1. Solve the Code

```
function sayHi() {
  console.log(name);
  console.log(age);
  var name = 'Hitesh';
  let age = 21;
}

sayHi();
```

Answer:- undefined and ReferenceError

Within the function, we first declare the name variable with the var keyword. This means that the variable gets hoisted (memory space is set up during the creation phase) with the default value of undefined, until we actually get to the line where we define the variable. We haven't defined the variable yet on the line where we try to log the name variable, so it still holds the value of undefined.
Variables with the let keyword (and const) are hoisted, but unlike var, don't get initialized. They are not accessible before the line we declare (initialize) them. This is called the "temporal dead zone". When we try to access the variables before they are declared, JavaScript throws a Reference Error.

2. Solve the Code

```
for (var i = 0; i < 3; i++) {
  setTimeout(() => console.log(i), 1);
}

for (let i = 0; i < 3; i++) {
  setTimeout(() => console.log(i), 1);
}
```

Answer:- 3 3 3 and 0 1 2

Because of the event queue in JavaScript, the setTimeout callback function is called *after* the loop has been executed. Since the variable i in the first loop was declared using the var keyword, this value was global. During the loop, we incremented the value of i by 1 each time, using the unary operator ++. By the time the setTimeout callback function was invoked, i was equal to 3 in the first example. In the second loop, the variable i was declared using the let keyword: variables declared with the let (and const) keyword are block-scoped (a block is anything between { }). During each iteration, i will have a new value, and each value is scoped inside the loop.

3. Solve the Code

```
const shape = {
  radius: 10,
  diameter() {
    return this.radius * 2;
  },
  perimeter: () => 2 * Math.PI * this.radius,
};

console.log(shape.diameter());
console.log(shape.perimeter());
```

Answer: 20 and NaN

Note that the value of diameter is a regular function, whereas the value of perimeter is an arrow function.

With arrow functions, the this keyword refers to its current surrounding scope, unlike regular functions! This means that when we call perimeter, it doesn't refer to the shape object, but to its surrounding scope (window for example).
There is no value radius on that object, which returns NaN.

4. Solve the Code

```
+true;
!'Lydia';
```

Answer: 1 and false
The unary plus tries to convert an operand to a number. true is 1, and false is 0.
The string 'Lydia' is a truthy value. What we're actually asking, is "is this truthy value falsy?". This returns false.

5. Which one is true

```
const bird = {
  size: 'small',
};

const mouse = {
  name: 'Mickey',
  small: true,
};
A: mouse.bird.size is not valid
B: mouse[bird.size] is not valid
C: mouse[bird["size"]] is not valid
D: All of them are valid
```

Answer: mouse.bird.size is not valid
In JavaScript, all object keys are strings (unless it's a Symbol). Even though we might not *type* them as strings, they are always converted into strings under the hood.

JavaScript interprets (or unboxes) statements. When we use bracket notation, it sees the first opening bracket [and keeps going until it finds the closing bracket]. Only then, it will evaluate the statement.
mouse[bird.size]: First it evaluates bird.size, which
is "small". mouse["small"] returns true

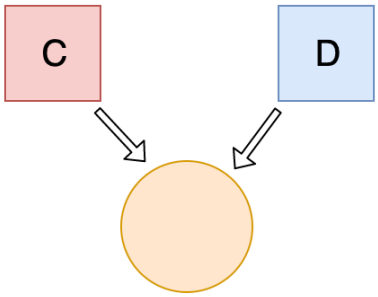
However, with dot notation, this doesn't happen. mouse does not have a key called bird, which means that mouse.bird is undefined. Then, we ask for the size using dot notation: mouse.bird.size. Since mouse.bird is undefined, we're actually asking undefined.size. This isn't valid, and will throw an error similar to Cannot read property "size" of undefined.

6. Solve the Code

```
let c = { greeting: 'Hey!' };
let d;

d = c;
c.greeting = 'Hello';
console.log(d.greeting);
```

Answer: Hello
In JavaScript, all objects interact by reference when setting them equal to each other.

