# Misc.
# Tools

## Comparison Operators

|  |  |
|---|---|
| `>` | Greater than |
| `<` | Less than |
| `<=` | Less than or equal to |
| `>=` | Greater than or equal to |
| `===` | Equal to |
| `!==` | **Not** equal to |

## Math and the modulo

Let's meet an interesting symbol called **modulo**. When `%` is placed between two numbers, the computer will divide the first number by the second, and then return the **remainder** of that division.

So if we do `23 % 10`, we divide 23 by 10 which equals 2 with 3 left over. So `23 % 10` evaluates to `3`.

**More examples**:

`17 % 5` evaluates to 2

`13 % 7` evaluates to 6

### ISNAN

If you call `isNaN` on something, it checks to see if that thing *is not* a number. So:

```
    isNaN('berry'); // => true
    isNaN(NaN); // => true
    isNaN(undefined); // => true
    isNaN(42);  // => false
```

## Confirm & Prompt

```
confirm("I feel awesome!");
confirm("I am ready to go.");
```

These boxes can be used on websites to*confirm* things with users.
You've probably seen them pop up when you try to delete important things or leave a website with unsaved changes.
Prompts are used to take an input from the user that is returned to the console.

```
prompt("Are you ok?");
```

## Return keyword

Nice job! Now, when we call a function, we don't always want to just print stuff. Sometimes, we just want it to `return` a value. We can then use that value (ie. the output from the function) in other code. Let's learn about the `return` keyword, then we'll see how to use functions with an if / else statement in the next exercise!
The `return` keyword simply gives the programmer back the value that comes out of the function. So the function runs, and when the `return` keyword is used, the function will immediately stop running and `return` the value.

## logical Operators.

**And** (&&)

The logical operator **and** is written in JavaScript like this: `&&`. It evaluates to `true` when *both* expressions are `true`; if they're not, it evaluates to `false`.

```
true && true;     // => true
true && false;    // => false
false && true;    // => false
false && false;   // => false
```

**Or** (||)
The logical operator **or** is written in JavaScript like this: `||`. It evaluates to `true` when *one or the other or both* expressions are `true`; if they're not, it evaluates to `false`.

```
true || true;      // => true
true || false;     // => true
false || true;     // => true
false || false;    // => false
```

The **or** operator is written with two vertical bars `||`. The vertical bar character is located right above the Enter key on your keyboard.

**Not** (!)

The logical operator **not** is written in JavaScript like this: `!`. It makes `true` expressions `false`, and vice-versa.

```
!true;    // => false
!false;   // => true
```

## Data Types

## Data Types (attributes)

Data comes in various **types**.
**a. numbers** are quantities, just like you're used to. You can do math with them.
**b. strings** are sequences of characters, like the letters `a-z`, spaces, and even numbers. These are all strings: `"Ryan"`, `"4"` and `"What is your name?"` Strings are extremely useful as labels, names, and content for your programs.

**c. booleans**. A boolean is either `true` or `false`.
For example, comparing two numbers returns a `true` or `false` result:

```
23 > 10 is true
5 < 4 is false
```

## Substrings

`"some word".substring(x, y)` where `x` is where you start chopping and `y` is where you finish chopping the original string.
The number part is a little strange. To select for the "he" in "hello", you would write this:

```
"hello". substring(0, 2);
```

Each character in a string is numbered starting from 0, like this:

```
0 1 2 3 4
| | | | |
h e l l o
```

The letter `h` is in position `0`, the letter `e` is in position `1`, and so on.
Therefore if you start at position `0`, and slice right up till position `2`, you are left with just `he`
**More examples**:

1. First 3 letters of "Batman"
`"Batman".substring(0,3);`

2. From 4th to 6th letter of "laptop"
`"laptop".substring(3,6);`

## Arrays

a. store **lists** of data

b. can store **different data types** at the same time

c. are **ordered** so the position of each piece of data is fixed

**Example**:

```
var names = ["Mao","Gandhi","Mandela"];

var sizes = [4, 6, 3, 2, 1, 9];

var mixed = [34, "candy", "blue", 11];
```

**Syntax**:

`var arrayName = [data, data, data];`

Any time you see data surrounded by `[ ]`, it is an array.

## Array positions

It's nice that we can put a list of data into an array. But now we need to learn how to get access to the data inside the array.

The position of things in arrays is fixed. So we just need to know the array name (here, it is `junkData`), and the position of the data we want, and we're done.

Small complication: the position (or the index) of each bit of data is counted starting from 0, not 1.

1. First element in the array: `junkData[0]`
2. Third element in the array: `junkData[2]`

Arrays have 0-based indexing, so we start counting the positions from 0.

## Object syntax

An object is like an array in this way, except its keys can be variables and strings, not just numbers.
Objects are just collections of information (keys and values) between curly braces, like this:

```
var myObject = {
    key: value,
    key: value,
    key: value
};
```

There are two ways to create an object: using **object literal notation** and using the **object constructor**.

Literal notation is just creating an object with curly braces, like this:

```
var myObj = {
    type: 'fancy',
    disposition: 'sunny'
};

var emptyObj = {};
```

When you use the constructor, the syntax looks like this:

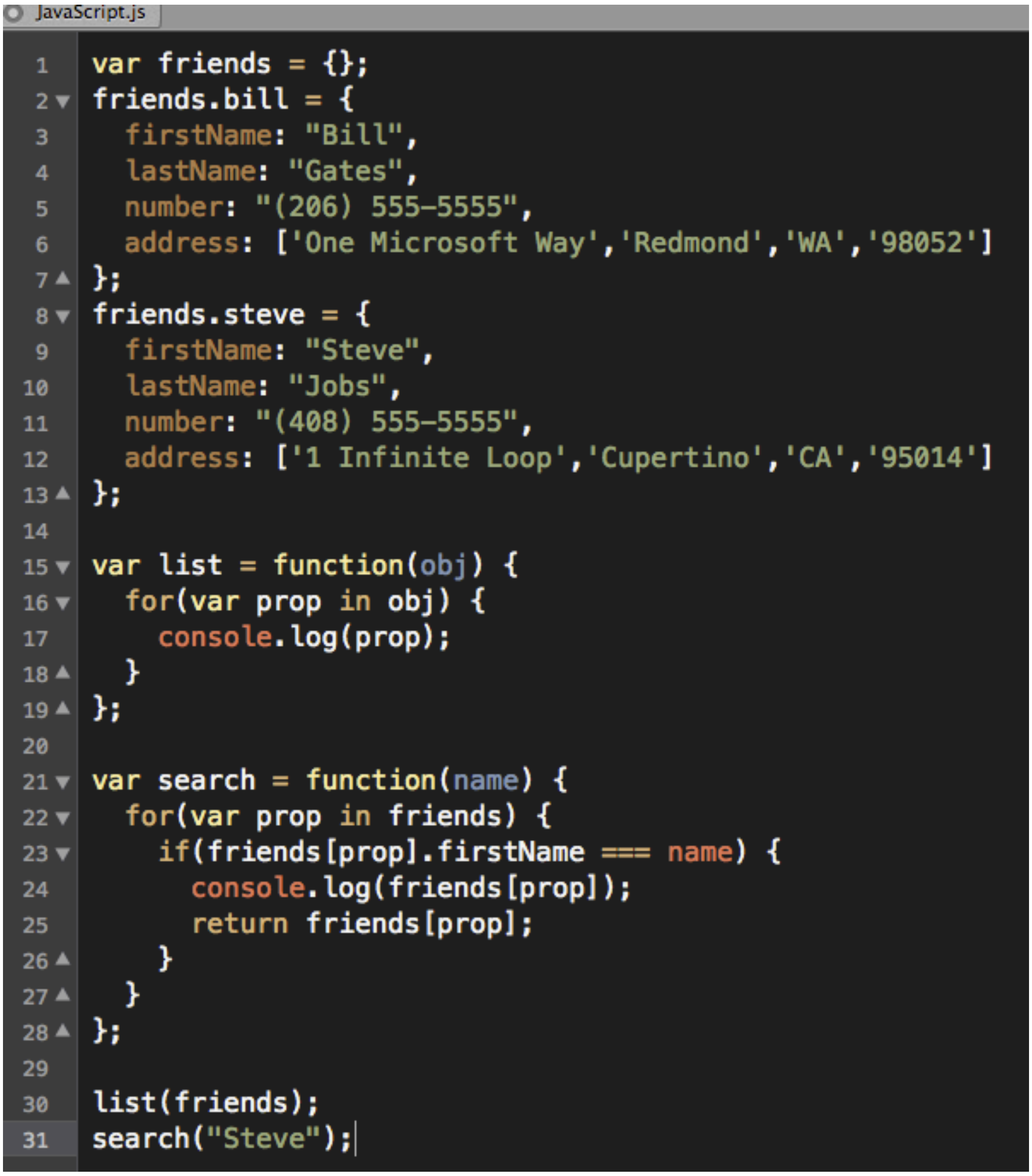
```
var myObj = new Object();
```

This tells JavaScript: "I want you to make me a `new` thing, and I want that thing to be an `Object`.

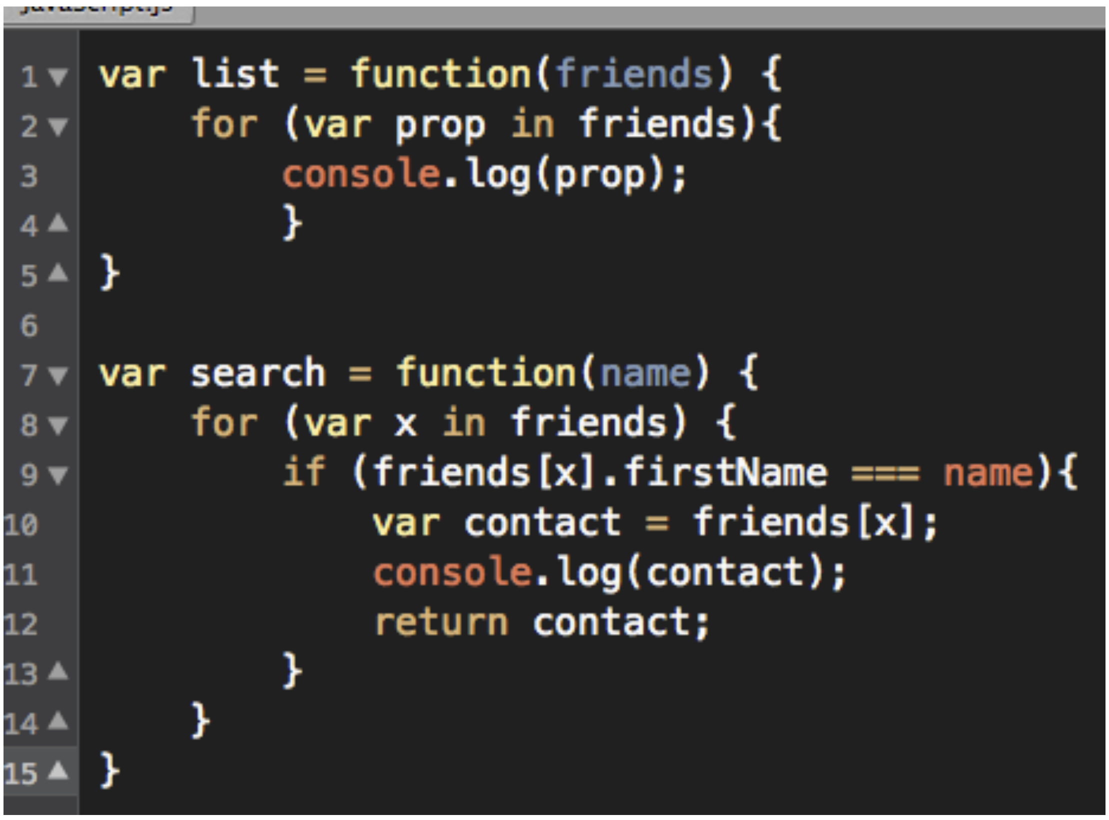
You can add keys to your object after you've created it in two ways **dot notation** and **bracket notation**:

```
myObj["name"] = "Charlie";
myObj.name = "Charlie";
```

Both are correct, and the second is shorthand for the first. See how this is sort of similar to arrays?


JavaScript.js

```
var friends = {};
friends.bill = {
  firstName: "Bill",
  lastName: "Gates",
  number: "(206) 555-5555",
  address: ['One Microsoft Way','Redmond','WA','98052']
};
friends.steve = {
  firstName: "Steve",
  lastName: "Jobs",
  number: "(408) 555-5555",
  address: ['1 Infinite Loop','Cupertino','CA','95014']
};

var list = function(obj) {
  for(var prop in obj) {
    console.log(prop);
  }
};

var search = function(name) {
  for(var prop in friends) {
    if(friends[prop].firstName === name) {
      console.log(friends[prop]);
      return friends[prop];
    }
  }
};

list(friends);
search("Steve");
```



```
var list = function(friends) {
    for (var prop in friends){
        console.log(prop);
        }
}

var search = function(name) {
    for (var x in friends) {
        if (friends[x].firstName === name){
            var contact = friends[x];
            console.log(contact);
            return contact;
        }
    }
}
```

## Properties

Each piece of information we include in an object is known as a **property**. Think of a property like a **category label** that belongs to some object. When creating an object, each property has a name, followed by `:` and then the **value** of that property. For example, if we want Bob's object to show he is 34, we'd type in `age: 34`.

`age` is the property, and `34` is the value of this property. When we have more than one property, they are separated by **commas**. The last property does not end with a comma.

# Misc.
# Tools

## Variables

We have learned how to do a few things now: make strings, find the length of strings, find what character is in the nth position, do basic math. Not bad for a day's work!

