1. JavaScript does only two things: It goes through the code line-by-line and runs each line (Thread of Execution). It threads its way down the code and executes it. The second is, it saves data as well as code (functions) in its memory for later use.

2. While running code, it stores values in the memory with their identifiers. The code of a function is saved in memory just like you would a string of characters.

3. A variable's value varies (by definition). It is better to call 'const' identifiers constants.

4. A function is like a mini-program/application so it needs its thread of execution and memory (when combined, called its Execution Context).

5. When we run a program, we create a global Execution Context. When we call a function, we create a mini Execution Context.

6. Any identifier declared in the function is stored in the Local Memory of the function's Execution Context, and not in the Global Memory. The arguments are also stored there.

7. We cannot store a function call expression but the result of its evaluation which is the return value of that function.

8. JavaScript can only work on one Thread of Execution at any given time.

9. When a function is called, the Thread of Execution weaves into the TOE of the function's Execution Context and weaves out to the previous TOE when the function returns.

10. JavaScript keeps track of what function is running by using the Call Stack.

11. When a function is called, it is added to the top of the Call Stack. JavaScript determines the current Thread of Execution and Local Scope of the program by looking at what is at the top of the Call Stack. JavaScript only deals with the top of the Call Stack at any given time.

12. When a function returns (implicitly or explicitly), it is popped off the top of the Call Stack.

13. The Call Stack will always have the Global Execution Context on it. Like all our code is inside a global function that is pushed to the Call Stack for the entire duration of the program.

14. The Execution Context is deleted once the function returns.

15. DRY (Don't Repeat Yourself).

16. Higher-Order Functions take in functions as arguments so their functionality is dynamic and decided at runtime, making them more generalized.

17. You can't edit functions once declared, but if you leave a space for another function (receive it as an argument), you can dynamically insert and change the functionality

in that space.

18. Any object, array, and function is saved in memory, and only a reference or pointer to that is passed around, which when used gives us access to the original object in memory.

19. Block scoping using ‘let’ and ‘const’ does not create a new Execution Context. It just creates its protected namespace. It is just a protected space in memory, only for the identifiers in that block.

20. Functions are first-class objects in JavaScript. They are fully featured objects and can be treated just like one.

21. A function that takes in another function is the Higher-Order Function and the function that is passed in as the argument is the Callback.

22. A function is a HOF, also if it returns out a function.

23. When returning a value from an Arrow Function without using curly braces and the ‘return’ keyword, JavaScript inserts it for you.

24. The name of a Callback function is the name of the parameter to which it was assigned.

25. Arrow Functions are lexically scoped for the value of the ‘this’ keyword.

26. We cannot call anonymous functions recursively.

27. Using Closure, we can define functions like once() (which can turn other functions into functions that only run once and never again) and memoize().

28. Memoization is an optimization technique that lets us not repeat calculations or tasks if we’ve already run them once using the same inputs.

29. JavaScript design patterns like the Module Pattern, Iterators, handling partial application, and maintaining state in Asynchronous Programming, all require or use Closure.

30. Every function call creates a brand new memory in a new Execution Context. So nothing of a function call remains once it completes running. The next run will always create a new and clean Execution Context.

31. The Local Memory of a function can also be called a (or its) Variable Environment or State (the live data being stored in an application at any particular moment).

32. Closure enables the function definition itself to have a permanent memory even before it is invoked. This helps the function to remember things from previous executions.

33. JavaScript is a Synchronous language, meaning we run a line and never look at that line ever again.

34. After a function has been returned from another function, it has zero relationships with the Higher-Order Function that returned it.

35. When you call a function by name, the thread or current line being run doesn't go to where the function was defined as we would visualize. It just runs the function that has been stored in memory by that name.

36. Even when we are calling functions that were defined inside another function, it doesn't go to the first line of the definition and start executing it. It takes the function from the memory of the current Execution Context and runs it.

37. When a function is popped off the Call Stack, its Execution Context is deleted as well, even if it returned another function.

38. When we return a function from another function, we return not only that function but all the data surrounding that function (scope) with it. The function definition with that data can be stored or called, which enables us to use variables that aren't really in the Local Memory of that function but is in the memory of its parent or up the scope chain. This is called Closure.

