ПМ3 Разработка модулей ПО.

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

Тема 2. Асинхронно программирование.

Лекция 11. Цепочки промисов и Promise.all.

Цель занятия:

Понять, как использовать цепочки промисов для последовательной обработки асинхронных операций и как обрабатывать несколько параллельных операций с помощью Promise.all.

Учебные вопросы:

1. Что такое цепочка промисов и как она работает?

2. Promise.all. Обработка нескольких параллельных промисов.

3. Обработка ошибок в цепочках промисов.

4. Promise.race, Promise.allSettled, Promise.any.

1. Что такое цепочка промисов и как она работает?

Promise — это объект в JavaScript, который представляет результат асинхронной операции:

- ожидание (pending),
- успех (fulfilled),
- ошибка (rejected).

Чтобы обработать результат, используется метод .then() (для успешного результата) и .catch() (для ошибки).

Цепочка промисов — это последовательность вызовов .then(), где каждый следующий .then() получает результат из предыдущего.

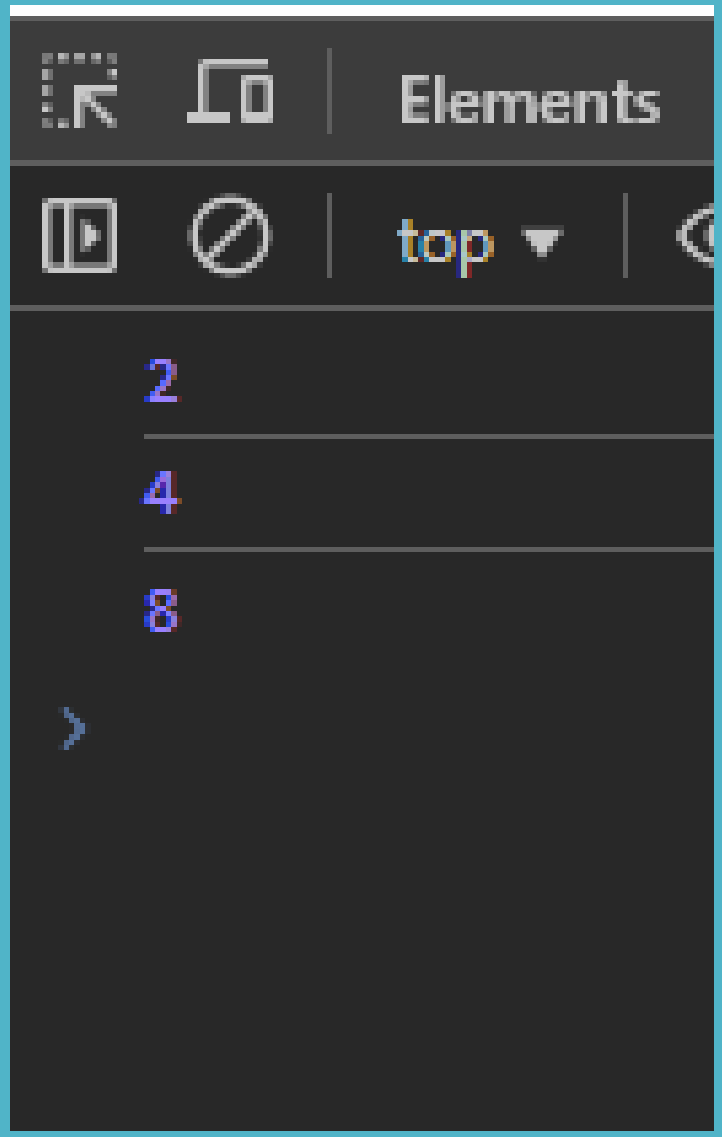
Таким образом можно выполнять несколько асинхронных операций по порядку, не погружаясь в «ад колбэков».

Принцип работы:

- Каждый вызов .then() возвращает новый промис.
- Результат функции внутри .then() передаётся в следующий .then().
- Если внутри .then() возвращается обычное значение → оно автоматически «оборачивается» в промис.
- Если возвращается другой промис → выполнение «ждёт» его завершения.

Пример цепочки. Здесь каждый .then() получает результат из предыдущего и передаёт его дальше.

