

# 8 javascript

## 8.1 基础知识

### basis

JavaScript

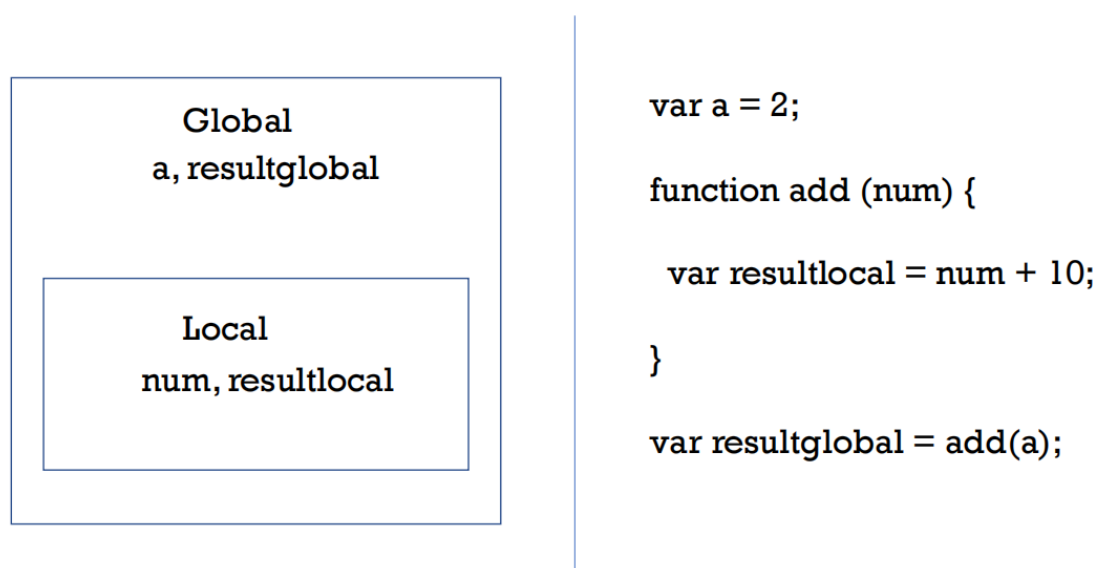
```
<script src="your JavaScript.js"></script>
<script> Write the code here </script>
```

HTML

```
<html>
<head> <title> Hello World</title> </head>
<body>
<script>
function doAnything(){
alert('Hello World');
}
doAnything();
</script>
</body>
</html>
```

## Execution Context

## Execution Context



执行上下文（Execution Context）是 JavaScript 中非常重要的概念之一，它描述了代码在运行时的环境和状态。每当 JavaScript 代码执行时，都会创建一个执行上下文，并按照一定的规则对其进行管理和执行。

一个执行上下文包含了以下信息：

1. **变量对象（Variable Object）**：用于存储在执行上下文中声明的变量、函数声明和形参。
2. **作用域链（Scope Chain）**：描述了在当前执行上下文中可以访问的变量和函数的范围。作用域链由当前执行上下文的变量对象和其父级执行上下文的变量对象构成。
3. **this 值**：指向当前执行上下文的对象。
4. **闭包信息**：当函数嵌套时，闭包信息记录了外部函数的变量对象和作用域链，以便内部函数可以访问外部函数的变量。

在 JavaScript 中，执行上下文的类型有全局执行上下文和函数执行上下文。全局执行上下文在页面加载时创建，并且只有一个，而函数执行上下文在每次函数调用时创建。执行上下文会按照调用栈的顺序进行管理，最后进入的执行上下文位于调用栈的顶部，称为当前执行上下文。

理解执行上下文对于了解 JavaScript 中的变量作用域、this 值、作用域链以及闭包等概念非常重要。

闭包是 JavaScript 中一个重要的概念，它指的是函数与其周围的词法环境的组合。这意味着函数可以访问在其定义时所处的词法作用域内的变量，即使在函数执行完成后，仍然可以访问这些变量。

下面是一个闭包的例子：

```
function outerFunction() {
    var outerVariable = 'I am from the outer function';

    function innerFunction() {
        console.log(outerVariable);
    }

    return innerFunction;
}

var closure = outerFunction(); // 调用外部函数，返回内部函数形成闭包
closure(); // 输出: "I am from the outer function"
```

HTML

```
var closure = outerFunction(); // 调用外部函数，返回内部函数形成闭包
closure(); // 输出: "I am from the outer function"
```

在这个例子中，`outerFunction` 是一个外部函数，它包含了一个内部函数 `innerFunction`。内部函数 `innerFunction` 访问了外部函数 `outerFunction` 中的变量 `outerVariable`。当我们调用 `outerFunction` 时，它返回了内部函数 `innerFunction`，并且形成了一个闭包。即使 `outerFunction` 执行完成后，闭包中的 `innerFunction` 仍然可以访问到 `outerVariable`。

闭包的一个重要特性是它可以“记住”其词法作用域，这使得闭包在很多情况下非常有用，比如在回调函数、事件处理程序、模块模式等方面的应用。

## Hoisting

## Hoisting

```
1 console.log(x);
2
3 var x = 10;
```

PROBLEMS OUTPUT DE

ja21121@C02DWCVPML7H Te  
Debugger attached.  
undefined

```
1 console.log(x);
2
3 //var x = 10;
```

PROBLEMS OUTPUT DEBUG CONSOLE

at Function.executeUserEntryP  
ja21121@C02DWCVPML7H Teaching 202  
Debugger attached.  
Waiting for the debugger to disco  
/Users/ja21121/Documents/Teaching  
console.log(x);  
^  
ReferenceError: x is not defined

