Namasthe JavaScript - Season 2

This Notes is taken from Namasthe JS Season 2 course from Youtube - Akshay Saini

Callback Hell

1. **The Good Part of Callbacks**

- **Asynchronous Code:** Callbacks are crucial for writing asynchronous code in JavaScript.
- **Single-Threaded Nature of JavaScript:** JavaScript is single-threaded and executes one thing at a time.
- **Waiting with Callbacks:** To execute code after a delay, like 5 seconds, a callback function is used with `setTimeout`.
- **E-commerce Example:** In an e-commerce scenario, callbacks manage the flow between creating an order and proceeding to payment, which are asynchronous operations.

2. **The Bad Part of Callbacks**
    - **Callback Hell**

    - **Definition:** Callback Hell occurs when multiple nested callbacks make the code unmanageable and grow horizontally, also known as the Pyramid of Doom.
- **Inversion of Control**
    - **Definition:** Inversion of control means losing control over your code when using callbacks.
    - **Explanation:** By passing a callback to an API (e.g., create order), the developer trusts that the API will execute the callback appropriately. There's a risk the callback might not be called, or might be called multiple times. There is a risk of blindly trusting external code to execute the callback as expected.

3. Example

```
// Callback Hell → Pyramid Of Doom Structure
const cart = ["shoes", "pants", "kurta"];

api.createOrder(cart, function () {
   api.proceedToPayment(function () {
      api.showOrderSummary(
      function () {
         api.updateWallet()
      })
   })
});
```

Promises

- **Introduction to Promises:** Promises are crucial for handling asynchronous operations in JavaScript
- **Promises as a Solution for issues with Callback:**
    - **Promise Return:** Instead of taking a callback, an API can return a promise.

    - **Promise Object:** A promise is initially an empty object that will eventually be filled with the result of the asynchronous operation.
    - **.then():** A callback function is attached to the promise object using `.then()`. This callback executes automatically when the promise is resolved (i.e., when the data is available).
    - **Control:** Promises provide trust and guarantee that the callback function will be called only once when the data is available.
- **Deep Dive into Promise Object:**
    - **Fetch API:** The `fetch` function, which returns a promise, demonstrates the structure of a promise object.
    - **Promise Structure:** A promise object has:
        - `PromiseState`: Indicates the state of the promise (`pending`, `fulfilled`, or `rejected`).
        - `PromiseResult`: Stores the data when the promise is fulfilled.
    - **Pending State:** Initially, the promise is in a `pending` state.
    - **Fulfilled State:** After the asynchronous operation completes, the promise transitions to the `fulfilled` state, and the `PromiseResult` is populated.
    - **Immutability:** Promise objects are immutable once resolved.
- **Interview Perspective: What is a Promise?**
    - **Placeholder:** A promise can be seen as a placeholder that will be filled later with a value.
    - **Container:** It can also be described as a container for a future value.
    - **MDN Definition:** 👉 ***A promise is an object representing the eventual completion of an asynchronous operation.***"

- **Callback Hell and Promise Chaining:**
  - **Callback Hell:** Nested callbacks make code hard to read and maintain.
  - **Promise Chaining:** Promises solve this with "promise chaining," using multiple `.then()` calls to handle sequential asynchronous operations in a cleaner, vertical fashion.
  - **`.then()` Chaining:** Each `.then()` returns a promise, allowing for chaining.
  - **Return:** It's important to return a value or a promise from each `.then()` to pass data down the chain.
- **Summary:**
  - Promises address the inversion of control issue with callbacks.
  - Promises provide guarantees of single execution and clear states.
  - Promise objects have `PromiseState` and `PromiseResult`.
  - Promise chaining resolves callback hell.
- **Example:**

```
const cart = ["shoes", "pants", "kurta"];

createOrder(cart, function (orderId) {
   proceedToPayment(orderId);
});

const promise = createOrder(cart);

promise.then(function (orderId) {
   proceedToPayment(orderId);
});
```

```
const cart = ["shoes", "pants", "kurta"];

createOrder(cart, function (orderId) {
  proceedToPayment(orderId, function (paymentInfp) {
    showOrderSummary(paymentInfp, function () {
      updateWalletBalance();
    });
  });
});

createOrder(cart)
  .then(function (orderId) {
    return proceedToPayment(orderId);
  })
  .then(function (paymentInfo) {
    return showOrderSummary(paymentInfo);
  })
  .then(function (paymentInfo) {
    return updateWalletBalance(paymentInfo);
  });
```

```
createOrder(cart)
  .then((orderId) => proceedToPayment(orderId))
  .then((paymentInfo) => showOrderSummary(paymentInfo))
  .then((paymentInfo) => updateWalletBalance(paymentInfo));
```

Creating a Promise, Chaining & Error Handling

- **Creating Promises:** Imagine you're ordering something online. The `createOrder` function is like placing your order. It gives you a promise, a sort of IOU. The `Promise` thing has two sides: `resolve` if everything goes well (you get an order ID), and `reject` if there's a problem (like your cart is empty).
- **Consuming Promises:** Once you have that promise, you use `.then()` to say what should happen when your order is confirmed. Think of it as setting up a delivery notification. The video also shows how to use `setTimeout` to

pretend there's a delay, like waiting for the server to respond.

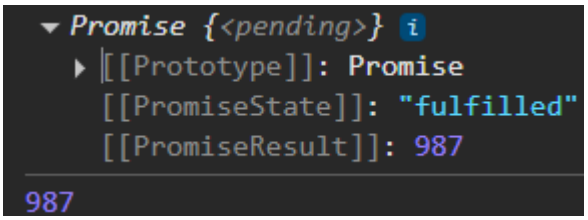
- **Error Handling:** What if something goes wrong? That's where `.catch()` comes in. It's like having a customer service line for when your order has issues. It's super important to handle errors gracefully, so your website doesn't break.
- **Promise Chaining:** Now, let's say after placing your order, you need to pay. The video introduces chaining with another function, `proceedToPayment`, which also gives you a promise. You use `.then()` again to chain these actions, one after the other. It's like a series of steps that happen in order. Make sure to return something in each `.then()` so the next step knows what's going on.
- **Advanced Error Handling:** You can use `.catch()` at different steps in the chain. If something breaks in the middle, the closest `.catch()` will handle it. But, if you have a `.then()` after a `.catch()`, that `.then()` will always run, no matter what happened before.

