Callback Hell

- Where we have dependencies on asynchronous API calls, we need to implement them with callback.
- If the dependencies are chaining a lot, it will make the code unreadable and unmaintainable. With this, code will starts to grow horizontally instead of vertically.
- This is called **callback hell**. And this structure is called **Pyramid of Doom**
- To resolve this, promise was introduced.

```
const cart = ["shoes", "pants", "kurta"];
api.createOrder(cart, function (){
    api.proceedToPayment(function (){
        api.showOrderSummary(function () {
            api.updateWallet();
        });
    });
});
```

Inversion of control is another problem when we use **callback hell**. Inversion of control says that we will lose control over our code when we use callbacks.

Considering the higher order function is some API written somewhere, When we pass callback function, we are blindly trusting that the function accepting the callback will call our function. We don't know how our function is treated. It may never get called, it may be called twice. We're giving our whole logic to other person's code. Thus, we're losing the control on our code. This is called **Inversion of Control**.

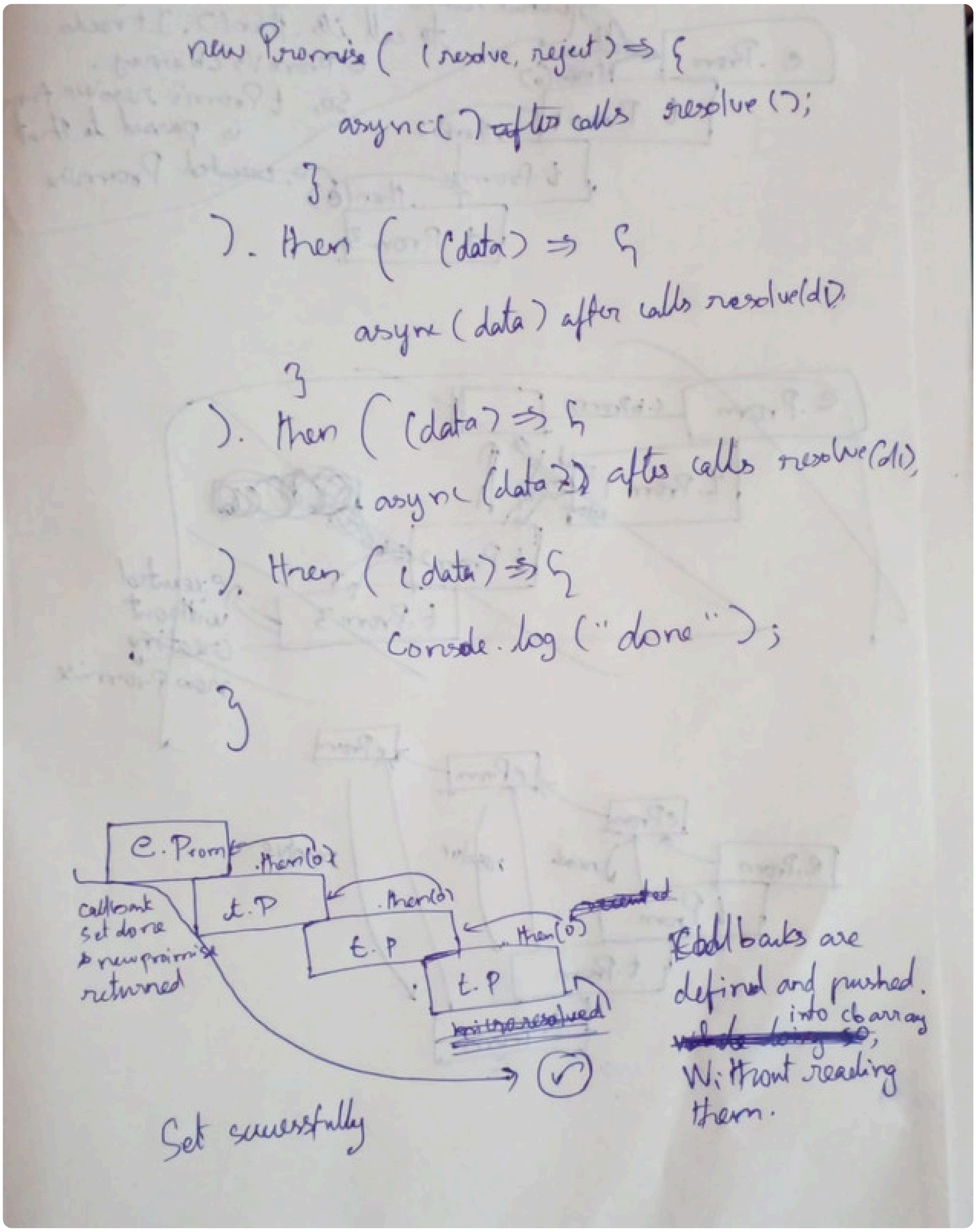
Promises

Promises are introduced to overcome the problem of **Callback Hell** and **Inversion of Control**.

According to MDN Docs, **Promise** is the object representing the eventual completion or failure of a asynchronous operation.

