1. JavaSript makes a new release every year. The features to be included in a release are suggested and a vote is conducted at the early part of a year. They are then officially ratified by ECMA (the organization that manages JavaScript standards). There are five stages (Stages 0-4) to the proposal process and if a feature has made it to the fifth stage (Stage 4), it is most definitely going to be included in the release. Features on the fourth stage and below (Stages 0-3) are postponed to the next release. These features could be of any size and impact (small or big). Releases are named based on the year that they get released (ES2020, ES2021, etc.).

2. The evolution of JavaScript is about moving towards being more declarative rather than imperative. For example, developers used to do many things using hacks in JavaScript for a long time, such as converting the ‘arguments’ pseudo-array into a real array for working with the argument list and so on. Newer releases of JavaScript have brought official support for such tasks to make it more declarative, like the Rest Operator which collects all the arguments in a real array for a function. This declarative approach makes the code more communicable.

3. Transpilers like Babel compile our code which is authored in the latest JavaScript standard, to code of an older standard so that it also works on older browsers. There is always going to be a gap between what we need to support and what the leading edge of technology is. We can bridge that gap using transpilers.

4. Template Strings (or Template Literals) can also be called Interpolated Literals. Interpolation is when we concatenate strings with other data values such as strings, numbers, booleans, etc. The imperative and pre-ES2015 (ES6) method to do this is by using the plus operator with the strings and data values, which requires us to mentally execute the code to figure out the final output of the whole operation. Template strings allow us to declare our final string and have placeholders for where we want to use the data values.

5. We declare a template string using the backtick (`) delimiter. We can then put a JavaScript expression inside of the curly braces of ‘${}’ in the string. The expression can be anything from a variable reference to an IIFE. A template string is not a template, but the string itself. It cannot be used over and over again like a template. It is similar to an IIFE as there is a bit of processing involved just after you describe it, where it constructs the string and inserts the data values into it.

6. Template strings also don’t require a backslash to insert a line break and continue the string from the next line. Regular strings require a backslash to do this. Continuing a string from the next line is called String Continuation. Something to note is that in template literals, line breaks, padding spaces, and tabs, persist in the resulting string, whereas in regular strings, they are ignored.

7. Template strings are declarative when compared to regular strings, as we are asking JavaScript to give us a string with data values in them, and JavaScript does just that without us having to manually figure out the string from confusing code.

8. We can control the processing of a template string that happens before it gives out a string. We can prefix template literals with a function’s name without any operator in between (whitespace between them doesn’t matter). This is called a Tagged Template Literal (or String) which is a special kind of function call. The return value of

that function is the string that gets constructed by the template literal. We can pre-process a string before using it.

9. The function used for a tagged template literal will receive an array of strings as its first argument, and the data values as individual arguments to the function (which can be collected in an array using the rest operator). The array of strings will contain the string literals in the template string (everything except the data values). We generally loop over the array of strings and concatenate the data values after every string literal in that array and return the resulting string.

10. For looping and interpolating inside a tag function, JavaScript makes it convenient for us by adding one more value in the array of strings, than the number of data values. The last string in the array of strings will be an empty string due to this. The first string in the array will also be empty if the template literal starts with a data value. \`${value}\` becomes ‘’ + value + ‘’. The array of strings have two strings in it as there is only one data value. We can safely interpolate without worrying about the existence of a data value, thanks to this behavior. The first string to be concatenated to the result will always be the first element in the array of strings, and if our loop is at the last element of the array, there won’t be a data value to be concatenated after it.

11. Tag functions that can be used with template literals can be found on NPM. There are tag functions for almost all kinds of processing that we would need and we can just import and use them with our template literals. Tagged template literals (or their tag functions) don’t necessarily have to return a string. They can return anything we want. One use-case of this is that we can construct regular expressions in a readable format inside a template literal, and use a tag function to parse it into a real regular expression and then return it. We can create an entire programming language inside of a template literal using a tag function (as an interpreter) if need be.
    Note: There is also a JSX tag function to write HTML in template literals, which returns the actual DOM elements created by that HTML (like in React). Original JSX requires a compiler since it is not native to JavaScript, but template strings are.

12. String Padding (prefixing or suffixing a string with one or more of the same character) and String Trimming were introduced to JavaScript as methods on the ‘String’ function’s prototype in the ES2017 and ES2019 standards. Instead of doing left and right padding, JavaScript now does start and end padding, since some languages may be of the RTL (Right To Left) format.

13. The ‘padStart’ function takes in two arguments. The first is the length to which the string should be padded. If the length of the string is less than or equal to the length to be padded, the string is not padded. If it is more, the string is padded with spaces on the left for LTR languages and right for RTL languages, till it grows to the specified pad length.
    ‘Hello’.padStart(5); // “Hello”
    ‘Hello’.padStart(10); // “   Hello”
    The second argument is a string from which the characters to pad the string with is taken. The characters in the string are pulled out one by one from the left-most character and padded to the target string till the padding length has been reached. If there aren’t enough characters to pull out in the string (argument), it starts pulling again from the beginning of the string till the padding completes.
    ‘Hello’.padStart(10, ‘<>’); // “<><><Hello”

```
‘Hello’.padStart(10, ‘*’); // “*****Hello”
```

14. The ‘padEnd’ function works the same as the ‘padStart’ function but pads at the end of the string (respecting the LTR or RTL behavior of the language). The string from which characters can be pulled out (the second argument) behaves the same, as it will still pull characters from the left-most character of the string. There is no function to pad from both sides in the standard as it isn’t very necessary.

