1).

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

let promise = job(); // calling job, resulting in a rejected promise

promise
  .then(function() {
    console.log('Success 1'); // This won't be executed because the promise is rejected
  })
  .then(function() {
    console.log('Success 2'); // Neither this one
  })
  .then(function() {
    console.log('Success 3'); // Nor this one
  })
  .catch(function() {
    console.log('Error 1'); // This will be executed
  })
  .then(function() {
    console.log('Success 4'); // This will be executed regardless of whether there was an error or not
  });
```

What is the output of the code above ?

ANS:

OUTPUT

Error 1

Success 4

2). Write a sleep function using a promise in javascript?

ANS:

We can create a sleep function in JavaScript using a Promise by utilizing the `setTimeout` function.

```
function sleep(ms) {
  return new Promise(resolve => setTimeout(resolve, ms));
}

// Usage example:
console.log('Start');
sleep(2000).then(() => {
  console.log('Sleeping for 2 seconds...');
  console.log('Woke up after 2 seconds');
});
```

Within this code:

The number of milliseconds to sleep, denoted by the argument ms, is required by the sleep function. Using new Promise, a Promise is created inside the sleep function. The Promise's resolution can be postponed by ms milliseconds using the setTimeout function.

After the allotted amount of time, the resolve function that was supplied to setTimeout is invoked, resolving the Promise.
Sleep(2000) provides a Promise that resolves in two seconds when called. After that, you can chain.then() to carry out tasks following a time of sleep.

3). What is the output of the following code?

```
const promise = new Promise(res => res(2));
promise.then(v => {
console.log(v);
return v * 2;
})
.then(v => {
console.log(v);
return v * 2;
})
.finally(v => {
```

```
console.log(v);
return v * 2;
})
.then(v => {

console.log(v);
});
```

ANS

The code given creates a Promise with a resolution function called res that resolves to the value 2 right away. The Promise is then chained to a number of handlers, each of which computes the resolved value. A finally handler is appended to the chain at the end.

Output will be

2

4

undefined

8

4).

```
console.log('start')
const fn = () => (new Promise((resolve, reject) => {
console.log(1);
resolve('success')
}))
console.log('middle')
fn().then(res => {
console.log(res)
})

console.log('end')
```

What is the output of this code snippet?

ANS:

"start" appears in the console log.
A function that yields a Promise is called fn. The console.log(1) function in this Promise executor function logs 1 to the console.
The console receives the log "middle".

The Promise chain is started when fn() is called. Consequently, the Promise executor's console.log(1) logs 1 to the console.
As resolve('success') is called, the Promise's value of'success' is resolved right away.
"end" appears in the console log.
Lastly, the Promise that fn() returned is chained to the then method. The callback function within

then is triggered when the Promise is resolved, logging "success" to the console.
Thus, the final product is:

start

middle

1

end

success

5). Write a function delay that returns a promise. And that promise should return a setTimeout that calls resolve after 1000ms.

ANS:

```
function delay(milliseconds) {
   return new Promise(resolve => {
      setTimeout(resolve, milliseconds);
   });
}


// Usage example:
console.log('Start');
delay(1000).then(() => {
   console.log('Resolved after 1000ms');
});
```

Within this code:

Before resolving the Promise, the delay function accepts a millisecond parameter that represents the delay.
Using new Promise, a promise is created inside the delay method. The resolution of the Promise can be postponed by milliseconds using the setTimeout function.

After the allotted amount of time, the resolve function that was supplied to setTimeout is invoked, resolving the Promise.
It returns a Promise that resolves in 1000 milliseconds when delay(1000) is called. After the delay, you can execute operations by chaining the.then() method.

6).

```
Promise.resolve('Success!')
.then(data => {
```

```
return data.toUpperCase()
})
.then(data => {
console.log(data)
})
```
What will the output be?

ANS:

SUCCESS!

The summary is as follows:

The expression "Success!" is returned as soon as Promise.resolve('Success!') generates a Promise. After invoking toUpperCase() to change the resolved value of "Success!" to uppercase, the first handler receives it and returns the outcome.

The result of the preceding then handler, which is 'SUCCESS!' (in uppercase), is sent to the second then handler, which uses console.log(data) to log it to the console.
'SUCCESS!' is the end result, then. Take note that the conversion done in the first then handler is the reason the string is in uppercase.

7).

```
var p = new Promise((resolve, reject) => {
reject(Error('The Fails!'))
})
.catch(error => console.log(error))
.then(error => console.log(error))
```
What will the output be?

ANS:

Error: The Fails!

Undefined

The summary is as follows:

An executor function is used to construct a Promise p, which promptly rejects with an error object containing the message "The Fails!"
Catching the rejection, the catch handler logs the error message 'Error: The Fails!' to the console. The reason for this is that an error object is passed to the catch handler when the reject function is called.

A then handler is then called. But the handler receives undefined since the prior operation—the rejection—did not yield a value.
The handler then logs an undefinable value to the console.
Thus the end result is undefined and 'Error: The Fails!' in that order.

8).

```
console.log('start')
setTimeout(() => {
console.log('setTimeout')
})
Promise.resolve().then(() => {
console.log('resolve')
})
console.log('end')
```

What will the output be?

ANS:

start

end

resolve

setTimeout

This is the reason why:

The console receives the log "start."
'setTimeout' is logged by calling the setTimeout function with an arrow function. This function will not execute until the main thread's execution is finished, even though it is scheduled to run asynchronously after a 0 millisecond timeout.

Promise.resolve() instantly resolves a Promise. Additionally scheduled to run asynchronously once the current script has completed processing is the then handler of this Promise.
The log reads "end" in the console.
Asynchronous operations (like the setTimeout callback and the then handler of the resolved Promise) are carried out after the main thread's execution is complete. Promise handlers are executed before setTimeout callbacks because they take precedence over tasks in the microtask queue. Consequently,

10).

```
console.log('start')
Promise.resolve(1).then((res) => {
console.log(res)
})
Promise.resolve(2).then((res) => {
console.log(res)
})
console.log('end')
```

What will the output be?

ANS:

start

end

1

2

The console receives the log "start."
Promise.resolve() generates two Promises. The numbers 1 and 2, respectively, immediately settle these promises.
The log reads "end" in the console. This occurs right away following the creation of the Promises but prior to the handlers being run.

Asynchronous calls are made to the then handlers of both Promises as soon as the JavaScript event loop is made available. They record to the console, respectively, the numbers 1 and 2.
Thus, "start," "end," 1, and 2 are the final result, in that sequence.