## Hoisting

```
Users > ja21121 > Documents > Teaching 2023 > JS h
1 console.log(calledbeforedeclared());
2
3 function calledbeforedeclared(){
4     var x = 10;
5     console.log(x);
6 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

ja21121@C02DWCVPML7H Teaching 2023 % node hoi  
Debugger attached.  
10

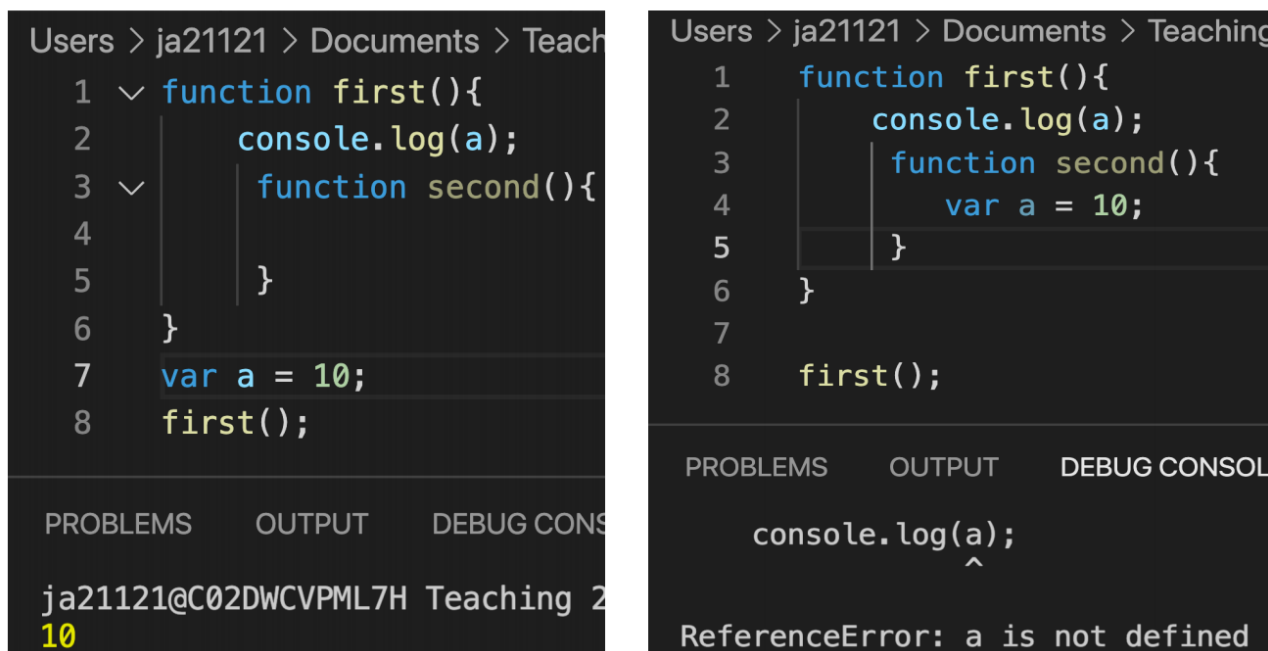
Hoisting（提升）是 JavaScript 中的一个概念，指的是在代码执行过程中，变量和函数声明会被提升到其所在作用域的顶部。这意味着可以在声明之前使用变量和函数，但它们的赋值或定义并不会提升。具体来说：

1. 变量声明会被提升，但赋值不会。
2. 函数声明会被提升，包括函数体内的函数。

3. 变量和函数的提升只会发生在其所在的作用域内，如果在函数内部声明的变量或函数，则它们只会在该函数内部被提升。

## Scope 作用域

### Scope



```
Users > ja21121 > Documents > Teaching
1  function first(){
2      console.log(a);
3      function second(){
4          var a = 10;
5      }
6  }
7  var a = 10;
8  first();

PROBLEMS  OUTPUT  DEBUG CONSOLE
ja21121@C02DWCVPML7H Teaching 2
10
```

```
Users > ja21121 > Documents > Teaching
1  function first(){
2      console.log(a);
3      function second(){
4          var a = 10;
5      }
6  }
7
8  first();

PROBLEMS  OUTPUT  DEBUG CONSOLE
console.log(a);
^
ReferenceError: a is not defined
```

作用域（Scope）和执行上下文（Execution Context）是 JavaScript 中两个重要的概念，它们有着不同的含义和作用。

#### 1. 作用域（Scope）：

- 作用域指的是变量和函数的可访问性和可见性范围。
- 在 JavaScript 中，作用域是词法（静态）作用域，即作用域在代码编写阶段就确定了，与函数的调用位置无关。
- 作用域定义了变量在代码中的可见性，包括全局作用域和局部作用域。

#### 2. 执行上下文（Execution Context）：

- 执行上下文是 JavaScript 中代码执行的环境，包含了当前代码的所有变量、函数、以及对外部环境的引用。
- 在 JavaScript 中，每次执行代码都会创建一个新的执行上下文。
- 执行上下文分为全局执行上下文和函数执行上下文，以及通过 `eval()` 和 `with` 创建的执行上下文。

简而言之，作用域决定了代码中变量的可见性和访问范围，而执行上下文则是代码执行的环境，包含了代码中的所有内容以及与外部环境的联系。

## Call Stack

当我们调用一个函数时，它会被添加到调用栈的顶部，然后在执行完成后从栈中移除。让我们通过一个简单的示例来说明：

