Star on GitHub

Category All

Sort by Expertise

EASY

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

Hide answer

The `alt` attribute provides alternative information for an image if a user cannot view it. The `alt` attribute should be used to describe any images except those which only serve a decorative purposes, in which case it should be left empty.

Good to hear

- Decorative images should have an empty `alt` attribute.
- Web crawlers use `alt` tags to understand image content, so they are considered important for Search Engine Optimization (SEO).
- Put the `.` at the end of `alt` tag to improve accessibility.

EASY

What is CSS BEM?

Hide answer

The BEM methodology is a naming convention for CSS classes in order to keep CSS more maintainable by defining namespaces to solve scoping issues. BEM stands for Block Element Modifier which is an explanation for its structure. A Block is a standalone component that is reusable across projects and acts as a "namespace" for sub components (Elements). Modifiers are used as flags when a Block or Element is in a certain state or is different in structure or style.

```
/* block component */
.block {
}

/* element */
.block__element {
}

/* modifier */
.block__element--modifier {
}
```

Here is an example with the class names on markup:

```
<nav class="navbar">
  <a href="/" class="navbar__link navbar__link--active"></a>
  <a href="/" class="navbar__link"></a>
  <a href="/" class="navbar__link"></a>
</nav>
```

In this case, `navbar` is the Block, `navbar__link` is an Element that makes no sense outside of the `navbar` component, and `navbar__link--active` is a Modifier that indicates a different state for the `navbar__link` Element.

Since Modifiers are verbose, many opt to use `is-*` flags instead as modifiers.

```
<a href="/" class="navbar__link is-active"></a>
```

These must be chained to the Element and never alone however, or there will be scope issues.

```
.navbar__link.is-active {
}
```

Good to hear

- Alternative solutions to scope issues like CSS-in-JS

EASY

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

Hide answer

Browsers have a cache to temporarily store files on websites so they don't need to be re-downloaded again when switching between pages or reloading the same page. The server is set up to send headers that tell the browser to store the file for a given amount of time. This greatly increases website speed and preserves bandwidth.

However, it can cause problems when the website has been changed by developers because the user's cache still references old files. This can either leave them with old functionality or break a website if the cached CSS and JavaScript files are referencing elements that no longer exist, have moved or have been renamed.

Cache busting is the process of forcing the browser to download the new files. This is done by naming the file something different to the old file.

A common technique to force the browser to re-download the file is to append a query string to the end of the file.

- `src="js/script.js"` => `src="js/script.js?v=2"`

The browser considers it a different file but prevents the need to change the file name.

EASY

What are the advantages of using CSS preprocessors?

Hide answer

CSS preprocessors add useful functionality that native CSS does not have, and generally make CSS neater and more maintainable by enabling DRY (Don't

Repeat Yourself) principles. Their terse syntax for nested selectors cuts down on repeated code. They provide variables for consistent theming (however, CSS variables have largely replaced this functionality) and additional tools like color functions (`lighten`, `darken`, `transparentize`, etc), mixins, and loops that make CSS more like a real programming language and gives the developer more power to generate complex CSS.

Good to hear

- They allow us to write more maintainable and scalable CSS
- Some disadvantages of using CSS preprocessors (setup, re-compilation time can be slow etc.)

EASY

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

Hide 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.

EASY

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

Hide 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.

EASY

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

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

Hide answer

Set the `.row` parent to `display: flex;` and use the `flex` shorthand property to give the column classes a `flex-grow` value that corresponds to its ratio value.

```
.row {
  display: flex;
}

.col-2 {
  flex: 2;
}

.col-7 {
  flex: 7;
}

.col-3 {
  flex: 3;
}
```

EASY

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

Hide answer

Yes to both. The W3 documents state that the tags represent the header(`<header>`) and footer(`<footer>`) areas of their nearest ancestor "section". So not only can the page `<body>` contain a header and a footer, but so can every `<article>` and `<section>` element.

Good to hear

- W3 recommends having as many as you want, but only 1 of each for each "section" of your page, i.e. body, section etc.

EASY

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

Hide answer

- `<header>` is used to contain introductory and navigational information about a section of the page. This can include the section heading, the author's name, time and date of publication, table of contents, or other navigational information.
- `<article>` is meant to house a self-contained composition that can logically be independently recreated outside of the page without losing its meaning. Individual blog posts or news stories are good examples.

- `<section>` is a flexible container for holding content that shares a common informational theme or purpose.
- `<footer>` is used to hold information that should appear at the end of a section of content and contain additional information about the section. Author's name, copyright information, and related links are typical examples of such content.

Good to hear

- Other semantic elements are `<form>` and `<table>`

EASY

What does lifting state up in React mean?

Hide answer

When several components need to share the same data, then it is recommended to lift the shared state up to their closest common ancestor. For example, if two child components share the same data, it is recommended to move the shared state to parent instead of maintaining the local state in both child components.

EASY

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

Hide answer

- `all`, which applies to all media type devices

- `print`, which only applies to printers
- `screen`, which only applies to screens (desktops, tablets, mobile etc.)
- `speech`, which only applies to screenreaders

EASY

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

Hide 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
```

EASY

In which states can a Promise be?

Hide 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.

EASY

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

Hide answer

Comments must be wrapped inside curly braces `{}` and use the `/* */` syntax.