NOTE: We attach callbacks to promise not pass them as arguments to promise

Example: When promise is fulfilled or resolved or success

```
const cart = ["shirt", "pant", "kurta"];

const promiseCreateOrder = createOrder(cart);

console.log(promiseCreateOrder);

// Consumer Part
promiseCreateOrder.then(function(orderId) {
   console.log(orderId);
   // proceedToPayment(orderId);
})

// Producer Part
```

```
function createOrder(cart) {
    const promise = new Promise(function(resolve, reject){
        const orderId = 987;
        if(!validateCart(cart)) {
            let err = new Error("Invalid Cart");
            reject(err);
        }
        else {
            setTimeout(function () {
                resolve(orderId);
            }, 5000);
        }
    });
    return promise;
}

function validateCart(cart) {
    return true;
}
```



Example: When promise is failed

```
const cart = ["shirt", "pant", "kurta"];

const promiseCreateOrder = createOrder(cart);

console.log(promiseCreateOrder);

// Consumer Part
promiseCreateOrder.then(function(orderId) {
    console.log(orderId);
```

```
    // proceedToPayment(orderId);
}).catch(function (error) {
    console.log(error.message);
})

// Producer Part
function createOrder(cart) {
    const promise = new Promise(function(resolve, reject){
        const orderId = 987;
        if(!validateCart(cart)) {
            let err = new Error("Invalid Cart");
            reject(err);
        }
        else {
            setTimeout(function () {
                resolve(orderId);
            }, 5000);
        }
    });
    return promise;
}

function validateCart(cart) {
    return false;
}
```

`Example: Different scenarios (Promise chaining)`

```
const cart = ["shirt", "pant", "kurta"];

// commented in scenario - 3
const promiseCreateOrder = createOrder(cart);
```

```
// Consumer Part
// Scenario - 1 with pass
// All the errors in the chain will go to one catch block.
promiseCreateOrder.then(function(orderId) {
    console.log(orderId);
    return orderId;
}).then(function(orderId) {
    return proceedToPayment(orderId);
}).then(function(paymentStatus) {
    console.log(paymentStatus);
}).catch(function (error) {
    console.log(error.message);
})

// Direct Also possible
// Scenario - 2 with fail
promiseCreateOrder.then(function(orderId) {
    return proceedToPayment(orderId);
}).then(function(paymentStatus) {
    console.log(paymentStatus);
}).catch(function (error) {
    console.log(error.message);
})

// Even if cart is invalid and want to make payment successful
// Scenario - 3 with fail
createOrder(cart).then(function(orderId) {
    console.log(orderId);
    return orderId;
}).catch(function (error) {
    console.log(error.message);
}).then(function(orderId) {
    return proceedToPayment(orderId);
}).then(function(paymentStatus) {
    console.log(paymentStatus);
})
```

```
// Producer Part
function createOrder(cart) {
    const promise = new Promise(function(resolve, reject){
        const orderId = 987;
        if(!validateCart(cart)) {
            let err = new Error("Invalid Cart");
            reject(err);
        }
        else {
            setTimeout(function () {
                resolve(orderId);
            }, 5000);
        }
    });
    return promise;
}

function validateCart(cart) {
    // scenario 1 and 2 true
    // scenario 3 false
    return true;
}

function proceedToPayment(orderId) {
    return new Promise(function(resolve, reject) {
        resolve("Payment successful");
    });
}
```

Scenario - 1

```
987
Payment successful
```

Scenario - 2

```
Payment successful
```

Scenario - 3

```
Invalid Cart
Payment successful
```

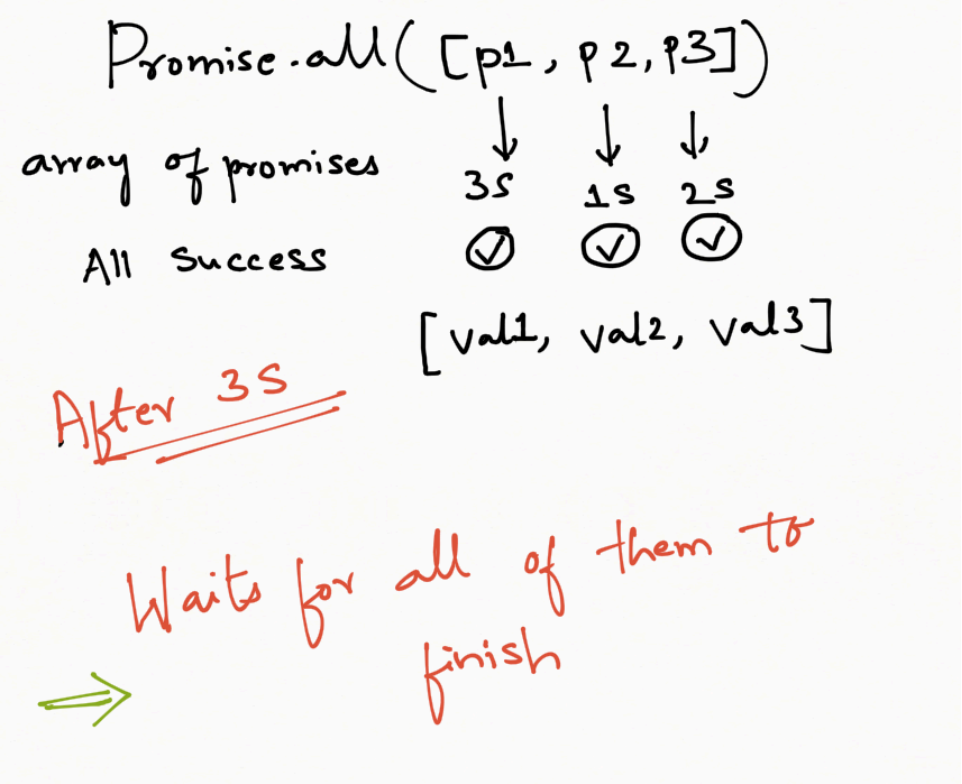
Promise APIs + Interview Questions (all, allSettled, race, any)

