Episode 1

- Everything in JS happens within an execution context
- Variable environment → Memory (key:value) | Code component → Thread of execution → Executes code one line at a time
- JS is a synchronous single threaded language
  - Synchronous → In a specific order
  - single threaded → Can execute one command at a time

Episode 2: How JS code is executed

* Global Execution Context is created in 2 phases
  1) Memory Creation phase → variable: undefined, function: function-body (whole code of function)
  2) Code execution phase
  The GEC is pushed on to the call stack
* When a function is called, a new execution context is created for the function (Mem creation phase, code exec phase) and is pushed on to the call stack. After function completes, execution context of the function is deleted

Episode 3: Hoisting in JS

* Hoisting is nothing but trying to access a variable even before it is defined / declared. Only declaration is hoisted and not initialization.
* Reason → Memory creation phase.
* Arrow functions and function expression just behaves like a variable during memory creation phase and is initialised to undefined.

Episode 4 : How function works

* For every function call, a new execution context is created and after function finished execution, it is deleted.
* Local variable > Global variable.

Episode 5: Shortest JS program:

* Shortest JS program → Empty js file
* window → a global object that is created along with the global execution context.
* JS engine in chrome → V8
* Global object created in chrome is called as window. So there are different JS engines, and in each JS engine the global object may be called with a different name.
* At global level, **this = window object**

Episode 6: Undefined V/S not defined

* Undefined → Undefined is a placeholder which is given to variables at memory creation phase. It means that the variable is not yet initialised.
* not defined → Means memory is not allocated for it.
* JS is a **loosely typed language**. That means that it doesnot attach any variable to a data type.

```
var x = 10;
x = "str";
x = true;
```

