1. The statement: “In JavaScript, everything is an object” is not true. Most values in JavaScript behave like an object.

2. The Primitive Types in JavaScript are undefined, boolean, string, symbol, number, and object.

3. Symbols are usually used to create pseudo-private or rather obscure keys on objects in the codebase of frameworks, more than in common code.

4. Some other types or other things that behave as types are: undeclared, null (JavaScript lists it as a primitive type but there are some quirks to it), function (more of a subtype of the object type or callable object), arrays (a subtype of the object type with numerical indexing, the ‘length’ property, and some utility methods on it).

5. BigInt is another primitive type that hasn’t officially been added to the language but will be soon. It is implemented in the V8 engine and is available in the Chrome Browser and Node.js environment.

6. In JavaScript, variables don’t have types, values do. So it may be appropriate to call them value types.

7. When using the ‘typeof’ operator with a variable, it does not return the type of the variable but rather the value that the variable holds. It can be used to determine what we can do with that specific value.

8. When unsetting a regular value like a number, we would use ‘undefined’, and in the case of an object, we would use ‘null’. This is said to be the reason in the specification as to why the ‘typeof’ operator when used with ‘null’ returns ‘object’.

9. BigInt will let us have integer values that are infinitely large (up to the memory limit).
   var x = 10n;
   typeof x; // ‘bigint’

10. In JavaScript, we can use the ‘typeof’ operator on an undeclared (not even uninitialized but undeclared) variable and it’ll still return ‘undefined’.

11. In ES6, block-scoped variables are not set to ‘undefined’ before they are declared. This uninitialized state is also known as the Temporal Dead Zone. The variable cannot be accessed when in this state or it will throw an error (TDZ error).

12. ‘NaN’ is a special (sentinel) value that indicates an invalid number. It doesn’t mean ‘Not a Number’ literally.

13. ‘NaN’ when used with any mathematical operator, always returns ‘NaN’ as the result.
    100 - ‘Hello’; // NaN;
    Here, the string ‘Hello’ is coerced into a number because the minus operator only works on numbers. ‘Hello’ is not a valid number so it gets set to the invalid number value ‘NaN’.

14. ‘NaN’ is the only value that doesn’t have the identity property. It is not equal to itself.
    var invalidNumber = Number(‘Hello’);

invalidNumber === invalidNumber; // false

15. The isNaN() function in JavaScript coerces its argument into a number and then checks whether that value is 'NaN'.
    isNaN('Hello'); // true

16. The Number.isNaN() method in ES6 checks whether the value passed in is 'NaN' and does not coerce the value beforehand.
    Number.isNaN('Hello'); // false

17. The 'typeof' operator when used on 'NaN' returns 'number'. 'NaN' is a number but just an invalid one.

18. Negative zero (-0) can only be checked for equality using the Object.is() method (and also some other operator) in JavaScript. Otherwise, it behaves just as zero (0).
    -0 === 0; // true
    Object.is(-0, 0); // false
    The Object.is() can be considered the more strict version of the triple-equals (===) operator. It can also check the equality of 'NaN' values although Number.isNaN() is semantically more appropriate. The Math.sign() method is also weird with zeroes as it returns -0 instead of -1 when -0 is passed in (the same of 0 as well).

19. Before defining a polyfill, we have to check whether the utility already exists in the current environment so that we don't override it.
    ```
    if (!Object.is) {
      Object.is = function (…) {…};
    }
    ```
    This is the Polyfill Pattern.

20. A way to check if a value is -0 is to divide 1 by that value. If the result is '-Infinity', the value is -0.
    1 / -0 === -Infinity; // true

21. Fundamental Objects are object representations of primitive values with similar behaviors. They are also called Built-In Functions or Native Functions. In them, these have to be instantiated using the 'new' keyword (as constructors): Object(), Array(), Function(), Date(), RegExp(), Error(). Others can be used with 'new' but shouldn't be, because they shouldn't be instantiated as objects (must be used as functions). They are: String(), Number(), and Boolean(). These functions when used as just functions coerces the argument to that respective primitive type.

22. In JavaScript, Type Conversion is also called Coercion.

23. Abstract Operations are conceptual operations that can be seen in the algorithms in the JavaScript specification.

24. ToPrimitive(hint) is an abstract operation that takes in a type hint which can either be 'string' or 'number'. How it works is, if we use a non-primitive like an object in a place where a primitive is required, like math or concatenation, it does the ToPrimitive(hint) abstract operation and gets its primitive value. If the type hint is 'number', it first calls valueOf() on that object and sees if the returned value is a primitive. If not, they call

the toString() method and checks if the return value is primitive and if yes, return it. If the type hint is ‘string’, it calls both the methods in the reverse order, that is toString() and then valueOf(). The two methods are present in every JavaScript object. If both the methods fail to return a primitive value, we get an error.

25. The algorithms in JavaScript are recursive. If a return value isn’t the required value, it does the operations again until it gets the required value or an error.

26. The ‘ToString’ abstract operation gives out the string form of a value. When we do ToString(object), it does ToPrimitive(string). -0 gets coerced to ‘0’ which is a corner case.

27. When using ‘ToString’ on arrays, it serializes the array. It takes of the square brackets and returns the values separated by commas. It doesn’t include ‘undefined’ or ‘null’ inside arrays, and it doesn’t care about the nesting of arrays either.
    [] ; // ‘’
    [1, 2, 3]; // ‘1,2,3’
    [null, undefined]; // ‘,’
    [[[], [], []], []]; // ‘,,,’
    [,,,]; // ‘,,,’

28. When using ‘ToString’ on objects, by default it returns ‘[object Object]’ which is returned by the toString() method on the object. It can be changed by overriding the toString() method on that object. The “Object” in the default return value of toString() is the string tag for all objects and can be changed by returning it from the ‘get [Symbol.toStringTag]()]’ method in ES6.
    Note: ‘get’ binds the property name with a function that will be called when we look up the property name’s value.

