ASYNCHRONOUS JAVASCRIPT "PROMISES"

ARIHARASUDHAN

THIS BOOK

This book is written (typed) by Ari, who hails from the South and has keen interests in Computer Science, Biology, and Tamil Literature. Occasionally, he updates his website, where you can reach out to him.

https://arihara-sudhan.github.io

Blocking Operations

An Operation is Blocking, if another operation has to wait until it completes. The Operations, such as reading from or writing to a drive are blocking in nature.

The Promise of Promises

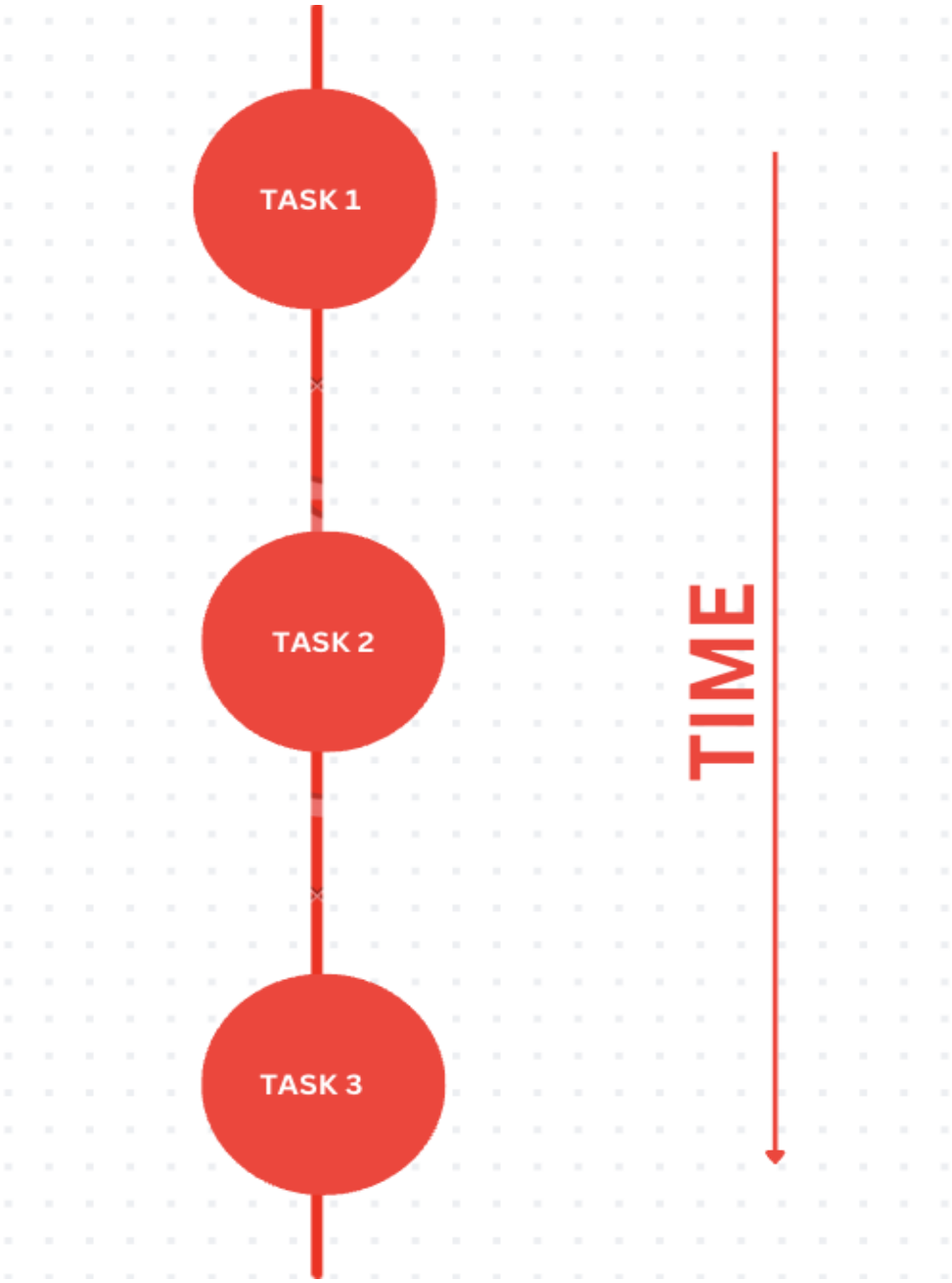
Promises promise a nonblocking strategy to perform I/O operations and such intensive operations. Promise is an object or a function (a function is an object in JavaScript). A Promise is a proxy for a value not necessarily known when the promise is created.

Asynchronous Model

Models are templates upon which the logics are fabricated within a compiler or interpreter. The three major models are,

