

ПМ3 Разработка модулей ПО.

РО 3.1 Понимать и применять принципы объектно-ориентированного и асинхронного программирования.

Тема 2. Асинхронно программирование.

Лекция 12. Async/Await и обработка ошибок.

Цель занятия:

**Познакомиться с синтаксисом
async/await для работы с
асинхронным кодом, научиться
правильно обрабатывать ошибки с
помощью try/catch.**

Учебные вопросы:

1. Что такое `async` и `await`: синтаксис и назначение.
2. Как `async`-функция возвращает значения (автоматическое обёртывание в `Promise`).
3. Использование `await` для последовательного выполнения асинхронных операций.
4. Обработка ошибок: конструкция `try/catch` в асинхронных функциях.
5. Сравнение стиля кода: промисы (`.then/.catch`) и `async/await`.
6. Практические примеры: переписывание кода с промисов на `async/await`.

1. Что такое `async` и `await`: синтаксис и назначение.

Проблема, которую решают `async/await`.

До появления `async/await` асинхронный код в JavaScript писали:

- через колбэки (callback hell),
- через промисы и `.then()`-цепочки.

Оба подхода работают, но код часто получается сложным для чтения.

Синтаксис `async/await` позволяет писать асинхронный код в стиле последовательного (синхронного) кода, делая его понятнее.

Ключевое слово `async`.

Используется для объявления асинхронной функции.

Любая `async`-функция всегда возвращает промис.

Если функция возвращает значение, оно автоматически оборачивается в `Promise.resolve()`.

👉 Пример:

```
async function hello() {  
  return "Привет!";  
}  
hello().then(result => console.log(result));  
// Выведет: "Привет!"
```

📌 Здесь функция вернула строку, но на самом деле `hello()` вернула **промис**, который успешно завершился этим значением.

Ключевое слово `await`.

Работает только внутри `асync-функций`.

Приостанавливает выполнение функции, пока промис не выполнится.

Возвращает результат промиса.

Пример:

```
function delay(ms) {  
  return new Promise(resolve => setTimeout(resolve, ms));  
}  
  
async function run() {  
  console.log("Начало");  
  await delay(2000);    // ждём 2 секунды  
  console.log("Прошло 2 секунды");  
}  
  
run();
```

 В момент вызова **await** функция ждёт промис, но поток JavaScript при этом не блокируется — другие задачи продолжают выполняться.

Совместное использование.

Обычно `async` и `await` используются вместе:

- `async` делает функцию асинхронной и возвращает промис.
- `await` позволяет пошагово «распаковывать» результаты асинхронных операций.

👉 Пример:

```
async function getData() {  
  let value1 = await Promise.resolve(10);  
  let value2 = await Promise.resolve(20);  
  console.log("Сумма:", value1 + value2);  
}
```

```
getData();
```

```
// Сумма: 30
```

Назначение `async/await`.

- Делает асинхронный код простым и читаемым.
- Убирает «лесенку» из `.then()`.
- Упрощает обработку ошибок (с помощью `try/catch`).
- Подходит как для последовательных, так и для параллельных операций.

Вывод:

- `async` и `await` — это синтаксический сахар над промисами.
- `async` превращает функцию в асинхронную и гарантирует возврат промиса.
- `await` позволяет писать асинхронный код в виде «обычного последовательного кода», делая программы понятнее и удобнее для поддержки.

2. Как `async`-функция возвращает значения.

Особенность `async`-функции.

Любая функция, объявленная с ключевым словом `async`, всегда возвращает промис.

Даже если в ней явно вернуть обычное значение, оно автоматически будет обернуто в `Promise.resolve()`.

Возврат обычного значения

👉 Пример:

```
async function f() {  
  return 42;  
}  
  
f().then(result => console.log(result));  
// Выведет: 42
```

📌 На самом деле `f()` возвращает промис, который завершается успешно и содержит число 42.

Возврат промиса.

Если `async`-функция возвращает промис, он используется напрямую.

Пример:

```
async function g() {  
  return Promise.resolve("Привет");  
}  
g().then(result => console.log(result));  
// Выведет: "Привет"
```

 Здесь значение не оборачивается повторно — используется исходный промис.

