JS SOLUTIONS

JS SOLUTIONS

1. What's the output?

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

sayHi();
```

- A: `Lydia` and `undefined`
- B: `Lydia` and `ReferenceError`
- C: `ReferenceError` and `21`
- D: `undefined` and `ReferenceError`

▼ **Answer**

Answer: D

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 `ReferenceError`.

---

2. What's the output?

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

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

- A: `0 1 2` and `0 1 2`

- B: `0 1 2` and `3 3 3`
- C: `3 3 3` and `0 1 2`

▼ **Answer**

Answer: C

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. What's the output?

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

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

- A: `20` and `62.83185307179586`
- B: `20` and `NaN`
- C: `20` and `63`
- D: `NaN` and `63`

▼ **Answer**

Answer: B

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).

Since there is no value `radius` in the scope of the arrow function, `this.radius` returns `undefined` which, when multiplied by `2 * Math.PI`, results in `NaN`.

---

4. What's the output?

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

- A: `1` and `false`
- B: `false` and `NaN`
- C: `false` and `false`

▼ **Answer**

Answer: A

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**

Answer: A

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. What's the output?

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

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

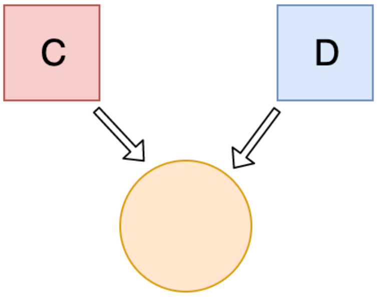
- A: `Hello`
- B: `Hey!`
- C: `undefined`
- D: `ReferenceError`
- E: `TypeError`

▼ **Answer**

Answer: A

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. What's the output?

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

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

- A: `true` `false` `true`
- B: `false` `false` `true`
- C: `true` `false` `false`
- D: `false` `true` `true`

▼ **Answer**

Answer: C

`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 (Equality 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 (Strict equality 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. What's the output?

```
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'));
```

- A: `orange`
- B: `purple`
- C: `green`
- D: `TypeError`

▼ **Answer**

Answer: D

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. What's the output?

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

- A: `{}`
- B: `ReferenceError: greetign is not defined`
- C: `undefined`

▼ **Answer**

Answer: A

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:

1. `global.greetign = {}` in Node.js
2. `window.greetign = {}`, `frames.greetign = {}` and `self.greetign` in browsers.
3. `self.greetign` in web workers.
4. `globalThis.greetign` in all environments.

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 happens when we do this?

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

bark.animal = 'dog';
```

- A: Nothing, this is totally fine!
- B: `SyntaxError`. You cannot add properties to a function this way.
- C: `"Woof"` gets logged.
- D: `ReferenceError`

▼ **Answer**

Answer: A

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. What's the output?

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

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

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

- A: `TypeError`
- B: `SyntaxError`
- C: `Lydia Hallie`
- D: `undefined` `undefined`

▼ **Answer**

Answer: A

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. What's the output?

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

const lydia = new Person('Lydia', 'Hallie');
const sarah = Person('Sarah', 'Smith');

console.log(lydia);
console.log(sarah);
```

- A: `Person {firstName: "Lydia", lastName: "Hallie"}` and `undefined`
- B: `Person {firstName: "Lydia", lastName: "Hallie"}` and `Person {firstName: "Sarah", lastName: "Smith"}`
- C: `Person {firstName: "Lydia", lastName: "Hallie"}` and `{}`
- D: `Person {firstName: "Lydia", lastName: "Hallie"}` and `ReferenceError`

▼ **Answer**

Answer: A

For `sarah`, 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 `"Sarah"` and `this.lastName` equals `"Smith"`. What we actually did, is defining `global.firstName = 'Sarah'` and `global.lastName = 'Smith'`. `sarah` itself is left `undefined`, since we don't return a value from the `Person` function.

---

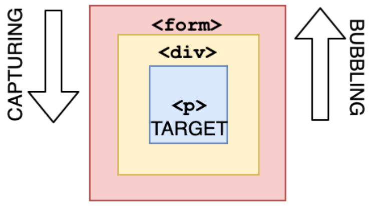
13. What are the three phases of event propagation?

- A: Target > Capturing > Bubbling
- B: Bubbling > Target > Capturing
- C: Target > Bubbling > Capturing
- D: Capturing > Target > Bubbling

▼ **Answer**

Answer: D

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.

- A: true
- B: false

▼ **Answer**

Answer: B

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. What's the output?

```javascript
function sum(a, b) {
  return a + b;
}

sum(1, '2');
```

- A: `NaN`
- B: `TypeError`
- C: `"12"`
- D: `3`

▼ **Answer**

Answer: C

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. What's the output?

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

- A: `1` `1` `2`
- B: `1` `2` `2`
- C: `0` `2` `2`
- D: `0` `1` `2`

▼ **Answer**

Answer: C

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. What's the output?

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

const person = 'Lydia';
const age = 21;

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

- A: `"Lydia"` `21` `["", " is ", " years old"]`
- B: `["", " is ", " years old"]` `"Lydia"` `21`
- C: `"Lydia"` `["", " is ", " years old"]` `21`

▼ **Answer**

Answer: B

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. What's the output?

```javascript
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 });
```

- A: `You are an adult!`
- B: `You are still an adult.`
- C: `Hmm.. You don't have an age I guess`

▼ **Answer**

Answer: C

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. What's the output?

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

getAge(21);
```

- A: `"number"`
- B: `"array"`
- C: `"object"`
- D: `"NaN"`

▼ **Answer**

Answer: C

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

20. What's the output?

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

getAge();
```

- A: `21`
- B: `undefined`
- C: `ReferenceError`
- D: `TypeError`

▼ **Answer**

Answer: C

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's the value of `sum`?

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

- A: `105`
- B: `"105"`
- C: `TypeError`
- D: `"10*10+5"`

▼ **Answer**

Answer: A

`eval` evaluates code 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);
```

- A: Forever, the data doesn't get lost.
- B: When the user closes the tab.
- C: When the user closes the entire browser, not only the tab.
- D: When the user shuts off their computer.

▼ **Answer**

Answer: B

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. What's the output?

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

console.log(num);
```

- A: `8`
- B: `10`
- C: `SyntaxError`
- D: `ReferenceError`

▼ **Answer**

Answer: B

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 and therefore can't be redeclared.

---

24. What's the output?

```javascript
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);
```

- A: `false` `true` `false` `true`

- B: `false` `true` `true` `true`
- C: `true` `true` `false` `true`
- D: `true` `true` `true` `true`

▼ **Answer**

Answer: C

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. What's the output?

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

- A: `{ a: "one", b: "two" }`
- B: `{ b: "two", a: "three" }`
- C: `{ a: "three", b: "two" }`
- D: `SyntaxError`

▼ **Answer**

Answer: C

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

- A: true
- B: false
- C: it depends

▼ **Answer**

Answer: A

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

---

27. What's the output?

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

- A: `1` `2`
- B: `1` `2` `3`
- C: `1` `2` `4`
- D: `1` `3` `4`

▼ **Answer**

Answer: C

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

---

28. What's the output?

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

const name = 'Lydia';

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

- A: `"Just give Lydia pizza already!"`
- B: `TypeError: not a function`
- C: `SyntaxError`
- D: `undefined`

▼ **Answer**

Answer: A

`String` is a built-in constructor, that 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. What's the output?

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

a[b] = 123;
```

```
a[c] = 456;

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

- A: `123`
- B: `456`
- C: `undefined`
- D: `ReferenceError`

▼ **Answer**

Answer: B

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. What's the output?

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

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

- A: `First` `Second` `Third`
- B: `First` `Third` `Second`
- C: `Second` `First` `Third`
- D: `Second` `Third` `First`

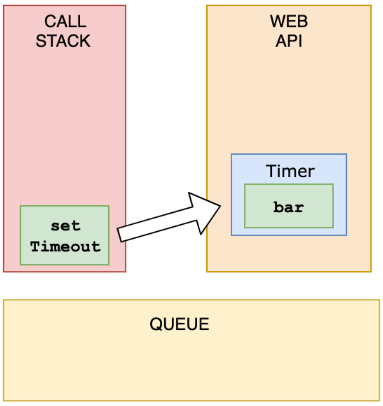
▼ **Answer**

Answer: B

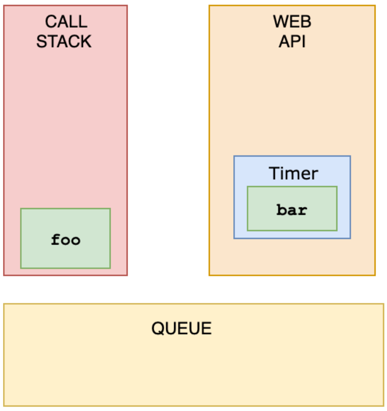
We have a `setTimeout` function and invoked it first. Yet, it was logged last.

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.

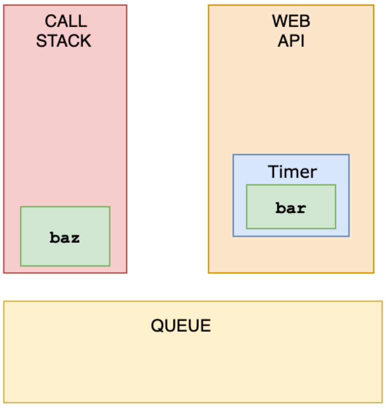
After the *callback* is pushed to the WebAPI, the `setTimeout` function itself (but not the callback!) is popped off the stack.



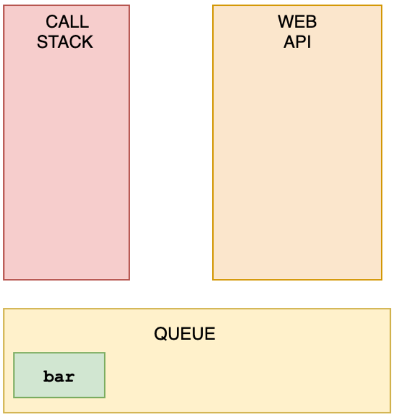
Now, `foo` gets invoked, and `"First"` is being logged.



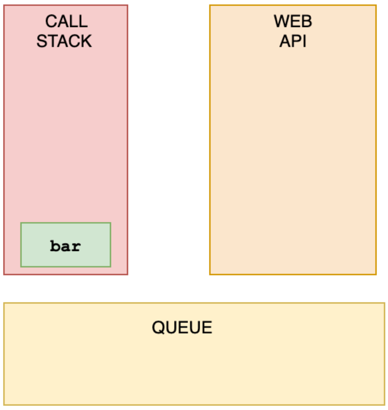
`foo` is popped off the stack, and `baz` gets invoked. `"Third"` gets logged.



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?

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

- A: Outer `div`
- B: Inner `div`
- C: `button`
- D: An array of all nested elements.

▼ **Answer**

Answer: C

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?

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

- A: `p` `div`
- B: `div` `p`
- C: `p`
- D: `div`

▼ **Answer**

Answer: A

If we click `p`, we see two logs: `p` and `div`. During event propagation, there are 3 phases: capturing, targeting, 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. What's the output?

```javascript
const person = { name: 'Lydia' };

function sayHi(age) {
  return `${this.name} is ${age}`;
}

console.log(sayHi.call(person, 21));
console.log(sayHi.bind(person, 21));
```

- A: `undefined is 21` `Lydia is 21`
- B: `function` `function`
- C: `Lydia is 21` `Lydia is 21`
- D: `Lydia is 21` `function`

▼ **Answer**

Answer: D

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. What's the output?

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

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

- A: `"object"`
- B: `"number"`
- C: `"function"`
- D: `"undefined"`

▼ **Answer**

Answer: B

The `sayHi` function returns the returned value of the immediately invoked function expression (IIFE). This function returned `0`, which is type `"number"`.

FYI: `typeof` can return the following list of values: `undefined`, `boolean`, `number`, `bigint`, `string`, `symbol`, `function` and `object`. Note that `typeof null` returns `"object"`.

---

35. Which of these values are falsy?

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

- A: `0`, `''`, `undefined`
- B: `0`, `new Number(0)`, `''`, `new Boolean(false)`, `undefined`
- C: `0`, `''`, `new Boolean(false)`, `undefined`
- D: All of them are falsy

▼ **Answer**

Answer: A

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. What's the output?

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

- A: `"number"`
- B: `"string"`
- C: `"object"`
- D: `"undefined"`

▼ **Answer**

Answer: B

`typeof 1` returns `"number"`.
`typeof "number"` returns `"string"`

---

37. What's the output?

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

- A: `[1, 2, 3, null x 7, 11]`
- B: `[1, 2, 3, 11]`
- C: `[1, 2, 3, empty x 7, 11]`
- D: `SyntaxError`

▼ **Answer**

Answer: C

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. What's the output?

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

- A: `1` `undefined` `2`
- B: `undefined` `undefined` `undefined`
- C: `1` `1` `2`
- D: `1` `undefined` `undefined`

▼ **Answer**

Answer: A

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**

Answer: A

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 behavior.

---

40. What's the output?

```
[[0, 1], [2, 3]].reduce(
  (acc, cur) => {
    return acc.concat(cur);
  },
  [1, 2],
);
```

- A: `[0, 1, 2, 3, 1, 2]`
- B: `[6, 1, 2]`
- C: `[1, 2, 0, 1, 2, 3]`
- D: `[1, 2, 6]`

▼ **Answer**

Answer: C

`[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. What's the output?

```
!!null;
!!'';
!!1;
```

- A: `false` `true` `false`
- B: `false` `false` `true`
- C: `false` `true` `true`
- D: `true` `true` `false`

▼ **Answer**

Answer: B

`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?

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

- A: a unique id
- B: the amount of milliseconds specified
- C: the passed function
- D: `undefined`

▼ **Answer**

Answer: A

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

---

43. What does this return?

