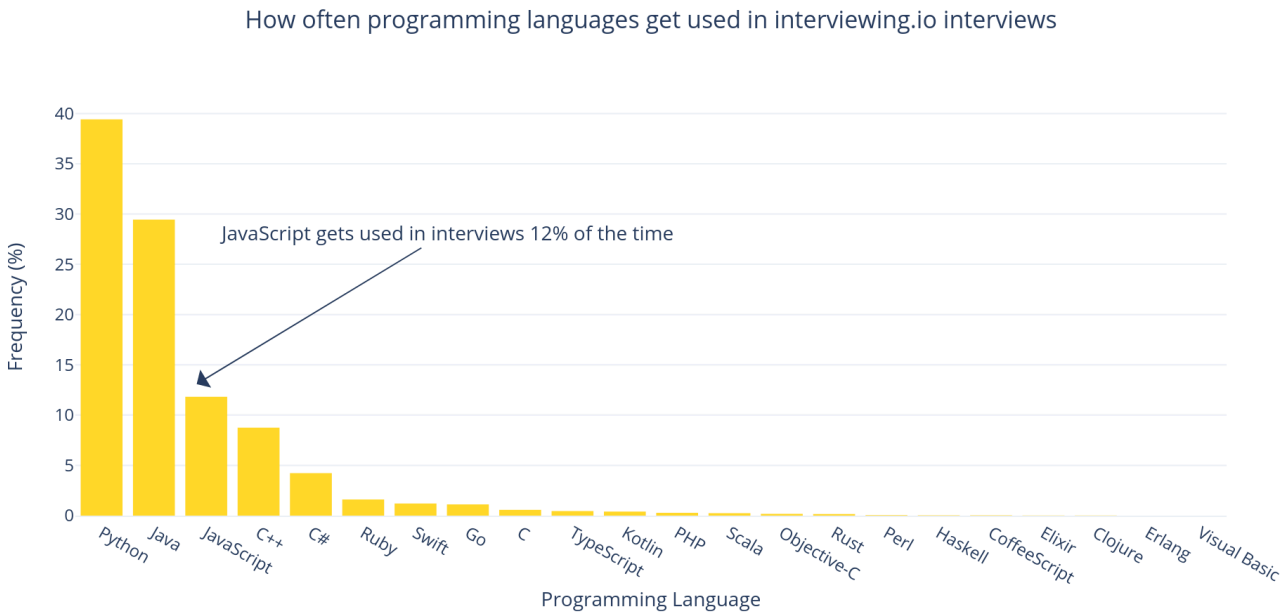
How is JavaScript Used in Technical Interviews?

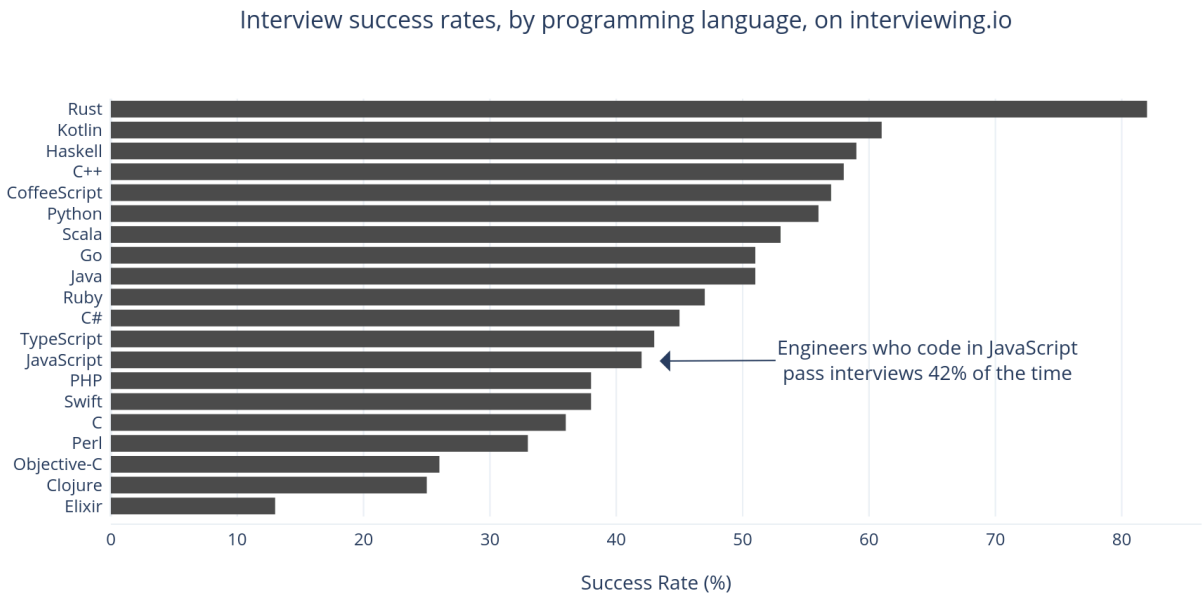
By Jai Pandya | Published: July 20, 2023

