编程指南

目录

1. JavaScript

1.1 变量命名规则

当你创建变量时，需要按照 camelCase（第一个单词小写，所有后续单词都首字母大写）规则写出变量名称。同时尝试使用可以准确、简洁地描述数据内容的变量名称。

```
var totalAfterTax = 53.03; // uses camelCase if the variable name is multiple words
var tip = 8; // uses lowercase if the variable name is one word
```

变量名称未遵守 camelCase 规则也不会导致 JavaScript 代码出错，但是所有编程语言都存在建议的样式指南，可以使代码保持一致、简洁、易于阅读。尤其是在创建大型项目的时候要这么做，因为有很多开发人员会处理到同一项目。

1.2 索引

你是否知道你可以访问字符串中的每个字符？要访问单个字符，你可以使用字符在字符串中的位置，叫做索引。只需在字

```
var quote = "Stay awhile and listen!";
console.log(quote[6]);
```

此外，你可以使用字符串的 `charAt()` 方法访问单个字符。例如，`quote.charAt(6)` 也会返回 `"w"`。你将在这门课程

1.3 转义字符串

```
"The man whispered, "please speak to me.""

Uncaught SyntaxError: Unexpected identifier
```

如果你尝试在字符串中使用引号，系统将显示如上所示的 SyntaxError。

因为你需要使用引号来表示字符串的起始和末尾位置，JavaScript 对你的字符串理解会出错，认为该字符串的内容是 "The man whispered, "，然后后面的内容是please speak to me.""并返回SyntaxError`。

如果你想在字符串中使用引号，并且不希望 JavaScript 误解你的意图，则需要采用不同的方式来表示引号。幸运的是，JavaScript 可以采用反斜杠字符 (\ ) 来表示。

```
"The man whispered, \"please speak to me.\""
```

返回： `"The man whispered, "please speak to me.""`

1.4 特殊字符

引号并不是唯一需要转义的特殊字符，实际上有很多。但是，为了方便，请参阅 JavaScript 中的一些常见特殊字符列表。

代码	字符

代码	字符
\	\ (反斜杠)
"	" (双引号)
'	' (单引号)
\n	newline换行符
\t	tab制表符
&	和号
\	反斜杠
\r	回车符
\b	退格符
\f	换页符

上表中列出的最后两个字符：换行符 \n 和制表符 \t 是比较特殊，因为他们可以向字符串添加额外的空白。换行符将添加换行，制表符将使一行内容转到下一行制表位。

```
"Up up\n\tdown down"

返回：
Up up
 down down
```

1.5 Null、Undefined 和 NaN

null 表示“空值”，而 undefined 表示“缺少值”。NaN 表示“非数字”，通常返回表示数字运算存在错误。

```
var x = null;

var x;
console.log(x);
> undefined
```

1.6 隐式类型转换

JavaScript 属于对类型要求不高的语言。

基本上，意味着当你编写 JavaScript 代码的时候，不需要指定数据类型。相反，当 JavaScript 引擎解析代码的时候，将自动转换为“相应的”数据类型。这就叫做隐式类型转换，当你之前尝试将字符串与数字相连的时候，就已经见过类似的示例。

```
"julia" + 1
返回："julia1"
```

在此示例中，JavaScript 在字符串 "julia" 后面添加了数字 1，形成字符串 "julia1"。在其他编程语言中，这段代码可能就会返回错误，但是在 JavaScript 中，数字 1 转换成了字符串 "1"，然后与字符串 "julia" 相连。

这种行为使得 JavaScript 不同于其他编程语言，但是对多种数据类型进行运算和比较操作时，可能会产生奇怪的结果。

```
习题 1/2
你认为 "Hello" % 10 的值会是多少？

NaN
```

定义： 强类型语言是一种当数据与预期类型不完全相符时很有可能会产生错误的语言。因为 JavaScript 是松散类型，所以你不需要指定数据类型；但是，这样可能会产生因为隐式类型转换而导致的错误，并且难以诊断。

