополнительная функциональность (Web API) берёт на себя работу с таймерами, интервалами, обработчиками событий. То есть когда мы регистрируем обработчик клика на кнопку — он попадает в окружение Web API. Именно оно знает, когда обработчик нужно вызвать.

Управление тем, как должны вызываться функции Web API, берёт на себя цикл событий (Event loop).

JS. try catch, async await, callback, event loop

Event loop

Цикл событий отвечает за выполнение кода, сбор и обработку событий и выполнение подзадач из очереди.

Именно цикл событий ответственен за то, что setTimeout() пропал из стека в прошлом примере. Чтобы увидеть картину целиком, давайте включим в нашу схему все недостающие части.

JS. try catch, async await, callback, event loop

Заметьте, что стек вызовов и очередь задач называются именно стеком и очередью. Потому что вызовы из стека работают по принципу «последний зашёл, первый вышел» (LIFO: last in, first out), а в очереди — по принципу «первый зашёл, первый вышел» (FIFO: first in, first out).

Очередь — структура данных, в которой элементы упорядочены так, что первый попавший в очередь элемент покидает её первым.

Таким образом цикл событий работает с асинхронным кодом — то есть таким, который выполняется не построчно.