Callback interview questions

give list and solution of 10 hard questions of callback which are most frequently asked in interviews

Absolutely, Vipul! Let's tackle **10 advanced JavaScript callback-related interview questions** with detailed explanations and solutions. I'll make them beginner-friendly but challenging enough for interviews.

---

1. Explain the output of this code (closure + callback)

```javascript
for (var i = 0; i < 3; i++) {
  setTimeout(function() {
    console.log(i);
  }, 1000);
}
```

Solution

- Output:

```
3
3
3
```

- Reason: `var` has **function scope**, so the same `i` is used in all callbacks. After the loop, `i = 3`.

Fix using closure:

```javascript
for (var i = 0; i < 3; i++) {
  (function(i){
    setTimeout(function() {
      console.log(i);
    }, 1000);
  })(i);
}
```

Output:

```
0
1
2
```

Or using `let` (block scope)

```javascript
for (let i = 0; i < 3; i++) {
  setTimeout(() => console.log(i), 1000);
}
```

---

2. Convert a function to use callbacks instead of return

```javascript
function add(a, b) {
  return a + b;
}
```

Solution

```javascript
function addCallback(a, b, callback) {
  callback(a + b);
}

addCallback(5, 10, function(result){
  console.log(result); // 15
});
```

---

3. Callback hell example

Problem: Nested asynchronous tasks

```javascript
doTask1(function(result1){
  doTask2(result1, function(result2){
    doTask3(result2, function(result3){
      console.log(result3);
    });
  });
});
```

Solution: Flatten using **named functions**

```javascript
function task1Callback(result1) {
  doTask2(result1, task2Callback);
}
function task2Callback(result2) {
  doTask3(result2, task3Callback);
}
function task3Callback(result3) {
  console.log(result3);
}
doTask1(task1Callback);
```

Or use **Promises** to avoid callback hell.

---

4. Implement `map` using callbacks

```javascript
function customMap(arr, callback) {
  let result = [];
  for(let i = 0; i < arr.length; i++) {
    result.push(callback(arr[i], i, arr));
  }
  return result;
}

console.log(customMap([1,2,3], x => x * 2)); // [2,4,6]
```

---

5. Asynchronous execution order

```javascript
console.log('Start');

setTimeout(() => console.log('Timeout 1'), 0);
Promise.resolve().then(() => console.log('Promise'));

console.log('End');
```

Solution

Output:

```sql
Start
End
Promise
Timeout 1
```

Explanation:

- Microtasks (Promises) run **before** macrotasks (setTimeout) even if timeout = 0.

---

6. Create a function that executes callbacks sequentially

```javascript
function runSequential(tasks) {
  let result = Promise.resolve();
  tasks.forEach(task => {
    result = result.then(task);
  });
  return result;
}

// Usage
```

```
runSequential([
  () => Promise.resolve(console.log('Task 1')),
  () => Promise.resolve(console.log('Task 2')),
  () => Promise.resolve(console.log('Task 3'))
]);
```

---

7. Callback with error handling

```javascript
function divide(a, b, callback) {
  if (b === 0) {
    callback('Division by zero error', null);
  } else {
    callback(null, a / b);
  }
}

divide(10, 0, (err, result) => {
  if (err) console.log(err);
  else console.log(result);
});
```

- **Output:** `Division by zero error`

---

8. Implement `filter` using callbacks

```javascript
function customFilter(arr, callback) {
  let result = [];
  for(let i = 0; i < arr.length; i++) {
    if(callback(arr[i], i, arr)) {
      result.push(arr[i]);
    }
  }
  return result;
}

console.log(customFilter([1,2,3,4], x => x % 2 === 0)); // [2,4]
```

---

9. Debounce using callback

```javascript
function debounce(func, delay) {
  let timeout;
  return function(...args) {
    clearTimeout(timeout);
    timeout = setTimeout(() => func.apply(this, args), delay);
  };
}

const handleInput = debounce((value) => console.log(value), 500);
handleInput('H');
handleInput('He');
handleInput('Hel'); // Only 'Hel' will print after 500ms
```

- **Use:** Avoid calling callback too frequently.