强类型编程语言代码的示例

```
int count = 1;
string name = "Julia";
double num = 1.2932;
float price = 2.99;
```

JavaScript 的对等代码

```
// equivalent code in JavaScript
var count = 1;
var name = "Julia";
var num = 1.2932;
var price = 2.99;
```

在下面的示例中，JavaScript 传入字符串"1"，并将其转换为 true，然后与布尔值 true 比较。

```
"1" == true
返回：true
```

当你使用 == 或 != 运算符时，JavaScript 首先将每个值转换为相同类型（如果不是相同类型的话）；因此叫做“转型”！这通常不是你希望出现的行为，比较值是否相等时使用 == 和 != 运算符并不是很好的做法。

1.7 绝对相等

相反，在 JavaScript 中，**最好使用绝对相等的方法看看数字、字符串或布尔型数值等在类型和值方面是否完全相同，而不用首先转换类型。要进行绝对比较，只需在 == 和 != 运算符的末尾添加一个 =。**

"1" === 1
返回：false

返回 false，因为字符串 "1" 和数字 1 并非具有相同的类型和值。

0 === false
返回：false

返回 false，因为数字 0 和布尔值 false 并非具有相同的类型和值。

习题 2/2
看看哪些表达式为 true。

"3" > 1 为 true，因为 3 大于 1（隐式类型转换）
true >= 0 为 true，因为 1 大于或等于 0（隐式类型转换）
1 !== false 为 true，因为 1 不等于 false（绝对相等）
3 === 3 为 true，因为 3 等于 3（绝对相等）

1.8 If...else 语句

If...else 语句使你能够根据是否满足一定的条件而执行特定的代码。

```
if (/* this expression is true */) {
  // run this code
} else {
  // run this code
}
```

这非常有用，因为你可以根据表达式的结果选择要执行的代码。例如：

```
var a = 1;
var b = 2;

if (a > b) {
  console.log("a is greater than b");
} else {
  console.log("a is less than or equal to b");
}
输出："a is less than or equal to b"
```

关于 if...else 语句的几个注意事项。

if 语句中的值始终转换为 true 或 false。根据该值，if 语句中的代码会运行，或 else 语句中的代码会运行。if 和 else 语句中的代码位于花括号 {...} 里，以便区分条件，并表明要运行哪段代码。

提示：编程时，有时候你可能只想使用 if 语句。但是，如果你尝试只使用 else 语句，就会出现错误 SyntaxError: Unexpected token else。出现此错误是因为 else 语句需要 if 语句才能运行。没有 if 语句的话，则不能使用 else 语句。

Else if 语句

在 JavaScript 中，你可以再用一个 if 语句来表示第二个检查条件，叫做 else if 语句。

```
var weather = "sunny";

if (weather === "snow") {
  console.log("Bring a coat.");
} else if (weather === "rain") {
  console.log("Bring a rain jacket.");
} else {
  console.log("Wear what you have on.");
}
输出：Wear what you have on.
```

通过额外添加一个 else if 语句，就增加了一个条件语句。

如果不下雪，代码就会跳到 else if 语句，看看是否下雨。 如果不下雨，代码就会跳到 else 语句。

本质上 else 语句充当的是默认条件，以防所有其他 if 语句都为 false。

逻辑表达式

逻辑表达式类似于数学表达式，但是逻辑表达式的结果是 true 或 false。

```
11 != 12
返回：true
```

当你写比较代码时，已经见过逻辑表达式了。比较是简单的逻辑表达式。

就像数学表达式使用 +、-、*、/ 和 % 一样，你也可以使用逻辑运算符创建更复杂的逻辑表达式。

逻辑运算符

逻辑运算符可以与布尔值（true 和 false）结合使用，创建复杂的逻辑表达式。

将两个布尔值与逻辑运算符相结合，可以创建返回另一个布尔值的逻辑表达式。下面的表格描述了不同的逻辑运算符：

运算符	含义	示例	使用方法
&&	逻辑 AND	value1 && value2	如果 value1 和 value2 都为 true，则返回 true
||	逻辑 OR	value1 || value2	如果 value1 或 value2（甚至二者！）为 true，则返回 true
!	逻辑 NOT	!value1	返回 value1 的相反值。如果 value1 为 true，则 !value1 为 false。

