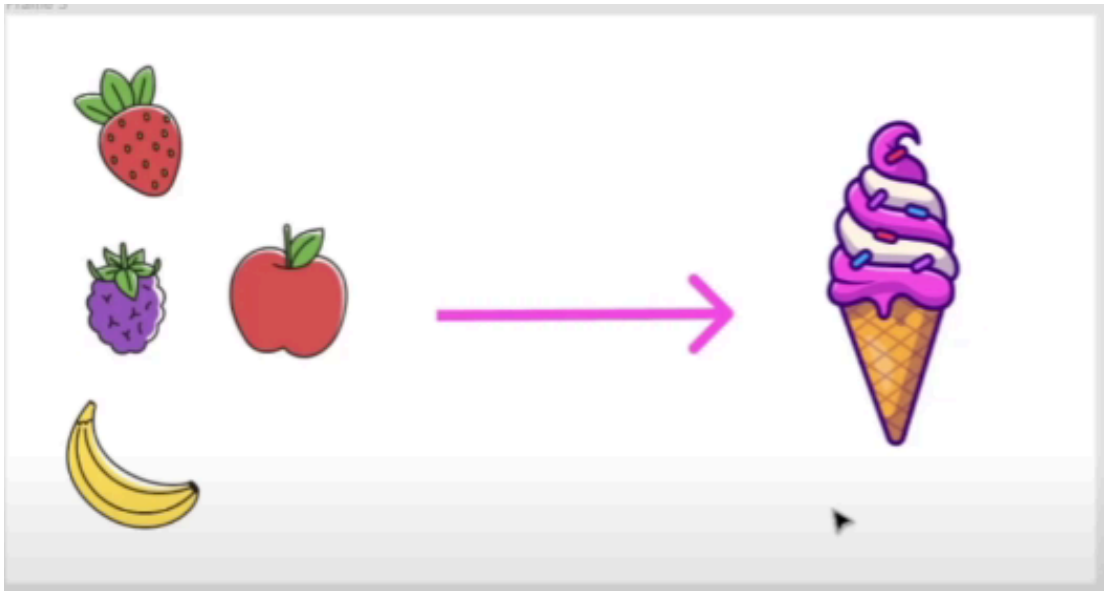
Asynchronous JavaScript (Async/Await, Promises, Callbacks)

Topics to Discuss

- Asynchronous Javascript?
- Synchronous Vs Asynchronous
- Callbacks
- Promises
- Async/Await
- Conclusion

Asynchronous Js

We will take an example of the making icecream to understand the concept



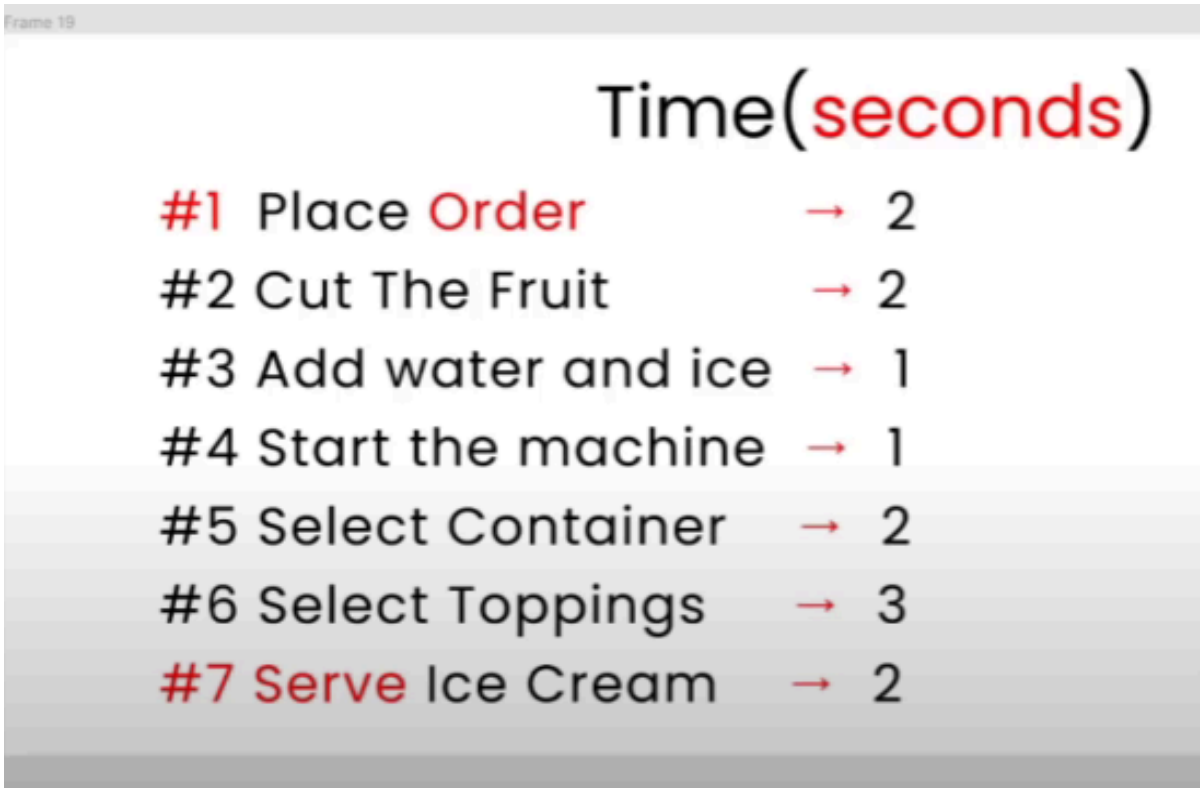
If you want to build projects efficiently, this concept is for you.

The theory of async JavaScript helps you break down big complex projects into smaller tasks.

