Adv Web Dev Arch
Lecture 6 (updated 2026)

how to implement
Microservices communication asynchronously
(intro to promises in JS)

Amir Amintabar, PhD

高级 Web 开发架构
第6讲（更新于2026年）

如何实现
微服务间的异步通信
（JavaScript 中 Promise 入门）

阿米尔・阿明塔巴尔，博士

Outline

- 1 Review
- 2 Asynch operations
- 3 Promises in JS
- 4 Promise status
- 5 .catch() and .finally()
- 6. In class activity
- 7. remarks on **response size** of GET vs POST requests
- 8. W3Schools website seems inaccurate

- Quizzes on promise during labs, this week
- No new labs this week. Next lab will be assigned after the midterm
- Midterm exam administrated on Learning Hub

大纲

- 1 复习
- 2 异步操作
- 3 JavaScript 中的 Promise
- 4 Promise 的状态
- 5 .catch() 和 .finally()
- 6。 课堂活动
- 7。 关于 GET 与 POST 请求的 **响应大小** 的说明
- 8。 W3Schools 网站内容似乎不够准确

- 。 本周实验课中有关 Promise 的测验
- 。 本周无新实验任务；下一次实验将在期中考试后布置
- 。期中考试在学习中心（Learning Hub）进行

review

1

- myDomain.com/api/customers

resource

- Every http request has a verb or method that determines its intention
- GET,PUT,DELETE,POST
- PUT: replacing an object entirely with something new
- PATCH: updating only a property of an object
- PUT vs POST
- PUT is idempotent, POST is not
- calling PUT once or several times successively has the same effect (that is no side effect), whereas successive identical POST requests may have additional effects, e.g. leading to placing an order several times.

Idempotent: can be applied multiple times without changing the result

审核

1

- myDomain.com/api/customers

资源

- 每个 HTTP 请求都包含一个动词（即方法），用于表明其意图
- GET ,PUT, DELETE ,POST
- PUT：用新内容完全替换某个对象
- PATCH：仅更新对象的某个属性
- PUT 与 POST 的区别
- PUT 是幂等的，而 POST 不是
- 连续调用一次或多次 PUT 具有相同的效果（即不会产生副作用），而连续发送多个相同的 POST 请求则可能产生额外影响，例如导致订单被重复提交。

幂等性：可被多次应用而不改变结果

Remember in JS can a function receive another function as parameter? first-class functions

- Example:

```
function foo(x) {
     alert(x);
}
function bar(func) {
     func("Hello World!");
}

//alerts "Hello World!"
bar(foo);
```

还记得在 JavaScript 中，函数可以将另一个函数作为参数接收吗? first-class functions

- 示例:

```
function foo(x) {
     alert(x);
}
function bar(func) {
     func("Hello World!");
}

//alerts "Hello World!"
bar(foo);
```

Remember Call backs

```
• function myDisplayer(some) {
    document.getElementById("demo").innerHTML = some;
  }
  function myCalculator(num1, num2, myCallback) {
    let sum = num1 + num2;
    myCallback(sum);
  }
  myCalculator(5, 5, myDisplayer);
```

- Example above says
- "here are arguments to do sth with them in this function"
- "here is the function to call once you are done"

记住回调函数

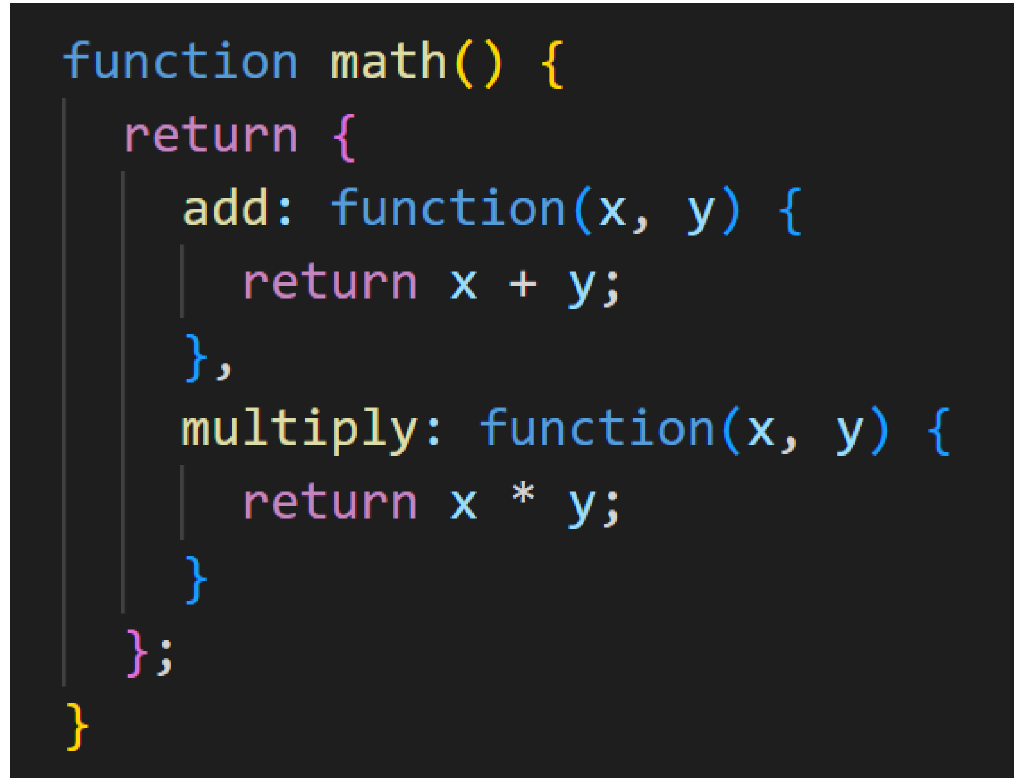
```
•函数 myDisplayer(some) {

    document.getElementById("demo").innerHTML = some;
  }
  functiyon m Cal(culator num1, 声明sum = num2, myCallback) {
    num1 + num2; myCallback(sum);

  }
  myCalculator(5, 5, myDisplayer);
```

- 上述示例说明：
- "此处为参数，用于在该函数中执行相应操作"
- "此处为完成操作后需调用的函数"

Function returning an object which has methods

- Q: What does this function return?

(what data type?)



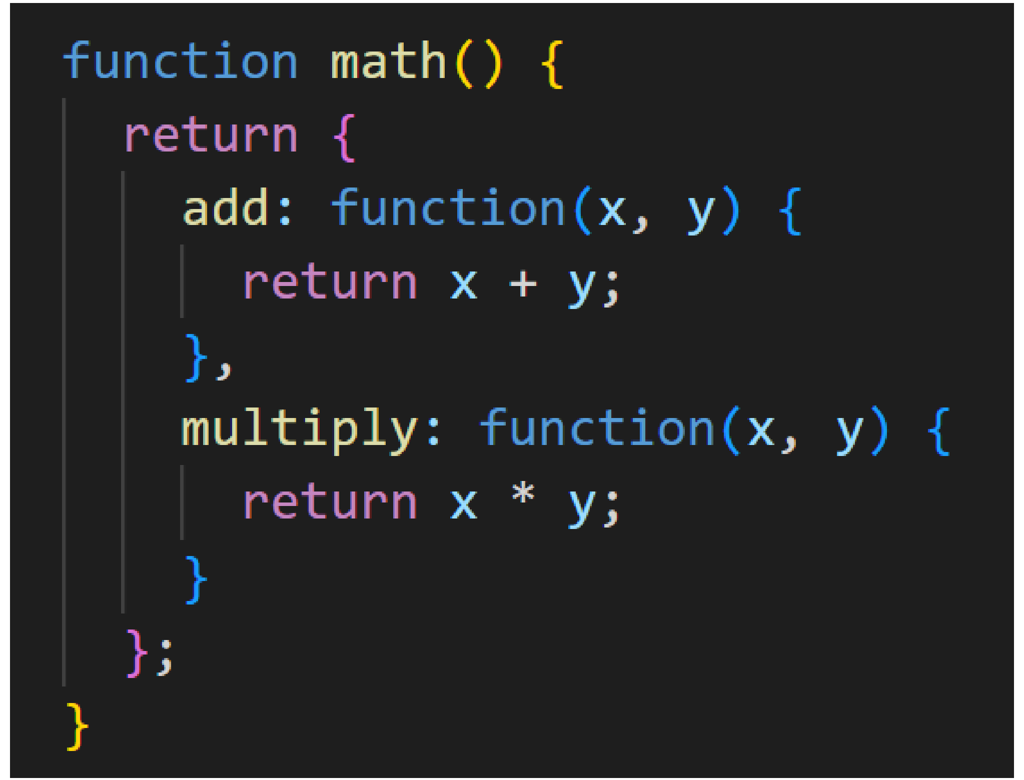
```
function math() {
  return {
    add: function(x, y) {
      return x + y;
    },
    multiply: function(x, y) {
      return x * y;
    }
  };
}
```

- Q: how would you add two numbers using the code snipped above?
- math().add(2,3)

返回一个包含方法的对象的函数

- Q：该函数返回什么?

（何种数据类型？）



```
function math() {
  return {
    add: function(x, y) {
      return x + y;
    },
    multiply: function(x, y) {
      return x * y;
    }
  };
}
```

- Q：如何使用上述代码片段实现两个数字相加?
- math().add(2, 3)

2

Asyc operations and callback hell!

2

异步操作与回调地狱！

What if a function does asynchronous operations ?

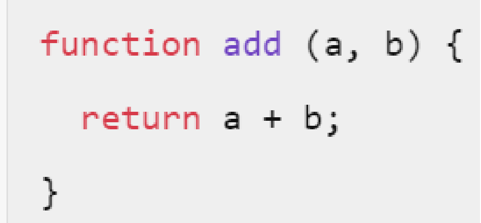
- Microservices communicate via the net using API calls
- and any communication via net is asynchronous

如果一个函数执行异步操作，会怎样?

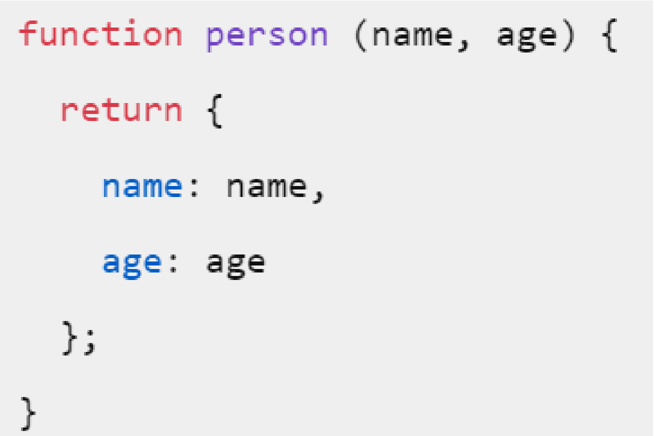
- 微服务通过网络使用 API 调用进行通信
- 且通过网络进行的任何通信均为异步方式。

problem

- In js Functions can only return a single value, whether of primitive data type (number, string etc) or a non- primitive data type such as an object.



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

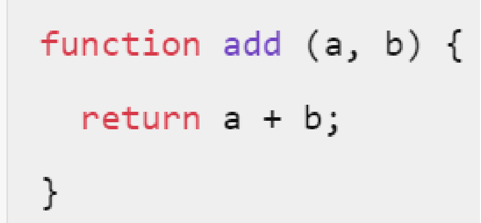


```
function person (name, age) {
  return {
    name: name,
    age: age
  };
}
```

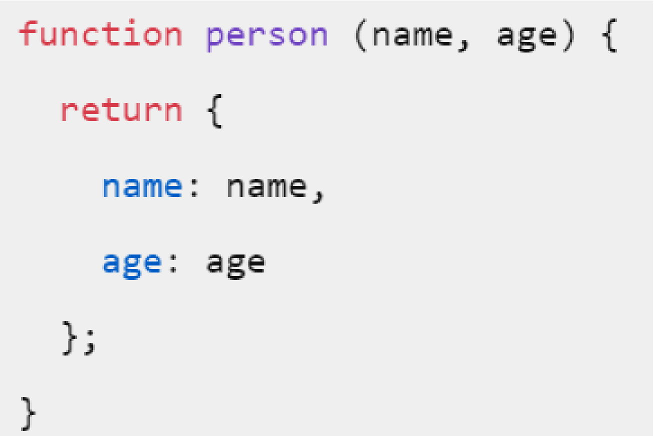
- Usually, we call the function in our code, and do stuff based on the retuned value. But what if the operation performed by a function is *asynchronous?* 😟
- Q: why functions doing asynchronous operations are more complicated?

问题

- 在 JavaScript 中，函数只能返回单个值，无论该值是原始数据类型还是其他类型。（如数字、字符串等）或非原始数据类型（例如对象）。



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



```
function person (name, age) {
  return {
    name: name,
    age: age
  };
}
```

- 通常，我们在代码中调用函数，并根据其返回值执行相应操作。但若函数执行的操作是异步的，又该如何处理？ 😟
- 问：为何执行异步操作的函数更为复杂？

- Dealing with functions doing asynchronous stuff, we are not only interested in the returned value, but **also in how it was returned**, was the operation successful or failed (timed-out, DB is down, authentication error etc)
- 1- Do something with the success
- 2- Or deal with the failure gracefully

- But functions return only one thing! (read with scream)
- Q: Whats your solution ?

- Lets simply return an object that has two callback methods, resolve() and reject()! We simply override them based on what we want ☺

- 在处理执行异步操作的函数时，我们不仅关注其返回值，**更关注该值是如何返回的**：操作是否成功完成，抑或失败（例如超时、数据库宕机、身份验证错误等）？

- 1- 在操作成功时执行相应处理
- 2- 或者以优雅的方式处理失败情况

- 但函数只能返回一个值！（请用夸张的语气朗读）
- 问：您的解决方案是什么?

- 我们只需返回一个包含两个回调方法的对象：resolve() 和 reject()！然后根据具体需求，简单地重写这两个方法即可 ☺

Promise in JavaScript

3

JavaScript 中的 Promise

3

Understanding promises

理解 Promise

Understanding Promises

- "Imagine yourself working in a company. Your boss **promises** you a raise someday.
- You don't know if you will get that raise until next quarter, you just wait(*pending*). Your boss can either really get you the raise(*fulfilled*), or stand you up and withhold (*reject*)it if she/he is not happy :(.

- That is a **promise**. A promise has 3 states:

1.Pending: You *don't know* if you will get the raise

2. Resolved (Fulfilled): Boss is happy, gets you the raise. describe what you would do in a function called resolve()

3.Rejected: Your boss is unhappy, withholds the raise. Lets describe what you would do in a function called reject()

理解 Promise

- "想象你在一家公司工作。你的老板 **承诺** 你将来会加薪。

- 你直到下个季度才能确定是否真能加薪，此时你只能等待（待定）。老板可能真的为你落实加薪（已兑现），也可能因不满意而食言并拒绝加薪（被拒绝）：（。

- 这就是一个**Promise**。Promise 具有三种状态:

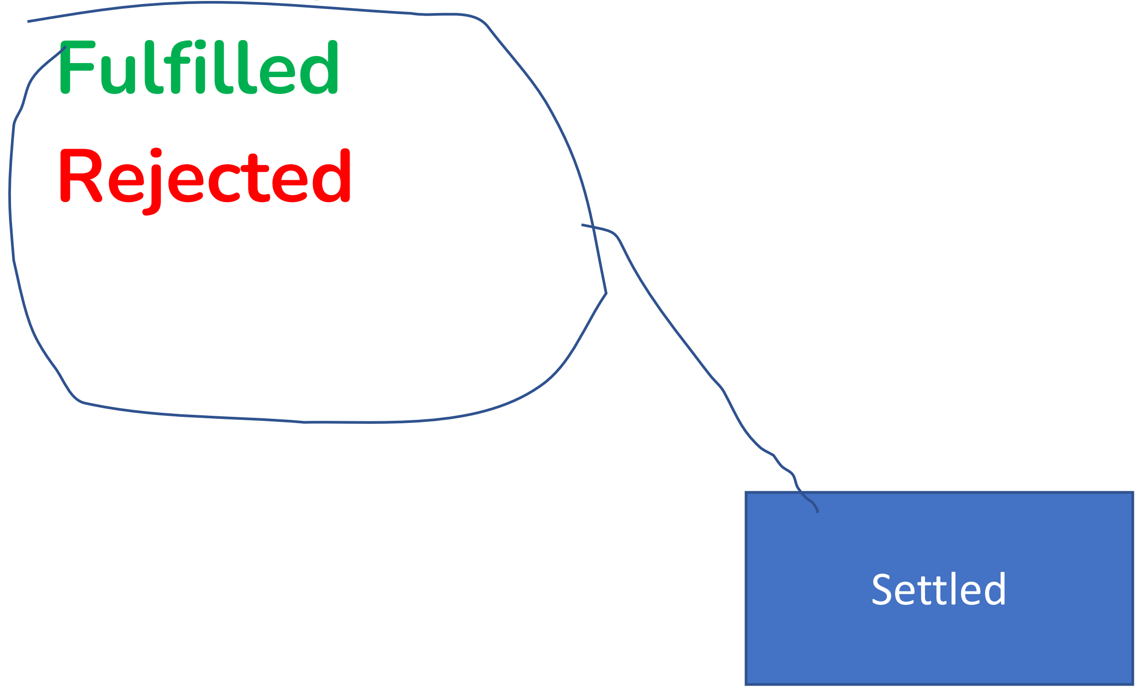
1. 待定（Pending）：你尚不确定能否获得加薪

2. 已解决（Resolved）（已兑现，Fulfilled）：老板满意，为你落实了加薪。请在名为 resolve() 的函数中描述你将执行的操作。

3. 被拒绝：你的老板不满意，因此暂缓加薪。请在名为 reject() 的函数中描述你将采取的行动。

Remember the three possible status of a JS promise

Pending



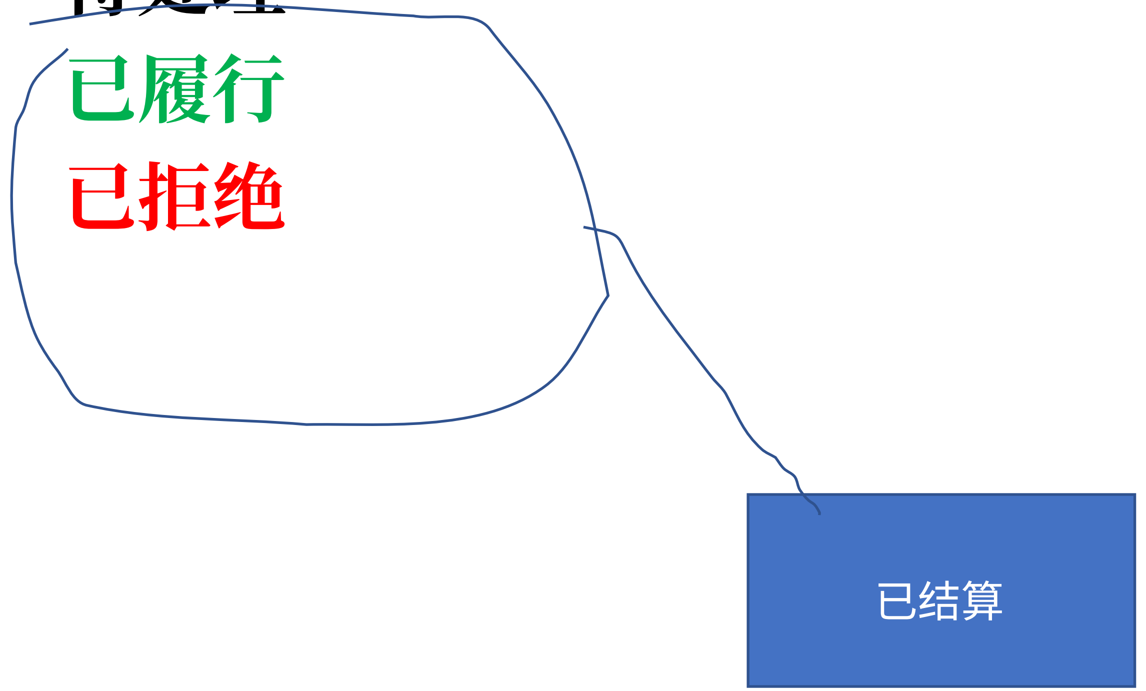
Fulfilled

Rejected

Settled

牢记 JavaScript Promise 的三种可能状态

待处理



已履行

已拒绝

已结算

- Js runs asynchronously. That means you never exactly know when things happen. Remember the restaurant with single waiter analogy. When you ask the waiter for sth, you never know when he is going to get back to you.
- You only get a promise. (he might get back to you , he might forget)
- When you make an AJAX call you never know when you are going to get back the result. Imagine other part of your code is pending for the result of this AJAX.

- JavaScript 以异步方式运行。这意味着你永远无法精确预知事件发生的时机。请回想一下“餐厅里只有一位服务员”的类比：当你向服务员提出请求时，你永远不知道他何时会回来回应你。
- 你得到的只是一个承诺。（他可能会回来回应你，也可能会忘记）
- 当你发起一个 AJAX 请求时，你永远无法预知结果何时返回。试想你的代码其他部分正等待该 AJAX 请求的结果。

Promise() is an object in JavaScript

1. The Promise object is used for asynchronous computations.
2. A Promise represents a single asynchronous operation that hasn't completed yet, but is expected in the future.
3. You pass a function to the constructor which gets executed the moment the promise object gets created
4. The executer function gets **reference of** two functions (**resolve** and **reject**) as input
5. The Promise object has a method called then()

let promise = new Promise(myExecuter(res~~()~~,rej~~()~~));

Promise() 是 JavaScript 中的一个对象

1. Promise 对象用于异步计算。
2. Promise 表示一个尚未完成、但预期将在未来完成的单一异步操作。
3. 您需向构造函数传入一个函数，该函数会在 Promise 对象创建的瞬间执行。
4. 执行器函数接收两个函数（**resolve** 和 **reject**）作为参数，这两个函数是其 **引用**。
5. Promise 对象有一个名为 then() 的方法

声明 一个 Promise = 新建 一个 Promise(执行器函数(成功回调(), 失败回调()));

Promise object constructor

```
let promise = new Promise(
   (resolve, reject)=> {
     // some condition to decide outcome of promise (reject or resolve)
     reject("Rejected!");
     resolve("Resolved!");
   }
);
promise.then(
   (resolveMes)=> {
     console.log(resolveMes);
   }
   ,
   (rejectMes)=> {
     console.log(rejectMes)
   }
);
```

1. Promise constructor to which we can pass a function called executer. This function itself gets two functions first one points to resolve, the second one points to reject

3. Promise executer function. Gets executed **immediately** upon creation of the promise object

4. resolve and reject invoke handlers **asynchronously** and **mutually exclusive** And run only once

Resolve, reject change promise status synchronously which lead to put handlers in microtask and run only after the current call stack finishes

Promise handlers do **NOT** wait for “all code in the program” to finish.

