Javascript Basic

Q: Closures（闭包）

```
闭包就是能够读取其他函数内部变量的函数，或者子函数在外调用，
子函数所在的父函数的作用域不会被释放。

function init() {
  var name = "Mozilla"; // name 是 init 创建的局部变量
  function displayName() {
    // displayName() 是内部函数，它创建了一个闭包
    console.log(name); // 使用在父函数中声明的变量
  }
  displayName();
}
init();
```

Callback Hell

```
Use promise, generator, async/await to solve
```

Promise

```
一个promise 可能有三种状态：等待（pending）、已完成（resolved，又称fulfilled）、
已拒绝（rejected）。

Promise.all、Promise.race、Promise.any的区别
all： 成功的时候返回的是一个结果数组，而失败的时候则返回最先被reject失败状态的值。
The Promise.all() static method takes an iterable of promises as input and
returns a single Promise.
This returned promise fulfills when all of the input's promises fulfill
(including when an empty iterable is passed),
with an array of the fulfillment values. It rejects when any of the input's
promises rejects, with this first rejection reason.

race： 哪个结果获得的快，就返回那个结果，不管结果本身是成功状态还是失败状态。
any： 返回最快的成功结果，如果全部失败就返回失败结果。


const myFirstPromise = new Promise((resolve, reject) => {
  // 当异步操作成功时，我们调用 resolve(...)，当其失败时，调用 reject(...)。
  // 在这个例子中，我们使用 setTimeout(...) 来模拟异步代码。
  // 在实际情况中，你可能会使用类似 XHR 或 HTML API 等。
  setTimeout(() => {
```

```
    resolve("成功！"); // 耶！一切顺利！
  }, 250);
});

myFirstPromise.then((successMessage) => {
  // successMessage 是我们在上面的 resolve(...) 函数中传入的任何内容。
  // 它不一定是字符串，但如果它只是一个成功的消息，那么它大概率是字符串。
  console.log(`耶！${successMessage}`);
});
```

addEventListener

```
addEventListener(type, listener)
addEventListener(type, listener, options)
addEventListener(type, listener, useCapture)

my_element.addEventListener("click", function (e) {
  console.log(this.className); // logs the className of my_element
  console.log(e.currentTarget === this); // logs `true`
});
```

前端事件流

```
事件捕获阶段
处于目标阶段
事件冒泡阶段

先捕获，后冒泡
```

What did `new` do?

```
new 操作符新建了一个空对象，
这个对象原型指向构造函数的prototype，
执行构造函数后返回这个对象。

// （1）首先创建了一个新的空对象
// （2）设置原型，将对象的原型设置为函数的 prototype 对象。
// （3）让函数的 this 指向这个对象，执行构造函数的代码（为这个新对象添加属性）
// （4）判断函数的返回值类型，如果是值类型，返回创建的对象。如果是引用类型，就返回这个引
用类型的对象。
```

`this`

this 是执行上下文中的一个属性，它指向最后一次调用这个方法的对象。
1.第一种是函数调用模式，当一个函数不是一个对象的属性时，直接作为函数来调用时，this 指向全局对象。

2.第二种是方法调用模式，如果一个函数作为一个对象的方法来调用时，this 指向这个对象。

3.第三种是构造器调用模式，如果一个函数用 new 调用时，函数执行前会新创建一个对象，this 指向这个新创建的对象。

4.第四种是 apply 、 call 和 bind 调用模式，这三个方法都可以显示的指定调用函数的 this 指向。

The value of `this` depends on in which context it appears: function, class, or global.

通过apply 和call 改变函数的this 指向，他们两个函数的第一个参数都是一样的表示要改变指向的那个对象，第二个参数，apply 是数组，而call 则是arg1,arg2...这种形式。通过bind 改变this 作用域会返回一个新的函数，这个函数不会马上执行。

```
let obj = {
  name: this.name,
  objAge: this.age,
  myLove: function (fm, t) {
    console.log(this.name + "年龄" + this.age, "来自" + fm + "去往" + t);
  },
};
let obj1 = {
  name: "huang",
  age: 22,
};
obj.myLove.call(obj1, "达州", "成都"); //huang年龄22来自达州去往成都
obj.myLove.apply(obj1, ["达州", "成都"]); //huang年龄22来自达州去往成都

obj.myLove.bind(obj1, "达州", "成都")(); // (注意前面有个调用)huang年龄22来自达州去往成都
```