Then you can use any of these three techniques – **callbacks, promises, or Async/await** – to run those small tasks in a way that you get the best results

Let's break down into smaller steps

Adding time function to understand the concept better



Synchronous Vs Asynchronous

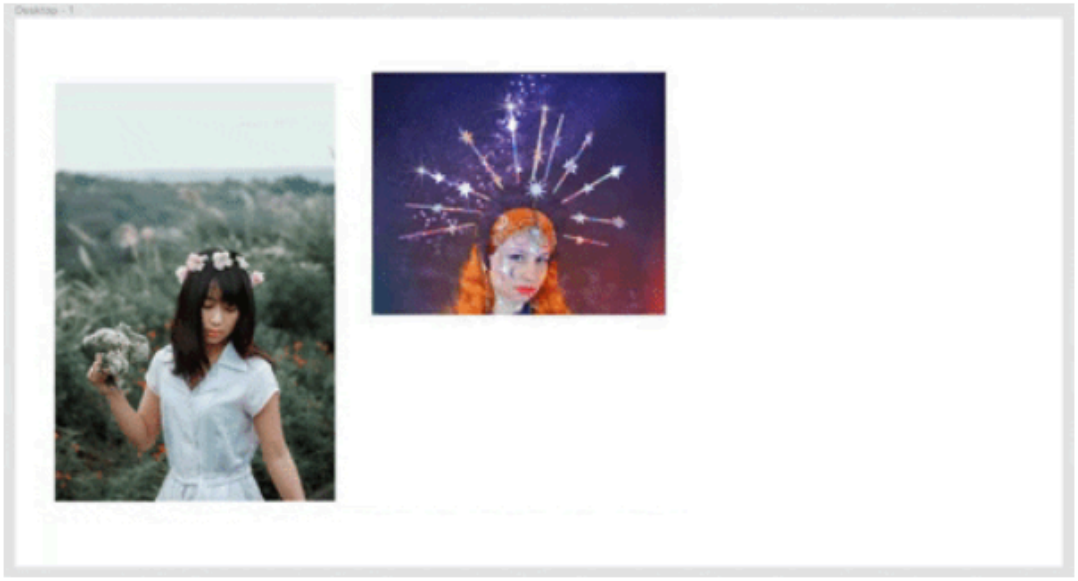
What is a Synchronous System?

Example of synchronous (one by one)

We have to count 10 toffees from a single hand, we will do it one by one by separating counted toffees from the rest, we can't count other toffees unless we have finished counting the previous toffees

Eg: Think of this as if you have just one hand to accomplish 10 tasks. So, you have to complete one task at a time.

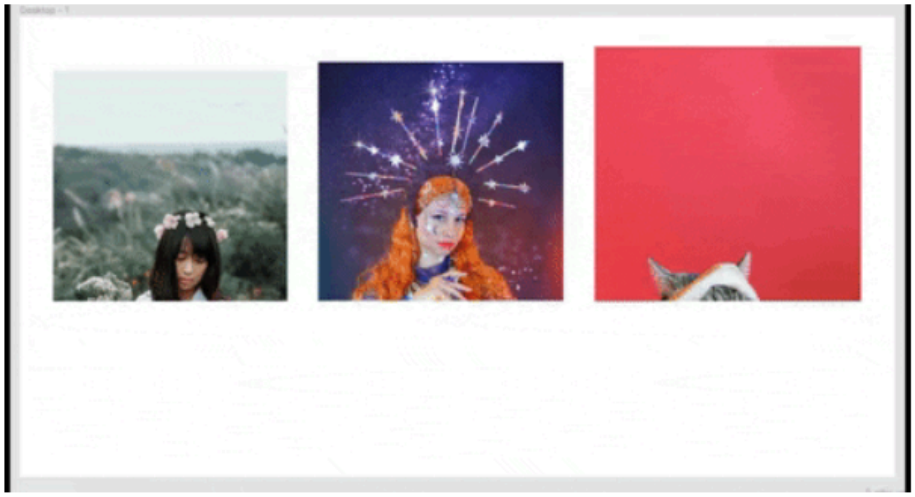
More examples



Here we can see that until the first image is loaded completely, the second image doesn't start loading.
Javascript is by default Synchronous(single-threaded). Think about it like one thread means one hand with which to do stuff

What is an Asynchronous System?

Asynchronous (no dependent)(they will be run independently)
In this system, tasks are completed independently
Eg

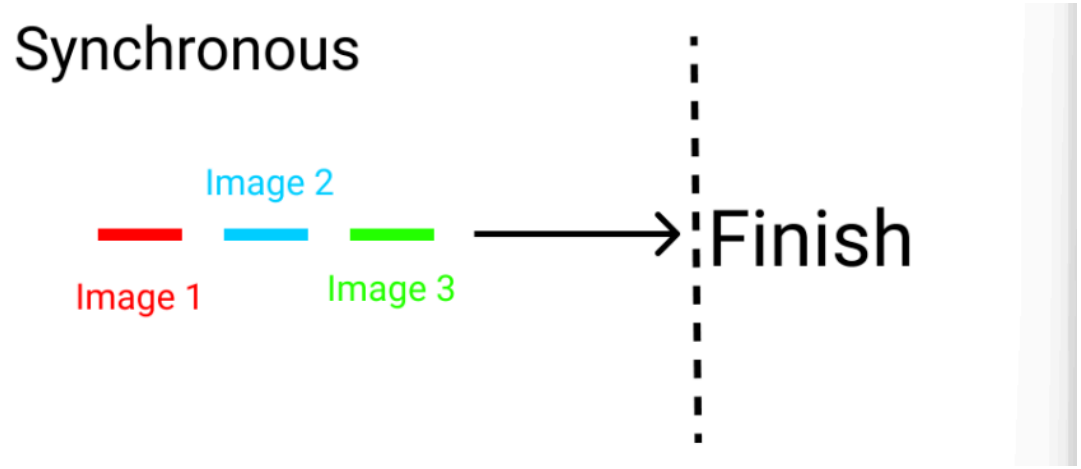


Here we can see that all three images load at the same time this is an example of asynchronous. All the images are loading at their own pace. None of them is waiting for any of the others

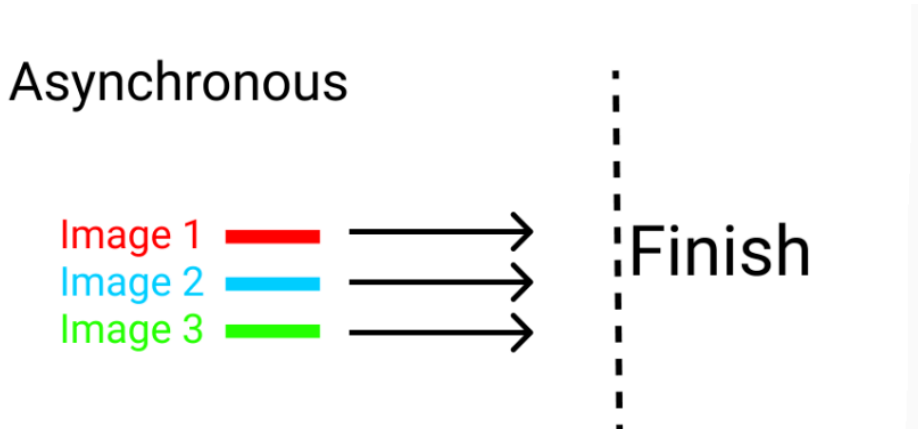
Summarize: Synchronous vs Asynchronous Js

Suppose there are three images of a marathon

- **Synchronous** system, three images are in the same lane. One can't overtake the other. The race is finished one by one. If image number 2 stops, the following image (1) also stops.



