Beginning Programming in JavaScript

Clarissa Littler

June 5, 2016

1 Introduction

1.1 What Is This Document?

This document is a part of a set of tutorial lectures for absolute beginners to learn programming. The mini-course is centered around JavaScript and on accomplishing certain small goals in JavaScript.

1.2 What This Document Isn't

1. A full course on JavaScript
2. An explanation of how to do web-programming
3. A reference for the entire JavaScript language

1.3 Why Programming?

First, let's talk about why someone would even want to learn computer programming in the first place. There's the obvious pragmatic reason of hoping to get a job as a programmer.

There's also all the things you can potentially *do* as a programmer.

These days we talk a lot about coding websites and mobile apps, which are massively important in their own right, but there's even more beyond that. In just the last few years we've reached a point where we can programmatically control the electronics in our houses, we can write code to control 3D printers, we can program AIs and robots and self-driving vehicles.

A larger and larger portion of our lives is going to be mediated by code and automation, and since I have a strong populist streak I'd love to see more people be able to take control of this dependence on automation and artificial intelligence.

Even beyond all of those good reasons for learning programming, there's also just the pleasure of it. Coding can be a lot like cooking or any other form of crafting: you're figuring out how to make something and just the act of *making* brings a certain pleasure with it.

1.4 What Is Programming?

Before we start talking about *how* to program, we need to discuss *what* programming is. The answer I think I lot of people would give is something like "programming is how you tell a computer to do things", which is true but I think we can go a little deeper than that by taking a detour through what "computer" and "computation" really mean.

Once upon a time, by which I mean the first half of the twentieth century, "computer" was a job title not a noun. A computer was a person who made calculations, often for physics experiments or firing solution tables for the military. In other words, computers were people whose job was to *perform computation*. As a historical side-note, computers were mostly women and these computers, in turn, were some of the first programmers.

Computers, whether people or machines, perform computations by executing a series of steps. Think back to how you learned to add big numbers, like the ones below, together with a pencil and paper.

```
    32347
   203423
+ 34323425
----------
      ...
```

How do you calculate this? You start at the rightmost column, add the numbers together, carry if you need to, and proceed to the next column.

If you do this on a piece of paper, your work-in-progress might look something like [1]

$$\begin{array}{r} {\scriptstyle 1\,1} \\ 32347 \\ 203423 \\ +\ 34323425 \\ \hline \cdots 35 \end{array}$$

In fact, every calculation you learned how to do in math classes all had some series of steps you were supposed to do to get the answer. You learned how to perform these steps with a pen and paper, and later a calculator, but the principle is fundamentally the same either way.

We can generalize this idea of taking steps and calculations to be about more than just numbers. Any procedure that takes

- a finite[2] number of steps
- a finite amount of "material"
- a finite amount of time to complete

can be described as a computation. This includes things like

- cooking
- following a map
- building furniture from a wordless diagram

[1] The formatting for this came from `http://tex.stackexchange.com/questions/95812/how-to-show-carries-in-long-addition`

And while tasks like these might seem like a bit of a digression when we're talking about *computers*, the point here is that anything that **can** be described in the "finite" way we did above can potentially be done by programmed machine. Self-driving cars can follow map directions to get to a destination. 3D printers can assemble solid objects given a blueprint in the right format and enough material.

Computation is not just number crunching or showing you a webpage. Now, you might be wondering what *can't* be described by a computation, and it turns out the answer is "quite a lot, but not a lot you'd care about". You're most likely to run into the limits of computation when writing programs that try to analyze other programs. This is why there's no perfect anti-virus program: it's physically impossible for there to be a program that can look at another program and determine, with perfect accuracy, whether or not its a virus. I'm not joking when I say "physically impossible". The limits of computation are as real and physical as the laws of motion. No amount of making computers faster can get around them.

The difference between a recipe and a "program" is the level of precision. A recipe can be short, to the point, and you can fill in the gaps because you're a person and you have experience you can draw on to make conclusions, to read between the lines. A computer doesn't have that ability. It needs instructions to be absolutely precise, to be 100% clear with no ambiguity. Writing in English, or any other natural language, isn't precise enough to be certain that you're telling the machine what you **think** you're telling it.

Because of this, programming languages tend to be small and with a very rigid, non-extensible grammar. The same way that in any language we speak there's a notion of "correct" and "incorrect" grammar, there's correct and incorrect grammar for a programming language. Unlike a natural language, where I can speak my native tongue of Texan and say "y'all'd've" and you probably know what I mean. Spend five minutes on tumblr and you'll see new idioms and words being coined constantly. It's really cool! Programming languages don't generally have this flexibility, though. Their grammar is set.

Instead, we have specialized, simple, languages for exactly describing what the computer should do. Unsurprisingly, we call these *programming languages*. There's many, many programming languages out there and some are good, some are bad, but most are just *different* ways of describing computations to the computer. For this course, we'll be specifically learning a language called JavaScript.

1.5 Why JavaScript?

In this mini-course we'll be learning JavaScript. Why JavaScript in particular, though? First, JavaScript is the language that makes interactive websites **work**. Now, if your first thought is "what's a non-interactive site?" then I suggest you try looking at one of the GeoCities archive projects to see what 90s web pages looked like. They were ugly and they were basically just static text, images, and links. Nothing changed when you interacted with it. The only real points of interaction were forms and links.

Obviously, that's nothing what websites look like now. We have animations as you hover over and click things. We have pages that change constantly as you're interacting with them. We even have rich games that can run in the browser. All of that is possible because we now have the ability to run code that creates this interactive experience. All of the code that runs in your web browser is in JavaScript.

Why? Well, like most things related to programming language adoption it's a matter of someone deciding to use it and eventually everyone else settling on the ready solution rather than inventing their own. The end result, though, is that every browser that exists, whether on a phone, or a laptop, or a desktop, or a tablet all have what's called an *interpreter* that can understand JavaScript programs and run them in order to make the page your own interactive. We'll talk more later about *what* happens in your browser when you visit a webpage.

Suffice it to say, JavaScript is a ubiquitous and important language now.