JavaScript Interview Stats

The selection of a programming language can often be a decisive factor in the arena of technical interviews. Based on the data collected from over 100k interviews on our platform, JavaScript emerged as the third most popular language of choice, being used in 12% of all technical interviews. JavaScript ranks just below popular programming languages like Python and Java, yet, when chosen, JavaScript delivers a pass rate of 42% in the interviews, showcasing its significance and effectiveness.

Below is a distribution of programming languages and their popularity in technical interviews as well as success rates in interviews by language.





One of the reasons for JavaScript's popularity is its versatility. It works on both the backend and frontend, making it a valuable tool for full-stack developers. Using a single language across multiple layers of an application can enhance productivity and improve workflow.

Of course, it wouldn't be a discussion about JavaScript without acknowledging the love-hate relationship it enjoys with programmers. JavaScript can be likened to that old friend you may sometimes find eccentric and unpredictable, yet they always surprise you with their resourcefulness and charm. No matter how you feel about it, one thing is sure — you can't ignore JavaScript.

Why JavaScript is Important for Technical Interviews

Unique Qualities of JavaScript

JavaScript stands out due to its adaptability and prevalence in web development. It's a flexible tool that handles both object-oriented and functional programming styles, making it ideal for showcasing diverse problem-solving skills in interviews. Its role as the sole native language of web browsers puts it front and center in web development, offering a clear edge during interviews. Its use in front-end and back-end development, thanks to tools like Node.js and frameworks like React, makes it a full-stack developer's dream.

Industry Significance of JavaScript

JavaScript's importance goes beyond its technical merits—it's also a leading language in the industry. The 2023 Stack Overflow Developer Survey marks it as the most commonly used

language for the eleventh year in a row. GitHub's Octoverse Report 2022 also attests to the same fact, with JavaScript being the most used language on GitHub. This sustained industry demand ensures that JavaScript expertise remains highly valued in technical interviews.

JavaScript Idioms & Idiosyncrasies

JavaScript was developed by Brendan Eich in just ten days in the year 1995 while he was working at Netscape Communications. Over the years, JavaScript has undergone numerous changes and enhancements. ECMAScript (ES), the standardized language specification, has overseen these transformations. One of the most significant shifts came with the release of ES6 (also known as ES2015) in June 2015, which introduced new syntax and powerful features that transformed the way JavaScript code was written. Since then, new versions of the specification have been released yearly, with the latest being ES2023.

While the specifications have evolved, JavaScript's core principles have remained unchanged. It's a dynamic, weakly typed, prototype-based language that supports object-oriented, imperative, and declarative programming styles. Additionally, it's a single-threaded, non-blocking, asynchronous language that uses an event loop to handle concurrency. In recent years, the emergence of TypeScript - a statically typed superset of JavaScript - highlights the evolving nature of JavaScript, offering type safety and improved tooling. In this section we'll learn about JavaScript's idioms and idiosyncrasies, that make it special.

Single Threaded Event Loop & Asynchronous Behavior

JavaScript is single-threaded, meaning it can process one operation at a time in a single sequence, or thread, of operations. While this might seem limiting, especially considering that many programming languages use multi-threading, JavaScript leverages this single-threaded nature using an event loop mechanism to handle asynchronous operations efficiently.

JavaScript's single-threaded nature helps it avoid the complexities of multithreading while manipulating DOM tree, making it easier to learn and use. But since it can only process one operation at a time, a long-running operation can block the thread and hang the system, causing what is known as a "blocking" operation.