```
JavaScript

function greet(name) {
  console.log("Hello, " + name + "!");
}

function sayHello() {
  let userName = "Alice";
  greet(userName);
}

sayHello();
```

在这个示例中，我们有两个函数：`greet` 和 `sayHello`。当 `sayHello` 函数被调用时，它将 `userName` 设置为 "Alice"，然后调用 `greet` 函数，并传递 `userName` 作为参数。

函数调用的顺序如下：

1. 首先，`sayHello` 函数被添加到调用栈的顶部。
2. 在 `sayHello` 函数内部，`userName` 变量被设置为 "Alice"。
3. 然后，`greet` 函数被调用，并且 "Alice" 被传递给 `name` 参数。
4. 在 `greet` 函数内部，它输出 "Hello, Alice!" 到控制台。
5. 最后，`greet` 函数执行完成，从调用栈中移除。
6. 接着，`sayHello` 函数执行完成，也从调用栈中移除。

在这个过程中，调用栈会根据函数调用的顺序动态地增加和减少，以跟踪函数的执行流程。

# Syntax

## 1. break:

JavaScript

```
for (var i = 0; i < 5; i++) {  
    if (i === 3) {  
        break;  
    }  
    console.log(i);  
}  
// 输出: 0 1 2
```

## 2. if...else:

JavaScript

```
var age = 20;  
if (age >= 18) {  
    console.log("You are an adult.");  
} else {  
    console.log("You are a minor.");  
}  
// 输出: You are an adult.
```

## 3. for:

JavaScript

```
for (var i = 0; i < 5; i++) {  
    console.log(i);  
}  
// 输出: 0 1 2 3 4
```

## 4. function:

JavaScript

```
function greet(name) {  
    console.log("Hello, " + name + "!");  
}  
greet("John");  
// 输出: Hello, John!
```

## 5. do...while:

JavaScript

```
var i = 0;
do {
  console.log(i);
  i++;
} while (i < 5);
// 输出: 0 1 2 3 4
```

## 6. var, let & const:

JavaScript

```
var a = 10; // 使用 var 声明变量
let b = 20; // 使用 let 声明变量
const PI = 3.14; // 使用 const 声明常量
```

## 7. return:

JavaScript

```
function add(a, b) {
  return a + b;
}
var result = add(3, 4);
console.log(result); // 输出: 7
```

## 8. switch:



```
var day = 2;
switch (day) {
  case 1:
    console.log("Monday");
    break;
  case 2:
    console.log("Tuesday");
    break;
  default:
    console.log("Other day");
}
// 输出: Tuesday
```

## 9. throw:

```
function divide(a, b) {
  if (b === 0) {
    throw "Division by zero is not allowed.";
  }
  return a / b;
}
try {
  var result = divide(10, 0);
  console.log(result);
} catch (error) {
  console.log("Error: " + error);
}
// 输出: Error: Division by zero is not allowed.
```

## 10. try...catch:

JavaScript

```
try {
  var result = x / y;
  console.log(result);
} catch (error) {
  console.log("An error occurred: " + error.message);
}
```

## 11. arrays:

JavaScript

```
var example = [1, "joe", 5.555];
```

比较运算符: `<`, `<=`, `>`, `>=`, `==`, `===`

逻辑运算符: `&&`, `||`, `!`

JavaScript

```
console.log("3.0" === 3); // false, 因为类型不同, 且值也不同
console.log("3.0" == 3);  // true, 因为值相同, JavaScript 会进行类型转换
```

JavaScript 不需要声明类型, 语言会自动进行类型推断。

大多数情况下, 它会正确地推断类型, 但当类型不匹配时, 可以使用 `parseInt` 来转换为整数。

许多运算符会自动转换类型, 因此在许多操作中会发生类型转换。

JavaScript

```
var name = prompt("What's your name?"); // 用户输入的值通常会被转换为字符串
var age = 20; // 整数类型
```

## Arrays

JavaScript

```
var name = `johndoe`;
console.log(name.length);           = 7
console.log(name[0]);               = 'j'
```

```
console.log(name[name.length]);      = undefined
console.log(name[name.length-1]);    = 'e'
```

JavaScript

```
var example = [1, joe, 5.555]
for var element in //index
example {
  console.log(element);
}
for var element of //value
example {
  console.log(element);
}
```

在这个例子中，`for...in` 循环会迭代数组的索引（即元素的键名），而 `for...of` 循环会迭代数组的元素值。

在 JavaScript 中，`for...in` 和 `for...of` 是两种不同的循环语句，用于迭代对象和数组的元素。它们之间的区别如下：

1. `for...in` 循环用于迭代对象的可枚举属性，通常用于遍历对象的键名。它会遍历对象自身的可枚举属性以及继承的可枚举属性。

JavaScript

```
const obj = { a: 1, b: 2, c: 3 };

for (let key in obj) {
  console.log(key); // 输出: 'a', 'b', 'c'
}
```

1. `for...of` 循环用于迭代可迭代对象（如数组、字符串、Map、Set 等）的元素，而不是对象的属性。它会迭代对象本身的元素，而不会迭代继承的属性。

JavaScript

```
const arr = [1, 2, 3];

for (let value of arr) {
  console.log(value); // 输出: 1, 2, 3
}
```

因此, `for...in` 适用于迭代对象的属性, 而 `for...of` 适用于迭代可迭代对象的元素。

1. `map()`: 通过将函数应用于原始数组中的每个元素来创建一个新数组。

```
const numbers = [1, 2, 3, 4];
const doubled = numbers.map(num => num * 2);
// doubled 是 [2, 4, 6, 8]
```

JavaScript

2. `filter()`: 通过由函数提供的测试来创建一个新数组, 其中包含所有通过测试的元素。

```
const numbers = [1, 2, 3, 4];
const evenNumbers = numbers.filter(num => num % 2 === 0);
// evenNumbers 是 [2, 4]
```

JavaScript

3. `reduce()`: 通过将函数应用于每个元素, 将数组减少为单个值。