To do more complex coding, we need a way to 'save' the values from our coding. We do this by defining a variable with a specific, case-sensitive name. Once you create (or **declare**) a variable as having a particular name, you can then call up that value by typing the variable name.

**Code**:
`var varName = data;`
**Example**:
a. `var myName = "Leng";`
b. `var myAge = 30;`
c. `var isOdd = true;`

## Change variable values

So far, we've seen
a. how to create a variable
b. how to use a variable

Let's now see how to change a variable's value. A variable's value is easily changed. Just pretend you are creating a new variable while using the same name of the existing variable!

**Example**:
`var myAge = "Thirty";`

Say I had a birthday and I want to change my age.

`myAge = "Thirty-one";`

Now the value of `myAge` is "Thirty-one"!

# Global vs Local Variables

Let's talk about an important concept: **scope**. Scope can be global or local.

Variables defined **outside** a function are accessible anywhere once they have been declared. They are called **global variables** and their scope is **global**. For example:

```
var globalVar = "hello";

var foo = function() {
    console.log(globalVar);  // prints "hello"
}
```

The variable `globalVar` can be accessed anywhere, even inside the function `foo`.
Variables defined **inside** a function are **local variables**. They cannot be accessed outside of that function. For example:

```
var bar = function() {
    var localVar = "howdy";
}

console.log(localVar);  // error
```

The variable `localVar` only exists inside the function `bar`. Trying to print `localVar` outside the function gives a error.

Check out the code in the editor. Until now you've been using the `var` keyword without really understanding why. The `var` keyword creates a new variable **in the current scope**. That means if `var` is used outside a function, that variable has a global scope. If `var` is used inside a function, that variable has a local scope.
On line 4 we have not used the `var` keyword, so when we log `my_number` to the console outside of the function, it will be 14.

# Function syntax (behaviour)

A function takes in inputs, does something with them, and produces an output.

Here's an example of a function:

```
var sayHello = function(name) {
    console.log('Hello ' + name);
};
```

1. First we declare a function using `var`, and then give it a name `sayHello`. The name should begin with a lowercase letter and the convention is to use lowerCamelCase where each word (except the first) begins with a capital letter.
2. Then we use the `function` keyword to tell the computer that you are making a function
3. The code in the parentheses is called a **parameter**. It's a placeholder word that we give a specific value when we call the function. Click "Stuck? Get a hint!" for more.
4. Then write your block of reusable code between `{ }`. Every line of code in this block must end with a `;`.

You can run this code by "calling" the function, like this:

```
sayHello("Emily");
```

Calling this function will print out `Hello Emily`.

## How does a function work?

Let's break down exactly how a computer thinks when it sees the code for a function.

```
var functionName = function( ) {
    // code code code
    // code code code
    // (more lines of code)
};
```

1. The `var` keyword declares a variable named `functionName`.
2. The keyword `function` tells the computer that `functionName` is a function and not something else.
3. Parameters go in the parentheses. The computer will look out for it in the code block.
4. The code block is the reusable code that is between the curly brackets `{ }`. Each line of code inside `{ }` must end with a semi-colon.
5. The entire function ends with a semi-colon.

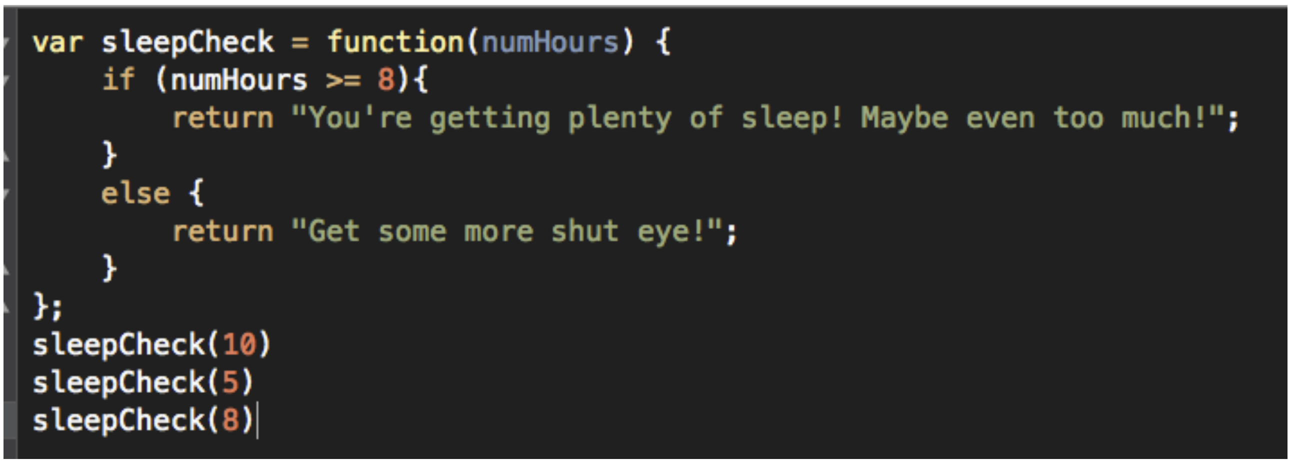
To use the function, we **call** the function by just typing the function's name, and putting a parameter value inside parentheses after it. The computer will run the reusable code with the specific parameter value substituted into the code.

## Functions with two parameters

So far we've only looked at functions with one parameter. But often it is useful to write functions with more than one parameter. For example, we can have the following function:

```
var areaBox = function(length, width) {
    return length * width;
};
```

With more than one parameter, we can create more useful functions
To call a function with more than one parameter, just enter a value for each parameter in the parentheses. For example, `areaBox(3,9);` would return the area of a box with a length of 3 and a width of 9.



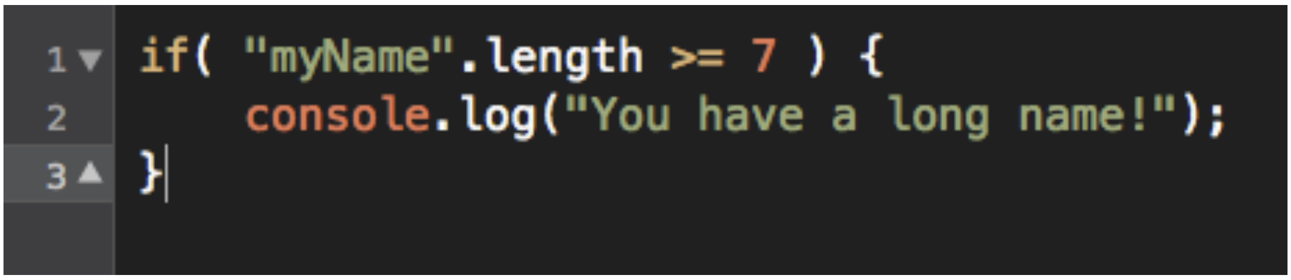
```
var sleepCheck = function(numHours) {
    if (numHours >= 8){
        return "You're getting plenty of sleep! Maybe even too much!";
    }
    else {
        return "Get some more shut eye!";
    }
};
sleepCheck(10)
sleepCheck(5)
sleepCheck(8)
```