15. The ‘trim’ function trims all kinds of whitespace (including new lines, and tabs) from a string (without mutating the original string) on both sides and returns it. ‘trimStart’ and ‘trimEnd’ does the trimming on the start and end of the string, respecting of the LTR or RTL behavior of the language in that string.

16. Destructuring means decomposing a data structure into its individual parts. The purpose of destructuring is to assign individual parts of a larger data structure to variables (or properties).

17. The imperative (pre-ES6) way of doing something similar to destructuring is by referencing a property on an object and assigning its value to a variable. In this pattern, it is difficult to figure out the structure of a data structure by looking at the code. It is not very communicable. This problem is usually solved by using code comments to document the structure of the object, but the problem with that is, the structure in the comment can get outdated in cases, like if the object is a response from an API call, and that API makes changes to that object internally. This is why destructuring was introduced. To provide official support for this pattern and make the code more communicable.
    ```
    var user = data[0];
    var userName = user.name;
    var userEmail = user.email ? user.email : ‘johndoe@example.com’;
    ```
    The above code can be rewritten like the following, using destructuring:
    ```
    var [
      {
        name: userName,
        email: userEmail = ‘johndoe@example.com’,
      }
    ] = data;
    ```

18. The pattern on the LHS of the assignment is not an array or an object literal. It is the syntax for a pattern, describing the value that is expected from the RHS (‘data’). It not only describes the structure of the value but also assigns individual parts of the value to variables. Line 3 of the above code tells JavaScript to create a variable ‘userName’ and assign it the value of the ‘name’ property which is on the first element (which is an object) of the ‘data’ array. The destructuring syntax tells JavaScript how to break down a data structure and assign individual parts of it to variables. It doesn’t care about the rest of the structure, but only the parts that we describe.
    ```
    var [, second] = [1, 2];
    second; // 2
    ```

19. Line 4 of the above code tells JavaScript to create a ‘userEmail’ variable and assign it the value of ‘data[0].email’. If the property (‘email’) doesn’t exist on that object (‘data[0]’), the variable is assigned the fallback value which is after the equal sign.

This is called a Default Value Expression, where a variable is assigned a fallback value if specified and when a property (individual part) that we are trying to destructure does not exist on that data structure.

20. If an individual part (property) that we are destructuring doesn’t exist on that data structure, and we haven’t written a default value expression for it, the variable is set to ‘undefined’. This is the same behavior when we reference an element at an index that doesn’t exist on an array.
    ```
    var array = [];
    var [first] = array;
    first; // undefined
    first = array[0]; // undefined
    ```

21. The default value expression does a strict equality check with ‘undefined’ and falls back to the default value only if the value of the property is ‘undefined’.
    ```
    var array = [1, , , null, 5];
    var [, second, third = 3, fourth = 4] = array;
    second; // undefined
    third; // 3
    fourth; // null
    ```

22. We can use the rest operator inside a destructuring pattern to collect everything else that wasn’t described in the pattern, from a data structure to a variable (works on both arrays and objects).
    ```
    var array = [1, 2, 3, 4, 5];
    var [first, …fromSecond] = array;
    fromSecond; // [2, 3, 4, 5]
    var object = {
      id: 1,
      name: ‘John Doe’,
      email: ‘johndoe@example.com’,
    };
    var { id, …details } = object;
    details; // { name: “John Doe”, email: “johndoe@example.com” }
    ```

23. The rest operator evaluates to an empty array if there is nothing to gather (or collect).
    ```
    var array = [1];
    var [first, …fromSecond] = array;
    fromSecond; // []
    ```
    This is the same behavior of when we use the ‘slice’ method instead of the rest operator.
    Note: The rest operator can only be used at the end of a destructuring pattern as it collects everything else till the end. There shouldn’t be any more destructuring in that level of that data structure after using the rest operator.

24. We can chain assignments to reduce the lines of code written.
    ```
    var data;
    var { name } = data = getData();
    ```
    Assignments are evaluated from right to left.

25. Destructuring only does the assignment. The declarations (if any) of the variables to which the individual parts are destructured will be handled by JavaScript (the compiler specifically).

26. Any variable that can be assigned a value is referred to as a Valid Left-Hand Side Target in the specification. Any valid left-hand side target can be the target of a destructured assignment, may it be variables or object properties.
{ name: user.name, email: user.emails[0] } = data;
Here, the values of the 'name' and 'email' properties on the 'data' object are assigned to the 'name' property and 'emails[0]' (position in the array) of the 'user' object.