They wait only for the **current call stack** to unwind.

2. Definition of the two handlers (functions) resolve and promise are given here, or could be given elsewhere but passed by reference

In this example, when reject handlers invokes, the resolve will be ignored because the promise is already settled.

5. The then() method itself returns a promise

Note: blue color indicates resolve contents, Red indicates reject ones

Promise 对象构造函数

```
声明 一个 Promise = 新建 一个 Promise(
   (resolve, reject)=> {
     // 某些条件，用于决定 Promise 的结果（拒绝或解决）
     reject("已拒绝！");
     resolve("已解决！");
   }
);
promise.then(
   (resolveMes)=> {
     console.log(resolveMes);
   }
   ,
   (rejectMes)=> {
     console.log(rejectMes)
   }
);
```

1. Promise 构造函数，可传入一个名为执行器（executor）的函数。该函数自身接收两个函数参数：第一个指向 resolve，第二个指向 reject。

3. Promise 执行器函数：在 Promise 对象创建时**立即执行**。

4. resolve 和 reject 调用处理程序**异步执行**且**互斥**

仅执行一次

resolve 和 reject 会同步改变 Promise 的状态，从而将处理程序加入微任务队列，并仅在当前调用栈执行完毕后运行。Promise 处理程序 **不会** 等待“整个程序的所有代码”执行完毕，而仅等待 **当前调用栈清空**。

2. 两个处理程序的定义（**即函数**）：resolve 和 promise 在此处给出，也可在其他位置定义，再通过引用方式传入

本示例中，当调用 reject 处理程序时，resolve 将被忽略，因为 Promise 已处于终态（settled）。

5. then() 方法本身返回一个 Promise。

注意：蓝色表示 resolve 的内容，红色表示 reject 的内容。

- resolve / reject do not invoke handlers directly; they change the promise’s state , JavaScript then queues the handlers (as you know queuing ➔ asynch)

- In fact resolve(),reject() **create** the promise’s value; .then()handlers **receives** the promise’s value
-

resolve is one specific function created by js internally, the first callback passed to .then(callback1(), callback(2)) is a different function they exist at different times they play different roles

- resolve / reject 并不会直接调用处理函数；它们仅改变 Promise 的状态，随后 JavaScript 会将这些处理函数加入任务队列（如你所知，该队列属于 ➔ 异步机制）

- 事实上，resolve() 和 reject() **创建** Promise 的值；而 .then() 处理函数 **接收** 该 Promise 的值
-

resolve 是 JavaScript 内部创建的一个特定函数；传递给 .then(callback1(), callback2()) 的第一个回调函数是另一个不同的函数，二者存在于不同时刻，且扮演着不同的角色。

Eager vs lazy

- Promises are eager. This means that **when a new Promise is constructed, it immediately starts executing, attempting to resolve the** asynchronous value (meaning the resolve, reject methods are called asynchronously by a synchronous executer method)

急切执行 vs 惰性执行

- Promise 是急切执行的。这意味着**每当创建一个新的 Promise 时 constructed, it immediately starts executing, attempting to 其异步值便立即被解析**（即 resolve 和 reject 方法由一个同步执行器方法异步调用） **执行** **执行，**

Promise Examples

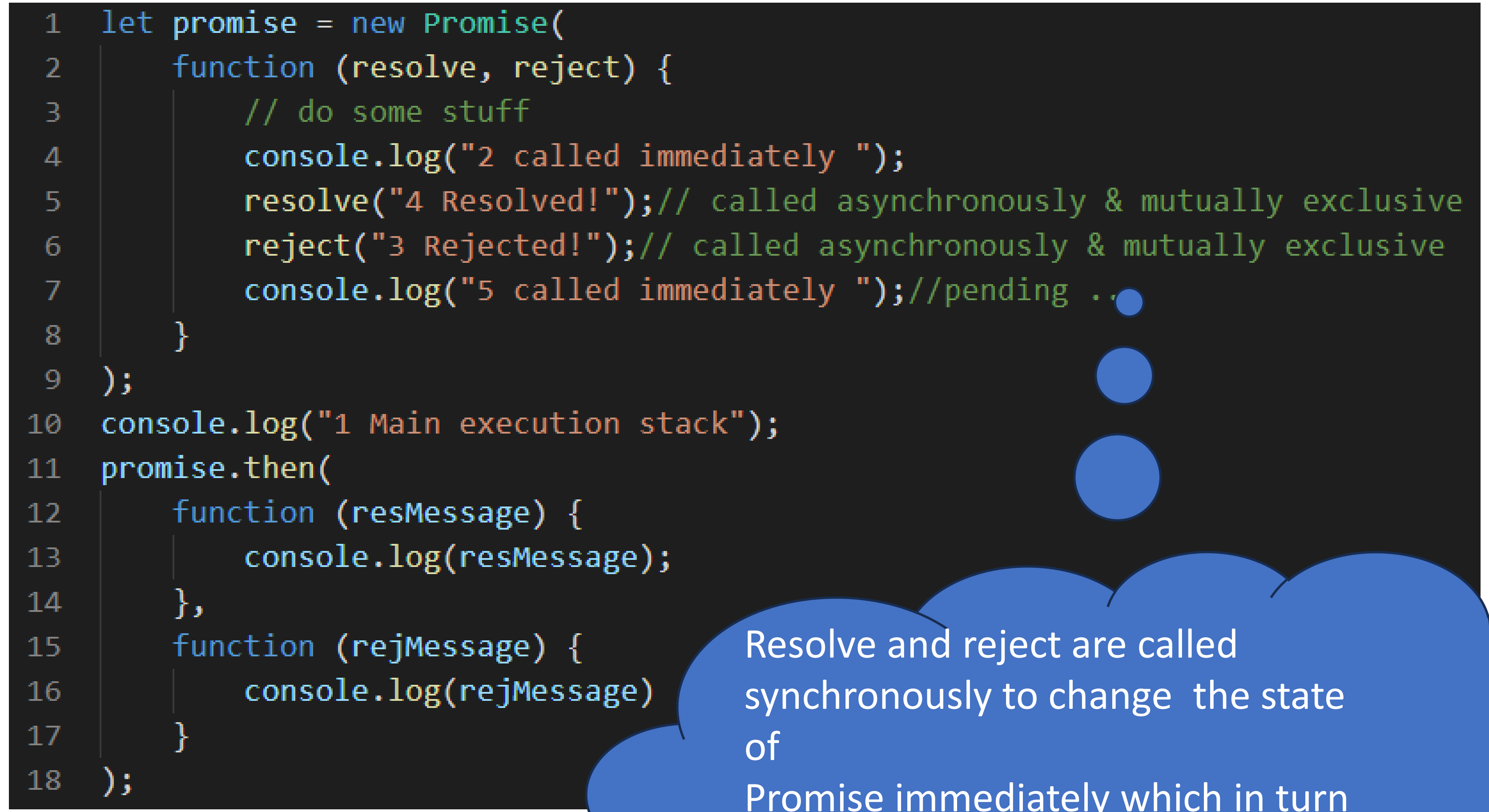
4

Promise 示例

4

Example 1:

- In what order the strings below will be logged?



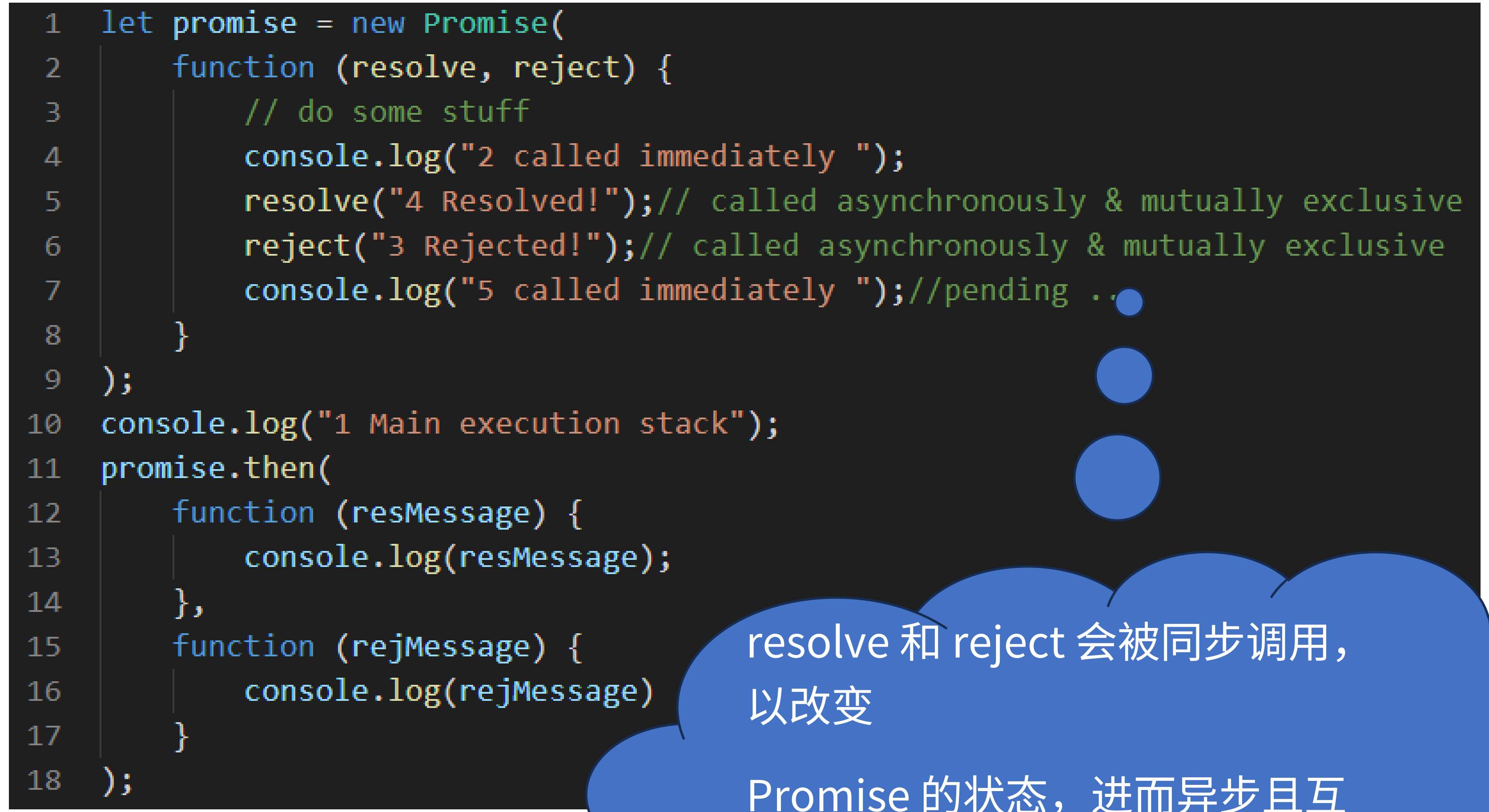
```
let promise = new Promise(
    function (resolve, reject) {
        // do some stuff
        console.log("2 called immediately ");
        resolve("4 Resolved!");// called asynchronously & mutually exclusive
        reject("3 Rejected!");// called asynchronously & mutually exclusive
        console.log("5 called immediately ");//pending ..
    }
);
console.log("1 Main execution stack");
promise.then(
    function (resMessage) {
        console.log(resMessage);
    },
    function (rejMessage) {
        console.log(rejMessage)
    }
);
```

Resolve and reject are called synchronously to change the state of Promise immediately which in turn invokes the corresponding handlers asynchronously and mutually exclusive

A:
2 (we said the executor will invoke the moment the promise constructor is called.
5 (2 and 5) execute synchronously
1 then this like executes synchronously as it is in the same execution stack
4 invokes asynchronously and mutually exclusive which means the main execution stack finishes first, then one of these two the invokes first, will block the other one from execution .

示例 1:

- 以下字符串将以何种顺序被输出?



```
let promise = new Promise(
    function (resolve, reject) {
        // do some stuff
        console.log("2 called immediately ");
        resolve("4 Resolved!");// called asynchronously & mutually exclusive
        reject("3 Rejected!");// called asynchronously & mutually exclusive
        console.log("5 called immediately ");//pending ..
    }
);
console.log("1 Main execution stack");
promise.then(
    function (resMessage) {
        console.log(resMessage);
    },
    function (rejMessage) {
        console.log(rejMessage)
    }
);
```

resolve 和 reject 会被同步调用，以改变

Promise 的状态，进而异步且互斥地调用相应的处理程序。

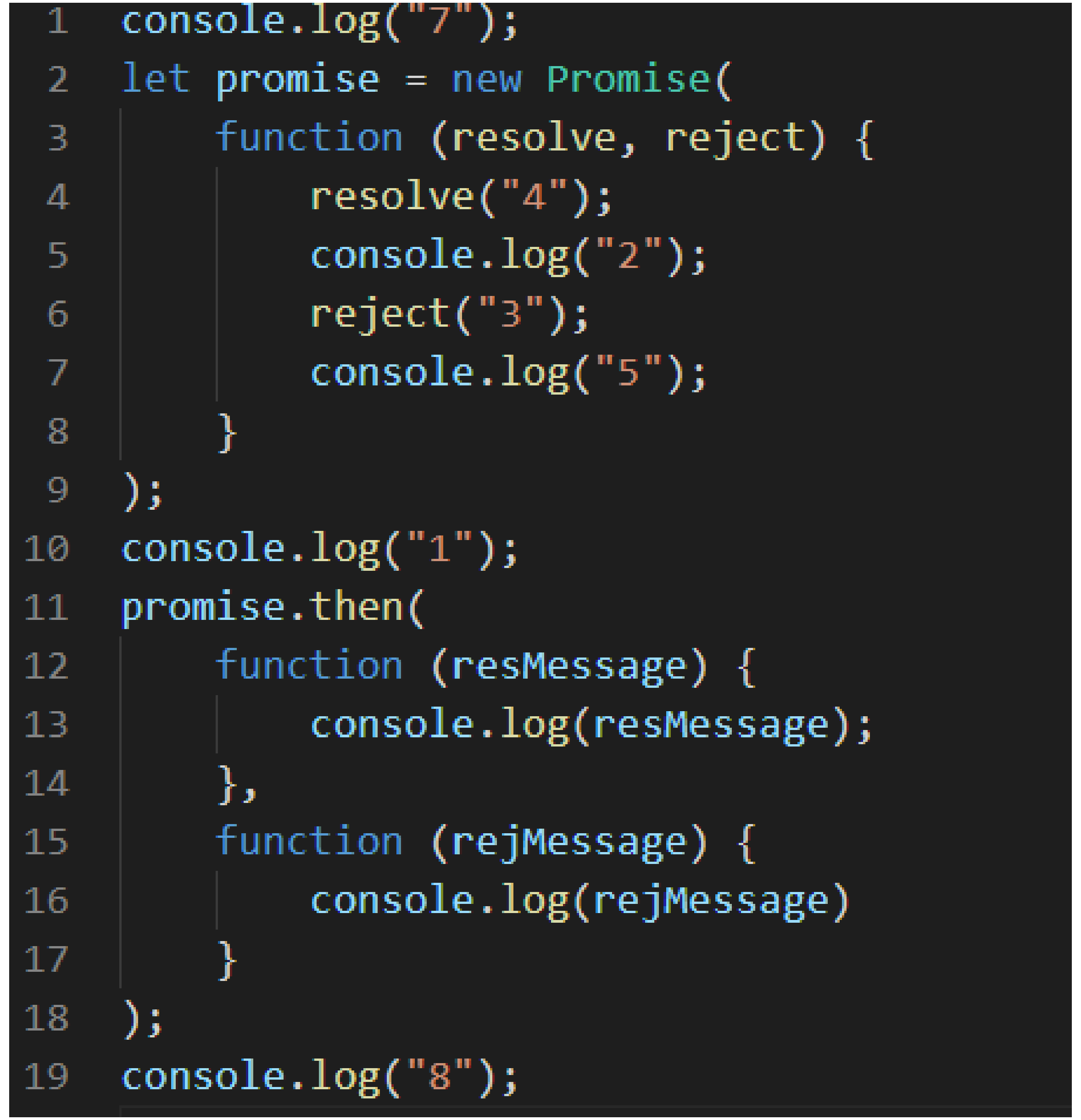
A:
2（我们已说明，执行器将在 Promise 构造函数被调用的瞬间立即执行。

5（2 和 5）同步执行**1** 随后，此行代码也同步执行，因为它位于同一执行栈中**4** 调用

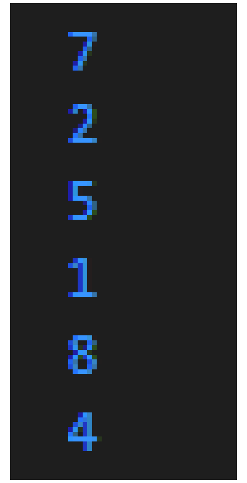
异步且互斥，这意味着主执行栈会先执行完毕

，然后这两个调用中先开始执行的那个，将阻塞另一个的执行。

Example 2: in what order the numbers will be logged?

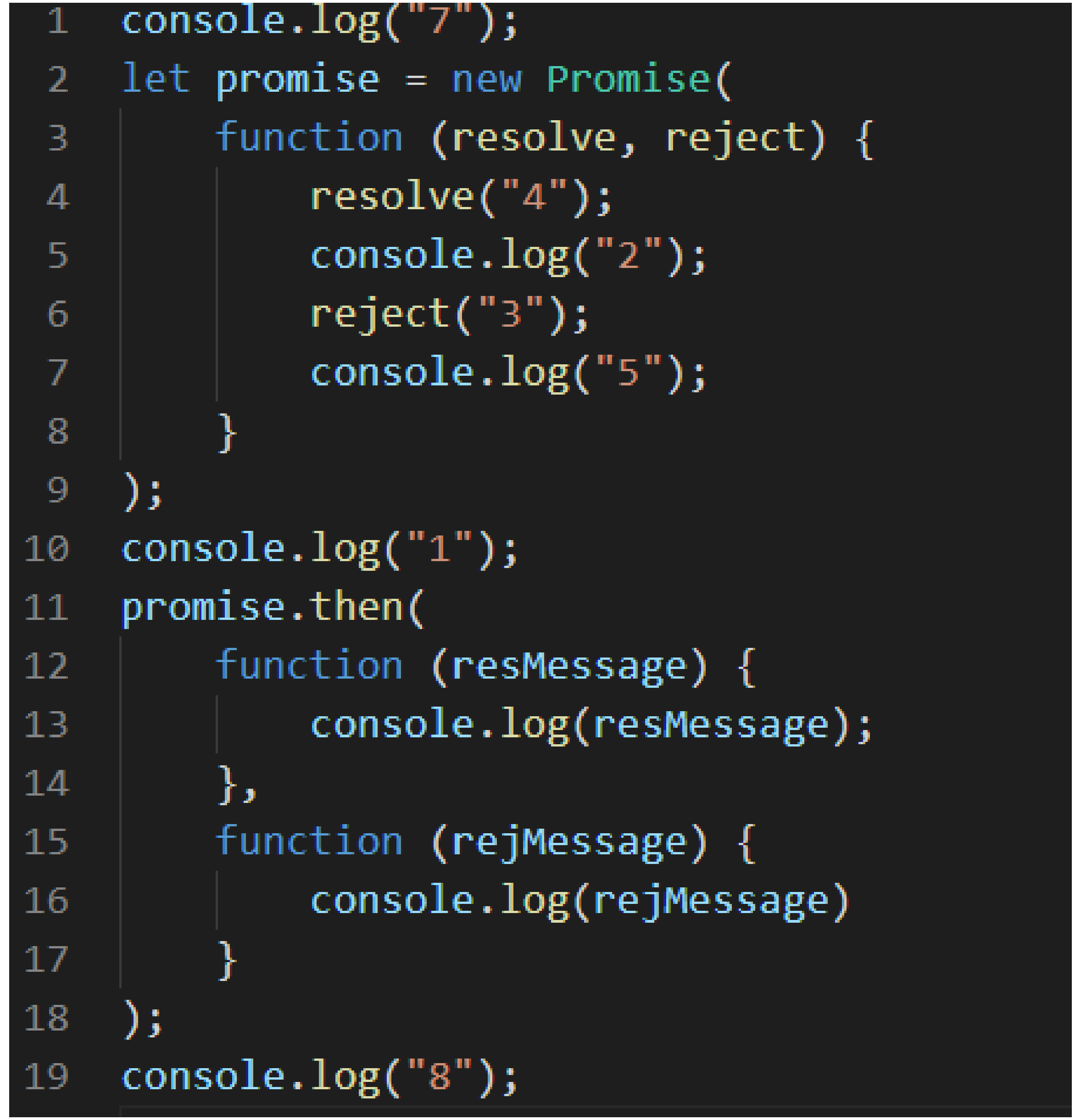


```
console.log("7");
let promise = new Promise(
    function (resolve, reject) {
        resolve("4");
        console.log("2");
        reject("3");
        console.log("5");
    }
);
console.log("1");
promise.then(
    function (resMessage) {
        console.log(resMessage);
    },
    function (rejMessage) {
        console.log(rejMessage)
    }
);
console.log("8");
```

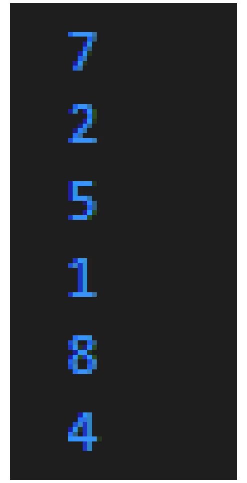


```
7
2
5
1
8
4
```

示例 2：数字将以何种顺序被记录？



```
console.log("7");
let promise = new Promise(
    function (resolve, reject) {
        resolve("4");
        console.log("2");
        reject("3");
        console.log("5");
    }
);
console.log("1");
promise.then(
    function (resMessage) {
        console.log(resMessage);
    },
    function (rejMessage) {
        console.log(rejMessage)
    }
);
console.log("8");
```



```
7
2
5
1
8
4
```

Example 3

```
let promise = new Promise(
    function (resolve, reject) {
        console.log(" 2 ");
        reject(" 3 ");
        resolve(" 4 ");
        console.log(" 5 ");
    }
);
promise.then(
    function (st) {
        console.log(st);
    },
    function (st) {
        console.log(st)
    }
);
console.log(" 1 ");
```

Q: In what order the console logs the numbers?

A:
2
5
1
3

Q: what happened to 4?