- **Asynchronous system**, the three images are in different lanes. They'll finish the race at their own pace. Nobody stops for anybody:



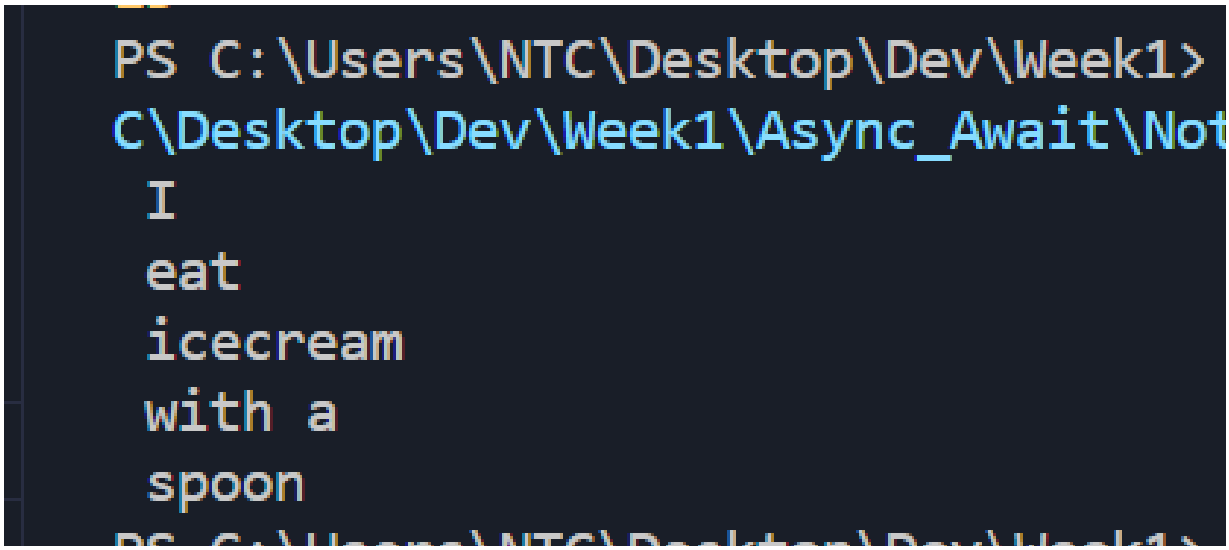
Synchronous and Asynchronous Code Examples

Synchronous Code Examples

Javascript runs from top to bottom, if anything gets stuck entire process is stuck.

```
// synchronous
console.log(" I  ")

console.log(" eat ")

console.log(" icecream ")

console.log(" with a ")

console.log(" spoon ")
```

Result:



Asynchronous Code Examples:

setTimeout() will be taken to another lane and after 4 sec it will be released to the main lane

```
// asynchronous
console.log(" I  ")

console.log(" eat ")

console.log(" with a ")

console.log(" spoon ")

setTimeout(()=>{
    // setTimeout() is an asynchronous function
    console.log(" icecream ")
    // prints ice cream after 4 seconds
},4000)

// Arrow function
// let abcded = (a,b,c,d) =>{console.log(a,b,c,d)};
```

Result:

I

eat

with a

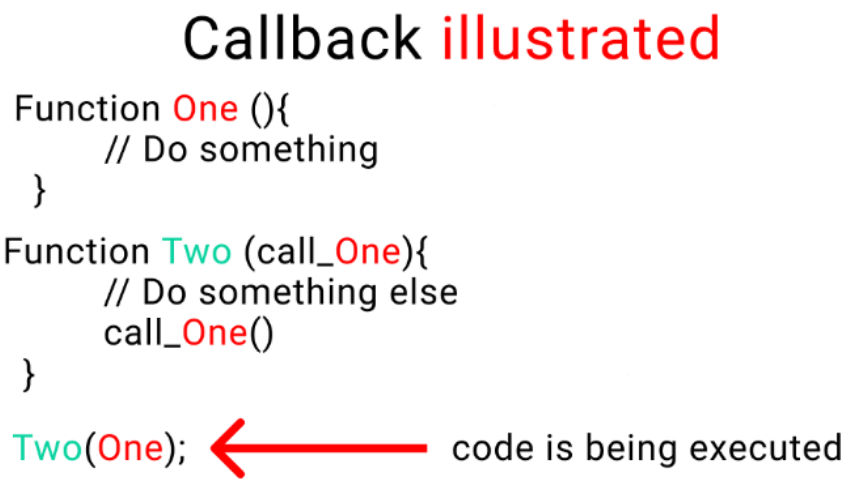
spoon

icecream

Callbacks

Calling a function inside another function is called a callback

Here's an illustration of a callback:



An example of a callback

Regular function

```
function one (){
    console.log(" Step one ");
}
function two (){
    console.log(" Step two ");
}
two();
one();
```

Step two

Step one

Eg of callback function:

```
function one (call_two){
    console.log(" Step one complete.please call step two ");
    call_two();
}
function two (){
    console.log(" Step two complete ");
}
one(two);
```

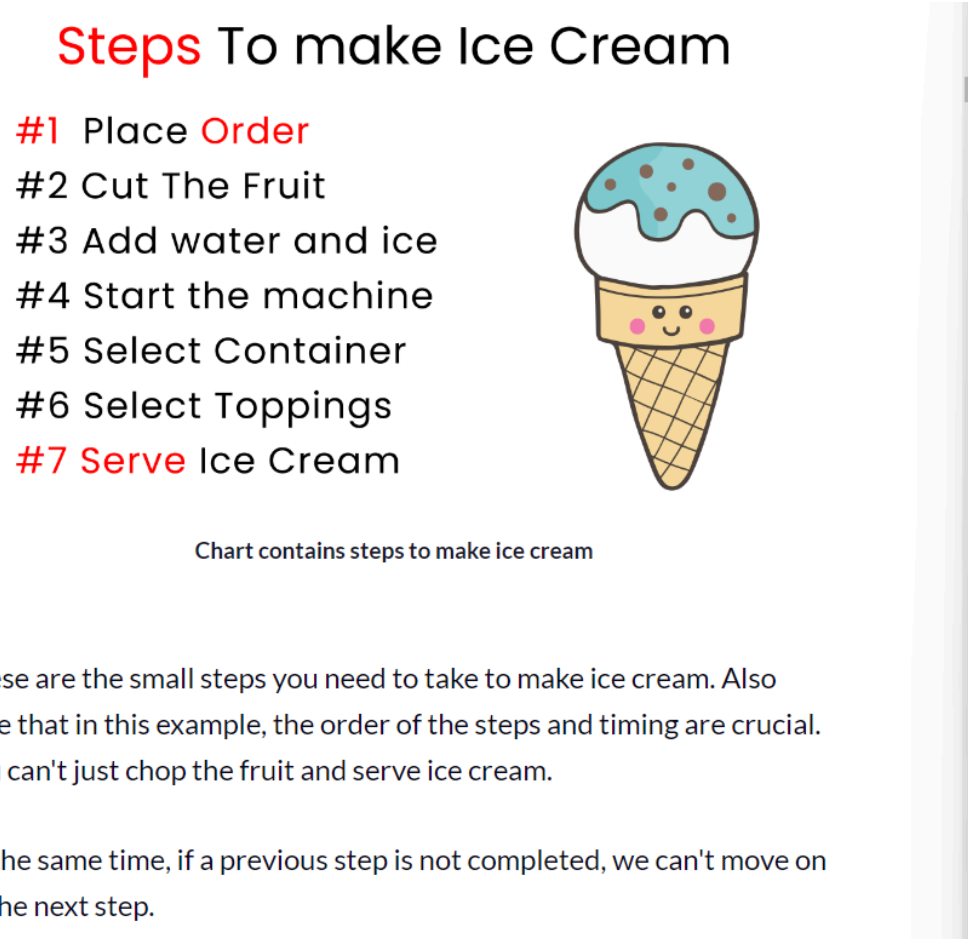
Output:

Step one is complete. please call step two

Step two complete

Why do we use callbacks?

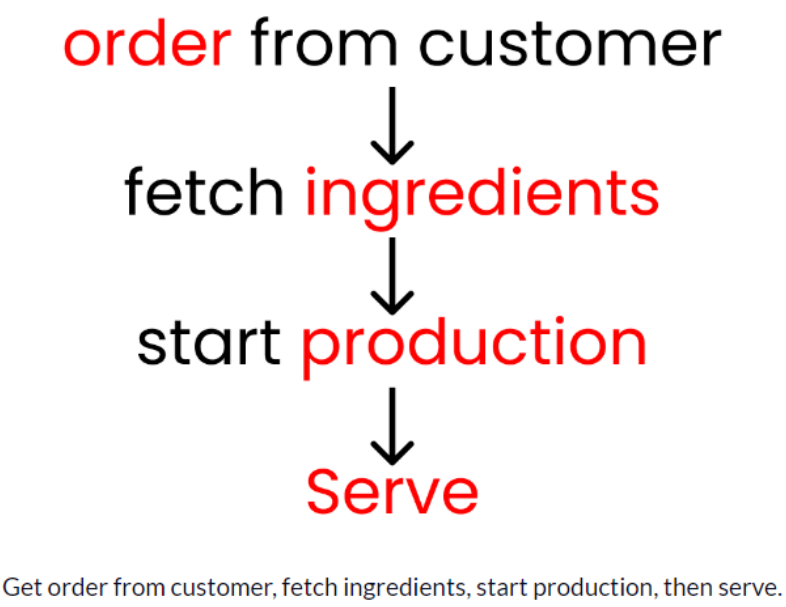
When doing a complex task, we break that task down into smaller steps. We use callbacks to help us establish a relationship between these steps according to time(optional) and order.



Eg: Of an ice cream shop business

We first need to know the relationship between customers and us.

This is how it all works: 👇



Get order from customer, fetch ingredients, start production, then serve.

We need to create two functions order and production.

If we don’t get the order we can’t start production, this is how we form a connection between two functions using CALLBACK

```
// forming connection between two functions using callback
let order = (call_production) => {
    console.log("Order placed, please call production");
    call_production();
}

let production = () => {
    console.log("Order received, starting production");
}
order(production);
//Pass the second function name as an argument
```

Output:

Order placed, please call production

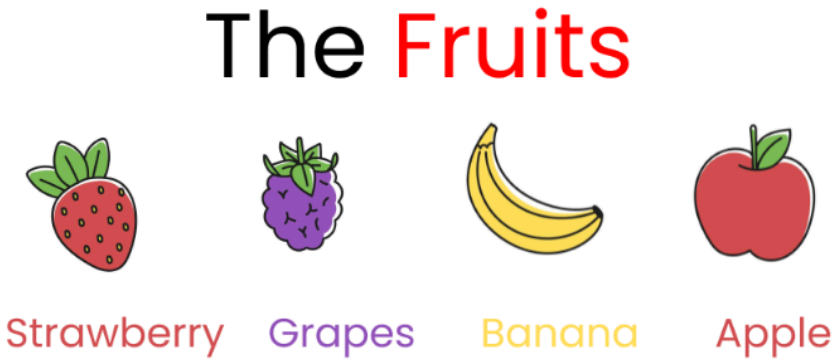
Order received, starting production

The shop will have two parts:

- The storeroom will have all the ingredients [Our **Backend**]
- We'll produce ice cream in our kitchen [The **Frontend**]

Let's store our data

We will store our ingredients inside an object.



```
let stocks  = {
    Fruits:["strawberry","grapes","banana","apple"],
}
console.log(stocks.Fruits[2])
```

We have other ingredients


```
let stocks  = {
    Fruits:["strawberry","grapes","banana","apple"],
    liquid:["water","ice"],
    holder:["cone","cup","stick"],
    toppings:["chocolate","peanuts"]
}
```