To overcome this, JavaScript uses an event-driven, non-blocking I/O model. It utilizes an event loop and a callback queue. When an asynchronous operation is encountered, it's offloaded to the browser's Web APIs, freeing up the main thread to continue executing other operations. The associated callback function is pushed into a task queue when the

asynchronous operation is completed. The event loop continually checks this queue and pushes any waiting callbacks back onto the main thread for execution as soon as it's free.

This unique design allows JavaScript to handle high I/O workloads efficiently without the complexity and potential issues of multi-threading, making it particularly well-suited for web development, where asynchronous operations like network requests, user interactions, and timers are common.

JavaScript

```
console.log("Fetching data...");

setTimeout(function() {
  console.log("Data fetch complete!");
}, 2000);

console.log("Waiting for data...");

// Output:
// Fetching data...
// Waiting for data...
// Data fetch complete!
```

JavaScript Execution Environment

The environments in which JavaScript runs, such as web browsers (Chrome, Safari, Firefox, etc.) or servers (Node.js, Deno, Bun, etc.), each provide unique features and behaviors. Although ECMAScript defines the standard specifications for JavaScript, not all environments implement these uniformly, leading to environment-specific quirks. For instance, a feature like the Fetch API, widely supported in modern web browsers, wasn't natively supported in Node.js until version 17.5 (with experimental flag). Therefore, understanding your JavaScript execution environment and its specific features is crucial for creating robust, cross-compatible code.

Type Coercion

As a weakly typed language, JavaScript can automatically convert values from one type to another, a behavior known as type coercion.

JavaScript

```
console.log(4 + "2"); // Output: "42"

let numStr = "42";
```

```
let num = +numStr; // '+' operator triggers type coercion.
console.log(num); // Output: 42 (a number, not a string)
```

This behavior of JavaScript may remind you of type casting seen in other languages. The key difference is that type casting (or type conversion) is explicitly done by the programmer, while type coercion is performed implicitly by the language. In JavaScript, it's important to understand when and how type coercion occurs to prevent unexpected outcomes.

Function Expressions

JavaScript treats functions as first-class objects so that they can be assigned to variables, stored in data structures, passed as arguments to other functions, and returned from other functions.

JavaScript

```
// function gets assigned to a variable
let calculateArea = function(radius) {
  return Math.PI * radius * radius;
};
```

Hoisting

Hoisting is a unique behavior of JavaScript where variable and function declarations are moved to the top of their containing scope during the compilation phase before the code has been executed.

JavaScript

```
greet(); // Output: Hello, Interviewing.io!

// Function declaration
function greet() {
  console.log('Hello, Interviewing.io!');
}
```

Closure

A closure is a function that has access to the variables of its outer function, even after the outer function has returned. This is possible because the inner function has access to the

outer function's scope, even after the outer function has finished executing. This helps create private variables and function factories.

JavaScript

```
function outerFunc() {
  let outerVar = 'I am outside!';
  function innerFunc() {
    console.log(outerVar);
  }
  return innerFunc;
}
let inner = outerFunc();
inner(); // Output: I am outside!
```

The `this` Keyword

In JavaScript, `this` is a special keyword that refers to the context in which a function is called. This can vary depending on how and where the function is invoked. In a method of an object, `this` refers to the object itself. In a simple function call, `this` refers to the global object (in non-strict mode) or is `undefined` (in strict mode).

JavaScript

```
const myObj = {
  value: 'Hello, World!',
  printValue: function() {
    console.log(this.value);
  }
};

myObj.printValue(); // Output: Hello, World!
```

Contrast this with Python, where the object context is passed explicitly as a parameter (`self`) to an instance method, and with Java, where `this` always refers to the current instance of the class.

Understanding the `this` keyword, and its context-dependent nature, is crucial for writing and debugging JavaScript code. It's also a source of frequent mistakes. We'll learn more about that in the next section.

Destructuring

Introduced in ES6, destructuring allows for quickly unpacking values from arrays or properties from objects. This can help simplify code and make it more readable.

JavaScript

```
let candidate = {
  name: "Alice",
  language: "JavaScript",
  experience: "3 years",
};

let { name, experience } = candidate;
console.log(name, experience); // Output: Alice, 3 years
```

Rest and Spread Operators

These operators provide convenient ways to handle collections of items and can often simplify the code written in an interview.