---

10. Implement `once` function using callback

```javascript
function once(fn) {
  let called = false;
  return function(...args) {
    if (!called) {
      called = true;
      return fn.apply(this, args);
    }
  };
}

const init = once(() => console.log('Initialized'));
init(); // Initialized
init(); // Nothing
```

✅ Key Takeaways:

- Always be clear about **synchronous vs asynchronous** callbacks.
- Avoid **callback hell** by using named functions or Promises.
- Microtasks (Promises) run **before** macrotasks (setTimeout).
- Common interview patterns: **`once`**, **`debounce`**, **`sequential execution`**, **`error handling`**.

If you want, I can make a **second set of 10 extremely tricky callback questions** that are **frequently asked in FAANG+ interviews**, with real code problems like **race conditions, nesting, and async-heavy challenges**.

Do you want me to do that?

yes

Perfect! Here’s a **second set of 10 advanced callback/async questions** that are considered tricky and are frequently asked in high-level interviews like FAANG. I’ll include detailed solutions and explanations.

---

1. Multiple asynchronous callbacks with varying delays

```javascript
function asyncTask(name, delay, callback) {
  setTimeout(() => {
    console.log(name);
    callback();
  }, delay);
}
```

```
asyncTask('A', 1000, () => {
  asyncTask('B', 500, () => {
    asyncTask('C', 100, () => {
      console.log('Done');
    });
  });
});
```

Explanation:

- Output:

```css
A
B
C
Done
```

- Even though B’s delay is smaller than A, it waits for A because of the **nested callbacks**.

Takeaway: Nested callbacks preserve order, not delay duration.

---

2. Callback with shared variable trap

```javascript
let arr = [1,2,3];
for (var i = 0; i < arr.length; i++) {
  setTimeout(() => console.log(arr[i]), 1000);
}
```

Output: `undefined undefined undefined`

- Reason: **`var i`** is function-scoped; after loop ends **`i=3`**, so **`arr[3]`** is **`undefined`**.

Fix:

```javascript
for (let i = 0; i < arr.length; i++) {
  setTimeout(() => console.log(arr[i]), 1000);
}
```

Output: **`1 2 3`**

---

3. Race condition with callbacks

```javascript
let data = null;

function fetchData(callback) {
  setTimeout(() => {
    data = 'Hello';
    callback(data);
  }, Math.random()*1000);
}

fetchData((res) => console.log(res));
console.log(data);
```

Output:

```pgsql
null
Hello  // after some random time
```

- Reason: `console.log(data)` runs **synchronously**, but fetchData callback runs asynchronously.
- **Lesson:** Always handle dependent code inside the callback.

---

4. Callback with array iteration (async trap)

```javascript
let items = [1,2,3];
items.forEach(async (item) => {
  await new Promise(res => setTimeout(res, 100));
  console.log(item);
});
console.log('Done');
```

Output:

```nginx
Done
1
2
3
```

Explanation: `forEach` doesn't wait for async functions.
Fix: Use `for..of`:

```javascript
async function process() {
  for (const item of items) {
    await new Promise(res => setTimeout(res, 100));
    console.log(item);
  }
  console.log('Done');
}
process();
```

---

5. Callback with `.bind()`

```javascript
function printName(prefix) {
  console.log(prefix + this.name);
}
const user = { name: 'Vipul' };

setTimeout(printName.bind(user, 'Hello '), 500);
```

Output: `Hello Vipul`

- **Takeaway:** `bind` fixes `this` for callbacks.

---

6. Callback inside loop with async closure

```javascript
for (var i = 0; i < 3; i++) {
  setTimeout((i) => console.log(i), 1000, i);
}
```

- Output: `0 1 2`
- Explanation: Using **setTimeout 3rd parameter** passes `i` directly to the callback, avoiding closure issues.

---

7. Handling multiple callbacks in parallel

```javascript
function task(name, cb) {
  setTimeout(() => {
    console.log(name);
    cb();
  }, Math.random()*1000);
}

let doneCount = 0;
function allDone() {
  doneCount++;
  if (doneCount === 3) console.log('All tasks finished');
}

task('A', allDone);
task('B', allDone);
task('C', allDone);
```

- Output order of A, B, C is **random**, but `'All tasks finished'` prints **after all three** finish.
- **Takeaway:** Classic **parallel callback completion** pattern.