```
const tree = (
  <div>
    {/* Comment */}
    <p>Text</p>
  </div>
)
```

EASY

What is a stateful component in React?

Hide 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()`.

EASY

What is a stateless component?

Hide 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.

INTERMEDIATE

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

Hide answer

If neither attribute is present, the script is downloaded and executed synchronously, and will halt parsing of the document until it has finished executing (default behavior). Scripts are downloaded and executed in the order they are encountered.

The `defer` attribute downloads the script while the document is still parsing but waits until the document has finished parsing before executing it, equivalent to

executing inside a `DOMContentLoaded` event listener. `defer` scripts will execute in order.

The `async` attribute downloads the script during parsing the document but will pause the parser to execute the script before it has fully finished parsing. `async` scripts will not necessarily execute in order.

Note: both attributes must only be used if the script has a `src` attribute (i.e. not an inline script).

```
<script src="myscript.js"></script>
<script src="myscript.js" defer></script>
<script src="myscript.js" async></script>
```

Good to hear

- Placing a `defer` script in the `<head>` allows the browser to download the script while the page is still parsing, and is therefore a better option than placing the script before the end of the body.
- If the scripts rely on each other, use `defer`.
- If the script is independent, use `async`.
- Use `defer` if the DOM must be ready and the contents are not placed within a `DOMContentLoaded` listener.

INTERMEDIATE

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 }
)
```

Hide 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))
  )
```

INTERMEDIATE

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 }
```

Hide 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)
```

INTERMEDIATE

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

Hide 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.

INTERMEDIATE

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

Hide 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

INTERMEDIATE

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

Hide answer

React's philosophy in the beginning was to align with the browser DOM API rather than HTML, since that more closely represents how elements are created. Setting a `class` on an element meant using the `className` API:

```
const element = document.createElement("div")
element.className = "hello"
```

Additionally, before ES5, reserved words could not be used in objects:

```
const element = {
  attributes: {
    class: "hello"
  }
}
```

In IE8, this will throw an error.

In modern environments, destructuring will throw an error if trying to assign to a variable:

```
const { class } = this.props // Error
const { className } = this.props // All good
const { class: className } = this.props // All good, but cumbersome!
```

However, `class` *can* be used as a prop without problems, as seen in other libraries like Preact. React currently allows you to use `class`, but will throw a warning and convert it to `className` under the hood. There is currently an open thread (as of January 2019) discussing changing `className` to `class` to reduce confusion.

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 }
```

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.

INTERMEDIATE

How do you compare two objects in JavaScript?

Hide 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)

INTERMEDIATE

What is CORS?

Hide 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.

INTERMEDIATE

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

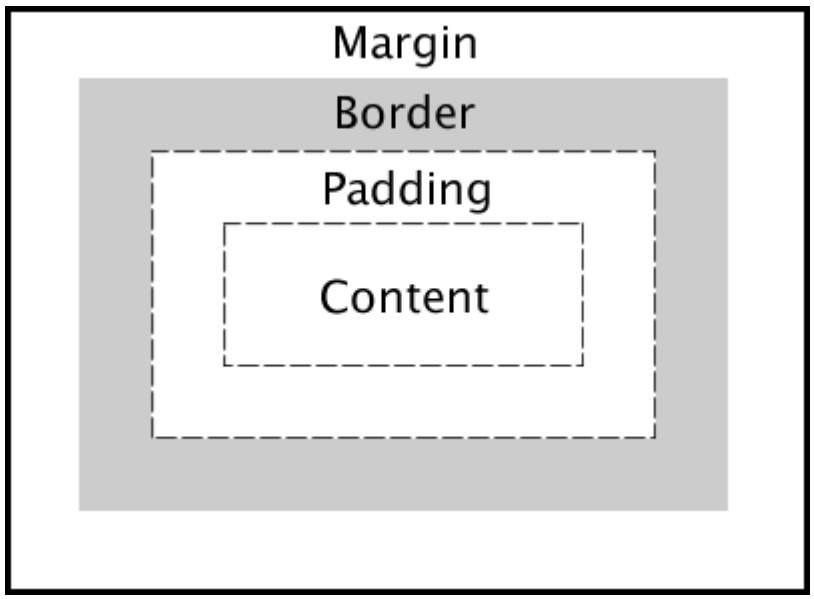
Hide answer

`Content`: The inner-most part of the box filled with content, such as text, an image, or video player. It has the dimensions `content-box width` and `content-box height`.

`Padding`: The transparent area surrounding the content. It has dimensions `padding-box width` and `padding-box height`.

`Border`: The area surrounding the padding (if any) and content. It has dimensions `border-box width` and `border-box height`.

Margin: The transparent outer-most layer that surrounds the border. It separates the element from other elements in the DOM. It has dimensions `margin-box width` and `margin-box height`.



Good to hear

- This is a very common question asked during front-end interviews and while it may seem easy, it is critical you know it well!
- Shows a solid understanding of spacing and the DOM

INTERMEDIATE

What is the DOM?

Hide 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.

INTERMEDIATE

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

Hide answer

Both `em` and `rem` units are based on the `font-size` CSS property. The only difference is where they inherit their values from.

- `em` units inherit their value from the `font-size` of the parent element
- `rem` units inherit their value from the `font-size` of the root element (`html`)

In most browsers, the `font-size` of the root element is set to `16px` by default.

Good to hear

- Benefits of using `em` and `rem` units

INTERMEDIATE

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

Hide 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

INTERMEDIATE

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

Hide 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

INTERMEDIATE

What are truthy and falsy values in JavaScript?