Поведение при ошибке.

Если внутри аsync-функции выбросить исключение (throw), оно превращается в отклонённый промис (Promise.reject).

👉 Пример:

```
async function h() {  
  throw new Error("Ошибка!");  
}  
  
h().catch(err => console.error(err.message));  
// Выведет: "Ошибка!"
```

📌 Таким образом, любое исключение автоматически конвертируется в отклонённый промис.

Итог.

- `async`-функция гарантированно возвращает промис.
- Если возвращается обычное значение → оно становится результатом успешно выполненного промиса.
- Если возвращается промис → он используется как есть.
- Если возникает ошибка → она превращается в `Promise.reject`.



Вывод:

- `async` делает поведение функций предсказуемым: всё, что они возвращают, всегда становится промисом.
- Это позволяет единообразно работать с асинхронным кодом — через `.then()` или с помощью `await`.

3. Использование `await` для последовательного выполнения асинхронных операций.

Проблема параллельных задач.

При использовании промисов часто приходится писать цепочки `.then()`, которые быстро становятся неудобными и плохо читаемыми.

С помощью `await` можно выполнять асинхронные шаги последовательно, почти как обычные инструкции.

Как работает await?

Ключевое слово `await` используется только внутри асинхронных функций.

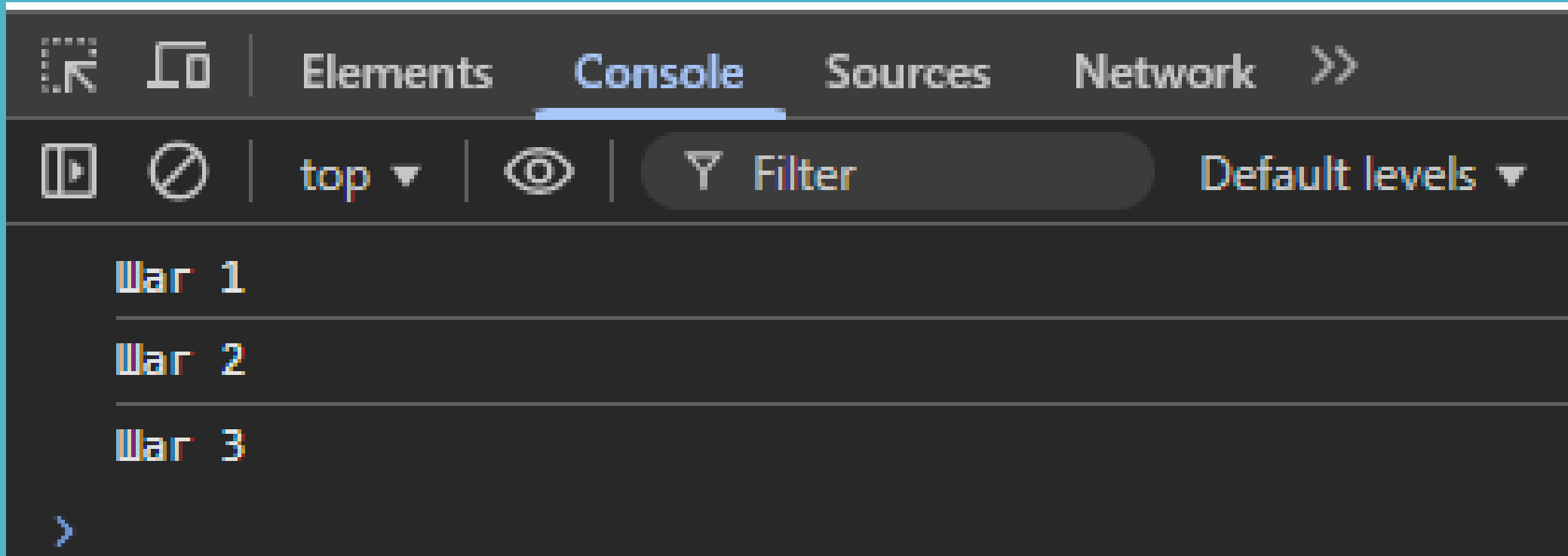
Оно «останавливает» выполнение функции до тех пор, пока промис не завершится.

