ch 3 and 6 notes, yay midterms!, going to be lazy again.
Eloquent JavaScript:
Chapter 3: Functions:
Summary:
Functions are the "bread and butter of JavaScript programming," where its definition is a regular variable definition with its value given being a function. There's a tradeoff in elegance vs. efficiency between recursion that keeps calling a function and just looping, but recursion really shines when used with complicated things like trees. One of the key features of functions is their local scope, where parameters and variables called within cannot be seen by outside the program, but they can still use the global variables outside. Finally, they concluded the chapter by mentioning how repeating the same code over and over is bad and that you should try to write a pure function that will always return the same value with the same arguments.

Notes:
Functions are the "bread and butter of JavaScript programming," where its definition is a regular variable definition with its value given being a function.

Functions expressions start with the keyword function:
-have a set of parameters
-and a body contain statements to be executed if called
    -has to be wrapped in braces

return value determines the value the function returns.

Parameters and Scopes:
The parameters to a function behave like regular variables, but their initial values are given by the *caller* of the function, not the code in the function itself.

variables created inside of them, including their parameters, are *local* to the function.

This “localness” of variables applies only to the parameters and to variables declared with the var keyword inside the function body. Variables declared outside of any function are called *global*, because they are visible throughout the program.

Nested Scope:
JavaScript distinguishes not just between *global* and *local* variables. Functions can be created inside other functions, producing several degrees of locality.

Each local scope can also see all the local scopes that contain it.

Functions as Values
Function variables usually simply act as names for a specific piece of the program.

A function value can do all the things that other values can do—you can use it in arbitrary expressions
    It is possible to store a function value in a new place, pass it as an argument to a function, and so on

Declaration Notation

There is a slightly shorter way to say “var square = function…”. Thefunction keyword can also be used at the

start of a statement, as in the following:

```
function square(x) {
  return x * x;
}
```

Function declarations are not part of the regular top-to-bottom flow of control. They are conceptually moved to the top of their scope and can be used by all the code in that scope.

Every time a function is called, the current context is put on top of this “call stack”. When the function returns, it removes the top context from the stack and uses it to continue execution.

Optional Arguments

JavaScript is extremely broad-minded about the number of arguments you pass to a function. If you pass too many, the extra ones are ignored. If you pass too few, the missing parameters simply get assigned the value undefined.

Closure

What happens to local variables when the function call that created them is no longer active?

local variables really are re-created for every call—different calls can’t trample on one another’s local variables.

This feature—being able to reference a specific instance of local variables in an enclosing function—is called *closure*.

Recursion

```
function power(base, exponent) {
  if (exponent == 0)
    return 1;
  else
    return base * power(base, exponent - 1);
}

console.log(power(2, 3));
// → 8
```

The function calls itself multiple times with different arguments to achieve the repeated multiplication.

In typical JavaScript implementations, it’s about 10 times slower than the looping version. Running through a simple loop is a lot cheaper than calling a function multiple times.

Often, though, a program deals with such complex concepts that giving up some efficiency in order to make the program more straightforward becomes an attractive choice.

Then there's a really nice example showing the steps that the recursion goes through, so check it out.

Growing Functions:
The first is that you find yourself writing very similar code multiple times. We want to avoid doing that since having more code means more space for mistakes to hide and more material to read for people trying to understand the program. So we take the repeated functionality, find a good name for it, and put it into a function.

Possibly breaking down into smaller functions for the functions to use.

Functions and Side Effects
Functions can be roughly divided into those that are called for their side effects and those that are called for their return value.

A pure function has the pleasant property that, when called with the same arguments, it always produces the same value (and doesn't do anything else).

Chapter 6 : Objects
Summary:
Objects in JavaScript contain a lot of elements used in object oriented programming that was mentioned throughout the chapter. Before that, most, if not all objects contains a prototype which is another object used as fallback sources of properties, but you can choose to not give an object a prototype. Constructors can make new objects as long as you use the new operator with its prototype being found in it property in constructor function. Now some of the big things mentioned and given huge examples were polymorphism (ability to process objects differently depending on their data type or class) and inheritance (an object or class is based on another object or class, using the same implementation specifying implementation to maintain the same behavior).

Notes:
Method:
Methods are simply properties that hold function values. This is a simple method.

When a function is called as a method—looked up as a property and immediately called, as in object.method()—the special variable this in its body will point to the object that it was called on.

Prototypes:
In addition to their set of properties, almost all objects also have a *prototype*. A prototype is another object that is used as a fallback source of properties. When an object gets a request for a property that it does not have, its prototype will be searched for the property, then the prototype's prototype, and so on.

So who is the prototype of that empty object? It is the great ancestral prototype, the entity behind almost all objects, Object.prototype.

Constructors:
In JavaScript, calling a function with the newkeyword in front of it causes it to be treated as a constructor. The constructor will have its this variable bound to a fresh object, and unless it explicitly returns another object value, this new object will be returned from the call.

An object created with new is said to be an *instance* of its constructor.

Constructors (in fact, all functions) automatically get a property namedprototype, which by default holds a plain, empty object that derives fromObject.prototype. Every instance created with this constructor will have this object as its prototype.

Overriding Derived Properties

Calling toString on an array gives a result similar to calling .join(",") on it—it puts commas between the values in the array. Directly callingObject.prototype.toString with an array produces a different string. That function doesn't know about arrays, so it simply puts the word "object" and the name of the type between square brackets.

Prototype Interference
A prototype can be used at any time to add new properties and methods to all objects based on it.

Oddly, toString did not show up in the for/in loop, but the in operator did return true for it. This is because JavaScript distinguishes between *enumerable*and *nonenumerable* properties.
All properties that we create by simply assigning to them are enumerable. The standard properties in Object.prototype are all nonenumerable, which is why they do not show up in such a for/in loop.

Prototype-less objects
You are allowed to pass null as the prototype to create a fresh object with no prototype. For objects like map, where the properties could be anything, this is exactly what we want.

Polymorphism

This is a simple instance of a powerful idea. When a piece of code is written to work with objects that have a certain interface—in this case, a toStringmethod—any kind of object that happens to support this interface can be plugged into the code, and it will just work.

This technique is called *polymorphism*—though no actual shape-shifting is involved. Polymorphic code can work with values of different shapes, as long as they support the interface it expects.

Then there's a huge section with an example of how it works, check It out!

Getters and Setters
When specifying an interface, it is possible to include properties that are not methods.

We can specify properties that, from the outside, look like normal properties but secretly have methods associated with them.

In object literal, the get or set notation for properties allows you to specify a function to be run when the property is read or written. You can also add such a property to an existing object, for example a prototype, using theObject.defineProperty function (which we previously used to create nonenumerable properties).

Inheritance:

This pattern is called *inheritance*. It allows us to build slightly different data types from existing data types with relatively little work. Typically, the new constructor will call the old constructor (using the call method in order to be able to give it the new object as its this value). Once this constructor has been called, we can assume that all the fields that the old object type is supposed to contain have been added.

The Instanceof Operator:

It is occasionally useful to know whether an object was derived from a specific constructor. For this, JavaScript provides a binary operator calledinstanceof. The operator will see through inherited types.