39. So it is like the function in memory has a permanent data store, which is its surrounding data from the place of its definition attached to it, which is not in its Local Memory. If it was in its Local Memory, it would've been reset on every call. But this data is persisted since it's the data attached to the function in memory.

40. The permanent data stored with the returned function is like a backpack (called Closure) attached to the function's definition (in memory). When a variable is not found in the Local Memory, it looks in the backpack to find it.

41. The function identifier itself becomes storage for the function definition and its Closure.

42. Not only when we return, but when we define a function, the function definition gets a hidden property called [[scope]], which is a link to all the surrounding data of that function. That hidden property is the backpack (or the link to it). It is returned out from a function with the function definition.

43. When it can't find a variable in the Local Memory, the [[scope]] intercepts and we look there next.

44. We cannot access [[scope]] directly in any way.

45. Closure is not only a permanent memory of a function but also a private memory, since we cannot access it without calling the function itself, which can be written and controlled as we want.

46. Function Decoration is when we pass a Callback to a HOF which returns another function that has access to that Callback. The Callback will be in the [[scope]]

property of that function definition, wherever it goes.

47. Function Decoration is when we edit a function. Above is how it is implemented using Closures. We just use the Callback (decorated function) via the backpack ([[scope]]).

48. When we complete running a function, its Execution Context is deleted and the actual memory used is also cleared. But if we have returned a function that uses anything in that memory, the actual memory with that label(s) is not cleared. A link to all the memory that is used in the returned function is included in the [[scope]] property.

49. In modern compilers, links to all the memory of the surrounding data are not included in the [[scope]] property. Only the variables (links to variables) referenced in the returned function are stored. If something that is not referenced is stored, we call that a Memory Leak, meaning we are storing a value with a label that will never be used anymore and wasting the computer's memory.

50. This backpack is also called C.O.V.E by some, an abbreviation of Closed Over Variable Environment.

51. Scope is some rules in any programming language that determines what data is available to you when running a certain line of code.

52. JavaScript is lexical or static scoped language as the physical position of a function determines the scope of a function for its lifetime. It takes the Variable Environment of its birthplace in its backpack because JavaScript is lexically scoped and that is the scope rules specified for it by whoever wrote it.

53. If it was dynamically scoped, it would look in the next Execution Context in the Call Stack, and [[scope]] wouldn't intercept the lookup.

54. Persistent Lexical or Static Scope Reference Data (PLSRD) is another name for the data (closure).

55. Where a function was born, determines what data it has access to when it is eventually run, wherever that may be.

56. The permanent data ([[scope]]) attached to the function definition is popularly known as the function's Closure.

57. The backpack is the result of JavaScript being a lexically scoped language. So it takes all the surrounding data with a function definition on its [[scope]] property, which is persisted even after the parent function completes running and its Execution Context is deleted.

58. The data in the backpack attached depends and differs based on the Execution Context that the function was defined in. It gets references to the actual memories used by that Execution Context, inside of [[scope]].

59. Memoization is about giving our functions memories of their previous input and output so that a function won't have to do its tasks again to get to a certain output if

that function had already been run once with the corresponding input.

60. Iterators give a function which when run gives out the next value in the sequence, as it stores a counter in the backpack or its Closure ([[scope]]). Generators also use Closure to get the memory up to where it yielded in the previous run.

61. Module Pattern lets you store data available to the entire codebase without polluting the Global Memory with its labels which also has a chance of being overwritten if it’s a variable. So the Module Pattern stores it in the backpack (Closure ([[scope]])). It also provides an API for interfacing with it cleanly.

62. JavaScript only has one thread. The same thread is used and goes through every Execution Context. Only one line can be run at a time.

63. Three core components of JavaScript are the Thread of Execution, the Variable Environment, and the Call Stack.

64. JavaScript has a lot of Facade Functions available in a browser, which is not available in the language itself, like Timer and Sockets.

65. These Facades like setTimeout() and ‘document’ do not do anything in JavaScript but do something or uses a feature of the browser itself, which we interface to using JavaScript.

