https://github1s.com/webber2408/Es6-Samples

1. ECMAScript (June 2020) ECMAScript® 2020 Language Specification
2. What is the code?
3. Statements, Variables (Symbolic placeholders for the value themselves, can be varied with time), literal values
   What happens behind the scene when we do var a = 2 (Compilation Phase - Part of the phase is to find and associate all declarations with their appropriate scopes, e.g. var a; is processed in the compilation phase and a=2; is processed in the execution phase)?
   1. [Var declaration in scope] Compiler checks if `a` exists in the current scope or not.
   2. [Var lookup in scope] Performs an LHS Lookup to check if `a` has been declared, if yes then assigns the value to it.
   LHS vs RHS Lookup:
   https://drive.google.com/file/d/1w5heP03n_hWl9Vnx40QNUaie0AYvEf9V/view?usp=sharing
   Note: If RHS lookup fails. ReferenceError occurs if LHS lookup fails, the variable is declared in the global scope (if not in strict mode, if in strict mode, the same ReferenceError will be thrown!).
4. Value Types (7) (Not Variable Types - Variables are just containers for values) - number/boolean/string/null/undefined/object/symbol(ES6)
   a. Subtypes of type “object”: array/function
   Note (Imp): Built-in type values can be wrapped to their object wrapper counterpart.
   e.g. A `string` value can be wrapped by a `String` object, when we call .toUpperCase() on the primitive string value as the `String` object defines the method, .toUpperCase() on its prototype. Similarly, a `boolean` value can be wrapped into a `Boolean` object and a `number` value can be wrapped into a `Number` object.
   Note: **Primitive Data types in JavaScript include Number, String, Boolean, Undefined, Null and Symbol. The Non-Primitive data type has only one member i.e. the Object**
5. Objects (Compound storage for any type of data)
6. Operators (=, first RHS is calculated and then assigned to LHS)
   a. Operator types
      i. Assignment
      ii. Math
      iii. Compound Assignment (+=, etc)
      iv. Increment / Decrement
      v. Object property access (.)
      vi. Equality & Non-Equality (== loose-equals, === strict-equals, !==, !=)
      (==) checks for value equality with coercion allowed and (===) checks for the value equality without allowing coercion.

https://github1s.com/webber2408/Es6-Samples

Why is “124” == 124? // JS implicitly converts LHS to its number equivalent. (aka Implicit Coercion). JS goes on through a number of steps to coerce one or both values to a different type until the types match, where then simple value equality can be checked. (Check the algorithm here, http://www.ecma-international.org/ecma-262/5.1/#sec-11.9.3)

Note: Forcing numeric comparison (+a == +b)
Forcing string comparison (“” + a == “” + b)
Forcing boolean comparison (!a == !b)
new String(“a”) == “a” (true) but new String(“a) == new String(“a”) (false) as Both the String Object are kept by reference and == or === only checks for the reference and not the actual value. And references here are different.

Explain 12

In JavaScript primitive types are passed around as values: meaning that each time a value is assigned, a copy of that value is created.

On the other side objects (including plain objects, array, functions, class instances) are references. If you modify the object, then all variables that reference that object are going to see the change.

The comparison operator distinguishes comparing values and references. 2 variables holding references are equal only if they reference exactly the same object, but 2 variables holding values are equal if they simply have 2 same values no matter where the value originates: from a variable, literal, etc.

vii. Comparison / Inequality (<= less than or loose-equals)
Explain 13
viii. Logical (&&, etc)
ix. The string concatenation operator (+) Explicit Coercion
x. Bitwise operator (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Expressions_and_Operators#Bitwise)

https://github1s.com/webber2408/Es6-Samples

xi. Below: Nullish Coalescing Operator (?? - assign value if not null) and Optional Chaining Operator (?.)

However, due to `||` being a boolean logical operator, the left-hand-side operand was coerced to a boolean for the evaluation and any *falsy* value (`0` , `''` , `NaN` , `null` , `undefined`) was not returned. This behavior may cause unexpected consequences if you consider `0` , `''` , or `NaN` as valid values.

```
const count = 0;
const text = "";

const qty = count || 42;
const message = text || "hi!";
console.log(qty);     // 42 and not 0
console.log(message); // "hi!" and not ""
```