First, variable c holds a value to an object. Later, we assign d with the same reference that c has to the object.



When you change one object, you change all of them.

7. Solve the Code

```
let a = 3;
let b = new Number(3);
let c = 3;

console.log(a == b);
console.log(a === b);
console.log(b === c);
```

Answer: true false false
new Number() is a built-in function constructor. Although it looks like a number, it's not really a number: it has a bunch of extra features and is an object.
When we use the == operator, it only checks whether it has the same *value*. They both have the value of 3, so it returns true.
However, when we use the === operator, both value *and* type should be the same. It's not: new Number() is not a number, it's an **object**. Both return false.

8. Solve the Code

```
class Chameleon {
  static colorChange(newColor) {
    this.newColor = newColor;
    return this.newColor;
  }

  constructor({ newColor = 'green' } = {}) {
    this.newColor = newColor;
  }
}

const freddie = new Chameleon({ newColor: 'purple' });
console.log(freddie.colorChange('orange'));
```

Answer: TypeError
The colorChange function is static. Static methods are designed to live only on the constructor in which they are created, and cannot be passed down to any children or called upon class instances. Since freddie is an instance of class Chameleon, the function cannot be called upon it. A TypeError is thrown.

9. Solve the Code

```
let greeting;
greeting = {}; // Typo!
console.log(greeting);
```

Answer: {}
It logs the object, because we just created an empty object on the global object! When we mistyped greeting as greetign, the JS interpreter actually saw this as global.greetign = {} (or window.greetign = {} in a browser).
In order to avoid this, we can use "use strict". This makes sure that you have declared a variable before setting it equal to anything.

10. What will happen if you run this code

```
function bark() {
  console.log('Wolf!');
}

bark.animal = 'dog';
```

Answer: Nothing this is totally fine
This is possible in JavaScript, because functions are objects! (Everything besides primitive types are objects)

A function is a special type of object. The code you write yourself isn't the actual function. The function is an object with properties. This property is invocable.

11. Solve the Code

```
function Person(firstName, lastName) {
  this.firstName = firstName;
  this.lastName = lastName;
}

const member = new Person('Hitesh', 'Anurag');
Person.getFullName = function() {
  return `${this.firstName} ${this.lastName}`;
};

console.log(member.getFullName());
```

Answer: TypeError
In JavaScript, functions are objects, and therefore, the method getFullName gets added to the constructor function object itself. For that reason, we can call Person.getFullName(), but member.getFullName throws a TypeError.
If you want a method to be available to all object instances, you have to add it to the prototype property:

```
Person.prototype.getFullName = function() {
  return `${this.firstName} ${this.lastName}`;
};
```

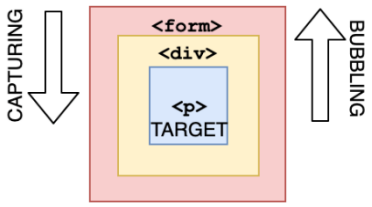
12. Solve the Code

```
function Person(firstName, lastName) {
  this.firstName = firstName;
  this.lastName = lastName;
}

const hitesh = new Person('Hitesh', 'Anurag');
const anurag = Person('Prasad', 'iNeuron');

console.log(hitesh);
console.log(anurag);
```

Answer: Person {firstName: "Hitesh", lastName: "Anurag"} and undefined
For anurag, we didn't use the new keyword. When using new, this refers to the new empty object we create. However, if you don't add new, this refers to the **global object**!
We said that this.firstName equals "Prasad" and this.lastName equals "iNeuron". What we actually did, is defining global.firstName = Prasad and global.lastName = 'Smith'. iNeuron itself is left undefined, since we don't return a value from the Person function.

13. What are the three phases of event propagation?

Answer: Capturing > Target > Bubbling
During the **capturing** phase, the event goes through the ancestor elements down to the target element. It then reaches the **target** element, and **bubbling** begins.