1. **Key Words**

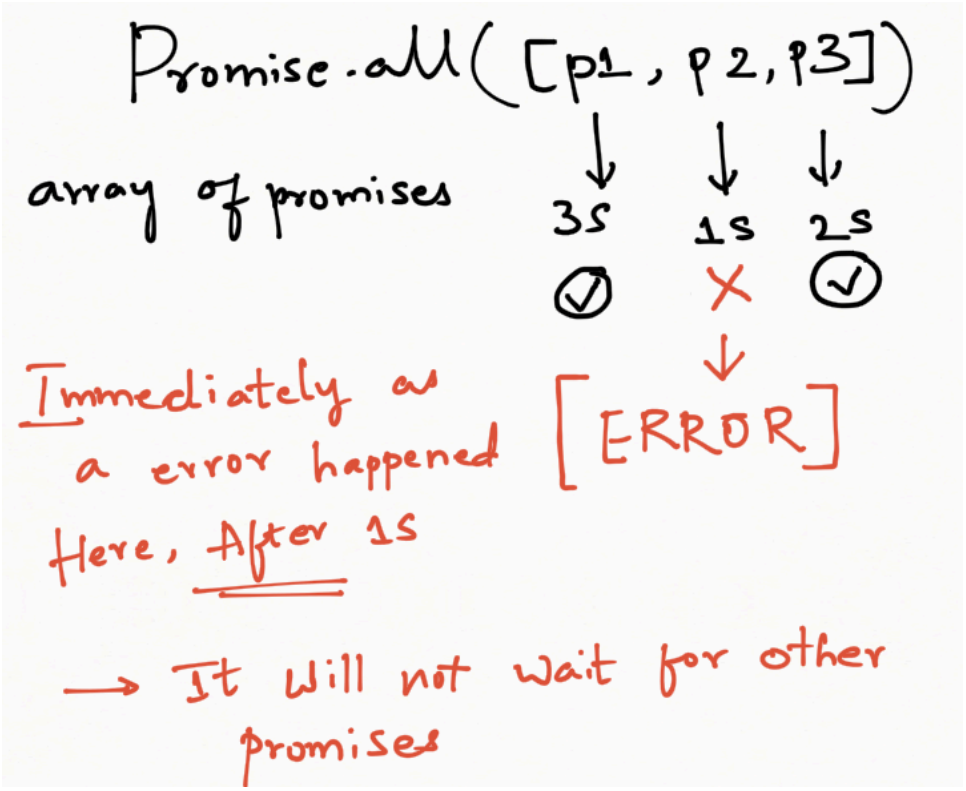
Settled - Got the result

- Success/Failure
- Resolve/Reject
- Fulfilled/Rejected

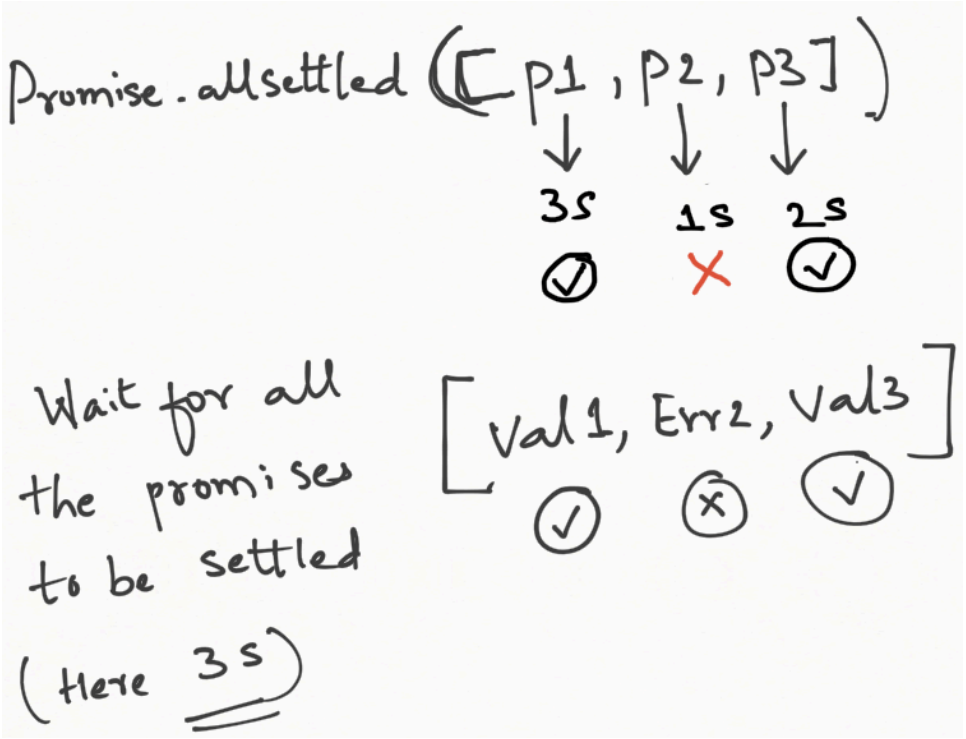
2. Promise.all() (If all are fulfilled)



