> `JavaScript` 进阶问题列表

进阶问题

1. 输出是什么?

```js
function sayHi() {
  console.log(name)
  console.log(age)
  var name = 'Lydia'
  let age = 21
}

sayHi()
```

- A: `Lydia` 和 `undefined`
- B: `Lydia` 和 `ReferenceError`
- C: `ReferenceError` 和 `21`
- D: `undefined` 和 `ReferenceError`

▼ 答案

答案: D

在函数内部，我们首先通过 `var` 关键字声明了 `name` 变量。这意味着变量被提升了（内存空间在创建阶段就被设置好了），直到程序运行到定义变量位置之前默认值都是 `undefined`。因为当我们打印 `name` 变量时还没有执行到定义变量的位置，因此变量的值保持为 `undefined`。

通过 `let` 和 `const` 关键字声明的变量也会提升，但是和 `var` 不同，它们不会被*初始化*。在我们声明（初始化）之前是不能访问它们的。这个行为被称之为暂时性死区。当我们试图在声明之前访问它们时，JavaScript 将会抛出一个 `ReferenceError` 错误。

---

2. 输出是什么?

```js
for (var i = 0; i < 3; i++) {
  setTimeout(() => console.log(i), 1)
}

for (let i = 0; i < 3; i++) {
  setTimeout(() => console.log(i), 1)
}
```

- A: `0 1 2` 和 `0 1 2`
- B: `0 1 2` 和 `3 3 3`
- C: `3 3 3` 和 `0 1 2`

▼ 答案

答案: C

由于 JavaScript 的事件循环，`setTimeout` 回调会在*遍历结束后*才执行。因为在第一个遍历中遍历 `i` 是通过 `var` 关键字声明的，所以这个值是全局作用域下的。在遍历过程中，我们通过一元操作符 `++` 来每次递增 `i` 的值。当 `setTimeout` 回调执行的时候，`i` 的值等于 3。

在第二个遍历中，遍历 `i` 是通过 `let` 关键字声明的：通过 `let` 和 `const` 关键字声明的变量是拥有块级作用域（指的是任何在 {} 中的内容）。在每次的遍历过程中，`i` 都有一个新值，并且每个值都在循环内的作用域中。

---

3. 输出是什么?

```js
const shape = {
  radius: 10,
  diameter() {
    return this.radius * 2
  },
  perimeter: () => 2 * Math.PI * this.radius
}

shape.diameter()
shape.perimeter()
```

- A: `20` and `62.83185307179586`
- B: `20` and `NaN`

- C: `20` and `63`
- D: `NaN` and `63`

▼ 答案

答案: B

注意 `diameter` 的值是一个常规函数，但是 `perimeter` 的值是一个箭头函数。

对于箭头函数， `this` 关键字指向的是它当前周围作用域（简单来说是包含箭头函数的常规函数，如果没有常规函数的话就是全局对象），这个行为和常规函数不同。这意味着当我们调用 `perimeter` 时， `this` 不是指向 `shape` 对象，而是它的周围作用域（在例子中是 `window` ）。

在 `window` 中没有 `radius` 这个属性，因此返回 `undefined` 。

---

4. 输出是什么?

```js
+true;
!"Lydia";
```

- A: `1` and `false`
- B: `false` and `NaN`
- C: `false` and `false`

▼ 答案

答案: A

一元操作符加号尝试将 bool 转为 number。 `true` 转换为 number 的话为 `1` ， `false` 为 `0` 。

字符串 `'Lydia'` 是一个真值，真值取反那么就返回 `false` 。

---

5. 哪一个是无效的?

```js
const bird = {
  size: 'small'
}
```

```
const mouse = {
  name: 'Mickey',
  small: true
}
```

- A: `mouse.bird.size`
- B: `mouse[bird.size]`
- C: `mouse[bird["size"]]`
- D: All of them are valid

▼ **答案**

答案: A

在 JavaScript 中，所有对象的 keys 都是字符串（除非对象是 Symbol）。尽管我们可能不会定义它们为字符串，但它们在底层总会被转换为字符串。

当我们使用括号语法时（[]），JavaScript 会解释（或者 unboxes）语句。它首先看到第一个开始括号 `[` 并继续前进直到找到结束括号 `]`。只有这样，它才会计算语句的值。

`mouse[bird.size]`：首先计算 `bird.size`，这会得到 `small`。`mouse["small"]` 返回 `true`。

然后使用点语法的话，上面这一切都不会发生。`mouse` 没有 `bird` 这个 key，这也就意味着 `mouse.bird` 是 `undefined`。然后当我们使用点语法 `mouse.bird.size` 时，因为 `mouse.bird` 是 `undefined`，这也就变成了 `undefined.size`。这个行为是无效的，并且会抛出一个错误类似 `Cannot read property "size" of undefined`。

---

6. 输出是什么?

```js
let c = { greeting: 'Hey!' }
let d

d = c
c.greeting = 'Hello'
console.log(d.greeting)
```

- A: `Hello`
- B: `undefined`

- C: `ReferenceError`
- D: `TypeError`

▼ 答案

答案: A

在 JavaScript 中，当设置两个对象彼此相等时，它们会通过*引用*进行交互。

首先，变量 `c` 的值是一个对象。接下来，我们给 `d` 分配了一个和 `c` 对象相同的引用。

因此当我们改变其中一个对象时，其实是改变了所有的对象。

---

7. 输出是什么?

```js
let a = 3
let b = new Number(3)
let c = 3

console.log(a == b)
console.log(a === b)
console.log(b === c)
```

- A: `true` `false` `true`
- B: `false` `false` `true`
- C: `true` `false` `false`
- D: `false` `true` `true`

▼ 答案

答案: C

`new Number()` 是一个内建的函数构造器。虽然它看着像是一个 number，但它实际上并不是一个真实的 number：它有一堆额外的功能并且它是一个对象。

当我们使用 `==` 操作符时，它只会检查两者是否拥有相同的*值*。因为它们的值都是 `3`，因此返回 `true`。

然后，当我们使用 `===` 操作符时，两者的值以及*类型*都应该是相同的。`new Number()` 是一个对象而不是 number，因此返回 `false`。

---

8. 输出是什么?

```js
class Chameleon {
  static colorChange(newColor) {
    this.newColor = newColor
    return this.newColor
  }

  constructor({ newColor = 'green' } = {}) {
    this.newColor = newColor
  }
}

const freddie = new Chameleon({ newColor: 'purple' })
freddie.colorChange('orange')
```

- A: `orange`
- B: `purple`
- C: `green`
- D: `TypeError`

▼ 答案

答案: D

`colorChange` 是一个静态方法。静态方法被设计为只能被创建它们的构造器使用（也就是 `Chameleon`），并且不能传递给实例。因为 `freddie` 是一个实例，静态方法不能被实例使用，因此抛出了 `TypeError` 错误。

---

9. 输出是什么?

```js
let greeting
greetign = {} // Typo!
console.log(greetign)
```

- A: `{}`
- B: `ReferenceError: greetign is not defined`

- C: `undefined`

▼ 答案

答案: A

代码打印出了一个对象，这是因为我们在全局对象上创建了一个空对象！当我们将 `greeting` 写错成 `greetign` 时，JS 解释器实际在上浏览器中将它视为 `global.greetign = {}` （或者 `window.greetign = {}` ）。

为了避免这个为题，我们可以使用 `"use strict"。这能确保当你声明变量时必须赋值。

---

10. 当我们这么做时，会发生什么?

```js
function bark() {
  console.log('Woof!')
}

bark.animal = 'dog'
```

- A: 正常运行!
- B: `SyntaxError` . 你不能通过这种方式给函数增加属性。
- C: `undefined`
- D: `ReferenceError`

▼ 答案

答案: A

这在 JavaScript 中是可以的，因为函数是对象！（除了基本类型之外其他都是对象）

函数是一个特殊的对象。你写的这个代码其实不是一个实际的函数。函数是一个拥有属性的对象，并且属性也可被调用。

---

11. 输出是什么?

```js
function Person(firstName, lastName) {
  this.firstName = firstName;
  this.lastName = lastName;
}
```

```
const member = new Person("Lydia", "Hallie");
Person.getFullName = function () {
  return `${this.firstName} ${this.lastName}`;
}

console.log(member.getFullName());
```

- A: `TypeError`
- B: `SyntaxError`
- C: `Lydia Hallie`
- D: `undefined` `undefined`

▼ 答案

答案: A

你不能像常规对象那样，给构造函数添加属性。如果你想一次性给所有实例添加特性，你应该使用原型。因此本例中，使用如下方式：

```js
Person.prototype.getFullName = function () {
  return `${this.firstName} ${this.lastName}`;
}
```

这才会使 `member.getFullName()` 起作用。为什么这么做有益的？假设我们将这个方法添加到构造函数本身里。也许不是每个 `Person` 实例都需要这个方法。这将浪费大量内存空间，因为它们仍然具有该属性，这将占用每个实例的内存空间。相反，如果我们只将它添加到原型中，那么它只存在于内存中的一个位置，但是所有实例都可以访问它！

---

12. 输出是什么?

```js
function Person(firstName, lastName) {
  this.firstName = firstName
  this.lastName = lastName
}

const lydia = new Person('Lydia', 'Hallie')
const sarah = Person('Sarah', 'Smith')

console.log(lydia)
console.log(sarah)
```

- A: `Person {firstName: "Lydia", lastName: "Hallie"}` and `undefined`
- B: `Person {firstName: "Lydia", lastName: "Hallie"}` and `Person {firstName: "Sarah", lastName: "Smith"}`
- C: `Person {firstName: "Lydia", lastName: "Hallie"}` and `{}`
- D: `Person {firstName: "Lydia", lastName: "Hallie"}` and `ReferenceError`

▼ **答案**

答案: A

对于 `sarah`，我们没有使用 `new` 关键字。当使用 `new` 时，`this` 引用我们创建的空对象。当未使用 `new` 时，`this` 引用的是**全局对象**（global object）。

我们说 `this.firstName` 等于 `"Sarah"`，并且 `this.lastName` 等于 `"Smith"`。实际上我们做的是，定义了 `global.firstName = 'Sarah'` 和 `global.lastName = 'Smith'`。而 `sarah` 本身是 `undefined`。

---

13. 事件传播的三个阶段是什么?