14. All object have prototypes.

Answer: false
All objects have prototypes, except for the **base object**. The base object is the object created by the user, or an object that is created using the new keyword. The base object has access to some methods and properties, such as .toString. This is the reason why you can use built-in JavaScript methods! All of such methods are available on the prototype. Although JavaScript can't find it directly on your object, it goes down the prototype chain and finds it there, which makes it accessible for you.

15. Solve the Code

```
1  function sum(a, b) {
2    return a + b;
3  }
4
5  sum(1, '2');
```

Answer: “12”
JavaScript is a **dynamically typed language**: we don't specify what types certain variables are. Values can automatically be converted into another type without you knowing, which is called *implicit type coercion*. **Coercion** is converting from one type into another.

In this example, JavaScript converts the number 1 into a string, in order for the function to make sense and return a value. During the addition of a numeric type (1) and a string type ('2'), the number is treated as a string. We can concatenate strings like "Hello" + "World", so what's happening here is "1" + "2" which returns "12".

16. Solve the Code

```
let number = 0;
console.log(number++);
console.log(++number);
console.log(number);
```

Answer: 0 2 2
The postfix unary operator ++:

1. Returns the value (this returns 0)
2. Increments the value (number is now 1)

The prefix unary operator ++:

1. Increments the value (number is now 2)
2. Returns the value (this returns 2)

This returns 0 2 2.

17. Solve the Code

```
function getPersonInfo(one, two, three) {
  console.log(one);
  console.log(two);
  console.log(three);
}

const person = 'Hitesh';
const age = 21;

getPersonInfo`${person} is ${age} years old`;
```

Answer: ["", " is ", " years old"] "Hitesh" 21
If you use tagged template literals, the value of the first argument is always an array of the string values. The remaining arguments get the values of the passed expressions!

18. Solve the Code

```
function checkAge(data) {
  if (data === { age: 18 }) {
    console.log('You are an adult!');
  } else if (data == { age: 18 }) {
    console.log('You are still an adult.');
  } else {
    console.log(`Hmm.. You don't have an age I guess`);
  }
}

checkAge({ age: 18 });
```

Answer: Hmm.. You don't have an age I guess
When testing equality, primitives are compared by their *value*, while objects are compared by their *reference*. JavaScript checks if the objects have a reference to the same location in memory.

The two objects that we are comparing don't have that: the object we passed as a parameter refers to a different location in memory than the object we used in order to check equality.

This is why both { age: 18 } === { age: 18 } and { age: 18 } == { age: 18 } return false.

19. Solve the Code

```
function getAge(...args) {
  console.log(typeof args);
}

getAge(21);
```

Answer: “Object”

The rest parameter (...args) lets us "collect" all remaining arguments into an array. An array is an object, so typeof args returns "object"

20. Solve the Code

```
function getAge() {
  'use strict';
  age = 21;
  console.log(age);
}

getAge();
```

Answer: ReferenceError
With "use strict", you can make sure that you don't accidentally declare global variables. We never declared the variable age, and since we use "use strict", it will throw a reference error. If we didn't use "use strict", it would have worked, since the property age would have gotten added to the global object.

21. What will be the value of sum

```
const sum = eval('10*10+5');
```

Answer: 105
eval evaluates codes that's passed as a string. If it's an expression, like in this case, it evaluates the expression. The expression is 10 * 10 + 5. This returns the number 105.

22. How long is cool secret accessible?

```
sessionStorage.setItem('cool_secret', 123);
```

Answer: When the user closes the tab.
The data stored in sessionStorage is removed after closing the *tab*.
If you used localStorage, the data would've been there forever, unless for example localStorage.clear() is invoked.

23. Solve the Code

```
var num = 8;
var num = 10;

console.log(num);
```

Answer: 10
With the var keyword, you can declare multiple variables with the same name. The variable will then hold the latest value.
You cannot do this with let or const since they're block-scoped.