```javascript
[...'Lydia'];
```

- A: `["L", "y", "d", "i", "a"]`
- B: `["Lydia"]`
- C: `[[], "Lydia"]`
- D: `[["L", "y", "d", "i", "a"]]`

▼ **Answer**

Answer: A

A string is an iterable. The spread operator maps every character of an iterable to one element.

---

44. What's the output?

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

const gen = generator(10);
```

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

- A: `[0, 10], [10, 20]`
- B: `20, 20`
- C: `10, 20`
- D: `0, 10 and 10, 20`

▼ **Answer**

Answer: C

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));
```

- A: `"one"`
- B: `"two"`
- C: `"two" "one"`
- D: `"one" "two"`

▼ **Answer**

Answer: B

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. What's the output?

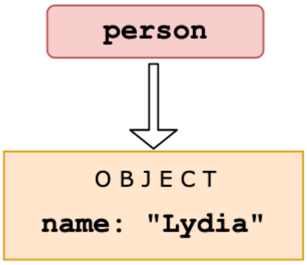
```
let person = { name: 'Lydia' };
const members = [person];
person = null;

console.log(members);
```

- A: `null`
- B: `[null]`
- C: `[{}]`
- D: `[{ name: "Lydia" }]`

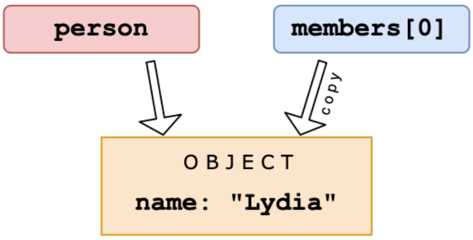
▼ **Answer**

Answer: D

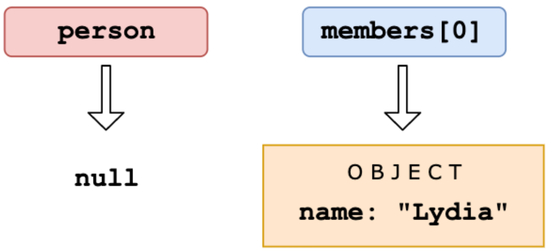
First, we declare a variable `person` with the value of an object that has a `name` property.



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. What's the output?

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

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

- A: `{ name: "Lydia" }, { age: 21 }`
- B: `"name", "age"`
- C: `"Lydia", 21`
- D: `["name", "Lydia"], ["age", 21]`

▼ **Answer**

Answer: B

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. What's the output?

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

- A: `"345"`
- B: `"75"`
- C: `12`
- D: `"12"`

▼ **Answer**

Answer: B

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`?

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

- A: `42`
- B: `"42"`
- C: `7`
- D: `NaN`

▼ **Answer**

Answer: C

Only the first number 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. What's the output?

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

- A: `[]`
- B: `[null, null, null]`
- C: `[undefined, undefined, undefined]`
- D: `[ 3 x empty ]`

▼ **Answer**

Answer: C

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`.

---

51. What's the output?

```
function getInfo(member, year) {
  member.name = 'Lydia';
  year = '1998';
}

const person = { name: 'Sarah' };
const birthYear = '1997';

getInfo(person, birthYear);

console.log(person, birthYear);
```

- A: `{ name: "Lydia" }, "1997"`
- B: `{ name: "Sarah" }, "1998"`
- C: `{ name: "Lydia" }, "1998"`
- D: `{ name: "Sarah" }, "1997"`

▼ **Answer**

Answer: A

Arguments are passed by *value*, unless their value is an object, then they're passed by *reference*. `birthYear` is passed by value, since it's a string, not an object. When we pass arguments by value, a *copy* of that value is created (see question 46).

The variable `birthYear` has a reference to the value `"1997"`. The argument `year` also has a reference to the value `"1997"`, but it's not the same value as `birthYear` has a reference to. When we update the value of `year` by setting `year` equal to `"1998"`, we are only updating the value of `year`. `birthYear` is still equal to `"1997"`.

The value of `person` is an object. The argument `member` has a (copied) reference to the *same* object. When we modify a property of the object `member` has a reference to, the value of `person` will also be modified, since they both have a reference to the same object. `person`'s `name` property is now equal to the value `"Lydia"`

52. What's the output?

```
function greeting() {
  throw 'Hello world!';
}

function sayHi() {
  try {
    const data = greeting();
    console.log('It worked!', data);
  } catch (e) {
    console.log('Oh no an error:', e);
  }
}

sayHi();
```

- A: `It worked! Hello world!`
- B: `Oh no an error: undefined`
- C: `SyntaxError: can only throw Error objects`
- D: `Oh no an error: Hello world!`

▼ **Answer**

Answer: D

With the `throw` statement, we can create custom errors. With this statement, you can throw exceptions. An exception can be a **string**, a **number**, a **boolean** or an **object**. In this case, our exception is the string `'Hello world!'`.

With the `catch` statement, we can specify what to do if an exception is thrown in the `try` block. An exception is thrown: the string `'Hello world!'`. `e` is now equal to that string, which we log. This results in `'Oh an error: Hello world!'`.

53. What's the output?

```
function Car() {
  this.make = 'Lamborghini';
  return { make: 'Maserati' };
}

const myCar = new Car();
console.log(myCar.make);
```

- A: `"Lamborghini"`

- B: `"Maserati"`
- C: `ReferenceError`
- D: `TypeError`

▼ **Answer**

Answer: B

When a constructor function is called with the `new` keyword, it creates an object and sets the `this` keyword to refer to that object. By default, if the constructor function doesn't explicitly return anything, it will return the newly created object.

In this case, the constructor function `Car` explicitly returns a new object with `make` set to `"Maserati"`, which overrides the default behavior. Therefore, when `new Car()` is called, the *returned* object is assigned to `myCar`, resulting in the output being `"Maserati"` when `myCar.make` is accessed.

---

54. What's the output?

```javascript
(() => {
  let x = (y = 10);
})();

console.log(typeof x);
console.log(typeof y);
```

- A: `"undefined", "number"`
- B: `"number", "number"`
- C: `"object", "number"`
- D: `"number", "undefined"`

▼ **Answer**

Answer: A

`let x = (y = 10);` is actually shorthand for:

```javascript
y = 10;
let x = y;
```

When we set `y` equal to `10`, we actually add a property `y` to the global object (`window` in the browser, `global` in Node). In a browser, `window.y` is now equal to `10`.

Then, we declare a variable `x` with the value of `y`, which is `10`. Variables declared with the `let` keyword are *block scoped*, they are only defined within the block they're declared in; the immediately invoked function expression (IIFE) in this case. When we use the `typeof` operator, the operand `x` is not defined: we are trying to access `x` outside of the block it's declared in. This means that `x` is not defined. Values who haven't been assigned a value or declared are of type `"undefined"`. `console.log(typeof x)` returns `"undefined"`.

However, we created a global variable `y` when setting `y` equal to `10`. This value is accessible anywhere in our code. `y` is defined, and holds a value of type `"number"`. `console.log(typeof y)` returns `"number"`.

---

55. What's the output?

```javascript
class Dog {
  constructor(name) {
    this.name = name;
  }
}

Dog.prototype.bark = function() {
  console.log(`Woof I am ${this.name}`);
};

const pet = new Dog('Mara');

pet.bark();

delete Dog.prototype.bark;

pet.bark();
```

- A: `"Woof I am Mara"`, `TypeError`
- B: `"Woof I am Mara"`, `"Woof I am Mara"`
- C: `"Woof I am Mara"`, `undefined`
- D: `TypeError`, `TypeError`

▼ **Answer**

Answer: A

We can delete properties from objects using the `delete` keyword, also on the prototype. By deleting a property on the prototype, it is not available anymore in the prototype chain. In this case, the `bark` function is not available anymore on the prototype after `delete Dog.prototype.bark`, yet we still try to access it.

When we try to invoke something that is not a function, a `TypeError` is thrown. In this case `TypeError: pet.bark is not a function`, since `pet.bark` is `undefined`.

---

56. What's the output?

```javascript
const set = new Set([1, 1, 2, 3, 4]);

console.log(set);
```

- A: `[1, 1, 2, 3, 4]`
- B: `[1, 2, 3, 4]`
- C: `{1, 1, 2, 3, 4}`
- D: `{1, 2, 3, 4}`

▼ **Answer**

Answer: D

The `Set` object is a collection of *unique* values: a value can only occur once in a set.

We passed the iterable `[1, 1, 2, 3, 4]` with a duplicate value `1`. Since we cannot have two of the same values in a set, one of them is removed. This results in `{1, 2, 3, 4}`.

---

57. What's the output?

```javascript
// counter.js
let counter = 10;
export default counter;
```

```javascript
// index.js
import myCounter from './counter';

myCounter += 1;

console.log(myCounter);
```

- A: `10`
- B: `11`
- C: `Error`
- D: `NaN`

▼ **Answer**

Answer: C

An imported module is *read-only*: you cannot modify the imported module. Only the module that exports them can change its value.

When we try to increment the value of `myCounter`, it throws an error: `myCounter` is read-only and cannot be modified.

---

58. What's the output?

```
const name = 'Lydia';
age = 21;

console.log(delete name);
console.log(delete age);
```

- A: `false`, `true`
- B: `"Lydia"`, `21`
- C: `true`, `true`
- D: `undefined`, `undefined`

▼ **Answer**

Answer: A

The `delete` operator returns a boolean value: `true` on a successful deletion, else it'll return `false`. However, variables declared with the `var`, `const`, or `let` keywords cannot be deleted using the `delete` operator.

The `name` variable was declared with a `const` keyword, so its deletion is not successful: `false` is returned. When we set `age` equal to `21`, we actually added a property called `age` to the global object. You can successfully delete properties from objects this way, also the global object, so `delete age` returns `true`.

---

59. What's the output?

```
const numbers = [1, 2, 3, 4, 5];
const [y] = numbers;

console.log(y);
```

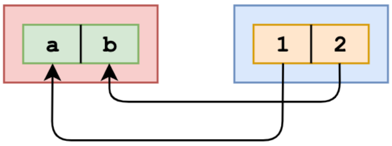
- A: `[[1, 2, 3, 4, 5]]`
- B: `[1, 2, 3, 4, 5]`
- C: `1`
- D: `[1]`

▼ **Answer**

Answer: C

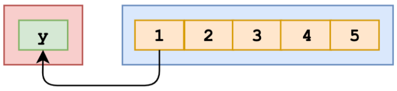
We can unpack values from arrays or properties from objects through destructuring. For example:

```
[a, b] = [1, 2];
```



The value of `a` is now `1`, and the value of `b` is now `2`. What we actually did in the question, is:

```
[y] = [1, 2, 3, 4, 5];
```



This means that the value of `y` is equal to the first value in the array, which is the number `1`. When we log `y`, `1` is returned.

---

60. What's the output?

```
const user = { name: 'Lydia', age: 21 };
const admin = { admin: true, ...user };

console.log(admin);
```

- A: `{ admin: true, user: { name: "Lydia", age: 21 } }`
- B: `{ admin: true, name: "Lydia", age: 21 }`
- C: `{ admin: true, user: ["Lydia", 21] }`
- D: `{ admin: true }`

▼ **Answer**

Answer: B

It's possible to combine objects using the spread operator `...`. It lets you create copies of the key/value pairs of one object, and add them to another object. In this case, we create copies of the `user` object, and add them to the `admin` object. The `admin` object now contains the copied key/value pairs, which results in `{ admin: true, name: "Lydia", age: 21 }`.

---

61. What's the output?

```
const person = { name: 'Lydia' };

Object.defineProperty(person, 'age', { value: 21 });

console.log(person);
console.log(Object.keys(person));
```

- A: `{ name: "Lydia", age: 21 }`, `["name", "age"]`
- B: `{ name: "Lydia", age: 21 }`, `["name"]`
- C: `{ name: "Lydia"}`, `["name", "age"]`

- D: `{ name: "Lydia"}`, `["age"]`

▼ **Answer**

Answer: B

With the `defineProperty` method, we can add new properties to an object, or modify existing ones. When we add a property to an object using the `defineProperty` method, they are by default *not enumerable*. The `Object.keys` method returns all *enumerable* property names from an object, in this case only `"name"`.

Properties added using the `defineProperty` method are immutable by default. You can override this behavior using the `writable`, `configurable` and `enumerable` properties. This way, the `defineProperty` method gives you a lot more control over the properties you're adding to an object.

---

62. What's the output?

```
const settings = {
  username: 'lydiahallie',
  level: 19,
  health: 90,
};

const data = JSON.stringify(settings, ['level', 'health']);
console.log(data);
```

- A: `"{"level":19, "health":90}"`
- B: `"{"username": "lydiahallie"}"`
- C: `"["level", "health"]"`
- D: `"{"username": "lydiahallie", "level":19, "health":90}"`

▼ **Answer**

Answer: A

The second argument of `JSON.stringify` is the *replacer*. The replacer can either be a function or an array, and lets you control what and how the values should be stringified.

If the replacer is an *array*, only the property names included in the array will be added to the JSON string. In this case, only the properties with the names `"level"` and `"health"` are included, `"username"` is excluded. `data` is now equal to `"{"level":19, "health":90}"`.

If the replacer is a *function*, this function gets called on every property in the object you're stringifying. The value returned from this function will be the value of the property when it's added to the JSON string. If the value is `undefined`, this property is excluded from the JSON string.

---

63. What's the output?