- A: Target > Capturing > Bubbling
- B: Bubbling > Target > Capturing
- C: Target > Bubbling > Capturing
- D: Capturing > Target > Bubbling

▼ **答案**

答案: D

在**捕获**（capturing）阶段中，事件从祖先元素向下传播到目标元素。当事件达到**目标**（target）元素后，**冒泡**（bubbling）才开始。

---

14. 所有对象都有原型。

- A: true
- B: false

▼ **答案**

答案: B

除了**基本对象**（base object），所有对象都有原型。基本对象可以访问一些方法和属性，比如 `.toString`。这就是为什么你可以使用内置的 JavaScript 方法！所有这些方法在原型上都是可用的。虽然 JavaScript 不能直接在对象上找到这些方法，但 JavaScript 会沿着原型链找到它们，以便于你使用。

---

15. 输出是什么?

```js
function sum(a, b) {
  return a + b
}

sum(1, '2')
```

- A: `NaN`
- B: `TypeError`
- C: `"12"`
- D: `3`

▼ **答案**

答案: C

JavaScript 是一种**动态类型语言**：我们不指定某些变量的类型。值可以在你不知道的情况下自动转换成另一种类型，这种类型称为**隐式类型转换**（implicit type coercion）。**Coercion** 是指将一种类型转换为另一种类型。

在本例中，JavaScript 将数字 `1` 转换为字符串，以便函数有意义并返回一个值。在数字类型（`1`）和字符串类型（`'2'`）相加时，该数字被视为字符串。我们可以连接字符串，比如 `"Hello" + "World"`，这里发生的是 `"1" + "2"`，它返回 `"12"`。

---

16. 输出是什么?

```js
let number = 0
console.log(number++)
console.log(++number)
console.log(number)
```

- A: `1` `1` `2`

- B: `1` `2` `2`
- C: `0` `2` `2`
- D: `0` `1` `2`

▼ 答案

答案: C

一元**后自增**运算符 `++`：

1. 返回值（返回 `0`）
2. 值自增（number 现在是 `1`）

一元**前自增**运算符 `++`：

1. 值自增（number 现在是 `2`）
2. 返回值（返回 `2`）

结果是 `0 2 2`.

---

17. 输出是什么?

```js
function getPersonInfo(one, two, three) {
  console.log(one)
  console.log(two)
  console.log(three)
}

const person = 'Lydia'
const age = 21

getPersonInfo`${person} is ${age} years old`
```

- A: `"Lydia"` `21` `["", " is ", " years old"]`
- B: `["", " is ", " years old"]` `"Lydia"` `21`
- C: `"Lydia"` `["", " is ", " years old"]` `21`

▼ 答案

答案: B

如果使用标记模板字面量，第一个参数的值总是包含字符串的数组。其余的参数获取的是传递的表达式的值！

---

18. 输出是什么？

```js
function checkAge(data) {
  if (data === { age: 18 }) {
    console.log('You are an adult!')
  } else if (data == { age: 18 }) {
    console.log('You are still an adult.')
  } else {
    console.log(`Hmm.. You don't have an age I guess`)
  }
}

checkAge({ age: 18 })
```

- A: `You are an adult!`
- B: `You are still an adult.`
- C: `Hmm.. You don't have an age I guess`

▼ 答案

答案: C

在测试相等性时，基本类型通过它们的值（value）进行比较，而对象通过它们的引用（reference）进行比较。JavaScript 检查对象是否具有对内存中相同位置的引用。

题目中我们正在比较的两个对象不是同一个引用：作为参数传递的对象引用的内存位置，与用于判断相等的对象所引用的内存位置并不同。

这也是 `{ age: 18 } === { age: 18 }` 和 `{ age: 18 } == { age: 18 }` 都返回 `false` 的原因。

---

19. 输出是什么？

```js
function getAge(...args) {
  console.log(typeof args)
}
```

```
getAge(21)
```

- A: `"number"`
- B: `"array"`
- C: `"object"`
- D: `"NaN"`

▼ 答案

答案: C

扩展运算符（`...args`）会返回实参组成的数组。而数组是对象，因此 `typeof args` 返回 `"object"`。

---

20. 输出是什么?

```js
function getAge() {
  'use strict'
  age = 21
  console.log(age)
}

getAge()
```

- A: `21`
- B: `undefined`
- C: `ReferenceError`
- D: `TypeError`

▼ 答案

答案: C

使用 `"use strict"`，你可以确保不会意外地声明全局变量。我们从来没有声明变量 `age`，因为我们使用 `"use strict"`，它将抛出一个引用错误。如果我们不使用 `"use strict"`，它就会工作，因为属性 `age` 会被添加到全局对象中了。

---

21. 输出是什么?

```js
const sum = eval('10*10+5')
```

- A: `105`
- B: `"105"`
- C: `TypeError`
- D: `"10*10+5"`

▼ 答案

答案: A

代码以字符串形式传递进来，`eval` 对其求值。如果它是一个表达式，就像本例中那样，它对表达式求值。表达式是 `10 * 10 + 5` 。这将返回数字 `105` 。

---

22. cool_secret 可访问多长时间?

```js
sessionStorage.setItem('cool_secret', 123)
```

- A: 永远，数据不会丢失。
- B: 当用户关掉标签页时。
- C: 当用户关掉整个浏览器，而不只是关掉标签页。
- D: 当用户关闭电脑时。

▼ 答案

答案: B

关闭 tab **标签页** 后，`sessionStorage` 存储的数据才会删除。

如果使用 `localStorage` ，那么数据将永远在那里，除非调用了 `localStorage.clear()` 。

---

23. 输出是什么?

```js
var num = 8
var num = 10

console.log(num)
```

- A: `8`
- B: `10`
- C: `SyntaxError`
- D: `ReferenceError`

▼ 答案

答案: B

使用 `var` 关键字，你可以用相同的名称声明多个变量。然后变量将保存最新的值。

你不能使用 `let` 或 `const` 来实现这一点，因为它们是块作用域的。

---

24. 输出是什么?

```js
const obj = { 1: 'a', 2: 'b', 3: 'c' }
const set = new Set([1, 2, 3, 4, 5])

obj.hasOwnProperty('1')
obj.hasOwnProperty(1)
set.has('1')
set.has(1)
```

- A: `false` `true` `false` `true`
- B: `false` `true` `true` `true`
- C: `true` `true` `false` `true`
- D: `true` `true` `true` `true`

▼ 答案

答案: C

所有对象的键（不包括 Symbol）在底层都是字符串，即使你自己没有将其作为字符串输入。这就是为什么 `obj.hasOwnProperty('1')` 也返回 `true`。

对于集合，它不是这样工作的。在我们的集合中没有 `'1'`：`set.has('1')` 返回 `false`。它有数字类型为 `1`，`set.has(1)` 返回 `true`。

---

25. 输出是什么?

```js
const obj = { a: 'one', b: 'two', a: 'three' }
console.log(obj)
```

- A: `{ a: "one", b: "two" }`
- B: `{ b: "two", a: "three" }`
- C: `{ a: "three", b: "two" }`
- D: `SyntaxError`

▼ 答案

答案: C

如果你有两个名称相同的键，则键会被替换掉。它仍然位于第一个键出现的位置，但是值是最后出现那个键的值。

---

26. JavaScript 全局执行上下文为你做了两件事：全局对象和 this 关键字。

- A: true
- B: false
- C: it depends

▼ 答案

答案: A

基本执行上下文是全局执行上下文：它是代码中随处可访问的内容。

---

27. 输出是什么?

```js
for (let i = 1; i < 5; i++) {
  if (i === 3) continue
  console.log(i)
}
```

- A: `1` `2`
- B: `1` `2` `3`
- C: `1` `2` `4`

- D: `1` `3` `4`

▼ 答案

答案: C

如果某个条件返回 `true`，则 `continue` 语句跳过本次迭代。

---

28. 输出是什么?

```js
String.prototype.giveLydiaPizza = () => {
  return 'Just give Lydia pizza already!'
}

const name = 'Lydia'

name.giveLydiaPizza()
```

- A: `"Just give Lydia pizza already!"`
- B: `TypeError: not a function`
- C: `SyntaxError`
- D: `undefined`

▼ 答案

答案: A

`String` 是内置的构造函数，我们可以向它添加属性。我只是在它的原型中添加了一个方法。基本类型字符串被自动转换为字符串对象，由字符串原型函数生成。因此，所有 string(string 对象)都可以访问该方法!

