JavaScript coding interview questions

Question 1: Can you write a function in JavaScript to reverse the order of words in a given string?

```
const reversedString = str => str.split(' ').reverse().join(' ');
```

In this function, the input string is split into an array of words using the `split(' ')` method. Then, the order of the words in the array is reversed with `reverse()`. Finally, the reversed array is joined back into a string using `join(' ')`. This concise line elegantly achieves the reversal of word order in the given string.

Question 2: Can you write a function in JavaScript to remove duplicate elements from an array?

```
const uniqueArray = arr => [...new Set(arr)];
```

This concise function utilizes the `Set` object in JavaScript, which automatically removes duplicate values. The spread operator (`...`) is then used to convert the set back into an array. This elegant solution capitalizes on the unique property of sets, simplifying the process of eliminating duplicates from an array.

Question 3: Can you write a function in JavaScript to merge two objects without overwriting existing properties?

```
const mergeObjects = (obj1, obj2) => ({ ...obj1, ...obj2 });
```

This function utilizes the spread (`...`) operator within an object literal to merge the properties of `obj1` and `obj2`. The order is important; properties of `obj2` will overwrite properties of `obj1` with the same name. This one-liner is effective for creating a new object that contains the combined properties of both objects without modifying the original objects.

Question 4: Can you write a function in JavaScript to get the current date in the format “YYYY-MM-DD”?

```
const currentDate = () => new Date().toISOString().split('T')[0];
```

In this function, the `Date` object is used to get the current date and time. The `toISOString` method is then employed to convert the date to a string in the ISO 8601 format. Finally, the string is split at the 'T' character, and only the part before 'T' (which represents the date) is extracted. This provides the current date in the desired "YYYY-MM-DD" format.

Question 5: Can you write a function in JavaScript to calculate the cumulative sum of an array?

Cumulative sums, or running totals, are used to display the total sum of data as it grows with time (or any other series or progression). This lets you view the total contribution so far of a

given measure against time. An example of a cumulative sum is: Input array => 10, 15, 20, 25, 30. Output array => 10, 25, 45, 70, 100.

```
const cumulativeSum = arr => arr.reduce((acc, num) => [...acc, acc.length
? acc[acc.length - 1] + num : num], []);
```

In this function, the `reduce` method is employed to iterate over the array, maintaining an accumulator (`acc`) that stores the cumulative sum at each step. The spread operator (`...`) is used to create a new array at each step, ensuring that the original array remains unchanged. This approach provides a concise way to calculate the cumulative sum of the input array.

Question 6: Can you write a function in JavaScript to split an array into chunks of a specified size?

```
const chunkArray = (arr, size) => Array.from({ length:
Math.ceil(arr.length / size) }, (_, i) => arr.slice(i * size, i * size +
size));
```

This one-liner function uses `Array.from` to create a new array based on the length of the original array divided by the specified chunk size. The arrow function inside `Array.from` then uses `slice` to extract chunks of the array, ensuring that the chunks do not exceed the array's boundaries.

Question 7: Can you write a one-liner in JavaScript to find the longest consecutive sequence of a specific element in an array?

To enhance comprehension of this idea, let's delve into an illustration. Consider a sequence comprising 10 elements, such as (1, 2, 3, 7, 8, 9, 4, 5, 6). In this scenario, the longest consecutive subsequence would be 3456, encompassing a total of four elements.

```
const longestConsecutiveSequence = (arr, element) =>
Math.max(...arr.join('').split(element).map(group => group.length));
```

In this simple function, the array is first joined into a single string, then split using the specified element. The resulting array consists of groups where the element occurs consecutively. The `map` function is used to transform this array into an array of lengths of consecutive sequences, and `Math.max(...)` finds the length of the longest consecutive sequence.

Question 8: Can you write a function in JavaScript to transpose a 2D matrix?

Creating a program to determine the transpose of a matrix involves the manipulation of its rows and columns. Essentially, the transpose is derived by converting the elements at position A[i][j] to A[j][i] for a matrix of size N by M, where N represents the number of rows and M represents the number of columns. This transformation entails swapping the roles of rows and columns, resulting in a rearranged matrix.

```
const transposeMatrix = matrix => matrix[0].map((col, i) => matrix.map(row
=> row[i]));
```

This function uses the `map` function twice to iterate over the rows and columns of the matrix. The outer `map` iterates over the columns, and the inner `map` iterates over the rows. By swapping the indices during the inner `map`, the original matrix is effectively transposed.