真值和假值

JavaScript 中的每个值都有固有的布尔值，在布尔表达式中评估该值时，该值就会转换为固有的布尔值。

这些固有的值称为真值或假值。

假值

结果为 false 的值称为 假值。例如，空字符串 "" 为假值，因为在布尔表达式中，"" 等于 false。

```
false == ""
返回：true
```

以下是所有假值的列表：

```
false
null
undefined
0
NaN
""
```

真值

如果结果为 true，则为真值。例如，1 是真值，因为在布尔环境下，1 等于 true。

```
true == 1
返回：true
```

以下是真值的一些其他示例：

```
true
42
"pizza"
{}
[]
```

本质上，如果不是假值，则为真值！

有时候，你可能会遇到以下类型的条件语句。

```
var isGoing = true;
var color;

if (isGoing) {
  color = "green";
} else {
  color = "red";
}

console.log(color);
```

> 输出："green"

在此示例中，变量 color 根据 isGoing 的值被赋为 "green" 或 "red"。这段代码可以运行，但是这么为变量赋值显得有点冗长。幸运的是，JavaScript 提供了其他方式。

提示： 使用 if(isGoing) 和使用 if(isGoing === true) 是一样的。此外，使用 if(!isGoing) 和使用 if(isGoing === false) 是一样的。

三元运算符

三元运算符可以帮助避免写出冗长的 if...else 语句。

conditional ? (if condition is true) : (if condition is false)
要使用三元运算符，首先提供 ? 左侧的条件语句。然后，在 ? 和 : 之间写出条件为 true 时将运行的代码，并在 : 右侧写出条件为 false 的代码。例如，你可以将上述示例代码重写为：

```
var isGoing = true;
var color = isGoing ? "green" : "red";
console.log(color);
```

> 输出："green"
> 此代码不仅替换了条件语句，还处理了 color 的变量赋值。

分解代码后会发现，条件 isGoing 位于 ? 的左侧。然后，如果条件为 true，那么 ? 后的第一个表达式将运行，如果条件为 false，则 : 后的第二个表达式将运行。

练习题

如果运行以下代码，将会向控制台输出什么内容？

```
var adult = true;
var preorder = true;

console.log("It costs $" + (adult ? "40.00" : "20.00") + " to attend the concert. Pick up your t

It costs $20.00 to attend the concert. Pick up your tickets at the gate.

It costs $20.00 to attend the concert. Pick up your tickets at the will call.

It costs $40.00 to attend the concert. Pick up your tickets at the gate.
```

Switch 语句

如果你的代码中重复出现 else if，每个条件都是基于相同的值，那么也许可以使用 switch 语句。

```
if (option === 1) {
  console.log("You selected option 1.");
} else if (option === 2) {
  console.log("You selected option 2.");
} else if (option === 3) {
  console.log("You selected option 3.");
} else if (option === 4) {
  console.log("You selected option 4.");
} else if (option === 5) {
  console.log("You selected option 5.");
} else if (option === 6) {
  console.log("You selected option 6.");
}
```

switch 语句是另一种将多个基于相同值的 else if 语句放到一起，并且不用使用条件语句的方式，只是根据某个值切换每段代码。

```
switch (option) {
  case 1:
    console.log("You selected option 1.");
  case 2:
    console.log("You selected option 2.");
  case 3:
    console.log("You selected option 3.");
  case 4:
    console.log("You selected option 4.");
  case 5:
    console.log("You selected option 5.");
  case 6:
    console.log("You selected option 6.");
}
```