27. { name } = data = getData();
data = { name } = getData();
Both lines above work the same way. 'data' receives the return value of 'getData' and not the subset that was extracted by the pattern. This is because it first does:
{ name } = getData();
The above line returns (or is evaluated to) the value on the RHS (the return value of 'getData') as an assignment expression always evaluates to the data that was subject to the assignment (the RHS).
data = { name } = getData();
// data = ({ name } = getData());
// data = return value of getData();

28. We can put a comma without a target variable when destructuring an array to skip the assignment of a value at a specific index position.
var [first, , third] = [1, 2, 3]; // The value at index 1 is not assigned to anything.
This is called Array Elision. It can be done anywhere in the pattern, including before the rest expression, to exclude some values from being gathered if we want. It is good to put the comma after a line break for better readability.

29. We can swap values of variables using array destructuring.
[x, y] = [y, x];

30. We can do destructuring on the parameters of functions and the target variables are automatically declared and assigned their values based on the pattern we describe.
function useData({ name: userName }) { … }
Here, the 'userName' variable is created automatically for us and assigned the value of the 'name' property on the argument that got passed in.
function useData([first]) { … }
Array destructuring also works the same way. We are saying that we don't care about the complete data structure but only its individual parts, which we assign to targets using a destructuring pattern.

31. If we try to do array or object destructuring with a value that is not an array or an object, a 'TypeError' is thrown. This behavior is similar to when we try to access a property or index on something that is not an object or an array. It just goes on to say that destructuring is just a declarative syntax for something that we used to do imperatively (assigning values of properties to variables). We can provide a fallback value in this case to avoid the error.
var data = getData() || [];

32. To prevent getting ‘TypeError’s with parameters, we can set default values for the parameters using the equal sign. We can go further and set default values for the destructured variables as well.
    ```
    function useData([first = 1] = []) { … }
    ```

33. We can destructure a nested data structure by nesting its inner structure(s) inside of the destructuring pattern itself.
    ```
    var [first, [second, third], fourth] = array;
    ```
    If the inner data structure that we are destructuring from inside the pattern turns out to be a value that cannot be destructured from (like ‘undefined’), we need to fall back to a default value. A default value can be specified, just like we would with a target variable, inside the destructuring pattern.
    ```
    var [first, [second, third] = [], fourth] = array;
    ```

34. When doing object destructuring, we specify the name of the property (the source) and the target to which we want to assign its value, and then an optional default value after an equal sign.
    ```
    var {
      source: target = defaultValue,
    } = object || defaultObject;
    ```
    The order of the properties doesn’t matter like with arrays.

35. While doing object destructuring, when we collect the remaining properties on a data structure into a target variable using the rest operator, it is called Object Rest. See 22.

36. Points 20, 21, 22, 23, 26, 30, 31, 32, and 33 apply to object destructuring as well.

37. The reason for flipping the order of the target and source when destructuring an object is so that the structure of that object can be written as it is, in the destructuring pattern.
    ```
    var object = {
      property: value,
      // target: source
    };
    var {
      property: target,
      // source: target
    } = object;
    ```
    Here, the position of the target and source is flipped, but the original structure of the object is persisted and observed in the destructuring pattern, making it easier to reason about and communicate, if we would just exclude the idea of a target and a source when destructuring. We just have to remember that the property name is written on the left side of the colon in both cases (inside object literals and destructuring patterns) and the thing that we’re assigning to it, or the thing that we assign it to, is on always on the right.

38. We cannot omit the ‘var’ keyword when object destructuring like how we can with array destructuring if the variables have already been declared because, as the expression starts with a curly brace, JavaScript will throw a syntax error. This is because the curly brace in JavaScript is overloaded with many different operations.

JavaScript will think that we are creating a block and not a destructuring pattern. So, to omit the declarations and just assign the destructured values to existing variables, we have to wrap the whole assignment expression in parenthesis.
```
({
  name: userName,
  email: userEmail,
} = data);
```

39. We can also chain assignments to make the expression not start with a block when we don't want to declare a target for destructuring on that expression itself. We also get a reference to the entire data structure this way.
    ```
    var data = { name } = getData();
    ```

40. If the property name and the target variable name are the same when destructuring a value, we just have to list the property name (or variable name) once.
    ```
    var { name } = user;
    ```
    The above line says, declare a variable 'name', grab the value of the 'name' property from the 'user' object, and assign it to the 'name' variable.
    ```
    var { name = 'John Doe' } = user;
    ```
    Here, the default value is assigned to the 'name' variable if there is no 'name' property on the 'user' object. It only works if a variable with the same name as that property exists in the lexical scope and can be assigned a value (when destructuring to already declared variables).

41. When destructuring nested objects, if we specify an object with default values in it as a fallback for a nested object and its properties, that fallback object is only used if the property which contains the nested object doesn't exist on the object from which we are destructuring.
    ```
    var object = { a: { b: 1 } };
    var { a: { b } = { b: 2 } } = object;
    ```
    If we do the above, the '{ b: 2 }' object is only used as the fallback if 'a' doesn't exist or is set to 'undefined'. If 'a' is indeed a property on 'object', '{b: 2}' is not used, and if 'b' doesn't exist on 'a', 'b' will not have a fallback value and becomes 'undefined'. So it is better to use empty objects as fallbacks for nested objects, and the default value expression syntax for all properties inside of those nested objects.
    ```
    var { a: { b = 2 } = {} } = object;
    ```

42. A feature of object destructuring which isn't there for array destructuring is that we can destructure a property more than once.
    ```
    var {
      a,
      b,
      b: { c },
    } = object;
    ```
    Here, we first destructure the value of the property 'b' to a new variable 'b'. In the next line, we destructure 'c' from inside the nested object 'b' to a new variable 'c'. We can destructure from the same property multiple times like this. This is especially useful in cases like the above where we need the nested object in a variable by itself, and its individual properties separately in other variables.