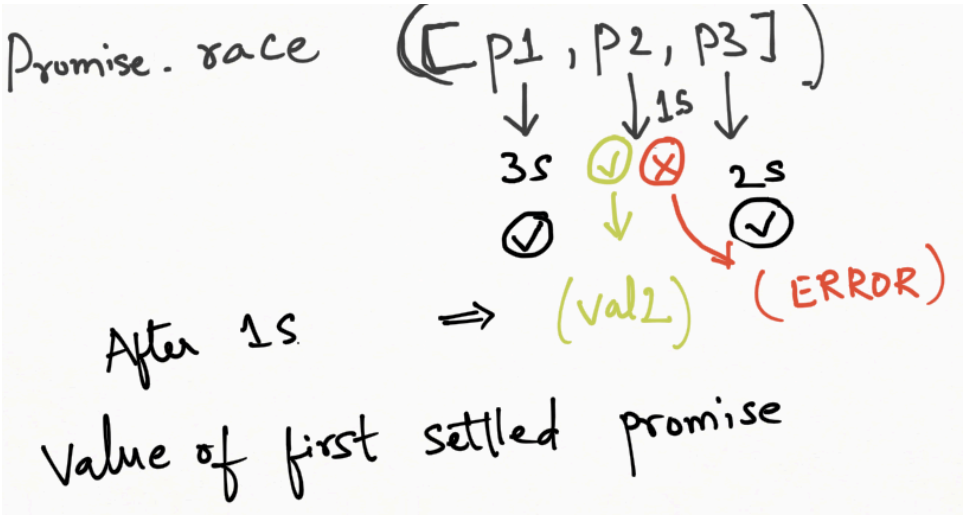
3. Promise.all() → Fail Fast (If any one fails)



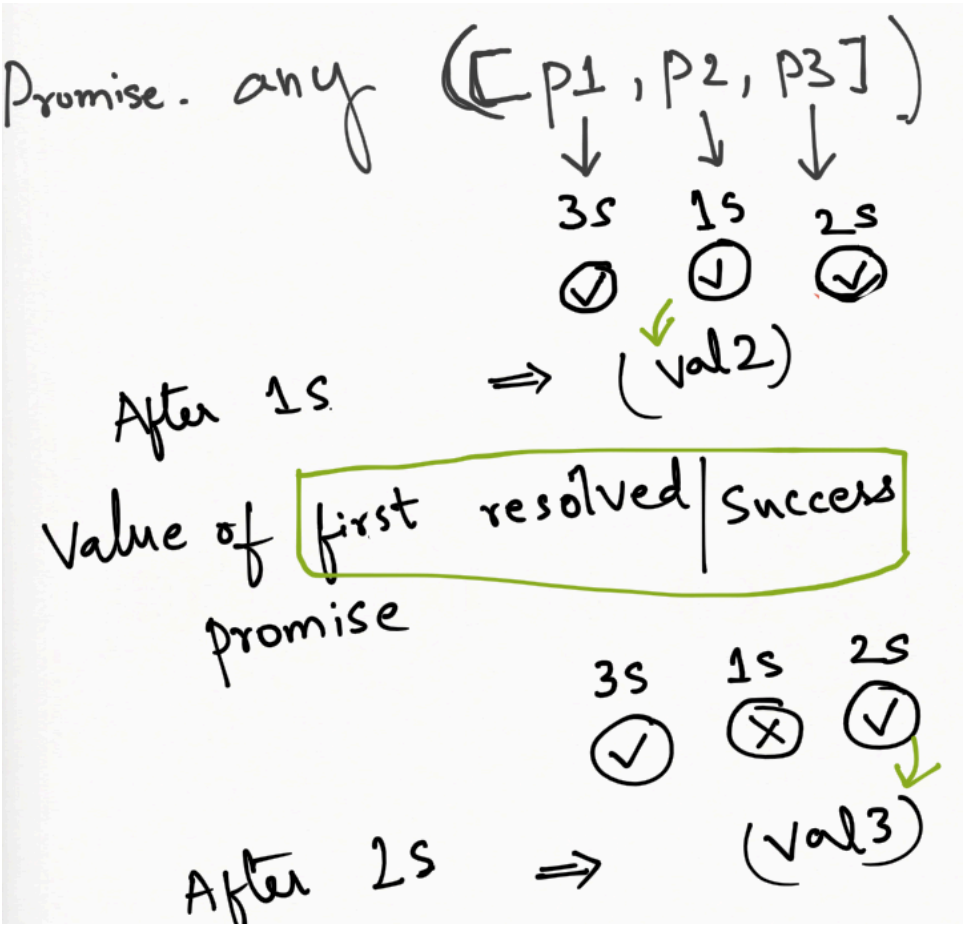
4. Promise.allSettled() will behave in a same way as Promise.all(), if all the promises are success. Below is the example if any one fails.



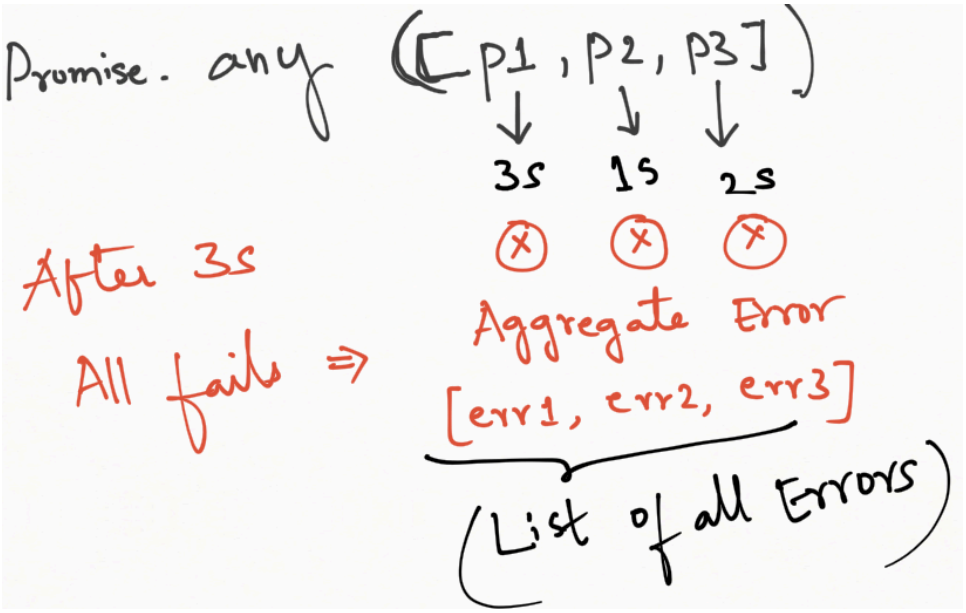
5. Promise.race() → Value of first settled promise irrespective of Fulfilled/Rejected.