Spread: While destructuring 'unpacks' elements from an array or properties from an object, the spread operator takes it further by allowing you to expand or 'spread out' these elements or properties in a new context. It's helpful when you want to combine arrays or to use an array's values as function arguments.

JavaScript

```
let candidate = {
  name: "Alice",
  basicSkills: ["HTML", "CSS"]
};

let updatedCandidate = {
  ...candidate, // Using spread operator to copy properties from candidate object
  advancedSkills: ["JavaScript", "React"]
};

console.log(updatedCandidate);
/* Output:
{
  name: "Alice",
  basicSkills: ["HTML", "CSS"],
  advancedSkills: ["JavaScript", "React"]
}
*/
```

Rest: The Rest operator collects multiple elements and condenses them into a single array. It's used in function arguments to allow the function to accept any number of parameters.

JavaScript

```
function getCandidateDetails({ name, ...skills }) {
  console.log(`Candidate ${name} has the following skills:`);
  console.log(`Basic: ${skills.basicSkills}`);
  console.log(`Advanced: ${skills.advancedSkills}`);
}

getCandidateDetails(updatedCandidate);
/* Output:
Candidate Alice has the following skills:
Basic: HTML,CSS
Advanced: JavaScript,React
*/
```

Common JavaScript Interview Mistakes

In the context of interviews, a deep understanding of JavaScript is critical. There are some common pitfalls that candidates often fall into. Recognizing these mistakes can greatly enhance your interview performance and overall coding skills.

Improper Use of 'this' Keyword

The `this` keyword in JavaScript can be tricky, as its context depends on how and where it's called. Let's consider an example where you are iterating over an array of numbers to calculate their sum:

JavaScript

```
class ArraySum {
  constructor(numbers) {
    this.numbers = numbers;
    this.sum = 0;
  }

  calculateSum() {
    this.numbers.forEach(function(num) {
      this.sum += num;
    });
  }
}

let obj = new ArraySum([1, 2, 3]);
```

```
obj.calculateSum();
console.log(obj.sum); // NaN
```

Here, this inside the `forEach` callback doesn't refer to the `ArraySum` instance, but to the global object (`undefined` in strict mode). This results in `NaN` because `undefined +` number in JavaScript is `NaN` .

The issue can be fixed using an arrow function:

JavaScript

```
class ArraySum {
  constructor(numbers) {
    this.numbers = numbers;
    this.sum = 0;
  }

  calculateSum() {
    this.numbers.forEach(num => {
      this.sum += num;
    });
  }
}

let obj = new ArraySum([1, 2, 3]);
obj.calculateSum();
console.log(obj.sum); // 6
```

The arrow function doesn't have its own `this` context, it inherits it from the surrounding code. Now `this` within the `forEach` callback correctly refers to the `ArraySum` instance, leading to the correct sum of numbers.

Using Array as a Queue without Time Complexity Considerations

JavaScript has no built-in queue data structure. Using an array as a queue is common during data structure and algorithm questions. However, it can be computationally expensive. Suppose you're implementing a Breadth-First Search (BFS) on a graph in an interview; you might use an array as a queue to hold nodes:

JavaScript

```
function bfs(graph, startNode) {
  let queue = [];
  // enqueue operation
  queue.push(startNode);
```

```
  while(queue.length > 0) {
    // dequeue operation, O(n)
    let node = queue.shift();
    console.log(node.value);

    for(let child of node.children) {
      queue.push(child);
    }
  }
}
```

`Array.prototype.shift()` has a time complexity of `O(n)` because it re-indexes every remaining element in the array. This can be a major inefficiency for large arrays.

You should always let your interviewer know you know this limitation. If the interviewer insists, you should be able to implement it using a linked list. This will give you a time complexity of `O(1)` for both enqueue and dequeue operations.

Mistakes with Type Coercion and Equality (== and ===)

Understanding JavaScript's type coercion in comparison operations is crucial, especially when dealing with different data types. Let's look at a simple but confusing example:

JavaScript

```
let a = '0';
let b = 0;

console.log(a == b); // true
console.log(a === b); // false
```

In the first log statement, JavaScript coerces the string '0' to a number due to the `==` operator, resulting in true. In the second log statement, the `===` operator checks both value and type, hence '0' (string) and 0 (number) are not considered equal.