Question 9: Can you write a function in JavaScript to convert a string containing hyphens and underscores to camel case?

Camel case refers to the stylistic convention in which a compound word or phrase is devoid of spaces or punctuation. Rather than employing spaces or punctuation marks, each distinct word within the compound is demarcated by the use of either lowercase or uppercase letters. For instance, the transformation of the string “secret_key_one” into camel case results in “secretKeyOne.”

```
const toCamelCase = str => str.replace(/[-_](.)/g, (_, c) =>
c.toUpperCase());
```

This function uses the `replace` method with a regular expression to match hyphens or underscores followed by any character. The callback function within `replace` converts the matched character to uppercase, effectively transforming the string to camel case.

Question 10: Can you write a line of code in JavaScript to swap the values of two variables without using a temporary variable?

```
[a, b] = [b, a];
```

This simple line of code uses destructuring assignment to swap the values of `a` and `b` without the need for a temporary variable. The right-hand side `[b, a]` creates a new array with the values of `b` and `a`, and the destructuring assignment `[a, b]` assigns these values back to `a` and `b` respectively.

Question 11: Can you write a function in JavaScript to create a countdown from a given number?

```
const countdown = n => Array.from({ length: n }, (_, i) => n - i);
```

This easy function uses `Array.from` to create an array of numbers counting down from the given number `n`. The length of the array is determined by `n`, and the arrow function inside `Array.from` calculates the values by subtracting the index `i` from `n`.

Example:

```
const countdownResult = countdown(5);
console.log(countdownResult); // Output: [5, 4, 3, 2, 1]
```

Question 12: Can you write a function in JavaScript to convert a string to an integer while handling non-numeric characters gracefully?

This function utilizes the unary plus (`+`) operator along with the `isNaN` function to convert a string to an integer.

The `isNaN` function checks if the result of the conversion is not a number, and in such cases, it returns a default value (`0` in this case).

```
const stringToInteger = str => +str === +str ? +str : 0;
```

In the first example, the `stringToInteger` function successfully converts the string "123" to the integer `123`. In the second example, it returns `0` as the string "abc" cannot be converted to a number.

```
stringToInteger("123"); // Output: 123
stringToInteger("abc"); // Output: 0
```

Question 13: Can you write a function in JavaScript to convert a decimal number to its binary representation?

This function leverages the built-in `toString` method with a base argument of `2` to convert a decimal number to its binary representation. It's a concise and built-in way to achieve this conversion in JavaScript.

```
const decimalToBinary = num => num.toString(2);
```

In this example, calling `decimalToBinary(10)` converts the decimal number `10` to its binary representation, which is the string `"1010"`.

```
const decimalNumber = 10;
const binaryRepresentation = decimalToBinary(decimalNumber);
// binaryRepresentation: "1010"
```

Question 14: Can you write a function in JavaScript to calculate the factorial of a given non-negative integer?

This function uses a ternary operator to check if the input `n` is 0. If true, it returns 1, as the factorial of 0 is 1. If false, it uses `Array.from` to create an array of numbers from 1 to `n` and then uses the `reduce` method to calculate the factorial.

```
const factorial = (n) => n === 0 ? 1 : Array.from({length: n}, (_, i) => i
+ 1).reduce((acc, num) => acc * num, 1);
```

An example:

```
const number = 5;
const result = factorial(number);
// result: 120
```

Question 15: Write a concise function to safely access a deeply nested property of an object without throwing an error if any intermediate property is undefined.

With the new version of JavaScript, we can use the optional chaining operator. However, for this interview question, we'll write a function ourselves.

```
const deepAccess = (obj, path) => path.split('.').reduce((acc, key) => acc
&& acc[key], obj);
```

This one-liner function utilizes the `reduce` method on the result of splitting the `path` string by dots. It iteratively accesses nested

properties, ensuring that each intermediate property exists before attempting to access the next one.