29. Anytime we need to do something numeric and we don’t have a number but a value of another type, the ToNumber abstract operation is invoked. ‘-0’ is surprisingly coerced to -0. An empty string (‘’) is coerced to 0 which is a corner case (should’ve been coerced to NaN). ‘null’ also becomes 0 unexpectedly whereas ‘undefined’ becomes NaN.

30. When invoking ‘ToNumber’ on an object (or array), ToPrimitive(number) is called. It first invokes the valueOf() method on the object which just returns the object itself. Then it invokes the toString() method. So every ‘ToNumber’ invocation on an object will return a primitive string, which is coerced later. This leads to this:
    [‘’]; // becomes ‘’ and then 0
    [‘0’]; // becomes ‘0’ and then 0
    [‘-0’]; // becomes ‘-0’ and then -0
    [null]; // becomes ‘’ and then 0
    [undefined]; // becomes ‘’ and then 0
    [1, 2, 3]; // becomes ‘1,2,3’ and then NaN
    [[[]]]; // becomes ‘’ and then 0
    Empty string becoming 0 is the root of all coercion evil, Kyle says.
    If we override the valueOf() method on the object, we can return what we want when ToPrimitive(number) is applied to it since that method is invoked first and the operation will end if a primitive value is obtained.

31. If we use a value that is not of the boolean type in a place that requires a boolean value, the ToBoolean abstract method is invoked. The operation is more a look-up rather than algorithmic. There is a look-up table of falsy values. If the value is one of the falsy values, it returns ‘false’ and otherwise ‘true’. The falsy values are: ‘’, 0, -0, null, NaN, false, undefined. It doesn’t invoke the ‘ToPrimitive’ or anything, it just does the look-up.
    []; // true

32. If any of the operands of a + operator is a string, Operator Overloading occurs and JavaScript performs string concatenation. If the other value is not a string, it is coerced to one. The minus operator however is not overloaded for strings but is only defined for numbers. So if any of the operands is not a number in a subtraction operation, it is coerced to one. It is the same for <, >, <=, and >=.

33. The Unary Plus operator invokes the ‘ToNumber’ abstract operation on any value after it to coerce it to a number.
    +’1’; // 1

34. Boxing is when JavaScript implicitly coerces (in theory) a primitive value into an object so that we can use methods on it as we would on any object.

35. If we create a boolean object using the Boolean fundamental object with the value ‘false’, the object itself if coerced to a boolean will return ‘true’, since ToBoolean only checks if the value is on the falsy list and the object is not on it.
    Boolean(new Boolean(false)); // true

36. The ‘ToNumber’ abstract operation strips of all forms of padding whitespace from a string before doing the coercion. So a string full of whitespace or newline characters is also coerced to 0.

37. If both operands are strings, the less than and greater than operator does an alphanumeric comparison.

38. The Abstract Equality Operator (==) does the Strict Equality Comparison (===) behind the scenes when the types of the two operands are the same. The Strict Equality Operator returns ‘false’ if the types of the two operands are different. If they are the same, it returns ‘false’ if either of them is ‘NaN’, and ‘true’ if the values of both are the same, and also if both are zeroes regardless of its sign.

39. The difference between abstract and strict comparison is that one allows coercion if the types are different and the other doesn’t.

40. When comparing two objects of the same structure, both equality checks return ‘false’. The pointer (or link or address) to the object is checked for equality (which is not the same in this case) and not the object itself because the pointer is what the variable holds.

41. When using the Abstract Equality Operator, if the operands are ‘null’ and ‘undefined’, the expression evaluates to ‘true’. ‘null’ and ‘undefined’ are coercively equal to each other and with nothing else.

42. If the operands are of different types, the Abstract Equality Comparison prefers to apply the ‘ToNumber’ abstract operation on the non-numeric ones. Double equal prefers numeric coercion.

43. If either of the operands is not a primitive value, the Abstract Equality Comparison applies the ‘ToPrimitive’ abstract operation on that operand. Double equal only compares primitives. If both the values are non-primitive values, the types are the same and hence the strict comparison occurs. If after the coercion of a primitive value both the operands are of the same type, the strict comparison is invoked since the algorithms are recursive. It runs the algorithm over and over again until one of the conditions match and compares the two primitives (maybe coerced to) to give us a result.

44. ‘ToPrimitive’ for arrays, when applied stringifies the array. This can make a number coercively equal to that number in an array like this:
    10 == [10]; // true because [10] becomes ‘10’ and then 10.

45. The coercion algorithm has a ‘ToPrimitive’ conversion step to it, only because, in the old days, people did ‘new String()’ to create strings so it had to be coercively equal when doing abstract comparisons.

46. Summary of Abstract Comparison algorithm:
    If the types are the same, do the strict comparison (===).
    If both are ‘null’ and ‘undefined’, return true.
    If there are non-primitives, perform the ‘ToPrimitive’ abstract operation on them.
    If there are only primitives, perform the ‘ToNumber’ abstract operation to all non-numbers.
    This is done recursively until a result is obtained or it errors.

47. The ‘!=’ operator just does the negation of the ‘==’ operator's result.
    x != y; // is the same is !(x == y).

48. We shouldn’t do an explicit comparison of any value with a boolean value since in many cases the coercion will mess it up and almost in all cases the implicit equivalent is better.
    [] == true; // ‘false’ because [] becomes ‘’ which becomes 0, and true becomes 1.
    [] == false; // ‘true’ because [] becomes ‘’ which becomes 0, and false becomes 0.

49. Checking to see if a method is on an object by referencing it, is called Duck Typing.

50. Scenarios to avoid with ‘==’:
    Don’t use it with 0 and ‘’ (or a string with only whitespace in it) on either side.
    Don’t use it with non-primitive values.
    Don’t use it with boolean values on either side (use ‘ToBoolean’ or ‘===’ if necessary).

51. In TypeScript, we are kind of layering on an extra requirement where the variable itself has a type to it rather than just the value.

52. When Inferencing in TypeScript, it guesses that we only mean to store values of a specific type (type of the value at initialization) inside a variable and throws an error

when we store a value of another type.