6. Promise.any() → Value of first resolved promise. Seeking for first success.



7. Promise.any() → Incase of all failures - **Aggregate error** - with array of all errors.

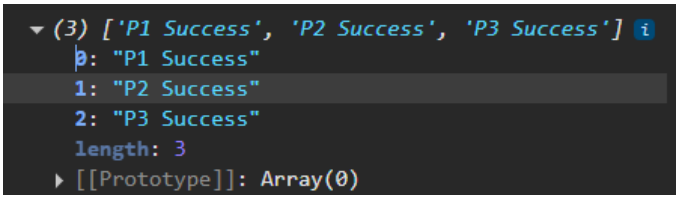


Examples:

1. Promise.all() → All Success

```
const p1 = new Promise((resolve, reject) ⇒ {
  setTimeout(() ⇒ resolve("P1 Success"), 3000);
});

const p2 = new Promise((resolve, reject) ⇒ {
  setTimeout(() ⇒ resolve("P2 Success"), 1000);
});

const p3 = new Promise((resolve, reject) ⇒ {
  setTimeout(() ⇒ resolve("P3 Success"), 2000);
});

Promise.all([p1, p2, p3]).then(res ⇒ {
console.log(res);
});
```

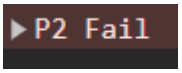
After 3 seconds.



2. Promise.all() → One Failure

```
const p1 = new Promise((resolve, reject) => {
  setTimeout(() => resolve("P1 Success"), 3000);
});

const p2 = new Promise((resolve, reject) => {
  setTimeout(() => reject("P2 Fail"), 1000);
});

const p3 = new Promise((resolve, reject) => {
  setTimeout(() => resolve("P3 Success"), 2000);
});

Promise.all([p1, p2, p3]).then(res => {
  console.log(res);
}).catch(err => {
  console.error(err);
});
```

After 1 second.

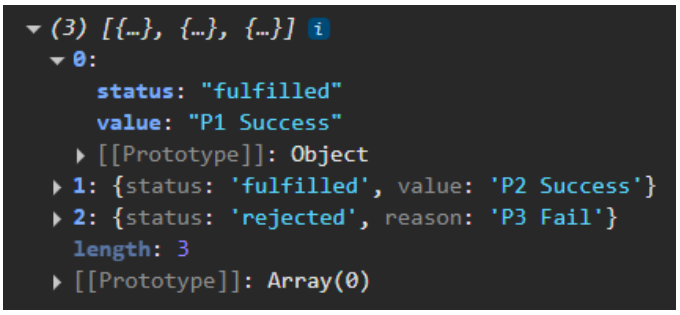


3. Promise.allSettled()

```
const p1 = new Promise((resolve, reject) => {
  setTimeout(() => resolve("P1 Success"), 3000);
});

const p2 = new Promise((resolve, reject) => {
```

```
  setTimeout(() => resolve("P2 Success"), 1000);
});

const p3 = new Promise((resolve, reject) => {
  setTimeout(() => reject("P3 Fail"), 2000);
});

Promise.allSettled([p1, p2, p3]).then(res => {
  console.log(res);
}).catch(err => {
  console.error(err);
});
```

After 3 seconds.

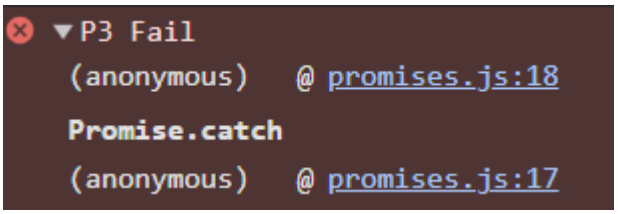


4. Promise.race()

```
const p1 = new Promise((resolve, reject) => {
  setTimeout(() => resolve("P1 Success"), 3000);
});

const p2 = new Promise((resolve, reject) => {
  setTimeout(() => resolve("P2 Success"), 5000);
});

const p3 = new Promise((resolve, reject) => {
  setTimeout(() => reject("P3 Fail"), 2000);
});

Promise.race([p1, p2, p3]).then(res => {
```

```
  console.log(res);
}).catch(err => {
  console.error(err);
});
```

After 2 seconds.



5. Promise.race()

   If two or more promises are taking same time, then result will be in the first occurred promise in the given array of promises.

```
const p1 = new Promise((resolve, reject) => {
  setTimeout(() => reject("P1 Fail"), 3000);
});

const p2 = new Promise((resolve, reject) => {
  setTimeout(() => resolve("P2 Success"), 2000);
});

const p3 = new Promise((resolve, reject) => {
  setTimeout(() => resolve("P3 Success"), 2000);
});

Promise.race([p1, p2, p3]).then(res => {
  console.log(res);
}).catch(err => {
  console.error(err);
});
```

After 2 seconds.

```
P2 Success
```

6. Promise.any()

```
const p1 = new Promise((resolve, reject) => {
  setTimeout(() => reject("P1 Fail"), 2000);
});

const p2 = new Promise((resolve, reject) => {
  setTimeout(() => resolve("P2 Success"), 3000);
});

const p3 = new Promise((resolve, reject) => {
  setTimeout(() => resolve("P3 Success"), 5000);
});

Promise.any([p1, p2, p3]).then(res => {
  console.log(res);
}).catch(err => {
  console.error(err);
});
```

After 3 seconds

```
P2 Success
```

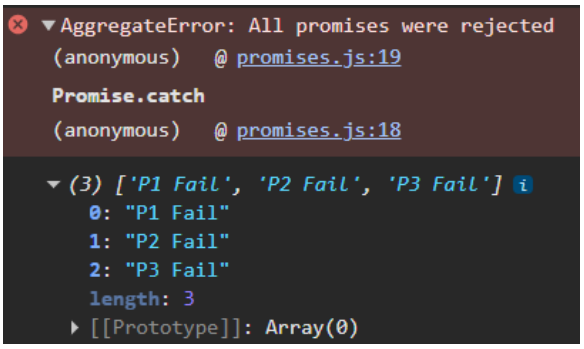
7. Promise.any()