VALID

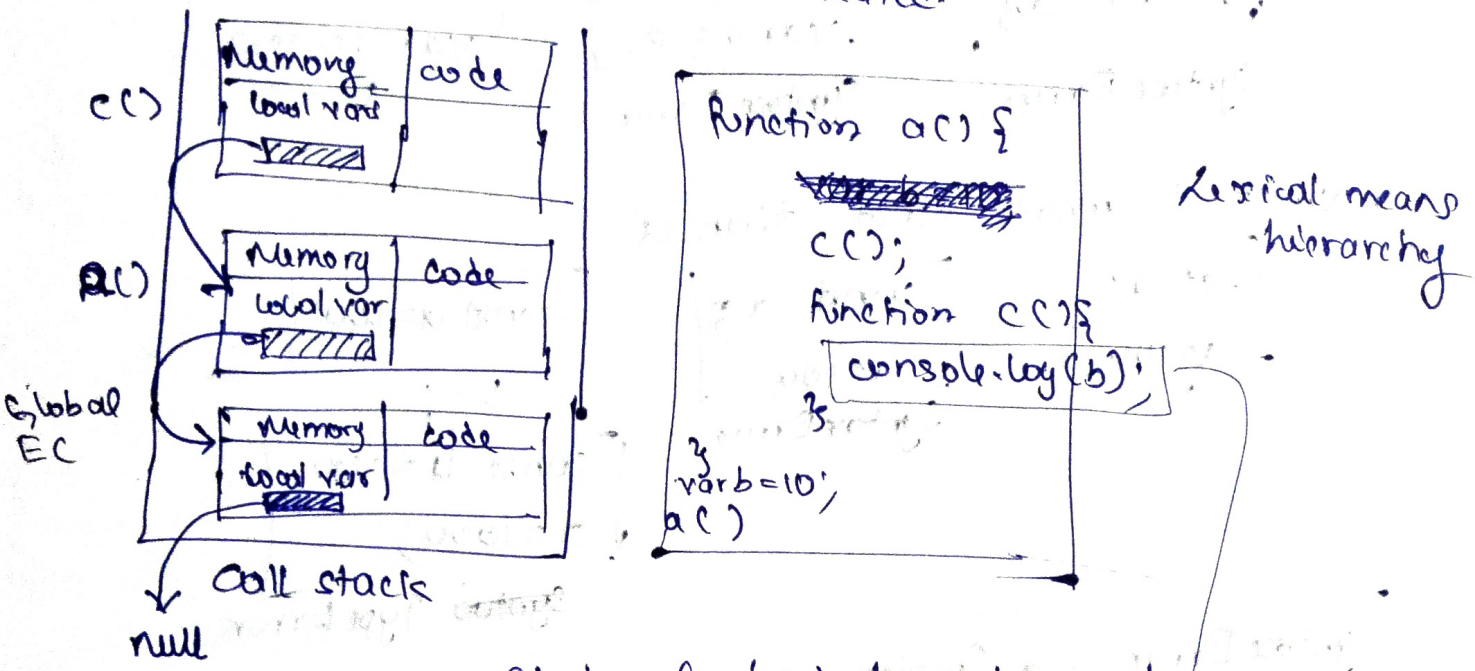
also called **weakly typed language**

Episode 7: Scope Chain & Lexical environment

* **Scope** → It means where you can access a variable or function

* **Lexical Environment:** local memory + Lexical environment of parent

Whenever an execution context is created a lexical environment is also created.



* **Scope chaining** :- 1) Check in local memory, if not present
2) Checks lexical parent ie. a()
3) If not present in a(), check in lexical parent ie. GEC.

Episode 8: let v/s const v/s var

- **let & const** → cannot access the variable until it is initialised. let and const are stored in a special place in memory and cannot be accessed until it is initialised. [special zone → Temporal Dead Zone]
- **temporal dead zone** → Time between declaration of let and const and its initialization. ie between hoisting and initialization

eg.
```
console.log(a);
let a = 10;
var b = 20;
```
→ ReferenceError: cannot access before initialization.

ReferenceError

* let is strict than var

```
let a = 10;
let a = 100;
```
SyntaxError

```
let a = 10;
var a = 10;
```
SyntaxError

```
var a = 10;
var a = 100;
```
✓

* const is more strict than a

```
let a;
a = 100
```
✓

```
const a;
a = 100
```
SyntaxError

```
const a = 100;
```
✓

```
const a = 100;
a = 1000;
```
~~SyntaxError~~ TypeError

SyntaxError → Problem in syntax of code
TypeError → assigning wrong value
ReferenceError → variable cannot be accessed.

let & const → Scope = temporal dead zone until initialised/after initialization → Block scope
var → ~~[redacted]~~ function scope

Block

Episode 9: Scope and Shadowing

* Block ~~[redacted]~~ : Grouping multiple statements together in a place where JS expects one statement.

eg: `if (condition) statement;` → after if ~~[redacted]~~ condition, JS actually expects a single statement. If we have multiple statements, then we use { } to group the statements. `if (condition) { ... }`

* **Block Scope** : what all variables we can access within the block.

* **Shadowing in JS**

```
var a = 100;
{
  var a=10;
  console.log(a)
} console.log(a)
```

o/p :- 10
10

var a=10; inside the block shadows the var a=100, outside the block.

We also see that a printed outside the block also gives 10 as o/p.

when we use var for shadowing it also modifies it because they both are pointing to same address

```
let b=100;
{
  let b=10;
  console.log(b);
}
console.log(b)
```

op- 10
100

let b=10 shadowed let b=100.
But didnot modify let b=100 because they are in different memory location.
let b=10 → block scope {}
let b=100 → has global scope

```
let a= 20
{
  var a =20
}
```

✗ ILLEGAL SHADOWING
because var a is trying to cross its boundary. as a is already present in global scope, var cannot shadow let here.

```
let a=20;
{
  let a=20;
}
```

✓

```
var a=20;
{
  let a=20;
}
```

✓

```
let a =20;
function c){
  var a=20;
}
```

→ var a is not crossing boundary as it is function scoped.

Scope rules and shadowing apply the same for even arrow function and function expressions.

Episode 10: Closures

Closure :- function + its lexical environment, i.e function bonded together with its lexical scope is called as Closure.

eg:

```
function x(){
    var a = 7;
    function y(){
        console.log(a);
    }
    return y;
}
var z = x();
console.log(z);
```

So the function y is returned by x() and the returned function is stored in z.