Hide 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
```

INTERMEDIATE

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

Hide 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),
    []
  )
```

INTERMEDIATE

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

Hide 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) < epsilc
approxEqual(0.1 + 0.2, 0.3) // true
```

Good to hear

- A simple solution to this problem

INTERMEDIATE

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

Hide 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.

INTERMEDIATE

What will the console log in this example?

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

Hide 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

INTERMEDIATE

How does hoisting work in JavaScript?

Hide 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

INTERMEDIATE

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

Hide answer

HTML specifications such as `HTML5` define a set of rules that a document must adhere to in order to be "valid" according to that specification. In addition, a specification provides instructions on how a browser must interpret and render such a document.

A browser is said to "support" a specification if it handles valid documents according to the rules of the specification. As of yet, no browser supports all aspects of the `HTML5` specification (although all of the major browser support most of it), and as a result, it is necessary for the developer to confirm whether the aspect they are making use of will be supported by all of the browsers on which they hope to display their content. This is why cross-browser support continues to be a headache for developers, despite the improved specificiations.

Good to hear

- `HTML5` defines some rules to follow for an invalid `HTML5` document (i.e., one that contains syntactical errors)
- However, invalid documents may contain anything, so it's impossible for the specification to handle all possibilities comprehensively.
- Thus, many decisions about how to handle malformed documents are left up to the browser.

INTERMEDIATE

What is the difference between HTML and React event handling?

Hide 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.

INTERMEDIATE

What are some differences that XHTML has compared to HTML?

Hide answer

Some of the key differences are:

- An XHTML element must have an XHTML `<DOCTYPE>`
- Attributes values must be enclosed in quotes
- Attribute minimization is forbidden (e.g. one has to use `checked="checked"` instead of `checked`)
- Elements must always be properly nested
- Elements must always be closed
- Special characters must be escaped

Good to hear

- Any element can be self-closed
- Tags ands attributes are case-sensitive, usually lowercase

INTERMEDIATE

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

Hide 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

INTERMEDIATE

What are inline conditional expressions?

Hide 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>
```

```
    )
  }
```

INTERMEDIATE

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

Hide 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.

INTERMEDIATE

What is the difference between lexical scoping and dynamic scoping?

Hide 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.

INTERMEDIATE

What are the lifecycle methods in React?

Hide answer

`getDerivedStateFromProps`: Executed before rendering on the initial mount and all component updates. Used to update the state based on changes in props over time. Has rare use cases, like tracking component animations during the lifecycle. There are only few cases where this makes sense to use over other lifecycle methods. It expects to return an object that will be the the new state, or

null to update nothing. This method does not have access to the component instance either.

`componentDidMount`: Executed after first rendering and here all AJAX requests, DOM or state updates, and set up eventListeners should occur.

`shouldComponentUpdate`: Determines if the component will be updated or not. By default, it returns true. If you are sure that the component doesn't need to render after state or props are updated, you can return a false value. It is a great place to improve performance as it allows you to prevent a rerender if component receives new prop.

`getSnapshotBeforeUpdate`: Invoked right after a component render happens because of an update, before `componentDidUpdate`. Any value returned from this method will be passed to `componentDidUpdate`.

`componentDidUpdate`: Mostly it is used to update the DOM in response to prop or state changes.

`componentWillUnmount`: It will be used to cancel any outgoing network requests, or remove all event listeners associated with the component.

`componentDidCatch`: Used in error boundaries, which are components that implement this method. It allows the component to catch JavaScript errors anywhere in the *child* component tree (below this component), log errors, and display a UI with error information.

INTERMEDIATE

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

Hide answer

There are four different phases of component's lifecycle:

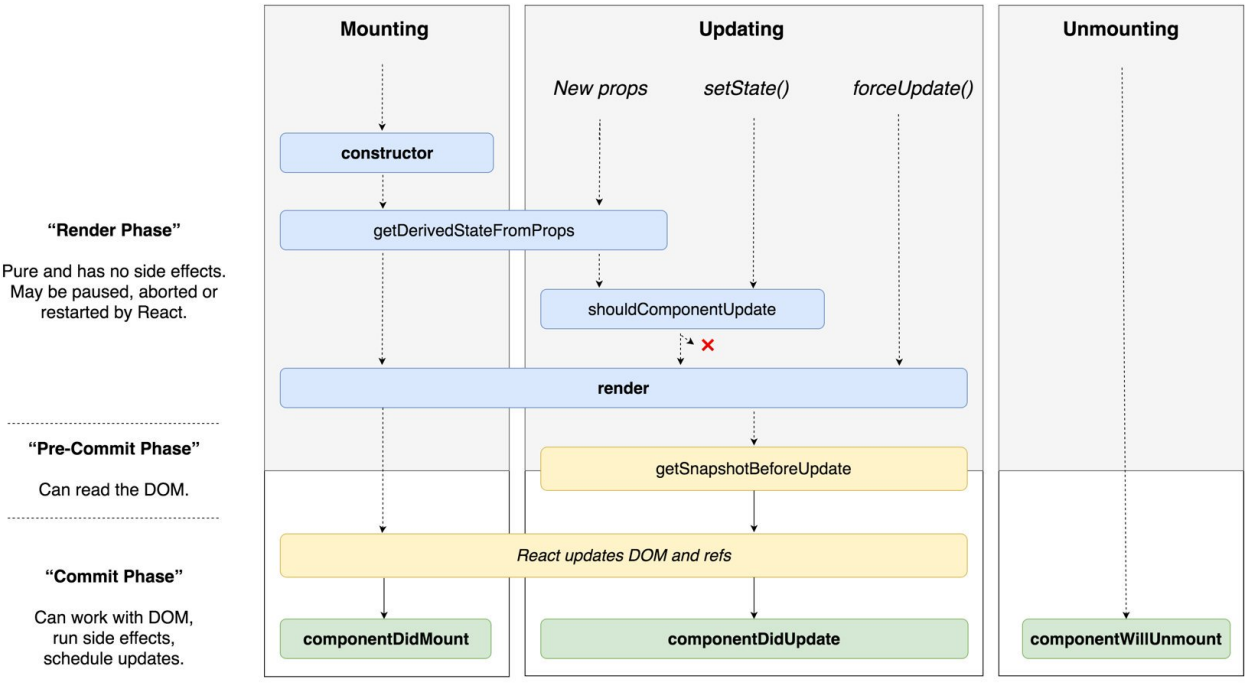
Initialization: In this phase, the component prepares setting up the initial state and default props.