As a best practice, it is always recommended to use the `===` operator.

Not Writing Idiomatic JavaScript

Idiomatic JavaScript means writing code that aligns with the community's accepted best practices and conventions. The following table shows some common mistakes that are not idiomatic JavaScript and how to fix them:

Non-idiomatic JavaScript	Idiomatic JavaScript	Explanation
`let x = new Array();`	`let x = [];`	Use literal notation to initialize arrays.
`let y = new Object();`	`let y = {};`	Use literal notation to initialize objects.
`for (let i = 0; i < array.length; i++) { console.log(array[i]); }`	`array.forEach(element => console.log(element));`	Use `forEach` for array iteration.
`if (a !== null && a !== undefined) {...}`	`if (a) {...}`	JavaScript treats `null`, `undefined`, `0`, `NaN`, `""` as falsy. Just use `if (a)` to check for these.
`let z; if (x) { z = y; } else { z = w; }`	`let z = x ? y : w;`	Use the ternary operator for simple conditional assignment.
`arr.indexOf(el) === -1`	`!arr.includes(el)`	Use `includes` to check if an array contains a specific element.
`for (let i = 0; i < users.length; i++) { if (users[i].age > 21) { adults.push(users[i]); }}`	`let adults = users.filter(user => user.age > 21);`	Use `filter` for creating a new array with all elements that pass a test.

Unintentionally Mutating Array or Objects

JavaScript is a language where arrays and objects are mutable and are passed by reference. Therefore, any changes to the array or object inside a function will reflect outside the function as well, leading to unintentional side effects.

Let's consider a simple array-based DFS approach where you're not properly managing mutations:

JavaScript

```
let graph = {
  'A': ['B', 'C'],
  'B': ['A'],
  'C': ['A', 'B', 'D', 'E'],
  'D': ['C', 'E', 'F'],
  'E': ['C', 'D'],
  'F': ['D']
};

let visited = [];

function dfs(node) {
  visited.push(node);
  for (let neighbor of graph[node]) {
    if (!visited.includes(neighbor)) {
      dfs(neighbor);
    }
  }
  return visited;
}

let pathFromA = dfs('A'); // ['A', 'B', 'C', 'D', 'E', 'F']
let pathFromB = dfs('B'); // ['A', 'B', 'C', 'D', 'E', 'F']
```

In the above code, we expect `pathFromA` and `pathFromB` to be different, but since `visited` is shared and gets mutated during each `DFS` run, `pathFromB` doesn't give us the expected result.

To fix this, we need to initialize `visited` within the function itself:

JavaScript

```
function dfs(node, visited = []) {
  visited.push(node);
  for (let neighbor of graph[node]) {
    if (!visited.includes(neighbor)) {
      dfs(neighbor, visited);
    }
  }
  return visited;
}

let pathFromA = dfs('A'); // ['A', 'B', 'C', 'D', 'E', 'F']
let pathFromB = dfs('B'); // ['B', 'A', 'C', 'D', 'E', 'F']
```

Now, `pathFromA` and `pathFromB` are different as expected. Understanding and managing mutations properly is crucial in JavaScript, particularly in tricky algorithms such as DFS.

Not Understanding the Sort() Method

JavaScript's built-in `Array.prototype.sort()` method can be a source of confusion, especially when sorting numerical arrays. If no compare function is supplied, `sort()` will convert items to strings and sort them in lexicographic (alphabetical) order, which can lead to unexpected results when dealing with numbers.

For instance, let's say you're working on a coding problem where you're given an array of integers, and you need to sort them in ascending order. You might think you could simply use `sort()`:

JavaScript

```
let arr = [10, 21, 4, 15];
arr.sort();
console.log(arr); // Outputs: [10, 15, 21, 4]
```

This output isn't what you'd expect if you wanted to sort numerically. It's because `sort()` converts the numbers to strings, and '10' is lexicographically less than '4'.

