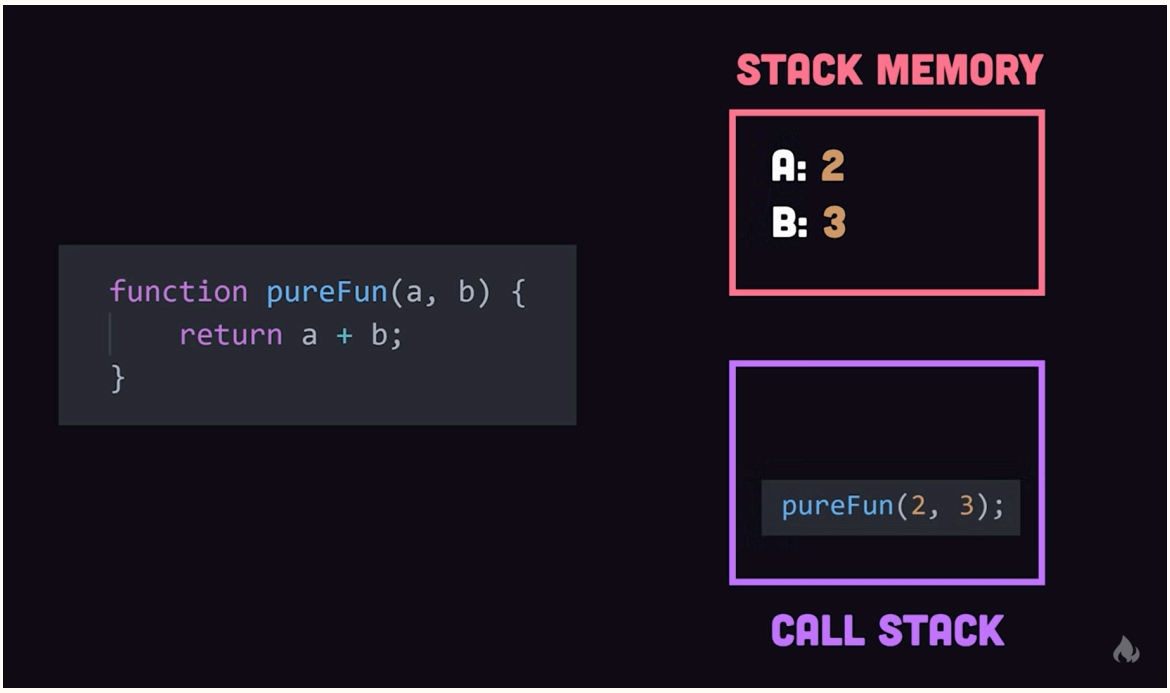
React js Interview Notes

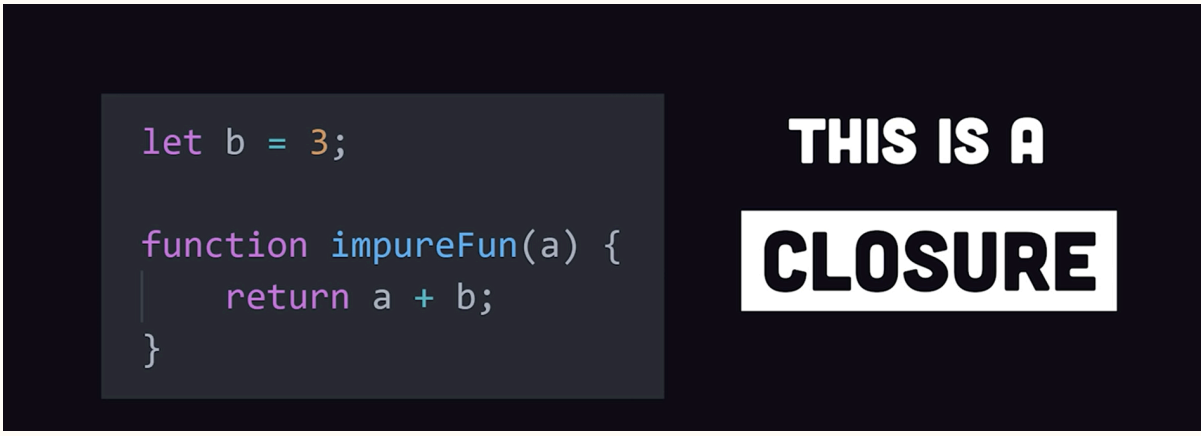
CLOSURES

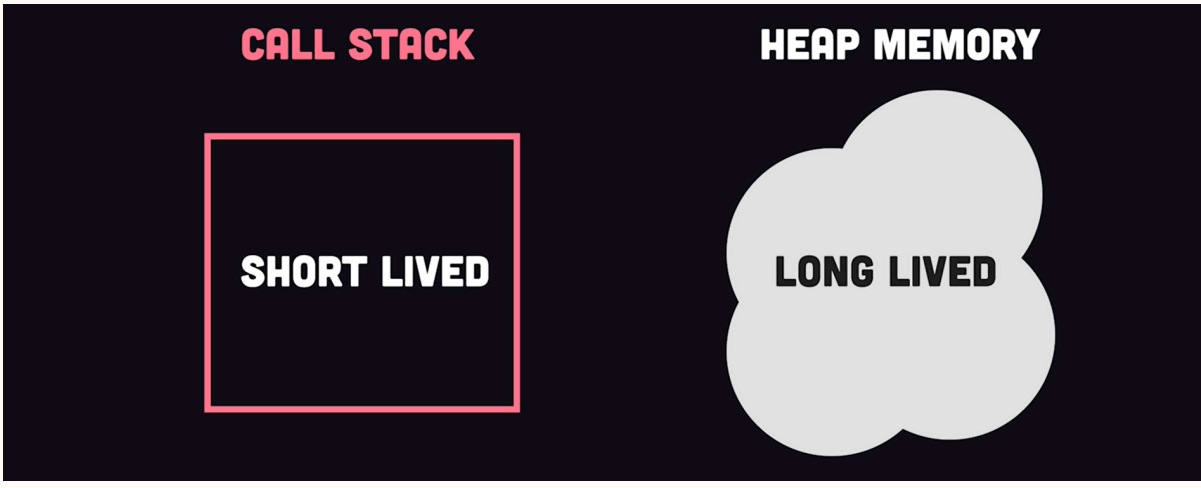
- **Closures:**
- It is used to preserve variables from outer scope *(Lexical Scope)* of a function in the inner scope of a function.
- Example #1:
  - If its a stateful function (Take works with the variable outside of its scope a.k.a Lexical environment) it would be having a closure
  - A pure function would not have a closure (Pure function: that works with its own arguments only)
  - 

  - It instantly removes from call stack => stack memory when the function executes unlike a stateful function that stores in Heap Memory and not in Stack Memory. The data removes from the Heap memory is with its builtin garbage collector.