---

8. Callback with early return

```javascript
function checkNumber(num, callback) {
  if (num < 0) return callback('Negative');
  callback(null, num * 2);
}

checkNumber(-5, (err, result) => {
  if(err) console.log(err);
  else console.log(result);
});
```

- Output: **`Negative`**
- **Takeaway:** Always handle **error-first callbacks** to prevent execution of success logic when error occurs.

---

9. Transform callback to Promise

```javascript
function asyncAdd(a, b, callback) {
  setTimeout(() => callback(a+b), 500);
}

// Transform
function asyncAddPromise(a, b) {
  return new Promise(resolve => asyncAdd(a, b, resolve));
}

asyncAddPromise(5, 10).then(console.log); // 15
```

- **Takeaway:** Many callback-based APIs can be wrapped in Promises.

---

10. Nested callbacks with conditional execution

```javascript
function step1(cb) { setTimeout(() => cb('S1'), 500); }
function step2(input, cb) { setTimeout(() => cb(input+'->S2'), 300); }
function step3(input, cb) { setTimeout(() => cb(input+'->S3'), 100); }

step1(s1 => {
  if(s1 === 'S1') {
    step2(s1, s2 => {
      if(s2.includes('S2')) {
        step3(s2, s3 => console.log('Final:', s3));
      }
    });
  }
});
```

- Output: **`Final: S1->S2->S3`**
- **Takeaway:** Always check conditions inside nested callbacks to avoid **unexpected execution**.

✅ Tips for Callback Questions in Interviews

1. Always check **synchronous vs asynchronous behavior**.
2. Watch for **variable scope issues** (`var` vs `let`).
3. Understand **microtasks vs macrotasks** (Promises vs setTimeout).
4. Learn **patterns**: sequential, parallel, once, debounce, error-first.
5. Practice **wrapping callbacks in Promises** for modern code.

If you want, I can make a **mega cheat-sheet of 20+ callback patterns with diagrams and output predictions**, which is extremely useful for interviews.

Do you want me to make that?

yes

Perfect, Vipul! I'll create a **mega callback cheat-sheet** for you. It will include **20+ patterns**, with **diagrams, output predictions, and explanations**—everything you need to ace interviews. I'll organize it so you can **quickly revise before interviews**.

Here's the complete cheat-sheet:

JavaScript Callback Mega Cheat-Sheet (20+ Patterns)

---

1. Basic Callback

```javascript
function greet(name, callback) {
  callback(`Hello, ${name}`);
}

greet('Vipul', msg => console.log(msg));
```

Output:

```
Hello, Vipul
```

Takeaway: Simple **passing a function as argument**.

---

2. Callback with return value trap

```javascript
function add(a, b, cb) { cb(a+b); }
```

```
console.log(add(5, 3, res => res));
```

Output:

```javascript
undefined
```

Reason: `cb` returns value to callback, not `add`. Always use callback **inside** async/sync code.

---

3. Async setTimeout Callback

```javascript
console.log('Start');
setTimeout(() => console.log('Callback'), 0);
console.log('End');
```

Output:

```sql
Start
End
Callback
```

Takeaway: `setTimeout` is async (macrotask).

---

4. Callback with var loop trap

```javascript
for(var i=0;i<3;i++){
  setTimeout(()=>console.log(i),100);
}
```

Output: `3 3 3`
Fix: use `let i` or closure.

---

5. Callback with let

```javascript
for(let i=0;i<3;i++){
  setTimeout(()=>console.log(i),100);
}
```

Output: `0 1 2`
Reason: `let` is block-scoped.

6. Closure in callback

```
javascript

for(var i=0;i<3;i++){
  (function(i){
    setTimeout(()=>console.log(i),100);
  })(i);
}
```

Output: `0 1 2`
Takeaway: Use IIFE to **capture variable**.

---

7. Error-first callback

```
javascript

function divide(a, b, cb){
  if(b===0) return cb('Error');
  cb(null, a/b);
}
divide(10,0,(err,res)=>{ if(err) console.log(err); else console.log(res); });
```

Output: `Error`

---

8. Nested Callback (Callback Hell)

```
javascript

task1(res1 => {
  task2(res2 => {
    task3(res3 => console.log(res3));
  });
});
```

Takeaway: Avoid using **Promises** for better readability.