Luckily, it's a fairly decent one. If you look online for people's opinions on JavaScript you'll find it gets some hate because there are some strange and counterintuitve aspects of the language, but they're also mostly avoidable unless you're running into someone else's code that uses them. "JavaScript: The Good Parts" by Crockford is a good reference for the nice, clean core of the language.

2 Basic Syntax and Translation

2.1 Loading and Running Code

Before we can do anything, we need to address how to *run* JavaScript programs. We've already established that every web browser has the ability to

run JavaScript code, but let's go over exactly how this works.

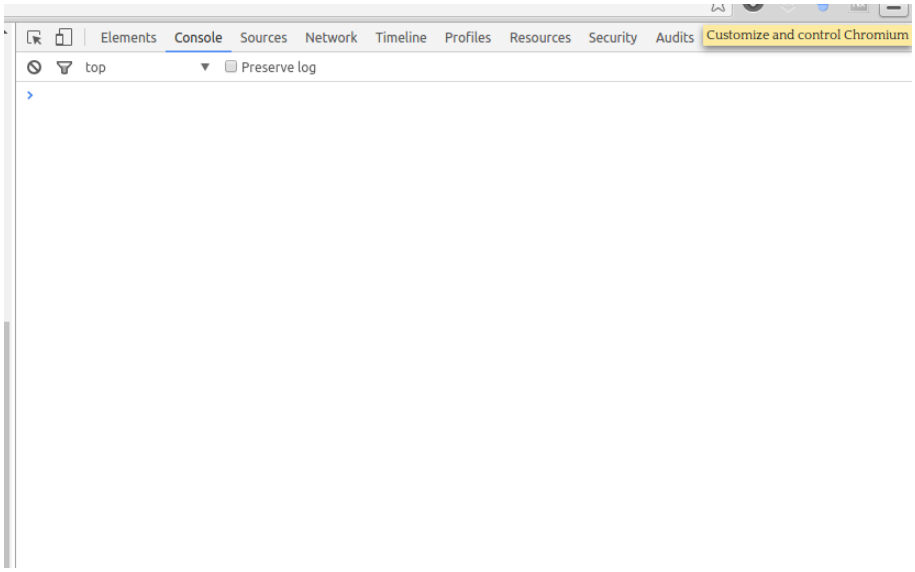
First, every browser has something called the *JavaScript console*. Each browser is slightly different in how you open the browser, but here's how to open the browser for the Big 3

Our answers here come from

http://webmasters.stackexchange.com/questions/8525/how-to-open-the-javascript-console-in-different-browsers

- Internet Explorer ⇒ hit f12 and the console is located in the "console" tab
- Chrome ⇒ `Ctrl + Shift + J`
- Firefox ⇒ `Ctrl + Shift + K`

Once you've opened the console you'll see something like



2.2 Nouns and Verbs

The same way that, in English, there's nouns that describe objects and verbs that describe actions, in a programming language there's a distinction between the "nouns" that describe data and the "verbs" that describe what to do with the data.

Data in a programming language are going to be things like numbers, pieces of text called strings, lists of things, and collections of things.

The actions in a programming language are things like reading in user input, printing out messages, changing the webpage, changing data, and storing data. There's even more complicated constructs to do things multiple times or to even store code so it can be reused again and again.

2.3 Descriptions vs. Algorithms

We've established that programs are detailed descriptions of instructions that are human readable but precise enough for a computer to understand.

There is a connection, though, between the ways we describe things to each other versus how we need to describe tasks to a computer.

For the rest of these notes we'll be explaining how to take a description of how to perform a task in English and translate it into JavaScript. There are going to be keywords that are important clues in how we take an English solution and turn it into real code. We'll highlight these keywords when we first introduce them by having them show in the color blue.

Our first example is that whenever we say print something, we know that in JavaScript this is going to turn into `console.log(thing-to-print)`. So whenever you see in a description such as

> Compare two numbers, `a` and `b`, and then print the value of the larger number.

You know that you're going to use `console.log` to print out something.

2.4 Running Code By Hand

The last bit of prologue before we start describing the JavaScript language is that throughout this document we'll be explaining how to evaluate code *by hand* if you wish to.

Now, that might seem an odd thing to do but it's a lot like learning arithmetic as a child. There's nothing wrong with using a calculator **once you know how it works**. First, though, you need to understand what the calculator is doing under the hood. Not even because it "builds character",

but because unless you know how to do calculations yourself on some level you won't know how to spot what a right answer and a wrong answer looks like and you won't have the skills to double check calculations.

It's easy to make typos and say something you didn't mean even entering things into a calculator and it's far easier still to say something you didn't mean at all when you're programming a computer. It's a useful skill to be able to check your code before you ever even run it.

To this end, with each new piece of JavaScript we introduce we'll explain how to evaluate the code by hand with a pen and paper. You don't have to format the paper the way I suggest, just as long as it's clear to you what the state of the progrma is.

2.5 Basic Expressions

2.5.1 Expressions and Values

In JavaScript, and a number of languages, there's a distinction between steps in a program and calculations that result in some kind of value. By calculations I mean things such as

- concatenating strings
- adding numbers
- printing values

Most expressions will return some kind of *value*. By *value* I mean the basic data of JavaScript: numbers are values, pieces of literal text are values, lists are values, and other kinds of data we'll end up seeing. There's even a value called *undefined* that's the value that corresponds to "this expression didn't return anything useful", which might seem odd at first but it's similar to the way the number 0 is the quantity of "no quantity".

2.5.2 Numbers

The first kind of data we'll look at are *numbers*. Numbers in JavaScript are just like numbers in math classes you took. The operations you're familiar with are all here: multiplication, division, addition, and subtraction.

In JavaScript, the symbols are pretty similar to what you may have seen before

name	symbol
addition	+
subtraction	−
division	/
multiplication	*

If you type in something like `10*(3-2)+5` into the console you'll see the JavaScript interpreter *evaluate* the expression and then return the value, which in this case is 15.

Go ahead and try a few arithmetic expressions just to see what happens.

1. Evaluation by hand This is our very first example of how to evaluate code **by hand**. Now, there's two pieces here that are important. The base numbers in JavaScript, the *literals* as they're often called in programming, just evaluate to themselves: the number `4` becomes the value 4, the number `0` becomes the value 0.