Now if we call z ie z(), it will still print 7 even though x doesnot exist

Why? Because when we returned y from x, not just the function, but its lexical scope was also returned.

```
z = x()
z()
  ↕
x()()
```

↓ z

Also when we returned y from the function x() what lexical scope of y is returned. and inside the lexical scope variables are not returned by value; reference to the variable is returned.

```
function x(){
    var a = 7;
    function y(){
        console.log(a);
    }
    a = 100;
    return y;
}
var z = x();
z();
```

o/p: 100

Applications / use of closures : setTimeout, memoize, currying ~~etc~~., data encapsulation & hiding etc

Episode 11: setTimeout + Closure

```
function x (){
    var i=1;
    setTimeout (function(){
        console.log(i);
    }, 3000);
    console.log ("Namaste JS");
}
x();
```

setTimeout (~~func~~ callback, time);

So when setTimeout is executed, the call back function forms a closure along with its lexical environment and the callback function is stored somewhere else in the browser. After the timeout, the function is pushed on to the call stack and executed.

Task → Print values 1, 2, 3, 4, 5 on a gap of 1 sec each.

A common approach which is wrong is

```
function x(){
    for(var i=1; i<=5; i++){
        setTimeout (function(){
            console.log (i);
        }, i*1000);
    }
    console.log ("Namaste JS");
}
x();
```

Why?

This is because of closure so when we called setTimeout every second, a closure of call back function is stored. So the closure contains i. But reference to i is stored and not its value. So all closures are pointing to same refrence of i. By the time the call back is executed i would already be 6.

op: Namaste JS
6
6
6
6
6

Solution → use let inside the loop. Since let has block scope, a new variable is created for every iteration. So i is a new copy all together for every iteration.

```
function x(){
    for( let i=0; i<=5; i++){
        setTimeout(()=>{
            console.log(i);
        }, i*1000);
    }
    console.log("Namaste JS");
}
x();
```

Output

```
Namaste JS
1
2
3
4
5
```

So each closure contains a different variable i.

How to do it without let?

Sol → Somehow, we have to give a new copy of i to closure ie call back function and we also know that var is function scoped. So the only solution is to enclose the setTimeout within another function and pass i as an argument to it.

```
function x(){
    for(var i=0; i<=5; i++){
        function close(x){
            setTimeout(function(){
                console.log(x);
            }, x*1000);
        }
        close(i);
    }
}
x();
```

So now x is a new copy every time.

Data abstraction & encapsulation using closures

Ep12: Intro

```
var counter = 0;
function increment () {
    counter++;
}
increment();
```

Here, counter variable is global and anybody can manipulate it.

↓ use closure to hide

```
function counter () {
    var cnt = 0;
    function increment () {
        cnt++;
    }
    return increment;
}
counter()()
```

So here we achieved data hiding - variable counter can be accessed by only the increment function i.e only inside the counter function.

↓ add decrement functionality

```
function counter () {
    var cnt = 0;
    function increment () {
        cnt++;
    }
    function decrement () {
        cnt--;
    }
    return { increment, decrement }
}
var counter1 = counter();
counter1.increment()
counter1.decrement()
```

Not optimal

Constructor function

```
function Counter () {
    var cnt = 0;
    this.increment = () => {
        cnt++;
    }
    this.decrement = () => {
        cnt--;
    }
}
var counter1 = new Counter();
counter1.increment()
counter1.decrement()
```

- **Disadvantage of closure**

1) Memory usage because closure's consume a lot of memory
2) Closed variables are not garbage collected till program completes.

- **Garbage Collector**

It is a software in the JS engine which frees up the unutilized memory.

- **Garbage collector with closures**

```
function a(){
    var x = 0;
    return function(){
        console.log(x);
    }
}
var z = a();
```

When the program is executed, coz the returned function forms a closure ⇒ Garbage collector will not free up memory of x and the function

- **Smart garbage collector**

```
function a(){
    var x = 0, y = 0;
    return function b(){
        console.log(x);
    }
}
var z = a();
```

Even though a closure is formed, the variable y=0 is garbage collected because it is not used within function b. → Smart garbage collection.