```
let num = 10;

const increaseNumber = () => num++;
const increasePassedNumber = number => number++;

const num1 = increaseNumber();
const num2 = increasePassedNumber(num1);

console.log(num1);
console.log(num2);
```

- A: `10`, `10`
- B: `10`, `11`
- C: `11`, `11`
- D: `11`, `12`

▼ **Answer**

Answer: A

The unary operator `++` *first returns* the value of the operand, *then increments* the value of the operand. The value of `num1` is `10`, since the `increaseNumber` function first returns the value of `num`, which is `10`, and only increments the value of `num` afterward.

`num2` is `10`, since we passed `num1` to the `increasePassedNumber`. `number` is equal to `10`(the value of `num1`). Again, the unary operator `++` *first returns* the value of the operand, *then increments* the value of the operand. The value of `number` is `10`, so `num2` is equal to `10`.

---

64. What's the output?

```
const value = { number: 10 };

const multiply = (x = { ...value }) => {
  console.log((x.number *= 2));
};

multiply();
multiply();
multiply(value);
multiply(value);
```

- A: `20`, `40`, `80`, `160`
- B: `20`, `40`, `20`, `40`

- C: `20`, `20`, `20`, `40`
- D: `NaN`, `NaN`, `20`, `40`

▼ **Answer**

Answer: C

In ES6, we can initialize parameters with a default value. The value of the parameter will be the default value, if no other value has been passed to the function, or if the value of the parameter is `"undefined"`. In this case, we spread the properties of the `value` object into a new object, so `x` has the default value of `{ number: 10 }`.

The default argument is evaluated at *call time*! Every time we call the function, a *new* object is created. We invoke the `multiply` function the first two times without passing a value: `x` has the default value of `{ number: 10 }`. We then log the multiplied value of that number, which is `20`.

The third time we invoke multiply, we do pass an argument: the object called `value`. The `*=` operator is actually shorthand for `x.number = x.number * 2`: we modify the value of `x.number`, and log the multiplied value `20`.

The fourth time, we pass the `value` object again. `x.number` was previously modified to `20`, so `x.number *= 2` logs `40`.

---

65. What's the output?

```
[1, 2, 3, 4].reduce((x, y) => console.log(x, y));
```

- A: `1` `2` and `3` `3` and `6` `4`
- B: `1` `2` and `2` `3` and `3` `4`
- C: `1` `undefined` and `2` `undefined` and `3` `undefined` and `4` `undefined`
- D: `1` `2` and `undefined` `3` and `undefined` `4`

▼ **Answer**

Answer: D

The first argument that the `reduce` method receives is the *accumulator*, `x` in this case. The second argument is the *current value*, `y`. With the reduce method, we execute a callback function on every element in the array, which could ultimately result in one single value.

In this example, we are not returning any values, we are simply logging the values of the accumulator and the current value.

The value of the accumulator is equal to the previously returned value of the callback function. If you don't pass the optional `initialValue` argument to the `reduce` method, the accumulator is equal to the first element on the first call.

On the first call, the accumulator (`x`) is `1`, and the current value (`y`) is `2`. We don't return from the callback function, we log the accumulator, and the current values: `1` and `2` get logged.

If you don't return a value from a function, it returns `undefined`. On the next call, the accumulator is `undefined`, and the current value is `3`. `undefined` and `3` get logged.

On the fourth call, we again don't return from the callback function. The accumulator is again `undefined`, and the current value is `4`. `undefined` and `4` get logged.

66. With which constructor can we successfully extend the `Dog` class?

```
class Dog {
  constructor(name) {
    this.name = name;
  }
};

class Labrador extends Dog {
  // 1
  constructor(name, size) {
    this.size = size;
  }
  // 2
  constructor(name, size) {
    super(name);
    this.size = size;
  }
  // 3
  constructor(size) {
    super(name);
    this.size = size;
  }
  // 4
  constructor(name, size) {
    this.name = name;
    this.size = size;
  }

};
```

- A: 1
- B: 2
- C: 3
- D: 4

▼ **Answer**

Answer: B

In a derived class, you cannot access the `this` keyword before calling `super`. If you try to do that, it will throw a ReferenceError: 1 and 4 would throw a reference error.

With the `super` keyword, we call that parent class's constructor with the given arguments. The parent's constructor receives the `name` argument, so we need to pass `name` to `super`.

The `Labrador` class receives two arguments, `name` since it extends `Dog`, and `size` as an extra property on the `Labrador` class. They both need to be passed to the constructor function on `Labrador`, which is done correctly using constructor 2.

---

67. What's the output?

```
// index.js
console.log('running index.js');
import { sum } from './sum.js';
console.log(sum(1, 2));

// sum.js
console.log('running sum.js');
export const sum = (a, b) => a + b;
```

- A: `running index.js`, `running sum.js`, `3`
- B: `running sum.js`, `running index.js`, `3`
- C: `running sum.js`, `3`, `running index.js`
- D: `running index.js`, `undefined`, `running sum.js`

▼ **Answer**

Answer: B

With the `import` keyword, all imported modules are *pre-parsed*. This means that the imported modules get run *first*, and the code in the file that imports the module gets executed *after*.

This is a difference between `require()` in CommonJS and `import`! With `require()`, you can load dependencies on demand while the code is being run. If we had used `require` instead of `import`, `running index.js`, `running sum.js`, `3` would have been logged to the console.

---

68. What's the output?

```
console.log(Number(2) === Number(2));
console.log(Boolean(false) === Boolean(false));
console.log(Symbol('foo') === Symbol('foo'));
```

- A: `true`, `true`, `false`
- B: `false`, `true`, `false`
- C: `true`, `false`, `true`
- D: `true`, `true`, `true`

▼ **Answer**

Answer: A

Every Symbol is entirely unique. The purpose of the argument passed to the Symbol is to give the Symbol a description. The value of the Symbol is not dependent on the passed argument. As we test equality, we are creating two entirely new symbols: the first `Symbol('foo')`, and the second `Symbol('foo')`. These two values are unique and not equal to each other, `Symbol('foo') === Symbol('foo')` returns `false`.

---

69. What's the output?

```javascript
const name = 'Lydia Hallie';
console.log(name.padStart(13));
console.log(name.padStart(2));
```

- A: `"Lydia Hallie"`, `"Lydia Hallie"`
- B: `" Lydia Hallie"`, `" Lydia Hallie"` (`"[13x whitespace]Lydia Hallie"`, `"[2x whitespace]Lydia Hallie"`)
- C: `" Lydia Hallie"`, `"Lydia Hallie"` (`"[1x whitespace]Lydia Hallie"`, `"Lydia Hallie"`)
- D: `"Lydia Hallie"`, `"Lyd"`,

▼ **Answer**

Answer: C

With the `padStart` method, we can add padding to the beginning of a string. The value passed to this method is the *total* length of the string together with the padding. The string `"Lydia Hallie"` has a length of `12`. `name.padStart(13)` inserts 1 space at the start of the string, because 12 + 1 is 13.

If the argument passed to the `padStart` method is smaller than the length of the array, no padding will be added.

---

70. What's the output?

```javascript
console.log('🥑' + '💻');
```

- A: `"🥑💻"`
- B: `257548`

- C: A string containing their code points
- D: Error

▼ **Answer**

Answer: A

With the `+` operator, you can concatenate strings. In this case, we are concatenating the string `"🥑"` with the string `"💻"`, resulting in `"🥑💻"`.

---

71. How can we log the values that are commented out after the console.log statement?

```javascript
function* startGame() {
  const answer = yield 'Do you love JavaScript?';
  if (answer !== 'Yes') {
    return "Oh wow... Guess we're done here";
  }
  return 'JavaScript loves you back ❤️';
}

const game = startGame();
console.log(/* 1 */); // Do you love JavaScript?
console.log(/* 2 */); // JavaScript loves you back ❤️
```

- A: `game.next("Yes").value` and `game.next().value`
- B: `game.next.value("Yes")` and `game.next.value()`
- C: `game.next().value` and `game.next("Yes").value`
- D: `game.next.value()` and `game.next.value("Yes")`

▼ **Answer**

Answer: C

A generator function "pauses" its execution when it sees the `yield` keyword. First, we have to let the function yield the string "Do you love JavaScript?", which can be done by calling `game.next().value`.

Every line is executed, until it finds the first `yield` keyword. There is a `yield` keyword on the first line within the function: the execution stops with the first yield! *This means that the variable `answer` is not defined yet!*

When we call `game.next("Yes").value`, the previous `yield` is replaced with the value of the parameters passed to the `next()` function, `"Yes"` in this case. The value of the variable `answer` is now equal to `"Yes"`. The condition of the if-statement returns `false`, and `JavaScript loves you back ❤️` gets logged.

---

72. What's the output?

```
console.log(String.raw`Hello\nworld`);
```

- A: `Hello world!`
- B: `Hello`
  `world`
- C: `Hello\nworld`
- D: `Hello\n`
  `world`

▼ **Answer**

Answer: C

`String.raw` returns a string where the escapes (`\n`, `\v`, `\t` etc.) are ignored! Backslashes can be an issue since you could end up with something like:

`` const path = `C:\Documents\Projects\table.html` ``

Which would result in:

`"C:DocumentsProjects able.html"`

With `String.raw`, it would simply ignore the escape and print:

`C:\Documents\Projects\table.html`

In this case, the string is `Hello\nworld`, which gets logged.

---

73. What's the output?

```
async function getData() {
  return await Promise.resolve('I made it!');
}

const data = getData();
console.log(data);
```

- A: `"I made it!"`
- B: `Promise {<resolved>: "I made it!"}`
- C: `Promise {<pending>}`
- D: `undefined`

▼ **Answer**

Answer: C

An async function always returns a promise. The `await` still has to wait for the promise to resolve: a pending promise gets returned when we call `getData()` in order to set `data` equal to it.

If we wanted to get access to the resolved value `"I made it"`, we could have used the `.then()` method on `data`:

`data.then(res => console.log(res))`

This would've logged `"I made it!"`

---

74. What's the output?

```javascript
function addToList(item, list) {
  return list.push(item);
}

const result = addToList('apple', ['banana']);
console.log(result);
```

- A: `['apple', 'banana']`
- B: `2`
- C: `true`
- D: `undefined`

▼ **Answer**

Answer: B

The `.push()` method returns the *length* of the new array! Previously, the array contained one element (the string `"banana"`) and had a length of `1`. After adding the string `"apple"` to the array, the array contains two elements, and has a length of `2`. This gets returned from the `addToList` function.

The `push` method modifies the original array. If you wanted to return the *array* from the function rather than the *length of the array*, you should have returned `list` after pushing `item` to it.

---

75. What's the output?

```javascript
const box = { x: 10, y: 20 };

Object.freeze(box);

const shape = box;
shape.x = 100;

console.log(shape);
```

- A: `{ x: 100, y: 20 }`
- B: `{ x: 10, y: 20 }`
- C: `{ x: 100 }`
- D: `ReferenceError`

▼ **Answer**

Answer: B

`Object.freeze` makes it impossible to add, remove, or modify properties of an object (unless the property's value is another object).

When we create the variable `shape` and set it equal to the frozen object `box`, `shape` also refers to a frozen object. You can check whether an object is frozen by using `Object.isFrozen`. In this case, `Object.isFrozen(shape)` would return true, since the variable `shape` has a reference to a frozen object.

Since `shape` is frozen, and since the value of `x` is not an object, we cannot modify the property `x`. `x` is still equal to `10`, and `{ x: 10, y: 20 }` gets logged.

---

76. What's the output?

```javascript
const { firstName: myName } = { firstName: 'Lydia' };

console.log(firstName);
```

- A: `"Lydia"`
- B: `"myName"`
- C: `undefined`
- D: `ReferenceError`

▼ **Answer**

Answer: D

By using [destructuring assignment](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment) syntax we can unpack values from arrays, or properties from objects, into distinct variables:

```javascript
const { firstName } = { firstName: 'Lydia' };
// ES5 version:
// var firstName = { firstName: 'Lydia' }.firstName;

console.log(firstName); // "Lydia"
```

Also, a property can be unpacked from an object and assigned to a variable with a different name than the object property:

```
const { firstName: myName } = { firstName: 'Lydia' };
// ES5 version:
// var myName = { firstName: 'Lydia' }.firstName;

console.log(myName); // "Lydia"
console.log(firstName); // Uncaught ReferenceError: firstName is not defined
```

Therefore, `firstName` does not exist as a variable, thus attempting to access its value will raise a `ReferenceError`.

Note: Be aware of the `global scope` properties:

```
const { name: myName } = { name: 'Lydia' };

console.log(myName); // "lydia"
console.log(name); // "" ----- Browser e.g. Chrome
console.log(name); // ReferenceError: name is not defined  ----- NodeJS
```

Whenever Javascript is unable to find a variable within the *current scope*, it climbs up the Scope chain and searches for it and if it reaches the top-level scope, aka **Global scope**, and still doesn't find it, it will throw a `ReferenceError`.

- In **Browsers** such as *Chrome*, `name` is a *deprecated global scope property*. In this example, the code is running inside *global scope* and there is no user-defined local variable for `name`, therefore it searches the predefined *variables/properties* in the global scope which is in the case of browsers, it searches through `window` object and it will extract the window.name value which is equal to an **empty string**.
- In **NodeJS**, there is no such property on the `global` object, thus attempting to access a non-existent variable will raise a ReferenceError.

---

77. Is this a pure function?

```
function sum(a, b) {
  return a + b;
}
```

- A: Yes
- B: No

▼ **Answer**

Answer: A

A pure function is a function that *always* returns the same result, if the same arguments are passed.

The `sum` function always returns the same result. If we pass `1` and `2`, it will *always* return `3` without side effects. If we pass `5` and `10`, it will *always* return `15`, and so on. This is the definition of a pure function.

---

78. What is the output?