Example:

```
const nestedObject = { a: { b: { c: 42 } } };
const propertyPath = 'a.b.c';
const result = deepAccess(nestedObject, propertyPath);
// result: 42
```

Question 16: Can you write a function in JavaScript to generate a random integer between a specified minimum and maximum value (inclusive)?

```
const randomInRange = (min, max) => Math.floor(Math.random() * (max - min
+ 1)) + min;
```

This simple function uses `Math.random()` to generate a random decimal between 0 (inclusive) and 1 (exclusive). By multiplying this value by the range (`max - min + 1`), it ensures that the result covers the entire specified range. `Math.floor` is then applied to round down to the nearest integer, and `min` is added to ensure the result falls within the desired range.

Example:

```
const minValue = 5;
const maxValue = 10;
const result = randomInRange(minValue, maxValue);
// result: 7
```

Question 17: Can you write a function in JavaScript to count the occurrences of each element in an array and return the result as an object?

```
const countOccurrences = (arr) => arr.reduce((acc, val) => (acc[val] =
(acc[val] || 0) + 1, acc), {});
```

This one-liner uses the `reduce` method to iterate over the array (`arr`). The accumulator (`acc`) is an object where each property represents an array element, and the value is the count of occurrences. The expression `(acc[val] = (acc[val] || 0) + 1, acc)` increments the count for the current element or initializes it to 1 if it's the first occurrence.

Example:

```
const inputArray = [1, 2, 2, 3, 4, 4, 4, 5];
const result = countOccurrences(inputArray);
// result: { 1: 1, 2: 2, 3: 1, 4: 3, 5: 1 }
```

Question 18: Can you write a function in JavaScript to capitalize the first letter of each word in a given sentence?

```
const capitalizeWords = (sentence) => sentence.replace(/\b\w/g, char =>
char.toUpperCase());
```

This one-liner uses a regular expression (`/\b\w/g`) to match the first character of each word in the sentence. The `replace` method is then used to replace each matched character with its uppercase equivalent, effectively capitalizing the first letter of each word.

Example:

```
const inputSentence = 'hello world, this is a test';
const result = capitalizeWords(inputSentence);
// result: 'Hello World, This Is A Test'
```

Question 19: Can you write a function in JavaScript to reverse a given string?

JavaScript doesn't have a built-in string reverse function. We need to write one utilizing array reverse function.

```
const reverseString = (str) => str.split('').reverse().join('');
```

This one-liner uses the `split('')` method to convert the input string into an array of characters, `reverse()` reverses the array, and `join('')` converts it back to a string. The result is a reversed version of the input string.

Example:

```
const inputString = 'hello';
const result = reverseString(inputString);
// result: 'olleh'
```

Question 20: Can you write a function in JavaScript to find the longest word in a given sentence?

```
const longestWord = (sentence) => sentence.split(' ').reduce((longest,
word) => word.length > longest.length ? word : longest, '');
```

This function uses the `split(' ')` method to break the input sentence into an array of words. The `reduce` method then finds the word with the maximum length by comparing the length of each word.

Example:

```
const inputSentence = 'The quick brown fox jumpss over the lazy dog';
const result = longestWord(inputSentence);
// result: 'jumpss'
```

Question 21: Can you write a function in JavaScript to rename a specific property in an object?

```
const renameProperty = (obj, oldName, newName) => ({ ...obj, [newName]:
obj[oldName], ...(delete obj[oldName], obj) });
```

This simple function uses the spread operator to create a shallow copy of the object, renaming the specified property in the process. The property with the new name is assigned the value of the property with the old name. The `delete` operator is then used to remove the property with the old name.

Example:

```
const person = { firstName: 'John', lastName: 'Doe', age: 30 };
const updatedPerson = renameProperty(person, 'firstName', 'first');
// updatedPerson: { first: 'John', lastName: 'Doe', age: 30 }
```

Question 22: Can you write a function in JavaScript to find the second-largest element in an array?

```
const secondLargest = (arr) => [...new Set(arr)].sort((a, b) => b - a)[1];
```

This concise function first uses `new Set` to remove duplicates from the array. Then it sorts the unique elements in descending order and returns the second element, which is the second-largest.