53. Scope is where you look for things. Specifically identifiers.

54. All variables in a program are either assigned to, or retrieved from. These are the two functions of any variable.

55. JavaScript is not exactly an interpreted language. It is compiled or we can at least say parsed. We can notice this when we get syntax errors before even running a single line of code.

56. There are four steps in compiling. Lexing, Tokenization, Parsing (turns the stream of tokens into an Abstract Syntax Tree), Code Generation (taking the Abstract Tree and producing another form of that program that can be executed).

57. Determining the Lexical Scopes and what identifiers are going to be available in them are done at the time of compilation. Specifically in the parsing phase where the AST is created. The compiler makes a plan for the Lexical Environment at that time.

58. The JavaScript code is parsed into an intermediate representation (similar to bytecode in Java) and that is handed to the JavaScript Virtual Machine which is embedded in the same JavaScript Engine. That intermediate code is interpreted line-by-line and one thing that happens is, every time it enters a new Lexical Scope which was planned in the parsing phase, it creates all the identifiers (or variables) in that scope according to the plan. JavaScript is a Two-Pass System.

59. The scopes can be thought of as colored buckets and the identifiers as colored marbles. The buckets are created when we encounter a function (or enter a block in ES6). Upon encountering an identifier declaration (while compiling), we put a marble of the same color as that bucket, in the bucket.

60. In the compilation phase, there will be a Compiler as well as a Scope Manager.

61. When encountering a formal declaration of a variable, the Compiler asks the Scope Manager (of that scope) to check whether there is a marble (analogy for identifier) of that same color in the current bucket (analogy for current scope). If yes, it doesn’t need to do anything. If no, it includes in the plan to create a new marble with the color of the current bucket and drop it in that bucket. This is the plan that is created in the compilation phase to declare an identifier in a scope.

62. When encountering a formal function declaration, the Compiler asks the Scope Manager to create another colored bucket, since a function will create a new scope. This new bucket will be inside the current bucket and will also be a marble inside the current bucket (with its color). So a function inside a scope is at the same time a colored marble (an identifier in that scope), as well as a new bucket (a new colored bucket inside that bucket). In the compilation phase, once that bucket has been created, it compiles the code inside that function rather than going to the next line after that function declaration. It then creates marbles for any formal declarations in that function, which is of the same color as that function’s bucket.

63. Every scope will have a Scope Manager. So when compiling the code inside a scope, the Compiler asks the Scope Manager of that scope (of that colored bucket) to create and put marbles of that same color, or new colored buckets in it. This is how Shadowing (and Masking) is possible. The Scope Manager of that scope doesn't use marbles of its parent scopes in the event of a formal declaration. It creates its own.

64. When the compiler encounters an identifier reference, it determines what color that marble is and in which bucket to find it. So we will have covered declarations and references in the compilation phase and we now have a plan for our Lexical Environment.

65. When we execute the code, all the 'var's or all the declarations will be gone, as well as all the Function Declarations, since we already made a plan of it all. So essentially, all the identifiers and scopes in our program are determined at compile time. They are created and used at the time of execution, but they are determined at the compilation phase and cannot be changed after that (cannot change the buckets, marbles, or the colors themselves). The decisions about scopes are based on author-time and not run-time.

66. The declarations and Scope Resolution of references are done at compilation time and the assignments and look-up of references (getting the actual value at that particular time) are done at execution time.

67. The Compiler picks up not only the color of the marble (the scope it is in) but also its position. The position can either be the LHS (left-hand side) which is assigned a value, or RHS (right-hand side) which retrieves its value. It can alternatively be referred to as the target (LHS) and the source (RHS).

68. At the time of execution, there is the Scope Manager and the JavaScript Engine (or Virtual Machine).

69. When a variable assignment statement is encountered, the JavaScript Engine asks the Scope Manager if a variable of that name has been declared at compilation time. If yes, the variable (or its location specifically) is fetched and the value is assigned to it. It is essentially a look-up. When it encounters a function call expression, the identifier is in a source position so the JavaScript Engine asks the Scope Manager if that variable has been declared. If yes, its value is pulled out and is executed and the bucket is created with all its marbles according to the Compiler's plan. If we get a value like 'null' or 'undefined' which is not legal to be executed as a function, we get a Type Error which is a run-time error.

70. A Type Error is thrown when we do something with a value that we're not allowed to do with its type.

71. If the JavaScript Engine can't find a marble in the current bucket regardless of its position (target or source), it goes and looks in the parent bucket for the marble. This is what we call Lexical Scope.

72. Function Definitions are also declarations as a whole so they too wouldn't be present in the code to be executed (the compiled code).

73. If an identifier that is being looked up as a target reference isn't found in any scope up to the global scope, the Scope Manager, instead of saying no and throwing an error, declares a variable in the global scope and hands it to the JavaScript Engine (this is possibly planned at compilation time itself). These automatically declared variables in the globals are called Dynamic Auto-Globals. It is not a good idea to automatically create variables like this. It is better to declare your variables. In Strict Mode, this wouldn't be possible. It can only be done in Sloppy Mode.

74. Strict Mode can be turned on by inserting the pragma "use strict", preferably on the top of the program or at the top of a function or a scope.

75. In Strict Mode, if a variable is not found, even in the Global Scope, the JavaScript Engine throws a Reference Error rather than automatically creating one. A Reference Error is when a variable is not found in scope. Tools like Babel and other Transpilers will put the Strict Mode in for you. Inside some of the features like Classes and Modules in ES6, Strict Mode is assumed and it is not necessary to put it in. Some errors are only visible when Strict Mode is ON.

76. Parameters are formal variables inside the scope of a function. They would behave just as a marble of the color of that function's bucket. The assignment of an argument to a parameter happens before the function starts executing.

77. A Reference Error is thrown if a source reference doesn't find a variable and it is not thrown for target references in Non-Strict Mode.