# Misc.
# Tools

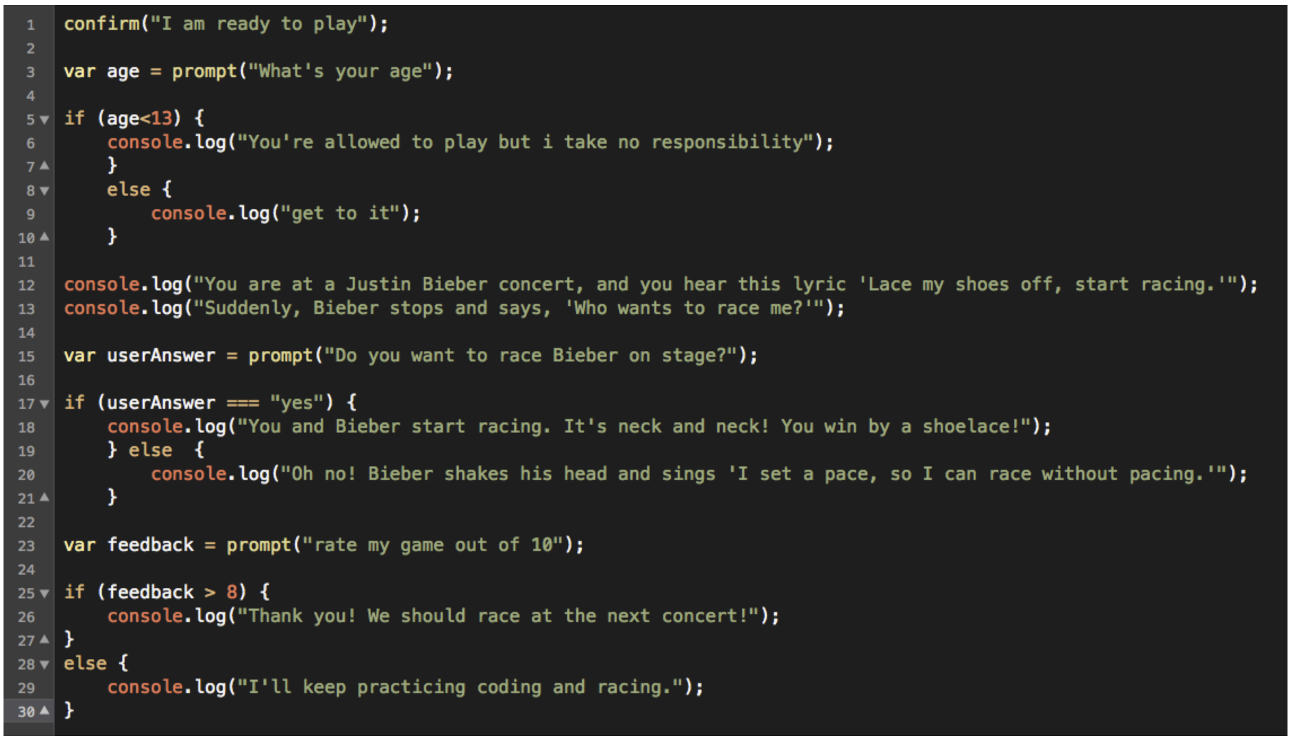
# CONTROL FLOW STATEMENTS

## if Statement

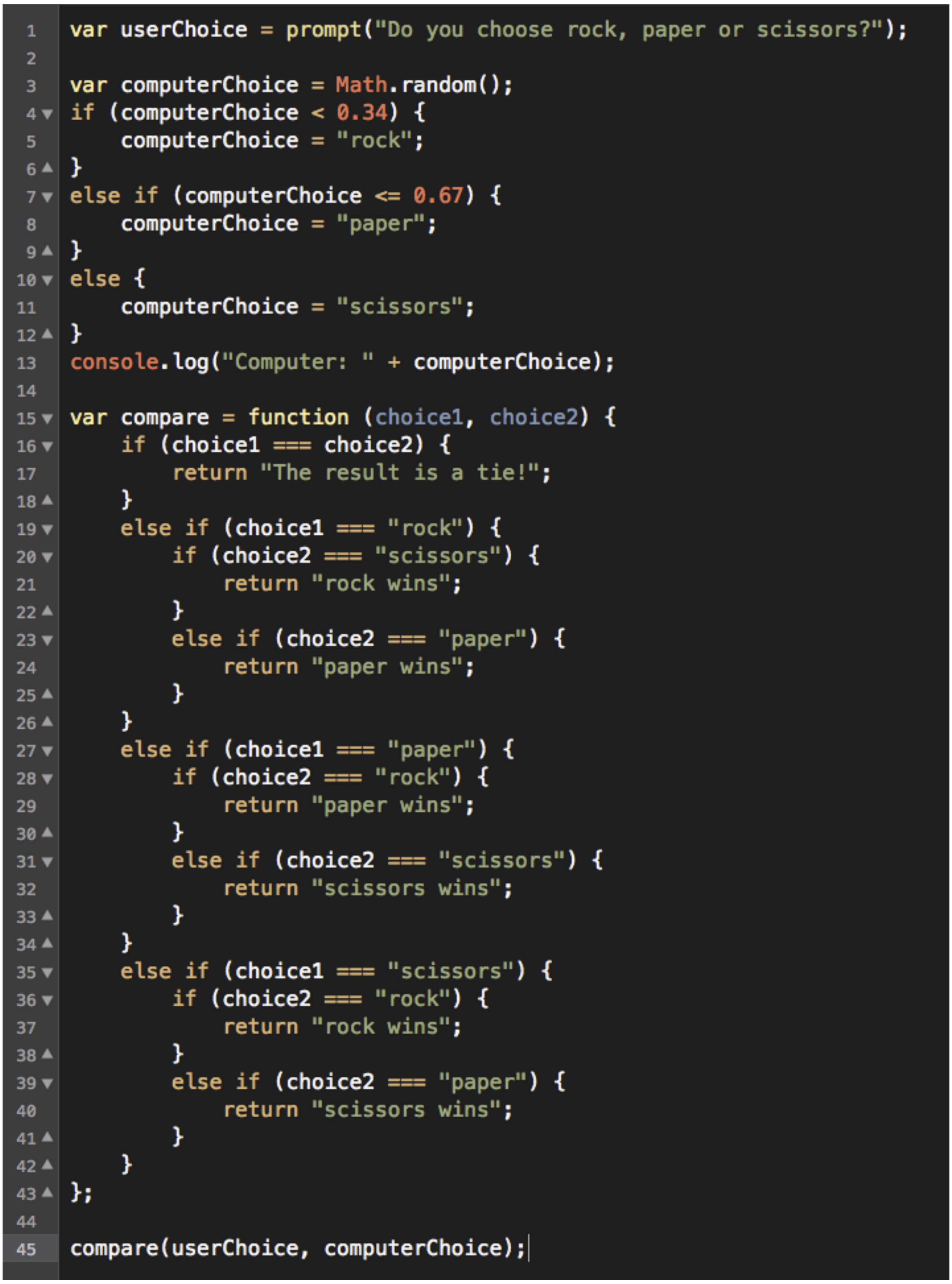
An `if` statement is made up of the `if` keyword, a condition like we've seen before, and a pair of curly braces `{ }`. If the answer to the condition is yes, the code inside the curly braces will run.



```
if( "myName".length >= 7 ) {
    console.log("You have a long name!");
}
```



```
confirm("I am ready to play");

var age = prompt("What's your age");

if (age<13) {
    console.log("You're allowed to play but i take no responsibility");
    }
    else {
        console.log("get to it");
    }

console.log("You are at a Justin Bieber concert, and you hear this lyric 'Lace my shoes off, start racing.'");
console.log("Suddenly, Bieber stops and says, 'Who wants to race me?'");

var userAnswer = prompt("Do you want to race Bieber on stage?");

if (userAnswer === "yes") {
    console.log("You and Bieber start racing. It's neck and neck! You win by a shoelace!");
    } else  {
        console.log("Oh no! Bieber shakes his head and sings 'I set a pace, so I can race without pacing.'");
    }

var feedback = prompt("rate my game out of 10");

if (feedback > 8) {
    console.log("Thank you! We should race at the next concert!");
}
else {
    console.log("I'll keep practicing coding and racing.");
}
```



```
var userChoice = prompt("Do you choose rock, paper or scissors?");

var computerChoice = Math.random();
if (computerChoice < 0.34) {
    computerChoice = "rock";
}
else if (computerChoice <= 0.67) {
    computerChoice = "paper";
}
else {
    computerChoice = "scissors";
}
console.log("Computer: " + computerChoice);

var compare = function (choice1, choice2) {
    if (choice1 === choice2) {
        return "The result is a tie!";
    }
    else if (choice1 === "rock") {
        if (choice2 === "scissors") {
            return "rock wins";
        }
        else if (choice2 === "paper") {
            return "paper wins";
        }
    }
    else if (choice1 === "paper") {
        if (choice2 === "rock") {
            return "paper wins";
        }
        else if (choice2 === "scissors") {
            return "scissors wins";
        }
    }
    else if (choice1 === "scissors") {
        if (choice2 === "rock") {
            return "rock wins";
        }
        else if (choice2 === "paper") {
            return "scissors wins";
        }
    }
};

compare(userChoice, computerChoice);
```