Episode 13: First Class Function

1) function statement → Normal func declaration function a(){ ... }

2) Function Expression → function assigned to a variable
→ var b = function a(){ ... }

- Diff b/w function statement and expression → Hoisting

3) Function declaration → Another name for function for function statement.

4) Anonymous function → function without a name.
→ Anonymous function donot have their own identity. They can only be used while returning a function or passed as an argument ie. IT IS USED WHERE FUNCTIONS ARE USED AS VALUES.
Eg: Function expression, call back func etc.

5) Named Function expression → Function expression in which we name the anonymous function.

Eg:
```
var b = function xyz (){
            console.log("B");
        };
```

xyz() → Invalid → xyz can only be used inside the function only, xyz acts like local variable.
b() → valid

6) First class function → Ability to use functions as values, ie functions passed as arguments to another function or functions returned by another function or function assigned to a variable. also called First class citizens

7) Arrow function → var a = () => { ... }

Ep 14: Callback functions

* Callback function :- function passed as an arguement to another function.

Call stack → Main thread of JS

* If any operation blocks the call stack → called → Blocking the main thread

Eg: ~~[illegible]~~ Say an operation x is a very heavy operation and takes a lot of time (say 20s). So during that time, JS wont be able to execute any other function and everything is blocked. sol → use async operation

Callback is heart of async function

- Event listener

element.addEventListener("event", callback);

What this does is it attaches the callback function to the event. the callback function is stored somewhere. When the event occurs, the callback function is pushed onto the call stack and executed.

- Event listener + Closure

Task: Display the no of times button is clicked.

~~1) use global variable~~

~~var count = 0;~~

~~document.querySelector('button').addEventListener(~~
~~"click", () => {console.log(count)});~~

1) To use global variable

```
var count = 0;
document.querySelector('button').addEventListener('click', () => {
    count++;
    console.log(count);
});
```

problem → count variable can be accessed by any one.

2) To use closure.

```
function attachEventListener() {
    var count = 0;
    document.querySelector("button").addEventListener("click",
    () => { count++; console.log(count); });
}
attachEventListener();
```

Why remove event listeners?

* Event listeners are heavy ie it takes alot of memory
* Event listener forms a closure as seen above and WKT closures are not garbage collected.

Ep 15: Event loop
Ep 16: JS engine
NOT READ

Higher Order Functions

Functions that take fuction as an arguement or return a function are called as **higher order functions**

Task: Write the logic to find area, diameter and circumference of circles whose radius if given in an array

```
const radius = [3, 1, 2, 4];

const calculateArea = function (radius) {
  const output = [];
  for (let i = 0; i < radius.length; i++) {
    output.push(Math.PI * radius[i] * radius[i]);
  }
  return output;
};

console.log(calculateArea(radius));

const caluculateCircumference = function (radius) {
  const output = [];
  for (let i = 0; i < radius.length; i++) {
    output.push(2 * Math.PI * radius[i]);
  }
  return output;
};

console.log(caluculateCircumference(radius));

const caluculateDiameter = function (radius) {
  const output = [];
  for (let i = 0; i < radius.length; i++) {
    output.push(2 * radius[i]);
  }
  return output;
};

console.log(caluculateDiameter(radi
```

```
const radius = [3, 1, 2, 4];

const area = function (radius) {
  return Math.PI * radius * radius;
};

const cicumference = function (radius) {
  return 2 * Math.PI * radius;
};

const diameter = function (radius) {
  return 2 * radius;
};

const calculate = function (radius, logic) {
  const output = [];
  for (let i = 0; i < radius.length; i++) {
    output.push(logic(radius[i]));
  }
  return output;
};

console.log(calculate(radius, area));
console.log(calculate(radius, cicumference));
console.log(calculate(radius, diameter));
```

Problem: Code repetetion

If we want to use our custom functions like the built in functions ie map, forEach etc we do this

```
Array.prototype.calculate = function (logic) {
  const output = [];
  for (let i = 0; i < arr.length; i++) {
    output.push(logic(arr[i]));
  }
  return output;
};
```