24. Solve the Code

```
const obj = { 1: 'a', 2: 'b', 3: 'c' };
const set = new Set([1, 2, 3, 4, 5]);

obj.hasOwnProperty('1');
obj.hasOwnProperty(1);
set.has('1');
set.has(1);
```

Answer: true true false true
All object keys (excluding Symbols) are strings under the hood, even if you don't type it yourself as a string. This is why obj.hasOwnProperty('1') also returns true.
It doesn't work that way for a set. There is no '1' in our set: set.has('1') returns false. It has the numeric type 1, set.has(1) returns true.

25. Solve the Code

```
const obj = { a: 'one', b: 'two', a: 'three' };
console.log(obj);
```

Answer: { a: "three", b: "two" }
If you have two keys with the same name, the key will be replaced. It will still be in its first position, but with the last specified value.

26. True or False

The JavaScript global execution context creates two things for you: the global object, and the "this" keyword.

Answer: true

The base execution context is the global execution context: it's what's accessible everywhere in your code.

27. Solve the Code

```
for (let i = 1; i < 5; i++) {
  if (i === 3) continue;
  console.log(i);
}
```

Answer: 1 2 4

The continue statement skips an iteration if a certain condition returns true.

28. Solve the Code

```
String.prototype.giveHiteshPizza = () => {
  return 'Just give Hitesh pizza already!';
};

const name = 'Hitesh';

console.log(name.giveLydiaPizza())
```

Answer: "Just give Hitesh pizza already!"

String is a built-in constructor, which we can add properties to. I just added a method to its prototype. Primitive strings are automatically converted into a string object, generated by the string prototype function. So, all strings (string objects) have access to that method!

29. Solve the Code

```
const a = {};
const b = { key: 'b' };
const c = { key: 'c' };

a[b] = 123;
a[c] = 456;

console.log(a[b]);
```

Answer: 456

Object keys are automatically converted into strings. We are trying to set an object as a key to object a, with the value of 123.

However, when we stringify an object, it becomes "[object Object]". So what we are saying here, is that a["[object Object]"] = 123. Then, we can try to do the same again. c is another object that we are implicitly stringifying. So then, a["[object Object]"] = 456.

Then, we log a[b], which is actually a["[object Object]"]. We just set that to 456, so it returns 456.

30. Solve the Code

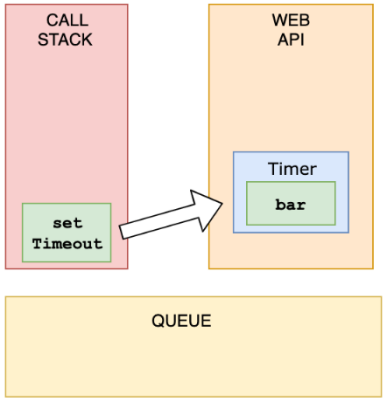
```
const foo = () => console.log('First');
const bar = () => setTimeout(() => console.log('Second'));
const baz = () => console.log('Third');

bar();
foo();
baz();
```

Answer: First Third Second

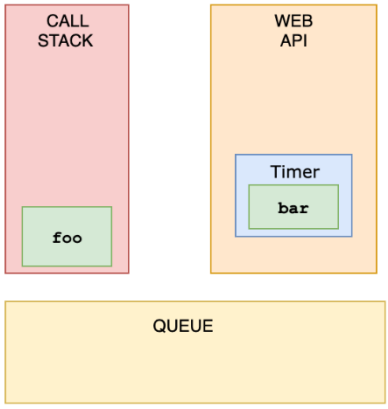
We have a setTimeout function and invoked it first. Yet, it was logged last.

This is because in browsers, we don't just have the runtime engine, we also have something called a WebAPI. The WebAPI gives us the setTimeout function to start with, and for example the DOM.

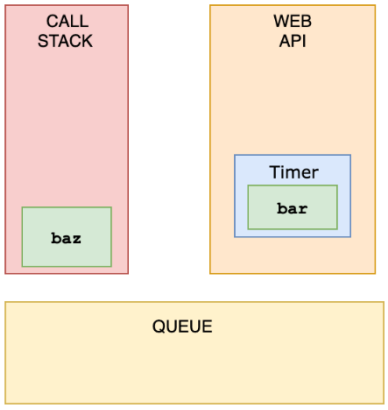
After the callback is pushed to the WebAPI, the setTimeout function itself (but not the callback!) is popped off the stack.