```
const numbers = [1, 2, 3, 4];
const sum = numbers.reduce((acc, curr) => acc + curr, 0);
// sum 是 10
```

JavaScript

4. `forEach()`: 为数组的每个元素执行一次提供的函数。

```
const numbers = [1, 2, 3, 4];
numbers.forEach(num => console.log(num));
// 输出: 1, 2, 3, 4
```

JavaScript

5. **push()**: 向数组末尾添加一个或多个元素，并返回数组的新长度。

JavaScript

```
const numbers = [1, 2, 3];  
numbers.push(4);  
// numbers 现在是 [1, 2, 3, 4]
```

6. **pop()**: 从数组中删除最后一个元素，并返回该元素。

JavaScript

```
const numbers = [1, 2, 3];  
const lastElement = numbers.pop();  
// lastElement 是 3, numbers 现在是 [1, 2]
```

7. **shift()**: 从数组中删除第一个元素，并返回该元素。

JavaScript

```
const numbers = [1, 2, 3];  
const firstElement = numbers.shift();  
// firstElement 是 1, numbers 现在是 [2, 3]
```

8. **unshift()**: 向数组开头添加一个或多个元素，并返回数组的新长度。

JavaScript

```
const numbers = [2, 3];  
numbers.unshift(1);  
// numbers 现在是 [1, 2, 3]
```

## Anonymous Functions

在这个例子中，我们使用 `map()` 方法将一个函数应用于数组中的所有元素，并返回一个新的数组，其中包含每个元素经过函数处理后的值。

首先，我们有一个名为 `age` 的数组，其中包含一些年龄值 `[20, 25, 30, 35]`。然后，我们使用 `map()` 方法来遍历这个数组，并对每个元素执行一个匿名

函数，该函数将当前元素的值加上 10。最后，`map()` 方法返回一个新的数组，其中包含每个元素加上 10 后的结果。

下面是具体的代码示例：

```
JavaScript

var age = [20, 25, 30, 35];

// 使用 map() 方法将每个元素的值加上 10
age = age.map(function(value) {
    return value + 10;
});

console.log(age); // 输出 [30, 35, 40, 45]
```

在上面的代码中，`map()` 方法遍历了数组 `age` 中的每个元素，并对每个元素执行了一个匿名函数，该函数将当前元素的值加上 10。最后，`map()` 返回一个新的数组，其中包含了每个元素加上 10 后的结果。

匿名函数（Anonymous Functions）是指在定义时不指定函数名称的函数。在 JavaScript 中，可以使用函数表达式来创建匿名函数。函数表达式可以赋值给变量，也可以作为其他函数的参数传递。

```
JavaScript

// 匿名函数赋值给变量
const greet = function() {
    console.log("Hello!");
};

// 调用匿名函数
greet();
```

在上面的示例中，我们定义了一个匿名函数，并将其赋值给变量 `greet`。然后，我们可以通过变量名 `greet` 来调用这个匿名函数，输出 "Hello!"。

匿名函数经常用作回调函数或者立即执行函数（Immediately Invoked Function Expressions, IIFE）。例如，在事件处理器中，可以使用匿名函数作为回调函数来处理事件。

```
// 匿名函数作为事件处理器的回调函数
document.addEventListener('click', function() {
    console.log('Document clicked!');
});
```

匿名函数的优势在于可以灵活地在需要时定义和使用函数，而不必为函数指定名称。因此，匿名函数在需要临时性或简单功能的情况下特别有用。

## Arrow Functions

箭头函数是 ES6 中的一种新的函数声明方式，它提供了一种更简洁的语法形式来定义函数，并且在某些情况下可以改变 `this` 的指向。

箭头函数的语法形式是用小括号括起来的参数列表，后跟一个箭头 `=>`，然后是函数体。如果函数体只有一条语句，可以省略大括号和 `return` 关键字，否则需要显式地使用大括号来包裹函数体，并且需要使用 `return` 来返回值。

下面是一个箭头函数的基本示例：

```
// 传统函数声明
function add(a, b) {
    return a + b;
}

// 箭头函数
const addArrow = (a, b) => a + b;

console.log(add(2, 3)); // 输出 5
console.log(addArrow(2, 3)); // 输出 5
```

箭头函数通常在匿名函数和回调函数中使用，它具有更简洁的语法形式，并且在一些情况下可以更清晰地表达函数的意图。此外，箭头函数内部的 `this` 是词法作用域，指向的是定义时所在的对象，而不是调用时所在的对象，这与传统的函数声明有所不同。

- Write shorter functions

```
<!DOCTYPE html>
<html>
<body>

<h1>JavaScript Functions</h1>
<h2>The Arrow Function</h2>

<p>This example shows the syntax of an Arrow Function,
and how to use it.</p>

<p id="demo"></p>
|
<script>
let hello = "";