Chaining of Promises

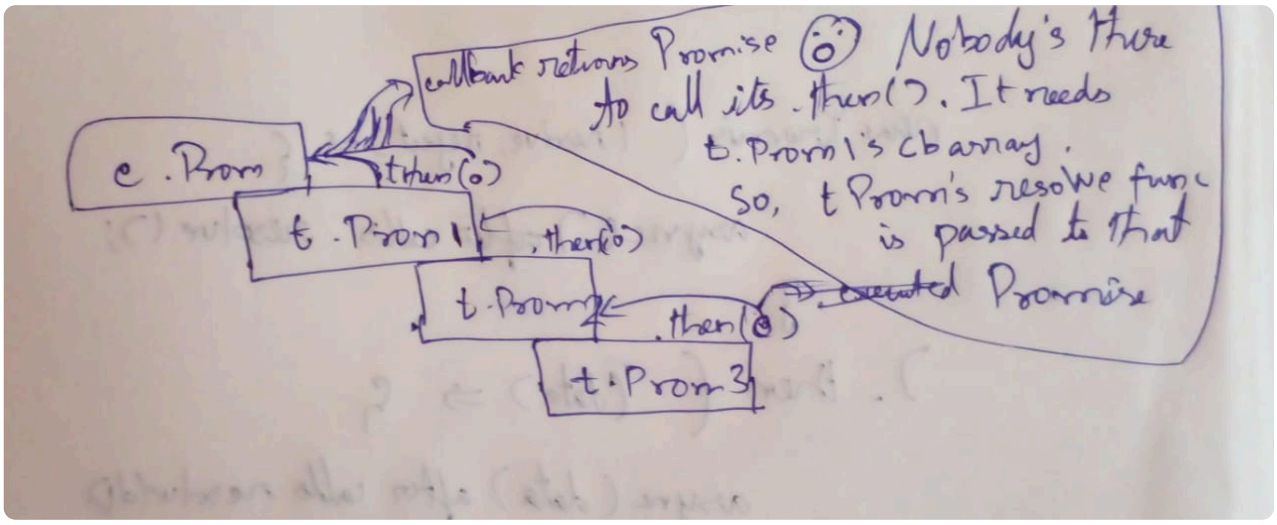
Implemention of promise [here](https://medium.com/swlh/implement-a-simple-promise-in-javascript-20c9705f197a) (https://medium.com/swlh/implement-a-simple-promise-in-javascript-20c9705f197a)

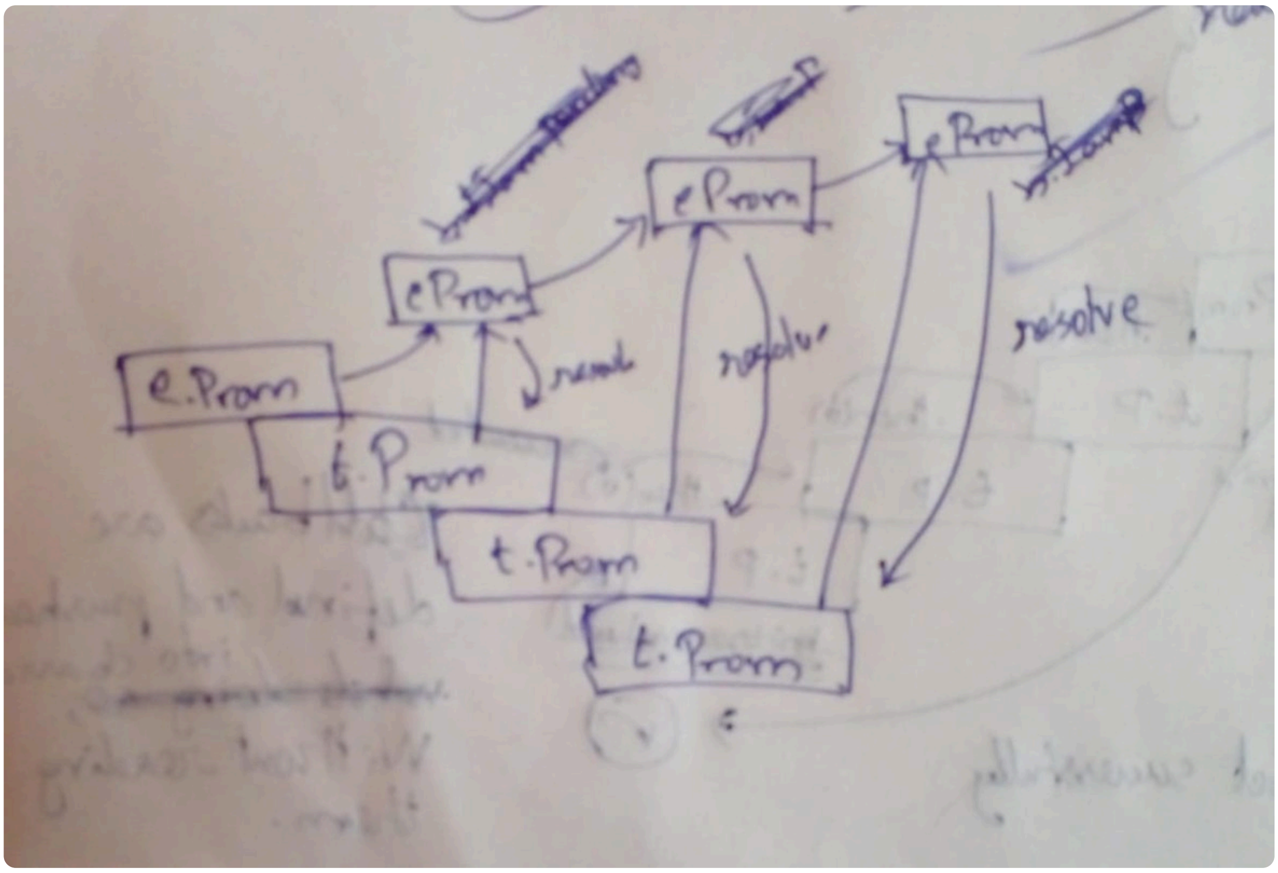


```
new Promise( (resolve, reject) => {
        async() ~~after~~ calls resolve();
    };
). then ( (data) => {
        async (data) after calls resolve(d1);
    }
). then ( (data) => {
        async (data2) after calls resolve(d1);
). then ( (data) => {
        console.log ("done");
}
```