示例 3

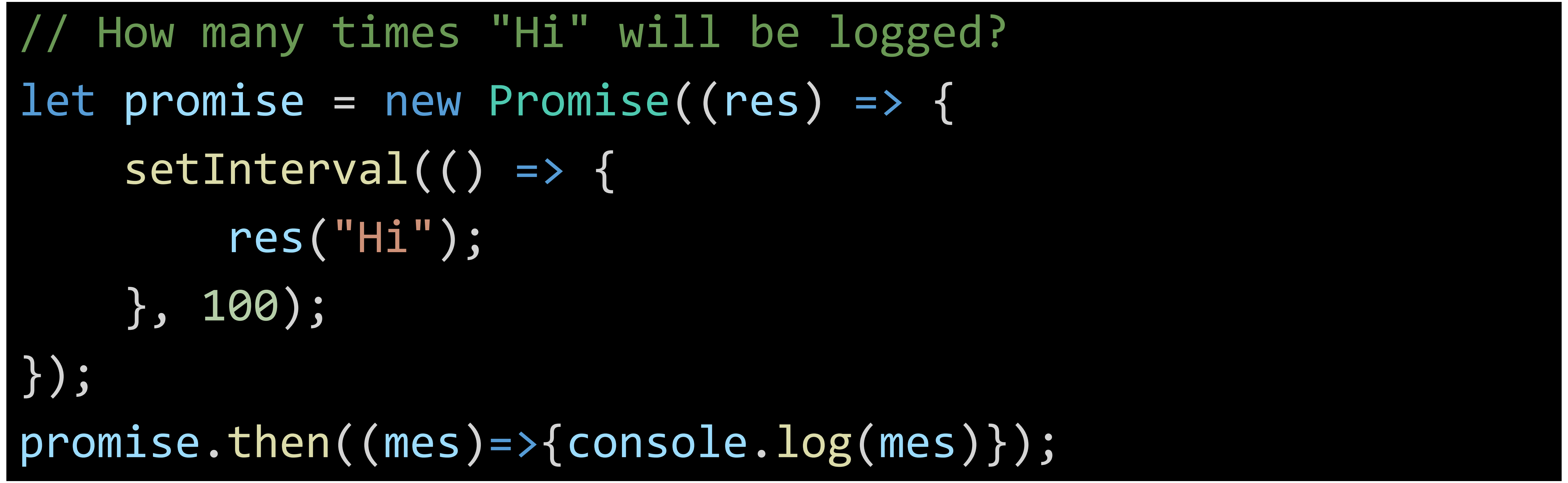
```
声明 一个 Promise = 新建 一个
Promise(函数 (resolve, reject)
{console.log(" 2 ");reject(" 3
");resolve(" 4 ");console.log("
5 ");});promise.then( 函数 (st)
{console.log(st);},函数 (st)
{
console.log(st)});console.log("
1 ");
```

Q: In what order the console logs the numbers?

A:
2
5
1
3

Q: what happened to 4?

Example 4: How many times 'Hi' will be logged?

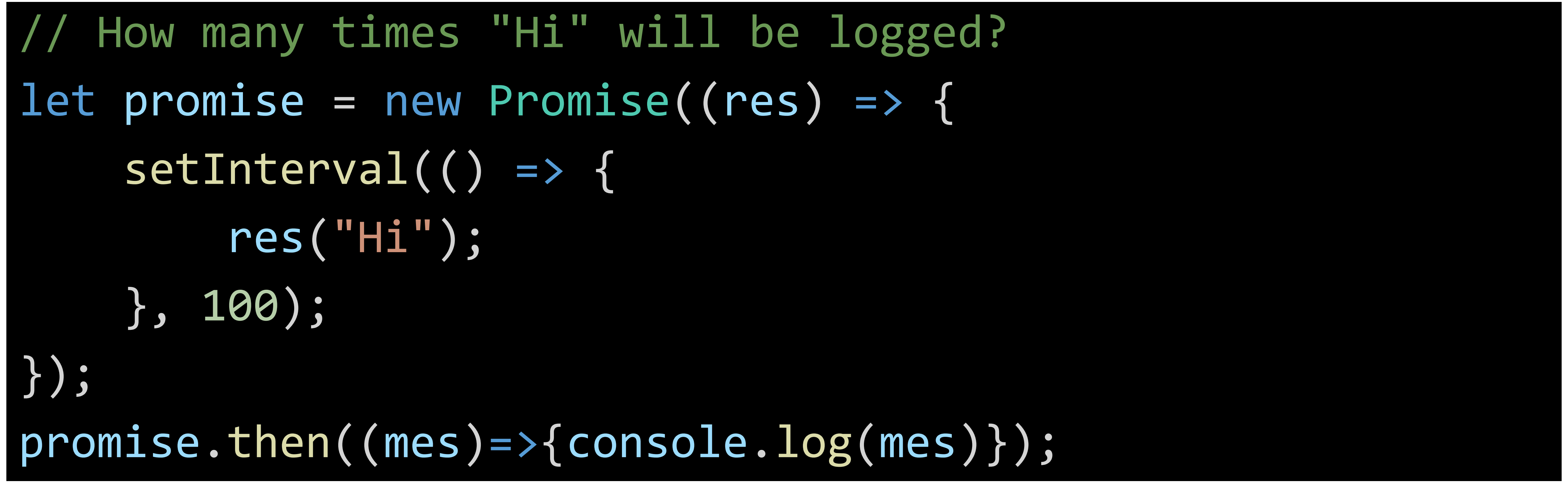


```
// How many times "Hi" will be logged?
let promise = new Promise((res) => {
    setInterval(() => {
        res("Hi");
    }, 100);
});
promise.then((mes)=>{console.log(mes)});
```

Tip: Remember we said promises are either pending or settled (settled: resolved or rejected).
Once they are settled, their state will not change
If the promise has already been resolved to a value to a rejection or to another promise,
then the res method does nothing .

A: Hi will be logged only once.

示例 4：'Hi' 将被记录多少次?



```
// How many times "Hi" will be logged?
let promise = new Promise((res) => {
    setInterval(() => {
        res("Hi");
    }, 100);
});
promise.then((mes)=>{console.log(mes)});
```

提示：请记住，我们之前提到过，Promise 的状态只有三种可能：待定（pending）或已确定（ settled）；而 “已确定” 又分为 “已解决（resolved）” 和 “已拒绝（rejected）” 。一旦 Promise 的状态变为已确定，其状态便不会再改变。如果该 Promise 已经被解决为某个值、被拒绝，或被解决为另一个 Promise，则调用 res 方法将不产生任何效果。

A：`console.log` 将仅执行一次。

Example 5:

- Promise.resolve() and Promise.reject() are shortcuts to manually create an already resolved or rejected promise respectively.
- In what order the numbers will be logged?

```
Promise.resolve().then(() => console.log(1));
console.log(2);
```

- Remember the note in the gray slide about functions passed to then(), which means resolve and reject, will never be called synchronously, even with an already-resolved promise
- Therefore the order is:

2

1

Examp 第 5 条：

- Promise.resolve() 和 Promise.reject() 分别是手动创建一个已解决或已拒绝的 Promise 的快捷方法。
- 数字将以何种顺序被打印出来?

```
Promise。resolve （）。then （（=> console。log （1））；

控制台.日志(2);
```

- 请回顾灰色幻灯片中关于传入 then() 的函数的说明，其中指出 resolve 和 reject 这两个函数永远不会被同步调用，即使 Promise 已处于已解决（resolved）状态。
- 因此，执行顺序为：

2

1

Chaining promises

If you remember the note 5 of the gray slide, The then() method returns a Promise. That’s why we can chain promises .

The then() method returns a Promise. It takes up to two arguments: callback functions for the success and failure cases of the Promise

function return (new Promise)).then(()=>{return (new Promise)}).then

promise.then()

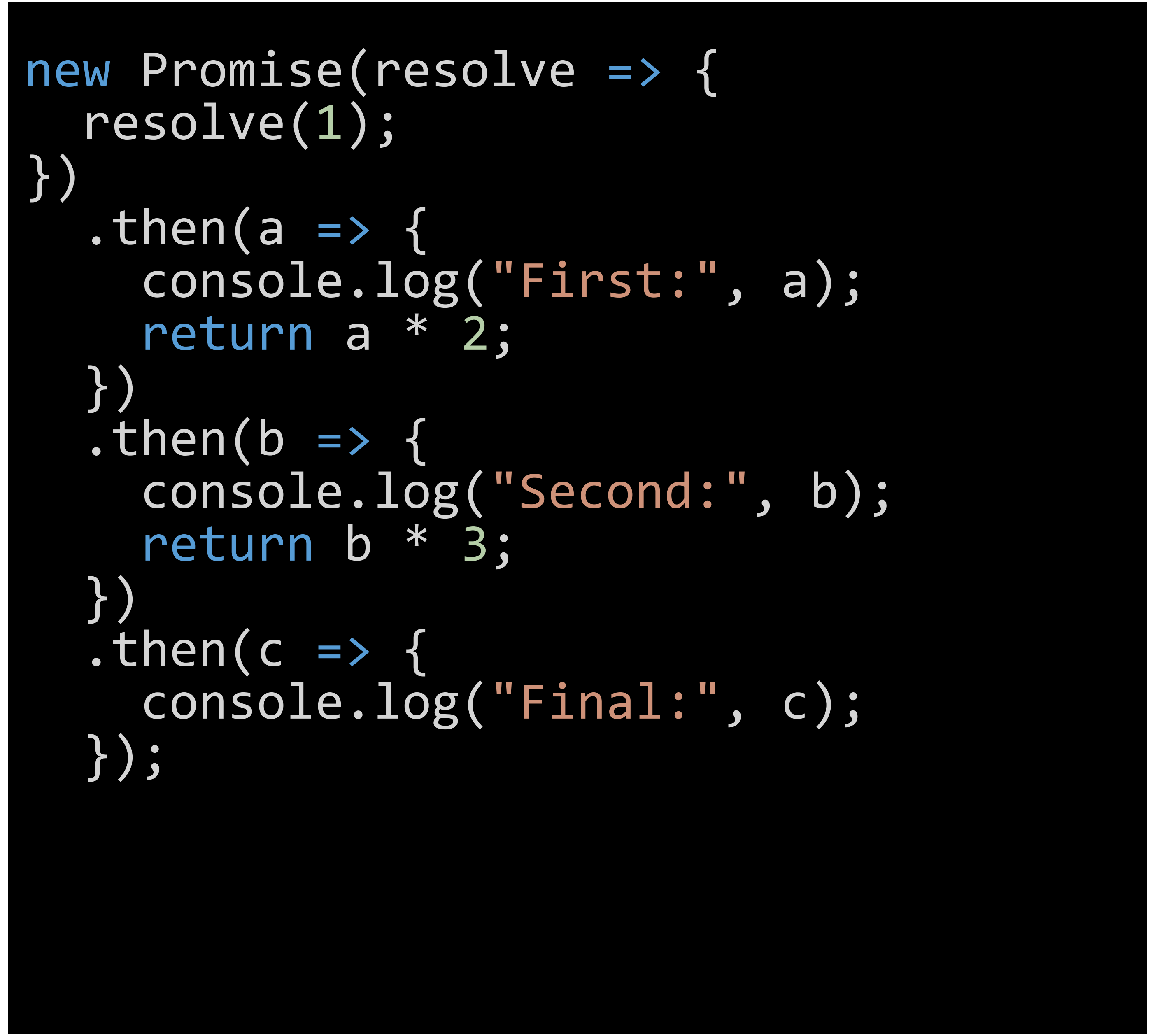
链式调用 Promise

如果您还记得灰色幻灯片中的第 5 条注释，那么 then() 方法会返回一个 Promise，这正是我们可以链式调用 Promise 的原因。

then() 方法返回一个 Promise，它最多接受两个参数：分别用于处理 Promise 成功和失败情况的回调函数。

函数返回（新建 Promise））.then（（）=>{返回（新建 Promise）}）.thenpromise.then()

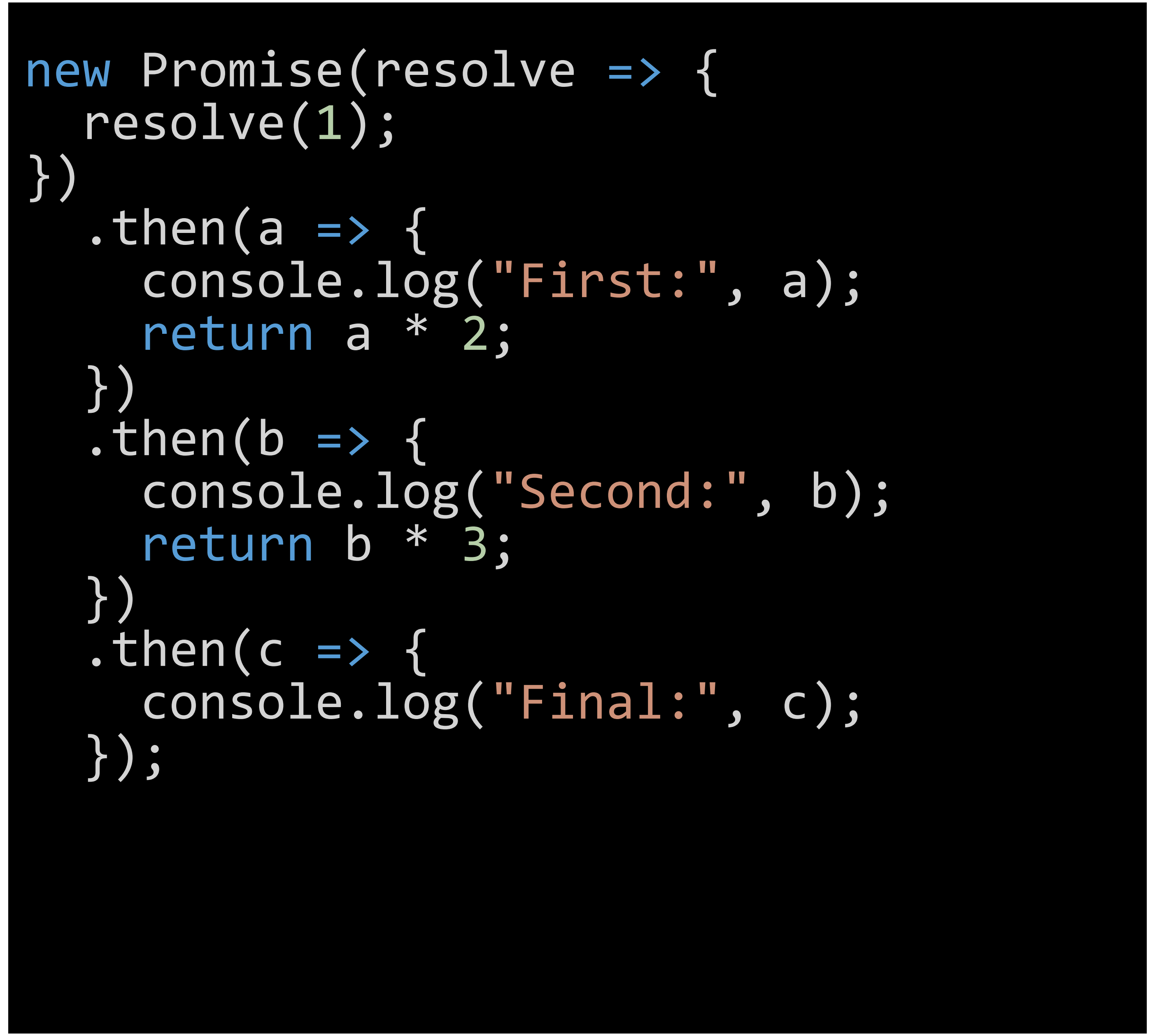
Example 6 (in-class activity)



```
new Promise(resolve => {
  resolve(1);
})
  .then(a => {
    console.log("First:", a);
    return a * 2;
  })
  .then(b => {
    console.log("Second:", b);
    return b * 3;
  })
  .then(c => {
    console.log("Final:", c);
  });
```

1. What will be logged ?

示例 6（课堂活动）

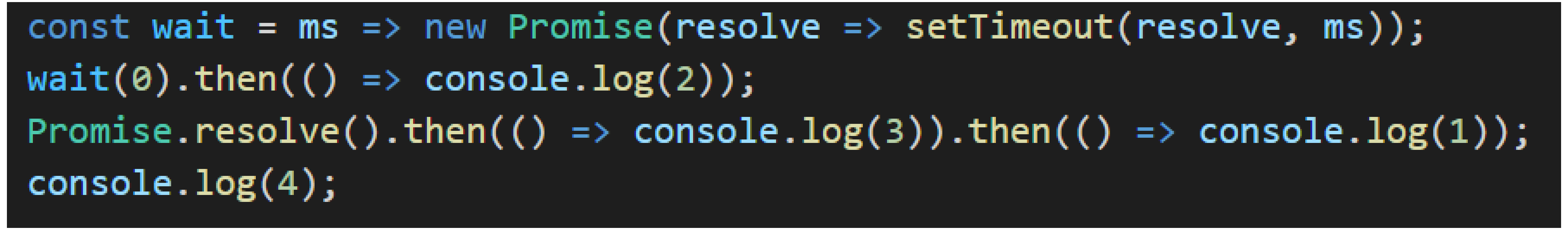


```
new Promise(resolve => {
  resolve(1);
})
  .then(a => {
    console.log("First:", a);
    return a * 2;
  })
  .then(b => {
    console.log("Second:", b);
    return b * 3;
  })
  .then(c => {
    console.log("Final:", c);
  });
```

1. 将会输出什么?

Example 7

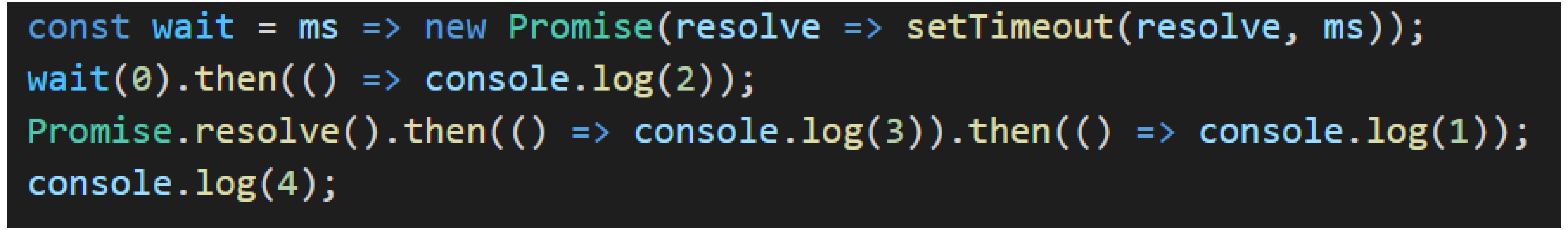
- In what order the numbers will be logged?



```
const wait = ms => new Promise(resolve => setTimeout(resolve, ms));
wait(0).then(() => console.log(2));
Promise.resolve().then(() => console.log(3)).then(() => console.log(1));
console.log(4);
```

示例 7

- 数字将以何种顺序被输出?



```
const wait = ms => new Promise(resolve => setTimeout(resolve, ms));
wait(0).then(() => console.log(2));
Promise.resolve().then(() => console.log(3)).then(() => console.log(1));
console.log(4);
```

Example 8 (in-class activity)

```
const p = new Promise((resolve, reject) => {
  const success = false;
  if (success) {
    resolve("Everything worked!");
  } else {
    reject("Something went wrong");
  }
});


p.then(result => {
  console.log("SUCCESS:", result);
})
  .catch(error => {
    console.log("ERROR:", error);
  });
```

1. What will be logged ?
2. What happens if we don’t handle reject and remove catch() ?

A: Unhandled reject error

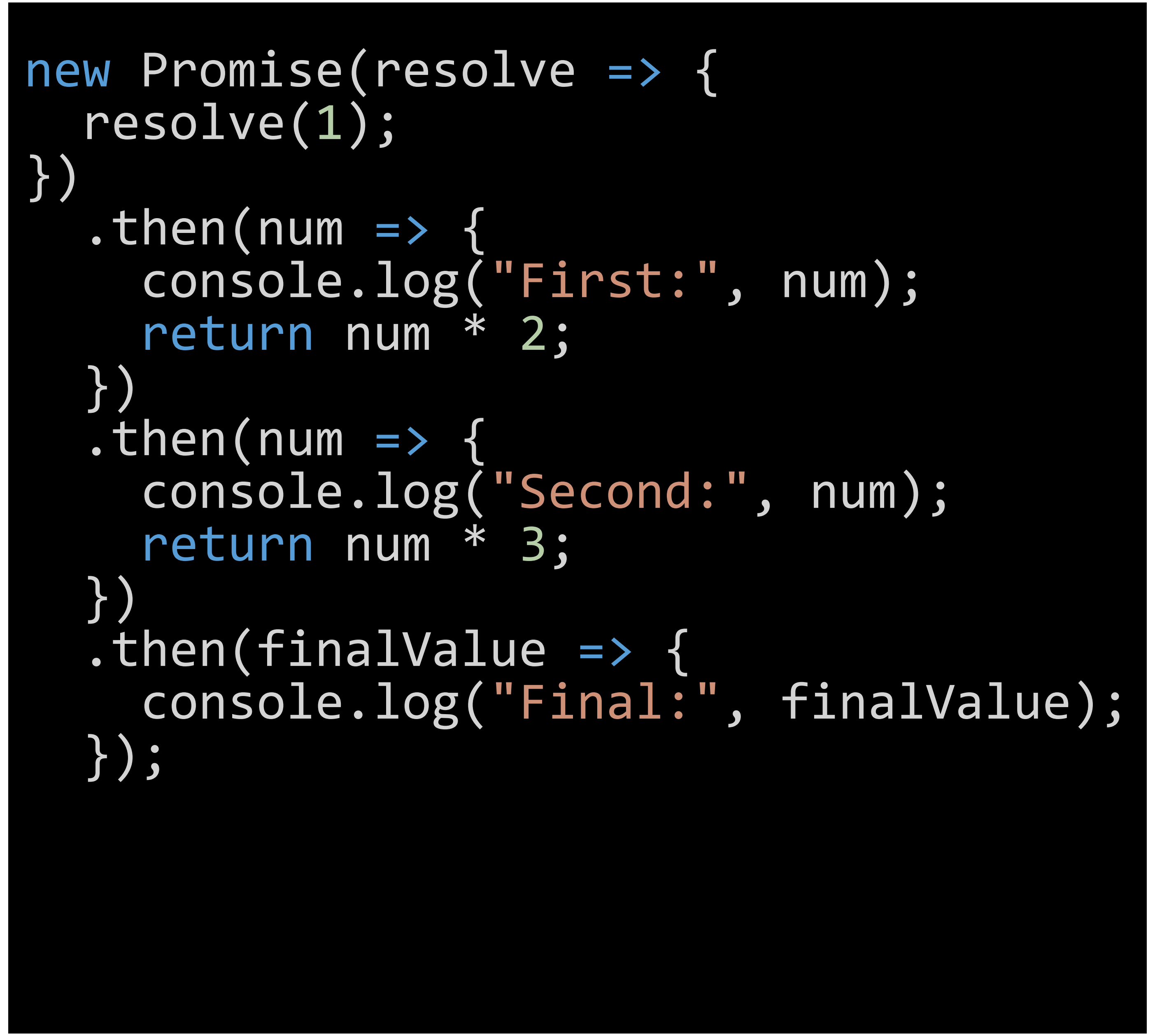
示例 8（课堂活动）

```
const p = new Promise((resolve, reject) => {
  const success = false;
  if (success) {
    resolve("Everything worked!");
  } else {
    reject("Something went wrong");
  }
});


p.then(result => {
  console.log("SUCCESS:", result);
})
  .catch(error => {
    console.log("ERROR:", error);
  });
```