- Example #1:

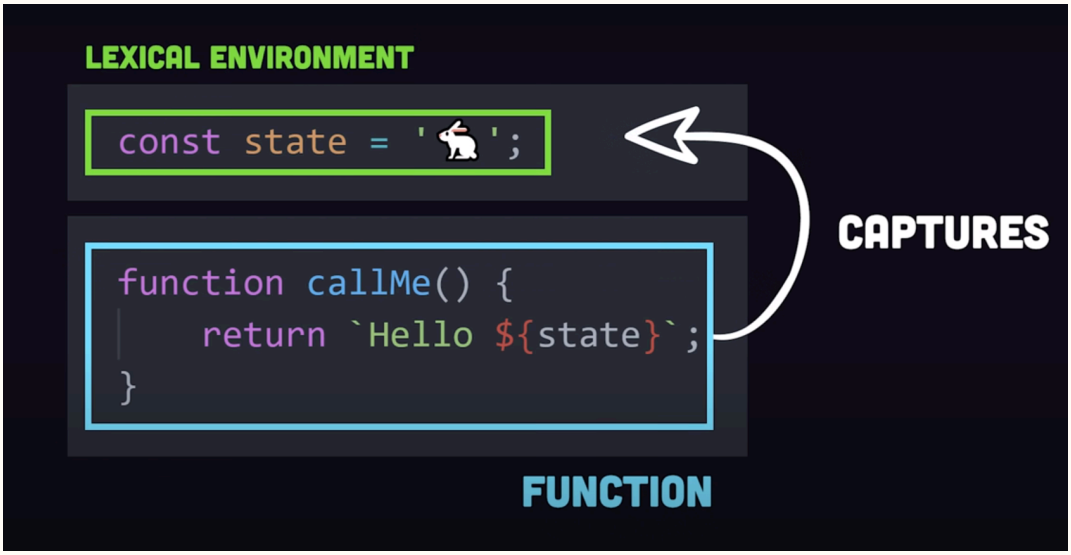

STACK MEMORY
A: 2
B: 3
function pureFun(a, b) {
return a + b;
}
pureFun(2, 3);
CALL STACK


let b = 3;
function impureFun(a) {
return a + b;
}
THIS IS A
CLOSURE


CALL STACK
SHORT LIVED
HEAP MEMORY
LONG LIVED

```
function outerFunction(outerVariable){
return function innerFunction(innerVariable){
console.log("OuterVariable:" , outerVariable)
console.log("Inner Variable:", innerVariable)
}
}

Const newFunction = outerFunction("outside")
newFunction("inside")
```

- Example #2:

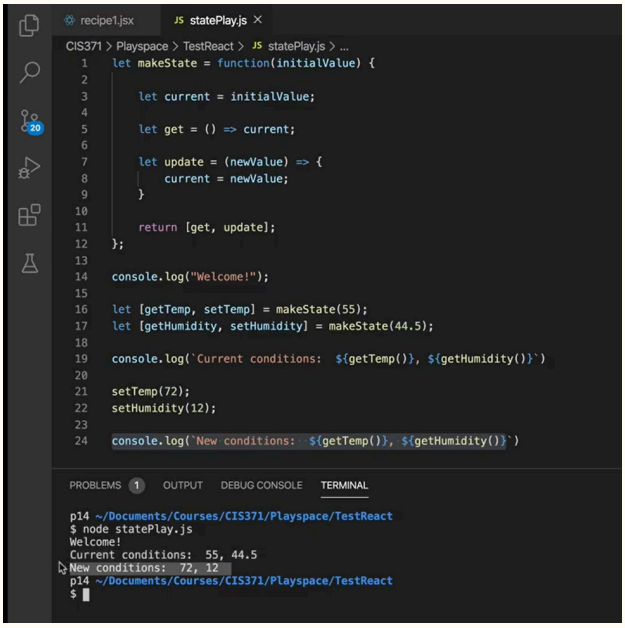


- Example #3:

```
for (var i = 0; i<3; i++){
const log = () =>{
console.log(i) //output ?
}
seTimeout (log,100);
}
```

Example#2

Under the hood how we can replicate useState hook , it also uses closures
Returning multiple functions from a function and how to access them using array destructuring. Each time makeState is called its making a closure with the values inside it



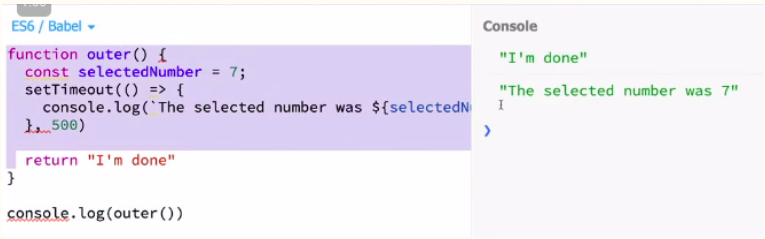
• Stale Closures:

- The Link for Stale Closures **LINK1**
- This Link will show why we need to use useRef to avoid state update in useEffect that causes a closure **Link2** **Link3**

Notes

Link3 further explains link 2 , about stale closures , how a factory function returns first and when to use useRef in useEffect etc

Closure Good example:



- ## Mistakes in Closures:

HOISTING

- Var is hoisted globally:

  Example#1

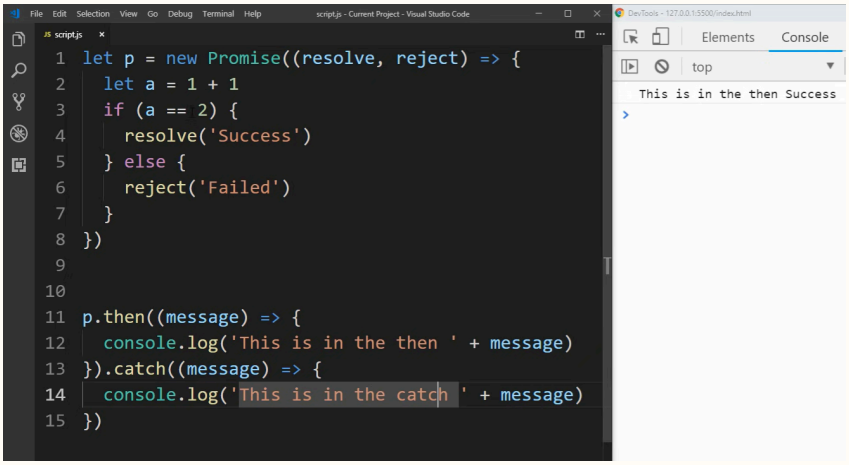
```
Var num = 8;
Function display (){
Var num = 10;
console.log(num)
}
display();
console.log(num) // 10 , 10 //because var hoisted globally
```