异步加载JS 的方法

1. defer：defer 特性告诉浏览器不要等待脚本。相反，浏览器将继续处理 HTML，构建 DOM。脚本会“在后台”下载，然后等 DOM 构建完成后，脚本才会执行。
<script defer src="xxx"/>

2. async: async 脚本会在后台加载，并在加载就绪时运行。DOM 和其他脚本不会等待它们，它们也不会等待其它的东西。
async 脚本就是一个会在加载完成时执行的完全独立的脚本。
<script async src="https://google-analytics.com/analytics.js"></script>

同时存在defer 和async，那么defer 的优先级比较高，脚本将在页面完成时执行。

```
创建script 标签，插入到DOM 中

3. 动态创建 DOM 方式
const script = document.createElement('script')
script.src = 'xxxx'

4. 使用 setTimeout 延迟方法
```

Js Garbage Collection

```
必要性：
由于字符串、对象和数组没有固定大小，所有当他们的大小已知时，才能对他
们进行动态的存储分配。JavaScript 程序每次创建字符串、数组或对象时，解释器都必
须分配内存来存储那个实体。只要像这样动态地分配了内存，最终都要释放这些内存以
便他们能够被再用，否则，JavaScript 的解释器将会消耗完系统中所有可用的内存，造
成系统崩溃。

JavaScript 的解释器可以检测到何时程序不再使用一个对象
了，当他确定了一个对象是无用的时候，他就知道不再需要这个对象，可以把它所占用
的内存释放掉了。

There are mainly 2 methods:

1. Reference-counting garbage collection:
(Note: No modern JavaScript engine uses reference-counting for garbage
collection anymore.)
引用计数法的意思就是每个值没引用的次数，
当声明了一个变量，并用一个引用类型的值赋值给改变量，则这个值的引用次数为1,；
相反的，如果包含了对这个值引用的变量又取得了另外一个值，则原先的引用值引用次
数就减1，当这个值的引用次数为0 的时候，说明没有办法再访问这个值了，因此就把
所占的内存给回收进来，这样垃圾收集器再次运行的时候，就会释放引用次数为0 的这
些值。
用引用计数法会存在内存泄露，下面来看原因：
function problem() {
var objA = new Object();
var objB = new Object();
objA.someOtherObject = objB;
objB.anotherObject = objA;
}
在这个例子里面，objA 和objB 通过各自的属性相互引用，这样的话，两个对象的引用
次数都为2，在采用引用计数的策略中，由于函数执行之后，这两个对象都离开了作用
域，函数执行完成之后，因为计数不为0，这样的相互引用如果大量存在就会导致内存
泄露。

2. Mark-and-sweep algorithm:
This algorithm reduces the definition of "an object is no longer needed" to
"an object is unreachable".

This algorithm assumes the knowledge of a set of objects called roots.
```

```
In JavaScript, the root is the global object.
Periodically, the garbage collector will start from these roots, find all
objects that are referenced from these roots,
then all objects referenced from these, etc.
Starting from the roots, the garbage collector will thus find all reachable
objects and collect all non-reachable objects.

3. generational garbage collection (in V8 engine)
新创建的对象被放置在新生代(Young Generation)中，如果这些对象能够长期存活，则被移动到老
生代(Old Generation)。垃圾回收器会更加频繁地清理新生代中的对象，
而老生代的对象因为生命周期较长，则不需要频繁清理。这种方式大大提高了垃圾回收的效率。

生代采用了标记清除法和标记压缩法。标记清除法首先会对内存中存活的对象进行标记，标记结束后清
除掉那些没有标记的对象。
由于标记清除后会造成很多的内存碎片，不便于后面的内存分配。所以了解决内存碎片的问题引入了标
记压缩法。
```

eval 是做什么的

```
它的功能是将对应的字符串解析成JS 并执行，应该避免使用JS，因为非常消耗性能（2
次，一次解析成JS，一次执行）
console.log(eval('2 + 2'));
// Expected output: 4

console.log(eval(new String('2 + 2')));
// Expected output: 2 + 2

console.log(eval('2 + 2') === eval('4'));
// Expected output: true

console.log(eval('2 + 2') === eval(new String('2 + 2')));
// Expected output: false
```