---

29. 输出是什么?

```js
const a = {}
const b = { key: 'b' }
const c = { key: 'c' }

a[b] = 123
a[c] = 456
```

```
console.log(a[b])
```

- A: `123`
- B: `456`
- C: `undefined`
- D: `ReferenceError`

▼ 答案

答案: B

对象的键被自动转换为字符串。我们试图将一个对象 `b` 设置为对象 `a` 的键，且相应的值为 `123`。

然而，当字符串化一个对象时，它会变成 `"[object Object]"`。因此这里说的是，`a["[object Object]"] = 123`。然后，我们再一次做了同样的事情，`c` 是另外一个对象，这里也有隐式字符串化，于是，`a["[object Object]"] = 456`。

然后，我们打印 `a[b]`，也就是 `a["[object Object]"]`。之前刚设置为 `456`，因此返回的是 `456`。

---

30. 输出是什么?

```js
const foo = () => console.log('First')
const bar = () => setTimeout(() => console.log('Second'))
const baz = () => console.log('Third')

bar()
foo()
baz()
```

- A: `First` `Second` `Third`
- B: `First` `Third` `Second`
- C: `Second` `First` `Third`
- D: `Second` `Third` `First`

▼ 答案

答案: B

我们有一个 `setTimeout` 函数，并首先调用它。然而，它是最后打印日志的。

这是因为在浏览器中，我们不仅有运行时引擎，还有一个叫做 `WebAPI` 的东西。 `WebAPI` 提供了 `setTimeout` 函数，也包含其他的，例如 DOM。

将 *callback* 推送到 WebAPI 后， `setTimeout` 函数本身(但不是回调！)将从栈中弹出。

现在， `foo` 被调用，打印 `"First"` 。

`foo` 从栈中弹出， `baz` 被调用. 打印 `"Third"` 。

WebAPI 不能随时向栈内添加内容。相反，它将回调函数推到名为 *queue* 的地方。

这就是事件循环开始工作的地方。一个**事件循环**查看栈和任务队列。如果栈是空的，它接受队列上的第一个元素并将其推入栈。

`bar` 被调用，打印 `"Second"` ，然后它被栈弹出。

---

31. 当点击按钮时，event.target是什么?

```html
<div onclick="console.log('first div')">
  <div onclick="console.log('second div')">
    <button onclick="console.log('button')">
      Click!
    </button>
  </div>
</div>
```

- A: Outer `div`
- B: Inner `div`
- C: `button`
- D: 一个包含所有嵌套元素的数组。

▼ 答案

答案: C

导致事件的最深嵌套的元素是事件的 target。你可以通过 `event.stopPropagation` 来停止冒泡。

---

32. 当您单击该段落时，日志输出是什么?

```html
<div onclick="console.log('div')">
  <p onclick="console.log('p')">
    Click here!
  </p>
</div>
```

- A: `p` `div`
- B: `div` `p`
- C: `p`
- D: `div`

▼ 答案

答案: A

如果我们点击 `p`，我们会看到两个日志：`p` 和 `div`。在事件传播期间，有三个阶段：捕获、目标和冒泡。默认情况下，事件处理程序在冒泡阶段执行（除非将 `useCapture` 设置为 `true`）。它从嵌套最深的元素向外传播。

---

33. 输出是什么?

```js
const person = { name: 'Lydia' }

function sayHi(age) {
  console.log(`${this.name} is ${age}`)
}

sayHi.call(person, 21)
sayHi.bind(person, 21)
```

- A: `undefined is 21` `Lydia is 21`

- B: `function` `function`
- C: `Lydia is 21` `Lydia is 21`
- D: `Lydia is 21` `function`

▼ 答案

答案: D

使用这两种方法，我们都可以传递我们希望 `this` 关键字引用的对象。但是， `.call` 是**立即执行**的。

`.bind` 返回函数的**副本**，但带有绑定上下文！它不是立即执行的。

---

34. 输出是什么?

```js
function sayHi() {
  return (() => 0)()
}

typeof sayHi()
```

- A: `"object"`
- B: `"number"`
- C: `"function"`
- D: `"undefined"`

▼ 答案

答案: B

`sayHi` 方法返回的是立即执行函数(IIFE)的返回值.此立即执行函数的返回值是 `0`， 类型是 `number`

参考：只有7种内置类型： `null` ， `undefined` ， `boolean` ， `number` ， `string` ， `object` 和 `symbol` 。 `function` 不是一种类型，函数是对象，它的类型是 `object` 。

---

35. 下面哪些值是 falsy?

```js
0
new Number(0)
('')
(' ')
new Boolean(false)
undefined
```

- A: `0`, `''`, `undefined`
- B: `0`, `new Number(0)`, `''`, `new Boolean(false)`, `undefined`
- C: `0`, `''`, `new Boolean(false)`, `undefined`
- D: All of them are falsy

▼ 答案

答案: A

只有 6 种 falsy 值:

- `undefined`
- `null`
- `NaN`
- `0`
- `''` (empty string)
- `false`

`Function` 构造函数, 比如 `new Number` 和 `new Boolean`，是 truthy。

---

36. 输出是什么?

```js
console.log(typeof typeof 1)
```

- A: `"number"`
- B: `"string"`
- C: `"object"`
- D: `"undefined"`

▼ 答案

答案: B

`typeof 1` 返回 `"number"`。 `typeof "number"` 返回 `"string"`。

---

37. 输出是什么?

```js
const numbers = [1, 2, 3]
numbers[10] = 11
console.log(numbers)
```

- A: `[1, 2, 3, 7 x null, 11]`
- B: `[1, 2, 3, 11]`
- C: `[1, 2, 3, 7 x empty, 11]`
- D: `SyntaxError`

▼ 答案

答案: C

当你为数组设置超过数组长度的值的时候， JavaScript 会创建名为 "empty slots" 的东西。它们的值实际上是 `undefined`。你会看到以下场景：

`[1, 2, 3, 7 x empty, 11]`

这取决于你的运行环境（每个浏览器，以及 node 环境，都有可能不同）

---

38. 输出是什么?

```js
(() => {
  let x, y
  try {
    throw new Error()
  } catch (x) {
    (x = 1), (y = 2)
    console.log(x)
  }
  console.log(x)
  console.log(y)
})()
```

- A: `1` `undefined` `2`

- B: `undefined` `undefined` `undefined`
- C: `1` `1` `2`
- D: `1` `undefined` `undefined`

▼ 答案

答案: A

`catch` 代码块接收参数 `x` 。当我们传递参数时，这与之前定义的变量 `x` 不同 。这个 `x` 是属于 `catch` 块级作用域的。

然后，我们将块级作用域中的变量赋值为 `1` ，同时也设置了变量 `y` 的值。现在，我们打印块级作用域中的变量 `x` ，值为 `1` 。

`catch` 块之外的变量 `x` 的值仍为 `undefined` ， `y` 的值为 `2` 。当我们在 `catch` 块之外执行 `console.log(x)` 时，返回 `undefined` ， `y` 返回 `2` 。

---

39. JavaScript 中的一切都是?

- A: 基本类型与对象
- B: 函数与对象
- C: 只有对象
- D: 数字与对象
-

▼ 答案

答案: A

JavaScript 只有基本类型和对象。

基本类型包括 `boolean` , `null` , `undefined` , `bigint` , `number` , `string` , `symbol` 。

---

40. 输出是什么?

```js
[[0, 1], [2, 3]].reduce(
  (acc, cur) => {
    return acc.concat(cur)
  },
```

```
  [1, 2]
)
```

- A: `[0, 1, 2, 3, 1, 2]`
- B: `[6, 1, 2]`
- C: `[1, 2, 0, 1, 2, 3]`
- D: `[1, 2, 6]`

▼ 答案

答案: C

`[1, 2]`是初始值。初始值将会作为首次调用时第一个参数 `acc` 的值。在第一次执行时，`acc` 的值是 `[1, 2]`， `cur` 的值是 `[0, 1]`。合并它们，结果为 `[1, 2, 0, 1]`。 第二次执行， `acc` 的值是 `[1, 2, 0, 1]`， `cur` 的值是 `[2, 3]`。合并它们，最终结果为 `[1, 2, 0, 1, 2, 3]`

---

41. 输出是什么?

```js
!!null
!!''
!!1
```

- A: `false` `true` `false`
- B: `false` `false` `true`
- C: `false` `true` `true`
- D: `true` `true` `false`

▼ 答案

答案: B

`null` 是 falsy。 `!null` 的值是 `true`。 `!true` 的值是 `false`。

`""` 是 falsy。 `!""` 的值是 `true`。 `!true` 的值是 `false`。

`1` 是 truthy。 `!1` 的值是 `false`。 `!false` 的值是 `true`。

---

42. `setInterval` 方法的返回值是什么?

```js
setInterval(() => console.log('Hi'), 1000)
```

- A: 一个唯一的id
- B: 该方法指定的毫秒数
- C: 传递的函数
- D: `undefined`

▼ 答案

答案: A

`setInterval` 返回一个唯一的 id。此 id 可被用于 `clearInterval` 函数来取消定时。

---

43. 输出是什么?

```js
[...'Lydia']
```

- A: `["L", "y", "d", "i", "a"]`
- B: `["Lydia"]`
- C: `[[], "Lydia"]`
- D: `[["L", "y", "d", "i", "a"]]`

▼ 答案

答案: A

string 类型是可迭代的。扩展运算符将迭代的每个字符映射成一个元素。

---

44. 输出是什么?

```js
function* generator(i) {
  yield i;
  yield i * 2;
}

const gen = generator(10);
```

```
console.log(gen.next().value);
console.log(gen.next().value);
```

- A: `[0, 10], [10, 20]`
- B: `20, 20`
- C: `10, 20`
- D: `0, 10 and 10, 20`

▼ 答案

答案: C

一般的函数在执行之后是不能中途停下的。但是，生成器函数却可以中途“停下”，之后可以再从停下的地方继续。当生成器遇到 `yield` 关键字的时候，会生成 `yield` 后面的值。注意，生成器在这种情况下不 *返回* (*return*)值，而是 *生成* (*yield*)值。

首先，我们用 `10` 作为参数 `i` 来初始化生成器函数。然后使用 `next()` 方法一步步执行生成器。第一次执行生成器的时候， `i` 的值为 `10` ，遇到第一个 `yield` 关键字，它要生成 `i` 的值。此时，生成器“暂停”，生成了 `10` 。

然后，我们再执行 `next()` 方法。生成器会从刚才暂停的地方继续，这个时候 `i` 还是 `10` 。于是我们走到了第二个 `yield` 关键字处，这时候需要生成的值是 `i*2` ， `i` 为 `10` ，那么此时生成的值便是 `20` 。所以这道题的最终结果是 `10,20` 。

45. 返回值是什么?

```js
const firstPromise = new Promise((res, rej) => {
  setTimeout(res, 500, "one");
});

const secondPromise = new Promise((res, rej) => {
  setTimeout(res, 100, "two");
});

Promise.race([firstPromise, secondPromise]).then(res => console.log(res));
```

- A: `"one"`
- B: `"two"`
- C: `"two" "one"`
- D: `"one" "two"`

▼ 答案

答案: B

当我们向 `Promise.race` 方法中传入多个 `Promise` 时，会进行 *优先* 解析。在这个例子中，我们用 `setTimeout` 给 `firstPromise` 和 `secondPromise` 分别设定了500ms和100ms的定时器。这意味着 `secondPromise` 会首先解析出字符串 `two` 。那么此时 `res` 参数即为 `two` ，是为输出结果。

