30-seconds / 30-seconds-of-interviews

A curated collection of common interview questions to help you prepare for your next interview. https://30secondsofinterviews.org

#interview #interview-questions #learning-resources #awesome-list #javascript #html #css #snippets #snippets-collection #education #learn-to-code

814 commits | 5 branches | 0 releases | 36 contributors | MIT

Branch: master | New pull request | Create new file | Upload files | Find File | Clone or download

30secondsofcode Travis build: 1201 [cron] — Latest commit `37f00dd` 5 days ago

.github	add pull request template	11 months ago
.travis	Update push.sh	6 months ago
data	Travis build: 1113 [cron]	2 months ago
docs	Travis build: 1201 [cron]	5 days ago
questions	fix typo	2 months ago
scripts	Travis build: 1095 [FORCED]	2 months ago
static-parts	Delete Gitter link	25 days ago
website	add recommended resource	2 months ago
.babelrc	Add 'show answer' button	11 months ago
.editorconfig	add editorconfig	11 months ago
.eslintignore	[FEATURE] Enforce style rules fix #46 !only merge after #48 (#49)	11 months ago
.eslintrc.js	[WIP][ENHANCEMENT] Dropdown filters (#51)	11 months ago
.gitignore	[FEATURE] Adding .idea in .gitignore (#148)	27 days ago
.prettierrc.js	conditional Travis Lint using prettier-eslint, prettify snippets (#59)	11 months ago
.travis.yml	fixing the push force --force-build	10 months ago
30s-favicon.ico	add favicon 😎	11 months ago
CODE_OF_CONDUCT.md	Create CODE_OF_CONDUCT.md	11 months ago
CONTRIBUTING.md	Update guidelines	11 months ago
LICENSE	Create LICENSE	11 months ago
README.md	Travis build: 1179 [cron]	24 days ago
digitalocean.png	DigitalOcean ❤️	9 months ago
frontendmasters.png	add sponsors	7 months ago
logo.jpg	update logo	11 months ago
logo.psd	Create new logo, update static files	11 months ago
logo.svg	add logo in SVG format	11 months ago
package-lock.json	wip recommended resources	2 months ago
package.json	wip recommended resources	2 months ago
promo.jpg	update promo img	11 months ago
question-template.md	update template	10 months ago
yarn.lock	Travis build: 1125 [cron]	2 months ago

README.md

30 Seconds of Interviews

A curated collection of common interview questions to help you prepare for your next interview.

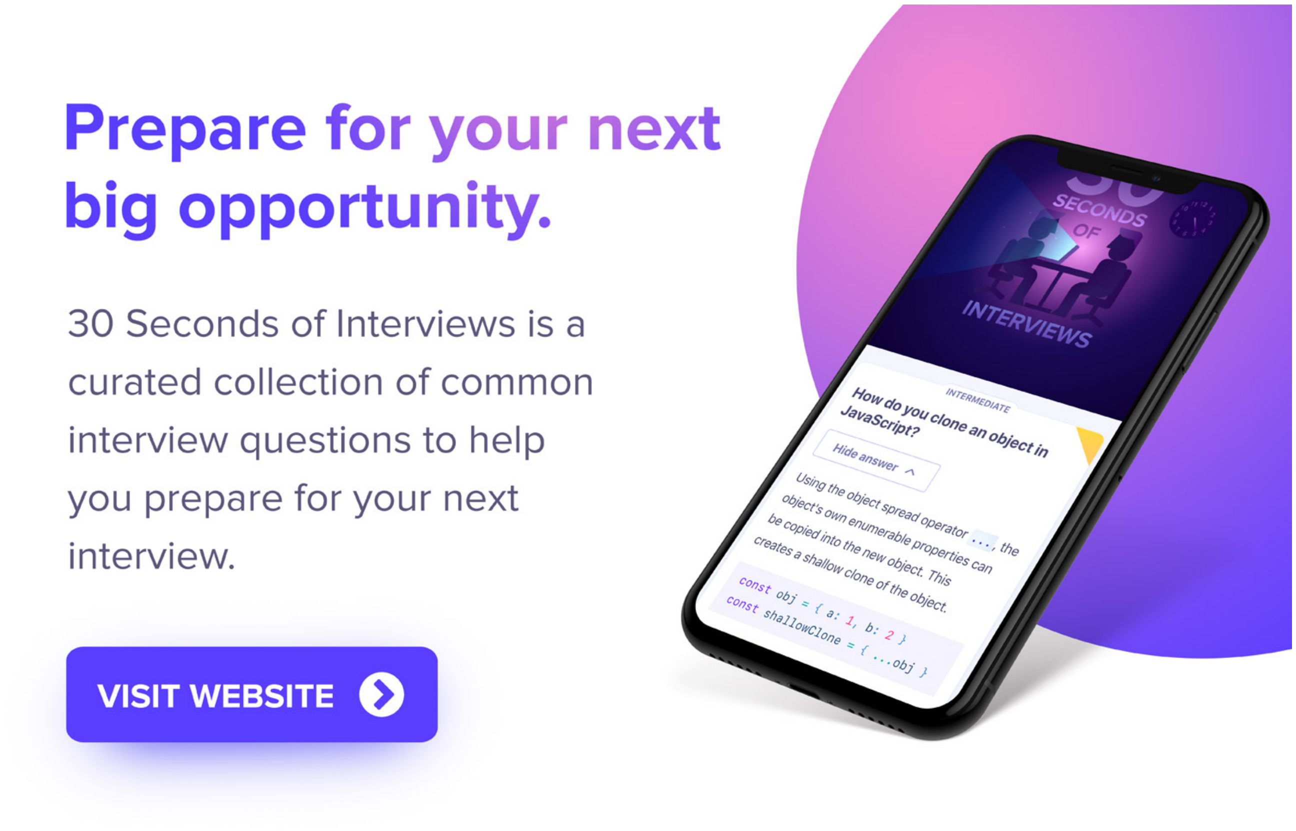
PRs welcome build passing Product Hunt vote licence MIT

> *This README is built using markdown-builder.*

Foreword

Interviews are daunting and can make even the most seasoned expert forget things under pressure. Review and learn what questions are commonly encountered in interviews curated by the community that's answered them and go prepared for anything they'll ask. By bringing together experience and real-world examples, you can go from being nervous to being prepared for that next big opportunity.

View online


Prepare for your next big opportunity.
30 Seconds of Interviews is a curated collection of common interview questions to help you prepare for your next interview.
VISIT WEBSITE
SECONDS OF INTERVIEWS
INTERMEDIATE
How do you clone an object in JavaScript?
Hide answer
Using the object spread operator ..., the object's own enumerable properties can be copied into the new object. This creates a shallow clone of the object.
const obj = { a: 1, b: 2 }
const shallowClone = { ...obj }

Our sponsors

DigitalOcean

Contributing

> 30 seconds of interviews is a community effort, so feel free to contribute in any way you can. Every contribution helps!

Do you have an excellent idea or know some cool questions that aren't on the list? Read the contribution guidelines and submit a pull request.

Join our Gitter channel to help with the development of the project.

Related projects

- 30 Seconds of Code
- 30 Seconds of CSS
- 30 Seconds of React
- 30 Seconds of Knowledge

Table of Contents

JavaScript

▶ View contents

React

▶ View contents

HTML

▶ View contents

CSS

▶ View contents

Node

▶ View contents

Security

▶ View contents

JavaScript

What is the difference between the equality operators `==` and `===` ?

▼ View answer

Triple equals (`===`) checks for strict equality, which means both the type and value must be the same. Double equals (`==`) on the other hand first performs type coercion so that both operands are of the same type and then applies strict comparison.