The nullish coalescing operator avoids this pitfall by only returning the second operand when the first one evaluates to either `null` or `undefined` (but no other falsy values):

```
const myText = ''; // An empty string (which is also a falsy value)

const notFalsyText = myText || 'Hello world';
console.log(notFalsyText); // Hello world

const preservingFalsy = myText ?? 'Hi neighborhood';
console.log(preservingFalsy); // '' (as myText is neither undefined nor
null)
```

```
const a = { duration: 50 };

a.duration ??= 10;
console.log(a.duration);
// expected output: 50

a.speed ??= 25;
console.log(a.speed);
// expected output: 25
```

Important link:
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Nullish_coalescing_operator

https://github1s.com/webber2408/Es6-Samples

xii. Comma operator: (Evaluates multiple statements and assigns the value from the last statement, e.g.
What are the outputs of let y = 24; y = (y+=2, y) // 26
let y = 24; y = (y+= 2, y+= 2) //28
and let x = (2,3) console.log(x) //3)

xiii. Unary operator (delete, typeof)
Delete operator: Should not use on array elements as it just replaces the element with undefined; it returns either true/false)
Typeof operator (returns always a string ("") value)
typeof(null) // "object" → Weird? You have to live with it!
typeof(String/Object/Array/Function/Boolean) // "function"
typeof(62) // "number"
typeof(doesnotexist) // "undefined"
typeof(undefined) // "undefined"
Void operator (evaluates the expression and returns undefined)
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/void

```
void 2 == '2';   // (void 2) == '2', returns false
void (2 == '2'); // void (2 == '2'), returns undefined
```

The `void` operator is often used merely to obtain the `undefined` primitive value

7. What does a = b*2 (Statement); mean to the computer?
8. Expressions (Statements are made up of one or more expressions)
9. How many expressions are there in a = b*2;
    a. 2 -> Literal value expression
    b. b -> variable expression
    c. b*2 -> arithmetic expression
    d. a = b*2 -> assignment expression
10. Expression Statements (A general expression that stands alone, e.g. b*2;)
11. More on expressions, call expression (A function statement)
12. Why is JS an interpreted language? (Because JS source code is processed each time it's run. But that's not entirely accurate, the JS Engine actually compiles the program on the fly and then immediately runs the compiled code)
13. How do we output to the user? (console.log() -> Function Statement)
14. Taking input -> window.prompt from the console
15. Coercing Types
    Let age = 24;
    Let age1 = Number(age)

https://github1s.com/webber2408/Es6-Samples

console.log(age) // “24” (number converted to a string when printed on-screen - Implicit Coercion)
console.log(age1) //24 (forced coercion / explicit coercion)

16. JS Comments (Comments should explain why and not what!).
17. Static typing/type enforcement Number age = 42; (This does not work in JS though!)
18. JavaScript uses weak typing/dynamic typing (Allows a variable to hold any type of value at any time without any type of enforcement)
19. Arrays (It’s an object which holds values in numerically indexed positions):
    Explain 4 5 5.1 5.2 5.3 5.4 5.5(Imp)
    Whenever you want to make an array, always think, Array.from()! E.g. Generating a sequence of numbers to a given number.
20. Conditional Statements: Something about if statement: The if statement expects a boolean but if you provide something else than boolean, implicit coercion will occur e.g. 0 & “” will become false and 99.99 & “free” will become true.
21. Loops: Explain how they work | Loop flowchart
    Explain 8
22. Function: A function is nothing but a block of code that we can call anytime we want to perform a task. Now we do not need to write the same code again and again to perform the same task. | Explain parameters and return statements.
    Explain 11 11.1
23. What are anonymous functions and what are their drawbacks?
    a. No useful name present in the stack trace, debugging is difficult.
    b. Can’t call itself now.
    c. Code less readable / understandable.
24. Scope (Technically called, Lexical Scope [write-time]) - A McDonald’s restaurant cannot serve you Thai food! It’s not in its scope!
    a. Each function has its own scope
    b. No two same variables can be declared in one scope (not let & const at least)
    c. The same name variables can exist in different scopes.

    Note: 1) Engines stop scope look-ups when it finds the first match.
    2) Block scope is a tool to extend the earlier Principle of Least Privilege from hiding information in functions to hiding information in blocks of our code.

    Note: var variables are '**function scope.** ' *What does this mean?* It means they are only available inside the function they're created in, or if not created inside a function, they are 'globally scoped. '
    Explain 1 1.1 1.2 3 11.2(Imp)
25. Scope Chaining - If not found in the current scope, the Engine looks into the next outer containing scope.
26. Dynamic Scope [run-time] - Dynamic Scope looks up the call stack (to check for the function through which it was called) in contrast to the lexical scope (which checks the next scope bubble) which looks up the nested scope chain to find the correct variable reference.

https://github1s.com/webber2408/Es6-Samples

```
function foo() {
        console.log( a ); // 3  (not 2!)
}

function bar() {
        var a = 3;
        foo();
}

var a = 2;

bar();
```

The Key Contrast Between Lexical and Dynamic Scoping: Lexical scope is write-time, whereas dynamic scope is run-time. Lexical scope care where a function was declared, but dynamic scope cares where a function was called from.

