Promises

(Need: Callback Hell)

```
function step1(callback) {
  setTimeout(() => {
    console.log('Step 1');
    callback();
  }, 1000);
}

function step2(callback) {
  setTimeout(() => {
    console.log('Step 2');
    callback();
  }, 1000);
}
```

```
function step3(callback) {
  setTimeout(() => {
    console.log('Step 3');
    callback();
  }, 1000);
}

step1(() => {
  step2(() => {
    step3(() => {
      console.log('All steps completed');
    });
  });
});
```

When multiple asynchronous operations need to be performed in sequence, callbacks can lead to deeply nested and hard-to-read code, often referred to as “callback hell.”

Promises

(States of Promise)

1. Definition: A promise is an object representing the eventual completion or failure of an asynchronous operation.
2. States of a Promise:
   - Pending: Initial state, neither fulfilled nor rejected.
   - Fulfilled: Operation completed successfully.
   - Rejected: Operation failed.

Promises

(Creation of Promise)

```
// Creating a Promise
let promise = new Promise((resolve, reject) => {
  // Asynchronous operation
  if (result()) {
    resolve('Success');
  } else {
    reject('Error');
  }
});
```

Promises are created using the Promise constructor, which takes an executor function with two arguments: resolve and reject.

Promises

(Handling of Promise)

```
// Handling a Promise: handle value
promise.then(value => {
  console.log(value); // 'Success'
});

// Handling a Promise: handle rejection
promise.catch(error => {
  console.error(error); // 'Error'
});

/* Handling a promise: Executes a block of
code regardless of the promise's outcome.*/
promise.finally(() => {
  console.log('Operation completed');
});
```

Promises have then, catch, and finally methods for handling the results of the asynchronous operation.

- then(): Used to handle fulfilment.
- catch(): Used to handle rejection.
- finally(): Executes a block of code regardless of the promise's outcome.

Promises

(Solving Callback Hell)

```
function step1() {
  return new Promise((resolve) => {
    setTimeout(() => {
      console.log('Step 1');
      resolve();
    }, 1000);
  });
}

function step2() {
  return new Promise((resolve) => {
    setTimeout(() => {
      console.log('Step 2');
      resolve();
    }, 1000);
  });
}
```

```
function step3() {
  return new Promise((resolve) => {
    setTimeout(() => {
      console.log('Step 3');
      resolve();
    }, 1000);
  });
}

step1()
  .then(() => step2())
  .then(() => step3())
  .then(() => {
    console.log('All steps completed');
  });
```

In this version, each step returns a Promise that resolves after a timeout. The steps are chained together using .then(), making the code more readable and easier to maintain.

Promises

(Error handling)

```
// Using Promises
fetch('https://api.example.com/data')
  .then((response) => response.json())
  .then((data) => console.log(data))
  .catch((error) => console.error('Fetch error:', error));

fetchData();
```

Errors in promises are handled using .catch() or by chaining .then() with a second callback for error handling.

Fetch API

```
fetch('https://jsonplaceholder.typicode.com/posts')
  .then(response => {
    if (!response.ok) {
      throw new Error('Network response was not ok ' + response.statusText);
    }
    return response.json();
  })
  .then(data => console.log(data))
  .catch(error => console.log(error));
```

- The Fetch API provides a modern way to make HTTP requests in JavaScript.
- It is a promise-based API, making it easier to handle asynchronous requests.