Good to hear

- Whenever possible, use triple equals to test equality because loose equality `==` can have unintuitive results.
- Type coercion means the values are converted into the same type.
- Mention of falsy values and their comparison.

Additional Links

- MDN docs for comparison operators

⬆ Back to top

What is the difference between an element and a component in React?

▼ View answer

An element is a plain JavaScript object that represents a DOM node or component. Elements are pure and never mutated, and are cheap to create.

A component is a function or class. Components can have state and take props as input and return an element tree as output (although they can represent generic containers or wrappers and don't necessarily have to emit DOM). Components can initiate side effects in lifecycle methods (e.g. AJAX requests, DOM mutations, interfacing with 3rd party libraries) and may be expensive to create.

```
const Component = () => "Hello"
const componentElement = <Component />
const domNodeElement = <div />
```

Good to hear

- Elements are immutable, plain objects that describe the DOM nodes or components you want to render.
- Components can be either classes or functions, that take props as an input and return an element tree as the output.

Additional Links

- React docs on Rendering Elements
- React docs on Components and Props

⬆ Back to top

What is the difference between the postfix `i++` and prefix `++i` increment operators?

▼ View answer

Both increment the variable value by 1. The difference is what they evaluate to.

The postfix increment operator evaluates to the value *before* it was incremented.

```
let i = 0
i++ // 0
// i === 1
```

The prefix increment operator evaluates to the value *after* it was incremented.

```
let i = 0
++i // 1
// i === 1
```

Good to hear

Additional Links

⬆ Back to top

In which states can a Promise be?

▼ View answer

A `Promise` is in one of these states:

- pending: initial state, neither fulfilled nor rejected.
- fulfilled: meaning that the operation completed successfully.
- rejected: meaning that the operation failed.

A pending promise can either be fulfilled with a value, or rejected with a reason (error). When either of these options happens, the associated handlers queued up by a promise's then method are called.

Good to hear

Additional Links

- Official Web Docs Promise

⬆ Back to top

What is a stateful component in React?

▼ View answer

A stateful component is a component whose behavior depends on its state. This means that two separate instances of the component if given the same props will not necessarily render the same output, unlike pure function components.

```
// Stateful class component
class App extends Component {
  constructor(props) {
    super(props)
    this.state = { count: 0 }
  }
  render() {
    // ...
  }
}

// Stateful function component
function App() {
  const [count, setCount] = useState(0)
  return // ...
}
```

Good to hear

- Stateful components have internal state that they depend on.
- Stateful components are class components or function components that use stateful Hooks.
- Stateful components have their state initialized in the constructor or with `useState()` .

Additional Links

- React docs on State and Lifecycle

⬆ Back to top

What is a stateless component?

▼ View answer

A stateless component is a component whose behavior does not depend on its state. Stateless components can be either functional or class components. Stateless functional components are easier to maintain and test since they are guaranteed to produce the same output given the same props. Stateless functional components should be preferred when lifecycle hooks don't need to be used.

Good to hear

- Stateless components are independent of their state.
- Stateless components can be either class or functional components.
- Stateless functional components avoid the `this` keyword altogether.

Additional Links

- React docs on State and Lifecycle

⬆ Back to top

Create a function `batches` that returns the maximum number of whole batches that can be cooked from a recipe.

```
/**
It accepts two objects as arguments: the first object is the recipe
for the food, while the second object is the available ingredients.
Each ingredient's value is number representing how many units there are.

`batches(recipe, available)`
*/

// 0 batches can be made
batches(
  { milk: 100, butter: 50, flour: 5 },
  { milk: 132, butter: 48, flour: 51 }
)
batches(
  { milk: 100, flour: 4, sugar: 10, butter: 5 },
  { milk: 1288, flour: 9, sugar: 95 }
)

// 1 batch can be made
batches(
  { milk: 100, butter: 50, cheese: 10 },
  { milk: 198, butter: 52, cheese: 10 }
)

// 2 batches can be made
batches(
  { milk: 2, sugar: 40, butter: 20 },
  { milk: 5, sugar: 120, butter: 500 }
)
```

▼ View answer

We must have all ingredients of the recipe available, and in quantities that are more than or equal to the number of units required. If just one of ingredients is not available or lower than needed, we cannot make a single batch.

Use `Object.keys()` to return the ingredients of the recipe as an array, then use `Array.prototype.map()` to map each ingredient to the ratio of available units to the amount required by the recipe. If one of the ingredients required by the recipe is not available at all, the ratio will evaluate to `NaN`, so the logical OR operator can be used to fallback to `0` in this case.

Use the spread `...` operator to feed the array of all the ingredient ratios into `Math.min()` to determine the lowest ratio. Passing this entire result into `Math.floor()` rounds down to return the maximum number of whole batches.

```
const batches = (recipe, available) =>
  Math.floor(
    Math.min(...Object.keys(recipe).map(k => available[k] / recipe[k] || 0))
  )
```

Good to hear

Additional Links

⬆ Back to top

Create a standalone function `bind` that is functionally equivalent to the method `Function.prototype.bind`.

```
function example() {
  console.log(this)
}
const boundExample = bind(example, { a: true })
boundExample.call({ b: true }) // logs { a: true }
```

▼ View answer

Return a function that accepts an arbitrary number of arguments by gathering them with the rest `...` operator. From that function, return the result of calling the `fn` with `Function.prototype.apply` to apply the context and the array of arguments to the function.

```
const bind = (fn, context) => (...args) => fn.apply(context, args)
```

Good to hear

Additional Links

⬆ Back to top

What is the purpose of callback function as an argument of `setState` ?

▼ View answer

The callback function is invoked when `setState` has finished and the component gets rendered. Since `setState` is asynchronous, the callback function is used for any post action.

```
setState({ name: "sudheer" }, () => {
  console.log("The name has updated and component re-rendered")
})
```

Good to hear

- The callback function is invoked after `setState` finishes and is used for any post action.
- It is recommended to use lifecycle method rather this callback function.

Additional Links

- React docs on `setState`

⬆ Back to top

What is a callback? Can you show an example using one?

▼ View answer

Callbacks are functions passed as an argument to another function to be executed once an event has occurred or a certain task is complete, often used in asynchronous code. Callback functions are invoked later by a piece of code but can be declared on initialization without being invoked.

As an example, event listeners are asynchronous callbacks that are only executed when a specific event occurs.

```
function onClick() {
  console.log("The user clicked on the page.")
}
document.addEventListener("click", onClick)
```

However, callbacks can also be synchronous. The following `map` function takes a callback function that is invoked synchronously for each iteration of the loop (array element).

```
const map = (arr, callback) => {
  const result = []
  for (let i = 0; i < arr.length; i++) {
    result.push(callback(arr[i], i))
  }
  return result
}
map([1, 2, 3, 4, 5], n => n * 2) // [2, 4, 6, 8, 10]
```

Good to hear

- Functions are first-class objects in JavaScript
- Callbacks vs Promises

Additional Links

- MDN docs for callbacks

⬆ Back to top

How do you clone an object in JavaScript?

▼ View answer

Using the object spread operator `...`, the object's own enumerable properties can be copied into the new object. This creates a shallow clone of the object.

```
const obj = { a: 1, b: 2 }
const shallowClone = { ...obj }
```

With this technique, prototypes are ignored. In addition, nested objects are not cloned, but rather their references get copied, so nested objects still refer to the same objects as the original. Deep-cloning is much more complex in order to effectively clone any type of object (Date, RegExp, Function, Set, etc) that may be nested within the object.