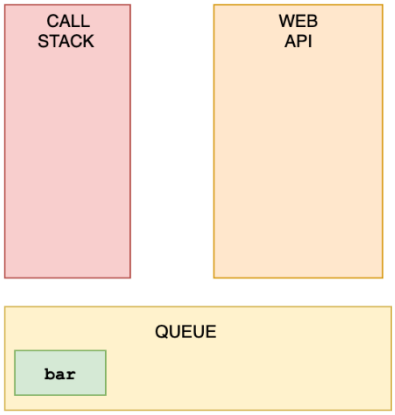
Now, foo gets invoked, and "First" is being logged.



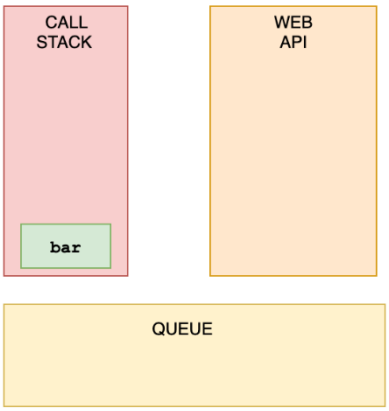
foo is popped off the stack, and baz gets invoked. "Third" gets logged.



The WebAPI can't just add stuff to the stack whenever it's ready. Instead, it pushes the callback function to something called the queue.



This is where an event loop starts to work. An event loop looks at the stack and task queue. If the stack is empty, it takes the first thing on the queue and pushes it onto the stack.



bar gets invoked, "Second" gets logged, and it's popped off the stack.

31. What is the event.target when clicking the button?

```
<div onclick="console.log('first div')">
  <div onclick="console.log('second div')">
    <button onclick="console.log('button')">
      Click!
    </button>
  </div>
</div>
```

Answer: button

The deepest nested element that caused the event is the target of the event. You can stop bubbling by event.stopPropagation

32. When you click the paragraph, what's the logged output?

```
<div onclick="console.log('div')">
  <p onclick="console.log('p')">
    Click here!
  </p>
</div>
```

Answer: p div
If we click p, we see two logs: p and div. During event propagation, there are 3 phases: capturing, target, and bubbling. By default, event handlers are executed in the bubbling phase (unless you set useCapture to true). It goes from the deepest nested element outwards.

33. Solve the Code

```
1  const person = { name: 'Hitesh' };
2
3  function sayHi(age) {
4    return `${this.name} is ${age}`;
5  }
6
7  console.log(sayHi.call(person, 21));
8  console.log(sayHi.bind(person, 21));
```

Answer: Hitesh is 21 function
With both, we can pass the object to which we want the this keyword to refer to. However, .call is also *executed immediately*!
.bind. returns a *copy* of the function, but with a bound context! It is not executed immediately.

34. Solve the Code

```
function sayHi() {
  return (() => 0)();
}

console.log(typeof sayHi());
```

Answer: “number”
The sayHi function returns the returned value of the immediately invoked function expression (IIFE). This function returned 0, which is type "number".
FYI: there are only 7 built-in types: null, undefined, boolean, number, string, object, and symbol. "function" is not a type, since functions are objects, it's of type "object".

35. Which of these values are falsy?

```
0;
new Number(0);
('');
(' ');
new Boolean(false);
undefined;
```

Answer: 0, '', undefined
There are 8 falsy values:

- undefined
- null
- NaN
- false
- '' (empty string)
- 0
- -0
- 0n (BigInt(0))

Function constructors, like new Number and new Boolean are truthy.

36.Solve the Code

```
console.log(typeof typeof 1);
```

Answer: string
typeof 1 returns "number". typeof "number" returns "string"

37.Solve the Code

```
const numbers = [1, 2, 3];
numbers[10] = 11;
console.log(numbers);
```

Answer: [1, 2, 3, empty x 7, 11]
When you set a value to an element in an array that exceeds the length of the array, JavaScript creates something called "empty slots". These actually have the value of undefined, but you will see something like:
[1, 2, 3, empty x 7, 11]
depending on where you run it (it's different for every browser, node, etc.)