e.Prom
.then()
t.P
.then()
t.P
.then()
t.P
callback set done & new promise returned
~~created~~
~~with the resolved~~
Set successfully

Callbacks are defined and pushed into cb array ~~while doing so~~, Without reading them.


callback returns Promise
Nobody's there to call its .then(). It needs t.Prom1's cb array. So, t.Prom's resolve func is passed to that ~~created~~ Promise
e.Prom
.then(o)
t.Prom1
.then(o)
t.Prom
.then(o)
t.Prom3



In the above picture, `e.Promise` are the promise returned by executors (handlers) whereas `t.promise` are promises returned by `.then()` invocation.

So, the Promises returned by `.then()` expects that function `onFulfilled` return normal data and we can resolve ourselves with that data.

This will only happen in the last `.then()` where the callback `onFullfilled` is synchronous and returns some value or `undefined`.

In other `.then()` returned promises, they unexpectedly returned Promises (`e.Promises`). So, we need to assign the callbacks passed for ourself (using `.then()`) to this internally returned `e.Promise`. So, in such cases, this promise won't resolve itself and gives it's `resolve` to that internally returned `e.Promise` by calling `e.Promise.then(resolve, reject);` . So, they can access current `p.Promise` 's callback and run them. This is how chains in promises are created.

Applications of Promises

It's not neccessary to know the internal implementation of promises. Promises are actually the syntactic sugar which inside uses normal callbacks only.

In this way, the promise solves the problem **Pyramid of Doom**.

When we use promises, let's say we use `fetch()` function. Here with this, we're not passing our function. Instead, we receive the promise object which is created by Browser.

Browser internally makes API call for the corresponding website and return a promise with pending.

Once, it gets the result, it will update the value of the promise with data received from API and fires the callbacks attached to the Promise. So, in such places, we're not passing our callbacks to APIs and passing to our browser's implementation only. So, in such cases, browser's Promise implementation is designed such that it will only call the attached callbacks only once. So, here, **Inversion of Control** is also solved.

What if we tend to use Promises returned by other code in our module instead of calling API ?

In such cases, If they use same Promise as of ECMAScript provides, they can't be able to call our callbacks more than once (as it is designed by ECMAScript people). If custom promise is returned, then that's not handled.

> Callbacks will be attached to promises using `.then()` . We can attach multiple callbacks to singlle promise objects. Those callbacks were executed when the promise resolved. They will be executed in the same order on which they attached.

Working of Promise

Promise have 3 states. `pending` , `fulfilled` , `rejected` .

Promises will have values. We will set the value by calling `resolve(value)`. This will set the value of the promise and also sets the status of promise as `fulfilled`. This value will be passed as the input for callback function attached to promise by `.then()` method.

Example

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

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

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

Understanding Rejection Propagation in Promises

When dealing with promise rejections, it’s important to understand how errors propagate in a promise chain and how `.catch()` works to handle those errors. Promises have a **built-in mechanism for propagating rejections**, which gives you flexibility in handling errors.

Rejection Propagation

Rejection Propagation

1. **Rejections Propagate Through the Chain:**
   - If a promise is rejected, the rejection propagates to the next `.catch()` or `.then(onRejected)` in the chain.
   - If no `.catch()` or rejection handler is present at a certain step in the chain, the rejection continues to propagate down the chain.

2. **Where to Handle Errors?**

    - You **can handle errors at specific points in the chain** (e.g., in the middle).
    - If you don't handle the error at intermediate steps, you can **handle it at the end of the chain** with a single `.catch()`.