RACING CONDITION

- In use Effect we have an API call with dependency , each time the dependency changes a new api call will trigger and all those api calls will be in closure , unless with change of dependency you first clear that call on mount then make a call on mount. Best Example is in LINK

Synchronous VS Asynchronous JavaScript

Promises:

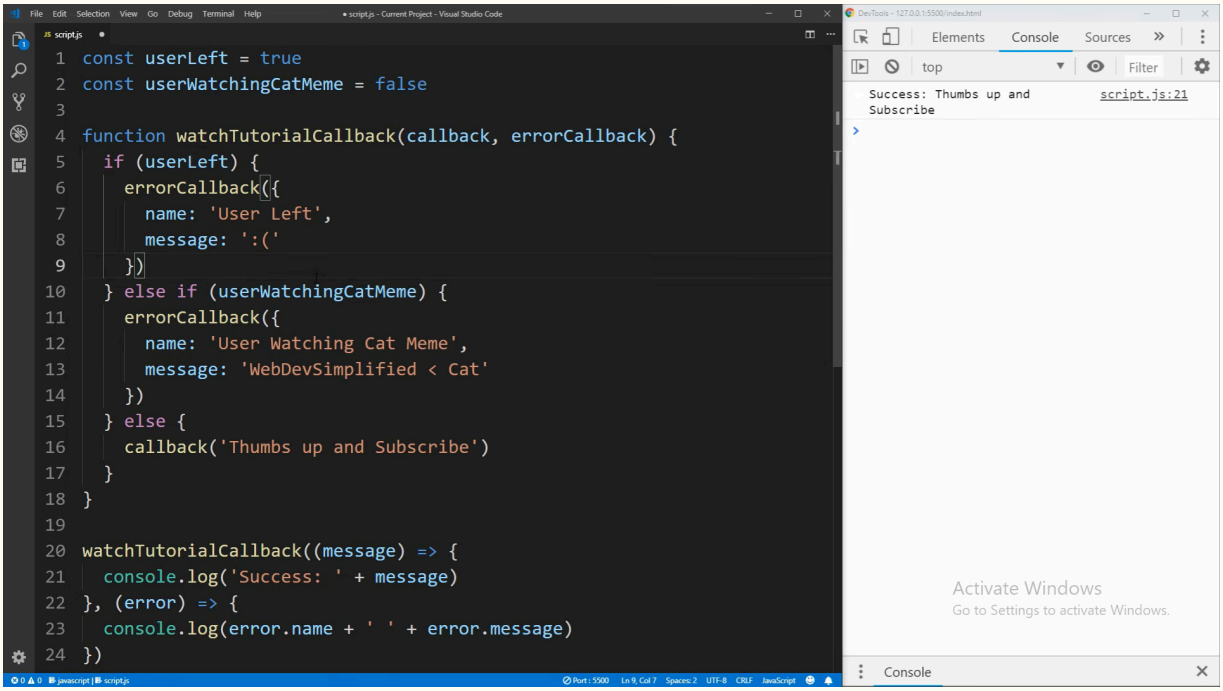


Why do we need to use Promises ?

If the promise is successful, it will produce a resolved value, but if something goes wrong then it will produce a reason why the promise failed.

Example#1:

=> To avoid callbacks, below is the code that mimics Promises but with callbacks.



```
const userLeft = true
const userWatchingCatMeme = false

function watchTutorialCallback(callback, errorCallback) {
  if (userLeft) {
    errorCallback({
      name: 'User Left',
      message: ':('
    })
  } else if (userWatchingCatMeme) {
    errorCallback({
      name: 'User Watching Cat Meme',
      message: 'WebDevSimplified < Cat'
    })
  } else {
    callback('Thumbs up and Subscribe')
  }
}

watchTutorialCallback((message) => {
  console.log('Success: ' + message)
}, (error) => {
  console.log(error.name + ' ' + error.message)
})
```

```
Success: Thumbs up and Subscribe   script.js:21
```

Now to modify the above code in Promises we do:

```
function watchTutorialPromise() {
  return new Promise((resolve, reject) => {
    if (userLeft) {
      reject({
        name: 'User Left',
        message: ':('
      })
    } else if (userWatchingCatMeme) {
      reject({
        name: 'User Watching Cat Meme',
        message: 'WebDevSimplified < Cat'
      })
    } else {
      resolve('Thumbs up and Subscribe')
    }
  })
}
```

```
User Watching Cat Meme
WebDevSimplified < Cat
```

```
watchTutorialPromise().then((message)=>{
console.log("Success",+message)
}).catch((error)=>{
console.log(error.name+" "+error.message)})
```

Callback Hell:

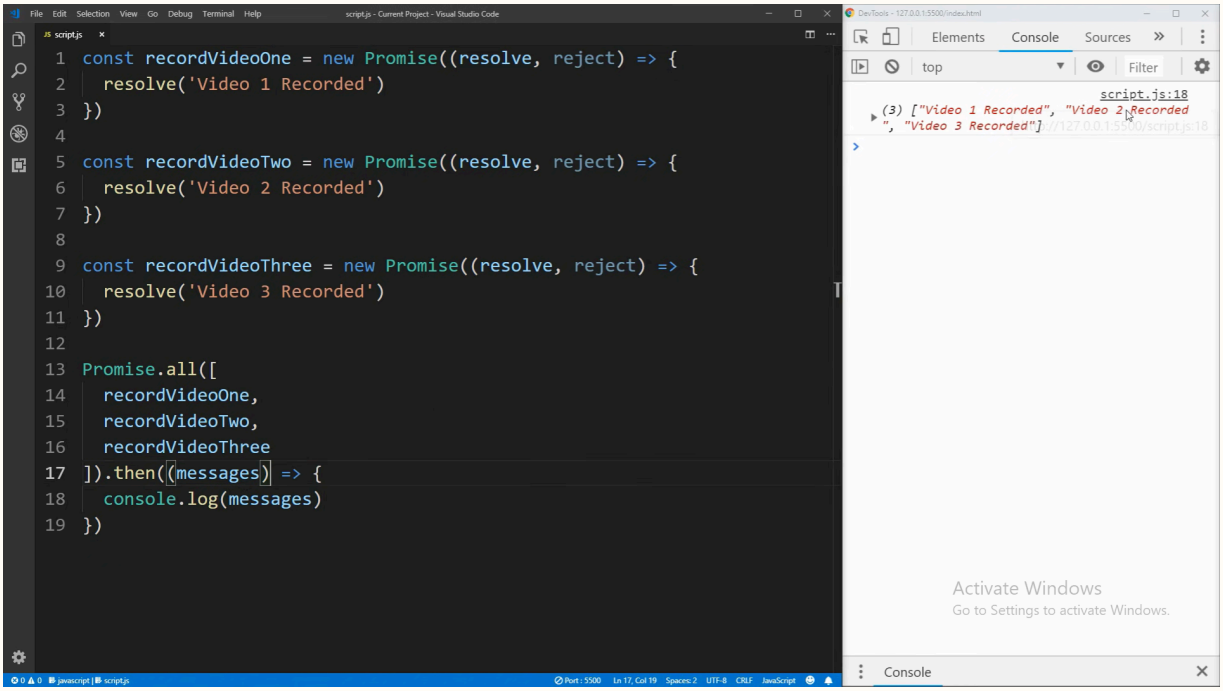
To avoid callback hell we had to use Promises:
But with Promises we came across .then Hell

Promise Types

Promise All:

It will execute when all 3 promises/API calls at a same time and will return the responses (all 3). Then Will execute only when all 3 of the APIs/Promises will resolve and neither will be rejected.

