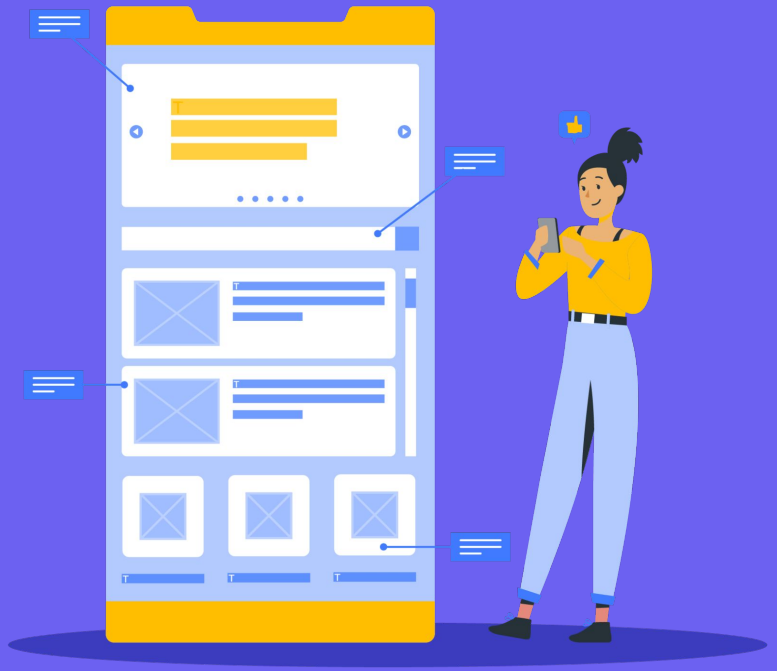
Promises, Promise Hell, Promise.all, Async/ Await, Iterator, Generator, Inversion of Control



Relevel
by Unacademy

Promises

An object that represents the completion (or failure) of an asynchronous operation and also its resulting value is known as a promise.

```
const newPromise = new Promise((resolve, reject) => {
   console.log("Promise created");
   setTimeout(() => {
       reject("Some Error");
   }, 1000); //reject after 1 sec

   console.log("Exit from promise")
});

console.log("First txt in sync");

newPromise
   .then((result) => {
       console.log("success:", result);
   })
   .catch((error) => {
       console.log("Faliure:", error);
   })
   .finally(() => {
       console.log("Promise completed");
   });

console.log("Last txt sync");
```

Promise Hell

Promise hell is a self created problem due to lack of understanding of Promises, unlike callback hell. In Promise hell as well we have to wait for other promises to return and it will fill the stack.

```
fetchStory()
    .then((story) => {
        return findText(story);
    })
    .then((txt) => {
        return print(txt);
    });

// single liners
fetchStory()
    .then((story) => findText(story))
    .then((txt) => print(txt));

// inline
fetchStory()
    .then(findText)
    .then(print);
```

Promise.all

It is a method that takes an array of promises and return new promise only when all promises will be fulfilled.
It works asynchronous. So, if there is independent promises. We can add it into Promise.all() and it will try to resolve it parally.

```
connectDB()
   .then((database) => {
       return findMenu(database)
           .then((menu) => {
               return getUser(database)
                   .then((user) => {
                       return suggestItems(menu, user);
                   });
           });
   });
```

```
const databasePromise = connectDB();
const menuPromise = menuPromise
   .then(findMenu);
const userPromise = databasePromise
   .then(getUser);
Promise.all([
   menuPromise,
   userPromise
])
   .then(([menu, user]) => suggestItems(menu, user));
```

Async/Await

Async/ Await is used to handle promises in synchronous code fashion with less code effort.

Async: We put this keyword before any function then, it will return a promise.

Await: This keyword is used before a promise inside the **Async block** to block the code until promise resolves or reject.