Here are our Steps, and the time each step will take to execute

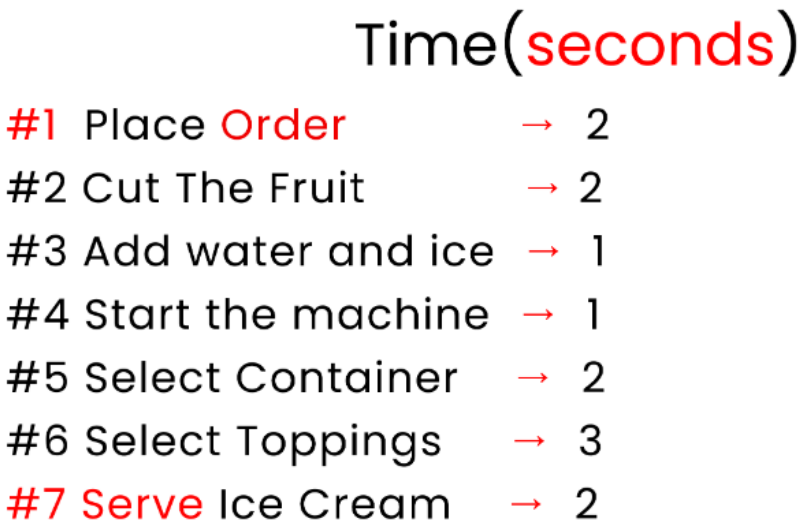
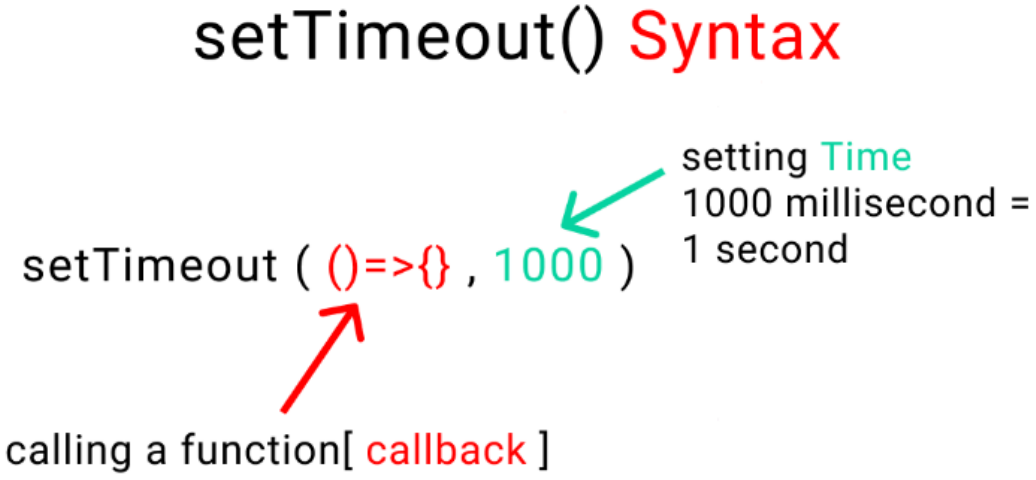


Chart contains steps to make ice cream

In this chart, we can see that step 1 which is to place the order, takes up 2 seconds. Then step 2 is to cut the fruit(2 seconds),....

We will use the setTimeout() to establish the timing



```
let stocks  = {
    Fruits:["strawberry","grapes","banana","apple"],
    liquid:["water","ice"],
    holder:["cone","cup","stick"],
    toppings:["chocolate","peanuts"]
}

// forming connection between two function using callback
let order = (Fruit_name,call_production) => {
    setTimeout(()=>{
        console.log(`${stocks.Fruits[Fruit_name]} was selected`)
        call_production();
        // if outside it would have started production before fruits were
selected
    },2000)

}

let production = () => {
    setTimeout(()=>{
        console.log("production has started")
        setTimeout(()=>{
            console.log("The fruit has been chopped")
            // We are going into callback hell
            setTimeout(()=>{
                console.log(`${stocks.liquid[0]} and ${stocks.liquid[1]}
was added`)

                setTimeout(()=>{
                    console.log("The machine was started")

                    setTimeout(()=>{
                        console.log(`icecream was placed on
${stocks.holder[0]}`)

                        setTimeout(()=>{
                            console.log(`${stocks.toppings[0]} was added
as toppings`)

                            setTimeout(()=>{
```

```
                                console.log("Serve icecream")
                            },2000)
                        },3000)
                    },2000)
                },1000)
            },1000)
        },2000)
    }, 0)
}
order(0,production);
// Javascript runs from top to bottom
```

```
let production = () =>{

  setTimeout(()=>{
    console.log("production has started")
    setTimeout(()=>{
      console.log("The fruit has been chopped")
      setTimeout(()=>{
        console.log(`${stocks.liquid[0]} and ${stocks.liquid[1]} Added`)
        setTimeout(()=>{
          console.log("start the machine")
          setTimeout(()=>{
            console.log(`Ice cream placed on ${stocks.holder[1]}`)
            setTimeout(()=>{
              console.log(`${stocks.toppings[0]} as toppings`)
              setTimeout(()=>{
                console.log("serve Ice cream")
              },2000)
            },3000)
          },2000)
        },1000)
      },1000)
    },2000)
  },0000)

};
```

Output:

strawberry was selected

production has started

The fruit has been chopped

water and ice was added

The machine was started

icecream was placed on cone

chocolate was added as toppings

Serve icecream

Callback hell

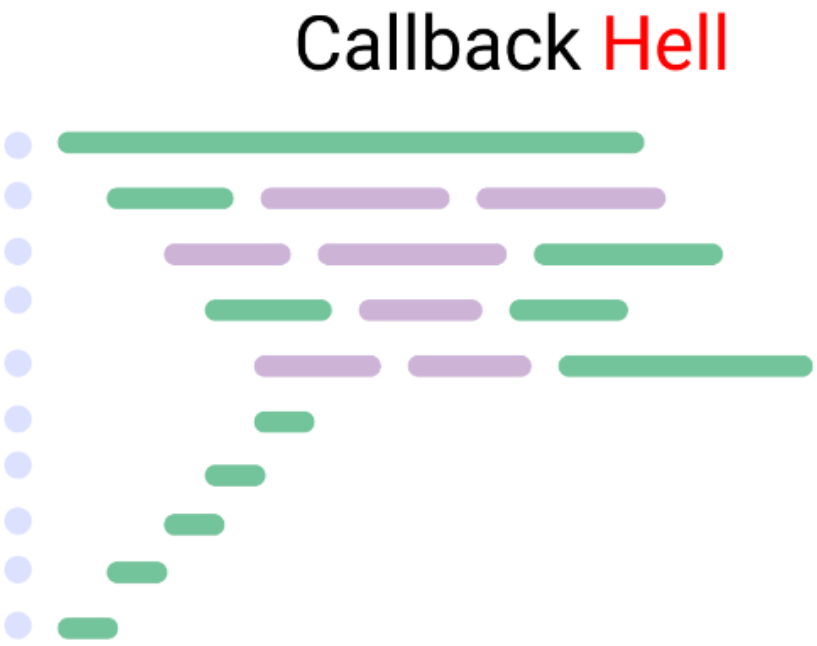


Illustration of Callback hell

What is solution for this?