这里，每个 else if 语句 (option === [value]) 被替换成了 case 条件 (case: [value])，这些条件封装在了 switch 语句中。

switch 语句一开始查看代码时，会查看第一个表达式等于传递给 switch 语句的结果的 case 条件。然后，将控制权转给该 case 条件，执行相关的语句。

所以，如果将 option 设为 3...

```
var option = 3;

switch (option) {
  ...
}
Prints:
You selected option 3.
You selected option 4.
You selected option 5.
You selected option 6.
```

...然后 switch 语句输出选项 3、4、5 和 6。

但是和顶部的原始 if...else 代码不完全一样吧？缺少了什么？

中断语句

中断语句可以用来终结一个 switch 语句并将控制权转给被终结语句后面的代码。向每个 case 条件添加 break 后，就解决了 switch 语句不断往下跳转到其他 case 条件的问题。

```
var option = 3;

switch (option) {
  case 1:
    console.log("You selected option 1.");
    break;
  case 2:
    console.log("You selected option 2.");
    break;
  case 3:
    console.log("You selected option 3.");
    break;
  case 4:
    console.log("You selected option 4.");
    break;
  case 5:
    console.log("You selected option 5.");
    break;
  case 6:
    console.log("You selected option 6.");
    break; // technically, not needed
}
```

输出：You selected option 3.

练习题

下面的 switch 语句结果是多少？

```
var month = 2;

switch(month) {
  case 1:
  case 3:
  case 5:
  case 7:
  case 8:
  case 10:
  case 12:
    days = 31;
    break;
  case 4:
  case 6:
  case 9:
  case 11:
    days = 30;
    break;
  case 2:
    days = 28;
}

console.log("There are " + days + " days in this month.");
```

"There are 28 days in this month"

在某些情况下，可以利用 switch 语句的“传递”行为。

例如，当你的代码结构是层级结构时。

```
var tier = "nsfw deck";
var output = "You'll receive "

switch (tier) {
  case "deck of legends":
    output += "a custom card, ";
  case "collector's deck":
    output += "a signed version of the Exploding Kittens deck, ";
  case "nsfw deck":
    output += "one copy of the NSFW (Not Safe for Work) Exploding Kittens card game and ";
  default:
    output += "one copy of the Exploding Kittens card game.";
}

console.log(output);
```

> 输出：You'll receive one copy of the NSFW (Not Safe for Work) Exploding Kittens card game and one copy of the Exploding Kittens card game.

> 在此示例中，根据成功的 Exploding Kittens Kickstarter 活动（由 Elan Lee 创建的一款有趣的桌游），每个后继层级通过添加更多的输出在下个层级上继续扩建。代码中没有任何中断语句的话，switch 语句跳到 "nsfw deck" 后，继续向下传递，直到到达 switch 语句的结尾。

同时，注意 default case。

```
var tier = "none";
var output = "You'll receive ";

switch (tier) {
  ...
  default:
    output += "one copy of the Exploding Kittens card game.";
}

console.log(output);
```

> 输出：You'll receive one copy of the Exploding Kittens card game.
> 你可以向 switch 语句中添加 default case，当没有任何与 switch 表达式相符的值时，将执行该语句。

循环

While 循环的各个部分

循环有各种类型，但是它们本质上都实现相同的操作：重复一定次数地执行特定操作。

任何循环都具有以下三大部分：

何时开始：设置循环的代码 — 例如定义某个变量的起始值。
何时停止：测试循环是否继续的逻辑条件。
如何转到下一项： 递增或递减步骤 — 例如，x = x * 3 或 x = x - 1
以下是包含所有这三部分的基本 while 循环。

```
var start = 0; // when to start
while (start < 10) { // when to stop
  console.log(start);
  start = start + 2; // how to get to the next item
}
```