---

46. 输出是什么?

```js
let person = { name: "Lydia" };
const members = [person];
person = null;

console.log(members);
```

- A: `null`
- B: `[null]`
- C: `[{}]`
- D: `[{ name: "Lydia" }]`

▼ 答案

答案: D

首先我们声明了一个拥有 `name` 属性的对象 `person` 。

然后我们又声明了一个变量 `members` . 将首个元素赋值为变量 `person` 。 当设置两个对象彼此相等时，它们会通过 *引用* 进行交互。但是当你将引用从一个变量分配至另一个变量时，其实只是执行了一个 *复制* 操作。（注意一点，他们的引用 *并不相同*!）

接下来我们让 `person` 等于 `null` 。

我们没有修改数组第一个元素的值，而只是修改了变量 `person` 的值,因为元素（复制而来）的引用与 `person` 不同。 `members` 的第一个元素仍然保持着对原始对象的引用。当我们输出

`members` 数组时，第一个元素会将引用的对象打印出来。

---

47. 输出是什么?

```js
const person = {
  name: "Lydia",
  age: 21
};

for (const item in person) {
  console.log(item);
}
```

- A: `{ name: "Lydia" }, { age: 21 }`
- B: `"name", "age"`
- C: `"Lydia", 21`
- D: `["name", "Lydia"], ["age", 21]`

▼ 答案

答案: B

在 `for-in` 循环中,我们可以通过对象的key来进行迭代,也就是这里的 `name` 和 `age` 。在底层，对象的key都是字符串（如果他们不是Symbol的话）。在每次循环中，我们将 `item` 设定为当前遍历到的key.所以一开始， `item` 是 `name` ，之后 `item` 输出的则是 `age` 。

---

48. 输出是什么?

```js
console.log(3 + 4 + "5");
```

- A: `"345"`
- B: `"75"`
- C: `12`
- D: `"12"`

▼ 答案

答案: B

当所有运算符的 *优先级* 相同时，计算表达式需要确定运算符的结合顺序，即从右到左还是从左往右。在这个例子中，我们只有一类运算符 `+`，对于加法来说，结合顺序就是从左到右。

`3 + 4` 首先计算，得到数字 `7`.

由于类型的强制转换， `7 + '5'` 的结果是 `"75"`. JavaScript将 `7` 转换成了字符串，可以参考问题15.我们可以用 `+` 号把两个字符串连接起来。 `"7" + "5"` 就得到了 `"75"`.

---

49. `num` 的值是什么?

```js
const num = parseInt("7*6", 10);
```

- A: `42`
- B: `"42"`
- C: `7`
- D: `NaN`

▼ **答案**

答案: C

只返回了字符串中第一个字母. 设定了 *进制* 后 (也就是第二个参数，指定需要解析的数字是什么进制: 十进制、十六机制、八进制、二进制等等.....), `parseInt` 检查字符串中的字符是否合法. 一旦遇到一个在指定进制中不合法的字符后，立即停止解析并且忽略后面所有的字符。

`*` 就是不合法的数字字符。所以只解析到 `"7"`，并将其解析为十进制的 `7`. `num` 的值即为 `7`.

---

50. 输出是什么?

```js
[1, 2, 3].map(num => {
  if (typeof num === "number") return;
  return num * 2;
});
```

- A: `[]`
- B: `[null, null, null]`
- C: `[undefined, undefined, undefined]`

- D: `[ 3 x empty ]`

▼ 答案

答案: C

对数组进行映射的时候，`num` 就是当前循环到的元素. 在这个例子中，所有的映射都是number类型，所以if中的判断 `typeof num === "number"` 结果都是 `true` .map函数创建了新数组并且将函数的返回值插入数组。

但是，没有任何值返回。当函数没有返回任何值时，即默认返回 `undefined` .对数组中的每一个元素来说，函数块都得到了这个返回值，所以结果中每一个元素都是 `undefined` .

---

51. 输出的是什么?

```js
function getInfo(member, year) {
  member.name = "Lydia";
  year = "1998";
}

const person = { name: "Sarah" };
const birthYear = "1997";

getInfo(person, birthYear);

console.log(person, birthYear);
```

- A: `{ name: "Lydia" }, "1997"`
- B: `{ name: "Sarah" }, "1998"`
- C: `{ name: "Lydia" }, "1998"`
- D: `{ name: "Sarah" }, "1997"`

▼ 答案

答案: A

普通参数都是 *值* 传递的，而对象则不同，是 *引用* 传递。所以说， `birthYear` 是值传递，因为他是个字符串而不是对象。当我们对参数进行值传递时，会创建一份该值的 *复制* 。（可以参考问题46）

变量 `birthYear` 有一个对 `"1997"` 的引用，而传入的参数也有一个对 `"1997"` 的引用，但二者的引用并不相同。当我们通过给 `year` 赋值 `"1998"` 来更新 `year` 的值的时候我们只是更新了 `year` （的引用）。此时 `birthYear` 仍然是 `"1997"` .

而 `person` 是个对象。参数 `member` 引用与之 *相同的* 对象。当我们修改 `member` 所引用对象的属性时, `person` 的相应属性也被修改了,因为他们引用了相同的对象. `person` 的 `name` 属性也变成了 `"Lydia"` .

---

52. 输出是什么?

```js
function greeting() {
  throw "Hello world!";
}

function sayHi() {
  try {
    const data = greeting();
    console.log("It worked!", data);
  } catch (e) {
    console.log("Oh no an error!", e);
  }
}

sayHi();
```

- A: `"It worked! Hello world!"`
- B: `"Oh no an error: undefined`
- C: `SyntaxError: can only throw Error objects`
- D: `"Oh no an error: Hello world!`

▼ **答案**

答案: D

通过 `throw` 语句，我么可以创建自定义错误。 而通过它，我们可以抛出异常。异常可以是一个**字符串**, 一个 **数字**, 一个 **布尔类型** 或者是一个 **对象**。在本例中，我们的异常是字符串 `'Hello world'` .

通过 `catch` 语句，我们可以设定当 `try` 语句块中抛出异常后应该做什么处理。在本例中抛出的异常是字符串 `'Hello world'` . `e` 就是这个字符串，因此被输出。最终结果就是 `'Oh an error: Hello world'` .

53. 输出是什么?

```js
function Car() {
  this.make = "Lamborghini";
  return { make: "Maserati" };
}

const myCar = new Car();
console.log(myCar.make);
```

- A: `"Lamborghini"`
- B: `"Maserati"`
- C: `ReferenceError`
- D: `TypeError`

▼ 答案

答案: B

返回属性的时候，属性的值等于 *返回的* 值，而不是构造函数中设定的值。我们返回了字符串 `"Maserati"`，所以 `myCar.make` 等于 `"Maserati"`.

54. 输出是什么?

```js
(() => {
  let x = (y = 10);
})();

console.log(typeof x);
console.log(typeof y);
```

- A: `"undefined", "number"`
- B: `"number", "number"`
- C: `"object", "number"`
- D: `"number", "undefined"`

▼ 答案

答案: A

`let x = y = 10;` 是下面这个表达式的缩写:

```js
y = 10;
let x = y;
```

我们设定 `y` 等于 `10` 时,我们实际上增加了一个属性 `y` 给全局对象(浏览器里的 `window`, Nodejs里的 `global`)。在浏览器中， `window.y` 等于 `10` .

然后我们声明了变量 `x` 等于 `y` ,也是 `10` .但变量是使用 `let` 声明的，它只作用于 *块级作用域* 仅在声明它的块中有效；就是案例中的立即调用表达式(IIFE)。使用 `typeof` 操作符时, 操作值 `x` 没有被定义：因为我们在 `x` 声明块的外部，无法调用它。这就意味着 `x` 未定义。未分配或是未声明的变量类型为 `"undefined"` . `console.log(typeof x)` 返回 `"undefined"` .

而我们创建了全局变量 `y` ，并且设定 `y` 等于 `10` .这个值在我们的代码各处都访问的到。 `y` 已经被定义了，而且有一个 `"number"` 类型的值。 `console.log(typeof y)` 返回 `"number"` .

---

55. 输出是什么?

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

Dog.prototype.bark = function() {
  console.log(`Woof I am ${this.name}`);
};

const pet = new Dog("Mara");

pet.bark();

delete Dog.prototype.bark;

pet.bark();
```

- A: `"Woof I am Mara"` , `TypeError`

- B: `"Woof I am Mara"`, `"Woof I am Mara"`
- C: `"Woof I am Mara"`, `undefined`
- D: `TypeError`, `TypeError`

▼ 答案

答案: A

我们可以用`delete`关键字删除对象的属性，对原型也是适用的。删除了原型的属性后，该属性在原型链上就不可用了。在本例中，函数`bark`在执行了`delete Dog.prototype.bark`后不可用, 然而后面的代码还在调用它。

当我们尝试调用一个不存在的函数时`TypeError`异常会被抛出。在本例中就是`TypeError: pet.bark is not a function`，因为`pet.bark`是`undefined`.

---

56. 输出是什么?

```js
const set = new Set([1, 1, 2, 3, 4]);

console.log(set);
```

- A: `[1, 1, 2, 3, 4]`
- B: `[1, 2, 3, 4]`
- C: `{1, 1, 2, 3, 4}`
- D: `{1, 2, 3, 4}`

▼ 答案

答案: D

`Set`对象是独一无二的值的集合：也就是说同一个值在其中仅出现一次。

我们传入了数组`[1, 1, 2, 3, 4]`，他有一个重复值`1`.以为一个集合里不能有两个重复的值，其中一个就被移除了。所以结果是`{1, 2, 3, 4}`.

---

57. 输出是什么?

```js
// counter.js
let counter = 10;
```

```
export default counter;
```

```js
// index.js
import myCounter from "./counter";

myCounter += 1;

console.log(myCounter);
```

- A: `10`
- B: `11`
- C: `Error`
- D: `NaN`

▼ 答案

答案: C

引入的模块是 *只读* 的: 你不能修改引入的模块。只有导出他们的模块才能修改其值。

当我们给 `myCounter` 增加一个值的时候会抛出一个异常： `myCounter` 是只读的，不能被修改。

---

58. 输出是什么?

