More About Promises in JavaScript

Sunday, 3 September 2023 7:15 PM

```
3
4  console.log("start");
5
6  function importantAction(username) {
7    setTimeout(() => {
8      return `Subscribe to ${username}`;
9    }, 1000);
10 }
11
12 const message = importantAction("Roadside Coder");
13
14 console.log(message);
15
16 console.log("stop");
17
```

```
start                    script.js:4
undefined                script.js:14
stop                     script.js:16
>
```

(o/p)

→ So, we get undefined on line 14 as JS waits for no one & whether `message` has value or not it will be printed.

→ Callback fn is inside setTimeout so it is asynchronous & it would be executed after synchronous code ends (through event loop).

→ So, to solve the above we use callbacks.

```
4  console.log("start");
5
6  function importantAction(username, cb) {
7    setTimeout(() => {
8      cb(`Subscribe to ${username}`);
9    }, 1000);
10 }
11
12 const message = importantAction("Roadside Coder", function (message) {
13   console.log(message);
14 });
15
16 console.log("stop");
17
```

```
start                                script.js:4
stop                                 script.js:16
Subscribe to Roadside Coder          script.js:13
>
```

(o/p)

→ Now, we pass callback fn inside importantAction() which calls it inside setTimeout & this callback fn prints the message. So it is basically used to handle async operations.

→ And since async callbacks pass through callback queue & they are passed in call stack by event loop once it becomes empty (once, synchronous code ends), so it is executed at last after all synchronous code ends.

→ Now, let's assume there is another fn that we need to run after the previous one is

```
12 function likeTheVideo(video, cb) {
13   setTimeout(() => {
14     cb(`Like the ${video} video`);
15   }, 1000);
16 }
17
18 const message = importantAction("Roadside Coder", function (message) {
19   console.log(message);
20   likeTheVideo("Javascript Interview Questions", (action) => {
21     console.log(action);
22   });
23 });
```

→ So, now likeTheVideo() works when line 19 has been executed.

→ We can see if there were many dependant fn's then this would create a pyramid of doom or callback hell. So to improve this, we have promises.

→

```
4  console.log("start");
5
6  const sub = new Promise((resolve, reject) => {
7    setTimeout(() => {
8      const result = false;
9      if (result) resolve("Subscribed to Roadside Coder");
10     else reject(new Error("Why aren't you subscribed to Roadside Coder?"));
11   }, 2000);
```

→ This is how we create a new promise which has 2 fn's resolve() & reject()

```
});

sub
  .then((res) => {
    console.log(res);
  })
  .catch((err) => {
    console.error(err);
  });

console.log("stop");
```

→ In order to do something when promise gets resolved, we attach a callback fn to .then() & to catch any error if promise is in rejected state, we attach a callback fn to .catch().

→ We can also use Promise.resolve independently & it will give us the same output and it will be an async operation.

```
console.log("start");

const sub = Promise.resolve("Subscribed to Roadside Coder");
console.log(sub);
sub.then((res) => console.log(res));

console.log("stop");
```

```
start                                          script.js:4
                                               script.js:7
▶ Promise {<fulfilled>: 'Subscribed to R
  oadside Coder'}
stop                                           script.js:10
Subscribed to Roadside Coder                   script.js:8
>
```

→ Fulfilled promise due to Promise.resolve()

(o/p)

→ It works the same but gives rejected state if we use Promise.reject().

→ To resolve callback hell in last code, we can use Promises:

```
function shareTheVideo(video) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve(`Share the ${video} video`);
    }, 1000);
  });
}
```

→ fn's will be returning promises now instead of calling callback fn's.

```
importantAction("Roadside Coder")
  .then((res) => {
    console.log(res);
    likeTheVideo("Javascript Interview Questions").then((res) => {
      console.log(res);
      shareTheVideo("Javascript Interview Questions").then((res) => {
        console.log(res);
      });
    });
  })
  .catch((err) => console.error(err));
```

→ But we can see that this is resulting in something like promise hell.

→ So, now what we can do is, we can use promise chaining.

```
importantAction("Roadside Coder")
  .then((res) => {
    console.log(res);
    return likeTheVideo("Javascript Interview Questions");
  })
  .then((res) => {
    console.log(res);
    return shareTheVideo("Javascript Interview Questions");
  })
  .then((res) => {
    console.log(res);
  })
  .catch((err) => console.error(err));
```

→ So this is promise chaining, here we return a promise which when resolved will make .then() fn call the callback fn inside the chain.