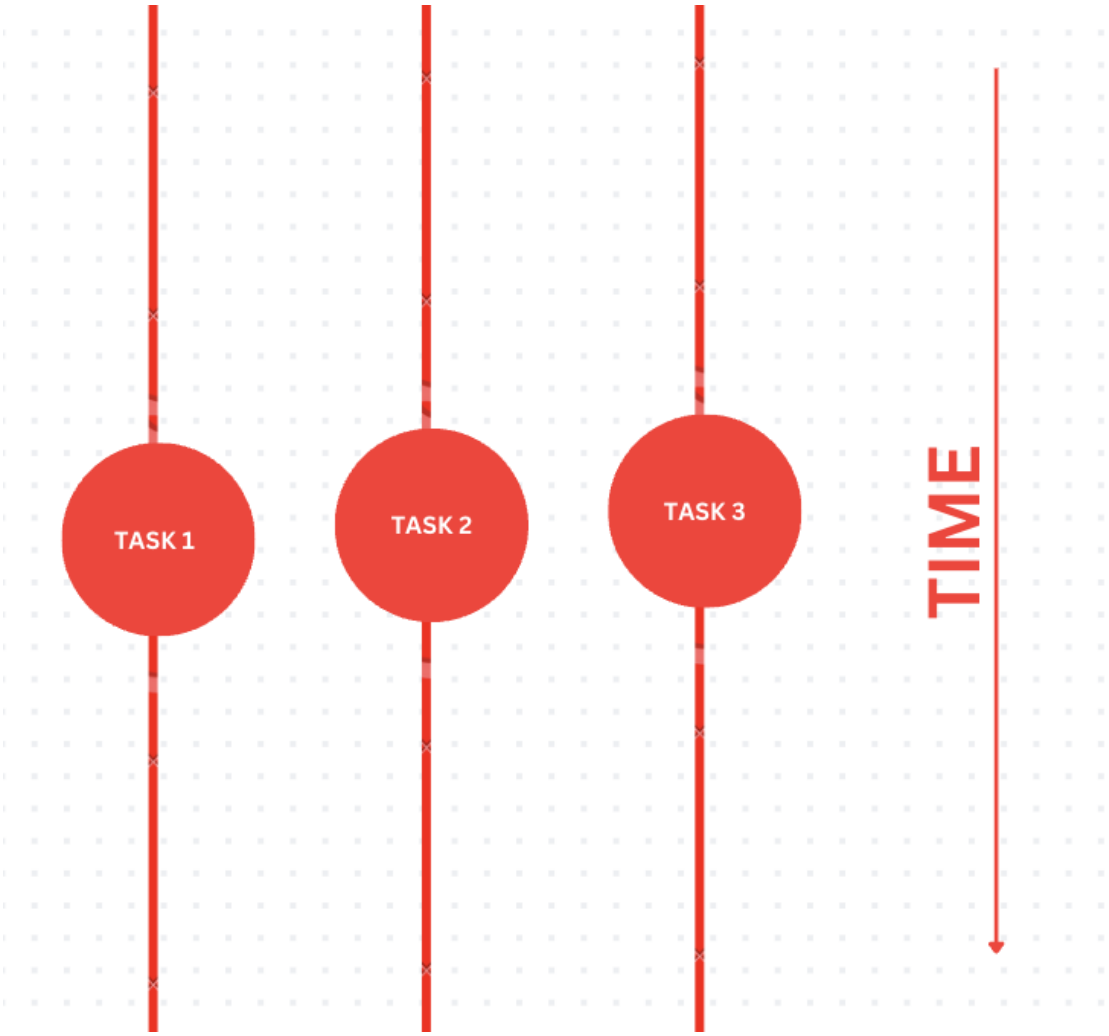
@ Single Threaded Synchronous

First task in the queue gets priority. One task follows the other.



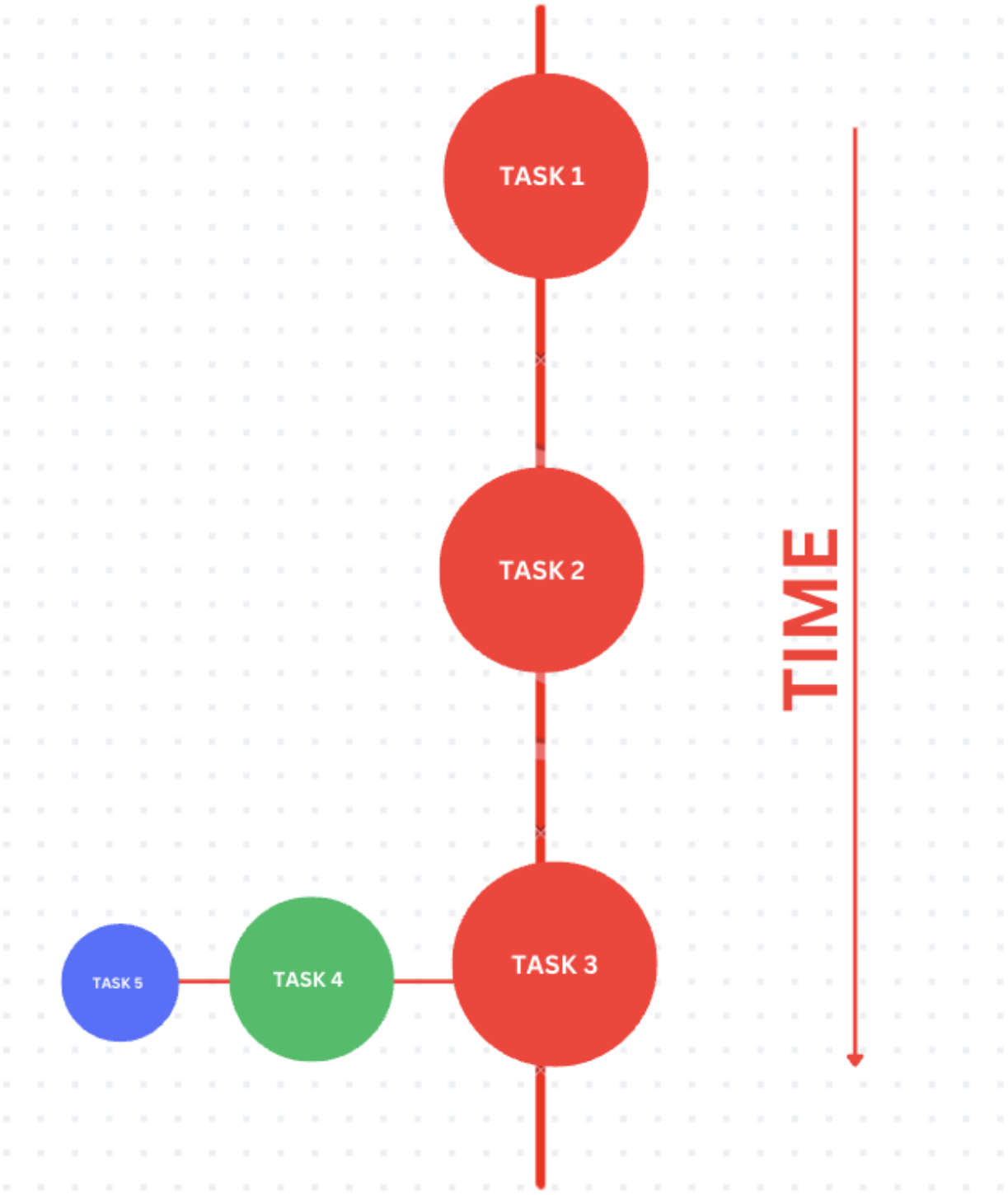
@ Multi Threaded Synchronous

Each tasks requires a separate thread in Multithreaded Synchronous Model. Multithreading is a way to introduce parallelism in our program. If there can be parallel parts which do not depend on result from another part in our program, we can make use of Multithreading.



@ Asynchronous

Tasks are interleaved with one another in a single thread.



In a threaded system, the operating system decides which thread (a smaller part of a program) runs and when it should stop to let another thread run. The programmer doesn't control this; the base program or operating system handles it. A Simple Example in Java for this:

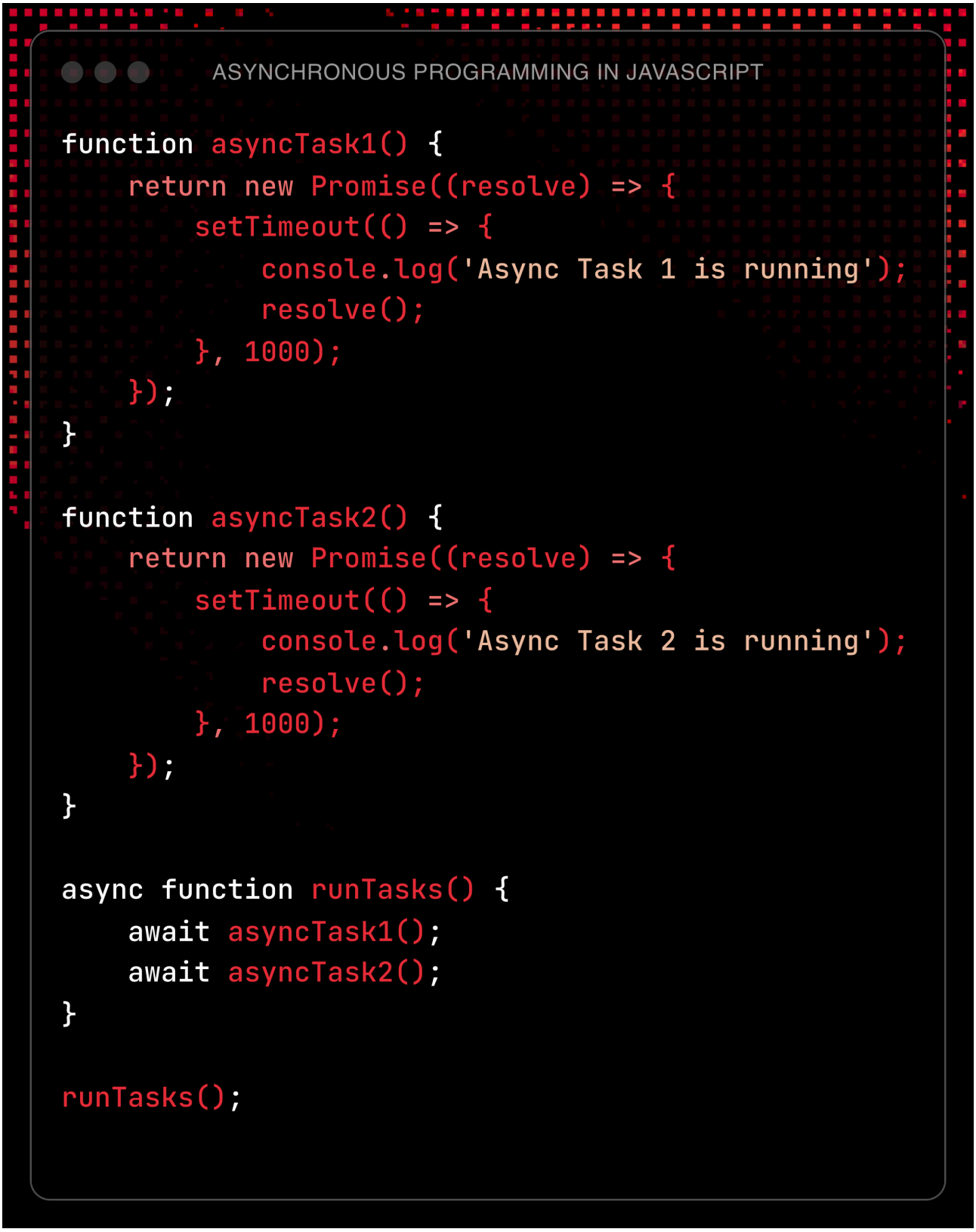
MULTITHREADING IN JAVA

```java
public class ThreadExample {
    public static void main(String[] args) {
        Thread thread1 = new Thread(new Task("Thread 1"));
        Thread thread2 = new Thread(new Task("Thread 2"));

        thread1.start();
        thread2.start();
    }
}

class Task implements Runnable {
    private String threadName;
    public Task(String threadName) {
        this.threadName = threadName;
    }
    @Override
    public void run() {
        for (int i = 0; i < 5; i++) {
            System.out.println(threadName + " is running");
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

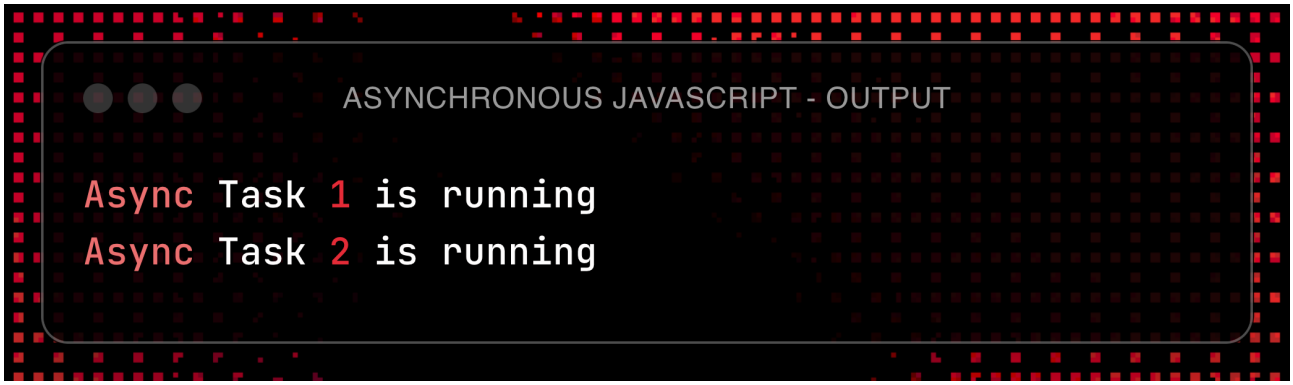
The Output would be like the following:

Here, Sometimes, Thread 2 or Thread 1 is first to be executed... It is completely on the hands (logics?) of The Operating System. In an asynchronous system, the programmer such as Ari has full control over which part of the program runs and when it stops.

A thread won't stop or change its state unless the programmer explicitly tells it to do so.



Here, we insist the task1 to be executed first using async-await. So, the output is definitely,



If one task uses the output of the other, the dependent task must be engineered to wait! The soul of an asynchronous system that can outperform synchronous systems almost dramatically is when the tasks are forced to wait or are blocked.

Non-Blocking Operations

When an asynchronous program encounters a task that will normally get blocked in a synchronous program, it will instead execute some other tasks that can still make progress. Because of this, asynchronous programs are also called non-blocking program.

JavaScript achieves Async

JavaScript uses some certain stuffs to achieve Asynchronous Programming. No suspense! Those are, A callback function, Event Listener, Publisher/Subscriber and A Promise Object.

CallBack

A Callback is a function passed to another function. That another function is, Higher Order Function.

CALLBACK PATTERN

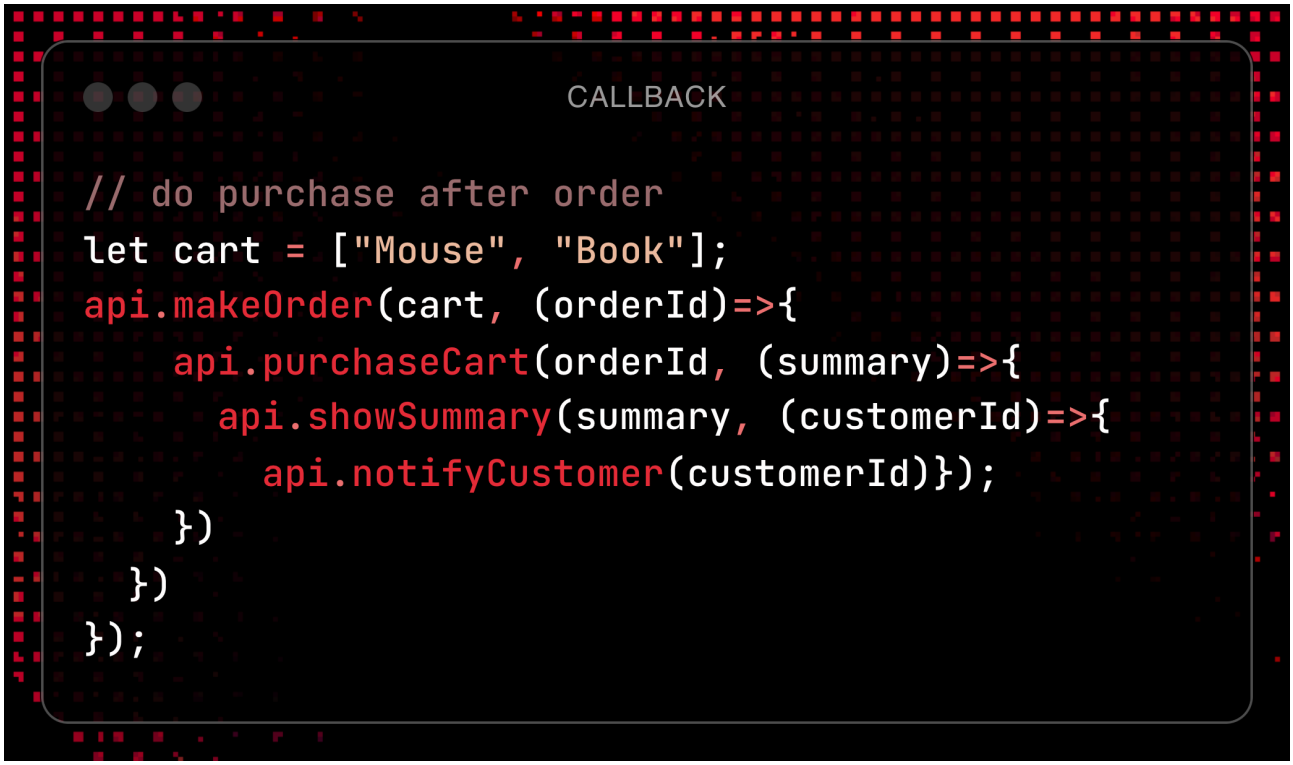
```
function fetchData(callback) {
  // Simulating an asynchronous operation
  setTimeout(() => {
    const data = 'Ente Tenkasi Tamizh Paingili!';
    callback(data);
  }, 1000);
}