```js
const name = "Lydia";
age = 21;

console.log(delete name);
console.log(delete age);
```

- A: `false` , `true`
- B: `"Lydia"` , `21`
- C: `true` , `true`
- D: `undefined` , `undefined`

▼ 答案

答案: A

`delete` 操作符返回一个布尔值： `true` 指删除成功，否则返回 `false` . 但是通过 `var` , `const` 或 `let` 关键字声明的变量无法用 `delete` 操作符来删除。

`name` 变量由 `const` 关键字声明，所以删除不成功:返回 `false` . 而我们设定 `age` 等于 `21` 时,我们实际上添加了一个名为 `age` 的属性给全局对象。对象中的属性是可以删除的，全局对象也是如此，所以 `delete age` 返回 `true` .

---

59. 输出是什么?

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

console.log(y);
```

- A: `[[1, 2, 3, 4, 5]]`
- B: `[1, 2, 3, 4, 5]`
- C: `1`
- D: `[1]`

▼ 答案

答案: C

我们可以通过解构赋值来解析来自对象的数组或属性的值，比如说：

```js
[a, b] = [1, 2];
```

`a` 的值现在是 `1` ， `b` 的值现在是 `2` .而在题目中，我们是这么做的:

```js
[y] = [1, 2, 3, 4, 5];
```

也就是说， `y` 等于数组的第一个值就是数字 `1` .我们输出 `y` ， 返回 `1` .

---

60. 输出是什么?

```js
const user = { name: "Lydia", age: 21 };
const admin = { admin: true, ...user };

console.log(admin);
```

- A: `{ admin: true, user: { name: "Lydia", age: 21 } }`
- B: `{ admin: true, name: "Lydia", age: 21 }`
- C: `{ admin: true, user: ["Lydia", 21] }`
- D: `{ admin: true }`

▼ 答案

答案: B

扩展运算符 `...` 为对象的组合提供了可能。你可以复制对象中的键值对，然后把它们加到另一个对象里去。在本例中，我们复制了 `user` 对象键值对，然后把它们加入到 `admin` 对象中。 `admin` 对象就拥有了这些键值对，所以结果为 `{ admin: true, name: "Lydia", age: 21 }` .

---

61. 输出是什么?

```js
const person = { name: "Lydia" };

Object.defineProperty(person, "age", { value: 21 });

console.log(person);
console.log(Object.keys(person));
```

- A: `{ name: "Lydia", age: 21 }` , `["name", "age"]`
- B: `{ name: "Lydia", age: 21 }` , `["name"]`
- C: `{ name: "Lydia"}` , `["name", "age"]`
- D: `{ name: "Lydia"}` , `["age"]`

▼ 答案

答案: B

通过 `defineProperty` 方法，我们可以给对象添加一个新属性，或者修改已经存在的属性。而我们使用 `defineProperty` 方法给对象添加了一个属性之后，属性默认为 *不可枚举(not*

enumerable). `Object.keys` 方法仅返回对象中 *可枚举(enumerable)* 的属性，因此只剩下了 `"name"` .

用 `defineProperty` 方法添加的属性默认不可变。你可以通过 `writable` , `configurable` 和 `enumerable` 属性来改变这一行为。这样的话， 相比于自己添加的属性， `defineProperty` 方法添加的属性有了更多的控制权。

---

62. 输出是什么?

```js
const settings = {
  username: "lydiahallie",
  level: 19,
  health: 90
};

const data = JSON.stringify(settings, ["level", "health"]);
console.log(data);
```

- A: `"{"level":19, "health":90}"`
- B: `"{"username": "lydiahallie"}"`
- C: `"["level", "health"]"`
- D: `"{"username": "lydiahallie", "level":19, "health":90}"`

▼ 答案

答案: A

`JSON.stringify` 的第二个参数是 *替代者(replacer)*. 替代者(replacer)可以是个函数或数组，用以控制哪些值如何被转换为字符串。

如果替代者(replacer)是个 *数组* ，那么就只有包含在数组中的属性将会被转化为字符串。在本例中，只有名为 `"level"` 和 `"health"` 的属性被包括进来， `"username"` 则被排除在外。 `data` 就等于 `"{"level":19, "health":90}"` .

而如果替代者(replacer)是个 *函数*，这个函数将被对象的每个属性都调用一遍。 函数返回的值会成为这个属性的值，最终体现在转化后的JSON字符串中（译者注：Chrome下，经过实验，如果所有属性均返回同一个值的时候有异常，会直接将返回值作为结果输出而不会输出JSON字符串），而如果返回值为 `undefined` ，则该属性会被排除在外。

---

63. 输出是什么?

```js
let num = 10;

const increaseNumber = () => num++;
const increasePassedNumber = number => number++;

const num1 = increaseNumber();
const num2 = increasePassedNumber(num1);

console.log(num1);
console.log(num2);
```

- A: `10`, `10`
- B: `10`, `11`
- C: `11`, `11`
- D: `11`, `12`

▼ 答案

答案: A

一元操作符 `++` *先返回* 操作值, *再累加* 操作值。 `num1` 的值是 `10`, 因为 `increaseNumber` 函数首先返回 `num` 的值，也就是 `10`，随后再进行 `num` 的累加。

`num2` 是 `10` 因为我们将 `num1` 传入 `increasePassedNumber`. `number` 等于 `10` （ `num1` 的值。同样道理， `++` *先返回* 操作值, *再累加* 操作值。） `number` 是 `10`，所以 `num2` 也是 `10`.

---

64. 输出什么?

```js
const value = { number: 10 };

const multiply = (x = { ...value }) => {
  console.log(x.number *= 2);
};

multiply();
```

```
multiply();
multiply(value);
multiply(value);
```

- A: `20` , `40` , `80` , `160`
- B: `20` , `40` , `20` , `40`
- C: `20` , `20` , `20` , `40`
- D: `NaN` , `NaN` , `20` , `40`

▼ 答案

答案: C

在ES6中，我们可以使用默认值初始化参数。如果没有给函数传参，或者传的参值为 `"undefined"` ，那么参数的值将是默认值。上述例子中，我们将 `value` 对象进行了解构并传到一个新对象中，因此 `x` 的默认值为 `{number: 10}` 。

默认参数在调用时才会进行计算，每次调用函数时，都会创建一个新的对象。我们前两次调用 `multiply` 函数且不传递值，那么每一次 `x` 的默认值都为 `{number: 10}` ，因此打印出该数字的乘积值为 `20` 。

第三次调用 `multiply` 时，我们传递了一个参数，即对象 `value` 。 `*=` 运算符实际上是 `x.number = x.number * 2` 的简写，我们修改了 `x.number` 的值，并打印出值 `20` 。

第四次，我们再次传递 `value` 对象。 `x.number` 之前被修改为 `20` ，所以 `x.number * = 2` 打印为 `40` 。

---

65. 输出什么?

```js
[1, 2, 3, 4].reduce((x, y) => console.log(x, y));
```

- A: `1` `2` and `3` `3` and `6` `4`
- B: `1` `2` and `2` `3` and `3` `4`
- C: `1` `undefined` and `2` `undefined` and `3` `undefined` and `4` `undefined`
- D: `1` `2` and `undefined` `3` and `undefined` `4`

▼ 答案

答案: D

`reducer` 函数接收4个参数:

1. Accumulator (acc) (累计器)
2. Current Value (cur) (当前值)
3. Current Index (idx) (当前索引)
4. Source Array (src) (源数组)

`reducer` 函数的返回值将会分配给累计器，该返回值在数组的每个迭代中被记住，并最后成为最终的单个结果值。

`reducer` 函数还有一个可选参数 `initialValue`，该参数将作为第一次调用回调函数时的第一个参数的值。如果没有提供 `initialValue`，则将使用数组中的第一个元素。

在上述例子， `reduce` 方法接收的第一个参数(Accumulator)是 `x`，第二个参数(Current Value)是 `y`。

在第一次调用时，累加器 `x` 为 `1`，当前值 `“y”` 为 `2`，打印出累加器和当前值： `1` 和 `2`。

例子中我们的回调函数没有返回任何值，只是打印累加器的值和当前值。如果函数没有返回值，则默认返回 `undefined`。 在下一次调用时，累加器为 `undefined`，当前值为“3”, 因此 `undefined` 和 `3` 被打印出。

在第四次调用时，回调函数依然没有返回值。 累加器再次为 `undefined`，当前值为“4”。 `undefined` 和 `4` 被打印出。

---

66. 使用哪个构造函数可以成功继承 `Dog` 类?

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

class Labrador extends Dog {
  // 1
  constructor(name, size) {
    this.size = size;
  }
  // 2
  constructor(name, size) {
    super(name);
    this.size = size;
  }
```

```
  // 3
  constructor(size) {
    super(name);
    this.size = size;
  }
  // 4
  constructor(name, size) {
    this.name = name;
    this.size = size;
  }

};
```

- A: 1
- B: 2
- C: 3
- D: 4

▼ 答案

答案: B

在子类中，在调用 `super` 之前不能访问到 `this` 关键字。 如果这样做，它将抛出一个 `ReferenceError` ： 1和4将引发一个引用错误。

使用 `super` 关键字，需要用给定的参数来调用父类的构造函数。 父类的构造函数接收 `name` 参数，因此我们需要将 `name` 传递给 `super` 。

`Labrador` 类接收两个参数， `name` 参数是由于它继承了 `Dog` ， `size` 作为 `Labrador` 类的额外属性，它们都需要传递给 `Labrador` 的构造函数，因此使用构造函数2正确完成。

---

67. 输出什么?

```js
// index.js
console.log('running index.js');
import { sum } from './sum.js';
console.log(sum(1, 2));

// sum.js
console.log('running sum.js');
export const sum = (a, b) => a + b;
```

- A: `running index.js`, `running sum.js`, `3`
- B: `running sum.js`, `running index.js`, `3`
- C: `running sum.js`, `3`, `running index.js`
- D: `running index.js`, `undefined`, `running sum.js`

▼ 答案

答案: B

`import`命令是编译阶段执行的，在代码运行之前。因此这意味着被导入的模块会先运行，而导入模块的文件会后执行。

这是CommonJS中`require()`和`import`之间的区别。使用`require()`，您可以在运行代码时根据需要加载依赖项。 如果我们使用`require`而不是`import`，`running index.js`，`running sum.js`，`3`会被依次打印。

---

68. 输出什么?