27. Garbage Collection
    Explain 20
28. Shadow Variables - The same variable declared at multiple nested scopes is called shadowing (i.e. the inner identifier shadows the outer identifier).
    Note: If want to access a global shadow variable, can use the window.a
    If the variable is not a global variable then it can't access it. (Non-global shadow variables cannot be accessed.)
    https://www.geeksforgeeks.org/variable-shadowing-in-javascript/
    Explain 19(Imp)
29. What can we do to avoid scope collision?
    a. Try hiding variables in the scope
    b. Use global namespaces
    c. Using modules
30. Practice
    https://drive.google.com/file/d/16ZJHzNgDKT59rMWYF4Vw0srav7uvdE2M/view?usp=sharing
    Solution: https://repl.it/@webber2408/BasicPracticeSolution#index.js
31. Closures: Closures happen as a result of writing code that relies on the lexical scope. The closure is when a function is able to remember and access its lexical scope even when that function is executing outside its lexical scope.
    Explain 10 10.1 10.2 (Great Example)
32. Modules (A way to implement closures!)
    Explain 15 15.1 15.2(Imp) 15.3 15.4_bar 15.4_foo

https://github1s.com/webber2408/Es6-Samples

33. Strict Mode: (ES5 added a “strict mode” to the language, which tightens the rules for certain behaviors. Generally, these restrictions are seen as keeping the code to a safer and more appropriate set of guidelines. Also, adhering to strict mode makes your code generally more optimizable by the engine.)
    Explain 14
34. this in JS
    Explain 16
35. Prototype
    Explain 17
36. Polyfilling vs Transpiling
    Explain 18
37. Reduce Polyfill

```
const arr = [1, 2, 3];

var result = arr.reduce((acc, itr) => {
  return acc+=itr;
}, 0);

console.log(result);

Array.prototype.reduce = function(foo, initValue){
  let self = this;
  let result = initValue;

  this.forEach(item => {
    result = foo(result, item);
  });

  return result;
}

var result = arr.reduce((acc, itr) => {
  return acc+=itr;
}, 0);

console.log(result);
```

https://github1s.com/webber2408/Es6-Samples

Object-Oriented JavaScript

1. Object [A collection of related data or functionality]
2. Dot Notation & Bracket Notation
3. Objects are sometimes called **associative arrays** — they map strings to values in the same way that arrays map numbers to values.
4. **this** keyword refers to the current object the code is being written inside.
   Why do we need `this`?
   Because it ensures that the correct values are used when a member's context changes, there can be two different object instances and we want to access their own methods and not of others.
5. Object names are called **namespaces**.
6. **abstraction** — creating a simple model of a more complex thing, which represents its most important aspects in a way that is easy to work with for our program's purposes.
7. **Object instances** - Objects that contain the data and functionality defined in the class.
   Instantiation - creating an object instance from a class.
8. **Polymorphism** - The ability of multiple object types to implement the same functionality is called polymorphism.
9. **Constructors -** special functions to define and initialize objects and their features.

https://github1s.com/webber2408/Es6-Samples

JavaScript Revision:

1. **Go through JS Masters.**

2. **Read YDKJS - Objects.**

3. **Read Mocha / Chai Blog** - https://javascript.info/testing-mocha, https://www.chaijs.com/api/assert/#method_equal

4. **Call(), Bind(), Apply(). - YDKJS Ch:2 Pg-18,19**

   > **Note:** This function is almost identical to `apply()`, except that `call()` accepts an **argument list**, while `apply()` accepts a **single array of arguments** — for example, `func.apply(this, ['eat', 'bananas'])` vs. `func.call(this, 'eat', 'bananas')`.

   The `call()` allows for a function/method belonging to one object to be assigned and called for a different object.

   Call() - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/call
   Apply() - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/apply

https://github1s.com/webber2408/Es6-Samples

To answer the part about when to use each function, use `apply` if you don't know the number of arguments you will be passing, or if they are already in an array or array-like object (like the `arguments` object to forward your own arguments. Use `call` otherwise, since there's no need to wrap the arguments in an array.