   You can test that yourself in the JavaScript console by just entering numbers and seeing that the *value* returned is just the number you entered.

   The arithmetic expressions evaluate in the normal order of operations[3]

2.5.3 Strings

One of the other incredibly important kinds of data are *strings*. Strings are pieces of text held within quotation marks, either double or single quotes. A programming language needs strings so that it can interact with text: either reading and understanding it or displaying it to the user.

You can make a string either like

[3]Please Excuse My Dear Aunt Sally

Please	Parentheses
Excuse	Exponent
My	Multiplication
Dear	Division
Aunt	Addition
Sally	Subtraction

```
"this is a string, or should I say 'a string'"
```

or like this

```
'this is a string, or should I say "a string"'
```

but there's a few things that *aren't* valid. You can't do

```
"this is a string, or should I say "a string""
```

because since you started the string with a double-quote its not obvious to the interpreter where you wanted the end of the string to be. If you want to represent quotations-within-strings you should really just switch between single and double quotes.

This also means that it isn't valid to mis-match the kinds of quotation marks. So something like

```
"this is a string'
```

will not work.

The most primitive operation on strings is the ability to *concatenate* text. Concatenate really just means "stick together" and, in JavaScript is *also* represented by the + symbol. If you enter something like

```
"this is one string" + " this is another string"
+ " and together we are..."
```

You will see "this is one string this is another string and together we are..." as one string. Now, you'll notice that we needed to put *space* at the beginning of " this is another string" and " and together we are..." in order for their to be a space between the pieces of the sentence. We could also have just as easily written

```
"this is one string " + "this is another string "
+ "and together we are..."
```

because all that matters is that the spaces are *somewhere*.

Leaving out spaces is a **very** common mistake, so get in the habit of paying attention to the spaces at the beginning or end of the string.

1. Evaluation by hand Strings evaluate to themselves, so the a valid string such as

   ```
   "my dog is named chicken"
   ```

   evaluates to the *value* "my dog is named chicken"

   String concatenation is evaluated by combining the two strings, being careful to not add any extra space.

2.6 Statements and Steps

One of the first things we need to discuss before we begin writing real programs is how to do more than a single step in a program.

We've already seen two basic kinds of expressions: we've seen numeric and string *literals*, where the thing you type **is** the value, and basic arithmetic operations that evaluate to a number just the way you'd expect.

Real programs, just like real directions, have many *steps*. These steps in JavaScript are called *statements*. Statements are **generally** separated by semi-colons (**;**), though not always. We'll be explicit about where they are unnecessary. [4]

Any expression can be put on a line by itself, like this

```
10;
20;
"lalalala";
```

but simple expressions don't really do anything. A more *interesting* expression that we looked at in the opening was `console.log`. We'll still delay a bit in explaining **why** `console.log` works the way it does, but we'll use it at the JavaScript console to print things out. The following simple program just prints out the numbers 1,2,3 in succession.

```
console.log(1);
console.log(2);
console.log(3);
```

If you point your browser to the file `consoleLogTest.html`, which includes the above code as a script, and then open up the console you should see the numbers 1,2,3 printed out.

2.6.1 Evaluating by hand

To evaluate a sequence of statements, just evaluate each statement in turn in the order they appear down the page.

[4] Technically there's many places where you **could** leave semi-colons out, but it's generally a bad habit. Why? Because it generally leads to very *strange* error messages when something goes wrong. Anything that makes your code harder to debug is generally a bad idea.

2.7 Variables

Now that we know how to do more than a single thing at a time, we need to deal with how data is stored and used later. In essentially every programming language we have some notion of *variables*.

Variables are something we're all familiar with in our speech in general. Have you ever heard a story where someone says "I have a friend, let's call her Anna, . . . ". For the rest of the story you know that "Anna" is the speaker's friend, even if that's not her real name. The name Anna "points" to the person.

Similarly, we have *pronouns* in English. We can say "he" or "him" and, if we've already established who "he" is, then you know who the "variable" points to. For example, in the sentence "Bob has three hats. He wears two of them each day." you know that "he" is Bob of the three hats.

Variables are the pronouns of a programming language. We make them like

```
var thisIsAVariable = 20;
```

where `var` is the start of the expression that tells JavaScript that you are *declaring* a variable, the name "thisIsAVariable" is the actual name of the variable, the *pronoun* you're making, and the expression to the right of the equals sign is the going to be evaluated to give the value the variable *pointing* at.

If you consider the English sentence "She waved at Anna, and she waved back" you'll notice that "she" means two different people within the same sentence. Similarly to English pronouns, in a programming language variables are allowed to refer to, to point to, different things at different times.

The act of making a variable refer to a piece of data is called *assignment* and we say that you're *assigning the variable*.

Assigning the variable has the following form

```
thisIsAVariable = "assigning a string instead";
```

```

```

In this case, we're assigning a string to `thisIsAVariable`.

You might wonder if it's possible to do *declare* a variable without *assigning* it and, in fact, you can

```
var thisIsntAssigned;
```

But this is like starting a conversation with "He says hi!". You have no idea who *he* is at the start and the person, or computer, you're talking to is just going to be confused. Now, you can clarify who *he* is by assigning a value later in the "conversation" but it's generally best to assign something to a variable when it's declared.

2.7.1 What can you name variables?

You might be wondering what variables can be named. While technically the rules are slightly broader than this, I recommend variables to use just letters when possible, starting with a lower case letter, and using alternate casing when a variable name is multiple words.

Now, some folks like to say you should give variables really long descriptive names like `howFarTheMisslesShouldGo` but I think it's good enough to give them a name that's distinctive and memorable such as `missleDist` for the /dist/ance the missles should go. I personally find very long multi-word names make it harder to skim code effectively.

2.7.2 Evaluating by hand

To evaluate variables by hand, first create a small two-column table on your paper with the headings "names" and "values". Fill the first column with the names of all the variables you see declared in the program and leave the second column empty for now. [5]

[5] Technically speaking this isn't **quite** how JavaScript works, because of something called *variable hoisting* where declarations are evaluated, but that's an advanced topic

Then, when you come to the line of a variable declaration, evaluate the expression to the right of the equals sign, if any, and fill in the value in the corresponding spot of the table. If there is no expression when the variable is declared, put `undefined` in the table instead.

