# JS Interview Questions

## 1. What is asynchronous programming, and why is it important in JavaScript? (JavaScript event loop)

Synchronous programming means that, barring conditionals and function calls, code is executed sequentially from top-to-bottom, blocking on long-running tasks such as network requests and disk I/O.
Asynchronous programming means that the engine runs in an event loop. When a blocking operation is needed, the request is started, and the code keeps running without blocking for the result. When the response is ready, an interrupt is fired, which causes an event handler to be run, where the control flow continues. In this way, a single program thread can handle many concurrent operations. That is why synchronous javascript has asynchronous effects.
User interfaces are asynchronous by nature, and spend most of their time waiting for user input to interrupt the event loop and trigger event handlers.
Node is asynchronous by default, meaning that the server works in much the same way, waiting in a loop for a network request, and accepting more incoming requests while the first one is being handled.
This is important in JavaScript, because it is a very natural fit for user interface code, and very beneficial to performance on the server.

**2.** Please give the output of the following code (help understand scope/closure):

```
const arr = [10, 12, 15, 21];
for (var i = 0; i < arr.length; i++) {
    setTimeout(function() {
        console.log('Index: ' + i + ', element: ' + arr[i]);
    }, 3000);
}
```

Output:

```
Index: 4, element: undefined
Index: 4, element: undefined
Index: 4, element: undefined
Index: 4, element: undefined
```

Two possible solutions:

```
// solution 1
const arr = [10, 12, 15, 21];
for (var i = 0; i < arr.length; i++) {
    // pass in the variable i so that each function has access to the correct index
    setTimeout(function(i_local) {
        return function() {
            console.log('The index of this number is: ' + i_local);
        }
    }(i), 3000);
}
```

```
// solution 2
for (let i = 0; i < arr.length; i++) {
  // using the ES6 let syntax, it creates a new binding, every single time the function is called
  // read more here: http://exploringjs.com/es6/ch_variables.html#sec_let-const-loop-heads
  setTimeout(function() {
    console.log('The index of this number is: ' + i);
  }, 3000);
}
```

## 3. Describe callback and callback doom, how to handle it?

A callback is a plain JavaScript function passed to some method as an argument or option. The cause of **callback hell** is when people try to write JavaScript in a way where execution happens visually from top to bottom

- keep code shallow so easy to trace
- modularize: separate functions into small parts and assemble them into bigger module
- promises: A promise is an object that may produce a single value some time in the future: either a resolved value, or a reason that it's not resolved (e.g., a network error occurred). A promise may be in one of 3 possible states: fulfilled, rejected, or pending. Promise users can attach callbacks to handle the fulfilled value or the reason for rejection.
- Using async/await functions (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function)

(another way of describing this question: how to handle asynchronous functions in JS? Callback, promises and async/await)

## 1. What the difference between arrow functions and normal functions?

1. **Lexical this and arguments**: Arrow functions don't have their own this or arguments binding.
2. **Arrow functions cannot be called with new**: It is more like a "static" function.

Check reference here.

## 2. Difference between function and class.

Actually in JS, class is "special" function. With function, you can do almost all procedure programing tasks smoothly, but in the context of object oriented programming, it can be a bit tricky by apply prototype to defining sub-classes
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/prototype.

## 3. Difference among let, const and var in terms of scope.

| | var | let | const |
|---|---|---|---|
| scope | Function scoped | Block scoped | Block scoped |
| accessibility | Undefined when accessing a variable before it's declared | ReferenceError when accessing a variable | ReferenceError when accessing a variable |

| | | | |
|---|---|---|---|
| | | before it’s declared | before it’s declared |
| assign | Can be re-assigned | Can be re-assigned | Cannot be re-assigned |

4. **How do you understand Scope and Closure.**
   A *closure* is the combination of a function and the lexical environment within which that function was declared. a.k.a. Nested functions have access to variables declared in their outer scope.
   See https://developer.mozilla.org/en-US/docs/Web/JavaScript/Closures and https://wsvincent.com/javascript-scope-closures/

5. **JavaScript Data Structure like Set, Map etc.**
   Check it here: JavaScript Set, JavaScript Map.