1. What will be logged ?
2. What happens if we don’t handle reject and remove catch() ?

A: 未处理的拒绝错误

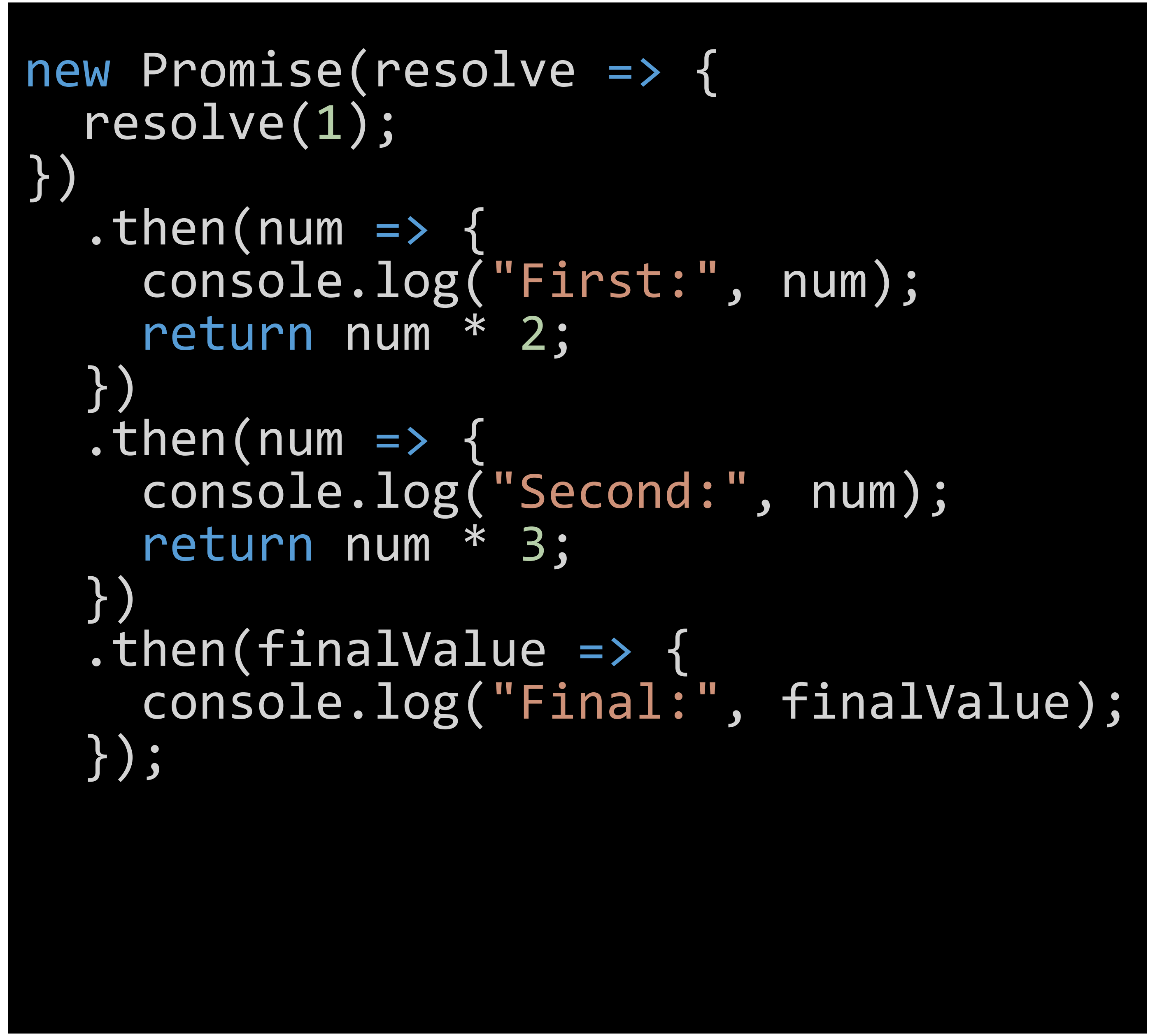
Example 9 (in-class activity)



```
new Promise(resolve => {
  resolve(1);
})
  .then(num => {
    console.log("First:", num);
    return num * 2;
  })
  .then(num => {
    console.log("Second:", num);
    return num * 3;
  })
  .then(finalValue => {
    console.log("Final:", finalValue);
  });
```

1. What will be logged ?

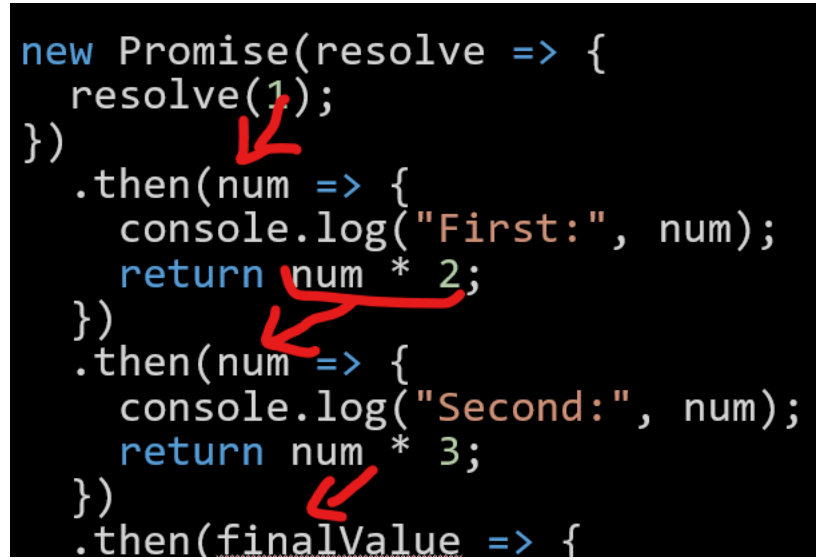
示例 9（课堂活动）



```
new Promise(resolve => {
  resolve(1);
})
  .then(num => {
    console.log("First:", num);
    return num * 2;
  })
  .then(num => {
    console.log("Second:", num);
    return num * 3;
  })
  .then(finalValue => {
    console.log("Final:", finalValue);
  });
```

1. 将输出什么?

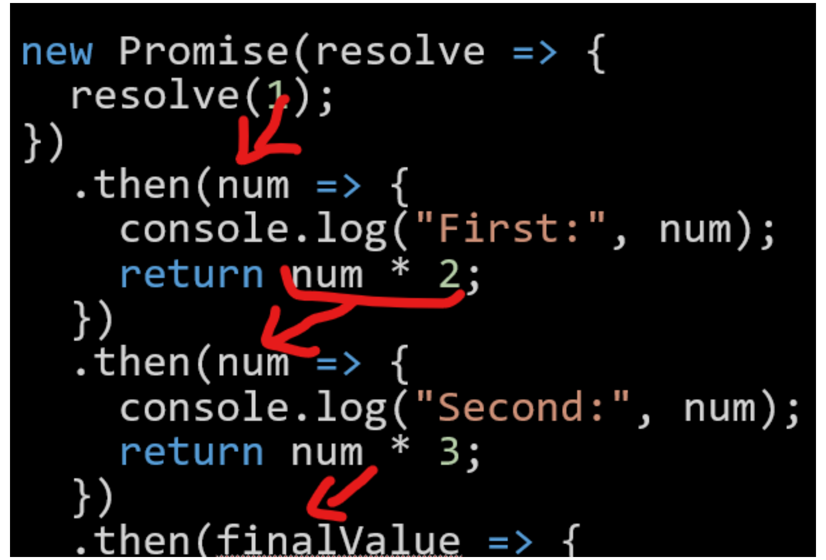
- Then() returns another promise and passes num*3 to the resolve of that new promise
- Q: why it does not pass the num*3 to the reject of the new promise?
- The value returned from .then() **is passed to the internal resolve function of the NEW promise created by .then()**.



```
new Promise(resolve => {
  resolve(1);
})
  .then(num => {
    console.log("First:", num);
    return num * 2;
  })
  .then(num => {
    console.log("Second:", num);
    return num * 3;
  })
  .then(finalValue => {
```

- The core rule (from the Promise spec, simplified)
- Returning a value is defined as success (passing to resolve of new promise)
  Throwing an error is defined as failure (passing to reject of new promise)

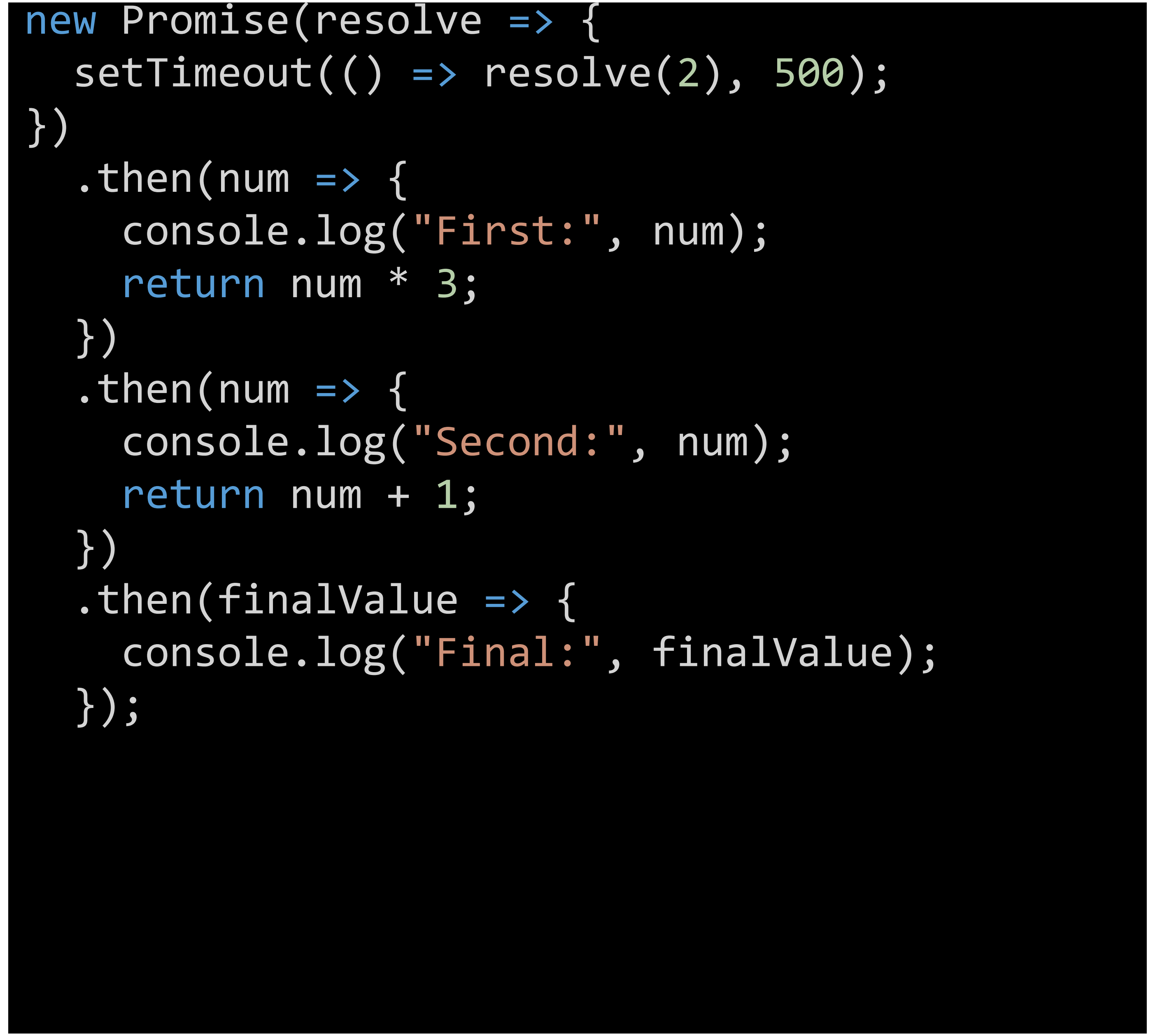
- Then() 方法返回另一个 Promise，并将 num*3 的值传递给该新 Promise 的 resolve 函数。
- 问：为什么它不将 num*3 的值传递给新 Promise 的 reject 函数?
- .then() 方法返回的值**会传入由 .then() 创建的新 Promise 的内部 resolve 函数**。



```
new Promise(resolve => {
  resolve(1);
})
  .then(num => {
    console.log("First:", num);
    return num * 2;
  })
  .then(num => {
    console.log("Second:", num);
    return num * 3;
  })
  .then(finalValue => {
```

- 核心规则（源自 Promise 规范，已简化）
- 返回一个值被定义为成功（即传递给新 Promise 的 resolve 函数）
  抛出一个错误被定义为失败（即传递给新 Promise 的 reject 函数）

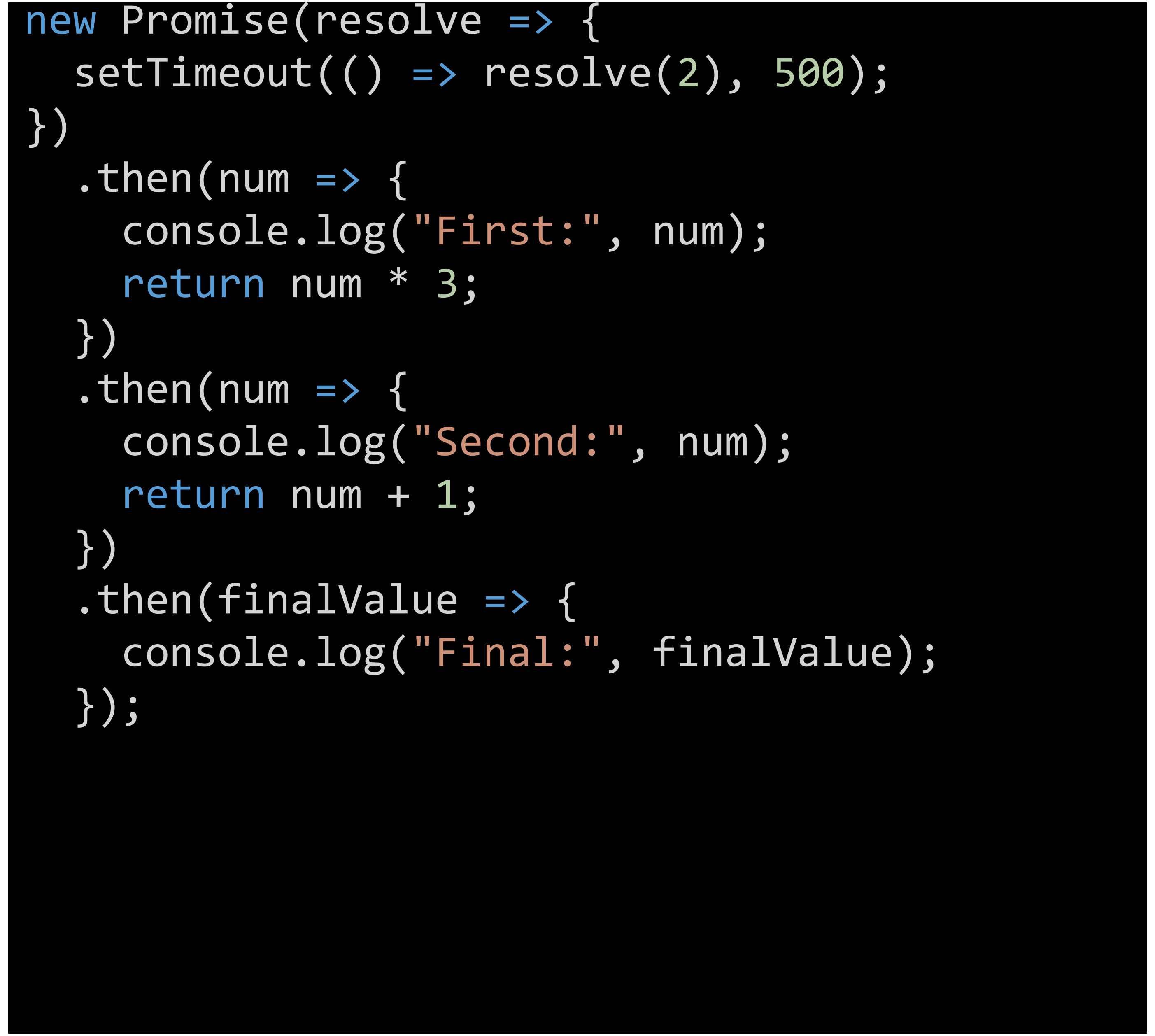
Example 10 (in-class activity)



```
new Promise(resolve => {
  setTimeout(() => resolve(2), 500);
})
  .then(num => {
    console.log("First:", num);
    return num * 3;
  })
  .then(num => {
    console.log("Second:", num);
    return num + 1;
  })
  .then(finalValue => {
    console.log("Final:", finalValue);
  });
```

1. What will be logged ?

Example 10 (in-class activity)



```
new Promise(resolve => {
  setTimeout(() => resolve(2), 500);
})
  .then(num => {
    console.log("First:", num);
    return num * 3;
  })
  .then(num => {
    console.log("Second:", num);
    return num + 1;
  })
  .then(finalValue => {
    console.log("Final:", finalValue);
  });
```

1. 将会输出什么?

- What is the difference between try catch error and promise resolve reject?
- Promises were designed to behave like **async try/catch blocks**.

```
// Synchronous code

try {
  const x = work();
  return x;    // success path
} catch (e) {
  throw e;     // failure path
}
```

```
//Promise .then() handler

.then(x => {
  return x;  // resolve
})
  .catch(e => {
    throw e;  // reject
  })
```

- try...catch 错误处理与 Promise 的 resolve/reject 之间有何区别?
- Promise 的设计初衷正是模拟 **异步的 try/catch 语句块**。

```
// Synchronous code

try {
  const x = work();
  return x;    // success path
} catch (e) {
  throw e;     // failure path
}
```

```
//Promise .then() handler

.then(x => {
  return x;  // resolve
})
  .catch(e => {
    throw e;  // reject
  })
```

Example 11 (in-class activity)

```
f1 = () =>{console.log(1)};
f2 = () =>{console.log(2)};
f3 = () =>{console.log(3)};

// async() does things asynchronously for us
function async(callMeBack) {
    setTimeout(() => {
        callMeBack();
    }, Math.floor(Math.random() * 1000)
    );
}
//we want f1,f2 and f3 execute in f1,f2, f3
// asynchronously How can we guarantee ?
// would below work? if not what to do?
async(f1);
async(f2);
async(f3);
```

we want f1,f2 and f3 execute in f1,f2, f3 asynchronously How can we guarantee ?
 would below work? if not what to do?

示例 11（课堂活动）

```
f1 = () =>{console.log(1)};
f2 = () =>{console.log(2)};
f3 = () =>{console.log(3)};

// async() does things asynchronously for us
function async(callMeBack) {
    setTimeout(() => {
        callMeBack();
    }, Math.floor(Math.random() * 1000)
    );
}
//we want f1,f2 and f3 execute in f1,f2, f3
// asynchronously How can we guarantee ?
// would below work? if not what to do?
async(f1);
async(f2);
async(f3);
```

我们希望 f1、f2 和 f3 在 f1、f2、f3 中异步执行。如何确保这一点？以下写法是否可行？如果不可行，又该如何处理？

Lets observe
how

promise status

5

changes

In JavaScript, **"promise states"** and **"promise status"** refer to the same concept: the internal, unchangeable condition of a `Promise` object at any given time. There are three possible states (or statuses): **`pending`**, **`fulfilled`**, and **`rejected`**.

