ПМ3 Разработка модулей ПО.

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

Тема 2. Асинхронно программирование.

Лекция 10. Промисы и создание new Promise.

Цель занятия:

Изучить концепцию Promise в JavaScript, его состояние и использование для работы с асинхронными задачами.

Учебные вопросы:

1. Обзор промисов. Создание промиса с помощью конструктора.
2. Методы промиса.
3. Примеры использования промисов.

1. Обзор промисов. Создание промиса с помощью конструктора.

- **Определение**

Promise — это объект в JavaScript, который представляет собой завершение (или отклонение) асинхронной операции и её результат.

Он используется для работы с асинхронным кодом, позволяя избежать "адских коллбеков" (callback hell) и улучшая читаемость и структуру кода.

Зачем нужен Promise?

Управление асинхронными операциями:

Промисы позволяют работать с результатами асинхронных операций, таких как загрузка данных, выполнение запросов к API и т.д., в удобной и **предсказуемой** форме.

Улучшение читаемости кода:

Использование промисов позволяет избежать вложенных коллбеков, делая код более линейным и читаемым.

Обработка ошибок:

Промисы позволяют удобно обрабатывать ошибки с помощью метода catch, что упрощает процесс отладки и управление исключениями.

Цепочки промисов:

Благодаря методам then и catch можно создавать цепочки промисов, что способствует более четкому разделению логики и улучшает порядок выполнения, поскольку каждый этап может обрабатывать результат предыдущего.

Статусы выполнения:

Promise имеет три состояния:

- **Pending** (ожидание): начальное состояние, операция еще не завершена.
- **Fulfilled** (выполнен): операция завершена успешно, и результат доступен.
- **Rejected** (отклонен): операция завершилась неудачно, и ошибка доступна.

Создание нового Promise.

Чтобы создать новый Promise, используется конструктор **new Promise()**. Он принимает в качестве аргумента одну функцию, называемую executor (исполнитель).

```
const myPromise = new Promise(function executor(resolve, reject) {
  // Здесь асинхронный код
});
```

Эта функция-исполнитель выполняется немедленно после создания промиса. В неё автоматически передаются две функции-колбэка, которые нужно использовать для управления состоянием промиса:

- **resolve**(value): вызывает этот колбэк, когда асинхронная операция успешно завершилась. Это переводит промис в состояние fulfilled (выполнен). В value можно передать результат операции.
- **reject**(reason): вызывает этот колбэк, когда асинхронная операция завершилась с ошибкой. Это переводит промис в состояние rejected (отклонён). В reason можно передать объект ошибки.

Пример создания промиса.

Допустим, мы хотим создать промис, который имитирует загрузку данных с задержкой в 2 секунды.

```
const fetchData = new Promise((resolve, reject) => {
  setTimeout(() => {
    const data = { id: 1, name: 'Sample Data' };
    const success = true; // Имитируем успешное завершение

    if (success) {
      resolve(data); // Успех
    } else {
      reject(new Error('Failed to fetch data')); // Ошибка
    }
  }, 2000); // Задержка 2 секунды
});
```

Объяснение кода:

```
const fetchData = new Promise((resolve, reject) => {
```

Создание промиса:

Здесь мы создаем новый объект Promise и передаем в него функцию-исполнитель (executor). Эта функция принимает два параметра: resolve и reject.

Эти параметры — это функции автоматически объявляются и будут использоваться для изменения состояния промиса.

Асинхронная операция:

```
setTimeout(() => {
```

Внутри промиса мы используем setTimeout, чтобы имитировать асинхронную операцию, которая занимает 2 секунды.

Имитирование результата:

```
const data = { id: 1, name: 'Sample Data' };
const success = true; // Имитируем успешное завершение
```

Мы создаем объект data, который представляет собой данные, которые мы хотим вернуть при успешном завершении. Переменная success используется для имитации результата операции (успеха или ошибки).

здание

Вызов resolve и reject:

```
if (success) {
  resolve(data); // Успех
} else {
  reject(new Error('Failed to fetch data')); // Ошибка
}
```

Если success равно true, мы вызываем функцию resolve(data). Это означает, что операция завершилась успешно, и мы передаем данные, которые будут доступны в then методах, когда промис будет выполнен.