78. Undefined means that a variable exists but does not have a value at the moment. It may have had a value before but currently, it doesn't have a value. Undeclared means that a variable has not been formally declared in the scope that we're looking in. There is no marble for it.

79. The Scope Chain can be conceptualized as a building with multiple floors in it. The ground floor is the scope with the variable reference and the top floor is the Global Scope. All the floors in between are the scopes leading from the current scope to the Global Scope. We look from the ground floor up, looking on each floor, till the top floor where we either will or won't find the variable.

80. Function Expressions (like functions assigned to variables or IIFEs) and their identifier name is not available in the scope that it is defined in. It is only available in its scope. When encountering a Function Expression, a new colored bucket is created and the identifier of that Function Expression is a marble of the same color as that created bucket. That is one of the differences between a Function Declaration and Function Expression. The latter puts their identifier into their scope. The value of that identifier cannot be changed. The identifier of that Function Expression is read-only in its scope.

81. A Function Expression (not a Function Declaration) without a name is an Anonymous Function Expression and if it has a name, it is a Named Function Expression. A Named Function Expression should be preferred because it allows for self-

referencing itself for things like recursion or unbinding an event handler from inside itself. Before this, we used to use ‘arguments.callee’ to get the reference to a function from inside it, but it has been deprecated. Another reason to prefer Named Function Expressions is that it shows the name of that function in the Stack Trace in the event of an error. It also leads to more self-documenting code.

82. The purpose of code is to communicate your intent better and not to improve your convenience to code.

83. Even if you assign an Anonymous Function Expression to a variable, it is still an Anonymous Function Expression that doesn’t have a self-referencable name which is read-only and is less debuggable and self-documenting. It is less debuggable in the case it is used as Callbacks since they do not use the parameter name as that function’s name. Arrow Functions are also Anonymous Function Expressions.

84. Function Declaration > Named Function Expression > Anonymous Function Expression.

85. The sort() function mutates the array that it is called upon while also returning a new array.

86. The Callback received by sort() will get two arguments, which are the two values that are compared at any time. If the first argument should come before the second one after sorting, we return a negative number (-1 usually). If the second one should come before the first, we return a positive number (1). If both are equal, we return 0.

87. Lexical Scope is the idea that a Compiler figures out the scope ahead of time before execution. The name Lexical Scope comes from the name of a stage in Parsing called the Lexer. The Lexical Scope is determined during compile-time. Most languages in existence are Lexically Scoped. Interpreted languages such as Bash Script are Dynamically Scoped.

88. At execution time, there is no look-up process where the JavaScript Engine communicates with the Scope Manager. All the marbles and their colors are determined at the compilation time for declarations and references. The scope of the variable is already known to JavaScript at execution time without looking it up. In some cases where it has to be looked up, it is only looked up once and never again because it now knows the marble color. This is a good optimization.

89. Lexical Scope is determined at author-time (compilation) and Dynamic Scope is determined at run-time (execution).

90. The Principle of Least Exposure also called the Principle of Least Privilege says that we should default to keeping every data private and only expose the minimal necessary. The advantages to this are, it reduces name collisions, it prevents other developers from misusing the data, and it is a protection from future refactoring (we can refactor private data freely compared to public data since it won’t be used by anything else in that codebase rather than inside its scope or Namespace).

91. There are two steps to calling a function. The first is that the value of the function is fetched based on the identifier name. The second step is the parenthesis to execute

it.

```
aFunction();
(aFunction)(); // Both of these have the same effect.
```

If we can put a variable inside the parenthesis (hence making it an expression) and immediately execute it, we can put a Function Expression inside the parenthesis and execute it too. This is called Immediately Invoked Function Expressions (IIFE).

92.
```
(function greet() {
  console.log('Hello!');
})();
```
This is not a Function Declaration because the 'function' keyword is not the first thing in the statement, hence making it a Function Expression. Instead of using the parenthesis to turn it into an expression, we can also use Unary Operators such as +, ^, - , ~, or the lesser-known delete, void, etc.

93. IIFEs can be used to turn statements into expressions if need be.
```
var teacher = (function getGreeting() {
  try {
    return fetchGreeting();
  catch (error) {
    return 'Hello!';
  }
})();
```
Here, the 'try…catch' statement has been turned into an expression.

94. We can use 'let' or 'const' inside a block to make that block a scope. Unless there is a 'let' or 'const' declaration inside a block, it is not a scope. We can use Block Scope instead of IIFE to create a scope in a lot of cases.
```
{
  let greeting = 'Hello';
  console.log(greeting);
}
```
'var' doesn't work here because it attaches itself to the scope of that function instead of the block.

95. Use 'let' to block scope and use 'var' to function scope.

96. Reassigning a 'const' will throw a Type Error.

97. Hoisting is just a dumbed-down explanation for the Two-Pass processing and Lexical Scope mechanism of JavaScript. The identifiers are included in the plan in the parsing phase and they are created when the scope gets created according to plan. JavaScript doesn't magically look through code to find all declarations and move them to the top of the scope's code. Function Expressions are not hoisted since they are not declarations even though the Compiler creates a bucket (scope) for them. In other words, only the target references are hoisted (or created when creating the scope according to the Compiler's plan). Also, Function Expressions cannot be hoisted in its containing scope because it doesn't exist there. It exists in its enclosing scope, where it is hoisted.

98. ‘let’ and ‘const’ also hoists. They also follow the Two-Pass system. But when they hoist, they don’t set the value of the variables to be ‘undefined’ as ‘var’ does when that scope is created. They just create the location for it but do not initialize it to any value. So when that variable is referenced before its ‘let’ or ‘const’ declaration, it throws a TDZ (Temporal Dead Zone) error because it is uninitialized and not ‘undefined’ like with ‘var’.

99. The TDZ exists for ‘const’. It wouldn’t be right for a ‘const’ identifier to have ‘undefined’ as its value before its declaration at execution time, where it gets assigned its constant value. So TDZ was introduced where we cannot access the value of a ‘const’ identifier before its declaration because it is uninitialized. Since it is being done for ‘const’, the Language Designers did it for ‘let’ too.