```javascript
const add = () => {
  const cache = {};
  return num => {
    if (num in cache) {
      return `From cache! ${cache[num]}`;
    } else {
      const result = num + 10;
      cache[num] = result;
      return `Calculated! ${result}`;
    }
  };
};

const addFunction = add();
console.log(addFunction(10));
console.log(addFunction(10));
console.log(addFunction(5 * 2));
```

- A: `Calculated! 20` `Calculated! 20` `Calculated! 20`
- B: `Calculated! 20` `From cache! 20` `Calculated! 20`
- C: `Calculated! 20` `From cache! 20` `From cache! 20`
- D: `Calculated! 20` `From cache! 20` `Error`

▼ **Answer**

Answer: C

The `add` function is a *memoized* function. With memoization, we can cache the results of a function in order to speed up its execution. In this case, we create a `cache` object that stores the previously returned values.

If we call the `addFunction` function again with the same argument, it first checks whether it has already gotten that value in its cache. If that's the case, the cache value will be returned, which saves execution time. Otherwise, if it's not cached, it will calculate the value and store it afterward.

We call the `addFunction` function three times with the same value: on the first invocation, the value of the function when `num` is equal to `10` isn't cached yet. The condition of the if-statement `num in cache` returns `false`, and the else block gets executed: `Calculated! 20` gets logged, and the value of the result gets added to the cache object. `cache` now looks like `{ 10: 20 }`.

The second time, the `cache` object contains the value that gets returned for `10`. The condition of the if-statement `num in cache` returns `true`, and `'From cache! 20'` gets logged.

The third time, we pass `5 * 2` to the function which gets evaluated to `10`. The `cache` object contains the value that gets returned for `10`. The condition of the if-statement `num in cache` returns `true`, and `'From cache! 20'` gets logged.

---

79. What is the output?

```
const myLifeSummedUp = ['☕', '💻', '🍷', '🍫'];

for (let item in myLifeSummedUp) {
  console.log(item);
}

for (let item of myLifeSummedUp) {
  console.log(item);
}
```

- A: `0` `1` `2` `3` and `"☕"` `"💻"` `"🍷"` `"🍫"`
- B: `"☕"` `"💻"` `"🍷"` `"🍫"` and `"☕"` `"💻"` `"🍷"` `"🍫"`
- C: `"☕"` `"💻"` `"🍷"` `"🍫"` and `0` `1` `2` `3`
- D: `0` `1` `2` `3` and `{0: "☕", 1: "💻", 2: "🍷", 3: "🍫"}`

▼ **Answer**

Answer: A

With a *for-in* loop, we can iterate over **enumerable** properties. In an array, the enumerable properties are the "keys" of array elements, which are actually their indexes. You could see an array as:

`{0: "☕", 1: "💻", 2: "🍷", 3: "🍫"}`

Where the keys are the enumerable properties. `0` `1` `2` `3` get logged.

With a *for-of* loop, we can iterate over **iterables**. An array is an iterable. When we iterate over the array, the variable "item" is equal to the element it's currently iterating over, `"☕"` `"💻"` `"🍷"` `"🍫"` get logged.

---

80. What is the output?

```
const list = [1 + 2, 1 * 2, 1 / 2];
console.log(list);
```

- A: `["1 + 2", "1 * 2", "1 / 2"]`
- B: `["12", 2, 0.5]`

- C: `[3, 2, 0.5]`
- D: `[1, 1, 1]`

▼ **Answer**

Answer: C

Array elements can hold any value. Numbers, strings, objects, other arrays, null, boolean values, undefined, and other expressions such as dates, functions, and calculations.

The element will be equal to the returned value. `1 + 2` returns `3`, `1 * 2` returns `2`, and `1 / 2` returns `0.5`.

---

81. What is the output?

```javascript
function sayHi(name) {
  return `Hi there, ${name}`;
}

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

- A: `Hi there,`
- B: `Hi there, undefined`
- C: `Hi there, null`
- D: `ReferenceError`

▼ **Answer**

Answer: B

By default, arguments have the value of `undefined`, unless a value has been passed to the function. In this case, we didn't pass a value for the `name` argument. `name` is equal to `undefined` which gets logged.

In ES6, we can overwrite this default `undefined` value with default parameters. For example:

`function sayHi(name = "Lydia") { ... }`

In this case, if we didn't pass a value or if we passed `undefined`, `name` would always be equal to the string `Lydia`

---

82. What is the output?

```javascript
var status = '😎';

setTimeout(() => {
  const status = '😍';

  const data = {
    status: '🥑',
    getStatus() {
      return this.status;
    },
  };

  console.log(data.getStatus());
  console.log(data.getStatus.call(this));
}, 0);
```

- A: `"🥑"` and `"😍"`
- B: `"🥑"` and `"😎"`
- C: `"😍"` and `"😎"`
- D: `"😎"` and `"😎"`

▼ **Answer**

Answer: B

The value of the `this` keyword is dependent on where you use it. In a **method**, like the `getStatus` method, the `this` keyword refers to *the object that the method belongs to*. The method belongs to the `data` object, so `this` refers to the `data` object. When we log `this.status`, the `status` property on the `data` object gets logged, which is `"🥑"`.

With the `call` method, we can change the object to which the `this` keyword refers. In **functions**, the `this` keyword refers to the *the object that the function belongs to*. We declared the `setTimeout` function on the *global object*, so within the `setTimeout` function, the `this` keyword refers to the *global object*. On the global object, there is a variable called *status* with the value of `"😎"`. When logging `this.status`, `"😎"` gets logged.

---

83. What is the output?

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

let city = person.city;
city = 'Amsterdam';

console.log(person);
```

- A: `{ name: "Lydia", age: 21 }`
- B: `{ name: "Lydia", age: 21, city: "Amsterdam" }`
- C: `{ name: "Lydia", age: 21, city: undefined }`
- D: `"Amsterdam"`

▼ **Answer**

Answer: A

We set the variable `city` equal to the value of the property called `city` on the `person` object. There is no property on this object called `city`, so the variable `city` has the value of `undefined`.

Note that we are *not* referencing the `person` object itself! We simply set the variable `city` equal to the current value of the `city` property on the `person` object.

Then, we set `city` equal to the string `"Amsterdam"`. This doesn't change the person object: there is no reference to that object.

When logging the `person` object, the unmodified object gets returned.

---

84. What is the output?

```javascript
function checkAge(age) {
  if (age < 18) {
    const message = "Sorry, you're too young.";
  } else {
    const message = "Yay! You're old enough!";
  }

  return message;
}

console.log(checkAge(21));
```

- A: `"Sorry, you're too young."`
- B: `"Yay! You're old enough!"`
- C: `ReferenceError`
- D: `undefined`

▼ **Answer**

Answer: C

Variables with the `const` and `let` keywords are *block-scoped*. A block is anything between curly brackets (`{ }`). In this case, the curly brackets of the if/else statements. You cannot reference a variable outside of the block it's declared in, a ReferenceError gets thrown.

85. What kind of information would get logged?

```
fetch('https://www.website.com/api/user/1')
  .then(res => res.json())
  .then(res => console.log(res));
```

- A: The result of the `fetch` method.
- B: The result of the second invocation of the `fetch` method.
- C: The result of the callback in the previous `.then()`.
- D: It would always be undefined.

▼ **Answer**

Answer: C

The value of `res` in the second `.then` is equal to the returned value of the previous `.then`. You can keep chaining `.then`s like this, where the value is passed to the next handler.

---

86. Which option is a way to set `hasName` equal to `true`, provided you cannot pass `true` as an argument?

```
function getName(name) {
  const hasName = //
}
```

- A: `!!name`
- B: `name`
- C: `new Boolean(name)`
- D: `name.length`

▼ **Answer**

Answer: A

With `!!name`, we determine whether the value of `name` is truthy or falsy. If the name is truthy, which we want to test for, `!name` returns `false`. `!false` (which is what `!!name` practically is) returns `true`.

By setting `hasName` equal to `name`, you set `hasName` equal to whatever value you passed to the `getName` function, not the boolean value `true`.

`new Boolean(true)` returns an object wrapper, not the boolean value itself.

`name.length` returns the length of the passed argument, not whether it's `true`.

---

87. What's the output?

```
console.log('I want pizza'[0]);
```

- A: `"""`
- B: `"I"`
- C: `SyntaxError`
- D: `undefined`

▼ **Answer**

Answer: B

In order to get a character at a specific index of a string, you can use bracket notation. The first character in the string has index 0, and so on. In this case, we want to get the element with index 0, the character `"I'`, which gets logged.

Note that this method is not supported in IE7 and below. In that case, use `.charAt()`.

---

88. What's the output?

```
function sum(num1, num2 = num1) {
  console.log(num1 + num2);
}

sum(10);
```

- A: `NaN`
- B: `20`
- C: `ReferenceError`
- D: `undefined`

▼ **Answer**

Answer: B

You can set a default parameter's value equal to another parameter of the function, as long as they've been defined *before* the default parameter. We pass the value `10` to the `sum` function. If the `sum` function only receives 1 argument, it means that the value for `num2` is not passed, and the value of `num1` is equal to the passed value `10` in this case. The default value of `num2` is the value of `num1`, which is `10`. `num1 + num2` returns `20`.

If you're trying to set a default parameter's value equal to a parameter that is defined *after* (to the right), the parameter's value hasn't been initialized yet, which will throw an error.

---

89. What's the output?

```javascript
// module.js
export default () => 'Hello world';
export const name = 'Lydia';

// index.js
import * as data from './module';

console.log(data);
```

- A: `{ default: function default(), name: "Lydia" }`
- B: `{ default: function default() }`
- C: `{ default: "Hello world", name: "Lydia" }`
- D: Global object of `module.js`

▼ **Answer**

Answer: A

With the `import * as name` syntax, we import *all exports* from the `module.js` file into the `index.js` file as a new object called `data` is created. In the `module.js` file, there are two exports: the default export, and a named export. The default export is a function that returns the string `"Hello World"`, and the named export is a variable called `name` which has the value of the string `"Lydia"`.

The `data` object has a `default` property for the default export, other properties have the names of the named exports and their corresponding values.

---

90. What's the output?

```javascript
class Person {
  constructor(name) {
    this.name = name;
  }
}

const member = new Person('John');
console.log(typeof member);
```

- A: `"class"`
- B: `"function"`
- C: `"object"`
- D: `"string"`

▼ **Answer**

Answer: C

Classes are syntactical sugar for function constructors. The equivalent of the `Person` class as a function constructor would be:

```
function Person(name) {
  this.name = name;
}
```

Calling a function constructor with `new` results in the creation of an instance of `Person`, `typeof` keyword returns `"object"` for an instance. `typeof member` returns `"object"`.

---

91. What's the output?

```
let newList = [1, 2, 3].push(4);

console.log(newList.push(5));
```

- A: `[1, 2, 3, 4, 5]`
- B: `[1, 2, 3, 5]`
- C: `[1, 2, 3, 4]`
- D: `Error`

▼ **Answer**

Answer: D

The `.push` method returns the *new length* of the array, not the array itself! By setting `newList` equal to `[1, 2, 3].push(4)`, we set `newList` equal to the new length of the array: `4`.

Then, we try to use the `.push` method on `newList`. Since `newList` is the numerical value `4`, we cannot use the `.push` method: a TypeError is thrown.

---

92. What's the output?

```
function giveLydiaPizza() {
  return 'Here is pizza!';
}

const giveLydiaChocolate = () =>
  "Here's chocolate... now go hit the gym already.";

console.log(giveLydiaPizza.prototype);
console.log(giveLydiaChocolate.prototype);
```

- A: `{ constructor: ...}` `{ constructor: ...}`
- B: `{}` `{ constructor: ...}`
- C: `{ constructor: ...}` `{}`

- D: `{ constructor: ...}` `undefined`

▼ **Answer**

Answer: D

Regular functions, such as the `giveLydiaPizza` function, have a `prototype` property, which is an object (prototype object) with a `constructor` property. Arrow functions however, such as the `giveLydiaChocolate` function, do not have this `prototype` property. `undefined` gets returned when trying to access the `prototype` property using `giveLydiaChocolate.prototype`.

---

93. What's the output?

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

for (const [x, y] of Object.entries(person)) {
  console.log(x, y);
}
```

- A: `name` `Lydia` and `age` `21`
- B: `["name", "Lydia"]` and `["age", 21]`
- C: `["name", "age"]` and `undefined`
- D: `Error`

▼ **Answer**

Answer: A

`Object.entries(person)` returns an array of nested arrays, containing the keys and objects:

`[ [ 'name', 'Lydia' ], [ 'age', 21 ] ]`

Using the `for-of` loop, we can iterate over each element in the array, the subarrays in this case. We can destructure the subarrays instantly in the for-of loop, using `const [x, y]`. `x` is equal to the first element in the subarray, `y` is equal to the second element in the subarray.

The first subarray is `[ "name", "Lydia" ]`, with `x` equal to `"name"`, and `y` equal to `"Lydia"`, which get logged.
The second subarray is `[ "age", 21 ]`, with `x` equal to `"age"`, and `y` equal to `21`, which get logged.

---

94. What's the output?

```
function getItems(fruitList, ...args, favoriteFruit) {
  return [...fruitList, ...args, favoriteFruit]
}

getItems(["banana", "apple"], "pear", "orange")
```

- A: `["banana", "apple", "pear", "orange"]`
- B: `[["banana", "apple"], "pear", "orange"]`
- C: `["banana", "apple", ["pear"], "orange"]`
- D: `SyntaxError`

▼ **Answer**

Answer: D

`...args` is a rest parameter. The rest parameter's value is an array containing all remaining arguments, **and can only be the last parameter**! In this example, the rest parameter was the second parameter. This is not possible, and will throw a syntax error.

```
function getItems(fruitList, favoriteFruit, ...args) {
  return [...fruitList, ...args, favoriteFruit];
}

getItems(['banana', 'apple'], 'pear', 'orange');
```

The above example works. This returns the array `[ 'banana', 'apple', 'orange', 'pear' ]`

---

95. What's the output?

```
function nums(a, b) {
  if (a > b) console.log('a is bigger');
  else console.log('b is bigger');
  return
  a + b;
}

console.log(nums(4, 2));
console.log(nums(1, 2));
```

- A: `a is bigger`, `6` and `b is bigger`, `3`
- B: `a is bigger`, `undefined` and `b is bigger`, `undefined`
- C: `undefined` and `undefined`
- D: `SyntaxError`

▼ **Answer**

Answer: B

In JavaScript, we don't *have* to write the semicolon (`;`) explicitly, however the JavaScript engine still adds them after statements. This is called **Automatic Semicolon Insertion**. A statement can for example be variables, or keywords like `throw`, `return`, `break`, etc.