> 输出：
> 0
> 2
> 4

> 6
> 8
> 如果循环缺少这三个部分的任一部分，那么就会出现问题。例如，缺少停止条件会导致循环一直执行下去！

千万别运行这段代码！

```
while (true) {
  console.log("true is never false, so I will never stop!");
}
```

如果你真的在控制台中运行了上述代码，你的浏览器标签页可能会崩溃。

> 警告：你之所以看到这段内容，是因为你没有聆听我们的警告，在控制台中运行了这段无限循环的代码。如果你的浏览器标签页崩溃了或卡住/不响应，可以采取几个方法来解决这一问题。如果你使用的是 Firefox，浏览器将弹出一个通知，告诉你脚本不响应，并给出终止该脚本的选项（执行这一选项）。如果你使用的是 Chrome，转到任务栏，并依次选择 Windows > 任务管理器。你可以通过任务管理器终止运行该脚本的特定标签页进程。如果你使用的不是 Firefox 或 Chrome，下载 Firefox 或 Chrome 吧;)

For 循环的各个部分

for 循环明确要求定义起始点、结束点和循环的每一个步骤。实际上，如果缺少这三个部分的任一部分，系统都会提示Uncaught SyntaxError: Unexpected token)。

```
for ( start; stop; step ) {
  // do this thing
}
```

下面这个 for 循环输出了 0-5 的每个值。注意区分 for 循环不同语句的分号：var i = 0; i < 6; i = i + 1

```
for (var i = 0; i < 6; i = i + 1) {
  console.log("Printing out i = " + i);
}
```

> Prints:
> Printing out i = 0
> Printing out i = 1
> Printing out i = 2
> Printing out i = 3

> Printing out i = 4
> Printing out i = 5

嵌套循环

你是否知道你还可以在循环里嵌套其他循环？请将以下嵌套循环粘贴到浏览器中，看看输出结果是多少：

```
for (var x = 0; x < 5; x = x + 1) {
  for (var y = 0; y < 3; y = y + 1) {
    console.log(x + "," + y);
  }
}
```

> 输出：
> 0, 0
> 0, 1
> 0, 2
> 1, 0
> 1, 1
> 1, 2
> 2, 0
> 2, 1
> 2, 2
> 3, 0
> 3, 1
> 3, 2
> 4, 0
> 4, 1
> 4, 2

注意输出内容的顺序。

对于外部循环里的每个 x 值，内部的 for 循环都完全执行了。外部循环从 x = 0 开始，然后内部循环执行完 y 的所有值：

```
x = 0 and y = 0, 1, 2 // corresponds to (0, 0), (0, 1), and (0, 2)
```

内部循环访问完 y 后，外部循环继续变成下一个值，即 x = 1，整个流程再执行一遍。

```
x = 0 and y = 0, 1, 2 // (0, 0) (0, 1) and (0, 2)
x = 1 and y = 0, 1, 2 // (1, 0) (1, 1) and (1, 2)
x = 2 and y = 0, 1, 2 // (2, 0) (2, 1) and (2, 2)
etc.
```

递增和递减

以下是到目前为止学到的运算符的总结内容：

```
x++ or ++x // same as x = x + 1 ++x但是先加再循环
x-- or --x // same as x = x - 1
x += 3 // same as x = x + 3
x -= 6 // same as x = x - 6
x *= 2 // same as x = x * 2
x /= 5 // same as x = x / 5
```

如何声明函数

函数使你能够将一段代码封装起来，并在程序中使用（经常会重复利用）。

有时候，函数具有参数，例如这节课开头部分的 pizza 按钮。reheatPizza() 具有一个参数：披萨块数。

```
function reheatPizza(numSlices) {
  // code that figures out reheat settings!
}
```

你还见过的 reverseString() 函数具有一个参数：要倒转的字符串。

```
function reverseString(reverseMe) {
  // code to reverse a string!
}
```