To correctly sort numbers in JavaScript, you need to supply a comparator function:

JavaScript

```
let arr = [10, 21, 4, 15];

// sort method is passed a comparator function
// if comparator(a, b) returns a negative number, a comes before b
// if comparator(a, b) returns a positive number, b comes before a
// if comparator(a, b) returns 0, a and b are unchanged with respect to each other
arr.sort((a, b) => a - b);

console.log(arr); // Outputs: [4, 10, 15, 21]
```

Using 'var' instead of 'let' or 'const'

The use of `var` is considered outdated in modern JavaScript (ES6 and later). Instead, `let` and `const` are preferred because they provide block scoping, reducing potential bugs and making the code easier to predict and understand.

- Use `const` when the variable should not be reassigned. This is often true for function declarations, imported modules, and configuration variables. Using `const` can help you catch errors where you accidentally try to reassign a variable.

- Use `let` when the variable will be reassigned. This is common in loops (for instance, counters), and in some algorithm implementations.

Choosing `let` or `const` appropriately in your code makes it more predictable and signals to other developers (and interviewers) that you understand the variable's purpose and lifecycle. This can make your code easier to read and maintain.

It's also important to note that `const` does not make the entire variable immutable, only the assignment. For instance, if you declare an object or an array with `const`, you can still modify the elements in the array or the object's properties. This can lead to unintentional behavior if not fully understood.

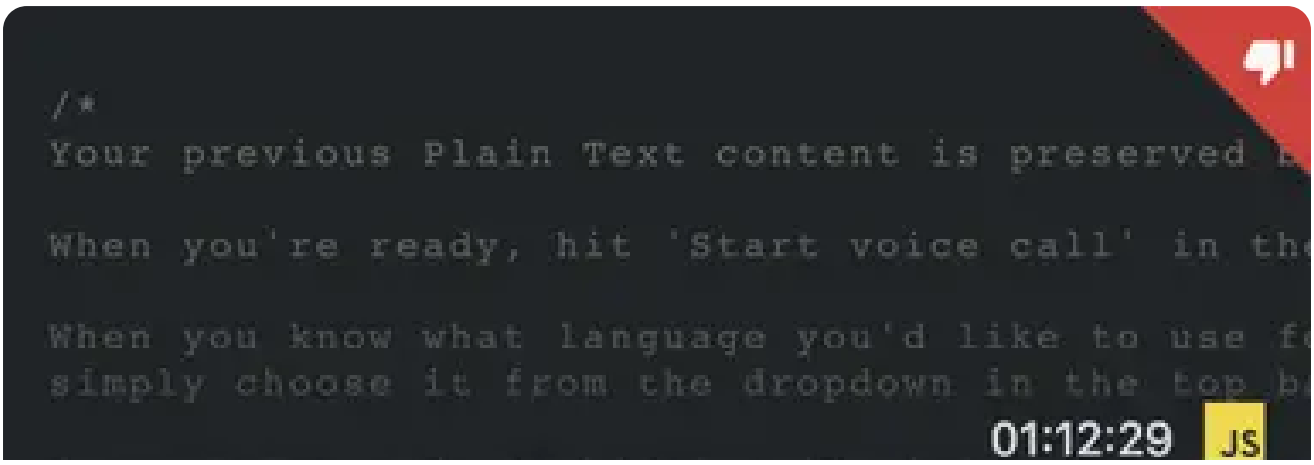
JavaScript

```
const obj = {};
obj.property = 'value'; // This is allowed

const arr = [];
arr.push(1); // This is allowed
```

JavaScript Interview Replays

Below you can find replays of mock interviews conducted on our platform in JavaScript. The questions asked in these interviews tend to be language-agnostic (rather than asking about language-specific details and idiosyncrasies), but in these cases, the interviewee chose JavaScript as the language they would work in.