100. Closure is when a function remembers its Lexical Scope, that is the variables outside itself (free variables), even when it is executed in another Lexical Scope. Even when a function is passed in as a Callback to another function or is returned out from a function, it can hold a reference to the scope that it was defined in. Wherever you may pass that function, it continues to have access to that reference.

101. A function closes over the variables around it and will have access to them regardless of the scope that it is invoked in. Closes over meaning that if a variable in the Lexical Scope of a function but not in the Local Scope of that function itself is used inside it, the function is said to be closed over that variable. But it is possible and more likely that the JavaScript Engine implements Closure as a linkage to the entire scope and not on a per variable basis. Technically it should be able to close over individual variables and remove variables that are not closed over. But currently, it’s on a per scope basis.

102. Closure is very useful and close to being a requirement of any programming language that has Lexical Scope and First Class Functions (functions that can be passed around like variables).

103. We have access to the actual variable and not a snapshot of it when a function has closed over it. Wherever that function goes, we have access to the original variable. We do not close over values in Closure. We close over variables and whenever we invoke that function, we can access the value in that variable at that particular moment because we have preserved a linkage to that variable (or to the scope that it is in).

```
for (var index = 0; index <= 3; index++) {
  setTimeout(function () {
    console.log(index);
  }, index * 1000);
}
// 4
// 4
// 4
```

Here there is only one ‘index’ variable. So the latest value of it is logged out because the Callback closed over the variable, and the value of that variable when that function was run was, in this case, 4.

104. If we use ‘let’ instead of ‘var’ on the ‘for’ loop (for, for…of, and for…in), a new variable is created on each iteration in the block scope of that ‘for’ loop. In that case, the Callback will close over a new variable on every iteration with a value that doesn’t change, and hence it will print:

```
// 1
// 2
// 3
```

JavaScript creates a new variable on each iteration and assigns it the value from the previous iteration (and also increments it after that in this case). If we use ‘const’ it throws an error as it tries to increment that variable after assigning, which is not legal for a constant.

105. Storing a set of functions and some data that they operate on in a single logical unit like an object can be called the Namespace Pattern (not official). It is effectively collecting that data and behaviors (functions) into a Namespace.

106. The Module Pattern is similar in concept to the Namespace Pattern but it introduces the idea of hiding or having control over the visibility of its data and behavior. It requires Encapsulation. A Module will have things that are public which is its public API and there will be private things that are not available for access.

107. Modules are implemented using Closures. The classic Module Pattern called the Revealing Module Pattern encapsulates its data and behavior together. The state of a module is held by its methods using Closure. We run an IIFE with the private state inside it, and we expose the methods that close over those private data. This way, the private data is not directly exposed to the public but can still be made use of inside the method that is being exposed, by the power of Closure. The IIFE that is run to instantiate a module is a Singleton (as in it runs only once and never again) but its scope is still available to the public API.

```
var workshop = (function Workshop(teacher) {
  var public = {
    ask,
  };
  return public;
  function ask(question) {
    console.log(`${teacher}, ${question}`);
  }
})(‘Kyle’);
workshop.ask(‘What is Closure?’);
// Kyle, What is Closure?
```

108. When a function that returns another function completes running, its scope is not garbage collected if the returned function has closed over variables in that scope. The scope persists and the returned function will have a linkage to that scope.

109. A Module is designed to change its state over time. If it doesn’t, then it’s not a Module. It is an over-engineered Namespace. We need Closure for a Module so that we can hide the state (that can change over time) from the public API of that Module. The Principle of Least Exposure/Privilege can also be observed here.

110. Instead of using a Singleton IIFE to instantiate a Module, we can create a Factory Function that can be called multiple times to instantiate Modules over and over again. We can just make the IIFE a normal Function Declaration and use it as a Factory Function. All instances created using the Factory Function will have their state which won't have any relation to each other's state.

111. To use ES6 Modules in Node.js, which doesn't have first-class support for them, we have to use the '.mjs' file extension instead of '.js'.

112. The new ES6 Module syntax works just as a normal Module would conceptually. You create a file and declare private variables and functions in it. You export what should be available to the public using the 'export' keyword. Everything that is exported is public and everything that is not exported is private.

113. ES6 Modules are file-based. You cannot have multiple Modules in a single file. A file can only contain one Module. The file essentially acts as a big function where the private identifiers are closed over by the public (exported) identifiers.

114. ES6 Modules are also Singletons. No matter how many times we import a Module, it only runs once. We get a reference to the same instance that was created on the first import on every subsequent import. We'd have to expose a Factory Function in our API ourselves for users to instantiate the Module over and over again.

115. We can import in two ways in a file. The first is the Named Import Syntax where we assign the reference (just the reference to that export in the Module) of the default export to an identifier. The name of the default export is 'default' itself. We can import individual methods from the Module this way.
import ask from 'workshop.mjs';
The above line imports the default export of 'workshop.mjs' with the identifier name 'ask' in our scope.
The second style is the Namespace Import where the identifier (just a variable) to which all the exports of the Module are imported, acts as a Namespace for that exports.
import * as workshop from 'workshop.mjs';

116. Modules exposed as UMD (Universal Module Definition) Style Modules interoperates with Browsers, Module Loaders, and Node.js.

117. The 'this' keyword inside a function references the Execution Context for that call, which is determined entirely by how the function was called. You cannot look at the definition of a function to figure out what its 'this' reference is going to be. The only thing that matters is, how that function gets invoked. 'this' can have a different reference every time it's called, which makes it more flexible and reusable.

118. We can pass in the value of the 'this' keyword to a function by using the 'call' method on the function's Prototype. Hence, we can change the value of 'this' to dynamically change the values used in that function (similar to what we can achieve from Dynamic Scope). The 'this' keyword exists so that we can invoke these functions in different contexts.

119. There are four different ways in which a function can be invoked, and the way it is invoked determines its ‘this’ reference. The ‘this’ chain also can be visualized as a building with a lot of floors, same as with Lexical Scope. But which building it is, is determined based on the way the function is invoked. The place where a function is called is its Call Site (not where it is referenced but called).