让我们观察一下

Promise 状态

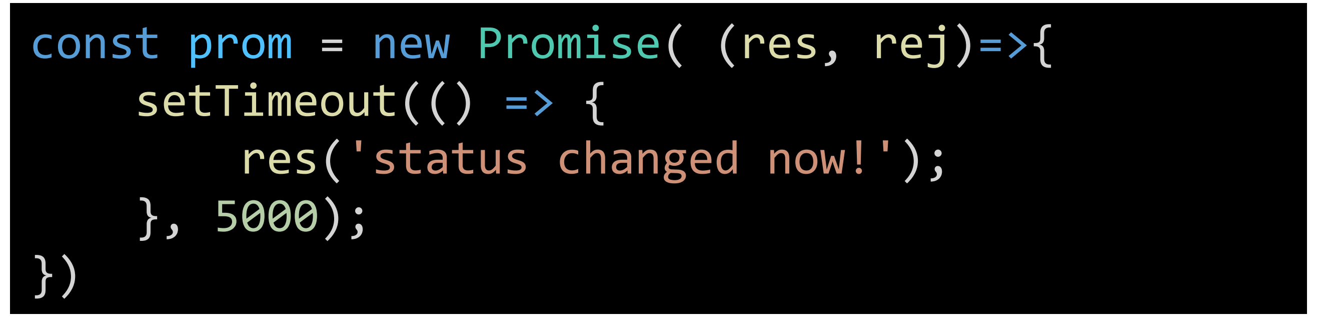
5

变更

在 JavaScript 中，**“Promise 状态”**和“**Promise 状况**”指代同一概念：即 Promise 对象在任意时刻所处的内部、不可变的状态。共有三种可能的状态（或状况）：**待定**（**pending**）、**已兑现**（**fulfilled**）和**已拒绝**（**rejected**）。

Try this code example 12

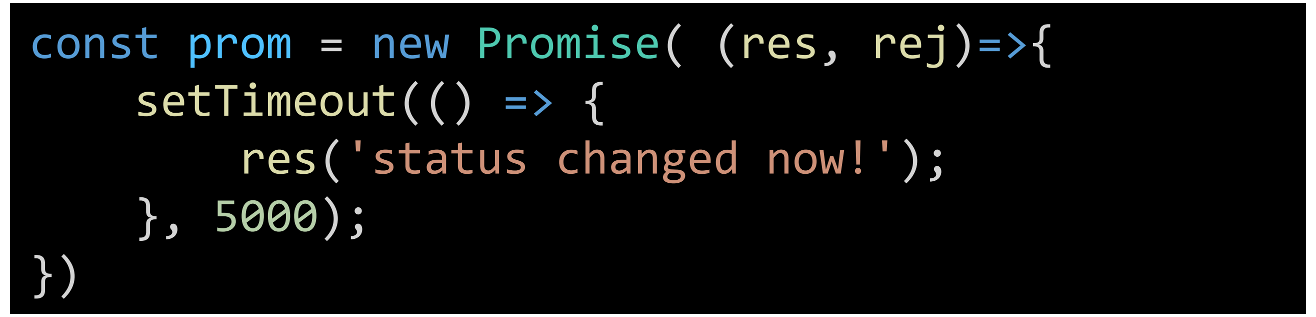
and observe the changes in the property values of the instantiated promse object over time



```
const prom = new Promise( (res, rej)=>{
    setTimeout(() => {
        res('status changed now!');
    }, 5000);
})
```

尝试此代码示例 12

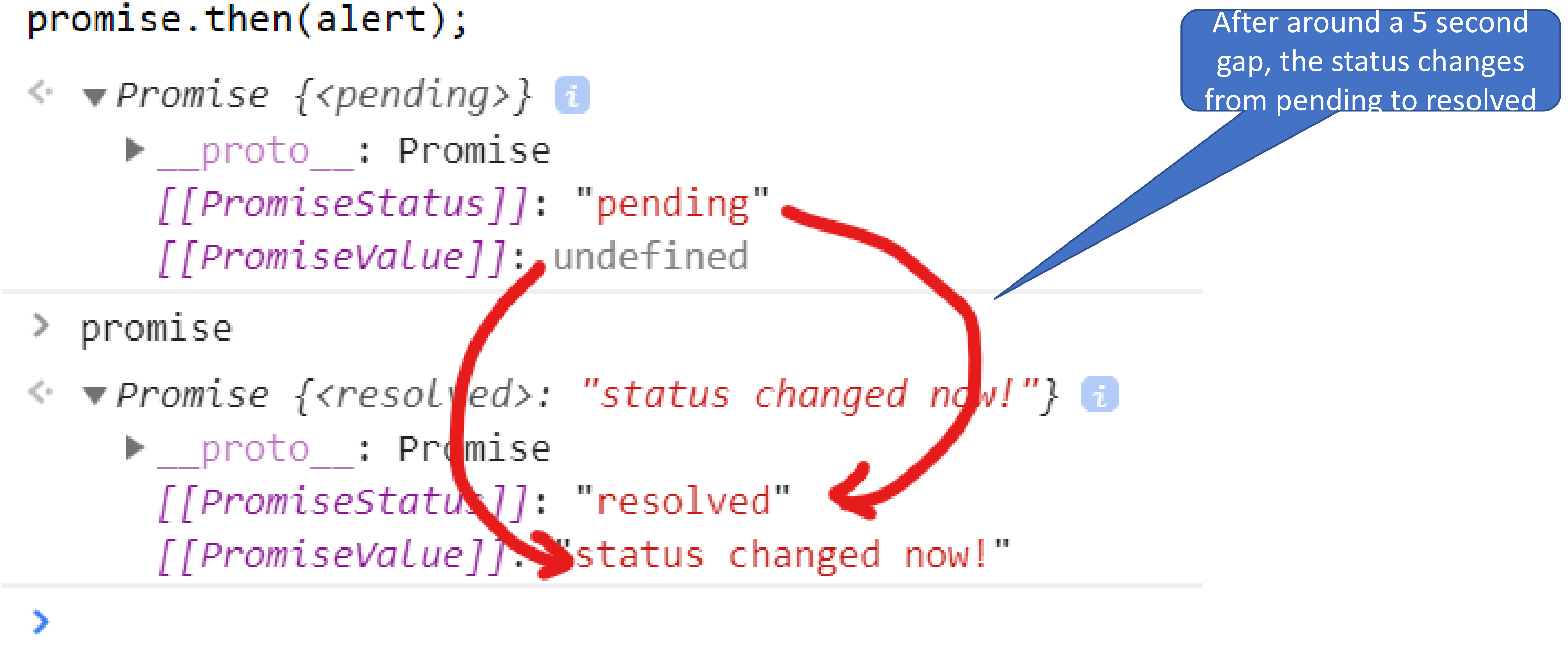
并观察实例化后的 Promise 对象各属性值随时间发生的变化



```
const prom = new Promise( (res, rej)=>{
    setTimeout(() => {
        res('status changed now!');
    }, 5000);
})
```

Example: observe how status and value change after 5 sec

```
let promise = new Promise(function (resolve, reject) {
    setTimeout(() => resolve("status changed now!"), 5000);
});
promise.then(alert);
```

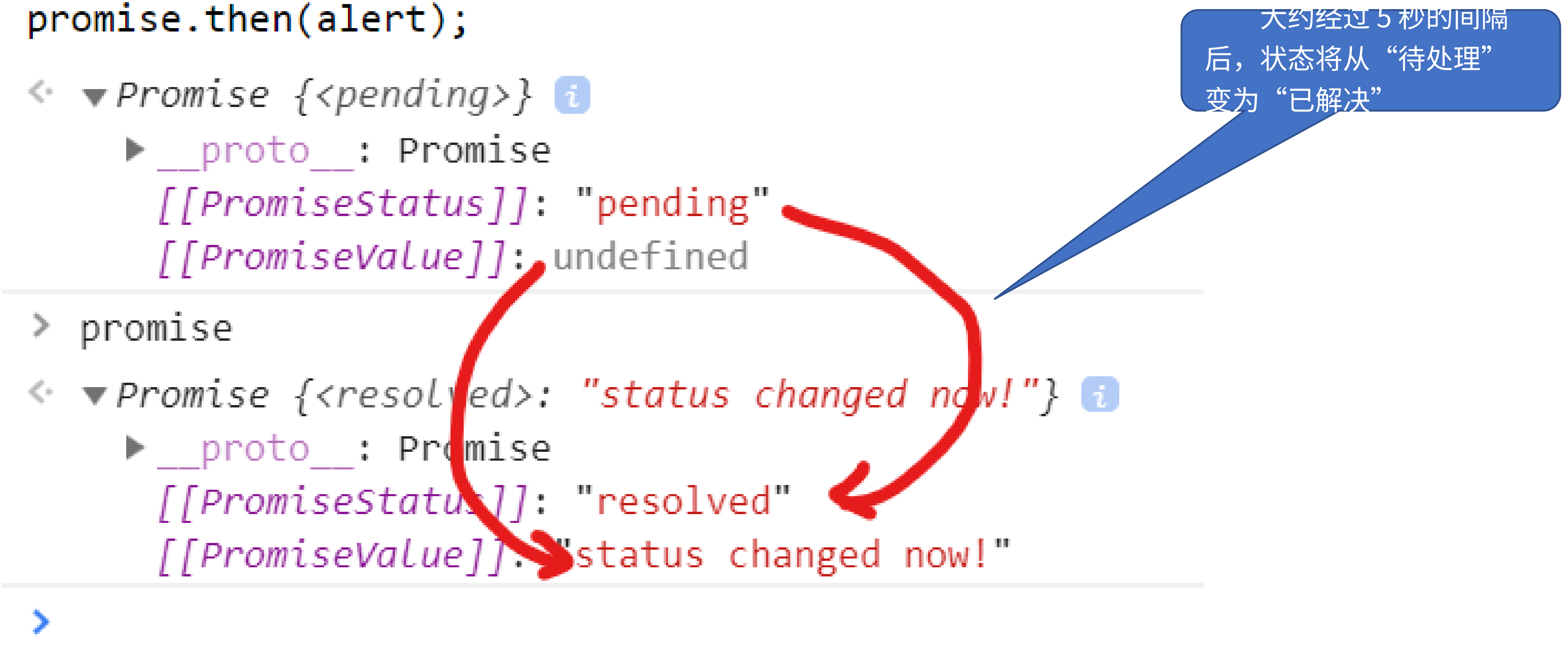


```
< ▼Promise {<pending>}
    ▶ __proto__: Promise
      [[PromiseStatus]]: "pending"
      [[PromiseValue]]: undefined
> promise
< ▼Promise {<resolved>: "status changed now!"}
    ▶ __proto__: Promise
      [[PromiseStatus]]: "resolved"
      [[PromiseValue]]: "status changed now!"
>
```

After around a 5 second gap, the status changes from pending to resolved

示例：观察状态和值在 5 秒后如何变化

```
let promise = new Promise(function (resolve, reject) {
    setTimeout(() => resolve("status changed now!"), 5000);
});
promise.then(alert);
```



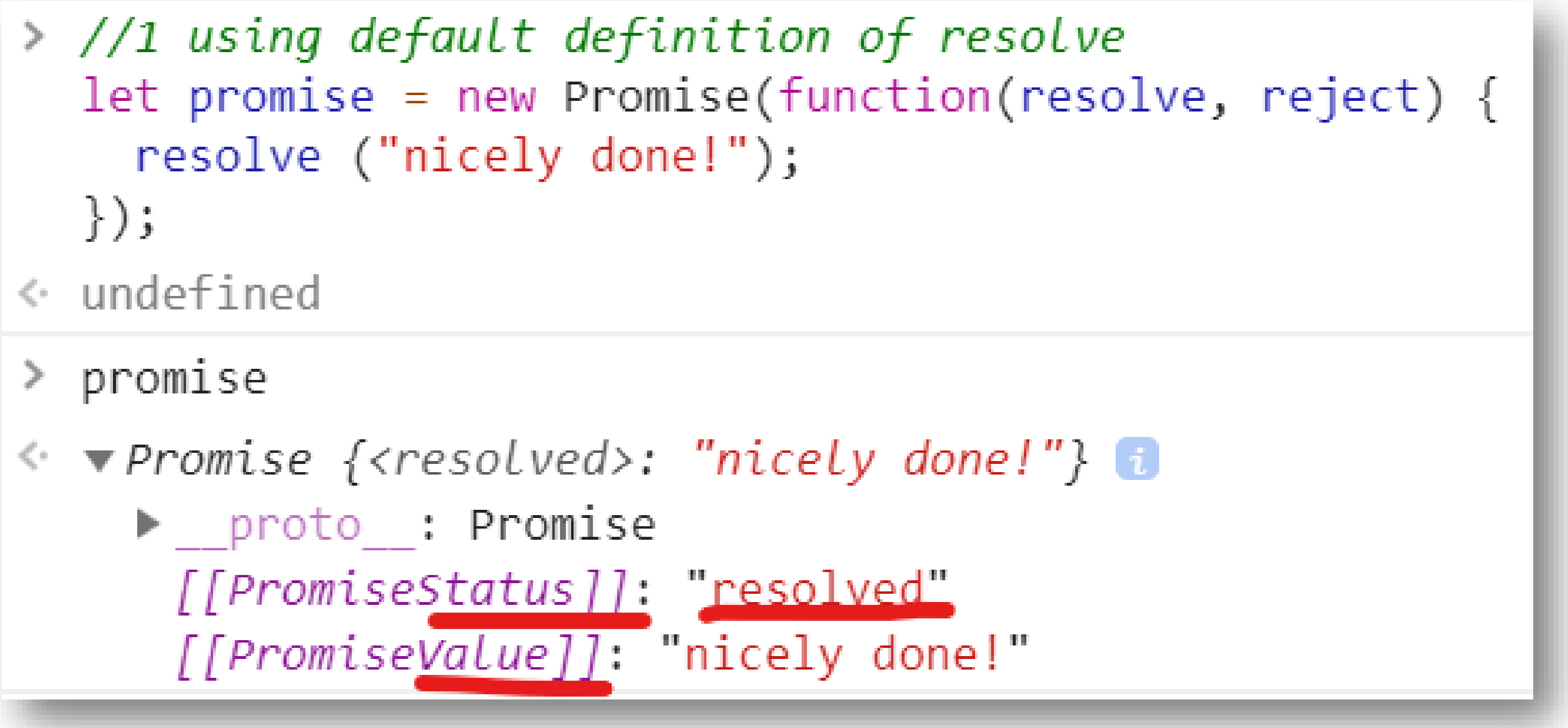
```
< ▼Promise {<pending>}
    ▶ __proto__: Promise
      [[PromiseStatus]]: "pending"
      [[PromiseValue]]: undefined
> promise
< ▼Promise {<resolved>: "status changed now!"}
    ▶ __proto__: Promise
      [[PromiseStatus]]: "resolved"
      [[PromiseValue]]: "status changed now!"
>
```

大约经过 5 秒的间隔后，状态将从 “待处理” 变为 “已解决”

Example 9: using default definition of resolve method

- what will be the status and value of the object promise after execution of this code?

```
let promise = new Promise(function(resolve, reject) {
  resolve ("nicely done!");
});
```

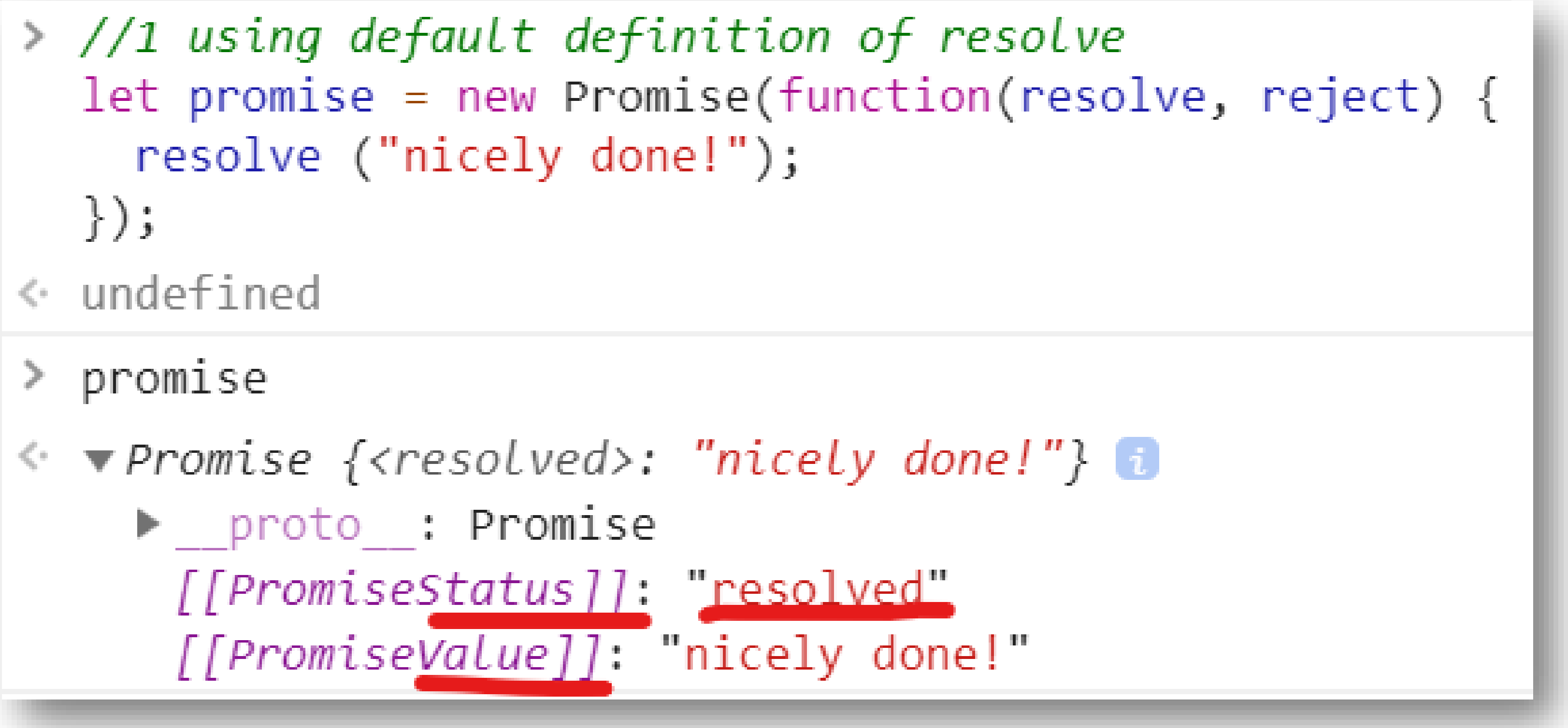


```
> //1 using default definition of resolve
  let promise = new Promise(function(resolve, reject) {
    resolve ("nicely done!");
  });
<· undefined
> promise
<· ▼Promise {<resolved>: "nicely done!"} i
    ▶__proto__: Promise
     [[PromiseStatus]]: "resolved"
     [[PromiseValue]]: "nicely done!"
```

示例 9：使用 resolve 方法的默认定义

- 执行此代码后，对象 promise 的状态和值将分别是什么？代码?

```
let promise = new Promise(function(resolve, reject) {
  resolve ("nicely done!");
});
```

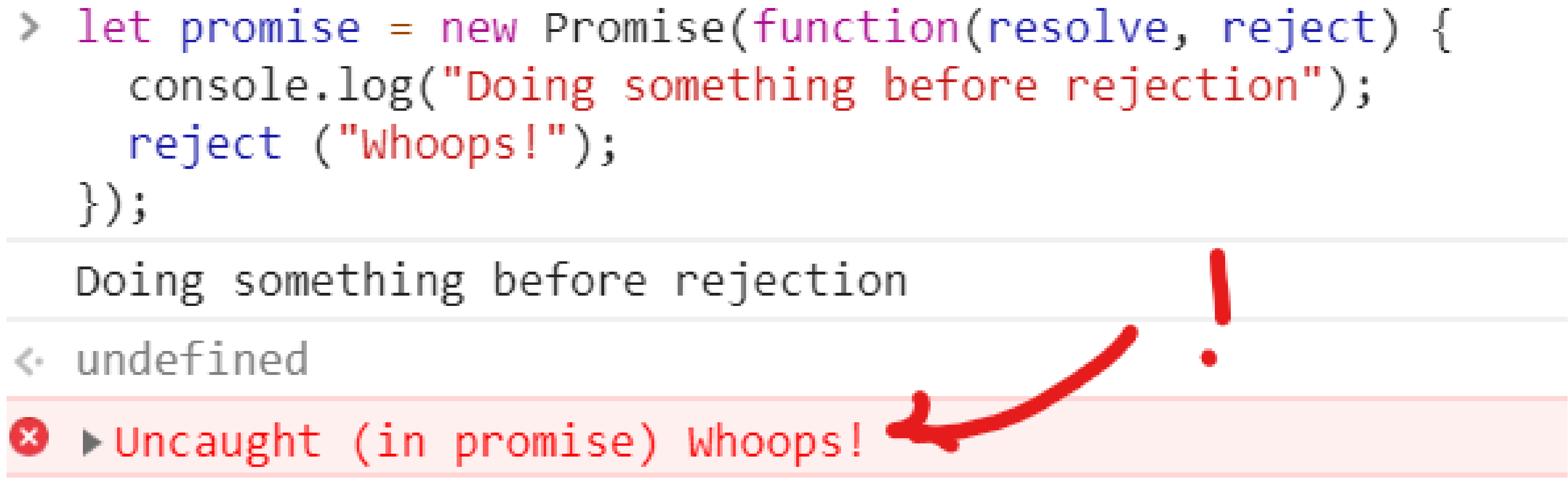


```
> //1 using default definition of resolve
  let promise = new Promise(function(resolve, reject) {
    resolve ("nicely done!");
  });
<· undefined
> promise
<· ▼Promise {<resolved>: "nicely done!"} i
    ▶__proto__: Promise
     [[PromiseStatus]]: "resolved"
     [[PromiseValue]]: "nicely done!"
```

Example 10: using default definition of reject method

- what will be the status and value of the object promise after execution of this code?

```
let promise = new Promise(function(resolve, reject) {
    console.log("Doing something before rejection");
    reject ("Whoops!");
});
```



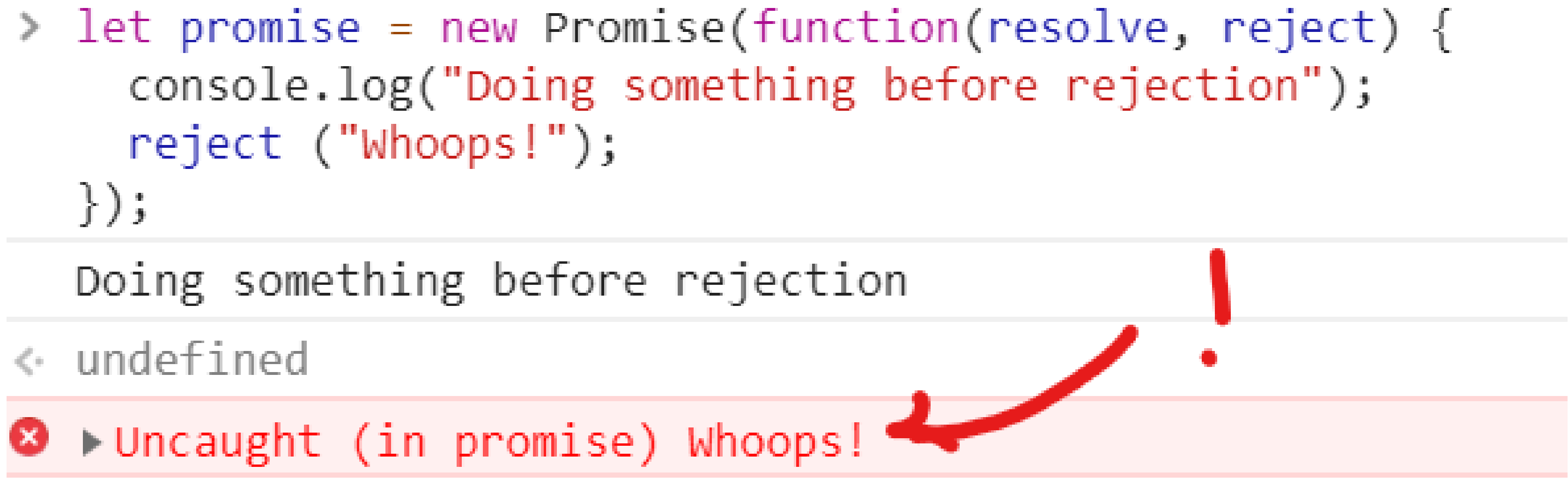
```
> let promise = new Promise(function(resolve, reject) {
    console.log("Doing something before rejection");
    reject ("Whoops!");
  });
  Doing something before rejection
<· undefined
ⓧ ▶Uncaught (in promise) Whoops!
```

Why?
Because we did not catch the error (we did not define the reject method)!
The js engine throws the error using default definition of reject! In fact rejection naturally means something went wrong