---

9. Sequential Execution

```
javascript

function runSequential(tasks){
  let result = Promise.resolve();
  tasks.forEach(task => result = result.then(task));
  return result;
}
```

Use: Ensures callbacks execute **one after another**.

---

10. Parallel Execution with Completion Callback

```
javascript

let done=0;
function allDone(){
  done++;
  if(done===3) console.log('All finished');
}
task('A',allDone); task('B',allDone); task('C',allDone);
```

Takeaway: Track completed tasks for parallel execution.

---

11. Async forEach Trap

```
javascript

[1,2,3].forEach(async item=>{
  await new Promise(r=>setTimeout(r,100));
  console.log(item);
});
console.log('Done');
```

Output:

```
nginx

Done
1
2
3
```

Fix: Use `for..of` loop with `await`.

---

12. setTimeout with parameters

```
javascript

for(var i=0;i<3;i++){
  setTimeout((i)=>console.log(i),100,i);
}
```

Output: `0 1 2`

Takeaway: Pass parameters directly to avoid closure issues.

---

13. Debounce Callback

```
javascript

function debounce(fn, delay){
  let t;
  return function(...args){
    clearTimeout(t);
    t=setTimeout(()=>fn(...args), delay);
  }
}
```

Use: Limit frequent execution (input events).

---

14. Once Callback

```
javascript

function once(fn){
  let called=false;
  return function(...args){
    if(!called){ called=true; return fn(...args); }
  }
}
```

Use: Run callback only **once** (e.g., initialization).

---

15. Callback with `this` binding

```
javascript

function greet(prefix){ console.log(prefix+this.name); }
const user={name:'Vipul'};
setTimeout(greet.bind(user,'Hello '),500);
```

Output: `Hello Vipul`

---

16. Wrapping Callback in Promise

```
javascript

function asyncAdd(a,b,cb){ setTimeout(()=>cb(a+b),500); }
const promiseAdd = (a,b)=>new Promise(res=>asyncAdd(a,b,res));
promiseAdd(5,10).then(console.log);
```

Output: 15

---

17. Conditional Nested Callback

```
javascript

step1(s1=>{
  if(s1==='S1'){
    step2(s1,s2=>{
      if(s2.includes('S2')) step3(s2,s3=>console.log('Final:',s3));
    });
  }
});
```

Output: `Final: S1->S2->S3`

18. Callback with setInterval

```
javascript

let count=0;
const id=setInterval(()=>{
  console.log('Tick');
  if(++count===3) clearInterval(id);
},100);
```

Output:

```
nginx

Tick
Tick
Tick
```

- **Takeaway:** Always clear interval to avoid infinite callback calls.

---

19. Callback with map/filter

```
javascript

function customMap(arr, cb){
  let res=[];
  for(let i of arr) res.push(cb(i));
  return res;
}
console.log(customMap([1,2,3], x=>x*2));
```

Output: `[2,4,6]`

---

20. Callback Order: Microtask vs Macrotask

```
javascript

console.log('Start');
setTimeout(()=>console.log('Timeout'),0);
Promise.resolve().then(()=>console.log('Promise'));
console.log('End');
```

Output:

```
sql

Start
End
Promise
Timeout
```

- Microtasks run before macrotasks.

21. Race Condition Example

```
javascript

let data=null;
setTimeout(()=>{ data='A'; console.log(data); }, 500);
console.log(data);
```

Output:

```
csharp

null
A
```

- Always handle async data in callback.

✅ Tips for Using This Cheat-Sheet

1. Always **check variable scope** (`var` vs `let`).
2. Understand **synchronous vs asynchronous** execution.
3. Learn **microtasks vs macrotasks** for Promises vs setTimeout.
4. Recognize **callback patterns**: sequential, parallel, once, debounce.
5. Practice **wrapping callbacks in Promises** for modern code.

Vipul, if you want, I can also make a **visual diagram cheat-sheet** showing **20+ callback patterns with arrows, execution order, and async flow**, which is insanely useful for interviews.

Do you want me to make that visual version?