值结果确实是 true 或者 false。

你还可以提供一个用于 if 条件不为真时的选择，我们将其称为 else 语句。考虑：

```
const ACCESSORY_PRICE = 9.99;

var bank_balance = 302.13;
var amount = 99.99;

amount = amount * 2;

// 是否可以提供额外的购买?
if ( amount < bank_balance ) {
     console.log( "I'll take the accessory!" );
     amount = amount + ACCESSORY_PRICE;
}
// 否则:
else {
    console.log( "No, thanks." );
}
```

在以上的示例中，如果 amount < bank_balance 为真，那么就会打印出 "I'll take the accessory!"，并在变量 amount 上加上 9.99。否则，else 语句就会礼貌地回答 "No, thanks."，并保持 amount 不变。

正如我们在 1.5 节中讨论的那样，不满足期望类型的值通常会被强制转换为需要的类型。if 语句需要布尔型的值，如果传递的值是非布尔型的，那么就会发生类型转换。

JavaScript 定义了一系列特定的值，这些值在强制转换为布尔型时会被认为是“假的”，它们会转化为 false，其中包括 0 和 "" 这样的值。任何不在这个列表中的其他值会自动成为“真的”，因此在强制转换为布尔型时会转化为 true。真值包括 99.99 和 "free" 这样的值。要想获得更多信息，参见 2.1.3 节中的“真与假”部分。

除 if 之外，还有其他形式的**条件判断**。比如，switch 语句可以用作一组 if..else 语句的简写（参见第 2 章）。循环（参见 1.10 节）通过一个**条件判断**来确定是继续还是停止。

有关测试条件判断时可能隐式发生的类型转换的更多信息，参见本系列《你不知道的 JavaScript（中卷）》第一部分中的第 4 章。

1.10 循环

商店忙碌时会有一个等候服务的顾客队列。如果这个队列中还有顾客，那么店员就要继续为下一位顾客服务。

重复一系列动作，直到不满足某个条件，换句话说，重复只发生在满足条件的情况下，这

就是程序循环的工作；循环有多种形式，但都满足基本的行为特性。

循环包括测试条件以及一个块（通常就是 `{ .. }`）。循环块的每次执行被称为一个**迭代**。

比如，`while` 循环和 `do..while` 循环形式展示了重复一个语句块直到一个条件判断求值不再为真这个概念：

```
while (numOfCustomers > 0) {
    console.log( "How may I help you?" );

    // 帮助顾客……

    numOfCustomers = numOfCustomers - 1;
}

// 对比：

do {
    console.log( "How may I help you?" );

    // 帮助顾客……

    numOfCustomers = numOfCustomers - 1;
} while (numOfCustomers > 0);
```

这些循环之间的唯一实际区别是，条件判断在第一次迭代执行前（`while`）检查还是在第一次迭代后（`do..while`）检查。

不管是哪种形式，如果条件判断测试结果为 `false`，那么都不会运行下一轮迭代。这意味着，如果第一次的条件判断为 `false`，那么 `while` 循环就不会执行，而 `do..while` 循环只会运行一次。

有时循环要在一组数字上迭代，比如从 `0` 到 `9`（10 个数字）。你可以设置一个 `i` 这样的循环迭代变量，初始为 `0`，然后每次迭代增加 `1`。

出于多种历史原因，编程语言几乎总是从零开始计数，也就是说，是从 `0` 而不是从 `1` 开始。如果不熟悉这种思维模式的话，一开始可能会感到非常迷惑。花点时间进行从 `0` 开始的计数训练，并适应这一点。

条件判断会在每次迭代时测试，这就好像是在循环内部有一个隐式的 `if` 语句。

我们可以通过 JavaScript 的 `break` 语句来结束循环。另外，我们也可以看到，很容易就会创建出一个如果不使用 `break` 机制就会陷入死循环的循环。

举例来说：

```
var i = 0;
```

```
// while..true循环会永久运行，不是吗？
while (true) {
    // 停止循环？
    if ((i <= 9) === false) {
        break;
    }

    console.log(i);
    i = i + 1;
}
// 0 1 2 3 4 5 6 7 8 9
```