Now we can use calculate function with any array: **arr.calculate(callback);**

```
console.log(radius.calculate(area));
```

BY SHIVKARAMAN

Array Methods

1. for each

```
arrayname.foreach(callback func)
arranName.foreach( (data, index) => {
        console.log(data, index);
});
```

2. filter -> Creates a new array with items for which the callback func returns true. The original array is unchanged

```
const users = [
  {name: 'shaun', premium: true},
  {name: 'yoshi', premium: false},
  {name: 'maric', premium: false},
  {name: 'chun-li', premium: true}
];

const premiumUsers = users.filter(user => {
  return user.premium;
});

console.log(premiumUsers);
```

Creates copy of object

```
▼ (2) [{…}, {…}]
  ▶ 0: {name: "shaun", premium: true}
  ▶ 1: {name: "chun-li", premium: true}
    length: 2
  ▶ __proto__: Array(0)
```

3. map -> creates a new array whose elements depend on the original array This mmethod can alter the original array too

```
const products = [
  {name: 'gold star', price: 20},
  {name: 'mushroom', price: 40},
  {name: 'green shells', price: 30},
  {name: 'banana skin', price: 10},
  {name: 'red shells', price: 50}
];

const saleProducts = products.map((product) => {
  if(product.price > 30){
    return {name: product.name, price: product.price / 2};
  } else {
    return product;
  }
});
```

```
product.price = product.price / 2;
```

This would alter the price in original array

This is object itself and not a copy

```
▼ (5) [{…}, {…}, {…}, {…}, {…}]
  ▶ 0: {name: "gold star", price: 20}
  ▶ 1: {name: "mushroom", price: 20}
  ▶ 2: {name: "green shells", price: 30}
  ▶ 3: {name: "banana skin", price: 10}
  ▶ 4: {name: "red shells", price: 25}
    length: 5
  ▶ __proto__: Array(0)
```

4. reduce -> Iterates over the array and returns a value depending upon the callback func
value-> can be of any data type

```
const scores = [10, 20, 60, 40, 70, 90, 30];

const result = scores.reduce((acc, curr) => {
  if(curr > 50){
    acc++;
  }
  return acc;
}, 0);
```

Accumulator

Current value (element of array)

Initial value of acc

5. find -> returns the first array element that satisfies the condition

```
const scores = [10, 5, 0, 40, 60, 10, 20, 70];

const firstHighScore = scores.find((score) => {
  return score > 50;
});
```

6. Sort -> sorts the array

arrayName.sort() -> this sorts elements if they are of primitive type, if the array is array of objects we need to pass in a callback function which helps sort the array -> this func looks only at the first part of any element

```
const players = [
  {name: 'mario', score: 20},
  {name: 'luigi', score: 10},
  {name: 'chun-li', score: 50},
  {name: 'yoshi', score: 30},
  {name: 'shaun', score: 70}
];

players.sort((a,b) => {
  if(a.score > b.score){
    return -1;
  } else if (b.score > a.score){
    return 1;
  } else {
    return 0;
  }
});
```

return type
1. +ve -> swap
2. 0 -> no change
3. -ve number -> no change

```
[10, 50, 20, 5, 35, 70, 45];
```