Here, we wrote a `return` statement, and another value `a + b` on a *new line*. However, since it's a new line, the engine doesn't know that it's actually the value that we wanted to return. Instead, it automatically added a semicolon after `return`. You could see this as:

```
return;
a + b;
```

This means that `a + b` is never reached, since a function stops running after the `return` keyword. If no value gets returned, like here, the function returns `undefined`. Note that there is no automatic insertion after `if/else` statements!

---

96. What's the output?

```
class Person {
  constructor() {
    this.name = 'Lydia';
  }
}

Person = class AnotherPerson {
  constructor() {
    this.name = 'Sarah';
  }
};

const member = new Person();
console.log(member.name);
```

- A: `"Lydia"`
- B: `"Sarah"`
- C: `Error: cannot redeclare Person`
- D: `SyntaxError`

▼ **Answer**

Answer: B

We can set classes equal to other classes/function constructors. In this case, we set `Person` equal to `AnotherPerson`. The name on this constructor is `Sarah`, so the name property on the new `Person` instance `member` is `"Sarah"`.

---

97. What's the output?

```javascript
const info = {
  [Symbol('a')]: 'b',
};

console.log(info);
console.log(Object.keys(info));
```

- A: `{Symbol('a'): 'b'}` and `["{Symbol('a')"]`
- B: `{}` and `[]`
- C: `{ a: "b" }` and `["a"]`
- D: `{Symbol('a'): 'b'}` and `[]`

▼ **Answer**

Answer: D

A Symbol is not *enumerable*. The Object.keys method returns all *enumerable* key properties on an object. The Symbol won't be visible, and an empty array is returned. When logging the entire object, all properties will be visible, even non-enumerable ones.

This is one of the many qualities of a symbol: besides representing an entirely unique value (which prevents accidental name collision on objects, for example when working with 2 libraries that want to add properties to the same object), you can also "hide" properties on objects this way (although not entirely. You can still access symbols using the `Object.getOwnPropertySymbols()` method).

---

98. What's the output?

```javascript
const getList = ([x, ...y]) => [x, y]
const getUser = user => { name: user.name, age: user.age }

const list = [1, 2, 3, 4]
const user = { name: "Lydia", age: 21 }

console.log(getList(list))
console.log(getUser(user))
```

- A: `[1, [2, 3, 4]]` and `SyntaxError`
- B: `[1, [2, 3, 4]]` and `{ name: "Lydia", age: 21 }`
- C: `[1, 2, 3, 4]` and `{ name: "Lydia", age: 21 }`
- D: `Error` and `{ name: "Lydia", age: 21 }`

▼ **Answer**

Answer: A

The `getList` function receives an array as its argument. Between the parentheses of the `getList` function, we destructure this array right away. You could see this as:

`[x, ...y] = [1, 2, 3, 4]`

With the rest parameter `...y`, we put all "remaining" arguments in an array. The remaining arguments are `2`, `3` and `4` in this case. The value of `y` is an array, containing all the rest parameters. The value of `x` is equal to `1` in this case, so when we log `[x, y]`, `[1, [2, 3, 4]]` gets logged.

The `getUser` function receives an object. With arrow functions, we don't *have* to write curly brackets if we just return one value. However, if you want to instantly return an *object* from an arrow function, you have to write it between parentheses, otherwise everything between the two braces will be interpreted as a block statement. In this case the code between the braces is not a valid JavaScript code, so a `SyntaxError` gets thrown.

The following function would have returned an object:

`const getUser = user => ({ name: user.name, age: user.age })`

---

99. What's the output?

```javascript
const name = 'Lydia';

console.log(name());
```

- A: `SyntaxError`
- B: `ReferenceError`
- C: `TypeError`
- D: `undefined`

▼ **Answer**

Answer: C

The variable `name` holds the value of a string, which is not a function, and thus cannot be invoked.

TypeErrors get thrown when a value is not of the expected type. JavaScript expected `name` to be a function since we're trying to invoke it. It was a string however, so a TypeError gets thrown: name is not a function!

SyntaxErrors get thrown when you've written something that isn't valid JavaScript, for example when you've written the word `return` as `retrun`.
ReferenceErrors get thrown when JavaScript isn't able to find a reference to a value that you're trying to access.

---

100. What's the value of output?

```javascript
// 🎉✨ This is my 100th question! ✨🎉

const output = `${[] && 'Im'}possible!
You should${'' && `n't`} see a therapist after so much JavaScript lol`;
```

- A: `possible! You should see a therapist after so much JavaScript lol`
- B: `Impossible! You should see a therapist after so much JavaScript lol`
- C: `possible! You shouldn't see a therapist after so much JavaScript lol`
- D: `Impossible! You shouldn't see a therapist after so much JavaScript lol`

▼ **Answer**

Answer: B

`[]` is a truthy value. With the `&&` operator, the right-hand value will be returned if the left-hand value is a truthy value. In this case, the left-hand value `[]` is a truthy value, so `"Im'` gets returned.

`""` is a falsy value. If the left-hand value is falsy, nothing gets returned. `n't` doesn't get returned.

---

101. What's the value of output?

```javascript
const one = false || {} || null;
const two = null || false || '';
const three = [] || 0 || true;

console.log(one, two, three);
```

- A: `false` `null` `[]`
- B: `null` `""` `true`
- C: `{}` `""` `[]`
- D: `null` `null` `true`

▼ **Answer**

Answer: C

With the `||` operator, we can return the first truthy operand. If all values are falsy, the last operand gets returned.

`(false || {} || null)`: the empty object `{}` is a truthy value. This is the first (and only) truthy value, which gets returned. `one` is equal to `{}`.

`(null || false || "")`: all operands are falsy values. This means that the last operand, `""` gets returned. `two` is equal to `""`.

`([] || 0 || "")`: the empty array `[]` is a truthy value. This is the first truthy value, which gets returned. `three` is equal to `[]`.

---

102. What's the value of output?

```javascript
const myPromise = () => Promise.resolve('I have resolved!');

function firstFunction() {
  myPromise().then(res => console.log(res));
  console.log('second');
}

async function secondFunction() {
  console.log(await myPromise());
  console.log('second');
}

firstFunction();
secondFunction();
```

- A: `I have resolved!`, `second` and `I have resolved!`, `second`
- B: `second`, `I have resolved!` and `second`, `I have resolved!`
- C: `I have resolved!`, `second` and `second`, `I have resolved!`
- D: `second`, `I have resolved!` and `I have resolved!`, `second`

▼ **Answer**

Answer: D

With a promise, we basically say *I want to execute this function, but I'll put it aside for now while it's running since this might take a while. Only when a certain value is resolved (or rejected), and when the call stack is empty, I want to use this value.*

We can get this value with both `.then` and the `await` keywords in an `async` function. Although we can get a promise's value with both `.then` and `await`, they work a bit differently.

In the `firstFunction`, we (sort of) put the myPromise function aside while it was running, but continued running the other code, which is `console.log('second')` in this case. Then, the function resolved with the string `I have resolved`, which then got logged after it saw that the callstack was empty.

With the await keyword in `secondFunction`, we literally pause the execution of an async function until the value has been resolved before moving to the next line.

This means that it waited for the `myPromise` to resolve with the value `I have resolved`, and only once that happened, we moved to the next line: `second` got logged.

103. What's the value of output?

```
const set = new Set();

set.add(1);
set.add('Lydia');
set.add({ name: 'Lydia' });

for (let item of set) {
  console.log(item + 2);
}
```

- A: `3`, `NaN`, `NaN`
- B: `3`, `7`, `NaN`
- C: `3`, `Lydia2`, `[object Object]2`
- D: `"12"`, `Lydia2`, `[object Object]2`

▼ **Answer**

Answer: C

The `+` operator is not only used for adding numerical values, but we can also use it to concatenate strings. Whenever the JavaScript engine sees that one or more values are not a number, it coerces the number into a string.

The first one is `1`, which is a numerical value. `1 + 2` returns the number 3.

However, the second one is a string `"Lydia"`. `"Lydia"` is a string and `2` is a number: `2` gets coerced into a string. `"Lydia"` and `"2"` get concatenated, which results in the string `"Lydia2"`.

`{ name: "Lydia" }` is an object. Neither a number nor an object is a string, so it stringifies both. Whenever we stringify a regular object, it becomes `"[object Object]"`. `"[object Object]"` concatenated with `"2"` becomes `"[object Object]2"`.

104. What's its value?

```
Promise.resolve(5);
```

- A: `5`
- B: `Promise {<pending>: 5}`
- C: `Promise {<fulfilled>: 5}`
- D: `Error`

▼ **Answer**

Answer: C

We can pass any type of value we want to `Promise.resolve`, either a promise or a non-promise. The method itself returns a promise with the resolved value (`<fulfilled>`). If you pass a regular function, it'll be a resolved promise with a regular value. If you pass a promise, it'll be a resolved promise with the resolved value of that passed promise.

In this case, we just passed the numerical value `5`. It returns a resolved promise with the value `5`.

---

105. What's its value?

```
function compareMembers(person1, person2 = person) {
  if (person1 !== person2) {
    console.log('Not the same!');
  } else {
    console.log('They are the same!');
  }
}

const person = { name: 'Lydia' };

compareMembers(person);
```

- A: `Not the same!`
- B: `They are the same!`
- C: `ReferenceError`
- D: `SyntaxError`

▼ **Answer**

Answer: B

Objects are passed by reference. When we check objects for strict equality (`===`), we're comparing their references.

We set the default value for `person2` equal to the `person` object, and passed the `person` object as the value for `person1`.

This means that both values have a reference to the same spot in memory, thus they are equal.

The code block in the `else` statement gets run, and `They are the same!` gets logged.

---

106. What's its value?

```
const colorConfig = {
  red: true,
  blue: false,
  green: true,
  black: true,
  yellow: false,
};

const colors = ['pink', 'red', 'blue'];

console.log(colorConfig.colors[1]);
```

- A: `true`
- B: `false`
- C: `undefined`
- D: `TypeError`

▼ **Answer**

Answer: D

In JavaScript, we have two ways to access properties on an object: bracket notation, or dot notation. In this example, we use dot notation (`colorConfig.colors`) instead of bracket notation (`colorConfig["colors"]`).

With dot notation, JavaScript tries to find the property on the object with that exact name. In this example, JavaScript tries to find a property called `colors` on the `colorConfig` object. There is no property called `colors`, so this returns `undefined`. Then, we try to access the value of the first element by using `[1]`. We cannot do this on a value that's `undefined`, so it throws a `TypeError`: `Cannot read property '1' of undefined`.

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. If we would've used `colorConfig[colors[1]]`, it would have returned the value of the `red` property on the `colorConfig` object.

---

107. What's its value?

```
console.log('❤️' === '❤️');
```

- A: `true`
- B: `false`

▼ **Answer**

Answer: A

Under the hood, emojis are unicodes. The unicodes for the heart emoji is `"U+2764 U+FE0F"`. These are always the same for the same emojis, so we're comparing two equal strings to each other, which returns true.

---

108. Which of these methods modifies the original array?

```
const emojis = ['✨', '🥑', '😍'];

emojis.map(x => x + '✨');
emojis.filter(x => x !== '🥑');
emojis.find(x => x !== '🥑');
emojis.reduce((acc, cur) => acc + '✨');
emojis.slice(1, 2, '✨');
emojis.splice(1, 2, '✨');
```

- A: `All of them`
- B: `map` `reduce` `slice` `splice`
- C: `map` `slice` `splice`
- D: `splice`

▼ **Answer**

Answer: D

With `splice` method, we modify the original array by deleting, replacing or adding elements. In this case, we removed 2 items from index 1 (we removed `'🥑'` and `'😍'`) and added the ✨ emoji instead.

`map`, `filter` and `slice` return a new array, `find` returns an element, and `reduce` returns a reduced value.

---

109. What's the output?

```
const food = ['🍕', '🍫', '🥑', '🍔'];
const info = { favoriteFood: food[0] };

info.favoriteFood = '🍝';

console.log(food);
```

- A: `['🍕', '🍫', '🥑', '🍔']`
- B: `['🍝', '🍫', '🥑', '🍔']`
- C: `['🍝', '🍕', '🍫', '🥑', '🍔']`
- D: `ReferenceError`

▼ **Answer**

Answer: A

We set the value of the `favoriteFood` property on the `info` object equal to the string with the pizza emoji, `'🍕'`. A string is a primitive data type. In JavaScript, primitive data types don't interact by reference.

In JavaScript, primitive data types (everything that's not an object) interact by *value*. In this case, we set the value of the `favoriteFood` property on the `info` object equal to the value of the first element in the `food` array, the string with the pizza emoji in this case (`'🍕'`). A string is a primitive data type, and interact by value (see my blogpost if you're interested in learning more)

Then, we change the value of the `favoriteFood` property on the `info` object. The `food` array hasn't changed, since the value of `favoriteFood` was merely a *copy* of the value of the first element in the array, and doesn't have a reference to the same spot in memory as the element on `food[0]`. When we log food, it's still the original array, `['🍕', '🍫', '🥑', '🍔']`.