When you assign a variable, evaluate the expression to the right of the equals sign then change the value in the table corresponding to the variable.

Finally, when a variable is *referenced*, just look up the value of the variable in the table and return that value.

2.8 Arrays

In every day life, we use *lists* constantly: todo lists, grocery lists, email lists, even your Facebook friends or Twitter followers is, in some sense, a general list of things.

More generally, the concept of a list describes everything from a hastily made scrawling of directions to general containers of thigns like bookshelves and clothes racks.

The key features are that there is an *order* to the elements of the "list", a beginning and an end, and there's a way to retrieve and replace what's in different spots in the "list", much like how you can grab a book from the middle of a bookcase and put a book in another spot later. In this generalized list you don't have to worry about the size per se the way you would with a bookcase. You can keep adding items to the list, wherever you want in the list.

These lists in JavaScript are called arrays and they're the first kind of compound container that we'll see in JavaScript. In JavaScript, we make arrays by putting expressions between square brackets and separated by commas as in `[1,3,"thing"]`. This example gives us an array that has three items in it: these three "slots" are numbered *starting with zero* [6]. More explicitly, we have that `1` is stored in the 0th place in the array, `3` is stored in the 1st place in the array, and `''thing''` is stored in the 2nd place in the array.

---

and one that you shouldn't run into as long as you always *declare* your variables *before* using them.

[6] This is called "0-indexing" and is almost universal in computer science. If you do any amount of programming you'll start unconciously counting things starting with the "0th" item.

In JavaScript, we can retrieve items from the array with the "square bracket" syntax as in the following example

```
var arr = [1,3,"thing"];
console.log(arr[0]);
console.log(arr[1]);
console.log(arr[2]);
```

In other words, we take

1. the name of the array (`arr`)
2. open square bracket (`[`)
3. the number corresponding to the place, or index, in the array (`0`)
4. a closing square bracket (`]`)

Very similarly, you can change what's in the slot of an array by assigning to the slot `arr[i]` as in the following exmaple

```
var arr = [2,4,6];
arr[0] = 1;
arr[1] = 3;
arr[2] = 5;
console.log(arr);
```

2.8.1 Exercises

Consider the following program:

```
var myVariable = "variable";
var myArray = [1,3,myVariable,7];

myArray[2] = 5;
```

Now, what's the value of myVariable at the end of the program? Explain why in terms of your understanding of variables and arrays.

2.9 Objects

In everyday life we have things like contact lists, directories, dictionaries, and glossaries. These are all kinds of data that map names to some kind of *information*.

All these general concepts of mapping are capture in JavaScript by *objects*. Objects are simply collections of names and values. We generally call the names *properties*.

You can make an object by including a list of pairs of names and expressions, separated by commas

```
{ name1 : 1, name2 : 2, name3 : "3"}
```

The colons between the names and values are important.

Once you have your data in an object, you can access the data two different ways. The first one is what people call the "dot syntax". It works like

```
var ourObject = {name1 : 10, name2 : "thing"};
console.log(ourObject.name1);
```

where to *get* the value connected to `name1` in the object `ourObject`, we put a dot between `ourObject` and `name1`. There's no quotation marks

needed. In this way object properties are much like variables: they are names that refer to values.

The other way you can refer to the properties of an object are with the "bracket syntax", where you put the name of the property **in quotes** and inside a pair of brackets instead. Our example above becomes,

```
var ourObject = {name1 : 10, name2 : "thing"};
console.log(ourObject["name1"]);
```

2.9.1 Why "object"?

The name object might seem a bit odd, but there's some intution for it.

First, think about how you might want to represent a physical *object* like a table in a computer program. If you want to represent a table in, say, a game then you probably only care about a few things about it such as

1. the table's location
2. the table's dimensions
3. the table's color

These attributes you care about are the table's *properties* and a collection of all of this data is the representation of a table as an object.

1. Exercises
   (a) How would you represent a user account for a social media site *as an object*?
   (b) How would you represent a car *as an object* in a racing game?

2.9.2 "Everything" is an object

In JavaScript, *essentially* everything is an object [7]. As such, almost everything in JavaScript has properties you can call on. Arrays, for example,

[7] This isn't quite true. Primitive data types such as booleans and numbers and strings aren't actually objects in-and-of themselves, but if you ask about their properties then

have a length property that tells you how many pieces of data are currently stored in the array. Try a couple of simple examples like the following in your console:

```
[1,2,3].length;
[10,"50",["one","hundred","chickens"].length].length;
```

2.10 Defining and calling functions

The next major concept we'll be discussing are *function*. Functions fill the role of being computations that are described **once** and used **many** times. We're actually very familiar with grouping and naming a series of steps, so familiar we might not even think about it! If I ask you to go to "google.com', you know exactly what I'm asking you to do and how to do it. You know how to

1. turn on your computer, if it isn't already
2. navigate with your mouse or keyboard to open a browser
3. type "google.com" into the URL bar of the browser
4. and hit enter so the page loads

We could break these steps down even further into describing how to turn on your computer, how to use a mouse, *how to even move your hand*, etc. The point, though, is that if I say "open google" or "go to google" you know what I'm asking you to do and can perform the entire sequence of steps.

That is what functions are in any programming language: they are a sequence of steps that is grouped together so it can be used again and again.

they get "wrapped" with an object that handles the properties and methods. For all intents and purposes, though, you can think of these primitive data as objects that just don't have a *type* of object.

We see this in any kind of instructions: cookbooks assume you know what it means to dice or julienne, knitting books assume you can knit or purl, etc.

All that being said, ou might *also* be familiar with the idea of functions from something like a math class in high school or college, where a function in math is defined like

$$f(x) := x + 2$$

and then you *call* this function by putting an argument between the parentheses like $f(2)$. Now, you don't need to remember this or have seen this before in a math class, but if you do have that experience it might help to remember how functions worked then.