```js
console.log(Number(2) === Number(2))
console.log(Boolean(false) === Boolean(false))
console.log(Symbol('foo') === Symbol('foo'))
```

- A: `true`, `true`, `false`
- B: `false`, `true`, `false`
- C: `true`, `false`, `true`
- D: `true`, `true`, `true`

▼ 答案

答案: A

每个`Symbol`都是完全唯一的。传递给`Symbol`的参数只是给`Symbol`的一个描述。`Symbol`的值不依赖于传递的参数。 当我们测试相等时，我们创建了两个全新的符号：第一个`Symbol('foo')`，第二个`Symbol('foo')`，这两个值是唯一的，彼此不相等，因此返回`false`。

---

69. 输出什么?

```js
const name = "Lydia Hallie"
console.log(name.padStart(13))
console.log(name.padStart(2))
```

- A: `"Lydia Hallie"`, `"Lydia Hallie"`
- B: `" Lydia Hallie"`, `" Lydia Hallie"` (`"[13x whitespace]Lydia Hallie"`, `"[2x whitespace]Lydia Hallie"`)
- C: `" Lydia Hallie"`, `"Lydia Hallie"` (`"[1x whitespace]Lydia Hallie"`, `"Lydia Hallie"`)
- D: `"Lydia Hallie"`, `"Lyd"`

▼ 答案

答案: C

使用 `padStart` 方法，我们可以在字符串的开头添加填充。传递给此方法的参数是字符串的总长度（包含填充）。字符串 `Lydia Hallie` 的长度为 `12`，因此 `name.padStart（13）` 在字符串的开头只会插入1（`13 - 12 = 1`）个空格。

如果传递给 `padStart` 方法的参数小于字符串的长度，则不会添加填充。

---

70. 输出什么?

```js
console.log("🥑" + "💻");
```

- A: `"🥑💻"`
- B: `257548`
- C: A string containing their code points
- D: Error

▼ 答案

答案: A

使用 `+` 运算符，您可以连接字符串。 上述情况，我们将字符串 `“🥑”` 与字符串 `”💻“` 连接起来，产生 `”🥑💻“`。

---

71. 如何能打印出 `console.log` 语句后注释掉的值?

```js
function* startGame() {
  const answer = yield "Do you love JavaScript?";
  if (answer !== "Yes") {
    return "Oh wow... Guess we're gone here";
  }
  return "JavaScript loves you back ♡";
}

const game = startGame();
console.log(/* 1 */); // Do you love JavaScript?
console.log(/* 2 */); // JavaScript loves you back ♡
```

- A: `game.next("Yes").value` and `game.next().value`
- B: `game.next.value("Yes")` and `game.next.value()`
- C: `game.next().value` and `game.next("Yes").value`
- D: `game.next.value()` and `game.next.value("Yes")`

▼ 答案

答案: C

`generator` 函数在遇到 `yield` 关键字时会"暂停"其执行。 首先，我们需要让函数产生字符串 `Do you love JavaScript?` ，这可以通过调用 `game.next().value` 来完成。上述函数的第一行就有一个 `yield` 关键字，那么运行立即停止了， `yield` 表达式本身没有返回值，或者说总是返回 `undefined` ，这意味着此时变量 `answer` 为 `undefined`

`next` 方法可以带一个参数，该参数会被当作上一个 `yield` 表达式的返回值。当我们调用 `game.next("Yes").value` 时，先前的 `yield` 的返回值将被替换为传递给 `next()` 函数的参数 `"Yes"` 。此时变量 `answer` 被赋值为 `"Yes"` ， `if` 语句返回 `false` ，所以 `JavaScript loves you back ♡` 被打印。

---

72. 输出什么?

```js
console.log(String.raw`Hello\nworld`);
```

- A: `Hello world!`
- B: `Hello`
  `world`

- C: `Hello\nworld`
- D: `Hello\n`
  `world`

▼ 答案

答案: C

`String.raw` 函数是用来获取一个模板字符串的原始字符串的，它返回一个字符串，其中忽略了转义符（ `\n` ， `\v` ， `\t` 等）。但反斜杠可能造成问题，因为你可能会遇到下面这种类似情况：

```js
const path = `C:\Documents\Projects\table.html`
String.raw`${path}`
```

这将导致：

`"C:DocumentsProjects able.html"`

直接使用 `String.raw`

```js
String.raw`C:\Documents\Projects\table.html`
```

它会忽略转义字符并打印： `C:\Documents\Projects\table.html`

上述情况，字符串是 `Hello\nworld` 被打印出。

---

73. 输出什么?

```js
async function getData() {
  return await Promise.resolve("I made it!");
}

const data = getData();
console.log(data);
```

- A: `"I made it!"`
- B: `Promise {<resolved>: "I made it!"}`
- C: `Promise {<pending>}`

- D: `undefined`

▼ 答案

答案: C

异步函数始终返回一个promise。`await` 仍然需要等待promise的解决：当我们调用 `getData()` 并将其赋值给 `data`，此时 `data` 为 `getData` 方法返回的一个挂起的promise，该promise并没有解决。

如果我们想要访问已解决的值 `"I made it!"`，可以在 `data` 上使用 `.then()` 方法：

`data.then(res => console.log(res))`

这样将打印 `"I made it!"`

---

74. 输出什么?

```js
function addToList(item, list) {
  return list.push(item);
}

const result = addToList("apple", ["banana"]);
console.log(result);
```

- A: `['apple', 'banana']`
- B: `2`
- C: `true`
- D: `undefined`

▼ 答案

答案: B

`push()` 方法返回新数组的长度。一开始，数组包含一个元素（字符串 `"banana"`），长度为1。 在数组中添加字符串 `"apple"` 后，长度变为2，并将从 `addToList` 函数返回。

`push` 方法修改原始数组，如果你想从函数返回数组而不是数组长度，那么应该在push `item` 之后返回 `list`。

---

75. 输出什么?

```js
const box = { x: 10, y: 20 };

Object.freeze(box);

const shape = box;
shape.x = 100;
console.log(shape)
```

- A: `{ x: 100, y: 20 }`
- B: `{ x: 10, y: 20 }`
- C: `{ x: 100 }`
- D: `ReferenceError`

▼ 答案

答案: B

`Object.freeze` 使得无法添加、删除或修改对象的属性（除非属性的值是另一个对象）。

当我们创建变量 `shape` 并将其设置为等于冻结对象 `box` 时， `shape` 指向的也是冻结对象。你可以使用 `Object.isFrozen` 检查一个对象是否被冻结，上述情况，`Object.isFrozen（shape）` 将返回 `true` 。

由于 `shape` 被冻结，并且 `x` 的值不是对象，所以我们不能修改属性 `x` 。 `x` 仍然等于 `10` ， `{x: 10, y: 20}` 被打印。

注意，上述例子我们对属性 `x` 进行修改，可能会导致抛出TypeError异常（最常见但不仅限于严格模式下时）。

---

76. 输出什么?

```js
const { name: myName } = { name: "Lydia" };

console.log(name);
```

- A: `"Lydia"`
- B: `"myName"`
- C: `undefined`
- D: `ReferenceError`

▼ 答案

答案: D

当我们从右侧的对象解构属性 `name` 时，我们将其值 `Lydia` 分配给名为 `myName` 的变量。

使用 `{name: myName}`，我们是在告诉JavaScript我们要创建一个名为 `myName` 的新变量，并且其值是右侧对象的 `name` 属性的值。

当我们尝试打印 `name`，一个未定义的变量时，就会引发 `ReferenceError`。

---

77. 以下是个纯函数么?

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

- A: Yes
- B: No

▼ 答案

答案: A

纯函数一种若输入参数相同，则永远会得到相同输出的函数。

`sum` 函数总是返回相同的结果。 如果我们传递 `1` 和 `2`，它将总是返回 `3` 而没有副作用。如果我们传递 `5` 和 `10`，它将总是返回 `15`，依此类推，这是纯函数的定义。

---

78. 输出什么?

```js
const add = () => {
  const cache = {};
  return num => {
    if (num in cache) {
```

```
        return `From cache! ${cache[num]}`;
      } else {
        const result = num + 10;
        cache[num] = result;
        return `Calculated! ${result}`;
      }
    };
  };

  const addFunction = add();
  console.log(addFunction(10));
  console.log(addFunction(10));
  console.log(addFunction(5 * 2));
```

- A: `Calculated! 20` `Calculated! 20` `Calculated! 20`
- B: `Calculated! 20` `From cache! 20` `Calculated! 20`
- C: `Calculated! 20` `From cache! 20` `From cache! 20`
- D: `Calculated! 20` `From cache! 20` `Error`

▼ **答案**

答案: C

`add` 函数是一个记忆函数。 通过记忆化，我们可以缓存函数的结果，以加快其执行速度。上述情况，我们创建一个 `cache` 对象，用于存储先前返回过的值。

如果我们使用相同的参数多次调用 `addFunction` 函数，它首先检查缓存中是否已有该值，如果有，则返回缓存值，这将节省执行时间。如果没有，那么它将计算该值，并存储在缓存中。

我们用相同的值三次调用了 `addFunction` 函数：

在第一次调用， `num` 等于 `10` 时函数的值尚未缓存，if语句 `num in cache` 返回 `false` ，else块的代码被执行： `Calculated! 20` ，并且其结果被添加到缓存对象， `cache` 现在看起来像 `{10: 20}` 。

第二次， `cache` 对象包含 `10` 的返回值。 if语句 `num in cache` 返回 `true` ， `From cache! 20` 被打印。

第三次，我们将 `5 * 2` (值为10)传递给函数。 `cache` 对象包含 `10` 的返回值。 if语句 `num in cache` 返回 `true` ， `From cache! 20` 被打印。

---

79. 输出什么?