Example: Handling Errors at Different Points

```
const promise = new Promise((resolve, reject) => {
  reject(new Error("Initial error")); // The promise is rejected here
});

// Example 1: Handling the error at the end
promise
  .then(value => {
    console.log("Resolved:", value); // Won't execute
  })
  .then(value => {
    console.log("Continued with:", value); // Won't execute
  })
  .catch(err => {
    console.log("Caught at the end:", err.message); // Output: Caught at the
end: Initial error
  });

// Example 2: Handling the error in the middle
promise
  .then(value => {
    console.log("Resolved:", value); // Won't execute
  })
  .catch(err => {
    console.log("Caught in the middle:", err.message); // Output: Caught in
the middle: Initial error
    return "Recovered"; // Resolves the chain with a new value
  })
  .then(value => {
    console.log("Continued with:", value); // Output: Continued with:
Recovered
  });
```

- In **Example 1**, the error propagates all the way to the end because there's no handler in the middle.
- In **Example 2**, the rejection is handled in the middle, and the chain continues as resolved.

The Role of `.then()` and `.catch()`

When you attach `.then()` and `.catch()` at different points, their behavior depends on **where the rejection originates** and whether there's an intermediate handler.

Key Points to Understand

1. **`.then()` Creates a New Promise:**
    - Every `.then()` call returns a new promise object.
    - If the `.then()` does not handle the rejection, the rejection propagates to the next promise in the chain.

2. **`.catch()` Handles Rejections:**
    - `.catch()` handles rejections for the promise to which it is attached.
    - If there's no rejection for that promise, `.catch()` does nothing.

3. **Rejections Not Handled by `.catch()`:**
    - If a rejection is not handled (neither by `.catch()` nor by `.then(onRejected)`), it propagates and eventually results in an **unhandled promise rejection**.

Behavior When `.then()` and `.catch()` Are Attached Separately

Let's analyze this scenario:

```
const promise = new Promise((resolve, reject) => {
  reject(new Error("Rejected in executor"));
});

promise
  .then(value => {
    console.log("Resolved with:", value); // Won't execute
  });

promise
  .catch(err => {
    console.log("Caught in .catch():", err.message);
  });

/*
Caught in .catch(): Rejected in executor
promise.js:2 Uncaught (in promise) Error: Rejected in executor
    at promise.js:2:12
    at new Promise (<anonymous>)
    at promise.js:1:17
*/
```

What Happens Here?

1. When `promise` is rejected, the rejection is available for the **original promise**.
2. The `.then()` attached to the promise creates a **new promise**, but it does not handle the rejection because there's no `onRejected` callback in the `.then()`.
3. The `.catch()` is attached directly to the **original promise**, so it catches the rejection and logs the error.

Behavior with Multiple `.then()` and `.catch()`

Let's take a more complex example:

```
const promise = new Promise((resolve, reject) => {
  reject(new Error("Initial rejection"));
});

promise
  .then(value => {
    console.log("Resolved with:", value); // Won't execute
  })
  .catch(err => {
    console.log("Caught at first .catch():", err.message); // Output: Caught
at first .catch(): Initial rejection
    throw new Error("Error after recovery");
  })
  .then(value => {
    console.log("Continued with:", value); // Won't execute
  })
  .catch(err => {
    console.log("Caught at second .catch():", err.message); // Output: Caught
at second .catch(): Error after recovery
  });
```

What Happens Here?

1. The rejection from the `promise` propagates to the first `.catch()` because there's no handler in the `.then()` .
2. The first `.catch()` handles the rejection but throws a new error (`Error after recovery`).
3. The new error propagates to the next `.catch()` .

Strong Emphasis on `.catch()` Propagation Capability

1. `.catch()` **has rejection propagation capability**:
    - If a rejection occurs at any point in the chain and there's no intermediate handler, `.catch()` will handle the error.
    - This is why you only need one `.catch()` at the end of the chain for general error handling.

2. Rejections propagate through promises in the chain until they are handled.

When to Use Multiple `.catch()`

Use multiple `.catch()` calls when:

1. You want to handle specific errors at certain points in the chain.
2. You want to recover from an error at one stage and continue with subsequent processing.

For example:

```
 1  new Promise((resolve, reject) => {
 2    reject(new Error("First rejection"));
 3  })
 4    .then(value => {
 5      console.log("Resolved with:", value); // Won't execute
 6    })
 7    .catch(err => {
 8      console.log("Caught first error:", err.message); // Output:
 9      // Caught first error: First rejection
10      throw new Error("Second rejection"); // Re-throw a new error
11    })
12    .then(value => {
13      console.log("Continued with:", value); // Won't execute
14    })
15    .catch(err => {
16      console.log("Caught second error:", err.message); // Output:
17      // Caught second error: Second rejection
18    });
```

Graceful handling and proceeding the promise chain:

If we want to proceed with the chaining eventhough the previous promise was rejected, we can write `.catch()` just below that method which handles the error gracefully. If so, then the remaining `.then()` methods that are present below the `.catch()` will be executed successfully. In above examples, it won't as the first catch method throws Error without gracefully handling them.

If the line no.9 is commented, then `.then()` with "Continued with: " will also execute.

So, if we want to execute the remaining `.then()` or wanted to handle errors differently when it occured in different places of process, then we can use multiple `.catch()` in multiple places of chain.

Promise APIs + Interview Questions

Promise.all()

Whenever we want multiple API calls to be executed simultaneously, we can go for `Promise.all()` . Here, `Promise.all()` will get the iterable as an input. It gets iterable of Promises. Here, we will consider array of Promises.