Mounting: The react component is ready to mount to the DOM. This phase covers the `getDerivedStateFromProps` and `componentDidMount` lifecycle methods.

Updating: In this phase, the component gets updated in two ways, sending the new props and updating the state. This phase covers the `getDerivedStateFromProps`, `shouldComponentUpdate`, `getSnapshotBeforeUpdate` and `componentDidUpdate` lifecycle methods.

Unmounting: In this last phase, the component is not needed and gets unmounted from the browser DOM. This phase includes the `componentWillUnmount` lifecycle method.

Error Handling: In this phase, the component is called whenever there's an error during rendering, in a lifecycle method, or in the constructor for any child component. This phase includes the `componentDidCatch` lifecycle method.



INTERMEDIATE

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

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

Hide 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

INTERMEDIATE

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

Hide 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).

INTERMEDIATE

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)
})
```

Hide 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.

INTERMEDIATE

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

Hide 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`.

RECOMMENDED RESOURCE

Frontend Masters Handbook

This is a guide that anyone could use to learn about the practice of front-end development. It broadly outlines and discusses the practice of front-end engineering: how to learn it and what tools are used when practicing it.

INTERMEDIATE

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

Hide 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)
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.

```
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.

```
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.

```
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.

INTERMEDIATE

What is the difference between a parameter and an argument?

Hide 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).

INTERMEDIATE

Does JavaScript pass by value or by reference?

Hide 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

INTERMEDIATE

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

Hide 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)} />
```

INTERMEDIATE

What are Promises?

Hide 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!

INTERMEDIATE

How does prototypal inheritance differ from classical inheritance?

Hide 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()`.

INTERMEDIATE

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)
```

Hide 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

INTERMEDIATE

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

Hide answer

The `rel="noopener"` is an attribute used in `<a>` elements (hyperlinks). It prevents pages from having a `window.opener` property, which would otherwise

point to the page from where the link was opened and would allow the page opened from the hyperlink to manipulate the page where the hyperlink is.

Good to hear

- `rel="noopener"` is applied to hyperlinks.
- `rel="noopener"` prevents opened links from manipulating the source page.

INTERMEDIATE

What is REST?

Hide answer

REST (REpresentational State Transfer) is a software design pattern for network architecture. A RESTful web application exposes data in the form of information about its resources.

Generally, this concept is used in web applications to manage state. With most applications, there is a common theme of reading, creating, updating, and destroying data. Data is modularized into separate tables like `posts`, `users`, `comments`, and a RESTful API exposes access to this data with:

- An identifier for the resource. This is known as the endpoint or URL for the resource.
- The operation the server should perform on that resource in the form of an HTTP method or verb. The common HTTP methods are GET, POST, PUT, and DELETE.

Here is an example of the URL and HTTP method with a `posts` resource:

- Reading: `/posts/` => GET

- Creating: `/posts/new` => POST
- Updating: `/posts/:id` => PUT
- Destroying: `/posts/:id` => DELETE

Good to hear

- Alternatives to this pattern like GraphQL

INTERMEDIATE

What does the following function return?

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

Hide 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

INTERMEDIATE

Are semicolons required in JavaScript?

Hide 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.

INTERMEDIATE

What is short-circuit evaluation in JavaScript?

Hide 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.

INTERMEDIATE

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

Hide answer

CSS sprites combine multiple images into one image, limiting the number of HTTP requests a browser has to make, thus improving load times. Even under the new HTTP/2 protocol, this remains true.

Under HTTP/1.1, at most one request is allowed per TCP connection. With HTTP/1.1, modern browsers open multiple parallel connections (between 2 to 8) but it is limited. With HTTP/2, all requests between the browser and the server are multiplexed on a single TCP connection. This means the cost of opening and closing multiple connections is mitigated, resulting in a better usage of the TCP connection and limits the impact of latency between the client and server. It could then become possible to load tens of images in parallel on the same TCP connection.

However, according to benchmark results, although HTTP/2 offers 50% improvement over HTTP/1.1, in most cases the sprite set is still faster to load than individual images.

To utilize a spritesheet in CSS, one would use certain properties, such as `background-image`, `background-position` and `background-size` to ultimately alter the `background` of an element.

Good to hear

- `background-image`, `background-position` and `background-size` can be used to utilize a spritesheet.

INTERMEDIATE

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

Hide 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.

INTERMEDIATE

What does the following code evaluate to?

```
typeof typeof 0
```

Hide answer

It evaluates to `"string"`.

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

INTERMEDIATE

What are JavaScript data types?