43. We can have array patterns inside of object patterns and vice versa in a destructuring pattern.
    var { a: [b, c] = [] } = object || {};

44. We can implement named arguments with the help of object destructuring. Instead of having to remember the position or order of the arguments to a function, we can have the function accept a single object as its argument, in which the values are passed in as named properties and destructured in the function's parameter list.
    function addUser({ name = 'John Doe', email = 'johndoe@example.com' }) { … }
    addUser({ email: 'janedoe@example.com' });
    Here, we don't have to remember the order of the arguments and can pass a single object with named properties that are destructured in the function's parameter list. A downside is that we'll have to remember the names of the properties that are destructured in the function definition. This can be countered by using a set of conventions for naming parameters of functions.

45. To merge two objects while replacing one's properties with the other's, we had to use library methods such as '_.extend' from the Underscore library.
    fetch(_.extend({}, defaultOptions, options));
    The problem with this is, it is not very readable, and we have to rely on the implementation of an external library to do it. We can solve this using a pattern called Destructuring & Restructuring. We destructure the 'options' object provided to our function while falling back to the default values for properties that were not included on that object. We then restructure a new object inside the function using those destructured values and return it.
    ```
    function getOptions({
      headers: [
        contentType = 'Content-Type: application/json',
        …otherHeaders,
      ],
      method = 'POST',
      url = DEFAULT_URL,
    } = {}) {
      return {
        headers: [contentType, …otherHeaders],
        method,
        url,
      };
    }
    fetch(getOptions()); // Call 'fetch' with the default options.
    fetch(getOptions(options)); // Call with our 'options' merged with the default options.
    ```
    In the above function, we destructure the passed in 'options' object while setting the properties that were not included on it, to their default values. We also use the rest operator to our convenience. Then we restructure the object using the destructured values and return it for use with the 'fetch' function.

46. The 'Array.prototype.find' method (ES6) takes in a callback which is run for every item in the array that it is used on while receiving each item as its argument on every call. If the callback returns a truthy value when run for an item in that array, that item is returned from the 'find' method call. The 'indexOf' prototype method does an identity check between its argument and each element in the array that it is used on,

to find a specific element in that array. It cannot find an item based on a custom condition like with ‘find’. The ‘find’ method returns ‘undefined’ if no element passes the truth test. This is indistinguishable from when an item in the array with the value ‘undefined’ passes the truth test. We can’t know if it didn’t find the element or the element it found is actually ‘undefined’.

47. The above indistinguishability problem can be solved by using the ‘findIndex’ method instead of ‘find’. The signature of ‘findIndex’ is similar to ‘find’ but it returns the numeric index of the element that is found instead of returning the element itself. If the element is not found, it returns -1 (just as ‘indexOf’ would because of historical reasons). It is like ‘indexOf’ in terms of functionality but allows us to use a custom condition rather than just doing an identity check.

48. We used to use the ‘~’ (Bitwise Not) operator to use the return value of the ‘indexOf’ method as a condition. That is, to check whether that element exists (was found) or not.
var array = [1, 2, 3, NaN];
array.indexOf(4) != -1; // ‘false’. Not very communicable as -1 doesn’t mean anything.
~array.indexOf(4); // -0 (falsy) because ~N = -(N + 1).
~array.indexOf(1); // -1
~array.indexOf(NaN); // -0 (falsy) because ‘NaN’ is not equal to itself.
As of ES2016, we can use the ‘includes’ method that is on every array to do this. It does the same as ‘indexOf’ but returns a boolean value that indicates whether the value was found in the array or not. It also compares reference values (similar to what ‘Object.is’ does), which makes the method even better.
array.includes(NaN); // ‘true’. ‘includes’ does a much better (stricter) check.
We can also provide a second argument to ‘includes’, which is the index from which we want to start the search from.
array.includes(2, 2); // ‘false’. 2 is at the index 1. We search starting from index 2.
array.includes(2, -2); // true
If the second argument (‘fromIndex’) is a negative number, the search starts from the index number ‘array.length + fromIndex’, then starts over from index number 0 when the end of the array has been reached and goes all the way till the end again.

49. The ‘flat’ method on arrays that was recently introduced (ES2019), flattens an array (removes nesting basically). We can provide a number as its argument which tells the function as to how many levels of nesting should it flatten. The default number of levels flattened is 1, if no argument has been passed in.
array = [1, [2, 3], [[]], [4, [5]], 6];
array.flat(0); // ‘array’ itself. Doesn’t flatten.
array.flat(/* default is 1 */); // [1, 2, 3, [], 4, [5], 6]
array.flat(2); // [1, 2, 3, 4, 5, 6]
array.flat(Infinity); // Flattens all levels of nesting.