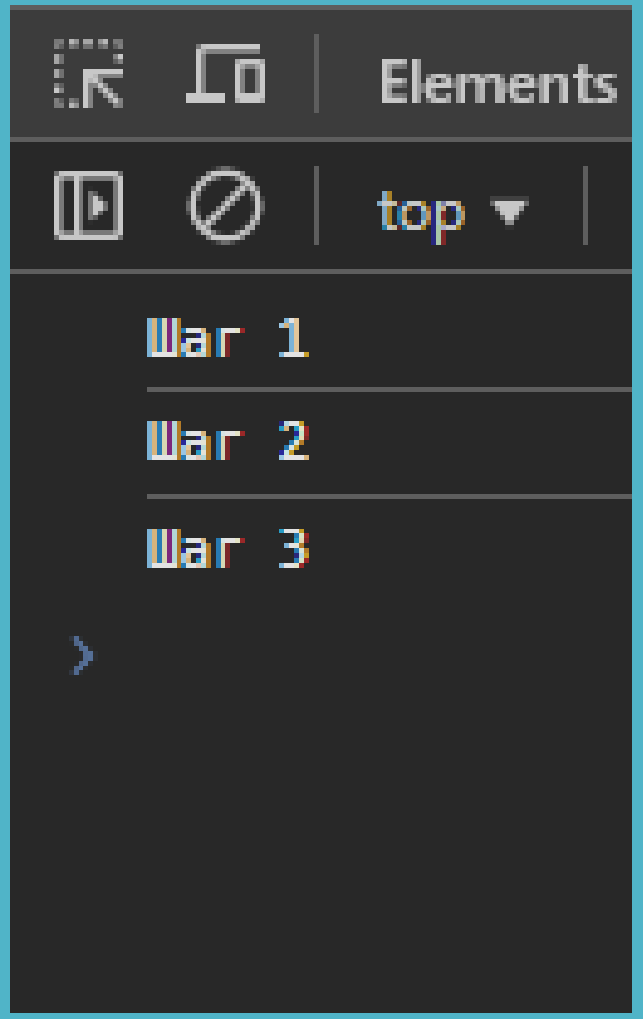
```
new Promise((resolve) => {
  resolve(2);
})
  .then((num) => {
    console.log(num); // 2
    return num * 2;
  })
  .then((num) => {
    console.log(num); // 4
    return num * 2;
  })
  .then((num) => {
    console.log(num); // 8
  });
```



Асинхронный пример. Здесь шаги выполняются последовательно, несмотря на асинхронные задержки.

```
function delay(ms) {
  return new Promise((resolve) => setTimeout(resolve, ms));
}

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



Зачем нужны цепочки:

- Для последовательного выполнения асинхронных задач.
- Чтобы упростить код и избежать вложенных колбэков («callback hell»).
- Для удобной передачи результатов от одной операции к следующей.

✅ Вывод:

Цепочка промисов — это механизм, позволяющий выполнять асинхронные операции шаг за шагом, передавая результат от одного .then() к другому.

Каждый .then() создаёт новый промис и может возвращать как значение, так и другой промис.

2. Promise.all. Обработка нескольких параллельных промисов.

Часто нужно запустить несколько асинхронных операций **параллельно** (например, загрузить данные с разных серверов или выполнить несколько задержек), а затем дождаться, когда все они завершатся, прежде чем продолжить.

Обычная цепочка промисов (.then) решает только последовательное выполнение. Для параллельных операций она неудобна.

Что такое Promise.all?

Promise.all — это статический метод класса Promise, который принимает массив (или другой итерируемый объект) промисов и возвращает новый промис.

Если все промисы успешно завершились, то возвращается массив их результатов в том же порядке, в котором они были переданы.

Если хотя бы один промис завершился с ошибкой, весь Promise.all завершается с ошибкой (reject).

Синтаксис:

```
Promise.all([promise1, promise2, promise3])
  .then((results) => {
    console.log(results); // массив результатов
  })
  .catch((error) => {
    console.error(error); // ошибка, если хотя бы один промис отклонён
  });
```

Пример работы. Три асинхронные операции выполняются параллельно. Результат вернётся только тогда, когда завершатся все три задержки.

```
function delay(ms, value) {
  return new Promise((resolve) => {
    setTimeout(() => resolve(value), ms);
  });
}
Promise.all([
  delay(1000, "A"),
  delay(2000, "B"),
  delay(1500, "C")
])
  .then((results) => {
    console.log(results); // ["A", "B", "C"]
  });
```

Когда полезно использовать Promise.all:

- Загрузка нескольких независимых ресурсов (например, разные API-запросы).
- Одновременный запуск нескольких асинхронных вычислений.
- Ожидание группы промисов, когда нужно продолжить только после их общего завершения.

Особенности:

- Порядок результатов сохраняется (по порядку промисов в массиве, а не по времени их выполнения).
- Если один промис завершился с ошибкой → весь Promise.all возвращает эту ошибку.
- Если нужно дождаться выполнения всех промисов, даже с ошибками, используют Promise.allSettled.

✅ Вывод: Promise.all позволяет запускать несколько промисов парал

3. Обработка ошибок в цепочках промисов.

Ошибки в Promise.all:

- Если хотя бы один промис завершится с ошибкой, весь Promise.all переходит в состояние rejected.
- Ошибка будет передана в .catch().
- Успешные результаты других промисов при этом игнорируются.

Пример:

```
function delay(ms, value, shouldFail = false) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      if (shouldFail) {
        reject(new Error("Ошибка: " + value));
      } else {
        resolve(value);
      }
    }, ms);
  });
}
```

В этом примере в .catch() попадёт ошибка из второго промиса, и весь Promise.all завершится с ней.