---

110. What does this method do?

```javascript
JSON.parse();
```

- A: Parses JSON to a JavaScript value
- B: Parses a JavaScript object to JSON
- C: Parses any JavaScript value to JSON
- D: Parses JSON to a JavaScript object only

▼ **Answer**

Answer: A

With the `JSON.parse()` method, we can parse JSON string to a JavaScript value.

```javascript
// Stringifying a number into valid JSON, then parsing the JSON string to a JavaScript value:
const jsonNumber = JSON.stringify(4); // '4'
JSON.parse(jsonNumber); // 4

// Stringifying an array value into valid JSON, then parsing the JSON string to a JavaScript value:
const jsonArray = JSON.stringify([1, 2, 3]); // '[1, 2, 3]'
JSON.parse(jsonArray); // [1, 2, 3]

// Stringifying an object  into valid JSON, then parsing the JSON string to a JavaScript value:
const jsonArray = JSON.stringify({ name: 'Lydia' }); // '{"name":"Lydia"}'
JSON.parse(jsonArray); // { name: 'Lydia' }
```

---

111. What's the output?

```
let name = 'Lydia';

function getName() {
  console.log(name);
  let name = 'Sarah';
}

getName();
```

- A: Lydia
- B: Sarah
- C: `undefined`
- D: `ReferenceError`

▼ **Answer**

Answer: D

Each function has its own *execution context* (or *scope*). The `getName` function first looks within its own context (scope) to see if it contains the variable `name` we're trying to access. In this case, the `getName` function contains its own `name` variable: we declare the variable `name` with the `let` keyword, and with the value of `'Sarah'`.

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 `ReferenceError`.

If we wouldn't have declared the `name` variable within the `getName` function, the javascript engine would've looked down the *scope chain*. The outer scope has a variable called `name` with the value of `Lydia`. In that case, it would've logged `Lydia`.

```
let name = 'Lydia';

function getName() {
  console.log(name);
}

getName(); // Lydia
```

---

112. What's the output?

```
function* generatorOne() {
  yield ['a', 'b', 'c'];
}

```

```
function* generatorTwo() {
  yield* ['a', 'b', 'c'];
}

const one = generatorOne();
const two = generatorTwo();

console.log(one.next().value);
console.log(two.next().value);
```

- A: `a` and `a`
- B: `a` and `undefined`
- C: `['a', 'b', 'c']` and `a`
- D: `a` and `['a', 'b', 'c']`

▼ **Answer**

Answer: C

With the `yield` keyword, we `yield` values in a generator function. With the `yield*` keyword, we can yield values from another generator function, or iterable object (for example an array).

In `generatorOne`, we yield the entire array `['a', 'b', 'c']` using the `yield` keyword. The value of `value` property on the object returned by the `next` method on `one` (`one.next().value`) is equal to the entire array `['a', 'b', 'c']`.

```
console.log(one.next().value); // ['a', 'b', 'c']
console.log(one.next().value); // undefined
```

In `generatorTwo`, we use the `yield*` keyword. This means that the first yielded value of `two`, is equal to the first yielded value in the iterator. The iterator is the array `['a', 'b', 'c']`. The first yielded value is `a`, so the first time we call `two.next().value`, `a` is returned.

```
console.log(two.next().value); // 'a'
console.log(two.next().value); // 'b'
console.log(two.next().value); // 'c'
console.log(two.next().value); // undefined
```

---

113. What's the output?

```
console.log(`${(x => x)('I love')} to program`);
```

- A: `I love to program`
- B: `undefined to program`
- C: `${(x => x)('I love') to program`

- D: `TypeError`

▼ **Answer**

Answer: A

Expressions within template literals are evaluated first. This means that the string will contain the returned value of the expression, the immediately invoked function `(x => x)('I love')` in this case. We pass the value `'I love'` as an argument to the `x => x` arrow function. `x` is equal to `'I love'`, which gets returned. This results in `I love to program`.

---

114. What will happen?

```javascript
let config = {
  alert: setInterval(() => {
    console.log('Alert!');
  }, 1000),
};

config = null;
```

- A: The `setInterval` callback won't be invoked
- B: The `setInterval` callback gets invoked once
- C: The `setInterval` callback will still be called every second
- D: We never invoked `config.alert()`, config is `null`

▼ **Answer**

Answer: C

Normally when we set objects equal to `null`, those objects get *garbage collected* as there is no reference anymore to that object. However, since the callback function within `setInterval` is an arrow function (thus bound to the `config` object), the callback function still holds a reference to the `config` object.
As long as there is a reference, the object won't get garbage collected.
Since this is an interval, setting `config` to `null` or `delete`-ing `config.alert` won't garbage-collect the interval, so the interval will still be called.
It should be cleared with `clearInterval(config.alert)` to remove it from memory.
Since it was not cleared, the `setInterval` callback function will still get invoked every 1000ms (1s).

---

115. Which method(s) will return the value `'Hello world!'`?

```javascript
const myMap = new Map();
const myFunc = () => 'greeting';

myMap.set(myFunc, 'Hello world!');
```

```
//1
myMap.get('greeting');
//2
myMap.get(myFunc);
//3
myMap.get(() => 'greeting');
```

- A: 1
- B: 2
- C: 2 and 3
- D: All of them

▼ **Answer**

Answer: B

When adding a key/value pair using the `set` method, the key will be the value of the first argument passed to the `set` function, and the value will be the second argument passed to the `set` function. The key is the *function* `() => 'greeting'` in this case, and the value `'Hello world'`. `myMap` is now `{ () => 'greeting' => 'Hello world!' }`.

1 is wrong, since the key is not `'greeting'` but `() => 'greeting'`.
3 is wrong, since we're creating a new function by passing it as a parameter to the `get` method. Object interacts by *reference*. Functions are objects, which is why two functions are never strictly equal, even if they are identical: they have a reference to a different spot in memory.

---

116. What's the output?

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

const changeAge = (x = { ...person }) => (x.age += 1);
const changeAgeAndName = (x = { ...person }) => {
  x.age += 1;
  x.name = 'Sarah';
};

changeAge(person);
changeAgeAndName();

console.log(person);
```

- A: `{name: "Sarah", age: 22}`
- B: `{name: "Sarah", age: 23}`
- C: `{name: "Lydia", age: 22}`
- D: `{name: "Lydia", age: 23}`

▼ **Answer**

Answer: C

Both the `changeAge` and `changeAgeAndName` functions have a default parameter, namely a *newly* created object `{ ...person }`. This object has copies of all the key/values in the `person` object.

First, we invoke the `changeAge` function and pass the `person` object as its argument. This function increases the value of the `age` property by 1. `person` is now `{ name: "Lydia", age: 22 }`.

Then, we invoke the `changeAgeAndName` function, however we don't pass a parameter. Instead, the value of `x` is equal to a *new* object: `{ ...person }`. Since it's a new object, it doesn't affect the values of the properties on the `person` object. `person` is still equal to `{ name: "Lydia", age: 22 }`.

---

117. Which of the following options will return `6`?

```javascript
function sumValues(x, y, z) {
  return x + y + z;
}
```

- A: `sumValues([...1, 2, 3])`
- B: `sumValues([...[1, 2, 3]])`
- C: `sumValues(...[1, 2, 3])`
- D: `sumValues([1, 2, 3])`

▼ **Answer**

Answer: C

With the spread operator `...`, we can *spread* iterables to individual elements. The `sumValues` function receives three arguments: `x`, `y` and `z`. `...[1, 2, 3]` will result in `1, 2, 3`, which we pass to the `sumValues` function.

---

118. What's the output?

```javascript
let num = 1;
const list = ['🥳', '🤠', '🥰', '🤪'];

console.log(list[(num += 1)]);
```

- A: `🤠`
- B: `🥰`
- C: `SyntaxError`
- D: `ReferenceError`

▼ **Answer**

Answer: B

With the `+=` operator, we're incrementing the value of `num` by `1`. `num` had the initial value `1`, so `1 + 1` is `2`. The item on the second index in the `list` array is 🥰, `console.log(list[2])` prints 🥰.

---

119. What's the output?

```javascript
const person = {
  firstName: 'Lydia',
  lastName: 'Hallie',
  pet: {
    name: 'Mara',
    breed: 'Dutch Tulip Hound',
  },
  getFullName() {
    return `${this.firstName} ${this.lastName}`;
  },
};

console.log(person.pet?.name);
console.log(person.pet?.family?.name);
console.log(person.getFullName?.());
console.log(member.getLastName?.());
```

- A: `undefined` `undefined` `undefined` `undefined`
- B: `Mara` `undefined` `Lydia Hallie` `ReferenceError`
- C: `Mara` `null` `Lydia Hallie` `null`
- D: `null` `ReferenceError` `null` `ReferenceError`

▼ **Answer**

Answer: B

With the optional chaining operator `?.`, we no longer have to explicitly check whether the deeper nested values are valid or not. If we're trying to access a property on an `undefined` or `null` value (*nullish*), the expression short-circuits and returns `undefined`.

`person.pet?.name`: `person` has a property named `pet`: `person.pet` is not nullish. It has a property called `name`, and returns `Mara`.
`person.pet?.family?.name`: `person` has a property named `pet`: `person.pet` is not nullish. `pet` does *not* have a property called `family`, `person.pet.family` is nullish. The expression returns `undefined`.
`person.getFullName?.()`: `person` has a property named `getFullName`: `person.getFullName()` is not nullish and can get invoked, which returns `Lydia Hallie`.
`member.getLastName?.()`: variable `member` is non-existent therefore a `ReferenceError` gets thrown!

---

120. What's the output?

```javascript
const groceries = ['banana', 'apple', 'peanuts'];

if (groceries.indexOf('banana')) {
  console.log('We have to buy bananas!');
} else {
  console.log(`We don't have to buy bananas!`);
}
```

- A: We have to buy bananas!
- B: We don't have to buy bananas
- C: `undefined`
- D: `1`

▼ **Answer**

Answer: B

We passed the condition `groceries.indexOf("banana")` to the if-statement. `groceries.indexOf("banana")` returns `0`, which is a falsy value. Since the condition in the if-statement is falsy, the code in the `else` block runs, and `We don't have to buy bananas!` gets logged.

---

121. What's the output?

```javascript
const config = {
  languages: [],
  set language(lang) {
    return this.languages.push(lang);
  },
};

console.log(config.language);
```

- A: `function language(lang) { this.languages.push(lang }`
- B: `0`
- C: `[]`
- D: `undefined`

▼ **Answer**

Answer: D

The `language` method is a `setter`. Setters don't hold an actual value, their purpose is to *modify* properties. When calling a `setter` method, `undefined` gets returned.

---

122. What's the output?

```javascript
const name = 'Lydia Hallie';

console.log(!typeof name === 'object');
console.log(!typeof name === 'string');
```

- A: `false` `true`
- B: `true` `false`
- C: `false` `false`
- D: `true` `true`

▼ **Answer**

Answer: C

`typeof name` returns `"string"`. The string `"string"` is a truthy value, so `!typeof name` returns the boolean value `false`. `false === "object"` and `false === "string"` both return`false`.

(If we wanted to check whether the type was (un)equal to a certain type, we should've written `!==` instead of `!typeof`)

---

123. What's the output?

```javascript
const add = x => y => z => {
  console.log(x, y, z);
  return x + y + z;
};

add(4)(5)(6);
```

- A: `4` `5` `6`
- B: `6` `5` `4`

- C: `4` `function` `function`
- D: `undefined` `undefined` `6`

▼ **Answer**

Answer: A

The `add` function returns an arrow function, which returns an arrow function, which returns an arrow function (still with me?). The first function receives an argument `x` with the value of `4`. We invoke the second function, which receives an argument `y` with the value `5`. Then we invoke the third function, which receives an argument `z` with the value `6`. When we're trying to access the value `x`, `y` and `z` within the last arrow function, the JS engine goes up the scope chain in order to find the values for `x` and `y` accordingly. This returns `4` `5` `6`.

---

124. What's the output?

```javascript
async function* range(start, end) {
  for (let i = start; i <= end; i++) {
    yield Promise.resolve(i);
  }
}

(async () => {
  const gen = range(1, 3);
  for await (const item of gen) {
    console.log(item);
  }
})();
```

- A: `Promise {1}` `Promise {2}` `Promise {3}`
- B: `Promise {<pending>}` `Promise {<pending>}` `Promise {<pending>}`
- C: `1` `2` `3`
- D: `undefined` `undefined` `undefined`

▼ **Answer**

Answer: C

The generator function `range` returns an async object with promises for each item in the range we pass: `Promise{1}`, `Promise{2}`, `Promise{3}`. We set the variable `gen` equal to the async object, after which we loop over it using a `for await ... of` loop. We set the variable `item` equal to the returned Promise values: first `Promise{1}`, then `Promise{2}`, then `Promise{3}`. Since we're *awaiting* the value of `item`, the resolved promise, the resolved *values* of the promises get returned: `1`, `2`, then `3`.

---

125. What's the output?

```javascript
const myFunc = ({ x, y, z }) => {
  console.log(x, y, z);
};

myFunc(1, 2, 3);
```

- A: `1` `2` `3`
- B: `{1: 1}` `{2: 2}` `{3: 3}`
- C: `{ 1: undefined }` `undefined` `undefined`
- D: `undefined` `undefined` `undefined`

▼ **Answer**

Answer: D

`myFunc` expects an object with properties `x`, `y` and `z` as its argument. Since we're only passing three separate numeric values (1, 2, 3) instead of one object with properties `x`, `y` and `z` ({x: 1, y: 2, z: 3}), `x`, `y` and `z` have their default value of `undefined`.

---

126. What's the output?

