You Don’t Know JavaScript Yet notes

Chapter 1:

Javascript has no real relation with Java. It was named Javascript since it was designed to appeal to mostly Java programmers. The official name for the language given by the ECMA Standards body is ECMAScript.

Javascript is a multi-paradigm language, meaning it can support procedural, class-oriented, or FP-style code, and that can be constantly changed throughout while you write code.

Another nice feature is that one of JS principles is backwards compatibility. Meaning that technically code that was written in the beginning when JS was created (1995) should still work today and will not become invalid JS code. A common saying among JS programmers is: “We don’t break the web!”.

Something odd about JS is that even though it’s backwards compatible, it is not forwards-compatible. Usually if you run a new ES feature on a browser running on an old ES engine, very probably the program will break and crash. Due to this kind of things, programmers must take special care when coding. A solution for this is transpiling. It is a process to convert newer syntax JS to and older syntax JS, compatible with what the engine is running.

JS also has something called *Strict Mode.* Strict Mode is a way of coding JS that enforces correct writing of the code. Meaning everything must be written correctly, without having missing semicolons or useless pieces of code. For example, if there’s a single extra semicolon somewhere, the code will throw an error. Strict mode can also be turned on per file, meaning that one file can have strict mode while another doesn’t. Or it can also be turned on per function, but this isn’t encouraged since it is better to turn on strict mode for the entire file.

Chapter 2:

In JS, each file is treated as its own separate program. This is important since one file may fail but that will not necessarily prevent the next file from being executed or from breaking the site. One way that multiple files can act as a single program is by sharing their state via the “global scope”. JS also supports a module format. Modules are also file-based. If a module is loaded through the *import* statement of by *<script type=”module”>* .

Value is a fundamental unit of information in a program. Values are data. Values in JS come in two forms: **primitive** and **object**. For example, some primitive value types are strings, Booleans, numbers, null, and undefined. Along with primitive values, there are object values. Arrays are a type of object value that is comprised of an ordered indexed list of data. Arrays can hold any type of value, whether it’s a primitive value or another object (also other arrays).

Objects are an unordered, keyed collection of any various types. Meaning that it can hold several types of values that can be accessed through the location name (or key) property. For example: name = {

first: "Kyle",

last: "Simpson",

age: 39,

specialties: [ "JS", "Table Tennis" ]

};

Variables are containers for values. There are a few ways to created variables: **var, let, const**. The **var** keyword creates a variable and gives you the option to initialize it. For example: var name = “Jose”. **Let** is similar to **var** but **let** is more widely used for block-scoped variables. **Const** is similar to **let**, except that **const** must be given a value when declared, and that value cannot be reassigned.

Even though functions could have many meanings, a function in JS is a collection of statements that can be invoked multiple times, could be provided different inputs, and may give back one or more outputs. Functions look a little like this:

function hello(name){

return amazingName;

}

Functions can also be assigned to a variable. For example:

var awesomeFunction = function(coolThings){

return amazingStuff;

}

As seen in the declarations above, functions can receive parameters and return values. Only a single value can be returned from a function. Although, if multiple values need to be returned, they can be wrapped in an array or in an object.

JS has several ways to compare values. For example, the double equals sign (==) and the triple equals sign (===) can be used. Both are similar but do differ in a way. The double equals sign only compares the value, so if both values compared are equal, the operation returns true. Meanwhile the triple equals sign compares value *and* type. So, if two values being compared are equal but have different types (for example one is a number and the other is a string), the operation will return false.

There are two major patterns for organizing code: classes and modules. Even though classes have many definitions, in JS they are a definition of a “type” of custom data structure that includes both data and behaviors that operate on that data. Classes define how that data structure works, but classes themselves are not concrete values. Something that is also available in through classes is Inheritance. This behavior allows child classed to access data from its parent class.

Like the class pattern, the module pattern has the same goal. Also, modules can “include” or “access” the data and behaviors of other modules, just like with classes. But they do have some differences, like the syntax for example. Classic modules have an outer function that runs at least once and returns an instance of the module with functions exposed that can operate with the module’s (hidden) data. In modern JS modules, there’s no wrapping function to define a module. Each file is its own module. Different to classic modules, modern modules aren’t instantiated but rather you just use “import” use a single instance of it.

Chapter 3:

The iterator pattern has been around for decades and suggests a “standardized” way to consume data from a source one chunk at a time. The idea for this is that it’s more helpful to iterate the data source, part by part, rather than handling the entire data set at once. One mechanism for these iterations is to use loops. For example, the *for…of* loop. Another mechanism to consume from iterators is the ***…*** operator. This operator has two forms: *spread* and *rest.* What *spread* does is that it retrieves all values from the iterator and places them into another container (array, argument list, etc). In short, what it does is that it copies the values from one container to another, with each value from the original container occupying a place in the new container. An iterable is a value that can be iterated over. ES6 defined basic data structures/collections types as iterables.

Closure is when a function remembers and continues to access variables from outside its scope, even when the function is executed in another scope. This is something that a lot of developers have probably used without realizing. Closure is part of the nature of a function. Objects don’t get closure, but functions do. To observe a closure, a function must be executed in a different scope to where it was originally defined. Closure is most common when working with asynchronous code, such as callbacks. For example:

**function** getSomeData(url) {

ajax(url,**function** onResponse(resp){

console.log(

`Response (from **${** url **}**): **${** resp **}**`

);

});

}

getSomeData("https://some.url/wherever");

// Response (from https://some.url/wherever): ...

The onResponse function is closed by the url function, that way it is preserved and remembered until the Ajax function returns. Closure allows the onResponse to be kept alive for as long as needed.

The *this* keyword in JS is one of the most used and one of the most misunderstood concepts in Javascript. One of the things that it does is that it exposes the *execution context* of a function. One way to understand the *execution context* is that it’s a tangible object whose properties are available for a function while it’s executing. When there is no *execution context* provided, the default value for the *this* keyword is the global *window* object.

A prototype is a characteristic of an object, and specifically resolution of a property access. A prototype is kind of a linkage between two objects. This prototype linkage happens when an object is created; it’s linked to another object that already exists. The purpose for this linkage (from object A to B) is so accesses against B for properties/methods that it does not have, are delegated to A to handle. This allows two or more objects to work together to perform a task. For example:

**var** homework = {

topic: "JS" };

The *homework* object only has the *topic* property. By default, the prototype linkage connects to the *Object.prototype.Object* which has many methods, like the toString() or valueOf() methods.

A prototype linkage can be created using the Object.create(…) method:

**var** homework = {

topic: "JS"

};

**var** otherHomework = Object.create(homework);

otherHomework.topic; // "JS"