hello = () => {
  return "Hello World!";
}

document.getElementById("demo").innerHTML = hello();
</script>
```

## JavaScript Functions

### The Arrow Function

This example shows the syntax of an Arrow Function, and how to use it.

Hello World!

## Events in JavaScript

JavaScript 中的事件是指用户与网页交互时发生的各种动作，例如点击、鼠标移动、键盘输入等。通过捕获和处理这些事件，我们可以实现网页的动态交互和响应用户操作的功能。

事件通常与 DOM 元素相关联，当事件发生时，可以触发相应的事件处理函数来执行特定的操作。常见的事件包括：

1. **click**: 鼠标点击事件，当用户点击元素时触发。
2. **mouseover**: 鼠标悬停事件，当鼠标移动到元素上方时触发。
3. **keydown**: 按键按下事件，当用户按下键盘上的按键时触发。
4. **submit**: 表单提交事件，当用户提交表单时触发。
5. **change**: 输入内容改变事件，当输入框的内容发生改变时触发。
6. **load**: 页面加载完成事件，当页面加载完成时触发。
7. **DOMContentLoaded**: DOM 加载完成事件，当 DOM 树构建完成时触发。

在 JavaScript 中，我们可以通过以下方式来处理事件：

1. 在 HTML 元素上直接添加事件属性，例如 `onclick`、`onmouseover` 等。
2. 使用 DOM API 在 JavaScript 中动态添加事件监听器，例如 `addEventListener` 方法。



3. 使用事件委托（Event Delegation）来管理事件，将事件监听器绑定到父元素而不是每个子元素上，可以提高性能并简化代码。

以下是一个简单的示例，演示了如何使用事件监听器来响应按钮的点击事件：

HTML

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Event Handling Example</title>
</head>
<body>
  <button id="myButton">Click Me</button>

  <script>
    // 获取按钮元素
    const button = document.getElementById('myButton');

    // 添加点击事件监听器
    button.addEventListener('click', function() {
      alert('Button clicked!');
    });
  </script>
</body>
</html>
```

JavaScript

```
<html>
<head> <title> Event Handlers </title> </head>
<body>
  <button onclick="alertName(event)">Button 1 </button>
  <button onclick="alertName(event)">Button 2 </button>
  <script>
    function alertName(event)
    {
      var trigger = event.srcElement;
      alert('You clicked on ' + trigger.innerHTML);
    }
  </script>
```

```
</body>
</html>
```

## 8.2 json

### The Basics

要在网页上显示数据，你可以使用各种编程语言。对于发送数据到网页的 JavaScript 来说，最常见的方式是使用 JSON（JavaScript Object Notation）。

JSON 支持以下数据类型：

- 数字（numbers）
- 布尔值（booleans）
- 字符串（strings）
- 空值（null）
- 数组（ordered sequences of values）
- 对象（objects）

然而，JSON 不支持以下内容：

- 函数（functions）
- 正则表达式（regular expressions）等
- 列表

JavaScript

```
Strings in JSON. { "name":"Joe" } (Strings in double quotes)
Numbers in JSON { "age":20 }
JSON can be objects. { "student":{ "name":"Joe", "age":30} }
```

```
1  ∨ const data = {
2      "name": "CyberJoe",
3      "hobby": "hacking",
4      "language" : ["JavaScript", "HTML", "CSS"]
5  }
6  // accessing JSON object
7  console.log(data.name);
8  console.log(data.hobby);
9  console.log(data.language[0]);
```

PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL

```
ja21121@C02DWCVPML7H Teaching 2023 % node json.js
Debugger attached.
CyberJoe
hacking
JavaScript
Waiting for the debugger to disconnect...
```

JavaScript

```
const person = {
  "name": "Alice",
  "age": 25
};

console.log(person.name); // 输出: Alice
console.log(person.age); // 输出: 25
```

stringify

```
Users > ja21121 > Documents > Teaching 2023 > JS index.js > ...
1  const obj = {name: "John", age: 30, city: "New York"};
2  const myJSON = JSON.stringify(obj);
3  console.log(myJSON);
```

JavaScript object to JSON object

PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL

```
ja21121@C02DWCVPML7H Teaching 2023 % node index.js
Debugger attached.
{"name":"John","age":30,"city":"New York"}
Waiting for the debugger to disconnect...
ja21121@C02DWCVPML7H Teaching 2023 %
```

parse

```
Users > ja21121 > Documents > Teaching 2023 > JS index.js > ...
1  const string = '{"name":"John","age":30,"city":"New York"}';
2  const myJSON = JSON.parse(string);
3  console.log(myJSON);
```

JSON object to JavaScript Object

PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL

```
ja21121@C02DWCVPML7H Teaching 2023 % node index.js
Debugger attached.
{ name: 'John', age: 30, city: 'New York' }
Waiting for the debugger to disconnect...
ja21121@C02DWCVPML7H Teaching 2023 %
```

`JSON.stringify()` 和 `JSON.parse()` 是 JSON 对象的两种常见方法，它们分别用于将 JavaScript 对象转换为 JSON 字符串和将 JSON 字符串解析为 JavaScript 对象。

- **JSON.stringify()**: 将 JavaScript 对象转换为 JSON 字符串。这在需要将 JavaScript 对象序列化为字符串以便进行传输或存储时非常有用。例如：

```
const obj = { name: "Alice", age: 25 };
const jsonString = JSON.stringify(obj);
console.log(jsonString); // 输出: {"name":"Alice","age":25}
```

- **JSON.parse()**: 将 JSON 字符串解析为 JavaScript 对象。这在接收到 JSON 格式的数据后需要将其转换为 JavaScript 对象时非常有用。例如:

```
const jsonString = '{"name":"Alice","age":25}';
const obj = JSON.parse(jsonString);
console.log(obj); // 输出: { name: "Alice", age: 25 }
```

总的来说, `JSON.stringify()` 用于序列化 JavaScript 对象为 JSON 字符串, 而 `JSON.parse()` 用于解析 JSON 字符串并将其转换为 JavaScript 对象。

## JSON vs XML

```
{
  "employees": [
    { "firstName": "John", "lastName": "Doe" },
    { "firstName": "Anna", "lastName": "Smith" },
    { "firstName": "Peter", "lastName": "Jones" }
  ]
}
```