//callback
function processData(data) {
  console.log('Data received:', data);
}

fetchData(processData); //Data received: Ente Tenkasi Tamizh Paingili!
```

The Higher Order Function can execute the callback at anytime. It is “called-back” and not executed immediately! For example, Once we do a makeOrder, purchaseCart is called back.

```
// do purchase after order
let cart = ["Mouse", "Book"];
api.makeOrder(cart, (orderId)=>{
    api.purchaseCart(orderId);
})
```

CallBack Hell

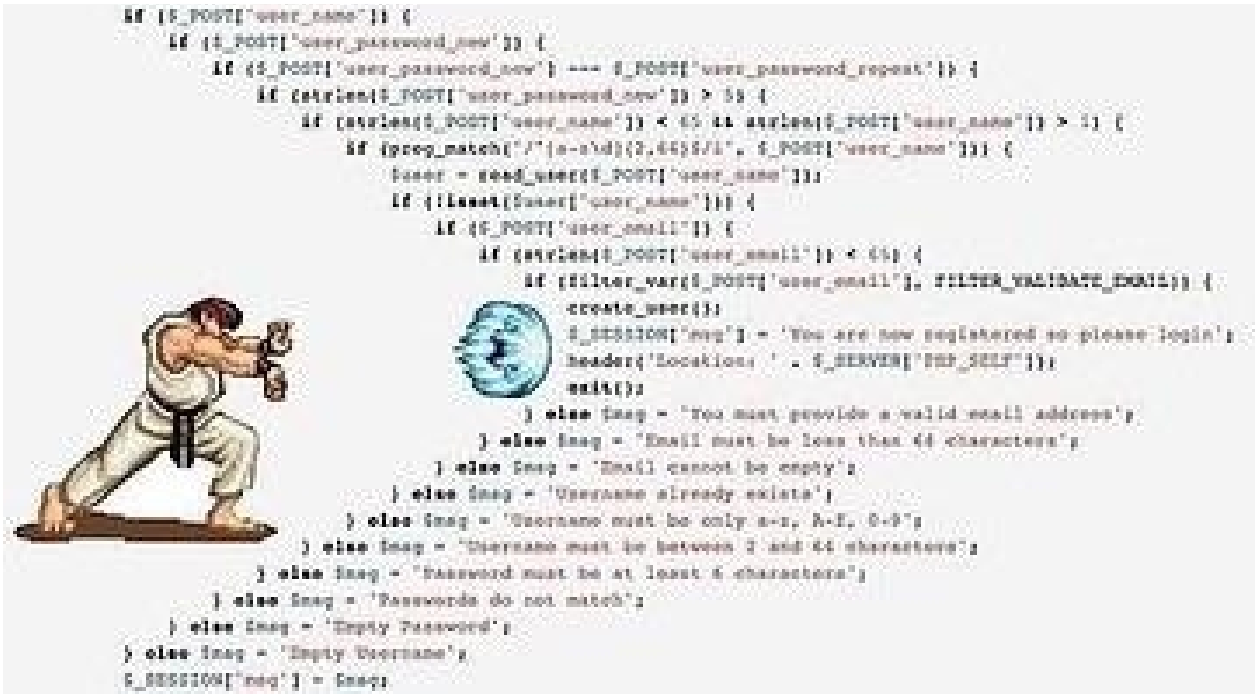
If we want to perform some more actions after purchaseCart, say showSummary, notifyCustomer and so on. Code starts to grow horizontally.



This is the so called, Callback Hell! If the callback is not handled properly, the problem of Callback Hell will come. The engineers will ask us, “what the hell is this?”. We should answer, “Callback Hell”.

Callback Hell is the scenario of working with nesting of callbacks which is messy and most of the time, out of control.

The callback hell is also called as, “The Pyramid of Doom”.



SOLUTION 1 : Keep It Simple Superman!

Break up the functionality to resolve such hells! Modularity should be there! Proper Naming Conventions should be followed.

Promise is a technique to avoid this Pyramid of Doom. It's better practice to not have large, complicated functions. Choosing function names for each of these levels will also help.

SOLUTION 2: To Break Up

SOLUTION TO CALLBACK HELL