Google Interviewer

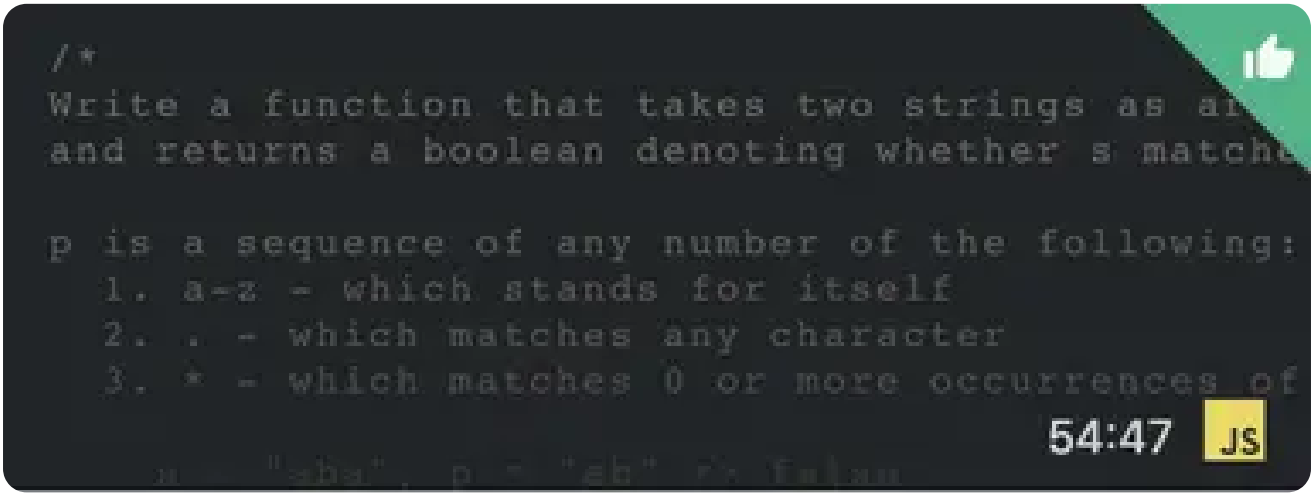
Meeting hour optimization

Intergalactic Avenger, a Google engineer, interviewed Stealthy Werewolf in JavaScript

Google Interviewer

Longest common subsequence of two strings

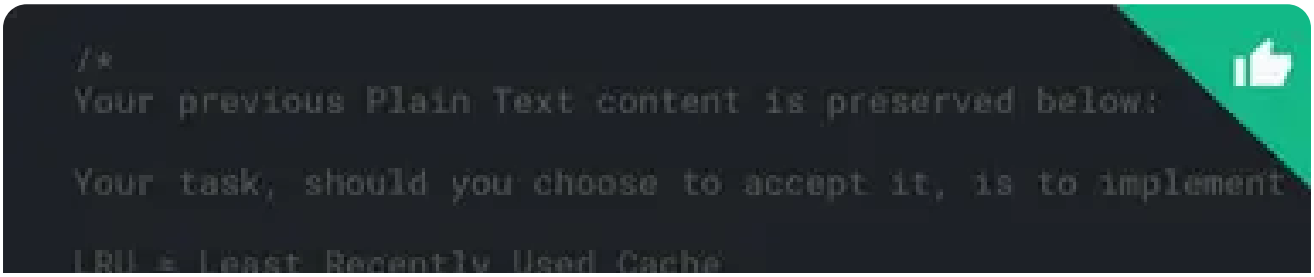
Paisley Wallaby, a Google engineer, interviewed Stealthy Dictaphone in JavaScript



Google Interviewer

Regular expression matcher

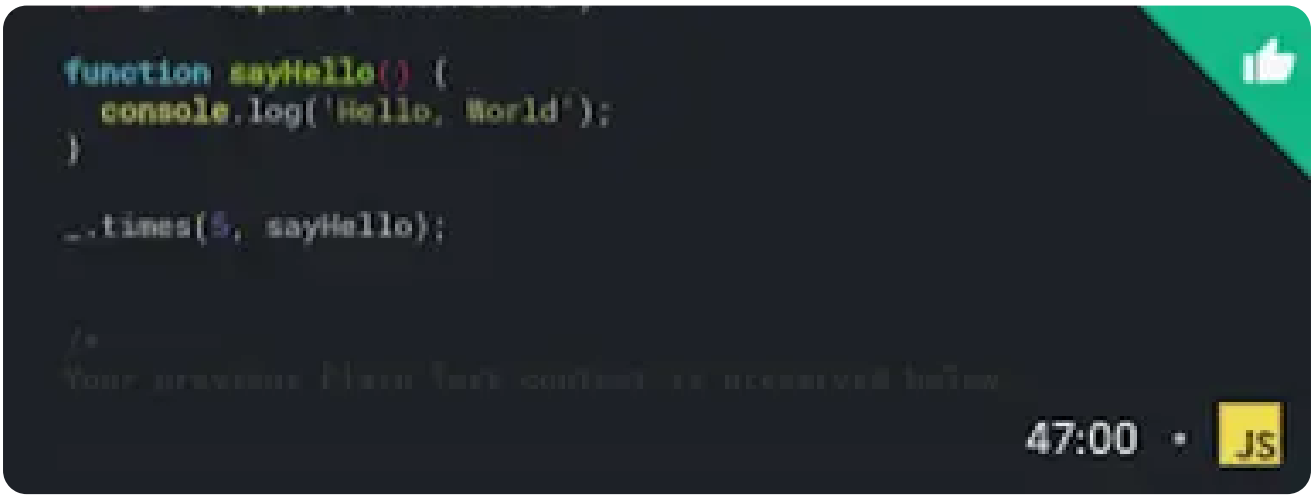
Paisley Wallaby, a Google engineer, interviewed Fresh Albatross in JavaScript