50. The ‘Array.prototype.flatMap’ method (ES2019) performs a ‘flat’ operation on the result of a ‘map’ operation on an array. The ‘flatMap’ method flattens and maps at the same time, the return value of the callback that it takes.
[1, 2, 3].map(value => [value * 2, String(value * 2)]).flat(); // [2, “2”, 4, “4”, 6, “6”]
Here, we have an intermediary array as the result of the ‘map’ call, and we perform ‘flat’ on it. The ‘flatMap’ method lets us do this with a single method call while being much more performant and having a better implementation with regards to our intent.

[1, 2, 3].flatMap(value => [value * 2, String(value * 2)]); // [2, “2”, 4, “4”, 6, “6”]
The flattening in ‘flatMap’ is only done for one level of nesting, and any additional flattening would require another call of the ‘flat’ method.

51. The ‘flatMap’ method can also be used as a method that adds or removes elements to the resulting array. A regular ‘map’ call returns a mapped array with the same number of elements as the array from which it was mapped. But by using ‘flatMap’, we can skip adding to the resulting array by returning an empty array from the callback, which gets flattened to nothing, or we can add multiple elements in a single iteration by returning an array of elements from the callback, which is flattened at that position.
[1, 2, 3, 4, 5, 6].flatMap(value => !(value % 2) ? [value, value * 2, value * 3] : []);
// [2, 4, 6, 4, 8, 12, 6, 12, 18]

52. The Iterator Pattern is when we have a data source (like an array) and we consume (use) the values in the array, one at a time. An iterator is an object with a ‘next’ method on it, which when called returns the next value from that data source. Iterators can be used inside of loops to consume all the values in a data source, one by one. In ES6, values of all basic built-in data types have been changed to iterables, that is, something that can be iterated over.

53. For instance, a string or an array can be used to construct an iterator, and we can iterate over it. This can be done by invoking the ‘[Symbol.iterator]’ special location (a meta method) on that value. The return value of the method call is an iterator that can be iterated over.
When we call the ‘next’ method on that iterator, we get an iterator result that has a ‘value’ property on it that contains the value of that iteration, along with a ‘done’ property, which is a boolean that tells us if there is anything more to be iterated.
var string = ‘Car’;
var iterator = string[Symbol.iterator]();
iterator.next(); // { done: false, value: “C” }
iterator.next(); // { done: false, value: “a” }
iterator.next(); // { done: false, value: “r” }
iterator.next(): // { done: true, value: undefined }
The ‘done’ property is true only when we iterate past all the values in the data source. When the iteration result has its ‘value’ as ‘undefined’ as the current iteration doesn’t have anything to consume, then we get ‘true’ as the value of the ‘done’ property. No matter how many times we call the ‘next’ method after we get past all the values in the data source, we always get ‘{ done: true, value: undefined }’.

54. Most of the values in JavaScript can be made an iterator out of. An iterator (the existing interface at least) also doesn’t go backward in a data source. It only goes forward when calling ‘next’.

55. The ‘for…of’ loop was introduced in ES6 to loop over iterables. It assigns the value of the ‘value’ property (of the iterator result) to the loop variable and calls the ‘next’ method (on the iterator) automatically at the beginning of every iteration.
for (let value of iterator) { … }
We can do the above because iterators are also iterables. We can also do,
for (let value of string) { … }
This is because a string is an iterable. Iterables, as well as iterators, can be iterated

over by the ‘for…of’ loop. The loop automatically creates an iterator for us in this case by invoking the ‘[Symbol.iterator]’ location on the value.

56. We can also use the spread operator to spread out all the values from an iterator (or iterable) to wherever it is valid to do so.
```
var string = ‘Car’;
var split = […string];
split; // [‘C’, ‘a’, ‘r’]
```
‘…’ and ‘for…of’ are official syntactic support for the iterator protocol.

57. The main use-case for iterators is that we can make our custom data structures into iterators by making them adhere to the iterator protocol. We can create a data structure that has a ‘next’ method on it that gives out an iterator result, and we get all this syntactic support for things like spreading and looping that data structure. It creates a standardized way of iterating through data sources.

58. Not all data structures are iterators. An object is not an iterator. If we try to loop through an object using a ‘for…of’ loop, we get a ‘TypeError’. We get a ‘TypeError’ because it tries to invoke the ‘[Symbol.iterator]’ location on that object, which doesn’t exist on it. So it ends up invoking ‘undefined’ which is illegal to do for that type.

59. We can define a ‘[Symbol.iterator]’ method on an object to make it an iterable. We can then manually create iterators of that object by calling that method. The ‘for…of’ and spread operator invokes that method when used with iterables to construct iterator instances out of them.
```
var object = {
  email: ‘johndoe@example.com’,
  name: ‘John Doe’,
  [Symbol.iterator]: function Iterator() {
    var index = 0;
    var keys = Object.keys(this);
    return {
      next: () => index < keys.length ?
        { done: false, value: this[keys[index++]] } :
        { done: true, value: undefined }
    };
  }
};
[…object]; // [‘John Doe’, ‘johndoe@example.com’]
```
It doesn’t throw a ‘TypeError’ and successfully spreads the values from that object because that data structure is now an iterable since it has the method that can construct an iterator instance out of it.