```
Promise.all([
  delay(1000, "A"),
  delay(2000, "B", true), // этот промис упадёт
  delay(1500, "C")
])
  .then((results) => {
    console.log("Все результаты:", results);
  })
  .catch((err) => {
    console.error("Произошла ошибка:", err.message);
  });
```

Особые случаи:

- Если нужно получить результаты всех промисов, даже с ошибками, используют Promise.allSettled.
- Иногда полезно в .catch() возвращать запасное значение, чтобы цепочка не прерывалась полностью.

Рекомендации:

- Всегда добавляйте .catch() в цепочки промисов для отладки и корректной обработки ошибок.
- При использовании Promise.all учитывайте, что падение одного промиса обрушит весь результат.
- Для «устойчивых» сценариев (например, загрузка сразу многих данных, где можно пропустить часть ошибок) применяйте Promise.allSettled.

✅ **Вывод:**

Ошибки в промисах обрабатываются с помощью .catch(). В цепочках .catch() перехватывает ошибку из любого предыдущего шага и позволяет продолжить выполнение. В Promise.all ошибка в одном промисе делает отклонённым весь результат — для более гибкой работы стоит использовать Promise.allSettled.

4. Promise.race, Promise.allSettled, Promise.any.

Promise.race.

- Принимает массив (или другой итерируемый объект) промисов.
- Возвращает первый завершившийся промис — не важно, был он успешным или с ошибкой.

Пример:

```
const p1 = new Promise((resolve) => setTimeout(() => resolve("Первый"), 1000));
const p2 = new Promise((resolve) => setTimeout(() => resolve("Второй"), 2000));

Promise.race([p1, p2])
  .then((result) => console.log("Результат:", result));
// Выведет: "Первый" (через 1 секунду)
```

Используется, когда нужно получить **самый быстрый результат** (например, при выборе самого быстрого сервера).

Promise.allSettled.

- Принимает массив промисов.
- Возвращает промис, который всегда выполняется успешно, когда завершатся все промисы (успешно или с ошибкой).
- Результат — массив объектов формата:

{ status: "fulfilled", value: ... }

{ status: "rejected", reason: ... }

Полезно, когда нужны результаты **всех операций**, даже если некоторые упали (например, загрузка набора файлов, где часть может не загрузиться).

Пример:

```
const p1 = Promise.resolve(10);
const p2 = Promise.reject("Ошибка");
const p3 = Promise.resolve(30);

Promise.allSettled([p1, p2, p3])
  .then((results) => console.log(results));
```

```
▼ (3) [{…}, {…}, {…}] i
  ▶ 0: {status: 'fulfilled', value: 10}
  ▶ 1: {status: 'rejected', reason: 'Ошибка'}
  ▶ 2: {status: 'fulfilled', value: 30}
```

Promise.any:

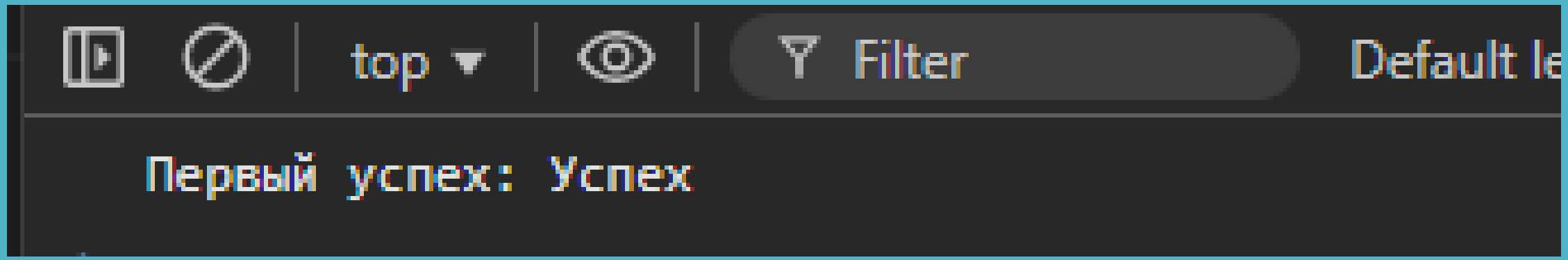
- Принимает массив промисов.
- Возвращает первый успешно завершившийся промис.
- Ошибки игнорируются, пока хотя бы один промис не выполнится успешно.
- Если все промисы завершились с ошибкой → возвращает ошибку типа AggregateError.

Используется, когда нужно дождаться **первого успешного результата**, а остальные можно игнорировать (например, когда делаем несколько запросов на разные источники данных и берём первый успешный).

Пример:

```
const p1 = Promise.reject("Ошибка 1");
const p2 = new Promise((resolve) => setTimeout(() => resolve("Успех"), 1000));
const p3 = Promise.reject("Ошибка 2");

Promise.any([p1, p2, p3])
  .then((result) => console.log("Первый успех:", result))
  .catch((err) => console.error("Все промисы упали:", err));
```



Сравнение методов:

Метод	Успешное завершение	Ошибка
Promise.all	Все промисы успешны → массив результатов	Один промис упал → вся операция с ошибкой
Promise.race	Первый завершившийся (успех или ошибка)	Ошибка, если первым завершился с ошибкой
Promise.allSettled	Всегда успешно, массив статусов	Ошибки не прерывают выполнение
Promise.any	Первый успешный промис	Ошибка, если все упали (AggregateError)

✅ Вывод (итоговое сравнение):

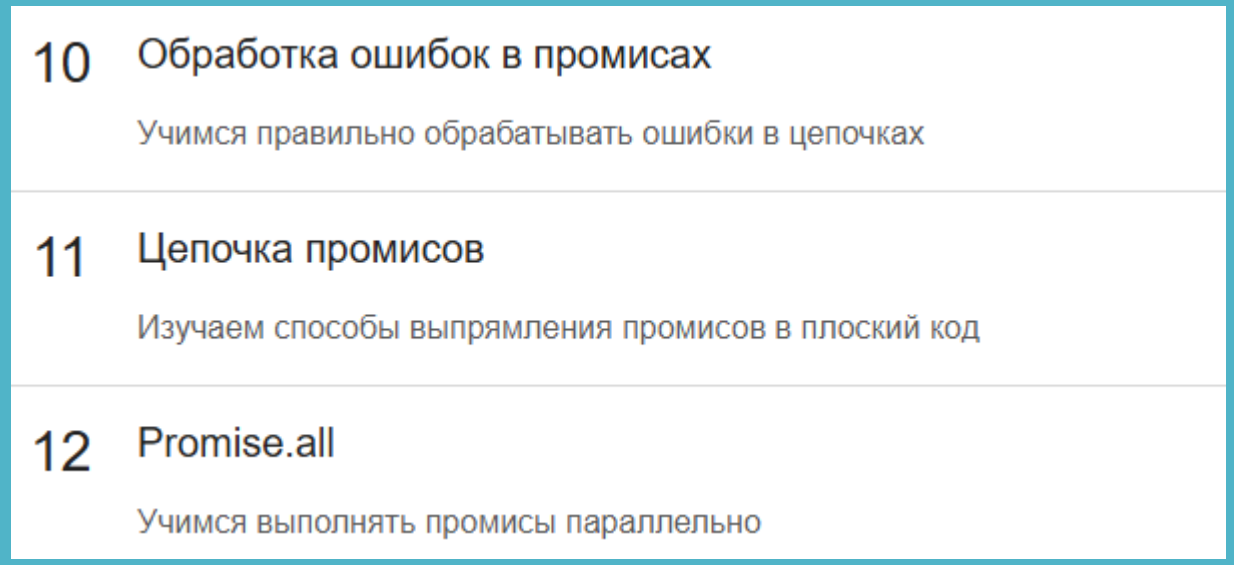
- **Promise.all** — ждём, пока выполнятся **все промисы успешно**.
  Если хотя бы один промис упадёт → весь результат будет ошибкой.
- **Promise.race** — ждём **первый завершившийся промис** (успех или ошибка).
- **Promise.allSettled** — ждём завершения **всех промисов**, получаем массив статусов (*успех/ошибка*), ошибки не прерывают выполнение.
- **Promise.any** — ждём **первый успешный результат**, ошибки игнорируем, пока не упадут все (тогда будет AggregateError).

Контрольные вопросы:

- Что произойдёт, если хотя бы один промис в Promise.all завершится с ошибкой?
- Чем отличается Promise.race от Promise.all?
- В чём особенность Promise.allSettled? Когда его стоит использовать?
- Чем отличается Promise.any от Promise.race?
- Что произойдёт, если все промисы в Promise.any завершатся с ошибкой?

Домашнее задание:

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



2. Повторить материал лекции.

Материалы лекций:

https://github.com/ShViktor72/Education

Обратная связь:

colledge20education23@gmail.com