```javascript
function getFine(speed, amount) {
  const formattedSpeed = new Intl.NumberFormat('en-US', {
    style: 'unit',
    unit: 'mile-per-hour'
  }).format(speed);

  const formattedAmount = new Intl.NumberFormat('en-US', {
    style: 'currency',
    currency: 'USD'
  }).format(amount);

  return `The driver drove ${formattedSpeed} and has to pay ${formattedAmount}`;
}

console.log(getFine(130, 300))
```

- A: The driver drove 130 and has to pay 300
- B: The driver drove 130 mph and has to pay $300.00
- C: The driver drove undefined and has to pay undefined
- D: The driver drove 130.00 and has to pay 300.00

▼ Answer

Answer: B

With the `Intl.NumberFormat` method, we can format numeric values to any locale. We format the numeric value `130` to the `en-US` locale as a `unit` in `mile-per-hour`, which results in `130 mph`. The numeric value `300` to the `en-US` locale as a `currency` in `USD` results in `$300.00`.

---

127. What's the output?

```
const spookyItems = ['👻', '🎃', '🕸'];
({ item: spookyItems[3] } = { item: '💀' });

console.log(spookyItems);
```

- A: `["👻", "🎃", "🕸"]`
- B: `["👻", "🎃", "🕸", "💀"]`
- C: `["👻", "🎃", "🕸", { item: "💀" }]`
- D: `["👻", "🎃", "🕸", "[object Object]"]`

▼ Answer

Answer: B

By destructuring objects, we can unpack values from the right-hand object, and assign the unpacked value to the value of the same property name on the left-hand object. In this case, we're assigning the value "💀" to `spookyItems[3]`. This means that we're modifying the `spookyItems` array, we're adding the "💀" to it. When logging `spookyItems`, `["👻", "🎃", "🕸", "💀"]` gets logged.

---

128. What's the output?

```
const name = 'Lydia Hallie';
const age = 21;

console.log(Number.isNaN(name));
console.log(Number.isNaN(age));

console.log(isNaN(name));
console.log(isNaN(age));
```

- A: `true` `false` `true` `false`
- B: `true` `false` `false` `false`
- C: `false` `false` `true` `false`
- D: `false` `true` `false` `true`

▼ Answer

Answer: C

With the `Number.isNaN` method, you can check if the value you pass is a *numeric value* and equal to `NaN`. `name` is not a numeric value, so `Number.isNaN(name)` returns `false`. `age` is a numeric value, but is not equal to `NaN`, so `Number.isNaN(age)` returns `false`.

With the `isNaN` method, you can check if the value you pass is not a number. `name` is not a number, so `isNaN(name)` returns true. `age` is a number, so `isNaN(age)` returns `false`.

---

129. What's the output?

```
const randomValue = 21;

function getInfo() {
  console.log(typeof randomValue);
  const randomValue = 'Lydia Hallie';
}

getInfo();
```

- A: `"number"`
- B: `"string"`
- C: `undefined`
- D: `ReferenceError`

▼ **Answer**

Answer: D

Variables declared with the `const` keyword are not referenceable before their initialization: this is called the *temporal dead zone*. In the `getInfo` function, the variable `randomValue` is scoped in the functional scope of `getInfo`. On the line where we want to log the value of `typeof randomValue`, the variable `randomValue` isn't initialized yet: a `ReferenceError` gets thrown! The engine didn't go down the scope chain since we declared the variable `randomValue` in the `getInfo` function.

---

130. What's the output?

```
const myPromise = Promise.resolve('Woah some cool data');

(async () => {
  try {
    console.log(await myPromise);
  } catch {
    throw new Error(`Oops didn't work`);
  } finally {
    console.log('Oh finally!');
  }
})();
```

- A: `Woah some cool data`
- B: `Oh finally!`
- C: `Woah some cool data` `Oh finally!`
- D: `Oops didn't work` `Oh finally!`

▼ **Answer**

Answer: C

In the `try` block, we're logging the awaited value of the `myPromise` variable: `"Woah some cool data"`. Since no errors were thrown in the `try` block, the code in the `catch` block doesn't run. The code in the `finally` block *always* runs, `"Oh finally!"` gets logged.

---

131. What's the output?

```javascript
const emojis = ['🥑', ['✨', '✨', ['🍕', '🍕']]];

console.log(emojis.flat(1));
```

- A: `['🥑', ['✨', '✨', ['🍕', '🍕']]]`
- B: `['🥑', '✨', '✨', ['🍕', '🍕']]`
- C: `['🥑', ['✨', '✨', '🍕', '🍕']]`
- D: `['🥑', '✨', '✨', '🍕', '🍕']`

▼ **Answer**

Answer: B

With the `flat` method, we can create a new, flattened array. The depth of the flattened array depends on the value that we pass. In this case, we passed the value `1` (which we didn't have to, that's the default value), meaning that only the arrays on the first depth will be concatenated. `['🥑']` and `['✨', '✨', ['🍕', '🍕']]` in this case. Concatenating these two arrays results in `['🥑', '✨', '✨', ['🍕', '🍕']]`.

---

132. What's the output?

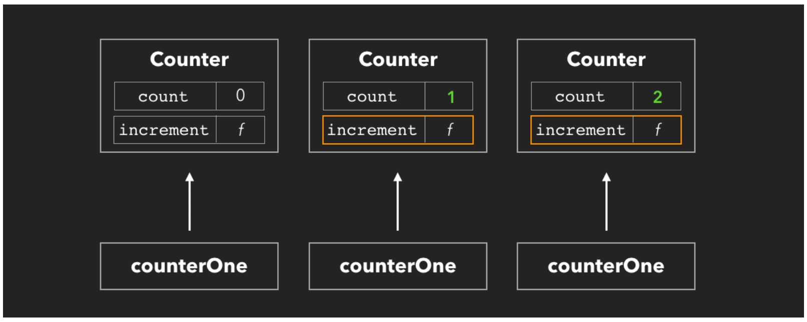
```javascript
class Counter {
  constructor() {
    this.count = 0;
  }

  increment() {
    this.count++;
  }
}

const counterOne = new Counter();
counterOne.increment();
counterOne.increment();

const counterTwo = counterOne;
counterTwo.increment();

console.log(counterOne.count);
```

- A: `0`
- B: `1`
- C: `2`
- D: `3`

▼ **Answer**

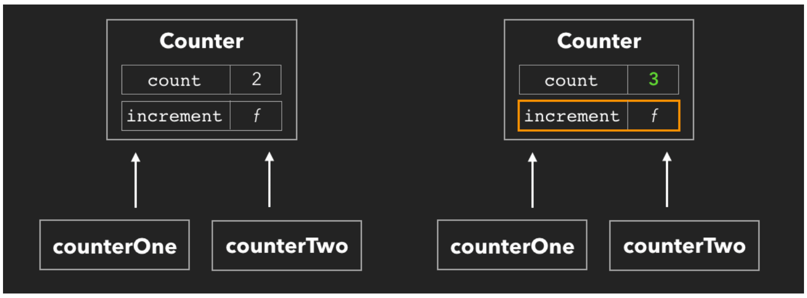
Answer: D

`counterOne` is an instance of the `Counter` class. The counter class contains a `count` property on its constructor, and an `increment` method. First, we invoked the `increment` method twice by calling `counterOne.increment()`. Currently, `counterOne.count` is `2`.



Then, we create a new variable `counterTwo`, and set it equal to `counterOne`. Since objects interact by reference, we're just creating a new reference to the same spot in memory that `counterOne` points to. Since it has the same spot in memory, any changes made to the object that `counterTwo` has a reference to, also apply to `counterOne`. Currently, `counterTwo.count` is `2`.

We invoke `counterTwo.increment()`, which sets `count` to `3`. Then, we log the count on `counterOne`, which logs `3`.



133. What's the output?

```
const myPromise = Promise.resolve(Promise.resolve('Promise'));

function funcOne() {
  setTimeout(() => console.log('Timeout 1!'), 0);
  myPromise.then(res => res).then(res => console.log(`${res} 1!`));
  console.log('Last line 1!');
}

async function funcTwo() {
  const res = await myPromise;
  console.log(`${res} 2!`)
  setTimeout(() => console.log('Timeout 2!'), 0);
  console.log('Last line 2!');
}

funcOne();
funcTwo();
```

- A: `Promise 1! Last line 1! Promise 2! Last line 2! Timeout 1! Timeout 2!`
- B: `Last line 1! Timeout 1! Promise 1! Last line 2! Promise2! Timeout 2!`
- C: `Last line 1! Promise 2! Last line 2! Promise 1! Timeout 1! Timeout 2!`
- D: `Timeout 1! Promise 1! Last line 1! Promise 2! Timeout 2! Last line 2!`

▼ **Answer**

Answer: C

First, we invoke `funcOne`. On the first line of `funcOne`, we call the *asynchronous* `setTimeout` function, from which the callback is sent to the Web API. (see my article on the event loop here.)

Then we call the `myPromise` promise, which is an *asynchronous* operation. Pay attention, that now only the first then clause was added to the microtask queue.

Both the promise and the timeout are asynchronous operations, the function keeps on running while it's busy completing the promise and handling the `setTimeout` callback. This means that `Last line 1!` gets logged first, since this is not an asynchonous operation.

Since the callstack is not empty yet, the `setTimeout` function and promise in `funcOne` cannot get added to the callstack yet.

In `funcTwo`, the variable `res` gets `Promise` because `Promise.resolve(Promise.resolve('Promise'))` is equivalent to `Promise.resolve('Promise')` since resolving a promise just resolves it's value. The `await` in this line stops the execution of the function until it receives the resolution of the promise and then keeps on running synchronously until completion, so `Promise 2!` and then `Last line 2!` are logged and the `setTimeout` is sent to the Web API. If the first then clause in `funcOne` had its own log statement, it would be printed before `Promise 2!`. However, it executed silently and put the second then clause in microtask queue. So, the second clause will be printed after `Promise 2!`.

Then the call stack is empty. Promises are *microtasks* so they are resolved first when the call stack is empty so `Promise 1!` gets to be logged.

Now, since `funcTwo` popped off the call stack, the call stack is empty. The callbacks waiting in the queue (`() => console.log("Timeout 1!")` from `funcOne`, and `() => console.log("Timeout 2!")` from `funcTwo`) get added to the call stack one by one. The first callback logs `Timeout 1!`, and gets popped off the stack. Then, the second callback logs `Timeout 2!`, and gets popped off the stack.

---

134. How can we invoke `sum` in `sum.js` from `index.js?`

```javascript
// sum.js
export default function sum(x) {
  return x + x;
}

// index.js
import * as sum from './sum';
```

- A: `sum(4)`
- B: `sum.sum(4)`
- C: `sum.default(4)`
- D: Default aren't imported with `*`, only named exports

▼ **Answer**

Answer: C

With the asterisk `*`, we import all exported values from that file, both default and named. If we had the following file:

```javascript
// info.js
export const name = 'Lydia';
export const age = 21;
export default 'I love JavaScript';
```

```
// index.js
import * as info from './info';
console.log(info);
```

The following would get logged:

```
{
  default: "I love JavaScript",
  name: "Lydia",
  age: 21
}
```

For the `sum` example, it means that the imported value `sum` looks like this:

```
{ default: function sum(x) { return x + x } }
```

We can invoke this function, by calling `sum.default`

---

135. What's the output?

```
const handler = {
  set: () => console.log('Added a new property!'),
  get: () => console.log('Accessed a property!'),
};

const person = new Proxy({}, handler);

person.name = 'Lydia';
person.name;
```

- A: `Added a new property!`
- B: `Accessed a property!`
- C: `Added a new property!` `Accessed a property!`
- D: Nothing gets logged

▼ **Answer**

Answer: C

With a Proxy object, we can add custom behavior to an object that we pass to it as the second argument. In this case, we pass the `handler` object which contains two properties: `set` and `get`. `set` gets invoked whenever we *set* property values, and `get` gets invoked whenever we *get* (access) property values.

The first argument is an empty object `{}`, which is the value of `person`. To this object, the custom behavior specified in the `handler` object gets added. If we add a property to the `person` object, `set` will get invoked. If we access a property on the `person` object, `get` gets invoked.

First, we added a new property `name` to the proxy object (`person.name = "Lydia"`). `set` gets invoked, and logs `"Added a new property!"`.

Then, we access a property value on the proxy object, and the `get` property on the handler object is invoked. `"Accessed a property!"` gets logged.

---

136. Which of the following will modify the `person` object?

```javascript
const person = { name: 'Lydia Hallie' };

Object.seal(person);
```

- A: `person.name = "Evan Bacon"`
- B: `person.age = 21`
- C: `delete person.name`
- D: `Object.assign(person, { age: 21 })`

▼ **Answer**

Answer: A

With `Object.seal` we can prevent new properties from being *added*, or existing properties to be *removed*.

However, you can still modify the value of existing properties.

---

137. Which of the following will modify the `person` object?

```javascript
const person = {
  name: 'Lydia Hallie',
  address: {
    street: '100 Main St',
  },
};

Object.freeze(person);
```

- A: `person.name = "Evan Bacon"`
- B: `delete person.address`
- C: `person.address.street = "101 Main St"`
- D: `person.pet = { name: "Mara" }`

▼ **Answer**

Answer: C

The `Object.freeze` method *freezes* an object. No properties can be added, modified, or removed.

However, it only *shallowly* freezes the object, meaning that only *direct* properties on the object are frozen. If the property is another object, like `address` in this case, the properties on that object aren't frozen, and can be modified.

---

138. What's the output?

```
const add = x => x + x;

function myFunc(num = 2, value = add(num)) {
  console.log(num, value);
}

myFunc();
myFunc(3);
```

- A: `2` `4` and `3` `6`
- B: `2` `NaN` and `3` `NaN`
- C: `2` `Error` and `3` `6`
- D: `2` `4` and `3` `Error`

▼ **Answer**

Answer: A

First, we invoked `myFunc()` without passing any arguments. Since we didn't pass arguments, `num` and `value` got their default values: num is `2`, and `value` is the returned value of the function `add`. To the `add` function, we pass `num` as an argument, which had the value of `2`. `add` returns `4`, which is the value of `value`.

Then, we invoked `myFunc(3)` and passed the value `3` as the value for the argument `num`. We didn't pass an argument for `value`. Since we didn't pass a value for the `value` argument, it got the default value: the returned value of the `add` function. To `add`, we pass `num`, which has the value of `3`. `add` returns `6`, which is the value of `value`.

---

139. What's the output?