60. Generators (ES6) are special functions. Then don’t execute their body when invoked, but returns an iterator. Generators must have an asterisk (*) prefixed to the function name in their definition. We can also use the ‘yield’ keyword inside of generators. We can yield values using that keyword. When we call the ‘next’ method on the iterator returned by a generator, we get the next value that is yielded by that generator.
```
function *generator() {
  yield 1;
  yield 2;
  return 3;
}
var iterator = generator();
iterator.next(); // { done: false, value: 1 }
iterator.next(); // { done: false, value: 2 }
iterator.next(); // { done: true, value: 3 }
```

Here, the result of the last ‘next’ call has its ‘done’ property set to ‘true’. This is because we returned that value from the generator using the ‘return’ keyword. The iterator knows that there cannot be another value after it, since we will have exited from the generator function after consuming that return value. When not using generators, the only way an iterator can know if all the values in a data source have been consumed is by trying to consume the next value by calling the ‘next’ method and not finding any value left in that data source.

61. It is not good to return a value from a generator. When we spread the values from the above iterator into an array, the returned value (3) does not get spread. This is because, as far as the spread operator is concerned, the ‘done’ property is ‘true’ for that iterator result so it stops spreading, and in the process ignores the result’s value. It is always recommended to yield values from a generator and not return them, to make them follow the iterator protocol.

62. Using generators, we can automate everything inside the ‘[Symbol.iterator]’ method definition, and focus only on yielding out the values. The generator will take care of everything else including returning an object (iterator instance) with a ‘next’ method on it, that returns iterator results with the values that we yielded.

```
var object = {
  email: ‘johndoe@example.com’,
  name: ‘John Doe’,
  *[Symbol.iterator] () {
    for (let key of Object.keys(this) {
      yield this[key];
    }
  }
};
[…object]; // [‘John Doe’, ‘johndoe@example.com’]
```

Generators are declarative compared to the imperative way of manually writing out the implementation for the ‘[Symbol.iterator]’ method.

63. A Tuple is just an array with two elements. We can create an iterator that iterates over the entries ([key, value]) of an object by yielding out ‘[key, this[key]]’ instead of ‘this[key]’. It is good to define multiple iterator constructors as properties of our data structure so that we can iterate over multiple factors like keys, values, entries, etc. of that data structure.

```
for (let entries of object.entries) { … }
```

We can define a ‘[Symbol.iterator]’ generator on the ‘entries’ property to make this functional.

64. The iterators pre-defined in JavaScript iterates over a snapshot of the iterable, which was taken when that iterator instance was created. The custom iterators that we create on our data structures may not have this implementation detail, and the iteration could be affected if we change the structure or values in that data structure,

and especially when we change it while we're iterating over it.

65. A set of features were added in ES2018 for improving regular expressions. Look-ahead is an assertion that says, when we are matching something in a regular expression, it is only matched if the thing immediately after it is also a match. We also use assertions for checking if a character matches at the beginning or end of a string (^ and $). Assertions are saying, in addition to the thing that is being matched, something else has to be true for it to match.
'Hello World'.match(/(l.)(?=o)/g); // ['ll']
Here, the '(?=o)' part says that we only want to match '(l.)' if it is followed by an 'o' character. Hence the 'll' in 'Hello' is matched as it is followed by an 'o'. This is a look-ahead assertion.
'Hello World'.match(/(l.)(?!o)/g); // ['lo', 'ld']
This is a negative look-ahead assertion and will match any '(l.)' that isn't followed by an 'o'. Hence matches 'lo' in 'Hello' and 'ld' in 'World'.

66. Look-behind assertions (opposite of look-ahead) (ES2018) were not supported by JavaScript for a long time, and we had to reverse the string being matched (or matched against) and use a reverse look-ahead pattern to achieve this. Now we have official support for this in JavaScript.
'Hello World'.match(/(?<=e)(l.)/g); // ['ll']
This matches 'll' as it is preceded by an 'e'.
'Hello World'.match(/?<!e)(l.)/g); // ['lo', 'ld']
This matches 'lo' and 'ld' as they are not preceded by an 'e'.

67. A Capture Group in a regular expression is an expression written inside a set of parenthesis. In regular expressions, parentheses are not just grouping operators, but also capturing operators. It groups some characters together and also pulls out the match of that capture group (the sub-pattern) separately, with the match result of the entire regular expression.
'Hello World'.match(/.(l.)/); // ['ell', 'll']
This matches 'll' as well, because it is a match of the expression inside the capture group '(l.)'.
We can reference the capture groups (the patterns in them) inside a regular expression by using '\n' (back-reference) where 'n' is the position of the capture group in the expression (starting from 1).
'Hello Workd'.match(/([jkl])o Wor\1/); // ['lo Work', 'l']
Here, 'lo Work' matched because '\1' refers to the pattern in the first capture group which is '[jkl]'. 'l' was matched after it separately, as it is the matched result of the capture group alone. The above expression is the same (conceptually) as doing:
'Hello Workd'.match(/([jkl])o Wor[jkl]/); // ['lo Work', 'l']
The back-reference doesn't create another capture group at that place. It just uses the pattern inside that referenced capture group (it uses '[jkl]' and not '([jkl])').