66. JavaScript is the thing we learned till now. Everything else (most of it) is features or functionalities provided by the browser.

67. Facade Functions are just like commands or messages which is sent down to the browser’s features.

68. The setTimeout() function call is exited from, once it tells the browser to set a timer of the specified duration and gives it a Callback to push to the Call Stack once the timer is complete. So JavaScript doesn’t wait for the completion of the timer and goes on to the next line after sending a message to the browser to set the timer.
Note: setTimeout() returns an ID for the timer.

69. The callback is not run inside setTimeout(). The setTimeout() call triggers the timer, which gets as arguments the duration, and a reference to the Callback in memory, which is to be invoked upon completion of the timer.

70. Once the timer completes, the Callback is enqueued into a queue called the Callback Queue, and not pushed directly to the Call Stack.

71. The Callbacks enqueued to the Callback Queue by any Facade Function is not pushed into the Call Stack unless there is nothing in the Call Stack and only after every synchronous code has been completed running, including the code in the global Execution Context.

72. The Event Loop checks whether the Call Stack is empty or not, before (or after) any line of code is run. If it is not empty, it doesn’t even look at the Callback Queue and lets the next line run. If it is empty and there is no more code to run, even in the

global Execution Context, it goes to the Callback Queue and dequeues the first Callback (if there is one) and pushes it into the Call Stack (calls the function).

73. The Event Loop is constantly running as long as the application is running.

74. When a Callback is invoked on the completion of a background feature (like fetch()), the result, or data from running the feature is passed into the Callback as its argument(s).

75. The above point essentially says that the result of that background feature is only available to the Execution Context of the Callback function which is run upon its completion.

76. Due to the above, a problem called Callback Hell occurs as we may have to run another background feature that uses the data received in the Callback, which forces us to pass in another Callback to another Facade Function. This can lead to multiple levels of nesting, making the code less readable.

77. Promises let us track the status of the background function that is being run, in our JavaScript code (in its memory).

78. For instance, when we run a network request using fetch(), it triggers the background feature (one prong), and it returns a special object called a Promise object (two-prong). Once the background feature has completed running, the Promise object gets filled in or updated with the resulting data from the feature. These functions can be called Two-Pronged Facade Functions.

79. The Promise object is like a placeholder object which allows us to keep track of the background feature in JavaScript's memory (local or global).

80. The background feature and the Promise object are mapped to each other or intimately linked. Once complete, the background feature will have a consequence on that Promise object which was returned when the background feature was triggered in the first place.

81. The Promise object returned from a Two-Pronged Facade Function will have two properties on it.
    ```
    {
      value: undefined,
      onFulfilled: [],
    }
    ```

82. Unlike a normal Facade Function, the two-pronged one does not take and enqueue a Callback on completion. It instead updates the 'value' property on the Promise object with the returned response data from the background feature.

83. A Promise object has a then() method on it. It takes in a Callback function and pushes it into the 'onFulfilled' hidden property, which is an array. It is essentially doing:
    ```
    futureData.onFulfilled.push(callback);
    ```

This cannot be done directly since it is a hidden method and cannot be accessed.

84. On completion of the background feature, the ‘value’ property of the Promise object linked to it is updated with the response data. Once it is updated, all the Callbacks in the ‘onFulfilled’ hidden property (array) are called, and the ‘value’ property’s value is passed in as their argument (so that any side-effects which use that data can be triggered).

85. Asynchronous means doing code out of order from when you saw it (not in its written position but deferred). The thread does not get blocked on a single line of code this way.

86. Two-Pronged functions are about triggering something in the background and getting a nice Promise object in the foreground to keep track of it.

87. The synchronous code gets the highest priority.

88. The Callback Queue is called the Task Queue in the specification.

89. Callbacks in Promise objects, like in the ‘onFulfilled’ array (inserted into it by then()), on completion of the background feature, gets enqueued to the Microtask Queue and not the Callback Queue. The event loop dequeues from the Microtask Queue before the Callback Queue if there is no code in the global Execution Context to run (not sure if ‘global’ is popped off the stack or not).

