What is callback hell? Explain different ways to solve callback hell with examples each.

→ Callback hell, also known as the Pyramid of Doom, is a common issue that arises in asynchronous programming when dealing with multiple nested callbacks. This occurs when you have a sequence of asynchronous operations that depend on the results of each other, and the code becomes deeply nested and hard to read. It makes the code difficult to maintain, understand, and debug, leading to poor code quality.

callback hell with a JavaScript example:

```
asyncFunc1(function(result1) {
  asyncFunc2(result1, function(result2) {
    asyncFunc3(result2, function(result3) {
      asyncFunc4(result3, function(result4) {
        // More nested callbacks...
      });
    });
  });
});
```

In this example, ‘asyncFunc1()’, ‘ asyncFunc2()’, ‘asyncFunc3()’, and ‘asyncFunc4()’ are asynchronous functions that take callbacks as arguments, and each function depends on the result of the previous one. To solve callback hell, there are several techniques and patterns available. Let's explore some of them:

- Using Async/Await with Promise.all:Using Async/Await with Promise.all:

→ If the operations don't depend on each other, you can run them concurrently using `Promise.all()` :

```
async function exampleWithPromiseAll() {
        try {
          const [result1, result2, result3, result4] = await
      Promise.all([
            asyncFunc1(),
            asyncFunc2(),
            asyncFunc3(),
            asyncFunc4(),
          ]);
          // The results here
        } catch (error) {
          // Handle errors
        }}
```

- Using Async Functions and Promisify:

→ Convert callback-based functions to return Promises using a utility like util.promisify(), if available:

```
const { promisify } = require('util');
const asyncFunc2Promise = promisify(asyncFunc2);
const asyncFunc3Promise = promisify(asyncFunc3);
const asyncFunc4Promise = promisify(asyncFunc4);

async function exampleWithPromisify() {
  try {
    const result1 = await asyncFunc1();
    const result2 = await asyncFunc2Promise(result1);
    const result3 = await asyncFunc3Promise(result2);
    const result4 = await asyncFunc4Promise(result3)
  } catch (error) {
    // Handle errors
  }
}
```

- Using Promises:

→ Promises provide a cleaner way to handle asynchronous operations and avoid excessive nesting. Modern JavaScript allows the use of ‘async/await’ to make asynchronous code look more like synchronous code:

```
async function exampleWithPromises() {
  try {
    const result1 = await asyncFunc1();
    const result2 = await asyncFunc2(result1);
    const result3 = await asyncFunc3(result2);
    const result4 = await asyncFunc4(result3);
    // Use the final result4 here
  } catch (error) {
    // Handle errors
  }
}
```