68. We had to use numeric references to reference capture groups inside of regular expressions till the introduction of Named Capture Groups (ES2018). We can name a capture group using the '(?<name>expression)' syntax. The capture group matches are also added to the 'groups' property on the result, which is an object that has the names of the capture groups and their matched values.
'Hello World'.match(/.(?<capture>l.)/); // ['ell', 'll', groups: { capture: 'll' }]
Here, 'capture' is the name of the capture group, and its matched value is available

by name, on the ‘groups’ object on the result. We can also reference a named capture group by name, inside the regular expression itself by using the ‘\k<name>’ syntax.
‘Hello Workd’.match(/(?<capture>[jkl])o Wor\k<capture>/); // [‘lo Work’, ‘l’]
Here, we don’t have to figure out the position of the capture group and can just reference it by its name.

69. We can also use a named capture group inside a ‘replace’ call. We can reference the value(s) which matched that capture group when replacing.
‘Hello World’.replace(/(?<capture>l.)/g, ‘-$<capture>-‘); // “He-ll-o Wor-ld-“
Here, the matched parts of the string are replaced with another string, that uses the matched values of the ‘capture’ capture group by using the ‘$<name>’ syntax. Multiple parts of the string are being replaced here because we’re using the ‘g’ (global) flag to match them all. If the first argument is not a regular expression, the second argument is considered as a normal string literal and not as something that includes a reference to a named capture group. If we’re replacing using a callback, we receive the named capture groups (‘groups’) object as the last argument to that callback.
```
‘Hello World’.replace(/(?<capture>l.)/g, (…arguments) => {
  const [, , , , groups] = arguments;
  return groups.capture.toUpperCase();
});
// HeLLo WorLD
```
It is harder to figure out the positions of capture groups in a regular expression, and the positions are subject to change, upon adding or removing a capture group from the expression. Hence, named capture groups are preferred and more semantic.

70. The ‘s’ flag turns on Dotall mode. It is a flag like ‘g’ (global), ‘m’ (multiline), ‘i’ (case-insensitive), etc. By default, the dot (.) character in a regular expression can match any character but the new-line character. If we turn on dotall mode, the dot character matches any character, including the new-line character (‘\n’). The ‘u’ flag turns on Unicode-aware mode for a regular expression. Regular expressions normally only consider ASCII characters, so this could be useful. It also gives us additional syntax for Unicode escape sequences in our regular expressions.

71. To match a string against a regular expression, we can use the ‘exec’ method on the regular expression and pass in the string to match, as its argument. When ‘exec’ is called on a regular expression that has the ‘g’ flag, it updates a ‘lastIndex’ property on the regular expression. This is so that on every subsequent call of ‘exec’, we get the next match from the matches. Hence, we can do this:
while (match = regularExpression.exec(string)) { … }
On every iteration, the next match is assigned to the ‘match’ variable until all the matches have been iterated through. ‘exec’ also gives us the ‘groups’ property on its result, which is an object with the regular expression’s named capture groups and their values.

72. The ‘(?:expression)’ syntax can be used to create a group without capturing its match. The ‘?’ symbol after any character or a group makes it optional for matching.

73. Before the introduction of async functions, we used to use callbacks and then promises to make our programs asynchronous. Then, because there were

generators and promises in ES6, tools like BlueBird (runners) used them to write asynchronous code synchronously. This is because a generator can be paused using the ‘yield’ keyword and then resumed from where it left off. So these tools pause the execution of a function upon an asynchronous operation call in it, that returns a promise. Then later when the promise resolves, the runner resumes the execution of the generator and provides the resolved value to it.

```
runner(function* setUsers() {
  // Yields a promise and pauses the execution of the generator.
  var users = yield readUsers();
  // Runner passes the resolved value in the ‘next’ call, which is assigned to ‘users’.
  // Generator resumes its execution.
  state.users = users;
});
```

The runner will take care of pausing and resuming the generator based on the events of the promise.

Note: A value can be passed into the ‘next’ method of an iterator. The value passed is used as the evaluated value of the ‘yield’ expression. In the above code, the value of ‘users’ will be whatever that was passed into the ‘next’ call, by the runner that handles it. The execution doesn’t resume from the next line. It resumes from the same line at where the generator yielded and evaluates the ‘yield’ expression to the argument passed into the ‘next’ call.

74. Async functions (ES2017) just provide syntactic support for the above pattern of pausing and resuming. We prefix the function definition with the ‘async’ keyword instead of creating a generator and using a runner. Inside the function’s body, we prefix promises with the ‘await’ keyword, instead of with the ‘yield’ keyword in generators. The execution of the function pauses till the promise is resolved, and later gives out the resolved value and resumes. Essentially they made the runner into official JavaScript syntax.

75. Nesting functions inside of async functions create function barriers where we cannot use the ‘await’ keyword inside of a nested function if the nested function is not an async function.

76. We cannot pass in async functions to methods like ‘forEach’ as callbacks. This is because async functions always return a promise, and ‘forEach’ does not know what to do with it. It doesn’t know to wait on a promise before running the callback again for the next value in the array. So the next iteration’s callback is called immediately and before the current iteration’s callback call finishes its execution. All the existing higher-order array methods are synchronous iterators that run over every value of an array eagerly. It doesn’t pause for a promise to resolve, before running the callback again on the next iteration. JavaScript needs to have eager asynchronous iterators (which doesn’t exist officially in JavaScript) for this purpose. Kyle has a library for this called Fasy.