Example:

```
const array = [5, 2, 8, 9, 2, 4, 7];
const result = secondLargest(array);
// result: 7
```

Question 23: Can you write a JavaScript function to group an array of objects by a specified property?

```
const groupByProperty = (arr, property) => arr.reduce((grouped, obj) => ({
...grouped, [obj[property]]: [...(grouped[obj[property]] || []), obj] }),
{});
```

This complex function uses the `reduce` method to iterate over the array of objects and group them based on the specified property. The spread operator (`...`) is used to create a new object with the grouping.

Example:

```
const people = [
  { id: 1, name: 'Alice', age: 25 },
```

```
  { id: 2, name: 'Bob', age: 30 },
  { id: 3, name: 'Alice', age: 28 },
];

const result = groupByProperty(people, 'name');
// result: { 'Alice': [ { id: 1, name: 'Alice', age: 25 }, { id: 3, name:
'Alice', age: 28 } ],
//           'Bob': [ { id: 2, name: 'Bob', age: 30 } ] }
```

Question 24: Can you write a JavaScript function to find the missing number in an array of consecutive integers from 1 to N?

```
const findMissingNumber = (arr) => (arr.length + 1) * (arr.length + 2) / 2
- arr.reduce((sum, num) => sum + num, 0);
```

This moderately complicated function uses the formula for the sum of consecutive integers from 1 to N and subtracts the sum of the array to find the missing number.

Example:

```
const arr = [1, 2, 3, 5, 6, 7, 8];
const result = findMissingNumber(arr);
// result: 4
```

Question 25: Can you write a JavaScript function to reverse the key-value pairs of an object?

```
const reverseObject = (obj) =>
Object.fromEntries(Object.entries(obj).map(([key, value]) => [value,
key]));
```

This function uses `Object.entries` to get an array of key-value pairs, then `map` is used to swap the positions of keys and values, and finally, `Object.fromEntries` is used to create a new object.

Example:

```
const originalObject = { a: 1, b: 2, c: 3 };
const reversedObject = reverseObject(originalObject);
// reversedObject: { '1': 'a', '2': 'b', '3': 'c' }
```

Question 26: Can you write a JavaScript function to check if a given string has balanced parentheses?

```
const isBalancedParentheses = (str) => str.split('').reduce((count, char)
=> count >= 0 ? count + (char === '(' ? 1 : char === ')' ? -1 : 0) : -1,
0) === 0;
```

This moderately complex function once again uses `reduce` to iterate over each character in the string, updating a count variable based on the presence of open and close parentheses. It checks if the count stays non-negative and eventually becomes zero, indicating balanced parentheses.

Example:

```
const str = '(a + b) * (c - d)';
const result = isBalancedParentheses(str);
// result: true
```

Question 27: Can you write a concise function in JavaScript to implement a simple debounce function that delays the

execution of a given function until after a specified time interval has passed without additional calls?

In JavaScript, debounce is a powerful technique used to optimize event handling by delaying the execution of a function until after a specified period of inactivity. It helps prevent excessive function calls triggered by rapid events, such as keystrokes or scroll movements.

```
const debounce = (func, delay) => {
  let timeout;
  return (...args) => {
    clearTimeout(timeout);
    timeout = setTimeout(() => func(...args), delay);
  }
};
```

The code above defines a `debounce` function that takes a function (`func`) and a delay time. It returns a new function that, when called, clears the previous timeout (if any) and sets a new timeout for the delayed execution of the provided function.

Example:

```
const delayedLog = debounce((text) => console.log(text), 1000);
delayedLog('Hello'); // Logs 'Hello' after 1000 milliseconds
delayedLog('World'); // Cancels the previous timeout and sets a new one
for 'World'
```

Question 28: Can you write a JavaScript function to truncate a given string to a specified length and append “…” if it exceeds that length?

```
const truncateString = (str, maxLength) => str.length > maxLength ?
str.slice(0, maxLength) + '...' : str;
```

This very simple function uses the ternary operator to check if the length of the string exceeds the specified `maxLength`. If true, it truncates the string and appends "..." using `slice`; otherwise, it returns the original string.