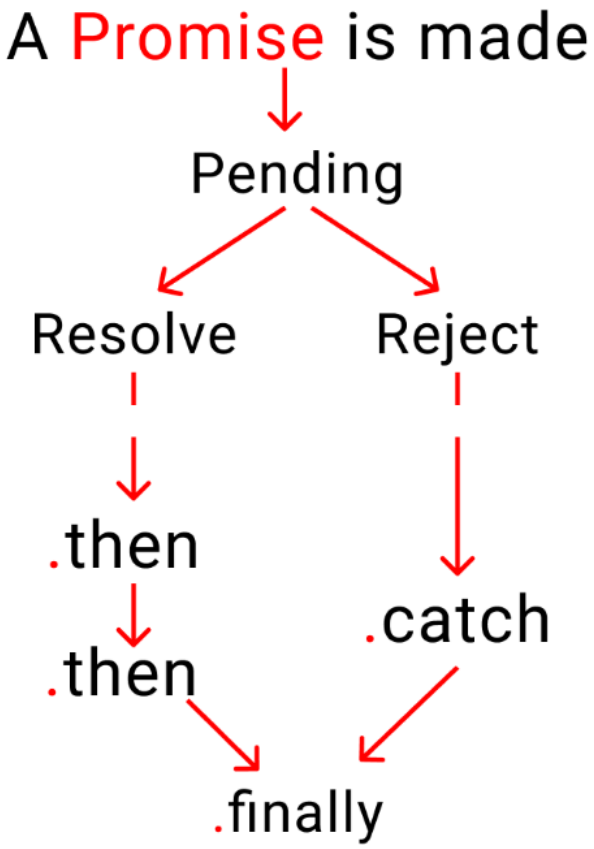
Promises were invented to solve the problem of callback hell and to better handle our tasks.

illustration of a promise format

Callback make relationship like : Parents -> Children -> grandchildren -> great grandchildren…….

But Promises take instruction. Hence we can keep our code neat and clean

Promise Cycle:



An illustration of the life of a promise

Eg

First of all promise is made that we will serve you an ice cream, it has two parts to this promise

Either it will be fulfilled or it will be rejected.

When the customer gets inside the stop just sits and thinks about what to order but hasn't ordered this state is called the **Pending** state. (Nothing is happening we didn't take orders, nor production has started)

Now suppose the customer asks for vanilla ice cream, if it is in stock then we can go to **Resolve** if we don't have it in stock then we have to **Reject.**

If we have that flavour then we have to go through **.then,** if we don't have Flavour we will send some message **catch** and the last step like closing steps **.finally**

We need to understand four more things first ->

- Relationship between time and work (eg steps like placing an order and cutting the fruits take 2 sec each)
- Promise chaining(**.then -> .then**)(first you have to do this then this then this)
- Error handling(**.catch)**
- The **.finally** handler

Relationship between time and work

Try to understand the code and see that, after two seconds of running it will display "strawberry was selected"

```
let stocks  = {
    Fruits:["strawberry","grapes","banana","apple"],
    liquid:["water","ice"],
    holder:["cone","cup","stick"],
    toppings:["chocolate","peanuts"]
}

// We are asking the question whether our shop is open or not
let is_shop_open = true;

let order = (time, work) => {
    return new Promise( (resolve, reject)=>{
        if(is_shop_open){
            setTimeout(()=>{
                resolve( work() )
                //The function inside the work will be called
            },time)
```

```
        }
        else{
            reject( console.log("Our shop is closed"))
        }
    } )
}
order(2000, () => {
    console.log(`${stocks.Fruits[0]} was selected`)
})
// ()=>console.log(`${stocks.Fruits[Fruit_name]} was selected)
//Even without curly brackets it will work
```

Promise Chaining

In this method we defining what we need to do when the first task is complete using the **.then** handler

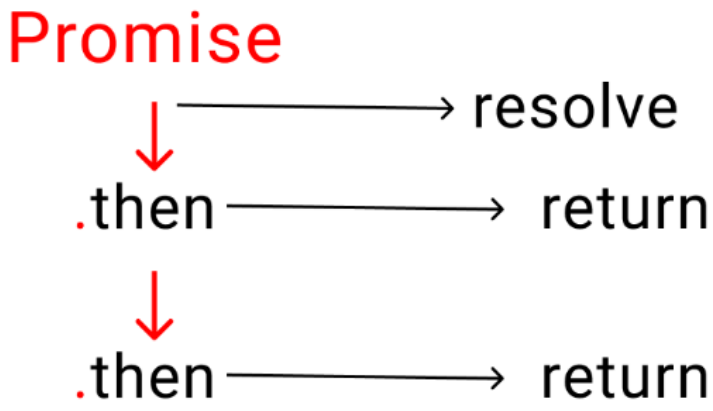


Illustration of promise chaining using .then handler

Example:

```
order(2000, () => {
    console.log(`${stocks.Fruits[0]} was selected`)
})
// promise chaining
.then(()=>{
    // if no return then it will not work
```

```
    return order(0, () => {
        console.log("production has started")
    })
})

.then(()=>{
    return order(2000, () => {
        console.log("The fruit has been chopped")
    })
})

.then(()=>{
    return order(1000, () => {
        console.log(`${stocks.liquid[0]} and ${stocks.liquid[1]} was
added`)
    })
})

.then(()=>{
    return order(1000, () => {
        console.log("The machine was started")
    })
})

.then(()=>{
    return order(2000, () => {
        console.log(`ice cream was placed on ${stocks.holder[0]}`)
    })
})

.then(()=>{
    return order(3000, () => {
        console.log(`${stocks.toppings[0]} was added as toppings`)
    })
})
.then(()=>{
    return order(2000, () => {
        console.log("Serve ice cream")
    })
});
```

```
//we cannot have semicolon between then, (;) this shows then() is
terminated
```

Error handling

We need a way to handle errors when something goes wrong. But first, we need to understand the promise cycle

Example :

To catch our errors, let's change our variable to false.

Let is_shop_open = false;

Which means our shop is close. We are not selling ice cream to our customer anymore.

To handle this we can use .catch handler. Just like .then it also returns a promise but only when our original promise is rejected

```
.then(()=>{
    return order(2000, () => {
        console.log("Serve icecream")
    })
})
.catch(()=>{
    console.log("Customer left")
    // will only work when our promise is rejected ()
})
```

Here's the result: 👇

A couple things to note about this code:

- The 1st message is coming from the `reject()` part of our promise
- The 2nd message is coming from the `.catch` handler

The .finally handler

The finally handler works regardless of whether our promise was resolved or rejected.

```
.finally(()=>{
    // will work everytime whether our promise is resolved or rejected
    console.log("Day ended, shop is closed")
})
```

Example:

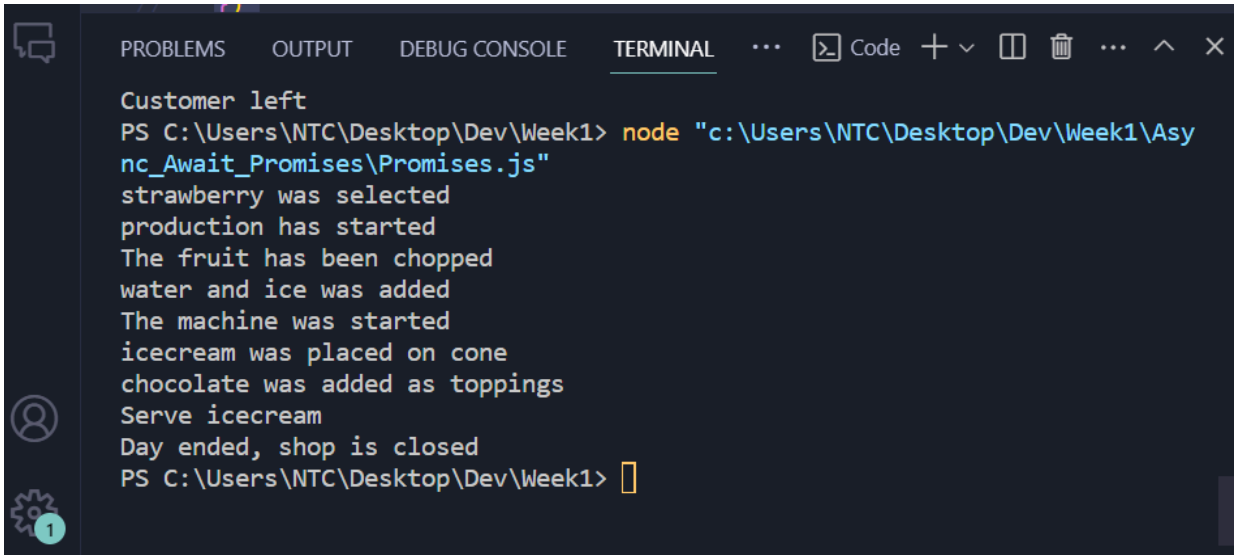
```
let stocks  = {
    Fruits:["strawberry","grapes","banana","apple"],
    liquid:["water","ice"],
    holder:["cone","cup","stick"],
```

```
    toppings:["chocolate","peanuts"]
}

// We are asking the question whether our shop is open or not
let is_shop_open = true;

let order = (time,work) => {
    return new Promise( (resolve, reject)=>{
        if(is_shop_open){
            setTimeout(()=>{
                resolve( work() )
                // the function inside the work will be called
            },time)
        }
        else{
            reject( console.log("Our shop is closed"))
        }
    } )
}
order(2000, () => {
    console.log(`${stocks.Fruits[0]} was selected`)
})
// promise chaining
.then(()=>{
    // if no return then it will not work
    return order(0, () => {
        console.log("production has started")
    })
})

.then(()=>{
    return order(2000, () => {
        console.log("The fruit has been chopped")
    })
})

.then(()=>{
    return order(1000, () => {
        console.log(`${stocks.liquid[0]} and ${stocks.liquid[1]} was
added`)
```

```
    })
})

.then(()=>{
    return order(1000, () => {
        console.log("The machine was started")
    })
})

.then(()=>{
    return order(2000, () => {
        console.log(`icecream was placed on ${stocks.holder[0]}`)
    })
})

.then(()=>{
    return order(3000, () => {
        console.log(`${stocks.toppings[0]} was added as toppings`)
    })
})
.then(()=>{
    return order(2000, () => {
        console.log("Serve icecream")
    })
})
.catch(()=>{
    console.log("Customer left")
    // will only work when our promise is rejected ()
})

.finally(()=>{
    // will work everytime whether our promise is resolved or rejected
    console.log("Day ended, shop is closed")
})
//we cannot have semicolon between then, (;) this shows then() is
terminated

// ()=>console.log(`${stocks.Fruits[Fruit_name]} was selected`)
// even without curly brackets it will work
```

```
// // dont use semicolon after curly brackets
// .then()
// .then()
// .then()
```

Output:



Async/ Await

This is supposed to be a better way to write promises and it helps us to keep our code simple and clean.

All you have to do is write the word **async** before any regular function and it becomes a promise.

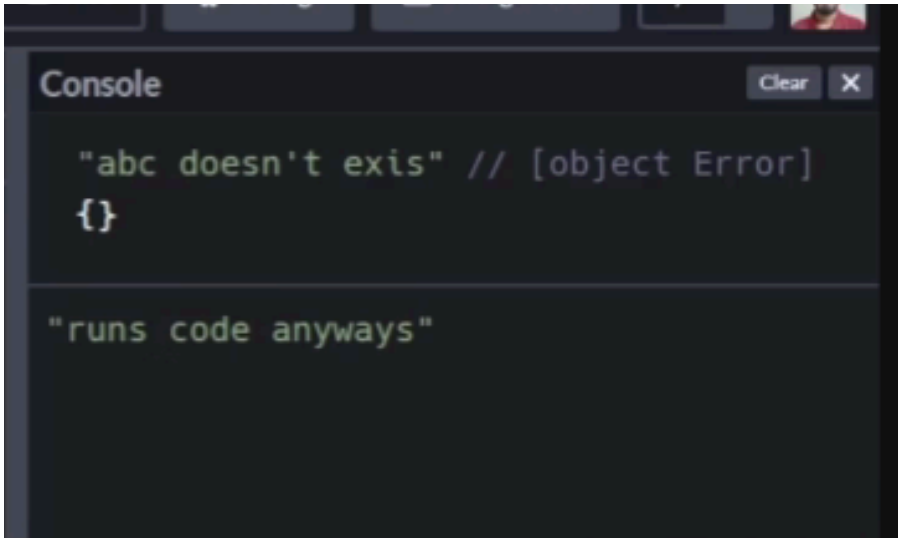
Promises Vs Async/ Await

```
let stocks  = {
    Fruits:["strawberry","grapes","banana","apple"],
    liquid:["water","ice"],
    holder:["cone","cup","stick"],
    toppings:["chocolate","peanuts"]
};
```

```
let is_shop_open = true;

let order = ()=>{
    return new Promise((resolve,reject)=>{
        if(is_shop_open){
            resolve();
        }
        else{
            reject();
        }
    })
}
order()
.then()
.then()
.then()
.catch()//promises is rejected
.finally()//always runs
```

Async/ Await **Try and Catch**

```
async function order(){
    try{
        await abc;
        // pointing out to function which does not exist
    }
    catch(error){
        console.log("abc doesn't exis",error)
    }
    finally{
        console.log("runs code anyways")
    }
}
order();
```



We need to understand

- How to use **try** and **catch** keywords
- How to use **await** keyword

await keyword

Lets go back to the ice cream shop suppose you have forgot to ask which topping to put on ice cream from a customer, chocolate or peanuts. So we need to stop our machine and go and ask our customer what they'd like on their ice cream.

Notice that only the kitchen is stopped and workers outside are still working.