120. The first way a function can be called is by Implicit Binding. This is when we store a method in an object (Namespace) and call that method using the dot operator. This tells JavaScript to call that function with its ‘this’ keyword pointing at that object. This is the same for all (or most) OOP languages. We can store the same function in two different objects, and when it is called, the ‘this’ keyword will be pointed to the object that it was called on, and the values will change accordingly.

121. The second way a function can be called is by Explicit Binding. This is when we call a function using the call() or apply() methods on it and by specifying the value of ‘this’ explicitly as its argument. A variation of Explicit Binding is called Hard Binding.
setTimeout(workshop.ask, 1000);
In the above case, there is no Call Site in that line. The function is just being passed as an argument. The function will be called by Explicit Binding inside the definition of setTimeout() where the ‘this’ reference is set to the Global Context. Hence, the value of ‘this’ inside of ask() will not be ‘workshop’ but rather the Global Context. We can use Hard Binding to solve this by using the bind() method on the function and passing in the context explicitly. It says that no matter how the function is invoked, it should always use the context that we passed into the bind() method. The bind() method returns a new function that is bound to the context that we’ve specified. It takes away the flexibility of ‘this’.

122. Lexical Scope is predictable whereas ‘this’ binding is flexible.

123. Rather than Hard Binding a this-aware method on an object to a context because we have to pass it around, it is better to create a Module and make use of Closure because the ‘this’ keyword was introduced for its flexibility and if we’re making it predictable, it’s better to use something that was made with predictability in mind (like the Lexical Scope).

124. The third way of invoking a function is by prefixing the ‘new’ keyword. It says that the ‘this’ reference of that function should be set to a new empty object. The new keyword has nothing specific to do with Classes but is syntactically made to look like that. You can effectively do the same thing using Explicit Binding.
var instance = createInstance.call({});
But the ‘new’ keyword also does more than just calling the function with its ‘this’ reference pointing to a new empty object.

125. The four things that the ‘new’ keyword does every time are, it creates a new empty object, it links that empty object to another object (see Prototype which comes later), it invokes the function after it and points its ‘this’ reference to that newly created object, and if the function doesn’t return an object, it implicitly returns the value of ‘this’. The ‘new’ keyword is hijacking the function to do all these and it doesn’t matter even if it’s an empty function, all these are done because it was

prefixed with ‘new’.

126. The fourth way of invoking a function is by Default Binding. It is the normal way of invoking a function. It means that if the value of ‘this’ has not been bound implicitly, explicitly, or using the ‘new’ keyword, it falls back to using the Global Context as the value of this in Non-Strict Mode, and ‘undefined’ if in Strict Mode. In Strict Mode, if a function is not invoked with any specific ‘this’ binding, the default behavior is to set ‘this’ as ‘undefined’.

127. It is ‘undefined’ in Strict Mode because it is not a good idea to define a this-aware function and invoke it without specifying a value for ‘this’. It is almost bad as automatically creating globals.

128. These are the only four ways available to invoke a function: Implicit Binding, Explicit Binding (+ Hard Binding), ‘new’ keyword, and Default Binding. You cannot figure this out by looking at the Function Definition but at the Call Site.

129. If a function is invoked by more than one of these ways:
```
new (workshop.ask.bind(workshop))();
```
The precedence goes like this: ‘new’ Keyword > Explicit Binding > Implicit Binding > Default Binding.
Note: bind() uses apply() internally (so it is Explicit Binding).

130. The ‘this’ keyword in an Arrow Function is not bound to its parent or anything. An Arrow Function just does not define a ‘this’ keyword in its scope at all, no matter how the function is invoked. This means that if we use ‘this’ inside an Arrow Function, it behaves as if ‘this’ is any other variable. It goes up the Lexical Scope and resolves its value.
```
var workshop = {
  teacher: ‘Kyle’,
  ask() {
    setTimeout(() => console.log(this.teacher + ‘?’), 1000);
  }
};
workshop.ask(); // “Kyle?”
```
In the above code, the ‘this’ keyword of the ask() method is set to ‘workshop’ at the Call Site of that function which is the last line, by Implicit Binding. An Arrow Function is passed into setTimeout() which closes over the ‘this’ identifier in the ask() method since in Arrow Functions, the ‘this’ keyword is not declared.
In the specification it says:
An Arrow Function does not define local bindings for ‘arguments’, ‘super’, ‘this’, or new.target.

131. When you prefix ‘new’ on an Arrow Function, you get an Exception.

132. An object is not a scope. If we define a method inside an object as an Arrow Function, the ‘this’ keyword does not resolve to that object. Because Arrow Functions resolve the value of ‘this’ lexically and objects are not Lexical Scope. It is the scope in which the object was defined in where the Arrow Function looks for ‘this’.
Note: The ‘this’ keyword in the Global Scope (can be considered as the Global

Function) resolves to the Global Object (‘window’ in Browser).

133. Without Lexical This (Arrow Functions), we would have to use hacks such as:
var self = this; // Not good.
, or
var that = this;
, or even better,
var context = this;
We store the value of ‘this’ in a variable inside the method and we use that variable inside the function that is returned or is used as a Callback so that the function closes over that variable and gets the intended value of ‘this’. Or we would have to Hard Bind the function using bind(). Arrow Functions are better in these scenarios.

134. Properties on an object aren’t scoped. They are just members of an object value. They don’t participate in Lexical Scope at all.

135. Classes don’t have to be statements but they can also be expressions and even Anonymous Expressions. They can have a constructor() method, as well as other methods on it, which doesn’t have to be separated by commas. We can also extend another Class using the ‘extends’ keyword where all methods of the Parent Class will be inherited (at least conceptually) and be available to the Child Class.

