**An High-Level Overview of JavaScript**

* High Level: Automatically, for example, manage resources because of abstraction. Makes languages easy to learn but has the downside of programs will not be as fast as low-level languages such as C.
* Garbage-collection: Algorithm in JavaScript engine that automatically removes old unused objects from the computer memory in order not to clog it up with unnecessary stuff.
* Interpreted or just-in-time compiled
* Multi-paradigm: A paradigm is an approach and mindset of structuring code, which will direct your coding style and technique.
  + JavaScript can be any:
    - Procedural programming
    - Object-oriented programming (OOP)
    - Function programming (FP)
* Prototype-based object-oriented: Use of classes such as Array to create, for example, a simple array. (Ex. const array = [1,2,3]; The array inherits from Array class so that array methods becomes usable such as push, pop, shift, etc.)
* First-class functions: Functions are simply treated as variables. We can pass them into other functions, and return them from functions.
* Dynamic: The data types become known at runtime.
  + Single-threaded:
    - Concurrency model is how the JavaScript engine handles multiple tasks happening at the same time.
    - JavaScript runs in one single thread.
* Non-blocking event loop: Takes long running tasks, executes them in the background, and puts them back in the main thread once they are finished.

**The JavaScript Engine and Runtime**

* JavaScript Engine: Program that executes JavaScript code
  + Google V8: Powers Chrome and nodeJS.
* JS Engine contains:
  + Call Stack: Code is executed using something called execution context.
  + Heap: Unstructured memory pool which stores all the objects that an application needs.
* Compilation vs Interpretation
  + Compilation: Entire code is converted into machine code at once, and written to a binary file that can be executed by a computer.A picture containing graphical user interface

    Description automatically generated
  + Interpretation: Interpreter runs through the source code and executes it line by line. Diagram

    Description automatically generated with medium confidence
    - JavaScript used to be interpreted language but it was really slow.
  + Just-in-time (JIT) compilation: Mix between compilation and interpretation. Entire code is converted into machine code at once, then executed immediately. JavaScript currently uses.Diagram

    Description automatically generated
* Graphical user interface, application

  Description automatically generatedModern Just-in-time Compilation of JavaScript
  + Parsing: Read the code. During the process, the code is parsed into a data structure called the Abstract Syntax Tree (AST). This works by first splitting up each line of code into pieces that are meaningful to the language such as const or function keywords and then saving all the pieces into a tree in a structured way. Also checks for syntax errors. The resulting tree is used later to produce machine code.
  + Compilation: Takes the generated AST and compiles it to machine code.
  + Execution: The machine code gets executed immediately. The code is executed in the JavaScript Engine’s call stack.
  + Modern JavaScript has a clever optimization strategy. It creates an unoptimized copy of the machine code in the beginning just so that it can start executing as fast as possible. In the background, this code is being optimized and recompiled during the already running program execution. Can be done multiple times. After each optimization, the unoptimized code is simply swept for the new more optimized code without stopping execution. Reason why V8 is so fast.Diagram

    Description automatically generated
* JavaScript Runtime
  + JavaScript Engine (in the browser)
    - Call Stack
    - Heap
  + Web APIs: Functionalities provided to the engine but not part of the JavaScript language. Accessible on window object.
  + Callback Queue: Data structure that contains all the callback functions that are ready to be executed.
    - When an event occurs, for example a click, the callback function is put in the callback queue. Then when the call stack is empty, the callback function is passed to the stack so that it can be executed. Happens with the help of the event loop. The event loop is the reason for non-blocking concurrency model.
  + NodeJS Runtime
    - Similar to browser except there are no web APIs and instead have C++ bindings and thread pool.

**Execution Contexts and the Call Stack**

**Diagram

Description automatically generatedWhat is an Execution Context?**

Execution of JavaScript code (after compilation):

* Creation of global execution context (for top-level code; NOT inside a function).
  + Function body executes when called.
  + Execution Context – Environment in which a piece of JavaScript code is executed. Stores all the necessary information for some code to be executed. Only one global execution context per JavaScript file.
* Execution of top-level code (inside global execution context).
* Execution of functions and waiting for callbacks.
  + One execution context will be created per function.

Diagram, schematic

Description automatically generated

**Execution Context in Detail**

What’s inside the execution context?

1. Variable Environment
   1. let, const, and var declaration
   2. Functions
   3. Arguments object (arrow functions do not have)
2. Scope Chain: Consists of references to variables that are located outside of the current function
3. this keyword (arrow functions do not have).

**The Call Stack**

Call Stack – The “place” where execution contexts get stacked on top of each other to keep track of where we are in the execution.

Two kinds:

* Global Execution Contexts – Always first on the call stack and last to leave the stack.
* Function Execution Contexts – Can have as many as there are functions in a file.

**Scope and the Scope Chain**

**Scoping and Scope in JavaScript**

Scoping – How program’s variables are organized and accessed.

Lexical scoping – Scoping is controlled by placement of functions and blocks in the code

Scope – Space or environment in which a certain variable is declared (variable environment in case of functions).

* Global
  + Outside of any function or block.
  + Variables declared in global scope are accessible everywhere.
* Function
  + Variables are accessible only inside function, NOT outside.
  + Also called local scope.
* Block (ES6)
  + Variables are accessible only inside block (block scoped). \*Only applies to let and const variables
  + Functions are also block scoped (only in strict mode).

Scope of a variable – Region of code where a certain variable can be accessed.

**Summary**

* Scoping asks the question “Where do variables live? “ or “Where can we access a certain variable, and where not?”
* There are 3 types of scope in JavaScript: global, function, and block.
* Only let and const variables are block-scoped. Variables declared with var end up in the closest function scope.
* In JavaScript, we have lexical scoping, the rules of where we can access variables are based on exactly where in the code functions and blocks are written.
* Every scope always has access to all the variables from all its outer scopes. This is the scope chain.
* When a variable is not in the current scope, the engine looks up in the scope chain until it finds the variable it’s looking for. This is called variable lookup.
* The scope chain is a one-way street; a cope will never have access to the variables of an inner scope.
* The scope chain in a certain scope is equal to adding together all the variable environments of all parent scopes.
* The scope chain has nothing to do with the order in which functions were called. It does not affect the scope chain at all.