示例 10：使用 reject 方法的默认定义

- 执行此代码后，该 Promise 对象的状态和值将分别是什么？代码？

```
let promise = new Promise(function(resolve, reject) {
    console.log("Doing something before rejection");
    reject ("Whoops!");
});
```



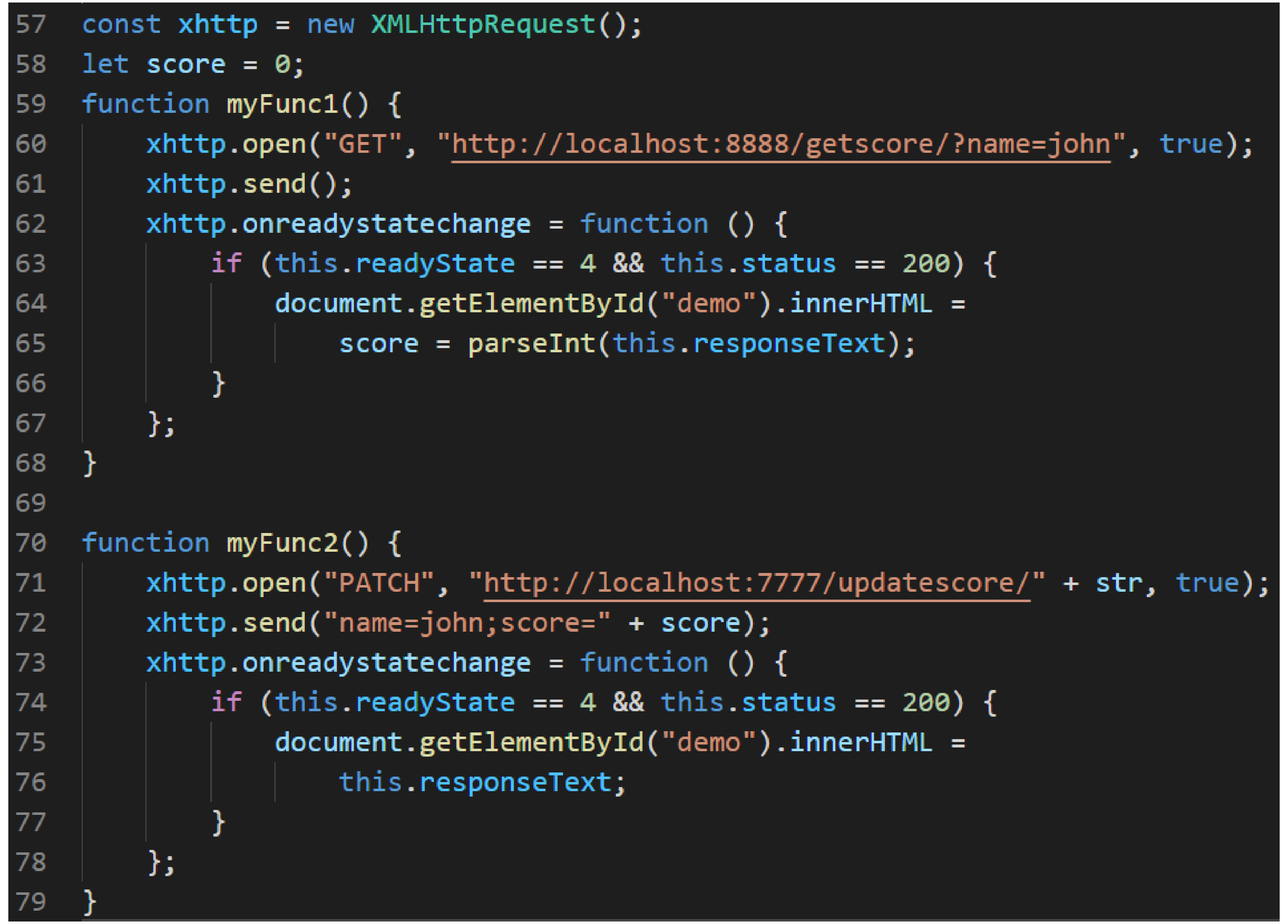
```
> let promise = new Promise(function(resolve, reject) {
    console.log("Doing something before rejection");
    reject ("Whoops!");
  });
  Doing something before rejection
<· undefined
ⓧ ▶Uncaught (in promise) Whoops!
```

Why?
因为我们未捕获该错误（即未定义 reject 方法）！ jreject 方法）！
JavaScript 引擎使用默认的 reject 定义抛出该错误！事实上
拒绝（rejection）自然意味着出现了某些问题

Q: How to execute two successive AJAX calls (second one only after first one is guaranteed finished)?

- Suppose we wish to
- make an Ajax call to server A to get something
- And then
- making another AJAX call to server B based on what you got from server A.
- **Q:** How can you **guarantee** the **right order**?
- **A:** calling myFunc2 inside myFunc1, line 65

Have you heard about callback hell?



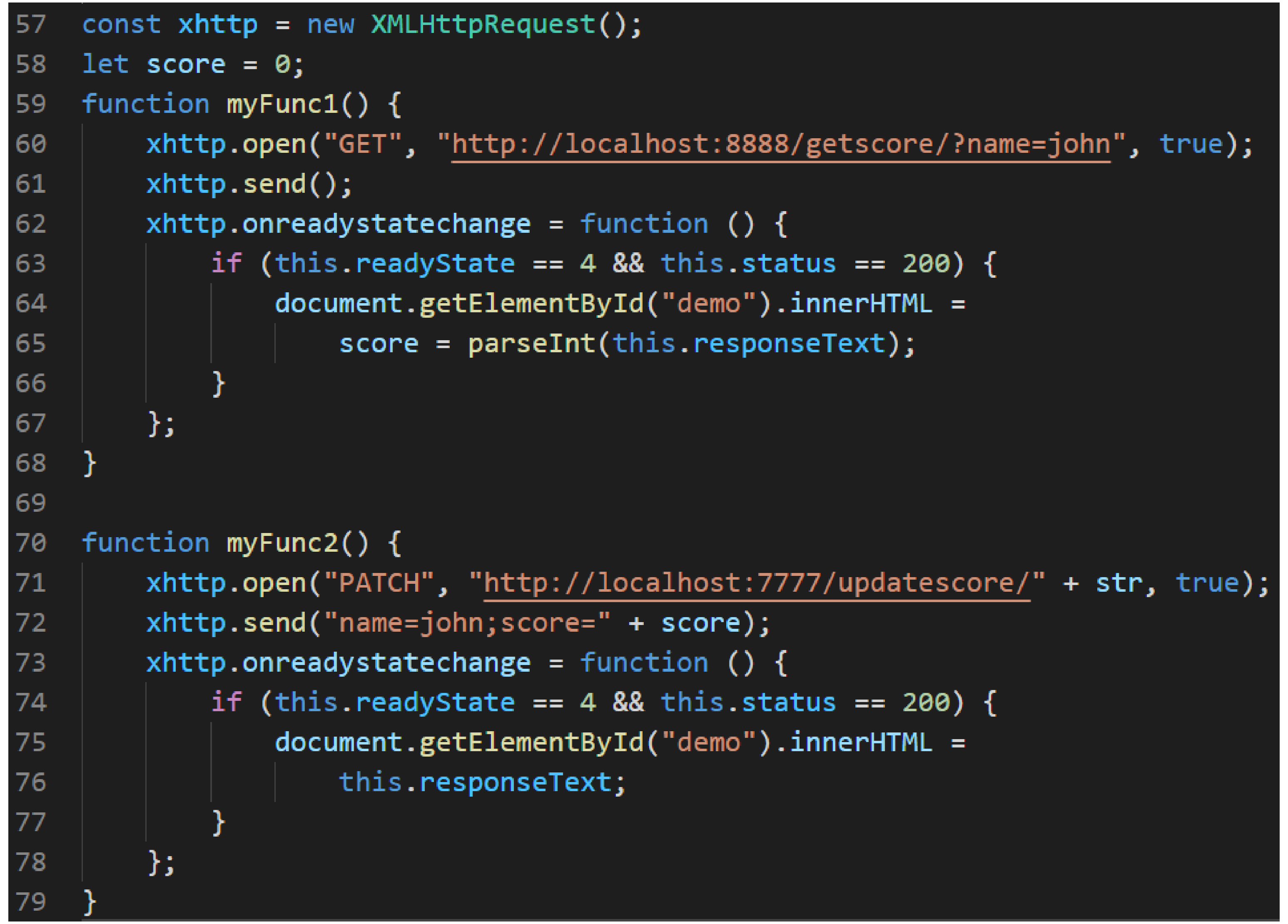
```
57  const xhttp = new XMLHttpRequest();
58  let score = 0;
59  function myFunc1() {
60      xhttp.open("GET", "http://localhost:8888/getscore/?name=john", true);
61      xhttp.send();
62      xhttp.onreadystatechange = function () {
63          if (this.readyState == 4 && this.status == 200) {
64              document.getElementById("demo").innerHTML =
65                  score = parseInt(this.responseText);
66          }
67      };
68  }
69
70  function myFunc2() {
71      xhttp.open("PATCH", "http://localhost:7777/updatescore/" + str, true);
72      xhttp.send("name=john;score=" + score);
73      xhttp.onreadystatechange = function () {
74          if (this.readyState == 4 && this.status == 200) {
75              document.getElementById("demo").innerHTML =
76                  this.responseText;
77          }
78      };
79  }
```

问：如何执行两个连续的 AJAX 请求（第二个请求仅在第一个请求确保完成之后才执行）？

第一个请求确保完成之后）？

- 假设我们希望
- 向服务器 A 发起一个 Ajax 请求，以获取某些数据
- 然后 **再**
- 根据从服务器 A 获取的数据，向服务器 B 发起另一个 AJAX 请求。
- **Q:** 如何**确保** 正确的**执行顺序**?
- **A：** 在 myFunc1 中调用 myFunc2，位于第 65

你听说过“回调地狱”吗?



```
57  const xhttp = new XMLHttpRequest();
58  let score = 0;
59  function myFunc1() {
60      xhttp.open("GET", "http://localhost:8888/getscore/?name=john", true);
61      xhttp.send();
62      xhttp.onreadystatechange = function () {
63          if (this.readyState == 4 && this.status == 200) {
64              document.getElementById("demo").innerHTML =
65                  score = parseInt(this.responseText);
66          }
67      };
68  }
69
70  function myFunc2() {
71      xhttp.open("PATCH", "http://localhost:7777/updatescore/" + str, true);
72      xhttp.send("name=john;score=" + score);
73      xhttp.onreadystatechange = function () {
74          if (this.readyState == 4 && this.status == 200) {
75              document.getElementById("demo").innerHTML =
76                  this.responseText;
77          }
78      };
79  }
```

.finally 6

3 main methods of the Promise object are

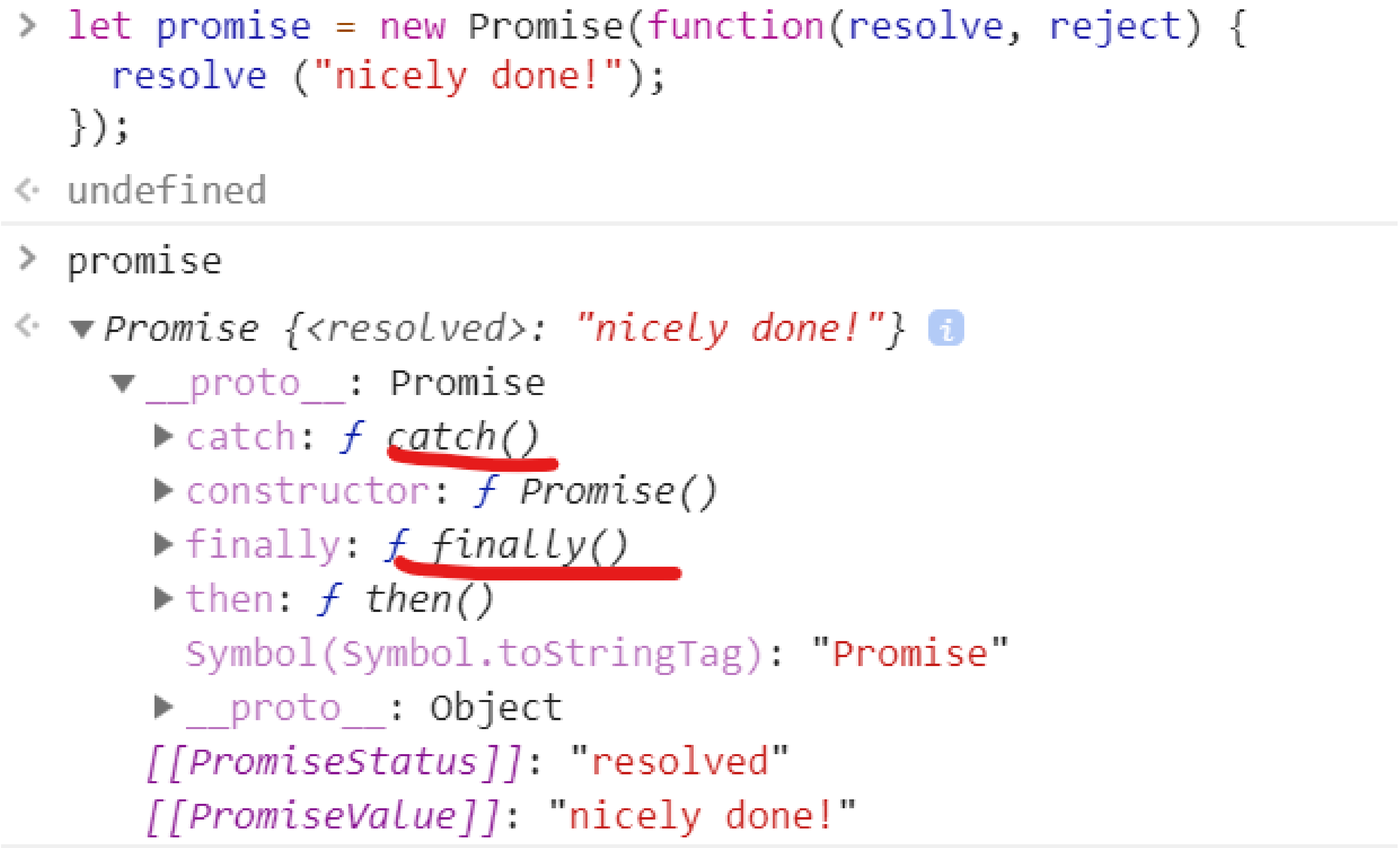
.then .catch .finally

..finally 6

Promise 对象的三个主要方法是.then、 .
catch 和.finally

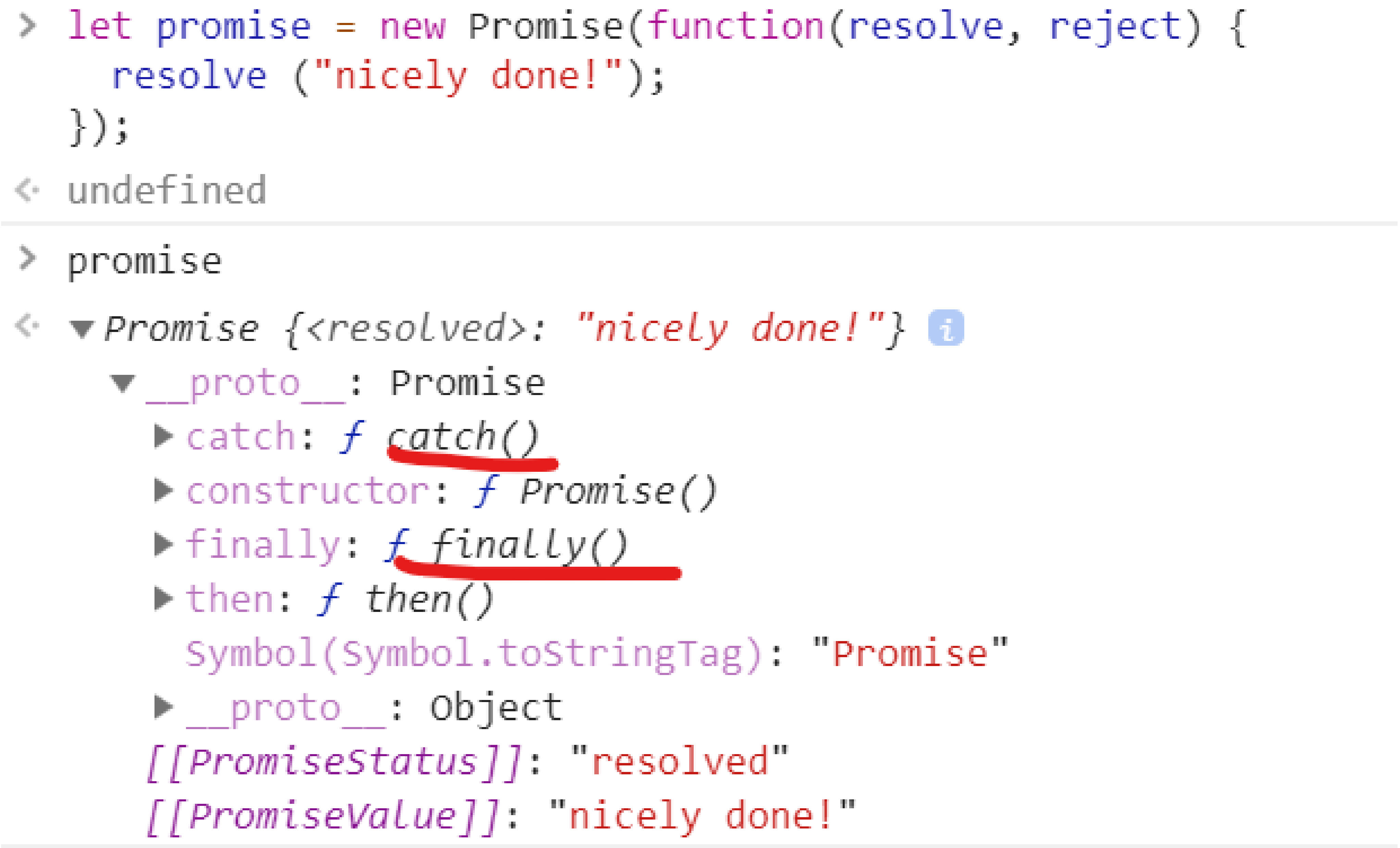
3 main methods of the Promise object are

What are these methods?



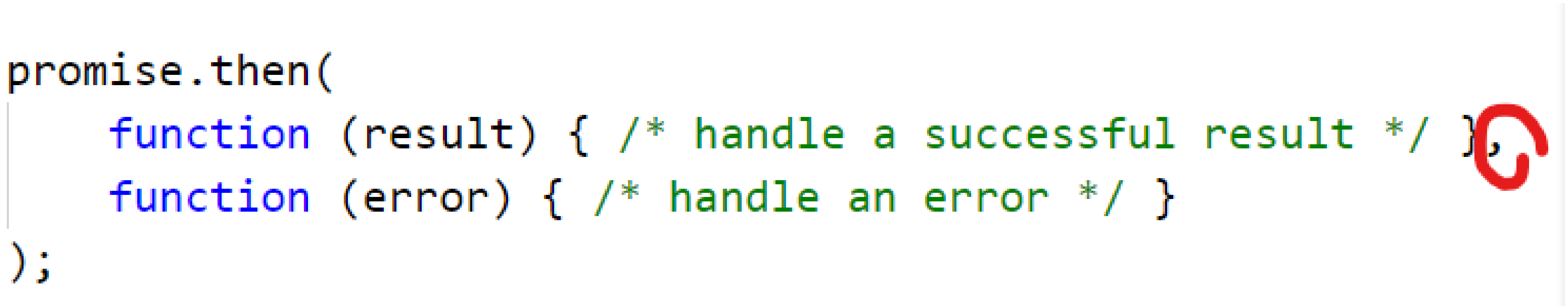
```
> let promise = new Promise(function(resolve, reject) {
    resolve ("nicely done!");
  });
< undefined
> promise
< ▼Promise {<resolved>: "nicely done!"} ℹ
    ▼__proto__: Promise
      ▶catch: ƒ catch()
      ▶constructor: ƒ Promise()
      ▶finally: ƒ finally()
      ▶then: ƒ then()
       Symbol(Symbol.toStringTag): "Promise"
      ▶__proto__: Object
     [[PromiseStatus]]: "resolved"
     [[PromiseValue]]: "nicely done!"
```

这些方法是什么?



```
> let promise = new Promise(function(resolve, reject) {
    resolve ("nicely done!");
  });
< undefined
> promise
< ▼Promise {<resolved>: "nicely done!"} ℹ
    ▼__proto__: Promise
      ▶catch: ƒ catch()
      ▶constructor: ƒ Promise()
      ▶finally: ƒ finally()
      ▶then: ƒ then()
       Symbol(Symbol.toStringTag): "Promise"
      ▶__proto__: Object
     [[PromiseStatus]]: "resolved"
     [[PromiseValue]]: "nicely done!"
```

.then method (we already know)

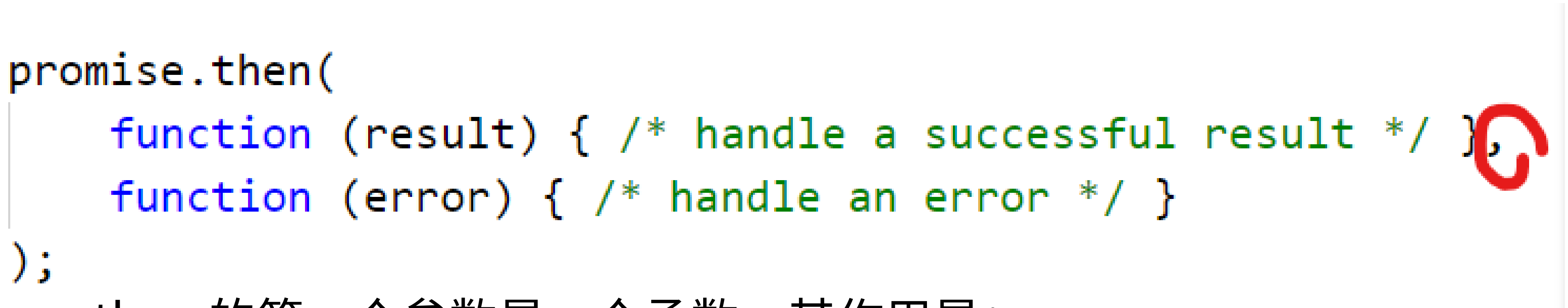


```
promise.then(
    function (result) { /* handle a successful result */ },
    function (error) { /* handle an error */ }
);
```

- The first argument of .then is a function that:
- runs when the Promise is resolved, and
- receives the result.

- The second argument of .then is a function that:
- runs when the Promise is rejected, and
- receives the error.