- This method returns a single Promise that fulfills when all of the promises in the iterable argument have fulfilled.
- If any of the promises in the iterable reject, the returned promise will be rejected with the reason of the first promise that was rejected.



Promise Race:

It will execute as soon as one of the API or Promise will resolve or Reject: It will return only one Return value , either that will fall in .then or .catch. Syntax: Promise.Race

Promise.allSettled:

This method returns a promise that resolves after all of the promises in the iterable have either fulfilled or rejected, with an array of objects describing the outcome of each promise.

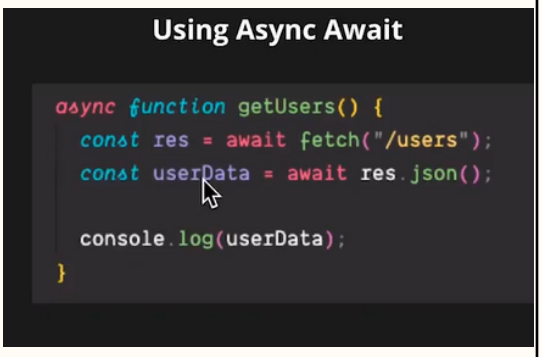
```
const promise1 = Promise.resolve(1);
const promise2 = Promise.reject('Error in Promise 2');

Promise.allSettled([promise1, promise2])
  .then(results => console.log(results))
  .catch(error => console.error(error));
```

Promise.any:

This method returns a promise that fulfills as soon as one of the promises in the iterable fulfills. It rejects with an AggregateError if all promises are rejected

Promise Vs Async Await:

1. Async Await is not an alternative to promise
2. Async Await is not different than promises
3. Async Await enables you to write asynchronous behavior code in a cleaner and readable way
4. Its nothin but a syntactic sugar or a wrapper created over promises to make code look synchronous

.Then / Promises	Async Await
	

```
function getUsers() {
  fetch("/users")
    .then((res) => res.json())
    .then((userData) => {
      console.log(userData);
    });

    console.log('Printing user data')
}
```

Output

Printing user data

userData { ...}

```
async function getUsers() {
  const res = await fetch("/users");
  const userData = await res.json();

  console.log(userData);
  console.log("Printing user data");
}
```

Output

userData { ...}

Printing user data


```
function getUserDetails() {
    fetch("/authenticate")
    .then((token) => {
        fetch("/userId", token)
        .then((userId) => {
            fetch("/userDetails", userId)
            .then((userData) => {
                console.log(userData);
            });
        })
    })
```

.then Hell

```
async function getUserDetails() {
    const token = await fetch("/authenticate");
    const userId = await fetch("/userId", token);
    const userDetails = await fetch("/userDetails", userId);

    console.log(userDetails);
}
```

Convert Below code into Async Await

```
function getUsersAvatar() {
    fetch("/users")
    .then((res) => res.json())
    .then((users) => {
        for (let user of users) {
            fetch(`/users/${user.id}`)
            .then((res) => res.json())
            .then((data) => console.log(data.imageUrl));
        }
    });
}
```

```
1  async function getAvatar(user){
2      const res = await fetch(`/users/${user.id}`)
3      const data = await res.json();
4
5      console.log(data.imageUrl)
6  }
7
8  async function getUsersAvatar() {
9      const response = await fetch("/users")
10     const users = await response.json()
11
12     for(let user of users){
13         getAvatar(user)
14     }
15
16 }
```

1. Await keyword pauses the execution of the function in which its declared
2. In this case , wrapping it inside another function paused the execution of that function and not the main function
3. Instead of for loop we can simple use .map or .forEach cause they have callback , we will e writing async keyword on start of their callback
4. For above code, user api calls will run in parallel, on the other

	hand if it's not wrapped in function then user apis will complete one by one in line

//	**WRONG:**
	 
	async function getUsersAvatar() {
	const response = await fetch("/users")
	const users = await response.json()
	
	for(let user of users){
	const res = await fetch(`/users/${user.id}`)
	const data = await res.json();
	
	console.log(data.imageUrl)
	}
	
	}

Mistakes in Async Await

CONCATENATION

Precedence is like DMAS rule (only + makes a string)

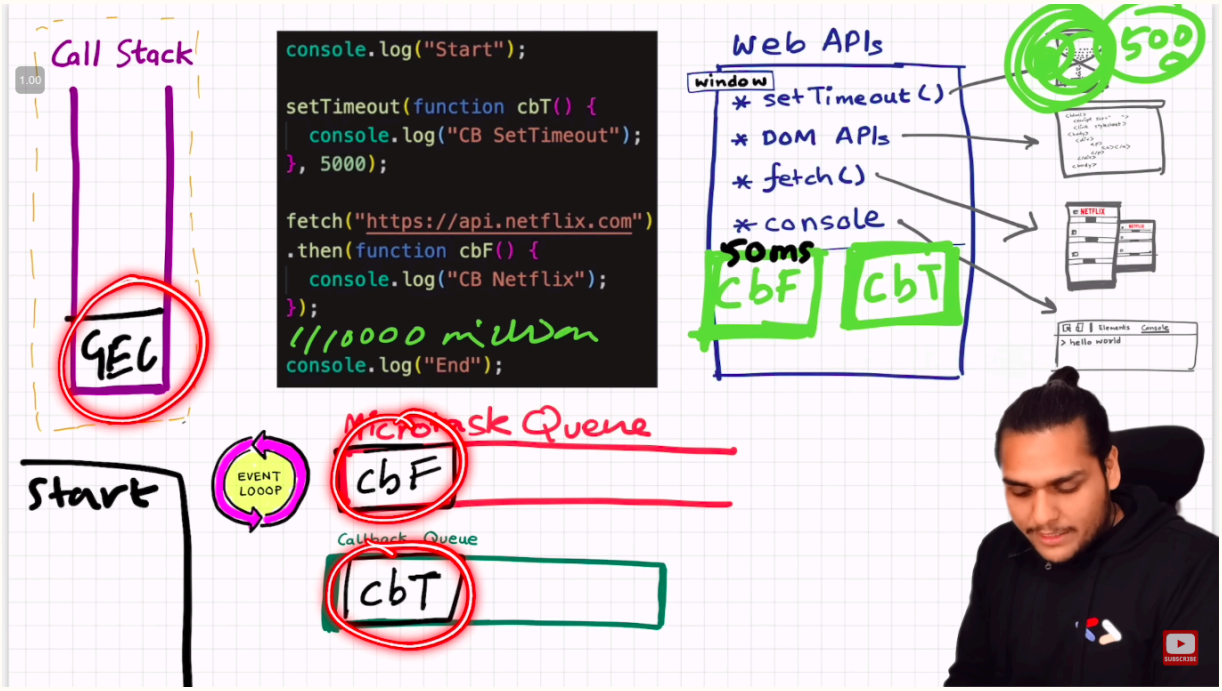
"1"+1 = "11"	"2"*4 = 8	"2" * "4" / "4" = 2
"1"+"1" = "11"	"2"*"2"=8	"4" / "4" =1
"1"+1 - 1 = 10	"2" * "4" + "4" = "84"	"4" / 4 =1