scores.sort(()

```
[10, 20, 35, 45, 5, 50, 70]
```

solution

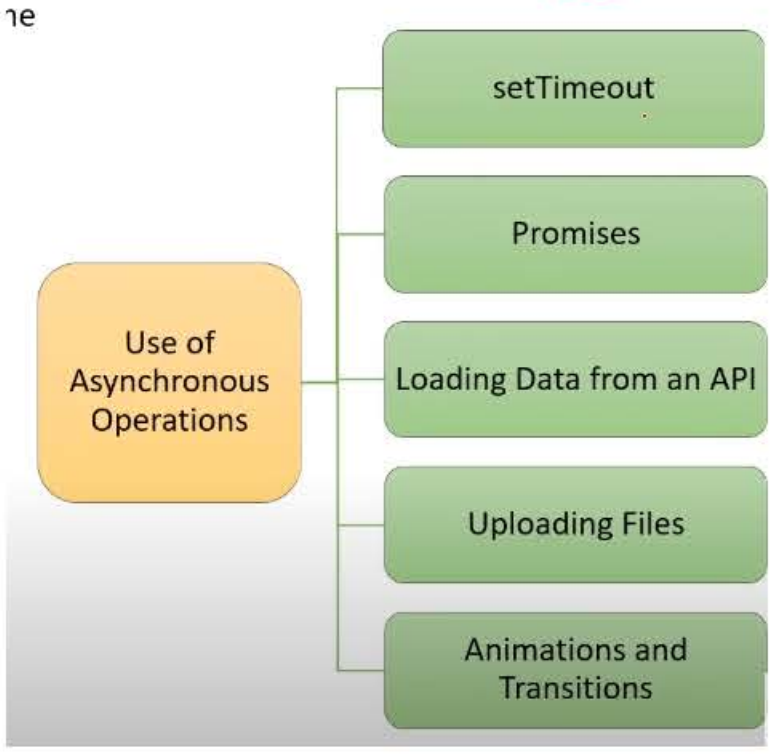
```
scores.sort((a,b) => b - a);
```

```
players.sort((a,b) => {
  return b.scor - a.score
})
```

SIMPLIFIED VERSION

Q. What are asynchronous operations in JavaScript? V. IMP.

- Asynchronous operations are operations that **do not block** the execution of the code.
- setTimeout is the asynchronous function which executes callback function after a specific period of time.



Callback Hell

Callback hell in JavaScript refers to a situation where multiple nested callbacks are used, leading to complex and hard-to-read code. This occurs when asynchronous operations are heavily nested within each other, making the code difficult to manage and understand.

javascript Save to grepper Copy code

```javascript
getUser(userId, function(user) {
    getPosts(user.id, function(posts) {
        getComments(posts[0].id, function(comments) {
            // Do something with comments
        });
    });
});
```

In this example, each function relies on the result of the previous one, and they're deeply nested. This can make the code hard to follow, debug, and maintain. Promises or async/await can help alleviate callback hell.

Q. What are promises in JavaScript? V. IMP.

- Important points about promises:

1. Promises in JavaScript are a way to handle **asynchronous operations.**
2. A promise represents a value that may not be available yet but will be available at some point in the **future**.
3. Promise can be in one of three states: **pending, resolved, or rejected.**

```javascript
const promise = new Promise((resolve, reject) => {
  // Perform asynchronous operation
  // If successful, call resolve(value)
  // If failed, call reject(error)
});
```

Promises in JS

Nesting of callback function in callback hell was messy and hard to read.
Instead we use PROMISES

```javascript
const getSomething = () => {

  return new Promise((resolve, reject) => {
    // fetch something
    //resolve('some data');
    reject('some error');
  });

};

getSomething().then((data) => {
  console.log(data);
}, (err) => {
  console.log(err);
});

// Better way of doing it

getSomething().then(data => {
  console.log(data);
}).catch(err => {
  console.log(err);
})
```

A normal function that returns a **Promise**

These two are functions that are built into the promise API

On Success

On Failure

Fired when resolve() is called in promise

Fired when reject() is called in promise

Concatenation involving different data types

1. **String + Number:**

```javascript
var result = "5" + 3;
console.log(result);
```

Output: `"53"`

Explanation: When a string and a number are concatenated using the `+` operator, JavaScript converts the number to a string and performs string concatenation.

1. **Number + String:**

```javascript
var result = 3 + "5";
console.log(result);
```

Output: `"35"`

Explanation: Similar to the previous example, the number is converted to a string, and string concatenation is performed.

1. **Number + Number + String:**

```javascript
var result = 2 + 2 + "5";
console.log(result);
```

Output: `"45"`

Explanation: The first two numbers are added (`2 + 2` results in `4`), and then the result is concatenated with the string `"5"`.

1. **String + Number + Number:**

```javascript
var result = "2" + 2 + 3;
console.log(result);
```

Output: `"223"`

Explanation: The first string is concatenated with the first number (`"2" + 2` results in `"22"`), and then the result is concatenated with the second number.