```
doSomething() {
  // do something now
}
CC() {
  ccccc() {
    ddddd() {
      doSomething()
    }
  }
}
main() {
  aaaaa() {
    bbbbb() {
      CC()
    }
  }
}
```

Inversion of Control

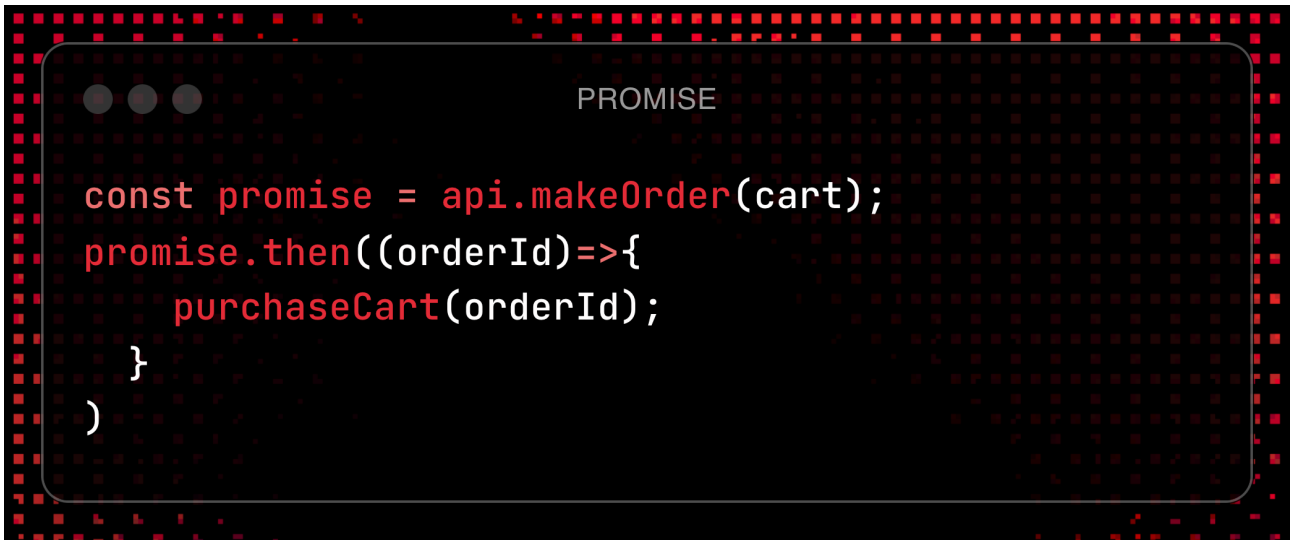
We blindly rely on the higher order function. It may call the callback or not. What if it calls the callback thrice? We don't control it! The control is completely in the hands of the higher order function which got our callback!

```
// do purchase after order
let cart = ["Mouse", "Book"];
api.makeOrder(cart, (orderId)=>{
    api.purchaseCart(orderId);
})
```

Are we sure about makeOrder() calling purchaseCart(), once done? This is called, Inversion of Control.

PROMISES TO AVOID IOC

To avoid the inversion of control, we can use Promise in our program. By this, the control will be in our hand. We write the fate of the callback!



Now, we did Inversion of Inversion of Control... Once the asynchronous operation is completed, then method will be executed. It actually takes our callback and calls it back. I smell an Inversion of Control here... Will the then() call the callback as expected? Please don't suspect here! It is the PROMISE by Promise! It will be called for sure!!!!

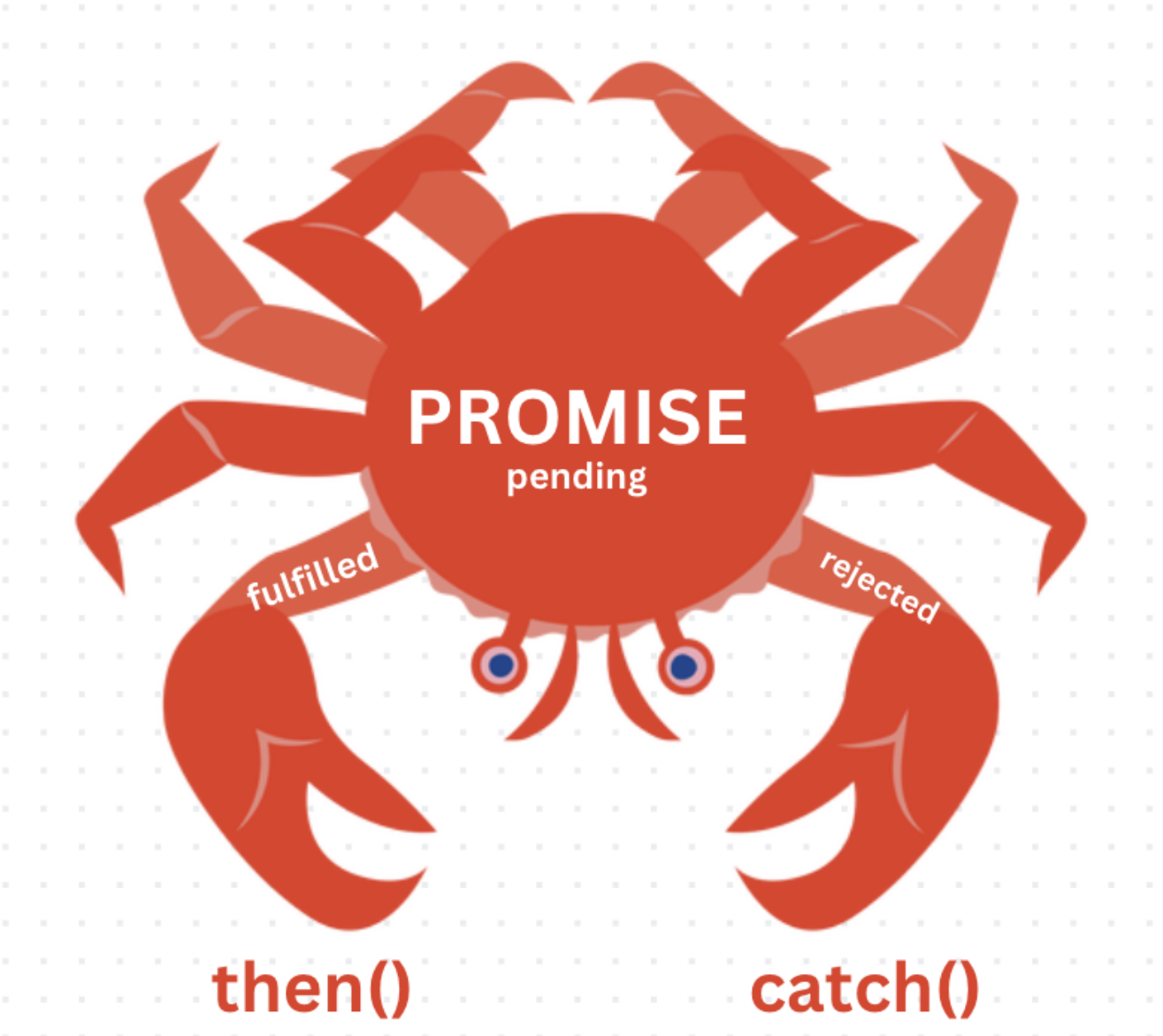
Let's see a promise object spit out by fetch call.

Promise can be only in three states.

Pending – FulFilled – Rejected

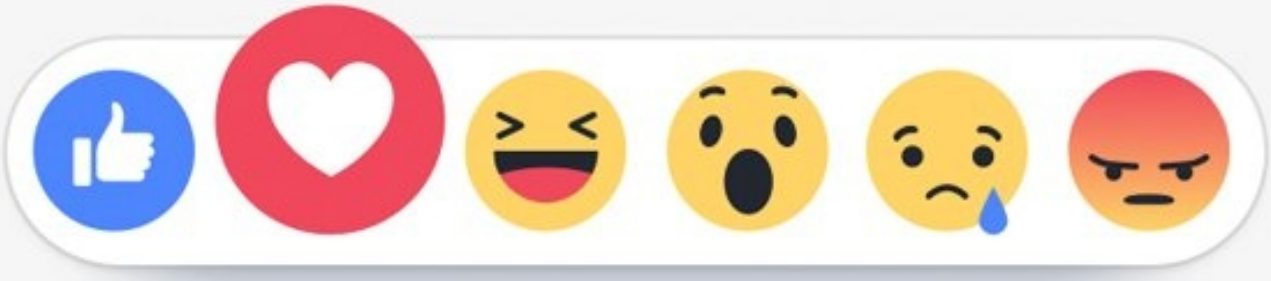
When created, the promise is in pending state. then() is executed only when the state is fulfilled. catch() is executed only when the state is rejected. Promise is widely defined as a placeholder that gets it's value later (after the completion of asynchronous function).

In other words, Promise is an object representing eventual completion or failure of an asynchronous operation.



PROMISES TO AVOID CBACK HELL

Can we avoid Callback Hell using Promises? YES! That's why we have this title here... How can we? Look at the following snippet, which has a pyramid of doom. We get a profile, then the posts, then the reactions, then we just show the heart reactions.



CALLBACK HELL