这并非是你在自己的程序中一定要效仿的一个实际循环形式。在这里展示只是为了说明问题。

虽然使用 `while`（或者 `do..while`）循环也可以手动完成任务，但还有一个专门为此设计的语法形式，我们将其称为 `for` 循环：

```
for (var i = 0; i <= 9; i = i + 1) {
    console.log( i );
}
// 0 1 2 3 4 5 6 7 8 9
```

在上述的示例中可以看到，两种情况下条件 `i <= 9` 对于前十次迭代都为 `true`，但当 `i` 为 `10` 时会变为 `false`。

`for` 循环有 3 个分句：初始化分句（`var i = 0`）、条件测试分句（`i <= 9`），以及更新分句（`i = i + 1`）。所以，如果你需要在循环迭代中计数，那么最紧凑、最容易理解和编写的形式就是 `for` 循环。

还有其他一些特殊的循环形式，专门用于在特定的值上迭代，比如对象属性（参见第 2 章），其中隐式的测试条件为是否所有的属性都已经处理完毕。无论循环的形式是什么，“循环到条件为否”这个概念是保持不变的。

1.11 函数

手机商店的店员应该不会随身携带计算器来计算税费和最后的应付金额。这个任务是她需要定义一次并多次复用的。很有可能的是，公司的收银设备（计算机或平板等）内置了这样的“函数”。

类似地，你的程序也几乎总是需要将代码的任务分割成可复用的片段，而不是一直重复编码。实现这一点的方法就是定义一个**函数**。

通常来说，函数是可以通过名字被“调用”的已命名代码段，每次调用，其中的代码就会运行。考虑：

```
function printAmount() {
    console.log( amount.toFixed( 2 ) );
}

var amount = 99.99;

printAmount(); // "99.99"

amount = amount * 2;

printAmount(); // "199.98"
```

函数可以接受参数，即你传入的值，也可以返回一个值：

```
function printAmount(amt) {
    console.log( amt.toFixed( 2 ) );
}

function formatAmount() {
    return "$" + amount.toFixed( 2 );
}

var amount = 99.99;

printAmount( amount * 2 );   // "199.98"

amount = formatAmount();
console.log( amount );    // "$99.99"
```

函数 `printAmount(..)` 接受一个名为 `amt` 的参数。函数 `formatAmount()` 返回一个值。你也可以在同一个函数中同时使用这两种技术。

通常来说，你会在计划多次调用的代码上使用函数，但只是将相关的代码组织到一起成为命名集合也是很有用的，即使只准备调用一次。

考虑：

```
const TAX_RATE = 0.08;

function calculateFinalPurchaseAmount(amt) {
    // 根据税费来计算新的数值
    amt = amt + (amt * TAX_RATE);

    // 返回新的数值
    return amt;
}

var amount = 99.99;
```

```
amount = calculateFinalPurchaseAmount( amount );

console.log( amount.toFixed( 2 ) );      // "107.99"
```

尽管 `calculateFinalPurchaseAmount(..)` 只被调用了一次，但将它的行为组织到一个独立的命名函数中使得使用其逻辑（`amount = calculateFinal...` 语句）的代码更为清晰。函数中的语句越多，其效果就会越明显。

作用域

如果你向手机商店的店员询问的手机型号是该商店里没有的，那么她也就无法将你想要的手机卖给你。她只能接触到商店里现有的手机。你也就不得不到另外一家商店尝试看看能否找到你想要的手机型号了。

编程中的一个术语可以表示这个概念：**作用域**（严格说是**词法作用域**）。在 JavaScript 中，每个函数都有自己的作用域。作用域基本上是变量的一个集合以及如何通过名称访问这些变量的规则。只有函数内部的代码才能访问这个函数作用域中的变量。

同一个作用域内的变量名是唯一的，所以不能有两个变量 `a` 一个接一个地放在一起。但是，同一个变量名 `a` 可以出现在不同的作用域中：

```
function one() {
    // 这个a只属于one()函数
    var a = 1;
    console.log( a );
}

function two() {
    // 这个a只属于two()函数
    var a=2;
    console.log( a );
}

one();      // 1
two();     // 2
```