All Failed → Return **Aggregate Error** with array of errors

```
const p1 = new Promise((resolve, reject) => {
  setTimeout(() => reject("P1 Fail"), 5000);
});

const p2 = new Promise((resolve, reject) => {
  setTimeout(() => reject("P2 Fail"), 1000);
```

```
});

const p3 = new Promise((resolve, reject) => {
  setTimeout(() => reject("P3 Fail"), 2000);
});

Promise.any([p1, p2, p3]).then(res => {
  console.log(res);
}).catch(err => {
  console.error(err);
  console.log(err.errors);
});
```

After 5 seconds



8. Promise.any()

If two or more promises are taking same time, then result will be in the first occurred successful promise in the given array of promises.

```
const p1 = new Promise((resolve, reject) => {
  setTimeout(() => reject("P1 Fail"), 5000);
});

const p2 = new Promise((resolve, reject) => {
  setTimeout(() => resolve("P2 Success"), 3000);
});

const p3 = new Promise((resolve, reject) => {
```

```
  setTimeout(() => resolve("P3 Success"), 3000);
});

Promise.any([p1, p2, p3]).then(res => {
  console.log(res);
}).catch(err => {
  console.error(err);
});
```

After 3 seconds.

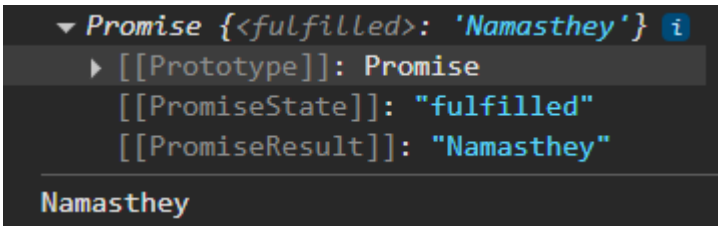
```
P2 Success
```

Async & Await

1. Async always returns a promise.

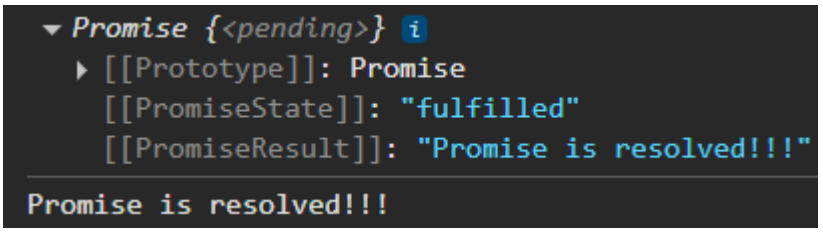
- When normal value is returned, It is wrapped in a promise and then it is returned.

```
async function getData() {
  return "Namasthey";
}

const data = getData();
console.log(data);

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



- When promise is returned, It is returned as it is.

```
const p = new Promise((resolve, reject) =>{
   resolve("Promise is resolved!!!");
});

async function getData() {
   return p;
}

const data = getData();
console.log(data);

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



2. Async and Await are used to handle promises.

3. **await keyword always should be used inside async function.**

```
const p = new Promise((resolve, reject) =>{
   resolve("Promise is resolved!!!");
});

async function handlePromise() {
   const val = await p;
   console.log(val);
}

handlePromise();

function getData() {
   p.then((res) => console.log(res));
```

```
}

getData();
```

```
Promise is resolved!!!
Promise is resolved!!!
```

4. Difference b/w normal approach and async await approach

- Normal approach (Here JS Engine will not wait until promise is resolved and moves to next lines of code.)

```
const p = new Promise((resolve, reject) =>{
   setTimeout(() => {
      resolve("Promise is resolved!!!");
   }, 10000);
});

function getData() {
   // JS Engine will not wait till promise is resolved
   p.then((res) => console.log(res));
   console.log("Namasthey JS");
}

getData();
```

```
Namasthey JS
Promise is resolved!!!
```

- aysnc await approach (Here JS Engine waits for promise to be resolved and then only moves to next lines of code in the async function)

```
const p = new Promise((resolve, reject) =>{
   setTimeout(() => {
      resolve("Promise is resolved!!!");
   }, 10000);
```

```
});

async function handlePromise() {
    console.log("Hello World!")
    // JS Engine waits for the promise to be resolved
    const val = await p;
    console.log("Namasthey JS");
    console.log(val);
}
```

This result is obtained after 10 seconds. Hello World! is printed immediately, but other lines are printed after 10 seconds.

```
Hello World!
Namasthey JS
Promise is resolved!!!
```

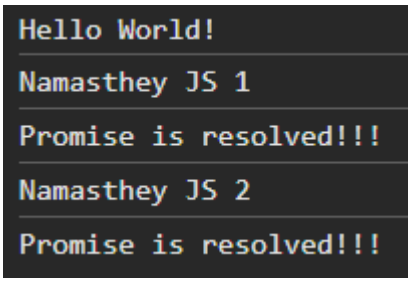
5. Other Scenarios

- Example 1:

```
const p = new Promise((resolve, reject) =>{
    setTimeout(() => {
        resolve("Promise is resolved!!!");
    }, 10000);
});

async function handlePromise() {
    console.log("Hello World!")
    // JS Engine waits for the promise to be resolved
    const val1 = await p;
    console.log("Namasthey JS 1");
    console.log(val1);

    const val2 = await p;
    console.log("Namasthey JS 2");
```

```
    console.log(val2);
}
```

Here, Hello World! is printed immediately and After 10 seconds all the logs in the function are printed at a time. It does wait here twice.