.then 方法（我们已经了解）

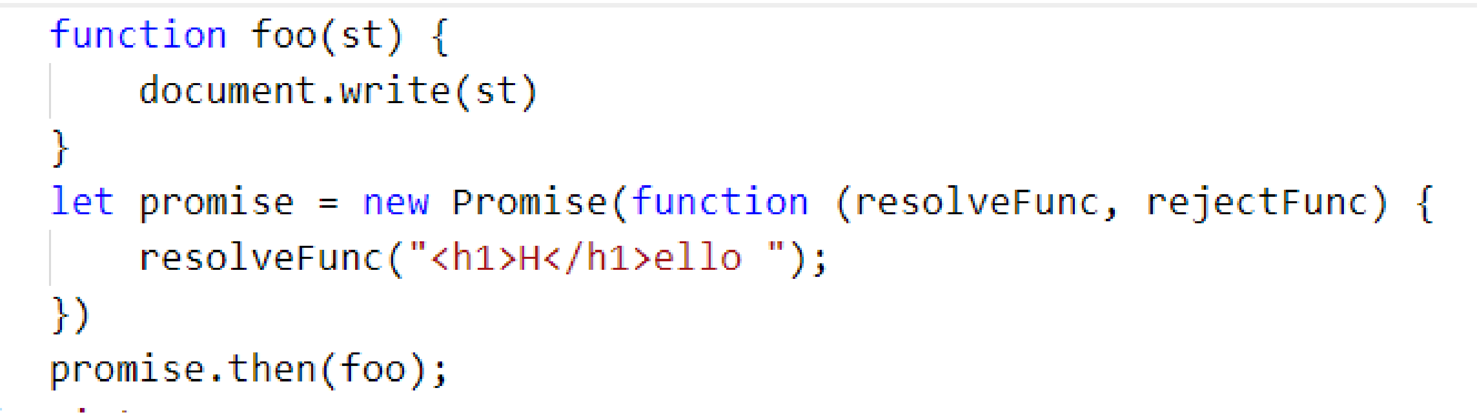


```
promise.then(
    function (result) { /* handle a successful result */ },
    function (error) { /* handle an error */ }
);
```

- .then 的第一个参数是一个函数，其作用是：
- 在 Promise 被成功解决（resolved）时执行；
- 接收该 Promise 的结果。

- .then 的第二个参数是一个函数，其作用是：
- 在 Promise 被拒绝（rejected）时执行；
- 收到该错误。

Example: What’s the output of the code below?

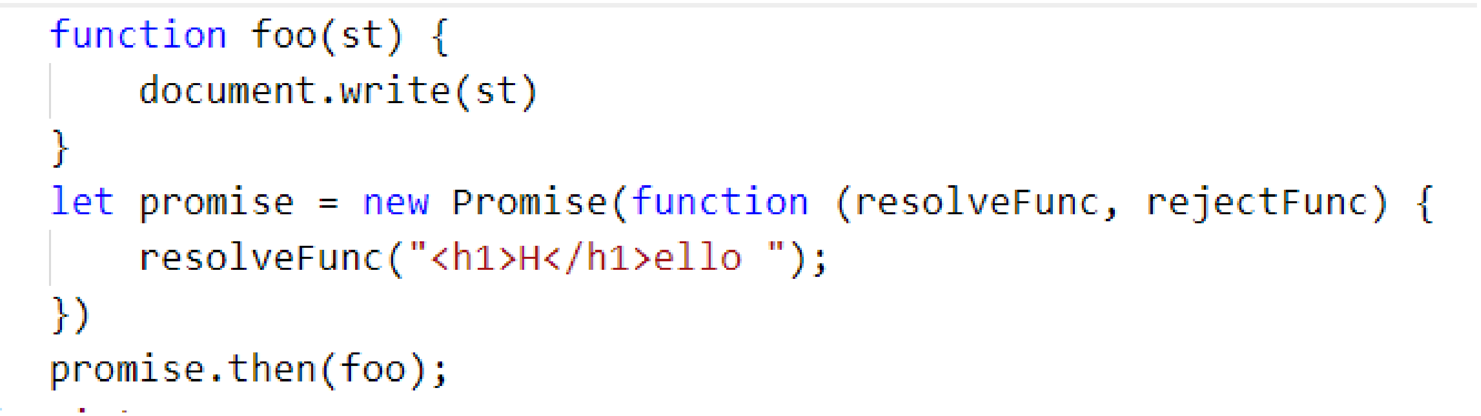


```
function foo(st) {
    document.write(st)
}
let promise = new Promise(function (resolveFunc, rejectFunc) {
    resolveFunc("<h1>H</h1>ello ");
})
promise.then(foo);
```

A: When we pass only one function to .then method, that function replaces the default definition of resolve method

So the code above will display 1 on the browser window

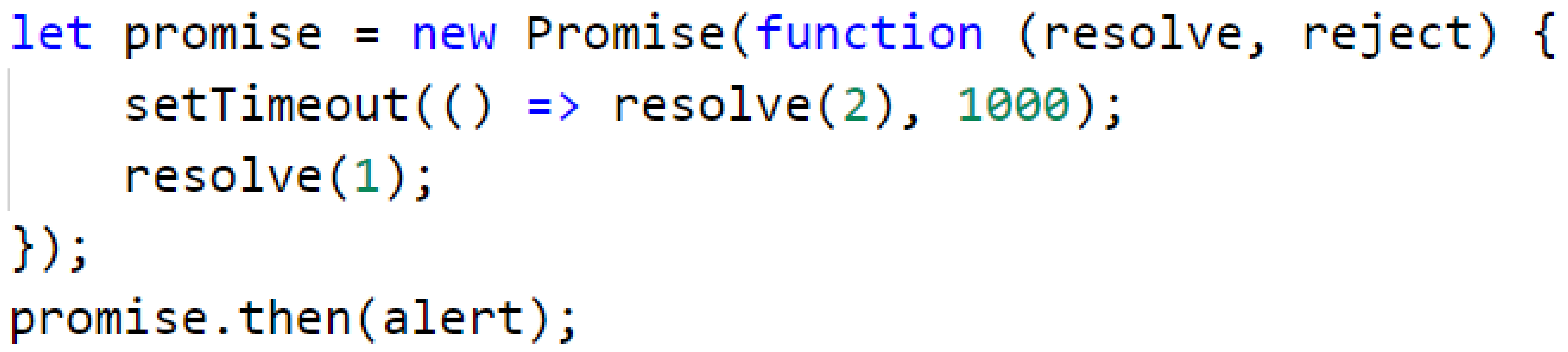
示例：以下代码的输出结果是什么?



```
function foo(st) {
    document.write(st)
}
let promise = new Promise(function (resolveFunc, rejectFunc) {
    resolveFunc("<h1>H</h1>ello ");
})
promise.then(foo);
```

A：当我们仅向 .then 方法传入一个函数时，该函数将取代 resolve 方法的默认定义。因此，上述代码将在浏览器窗口中显示 1。

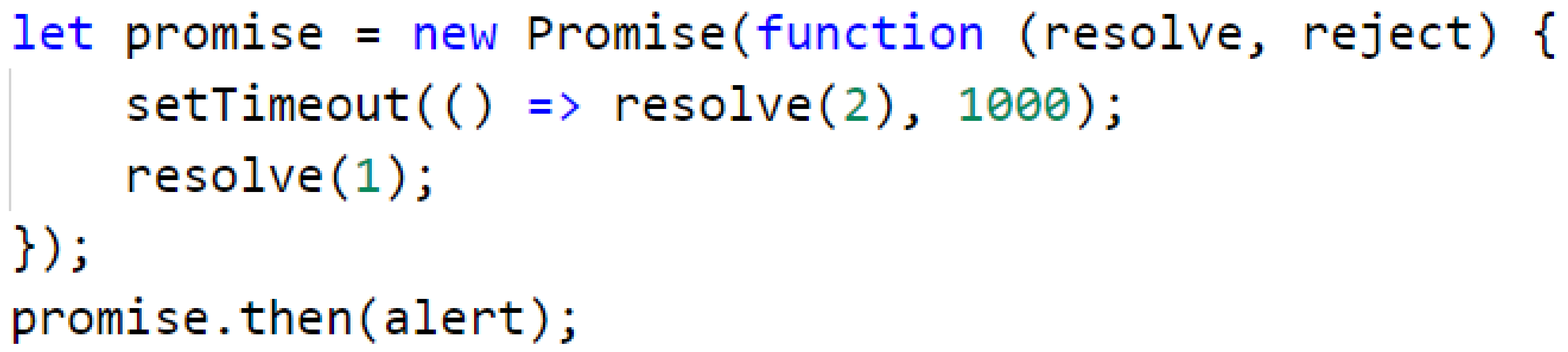
Example: What’s the output of the code below?



```
let promise = new Promise(function (resolve, reject) {
    setTimeout(() => resolve(2), 1000);
    resolve(1);
});
promise.then(alert);
```

it will alert “1” on the browser window and ignore the second resolve

示例：以下代码的输出结果是什么?



```
let promise = new Promise(function (resolve, reject) {
    setTimeout(() => resolve(2), 1000);
    resolve(1);
});
promise.then(alert);
```

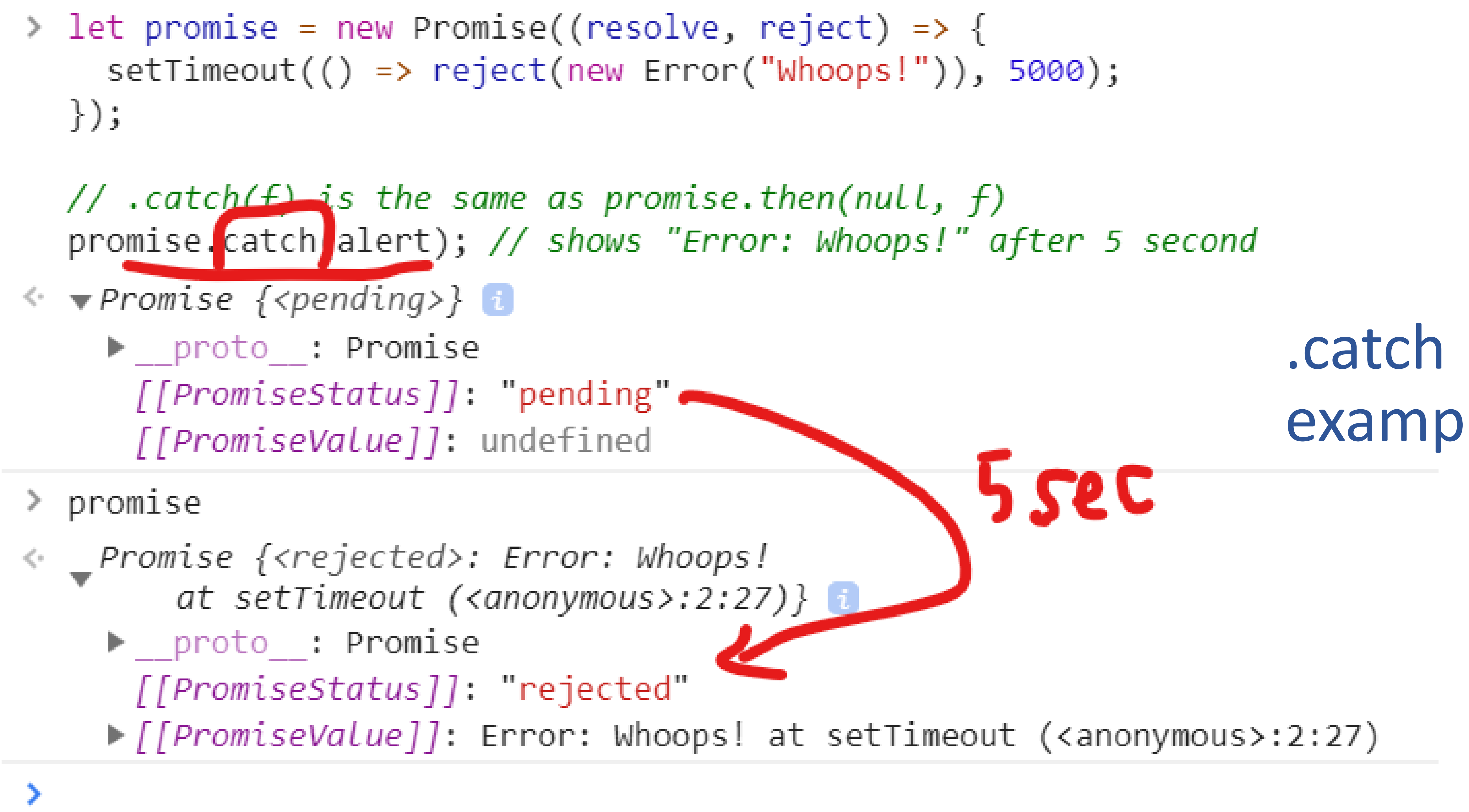
它将在浏览器窗口中弹出提示“1”，并忽略第二个 resolve。

.catch method

- If we’re interested only in errors, then we can use null as the first argument:
- .then(null, **errorHandlingFunction**)
- Is the same as
- .catch(**errorHandlingFunction**)

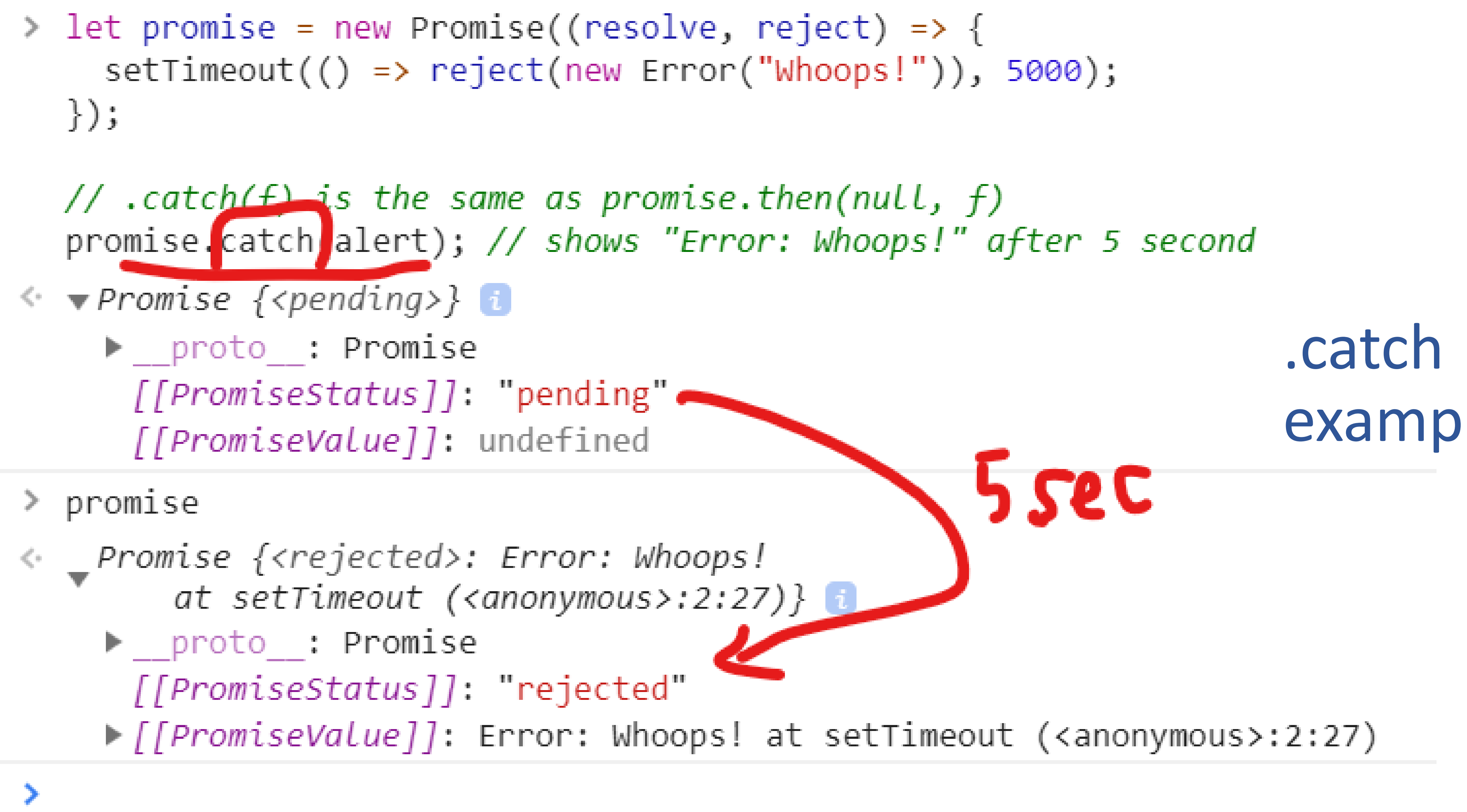
.catch 方法

- 如果我们只关心错误，则可将第一个参数设为 null：
- 。然后（空值，**错误处理函数**）
- 等价于
- .catch(**错误处理函数**)



```
> let promise = new Promise((resolve, reject) => {
    setTimeout(() => reject(new Error("Whoops!")), 5000);
  });

  // .catch(f) is the same as promise.then(null, f)
  promise.catch(alert); // shows "Error: Whoops!" after 5 second
<· ▼Promise {<pending>} i
    ▶__proto__: Promise
     [[PromiseStatus]]: "pending"
     [[PromiseValue]]: undefined
> promise
<· ▼Promise {<rejected>: Error: Whoops!
       at setTimeout (<anonymous>:2:27)} i
    ▶__proto__: Promise
     [[PromiseStatus]]: "rejected"
    ▶[[PromiseValue]]: Error: Whoops! at setTimeout (<anonymous>:2:27)
>
```

.catch
example



```
> let promise = new Promise((resolve, reject) => {
    setTimeout(() => reject(new Error("Whoops!")), 5000);
  });

  // .catch(f) is the same as promise.then(null, f)
  promise.catch(alert); // shows "Error: Whoops!" after 5 second
<· ▼Promise {<pending>} i
    ▶__proto__: Promise
     [[PromiseStatus]]: "pending"
     [[PromiseValue]]: undefined
> promise
<· ▼Promise {<rejected>: Error: Whoops!
       at setTimeout (<anonymous>:2:27)} i
    ▶__proto__: Promise
     [[PromiseStatus]]: "rejected"
    ▶[[PromiseValue]]: Error: Whoops! at setTimeout (<anonymous>:2:27)
>
```

5 sec

.catch
example

.finally

- Just like there's a finally clause in a regular try {...} catch {...}, there's finally in promises.

- The call **.finally(f)**
- is similar to
- .then(f, f)
- in the sense that it always runs when the promise is settled: be it resolve or reject. (src: javascript.info)

- Note: there are more into .finally method that is outside the scope of this class.

.finally

- 正如常规的 try {...} catch {...} 语句中存在 finally 子句一样，Promise 中也存在 finally 方法。

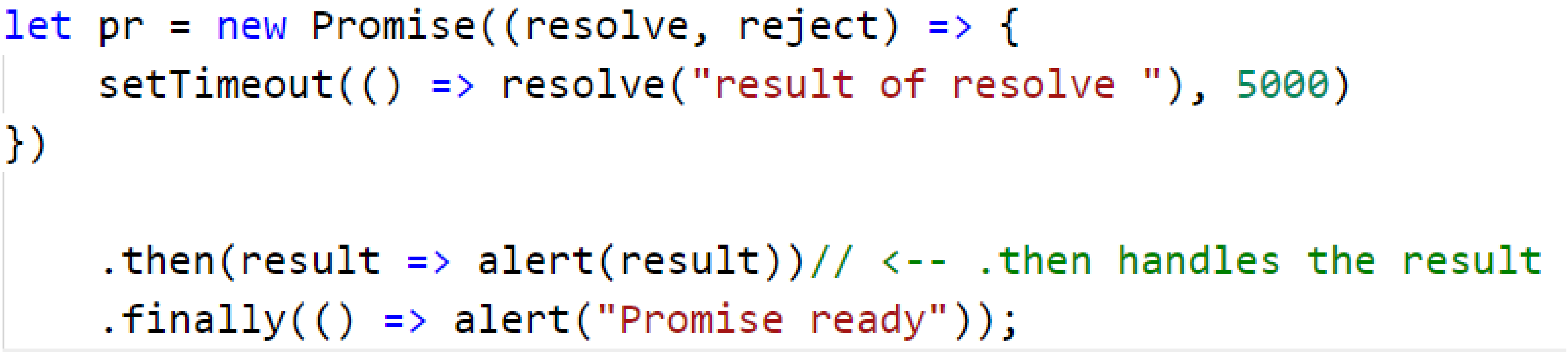
  Promise。

- 调用 **.finally(f)**
- 类似于
- .then(f, f)
- 其含义是：无论 Promise 最终是成功（resolve）还是失败（reject），该方法都会执行。（来源：javascript.info）

- 注意：.finally 方法还有更多细节，但超出了本课程的范围。

.finally method example

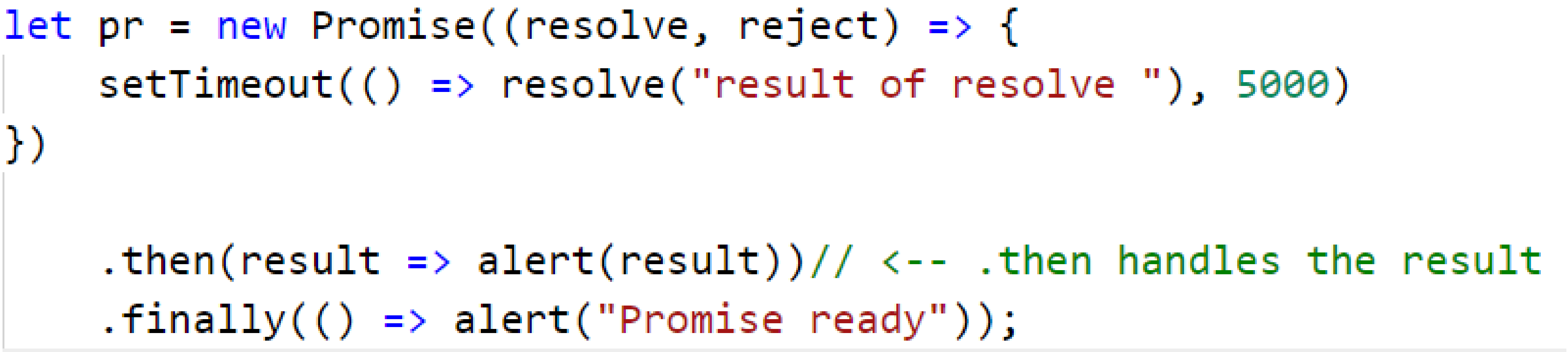
- Remember if resolve function executes, reject function will be ignore and vice versa. However regardless of how the promise settles the finally method will be called



```
let pr = new Promise((resolve, reject) => {
    setTimeout(() => resolve("result of resolve "), 5000)
})

    .then(result => alert(result))// <-- .then handles the result
    .finally(() => alert("Promise ready"));
```