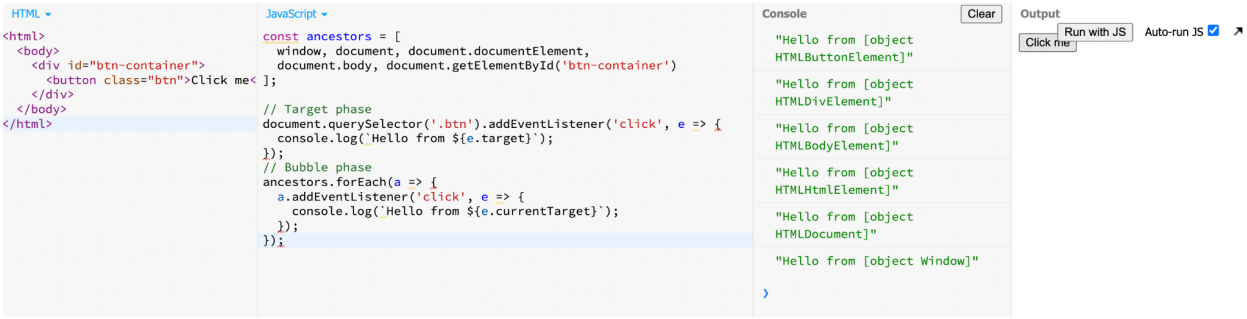
```
f.call(thisObject, a, b, c); // Fixed number of arguments

f.apply(thisObject, arguments); // Forward this function's arguments

var args = [];
while (...) {
    args.push(some_value());
}
f.apply(thisObject, args); // Unknown number of arguments
```

When I'm not passing any arguments (like your example), I prefer `call` since I'm *calling* the function. `apply` would imply you are *applying* the function to the (non-existent) arguments.

There shouldn't be any performance differences, except maybe if you use `apply` and wrap the arguments in an array (e.g. `f.apply(thisObject, [a, b, c])` instead of `f.call(thisObject, a, b, c)`). I haven't tested it, so there could be differences, but it would be very browser specific. It's likely that `call` is faster if you don't already have the arguments in an array and `apply` is faster if you do.

Bind() - The bind() method creates a new function that, when called, has its keyword set to the provided value, with a given sequence of arguments preceding any provided when the new function is called. *Advantage? Can pre-set some of the arguments of the target function.*

5. **Event Delegation / Propagation / Bubbling / Capturing.**
   https://www.30secondsofcode.org/articles/s/javascript-event-bubbling-capturing-delegation
   https://jsbin.com/kaqagederi/2/edit?html,js,console,output
   Most events bubble but some like “focus” do not!
   (https://codepen.io/cferdinandi/pen/pqJZdK?editors=1111)
   https://gomakethings.com/whats-the-difference-between-javascript-event-delegation-bubbling-and-capturing/#:~:text=tl%3Bdr%3A%20event%20delegation%20is,events%20that%20don't%20bubble.



https://github1s.com/webber2408/Es6-Samples

The event propagation mode determines in which order the elements receive the event. ***With bubbling, the event is first captured and handled by the innermost element and then propagated to outer elements****. With capturing, the event is first captured by the outermost element and propagated to the inner elements.*

6. Throttling and Debouncing.

7. **Difference between Map() and WeakMap()**
   https://www.geeksforgeeks.org/what-is-the-difference-between-map-and-weakmap-in-javascript/

Map	WeakMap
A Map is an unordered list of key-value pairs where the key and the value can be of any type like string, boolean, number, etc.	In a Weak Map, every key can only be an object and function. It used to store weak object references.
Maps are iterable.	WeakMaps are not iterable.
Maps will keep everything even if you don't use them.	WeakMaps holds the reference to the key, not the key itself.
The garbage collector doesn't remove a key pointer from "Map" and also doesn't remove the key from memory.	The garbage collector goes ahead and removes the key pointer from "WeakMap" and also removes the key from memory. WeakMap allows the garbage collector to do its task but not the Map.
Maps have some properties : .set, .get, .delete, .size, .has, .forEach, Iterators.	WeakMaps have some properties : .set, .get, .delete, .has.
You can create a new map by using a **new Map()**.	You can create a new WeakMap by using a **new WeakMap()**.

8. Reduce Polyfill.
9. Event Loop.
10. How DOM Works - Google Keep.
11. Currying in JS.
12. Latest ECMAScript Developments.
13. Revise OOPS Concepts.
14. What is TDD vs BDD?
15. JS Design Patterns.

https://github1s.com/webber2408/Es6-Samples

16. Vanilla JS Handling forms.
17. Revise CSS.