CommonJs, AMD (Asynchronous Module Definition), CMD(Common Module Definition), ES6

```
commonjs:

//payments.js
var customerStore = require('store/customer'); // import module
//store/customer.js
function customerStore(){
    return customers.get('store);
}
modules.exports = customerStore;

AMD:
AMD was born as CommonJS wasn’t suited for the browsers early on. As the
name implies, it supports asynchronous module loading.
```

```
Used require.js to achieven

define(['module1', ',module2'], function(module1, module2) {
  console.log(module1.setName());
});
The function is called only when the requested modules are finished
loading.

CMD:
它与 AMD（Asynchronous Module Definition）类似，但在模块加载和执行时有不同的理念。


ES6:
//------ lib.js ------
export const sqrt = Math.sqrt;
export function square(x) {
    return x * x;
}
export function diag(x, y) {
    return sqrt(square(x) + square(y));
}
//------ main.js ------
import { square, diag } from 'lib';
console.log(square(11)); // 121
console.log(diag(4, 3)); // 5
```

实现一个once 函数，传入函数参数只执行一次

```
var once = function(fn) {
    let flag = false
    return function(...args){
        if (flag) {
            return undefined
        }
        flag = true
        return fn(...args)
    }
};
/**
 * let fn = (a,b,c) => (a + b + c)
 * let onceFn = once(fn)
 *
 * onceFn(1,2,3); // 6
 * onceFn(2,3,6); // returns undefined without calling fn
 */
```

如何实现一个私有变量，用getName 方法可以访问，不能直接访问

```
1. closure
function createPerson(name) {
  let _name = name; // 私有变量
  return {
    getName: function() {
      return _name;
    }
  };
}

const person = createPerson('Alice');
console.log(person.getName()); // 输出: Alice
console.log(person._name);     // 输出: undefined

2. 使用 ES6 私有字段（Private Fields）
class Person {
  #name; // 私有字段

  constructor(name) {
    this.#name = name;
  }

  getName() {
    return this.#name;
  }
}

const person = new Person('David');
console.log(person.getName()); // 输出: David
console.log(person.#name);     // 语法错误，无法直接访问私有字段
```

==和===、以及Object.is 的区别

```
console.log(undefined == null); // Outputs: true

console.log(undefined === null); // Outputs: false

console.log(typeof undefined); // Outputs: "undefined"
console.log(typeof null);      // Outputs: "object"

Object.is 主要的区别就是+0！=-0 而NaN==NaN
```

setTimeout vs setInterval

```
setTimeout 用于在指定的延迟时间后执行一次函数。
setTimeout(function, delay, arg1, arg2, ...);
```

```
setInterval 用于按照指定的时间间隔周期性地执行函数，直到被明确取消。
setInterval(function, interval, arg1, arg2, ...);

取消方式：
setTimeout：使用 clearTimeout(timeoutId) 取消。
setInterval：使用 clearInterval(intervalId) 取消。
```

Js 是单线程的，那么它是怎么处理异步操作的？Eventloop

JavaScript 是一门单线程语言，即在同一时间只能执行一个任务。然而，它通过事件循环（Event Loop）机制有效地处理异步操作，如网络请求、定时器和用户交互等，从而避免阻塞主线程。

事件循环机制

事件循环是 JavaScript 处理异步任务的核心。它协调调用栈（Call Stack）和任务队列（Task Queue）之间的关系，确保异步任务在适当的时机执行。

主要组件：

1. 调用栈（Call Stack）：存储当前正在执行的函数。当函数被调用时，会被压入栈中；执行完毕后，从栈中弹出。

2. Web APIs：由浏览器提供的功能，如 setTimeout、HTTP 请求和 DOM 事件等。当执行异步操作时，这些任务会被交由浏览器处理，而非阻塞主线程。

3. 任务队列（Task Queue）：

  宏任务队列（Macro Task Queue）：包含如 setTimeout、setInterval 和 I/O 操作等任务。
  微任务队列（Micro Task Queue）：包含如 Promise 回调和 MutationObserver 等任务。微任务的优先级高于宏任务。

执行流程：

同步代码直接在调用栈中执行。
异步操作由 Web APIs 处理，完成后将回调函数放入相应的任务队列。
事件循环首先检查调用栈是否为空，若为空，则依次处理微任务队列中的所有任务，然后再处理宏任务队列中的第一个任务。

How the Event Loop Works:

When JavaScript code runs, synchronous tasks are executed directly on the call stack.