→ Promise Combinators: These help us to execute more than one promise at a time.

1) Promise.all(): If we provide all the promises to Promise.all() then it is going to run all promises in parallel & then going to return array of all fulfilled promises. But if one of the promise fails then it is going to fail the complete Promise.all().

```
Promise.all([
  importantAction("Roadside Coder"),
  likeTheVideo("Javascript Interview Questions"),
  shareTheVideo("Javascript Interview Questions"),
])
```

```
                                               script.js:36
(3) ['Subscribe to Roadside Coder', 'L
ike the Javascript Interview Questions
video', 'Share the Javascript Intervie
w Questions video']
```

```
.then((res) => {
  console.log(res);
})
.catch((err) => {
  console.error(err);
});
```

```
0: "Subscribe to Roadside Coder"
1: "Like the Javascript Interview Que
2: "Share the Javascript Interview Qu
length: 3
▶ [[Prototype]]: Array(0)
```

→ All promises above got resolved, so .then() 's attached callback fn will run otherwise it will just give error & .then() won't work if any one of the promises fail.

(o/p: array of all promises' 'res')

2) Promise.race() : Its syntax is exactly the same as Promise.all() but the difference being, it returns the first promise which fulfills or fails.

```
Promise.race([
  importantAction("Roadside Coder"),
  likeTheVideo("Javascript Interview Questions"),
  shareTheVideo("Javascript Interview Questions"),
])
  .then((res) => {
    console.log(res);
  })
  .catch((err) => {
    console.error("Error: Promises failed", err);
  });
```

→ Thus, it will output only that promise's response which succeeds or fails the first.

3) Promise.allSettled(): It is exactly same as Promise.all(), the difference being that it returns all the promises even if one gets rejected unlike Promise.all() which does not go to the attached callback fn inside .then() if 1 promise fails.

```
Promise.allSettled([
  importantAction("Roadside Coder"),
  likeTheVideo("Javascript Interview Questions"),
  shareTheVideo("Javascript Interview Questions"),
])
  .then((res) => {
    console.log(res);
  })
  .catch((err) => {
    console.error("Error: Promises failed", err);
  });
```

```
script.js:36
▼ (3) [{…}, {…}, {…}]
  ▶ 0: {status: 'fulfilled', value: 'Subs
  ▶ 1: {status: 'rejected', reason: 'Like
  ▶ 2: {status: 'fulfilled', value: 'Shar
    length: 3
  ▶ [[Prototype]]: Array(0)
```

(o/p)

4) Promise.any(): It is same as Promise.race() but it returns the first fulfilled promise & ignores all the rejected promises. If all promises fail, then it returns error that all promises were rejected.

```
Promise.any([
  importantAction("Roadside Coder"),
  likeTheVideo("Javascript Interview Questions"),
  shareTheVideo("Javascript Interview Questions"),
])
  .then((res) => {
    console.log(res);
  })
  .catch((err) => {
    console.error("Error: Promises failed", err);
  });
```

```
▼Error: Promises failed    script.js:39
AggregateError: All promises were
rejected
    (anonymous)          @ script.js:39
Promise.catch (async)
    (anonymous)          @ script.js:38
```

(if all promises return rejected state)

→ Async/Await : A more modern approach of handling promises when we want promises to be executed one after the another.

```
const result = () => {
  const message1 = importantAction("Roadside Coder");
};
```

→ As importantAction() returns a promise if directly store it inside a variable it will give us the Promise object. We can use async/await instead of .then() to get the value when promise gets resolved.

```
async function fewaef(params) {}

const result = async () => {
  const message1 = await importantAction("Roadside Coder");
};
```

→ example of using async/await in normal fn & arrow fn, so now this fn behaves asynchronously & message1 gets the value when promise is resolved.

```
30 const result = async () => {
31   const message1 = await importantAction("Roadside Coder");
32   const message2 = await likeTheVideo("Javascript Interview Questions");
33   const message3 = await shareTheVideo("Javascript Interview Questi
34
35   console.log({ message1, message2, message3 });
36 };
37
38 result();
```

result(): Promise<void>

→ Now this will work as a promise chain & all are executed one after the other.

```
script.js:35
{message1: 'Subscribe to Roadside Code
r', message2: 'Like the Javascript Int
erview Questions video', message3: 'Sh
are the Javascript Interview Questions
video'}
  message1: "Subscribe to Roadside Code
  message2: "Like the Javascript Interv
  message3: "Share the Javascript Inter
  [[Prototype]]: Object
```

→ o/p of above.