136. A Class will also have a ‘super’ keyword in it. It allows us to do Relative Polymorphism. You can refer to an instance of the Parent Class using the ‘super’ keyword and call its methods. When called as a function, super() invokes the constructor() of the Parent Class. This is particularly useful when a Child Class shadows a method in its Parent Class. The method in the Child Class will be used by its instances, but inside of the Child Class, we can use the method with the same name in the Parent Class using ‘super’. Before ES6 Classes, there was no ‘super’ keyword or any equivalent, hence this was not possible. So Classes are not just Syntactic Sugar but have many features of their own and many to come in the future.

137. Even though there are syntactic changes, the way the functions are called and the rules for ‘this’ binding are the same for Classes as well. The methods inside Classes work like any other function and are not hard-bound. To make it hard-bound, there is a pattern where the methods are declared inside the constructor() as Arrow Functions instead of being in the Prototype (which is not good). The very idea of a Class System is to make use of the Prototype feature and dynamic ‘this’. It’s better to use Modules if we want Lexical This in most places.

138. If extending a Parent Class, we have to call super() in the Child Class’s constructor() in the first line of its definition. If we don’t define a constructor() for the Child Class, it is called automatically.

139. Objects are created using Constructor calls in JavaScript. This is when we use ‘new’ on a function call expression. A new object is constructed, used as the ‘this’ binding of that function, and returned.

140. In most languages, when creating an Instance from a Class, everything that was inherited is flattened and copied to the Instance as it is much easier than looking up the method in the Class tree up to the top. Traditionally, Class is a copy

mechanism where when once copied, the Class and its Instance have no binding to each other (at least in our mental model). The Class hierarchy chain is maintained only when there is Relative Polymorphism and we need to access the method of a Parent Class. It is called a Virtual Table (C++).

141. JavaScript doesn’t copy methods like this. The Constructor call constructs an object which is linked to its Prototype. The Prototype is not copied to the Instance but the Instance is linked to it. We define methods on the ‘prototype’ property of the Constructor (function) and when we call the Constructor (call with ‘new’), the constructed object is linked to the ‘prototype’ property.

142. There is an ‘Object’ function (which also serves as a Namespace for a lot of important methods like keys(), values(), etc.) in JavaScript and it has a ‘prototype’ property on it. All objects are descended from this ‘prototype’ property. It contains a lot of utility methods such as valueOf(), toString(), etc. That ‘prototype’ property has a reference back to the ‘Object’ function called constructor() on it (which is a method). The ‘Object’ function is not its Constructor but it is referenced by the constructor() method of its Prototype to give a false sense of Class systems being native to JavaScript.

143. Every function when created will create with it a ‘prototype’ property that has a reference back to that function on its constructor() method. When we use the ‘new’ keyword when calling that function, it constructs a new object, it links that object to the ‘prototype’ property of that function, it invokes that function with its ‘this’ referencing the newly created object, and it returns the value of ‘this’. When we look up a property or method on that newly created object, and if it doesn’t have that property on it, that property is looked up in the ‘prototype’ property of the Constructor via the linkage between the object and the Constructor’s Prototype. This is called the Prototype Chain or [[proto]] (in the specification) of that object.

144. The ‘this’ keyword will refer to the object in the Call Site regardless of whether the method was found on that object or its Constructor’s Prototype. It follows the rules of Implicit Binding. We can essentially share a single method with an infinite number of objects by making use of the dynamic ‘this’ system and the Prototype linkage.

145. If we look up the constructor() method on an Instance, we can see that it references the function that constructed that object. This is not because that is the case (the ‘new’ keyword constructed it), but because the Instance does not have a constructor() method on it, so it looks up to the ‘prototype’ property on the Constructor via the linkage and finds constructor() there, which was created when we defined that function, just for this effect.

146. When we look up the Dunder Proto (__proto__) property of an object, it returns the Prototype of that object’s Constructor. But there is no Dunder Proto property on that object. When we look it up, it goes up the Prototype Chain and looks in the Prototype of its Constructor via the linkage, but still won’t be able to find a Dunder Proto there. So it goes further up to the Prototype property of the Constructor’s Prototype’s Constructor, which is ‘Object.prototype’, where we’ll find the Dunder Proto property, which is a Getter. We call that Getter with the object (the first one) as its context due to Implicit Binding and get the hidden internal linkage

([[proto]]) in return, which in this case returns the Prototype of its Constructor.
instance.__proto__ === Class.prototype; // true;
Class.prototype.__proto__ === Object.prototype; // true;
Note: Functions have a ‘prototype’ property (not objects) and objects can resolve the ‘__proto__’ property using the Prototype Linkage (Chain) and the behavior of ‘this’.

147. We cannot override the binding of a function that has been Explicitly Bound using the bind() method unless by prefixing the function call with the ‘new’ keyword, which will override the binding and use the newly created object as the context due to the precedence rules.

148. We can change or rewire the Prototype Chain of an object by using the Dunder Proto as a Setter (changing the value of the ‘__proto__’ property) as of ES6. We can change the linkage to another object from its Constructor’s Prototype this way if we want. We can also use Object.setPrototypeOf() to do this same thing in a much more formal way.

149. Arrow Functions do not have a ‘prototype’ property on them. This is the reason it throws an Exception when we call it with the ‘new’ keyword. There is no object to link the instance to.

150. The ‘super’ keyword is statically bound the first time it is created, and will not change even if the linkage is changed (Prototype Chain is rewired). ‘super’ can be used inside of methods in an object literal as well. This only works on Concise Methods (methods defined without using colons).

151. If we override a method on the Prototype Chain of an object by defining a method with the same name on that object, that is called Shadowing Prototypes. We cannot access the method on the Prototype Chain using the ‘this’ keyword since it has been shadowed.
Note: On an event of Infinite Recursion, we run out of space to insert stacks into the Call Stack I guess. Hence it throws an Exception.