Hide 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

INTERMEDIATE

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

Hide 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.

```
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.

INTERMEDIATE

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

Hide 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.

HARD

What is Big O Notation?

Hide 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]]] : a
  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.

HARD

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) {
          // ...
        })
      })
    })
  })
})
```

Hide 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.

HARD

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

Hide 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()`.

HARD

What is the `children` prop?

Hide 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:

```jsx
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.

HARD

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

Hide 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.

HARD

What is context?

Hide 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.

HARD

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

Hide answer

The General Sibling Selector `~` selects all elements that are siblings of a specified element.

The following example selects all `<p>` elements that are siblings of `<div>` elements:

```
div ~ p {
  background-color: blue;
}
```

The Adjacent Sibling Selector `+` selects all elements that are the adjacent siblings of a specified element.

The following example will select all `<p>` elements that are placed immediately after `<div>` elements:

```
div + p {
  background-color: red;
}
```

HARD

Can you describe how CSS specificity works?

Hide answer

Assuming the browser has already determined the set of rules for an element, each rule is assigned a matrix of values, which correspond to the following from highest to lowest specificity:

- Inline rules (binary - 1 or 0)
- Number of id selectors
- Number of class, pseudo-class and attribute selectors
- Number of tags and pseudo-element selectors

When two selectors are compared, the comparison is made on a per-column basis (e.g. an id selector will always be higher than any amount of class selectors, as ids have higher specificity than classes). In cases of equal specificity between multiple rules, the rules that comes last in the page's style sheet is deemed more specific and therefore applied to the element.

Good to hear

- Specificity matrix: [inline, id, class/pseudo-class/attribute, tag/pseudo-element]
- In cases of equal specificity, last rule is applied

HARD

What are error boundaries in React?

Hide answer

Error boundaries are React components that catch JavaScript errors anywhere in their child component tree, log those errors, and display a fallback UI instead of the component tree that crashed.

A class component becomes an error boundary if it defines a new lifecycle method called `componentDidCatch`.

```
class ErrorBoundary extends React.Component {
  constructor(props) {
    super(props)
    this.state = { hasError: false }
  }

  componentDidCatch(error, info) {
    // Display fallback UI
    this.setState({ hasError: true })
    // You can also log the error to an error reporting service
    logErrorToMyService(error, info)
  }

  render() {
    if (this.state.hasError) {
      // You can render any custom fallback UI
      return <h1>Something went wrong.</h1>
    }
    return this.props.children
  }
}
```

HARD

What is event-driven programming?

Hide 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.

HARD

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

Hide answer

A focus ring is a visible outline given to focusable elements such as buttons and anchor tags. It varies depending on the vendor, but generally it appears as a blue outline around the element to indicate it is currently focused.

In the past, many people specified `outline: 0;` on the element to remove the focus ring. However, this causes accessibility issues for keyboard users because the focus state may not be clear. When not specified though, it causes an unappealing blue ring to appear around an element.

In recent times, frameworks like Bootstrap have opted to use a more appealing `box-shadow` outline to replace the default focus ring. However, this is still not ideal for mouse users.

The best solution is an upcoming pseudo-selector `:focus-visible` which can be polyfilled today with JavaScript. It will only show a focus ring if the user is using a keyboard and leave it hidden for mouse users. This keeps both aesthetics for mouse use and accessibility for keyboard use.

HARD

What are fragments?

Hide answer

Fragments allow a React component to return multiple elements without a wrapper, by grouping the children without adding extra elements to the DOM. Fragments offer better performance, lower memory usage, a cleaner DOM and can help in dealing with certain CSS mechanisms (e.g. tables, Flexbox and Grid).

```
render() {
  return (
    <React.Fragment>
      <ChildA />
      <ChildB />
      <ChildC />
    </React.Fragment>
  );
}

// Short syntax supported by Babel 7
render() {
  return (
    <>
      <ChildA />
```

```
        <ChildB />
        <ChildC />
      </>
    );
  }
```

Good to hear

- Fragments group multiple elements returned from a component, without adding a DOM element around them.

HARD

What is functional programming?

Hide answer

Functional programming is a paradigm in which programs are built in a declarative manner using pure functions that avoid shared state and mutable data. Functions that always return the same value for the same input and don't produce side effects are the pillar of functional programming. Many programmers consider this to be the best approach to software development as it reduces bugs and cognitive load.

Good to hear

- Cleaner, more concise development experience
- Simple function composition
- Features of JavaScript that enable functional programming (`.map`, `.reduce` etc.)
- JavaScript is multi-paradigm programming language (Object-Oriented Programming and Functional Programming live in harmony)

HARD

What are higher-order components?

Hide answer

A higher-order component (HOC) is a function that takes a component as an argument and returns a new component. It is a pattern that is derived from React's compositional nature. Higher-order components are like **pure components** because they accept any dynamically provided child component, but they won't modify or copy any behavior from their input components.