→ But if one promise fails:

```
Uncaught (in promise)        script.js:36
Like the Javascript Interview Questions
video
```

→ we get this error as we've not done error handling.

→ So, we wrap this inside try/catch block for error handling.

```
30 const result = async () => {
31   try {
32     const message1 = await importantAction("Roadside Coder");
33     const message2 = await likeTheVideo("Javascript Interview Questions");
34     const message3 = await shareTheVideo("Javascript Interview Questions");
35
36     console.log({ message1, message2, message3 });
37   } catch (error) {
38     console.error("Promises Failed", error);
39   }
40 };
```

→ Thus, this is the cleanest way of resolving promises.

→ Output:

```
4  console.log("start");
5
6  const promise1 = new Promise((resolve, reject) => {
7    console.log(1);
8    resolve(2);
9  });
10
11 promise1.then((res) => {
12   console.log(res);
13 });
14
15 console.log("end");
16
```

```
start                    script.js:4
1                        script.js:7
end                      script.js:15
2                        script.js:12
>
```

(o/p)

→ So first all synchronous is printed. JS encounters "start" & prints it, next in our execution context, Promise is created & "1" is printed.

→ Then, callback fn in line 11 is asynchronous so it is stored in the web API environment,

but since JS waits for no one, so "end" is printed.
→ Now our callback fn is in callback queue & as soon as callstack gets empty when all synchronous code is executed, event loop notices this & pushes callback fn's EC to the callstack & finally "2" is printed.

→ Output:

```
4  console.log("start");
5
6  const promise1 = new Promise((resolve, reject) => {
7    console.log(1);
8    resolve(2);
9    console.log(3);
10 });
11
12 promise1.then((res) => {
13   console.log(res);
14 });
15
16 console.log("end");
17
```

```
start                script.js:4
1                    script.js:7
3                    script.js:9
end                  script.js:16
2                    script.js:13
>
```

→ This also works exactly the same as before & hence o/p is this. Since the only async operation is the callback fn thus, it is executed in last.

→ Output, when there is a slight variation in above:

```
4  console.log("start");
5
6  const promise1 = new Promise((resolve, reject) => {
7    console.log(1);
8    console.log(3);
9  });
10
11 promise1.then((res) => {
12   console.log("Result:" + res);
13 });
14
15 console.log("end");
16
```

```
start                script.js:4
1                    script.js:7
3                    script.js:8
end                  script.js:15
>
```

→ So, here, since promise never gets resolved, so lines 11–13 will never work. Hence, we get the above output.

→ Output:

```
4  console.log("start");
5
6  const fn = () =>
7    new Promise((resolve, reject) => {
8      console.log(1);
9      resolve("success");
10   });
11
12 console.log("middle");
13
14 fn().then((res) => {
15   console.log(res);
16 });
17
18 console.log("end");
19
```

```
start                script.js:4
middle               script.js:12
1                    script.js:8
end                  script.js:18
success              script.js:15
>
```

(o/p)

→ So, here first "start" gets printed then "middle" gets printed, then we invoke fn(), so promise is created & "1" is printed.
→ After that our callback fn gets stored in the call back queue & when "end" is printed, our synchronous code ends, event loop pushes our callback fn to the callstack & "success" is printed.

→ Output :

```
function job() {
  return new Promise(function (resolve, reject) {
    reject();
  });
}

let promise = job();

promise
  .then(function () {
    console.log("Success 1");
  })
  .then(function () {
    console.log("Success 2");
  })
  .then(function () {
    console.log("Success 3");
  })
  .catch(function () {
    console.log("Error 1");
  })
  .then(function () {
    console.log("Success 4");
  });
```

```
Error 1                    script.js:23
Success 4                  script.js:26
>
```

(o/p)

→ Here, job() returns a promise which only has reject().

→ So, "promise" variable has the promise returned.

→ Now, since promise is rejected, so catch() will run. But as we have another .then() after catch() so the last then() will also run. Hence, we get this output.