When such promises created, all will be executed.

Case 1 - All Promise fulfilled

Let's say p1, p2 and p3 are the promises that are passed to `Promise.all()` . p1 takes 1s, p2 takes 3s and p3 takes 2 seconds. In this case, `Promise.all()` waits for all the promise to be completed (waits for 3s) and gives the array of results (results from those promises).

```
const p1 = new Promise((resolve, reject) => {
  setTimeout(() => resolve("P1 Success"), 1000);
});

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

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

Promise.all([p1, p2, p3]).then((res) => {
  console.log(res);
});
// [P1 Success, P2 Success, P3 Success] (after 3 seconds)
```

Case 2 - Any of Promises failed

Here, In same example, p3 got failed after 2 seconds. In such case, as soon as p3 got failed, `Promise.all()` will also fails and throws the same error message that p3 thrown. It doesn't wait for other remaining promises to complete. As soon as it gets rejection in any of its promises, it rejects with same error.(single error value will be returned to catch block).

```
const p1 = new Promise((resolve, reject) => {
  setTimeout(() => resolve("P1 Success"), 1000);
});

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

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

Promise.all([p1, p2, p3]).then((res) => {
  console.log(res);
}).catch(err => {
    console.error(err);
})
// P3 Fails (after 2 seconds)
```

Promise.allSettled()

Case 1 - All Promise fulfilled

In this, the same behaviour of `Promise.all()` will be observed. So, it will return the array of all succeed promises after all promise get settled (succeeded).

Case 2 - Any of Promises failed

In this case, it didn't suddenly returns or throws error. Instead, it will wait for all the three promises to be settled (fulfilled or rejected) (will wait for 3 seconds), and will return the final array of results.

Let's say p1, p2, p3 are the promises and p1 fulfilled after 1seconds, p2 fulfilled after 3s and p3 rejected after 2s, then the final `Promise.allSettled()` will return [val1, val2, error3] after 3s.

So, in the cases where we need to show outputs of 5 APIs in 5 cards, we can use `Promise.allSettled()` and in there, if any of the `Promise` fails, just show the remaining cards.

```
const p1 = new Promise((resolve, reject) => {
  setTimeout(() => resolve("P1 Success"), 1000);
});

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

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

Promise.all([p1, p2, p3]).then((res) => {
  console.log(res);
}).catch(err => {
    console.error(err);
});
//
/*  [
  { status: 'fulfilled', value: 'P1 Success' },
  { status: 'fulfilled', value: 'P2 Success' },
  { status: 'rejected', reason: 'P3 Fails' }
] */
//    (after 3 seconds) (from .then())
```

Promise.race()

Here, as the name suggests, it's a race. Whichever promise get settled first, it will return the result of that promise. Thus, the return of the `Promise.race()` is a single value, not an array.

Here, the first settling promise may be succeed or may be got rejected. If got rejected, then that error will be return by `Promise.race()` . So, irrespective of success or failure, first settling promise will be considered and result of that promise will be returned by `Promise.race()` .

```
const p1 = new Promise((resolve, reject) => {
  setTimeout(() => resolve("P1 Success"), 1000);
});

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

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

Promise.all([p1, p2, p3]).then((res) => {
  console.log(res);
}).catch(err => {
    console.error(err);
});
/*
 * P1 Success (after 1s from .then())
 */
```

```
const p1 = new Promise((resolve, reject) => {
  setTimeout(() => reject("P1 Fails"), 1000);
});

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

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

Promise.all([p1, p2, p3]).then((res) => {
  console.log(res);
}).catch(err => {
    console.error(err);
});
/*
 * P1 Fails (after 1s from .catch())
 */
```

Promise.any()

Here, it's similar to `Promise.race()` but the difference lies in that, `Promise.any()` waits for the first promise to be resolved (succeeded - fulfilled) instead of first promise to be settled. And thus, it will return the value of the first resolved promise.

If all of the promises failed, then after the last promise fails, the `Promise.any()` returns the aggregated error. It means that it returns the array of errors returned by all the Promises. But inroder to get that array of errors, we need to access them through `.errors` property.

```
const p1 = new Promise((resolve, reject) => {
  setTimeout(() => reject("P1 Fails"), 1000);
});

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

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

Promise.all([p1, p2, p3]).then((res) => {
  console.log(res);
}).catch(err => {
    console.error(err);
});
/*
 * P2 Success (after 3s from .then())
 */
```

```
const p1 = new Promise((resolve, reject) => {
  setTimeout(() => reject("P1 Fails"), 1000);
});

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

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

Promise.all([p1, p2, p3]).then((res) => {
  console.log(res);
}).catch(err => {
    console.error(err);
    console.log(err.errors);
});
/*
 * Aggregate Error: All promise were rejected
 * [P1 Fails, P2 Fails, P3 Fails]
 * (after 3s from .catch())
 */
```

Settled	- Got the result
resolve	reject
success	failure
fulfilled	rejected

Async await

What is async in JavaScript ?

`async` is a keyword where we can use it before the function keyword on function declaration to make the function asynchronous.