在这两种情况下，参数都作为变量在花括号里列在函数名称后面。此外，如果有多个参数，直接用逗号分隔就行。

```
function doubleGreeting(name, otherName) {
  // code to greet two people!
}
```

但是，函数也可以没有任何参数。直接封装一些代码并执行某项任务。在这种情况下，直接将小括号留空就行了。例如，下面这个函数直接输出 "Hello!"。

```
// accepts no parameters! parentheses are empty
function sayHello() {
  var message = "Hello!";
  console.log(message);
}
```

如果你将上述任何一个函数复制到 JavaScript 控制台中，可能不会注意到任何情况。实际上，可能会看到返回了 undefined。当控制台无法明确返回任何内容时，使用特殊的 return 关键字就会出现默认的 undefined 返回值。

返回语句

在上述 sayHello() 函数中，我们使用 console.log 向控制台输出了值，但是没有用返回语句明确返回内容。你可以使用关键字 return，后面紧跟着你要返回的表达式或值写一个返回语句。

```
// declares the sayHello function
function sayHello() {
  var message = "Hello!";
  return message; // returns value instead of printing it
}
```

如何运行函数

现在，为了让函数执行任务，你需要调用函数，方法是使用函数名称，后面是小括号，其中包含任何传入的参数。函数就像机器，你可以构造机器，但是如果不开机的话，机器肯定不能运转。下面的示例演示了如何调用之前的 sayHello() 函数，然后将返回值输出到控制台中：

```
// declares the sayHello function
function sayHello() {
  var message = "Hello!";
  return message; // returns value instead of printing it
}

// function returns "Hello!" and console.log prints the return value
console.log(sayHello());
```

> 输出：Hello!

Parameter 与 Argument

一开始很难判断某项内容是 parameter 还是 argument。关键区别在于它们出现在代码中的何处。parameter 始终是变量名称，并出现在函数声明中。相反，argument 始终是一个值（即任何 JavaScript 数据类型：数字、布尔值等），并且始终出现在函数调用代码中。

返回与日志

请务必明白返回和输出并不是一回事。向 JavaScript 控制台输出值仅显示一个值（你可以查看该值并调试代码），但是该值的作用也仅此而已。因此，务必仅使用 console.log 在 JavaScript 控制台中测试你的代码。

请将以下函数声明和函数调用代码粘贴到 JavaScript 控制台中，看看日志（输出）和返回之间的区别：

```
function isThisWorking(input) {
  console.log("Printing: isThisWorking was called and " + input + " was passed in as an argument
  return "Returning: I am returning this string!";
}

isThisWorking(3);
```

> 输出： "Printing: isThisWorking was called and 3 was passed in as an argument"
> 返回： "Returning: I am returning this string!"

如果你没有明确定义返回值，函数默认地将返回 undefined。

```
function isThisWorking(input) {
  console.log("Printing: isThisWorking was called and " + input + " was passed in as an argument
}
isThisWorking(3);
```

> 输出： "Printing: isThisWorking was called and 3 was passed in as an argument"
> 返回： undefined

使用返回值

函数能够返回值很不错，但是如果不使用该值执行某项操作，又有什么用呢？

函数的返回值可以存储在变量中或在整个程序期间作为参数使用。此处有个将两个数字相加的函数，另一个函数将数字除以 2。我们可以通过以下方法求出 5 和 7 的平均值：使用 add() 函数将一组数字相加，然后将 add(5, 7) 求出的两个数字的和作为参数传入函数 divideByTwo() 中。

最后，我们甚至可以将最终答案存储到变量 average 中，并使用该变量执行更多的运算！

```
// returns the sum of two numbers
function add(x, y) {
  return x + y;
}


// returns the value of a number divided by 2
function divideByTwo(num) {
  return num / 2;
}


var sum = add(5, 7); // call add function is stored in the sum variable
var average = divideByTwo(sum); // call divideByTwo function and store in answer v
```