→ Output :

```
function job(state) {
  return new Promise(function (resolve, reject) {
    if (state) {
      resolve("success");
    } else {
      reject("error");
    }
  });
}

let promise = job(true);

promise
  .then(function (data) {
    console.log(data);

    return job(false);
  })
  .catch(function (error) {
    console.log(error);

    return "Error caught";
  })
  .then(function (data) {
    console.log(data);

    return job(true);
  })
  .catch(function (error) {
    console.log(error);
  });
```

```
// success
// error
// Error caught
```

(o/p)

→ Here, we are assigning 'job(true)' to 'promise'. So it returns a resolved promise.

→ As soon as we get a resolved promise, callback fn attached to .then() works & "success" is consoled. This callback fn again returns a promise.

→ Now, line 20 returns a rejected promise, so callback attached to .catch() works & "error" is printed. "Error caught" is returned.

→ Now line 28 is treated as a resolved promise, so callback fn attached to .then() runs & "Error caught" is printed. Now, this callback fn returns a resolved promise on line 30. But since, there is no .then() after line 31 so nothing is printed & promise chain ends.

→ Output :

```
function job(state) {
  return new Promise(function (resolve, reject) {
    if (state) {
      resolve("success");
    } else {
      reject("error");
    }
  });
}

let promise = job(true);

promise
  .then(function (data) {
    console.log(data);

    return job(true);
  })
.then(function (data) {
  if (data !== "victory") {
    throw "Defeat";
```

```
success                    script.js:18
Defeat                     script.js:32
error                      script.js:40
Error caught               script.js:44
Success: test              script.js:48
>
```

(o/p)

→ This works the same as before.

We just have to keep few things in mind.

. This will throw error & it will go to the

```
  }
  return job(true);
})
.then(function (data) {
  console.log(data);
})
.catch(function (error) {
  console.log(error);
  return job(false);
})
.then(function (data) {
  console.log(data);
  return job(true);
})
.catch(function (error) {
  console.log(error);
  return "Error caught";
})
.then(function (data) {
  console.log(data);
  return new Error("test"); // Not Returning a promise
})
.then(function (data) {
  console.log("Success:", data.message);
})
.catch(function (data) {
  console.log("Error:", data.message);
});
```

next.catch()

→ This is a fulfilled promise

→ Not a rejected promise. This is just a normal text.

→ Implement this logic:

So, we have to create a promise named 'firstPromise' which resolves on a text. Then we creat a promise named 'secondPromise' which resolves on the firstPromise. Then we resolve our 'secondPromise', o/p of which we need to pass to 'firstPromise' & print the value of text.

```
const firstPromise = new Promise((resolve, reject) => {
  resolve("First!");
});

const secondPromise = new Promise((resolve, reject) => {
  resolve(firstPromise);
});

secondPromise
  .then((res) => {
    return res;
  })
  .then((res) => console.log(res));
```

→ When secondPromise gets resolved, callback fn in line 13 is run.

→ Now line 14 will return a promise as well, as soon as it is resolved, we go to the next .then() callback fn.

→ We get the value when firstPromise is resolved, and since the returned value is "First!", so it gets printed.

→ Implement this using async/await instead of then/catch:

```
function loadJson(url) {
  return fetch(url).then((response) => {
    if (response.status == 200) {
      return response.json();
    } else {
      throw new Error(response.status);
    }
  });
}

loadJson("https://fakeurl.com/no-such-user.json").catch((err) =>
  console.log(err)
);
```

=

```
async function loadJson(url) {
  let response = await fetch(url);

  if (response.status == 200) {
    let json = await response.json();
    return json;
  }

  throw new Error(response.status);
}

loadJson("https://fakeurl.com/no-such-user.json").catch((err) =>
  console.log(err)
);
```

→ Implement a fn which takes an array of promises & resolves them recursively:

```
function importantAction(username) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve(`Subscribe to ${username}`);
    }, 1000);
  });
}

function likeTheVideo(video) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve(`Like the ${video} video`);
    }, 1000);
  });
}

function shareTheVideo(video) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve(`Share the ${video} video`);
    }, 1000);
  });
}
```

→ Assuming these are the promises that will be passed to our fn.

```
function promRecurse(funcPromises) {
  // Write Implementation Here
}
```

→ Passing the promises to our fn in an array.

```
promRecurse([
  importantAction("Roadside Coder"),
  likeTheVideo("Javascript Interview Questions"),
  shareTheVideo("Javascript Interview Questions"),
]);
```

→ promRecurse() implementation :

```
function promRecurse(funcPromises) {
  if (funcPromises.length === 0) return;

  const currPromise = funcPromises.shift();

  currPromise
    .then((res) => console.log(res))
    .catch((err) => console.error(err));

  promRecurse(funcPromises);
}
```

→ promRecurse() accepts an array of promises as parameters

→ We check whether funcPromises is an empty array or not (base condition of recursion)

→ In the 31, when we use shift() we get the zeroth index's promise in currPromise. And funcPromises now only has 2 elements.