Asynchronous tasks (e.g., setTimeout, Promises, I/O operations) are offloaded to the browser's Web APIs or Node.js APIs.
Once these tasks complete, their callbacks are placed into the task queue.

The event loop checks the call stack. If it's empty, it takes the first callback from the task queue and

```
pushes it onto the call stack for execution.

Microtasks and Macrotasks:

Tasks in JavaScript are categorized into microtasks and macrotasks, each
with its own queue:

  Macrotasks: Include events like setTimeout, setInterval, and I/O
operations.

  Microtasks: Include Promises' .then() handlers and process.nextTick() in
Node.js.
```

Promise 代码的执行顺序

```
new Promise(function (resolve, reject) {
  console.log(2);
  resolve();
})
  .then(function () {
    console.log(3);
  })
  .then(function () {
    console.log(4);
  });

process.nextTick(function () {
  console.log(5);
});

console.log(6);

//输出2,6,5,3,4,1

同步任务：立即执行主线程上的同步代码。
  console.log(2)：输出 2。
  console.log(6)：输出 6。

微任务（Microtasks）：在当前事件循环结束前执行的任务，包括 process.nextTick 和
Promise 的回调。
  process.nextTick 的回调：输出 5。
  Promise 的第一个 then 回调：输出 3。
  Promise 的第二个 then 回调：输出 4。

宏任务（Macrotasks）：在下一个事件循环中执行的任务，如 setTimeout。

setTimeout 的回调：输出 1。

script(主程序代码)—>process.nextTick—>Promises...——>setTimeout——>setInterval
——>setImmediate——> I/O——>UI rendering
```

```
sync task => nextTicket => promise.then()/promise resolve => setTimeout
```

Deep Copy

```
1. Using JSON.parse() and JSON.stringify():

const original = { a: 1, b: { c: 2 } };
const copy = JSON.parse(JSON.stringify(original));

2. Using Recursive Function:

function deepClone(obj, hash = new WeakMap()) {
  if (obj === null || typeof obj !== 'object') return obj;
  if (hash.has(obj)) return hash.get(obj); // Handle circular references

  const clone = Array.isArray(obj) ? [] : {};
  hash.set(obj, clone);

  for (const key in obj) {
    if (obj.hasOwnProperty(key)) {
      clone[key] = deepClone(obj[key], hash);
    }
  }
  return clone;
}

const original = { a: 1, b: { c: 2 } };
const copy = deepClone(original);
```

数组去重

```
Array.from(new Set(array))
```

How to check if it's an array?

```
DON't use typeof => will return object

Use Array.isArray() or instanceof

const arr = [1, 2, 3];
console.log(arr instanceof Array);
```

JS 实现跨域

JSONP：适用于仅需 GET 请求且后端支持的简单场景。

CORS：适用于后端可配置响应头的场景，支持多种请求方法。 (Access-Control-Allow-Origin)

代理服务器：适用于前后端分离且需要统一代理的场景。

服务器端设置跨域头：适用于后端代码可修改的项目。

Webpack

Webpack 是一个用于现代 JavaScript 应用程序的静态模块打包工具。
它的主要功能是将项目中的各种资源（如 JavaScript、CSS、图片等）视为模块，构建一个依赖图，
然后将这些模块打包成一个或多个bundle，以供浏览器使用。

how to reduce bundle size?
1. 启用生产模式（Production Mode）： 确保在生产环境中使用 Webpack 的生产模式，它会自动进行代码压缩和优化。
webpack --mode production

2. 代码分割（Code Splitting）： 将代码拆分为更小的块，实现按需加载，减少初始加载时间。可以在 Webpack 配置中使用 SplitChunksPlugin

```
module.exports = {
  // ...
  optimization: {
    splitChunks: {
      chunks: 'all',
    },
  },
};
```

3. 分析打包内容： 使用如 Webpack Bundle Analyzer 等工具，可视化并了解打包内容，识别需要优化或移除的大型模块或依赖。

```
const { BundleAnalyzerPlugin } = require('webpack-bundle-analyzer');

module.exports = {
  // ...
  plugins: [
    new BundleAnalyzerPlugin(),
  ],
};
```