## Starting the for loop

Congratulations! You've just run your first `for` loop. But what you're probably really keen to do is write your own `for` loop. Below is the general syntax of the `for` loop. We want to focus on the first line in the next few exercises.

**Syntax**

```
for (var i = 1; i < 11; i = i + 1) {
    /* your code here */;
}
```

Every `for` loop makes use of a counting variable. Here, our variable is called `i` (but it can have any name). The variable has many roles. The first part of the for loop tells the computer to start with a value of 1 for `i`. It does this by declaring the variable called `i` and giving it a value of `1`.

When the `for` loop executes the code in the code block—the bit between `{ }`—it does so by starting off where `i = 1`.

## Ending the for loop

We know how to control where the `for` loop starts. How do we control where it ends? Well, the second part of the `for` loop determines that.

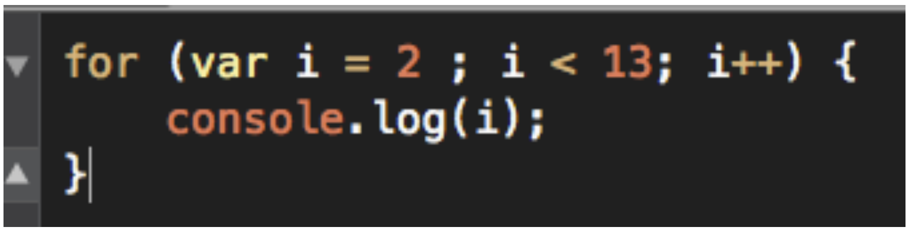
**Syntax**

```
for (var i = 1; i < 11; i = i + 1) {
    code code code;
    }
```

Here, this for loop will keep running until `i = 10` ( *i.e.* while `i < 11`). So when `i = 2`, or `i = 9`, the for loop will run. But once `i` is no longer less than 11, the loop will stop.

**Rules to learn**
a. A more efficient way to code to increment up by 1 is to write `i++`.
b. We decrement down by 1 by writing `i--`.
c. We can increment up by any value by writing `i += x`, where x is how much we want to increment up by. *e.g.*, `i += 3` counts up by 3s.
d. We can decrement down by any value by writing `i -= x`. (See the Hint for more.)
e. Be **very** careful with your syntax—if you write a loop that can't properly end, it's called an **infinite loop**. It will crash your browser!

## How does it work?



```
for (var i = 2 ; i < 13; i++) {
    console.log(i);
}
```

We've gone through the three bits of syntax for a `for` loop. But how exactly does it work? Let's imagine the steps the computer takes to run the `for` loop on the right.

1. It starts off with `i = 2`
2. It then asks: Is `i` currently less than 13? Because `i = 2`, this is true and we continue.

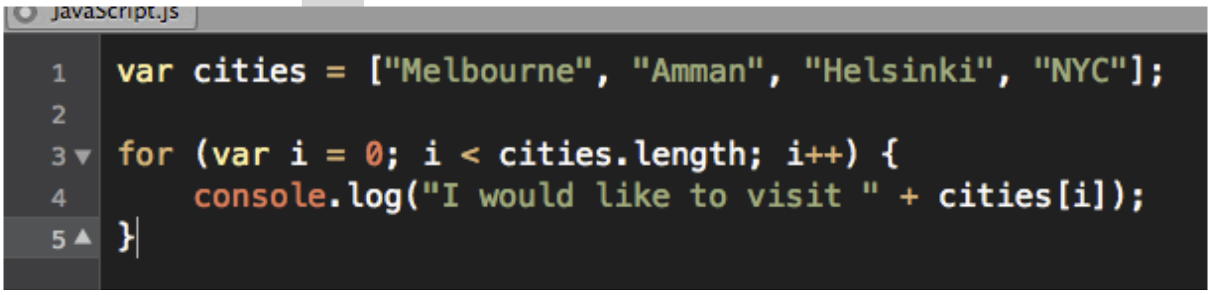
3. We do NOT increment now. Instead, if the condition is met, we run the code block.
4. Here, the code block prints out the value of `i`. It is currently 2 so 2 will be printed out.
5. Once the code block is finished, the `for` loop then increments / decrements. Here, we add 1.
6. Now `i = 3`. We check if it is less than 13. If it is true, we run the code block.
7. The code block runs, and then we increment.
8. We repeat these steps until the condition `i < 13` is no longer true.