1. **Number + String + Number:**

```javascript
var result = 1 + "2" + 3;
console.log(result);
```

Output: `"123"`

Explanation: The first number is concatenated with the first string (`1 + "2"` results in `"12"`), and then the result is concatenated with the second number.

These examples demonstrate the behavior of JavaScript's type coercion when performing operations involving different data types. The `+` operator is used for both addition and string concatenation, depending on the types of the operands.

BY SHIVKARAMAN

1. **Number + Boolean:**

```javascript
var result = 5 + true;
console.log(result);
```

Output: `6`

Explanation: In JavaScript, `true` is treated as `1` when used in a numeric context. Therefore, `5 + true` equals `5 + 1`, resulting in `6`.

1. **String + Boolean:**

```javascript
var result = "hello" + false;
console.log(result);
```

Output: `"hellofalse"`

Explanation: The boolean `false` is implicitly converted to a string and concatenated with the string `"hello"`.

1. **Number + Undefined:**

```javascript
var result = 10 + undefined;
console.log(result);
```

Output: `NaN` (Not-a-Number)

Explanation: When trying to perform arithmetic operations involving `undefined`, JavaScript converts `undefined` to `NaN`.

1. **String + Null:**

```javascript
var result = "world" + null;
console.log(result);
```

Output: `"worldnull"`

Explanation: The `null` value is implicitly converted to a string and concatenated with the string `"world"`.

1. **Boolean + String:**

```javascript
var result = true + "123";
console.log(result);
```

Output: `"true123"`

Explanation: The boolean `true` is implicitly converted to a string and concatenated with the string `"123"`.

These examples further showcase how JavaScript handles type conversions when different data types are involved in operations. It's essential to understand these behaviors, especially when dealing with mixed data types in your code.

How 'this' keyword behaves in different context

1. **Global Context:**

```javascript
console.log(this === window);
```

Output: `true`

Explanation: In the global context (outside any function), `this` refers to the global object, which is `window` in a browser environment.

1. **Function Context:**

```javascript
function foo() {
    console.log(this === window);
}
foo();
```

Output: `true`

Explanation: When a function is called in the global context, `this` still refers to the global object (`window`).

1. **Function in Strict Mode:**

```javascript
"use strict";
function bar() {
    console.log(this === undefined);
}
bar();
```

Output: `true`

Explanation: In strict mode, when a function is called without a specific `this` value, `this` is `undefined`.

1. **Object Method:**

```javascript
var person = {
    name: "Alice",
    sayHi: function() {
        console.log("Hi, my name is " + this.name);
    }
};
person.sayHi();
```

Output: `"Hi, my name is Alice"`

Explanation: In an object method, `this` refers to the object itself. So, `this.name` refers to the `name` property of the `person` object.

1. **Arrow Functions:**

```javascript
var obj = {
    value: 42,
    getValue: () => {
        console.log(this.value);
    }
};
obj.getValue();
```

Output: `undefined`

Explanation: Arrow functions do not have their own `this` context. They inherit `this` from their containing scope (in this case, the global scope), where `value` is not defined.

1. **Constructor Function:**

```javascript
function Car(make) {
    this.make = make;
}
var myCar = new Car("Toyota");
console.log(myCar.make);
```

Output: `"Toyota"`

Explanation: Inside a constructor function, `this` refers to the newly created instance of the object.

1. **Event Handlers:**

```html
<button id="myButton">Click me</button>
<script>
document.getElementById("myButton").addEventListener("click", function() {
    console.log(this === document.getElementById("myButton"));
});
</script>
```

Output: `true`

Explanation: In an event handler, `this` refers to the DOM element that triggered the event (`<button>` in this case).

1. **Explicit `this` Binding:**

```javascript
function greet() {
    console.log("Hello, " + this.name);
}
var person1 = { name: "Alice" };
var person2 = { name: "Bob" };
greet.call(person1);
greet.apply(person2);
```

Output: `"Hello, Alice"`
`"Hello, Bob"`

Explanation: Using `call()` and `apply()` methods, you can explicitly set the value of `this` in a function to a specific object.