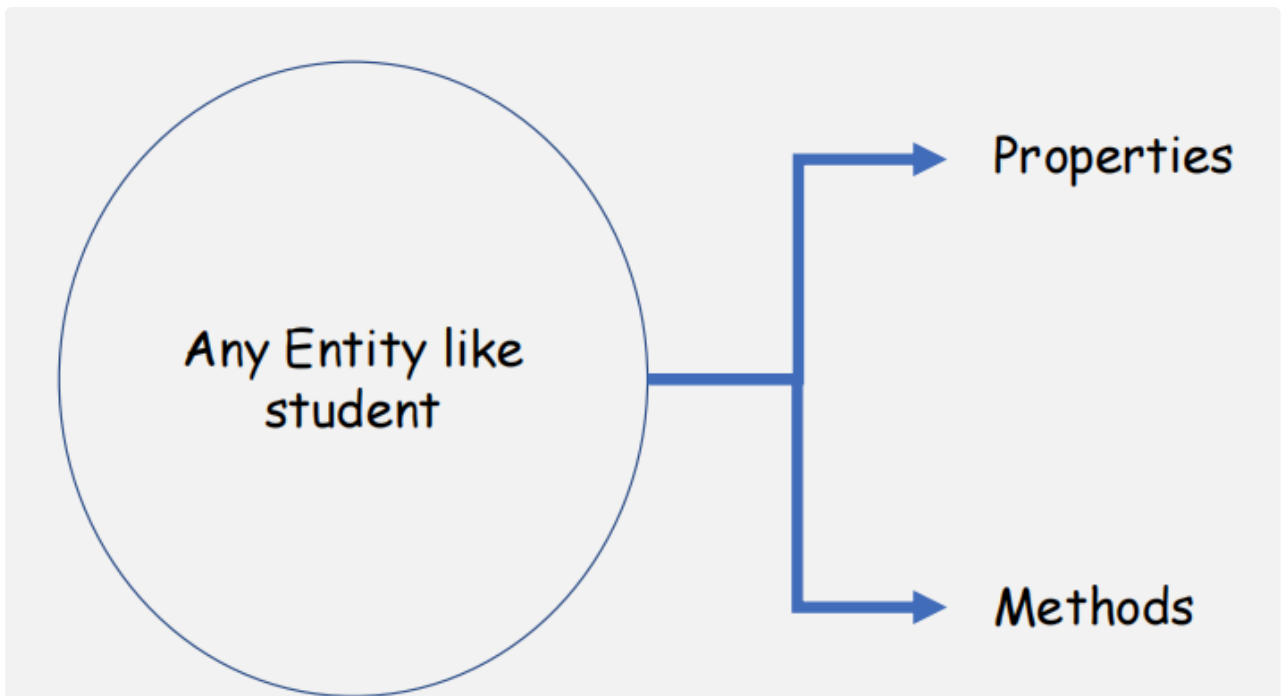
```
const data = '{"employees":[{"firstName":"John","lastName":"Doe"}, {"firstName":"Anna","lastName":"Smith"}, {"firstName":"Peter","lastName":"Jones"}]}'
const parsedData = JSON.parse(data);

console.log(parsedData.employees[0].firstName); // 输出: John
console.log(parsedData.employees[1].lastName); // 输出: Smith
```

在这里, `parsedData.employees` 让我们可以访问员工对象数组, 并且我们可以使用数组表示法访问单个员工的属性。

## 8.3 Object Oriented Paradigm

### Modelling a system



We model such entities using a Class

```
class Students
```

```
  name
```

```
  course
```

```
  marks
```