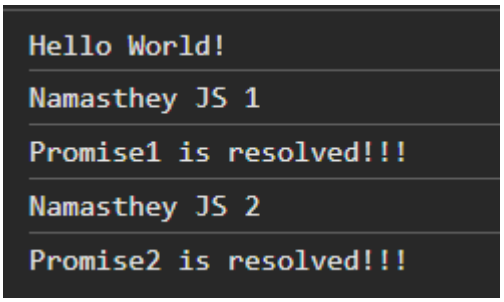


- Example 2:

```
const p1 = new Promise((resolve, reject) =>{
    setTimeout(() => {
        resolve("Promise1 is resolved!!!");
    }, 10000);
});

const p2 = new Promise((resolve, reject) =>{
    setTimeout(() => {
        resolve("Promise2 is resolved!!!");
    }, 5000);
});

async function handlePromise() {
    console.log("Hello World!")
    const val1 = await p1;
    console.log("Namasthey JS 1");
    console.log(val1);

    const val2 = await p2;
    console.log("Namasthey JS 2");
    console.log(val2);
}
```

```
handlePromise();
```

Here also Hello World! is printed immediately and After 10 seconds all the logs in the function are printed at a time. It does wait here twice.

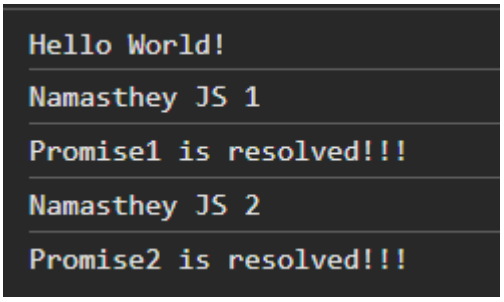


- Example 3:

```
const p1 = new Promise((resolve, reject) =>{
    setTimeout(() => {
        resolve("Promise1 is resolved!!!");
    }, 5000);
});

const p2 = new Promise((resolve, reject) =>{
    setTimeout(() => {
        resolve("Promise2 is resolved!!!");
    }, 10000);
});

async function handlePromise() {
    console.log("Hello World!")
    const val1 = await p1;
    console.log("Namasthey JS 1");
    console.log(val1);

    const val2 = await p2;
    console.log("Namasthey JS 2");
    console.log(val2);
}
```

```
handlePromise();
```

Here also Hello World! is printed immediately and After 5 seconds Namasthey JS 1 and Promise1 is resolved!!! are printed and after another 5 seconds Namasthey JS 2 and Promise2 is resolved!!! are printed.



6. Generally when we say JS Engine waits for the promise to be resolved in async await approach, It does not mean that JS Engine is waiting actually. It seems to be but what is happening here is when JS engine notices await, it terminates the async function from call stack and waits for promise to be resolved. Once resolved it again calls async function to call stack and executes from the same line from where it is left previously.

7. In more techinal terms:

When an `await` keyword is encountered within an `async` function, the JavaScript engine doesn't literally halt its execution. Instead, it gracefully pauses the execution of that specific `async` function. The engine then relinquishes the call stack, allowing other tasks in the event loop (like handling user interactions or other pending promises) to proceed. Once the awaited promise settles (either resolves or rejects), the JavaScript engine schedules the `async` function to be resumed. When the call stack becomes available again, the engine pushes the `async` function back onto the stack and execution continues from the exact line following the `await`.

Here's a breakdown of why this refined statement is more precise:

- **"gracefully pauses" instead of "terminates":** While the function does leave the call stack, "terminates" might imply a complete stop. "Pauses" better captures the temporary suspension.
- **"relinquishes the call stack"**: This clearly explains what happens to the function's execution context.
- **"allowing other tasks in the event loop... to proceed"**: This highlights the non-blocking nature of the operation and what the engine does while waiting.
- **"settles (either resolves or rejects)"**: This clarifies that the resumption happens regardless of the promise's outcome.
- **"schedules the `async` function to be resumed"**: This emphasizes that it's not an immediate return to execution but a scheduling within the event loop.
- **"When the call stack becomes available again"**: This adds context about when the resumed execution can actually occur.
- **"from the exact line following the `await`"**: This reinforces the point about maintaining the function's state.

8. Working of fetch and Real time example Aysnc Await

Fetch: → fetch() ==Promise>> Response.json ==Promise>> JsonValue

→ fetch().then((res) => res.json()).then((res) => console.log(json));

```
const API_URL = "https://api.github.com/users/Hruthik997";

async function handlePromise() {
  const responseJson = await fetch(API_URL);

  const jsonValue = await responseJson.json();

  console.log(jsonValue);

}
```

```
handlePromise();
```

9. Error Handling (Using try catch block)

```
const API_URL = "https://InvalidUrl";

async function handlePromise() {
   try {
      const responseJson = await fetch(API_URL);

      const jsonValue = await responseJson.json();

      console.log(jsonValue);
   }
   catch (err) {
      console.log(err);
   }
}

handlePromise()
```

```
const API_URL = "https://InvalidUrl";

async function handlePromise() {
  const responseJson = await fetch(API_URL);
  const jsonValue = await responseJson.json();
  console.log(jsonValue);
}

handlePromise().catch(err => console.log(err));
```

This keyword

- **"this" in the global scope:**
  - Refined Statement: In the global execution context, outside of any function, "this" refers to the global object. In browsers, this is typically the `window` object. In Node.js, it's the `global` object.
  - Explanation: The global scope is the outermost scope in a JavaScript environment. `this` behaves consistently here.
- **"this" in a function:**
  - The value of "this" inside a regular function is determined by how the function is called (its invocation context), and is affected by strict mode:
    - Non-strict mode: If the function is called as a standalone function, "this" is bound to the global object. If "this" is `undefined` or `null`, it is replaced with the global object.
    - Strict mode: If the function is called as a standalone function, "this" is `undefined`.
  - Explanation: It's crucial to distinguish between strict and non-strict modes. "Standalone" here means the function is invoked without being attached to an object (e.g., `myFunction()`).
- **"this" in a method:**
  - When a function is called as a method of an object, "this" is set to the object on which the method is called.
  - Explanation: This is the most common and intuitive use of "this." For example, in `obj.myMethod()`, "this" inside `myMethod` refers to `obj`.
- **call, apply, and bind:**
  - The `call()`, `apply()`, and `bind()` methods allow you to explicitly set the value of "this" when invoking a function. This enables you to call a function with a different "this" value than it would normally have.