Other alternatives include:

- `JSON.parse(JSON.stringify(obj))` can be used to deep-clone a simple object, but it is CPU-intensive and only accepts valid JSON (therefore it strips functions and does not allow circular references).
- `Object.assign({}, obj)` is another alternative.
- `Object.keys(obj).reduce((acc, key) => (acc[key] = obj[key], acc), {})` is another more verbose alternative that shows the concept in greater depth.

Good to hear

- JavaScript passes objects by reference, meaning that nested objects get their references copied, instead of their values.
- The same method can be used to merge two objects.

Additional Links

- MDN docs for Object.assign()
- Clone an object in vanilla JS

⬆ Back to top

How do you compare two objects in JavaScript?

▼ View answer

Even though two different objects can have the same properties with equal values, they are not considered equal when compared using `==` or `===`. This is because they are being compared by their reference (location in memory), unlike primitive values which are compared by value.

In order to test if two objects are equal in structure, a helper function is required. It will iterate through the own properties of each object to test if they have the same values, including nested objects. Optionally, the prototypes of the objects may also be tested for equivalence by passing `true` as the 3rd argument.

Note: this technique does not attempt to test equivalence of data structures other than plain objects, arrays, functions, dates and primitive values.

```
function isDeepEqual(obj1, obj2, testPrototypes = false) {
  if (obj1 === obj2) {
    return true
  }

  if (typeof obj1 === "function" && typeof obj2 === "function") {
    return obj1.toString() === obj2.toString()
  }

  if (obj1 instanceof Date && obj2 instanceof Date) {
    return obj1.getTime() === obj2.getTime()
  }

  if (
    Object.prototype.toString.call(obj1) !==
      Object.prototype.toString.call(obj2) ||
    typeof obj1 !== "object"
  ) {
    return false
  }

  const prototypesAreEqual = testPrototypes
    ? isDeepEqual(
        Object.getPrototypeOf(obj1),
        Object.getPrototypeOf(obj2),
        true
      )
    : true

  const obj1Props = Object.getOwnPropertyNames(obj1)
  const obj2Props = Object.getOwnPropertyNames(obj2)

  return (
    obj1Props.length === obj2Props.length &&
    prototypesAreEqual &&
    obj1Props.every(prop => isDeepEqual(obj1[prop], obj2[prop]))
  )
}
```

Good to hear

- Primitives like strings and numbers are compared by their value
- Objects on the other hand are compared by their reference (location in memory)

Additional Links

- Object Equality in JavaScript
- Deep comparison between two values

⬆ Back to top

What is CORS?

▼ View answer

Cross-Origin Resource Sharing or CORS is a mechanism that uses additional HTTP headers to grant a browser permission to access resources from a server at an origin different from the website origin.

An example of a cross-origin request is a web application served from `http://mydomain.com` that uses AJAX to make a request for `http://yourdomain.com`.

For security reasons, browsers restrict cross-origin HTTP requests initiated by JavaScript. `XMLHttpRequest` and `fetch` follow the same-origin policy, meaning a web application using those APIs can only request HTTP resources from the same origin the application was accessed, unless the response from the other origin includes the correct CORS headers.

Good to hear

- CORS behavior is not an error, it's a security mechanism to protect users.
- CORS is designed to prevent a malicious website that a user may unintentionally visit from making a request to a legitimate website to read their personal data or perform actions against their will.

Additional Links

- MDN docs for CORS

⬆ Back to top

What is the DOM?

▼ View answer
The DOM (Document Object Model) is a cross-platform API that treats HTML and XML documents as a tree structure consisting of nodes. These nodes (such as elements and text nodes) are objects that can be programmatically manipulated and any visible changes made to them are reflected live in the document. In a browser, this API is available to JavaScript where DOM nodes can be manipulated to change their styles, contents, placement in the document, or interacted with through event listeners.

Good to hear

- The DOM was designed to be independent of any particular programming language, making the structural representation of the document available from a single, consistent API.
- The DOM is constructed progressively in the browser as a page loads, which is why scripts are often placed at the bottom of a page, in the `<head>` with a `defer` attribute, or inside a `DOMContentLoaded` event listener. Scripts that manipulate DOM nodes should be run after the DOM has been constructed to avoid errors.
- `document.getElementById()` and `document.querySelector()` are common functions for selecting DOM nodes.
- Setting the `innerHTML` property to a new value runs the string through the HTML parser, offering an easy way to append dynamic HTML content to a node.

Additional Links

- MDN docs for DOM

⬆ Back to top

What is event delegation and why is it useful? Can you show an example of how to use it?

▼ View answer
Event delegation is a technique of delegating events to a single common ancestor. Due to event bubbling, events "bubble" up the DOM tree by executing any handlers progressively on each ancestor element up to the root that may be listening to it.

DOM events provide useful information about the element that initiated the event via `Event.target`. This allows the parent element to handle behavior as though the target element was listening to the event, rather than all children of the parent or the parent itself.

This provides two main benefits:

- It increases performance and reduces memory consumption by only needing to register a single event listener to handle potentially thousands of elements.
- If elements are dynamically added to the parent, there is no need to register new event listeners for them.

Instead of:

```
document.querySelectorAll("button").forEach(button => {
  button.addEventListener("click", handleButtonClick)
})
```

Event delegation involves using a condition to ensure the child target matches our desired element:

```
document.addEventListener("click", e => {
  if (e.target.closest("button")) {
    handleButtonClick()
  }
})
```

Good to hear

- The difference between event bubbling and capturing

Additional Links

- Event Delegation

⬆ Back to top

What is the difference between an expression and a statement in JavaScript?

▼ View answer

There are two main syntactic categories in JavaScript: expressions and statements. A third one is both together, referred to as an expression statement. They are roughly summarized as:

- **Expression**: produces a value
- **Statement**: performs an action
- **Expression statement**: produces a value and performs an action

A general rule of thumb:

> If you can print it or assign it to a variable, it’s an expression. If you can’t, it’s a statement.

Statements

```
let x = 0

function declaration() {}

if (true) {
}
```

Statements appear as instructions that do something but don't produce values.

```
// Assign `x` to the absolute value of `y`.
var x
if (y >= 0) {
  x = y
} else {
  x = -y
}
```

The only expression in the above code is `y >= 0` which produces a value, either `true` or `false`. A value is not produced by other parts of the code.

Expressions

Expressions produce a value. They can be passed around to functions because the interpreter replaces them with the value they resolve to.

```
5 + 5 // => 10

lastCharacter("input") // => "t"

true === true // => true
```

Expression statements

There is an equivalent version of the set of statements used before as an expression using the conditional operator:

```
// Assign `x` as the absolute value of `y`.
var x = y >= 0 ? y : -y
```

This is both an expression and a statement, because we are declaring a variable `x` (statement) as an evaluation (expression).

Good to hear

- Function declarations vs function expressions

Additional Links

- What is the difference between a statement and an expression?

⬆ Back to top

What are truthy and falsy values in JavaScript?

▼ View answer

A value is either truthy or falsy depending on how it is evaluated in a Boolean context. Falsy means false-like and truthy means true-like. Essentially, they are values that are coerced to `true` or `false` when performing certain operations.

There are 6 falsy values in JavaScript. They are:

- `false`
- `undefined`
- `null`
- `""` (empty string)
- `NaN`
- `0` (both `+0` and `-0`)

Every other value is considered truthy.