Asynchronous function will always return a promise.

If we return some non-promise value, it will automatically be wrapped with Promise and returned. If we return explicit promise, then corresponding promise will be returned when the function get called.

```
async function getData(){
    return "Hello";
}
const dataPromise = getData(); // will return Promise (here the status of the
promise if fulfilled as we didn't do any async operation inside the getData()
function to return the data)
dataPromise.then(res => console.log(res)); // Hello
```

```
async function getData(){
    return new Promise((resolve, reject) => {
        resolve("Hello");
    });
}

const dataPromise = getData();
dataPromise.then(res => console.log(res));
console.log("EOP");
// EOP
// Hello
```

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

const dataPromise = getData();
dataPromise.then(res => console.log(res));
console.log("EOP");
// EOP
// Hello
```

```
async function getData(){
    console.log("getData() called");
}

const dataPromise = getData();
dataPromise.then(res => console.log(res));
console.log("EOP");
// getData() called
// EOP
// undefined
```

Using await with async

`async` and `await` combo is just used to handle Promises.

How do we used to handle Promises before `async` and `await` ?

Example 1

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

function handlePromise(){
    p.then((val) => {
        console.log(val);
    })
}

handlePromise();
console.log("EOP");

// EOP
// Promise resolved
```

Example 2

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

function handlePromise(){
    p.then(res => console.log(res));
    console.log("EOH - End of HandlePromise");
}

handlePromise();
console.log("EOP");

/*
async-await.js:13 EOH - End of HandlePromise
async-await.js:17 EOP
async-await.js:12 Promise resolved
*/
```

How do we handle Promise with `async` and `await` ?

The very important point is that `await` keyword must be used only inside `async` functions and only before `Promises` or some function returning `Promises` like `fetch()` .

Example 1

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

async function handlePromise(){
    const val = await p;
    // manipulate or process the value
    console.log(val);
}

handlePromise();
console.log("EOP");

// EOP
// Promise resolved
```

Example 2

Here, one of the weird behaviour of JavaScript will be noticed 😳.
JavaScript actually waits for `await p` line to be completed.

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

async function handlePromise(){
    const val = await p;
    // manipulate or process the value
    console.log(val);
    console.log("EOH - End of HandlePromise");
}

handlePromise();
console.log("EOP");

/*
async-await.js:20 EOP (immediately)
async-await.js:10 Promise resolved (after 10s)
async-await.js:11 EOH - End of HandlePromise (after 10s)
*/
```

Example 3

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

async function handlePromise(){
    console.log("SOH - Starting of HandlePromise");
    const val = await p;
    // manipulate or process the value
    console.log(val);
    console.log("EOH - End of HandlePromise");
}

handlePromise();

/*
async-await.js:8 SOH - Starting of HandlePromise
async-await.js:11 Promise resolved
async-await.js:12 EOH - End of HandlePromise
*/
```

Example 4

Do you think so ?? JS waiting for `await` ?

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

async function handlePromise(){
    console.log("SOH - Starting of HandlePromise");
    const val = await p;
    // manipulate or process the value
    console.log(val);
    console.log("EOH - End of HandlePromise");
}

setTimeout(() => {
    console.log("Meanwhile ... 😅😅");
    console.log("I am not waiting brooo .... Do you think that I will wait
😎");
    console.log("Varta mame ... Durrrrr 😅")
}, 3000);

handlePromise();

/*
async-await.js:8 SOH - Starting of HandlePromise
async-await.js:16 Meanwhile ... 😅😅
async-await.js:17 I am not waiting brooo .... Do you think that I will wait
😎
async-await.js:18 Varta mame ... Durrrrr 😅
async-await.js:11 Promise resolved
async-await.js:12 EOH - End of HandlePromise
*/
```

Example 5

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

async function handlePromise(){
    console.log("SOH - Starting of HandlePromise");
    const val = await p;
    // manipulate or process the value
    console.log(val);
    console.log("EOH - End of HandlePromise");
}

handlePromise();
console.log("EOP");

/*
async-await.js:8 SOH - Starting of HandlePromise
async-await.js:16 EOP
async-await.js:11 Promise resolved
async-await.js:12 EOH - End of HandlePromise
*/
```

Example 6

```
const p = new Promise((resolve, reject) => {
    setTimeout(() => {
        resolve("Promise resolved");
    }, 5000);
});


async function handlePromise(){
    console.log("SOH - Starting of HandlePromise");
    const val = await p;
    // manipulate or process the value
    console.log(val);
    console.log("EOH - End of HandlePromise");

    setTimeout(() => {console.log("Settimeout(0) inside handlePromise")}, 0);

    const val2 = await p;
    console.log(val2);
    console.log("EOH - End of HandlePromise");
}

handlePromise();
console.log("EOP");

/*
async-await.js:14 SOH - Starting of HandlePromise
async-await.js:30 EOP
async-await.js:17 Promise resolved
async-await.js:18 EOH - End of HandlePromise
async-await.js:23 Promise resolved
async-await.js:24 EOH - End of HandlePromise
async-await.js:20 Settimeout(0) inside handlePromise
*/
```