```
api.getProfile((profileId) => {
  api.getPosts((posts) => {
    api.getReactions((reactions) => {
      api.showHearts(reactions);
    });
  });
});
```

To avoid this, we can use Promise Chaining. Let's refactor with Promises.

CALLBACK HELL

```
api.getProfile()
  .then((profileId) => {
    return api.getPosts(profileId);
  })
  .then((posts) => {
    return api.getReactions(posts);
  })
  .then((reactions) => {
    api.showHearts(reactions);
  })
  .catch((error) => {
    console.error('An error occurred:', error);
  });
```

Don't forget to return a Promise from a then() so that we can access then of it. Thus, we avoid the horizontal growth of code.

We can also create our own promise as shown below.

We can just make use of the Promise() constructor to create a promise object. The constructor has two params

`resolve:` A function that is called when the asynchronous operation completes successfully. It takes a

single argument, which is the result of the operation.

reject: A function that is called when the asynchronous operation fails. It takes a single argument, which is the reason for the failure (usually an error message or an error object).

The part where we define our promise is called, Publisher. The part where we use then, catch is Subscriber. We can also create promises for chaining just the same... When we want to call getName after the successful completion of getProfile, we can write a Promise for getName also as show in the following code snippet.

CREATING CHAIN

CREATING PROMISES FOR CHAINING

```
getProfile("ARI")
    .then((profile)=>{
        return getName(profile);
    })
    .then((name)=>{
        console.log(name);
    })
    .catch((err)=>{
        console.log(err);
    })

function getProfile(name) {
    return new Promise((resolve, reject)=>{
        if(!name){
            reject("NAME NOT VALID");
        }
        setTimeout(()=>{
            console.log("RESOLVED");
            resolve({name: "ARIHARASUDHAN", age: 21})}, 2000);
    })
}

function getName(profile) {
    return new Promise((resolve, reject)=> {
        if(profile.name){
            resolve(profile.name)
        } else {
            reject("UNNAMED")
        }
    })
}
```

If we want to check error for each promise, we can give them in respective order.

ERROR HANDLING FOR EACH PROMISES

```
getProfile("ARI")
    .then((profile)=>{
        return getName(profile);
    })
    .catch((err)=>{
        consol.log("ERROR IN GETTING PROFILE")
    })
    .then((name)=>{
        console.log(name);
    })
    .catch((err)=>{
        console.log("ERROR IN GETTING NAME");
    })

function getProfile(name) {
    return new Promise((resolve, reject)=>{
        if(!name){
            reject("NAME NOT VALID");
        }
        setTimeout(()=>{
            console.log("RESOLVED");
            resolve({name: "ARIHARASUDHAN", age: 21})}, 2000);
    })
}

function getName(profile) {
    return new Promise((resolve, reject)=> {
        if(profile.name){
            resolve(profile.name)
        } else {
            reject("UNNAMED")
        }
    })
}
```

PROMISES API

✳ Promise.all

To avoid the inversion of control, we can use Promise in our program. By

PROMISE ALL

```
const promise1 = Promise.resolve("DONE");
const promise2 = 42;
const promise3 = new Promise((resolve, reject) => setTimeout(resolve, 100, 'foo'));

Promise.all([promise1, promise2, promise3]).then((values) => {
      console.log(values);
    }).catch((error) => {
      console.log('Error:', error);
});
```

Output : ['DONE', 42, 'foo']

By Promise.all, we can check multiple promises. It is like, all or none. If any one promise is rejected, Promise.all will reject! If any one promise rejects, catch will be executed.

✳ Promise.allSettled

Settled means, all the promises are done! Whether fulfilled or rejected, all we need is the result of each promises. (By this, we can check which promise rejected).

PROMISE ALL SETTLED

```
const promise1 = Promise.reject("ERROR");
const promise2 = 42;
const promise3 = new Promise((resolve, reject) => setTimeout(resolve, 100, 'foo'));

Promise.allSettled([promise1, promise2, promise3]).then((values) => {
      console.log(values);
    }).catch((error) => {
      console.log('Error:', error);
});
```

Output will be like,

```
[
  {status: 'rejected',reason: 'ERROR'},
  { status: 'fulfilled', value: 42 },
  { status: 'fulfilled', value: 'foo' }
]
```

✳ Promise.race

The promise which fulfils or rejects first will be taken (Only race where we also consider the first loser)

PROMISE RACE

```
const promise1 = new Promise((resolve) => setTimeout(resolve, 500, 'first'));
const promise2 = new Promise((resolve,reject) => setTimeout(reject, 100, 'second'));