A value's truthiness can be examined by passing it into the `Boolean` function.

```
Boolean("") // false
Boolean([]) // true
```

There is a shortcut for this using the logical NOT `!` operator. Using `!` once will convert a value to its inverse boolean equivalent (i.e. not false is true), and `!` once more will convert back, thus effectively converting the value to a boolean.

```
!!"" // false
!![] // true
```

Good to hear

Additional Links

- Truthy on MDN
- Falsy on MDN

⬆ Back to top

Generate an array, containing the Fibonacci sequence, up until the nth term.

▼ View answer

Initialize an empty array of length `n`. Use `Array.prototype.reduce()` to add values into the array, using the sum of the last two values, except for the first two.

```
const fibonacci = n =>
  [...Array(n)].reduce(
    (acc, val, i) => acc.concat(i > 1 ? acc[i - 1] + acc[i - 2] : i),
    []
  )
```

Good to hear

Additional Links

- Similar problem

⬆ Back to top

What does `0.1 + 0.2 === 0.3` evaluate to?

▼ View answer

It evaluates to `false` because JavaScript uses the IEEE 754 standard for Math and it makes use of 64-bit floating numbers. This causes precision errors when doing decimal calculations, in short, due to computers working in Base 2 while decimal is Base 10.

```
0.1 + 0.2 // 0.30000000000000004
```

A solution to this problem would be to use a function that determines if two numbers are approximately equal by defining an error margin (epsilon) value that the difference between two values should be less than.

```
const approxEqual = (n1, n2, epsilon = 0.0001) => Math.abs(n1 - n2) < epsilon
approxEqual(0.1 + 0.2, 0.3) // true
```

Good to hear

- A simple solution to this problem

Additional Links

- A simple helper function to check equality
- Fix "0.1 + 0.2 = 0.300000004" in JavaScript

⬆ Back to top

What is the difference between the array methods `map()` and `forEach()` ?

▼ View answer

Both methods iterate through the elements of an array. `map()` maps each element to a new element by invoking the callback function on each element and returning a new array. On the other hand, `forEach()` invokes the callback function for each element but does not return a new array. `forEach()` is generally used when causing a side effect on each iteration, whereas `map()` is a common functional programming technique.

Good to hear

- Use `forEach()` if you need to iterate over an array and cause mutations to the elements without needing to return values to generate a new array.
- `map()` is the right choice to keep data immutable where each value of the original array is mapped to a new array.

Additional Links

- MDN docs for forEach
- MDN docs for map
- JavaScript — Map vs. ForEach

⬆ Back to top

What will the console log in this example?

```
var foo = 1
var foobar = function() {
  console.log(foo)
  var foo = 2
}
foobar()
```

▼ View answer

Due to hoisting, the local variable `foo` is declared before the `console.log` method is called. This means the local variable `foo` is passed as an argument to `console.log()` instead of the global one declared outside of the function. However, since the value is not hoisted with the variable declaration, the output will be `undefined` , not `2` .

Good to hear

- Hoisting is JavaScript's default behavior of moving declarations to the top
- Mention of `strict` mode

Additional Links

- MDN docs for hoisting

⬆ Back to top

How does hoisting work in JavaScript?

▼ View answer

Hoisting is a JavaScript mechanism where variable and function declarations are put into memory during the compile phase. This means that no matter where functions and variables are declared, they are moved to the top of their scope regardless of whether their scope is global or local.

However, the value is not hoisted with the declaration.

The following snippet:

```
console.log(hoist)
var hoist = "value"
```

is equivalent to:

```
var hoist
console.log(hoist)
hoist = "value"
```

Therefore logging `hoist` outputs `undefined` to the console, not `"value"` .

Hoisting also allows you to invoke a function declaration before it appears to be declared in a program.

```
myFunction() // No error; logs "hello"
function myFunction() {
  console.log("hello")
}
```

But be wary of function expressions that are assigned to a variable:

```
myFunction() // Error: `myFunction` is not a function
var myFunction = function() {
  console.log("hello")
}
```

Good to hear

- Hoisting is JavaScript's default behavior of moving declarations to the top
- Functions declarations are hoisted before variable declarations

Additional Links

- MDN docs for hoisting
- Understanding Hoisting in JavaScript

⬆ Back to top

What is the difference between HTML and React event handling?

▼ View answer

In HTML, the attribute name is in all lowercase and is given a string invoking a function defined somewhere:

```
<button onclick="handleClick()"></button>
```

In React, the attribute name is camelCase and are passed the function reference inside curly braces:

```
<button onClick={handleClick} />
```

In HTML, `false` can be returned to prevent default behavior, whereas in React `preventDefault` has to be called explicitly.

```
<a href="#" onclick="console.log('The link was clicked.'); return false" />
```

```
function handleClick(e) {
  e.preventDefault()
  console.log("The link was clicked.")
}
```

Good to hear

- HTML uses lowercase, React uses camelCase.

Additional Links

- React docs on Handling Events

⬆ Back to top

What is the reason for wrapping the entire contents of a JavaScript source file in a function that is immediately invoked?

▼ View answer

This technique is very common in JavaScript libraries. It creates a closure around the entire contents of the file which creates a private namespace and thereby helps avoid potential name clashes between different JavaScript modules and libraries. The function is immediately invoked so that the namespace (library name) is assigned the return value of the function.

```
const myLibrary = (function() {
  var privateVariable = 2
  return {
    publicMethod: () => privateVariable
  }
})()
privateVariable // ReferenceError
myLibrary.publicMethod() // 2
```

Good to hear

- Used among many popular JavaScript libraries
- Creates a private namespace

Additional Links

- MDN docs for closures

⬆ Back to top

What are inline conditional expressions?

▼ View answer

Since a JSX element tree is one large expression, you cannot embed statements inside. Conditional expressions act as a replacement for statements to use inside the tree.

For example, this won't work:

```
function App({ messages, isVisible }) {
  return (
    <div>
      if (messages.length > 0) {
        <h2>You have {messages.length} unread messages.</h2>
      } else {
        <h2>You have no unread messages.</h2>
      }
      if (isVisible) {
        <p>I am visible.</p>
      }
    </div>
  )
}
```

Logical AND `&&` and the ternary `? :` operator replace the `if` / `else` statements.

```
function App({ messages, isVisible }) {
  return (
    <div>
      {messages.length > 0 ? (
        <h2>You have {messages.length} unread messages.</h2>
      ) : (
        <h2>You have no unread messages.</h2>
      )}
      {isVisible && <p>I am visible.</p>}
    </div>
  )
}
```

Good to hear

Additional Links

- React docs on Conditional Rendering

⬆ Back to top

What is a key? What are the benefits of using it in lists?

▼ View answer

Keys are a special string attribute that helps React identify which items have been changed, added or removed. They are used when rendering array elements to give them a stable identity. Each element's key must be unique (e.g. IDs from the data or indexes as a last resort).

```
const todoItems = todos.map(todo => <li key={todo.id}>{todo.text}</li>)
```

- Using indexes as keys is not recommended if the order of items may change, as it might negatively impact performance and may cause issues with component state.
- If you extract list items as a separate component then apply keys on the list component instead of the `<li>` tag.

Good to hear

- Keys give elements in a collection a stable identity and help React identify changes.
- You should avoid using indexes as keys if the order of items may change.
- You should lift the key up to the component, instead of the `<li>` element, if you extract list items as components.

Additional Links

- React docs on Lists and Keys

⬆ Back to top

What is the difference between lexical scoping and dynamic scoping?

▼ View answer

Lexical scoping refers to when the location of a function's definition determines which variables you have access to. On the other hand, dynamic scoping uses the location of the function's invocation to determine which variables are available.