EVENT LOOP

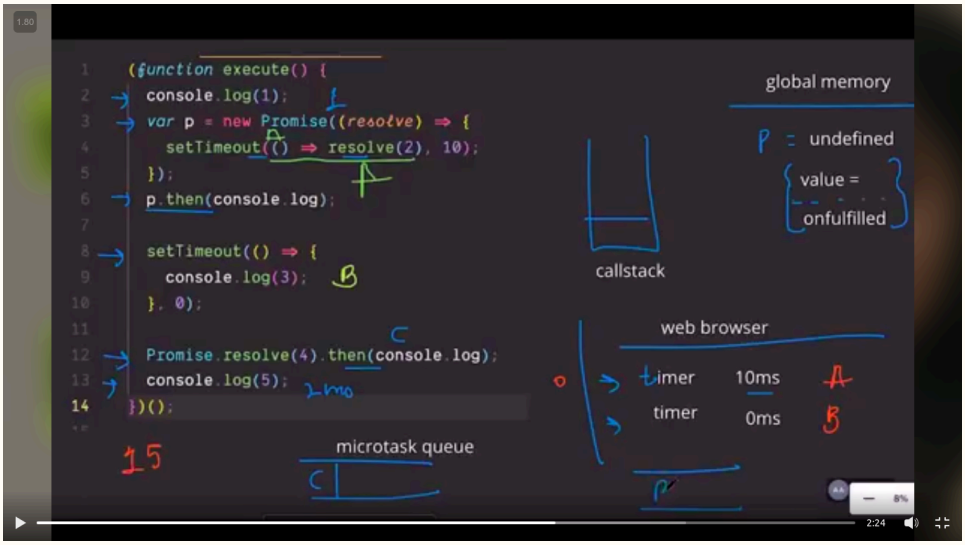
Javascript is a Single synchronous Threaded Language

Notes:

1. Call Stack that is responsible for execution is in **JS Engine**.
2. Browser Gives us **Web APIs**, To use the super powers of browser
3. If there are callbacks they will directly fall in call Stack but if there is a webAPI like setTimeout it will sent/Register it to (WEB API). After that it will decide based on priority whether to put it in Callback Queue or MicroTask Queue
4. Event Loop basically continuously checks the call stack if its empty or not , if it is, then it checks the callback queue or microTask queue.
5. **Important:** If code is executing and there are callbacks in Callback Queue. It will not execute because CallStack is not Empty.

6. 

7. **Important:** All the Callback functions that come from Promises will come in Micro Task Queue. Like **Fetch()** , OR Mutation Observer.
8. If there are multiple CB in MQ , all them will execute then CB in CQ will execute. It is called **STARVATION.** IT happens when there are many tasks inside MQ and the Task in CQ has to wait for a long time.
9. Explanation:
    a. Event Loop will always look at callStack if its empty or not if its empty it then executes Queue
    b. In above example when thread of execution comes at setTimeout and Fetch it sends it to Web API, once its resolved it sends the callbacks to their respective Queues
    c. Fetch web api (ALL the Callbacks comes from PROMISES and Mutation Observer) will send it to MICRO Task Queue and all other callbacks from other WebAPIs will sent to CALLBACK queue.
    d. **IMP:** From WEB API's , it will only sends the callback to any Queue once its completed (Once Timer is completed || Once Promise is Resolved || Once Event handler triggers(DOM API)).

e. 

f. Guess the Above Answer, Here A is in WebBrowser as Timer and not as promise , How i Did was that , A will be in MiroTask queue before C. Once the A will execute from MicroTask , it will send A into now web browser because A is now setTimeout it would wait for its timer to complete.
But this approach is wrong though the output is same
Correct will be that, A will be in WebBrowser as Timer because promise's settled callback will be in MicroTask but in ourcase of A , in settled state it is just loging the number so it would just display if there were to be a callback it would send that in MicroTask queue.

g. 

Link

h. In above example see that 1 comes before 3 , why? because the thread of execution is at line 2 then it immediately resolves and sends it to MicroTask and line 6 executes , once the call stack empties it executes result .
    i. If resolve(2) as wrapped in another setTimeout it will execute first and once its executes and settles it then sends its callback to MicroTask .

i. Remember : Javascript is Single Threaded means JS Engine will execute one line at a time but the Browser executes multiple things making it asynchronous.

10. Most **IMPORTANT** concept of Async Await with Event Loop ***LINK***.

a.

```
async function test() {
  console.log('Before async await');
  const num1 = await testPromise1();
  const num2 = await testPromise2();
  const num3 = await testPromise3();
  console.log(num1 + num2 + num3);
  console.log('After async await');
}
console.log('Before calling test');
test()
console.log('After calling test');
```

b. Guess the output of above, remember **async await DOES NOT BLOCK THE THREAD OF EXECUTION**. It is nothing but syntactic sugar over Promises. Instead of writing nested promises we can write async await

The big Question arrises if async await is a syntactical sugar then why does async function waits . its because only A will wait because its a promise . and not C even though it is in async.

```
async function getData(){
    console.log(C)
let A =  await Promise.resolve("A")
    console.log(A)
}
getData()
console.log("B")

//answer:  C B A
```

ARRAY + STRING METHODS

W3 Scools :