Promise.race([promise1, promise2]).then((value) => {
    console.log(value);
}).catch((err)=>{
    console.log("ERROR in "+err);
});
```

Output will be: ERROR in second

✳ Promise.any

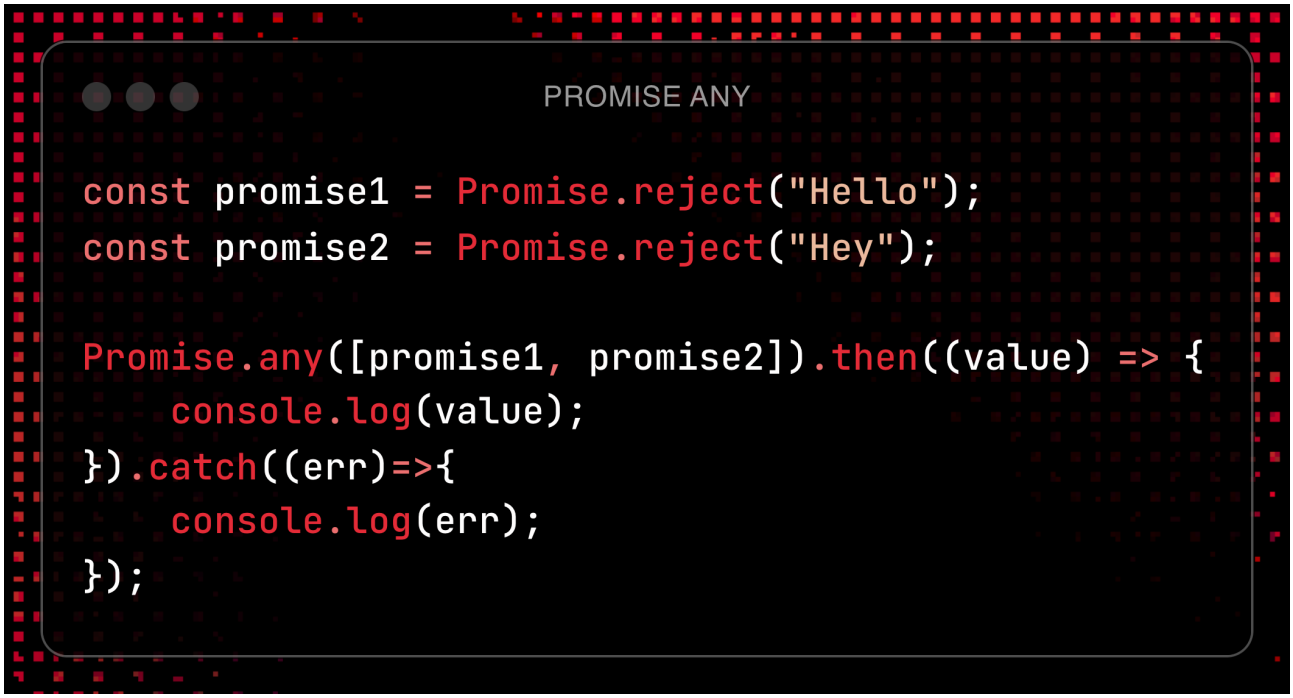
Promise.any is just like a race... but, a winner seeking race! The first promise fulfilled is taken..

PROMISE ANY

```
const promise1 = Promise.reject("Hello");
const promise2 = Promise.resolve("Hey");

Promise.any([promise1, promise2]).then((value) => {
    console.log(value);
}).catch((err)=>{
    console.log(err);
})
```

OP: HEY

If all the promises reject, then a list of error stuffs will be there.



It is an aggregate error.

ERROR!
[AggregateError: All promises were rejected] {
[errors]: ['Hello', 'Hey']
}

NOTE:

〉 These all methods are applied on list of promises

〉 We can directly reject/resolve using Promise.resolve(), Promise.reject()

〉 Along with then and catch, we can also use finally which will be always executed irrespective of what happens above.

PROMISE ANY

```
const promise1 = Promise.reject("Hello");
const promise2 = Promise.resolve("Hey");

Promise.any([promise1, promise2]).then((value) => {
    console.log(value);
}).catch((err)=>{
    console.log(err);
}).finally(()=>{
    console.log("I AM GARBAGE COLLECTOR");
});
```

Output:

Hey

I AM GARBAGE COLLECTOR

Async - Await

The keywords, async and await are used to handle asynchronous operations more efficiently and readably than traditional callback-based or promise-based approaches.

Async: The async keyword is used to declare an asynchronous function. This means the function will always return a promise, even if you don't explicitly return one. If the function returns a value, the promise will be resolved with that value. If the function throws an error, the promise will be rejected with that error.

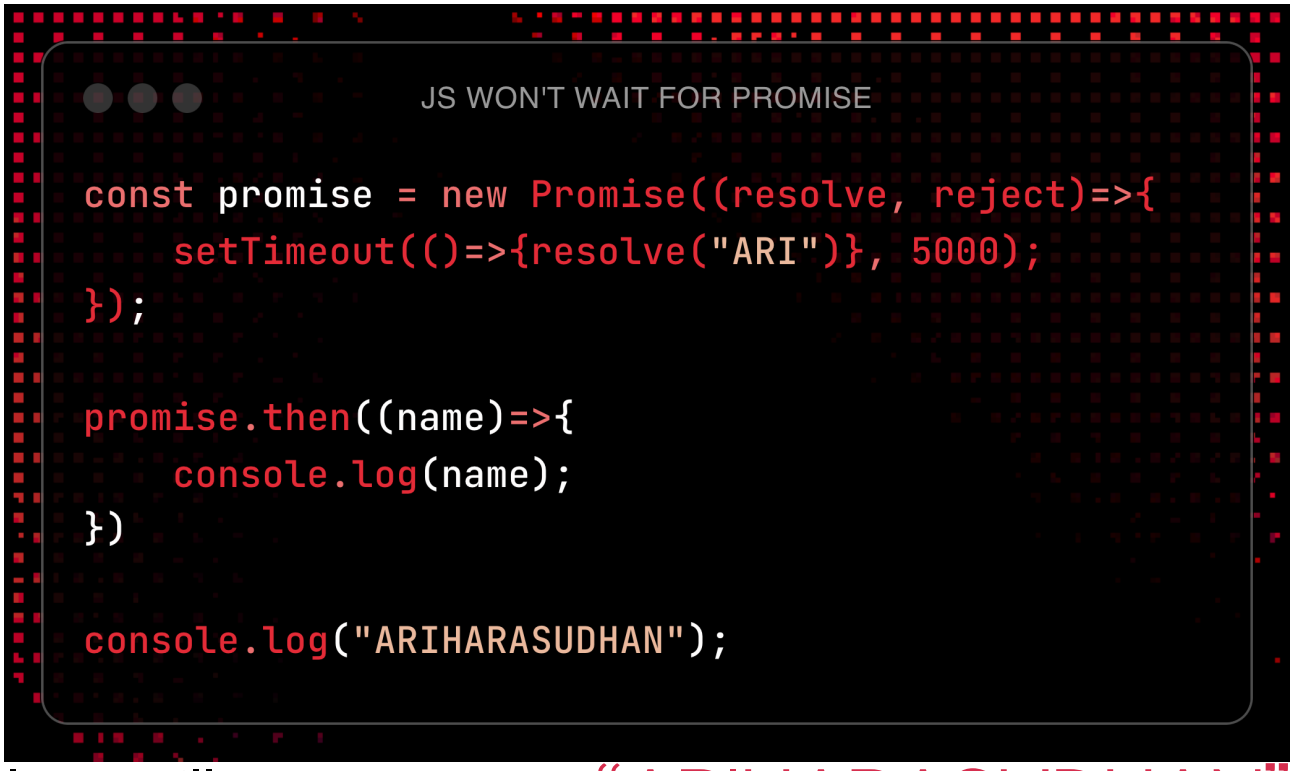
So, how can we take the value from the promise? We know how to do then-catch...

ASYNC