Namely, that a function has three parts:

1. Its name (f)
2. Its parameters (x)
3. Its body ($x + 2$)

and that when you *call* a function, you use the *name* of the function and then wrap the arguments you're providing to the function in parentheses ($f(2)$). Then you evaluate the body of a function by substituting the *arguments* for the *parameters* and then executing what's left now that the parameters are gone ($2 + 2$).

JavaScript works almost the same way, but more verbosely: let's see the same function defined in JavaScript.

```
function f (x) {
    return x + 2;
}
console.log(f(2));
```

There's still a name for the function (`f`) and we still put the sparameters in parentheses and we still call the function the same way as before.

The difference is that we put the *body* of the function between two braces and we have this keyword return that we haven't seen before. This reflects that in **programming** we sometimes want to call functions that don't give back any useful value. In fact, we've been using one this whole time: namely, `console.log`. You may have noticed that when you type `console.log(..)` in the console you see the value of the argument printed out **and** you'll see `undefined` returned from the function. Any function that doesn't explicitly return a value will return `undefined`. For example, the following function

```
function test (x,y) {
    console.log(x);
    console.log(y);
    console.log(x);
}
```

will print out the first argument, the second argument, and then the first again. It will return `undefined`.

One other difference between functions in programming and functions in math is that functions in programming can have **no** arguments. Consider the following code

```

```

2.10.1 Function scope

Functions act like mini-programs within your main program, and as such they can have their own variables that are independent of the larger program.

Try running the following program and seeing what happens:

```
var myVar = 0;
function test () {
    var myVar = 10;
    var testVar = 20;
```

```
}

test();

console.log(myVar);
console.log(testVar);
```

If you run this code, you'll see that the final values of `myVar` and `testVar` are `0` and `undefined` respectively. Recall that we described making variables as making a table, a place, that connects the name of the variable to the value the variable points to. However, when you declare a variable **within** a function, the variable goes into a separate table than a variable declared at the top-level of the program. When the function is done executing, there's no way to refer to refer to this special table anymore. [8]

In JavaScript, when you're looking up a variable you look at the most recently declared

2.11 Choices

Often when we're discussing instructions there's a notion of *choice*. If it's raining, take an umbrella. If you see the purple Little Free Library, take a right, otherwise you should keep walking. If the avocados are ripe, make tacos.

We can make these choices in programming as well. If the username and password match, log the user in. If the user clicks send, send the email.

2.11.1 Booleans

In order to make these choices, though, we need for the programming language to understand what it means for something to be "true".

While truth is a pretty complicated concept if you ask a philosopher, for the purposes of basic programming it's quite simple: there is a kind of

[8] Mostly true! There's actually a way to keep around the ability to refer to these variables after the function is done. They're called *closures* and we'll cover them later in these notes.

data called a *boolean*. Booleans are either `true` or `false`. In order to make decisions, we need operations and functions that will return booleans.

For example, we have the equality operator, `===`, and the less than operator `<`, and the greater than operator `>`. These operations follow our intuitive notions of what they should mean. `2 < 5` is true, but `2 === 5` is false.

We also have operations *on* booleans such as or (`||`), and (`&&`), and not (`!`) that represent logical operations. The or operation (`a||b`) is true whenever `a` is true **or** `b` is true or both are true. The and operation `a&&b` is true *only* when `a` is true **and** `b` is true. The not operation `!a` is true only when `a` is false.

These correspond to how we make decisions. If you're hungry and you have a snack, then eat it. If you're not bored, keep working. If you have a date or it's been more than a month, then shower.

1. Truthy and falsy JavaScript's notion of booleans is slightly more flexible than most programming languages. You can actually use any kind of data like booleans. All data in JavaScript is either *truthy*, or is treated like the boolean `true` in operations, or it is `falsy` and is treated like `false`.

2. Evaluation by hand Both `true` and `false` are literals, just like basic strings and numbers, and they evaluate to the values `true` and `false` respectively.

   The boolean operations `&&`, `||`, and `!` are operators summarized below.

   (a) Boolean operations on Booleans And:

argument 1	argument 2	result
true	true	true
true	false	false
false	true	false
false	false	false

   Or:

argument 1	argument 2	result
true	true	true
true	false	true
false	true	true
false	false	false

Not:

argument	result
true	false
false	true

(b) Boolean operations in general We've skipped over an important detail about how the boolean operations `||` and `&&` work: something often called "shortcircuiting evaluation".

If you'll notice back at the tables in our previous section then you'll see that if the *first* argument to `||` evaluates to true (or a truthy value), then we don't *need* to evaluate the second argument to `||` to know that the expression will evaluate to `true`. In fact, JavaScript (and most languages) won't evaluate the second argument to `||` if the first argument is true. If you type the following code into the console

```
5 || console.log(10);
```

You should just get the number `5` without the number `10` being printed out.

Similarly, you'll see that for `&&` that the second argument only *needs* to be evaluated if the first argument is false or falsey.

Thus we can summarize the tables more generally as

`a && b`:

a	b	result
truthy	evaluated	b
falsey	unevaluated	a

`a || b`:

a	b	result
truthy	unevaluated	a
falsey	evaluated	b

`! a`

a	result
truthy	false
falsey	true

2.11.2 If-statements

As you can imagine from the way I've been emphasizing the word "if", it's somehow important to the syntax of making choices.

We call these "if-statements", and we use them like

```
if (2 < 5) {
    console.log("two is less than five");
}
```

If-statements are the first exception we've seen to the rule that all-statements end with a semi-colon.

What if you have alternatives in mind: code that you want to run if the condition *isn't true*? In that case you need the keyword `else`, as in

```
if (2 === 5) {
    console.log("two is equal to five");
}
else {
    console.log("two wasn't equal to five");
}
```

If you have complicated conditions you can even chain if-else statements like

```
var thing = 10;

if (thing > 20) {
    console.log("print one thing");
}
else if (thing < 0) {
    console.log("print another thing");
}
else {
    console.log("THE THING");
}
```

1. Evaluation by hand To evaluate an if-statement of the form

```
if(condition){
    ...
}
else {
    ...
}
```

first evaluate the condition. If it is *truthy*, execute the code in the body of the if-clause and, when you're done, move onto the next statement after the if-statement. If the condition is *falsy*, then evaluate the code inside the else-clause and, when you're done, move onto the next statement after the if-statement. If there's no else-clause, then simply move onto the next statement if the condition is falsy.