```
const EnhancedComponent = higherOrderComponent(WrappedComponent)
```

Good to hear

- They can be used for state abstraction and manipulation, props manipulation, render high jacking, etc.

HARD

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

Hide answer

With HTML5, web pages can store data locally within the user's browser. The data is stored in name/value pairs, and a web page can only access data stored by itself.

Differences between `localStorage` and `sessionStorage` regarding lifetime:

- Data stored through `localStorage` is permanent: it does not expire and remains stored on the user's computer until a web app deletes it or the user asks the browser to delete it.
- `sessionStorage` has the same lifetime as the top-level window or browser tab in which the data got stored. When the tab is permanently closed, any data stored through `sessionStorage` is deleted.

Differences between `localStorage` and `sessionStorage` regarding storage scope: Both forms of storage are scoped to the document origin so that documents with different origins will never share the stored objects.

- `sessionStorage` is also scoped on a per-window basis. Two browser tabs with documents from the same origin have separate `sessionStorage` data.
- Unlike in `localStorage`, the same scripts from the same origin can't access each other's `sessionStorage` when opened in different tabs.

Good to hear

- Earlier, this was done with cookies.
- The storage limit is far larger (at least 5MB) than with cookies and its faster.
- The data is never transferred to the server and can only be used if the client specifically asks for it.

HARD

Explain the differences between imperative and declarative programming.

Hide answer

These two types of programming can roughly be summarized as:

- Imperative: **how** to achieve something
- Declarative: **what** should be achieved

A common example of declarative programming is CSS. The developer specifies CSS properties that describe what something should look like rather than how to achieve it. The "how" is abstracted away by the browser.

On the other hand, imperative programming involves the steps required to achieve something. In JavaScript, the differences can be contrasted like so:

Imperative

```
const numbers = [1, 2, 3, 4, 5]
const numbersDoubled = []
for (let i = 0; i < numbers.length; i++) {
  numbersDoubled[i] = numbers[i] * 2
}
```

We manually loop over the numbers of the array and assign the new index as the number doubled.

Declarative

```
const numbers = [1, 2, 3, 4, 5]
const numbersDoubled = numbers.map(n => n * 2)
```

We declare that the new array is mapped to a new one where each value is doubled.

Good to hear

- Declarative programming often works with functions and expressions. Imperative programming frequently uses statements and relies on low-level

features that cause mutations, while declarative programming has a strong focus on abstraction and purity.
- Declarative programming is more terse and easier to process at a glance.

HARD

What is memoization?

Hide answer

Memoization is the process of caching the output of function calls so that subsequent calls are faster. Calling the function again with the same input will return the cached output without needing to do the calculation again.

A basic implementation in JavaScript looks like this:

```
const memoize = fn => {
  const cache = new Map()
  return value => {
    const cachedResult = cache.get(value)
    if (cachedResult !== undefined) return cachedResult
    const result = fn(value)
    cache.set(value, result)
    return result
  }
}
```

Good to hear

- The above technique returns a unary function even if the function can take multiple arguments.

- The first function call will be slower than usual because of the overhead created by checking if a cached result exists and setting a result before returning the value.
- Memoization increases performance on subsequent function calls but still needs to do work on the first call.

HARD

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

Hide answer

In JavaScript classes, the methods are not bound by default. This means that their `this` context can be changed (in the case of an event handler, to the element that is listening to the event) and will not refer to the component instance. To solve this, `Function.prototype.bind()` can be used to enforce the `this` context as the component instance.

```
constructor(props) {
  super(props);
  this.handleClick = this.handleClick.bind(this);
}

handleClick() {
  // Perform some logic
}
```

- The `bind` approach can be verbose and requires defining a `constructor`, so the new public class fields syntax is generally preferred:

```
handleClick = () => {
  console.log('this is:', this);
}

render() {
  return (
    <button onClick={this.handleClick}>
      Click me
    </button>
  );
}
```

- You can also use an inline arrow function, because lexical `this` (referring to the component instance) is preserved:

```
<button onClick={e => this.handleClick(e)}>Click me</button>
```

Note that extra re-rendering can occur using this technique because a new function reference is created on render, which gets passed down to child components and breaks `shouldComponentUpdate` / `PureComponent` shallow equality checks to prevent unnecessary re-renders. In cases where performance is important, it is preferred to go with `bind` in the constructor, or the public class fields syntax approach, because the function reference remains constant.

Good to hear

- You can either bind methods to the component instance context in the constructor, use public class fields syntax, or use inline arrow functions.

HARD

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

Hide answer

The two terms can be contrasted as:

- Mutable: subject to change
- Immutable: cannot change

In JavaScript, objects are mutable while primitive values are immutable. This means operations performed on objects can change the original reference in some way, while operations performed on a primitive value cannot change the original value.

All `String.prototype` methods do not have an effect on the original string and return a new string. On the other hand, while some methods of `Array.prototype` do not mutate the original array reference and produce a fresh array, some cause mutations.

```
const myString = "hello!"
myString.replace("!", "") // returns a new string, cannot mutate the origina

const originalArray = [1, 2, 3]
originalArray.push(4) // mutates originalArray, now [1, 2, 3, 4]
originalArray.concat(4) // returns a new array, does not mutate the original
```

Good to hear

- List of mutating and non-mutating array methods

HARD

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

Hide answer

`NaN` (Not-a-Number) is the only value not equal to itself when comparing with any of the comparison operators. `NaN` is often the result of meaningless math computations, so two `NaN` values make no sense to be considered equal.

Good to hear

- The difference between `isNaN()` and `Number.isNaN()`
- `const isNaN = x => x !== x`

HARD

What is the event loop in Node.js?

Hide answer

The event loop handles all async callbacks. Callbacks are queued in a loop, while other code runs, and will run one by one when the response for each one has been received.

Good to hear

- The event loop allows Node.js to perform non-blocking I/O operations, despite the fact that JavaScript is single-threaded

HARD

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)))
```

Hide answer

Gather all supplied arguments using the rest operator `...` and return a unary function that uses `Array.prototype.reduce()` to run the value through the series of functions before returning the final value.

```
const pipe = (...fns) => x => fns.reduce((v, fn) => fn(v), x)
```

Good to hear

- Function composition is the process of combining two or more functions to produce a new function.