Если success равно false, мы вызываем reject(new Error('Failed to fetch data')). Это означает, что операция завершилась неудачно, и мы передаем объект Error, который будет доступен в catch методах, когда промис будет отклонен.

Задержка:

```
  }, 2000); // Задержка 2 секунды
});
```

setTimeout создает задержку в 2 секунды, после чего код внутри функции будет выполнен. Это имитирует время, необходимое для выполнения асинхронной операции.

Вызов функций.

Объявление:

- Функции resolve и reject объявлены автоматически при создании объекта Promise. Они не требуют явного объявления, так как они передаются как параметры функции-исполнителя.

Вызов:

- **resolve** вызывается в случае успешного завершения операции, передавая результат (data).
- **reject** вызывается в случае неудачи, передавая объект ошибки.

Заключение

Таким образом, этот код создает промис, который выполняет асинхронную операцию с имитацией задержки. В зависимости от значения переменной **success**, промис либо успешно завершится с данными, либо будет отклонен с ошибкой.

2. Методы промиса.

- **.then**(onFulfilled) → обрабатывает результат при успехе.
- **.catch**(onRejected) → обрабатывает ошибку.
- **.finally**(callback) → выполняется в любом случае (для «финальной» логики: очистка, лоадер и т. д.).

После создания промиса, мы можем использовать его методы **.then()** и **.catch()** для обработки результатов.

Метод **.then()** используется для обработки успешного завершения промиса. Он принимает два необязательных аргумента:

- Функция-колбэк для успешного выполнения (onFulfilled).
- Функция-колбэк для отклонения (onRejected).

Например:

onFulfilled onRejected

```
fetchData
  .then(data => {console.log('Данные получены:', data);}, () => {console.log("error!")})
```

Чаще всего .then() используют для первого аргумента. Внутри него доступен результат, переданный в resolve().

```
fetchData.then(result => {
  console.log('Данные успешно получены:', result);
});
```

```
Данные успешно получены: { id: 1, name: 'Sample Data' }
```

Метод .catch() — это более удобный способ обработки ошибок, чем второй аргумент в .then(). Он используется для перехвата ошибок (когда был вызван reject()).

Если success = false:

```
fetchData.catch(error => {
  console.error('Произошла ошибка:', error.message);
});
```

```
Произошла ошибка: Failed to fetch data
```

Важно, что .then() всегда возвращает новый промис, что позволяет выстраивать цепочку операций.

```
fetchData
  .then(result => {
    // Этот код выполняется, если промис успешен
    console.log('Данные успешно получены:', result);
    result.name = "New data";
    return result; // Возвращает новый промис с этим значением
  })
  .then(newValue => {
    // Этот код получает 'Новое значение' из предыдущего then
    console.log('Продолжение цепочки:', newValue);
  });
```

Метод .catch() встраивается в цепочку промисов после одного или нескольких вызовов .then(), чаще всего в конец цепочки, чтобы обрабатывать ошибки.

```
fetchData
  .then(result => {
    // Этот код выполняется, если промис успешен
    console.log('Данные успешно получены:', result);
    result.name = "New data";
    return result; // Возвращает новый промис с этим значением
  })
  .then(newValue => {
    // Этот код получает 'Новое значение' из предыдущего then
    console.log('Продолжение цепочки:', newValue);
  })
  .catch(error => {console.log('Error: ', error)});
```

Метод .finally() выполняется всегда, независимо от того, был промис успешно завершен или отклонен.

Это идеальное место для выполнения очистки ресурсов, например, скрытия индикатора загрузки или закрытия соединения.