```
async function myName(){
    return "Ari";
}

const promise = myName();
promise.then((name)=>{
    console.log(name) // Ari
}).catch((err)=>{
    console.log(err)
})
```

Await: The await keyword can only be used inside an async function. It makes JavaScript wait until the promise is resolved and returns the result. It pauses the execution of the async function, allowing other operations to run until the promise settles.

JavaScript Engine won't wait for a promise to be completed. It will execute the next statements.



It will output, "ARIHARASUDHAN" followed by "ARI". But, if we want to block for a while, yes! We can do that... We have to use the await keyword in front of the promise. Code that uses async and await is easier to read and understand than nested callbacks or chained promises.

ASYNC AWAIT

```
const promise = new Promise((resolve, reject)=>{
    setTimeout(()=>{resolve("ARI")}, 5000);
});
async function showName(){
    const name = await promise;
    console.log(name);
    console.log("HELLO");
}
showName(); // "ARI" followed by "HELLO"
```

Using try...catch with async/await provides a clean way to handle errors.

ASYNC AWAIT - ERROR HANDLING

```
const promise = new Promise((resolve, reject)=>{
    setTimeout(()=>{reject("ERROR (SUMMA)")}, 5000);
});
async function showName(){
    try{
        const name = await promise;
        console.log(name);
        console.log("HELLO");
    } catch(e){
        console.log(e);
    }
}
showName(); // ERROR (SUMMA)
```

There will be situations where we need to wait for the data from the previous lines of code. In such cases, using `await` will be very helpful.

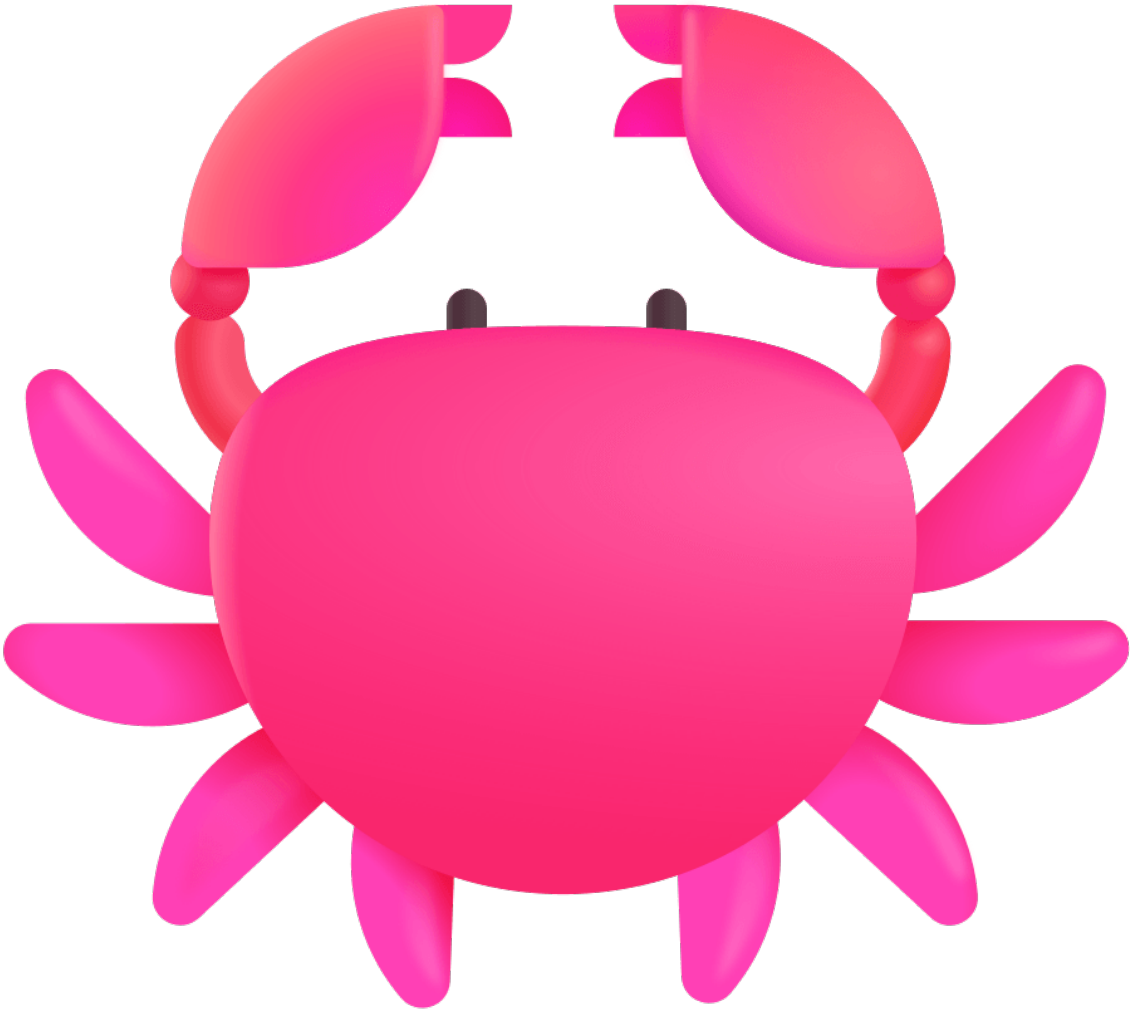
WITHOUT BLOCKING

```
function fetchData() {
    let data;
    setTimeout(() => {
        data = "Data fetched!";
    }, 2000);
    return data;
}
const result = fetchData();
console.log(result); // undefined
```

We have to wait until we get data.

WITH BLOCKING

```
function fetchData() {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            resolve("Data fetched!");
        }, 2000);
    });
}
async function getData() {
    const result = await fetchData();
    console.log(result);
}
getData(); // Data fetched!
```



https://arihara-sudhan.github.io