Example:

```
const s = 'This is a very long string that needs to be truncated.';
const maxLen = 20;
const result = truncateString(s, maxLen);
// result: 'This is a very lon...'
```

Question 29: Can you write a throttle function in JavaScript to implement a simple throttle function that limits the execution of a given function to once every specified time interval?

```
const throttle = (func, delay) => {
  let throttled = false;
  return (...args) => {
    if (!throttled) {
      func(...args);
      throttled = true;
      setTimeout(() => throttled = false, delay);
    }
  }
};
```

The above code defines a `throttle` function that takes a function (`func`) and a delay time. It returns a new function that, when called, checks if it's throttled. If not throttled, it executes the provided

function and sets a timeout to enable the next execution after the delay.

Example:

```
const throttledLog = throttle((text) => console.log(text), 1000);
throttledLog('Hello'); // Logs 'Hello'
throttledLog('World'); // Does not log 'World' because it's within the
1000ms throttle interval
```

Question 30: Can you write a JavaScript function to check if a given string has all unique characters?

```
const hasUniqueCharacters = (str) => new Set(str).size === str.length;
```

This simple one-liner function uses a `Set` to create a collection of unique characters from the string. It then compares the size of the set with the length of the original string to determine if all characters are unique.

Example:

```
const s1 = 'abcdef';
const result1 = hasUniqueCharacters(s1); // result1: true

const s2 = 'hello';
const result2 = hasUniqueCharacters(s2); // result2: false
```

Question 31: Can you write a function in JavaScript to convert each string in an array of strings to uppercase?

```
const convertToUppercase = (arr) => arr.map(str => str.toUpperCase());
```

The tiny function `convertToUppercase` uses the `map` method to iterate over each string in the array and applies the `toUpperCase` method to convert it to uppercase.

Example:

```
const arr = ['apple', 'banana', 'cherry'];
const result = convertToUppercase(arr);
// result: ['APPLE', 'BANANA', 'CHERRY']
```

Question 32: Can you write a JavaScript function to find the first non-repeated character in a given string?

```
const firstNonRepeatedChar = (str) => str.split('').find(char =>
str.indexOf(char) === str.lastIndexOf(char));
```

This simple function uses the `split` method to convert the string into an array of characters and the `find` method to locate the first character that has the same index when searched forward and backward in the string.

Example:

```
const s1 = 'programming';
const r1 = firstNonRepeatedChar(s1); // r1: 'o'
```

```
const s2 = 'hello';
const r2 = firstNonRepeatedChar(s2); // r2: 'h'
```

Question 33: Can you write a JavaScript function to find the longest word in a sentence?

```
const longestWord = (sentence) => sentence.split(' ').reduce((longest,
word) => word.length > longest.length ? word : longest, '');
```

The function `longestWord` uses the `split` method to break the sentence into an array of words and the `reduce` method to find the word with the maximum length.

Example:

```
const s = 'The quick brown fox jumps over the lazy dog';
const r = longestWord(s);
console.log(r); // r: 'quick'
```

Question 34: Can you write a JavaScript function to flatten a nested object?

```
const flattenObject = (obj) => Object.assign({}, ...(function
flattenObj(o) { return [].concat(...Object.keys(o).map(k => typeof o[k]
=== 'object' ? flattenObj({ [k]: o[k] }) : { [k]: o[k] }))})(obj));
```

This is a tricky function that needs a bit of understanding. This function uses a recursive function `flattenObj` to flatten the nested object. The `Object.assign` method is then used to merge the flattened objects into a single flat object.

Example:

```
const o = { a: 1, b: { c: 2, d: { e: 3 } } };
const r = flattenObject(o);
// r: { a: 1, 'b.c': 2, 'b.d.e': 3 }
```

Question 35: Can you write a JavaScript function to rotate the elements of an array to the right by a specified number of positions?

```
const rotateArray = (arr, positions) => arr.slice(-
positions).concat(arr.slice(0, -positions));
```

This simple function uses the `slice` method to extract the last `positions` elements and the remaining elements, and then concatenates them in reverse order to achieve the rotation.

Example:

```
const arr = [1, 2, 3, 4, 5];
const pos = 2;
const result = rotateArray(arr, pos);
// result: [4, 5, 1, 2, 3]
```

Question 36: Can you write a JavaScript function to convert a given number of minutes into hours and minutes?

```
const convertToHoursAndMinutes = (minutes) => `${Math.floor(minutes /
60)}h ${minutes % 60}m`;
```

This simple function uses the division operator and the modulo operator to calculate the number of hours and the remaining minutes.