- .finally() не получает никаких аргументов (ни результата, ни ошибки).
- Он не изменяет результат или ошибку промиса.
- Он просто позволяет выполнить код после завершения промиса.

```
fetchData
  .then(result => {
    // Этот код выполняется, если промис успешен
    console.log('Данные успешно получены:', result);
    result.name = "New data";
    return result; // Возвращает новый промис с этим значением
  })
  .then(newValue => {
    // Этот код получает 'Новое значение' из предыдущего then
    console.log('Продолжение цепочки:', newValue);
  })
  .catch(error => {console.log('Error: ', error)})
  .finally(() => {
    // Этот код выполнится всегда
    // Закрываем соединение с БД
    console.log('Операция завершена.');
  })
```

Ошибок нет:

```
Данные успешно получены: { id: 1, name: 'Sample Data' }
Продолжение цепочки: { id: 1, name: 'New data' }
Операция завершена.
```

Ошибка:

```
Error:  Error: Failed to fetch data
    at Timeout._onTimeout (c:\Users\user\Documents\JScode\test1\test.js:9:14)
    at listOnTimeout (node:internal/timers:588:17)
    at process.processTimers (node:internal/timers:523:7)
Операция завершена.
```

3. Примеры использования промисов.

Пример:

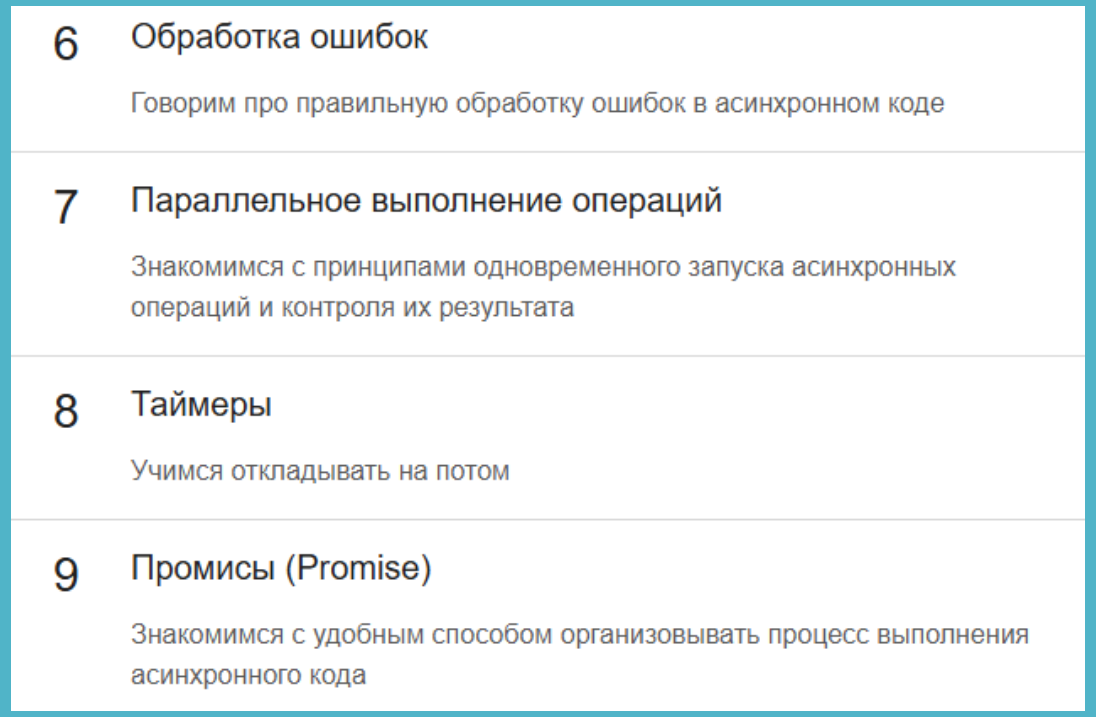
```
const promise = new Promise((resolve, reject) => {
  setTimeout(() => {
    const success = Math.random() > 0.3; // 70% успеха
    if (success) { resolve("✅ Успех!");}
    else { reject("❌ Ошибка!"); } }, 1000);
});

promise
  .then((result) => { console.log("Результат:", result); })
  .catch((error) => { console.error("Ошибка:", error); })
  .finally(() => { console.log("Завершено!"); });
```

Контрольные вопросы:

Домашнее задание:

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



2. Повторить материал лекции.

Материалы лекций:

https://github.com/ShViktor72/Education2025

Обратная связь:

colledge20education23@gmail.com