HARD

What are portals in React?

Hide answer

Portal are the recommended way to render children into a DOM node that exists outside the DOM hierarchy of the parent component.

```
ReactDOM.createPortal(child, container)
```

The first argument (`child`) is any renderable React child, such as an element, string, or fragment. The second argument (`container`) is a DOM element.

HARD

How to apply prop validation in React?

Hide answer

When the application is running in development mode, React will automatically check for all props that we set on components to make sure they are the correct data type. For incorrect data types, it will generate warning messages in the console for development mode. They are stripped in production mode due to their performance impact. Required props are defined with `isRequired`.

For example, we define `propTypes` for component as below:

```
import PropTypes from "prop-types"

class User extends React.Component {
  static propTypes = {
    name: PropTypes.string.isRequired,
    age: PropTypes.number.isRequired
  }

  render() {
    return (
      <h1>Welcome, {this.props.name}</h1>
      <h2>Age, {this.props.age}
```

```
      )
    }
  }
```

Good to hear

- We can define custom `propTypes`
- Using `propTypes` is not mandatory. However, it is a good practice and can reduce bugs.

HARD

What is a pure function?

Hide answer

A pure function is a function that satisfies these two conditions:

- Given the same input, the function returns the same output.
- The function doesn't cause side effects outside of the function's scope (i.e. mutate data outside the function or data supplied to the function).

Pure functions can mutate local data within the function as long as it satisfies the two conditions above.

Pure

```
const a = (x, y) => x + y
const b = (arr, value) => arr.concat(value)
const c = arr => [...arr].sort((a, b) => a - b)
```

Impure

```
const a = (x, y) => x + y + Math.random()
const b = (arr, value) => (arr.push(value), arr)
const c = arr => arr.sort((a, b) => a - b)
```

Good to hear

- Pure functions are easier to reason about due to their reliability.
- All functions should be pure unless explicitly causing a side effect (i.e. `setInnerHTML`).
- If a function does not return a value, it is an indication that it is causing side effects.

HARD

What is recursion and when is it useful?

Hide answer

Recursion is the repeated application of a process. In JavaScript, recursion involves functions that call themselves repeatedly until they reach a base condition. The base condition breaks out of the recursion loop because otherwise the function would call itself indefinitely. Recursion is very useful when working with data structures that contain nesting where the number of levels deep is unknown.

For example, you may have a thread of comments returned from a database that exist in a flat array but need to be nested for display in the UI. Each comment is either a top-level comment (no parent) or is a reply to a parent comment. Comments can be a reply of a reply of a reply... we have no knowledge beforehand the number of levels deep a comment may be. This is where recursion can help.

```
const nest = (items, id = null, link = "parent_id") =>
  items
    .filter(item => item[link] === id)
    .map(item => ({ ...item, children: nest(items, item.id) }))

const comments = [
  { id: 1, parent_id: null, text: "First reply to post." },
  { id: 2, parent_id: 1, text: "First reply to comment #1." },
  { id: 3, parent_id: 1, text: "Second reply to comment #1." },
  { id: 4, parent_id: 3, text: "First reply to comment #3." },
  { id: 5, parent_id: 4, text: "First reply to comment #4." },
  { id: 6, parent_id: null, text: "Second reply to post." }
]

nest(comments)
/*
[
  { id: 1, parent_id: null, text: "First reply to post.", children: [...] },
  { id: 6, parent_id: null, text: "Second reply to post.", children: [] }


]
*/
```

In the above example, the base condition is met if `filter()` returns an empty array. The chained `map()` won't invoke the callback function which contains the recursive call, thereby breaking the loop.

Good to hear

- Recursion is useful when working with data structures containing an unknown number of nested structures.
- Recursion must have a base condition to be met that breaks out of the loop or it will call itself indefinitely.

HARD

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

Hide answer

Refs provide a way to access DOM nodes or React elements created in the render method. Refs should be used sparringly, but there are some good use cases for refs, such as:

- Managing focus, text selection, or media playback.
- Triggering imperative animations.
- Integrating with third-party DOM libraries.

Refs are created using `React.createRef()` method and attached to React elements via the `ref` attribute. In order to use refs throughout the component, assign the `ref` to the instance property within the constructor:

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props)
    this.myRef = React.createRef()
  }

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

Refs can also be used in functional components with the help of closures.

Good to hear

- Refs are used to return a reference to an element.
- Refs shouldn't be overused.
- You can create a ref using `React.createRef()` and attach to elements via the `ref` attribute.

HARD

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

Hide answer

Static methods belong to a class and don't act on instances, while instance methods belong to the class prototype which is inherited by all instances of the class and acts on them.

```
Array.isArray // static method of Array
Array.prototype.push // instance method of Array
```

In this case, the `Array.isArray` method does not make sense as an instance method of arrays because we already know the value is an array when working with it.

Instance methods could technically work as static methods, but provide terser syntax:

```
const arr = [1, 2, 3]
arr.push(4)
Array.push(arr, 4)
```

Good to hear

- How to create static and instance methods with ES2015 class syntax

HARD

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

Hide answer

The `this` keyword is an object that represents the context of an executing function. Regular functions can have their `this` value changed with the methods `call()`, `apply()` and `bind()`. Arrow functions implicitly bind `this` so that it refers to the context of its lexical environment, regardless of whether or not its context is set explicitly with `call()`.

Here are some common examples of how `this` works:

Object literals

`this` refers to the object itself inside regular functions if the object precedes the invocation of the function.