Except (at(), constructor(), copyWithin(), entries() **)**

Important :

every()	returns true if all conditions return true

fill()	.fill("Kiwi", 2, 4) Fills the defined indexes with a given value. `array.fill(value, start, end) , end is not included`
filter()	return items if a condition returns true
find()	same as filter but it returns only first value if a condition returns true
findIndex()	Returns the index of found element
flat()	flattens the array , we can specify the depth
map()	for loop with return , i can store it in a variable
forEach(),	do not return
includes()	returns true if an element matches we can also pass a starting position to start searching for that element in an array
indexOf()	it returns the initial index of element that we give in param , (STRING and ARRAY) . The index of the first element with a specified value
findIndex()	The index of the first element that passes a test
isArray()	checks if its an array or not
join()	Returns an array in between commas as a string but we can give our own join string like this: fruits.join(" and ")
concat(),	concat two or more arrays into one
lastIndexOf()	gives the index of last matching element
pop()	removes from back
prototype(),	we can define object methods with it and then use them with object.newFunction()
push()	adds an element to the end of an array
shift()	Removes from start
slice()	fruits.slice(1, 3); fruits.slice(-3, -1); we get only the slice

splice()	const fruits = ["Banana", "Orange", "Apple", "Mango"];
	// At position 2, add 2 elements: (on place of apple -> lemon)
	fruits.splice(2, 0, "Lemon", "Kiwi") // if 1 instead of 0 Apple will be removed

	// At position 2, remove 2 items:
	fruits.splice(2, 2);

some()	if anything matches/returns true
toString(),	it does not returns an array like join but only string
valueOf():	//not specific to arrays: "*In JavaScript, the valueOf() method is not specific to arrays; rather, it's a method tha is inherited by all objects, including arrays. The valueOf(method is automatically called by JavaScript whenever an object is to be represented as a primitive value.* *For arrays, the valueOf() method returns the array itself. This method is called implicitly in certain situations, such as when an array is used in a context where a primitive value is expected. The primary use case is for type coercion, where an object needs to be converted to a primitive type.*")

Primitive:

There are 7 primitive types:
String, number, bigint, boolean, null, undefined, and symbol (IMMUTABLE)

UseCallback and useMemo Hook:

Description:
useCallback hook is used when we are sending a function to another component, so on each setState component will re-render thus it will make a new instance for that function. Similarly use useMemo only when we are using a complex method that returns a result. Whether we are using that same component or passing that in another component.
If the value is non primitive like arr or obj and we are passing it in the method and its not expensive , it will re-render the child component in which we are passing it in, so we should use React.memo in that child component.

Summary Final: use Ripplet IDE for ref or for practice

1. useCallback are used for functions , means that the variable in which we are storing can either be a function or a value like(arr,obj,string,number), if it is function we will use callback and if its a value either primitive or nonprimitive we will use useMemo.

2. Use both callback and useMemo only for optimization and for stopping re-rendering if the component or the memoized value or function is not that expensive.
3. As we know even a single state will re-render the whole component thus re-rendering its child components too, So only use React.Memo with those if it will really optimize the component by stopping the unwanted re-rendering.
4. We can pass setState as it is, but we should not and we should always pass a callback function with proper naming convention.
5. Never forget to use the var in dependency array to un memoize that value or function.
6. Even if we use React.Memo on child and donot use useCallback or usememo on the methods or values(non-primitive) that are passed as props, the child component will still re-render because function and non-primitive values does have a unique instance on every instance.

useDeferredValue:

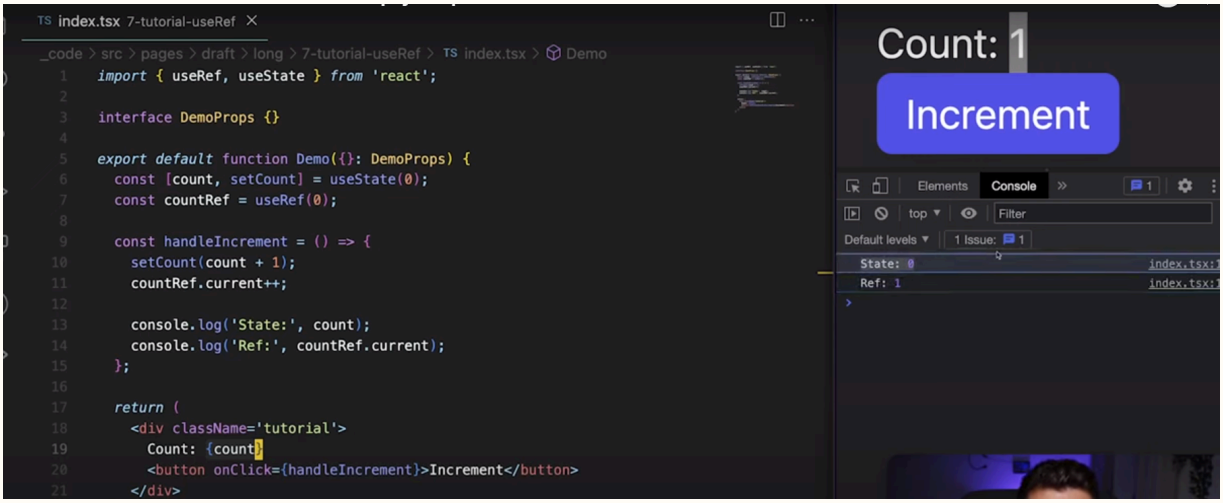
```
import { useState } from 'react';

import SlowList from './SlowList';

const Demo = () => {
  const [query, setQuery] = useState('');

  return (
    <div className='tutorial'>
      <input
        type='text'
        value={query}
        onChange={(e) => setQuery(e.target.value)}
        placeholder='Search...'
      />
      <SlowList text={query} />
    </div>
  );
};

export default Demo;
```

```tsx
import { memo } from 'react';

const SlowList = memo(({ text }: { text: string }) => {
  const items = [];
  for (let i = 0; i < 250; i++) {
    items.push(<SlowItem key={i} text={text} />);
  }
  return <ul className='items'>{items}</ul>;
});

const SlowItem = ({ text }: { text: string }) => {
  let startTime = performance.now();
  while (performance.now() - startTime < 1) {
    // Do nothing for 1 ms per item to emulate extremely slow code
  }

  return <li className='item'>Text: {text}</li>;
};

export default SlowList;
```

We made our app work with the slowness of our component. Due to only one render after typing something, not re-rendering on each key press.
Explanation: If i type only one letter it means we ill do 1 re-render but we are doing 2 re-renders , one from the actual state query and one from deferred value .
If i type 2 letters, we will have 3 re-renders 2 re-renders from the actual state query and one re-render from the Deferred value that has same value as in query

useRef:

When ever we need a value without re-rendering:



IntrestingInfo For set state, to access the updated value we only get it in re-render that is caused by that state change.
useRef: we can access its value right away no need to wait for re-render because useRef wont even trigger a re-render

ForwardRef():

I want to access the ref of my child component from the parent component, simple is that . If i want to do vice versa like accessing parent ref from child it's a wrong approach and must be corrected, ref control should be in parent or where its using and not in the child component.