.finally 方法示例

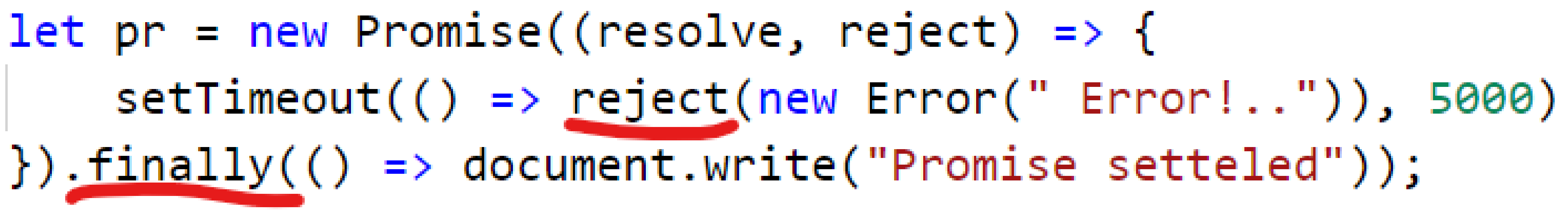
- 请注意：若 resolve 函数执行，则 reject 函数将被忽略，反之亦然。但无论 Promise 最终是成功还是失败，finally 方法均会被调用。



```
let pr = new Promise((resolve, reject) => {
    setTimeout(() => resolve("result of resolve "), 5000)
})

    .then(result => alert(result))// <-- .then handles the result
    .finally(() => alert("Promise ready"));
```

.finally method example 2

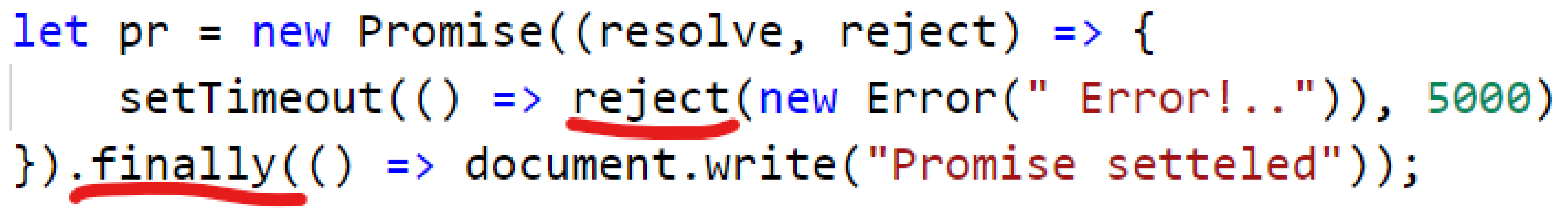
- Remember if resolve function executes, reject function will be ignore and vice versa. However regardless of how the promise settles the finally method will be called .
- In example below we did not catch error nor we defined reject function.
- Remember the default function would terminate the script



```
let pr = new Promise((resolve, reject) => {
    setTimeout(() => reject(new Error(" Error!..")), 5000)
}).finally(() => document.write("Promise setteled"));
```

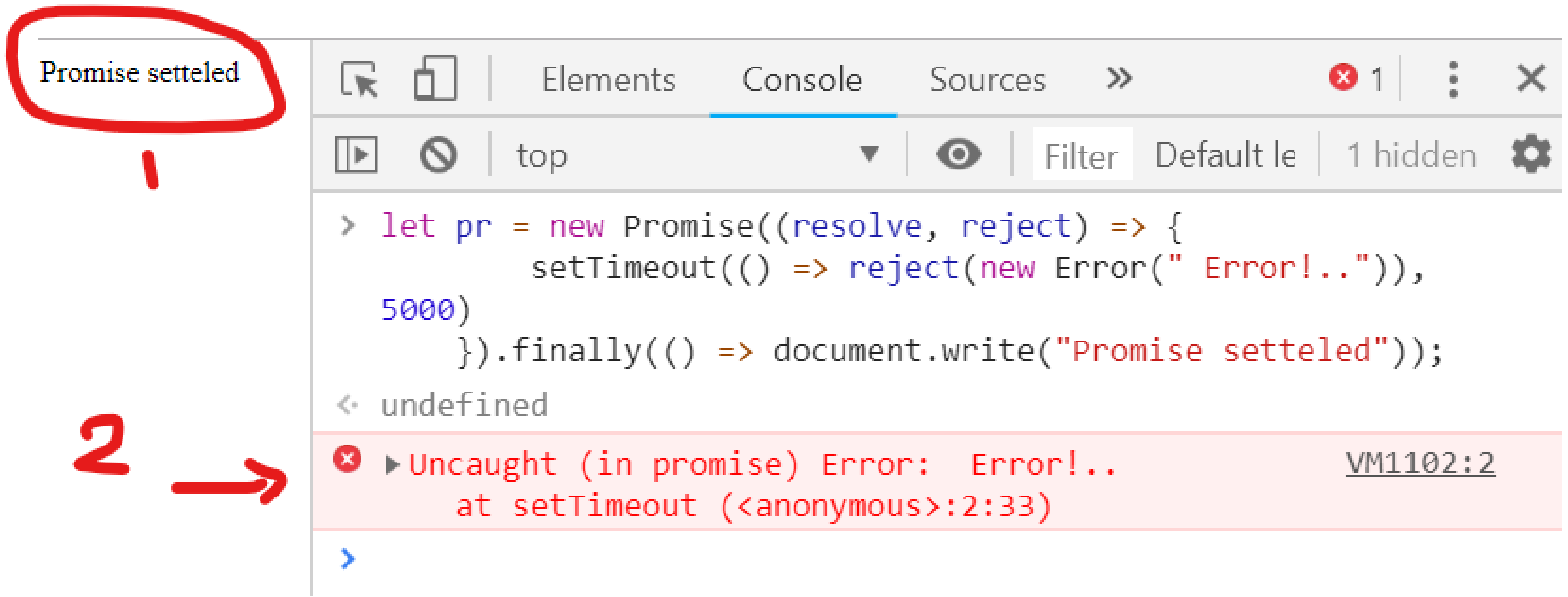
.finally 方法示例 2

- 请记住：若 resolve 函数执行，则 reject 函数将被忽略；反之亦然。但无论 Promise 最终是成功还是失败，finally 方法都会被执行。
- 在下方示例中，我们既未捕获错误，也未定义 reject 函数。
- 请注意，若未处理错误，其默认行为将导致脚本终止。



```
let pr = new Promise((resolve, reject) => {
    setTimeout(() => reject(new Error(" Error!..")), 5000)
}).finally(() => document.write("Promise setteled"));
```

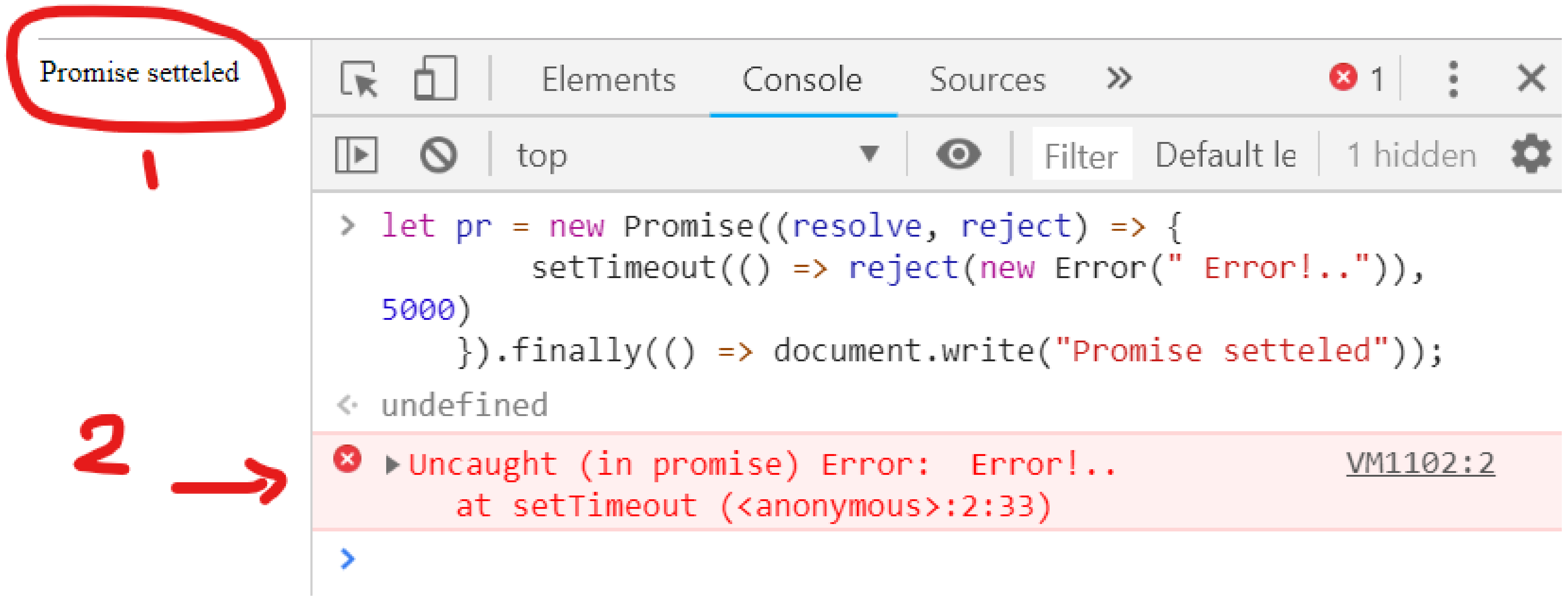
.finally method example 2 cont…



Promise setteled

```
> let pr = new Promise((resolve, reject) => {
          setTimeout(() => reject(new Error(" Error!..")),
  5000)
      }).finally(() => document.write("Promise setteled"));
<· undefined
✖ ▸Uncaught (in promise) Error:  Error!..                  VM1102:2
      at setTimeout (<anonymous>:2:33)
>
```

.finally 方法示例 2（续）…



Promise setteled

```
> let pr = new Promise((resolve, reject) => {
          setTimeout(() => reject(new Error(" Error!..")),
  5000)
      }).finally(() => document.write("Promise setteled"));
<· undefined
✖ ▸Uncaught (in promise) Error:  Error!..                  VM1102:2
      at setTimeout (<anonymous>:2:33)
>
```

Async/Await

- built on top of promises, allowing you to write asynchronous code that looks more like synchronous code.
- async: When a function is declared as async, it automatically returns a promise.
- await: This is used inside async functions to wait for a promise to resolve or reject.

- Good exapls at https://www.w3schools.com/js/js_async.as

异步/等待

- 基于 Promise 构建，使您能够编写外观更接近同步代码的异步代码。

- async：当函数被声明为 async 时，它会自动返回一个 Promise。
- await：在 async 函数内部使用，用于等待 Promise 的完成或拒绝。

- 优秀的示例位于 https://www.w3schools.com/js/js_async.as

In class activity 6

- Q1: For the given API server, measure the time diff from the time you sent a GET request, to the time you receive the response

  Q2: Measure the time diff from the time you receive the response header, to the time you receive the response

课堂活动 6

- 问题1：针对给定的API服务器，测量从你发送请求到收到响应头之间的时间差。GET 请求，直至您收到响应为止

  问题2：测量从你接收到响应头开始，到完全接收响应内容为止的时间差。

GET vs POST

- So we stated that there is a limitation on data size being sent over a single GET **request**. However, there is no such limitation on POST request
- Q: what about the **responses** of GET vs POST? Is there such limit for their responses too?

7

- A: no! There are no standardized limitations on the size of responses for GET or POST requests specified by the HTTP protocol.
  Needless to mention practical limitations can be influenced by server configurations, network constraints, or browser limitations.

GET 与 POST

- 因此，我们指出，通过单个 GET **请求** 发送的数据大小存在限制。但 POST 请求则无此类限制。
- 问：GET 与 POST 的 **响应** 有何区别？它们的响应大小是否也存在类似限制？

7

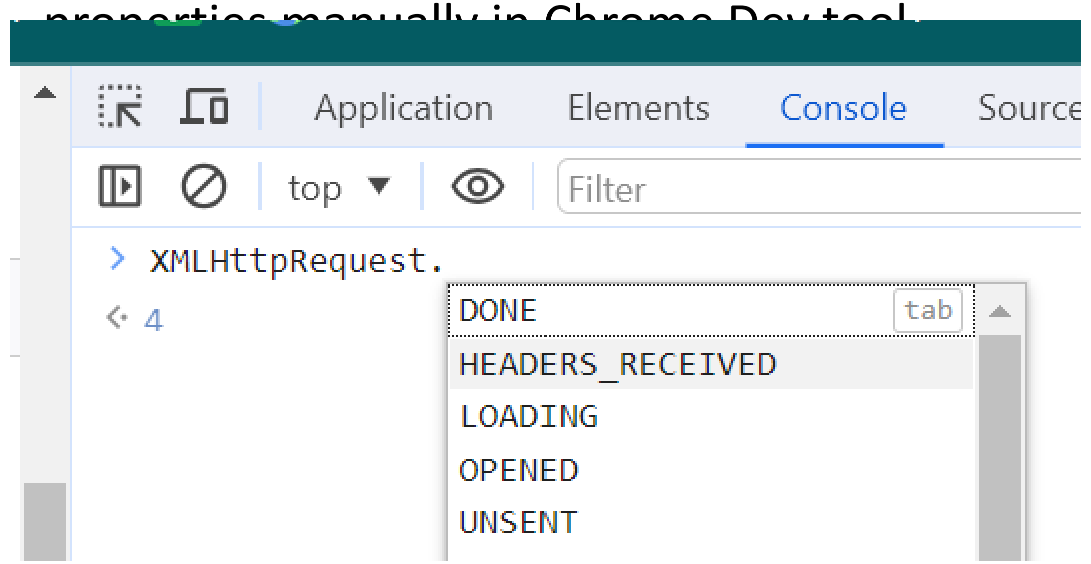
- 答：没有！HTTP 协议并未对 GET 或 POST 请求的响应大小规定任何标准化限制。当然，实际限制可能受服务器配置、网络条件或浏览器限制等因素影响。

W3schools is inaccurate about onreadystatechanged

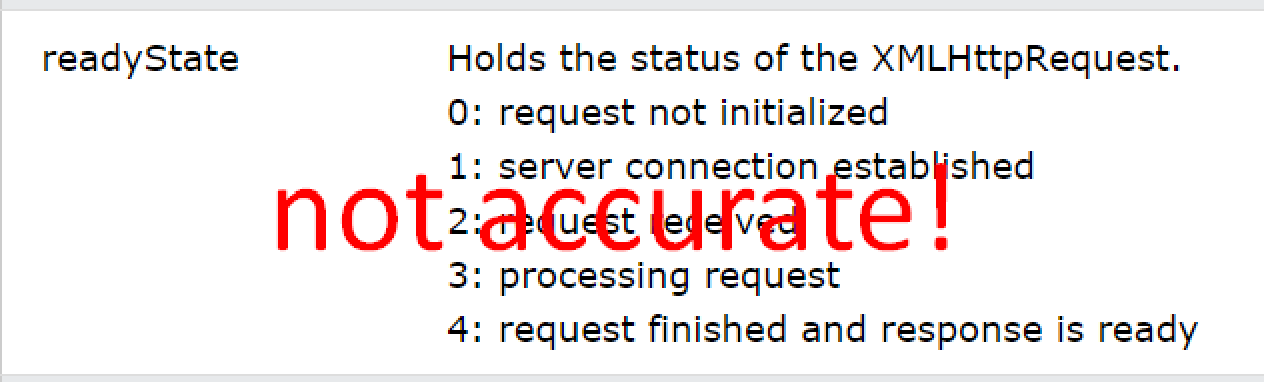
- Turns out https://www.w3schools.com/xml/ajax_xmlhttprequest_response.asp is not accurate on ready states status values ! The one we covered in lecture is accurate

8

- Accurate one:
  0 (UNSENT): The XHR object has been created, but open() has not been called yet.
- 1 (OPENED): open() has been called.
- 2 (HEADERS_RECEIVED): send() has been called, and the headers of the response are available.
- 3 (LOADING): The response is being received. As data comes in, the responseText property is updated.
- 4 (DONE): The operation is complete, and either the request has been successfully completed (status code 2xx) or an error occurred.
- So verified that by looking into the XMLHttpRequest class properties manually in Chrome Dev tool.



```
> XMLHttpRequest.
<· 4
DONE
HEADERS_RECEIVED
LOADING
OPENED
UNSENT
```



```
readyState      Holds the status of the XMLHttpRequest.
                0: request not initialized
                1: server connection established
                2: request received
                3: processing request
                4: request finished and response is ready
```

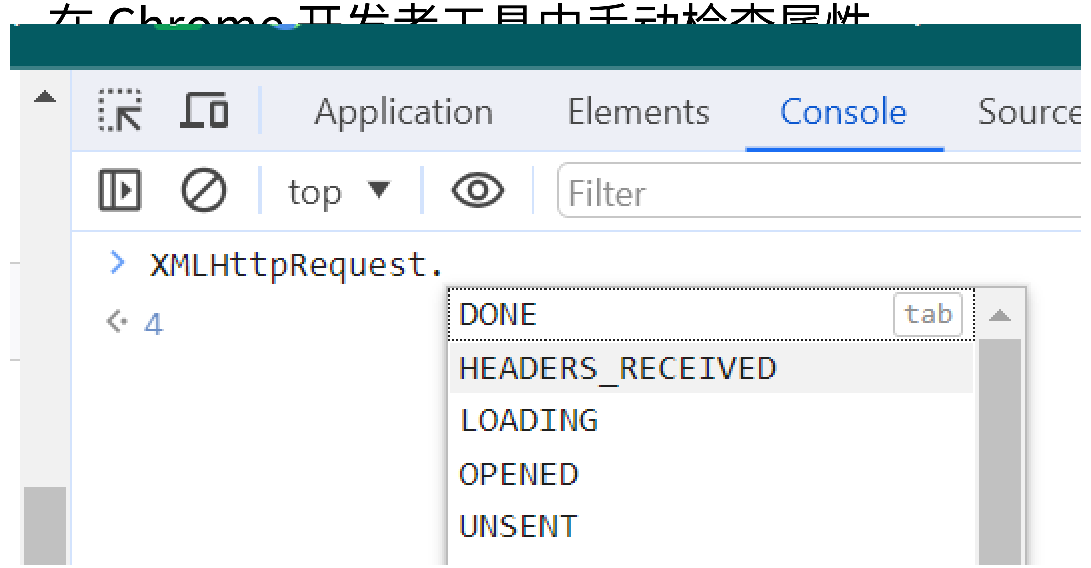
not accurate!

W3Schools 关于 onreadystatechange 的说明不准确 y状态通道 ged

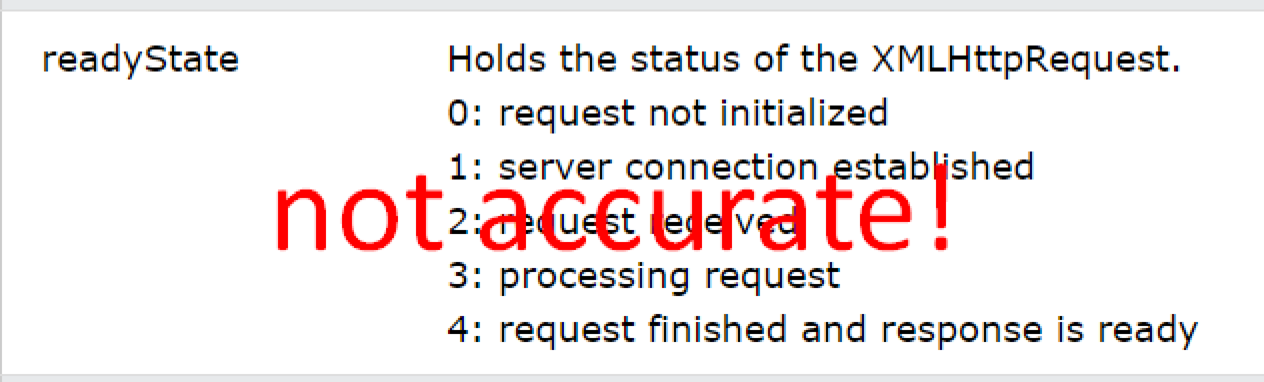
- 事实证明https://www.w3schools.com/xml/ajax_xmlhttprequest_response.asp关于就绪状态（readyState）取值的说明并不准确！我们在课堂上讲解的版本才是准确的

8

- 准确的定义如下：
  0（UNSENT）：XMLHttpRequest 对象已创建，但尚未调用 open() 方法。
- 1 （OPENED）：已调用 open() 方法。
- 2 (HEADERS_RECEIVED)：已调用 send() 方法，响应头信息已可用。
- 3 (LOADING)：正在接收响应。随着数据陆续到达，responseText 属性将被持续更新。
- 4 (DONE)：操作已完成，请求已成功完成（状态码为 2xx）或发生了错误。
- 因此，通过查阅 XMLHttpRequest 类验证了这一点在 Chrome 开发者工具中手动检查属性



```
> XMLHttpRequest.
<· 4
DONE
HEADERS_RECEIVED
LOADING
OPENED
UNSENT
```



```
readyState      Holds the status of the XMLHttpRequest.
                0: request not initialized
                1: server connection established
                2: request received
                3: processing request
                4: request finished and response is ready
```

not accurate!

References and resources

- https://scotch.io/tutorials/javascript-promises-for-dummies#toc-chaining-promises
- Spring.io
- MDN mozlla
- https://medium.com/front-end-weekly/ajax-async-callback-promise-e98f8074ebd7
- https://zapier.com/learn/apis/chapter-6-api-design/
- https://restfulapi.net/versioning/
- RESTful Web API Design with Node.js 10 - Third Edition
- https://phil.tech/2016/http-rest-api-file-uploads/ (for your term project)
- chatGPT

参考文献与资源

- https://scotch.io/tutorials/javascript-promises-for-dummies#toc-链接式调用 Promise
- Spring.io
- MDN Mozilla
- https://medium.com/front-end-weekly/ajax-async-callback-promise-e98f8074ebd7
- https://zapier.com/learn/apis/chapter-6-api-design/
- https://restfulapi.net/versioning/
- 使用 Node.js 10 进行 RESTful Web API 设计（第三版）
- https://phil.tech/2016/http-rest-api-file-uploads/（供你的课程设计使用）
- chatGPT