90. Calling a Callback (pushing to the Call Stack) in the Callback or Microtask Queue as well as passing into it the arguments (data) or the results from the background feature is done automatically by JavaScript.

91. The background feature only updates the value of the Promise object. Enqueueing the Callbacks in the ‘onFulfilled’ array to the Microtask Queue is done by or is a feature of the Promise object itself.

92. The ‘value’ property on a Promise object cannot be accessed directly as it is also hidden. It can only be accessed inside the body of the Callbacks in the ‘onFulfilled’ array when JavaScript invokes them and passes the ‘value’ property’s value as their argument.

93. The Callbacks you pass for background features have to be sensitive about how many and what all arguments they will receive when JavaScript invokes them.

94. In the case of Microtask Queue, if you resolve another Promise inside a Callback in the queue, the Callback associated with that resolved Promise is enqueued into the Microtask Queue itself. So there is a possibility that it can recursively fill in the Microtask Queue and the Callback Queue will never be reached by the Event Loop, as it always checks the Microtask Queue before the Callback Queue. This is called starving the Callback Queue.

95. In Node.js, there is caution around using the process.nextTick() method as it enqueues items to the Microtask Queue and can lead to starvation of the Callback

Queue.

96. In Facade Functions that take in a Callback, the Callback is usually enqueued to the Callback Queue (Task Queue) and pushed to the Call Stack upon completion of the background feature. In a Two-Pronged Facade Function that usually returns a Promise object, the Callback(s) added to it using the then() method is enqueued into the Microtask Queue and is invoked upon the update of the Promise object's 'value' property, and before any callbacks in the Callback Queue.

97. The priority of the queues only matters if both queues have callbacks in them at a particular time. If a fetch() background feature takes more time than setTimeout(), the Callback set by setTimeout() is run first since the response of the fetch() call isn't received yet, and hence the Callbacks in the 'onFulfilled' array is not enqueued to the Microtask Queue, therefore making the Microtask Queue empty.

98. A Promise object also has an 'onRejected' hidden property which is an empty array initially. We can populate the array with Callbacks using the catch() method on the Promise object or by passing the Callback as the second argument of the then() method.

99. If an error occurs in the background feature (the Promise is rejected), the Callbacks in the 'onRejection' array are enqueued to the Microtask Queue, and later invoked with the error as their argument. The Callbacks in the 'onFulfilled' array are not enqueued or invoked in this case.

100. The Prototype Chain in JavaScript allows us to emulate OOP.

101. In procedural code, it is not clear as to which data a function applies to, or where that function is, in a large codeset. In OOP, the function is a method on the data itself.

102. Every coding paradigm (like OOP) wants to make your code easy to reason about, easy to add features (new functionality), and be efficient and performant.

103. Encapsulation is when we protect and bundle up data and its functionality in one place (like in an object).

104. Object.create(null) returns an empty object. The argument of create() is not added as direct properties of the returned object.

105. A variable whose value is to be returned from a function call is uninitialized until that function returns a value and the value gets assigned to the variable.

106. The problem with making a function to create, populate, and return objects is that the methods on each object created by that function will take up individual units of memory per method, which will be a loss of space since they will all have the same function definitions. And to add a new feature to the created objects, we have to add it to every object individually, which is a lot of processing.

107. If we want to use a single copy of a method for every instance object, we can create a new Function Store object that has that method. Then when an instance

object looks for that method on it and doesn't find it, the JavaScript Interpreter (which reads JavaScript code line-by-line) looks in the Function Store object, which will have the method on it. For this to work, we need to link the created instance objects to the Function Store object that has all the shared methods.

108. Linking these objects can be done using the Object.create() method.

109. The looking up of methods in the common store (another object) is called the Prototype Chain. We pass in the shared function store to Object.create() when creating an instance object, to link that object to the function store, so that it looks for the method on the function store, if not found within itself.
const user = Object.create(userFunctionStore);

110. The link (or pointer) to the Function Store object is stored in the hidden '__proto__' property, also called Dunder Proto, which is on every object in JavaScript.
Note: Not true but the behavior is similar. See Kyle's course for more details.