Good to hear

- Lexical scoping is also known as static scoping.
- Lexical scoping in JavaScript allows for the concept of closures.
- Most languages use lexical scoping because it tends to promote source code that is more easily understood.

Additional Links

- Mozilla Docs Closures & Lexical Scoping

⬆ Back to top

Create a function that masks a string of characters with `#` except for the last four (4) characters.

```
mask("123456789") // "#####6789"
```

▼ View answer

> There are many ways to solve this problem, this is just one one of them.

Using `String.prototype.slice()` we can grab the last 4 characters of the string by passing `-4` as an argument. Then, using `String.prototype.padStart()`, we can pad the string to the original length with the repeated mask character.

```
const mask = (str, maskChar = "#") =>
  str.slice(-4).padStart(str.length, maskChar)
```

Good to hear

- Short, one-line functional solutions to problems should be preferred provided they are efficient

Additional Links

⬆ Back to top

What is a MIME type and what is it used for?

▼ View answer

`MIME` is an acronym for `Multi-purpose Internet Mail Extensions`. It is used as a standard way of classifying file types over the Internet.

Good to hear

- A `MIME type` actually has two parts: a type and a subtype that are separated by a slash (/). For example, the `MIME type` for Microsoft Word files is `application/msword` (i.e., type is application and the subtype is msword).

Additional Links

- MIME Type MDN

⬆ Back to top

NodeJS often uses a callback pattern where if an error is encountered during execution, this error is passed as the first argument to the callback. What are the advantages of this pattern?

```
fs.readFile(filePath, function(err, data) {
  if (err) {
    // handle the error, the return is important here
    // so execution stops here
    return console.log(err)
  }
  // use the data object
  console.log(data)
})
```

▼ View answer
Advantages include:

- Not needing to process data if there is no need to even reference it
- Having a consistent API leads to more adoption
- Ability to easily adapt a callback pattern that will lead to more maintainable code

As you can see from below example, the callback is called with null as its first argument if there is no error. However, if there is an error, you create an Error object, which then becomes the callback's only parameter. The callback function allows a user to easily know whether or not an error occurred.

This practice is also called the *Node.js error convention*, and this kind of callback implementations are called *error-first callbacks*.

```
var isTrue = function(value, callback) {
  if (value === true) {
    callback(null, "Value was true.")
  } else {
    callback(new Error("Value is not true!"))
  }
}

var callback = function(error, retval) {
  if (error) {
    console.log(error)
    return
  }
  console.log(retval)
}

isTrue(false, callback)
isTrue(true, callback)

/*
  { stack: [Getter/Setter],
    arguments: undefined,
    type: undefined,
    message: 'Value is not true!' }
  Value was true.
*/
```

Good to hear

- This is just a convention. However, you should stick to it.

Additional Links

- The Node.js Way Understanding Error-First Callbacks
- What are the error conventions?

⬆ Back to top

What is the difference between `null` and `undefined` ?

▼ View answer

In JavaScript, two values discretely represent nothing - `undefined` and `null` . The concrete difference between them is that `null` is explicit, while `undefined` is implicit. When a property does not exist or a variable has not been given a value, the value is `undefined` . `null` is set as the value to explicitly indicate "no value". In essence, `undefined` is used when the nothing is not known, and `null` is used when the nothing is known.

Good to hear

- `typeof undefined` evaluates to `"undefined"` .
- `typeof null` evaluates `"object"` . However, it is still a primitive value and this is considered an implementation bug in JavaScript.
- `undefined == null` evaluates to `true` .

Additional Links

- MDN docs for null
- MDN docs for undefined

⬆ Back to top

Describe the different ways to create an object. When should certain ways be preferred over others?

▼ View answer

Object literal

Often used to store one occurrence of data.

```
const person = {
  name: "John",
  age: 50,
  birthday() {
    this.age++
  }
}
person.birthday() // person.age === 51
```

Constructor

Often used when you need to create multiple instances of an object, each with their own data that other instances of the class cannot affect. The `new` operator must be used before invoking the constructor or the global object will be mutated.

```
function Person(name, age) {
  this.name = name
  this.age = age
}
Person.prototype.birthday = function() {
  this.age++
}
const person1 = new Person("John", 50)
```

```js
const person2 = new Person("Sally", 20)
person1.birthday() // person1.age === 51
person2.birthday() // person2.age === 21
```

Factory function

Creates a new object similar to a constructor, but can store private data using a closure. There is also no need to use `new` before invoking the function or the `this` keyword. Factory functions usually discard the idea of prototypes and keep all properties and methods as own properties of the object.

```js
const createPerson = (name, age) => {
  const birthday = () => person.age++
  const person = { name, age, birthday }
  return person
}
const person = createPerson("John", 50)
person.birthday() // person.age === 51
```

`Object.create()`

Sets the prototype of the newly created object.

```js
const personProto = {
  birthday() {
    this.age++
  }
}
const person = Object.create(personProto)
person.age = 50
person.birthday() // person.age === 51
```

A second argument can also be supplied to `Object.create()` which acts as a descriptor for the new properties to be defined.

```js
Object.create(personProto, {
  age: {
    value: 50,
    writable: true,
    enumerable: true
  }
})
```

Good to hear

- Prototypes are objects that other objects inherit properties and methods from.
- Factory functions offer private properties and methods through a closure but increase memory usage as a tradeoff, while classes do not have private properties or methods but reduce memory impact by reusing a single prototype object.

Additional Links

⬆ Back to top

What is the difference between a parameter and an argument?

▼ View answer

Parameters are the variable names of the function definition, while arguments are the values given to a function when it is invoked.

```
function myFunction(parameter1, parameter2) {
  console.log(arguments[0]) // "argument1"
}
myFunction("argument1", "argument2")
```

Good to hear

- `arguments` is an array-like object containing information about the arguments supplied to an invoked function.
- `myFunction.length` describes the arity of a function (how many parameters it has, regardless of how many arguments it is supplied).

Additional Links

⬆ Back to top

Does JavaScript pass by value or by reference?

▼ View answer
JavaScript always passes by value. However, with objects, the value is a reference to the object.

Good to hear

- Difference between pass-by-value and pass-by-reference

Additional Links

- JavaScript Value vs Reference

⬆ Back to top

How do you pass an argument to an event handler or callback?

▼ View answer
You can use an arrow function to wrap around an event handler and pass arguments, which is equivalent to calling `bind` :

```
<button onClick={() => this.handleClick(id)} />
<button onClick={this.handleClick.bind(this, id)} />
```

Good to hear

Additional Links

- React docs on Handling Events

⬆ Back to top

What are Promises?

▼ View answer
The `Promise` object represents the eventual completion (or failure) of an asynchronous operation, and its resulting value. An example can be the following snippet, which after 100ms prints out the result string to the standard output. Also, note the catch, which can be used for error handling. `Promise` s are chainable.

```
new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve("result")
  }, 100)
})
  .then(console.log)
  .catch(console.error)
```

Good to hear

- Take a look into the other questions regarding `Promise` s!

Additional Links

- Master the JavaScript Interview: What is a Promise?

⬆ Back to top

How does prototypal inheritance differ from classical inheritance?

▼ View answer

In the classical inheritance paradigm, object instances inherit their properties and functions from a class, which acts as a blueprint for the object. Object instances are typically created using a constructor and the `new` keyword.

In the prototypal inheritance paradigm, object instances inherit directly from other objects and are typically created using factory functions or `Object.create()` .