При этом главный поток JavaScript не блокируется — выполняются другие задачи.

Пример: шаги с задержкой.

```
function delay(ms, value) {  
  return new Promise(resolve => setTimeout(() => resolve(value), ms));  
}  
  
async function runSteps() {  
  const step1 = await delay(1000, "Шаг 1");  
  console.log(step1);  
  const step2 = await delay(1000, "Шаг 2");  
  console.log(step2);  
  const step3 = await delay(1000, "Шаг 3");  
  console.log(step3);  
}  
  
runSteps();
```


 Здесь каждое выполнение дожидается предыдущего.
Результат:



Сравнение с цепочкой промисов

Тот же пример без await выглядел бы так:

```
delay(1000, "Шаг 1")
  .then(res => {
    console.log(res);
    return delay(1000, "Шаг 2");
  })
  .then(res => {
    console.log(res);
    return delay(1000, "Шаг 3");
  })
  .then(res => console.log(res));
```

 Функционально результат тот же, но async/await выглядит компактнее и понятнее.

Пример с вычислениями:

```
async function calculate() {  
  let a = await Promise.resolve(5);  
  let b = await Promise.resolve(10);  
  let sum = a + b;  
  console.log("Сумма:", sum);  
}
```

```
calculate();  
// Сумма: 15
```


Когда использовать?

- Когда нужно выполнить задачи строго по порядку (например, запрос → обработка → сохранение).
- Когда важна читаемость и простота кода.
- В сочетании с `try/catch` для удобной обработки ошибок.

Вывод:

- `await` позволяет «развернуть» промисы в пошаговый код.
- Асинхронные операции выполняются последовательно, и это делает код проще, чем длинные цепочки `.then()`.

4. Обработка ошибок: конструкция `try/catch` в асинхронных функциях.

Проблема обработки ошибок.

В цепочках промисов ошибки перехватываются через `.catch()`.

При использовании `async/await` удобно применять знакомую конструкцию `try/catch`, которая делает код более «синхронным» на вид.

Синтаксис:

```
async function example() {  
  try {  
    // код, который может вызвать ошибку  
  } catch (error) {  
    // обработка ошибки  
  }  
}
```

Простой пример:

```
async function getData() {  
  try {  
    let result = await Promise.reject("Что-то пошло не так");  
    console.log("Результат:", result);  
  } catch (err) {  
    console.error("Ошибка перехвачена:", err);  
  }  
}  
getData();
```




► Ошибка перехвачена: Что-то пошло не так

 Здесь `Promise.reject` выбрасывает ошибку, и она сразу попадает в блок `catch`.

Несколько await в одном try.

Можно использовать один блок try/catch для целой последовательности асинхронных операций.

```
async function process() {  
  try {  
    let user = await Promise.resolve("Пользователь");  
    console.log("Загружен:", user);  
  
    let details = await Promise.reject("Нет доступа к данным");  
    console.log("Детали:", details); // не выполнится  
  } catch (error) {  
    console.error("Ошибка во время обработки:", error);  
  }  
}  
process();
```

 Как только ошибка возникла, выполнение внутри try прерывается, управление передаётся в catch.

Локальная обработка ошибок.

Иногда удобно использовать несколько блоков try/catch для разных этапов.

```
async function pipeline() {  
  try {  
    let step1 = await Promise.resolve("Шаг 1 завершён");  
    console.log(step1);  
  } catch (e) {  
    console.error("Ошибка на шаге 1:", e);  
  }  
  try {  
    let step2 = await Promise.reject("Сбой на шаге 2");  
    console.log(step2);  
  } catch (e) {  
    console.error("Ошибка на шаге 2:", e);  
  }  
}  
pipeline();
```

Совместимость с .catch()

async-функция всегда возвращает промис, поэтому можно комбинировать try/catch и .catch().

```
async function task() {  
  throw new Error("Ошибка внутри async");  
}  
task()  
  .then(() => console.log("OK"))  
  .catch(err => console.error("Перехвачено через .catch:", err.message));
```

Когда использовать try/catch?