Example 7

```
const p = new Promise((resolve, reject) => {
    setTimeout(() => {
        resolve("Promise resolved");
    }, 5000);
});

const p2 = new Promise((resolve, reject) => {
    setTimeout(() => {
        resolve("Promise resolved");
    }, 6000);
});

async function handlePromise(){
    console.log("SOH - Starting of HandlePromise");
    const val = await p;
    // manipulate or process the value
    console.log(val);
    console.log("EOH - End of HandlePromise");

    setTimeout(() => {console.log("Settimeout(0) inside handlePromise")}, 0);

    const val2 = await p2;
    console.log(val2);
    console.log("EOH - End of HandlePromise");
}



handlePromise();
console.log("EOP");

/*
async-await.js:14 SOH - Starting of HandlePromise
async-await.js:30 EOP
async-await.js:17 Promise resolved
async-await.js:18 EOH - End of HandlePromise
async-await.js:20 Settimeout(0) inside handlePromise
async-await.js:23 Promise resolved
async-await.js:24 EOH - End of HandlePromise
*/
```

How `async` and `await` works behind the scenes

Let's take example 6.

- Here, promise `p` and `p2` is created when JS executing them.
- After that, when it reaches invocation of `handlePromise()` , it will go inside it.
- Then "SOH - Starting of HandlePromise" is printed on the screen.

- As soon as `await` is encountered, JS will check whether the mentioned promise is settled or nor (fulfilled or rejected). If it is settled, definitely value must be assigned to promise. So, it will take that value and assign it into `val`. And it will move executing other lines as normal.
- But what if, promise `p` is not resolved yet. In such cases, the function will suspended and got out of the stack.
- Meanwhile, if there is any other event happened like clicking button, or any other setTimeout() fired, that will be fetched from callback queue and loaded into main JS callstack and will be executed there.
- Once the promise `p` get resolved, it will again get pulled to JS callstack (it will wait on callback queue until that time)
- Once it pulled, it will start executing. Next two `console.log()` lines will be printed.
- When the next `setTimeout()` is detected, it registers that `seteTimeout()` and move to next `await` line.
- At that time, promise `p` will be resolved. As it is so, the value will be fetched and stored into `val2` and below two lines will be executed.
- After that, the `setTimeout()` will be waiting in callback queue. That will be loaded into the memory and the callback of `setTimeout()` will be executed.

When such `await` is noticed, this will be the behaviour of JavaScript. One example 7, the second `await` actually tied with `p2` which takes one seconds more (6s totally). So, in that case, the `handlePromise` will again be suspended. Meanwhile `setTimeout(0)` will be waiting in callback queue. That will be loaded into callstack and executed. And after 1s (after that `p2` get resolved), the resume of `handlePromise` will again be loaded from callback queue into callstack and will be executed.

Working of `fetch` :

- `fetch` will return a promise.
- When it's resolved, we will get a Response object. It will be in ReadableStream.
- When we want to parse it to json object, we need to call `.json()` on that. And also `.json()` is an asynchronous operation and it will return Promise.
- After that promise is resolved, we will get the value.

So, the handling of fetch will be

```
async function handleFetch(){
    const response = await fetch(API_URL);
    const jsonData = await response.json();
    console.log(jsonData);
}
handleFetch();
```

(or)

```
function handleFetch(){
    const promise = fetch(API_URL);
    promise.then(function (response) {
        return response.json();
    }).then(function (data) {
        console.log(data);
    })
}
handleFetch();
```

Real life example

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Run JavaScript File</title>
</head>
<body>
    <h1>JavaScript File Execution</h1>
    <div class = "profile"></div>
    <script src="fetch.js"></script>
</body>
</html>
```

```
async function handleFetch(){
    const response = await
fetch("https://api.github.com/users/Danusshkumar");
    const data = await response.json();
    console.log(data);
    let profileDiv = document.getElementsByClassName("profile")[0];
    let nameElement = document.createElement("h2");
    nameElement.textContent = data.login;
    let avatarElement = document.createElement("img");
    avatarElement.setAttribute("src", data.avatar_url);
    profileDiv.appendChild(nameElement);
    profileDiv.appendChild(avatarElement);
}

handleFetch();
```

Error Handling

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

// await can only be used inside an async function
async function handlePromise() {
  try {
    const data = await fetch(API_URL);
    const jsonValue = await data.json();
    console.log(jsonValue);
  } catch (err) {
    console.log(err);
  }
}

handlePromise();
```

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

// await can only be used inside an async function
async function handlePromise() {
  const data = await fetch(API_URL);
  const jsonValue = await data.json();
  console.log(jsonValue);
}

handlePromise().catch((err) => console.log(err)); // error get passed to the
promise that will be returned by `handlePromise`
```

`async` and `await` are syntactic sugar over the `Promises`.
Under the hood, JavaScript handles this `async`, `await` with `.catch()` and `.then()` methods only.

Usage of `async`, `await` syntax makes our code readable. So, for readability, we can go for this `async / await` syntax. But, EOD, the choice is upto Developer's preference.

this keyword in JavaScript

Invocation without reference

- `this` keyword in global context refers to the global object. Global object will be given by JS Runtime. In browser, it's window object.
- Inside function, the `this` keyword will work depends on its mode. If it's a non-strict mode, it will refers to the global object.
- In strict mode, the value of `this` inside function will be `undefined`

Whenever the `this` keyword is null or undefined, the value of `this` keyword will be replaced with globalObject by JavaScript. This will happen only in non-strict mode. This phenomenon is called **this substitution**.

That's why we encounter `undefined` in strict mode.