77. The ‘await’ keyword can only wait on promises. It cannot wait on a custom implementation of a promise that doesn’t have a ‘then’ method on it, or on a thunk which is a function representation of a future value. With a generator, we can yield and pause on any type of future value as necessary, and not just on promises.

78. Another problem with async functions is starvation, which is caused due to the scheduling algorithm. Promises use the micro-task queue and not the callback queue. Promises defer functionality to after receiving a future value, or to the next “tick” as people say. This functionality is queued into the micro-task queue upon receiving the future value. The callbacks in the micro-task queue are run before the callbacks in the callback queue, which contains all the triggered events’ handlers and such. So it may lead to starvation of the callback queue, as the functions in it may never get to run if callbacks are continuously being added to the micro-task queue. It can intentionally or accidentally create infinite loops, where promises can resolve promises inside themselves and keep on adding callbacks to the micro-task queue and starve the callback queue.

79. Starvation creates a form of denial of service. It creates a form of deadlock. CSP (Communicating Sequential Processes) is a concurrency pattern in programming.

80. Another problem with async functions is that they cannot be canceled externally, once initiated. This can lead to starvation, as the function may continuously keep adding callbacks to the micro-task queue, and we have no way of stopping it externally. This can be solved using cancellation tokens. The ‘fetch’ (facade) function has this feature where we can pass to it a cancellation token, and we can cancel the request externally if needed, by calling abort on the cancellation token. Kyle has a library to do this for all asynchronous functions that use generators with runners, called CAF. A cancellation token is passed into the asynchronous function, and when abort is called on the token externally, a signal is sent into the function that tells it to stop running. This is particularly useful when we want to cancel an asynchronous operation if it takes too long to complete, and display a timeout message for it.

81. Async generators (ES2018) are special functions that can wait and pull out values from promises, like in async functions, and also yield or push out values out of them like in generators. It is a combination of an async function and a generator. We can use both ‘yield’ and ‘await’ keywords in an async generator. A use-case of this is that, when we are looping through a list of URLs and fetching them asynchronously, we can only make use of the fetched results once all the URLs in that list have been fetched from, but, with async generators, we can yield out a fetched result from each loop and make use of it without having to wait around for the completion of the entire loop. We can essentially pull out and consume data as they get resolved, one by one. We can lazily push out results of asynchronous operations from our function one by one instead of all at once and pull it out lazily from elsewhere as they’re being pushed out from the function. This (pulling and pushing) can be done using a runner function and a generator, but is not very ideal as we are overloading the meaning of the ‘yield’ keyword, which will make it harder for us to determine its meaning at a specific line, and also makes the code less readable.
async function* asyncGenerator() { … }

82. Instead of using an async function that gives us a promise which resolves once all the asynchronous operations inside it are complete, we can use an async generator that will yield out the results of each asynchronous operation as they complete. But we cannot use an async generator’s iterator with a ‘for…of’ loop or the spread operator. This is because the iterator result of an async generator’s iterator cannot be determined until the asynchronous operation to get that yield value is complete. The loop (‘for…of’) will be eagerly executing (without waiting for the promise resolution

inside our function) but we're lazily pushing out values (only after each promise resolution) from the function. We have to yield out a value before the promise gets resolved to a value. So instead of returning an iterator result upon calling the 'next' method, async generators return a promise which resolves to the iterator result. We get a promise for the iterator result and not an iterator result with a promise in it because there could be cases where we cannot determine the value of the 'done' property until we receive the resolved value of the asynchronous operation. And a 'for…of' loop or the spread operator cannot call the 'next' method on a promise because the method only exists on an iterator result. Hence, we cannot use an async generator's iterator with a 'for…of' loop or the spread operator.

83. To consume the results of an async generator's iterator, we'll have to loop through it inside of an async function and await the result of each promise yielded by the 'next' call.

```
async function consumeAsyncGenerator() {
  var iterator = asyncGenerator();
  while (true) {
    var result = await iterator.next();
    if (result.done) break;
    console.log(result.value);
  }
}
consumeAsyncGenerator();
```

The above code will lazily consume each value yielded from the async generator. This is called Lazy Asynchronous Iteration. See 76 to see about eager asynchronous iterations.

84. In the interest of being more declarative, JavaScript provided syntactic support for the above pattern and introduced the 'for await…of' loop. The loop loops over an async generator's iterator and automatically awaits each promise received from the 'next' call to get an iterator result, before deciding whether it needs to do another iteration and if yes, assigns the value of that iterator result to the loop variable.

```
for await (let value of asyncGenerator()) { … }
```

We now have generators that can push out values immediately, we have async functions that can pull out values from promises (asynchronous operations), we have async generators that can pull out values from promises as well as push them out lazily (as they get pulled out), we have libraries like Fasy to implement eager asynchronous iterations, and finally, we have lazy asynchronous iterations using async generators and the 'for await…of' loop.
Note: In programming, eager means the action happens immediately as it is encountered, and lazy means we can defer the action or not do it at all if needed.