1. Some functional utility methods are: each(), map(), reduce(), etc.

2. Functional Programming is about breaking your code into verbs whereas OOP is about breaking it into nouns.

3. Functional Programming is a style where we pass and return functions to and from functions, do things with its arguments/parameters, etc.

4. Functional Reactive Programming is a functional programming style with Observable Streams.

5. Pure Functions are functions without any side effects and are easier to test. It just takes in some input and gives back an output.

6. each(), map(), etc. will help to prevent off-by-one index errors in loops.

7. Anything that uses a dot (.) operator in JavaScript is an object.

8. If it's not an assignment, it's a lookup.

9. When assigning a primitive value to a variable, the value is stored directly in the memory with the variable name as its label (store by value).

10. When assigning a value to a property of an object, the object by itself has only the property names, and the property names hold references to the memory where its value is stored (store by reference).

11. When reassigning a property of an object, only the reference held by the property is updated. The memory with the old value is garbage collected, and the new value is stored in another place in memory, which is then pointed to by the object's property name.

12. When passing primitive values (stored by value) to functions, they are stored in a new place in memory every time. For reference values (stored by reference), the memory is shared instead of creating a new one. You're passing a pointer to the object and not the object itself. Since the value of the pointer is the same for every function that uses it, all functions that use that pointer value is working on the same object.

13. Arrays are objects. They just have some additions (like push(), pop(), etc.) but fundamentally they're just objects. We can add properties and methods to them as well.

14. In an array, the index is just a property of the array object, which is stringified upon use in square brackets.
    array[0] === array['0']; // true

15. We use bracket notation instead of dot notation (especially on arrays) when the property name is not a valid variable name. For arrays, we put the index number in square brackets instead of using the dot operator, simply because a number is not a

valid variable name.

16. The property/method name of an object is always a string. The bracket notation evaluates the expression inside it and coerces it into a string, and the dot notation does the same with the provided field name without the evaluation part.

17. The length of an array depends on the highest numerical property name (index). Length = Highest Index + 1;

18. If using invalid variable names as property names inside an object literal, it has to be put inside quotes (as a string), just as it would be in its bracket notation.

19. The length of an array is determined by the numerical indices with values added to it. Numerical indices are special to an array when to compared to normal properties. Normal properties are not counted for array length.

20. We use dots instead of square brackets just to reduce the number of characters written.

21. Destructuring means creating variables and taking the values for them out of an object or an array.

22. When destructuring, there will be a target and a source.
    const [first] /* Target */ = [1, 2, 3] /* Source */;

23. Assignments can also be done with destructuring rather than just initialization.
    let first;
    [first] = [1, 2, 3];

24. Array destructuring happens in order of the elements of an array. The first element is stored in the first destructured variable.

25. If there is no value to destructure, undefined is assigned to the variable.

26. We cannot reassign a ‘const’ variable’s value. But we can reassign the properties of an object stored inside a ‘const’ variable. Because we are not changing the pointer in the variable but a property on it.

27. The freeze(object) method on the ‘Object’ function will freeze the passed object (the original object itself) one layer deep so that its properties cannot be reassigned. It doesn’t throw an error but prevents reassignment.

28. In the case of destructuring objects, the variable is assigned the value of the property with the same name inside the object and not the pointer that the property stores (unless it’s not a primitive value).

29. We can destructure an element at a specific index of an array by using object destructuring on the array since an array is fundamentally just an object.
    const { ‘1’: second } = [1, 2, 3];
    second; // 2

30. You can also use bracket notation when object destructuring objects or arrays, which will let you evaluate an expression as the property name to destructure.
    ```
    const { [1 + 1]: third } = [1, 2, 3];
    third; // 3
    ```

31. When it comes to arrays, we can push a value or assign a value to any position by specifying its index.

32. A ‘for…in’ loop is used to loop over the properties of an object.

33. The variable itself which holds an object contains only a pointer to that object in memory. That pointer is what is passed around.

34. Turning some data into a data structure could be called Hydration.

35. The _.each() function takes an array (iteratee) and a callback function (iterator function) which will be called with each element in the passed array.

36. The each() function is just an abstract looping mechanism for arrays. It helps in preventing errors (off-by-one index errors) and is more readable in general.