Example:

```
const t = 125;
const r = convertToHoursAndMinutes(t);
// r: '2h 5m'
```

Question 37: Can you write a JavaScript function to generate a random password of a specified length?

```
const generateRandomPassword = (length) => Array.from({ length }, () =>
String.fromCharCode(Math.floor(Math.random() * 94) + 33)).join('');
```

This simple one-liner functino uses `Array.from` to create an array of the specified length and fills it with random characters by generating random Unicode values within the printable ASCII range.

Example:

```
const len = 12;
const pass = generateRandomPassword(len);
// pass: '$2#XrGp^@L9'
```

Question 38: Can you write a JavaScript function to convert an RGB color to its hexadecimal representation?

```
const rgbToHex = (r, g, b) => `#${(1 << 24 | r << 16 | g << 8 |
b).toString(16).slice(1)}`;
```

This function uses bitwise operations and string formatting to convert the RGB values to a hexadecimal representation with the `#` prefix.

Example:

```
const r = 255;
const g = 127;
const b = 63;
const color = rgbToHex(r, g, b);
// color: '#ff7f3f'
```

Question 39: Can you write a JavaScript function to check if a given string has balanced brackets?

This function is a variation of the parentheses' problem.

```
const areBracketsBalanced = (str) => !str.split('').reduce((count, char)
=> (char === '(' || char === '[' || char === '{') ? ++count : (char ===
')' || char === ']' || char === '}') ? --count : count, 0);
```

This function uses the `reduce` method to iterate through the string, incrementing the count for opening brackets and decrementing for closing brackets. The result is `true` if the count is zero, indicating balanced brackets.

Example:

```
const s1 = '({[]})';
const r1 = areBracketsBalanced(s1); // r1: true

const s2 = '({[})';
const r2 = areBracketsBalanced(s2); // r2: false
```

Question 40: Can you write a JavaScript function to generate a unique identifier?

```
const generateUniqueId = () => Math.random().toString(36).substr(2, 9);
```

This function uses `Math.random()` to generate a random decimal between 0 and 1, converts it to a base-36 string, and then takes a substring to get a unique identifier.

Example:

```
const uniqueId = generateUniqueId();
// uniqueId: 'abc123xyz'
```

Question 41: Can you write a JavaScript function that returns a promise which resolves after a specified delay?

```
const delayPromise = (ms) => new Promise(resolve => setTimeout(resolve,
ms));
```

This simple function creates a Promise that resolves after the specified delay using the `setTimeout` function.

Example:

```
const delay = 2000;
delayPromise(delay).then(() => console.log('Promise resolved after
delay'));
```

Question 42: Can you write a JavaScript function to convert an object to a query string?

```
const objectToQueryString = (obj) => Object.entries(obj).map(([key,
value]) =>
`${encodeURIComponent(key)}=${encodeURIComponent(value)}`).join('&');
```

This function uses `Object.entries` to convert the object into an array of key-value pairs, then uses `map` to encode each pair, and finally, `join` to concatenate them into a query string.

Example:

```
const o = { name: 'John Doe', age: 30, city: 'Example City' };
const qs = objectToQueryString(o);
// qs: 'name=John%20Doe&age=30&city=Example%20City'
```

Question 43: Can you write a JavaScript function to check if two objects have the same properties (regardless of order)?

```
const haveSameProperties = (obj1, obj2) =>
JSON.stringify(Object.keys(obj1).sort()) ===
JSON.stringify(Object.keys(obj2).sort());
```

This very simple function converts the sorted array of keys for each object to JSON strings and compares them to check if both objects have the same properties.

Example:

```
const o1 = { name: 'John', age: 30, city: 'Example City' };
const o2 = { age: 30, name: 'John', city: 'Example City' };

const r = haveSameProperties(o1, o2);
// r: true
```

Question 44: Can you write a JavaScript function to count the occurrences of each word in a given sentence?

```
const countWordOccurrences = (sentence) => sentence.split('
').reduce((countMap, word) => ({ ...countMap, [word]: (countMap[word] ||
0) + 1 }), {});
```

This one-liner uses the `split` method to break the sentence into an array of words and the `reduce` method to count the occurrences of each word in an object.