If you're evaluating a if-else-chain of the form

```
if (condition1) {
        ...
}
else if (condition2) {
        ...
}
    ...
else {

}
```

Evaluate each conditional and, if it is true, evaluate the body of the if-clause and if it is not true move on to the next conditional test.

2.12 Repetition

Very often, there are *subtasks* when we're performing a task. We need to do something again and again. This repetition has a couple of obvious forms and a more subtle one. The two main ones are

1. doing something a *number* of times

2. doing something until there's some change

2.12.1 For-loops

For the first kind of repetition think of times you've said or heard instructions like,

- cut three onions
- put every book on the shelf
- send a letter to each address on the list
- do 20 jumping jacks

When the instruction lists either a number of times to perform an action or specifies a collection of *things* that you need to act on. Both of these are going to be handled in JavaScript with what are called **for loops**.

The basic structure of a for-loop is something like

```
for(var i=0; i < 10; i = i+1){
    console.log(i);
}
```

where you have

1. the JavaScript keyword `for`, followed by three semi-colon separated things in parentheses:
   (a) the initialization of a variable to be used to count (`var i =0`)
   (b) how you know when you're done (`i < 10`)
   (c) what the next step should be (`i = i+1`)
2. the body of the function (`console.log(i)`) in braces

Exercise: Putting all these pieces together, what's your guess about what this code does?

Now, our examples above for real-life analogues of for loops mention the ideas of for each and for every, but it might not be obvious how for-loops accomplish that! So, generally, whenever you have a task that's for each thing in some collection then this means that you have some kind of list, which in JavaScript means an *array*, and then you want to do *something* for each element of the array.

In JavaScript, and many other languages with for-loops, this means you loop over all the valid slots, the indexes, in the array. What are the valid indexes of an array, though?

Consider the array `["thing 1","thing 2","chicken"]`. This array has *three elements*. The valid indexes are then `0`, `1`, and `2`. In fact, for every array of length `n` then the valid indices are 0 through `n-1`, which are all the numbers greater than or equal to `0` *less than* `n`.. So if we want to print out every element of an array we can do something like

```
var arr = ...;

for(var i=0; i < arr.length; i = i+1){
    console.log(arr[i]);
}
```

Now, for-loops can do more than just count **up** by one, even though that's probably the most common use you'll see. [9] You can count down, like in the following code

```
for(var i = 10; i > 0; i = i - 1){
    console.log(i);
}
```

1. Evaluating by hand For loops are

[9] I once taught a class where a student hadn't realized that you were even allowed to put something other than `i++` as the third clause of the for-loop

2.12.2 While-loops

1. General intuition While loops are the other very basic form of repetition in JavaScript. In terms of our everyday experiences, while-loops correspond to the process of doing a task while something else is happening or until something happens.

   Examples of this kind of iteration are

   - driving until you see a house with a rainbow streamer on the post box
   - occassionally stirring until the soy curls are browned on all sides
   - working on your homework until you're too tired
   - while it's raining, keep your umbrella out

   In each of these examples you're still trying to do a task repeatedly, checking to see if something has changed. The only real difference between while and until is how we think of the condition. When we say while, this means that we're going to keep doing *something* **while** some condition is true and when it **stops** being true, stop the task. When we say until, we mean that we're going to keep doing *something* **until** some condition **becomes** true.

   Exercise: Take all the English sentences describing while-loops and change them to use until if they used while and while if they used until

2. Syntax The way you write a while-loop is the following skeleton

```
while (/condition/){
        ...
}
```

   This means that while the condition between the parentheses is true, then run the code inside the while loop.

As an example, here's a while-loop that will double a number until it is larger than 10000

```
var counter = 10;
while (counter < 10000) {
    counter = 2 * counter;
}
console.log(counter);
```

Exercise: Without running the code, what do you think will be printed to the console?

Exercise: What's wrong with the following code and what makes it different than the code above?

```
while (var counter = 10 && counter < 10000){
    counter = 2 * counter;
}

console.log(counter);
```

I want to emphasize that the condition of the while loop is evaluated *every* time we restart the loop. This is in contrast to the for-loop where in a for loop such as

```
for(clause1; clause2; clause3){
        ...
}
```

where `clause2` is evaluated every time but `clause1` is only evaluated once and `clause3` is evaluated only at the end of a loop.

3. Evaluating by hand To evaluate a while loop by hand, *first* evaluate the condition of the while loop. If it evaluates to a *truthy* value, then execute the body of the loop. If it evaluates to a *falsy* value, then

skip to the first line of code after the end of the loop. Whenever you finish executing the body of the loop, repeat this process starting from evaluating the condition of the while loop.

3 Closures and Scope

3.1 Inner and outer functions

What happens when we define a function *inside* another function?

We've talked a little about variables and scope. To review, we know that when you *execute* a function you are creating a new scope, or "variable table", that, whenever you use a variable, is checked before all the other scopes that are in play. Much of the time, this new scope is no longer used after you exit the function, but there's one case where it can live on indefinitely: when you have an *inner* function that you return as a value.

This is a consequence of JavaScript's scoping rules, which say that the place you look for the value of a variable is defined by where the variable is located in the *text* of the program. This is called *lexical* scoping. That's a bit abstract so let's look at a few examples.

First, here's a simple example. In the following program a function is defined and called: what value will it print out?

```
var number = 0;

function testFunction () {
    var number = 20;
    console.log(number);
}

testFunction();
```

If you test this yourself you'll see that it prints *20* because when you call the function, you create a *new* variable named `number` in the **new** scope of the function and when you use the variable `number` in the `console.log` call, you look up the value in the new scope and find the value of 20.