Properties set as `this` do not refer to the object.

```
var myObject = {
  property: this,
  regularFunction: function() {
    return this
  },
  arrowFunction: () => {
    return this
  },
  iife: (function() {
    return this
```

```
  })()
}
myObject.regularFunction() // myObject
myObject["regularFunction"]() // my Object

myObject.property // NOT myObject; lexical `this`
myObject.arrowFunction() // NOT myObject; lexical `this`
myObject.iife // NOT myObject; lexical `this`
const regularFunction = myObject.regularFunction
regularFunction() // NOT myObject; lexical `this`
```

Event listeners

`this` refers to the element listening to the event.

```
document.body.addEventListener("click", function() {
  console.log(this) // document.body
})
```

Constructors

`this` refers to the newly created object.

```
class Example {
  constructor() {
    console.log(this) // myExample
  }
}
const myExample = new Example()
```

Manual

With `call()` and `apply()`, `this` refers to the object passed as the first argument.

```
var myFunction = function() {
  return this
}
myFunction.call({ customThis: true }) // { customThis: true }
```

Unwanted `this`

Because `this` can change depending on the scope, it can have unexpected values when using regular functions.

```
var obj = {
  arr: [1, 2, 3],
  doubleArr() {
    return this.arr.map(function(value) {
      // this is now this.arr
      return this.double(value)
    })
  },
  double() {
    return value * 2
  }
}
obj.doubleArr() // Uncaught TypeError: this.double is not a function
```

Good to hear

- In non-strict mode, global `this` is the global object (`window` in browsers), while in strict mode global `this` is `undefined`.
- `Function.prototype.call` and `Function.prototype.apply` set the `this` context of an executing function as the first argument, with `call` accepting a variadic number of arguments thereafter, and `apply` accepting an array as the second argument which are fed to the function in a variadic manner.
- `Function.prototype.bind` returns a new function that enforces the `this` context as the first argument which cannot be changed by other functions.

- If a function requires its `this` context to be changed based on how it is called, you must use the `function` keyword. Use arrow functions when you want `this` to be the surrounding (lexical) context.

HARD

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

Hide answer

The main purpose is to avoid manipulating the DOM directly and keep the state of an application in sync with the UI easily. Additionally, they provide the ability to create components that can be reused when they have similar functionality with minor differences, avoiding duplication which would require multiple changes whenever the structure of a component which is reused in multiple places needs to be updated.

When working with DOM manipulation libraries like jQuery, the data of an application is generally kept in the DOM itself, often as class names or `data` attributes. Manipulating the DOM to update the UI involves many extra steps and can introduce subtle bugs over time. Keeping the state separate and letting a framework handle the UI updates when the state changes reduces cognitive load. Saying you want the UI to look a certain way when the state is a certain value is the declarative way of creating an application, instead of the imperative way of manually updating the UI to reflect the new state.

Good to hear

- The virtual DOM is a representation of the real DOM tree in the form of plain objects, which allows a library to write code as if the entire document is

thrown away and rebuilt on each change, while the real DOM only updates what needs to be changed. Comparing the new virtual DOM against the previous one leads to high efficiency as changing real DOM nodes is costly compared to recalculating the virtual DOM.

- JSX is an extension to JavaScript that provides XML-like syntax to create virtual DOM objects which is transformed to function calls by a transpiler. It simplifies control flow (if statements/ternary expressions) compared to tagged template literals.

HARD

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

Hide answer

Including `'use strict'` at the beginning of your JavaScript source file enables strict mode, which enforces more strict parsing and error handling of JavaScript code. It is considered a good practice and offers a lot of benefits, such as:

- Easier debugging due to eliminating silent errors.
- Disallows variable redefinition.
- Prevents accidental global variables.
- Oftentimes provides increased performance over identical code that is not running in strict mode.
- Simplifies `eval()` and `arguments`.
- Helps make JavaScript more secure.

Good to hear

- Eliminates `this` coercion, throwing an error when `this` references a value of `null` or `undefined`.
- Throws an error on invalid usage of `delete`.
- Prohibits some syntax likely to be defined in future versions of ECMAScript

HARD

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

Hide answer

The virtual DOM (VDOM) is a representation of the real DOM in the form of plain JavaScript objects. These objects have properties to describe the real DOM nodes they represent: the node name, its attributes, and child nodes.

```
<div class="counter">
  <h1>0</h1>
  <button>-</button>
  <button>+</button>
</div>
```

The above markup's virtual DOM representation might look like this:

```
{
  nodeName: "div",
  attributes: { class: "counter" },
  children: [
    {
      nodeName: "h1",
      attributes: {},
      children: [0]
```

```
      },
      {
        nodeName: "button",
        attributes: {},
        children: ["-"]
      },
      {
        nodeName: "button",
        attributes: {},
        children: ["+"]
      }
    ]
  }
```

The library/framework uses the virtual DOM as a means to improve performance. When the state of an application changes, the real DOM needs to be updated to reflect it. However, changing real DOM nodes is costly compared to recalculating the virtual DOM. The previous virtual DOM can be compared to the new virtual DOM very quickly in comparison.

Once the changes between the old VDOM and new VDOM have been calculated by the diffing engine of the framework, the real DOM can be patched efficiently in the least time possible to match the new state of the application.

Good to hear

- Why accessing the DOM can be so costly.

This is a guide that anyone could use to learn about the practice of front-end development. It broadly outlines and discusses the practice of front-end engineering: how to learn it and what tools are used when practicing it.