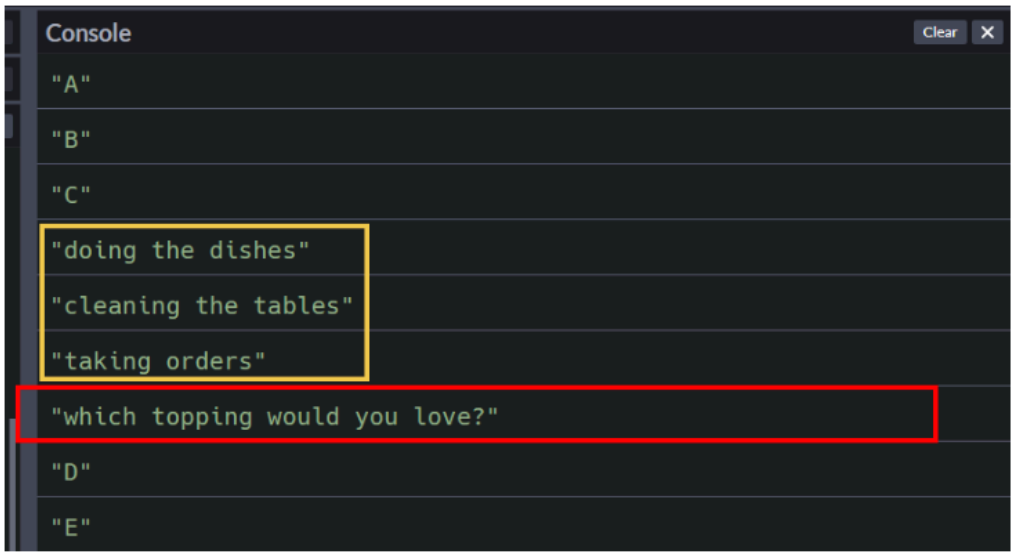
Here comes the implementation of the await keyword

Examples:

```
let stocks  = {
    Fruits:["strawberry","grapes","banana","apple"],
    liquid:["water","ice"],
    holder:["cone","cup","stick"],
    toppings:["chocolate","peanuts"]
};
```

```
let is_shop_open = true;

// made promise that we will go outside the kitchen and ask what toppings
you want
let toppings_choice = () => {
    return new Promise( (resolve, reject) => {
        setTimeout(()=>{
            resolve(console.log("which topping would you love?"))
        },3000)
    })
}
async function kitchen(){
    // There are steps that we need to follow inside the kitchen (we are
working here and we realize that we dont know which topping to add after C
step)
    console.log(" A ");
    console.log(" B ");
    console.log(" C ");
    await toppings_choice()
    console.log(" D ");
    console.log(" E ");
}
kitchen();
// meanwhile chef is outside the kitchen other works like these will still
run.
console.log("doing the dishes");
console.log("cleaning the tables");
console.log("taking others orders");
```

Output:



Small note

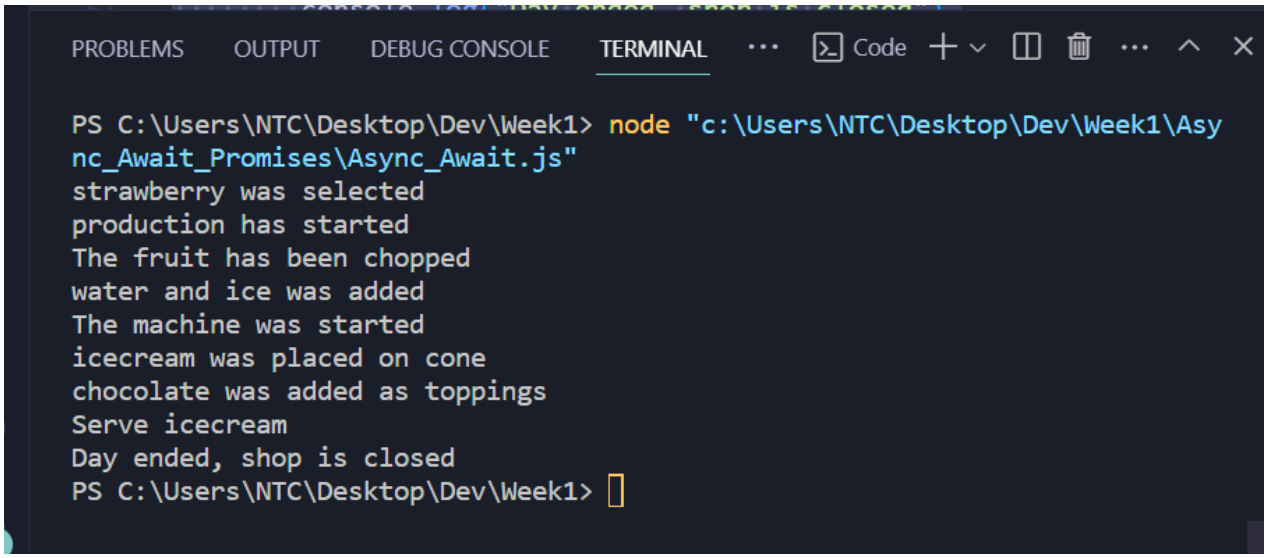
When using Async/ Await, you can also use the `.then`, `.catch`, and `.finally` handlers as well which are a core part of promises.

Final Code:

```
let stocks  = {
    Fruits:["strawberry","grapes","banana","apple"],
    liquid:["water","ice"],
    holder:["cone","cup","stick"],
    toppings:["chocolate","peanuts"]
};

let is_shop_open = true;

function time(ms){
    return new Promise((resolve,reject)=>{
        if(is_shop_open){
            setTimeout(resolve,ms);
        }
        else{
            reject(console.log("Shop is closed"))
        }
```

```
    })
}

async function kitchen(){
    try{
        await time(2000)
        console.log(`${stocks.Fruits[0]} was selected`)

        await time(0)//take zero seconds
        console.log("production has started")

        await time(2000)
        console.log("The fruit has been chopped")

        await time(1000)
        console.log(`${stocks.liquid[0]} and ${stocks.liquid[1]} was
added`)

        await time(1000)
        console.log("The machine was started")

        await time(2000)
        console.log(`icecream was placed on ${stocks.holder[0]}`)

        await time(3000)
        console.log(`${stocks.toppings[0]} was added as toppings`)

        await time(2000)
        console.log("Serve icecream")
    }
    catch(error){
        console.log("Customer left",error)
    }
    finally{
        console.log("Day ended, shop is closed")
    }
}

kitchen();
```

Output:


PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
Code
PS C:\Users\NTC\Desktop\Dev\Week1> node "c:\Users\NTC\Desktop\Dev\Week1\Async_Await_Promises\Async_Await.js"
strawberry was selected
production has started
The fruit has been chopped
water and ice was added
The machine was started
icecream was placed on cone
chocolate was added as toppings
Serve icecream
Day ended, shop is closed
PS C:\Users\NTC\Desktop\Dev\Week1>