```js
const myLifeSummedUp = ["☕", "💻", "🍷", "🍺"]

for (let item in myLifeSummedUp) {
  console.log(item)
}

for (let item of myLifeSummedUp) {
  console.log(item)
}
```

- A: `0` `1` `2` `3` and `"☕"` `"💻"` `"🍷"` `"🍺"`
- B: `"☕"` `"💻"` `"🍷"` `"🍺"` and `"☕"` `"💻"` `"🍷"` `"🍺"`
- C: `"☕"` `"💻"` `"🍷"` `"🍺"` and `0` `1` `2` `3`
- D: `0` `1` `2` `3` and `{0: "☕", 1: "💻", 2: "🍷", 3: "🍺"}`

▼ 答案

答案: A

通过 `for-in` 循环，我们可以遍历一个对象**自有的**、**继承的**、**可枚举的**、**非Symbol的**属性。在数组中，可枚举属性是数组元素的“键”， 即它们的索引。 类似于下面这个对象：

`{0: "☕", 1: "💻", 2: "🍷", 3: "🍺"}`

其中键则是可枚举属性，因此 `0` ， `1` ， `2` ， `3` 被记录。

通过 `for-of` 循环，我们可以迭代**可迭代对象**（包括 `Array` ， `Map` ， `Set` ， `String` ， `arguments` 等）。当我们迭代数组时，在每次迭代中，不同属性的值将被分配给变量 `item` ，因此 “☕” ， “💻” ， “🍷” ， “🍺” 被打印。

80. 输出什么?

```js
const list = [1 + 2, 1 * 2, 1 / 2]
console.log(list)
```

- A: `["1 + 2", "1 * 2", "1 / 2"]`
- B: `["12", 2, 0.5]`
- C: `[3, 2, 0.5]`
- D: `[1, 1, 1]`

▼ 答案

答案: C

数组元素可以包含任何值。 数字，字符串，布尔值，对象，数组， `null` ， `undeifned` ，以及其他表达式，如日期，函数和计算。

元素将等于返回的值。 `1 + 2` 返回 `3` ， `1 * 2` 返回'2 `, '1 / 2` 返回 `0.5` 。

---

81. 输出什么?

```js
function sayHi(name) {
  return `Hi there, ${name}`
}

console.log(sayHi())
```

- A: `Hi there,`
- B: `Hi there, undefined`
- C: `Hi there, null`
- D: `ReferenceError`

▼ 答案

答案: B

默认情况下，如果不给函数传参，参数的值将为 `undefined` 。 上述情况，我们没有给参数 `name` 传值。 `name` 等于 `undefined` ，并被打印。

在ES6中，我们可以使用默认参数覆盖此默认的 `undefined` 值。 例如：

`function sayHi（name =“Lydia”）{...}`

在这种情况下，如果我们没有传递值或者如果我们传递 `undefined` ， `name` 总是等于字符串 `Lydia`

---

82. 输出什么?

```js
var status = "😊"

setTimeout(() => {
  const status = "😎"

  const data = {
    status: "🍆",
    getStatus() {
      return this.status
    }
  }

  console.log(data.getStatus())
  console.log(data.getStatus.call(this))
}, 0)
```

- A: `"🍆"` and `"😎"`
- B: `"🍆"` and `"😊"`
- C: `"😎"` and `"😊"`
- D: `"😊"` and `"😊"`

▼ 答案

答案: B

`this` 关键字的指向取决于使用它的位置。 在**函数**中，比如 `getStatus` ， `this` 指向的是调用它的对象，上述例子中 `data` 对象调用了 `getStatus` ，因此 `this` 指向的就是 `data` 对象。 当我们打印 `this.status` 时， `data` 对象的 `status` 属性被打印，即 `"🍆"` 。

使用 `call` 方法，可以更改 `this` 指向的对象。 `data.getStatus.call(this)` 是将 `this` 的指向由 `data` 对象更改为全局对象。在全局对象上，有一个名为 `status` 的变量，其值为 `”😊“` 。 因此打印 `this.status` 时，会打印 `“😊”` 。

---

83. 输出什么?

```js
const person = {
  name: "Lydia",
  age: 21
}

let city = person.city
city = "Amsterdam"

console.log(person)
```

- A: `{ name: "Lydia", age: 21 }`
- B: `{ name: "Lydia", age: 21, city: "Amsterdam" }`
- C: `{ name: "Lydia", age: 21, city: undefined }`
- D: `"Amsterdam"`

▼ 答案

答案: A

我们将变量`city`设置为等于`person`对象上名为`city`的属性的值。 这个对象上没有名为`city`的属性，因此变量`city`的值为`undefined`。

请注意，我们没有引用`person`对象本身，只是将变量`city`设置为等于`person`对象上`city`属性的当前值。

然后，我们将`city`设置为等于字符串`“Amsterdam”`。 这不会更改person对象：没有对该对象的引用。

因此打印`person`对象时，会返回未修改的对象。

---

84. 输出什么?

```js
function checkAge(age) {
  if (age < 18) {
    const message = "Sorry, you're too young."
  } else {
    const message = "Yay! You're old enough!"
  }

  return message
}
```

```
console.log(checkAge(21))
```

- A: `"Sorry, you're too young."`
- B: `"Yay! You're old enough!"`
- C: `ReferenceError`
- D: `undefined`

▼ 答案

答案: C

`const` 和 `let` 声明的变量是具有**块级作用域**的，块是大括号（ `{}` ）之间的任何东西, 即上述情况 `if / else` 语句的花括号。 由于块级作用域，我们无法在声明的块之外引用变量，因此抛出 `ReferenceError` 。

---

85. 什么样的信息将被打印?

```js
fetch('https://www.website.com/api/user/1')
  .then(res => res.json())
  .then(res => console.log(res))
```

- A: `fetch` 方法的结果
- B: 第二次调用 `fetch` 方法的结果
- C: 前一个 `.then()` 中回调方法返回的结果
- D: 总是 `undefined`

▼ 答案

答案: C

第二个 `.then` 中 `res` 的值等于前一个 `.then` 中的回调函数返回的值。 你可以像这样继续链接 `.then` ，将值传递给下一个处理程序。

---

86. 哪个选项是将 `hasName` 设置为 `true` 的方法，前提是不能将 `true` 作为参数传递?

```js
function getName(name) {
  const hasName = //
}
```

- A: `!!name`
- B: `name`
- C: `new Boolean(name)`
- D: `name.length`

▼ 答案

答案: A

使用逻辑非运算符 `!` ，将返回一个布尔值，使用 `!! name` ，我们可以确定 `name` 的值是真的还是假的。 如果 `name` 是真实的，那么 `!name` 返回 `false` 。 `!false` 返回 `true` 。

通过将 `hasName` 设置为 `name` ，可以将 `hasName` 设置为等于传递给 `getName` 函数的值，而不是布尔值 `true` 。

`new Boolean（true）` 返回一个对象包装器，而不是布尔值本身。

`name.length` 返回传递的参数的长度，而不是布尔值 `true` 。

87. 输出什么?

```js
console.log("I want pizza"[0])
```

- A: `"""`
- B: `"I"`
- C: `SyntaxError`
- D: `undefined`

▼ 答案

答案: B

可以使用方括号表示法获取字符串中特定索引的字符，字符串中的第一个字符具有索引0，依此类推。 在这种情况下，我们想要得到索引为0的元素，字符 `'I'` 被记录。

请注意，IE7及更低版本不支持此方法。 在这种情况下，应该使用 `.charAt（）`

88. 输出什么?

```js
function sum(num1, num2 = num1) {
  console.log(num1 + num2)
}

sum(10)
```

- A: `NaN`
- B: `20`
- C: `ReferenceError`
- D: `undefined`

▼ 答案

答案: B

您可以将默认参数的值设置为函数的另一个参数，只要另一个参数定义在其之前即可。 我们将值 `10` 传递给 `sum` 函数。 如果 `sum` 函数只接收1个参数，则意味着没有传递 `num2` 的值，这种情况下， `num1` 的值等于传递的值 `10` 。 `num2` 的默认值是 `num1` 的值，即 `10` 。 `num1 + num2` 返回 `20` 。

如果您尝试将默认参数的值设置为后面定义的参数，则可能导致参数的值尚未初始化，从而引发错误。比如：

```js
function test(m = n, n = 2) {
    console.log(m, n)
}
test() // Uncaught ReferenceError: Cannot access 'n' before initialization
test(3) // 3 2
test(3, 4) // 3 4
```

89. 输出什么?

```js
// module.js
export default () => "Hello world"
```

```
export const name = "Lydia"

// index.js
import * as data from "./module"

console.log(data)
```

- A: `{ default: function default(), name: "Lydia" }`
- B: `{ default: function default() }`
- C: `{ default: "Hello world", name: "Lydia" }`
- D: Global object of `module.js`

▼ 答案

答案: A

使用`import * as name`语法，我们将`module.js`文件中所有`export`导入到`index.js`文件中，并且创建了一个名为`data`的新对象。 在`module.js`文件中，有两个导出：默认导出和命名导出。 默认导出是一个返回字符串“Hello World”的函数，命名导出是一个名为`name`的变量，其值为字符串`“Lydia”`。

`data`对象具有默认导出的`default`属性，其他属性具有指定exports的名称及其对应的值。

---

90. 输出什么?

```
class Person {
  constructor(name) {
    this.name = name
  }
}

const member = new Person("John")
console.log(typeof member)
```

js

- A: `"class"`
- B: `"function"`
- C: `"object"`
- D: `"string"`

▼ 答案

答案: C

类是构造函数的语法糖，如果用构造函数的方式来重写 `Person` 类则将是：

```js
function Person() {
  this.name = name
}
```

通过 `new` 来调用构造函数，将会生成构造函数 `Person` 的实例，对实例执行 `typeof` 关键字将返回 `"object"` ，上述情况打印出 `"object"` 。

---

91. 输出什么?