        - `call()` and `apply()`: Immediately invoke the function with the specified "this" value. `call()` takes arguments individually, while `apply()` takes them as an array.
        - `bind()`: Creates a new function that, when called, has its "this" value permanently set to the provided value.
    - Explanation: These methods provide powerful ways to control the context of function execution.
- **"this" in arrow functions:**
    - Arrow functions do not have their own "this" binding. Instead, they lexically inherit the "this" value from their enclosing execution context (the context where they are defined).
    - Explanation: Arrow functions behave differently from regular functions. Their "this" is determined by the surrounding code, not by how they are called.
- **"this" in DOM event handlers:**
    - Inside a DOM event handler function, "this" generally refers to the specific DOM element that triggered the event (the event target).
    - Explanation: When an event occurs (like a click), and a function is called to handle that event, "this" within that function will usually point to the HTML element that was clicked or otherwise interacted with.

Examples:

```
"use strict";

// *** this in global space ***
console.log(this); // Global object -> Window in browsers and Global in Nod
e JS

// *** this inside function ***
function x () {
   // the value depends upon strict and non strict mode
```

```
    // In Strict mode → undefined, In Non Strict mode → windows object
    console.log(this);
}

// this inside non-strict mode (this substitution)
/**If the value of this keyword is undefined or null, this will be replaced with
globalObject only in non strict mode */


// value of this keyword depends on how the function is called (window)


x(); // When you call it without any reference of an object → undefined
window.x(); // → In this case again this is equal to windows object

// *** this inside a object's method ***
const obj = {
    a:10,
    // Here x is a method of object obj
    x: function () {
        console.log(this);
        console.log(this.a);  // 10
    }
}

obj.x(); // this represents object obj itself


// *** call apply and bind methods (sharing methods) ***

const student1 = {
    name: "Hruthik",
    printName: function name() {
        console.log(this.name);
    }
}

student1.printName();
```

```
const student2 = {
   name: "Harsha"
}

// Here I want to use printName function on obj student2
student1.printName.call(student2);
// → Here basically you are replacing this object of student1 object with stu
dent2

// *** this inside arrow functions ***
const obj2 = {
   a:10,
   // Here x is a method of object obj
   x: () => {
      console.log(this);
   }
}

obj2.x(); // In this case this keyword refers to value of enclosing lexical cont
ext → Here, Global Object

const obj3 = {
   a: 10,
   // Here x is a method of object obj
   x: function () {
      // enclosing lexical context
      const y = () => {
         console.log(this);
      };
      y();
   }
}

obj3.x(); // Here this keyword refers to object obj3


// *** this inside DOM elements *** ⇒ Reference to HTML element
```

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial
    <title>Document</title>
  </head>
  <body>
    <h1>Namaste JavaScript</h1>
    <button onclick="alert(this.tagName)">Click Me</button>
    <script src="./index.js"></script>
  </body>
</html>
```

Call, apply and bind methods

1. call, apply, and bind are indeed about manipulating the this context of a function, allowing you to "borrow" or reuse functions with different objects. Here's a more detailed explanation with examples:
2. **Understanding this (Refer the above section)**

Before diving into the methods, it's crucial to understand this in JavaScript. The value of this depends on *how* a function is called. It can refer to:

- The global object (window in browsers).
- An object that called the function (as a method).
- A new object (in a constructor function).
- Explicitly defined context using call, apply, or bind.

3. **call()**

- **Purpose:** Calls a function with a specified this value and arguments provided *individually*.
- **Syntax:** functionName.call(thisArg, arg1, arg2, ...)
- thisArg: The value of this inside functionName.
- arg1, arg2, ...: Arguments to be passed to functionName.

Example:

```
const student = {

firstName: "John",

lastName: "Doe",

};

function printFullName(city, state) {

console.log(this.firstName + " " + this.lastName + ", " + city + ", " + state);

}

printFullName.call(student, "Jammu", "J&K"); // Output: John Doe, Jammu, J&K
```

- Here, call() is used to invoke printFullName.
- The this value inside printFullName is set to the student object. So, this.firstName refers to student.firstName, and this.lastName refers to student.lastName.
- "Jammu" and "J&K" are passed as individual arguments to printFullName.

4. **apply()**

- **Purpose:** Similar to call(), but takes arguments as an *array* (or an array-like object).
- **Syntax:** functionName.apply(thisArg, [arg1, arg2, ...])
- thisArg: The value of this inside functionName.
- [arg1, arg2, ...] : An array of arguments to be passed to functionName.

Example:

```
const student = {
```

```
firstName: "John",

lastName: "Doe",

};

function printFullName(city, state) {

console.log(this.firstName + " " + this.lastName + ", " + city + ", " + state);

}

printFullName.apply(student, ["Jammu", "J&K"]); // Output: John Doe, Jam
mu, J&K
```

- The key difference is that the arguments "Jammu" and "J&K" are passed as an array: ["Jammu", "J&K"].

5. **bind()**

- **Purpose:** Creates a *new* function with the this value bound to the specified object. It does *not* immediately execute the function.
- **Syntax:** const boundFunction = functionName.bind(thisArg, arg1, arg2, ...)
- thisArg: The value of this inside the *new* function.
- arg1, arg2, ...: (Optional) Arguments to be *pre-pended* to any arguments passed when the bound function is eventually called.
- boundFunction: The new function created by bind(). You must call this function to execute the original function.

Example:

```
const student = {

firstName: "John",
```

```
lastName: "Doe",

};

function printFullName(city, state) {

console.log(this.firstName + " " + this.lastName + ", " + city + ", " + state);

}

const printStudent = printFullName.bind(student, "Udhampur", "J&K");

printStudent(); // Output: John Doe, Udhampur, J&K
```

- `bind() creates a new function called printStudent.`
- `Inside printStudent, this is permanently set to the student object, and the argument "Udhampur" is pre-set.`
- `When printStudent() is called, it executes printFullName with the bound this (student) and the pre-set "Udhampur" and "J&K".`