此外，作用域是可以彼此嵌套的，就好像生日聚会上的小丑可以在一个气球内部吹起另一个气球那样。如果一个作用域嵌套在另外一个作用域内，那么内层作用域中的代码可以访问这两个作用域中的变量。

考虑：

```
function outer() {
    var a = 1;

    function inner() {
        var b = 2;
```

```
        // 这里我们既可以访问a，也可以访问b
        console.log( a + b );    // 3
    }

    inner();

    // 这里我们只能访问a
    console.log( a );         // 1
}

outer();
```

词法作用域的规则表明，一个作用域内的代码可以访问这个作用域内以及任何包围在它之外的作用域中的变量。

因此，`inner()` 函数内部的代码可以访问变量 `a` 和 `b`，但是 `outer()` 中的代码只能访问 `a`，不能访问 `b`，因为这个变量 `b` 只在 `inner()` 函数内部。

我们来回顾一下前面的代码片段：

```
const TAX_RATE = 0.08;

function calculateFinalPurchaseAmount(amt) {
    // 根据税费来计算新的数值
    amt = amt + (amt * TAX_RATE);

    // 返回新的数值
    return amt;
}
```

因为词法作用域的缘故，我们可以从函数 `calculateFinalPurchaseAmount(..)` 的内部访问常量（变量）`TAX_RATE`，尽管我们并没有将它传递进去。

有关词法作用域的更多信息，参见本系列《你不知道的 JavaScript（上卷）》第一部分中的前三章。

1.12 实践

在编程学习中，实践是绝对无法替代的。无论我如何在这里阐释说明，理论都无法让你成为一个程序员。

谨记这一点。我们来尝试针对在本章中学习到的概念进行一些练习。我会给出“需求”，你先试着解决这些“需求”。然后查看以下列出的代码，看看我是如何解决的。

- 编写程序以计算购买手机所需的总金额。你需要一直购买手机（提示：循环！）直到银行账号中的资金不足。而且，只要价格低于你的心理预期值，那么就要为每个手机购买附件。

- 计算总金额后再加上税费，然后以适当的格式打印出计算出的总金额。
- 最后，检查银行账号的余额，确认是否能买得起。
- 需要为“税率”、“手机价格”、“附件价格”和“预算阈值”建立一些常量，为“银行账号的余额”建立变量。
- 你应该定义一些函数来计算税费，格式化价格加上“$”符号并保留两位小数。
- **附加题**：试着在这个程序中集成输入，你可以使用 1.3.2 节中介绍的 `prompt(..)`。比如，你可以提示用户输入他们的银行账号余额。享受吧，发挥你的创造力！

好了，你现在可以开始实践了。在你自己尝试之前不要先看我的代码！

因为本书是一本关于 JavaScript 的书，显然我会使用 JavaScript 来完成这个练习。但你也可以根据个人意愿而使用其他语言来实现。

以下是我针对上述练习而设计的 JavaScript 解决方案：

```
const SPENDING_THRESHOLD = 200;
const TAX_RATE = 0.08;
const PHONE_PRICE = 99.99;
const ACCESSORY_PRICE = 9.99;

var bank_balance = 303.91;
var amount = 0;

function calculateTax(amount) {
    return amount * TAX_RATE;
}

function formatAmount(amount) {
    return "$" + amount.toFixed( 2 );
}

// 如果还有余额，那么继续购买手机
while (amount < bank_balance) {
    // 购买新的手机！
    amount = amount + PHONE_PRICE;

    // 是否可以负担得起附件？
    if (amount < SPENDING_THRESHOLD) {
        amount = amount + ACCESSORY_PRICE;
    }
}

// 别忘了交税
amount = amount + calculateTax( amount );

console.log(
    "Your purchase: " + formatAmount( amount )
```

```
);
// 你的购买金额：$334.76

// 你真的可以负担得起本次购买吗？
if (amount > bank_balance) {
    console.log(
        "You can't afford this purchase. :("
    );
}
// 你无法负担本次购买。 :(
```