```js
let newList = [1, 2, 3].push(4)

console.log(newList.push(5))
```

- A: `[1, 2, 3, 4, 5]`
- B: `[1, 2, 3, 5]`
- C: `[1, 2, 3, 4]`
- D: `Error`

▼ 答案

答案: D

`.push` 方法返回数组的长度，而不是数组本身！ 通过将 `newList` 设置为 `[1,2,3].push(4)` ，实际上 `newList` 等于数组的新长度： `4` 。

然后，尝试在 `newList` 上使用 `.push` 方法。 由于 `newList` 是数值 `4` ，抛出TypeError。

---

92. 输出什么?

```js
function giveLydiaPizza() {
  return "Here is pizza!"
}

const giveLydiaChocolate = () => "Here's chocolate... now go hit the gym already."
```

```
console.log(giveLydiaPizza.prototype)
console.log(giveLydiaChocolate.prototype)
```

- A: `{ constructor: ...}` `{ constructor: ...}`
- B: `{}` `{ constructor: ...}`
- C: `{ constructor: ...}` `{}`
- D: `{ constructor: ...}` `undefined`

▼ 答案

答案: D

常规函数，例如 `giveLydiaPizza` 函数，有一个 `prototype` 属性，它是一个带有 `constructor` 属性的对象（原型对象）。 然而，箭头函数，例如 `giveLydiaChocolate` 函数，没有这个 `prototype` 属性。 尝试使用 `giveLydiaChocolate.prototype` 访问 `prototype` 属性时会返回 `undefined` 。

---

93. 输出什么?

```js
const person = {
  name: "Lydia",
  age: 21
}

for (const [x, y] of Object.entries(person)) {
  console.log(x, y)
}
```

- A: `name` `Lydia` and `age` `21`
- B: `["name", "Lydia"]` and `["age", 21]`
- C: `["name", "age"]` and `undefined`
- D: `Error`

▼ 答案

答案: A

`Object.entries()` 方法返回一个给定对象自身可枚举属性的键值对数组，上述情况返回一个二维数组，数组每个元素是一个包含键和值的数组：

`[['name', 'Lydia'], ['age', 21]]`

使用 `for-of` 循环，我们可以迭代数组中的每个元素，上述情况是子数组。 我们可以使用 `const [x, y]` 在 `for-of` 循环中解构子数组。 `x` 等于子数组中的第一个元素， `y` 等于子数组中的第二个元素。

第一个子阵列是 `[“name”, “Lydia”]` ，其中 `x` 等于 `name` ，而 `y` 等于 `Lydia` 。 第二个子阵列是 `[“age”, 21]` ，其中 `x` 等于 `age` ，而 `y` 等于 `21` 。

---

94. 输出什么?

```js
function getItems(fruitList, ...args, favoriteFruit) {
  return [...fruitList, ...args, favoriteFruit]
}

getItems(["banana", "apple"], "pear", "orange")
```

- A: `["banana", "apple", "pear", "orange"]`
- B: `[["banana", "apple"], "pear", "orange"]`
- C: `["banana", "apple", ["pear"], "orange"]`
- D: `SyntaxError`

▼ **答案**

答案: D

`...args` 是剩余参数，剩余参数的值是一个包含所有剩余参数的数组，**并且只能作为最后一个参数**。上述示例中，剩余参数是第二个参数，这是不可能的，并会抛出语法错误。

```js
function getItems(fruitList, favoriteFruit, ...args) {
  return [...fruitList, ...args, favoriteFruit]
}

getItems(["banana", "apple"], "pear", "orange")
```

上述例子是有效的，将会返回数组： `[ 'banana', 'apple', 'orange', 'pear' ]`

95. 输出什么?

```js
function nums(a, b) {
  if
  (a > b)
  console.log('a is bigger')
  else
  console.log('b is bigger')
  return
  a + b
}

console.log(nums(4, 2))
console.log(nums(1, 2))
```

- A: `a is bigger`, `6` and `b is bigger`, `3`
- B: `a is bigger`, `undefined` and `b is bigger`, `undefined`
- C: `undefined` and `undefined`
- D: `SyntaxError`

▼ **答案**

答案: B

在JavaScript中，我们不必显式地编写分号(`;`)，但是JavaScript引擎仍然在语句之后自动添加分号。这称为**自动分号插入**。例如，一个语句可以是变量，或者像 `throw` 、 `return` 、 `break` 这样的关键字。

在这里，我们在新的一行上写了一个 `return` 语句和另一个值 `a + b` 。然而，由于它是一个新行，引擎并不知道它实际上是我们想要返回的值。相反，它会在 `return` 后面自动添加分号。你可以这样看:

```js
  return;
  a + b
```

这意味着永远不会到达 `a + b` ，因为函数在 `return` 关键字之后停止运行。如果没有返回值，就像这里，函数返回 `undefined` 。注意，在 `if/else` 语句之后没有自动插入!

---

96. 输出什么?

```js
class Person {
  constructor() {
    this.name = "Lydia"
  }
}

Person = class AnotherPerson {
  constructor() {
    this.name = "Sarah"
  }
}

const member = new Person()
console.log(member.name)
```

- A: `"Lydia"`
- B: `"Sarah"`
- C: `Error: cannot redeclare Person`
- D: `SyntaxError`

▼ 答案

答案: B

我们可以将类设置为等于其他类/函数构造函数。 在这种情况下，我们将 `Person` 设置为 `AnotherPerson` 。 这个构造函数的名字是 `Sarah` ，所以新的 `Person` 实例 `member` 上的name属性是 `Sarah` 。

---

97. 输出什么?

```js
const info = {
  [Symbol('a')]: 'b'
}

console.log(info)
console.log(Object.keys(info))
```

- A: `{Symbol('a'): 'b'}` and `["{Symbol('a')"]`
- B: `{}` and `[]`
- C: `{ a: "b" }` and `["a"]`

- D: `{Symbol('a'): 'b'}` and `[]`

▼ 答案

答案: D

`Symbol`类型是不可枚举的。`Object.keys`方法返回对象上的所有可枚举的键属性。`Symbol`类型是不可见的，并返回一个空数组。 记录整个对象时，所有属性都是可见的，甚至是不可枚举的属性。

这是`Symbol`的众多特性之一：除了表示完全唯一的值（防止对象意外名称冲突，例如当使用2个想要向同一对象添加属性的库时），您还可以`隐藏`这种方式对象的属性（尽管不完全。你仍然可以使用`Object.getOwnPropertySymbols()`方法访问`Symbol`。

---

98. 输出什么?

```js
const getList = ([x, ...y]) => [x, y]
const getUser = user => { name: user.name, age: user.age }

const list = [1, 2, 3, 4]
const user = { name: "Lydia", age: 21 }

console.log(getList(list))
console.log(getUser(user))
```

- A: `[1, [2, 3, 4]]` and `undefined`
- B: `[1, [2, 3, 4]]` and `{ name: "Lydia", age: 21 }`
- C: `[1, 2, 3, 4]` and `{ name: "Lydia", age: 21 }`
- D: `Error` and `{ name: "Lydia", age: 21 }`

▼ 答案

答案: A

`getList`函数接收一个数组作为其参数。 在`getList`函数的括号之间，我们立即解构这个数组。 您可以将其视为：

`[x, ...y] = [1, 2, 3, 4]`

使用剩余的参数`... y`，我们将所有剩余参数放在一个数组中。 在这种情况下，其余的参数是`2`，`3`和`4`。 `y`的值是一个数组，包含所有其余参数。 在这种情况下，`x`的值等于

`1`，所以当我们打印`[x, y]`时，会打印`[1, [2,3,4]]`。

`getUser`函数接收一个对象。对于箭头函数，如果只返回一个值，我们不必编写花括号。但是，如果您想从一个箭头函数返回一个对象，您必须在圆括号之间编写它，否则不会返回任何值!下面的函数将返回一个对象:

```
const getUser = user => ({ name: user.name, age: user.age })
```

由于在这种情况下不返回任何值，因此该函数返回`undefined`。

---

99. 输出什么?

```js
const name = "Lydia"

console.log(name())
```

- A: `SyntaxError`
- B: `ReferenceError`
- C: `TypeError`
- D: `undefined`

▼ **答案**

答案: C

变量`name`保存字符串的值，该字符串不是函数，因此无法调用。

当值不是预期类型时，会抛出`TypeErrors`。 JavaScript期望`name`是一个函数，因为我们试图调用它。 但它是一个字符串，因此抛出`TypeError`： `name is not a function`

当你编写了一些非有效的JavaScript时，会抛出语法错误，例如当你把`return`这个词写成`retrun`时。 当JavaScript无法找到您尝试访问的值的引用时，抛出`ReferenceErrors`。

---

100. 输出什么?

```js
// 🎉✨ This is my 100th question! ✨🎉

const output = `${[] && 'Im'}possible!
You should${'' && `n't`} see a therapist after so much JavaScript lol`
```

- A: `possible! You should see a therapist after so much JavaScript lol`
- B: `Impossible! You should see a therapist after so much JavaScript lol`
- C: `possible! You shouldn't see a therapist after so much JavaScript lol`
- D: `Impossible! You shouldn't see a therapist after so much JavaScript lol`

▼ 答案

答案: B

`[]`是一个真值。 使用`&&`运算符，如果左侧值是真值，则返回右侧值。 在这种情况下，左侧值`[]`是一个真值，所以返回`Im`。

`""`是一个假值。 如果左侧值是假的，则不返回任何内容。 `n't`不会被退回。

101.输出什么?

```js
const one = (false || {} || null)
const two = (null || false || "")
const three = ([] || 0 || true)

console.log(one, two, three)
```

- A: `false` `null` `[]`
- B: `null` `""` `true`
- C: `{}` `""` `[]`
- D: `null` `null` `true`

▼ 答案

答案: C

使用`||`运算符，我们可以返回第一个真值。 如果所有值都是假值，则返回最后一个值。

`(false || {} || null)`：空对象`{}`是一个真值。 这是第一个（也是唯一的）真值，它将被返回。 `one`等于`{}`。

`(null || false ||"")`：所有值都是假值。 这意味着返回传递的值`""`。 `two`等于`""`。

`([] || 0 ||"")`：空数组`[]`是一个真值。 这是第一个返回的真值。 `three`等于`[]`。