→ So when the zeroth index's promise is resolved, callback fn attached to then() is run & "res" is consoled.

→ Finally, we recursively call promRecurse() again with the remaining array (funcPromises) after using shift().

→ Polyfill of Promise :

```
const examplePromise = new PromisePolyFill((resolve, reject) => {
  setTimeout(() => {
    resolve(2);
  }, 1000);
});

examplePromise
  .then((res) => {
    console.log(res);
  })
  .catch((err) => console.error(err));
```

→ Example of how promise polyfill is used.

→ Callback fn inside "PromisePolyFill" is the 'executor' that is passed to 'PromisePolyfill'.

```
function PromisePolyFill(executor) {
  let onResolve,
    onReject,
    isFulfilled = false,
    isRejected = false,
    isCalled = false,
    value;

  function resolve(val) {
    isFulfilled = true;
    value = val;

    if (typeof onResolve === "function") {
      onResolve(val);
      isCalled = true;
    }
  }

  function reject(val) {
    isRejected = true;
    value = val;
    if (typeof onReject === "function") {
      onReject(val);
      isCalled = true;
    }
  }

  this.then = function (callback) {
    onResolve = callback;

    if (isFulfilled && !isCalled) {
      isCalled = true;
      onResolve(value);
    }

    return this;
  };

  this.catch = function (callback) {
    onReject = callback;

    if (isRejected && !isCalled) {
      isCalled = true;
      onReject(value);
    }

    return this;
  };

  try {
    executor(resolve, reject);
  } catch (error) {
    reject(error);
  }
}
```

→ Polyfill of promise.

→ This code handles async as well as sync operations.

→ For an async operation, our callback fn in line 9-11 gets stored in the callback queue & first our synchronized code runs, so first the .then() fn runs.

→ we receive the callback here & store it in "onResolve" variable. Since "isFulFilled" is false, so if condition does not run (which is for sync operations) & we return 'this'.

→ Here 'this' is the object created when the constructor fn is called (PromisePolyfill) using 'new' keyword. The object contains 'then' & 'catch' methods. We also return 'this' to enable fn chaining

→ Now, when our .then() has been called, our synchronous code ends & event loop pushes the callback fn which is inside callback queue to the callstack. And so our 'resolve' fn runs.

→ We store the arguments' value in 'value' & set isFulfilled to true (for handling for sync operation later). Then we check if "onResolve" is a fn or not (for handling sync operation later), since it is a fn right now, so "onResolve" is called with value & "isCalled" is set to true, so that "onResolve" may not run again. When onResolve() is called, we call the callback fn & "res" is consoled.

→ Now, for a sync operation lets imagine if there was no setTimeout & directly resolve() was called.

→ Now, in this case even before .then() is called our resolve() is called because callback fn or executor isn't going to the callback queue.

→ So, resolve() is called & "isFulfilled" is set to true & we store arguments' value in 'value'. But now if condition does not run as onResolve is not a fn right now (it has not been assigned with callback's value as .then() has not run yet)

→ Now, we go to .then() in synchronous code on line 15, so our lines 31-40 are run. We store our callback in "onResolve".

→ Now, the if condition runs as 'isFulfilled' has become true alread & 'isCalled' is false (i.e 'onResolve()' has not been called). So now we set "isCalled" to true & call 'onResolve()' with our arguments' value.

→ Same happens with reject() async & sync operations.

→ We also set our "executor (resolve, reject)" in try {} -- catch {} to catch any error.

→ If we directly use Promise.resolve() or Promise.reject(), to handle that:

```
71
72 PromisePolyFill.resolve = (val) => {
73   return new PromisePolyFill(function executor(resolve, reject) {
74     resolve(val);
75   });
76 };
77
78 PromisePolyFill.reject = (val) => {
79   return new PromisePolyFill(function executor(resolve, reject)
80     reject(val);
81   });
82 };
83
```

→ So, now if they are directly called, then our new Promise object is created (since this will be async, so .then() will run first) & resolve() will be called directly.

→ Polyfill for Promise.all() :