37. The iterator function receives the current iterated element, the index, and the entire data structure as its arguments.

38. The data structure itself is passed into the iterator function in case if the data structure passed into each() function is an array/object literal.
    ```
    _.each([1, 2, 3], function (number, index, list) {
      /* ‘list’ is only accessible here since it is not stored anywhere else. */
    });
    ```

39. The _.map() function is used to transform an array into another array since it always returns a new array.

40. If you don’t want to set up a project to test a library, you can go to the library’s website and work on the console window, since there’s a chance that the library will be loaded there.

41. The _.filter() function is used to collect and return only a set of values from the array that it is used upon, depending on the boolean value returned by the callback function that has to be passed to it. The element is included in the returned array if the callback’s return value is true (or truthy).

42. These kinds of functions (like filter()) are also referred to as truth tests.

43. You can omit the curly braces for control statements if it only contains a single statement inside it. The number of lines doesn’t matter.

44. The anatomy of a function is as follows:
    ```
    var add /* Function Name */ = function (a, b /* Parameters */) { /*
    Declaration/Definition */
      return a + b; /* Function Body */
    ```

```
};
add(1, 2, 3); /* Invocation/Call Time */
```

45. Parameters are variables that have no value until the function gets called with arguments, which are the values for the parameters.

46. Logging to the console or changing variables in shared or parent scopes are also side effects.

47. An arrow function automatically binds the context (value of ‘this’) to its parent context. Note: Not true but the behavior is similar. See Kyle’s course for more details.

48. All normal functions have a ‘this’ keyword that gets bound at call-time. But arrow functions do not have that. They reach up to their parent scope and grabs the value of ‘this’ in that scope.

49. Arrow functions replace the need for bind() as well as storing ‘this’ in a variable inside a function.

50. The ‘arguments’ keyword contains all the arguments passed into the function in an object-like array in the case of a regular function. It is not there in arrow functions.

51. Projecting is taking a value out of a data structure and turning it into another data structure.

52. A tuple is an array of arrays with only two elements in each array.

53. The rest operator gathers all the extra arguments and puts them in an array when used in a function’s parameter list.

54. The ‘arguments’ identifier available in regular functions is just a pseudo array, which is just an array-like object that doesn’t have the handy array methods and contains the arguments passed into the function as its elements.

55. To call an array method on the ‘arguments’ identifier (or any pseudo array), we have to use the call() method.
    ```
    Array.prototype.slice.call(arguments, 0, 10);
    ```

56. The ‘arguments’ pseudo array only cares about and contains the arguments passed into the function at call-time. It won’t differ if the parameter list contains a rest operator or default parameters.

57. The OR (||) operator evaluates to the first truthy operand from left to right in an expression. The AND (&&) operator evaluates to the last truthy operand in a sequence of truthy operands from left to right. They are coerced to Boolean when in an ‘if’ statement’s condition.

58. To convert a pseudo array into an array, we can use the call() method with the slice() method.
    ```
    const array = Array.prototype.slice.call(arguments);
    ```
    This calls the slice() method in the Prototype of the Array Constructor, but sets the

‘arguments’ pseudo array as the context (this) by passing it into call(). It is the same as doing arguments.slice(), if ‘arguments’ was a regular array. The call() method is a prototype method on every function, used to set its context (this) when calling.

59. The from() static method on the ‘Array’ class was introduced in ES6 to turn a pseudo array into a regular array. It has more features such as taking iterables and giving it access to array methods.

60. Array-like objects may even have a length but not methods like push().

61. The ‘debugger’ keyword when used will stop the execution at that line and open the debugging tools of your environment, where you can inspect the current state of the program and a lot more things.

62. Functions are also objects. So you can add properties and prototype properties to them.

63. Scope is just an area where a variable has access to some value.

64. A global variable is available to the whole codebase. We specify it by omitting the ‘var’ keyword (not a good idea) or attaching it as a property of the ‘window’ object.

65. Local variables are variables inside functions or blocks.

66. In testing, the describe() function takes two arguments: a description for the block, and a callback with all the tests.

67. The beforeEach() function takes a callback which is called before calling each it() function.

68. Each it() function call is a test in which the expect() function should pass for the test to pass. it() takes in a description for the test and a callback with the expect() call statements. expect() takes in a condition.