```
"use strict";
function x(){
    console.log(this);
}
x(); // will print undefined



function x(){
    console.log(this);
}
x(); // will print globalObject
```

Invocation with reference

> The value of `this` keyword depends on how the function is called.

Even with strict mode,

```
"use strict";
function x(){
    console.log(this);
}
window.x(); // will print globalObject
```

When the function that's using `this` is called without any reference, then it will become undefined. But, in above, we're calling it with `window` object. That's why `this` got binded to the object which is called.

```
const obj = {
    a : 5,
    func : function (){
        console.log(this);
    }
};

obj.func(); // prints the `obj` object
```

We can override the `this` value with some functions.

```
const student1 = {
    name : "Name 1",
    printName : function (){
        console.log(this.name);
    }
};

student1.printName(); // Name 1

const student2 {
    name : "Name 2"
};
student.printName.call(student2); // Name 2

// Above example is not original usecase. Bcz, the method that's gonna be
shared for all the students must be defined outside such student objects
```

`call()`, `apply()`, and `bind()`

These methods allow us to explicitly set the `this` value (context) for a function, enabling more control over its execution.

1. `call()`

- **Description:** Invokes a function immediately, allowing you to set the value of `this` and pass arguments one by one.
- **Syntax:** `func.call(thisArg, arg1, arg2, ...)`

Example:

```
const person = {
  name: "John",
};

function introduce(age, city) {
  console.log(`My name is ${this.name}, I am ${age} years old, and I live in
${city}.`);
}

introduce.call(person, 25, "New York");
// Output: My name is John, I am 25 years old, and I live in New York.
```

Real-Life Use Case:

When you have a method in one object and want to use it in the context of another object:

```
const car = { brand: "Tesla" };
const bike = { brand: "Yamaha" };

function showBrand() {
  console.log(`This is a ${this.brand}`);
}

showBrand.call(car); // Output: This is a Tesla
showBrand.call(bike); // Output: This is a Yamaha
```

2. `apply()`

- **Description:** Similar to `call()`, but arguments are passed as an array.
- **Syntax:** `func.apply(thisArg, [argsArray])`

Example:

```
const person = {
  name: "Alice",
};

function introduce(age, city) {
  console.log(`My name is ${this.name}, I am ${age} years old, and I live in
${city}.`);
}

introduce.apply(person, [30, "London"]);
// Output: My name is Alice, I am 30 years old, and I live in London.
```

Real-Life Use Case:

When arguments are already in an array format, such as in mathematical calculations:

```
const numbers = [3, 6, 1, 9];
const max = Math.max.apply(null, numbers); // Uses apply to pass an array
console.log(max); // Output: 9
```

3. bind()

- **Description:** Returns a new function with a specific `this` value and optional arguments. **Does not invoke the function immediately.**
- **Syntax:** `func.bind(thisArg, arg1, arg2, ...)`

Example:

```
const person = {
  name: "Emma",
};

function greet(greeting, punctuation) {
  console.log(`${greeting}, my name is ${this.name}${punctuation}`);
}

const boundGreet = greet.bind(person, "Hello");
boundGreet("!"); // Output: Hello, my name is Emma!
```

Real-Life Use Case:

When you need a function with a fixed context for later execution (e.g., event handlers):

```
const button = document.querySelector("button");
const user = {
  name: "Sophia",
  sayHello() {
    console.log(`Hello, ${this.name}!`);
  },
};

// Fix the context of `this` for the event listener
button.addEventListener("click", user.sayHello.bind(user));
```

Key Differences and Use Cases:

Feature	`call()`	`apply()`	`bind()`
Execution	Invokes the function immediately	Invokes the function immediately	Returns a new function (not executed)
Arguments	Passed one by one	Passed as an array	Can partially apply arguments

`this` inside arrow function

- Arrow functions don't have `this` binded with its own. It will inherit `this` from its lexical context of where it is defined.

```
const obj = {
    a : 5,
    func : () => {
        console.log(this);
    }
};

obj.func(); // prints globalObject
```

`this` in DOM

When the function is passed as an callback for eventlisteners, the `this` will refer the element in which the function was attached.

```
document.querySelector("button").addEventListener("click", function () {
  console.log(this.textContent); // Logs the text of the clicked button
});
```

`this` in Constructors

Refer Namaste JavaScript Season I (/FUiOImzMTUWxqbVHCxXDAg) to deep dive into this.

When the function called with `new`, it will become a constructor function call. With that, a new object will be created and binded with that constructor function. Eventhough, we didn't return anything, the created object will be returned. `this` inside such constructor function refers to that newly created object.

One can also understand this `this` with this (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/this)