Iterator Pattern:

Iterators:

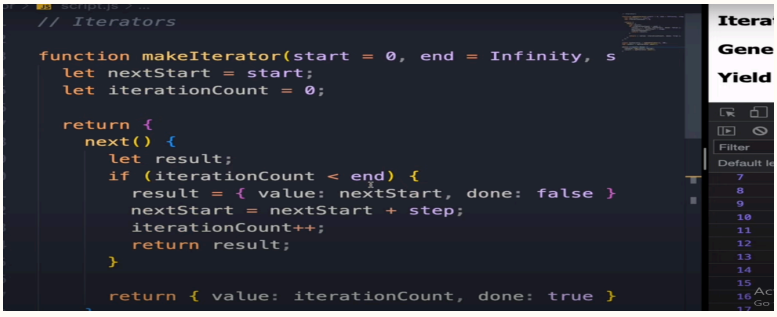
Example:
for(const val of [1, 2, 3, 4, 5]){
console.log(val)
}

Above, how did it know to traverse and to loop the values of the array?
Because in Array there is a built in iterator function that whenever we get an array how to print it.

To customize these built in iteration functions of an array we have

1. Iteration protocols
2. Generators
3. Yields

Custom Iterator:



Example present in docs. We donot need to cramb it . its a custom iterator and our js wont be able to use it in for loop.

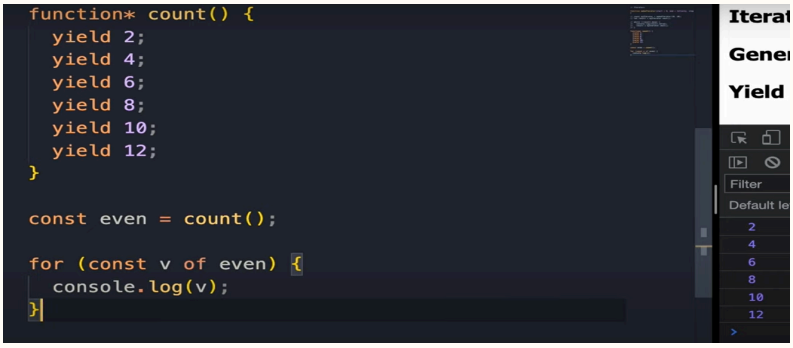
Yield:

Yield is nothing but the keyword to display the value in generator function

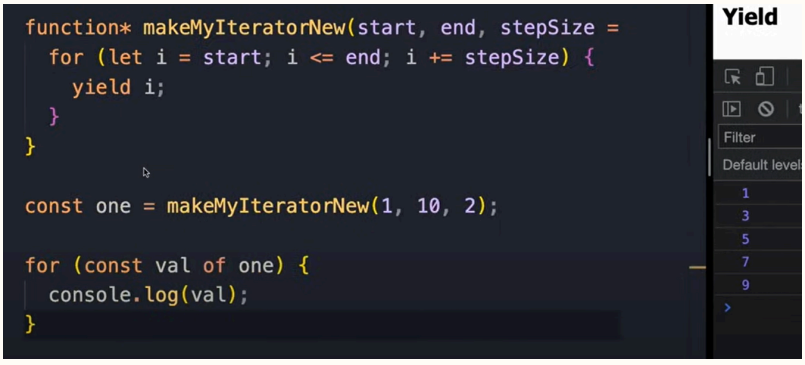
Generator Function:

It provides an alternative to iterator who is easy to use.
Syntax = function*

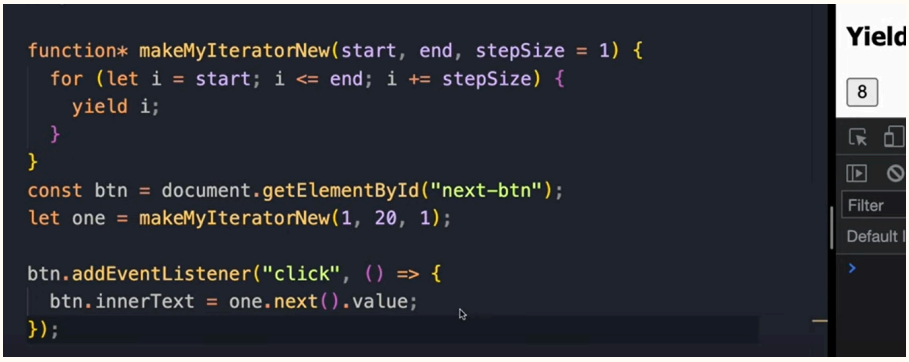
Below is the simple non used basic example of how to use the generator function with the yield.



Below is the actual implementation of yield in generator function.
Here our generator function is starting and ending with our desired stepsize

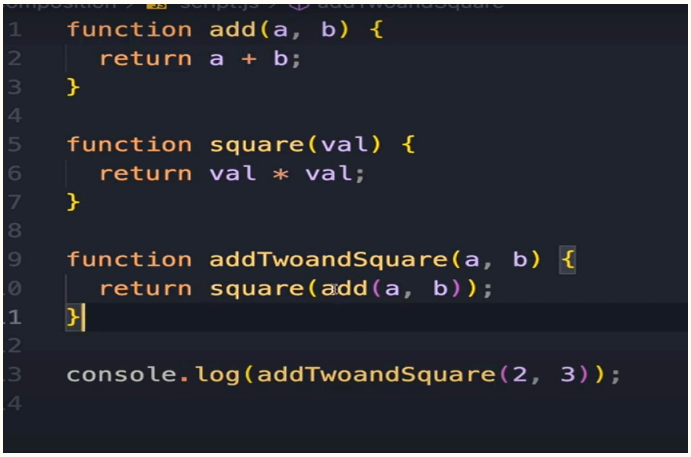


UseCase:
Below is the one usecase . In our custom iterator we can hold the value of iteration unlike the conventional for loop default iterator. How with the use of generator function's built in methods .next and .done

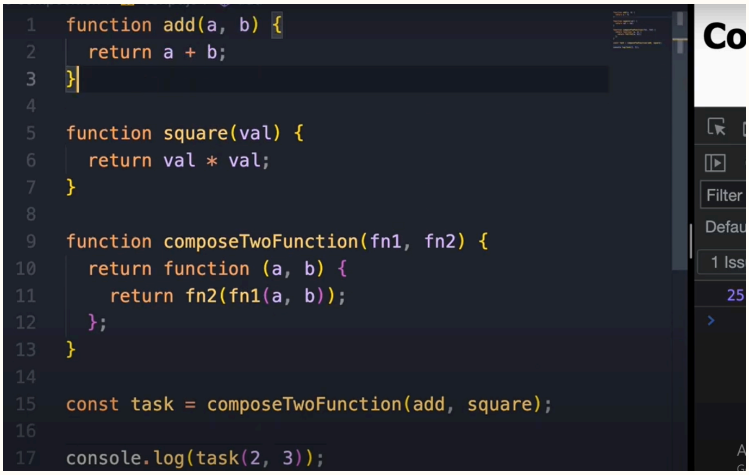


Composition:

Below is the conventional method to use multiple functions continuously , it can be tedious if there are too many functions we have to create a new composition function for each use case.