So, these keyword helps us to get write a synchronous fashion code with cleaner syntax.

```
const first = () => new Promise(resolve => {
   setTimeout(() => resolve('first()'), 1000);
});
const second = () => new Promise(resolve => {
   setTimeout(() => resolve('second()'), 1000);
});
const third = () => new Promise(resolve => {
   setTimeout(() => resolve('third()'), 1000);
});

// First Apporach - Asynchronous
Promise.all([first(), second(), third(), { someKey: someValue }])
   .then((data) => {
       console.log("success:", data);
   })
   .catch((err) => {
       console.log("error:", err);
   })

// Second Approach - Synchronous
const process = async () => {
   const first = await a();
   const second = await b();
   const third = await c();

   return [first, second, third];
};

process
   .then((data) => {
       console.log("success:", data);
   })
   .catch((err) => {
       console.log("error:", err);
   })
```

Iterators and Generators

Iterators:
These are the object with a sequence and able to return values when terminates.
These follows Iterators protocol by having a next() method that returns two properties:

1. Value: The next value in the sequence.
2. Done: Indicates whether that was the last value in the sequence or not by returning true and false. True when ends, false in the middle of iteration.

Generators:
So, to create a iterator we need careful programing. Generator makes it very easier. It helps us to define the algorithm of iteration in a single function where execution is not continuous,

It is a special type of iterator, and written with syntax function*.

```
function rangeIterator(begin = 0, end = Infinity, step = 1) {
    let nextIndex = begin;
    let iterationCount = 0;

    const iterator = {
        next: function() {
            let result;
            if (nextIndex < end) {
                result = { value: nextIndex, done: false }
                nextIndex += step;
                iterationCount++;
                return result;
            }
            return { value: iterationCount, done: true }
        }
    };
    return iterator;
}

const it = rangeIterator(1, 20, 3);

let response = it.next();
while (!response.done) {
console.log(response.value); // 1 4 7 10 13 16 19
response = it.next();
}

console.log("Iterated over sequence of size: ", response.value); // Iterated over
sequence of size:  7
```

```
function* rangeIterator(begin = 0, end = 100, step = 1) {
    let iterationCount = 0;
    for (let i = begin; i < end; i += step) {
        iterationCount++;
        yield i;
    }
    return iterationCount;
}
```

Inversion of Control

An abstract programming principle based on the flow of control that should be managed completely by some specific implementation of the framework, which is external to your code.

```
function filter(array, filterFunction) {
  let newArray = []
  for (let index = 0; index < array.length; index++) {
    const element = array[index]
    if (filterFunction(element)) {
      newArray[newArray.length] = element
    }
  }
  return newArray
}
 const array = [0, 1, undefined, 2, null, 3, 'four', ''];
 console.log(filter(
  array,
  el => el !== null && el !== undefined,
))
// [0, 1, 2, 3, 'four', '']
 console.log(filter(array, el => el !== undefined))
// [0, 1, 2, null, 3, 'four', '']
 console.log(filter(array, el => el !== null))
// [0, 1, 2, undefined, 3, 'four', '']
 console.log(filter(array,
  el => el !== undefined && el !== null && el !== 0,
))
// [1, 2, 3, 'four', '']
 console.log(filter(array,
  el => el !== undefined && el !== null && el !== '',
))
// [0, 1, 2, 3, 'four']
```

MCQs

1. Output of the code:

```
var p = new Promise((resolve, reject) => {
   reject(Error('The Fails!'))
 })
 p.catch(error => console.log(error.message))
 p.catch(error => console.log(error.message))
```

A. Print once
B. Print twice
C. Exit with error
D. No warning

MCQs

2. Output of the code:

```
Promise.resolve('A')
 .then(() => {
   throw Error('OMG')
 })
 .catch(error => {
   return 'Works'
 })
 .then(data => {
   throw Error('Not worked')
 })
 .catch(error => console.log(error.message))
```

A. Print “OMG” and “Not worked”
B. Print “Works”
C. Print “OMG”
D. Print “Not Worked”

MCQs

3. Which of the following method is used to get value from generator-iterator objects in JavaScript?
   A. next
   B. yield
   C. done
   D. stop

4. Promise.all is rejected if any of the elements are rejected.
   A. True
   B. False

5. What will be state and value of promise p in the code.

```
var p = new Promise(function(resolve, reject) {

    throw "Sorry";

}).

then((data) => console.log(data), (data) => data);
```

   A. State: *fulfilled*, value: "Sorry"
   B. State: *rejected*, value: "Sorry"
   C. State: *fulfilled*, value: undefined
   D. State: *rejected*, value: undefined

Practice/HW

1. Implement a map using the Inversion of Control principle that will perform operations on an array such as:
   a. Squaring the elements
   b. Dividing them by 5.
   c. The root of the element
   d. etc
2. Implement a generator that will return power all the powers of 3.
3. Program to explain difference between Promise.all() and Promise.race().

Thank you