38.Solve the Code

```
(() => {
  let x, y;
  try {
    throw new Error();
  } catch (x) {
    (x = 1), (y = 2);
    console.log(x);
  }
  console.log(x);
  console.log(y);
})();
```

Answer: 1 undefined 2

The catch block receives the argument x. This is not the same x as the variable when we pass arguments. This variable x is block-scoped.
Later, we set this block-scoped variable equal to 1, and set the value of the variable y.
Now, we log the block-scoped variable x, which is equal to 1.
Outside of the catch block, x is still undefined, and y is 2. When we want to console.log(x) outside of the catch block, it returns undefined, and y returns 2.

39. Everything in JavaScript is either a...

a) primitive or object
b) function or object
c) trick question! only objects
d) number or object

Answer: primitive or object
JavaScript only has primitive types and objects.
Primitive types are boolean, null, undefined, bigint, number, string, and symbol.
What differentiates a primitive from an object is that primitives do not have any properties or methods; however, you'll note that 'foo'.toUpperCase() evaluates to 'FOO' and does not result in a TypeError. This is because when you try to access a property or method on a primitive like a string, JavaScript will implicitly wrap the primitive type using one of the wrapper classes, i.e. String, and then immediately discard the wrapper after the expression evaluates. All primitives except for null and undefined exhibit this behaviour.

40. Solve the Code

Answer: [1, 2, 0, 1, 2, 3]
[1, 2] is our initial value. This is the value we start with, and the value of the very first acc. During the first round, acc is [1,2], and cur is [0, 1]. We concatenate them, which results in [1, 2, 0, 1].
Then, [1, 2, 0, 1] is acc and [2, 3] is cur. We concatenate them, and get [1, 2, 0, 1, 2, 3]

41. Solve the Code

```
1  !!null;
2  !!'';
3  !!1;
```

Answer: false false true
null is falsy. !null returns true. !true returns false.
"" is falsy. !"" returns true. !true returns false.
1 is truthy. !1 returns false. !false returns true.

42. What does the setInterval method return in the browser?

```
setInterval(() => console.log('Hi'), 1000);
```

Answer: a unique id
It returns a unique id. This id can be used to clear that interval with the clearInterval() function.

43. What does this return?

```
[...'Hitesh'];
```

Answer: [“H”, “i”, “t”, “e”, “s”, “h”]
A string is an iterable. The spread operator maps every character of an iterable to one element.

44. Solve the Code

```
function* generator(i) {
  yield i;
  yield i * 2;
}

const gen = generator(10);

console.log(gen.next().value);
console.log(gen.next().value);
```

Answer: 10 20
Regular functions cannot be stopped mid-way after invocation. However, a generator function can be "stopped" midway, and later continue from where it stopped. Every time a generator function encounters a yield keyword, the function yields the value specified after it. Note that the generator function in that case doesn't *return* the value, it *yields* the value.

First, we initialize the generator function with i equal to 10. We invoke the generator function using the next() method. The first time we invoke the generator function, i is equal to 10. It encounters the first yield keyword: it yields the value of i. The generator is now "paused", and 10 gets logged.
Then, we invoke the function again with the next() method. It starts to continue where it stopped previously, still with i equal to 10. Now, it encounters the next yield keyword, and yields i * 2. i is equal to 10, so it returns 10 * 2, which is 20. This results in 10, 20.