Good to hear

Additional Links

- MDN docs for inheritance and the prototype chain

⬆ Back to top

What is the output of the following code?

```
const a = [1, 2, 3]
const b = [1, 2, 3]
const c = "1,2,3"

console.log(a == c)
console.log(a == b)
```

▼ View answer

The first `console.log` outputs `true` because JavaScript's compiler performs type conversion and therefore it compares to strings by their value. On the other hand, the second `console.log` outputs `false` because Arrays are Objects and Objects are compared by reference.

Good to hear

- JavaScript performs automatic type conversion
- Objects are compared by reference
- Primitives are compared by value

Additional Links

- JavaScript Value vs Reference

⬆ Back to top

What does the following function return?

```
function greet() {
  return
  {
    message: "hello"
  }
}
```

▼ View answer

Because of JavaScript's automatic semicolon insertion (ASI), the compiler places a semicolon after `return` keyword and therefore it returns `undefined` without an error being thrown.

Good to hear

- Automatic semicolon placement can lead to time-consuming bugs

Additional Links

- Automatic semicolon insertion in JavaScript

⬆ Back to top

Are semicolons required in JavaScript?

▼ View answer

Sometimes. Due to JavaScript's automatic semicolon insertion, the interpreter places semicolons after most statements. This means semicolons can be omitted in most cases.

However, there are some cases where they are required. They are not required at the beginning of a block, but are if they follow a line and:

1. The line starts with `[`

```
const previousLine = 3
;[1, 2, previousLine].map(n => n * 2)
```

2. The line starts with `(`

```
const previousLine = 3
;(function() {
  // ...
})()
```

In the above cases, the interpreter does not insert a semicolon after `3`, and therefore it will see the `3` as attempting object property access or being invoked as a function, which will throw errors.

Good to hear

- Semicolons are usually optional in JavaScript but have edge cases where they are required.
- If you don't use semicolons, tools like Prettier will insert semicolons for you in the places where they are required on save in a text editor to prevent errors.

Additional Links

⬆ Back to top

What is short-circuit evaluation in JavaScript?

▼ View answer

Short-circuit evaluation involves logical operations evaluating from left-to-right and stopping early.

```
true || false
```

In the above sample using logical OR, JavaScript won't look at the second operand `false`, because the expression evaluates to `true` regardless. This is known as short-circuit evaluation.

This also works for logical AND.

```
false && true
```

This means you can have an expression that throws an error if evaluated, and it won't cause issues.

```
true || nonexistentFunction()
false && nonexistentFunction()
```

This remains true for multiple operations because of left-to-right evaluation.

```
true || nonexistentFunction() || window.nothing.wouldThrowError
true || window.nothing.wouldThrowError
true
```

A common use case for this behavior is setting default values. If the first operand is falsy the second operand will be evaluated.

```
const options = {}
const setting = options.setting || "default"
setting // "default"
```

Another common use case is only evaluating an expression if the first operand is truthy.

```
// Instead of:
addEventListener("click", e => {
  if (e.target.closest("button")) {
    handleButtonClick(e)
  }
})

// You can take advantage of short-circuit evaluation:
addEventListener(
  "click",
  e => e.target.closest("button") && handleButtonClick(e)
)
```

In the above case, if `e.target` is not or does not contain an element matching the `"button"` selector, the function will not be called. This is because the first operand will be falsy, causing the second operand to not be evaluated.

Good to hear

- Logical operations do not produce a boolean unless the operand(s) evaluate to a boolean.

Additional Links

- JavaScript: What is short-circuit evaluation?

⬆ Back to top

What is the difference between synchronous and asynchronous code in JavaScript?

▼ View answer
Synchronous means each operation must wait for the previous one to complete.

Asynchronous means an operation can occur while another operation is still being processed.

In JavaScript, all code is synchronous due to the single-threaded nature of it. However, asynchronous operations not part of the program (such as `XMLHttpRequest` or `setTimeout`) are processed outside of the main thread because they are controlled by native code (browser APIs), but callbacks part of the program will still be executed synchronously.

Good to hear

- JavaScript has a concurrency model based on an "event loop".
- Functions like `alert` block the main thread so that no user input is registered until the user closes it.

Additional Links

⬆ Back to top

What does the following code evaluate to?

```
typeof typeof 0
```

▼ View answer
It evaluates to `"string"` .

`typeof 0` evaluates to the string `"number"` and therefore `typeof "number"` evaluates to `"string"` .

Good to hear

Additional Links

- MDN docs for typeof

⬆ Back to top

What are JavaScript data types?

▼ View answer
The latest ECMAScript standard defines seven data types, six of them being primitive: `Boolean` , `Null` , `Undefined` , `Number` , `String` , `Symbol` and one non-primitive data type: `Object` .

Good to hear

- Mention of newly added `Symbol` data type
- `Array` , `Date` and `function` are all of type `object`
- Functions in JavaScript are objects with the capability of being callable

Additional Links

- MDN docs for data types and data structures
- Understanding Data Types in JavaScript

⬆ Back to top

What are the differences between `var`, `let`, `const` and no keyword statements?

▼ View answer

No keyword

When no keyword exists before a variable assignment, it is either assigning a global variable if one does not exist, or reassigns an already declared variable. In non-strict mode, if the variable has not yet been declared, it will assign the variable as a property of the global object (`window` in browsers). In strict mode, it will throw an error to prevent unwanted global variables from being created.

var

`var` was the default statement to declare a variable until ES2015. It creates a function-scoped variable that can be reassigned and redeclared. However, due to its lack of block scoping, it can cause issues if the variable is being reused in a loop that contains an asynchronous callback because the variable will continue to exist outside of the block scope.

Below, by the time the the `setTimeout` callback executes, the loop has already finished and the `i` variable is `10`, so all ten callbacks reference the same variable available in the function scope.

```
for (var i = 0; i < 10; i++) {
  setTimeout(() => {
    // logs `10` ten times
    console.log(i)
  })
}

/* Solutions with `var` */
for (var i = 0; i < 10; i++) {
  // Passed as an argument will use the value as-is in
  // that point in time
  setTimeout(console.log, 0, i)
}

for (var i = 0; i < 10; i++) {
  // Create a new function scope that will use the value
  // as-is in that point in time
  ;(i => {
    setTimeout(() => {
      console.log(i)
    })
  })(i)
}
```

let

`let` was introduced in ES2015 and is the new preferred way to declare variables that will be reassigned later. Trying to redeclare a variable again will throw an error. It is block-scoped so that using it in a loop will keep it scoped to the iteration.

```
for (let i = 0; i < 10; i++) {
  setTimeout(() => {
    // logs 0, 1, 2, 3, ...
    console.log(i)
  })
}
```

const

`const` was introduced in ES2015 and is the new preferred default way to declare all variables if they won't be reassigned later, even for objects that will be mutated (as long as the reference to the object does not change). It is block-scoped and cannot be reassigned.

```js
const myObject = {}
myObject.prop = "hello!" // No error
myObject = "hello" // Error
```

Good to hear

- All declarations are hoisted to the top of their scope.
- However, with `let` and `const` there is a concept called the temporal dead zone (TDZ). While the declarations are still hoisted, there is a period between entering scope and being declared where they cannot be accessed.
- Show a common issue with using `var` and how `let` can solve it, as well as a solution that keeps `var` .
- `var` should be avoided whenever possible and prefer `const` as the default declaration statement for all variables unless they will be reassigned later, then use `let` if so.

Additional Links

- `let` vs `const`

⬆ Back to top

What is a cross-site scripting attack (XSS) and how do you prevent it?

▼ View answer

XSS refers to client-side code injection where the attacker injects malicious scripts into a legitimate website or web application. This is often achieved when the application does not validate user input and freely injects dynamic HTML content.