Below is the way that we can make a composite function with the help of closure.



Below is the way to write the compose function using latest ES

```
function composeTwoFunction(fn1, fn2) {
  return function (a, b) {
    return fn2(fn1(a, b));
  };
}

const c2f = (fn1, f2) => (a, b) => f2(fn1(a, b));

const task = c2f(multiply, square);

console.log(task(2, 3));
```

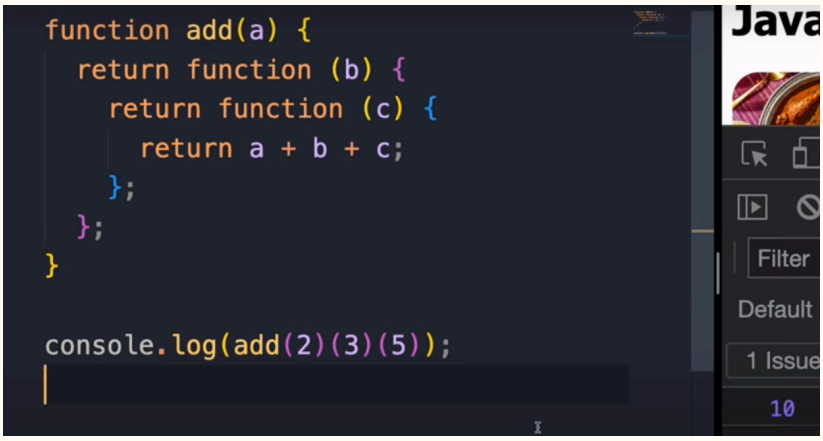
If we want to create a generic compose function for many functions and arguments we can use this. Take a look at reduce method too once

```
function compose(...fns) {
  return function (...value) {
    return fns.reduce((a, b) => b(a), value);
  };
}
```

Below the new way to write the above same function

```
function compose(...fns) {
  return function (...value) {
    return fns.reduce((a, b) => b(a), value);
  };
}

const composeAll =
  (...fns) =>
  (...val) =>
    fns.reduce((a, b) => b(a), val);
```

Currying in js:



Famous interview question related to Currying
Is **Infinite Recursive currying** and **Function currying and Non currying same function.**

Recursive Currying:

```
function sum(a) {
  return function (b) {
    if (!b) {
      return a;
    }
    return sum(a + b);
  };
}
console.log(sum(1)(2)(3)(4)(5)(6)()); //21
```

Approach for currying and Non currying same function:
The approach could be to check the function arguments length if its 1 solve it as a curring function else solve it as a normal function.

```
// Curried version of add function
function add(...args) {
    console.log("arguments: ",args)
  // Store the accumulated sum in a closure variable
  let sum = args.reduce((acc, val) => acc + val, 0);
  // Inner function to handle the currying
  function innerAdd(...nextArgs) {
    // If there are no arguments, return the accumulated sum
    if (nextArgs.length === 0) {
      return sum;
    }
    // Otherwise, update the accumulated sum and return a new function
    sum += nextArgs.reduce((acc, val) => acc + val, 0);
```

```
    return innerAdd;
  }
  // Return the inner function to enable currying
  return innerAdd;
}

// Usage examples
console.log(add(1)(2)(3)());          // Output: 6
console.log(add(1, 2)(3)());          // Output: 6
console.log(add(1, 2)(3, 4)());       // Output: 10
console.log(add(1)(2)(3)(4, 5)());    // Output: 15
```

Output:

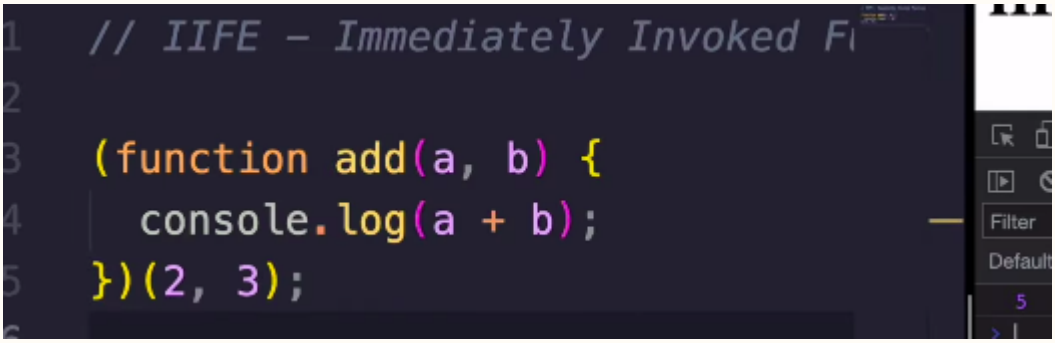
```
node /tmp/pJKnC7G0ep.js
arguments:  [ 1 ]
6
arguments:  [ 1, 2 ]
6
arguments:  [ 1, 2 ]
10
arguments:  [ 1 ]
15
```

Promisification:

It is the conversion of a function that accepts the callback and returns the promise.
Nothing to see much it's a rare concept .

IFFE:

Immediately invoked function



Advantages:
We can initialize a function in iffe , so that it wont be globally present
We can do async functions
WE can use IFFE to create Private and Public Variables

```
const myModule = (function() {
  const privateVar = 'This is private';

  const publicVar = 'This is public';

  function publicFunction() {
    console.log('Public function called');
  }

  return {
    publicVar,
    publicFunction
  };
})();
console.log(myModule.publicVar);
myModule.publicFunction();
```

Here balance is the private arable that i cannot access.

Reconciliation:

1. createRoot method in main.js or app.js : basically createRoot function is responsible for creating virtual Dom .
2. Now A thing comes in mind that an api call resolves and then instantly another api call resolves OR there are frequent changes so its better to ignore the intermediate updates and do only the last update. For which React team designs **Fiber.**
3. What react uses to update Virtual DOM is Fiber
4. Fiber helps in hydration
5. Diffing algorithm just compare both trees.

Signals:

Need to use Signals in React ?

1. It changes only those components that uses the value of signal
2. It does not require to follow the hooks syntax means it can be used on top of component and elsewhere too
3. It has its own useEffect

4. E.g in useState if th main state is in parent and its value is used in child so once the value updates parent updates so does its all childs but when we use signal it only updates that child who is using the value of that signal