JS 中继承实现的几种方式

1. 原型链继承

每个 JavaScript 对象都有一个内部链接，称为 [[Prototype]]，指向另一个对象。当访问对象的属性或方法时，JavaScript 会沿着原型链向上查找，
直到找到相应的属性或方法，或到达原型链的顶端（即 null）。

```
function Animal(name) {
  this.name = name;
}

Animal.prototype.eat = function() {
  console.log(this.name + ' is eating.');
};

function Dog(name, breed) {
  Animal.call(this, name); // 调用父类构造函数
  this.breed = breed;
}

Dog.prototype = Object.create(Animal.prototype); // 设置原型链
Dog.prototype.constructor = Dog; // 修正构造函数指向

Dog.prototype.bark = function() {
  console.log(this.name + ' is barking.');
};

const dog = new Dog('Buddy', 'Labrador');
dog.eat();  // 输出：Buddy is eating.
dog.bark(); // 输出：Buddy is barking.
```

2. ES6 类继承

ES6 引入了 class 语法，使继承的实现更加直观。class 语法是基于原型的语法糖，底层机制仍然是原型链继承。

```
class Animal {
  constructor(name) {
    this.name = name;
  }

  eat() {
    console.log(this.name + ' is eating.');
  }
}

class Dog extends Animal {
  constructor(name, breed) {
    super(name); // 调用父类构造函数
    this.breed = breed;
  }

  bark() {
    console.log(this.name + ' is barking.');
  }
}
```

```
const dog = new Dog('Buddy', 'Labrador');
dog.eat();  // 输出：Buddy is eating.
dog.bark(); // 输出：Buddy is barking.
```

JS 原型链

在 JavaScript 中，原型链是实现对象继承的核心机制。每个对象都有一个内部属性 [[Prototype]]，指向另一个对象，即其原型。当访问对象的某个属性或方法时，JavaScript 会首先在该对象自身查找；如果未找到，则沿着原型链向上查找，直到找到该属性或方法，或到达原型链的顶端（即 null）

```
Object.getPrototypeOf()
```

let, var, const

提起这三个最明显的区别是var 声明的变量是全局或者整个函数块的，而let,const 声明的变量是块级的变量，

```
var 声明的变量存在变量提升(hoisting)，let,const 不存在，
console.log(a); // 输出：undefined
var a = 5;

console.log(b); // 报错：b 未定义
let b = 10;
```

let 声明的变量允许重新赋值，const 不允许。

ES6 箭头函数的特性

箭头函数与普通函数的区别在于：
1、箭头函数没有this，所以需要通过查找作用域链来确定this 的值，这就意味着如果箭头函数被非箭头函数包含，this 绑定的就是最近一层非箭头函数的this，
2、箭头函数没有自己的arguments 对象，但是可以访问外围函数的arguments 对象
3、不能通过new 关键字调用，同样也没有new.target 值和原型

Js symbol

在 JavaScript 中，Symbol 是一种独特且不可变的基本数据类型，主要用于为对象添加唯一的属性键，避免属性名冲突。

```
const sym1 = Symbol('desc');
const sym2 = Symbol('desc');
console.log(sym1 === sym2); // 输出：false
```

```
const mySymbol = Symbol('uniqueKey');
const obj = {
  [mySymbol]: 'value'
};
console.log(obj[mySymbol]); // 输出：value
```

Babel

```
Babel 是一个工具链，主要用于将采用 ECMAScript 2015+ 语法编写的代码转换为向后兼容的 JavaScript 语法，
以便能够运行在当前和旧版本的浏览器或其他环境中。下面列出的是 Babel 能为你做的事情：

- 语法转换: 将最新的 JavaScript 语法转换为旧版本兼容的语法。
- Polyfill: 通过 Polyfill 方式在目标环境中添加缺失的功能（通过引入第三方 polyfill 模块，例如 core-js）
- 源码转换（codemods）:对源代码进行批量修改或重构 => 开发者可以高效地对大型代码库进行批量修改，提升开发效率，减少人为错误。
 (for example, want to change all `var` to `let`)
```

Js proxy

```
JavaScript 的 Proxy 对象 是 ECMAScript 6（ES6）引入的一项强大特性，
允许开发者创建一个代理对象来拦截并自定义对另一个对象的基本操作，如属性访问、赋值、枚举、函数调用等。

const proxy = new Proxy(target, handler);
```

