**Backend**

1. Aaa
2. 333
3. 333

**Frontend**

1. **What is NaN? how to make NaN value happen? how to check value is NaN? (2 ways) difference? Is a string NaN?**

Not a Number. NaN is the value that's returned when you attempt to perform an operation that's supposed to return a numerical result, but the operation doesn't make sense.

var ans = “hello” / 2;

* console.log(total === total);*//Output: false. Only NaN != NaN*
* console.log(Number.isNaN(total));*//Output: true. Check is NaN or not.*

In JavaScript, a string is not NaN by itself, but if you try to perform a mathematical operation using a string that cannot be converted to a number, JavaScript will return NaN (Not a Number).

1. **== vs ===**

do not use == to compare in js. ===: it will compare both value and type; == : only compares value.

1. **js primitive types**

ES5 js primitive types: string, boolean, number, null, undefine

1. **what is enclosure**

Closure is talking about function inside of another function. When outer function is executed, js will create a closure object to store all the outer function's local variables being referenced in the returned inner function.

* Pros for closure: Avoid creating a global variable, to create a private variable
* Cons for closure: Memory leak may happen.

1. **what is hoisting? Why hoisting is good?** **when hoisting is bad/not wanted?**

js will move variable declarations and function declarations to the top of the CURRENT scope.

It allows us to call functions before they appear in our code. This gives us the freedom to structure our code in the way that makes the most sense to us, without worrying about the order of function declarations.

If we try to access a variable before it's declared and is unaware that JavaScript hoists variable declarations, we might be surprised by the undefined value; only the declarations are hoisted, not initializations; let and const declarations are hoisted but not initialized to undefined like var. Trying to access them before declaration will result in a Reference Error.

1. **what is js object (dont say primitive types)? ES5 build-in Objects**

* key/value pair: Type of key is a string, value can be anything. We can omit the quotes on key if it doesn't contain special characters.(&, \_) are not special, so they can be used without ''.
* function
* Date
* Regular expression

Object, Function, array, Date, Math, JSON, Error.

1. **js inside steps to access: var a = {}; console.log(a.b);**

* A new, empty object is created and assigned to the variable a. {} is shorthand for "new Object()".
* Then, console.log(a.b); is executed. It tries to access the property b of object a.
* Since a is an empty object, it does not have a property named b. Therefore, a.b is undefined.
* console.log(undefined); is executed, which logs undefined to the console.

So, in the end, the output of the code will be undefined because a does not have a property named b.

1. **how to handle/stop bubbling?**

* Event bubbling can be stopped using the stopPropagation() method. In this case, if a click event is triggered on the specified element, the event will not bubble up to any parent elements.

element.addEventListener('click', function(event) {

event.stopPropagation();

});

* stopImmediatePropagation() will not only stop the event from bubbling up, but also prevent any remaining event handlers on the same element from being executed.

element.addEventListener('click', function(event) {

event.stopImmediatePropagation();

});

1. **do you know fetchAPI? js build-in way to send GET request**

The Fetch API is a modern, promise-based mechanism included in the Web APIs exposed by modern browsers for making asynchronous HTTP requests, similar to the older XMLHttpRequest. The Fetch API is more powerful and flexible, also has a more modern, promise-based API that is often considered easier to understand and use. Fetch API is built into modern browsers, but it's not available in older browsers, and some features are not yet fully supported in all modern browsers.

fetch('https://api.example.com/data', {

method: 'GET',

headers: 'Content-Type': 'application/json'

})

.then(response => response.json())

.then(data => console.log(data))

.catch(error => console.error('Error:', error));

* fetch('https://api.example.com/data') makes a GET request to the specified URL. fetch() returns a Promise that resolves to the Response object representing the response to the request.
* .then(response => response.json()) takes the Response object and reads its body as JSON, returning another Promise that resolves to the JSON data.
* .then(data => console.log(data)) logs the JSON data to the console.
* .catch(error => console.error('Error:', error)) catches any errors that may occur during any of the above steps.

1. **how to check the request is success**

In the Fetch API, you can check if a request was successful by examining the ok property of the Response object.

fetch('https://api.example.com/data')

.then(response => {

if (!response.ok) {

throw new Error('HTTP error, status = ' + response.status);

}

return response.json();

})

.then(data => console.log(data))

.catch(error => console.error(error));

When using XMLHttpRequest, you would typically check the readyState and status properties in the onreadystatechange event handler

var xhr = new XMLHttpRequest();

xhr.onreadystatechange = function() {

if (xhr.readyState == 4 && xhr.status == 200) {

console.log(xhr.responseText);

}

}

xhr.open('GET', 'https://api.example.com/data', true);

xhr.send();

1. **what’s async and await**

async and await are extensions of promises in JavaScript that provide a simpler and cleaner syntax to work with asynchronous operations, like fetching data from a server, reading files, or timeouts.