What about a slightly more complicated example? Instead of defining a function that prints, we define a function that **returns** a function that prints. What value does this program print out now?

```
var number = 0;

function testFunction () {
    var number = 20;
    return function () {
        console.log(number);
    }
}

var fun = testFunction();

fun();
```

It *still* prints out 20! You might think that when we call `fun()` at the bottom of the program the code

```
function () {
    console.log(number);
}
```

would look at the value at the *global* scope, not the scope of `testFunction`. Languages that do that are called *dynamically* scoped. Instead, JavaScript, as a *lexically* scoped language looks at what the `number` would have pointed to at the time the function was *defined*.

Meanwhile, if we print out `console.log(number)` at global scope we'll print out the number 0 instead of 20.

Further, variables defined in the outer function are *only* visible to the inner function. If we try an example such as

```
function outerFun () {
    var thing = "I'M HIDDEN";
    return function () {
        return thing;
    }
}

var thinger = outerFun();

console.log(thinger());
console.log(thing);
```

Then we'll

For example, if you navigate to the file `counterExample.html`, which runs the following code

```
function outer () {
    var counter = 0;
    return function (){
        counter = counter + 1;
        console.log(counter);
    }
}

inc = outer();

inc();
inc();
inc();
```

you can see that the number the function `inc` prints out changes each time, because the variable `counter` which was defined in the scope of the `outer` function is still being referred to by the body of the function defined inside `outer`.

As an exercise, try running through this example by hand. Keep in mind the rules of scope for inner functions.

We call a function like `inc` that can refer to variables "hidden" from normal view a *closure*.

3.2 Closures and objects

Closures become even more useful once we combine them with objects. For example, let's say we want to represent a counter as an object like

```
var counter = { value : 0,
                inc : function () {
                    counter.value = counter.value + 1;
                    return counter.value;
                },
                dec : function () {
                    counter.value = counter.value - 1;
                    return counter.value;
                }
              }
```

but there's a slight problem with object. You can just change the `value` property to whatever you want by setting the property as normal as in the following example

```
counter.value = "thing";
```

which means that if later in your program you call `counter.inc` then you'll get a rather unexpected result:

```
thing1
```

4 Advanced Iteration

5 Appendix: Evaluating Code By Hand

5.1 General Rules and Setup for Interpreting a Program

First, mark down a box labeled "current line". Every step you take, make a note of what line you're on.

You'll start at the first line of the program and, unless some rule specifies otherwise, go to the *next* line of code after you're finished with each line.

Also make a special section labled "output", which you'll use every time something is written to the console by the program.

If a line of code is an expression **only**, evaluate the expression as normal then **throw away** the return value of the expression.

5.1.1 Variable declaration

Look at your program. For all of the instances you see of `var name` or `var name = expression` (that isn't in the body of a function (and if you haven't seen functions yet, don't worry)), make a table that looks like

name1 name2 name3 name4 ...

It should have one column for each variable name.

You don't actually fill anything **in** to start, instead if there's a `= expression` portion of the variable declaration you wait until the line in question is reached before filling in the entry in the table according to the rules of the assignment expression.

5.2 Expressions

If an **expression** is the only thing on the line, evaluate the expression according to the appropriate rules for that expression.

5.2.1 Arithmetic

Numbers evaluate to themselves. Arithmetic operations evaluate exactly according to their them to: `+` is addition, `-` is subtraction, etc.

5.2.2 Strings

Strings evaluate to themselves. The `+` operator "concatenates" two strings together.

5.2.3 Booleans

`true` evaluates to `true`, `false` evaluates to `false`.

The boolean operator `!` takes an expression. Evaluate `! exp` by first evaluating the expression `exp`. If it returns a truthy value, then return `false`. If it returns a falsy value, then return `true`.

The short-circuiting operators `&&` and `||` have special rules. `exp1 && exp2` is evaluated by first evaluating `exp1`, if it is truthy then evaluate `exp2` and return its value. If it is falsy, then return the value of `exp1`.

`exp1 || exp2` is evaluated by first evaluating `exp1`. If it is truthy then return the value of `exp1`. If it is falsy then evaluate `exp2` and return its value.

As a reminder, falsy values are `NaN`, `null`, `undefined`, `0`, `~"~"~`, and `false`. Everything else is truthy.

5.2.4 Assignment

Assignment is always of the form `name = expression`. First, you evaluate the expression based on the kind of expression it is, then fill whatever value it returns **into** the appropriate entry in the table.

The value you wrote into the table is also the value returned by the expression.

5.2.5 Output to console

For purposes of "being the interpreter", we're going to treat the function `console.log` as a special operation. When you see an expression of the form `console.log(exp)`, evaluate the expresion that is the argument, then write the value in the output column you've set aside. As an expression, `console.log` returns `undefined`.

5.2.6 typeof

The `typeof` operator takes an *expression* as an argument. Evaluate this expression is and return, as a string, the type of the value returned according to the following rules

- numbers return "number"
  - this includes `NaN` and `Infinity`
- strings return "string"
- undefined returns "undefined"
- objects return "object"
- booleans return "boolean"

5.2.7 Variables resolution

To evaluate a variable, you have to first consider where the variable's `var` statement is and you then you examine the corresponding table that you made. If there is an overlap in names between two tables that are both visible from a point in the code, precedence goes to the more recently created table.

5.2.8 Function calls

A function is called when it is passed zero or more arguments. For example, `fun()`, `fun(1)`, `fun(1,2)`, etc. are all valid function calls.

A function call is evaluated by:

1. substituting the passed in values for the arguments of the function, which means everywhere the formal argument was seen in the function body, rewrite it to be the corresponding value
2. evaluate the body like you would a new program
   (a) make a variable table
   (b) evaluate each statement sequentially
   (c) if there is a return statement, then **stop** executing the function, go back to the point of where the function was called and hand back the value of the expression passed to the `return`
   (d) if there is no return statement by the end of the function, return `undefined`

1. A caveat on variable tables for functions After exiting the function, if there is nothing else that can reference the function's local variable table, then you may erase the table.

   If, on the other hand, that table is still visible to some entity in the program, you may **not** erase it and must keep the variable table in play.