For example, a comment system will be at risk if it does not validate or escape user input. If the comment contains unescaped HTML, the comment can inject a `<script>` tag into the website that other users will execute against their knowledge.

- The malicious script has access to cookies which are often used to store session tokens. If an attacker can obtain a user's session cookie, they can impersonate the user.
- The script can arbitrarily manipulate the DOM of the page the script is executing in, allowing the attacker to insert pieces of content that appear to be a real part of the website.
- The script can use AJAX to send HTTP requests with arbitrary content to arbitrary destinations.

Good to hear

- On the client, using `textContent` instead of `innerHTML` prevents the browser from running the string through the HTML parser which would execute scripts in it.
- On the server, escaping HTML tags will prevent the browser from parsing the user input as actual HTML and therefore won't execute the script.

Additional Links

- Cross-Site Scripting Attack (XSS)

⬆ Back to top

What is Big O Notation?

▼ View answer

Big O notation is used in Computer Science to describe the time complexity of an algorithm. The best algorithms will execute the fastest and have the simplest complexity.

Algorithms don't always perform the same and may vary based on the data they are supplied. While in some cases they will execute quickly, in other cases they will execute slowly, even with the same number of elements to deal with.

In these examples, the base time is 1 element = `1ms` .

O(1)

```
arr[arr.length - 1]
```

- 1000 elements = `1ms`

Constant time complexity. No matter how many elements the array has, it will theoretically take (excluding real-world variation) the same amount of time to execute.

O(N)

```
arr.filter(fn)
```

- 1000 elements = `1000ms`

Linear time complexity. The execution time will increase linearly with the number of elements the array has. If the array has 1000 elements and the function takes 1ms to execute, 7000 elements will take 7ms to execute. This is because the function must iterate through all elements of the array before returning a result.

O([1, N])

```
arr.some(fn)
```

- 1000 elements = `1ms <= x <= 1000ms`

The execution time varies depending on the data supplied to the function, it may return very early or very late. The best case here is O(1) and the worst case is O(N).

O(NlogN)

```
arr.sort(fn)
```

- 1000 elements ~= `10000ms`

Browsers usually implement the quicksort algorithm for the `sort()` method and the average time complexity of quicksort is O(NlgN). This is very efficient for large collections.

O(N^2)

```
for (let i = 0; i < arr.length; i++) {
  for (let j = 0; j < arr.length; j++) {
    // ...
  }
}
```

- 1000 elements = `1000000ms`

The execution time rises quadratically with the number of elements. Usually the result of nesting loops.

O(N!)

```
const permutations = arr => {
  if (arr.length <= 2) return arr.length === 2 ? [arr, [arr[1], arr[0]]] : arr
  return arr.reduce(
    (acc, item, i) =>
      acc.concat(
        permutations([...arr.slice(0, i), ...arr.slice(i + 1)]).map(val => [
          item,
          ...val
        ])
      ),
    []
  )
}
```

- 1000 elements = `Infinity` (practically) ms

The execution time rises extremely fast with even just 1 addition to the array.

Good to hear

- Be wary of nesting loops as execution time increases exponentially.

Additional Links

- Big O Notation in JavaScript

⬆ Back to top

How can you avoid callback hells?

```
getData(function(a) {
  getMoreData(a, function(b) {
    getMoreData(b, function(c) {
      getMoreData(c, function(d) {
        getMoreData(d, function(e) {
          // ...
        })
      })
    })
  })
})
```

▼ View answer

Refactoring the functions to return promises and using `async/await` is usually the best option. Instead of supplying the functions with callbacks that cause deep nesting, they return a promise that can be `await` ed and will be resolved once the data has arrived, allowing the next line of code to be evaluated in a sync-like fashion.

The above code can be restructured like so:

```
async function asyncAwaitVersion() {
  const a = await getData()
  const b = await getMoreData(a)
  const c = await getMoreData(b)
  const d = await getMoreData(c)
  const e = await getMoreData(d)
  // ...
}
```

There are lots of ways to solve the issue of callback hells:

- Modularization: break callbacks into independent functions

- Use a control flow library, like async
- Use generators with Promises
- Use async/await (from v7 on)

Good to hear

- As an efficient JavaScript developer, you have to avoid the constantly growing indentation level, produce clean and readable code and be able to handle complex flows.

Additional Links

- Avoiding Callback Hell in Node.js
- Asynchronous JavaScript: From Callback Hell to Async and Await

⬆ Back to top

Which is the preferred option between callback refs and findDOMNode()?

▼ View answer

Callback refs are preferred over the `findDOMNode()` API, due to the fact that `findDOMNode()` prevents certain improvements in React in the future.

```
// Legacy approach using findDOMNode()
class MyComponent extends Component {
  componentDidMount() {
    findDOMNode(this).scrollIntoView()
  }

  render() {
    return <div />
  }
}

// Recommended approach using callback refs
class MyComponent extends Component {
  componentDidMount() {
    this.node.scrollIntoView()
  }

  render() {
    return <div ref={node => (this.node = node)} />
  }
}
```

Good to hear

- Callback refs are preferred over `findDOMNode()` .

Additional Links

- React docs on Refs and the DOM

⬆ Back to top

What is the `children` prop?

▼ View answer

`children` is part of the props object passed to components that allows components to be passed as data to other components, providing the ability to compose components cleanly. There are a number of methods available in the React API to work with this prop, such as `React.Children.map`, `React.Children.forEach`, `React.Children.count`, `React.Children.only` and `React.Children.toArray`. A simple usage example of the children prop is as follows:

```
function GenericBox({ children }) {
  return <div className="container">{children}</div>
}

function App() {
  return (
    <GenericBox>
      <span>Hello</span> <span>World</span>
    </GenericBox>
  )
}
```

Good to hear

- Children is a prop that allows components to be passed as data to other components.
- The React API provides methods to work with this prop.

Additional Links

- React docs on Children

⬆ Back to top

What is a closure? Can you give a useful example of one?

▼ View answer
A closure is a function defined inside another function and has access to its lexical scope even when it is executing outside its lexical scope. The closure has access to variables in three scopes:

- Variables declared in its own scope
- Variables declared in the scope of the parent function
- Variables declared in the global scope

In JavaScript, all functions are closures because they have access to the outer scope, but most functions don't utilise the usefulness of closures: the persistence of state. Closures are also sometimes called stateful functions because of this.

In addition, closures are the only way to store private data that can't be accessed from the outside in JavaScript. They are the key to the UMD (Universal Module Definition) pattern, which is frequently used in libraries that only expose a public API but keep the implementation details private, preventing name collisions with other libraries or the user's own code.

Good to hear

- Closures are useful because they let you associate data with a function that operates on that data.
- A closure can substitute an object with only a single method.
- Closures can be used to emulate private properties and methods.

Additional Links

- MDN docs for closures
- What is a closure
- I never understood JavaScript closures

⬆ Back to top

What is context?

▼ View answer

Context provides a way to pass data through the component tree without having to pass props down manually at every level. For example, authenticated user, locale preference, UI theme need to be accessed in the application by many components.

```
const { Provider, Consumer } = React.createContext(defaultValue)
```

Good to hear

- Context provides a way to pass data through a tree of React components, without having to manually pass props.
- Context is designed to share data that is considered *global* for a tree of React components.

Additional Links

- React docs on Context

⬆ Back to top

What is event-driven programming?