`for` loops only run when the condition is `true`.

It is important that there is a way for the `for` loop to end. If the `for` loop is always going to be true, then you will be stuck in an infinite loop and your browser will crash! Look at the code. It is bad.

# Arrays I

For arrays, a useful way to systematically access every element in the array is to use a `for` loop!



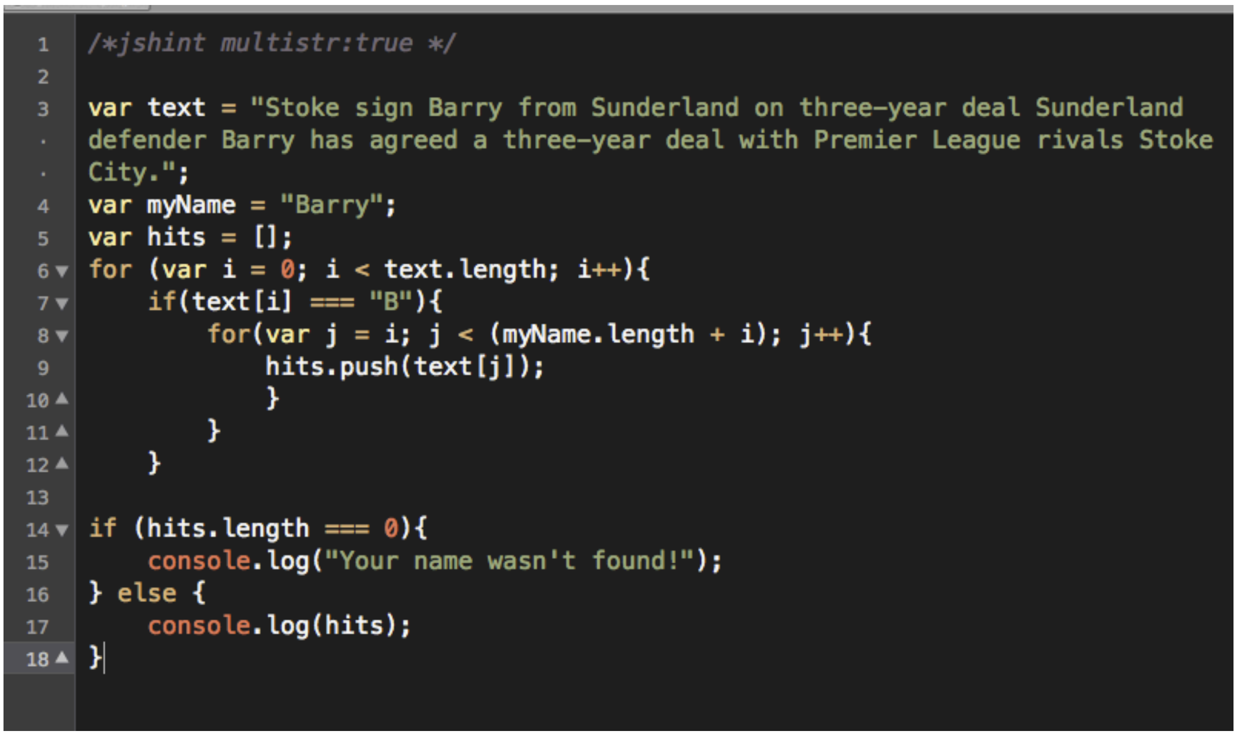
```
1  var cities = ["Melbourne", "Amman", "Helsinki", "NYC"];
2
3  for (var i = 0; i < cities.length; i++) {
4      console.log("I would like to visit " + cities[i]);
5  }
```

**How does it work?**

1. Line 3 declares the array. It has 4 elements.
2. We then start the `for` loop on line 5.
3. We see `i` starts off at value 0.
4. The `for` loop runs until `i < 4` (because `cities.length` equals 4. The array `cities` has 4 elements in it; see the Hint for more.)
5. We will increment `i` by 1 each time we loop over.
6. We print out `cities[0]`, which is `"Melbourne"`.
7. We then start the loop again. Except now `i = 1`.
8. It will print out `cities[1]`, which is `"Amman"`.
9. This continues until `i` is no longer less than `cities.length`.

```
for (var i = start; i < end; i++) {
  // do something
}
```

The counter variable `i` starts at "start", and stops looping when it reaches "end."



```
/*jshint multistr:true */

var text = "Stoke sign Barry from Sunderland on three-year deal Sunderland
defender Barry has agreed a three-year deal with Premier League rivals Stoke
City.";
var myName = "Barry";
var hits = [];
for (var i = 0; i < text.length; i++){
    if(text[i] === "B"){
        for(var j = i; j < (myName.length + i); j++){
            hits.push(text[j]);
            }
        }
    }

if (hits.length === 0){
    console.log("Your name wasn't found!");
} else {
    console.log(hits);
}
```

heterogenous arrays
multidimensional arrays
jagged arrays

## While syntax

The syntax looks like this:

```
while(condition){
    // Do something!
}
```

As long as the condition evaluates to `true`, the loop will continue to run. As soon as it's `false`, it'll stop. (When you use a number in a condition, as we did earlier, JavaScript understands `1` to mean `true` and `0` to mean `false`.)

## Brevity is the soul of programming

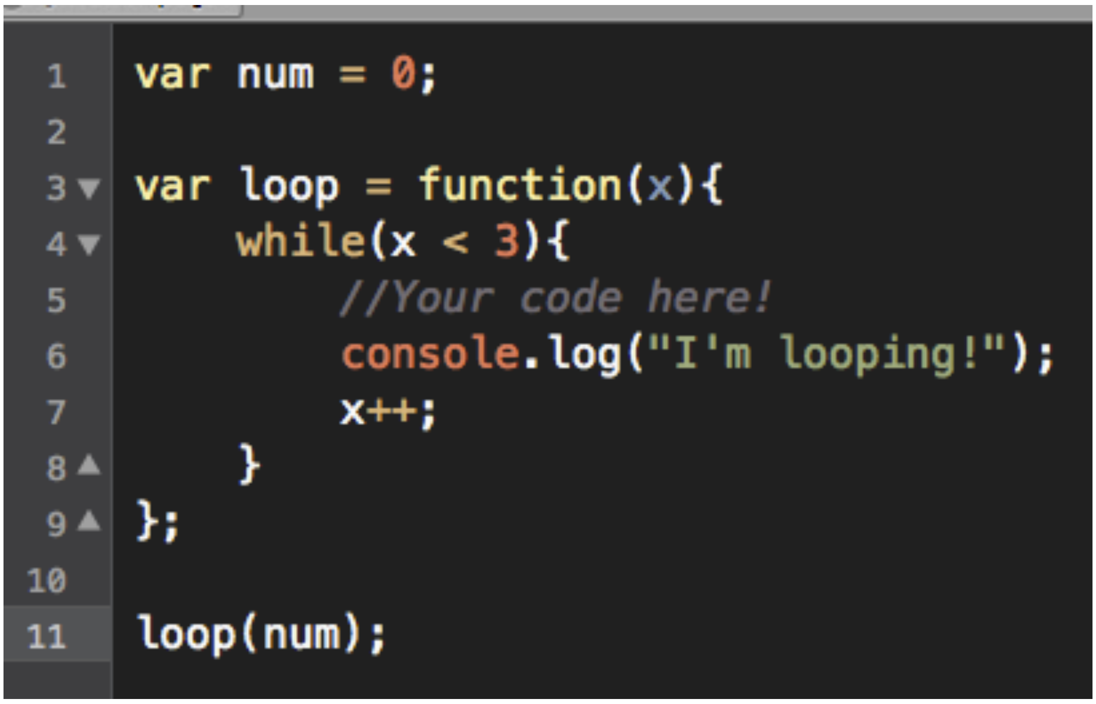
You may have noticed that when we give a variable the boolean value `true`, we check that variable directly—we don't bother with`===`. For instance,

```
var bool = true;
while(bool){
    //Do something
}
```