5.3 Object specific expressions

5.3.1 General object layout

An object is represented as a table a list of pairs of

- a property name
- the value corresponding to the property

5.3.2 Objects and variables

An important note about variables and objects. A variable never holds a literal object. Instead, what the variable contains is an "arrow" that points to the object. The "value" of an object is, then, simply the pointer rather than the object itself.

The implication of this is that there's no

5.3.3 Object creation with new

Objects can be created using the `new Constr()` syntax. This is evaluated by

1. creating a new object
2. setting the `.constructor` property to the constructor function
3. running the constructor function with `this` bound to the new object
4. returning **a pointer to** the new object after the constructor function finishes running

An object created with the `{}` or `{ prop : val, prop : val, ...}` syntax is equivalent to an object created using `new Object()` that then has the corresponding properties, if any, set.

5.3.4 Object property access and assignment

An object's properties can be accessed through two methods: the "dot" syntax `obj.prop` or the "array" syntax `obj["prop"]`. These are evaluated identically, the only distinction is the names that are allowed to be used for the properties: the array syntax is far more permissive with allowed names.

You evaluate property access by looking up the value of the property in the object and returning it. If the property isn't in the table corresponding to the object, first check the prototype of the constructor of the object. If the property isn't in the prototype or the prototype's prototype etc., then return undefined. When searching for a property, the first place you find it takes precedence and you return with **that value** immediately and do not continue searching up the prototype tree.

You evaluate property **assignment** by first evaluating the expression to the right of the `=` and putting that value into the table corresponding to the object, making a new space for the property if there isn't already one in the object.

5.3.5 this

The statement `this` acts like a variable with special evaluation rules. There's two different ways in which `this` can be used

1. in the constructor of an object
2. in a function to be called **by** an object

In the first case, when `new Cons()` is called to make a new object using the constructor `Cons`, `this` is a reference to the fresh object that is being constructed. See also the section on object creation.

In the second case, when a function is called **as a method**, `this` points to the parent object.

If `this` is encountered outside of these two cases, then it resolves to the "global object" of the program.

5.4 For loops

A basic for loop has the form

```
for (initialization; condition_for_continuing; next_step){
    statement1;
    statement2;
    statement3;
    ...
}
```

It's not **strictly** required, but you should make the "initialization" code only be of the form `var name = exp` or `name = exp`. The condition for continuing the loop should be an expression that returns a boolean. The next step slot should be an assignment expression that modifies the variable named in the initialization.

The rule is that you

1. execute the code in the "initialization" slot
2. evaluate the condition for continuing
   (a) if it is truthy, go to step (3)
   (b) if it falsey, jump to the line of code **after** the end of the for loop
3. execute the statements in the for loop
4. execute the code in the "next step" part of the for loop
5. go to step (2)

5.5 While loops

A while loop has the form

```
while (condition){
    statement1;
    statement2;
    statement3;
    ...
}
```

The rule is that you

1. evaluate the condition
   (a) if it is truthy, go to step (2)
   (b) if it is falsey, jump to the line of code **after** the end of the while loop
2. execute the statements in the while loop
3. go to step (1)

5.6 If statements

If statements have the basic form

```
if (condition){
    statement1;
    statement2;
    ...
}
else {
    morestatement1;
    morestatement2;
    morestatement3;
}
```

The rule for them is that you

1. evaluate the condition
    (a) if it is truthy, perform the statements listed between the braces of the "if"
    (b) if it is falsy, perform the statements listed between the braces of the "else"

The other form of if-statement is to leave out the `else` branch. In this case, our rule reads

1. evaluate the condition
    (a) if it is truthy, perform the statements listed between the braces of the "if"
    (b) if it is falsy, do nothing

5.7 Function declarations

There are two function declarations. There is the **expression** form which has the following syntax

```
function (arg1, arg2, ...) {
    statement1;
    statement2;
    statement3;
}
```

This evaluates to a function value, which in our pen and paper we'll represent as a box that

- contains the list of arguments to the function
- the lines of code for the body of the function
- an arrow pointing to the variable table within which the function was defined (this is important for calling functions!)

The second kind of function declaration, which is a **statement**, is the named function declaration, which has the following syntax.

```
function name (arg1,arg2,arg3) {
    statement1;
    statement2;
    statement3;
}
```

You evaluate this by treating it as equivalent to

```
var name = function (...){
   ...
};
```

6 Appendix: Numbers That Aren't Numbers

6.1 When is a number not a number?

When is a number not a number? When it's *not a number*! Now that sounds like some kind of bad joke, but there is actually a "number" called `NaN`, which stands for "not a number", in JavaScript. This "number" is how JavaScript denotes that at some point a numerical calculation *went wrong*, for example if you type the expression `3 / ''chicken''` into the console then you'll get back `NaN`. You may have already discovered it if you tried something that is ill-defined mathematically such as `0/0`.

Once you're trapped in `NaN` land you can't get back out again.

> Try typing the following expressions into the console just to see what happens:
>
> 1. `NaN + NaN`
> 2. `NaN * 0`
> 3. `1 + NaN`

6.2 Infinity

The other odd number you might encounter in JavaScript is `Infinity`. `Infinity` is the "number" you get when you perform an operation such as dividing a non-zero number by zero, such as `1/0`.

Now, as you might expect a number of operations on `Infinity` still yield `Infinity`. For example,

1. `0 + Infinity`
2. `Infinity - 10`
3. `Infinity * 2`
4. `Infinity / 0`

will all output `Infinity`, but `Infinity*0` is going to be `NaN`.

6.3 Why have these?

In day-to-day programming, you won't encounter these pseudo-numbers very often. They're much like `undefined` in that their use is to signal that *something went wrong*, but instead of `undefined` which could be the result of many different kinds of errors, `Infinity` and `NaN` have very particular causes involving numeric operations.

The fact that JavaScript is so nice about dividing by zero is somewhat rare. An older programming language such as C or C++ has "undefined behavior" when dividing by zero, which means that you can't rely on your program behaving predictably when this error happens.

6.4 Evaluation By Hand

`Nan` and `Infinity` evaluate to themselves. Any numeric operation involving `NaN` will result again in `NaN`. Any numeric operation on `Infinity`, other than multiplying by `0`, will give back `Infinity`.