1. What happens when you use *typeof* with *let* and *const*. How does it change the way we added validation using *typeof*?  
   Give 2 coding examples.

**Answer**: typeof is not safe to use for let and const as it gives error.

function testType () {  
 typeof a;  
 let a = ‘testing’;  
}

testType(); // Uncaught ReferenceError: a is not defined at testType (<anonymous>:2:3) at <anonymous>:5:1

Same for const key word as well.

Before ECMAScript 2015, typeof was always guaranteed to return a string for any operand it was supplied with. Even with undeclared identifiers, typeof will return 'undefined'. Using typeof could never generate an error.

But with the addition of block-scoped let and const, using typeof on let and const variables (or using typeof on a class) in a block before they are declared will throw a ReferenceError. Block scoped variables are in a "temporal dead zone" from the start of the block until the initialization is processed, during which, it will throw an error if accessed.

1. Is *var* dead? Should we never use *var* as a good practice? How should we decide where to use *var*, *let* and *const*.

No var is not dead. Yes let and const improves scope related things but when we need global variable we can use var.The scope of a variable defined with var is function scope or declared outside any function, global.The scope of a variable defined with let is block scope.  
Examples.  
function varAndLetTest(){  
 {  
 var varMsg = "var test";  
 }  
 console.log(varMsg);  
 {   
let letMsg = "let test";  
 }  
 console.log(letMsg);   
} varAndLetTest();

19:57:05.470VM278:5 var test

19:57:05.491

VM278:11 Uncaught ReferenceError: letMsg is not defined at varAndLetTest (<anonymous>:11:14) at <anonymous>:13:1

1. Understand Lexical variables. *this* keyword and how arrow function affects *this* of a function. Give 2 coding examples.

Answer: Lexical variables are those variables that can only referenced at the textual location of the code that creates it. And they exist as long as some form reference to it even if the construct that created the variable exited

In classic function expressions, the this keyword is bound to different values based on the context in which it is called. With arrow functions however, this is lexically bound. It means that it uses this from the code that contains the arrow function.

======= this with Regular function ===========================

var countup = {

    counter: 0,

    start:function(){

        setInterval(function(){

            this.counter++; // \*\*INCORRECT\*\*- doesn't increment countup's counter property

        }, 1000);

    }

};

countup.start();

============= end ========================================================================

============================This with work around in regular function=======================

var countup = {

    counter: 0,

    start:function(){

        var countup = this; // cache reference to the countup object

        setInterval(function(){

            countup.counter++; // NOW CORRECT- increments countup's counter property

        }, 1000);

    }

};

countup.start();

=============================end===============================

=======================this with arrow function================================================

var countup = {

    counter: 0,

    start:function(){

        setInterval( () => {

            this.counter++; // Increments countup's counter property

        }, 1000);

    }

};

countup.start();

===========================end===================================================================

1. List down scenarios where we cannot use arrow function over normal function.  
   Give 2 examples each of 4 possibilities discussed. Mention atleast one additional possibility.

Answer: Below are downsides:

In methods on an object:

class Counter {  
 counter = 0;  
 handleClick = () => {  
 this.counter++;  
 }  
}

In this way, even if handleClick were called with by an event handler rather than in the context of an instance of Counter, it would still have access to the instance's data.

Deep Callchains:

Another place where arrow functions can get you in trouble is when they are going to be used in many different combinations, particularly in deep chains of function calls.

The core reason is the same as with anonymous functions: they give really bad stacktraces.

### Functions With Dynamic Context:

### If “this” is bound dynamically.If you use arrow functions in these locations, that dynamic binding will not work, and you (or someone else working with your code later) may get very confused as to why things aren't working as expected.

1. How does *rest parameters* and *default parameters* completely removes the use of arguments object?  
   Give 2 coding examples.

Rest parameters are a good candidate to replace arguments object as they can be used for similar requirements.

function sumAll(...args) { // args is the name for the array  
 let sum = 0;  
 for (let arg of args) sum += arg;  
 return sum;  
}

alert( sumAll(1) ); // 1  
alert( sumAll(1, 2) ); // 3  
alert( sumAll(1, 2, 3) ); // 6

Also below are some downsides of arguments object which can work well with rest parameters.

Although arguments is both array-like and iterable, it’s not an array. It does not support array methods, so we can’t call arguments.map(...) for example.

**Arrow functions do not have "arguments"**

If we access the arguments object from an arrow function, it takes them from the outer “normal” function.

Here’s an example:

function f() {  
 let showArg = () => alert(arguments[0]);  
 showArg();  
}  
f(1); // 1

Also, it always contains all arguments. We can’t capture them partially, like we did with rest parameters.

So when we need these features, then rest parameters are preferred.

1. Explain *rest parameter* v/s *apply()* v/s *arguments*.   
   Give 2 coding examples.

Rest parameters vs argument:

**// Below is the example of arguments**

function showName() {

alert( arguments.length );

alert( arguments[0] );

alert( arguments[1] );

// it's iterable

// for(let arg of arguments) alert(arg);

}

// shows: 2, Julius, Caesar

showName("Julius", "Caesar");

// shows: 1, Ilya, undefined (no second argument)

showName("Ilya")

**//Below is the example of rest parameters**

function showName(firstName, lastName, ...titles) {

alert( firstName + ' ' + lastName ); // Julius Caesar

// the rest go into titles array

// i.e. titles = ["Consul", "Imperator"]

alert( titles[0] ); // Consul

alert( titles[1] ); // Imperator

alert( titles.length ); // 2

}

showName("Julius", "Caesar", "Consul", "Imperator");

1. Can we use *de-structuring* with objects?

Answer: Yes we can use de-structuring with objects.

1. What are iterators in ES6? What is the relation between iterators and for-of loop.

Iterator is described by a structure that contains a pointer to the next element in the iteration

An iterator must be an object with a function named next that returns an object with the keys: value — the current item in the iteration; and done— true if the iteration has finished, false otherwise.

const iterator = {

next() {

value: any,

done: boolean

}

}

1. Practical assignments:
   1. **ex\_blockScope.js**: Play around and change *var* with *let* or *const* wherever applicable. Make sure that the test case at the bottom return *true*.
   2. **ex\_arrowFunctions.js**: You’ll notice a lot of functions in this file. Replace them with best implementation of *arrow functions* wherever applicable.  
      Remember, this file returns *true*. So when you make changes, your code should return *true*.
   3. **ex\_restSpreadDefault.js**: Use *Rest, Spread and Default parameters* to write the foo function such that the console.log returns “281012” string.
   4. **ex\_destructuring.js**: Here we have is a check function that passes in an object but that response function is receiving an object. So here we’ll need to do some *object destructuring* to get the desired output.
   5. **ex\_templateString.js**: In this exercise, modify the line 8 and add interpolation expressions in place of the underscores and then add code to the “upper” function such that the console.log returns true.