is the same thing as

```
var bool = true;
while(bool === true){
    //Do something
}
```

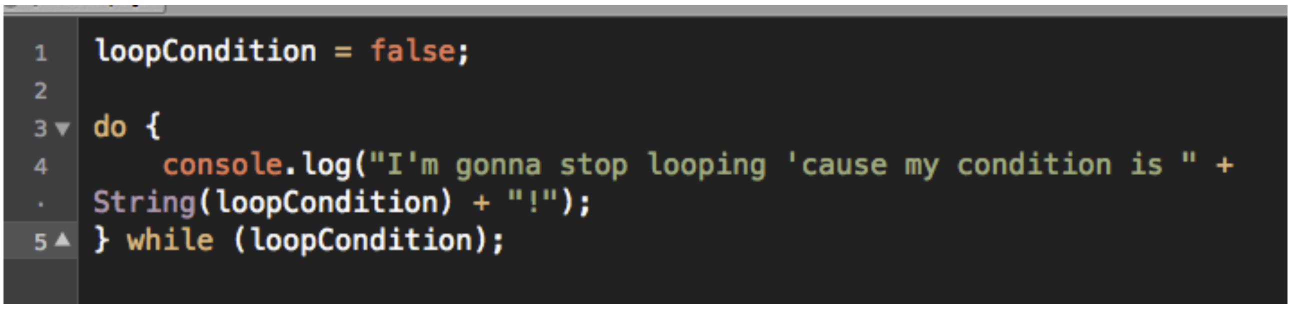
but the first one is faster to type. Get in the habit of typing exactly as much as you need to, and no more!



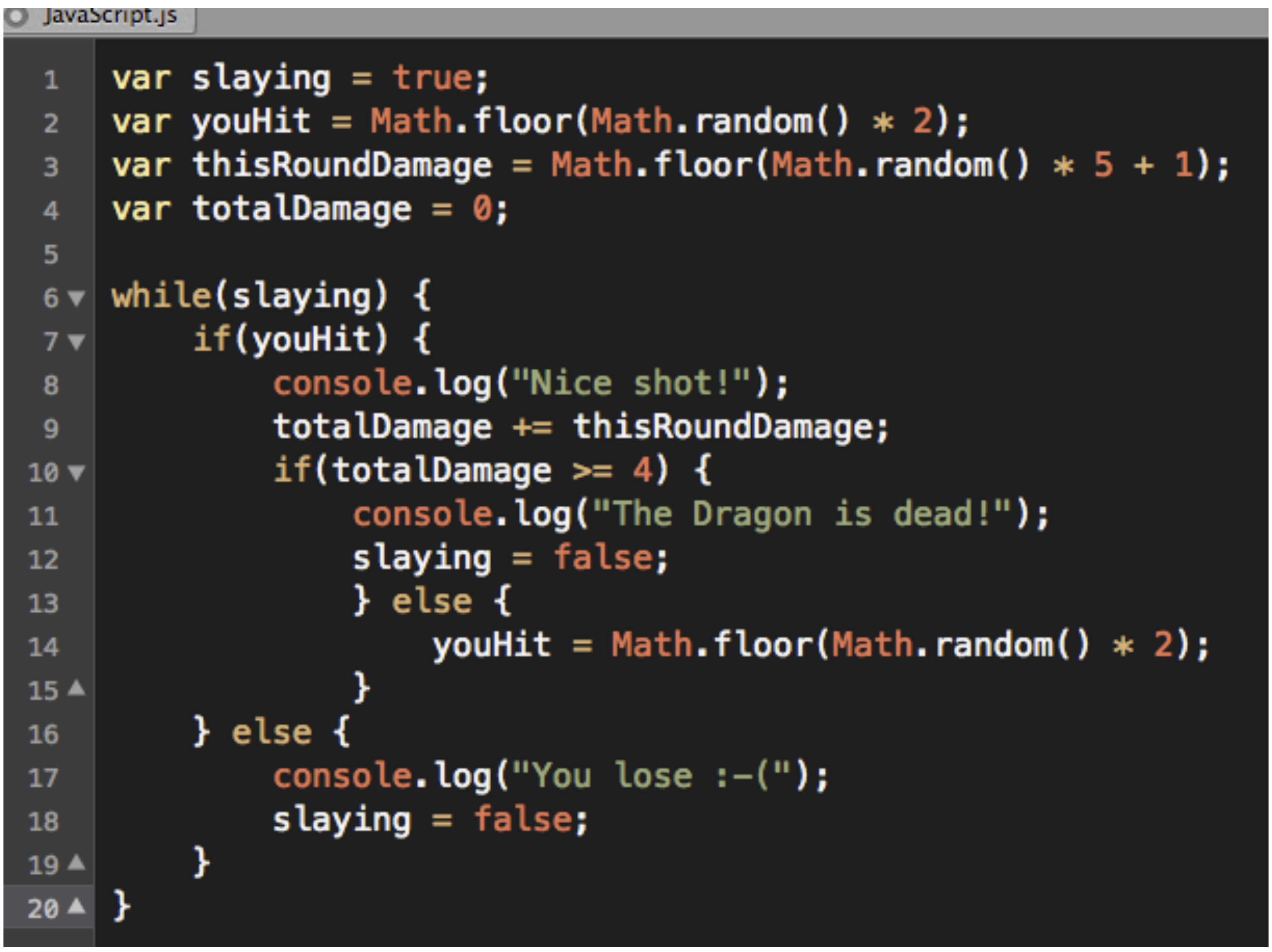
```
var num = 0;

var loop = function(x){
    while(x < 3){
        //Your code here!
        console.log("I'm looping!");
        x++;
    }
};

loop(num);
```