运行这个 JavaScript 程序最简单的方法是，将其输入到你手边浏览器的开发者终端中。

你是如何实现的呢？你现在已经看到了我的代码，不妨再试一下。你可以修改其中的一些常量，看看这个程序在不同的值之下是怎么运行的。

1.13 小结

学习编程并不必然是复杂、费力的过程。你需要熟悉几个基本的概念。

这些概念如同积木。要想构造高塔，首先要将积木一层层摞在一起。编程也是如此。以下是一些核心的编程积木块。

- 在值上执行动作需要**运算符**。
- 执行各种类型的动作需要**值和类型**，比如，对数字进行数学运算，用字符串输出。
- 在程序的执行过程中需要**变量**来保存数据（也就是**状态**）。
- 需要 `if` 这样的条件判断来作出决策。
- 需要循环来重复任务，直到不满足某个条件。
- 需要**函数**将代码组织为逻辑上可复用的块。

代码注释是编写可读代码的一种有效方法，能让你的代码更易于理解和维护，如果以后出现问题的话也更加容易进行修复。

最后，不要忽略练习的威力，学习如何编写代码的最好方法就是不断编写代码。

很高兴你已经在开始学习如何编码的路上了！继续前进吧！不要忘了查阅其他的编程入门资源（书籍、博客和在线培训等）。本章和本部分就是一个很好的起点，但它们仅仅只是一个概要介绍。

下一章将会介绍本章出现的多个概念，但会从 JavaScript 的角度来解释，这会强调多个主要的主题，这些主题是在本系列其他书中深入详细介绍的。

第 2 章

深入 JavaScript

我们在前一章中介绍了编程的基本组件，如变量、循环、条件判断和函数。当然，其中展示的所有代码都是用 JavaScript 语言编写的。而本章的主要关注点是作为 JavaScript 开发者在开始编写 JavaScript 代码时所需要了解的知识。

我们将在本章中介绍很多概念，这些概念需要阅读其他的“你不知道的 JavaScript”系列图书才能完全掌握。你可以将本章看作本系列其他图书将要详细介绍的主题的概论。

尤其重要的一点是，如果你还只是 JavaScript 方面的新手，那么应该多花点时间查看这些概念并反复练习本章中的示例代码。坚固的基础都是一点一点构造起来的，因此不要期望第一次阅读就能够马上完全理解这些概念。

深入学习 JavaScript 的旅程这就开始了。

正如我在第 1 章中所说的，在阅读和学习本章的过程中，你绝对应该亲自试验一下所有的代码示例。记住，本章中的部分代码假定了编写本部分时 JavaScript 最新版本（JavaScript 规范的官方名称 ECMAScript 第 6 版，一般称为“ES6”）中引入的功能。如果你恰好在使用 ES6 前的旧版浏览器，那么这些代码可能无法正常运行。你应该使用浏览器（如 Chrome、Firefox 或 IE 等）的更新版本。

2.1 值与类型

我们在第 1 章中已经提到过，JavaScript 的值有类型，但变量无类型。以下是可用的内置类型：

- 字符串
- 数字
- 布尔型
- null 和 undefined
- 对象
- 符号（ES6 中新增的）

JavaScript 提供了一个 typeof 运算符，该运算符可以用来查看值的类型：

```
var a;
typeof a;          // "undefined"

a = "hello world";
typeof a;          // "string"

a=42;
typeof a;          // "number"

a = true;
typeof a;          // "boolean"

a = null;
typeof a;          // "object"--诡异，这是bug

a = undefined;
typeof a;          // "undefined"


a={b:"c"};
typeof a;          // "object"
```

typeof 运算符的返回值永远是这 6 个（对 ES6 来说是 7 个）字符串值之一。也就是说，typeof "abc" 返回 "string"，而不是 string。

请注意这段代码中的变量是如何持有多个不同类型的值的，和表面看起来不同，typeof 并不是在询问“a 的类型”，而是“a 中当前值的类型”。在 JavaScript 中，只有值有类型；变量只是这些值的容器。

typeof null 是一个有趣的示例，你期望它返回的会是 "null"，但它返回的却是 "object"，这大概会让你觉得很意外。

这是 JavaScript 中存在已久的一个 bug，也似乎是一个永远不会被修复的 bug。Web 上的太多代码都依赖于这个 bug，因此，修复它会导致大量的新 bug！

另外，注意 a = undefined。我们显式地将 a 的值设置为 undefined，但从行为上来说，这