45. What does this return?

```
const firstPromise = new Promise((res, rej) => {
  setTimeout(res, 500, 'one');
});

const secondPromise = new Promise((res, rej) => {
  setTimeout(res, 100, 'two');
});

Promise.race([firstPromise, secondPromise]).then(res => console.log(res));
```

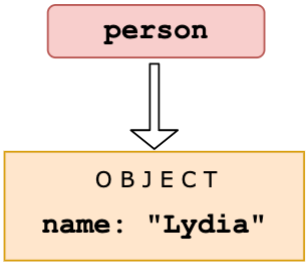
Answer: "two"
When we pass multiple promises to the Promise.race method, it resolves/rejects the *first* promise that resolves/rejects. To the setTimeout method, we pass a timer: 500ms for the first promise (firstPromise), and 100ms for the second promise (secondPromise). This means that the secondPromise resolves first with the value of 'two'. res now holds the value of 'two', which gets logged.

46. Solve the Code

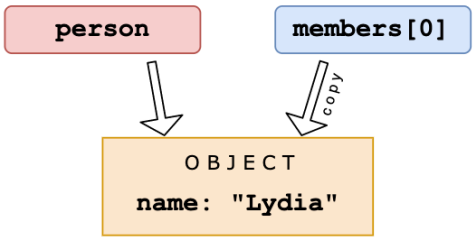
```
let person = { name: 'Hitesh' };
const members = [person];
person = null;

console.log(members);
```

Answer: [{ name: "Hitesh" }]

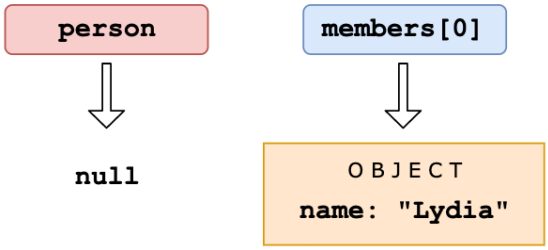
First, we declare a variable person with the value of an object that has a name property.



Then, we declare a variable called members. We set the first element of that array equal to the value of the person variable. Objects interact by reference when setting them equal to each other. When you assign a reference from one variable to another, you make a copy of that reference. (note that they don't have the same reference!)



Then, we set the variable person equal to null.



We are only modifying the value of the person variable, and not the first element in the array, since that element has a different (copied) reference to the object. The first element in members still holds its reference to the original object. When we log the members array, the first element still holds the value of the object, which gets logged.

47. Solve the Code

```
const person = {
  name: 'Hitesh',
  age: 21,
};

for (const item in person) {
  console.log(item);
}
```

Answer: "name", "age"
With a for-in loop, we can iterate through object keys, in this case name and age. Under the hood, object keys are strings (if they're not a Symbol). On every loop, we set the value of item equal to the current key it's iterating over. First, item is equal to name, and gets logged. Then, item is equal to age, which gets logged.

48. Solve the Code

```
console.log(3 + 4 + '5');
```

Answer: "75"
Operator associativity is the order in which the compiler evaluates the expressions, either left-to-right or right-to-left. This only happens if all operators have the *same* precedence. We only have one type of operator: +. For addition, the associativity is left-to-right.
3 + 4 gets evaluated first. This results in the number 7.
7 + '5' results in "75" because of coercion. JavaScript converts the number 7 into a string, see question 15. We can concatenate two strings using the +operator. "7" + "5" results in "75".

49. What's the value of num?

```
const num = parseInt('7*6', 10);
```

Answer: 7
Only the first numbers in the string is returned. Based on the *radix* (the second argument in order to specify what type of number we want to parse it to: base 10, hexadecimal, octal, binary, etc.), the parseInt checks whether the characters in the

string are valid. Once it encounters a character that isn't a valid number in the radix, it stops parsing and ignores the following characters.
* is not a valid number. It only parses "7" into the decimal 7. num now holds the value of 7.

50. Solve the Code

```
[1, 2, 3].map(num => {
  if (typeof num === 'number') return;
  return num * 2;
});
```

Answer: [undefined, undefined, undefined]
When mapping over the array, the value of num is equal to the element it's currently looping over. In this case, the elements are numbers, so the condition of the if statement typeof num === "number" returns true. The map function creates a new array and inserts the values returned from the function.
However, we don't return a value. When we don't return a value from the function, the function returns undefined. For every element in the array, the function block gets called, so for each element we return undefined.