} Properties

```
  calculategrades
```

} Methods

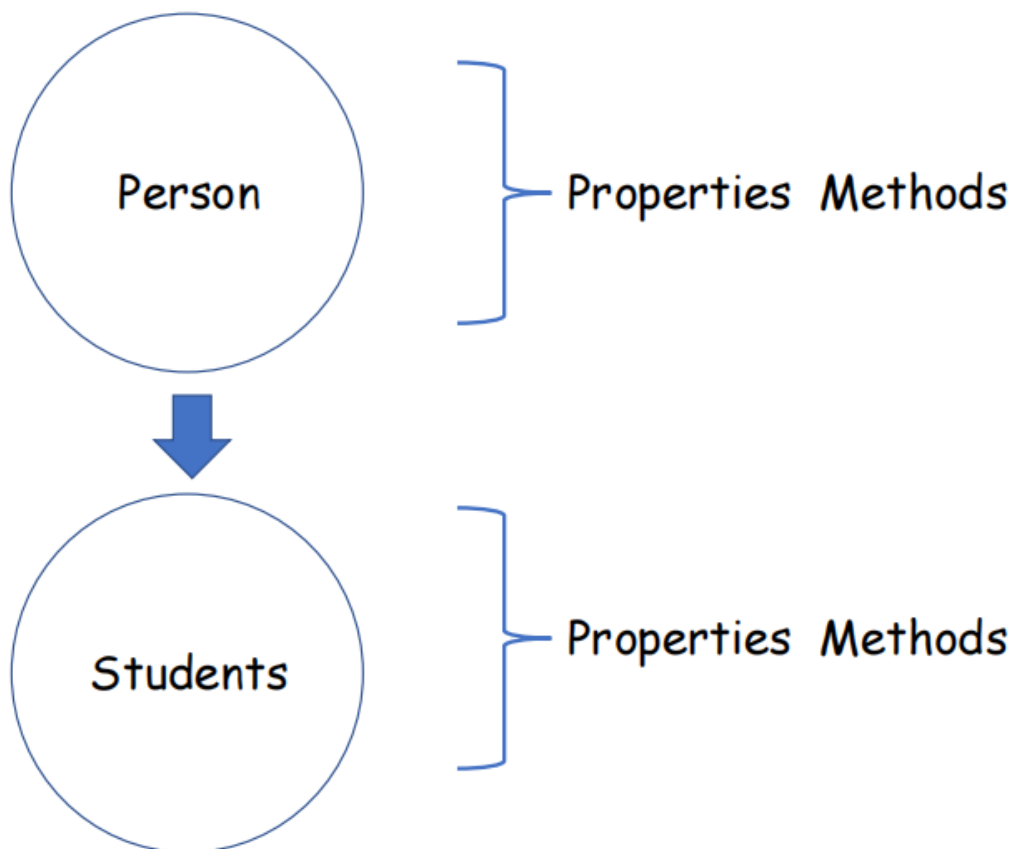
```
  students
```

} Constructor

类并不存在于内存中，而是类的实例，也就是对象存在于内存中。类具有一个特定的方法称为构造函数，构造函数用于初始化对象的值。

joe — new students ( Joe Gardiner, Software Tools, 100)

## Inheritance



```
Users > ja21121 > Documents > Teaching 2023 > JS class.js
1  class Students {
2    |   name;
3    constructor(name){
4      |   this.name = name;
5
6    }
7    myname(){
8      |   console.log(this.name);
9    }
10 }
11 const joe = new Students("Joe Gardiner");
12 joe.myname();
```

PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL

Joe Gardiner  
ja21121@C02DWCVPML7H Teaching 2023 %

JavaScript 对象是轻量级的

- 可以使用文字字面量创建对象，而无需使用类
- 也可以使用带有构造函数的函数来创建对象



## Literals

```
Users > ja21121 > Documents > Teaching 2023 >  
1  ∨ const Students = {  
2      name: "Joe Gardiner",  
3      course: "Software Tools",  
4  }  
5  
6  
7  console.log(Students.name);  
8  console.log(Students.course);  
  
PROBLEMS    OUTPUT    DEBUG CONSOLE    TER  
ja21121@C02DWCVPML7H Teaching 2023 % node  
Joe Gardiner  
Software Tools  
ja21121@C02DWCVPML7H Teaching 2023 %
```

## Functions

```
Users > ja21121 > Documents > Teaching 2023 > JS olite  
1  ∨ function Students(name){  
2      |   this.name = name;  
3  }  
4  
5  var joe = new Students("Joe Gardiner");  
6  console.log(joe.name);  
7  
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL  
ja21121@C02DWCVPML7H Teaching 2023 % node olite  
Joe Gardiner
```

Inheritance

```
7   Students.prototype.city="Bristol";
8   console.log(joe.city);
9   var marvin = new Students("Marvin Kopo");
10  console.log(marvin.name);
11  console.log(marvin.city);
12
```

PROBLEMS

OUTPUT

DEBUG CONSOLE

TERMINAL

```
ja21121@C02DWCVPML7H Teaching 2023 % node olitera
Joe Gardiner
Bristol
Marvin Kopo
Bristol
```

## 8.4 Aysnchronous programming

### Basics

JavaScript 具有同步执行的特性，它会按照代码的编写顺序一行一行地执行。

然而，在现实世界中，有时我们需要等待某些操作完成，例如从服务器获取数据或执行耗时的计算，但这不应该阻止程序的其余部分继续执行。为了处理这种情况，JavaScript 引入了异步执行的概念，可以通过回调函数和 promises 来实现。

- **回调函数 (Callbacks)**：回调函数是一种在异步操作完成后执行的函数。我们可以将回调函数作为参数传递给异步函数，在异步操作完成后调用回调函数来处理结果。
- **Promises**：Promise 是一种表示异步操作最终完成或失败的对象。它可以让我们以更加直观和易于理解的方式编写异步代码，避免了回调函数中出现的回调地狱问题。Promise 提供了 `then()` 方法来处理操作成功的情况，以及 `catch()` 方法来处理操作失败的情况。

### Callbacks

JavaScript

```
setTimeout(function() {  
    console.log("sss");  
}, 1000); // 在此处添加时间参数，以毫秒为单位
```

```
function goFirst(callback) {  
    console.log("aaa");  
    callback();  
}
```

```
function goSecond() {  
    console.log("bbb");  
}
```

```
goFirst(goSecond);
```

aaa

bbb

sss // 延迟1秒后输出

```
setTimeout(function() {  
    console.log("aaa");  
    setTimeout(function() {  
        console.log("bbb");  
        setTimeout(function() {  
            console.log("ccc");  
        }, 5000);  
    }, 5000);  
}, 5000);
```

"aaa"、"bbb"、"ccc"

假设我们有来自第三方的代码：

- 我们发起了多个数据库请求。
- 回调可能导致我们失去对代码的控制。
- 回调可能无法完成我们期望代码执行的任务。
- 导致我们失去对代码的信任。

# Promises

Promise 对象表示异步操作的最终完成（或失败）以及其结果值。（MDN 文档）

- Promise（承诺）用于处理异步操作。
- 异步意味着我们依赖于用户或其他任务的完成。
- 例如，如果用户正在浏览并在您的网站上选择要购买的商品。
- Promise 可以是挂起状态、已完成状态和已拒绝状态。
- Promise 对象是不可变的。

```
const link = "https://api.github.com/users/parthadc9";

const response = fetch(link);

response.then(function(data){
  console.log(data);
})
```

The screenshot displays a web browser's developer console with the following elements:

- Code Editor:** Shows a script with 13 lines. Lines 5, 7, 8, and 11 are highlighted with blue arrows, indicating the current execution point. The code is:

```
1
2
3 <script>
4
5   const link =
6     "https://api.github.com/users/parthadc9";
7   const response =
8     fetch(link);
9   response.then(function
10    (data){
11     console.log(data);
12   })
13 </script>
```
- Scope Chain Panel:** Located on the right, it shows the current execution context. The "Watch Expressions" section is empty. The "Global Lexical Environment" section shows the following variables:
  - link:** "https://api.github.com/us" (String)
  - response:** Promise (Object)
    - status:** "pending" (String)
  - Promise Prototype:** A list of methods: `catch(rejectionHandler)`, `constructor: function()`, `finally()`, `then(resolvedHandler, rejectionHar`, and `Symbol(Symbol.toStringTag): "Pro`.
  - Object Prototype:** A placeholder for the object's prototype.
  - Global Variables:** A placeholder for global variables.

```
const response = fetch();

response.then(function(){
  //remember to return
})
.then (function(){ // OR .then (returnedvalue => next function())
  //remember to return
})
```

For Each

```
const numbers = [1, 2, 3, 4, 5];

for (i = 0; i < numbers.length; i++) {
  console.log(numbers[i]);
}

numbers.forEach(function(number) {
  console.log(number);
});

numbers.forEach(number => console.log(number));
```

```
1  const numbers = [1, 2, 3, 4, 5];
2  numbers.forEach((number, index, array) => {
3    console.log('Index: ' + index + ' Value: ' + number);
4  });
```

PROBLEMS	OUTPUT	DEBUG CONSOLE	TERMINAL
----------	--------	---------------	----------

```
ja21121@C02DWCVPML7H Teaching 2023 % node foreach.js
Debugger attached.
Index: 0 Value: 1
Index: 1 Value: 2
Index: 2 Value: 3
Index: 3 Value: 4
Index: 4 Value: 5
```

## 8.5.1 Exercises

[https://docs.google.com/document/d/1hLM38b6Y9xhXtD-CAbp6fOr9IMm8P-HfMHfXNQHD\\_zA/mobilebasic](https://docs.google.com/document/d/1hLM38b6Y9xhXtD-CAbp6fOr9IMm8P-HfMHfXNQHD_zA/mobilebasic)

CSS

```
.env
MOVIES_API_KEY=39847290dee6c68d3e6c90b9b90ee4ae

sever
const http = require('http');
const fs = require('fs');
const path = require('path');
require('dotenv').config();

const PORT = process.env.PORT || 3000;

const MOVIES_API_KEY = process.env.MOVIES_API_KEY;

const SEARCH_API = `https://api.themoviedb.org/3/search/movie?api_key=
const API_URL_HOME = `https://api.themoviedb.org/3/discover/movie?api_

const server = http.createServer((req, res) => {
  let filePath = path.join(__dirname, req.url === '/' ? 'index.html'
  let extname = path.extname(filePath);
  let contentType = 'text/html';

  switch (extname) {
    case '.js':
      contentType = 'text/javascript';
      break;
    case '.css':
      contentType = 'text/css';
      break;
    case '.json':
      contentType = 'application/json';
      break;
```

```

    case '.png':
      contentType = 'image/png';
      break;
    case '.jpg':
      contentType = 'image/jpg';
      break;
  }

  fs.readFile(filePath, 'utf8', (err, content) => {
    if (err) {
      if (err.code === 'ENOENT') {
        // Page not found
        res.writeHead(404, { 'Content-Type': 'text/html' });
        res.end('<h1>404 Not Found</h1><p>The page you are loo
      } else {
        // Server error
        res.writeHead(500);
        res.end(`Server Error: ${err.code}`);
      }
    } else {
      // Success
      content = content.replace('{{API_URL_HOME}}', API_URL_HOME);
      content = content.replace('{{SEARCH_API}}', SEARCH_API);

      res.writeHead(200, { 'Content-Type': contentType });
      res.end(content, 'utf8');
    }
  });
});

server.listen(PORT, () => {
  console.log(`Server is running on port ${PORT}`);
});

```

CSS

```
const IMG_PATH = 'https://image.tmdb.org/t/p/w500'
```

```

// const SEARCH_API = 'https://api.themoviedb.org/3/search/movie?api_k
// const API_URL_HOME = 'https://api.themoviedb.org/3/discover/movie?a
//             include_adult=false&include_video=false&language

const SEARCH_API = "{{SEARCH_API}}";
const API_URL_HOME = "{{API_URL_HOME}}";

const main = document.getElementById("main");
const form = document.getElementById("form");
const search = document.getElementById("search");

async function getMovies(url) {

    const apiRes = await fetch(url);
    const resJson = await apiRes.json();

    if (!apiRes.ok) {
        console.error("Error fetching movies: ", apiRes.statusText);
    }
    showMovies(resJson.results);
}

function showMovies(movies) {
    main.innerHTML = ''

    movies.forEach((movie) => {
        const { title, poster_path, vote_average, overview } = movie

        const movieEl = document.createElement('div')
        movieEl.classList.add('movie')

        movieEl.innerHTML = `
            
            <div class="movie-info">
            <h3>${title}</h3>
            <span class="${getClassByRate(vote_average)}">${vote_average}</div>
            <div class="overview">

```



```

        <h3>Overview</h3>
        ${overview}
    </div>
    `
    main.appendChild(movieEl)
  })
}

function getClassByRate(vote) {
  if (vote >= 8) {
    return ".movie-info blue";
  } else if (vote >= 5) {
    return ".movie-info black"
  } else return ".movie-info orange";
}

// Get initial movies
getMovies(API_URL_HOME);

form.addEventListener('submit', (e) => {
  e.preventDefault()

  const searchTerm = search.value // we create a var with the search

  if(searchTerm && searchTerm !== '') { // and if the term exists
    getMovies(SEARCH_API + searchTerm);

    search.value = ''
  } else {
    window.location.reload();
  }
})

```