Example:

```
const s = 'the quick brown fox jumps over the lazy dog jumps over the
fence';
const r = countWordOccurrences(s);
console.log(r);
/*
r:
{
  'the': 3,
  'quick': 1,
  'brown': 1,
  'fox': 1,
  'jumps': 2,
```

```
  'over': 2,
  'lazy': 1,
  'dog': 1,
  'fence': 1
}
*/
```

Question 45: Can you write a JavaScript function to generate a random color in hexadecimal format?

```
const randomColor = () =>
`#${Math.floor(Math.random()*16777215).toString(16)}`;
```

This simple function uses `Math.random()` to generate a random number, `Math.floor` to round it down, and `toString(16)` to convert it to a hexadecimal string. The resulting string is in the format `#RRGGBB`.

Example:

```
const color = randomColor();
// color: '#1a2b3c'
```

Question 46: Can you write a JavaScript function to implement a simple caching function that stores the result of a function for a given input and returns the cached result if the same input is provided again?

```
const createCache = (func) => {
  const cache = new Map();
  return (...args) => cache.has(JSON.stringify(args)) ?
cache.get(JSON.stringify(args)) : (cache.set(JSON.stringify(args),
func(...args)), cache.get(JSON.stringify(args)));
};
```

This function creates a closure with a cache (using `Map`) and returns a function that checks if the cache has the result for the given arguments. If not, it calculates the result, stores it in the cache, and returns the result.

Example:

```
const add = (a, b) => a + b;
const cachedAdd = createCache(add);

cachedAdd(2, 3); // Output: 5 (calculated and cached)
cachedAdd(2, 3); // Output: 5 (returned from cache)
```

Question 47: Can you write a JavaScript function to generate an array of specified length filled with random numbers?

```
const generateRandomArray = (length) => Array.from({ length }, () =>
Math.floor(Math.random() * 100));
```

This simple function uses `Array.from` to create an array of the specified length and fills it with random numbers between 0 and 99.

Example:

```
const arr = generateRandomArray(5);
// arr: [42, 18, 75, 3, 91]
```

Question 48: Can you write a simple event emitter in JavaScript?

```
const createEventEmitter = () => {
  const listeners = new Map();
```

```
  return {
    on: (event, listener) => listeners.has(event) ?
listeners.get(event).push(listener) : listeners.set(event, [listener]),
    emit: (event, ...args) => listeners.get(event)?.forEach(listener =>
listener(...args)),
    off: (event, listener) => listeners.set(event,
listeners.get(event)?.filter(l => l !== listener)),
  };
};
```

This complicated function creates a simple event emitter with `on` to add listeners, `emit` to trigger events, and `off` to remove listeners.

Example:

```
const eventEmitter = createEventEmitter();

const greetListener = (name) => console.log(`Hello, ${name}!`);
eventEmitter.on('greet', greetListener);

eventEmitter.emit('greet', 'Alice'); // Output: 'Hello, Alice!'
eventEmitter.off('greet', greetListener);

eventEmitter.emit('greet', 'Bob'); // No output (listener removed)
```

Question 49: Can you write a JavaScript function to implement a basic queue using arrays with `enqueue` and `dequeue` operations?

```
const createQueue = () => ({
  items: [],
  enqueue: (item) => (items.push(item)),
  dequeue: () => items.shift()
});
```

This simple function creates a simple queue object with an array (`items`) and methods (`enqueue` and `dequeue`) to add and remove items from the queue.

Example:

```
const queue = createQueue();
queue.enqueue('item1');
queue.enqueue('item2');
queue.dequeue(); // Output: 'item1'
queue.dequeue(); // Output: 'item2'
```

Question 50: Can you write a JavaScript function to implement a basic stack using arrays with `push` and `pop` operations?

```
const createStack = () => ({
  items: [],
  push: (item) => items.push(item),
  pop: () => items.pop()
});
```

This function creates a simple stack object with an array (`items`) and methods (`push` and `pop`) to add and remove items from the stack.

Example:

```
const stack = createStack();
stack.push('item1');
stack.push('item2');
stack.pop(); // Output: 'item2'
stack.pop(); // Output: 'item1'
```