Summary of Asynchronous JavaScript

🕰️ What is Synchronous?

- Synchronous Programming means our code runs line by line. Each line must be completed before we move on the next line.
- For Example: We are preparing Cake. Now for preparing Cake is a step by step process. We have to follow those steps line by line. We can't jump directly from Step 1 to Step 4. We have to do in order like Step 1 then Step 2, 3 and then Step 4.
- That's why Synchronous is also called as blocking way.

⚡ What is Asynchronous?

- In Asynchronous Programming also our code runs line by line. But for moving forward we don't need to wait to finish that task. We can start a task and if it takes more time, then we can move to the next line.
- We can start a task, and while we wait for it to complete, we can do something else.
- For example: You are preparing a cake and also you are also watching this course. Here when you put the cake in the oven for baking, it will take time. So at that time, you can do other things instead of just waiting for cake to bake.
- That's why Asynchronous is also called as non-blocking way.
- We will use Asynchronous JavaScript when we want to perform non-blocking operations like:
    - Make Database Queries
    - API Calls
    - Reading/writing files etc...

🔁 Methods of Making Synchronous Code work like Asynchronous Code:

- Callback Functions
- Promises

📞 Callback Functions:

- Callback function is the function which we pass as argument in another function and executed after a specific task is completed.
- For Example: In setTimeout function we pass function as argument. That is called as callback function.

```
setTimeout(() => {
   console.log("This runs after 3 seconds!!")
}, 3000);
```

- This is very important concept for handling asynchronous work.

```
function fetchStudent(id, callback) {
  // fetch and update student
  setTimeout(() => {
    console.log("Fetching student data from the database...");

    callback({ id: id, name: "Harley", e_num: 500 });
  }, 3000);
}

console.log("Start");

fetchStudent(1, (student) => {
  console.log("Student", student);
});

console.log("End");
```

🔥 Callback Hell:

- Now when we have more callback functions in another callback then we face the issue of callback hell which is the nested structure of callbacks and it makes our code hard to manage and understand.
- So for solving this callback hell problem, we turn our anonymous functions into named functions. Basically we separately define this functions. Refer this code ⬇️

```
function fetchStudent(id, callback) {
  // fetch and update student
  setTimeout(() ⇒ {
    console.log("Fetching student data from the database...");

    callback({ id: id, name: "Harley", e_num: 500 });
  }, 3000);
}

function getResult(enrollment, callback) {
  setTimeout(() ⇒ {
    console.log("Fetching result from the database...");

    callback({ resultId: enrollment, percentage: 70 });
  }, 3000);
}

console.log("Start");

const printStudent = (student) ⇒ {
  console.log("Student", student);
  getResult(student.e_num, printResult);
};

const printResult = (result) ⇒ {
  console.log("Result", result);
};

fetchStudent(1, printStudent);
```

```
console.log("End");
```

🤝 Promises:

- A promise is an object which is able to hold the result of an asynchronous operation. In other words, Promise is promise you - to give you the result of the asynchronous operation or give you an error.

```
// Create promise
const pr = new Promise((resolve, reject) ⇒ {
  setTimeout(() ⇒ {
    // Getting data from the database
    const student = { id: 1, name: "Harley" };
    const status = false;

    if (status) {
      resolve(student);
    } else {
      reject(new Error("This is error message"));
    }
  }, 3000);
});

// Consume promise
pr.then((result) ⇒ console.log(result)).catch((error) ⇒ console.log(error));
```

- Now when we create a `new promise`, it is in a pending state (performing the asynchronous task). After that If that asynchronous task completed successfully then promise in fulfilled state and if there is any error occurs, then the promise is in the rejected state.
- So we can get resolve value in then method and reject error in catch method. For Example:

```
console.log("Start");
```

```
// fetchStudent(1, (student) => {
//   getResult(student.enrollment, (result) => {
//     console.log("Result", result);
//   });
// });

fetchStudent(1)
  .then((student) => getResult(student.enrollment))
  .then((result) => console.log("Result", result))
  .catch((error) => console.log(error));

console.log("End!");

function fetchStudent(id) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      console.log("Fetching Student Data from Database...");
      resolve({ id: id, name: "Harley", enrollment: 500 });
    }, 3000);
  });
}

function getResult(enrollment) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      console.log("Fetching Result from Database...");
      // reject(new Error("Result not found!"));
      resolve({ resultId: enrollment, percentage: 70 });
    }, 3000);
  });
}
```

⏳ Async-await:

- With promises we have to write then method again and again if we have multi. We can get rid of that by using Async/await. (Almost 80% of developers use Async/await)

- Async/await helps us to write asynchronous code that looks like synchronous code. When we call the function (which is returning a promise), we can add await operator before that function.
- But to use `await` we have to move our code in a function and add an `async` operator before that function. That’s it. We don’t need anything else.

```
async function printResult() {
  try {
    const student = await fetchStudent(1);
    const result = await getResult(student.enrollment);
    console.log("result", result);
  } catch (error) {
    console.log("error", error);
  }
}

printResult();
```

The ultimate Node JS Course ~ Code Bless You❤️