Pivotal Labs Interviewer

LRU Cache

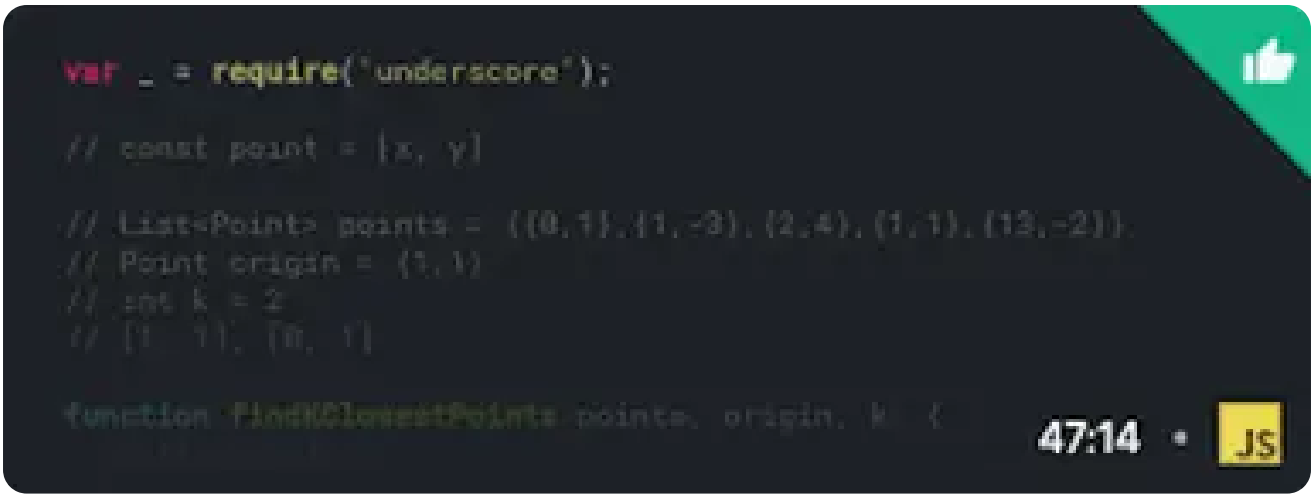
Fearsome Sandwich, a Pivotal Labs engineer, interviewed Special Chameleon in JavaScript



Google Interviewer

Memory efficient lookup

Intergalactic Avenger, a Google engineer, interviewed Samurai Razor in JavaScript



Microsoft Interviewer

K closest points

Indelible Raven, a Microsoft engineer, interviewed Nimble Pumpkin in JavaScript

See more like this

About interviewing.io

interviewing.io is a **mock interview practice platform**. We've hosted over 100K mock interviews, conducted by senior engineers from FAANG & other top companies. We've drawn on data from these interviews to bring you the best interview prep resource on the web.

We know exactly what to do and say to get the company, title, and salary you want.

Interview prep and job hunting are chaos and pain. We can help. Really.

Netflix interview questions

Microsoft interview questions

Popular Interview Questions

Reverse string

Longest substring without repeating characters

Longest common subsequence

Container with most water

Reverse linked list

K closest points to origin

Kth smallest element

Reverse words in a string

Guides

Amazon Leadership Principles

System Design Interview Guide

FAANG Hiring Process Guide

Company

For engineers

For employers

Blog

Press

FAQ

Security

Log in

Privacy Policy

Terms of Service