▼ View answer

Event-driven programming is a paradigm that involves building applications that send and receive events. When the program emits events, the program responds by running any callback functions that are registered to that event and context, passing in associated data to the function. With this pattern, events can be emitted into the wild without throwing errors even if no functions are subscribed to it.

A common example of this is the pattern of elements listening to DOM events such as `click` and `mouseenter`, where a callback function is run when the event occurs.

```
document.addEventListener("click", function(event) {
  // This callback function is run when the user
  // clicks on the document.
})
```

Without the context of the DOM, the pattern may look like this:

```
const hub = createEventHub()
hub.on("message", function(data) {
  console.log(`${data.username} said ${data.text}`)
})
hub.emit("message", {
  username: "John",
  text: "Hello?"
})
```

With this implementation, `on` is the way to *subscribe* to an event, while `emit` is the way to *publish* the event.

Good to hear

- Follows a publish-subscribe pattern.
- Responds to events that occur by running any callback functions subscribed to the event.
- Show how to create a simple pub-sub implementation with JavaScript.

Additional Links

- MDN docs on Events and Handlers
- Understanding Node.js event-driven architecture

⬆ Back to top

What are fragments?

▶ View answer

⬆ Back to top

What is functional programming?

▶ View answer

⬆ Back to top

Explain the differences between imperative and declarative programming.

▶ View answer

⬆ Back to top

What is memoization?

▶ View answer

⬆ Back to top

How do you ensure methods have the correct `this` context in React component classes?

▶ View answer

⬆ Back to top

Contrast mutable and immutable values, and mutating vs non-mutating methods.

▶ View answer

⬆ Back to top

What is the only value not equal to itself in JavaScript?

▶ View answer

⬆ Back to top

What is the event loop in Node.js?

▶ View answer

⬆ Back to top

Create a function `pipe` that performs left-to-right function composition by returning a function that accepts one argument.

```
const square = v => v * v
const double = v => v * 2
const addOne = v => v + 1
const res = pipe(square, double, addOne)
res(3) // 19; addOne(double(square(3)))
```

▶ View answer

⬆ Back to top

What are portals in React?

▶ View answer

⬆ Back to top

What is a pure function?

▶ View answer

⬆ Back to top

What is recursion and when is it useful?

▶ View answer

⬆ Back to top

What are refs in React? When should they be used?

▶ View answer

⬆ Back to top

Explain the difference between a static method and an instance method.

▶ View answer

⬆ Back to top

What is the `this` keyword and how does it work?

▶ View answer

⬆ Back to top

What is the purpose of JavaScript UI libraries/frameworks like React, Vue, Angular, Hyperapp, etc?

▶ View answer

⬆ Back to top

What does `'use strict'` do and what are some of the key benefits to using it?

▶ View answer

⬆ Back to top

What is a virtual DOM and why is it used in libraries/frameworks?

▶ View answer

⬆ Back to top

React

What is the difference between an element and a component in React?

▶ View answer

⬆ Back to top

What does lifting state up in React mean?

▶ View answer

⬆ Back to top

How do you write comments inside a JSX tree in React?

▶ View answer

⬆ Back to top

What is a stateful component in React?

▶ View answer

⬆ Back to top

What is a stateless component?

▶ View answer

⬆ Back to top

What is the purpose of callback function as an argument of `setState` ?

▶ View answer

⬆ Back to top

Why does React use `className` instead of `class` like in HTML?

▶ View answer

⬆ Back to top

What is the difference between HTML and React event handling?

▶ View answer

⬆ Back to top

What are inline conditional expressions?

▶ View answer

⬆ Back to top

What is a key? What are the benefits of using it in lists?

▶ View answer

⬆ Back to top

What are the lifecycle methods in React?

▶ View answer

⬆ Back to top

What are the different phases of the component lifecycle in React?

▶ View answer

⬆ Back to top

How do you pass an argument to an event handler or callback?

▶ View answer

⬆ Back to top

Which is the preferred option between callback refs and findDOMNode()?

▶ View answer

⬆ Back to top

What is the `children` prop?

▶ View answer

⬆ Back to top

What is context?

▶ View answer

⬆ Back to top

What are error boundaries in React?

▶ View answer

⬆ Back to top

What are fragments?

▶ View answer

⬆ Back to top

What are higher-order components?

▶ View answer

⬆ Back to top

How do you ensure methods have the correct `this` context in React component classes?

▶ View answer

⬆ Back to top

What are portals in React?

▶ View answer

⬆ Back to top

How to apply prop validation in React?

▶ View answer

⬆ Back to top

What are refs in React? When should they be used?

▶ View answer

⬆ Back to top

HTML

What is the purpose of the `alt` attribute on images?

▶ View answer

⬆ Back to top

What is the purpose of cache busting and how can you achieve it?

▶ View answer

⬆ Back to top

Can a web page contain multiple `<header>` elements? What about `<footer>` elements?

▶ View answer

⬆ Back to top

Briefly describe the correct usage of the following HTML5 semantic elements: `<header>` , `<article>` , `<section>` , `<footer>`

▶ View answer

⬆ Back to top

What are `defer` and `async` attributes on a `<script>` tag?

▶ View answer

⬆ Back to top

What is the DOM?

▶ View answer

⬆ Back to top

Discuss the differences between an HTML specification and a browser's implementation thereof.

▶ View answer

⬆ Back to top

What is the difference between HTML and React event handling?

▶ View answer

⬆ Back to top

What are some differences that XHTML has compared to HTML?

▶ View answer

⬆ Back to top

Where and why is the `rel="noopener"` attribute used?

▶ View answer

⬆ Back to top

What is HTML5 Web Storage? Explain `localStorage` and `sessionStorage` .

▶ View answer

⬆ Back to top

CSS

What is CSS BEM?

▶ View answer

⬆ Back to top

What are the advantages of using CSS preprocessors?

▶ View answer

⬆ Back to top

Using flexbox, create a 3-column layout where each column takes up a `col-{n} / 12` ratio of the container.

```html
<div class="row">
  <div class="col-2"></div>
  <div class="col-7"></div>
  <div class="col-3"></div>
</div>
```

▶ View answer

⬆ Back to top

Can you name the four types of `@media` properties?

▶ View answer

⬆ Back to top

Describe the layout of the CSS Box Model and briefly describe each component.

▶ View answer

⬆ Back to top

What is the difference between `em` and `rem` units?

▶ View answer

⬆ Back to top

What are the advantages of using CSS sprites and how are they utilized?

▶ View answer

⬆ Back to top

What is the difference between '+' and '~' sibling selectors?.

▶ View answer

⬆ Back to top

Can you describe how CSS specificity works?

▶ View answer

⬆ Back to top

What is a focus ring? What is the correct solution to handle them?

▶ View answer

⬆ Back to top

Node

NodeJS often uses a callback pattern where if an error is encountered during execution, this error is passed as the first argument to the callback. What are the advantages of this pattern?

```
fs.readFile(filePath, function(err, data) {
  if (err) {
    // handle the error, the return is important here
    // so execution stops here
    return console.log(err)
  }
  // use the data object
  console.log(data)
})
```

▶ View answer

⬆ Back to top

What is REST?

▶ View answer

⬆ Back to top

How can you avoid callback hells?

```
getData(function(a) {
  getMoreData(a, function(b) {
    getMoreData(b, function(c) {
      getMoreData(c, function(d) {
        getMoreData(d, function(e) {
          // ...
        })
      })
    })
  })
})
```

▶ View answer

⬆ Back to top

What is the event loop in Node.js?

▶ View answer

⬆ Back to top

Security

What is a cross-site scripting attack (XSS) and how do you prevent it?

▶ View answer

⬆ Back to top

License