111. When JavaScript can't find a property or a method on an object, it goes through the link in its hidden '__proto__' property to the Function Store object and looks for the property there. This is called Prototype Chain in JavaScript. It is a feature in JavaScript like Lexical Scope.

112. We can see '__proto__' in the Developer Console unlike the [[scope]] and 'onFulfilled' hidden properties.

113. The 'this' identifier is an implicit parameter that is automatically passed into all functions (including 'global') in JavaScript (like the 'arguments' pseudo array) when that function is called. The value of 'this' is the object on which the function is called on. That is the object on the LHS of the dot in a method call.

114. Every object in JavaScript has a '__proto__' property. For a normal object, it is the Object.prototype object by default. We are setting (replacing) the link on the '__proto__' property using Object.create(). But the '__proto__' of that object which is passed in as the argument to Object.create() will have its '__proto__' pointing to the default value. So every object has access to the methods on the Object.prototype object because of the prototypal nature of JavaScript. If it doesn't find the method on its '__proto__' object (pointed to), it looks in the '__proto__' object of the '__proto__' object, all the way up to the root.

115. The '__proto__' property of Object.prototype has the value 'null' since it is the root Prototype object of all objects up the chain (there is nothing above it).

116. When declaring and calling a function inside a method definition and later calling that method, the value of 'this' inside the declared function won't point to the object on which that method was called upon. Because the value of the 'this' identifier is received as an implicit parameter by every function and that value is a pointer to the object on which that function was called upon (defaults to the 'window' object if no object was specified). It has nothing to do with Lexical Scope. It is passed into a function at call-time like the 'arguments' pseudo array.

117. We can store the value of ‘this’ inside a variable (usually named ‘that’) in the method’s Local Memory, and the declared function can reference that variable to get the intended value of ‘this’ from the method’s Local Memory using Lexical Scope.

118. We can also explicitly pass the value of ‘this’ using the call(), apply(), or bind() methods which are on every function’s ‘__proto__’ referenced object. The first argument passed into these methods is the value of ‘this’.

119. In the case of Arrow Functions, it doesn’t get passed or declared the implicit ‘this’ parameter (just like with ‘arguments’ and Arrow Functions). So when there is a reference to ‘this’ inside an Arrow Function, it looks up its parent’s scope (Lexical Scope) and gets the value of ‘this’ from there.

120. Arrow Functions help us to avoid making a copy of ‘this’ for reference inside an inner function in a method (see 117), and also to avoid binding the value of ‘this’.

121. When using the ‘new’ keyword before a function, a new object is created and that object is received automatically as the value of ‘this’ implicit parameter by that function. It also returns the newly created object implicitly after the function completes its execution.

122. When creating a function, an object is also attached to it. When using parenthesis with the identifier, it behaves like a function, and when using a dot, it behaves like an object.

123. Every function has a ‘prototype’ property on it, which is an empty object by default.

124. When creating a new instance object using the ‘new’ keyword and a normal function, the ‘__proto__’ of the newly created object is linked to the ‘prototype’ property of the function’s object version by default. We have to specify any shared methods as methods of the function’s ‘prototype’ property.

125. The ‘new’ keyword can be called a modifier because it alters the behavior of a function’s Execution Context. It inserts stuff in there (like ‘this’) automatically for us when used.

126. A function is always a function + object combo.

127. When we call a function with the ‘new’ keyword before it, nothing special is happening. It is just a normal function call. But the ‘new’ keyword does some stuff inside the Execution Context of the function like creating an empty object, passing it in automatically as the implicit ‘this’ parameter, and returning the newly created object after setting its ‘__proto__’ property to reference the function’s ‘prototype’ property and running the function’s body.

128. To identify functions that need to be called with the ‘new’ keyword, we capitalize the first letter of the function’s identifier name as a common practice. It has no impact on the execution of the code.

129. A Class (in JavaScript) is just some syntactic sugar for the above pattern ('new' + function). It lets us define the Constructor (a function that constructs an object) and the shared functions at the same place.

130. A Class just creates a function + object combo. In it, there is a constructor() method which is the function bit of the combo. It is run when the Class is called with 'new' and parenthesis. All the other methods declared in the Class are stored on the 'prototype' property of the object bit of the combo.