69. In JavaScript, there is a function-based scope system where a variable can find the value it has in the function that it is in, or in its parent scope, both of which when combined become its scope.

70. You can only look up in scope up to the root object (window).

71. When a variable with the same name exists in the outer scope, it is known as masking (the variable in the inner scope masks the one in the outer scope). And if a variable with the same name exists in an inner function’s scope, it is called shadowing.

72. A variable is available to its scope and its inner scopes and never the other way around.

73. ‘let’ and ‘const’ keywords create block scopes.

74. In Chrome Developer Tools, we can see variables getting garbage collected as it won’t exist if not referenced in any future code when debugging.

75. The scope is determined based on the hierarchy of the code and not its execution. Where a function is called makes no difference to its scope, which is determined by where the function is declared. It is called Lexical Scope. It is as you would read it, and not in the order in which it is executed.

76. The variables in outer scopes are only referenced and hence the latest values are obtained always. The order of writing or declaring doesn’t factor in, and the execution order is what determines the value of a variable at any specific time.

77. A variable masking another variable with the same name in an outer scope means we won’t have access to the variable in the outer scope, since another one will be found in the current scope itself.

78. Every function call creates a brand new function scope (execution context actually).

79. Once the function finishes running, the created scope is thrown away.

80. JavaScript throws an error when redeclaring a variable in the same scope only if you use the ‘let’ keyword and not the ‘var’. The declaration part is ignored if ‘var’ is used in the assignment of an already declared variable created using the ‘var’ keyword.

81. Number.isNaN(1 + undefined); // true
    isNaN() checks whether the argument is or can be converted into a number.
    Number.isNaN() checks if the value is ‘NaN’.

82. The scope of an already run function is not thrown away if some other inner function references a variable that is in that scope in another context. The scope is persisted for that variable reference in the inner function.

83. A function exits to or resumes execution from where it was called, and not from where it was declared.

84. When declaring a variable with ‘var’, it gets hoisted to the top of the local scope with a value of ‘undefined’.

85. Hoisting is something that happens when your code is being compiled and optimized.

86. Higher-Order Functions can take in functions as their arguments as well as return functions from them.

87. Callbacks are just functions we pass into other functions as arguments.

88. If using the spread operator with an array inside the argument list of a function, it spreads the elements in the array and passes them as individual arguments to the function.

89. The apply() method is there so that we can call functions with the elements in the ‘arguments’ pseudo array of another function. The context and an array of arguments

can be passed into the apply() method of a function.

90. We can also do [].slice.call(arguments, 1, 2, 3) instead of Array.prototype.slice.call(arguments, 1, 2, 3).

91. The _.reduce() function takes in an object, an iteratee, and an accumulator. The iterate (callback) function receives the initial value of the accumulator in its first invocation, and then the value returned from that function in every subsequent call. The final returned value of the iteratee is returned from the function call. It's called reduce because it reduces a list to a single value.
    Iteratee signature: (accumulator, value, key, object) => Next Accumulator

92. When only assigning to a variable (not initializing), the value of the variable is returned from that statement. When initializing, 'undefined' is returned.

93. The _.eachRight() function is the same as _.each() but we go from right to left of the array on each iteration.

94. Currying is when you can break a function into multiple function calls that make up the whole argument list of the original function.
    add(1)(2)(3)(4); // 10
    Each function call except the last one returns another function.

95. Composing is just combining multiple functions to create a new function. The values passed in as arguments to the composed function's call, are passed into the last function in the argument list of the compose function call, and the return value of that function is passed into the function before it, and so on.

96. A curried function just accumulates all the arguments passed into it and returns a function every time until all the arguments required by the original function have been passed in. Once all the arguments have been received, it executes the stored function.

97. An asynchronous function does not block the main thread and is called at the end of the main thread.

98. A closure is when a function returns another (child) function from within, and the child function retains access to its scope, regardless of the context in which it is invoked.

99. Variable lookups are done at execution time. So when it comes to closures, even if a variable is declared after a function definition in which that variable is being used, and that function is returned only after the variable declaration, the function will still have access to the latest value of the variable since it will be declared by the time that function gets called.

100. A closure can be used to restrict access to variables, which you wouldn't want to change accidentally or unnecessarily.