Js stack vs heap

```
栈：原始数据类型（Undefined、Null、Boolean、Number、String）
堆：引用数据类型（对象、数组和函数）

堆和栈的概念存在于数据结构中和操作系统内存中。

在数据结构中，栈中数据的存取方式为先进后出。而堆是一个优先队列，是按优先级来进行排序的，优先级可以按照大小来规定。完全
二叉树是堆的一种实现方式。

在操作系统中，内存被分为栈区和堆区。

栈区内存由编译器自动分配释放，存放函数的参数值，局部变量的值等。其操作方式类似于数据结构中的栈。

堆区内存一般由程序员分配释放，若程序员不释放，程序结束时可能由垃圾回收机制回收。
```

内部属性 [[Class]] 是什么？

所有 typeof 返回值为 "object" 的对象（如数组）都包含一个内部属性 [[Class]]（我们可以把它看作一个内部的分类，而非
传统的面向对象意义上的类）。这个属性无法直接访问，一般通过
Object.prototype.toString(..) 来查看。例如：

```
Object.prototype.toString.call( [1,2,3] );
// "[object Array]"

Object.prototype.toString.call( /regex-literal/i );
// "[object RegExp]"
```

如何判断一个对象是否属于某个类？

第一种方式是使用 A instanceof B 运算符来判断B构造函数的 prototype 属性是否出现在A对象的原型链中的任何位置。

第二种方式，如果需要判断的是某个内置的引用类型的话，可以使用
Object.prototype.toString() 方法来打印对象的
[[Class]] 属性来进行判断。

Ajax 是什么?

2005 年 2 月，AJAX 这个词第一次正式提出，它是 Asynchronous JavaScript and XML 的缩写，
指的是通过 JavaScript 的 异步通信，从服务器获取 XML 文档从中提取数据，再更新当前网页的对应部分，而不用刷新整个网页。

具体来说，AJAX 包括以下几个步骤。

1.创建 XMLHttpRequest 对象，也就是创建一个异步调用对象
2.创建一个新的 HTTP 请求，并指定该 HTTP 请求的方法、URL 及验证信息
3.设置响应 HTTP 请求状态变化的函数
4.发送 HTTP 请求
5.获取异步调用返回的数据
6.使用 JavaScript 和 DOM 实现局部刷新

如何判断当前脚本运行在浏览器还是 node 环境中？（阿里）

```
this === window ? 'browser' : 'node';
```

通过判断 Global 对象是否为 window，如果不为 window，当前脚本没有运行在浏览器中。

什么是“前端路由”？

（1）什么是前端路由？

前端路由就是把不同路由对应不同的内容或页面的任务交给前端来做，之前是通过服务端根据 url 的不同返回不同的页面实现的。

（2）什么时候使用前端路由？

在单页面应用，大部分页面结构不变，只改变部分内容的使用

（3）前端路由有什么优点和缺点？

优点：用户体验好，不需要每次都从服务器全部获取，快速展现给用户

缺点：单页面无法记住之前滚动的位置，无法在前进，后退的时候记住滚动的位置

前端路由一共有两种实现方式，一种是通过 hash 的方式，一种是通过使用 pushState 的方式。

如何判断一个对象是否为空对象？

```
Object.keys(obj).length === 0
```

Difference for output

```
for(var i=0;i<5;i++){
  setTimeout(function(){ console.log(i); },1000);
}

console.log(i)

// Output: 5 5 5 5 5

let i;
for (i = 0; i < 5; i++) {
  setTimeout(function() {
    console.log(i);
  }, 1000);
}
console.log(i); // 5

//Output: 0 1 2 3 4

Or use closure:
```

```
for (var i = 0; i < 5; i++) {
  (function(i) {
    setTimeout(function() {
      console.log(i);
    }, i * 1000);
  })(i);
}

// Output: 0 1 2 3 4
```

What structure is an object made of in Javascript?

```
an object is a collection of key-value pairs where keys are strings or
symbols,
and values can be any data type, including other objects.

Properties
  Data Properties: These are key-value pairs where the key is a string or
symbol, and the value is any valid JavaScript data type.

  Accessor Properties: These are properties defined by getter and setter
functions instead of a value.

Prototype Chain:
  ach object has an internal link to another object called its prototype.
```