## The 'do' / 'while' loop

Sometimes you want to make sure your loop runs *at least one time* no matter what. When this is the case, you want a modified `while` loop called a `do`/`while` loop.
This loop says: "Hey! Do this thing one time, *then* check the condition to see if we should keep looping." After that, it's just like a normal `while`: the loop will continue so long as the condition being evaluated is true.



```
loopCondition = false;

do {
    console.log("I'm gonna stop looping 'cause my condition is " +
String(loopCondition) + "!");
} while (loopCondition);
```



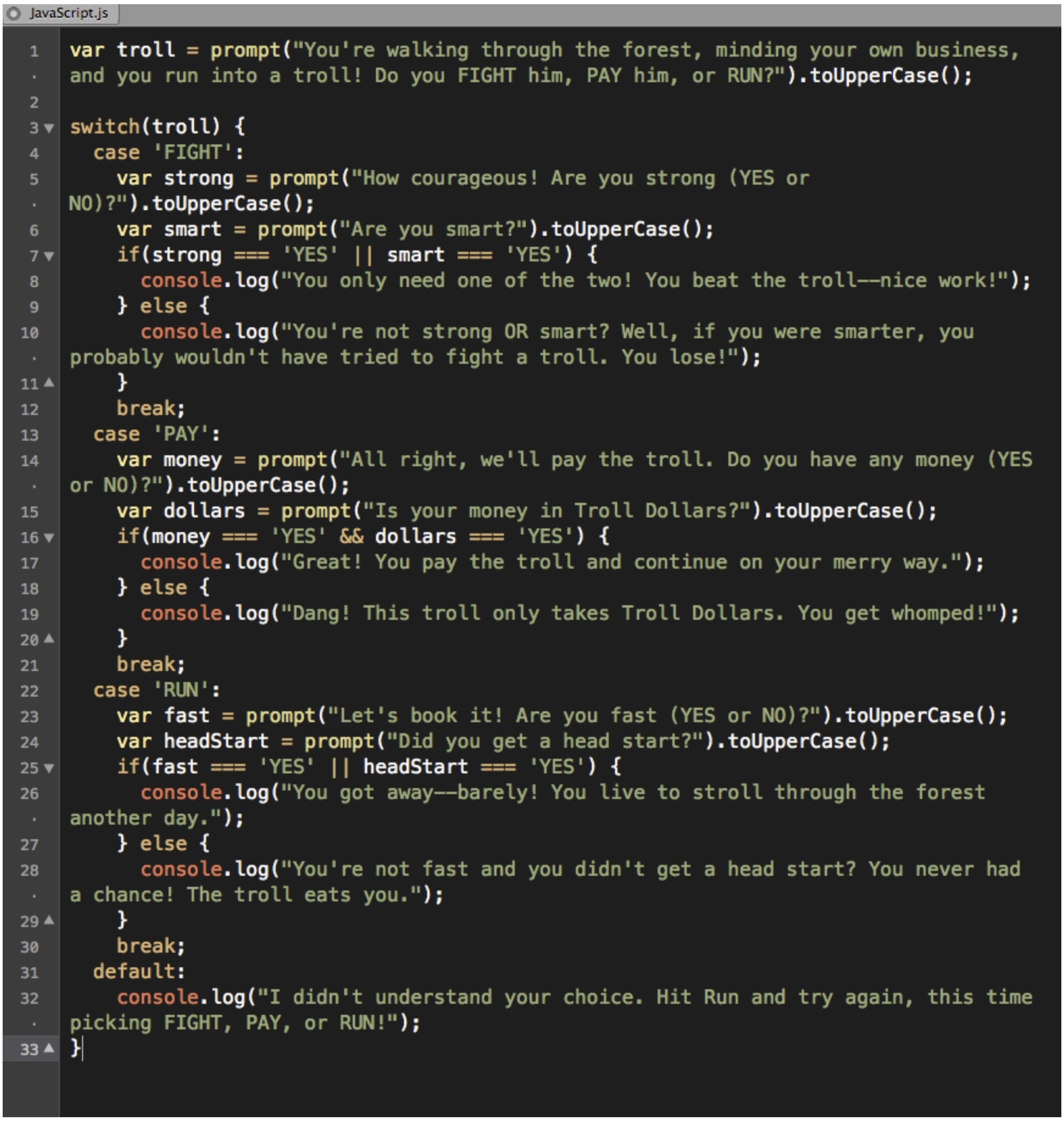
```
var slaying = true;
var youHit = Math.floor(Math.random() * 2);
var thisRoundDamage = Math.floor(Math.random() * 5 + 1);
var totalDamage = 0;

while(slaying) {
    if(youHit) {
        console.log("Nice shot!");
        totalDamage += thisRoundDamage;
        if(totalDamage >= 4) {
            console.log("The Dragon is dead!");
            slaying = false;
            } else {
                youHit = Math.floor(Math.random() * 2);
            }
    } else {
        console.log("You lose :-(");
        slaying = false;
    }
}
```

## Adding to an existing switch

The switch statement is put together like this:

```
switch (/*Some expression*/) {
    case 'option1':
        // Do something
        break;
    case 'option2':
        // Do something else
        break;
    case 'option3':
        // Do a third thing
        break;
    default:
       // Do yet another thing
}
```

JavaScript will try to match the expression between the `switch()` parentheses to each `case`. It will run the code below each case if it finds a match, and will execute the `default` code if no match is found.



```javascript
var troll = prompt("You're walking through the forest, minding your own business, and you run into a troll! Do you FIGHT him, PAY him, or RUN?").toUpperCase();

switch(troll) {
  case 'FIGHT':
    var strong = prompt("How courageous! Are you strong (YES or NO)?").toUpperCase();
    var smart = prompt("Are you smart?").toUpperCase();
    if(strong === 'YES' || smart === 'YES') {
      console.log("You only need one of the two! You beat the troll--nice work!");
    } else {
      console.log("You're not strong OR smart? Well, if you were smarter, you probably wouldn't have tried to fight a troll. You lose!");
    }
    break;
  case 'PAY':
    var money = prompt("All right, we'll pay the troll. Do you have any money (YES or NO)?").toUpperCase();
    var dollars = prompt("Is your money in Troll Dollars?").toUpperCase();
    if(money === 'YES' && dollars === 'YES') {
      console.log("Great! You pay the troll and continue on your merry way.");
    } else {
      console.log("Dang! This troll only takes Troll Dollars. You get whomped!");
    }
    break;
  case 'RUN':
    var fast = prompt("Let's book it! Are you fast (YES or NO)?").toUpperCase();
    var headStart = prompt("Did you get a head start?").toUpperCase();
    if(fast === 'YES' || headStart === 'YES') {
      console.log("You got away--barely! You live to stroll through the forest another day.");
    } else {
      console.log("You're not fast and you didn't get a head start? You never had a chance! The troll eats you.");
    }
    break;
  default:
    console.log("I didn't understand your choice. Hit Run and try again, this time picking FIGHT, PAY, or RUN!");
}
```