152. If we want to access a shadowed method of an object, we can access it using the Dunder Proto, which will give us the Prototype Chain of the current object, where that method is. We can do this up to any level in the Prototype Chain by chaining Dunder Protos. But it will be called with that Prototype object as its context, so we have to bind the value of ‘this’ explicitly. This can be referred to as Explicit Pseudo Polymorphism. It is not real Relative Polymorphism (which is achieved using the ‘super’ keyword). We cannot shadow methods without using the Class syntax and still achieve Relative Polymorphism or we have to do Explicit Pseudo Polymorphism.

```
instance.ask = function () {
  this.__proto__.ask.call(this);
};
```

153. Why shadow methods? Because it is the purpose of Relative Polymorphism. So that we can override methods on the Prototype Chain but call the methods on the Prototype Chain using the ‘super’ keyword, and hence extending the functionality of the Child Class’s methods.

154. To achieve Prototypal Inheritance without using the Class syntax, we have to do this:

```
function ParentClass(value) {
  this.value = value;
}
function ChildClass(value) {
  ParentClass.call(this, value);
}
ChildClass.prototype = Object.create(ParentClass.prototype); // The ‘extends’ clause.
```

Essentially, we linked the Prototype of a Constructor to the Prototype of another Constructor, which it should extend (or inherit conceptually) from. Object.create() does two things. It first creates an empty object and then links it to another object as its Prototype Chain (the first two steps of the ‘new’ keyword’s algorithm). So here, we link the ‘prototype’ property of the Child Class to the Prototype of the Parent Class. Now, ‘ChildClass.prototype’ is Prototype Linked to ‘ParentClass.Prototype’. It is the same as doing:

```
ChildClass.prototype.__proto__ = Workshop.prototype;
```

Now when we do:

```
const instance = new ChildClass();
// ‘instance’ is Prototype Linked to ‘ChildClass.prototype’ and that is linked to
// ‘ParentClass.prototype’.
```

155. Even if a method is not found on an object but its Prototype Chain, it is stilled called with that object as its context because of Implicit Binding rules. As said in the previous course, it can be seen as the dot operator passing the object on its left as an argument, which is assigned to the Implicit Parameter ‘this’.

156. To summarize, in OOP systems of most traditional languages, when we make a Class and then instantiate it, it is conceptually and sometimes physically copying from that Class to the Instance. When you create a Child Class that extends from that Class, we are copying things to the Child Class from it. When we instantiate the Child Class, there is more copying. In the case of Prototypes, the Instances and Child Classes are linked to the Parent Class. This is referred to as Prototypal Inheritance. But it is not an inheritance in practice, but a linkage.

157. The Prototype system in JavaScript is called the Behavior Delegation system and not an Inheritance or Class system. They tried to achieve the Class system using this Behavior Delegation system. The Prototype system is more powerful since a Class system can be implemented in a Prototypal language but not vice versa.

158. JavaScript and Lua are Object-Oriented Languages whereas others can call themselves Class-Oriented Languages since we can create objects in both these languages without using Classes.

159. Objects Linked to Other Objects (OLOO) is a Design Pattern made by Kyle for JavaScript. It doesn’t use Classes, Constructors, the ‘new’ keyword, or Prototypes (directly). We first create an object with some methods. To create an instance of it, we create another object using Object.create() and pass in the first object, so that the instance will have a Prototype Linkage to the first object. Hence we can use the methods in the first object using the Prototype Chain and Implicit Binding. To create another object like a Child Class, we can use Object.assign() and Object.create() to

link it to the parent object and create its methods on it. It can be instantiated using Object.create() as well.

```
var ParentObject = {
  getValue() {
    return this.value;
  }
};
var ChildObject = Object.assign(
  Object.create(ParentObject),
  {
    logValue() {
      console.log(this.getValue());
    }
  }
}
var instance = Object.create(ChildObject);
instance.value = 1;
instance.logValue(); // 1
```

160. The Object.create() method just creates a new empty function, sets its ‘prototype’ property to the object that was passed in, and then invokes that function as a Constructor (with ‘new’) and returns the constructed object. We essentially construct an object linked to another object that we passed in, using a Constructor, its Prototype, and the ‘new’ keyword.

```
Object.create = function create(prototype) {
  function constructor() {};
  constructor.prototype = prototype;
  return new constructor();
};
```

161. Composition Through Inheritance means that we solve problems using Inheritance and the Class system. We create Classes that inherit from the previous one like in a waterfall structure and create an instance of the latest Class and it will have access to all the methods of every Class before it on a single Instance.
ClassA > ClassB > instance

162. This pattern changed to Composition Over Inheritance which is keeping the Classes separate instead of chaining them as Child Classes, but instantiating one Class to a property on the instance of the other Class.
ClassA > instanceA
ClassB > instanceA.instanceB

163. This pattern too changed to Mixin Compositions as both the Classes are instantiated individually and everything from one instance is copied over to the other.
ClassA > instanceA
ClassB > instanceB
Copy all methods from ‘instanceB’ to ‘instanceA’.

164. With the Behavior Delegation pattern, we stop thinking about in terms of Parent and Child and start thinking about Peer-to-Peer. We can have ‘ObjectA’ linked to ‘ObjectB’ or vice versa. We can just link one object to the other, which keeps those

objects separate but lets us use the methods on one object on the other via that link, but with the context of the calling object. We could say that methods are delegated to the other object via the link. This is a two-way link because the context is the object on which the method is called on. We have access to all methods on the other object, and any of that methods will have access to the calling object by the ‘this’ reference. Both objects can talk and share at the time of the method call.

165. How it works is, let's say we have ‘ObjectA’ and ‘ObjectB’ where ‘ObjectB’ is Prototype Linked to ‘ObjectA’. When we call a method on ‘ObjectB’ and it doesn’t find that method on it, it delegates it to ‘ObjectA’ where that method is found. Now the ‘this’ keyword inside of that method in ‘ObjectA’ points to ‘ObjectB’ because of Implicit Binding. So inside that method, we can access all the methods of ‘ObjectB’. We can also access all the methods in ‘ObjectA’ because ‘this’ (‘ObjectB’) has a link to ‘ObjectA’ on it. Hence we can access the methods of both the objects inside that method, making it a Shared Context (Context shared between two objects).