* async is a keyword that you put in front of a function declaration to turn it into an async function. An async function is a function that knows how to expect the possibility of the await keyword being used to invoke asynchronous code.
* await is a keyword that you use in an async function to pause the execution of the function and wait for a promise to resolve or reject. await only works inside async functions.

1. **write aschyncronized in ES5**

Asynchronous operations were typically managed with callbacks or Promises. A promise is an object that wraps an async operation that may succeed or fail.

Promise has 3 States:

1. pending: when a promise is created, it's in a pending state

2. resolved: when async operation successfully finished... .then() to handle resolved promise

3. rejected: when the async operation failed, ... .catch() to handle a rejected promise

*// A function that returns a Promise*

function promiseFunction() {

return new Promise(function(resolve, reject) {

var condition = true; // This can be any condition, for this example, it's just set to true

if(condition) {

resolve('Promise is resolved successfully.');

} else {

reject('Promise is rejected');

}

});

}

*// Using the Promise*

promiseFunction().then(function(successMessage) {

console.log('Success: ' + successMessage);

}).catch(function(errorMessage) {

console.log('Error: ' + errorMessage);

});

1. **how to define user type in typescript**

interface User {

id: number;

name: string;

email: string;

}

let user: User = {

id: 1,

name: 'John Doe',

email: 'john@example.com'

};

1. **how to send request with AXIOS library**

Axios is a popular, promise-based HTTP client that sports an easy-to-use API and can be used in both the browser and Node.js.

const axios = require('axios');

axios.get('https://api.example.com/data')

.then(function (response) {

*// handle success*

console.log(response.data);

})

.catch(function (error) {

*// handle error*

console.log(error);

});

1. **event loop vs event propagation**

The event loop and event propagation are two separate concepts - the event loop is about how JavaScript handles tasks and asynchronous operations, while event propagation is about how events travel through the DOM.

Event Loop: In JavaScript, the event loop is a mechanism that constantly checks the call stack to see if it's empty. When the call stack is empty, it takes the first task from the task queue (also known as the event queue) and pushes it to the call stack. This process continues in a loop, hence the name event loop.

Event Propagation: Event propagation is a concept in the Document Object Model (DOM) that defines how events propagate or travel through the document tree (from the target element to the root and vice versa).

* Capturing phase: The event starts from the root and goes down to the target element.
* Target phase: The event has reached the target element.
* Bubbling phase: The event bubbles up from the target element back to the root.

You can control event propagation using methods like stopPropagation() and stopImmediatePropagation(). The former stops the event from continuing to the rest of the elements in the propagation path, while the latter stops the remaining event handlers on the same element from being executed.

1. **how do child-to-parent communicate in angular?**

First, in the child component, we use @Output() decorator on an EventEmitter, we call emit() method to send the event that carries the data to parent, child -> parent:

@Component({

selector: 'app-child',

template: `<button (click)="sendMessage()">Send Message</button>`

})

export class ChildComponent {

@Output() messageEvent = new EventEmitter<string>();

sendMessage() {

this.messageEvent.emit('Hello from Child Component!');

}

}

Then, in the parent component's template, use event binding to listen to this messageEvent and call a function to handle this event:

@Component({

selector: 'app-parent',

template: `

<app-child (messageEvent)="receiveMessage($event)"></app-child>

<p>{{message}}</p>

`

})

export class ParentComponent {

message: string = '';

receiveMessage($event: string) {

this.message = $event;

}

}

**Git**

1. **git stash, git reflog, git rebase, git merge vs git rebase, git cherry-pick, git tag**

* git stash: This command saves changes that you have made but do not want to commit immediately. You can apply the stashed changes later.
* git reflog: This command shows a log of where your HEAD and branch references have been. It can be useful to recover changes lost by mistaken force pushes or hard resets.
* git rebase: This command moves or combines a sequence of commits to a new base commit. It is useful for maintaining a linear project history.
* git merge vs git rebase: git merge takes the contents of a source branch and integrates it with the target branch. git rebase moves or combines the commits to a new base commit. merge keeps the history of commits as it is whereas rebase provides a much cleaner project history.
* git cherry-pick: This command applies the changes introduced by some existing commits. The changes are applied as a new commit in the current branch.
* git tag: This command is used to mark specific points in history as being important. Typically people use this functionality to mark release points (v1.0, v2.0, etc.).