- Когда нужно перехватить ошибку в нескольких await подряд.
- Когда ошибки нужно обрабатывать поэтапно.
- Когда код должен оставаться читаемым и линейным.

Вывод:

- Обработка ошибок в async/await выполняется привычным способом — через try/catch.
- Это делает код более ясным по сравнению с .catch(), особенно если есть несколько последовательных шагов.


5. Сравнение стиля кода: промисы (.then/.catch) и async/await.

Работа с промисами через .then()/.catch()

Когда мы используем промисы в «чистом» виде, код строится на последовательности методов .then() и .catch().

Пример:

```
function delay(ms, value) {  
  return new Promise(resolve => setTimeout(() => resolve(value), ms));  
}  
delay(1000, "Шаг 1")  
  .then(res => {  
    console.log(res);  
    return delay(1000, "Шаг 2");  
  })  
  .then(res => {  
    console.log(res);  
    return delay(1000, "Шаг 3");  
  })  
  .then(res => {  
    console.log(res);  
  })  
  .catch(err => {  
    console.error("Ошибка:", err);  
  });
```


 Здесь шаги выполняются последовательно, но из-за вложенности `.then()` код выглядит как «лесенка».

Работа с `async/await`

Те же операции можно выразить через `async/await`, и код становится похож на обычный синхронный.

Пример:

```
function delay(ms, value) {  
  return new Promise(resolve => setTimeout(() => resolve(value), ms));  
}  
  
async function runSteps() {  
  try {  
    let step1 = await delay(1000, "Шаг 1");  
    console.log(step1);  
  
    let step2 = await delay(1000, "Шаг 2");  
    console.log(step2);  
  
    let step3 = await delay(1000, "Шаг 3");  
    console.log(step3);  
  } catch (err) {  
    console.error("Ошибка:", err);  
  }  
}  
  
runSteps();
```

 Теперь структура линейная: каждый шаг выполняется после предыдущего, а ошибки ловятся через привычный try/catch.

Сравнение читаемости.

`.then/.catch`:

- Хорошо видно, где каждый шаг обрабатывает результат.
- Код становится менее читаемым при большом количестве последовательных операций.
- Более естественен для параллельных операций (`Promise.all`, `Promise.race`).

`async/await`:

- Выглядит как синхронный код → легче понимать.
- Удобен для длинных последовательностей операций.
- Ошибки удобно обрабатывать через `try/catch`.
- Иногда приходится комбинировать с методами промисов (`Promise.all`) для параллельных действий.

Когда использовать.

- Если задачи последовательные и важна читаемость → лучше `async/await`.
- Если задачи параллельные → часто проще и короче через `Promise.all + .then()`.

В реальных проектах оба подхода часто сочетаются.



Вывод:

- `async/await` — это синтаксический сахар над промисами.
- Код с `async/await` проще и ближе к привычному пошаговому стилю, тогда как `.then/.catch` остаётся полезным для параллельных операций и более низкоуровневой работы с промисами.

Контрольные вопросы:

- Что возвращает функция, объявленная с ключевым словом `async`?
- В чём отличие работы с асинхронными операциями через `.then()` и через `await`?
- Что произойдёт, если внутри `async`-функции выбросить исключение?
- Как правильно перехватить ошибку в `async`-функции?
- Можно ли использовать `await` вне `async`-функции?
- Почему `await` «останавливает» выполнение функции, но не блокирует весь поток JavaScript?
- Как можно переписать цепочку промисов на `async/await`?
- В чём преимущества `async/await` по сравнению с использованием только промисов?

Домашнее задание:

1. <https://ru.hexlet.io/courses/js-asynchronous-programming>

13 new Promise

Учимся создавать промисы из колбеков

14 Async/Await

Знакомимся с самым современным способом писать асинхронный код как синхронный

2. Повторить материал лекции.

Материалы лекций:

<https://github.com/ShViktor72/Education2025>

Обратная связь:

colledge20education23@gmail.com