```
class Counter {
  #number = 10

  increment() {
    this.#number++
  }
  getNum() {
    return this.#number
  }
}

const counter = new Counter()
counter.increment()

console.log(counter.#number)
```

- A: `10`
- B: `11`
- C: `undefined`
- D: `SyntaxError`

▼ **Answer**

Answer: D

In ES2020, we can add private variables in classes by using the `#`. We cannot access these variables outside of the class. When we try to log `counter.#number`, a SyntaxError gets thrown: we cannot access it outside the `Counter` class!

---

140. What's missing?

```
const teams = [
  { name: 'Team 1', members: ['Paul', 'Lisa'] },
  { name: 'Team 2', members: ['Laura', 'Tim'] },
];

function* getMembers(members) {
  for (let i = 0; i < members.length; i++) {
    yield members[i];
  }
}

function* getTeams(teams) {
  for (let i = 0; i < teams.length; i++) {
    // ✨ SOMETHING IS MISSING HERE ✨
  }
}

const obj = getTeams(teams);
```

```
obj.next(); // { value: "Paul", done: false }
obj.next(); // { value: "Lisa", done: false }
```

- A: `yield getMembers(teams[i].members)`
- B: `yield* getMembers(teams[i].members)`
- C: `return getMembers(teams[i].members)`
- D: `return yield getMembers(teams[i].members)`

▼ **Answer**

Answer: B

In order to iterate over the `members` in each element in the `teams` array, we need to pass `teams[i].members` to the `getMembers` generator function. The generator function returns a generator object. In order to iterate over each element in this generator object, we need to use `yield*`.

If we would've written `yield`, `return yield`, or `return`, the entire generator function would've gotten returned the first time we called the `next` method.

---

141. What's the output?

```
const person = {
  name: 'Lydia Hallie',
  hobbies: ['coding'],
};

function addHobby(hobby, hobbies = person.hobbies) {
  hobbies.push(hobby);
  return hobbies;
}

addHobby('running', []);
addHobby('dancing');
addHobby('baking', person.hobbies);

console.log(person.hobbies);
```

- A: `["coding"]`
- B: `["coding", "dancing"]`
- C: `["coding", "dancing", "baking"]`
- D: `["coding", "running", "dancing", "baking"]`

▼ **Answer**

Answer: C

The `addHobby` function receives two arguments, `hobby` and `hobbies` with the default value of the `hobbies` array on the `person` object.

First, we invoke the `addHobby` function, and pass `"running"` as the value for `hobby` and an empty array as the value for `hobbies`. Since we pass an empty array as the value for `hobbies`, `"running"` gets added to this empty array.

Then, we invoke the `addHobby` function, and pass `"dancing"` as the value for `hobby`. We didn't pass a value for `hobbies`, so it gets the default value, the `hobbies` property on the `person` object. We push the hobby `dancing` to the `person.hobbies` array.

Last, we invoke the `addHobby` function, and pass `"baking"` as the value for `hobby`, and the `person.hobbies` array as the value for `hobbies`. We push the hobby `baking` to the `person.hobbies` array.

After pushing `dancing` and `baking`, the value of `person.hobbies` is `["coding", "dancing", "baking"]`

---

142. What's the output?

```
class Bird {
  constructor() {
    console.log("I'm a bird. 🦢");
  }
}

class Flamingo extends Bird {
  constructor() {
    console.log("I'm pink. 🌸");
    super();
  }
}

const pet = new Flamingo();
```

- A: `I'm pink. 🌸`
- B: `I'm pink. 🌸` `I'm a bird. 🦢`
- C: `I'm a bird. 🦢` `I'm pink. 🌸`
- D: Nothing, we didn't call any method

▼ **Answer**

Answer: B

We create the variable `pet` which is an instance of the `Flamingo` class. When we instantiate this instance, the `constructor` on `Flamingo` gets called. First, `"I'm pink. 🌸"` gets logged, after which we call `super()`. `super()` calls the constructor of the parent class, `Bird`. The constructor in `Bird` gets called, and logs `"I'm a bird. 🦢"`.

143. Which of the options result(s) in an error?

```javascript
const emojis = ['🎄', '🎅', '🎁', '⭐'];

/* 1 */ emojis.push('🦌');
/* 2 */ emojis.splice(0, 2);
/* 3 */ emojis = [...emojis, '🥂'];
/* 4 */ emojis.length = 0;
```

- A: 1
- B: 1 and 2
- C: 3 and 4
- D: 3

▼ **Answer**

Answer: D

The `const` keyword simply means we cannot *redeclare* the value of that variable, it's *read-only*. However, the value itself isn't immutable. The properties on the `emojis` array can be modified, for example by pushing new values, splicing them, or setting the length of the array to 0.

144. What do we need to add to the `person` object to get `["Lydia Hallie", 21]` as the output of `[...person]`?

```javascript
const person = {
  name: "Lydia Hallie",
  age: 21
}

[...person] // ["Lydia Hallie", 21]
```

- A: Nothing, object are iterable by default
- B: `*[Symbol.iterator]() { for (let x in this) yield* this[x] }`
- C: `*[Symbol.iterator]() { yield* Object.values(this) }`
- D: `*[Symbol.iterator]() { for (let x in this) yield this }`

▼ **Answer**

Answer: C

Objects aren't iterable by default. An iterable is an iterable if the iterator protocol is present. We can add this manually by adding the iterator symbol `[Symbol.iterator]`, which has to return a generator object, for example by making it a generator function `*[Symbol.iterator]() {}`. This generator function has to yield the `Object.values` of the `person` object if we want it to return the array `["Lydia Hallie", 21]`: `yield* Object.values(this)`.

---

145. What's the output?

```javascript
let count = 0;
const nums = [0, 1, 2, 3];

nums.forEach(num => {
    if (num) count += 1
})

console.log(count)
```

- A: 1
- B: 2
- C: 3
- D: 4

▼ **Answer**

Answer: C

The `if` condition within the `forEach` loop checks whether the value of `num` is truthy or falsy. Since the first number in the `nums` array is `0`, a falsy value, the `if` statement's code block won't be executed. `count` only gets incremented for the other 3 numbers in the `nums` array, `1`, `2` and `3`. Since `count` gets incremented by `1` 3 times, the value of `count` is `3`.

---

146. What's the output?

```javascript
function getFruit(fruits) {
    console.log(fruits?.[1]?.[1])
}

getFruit([['🍊', '🍌'], ['🍍']])
getFruit()
getFruit([['🍍'], ['🍊', '🍌']])
```

- A: `null`, `undefined`, 🍌
- B: `[]`, `null`, 🍌
- C: `[]`, `[]`, 🍌
- D: `undefined`, `undefined`, 🍌

▼ **Answer**

Answer: D

The `?` allows us to optionally access deeper nested properties within objects. We're trying to log the item on index `1` within the subarray that's on index `1` of the `fruits` array. If the subarray on index `1` in the `fruits` array doesn't exist, it'll simply return `undefined`. If the subarray on index `1` in the `fruits` array exists, but this subarray doesn't have an item on its `1` index, it'll also return `undefined`.

First, we're trying to log the second item in the `['🍍']` subarray of `[['🍊', '🍌'], ['🍍']]`. This subarray only contains one item, which means there is no item on index `1`, and returns `undefined`.

Then, we're invoking the `getFruits` function without passing a value as an argument, which means that `fruits` has a value of `undefined` by default. Since we're conditionally chaining the item on index `1` of`fruits`, it returns `undefined` since this item on index `1` does not exist.

Lastly, we're trying to log the second item in the `['🍊', '🍌']` subarray of `['🍍'], ['🍊', '🍌']`. The item on index `1` within this subarray is `🍌`, which gets logged.

---

147. What's the output?

```
class Calc {
    constructor() {
        this.count = 0
    }

    increase() {
        this.count++
    }
}

const calc = new Calc()
new Calc().increase()

console.log(calc.count)
```

- A: `0`
- B: `1`
- C: `undefined`
- D: `ReferenceError`

▼ **Answer**

Answer: A

We set the variable `calc` equal to a new instance of the `Calc` class. Then, we instantiate a new instance of `Calc`, and invoke the `increase` method on this instance. Since the count property is within the constructor of the `Calc` class, the count property is not shared on the prototype of `Calc`. This means that the value of count has not been updated for the instance calc points to, count is still `0`.

---

148. What's the output?

```javascript
const user = {
    email: "e@mail.com",
    password: "12345"
}

const updateUser = ({ email, password }) => {
    if (email) {
        Object.assign(user, { email })
    }

    if (password) {
        user.password = password
    }

    return user
}

const updatedUser = updateUser({ email: "new@email.com" })

console.log(updatedUser === user)
```

- A: `false`
- B: `true`
- C: `TypeError`
- D: `ReferenceError`

▼ **Answer**

Answer: B

The `updateUser` function updates the values of the `email` and `password` properties on user, if their values are passed to the function, after which the function returns the `user` object. The returned value of the `updateUser` function is the `user` object, which means that the value of updatedUser is a reference to the same `user` object that `user` points to. `updatedUser === user` equals `true`.

---

149. What's the output?

```javascript
const fruit = ['🍌', '🍊', '🍎']

fruit.slice(0, 1)
fruit.splice(0, 1)
fruit.unshift('🍇')

console.log(fruit)
```

- A: `['🍌', '🍊', '🍎']`
- B: `['🍊', '🍎']`
- C: `['🍇', '🍊', '🍎']`
- D: `['🍇', '🍌', '🍊', '🍎']`

▼ **Answer**

Answer: C

First, we invoke the `slice` method on the fruit array. The slice method does not modify the original array, but returns the value that it sliced off the array: the banana emoji.
Then, we invoke the `splice` method on the fruit array. The splice method does modify the original array, which means that the fruit array now consists of `['🍊', '🍎']`.
At last, we invoke the `unshift` method on the `fruit` array, which modifies the original array by adding the provided value, '🍇' in this case, as the first element in the array. The fruit array now consists of `['🍇', '🍊', '🍎']`.

---

150. What's the output?

```javascript
const animals = {};
let dog = { emoji: '🐶' }
let cat = { emoji: '🐈' }

animals[dog] = { ...dog, name: "Mara" }
animals[cat] = { ...cat, name: "Sara" }

console.log(animals[dog])
```

- A: `{ emoji: "🐶", name: "Mara" }`
- B: `{ emoji: "🐈", name: "Sara" }`
- C: `undefined`
- D: `ReferenceError`

▼ **Answer**

Answer: B

Object keys are converted to strings.

Since the value of `dog` is an object, `animals[dog]` actually means that we're creating a new property called `"[object Object]"` equal to the new object. `animals["[object Object]"]` is now equal to

`{ emoji: "🐶", name: "Mara"}`.

`cat` is also an object, which means that `animals[cat]` actually means that we’re overwriting the value of `animals["[object Object]"]` with the new cat properties.

Logging `animals[dog]`, or actually `animals["[object Object]"]` since converting the `dog` object to a string results `"[object Object]"`, returns the `{ emoji: "🐈", name: "Sara" }`.

151. What's the output?

```javascript
const user = {
    email: "my@email.com",
    updateEmail: email => {
        this.email = email
    }
}

user.updateEmail("new@email.com")
console.log(user.email)
```

- A: `my@email.com`
- B: `new@email.com`
- C: `undefined`
- D: `ReferenceError`

▼ **Answer**

Answer: A

The `updateEmail` function is an arrow function, and is not bound to the `user` object. This means that the `this` keyword is not referring to the `user` object, but refers to the global scope in this case. The value of `email` within the `user` object does not get updated. When logging the value of `user.email`, the original value of `my@email.com` gets returned.

152. What's the output?

```javascript
const promise1 = Promise.resolve('First')
const promise2 = Promise.resolve('Second')
const promise3 = Promise.reject('Third')
const promise4 = Promise.resolve('Fourth')

const runPromises = async () => {
    const res1 = await Promise.all([promise1, promise2])
    const res2  = await Promise.all([promise3, promise4])
    return [res1, res2]
}

runPromises()
    .then(res => console.log(res))
    .catch(err => console.log(err))
```

- A: `[['First', 'Second'], ['Fourth']]`
- B: `[['First', 'Second'], ['Third', 'Fourth']]`
- C: `[['First', 'Second']]`
- D: `'Third'`

▼ **Answer**

Answer: D

The `Promise.all` method runs the passed promises in parallel. If one promise fails, the `Promise.all` method *rejects* with the value of the rejected promise. In this case, `promise3` is rejected with the value `"Third"`. We’re catching the rejected value in the chained `catch` method on the `runPromises` invocation to catch any errors within the `runPromises` function. Only `"Third"` gets logged, since `promise3` is rejected with this value.

---

153. What should the value of `method` be to log `{ name: "Lydia", age: 22 }`?

```
const keys = ["name", "age"]
const values = ["Lydia", 22]

const method = /* ?? */
Object[method](keys.map((_, i) => {
    return [keys[i], values[i]]
})) // { name: "Lydia", age: 22 }
```

- A: `entries`
- B: `values`
- C: `fromEntries`
- D: `forEach`

▼ **Answer**

Answer: C

The `fromEntries` method turns a 2d array into an object. The first element in each subarray will be the key, and the second element in each subarray will be the value. In this case, we’re mapping over the `keys` array, which returns an array that the first element is the item on the key array on the current index, and the second element is the item of the values array on the current index.

This creates an array of subarrays containing the correct keys and values, which results in `{ name: "Lydia", age: 22 }`

---

154. What's the output?

```
const createMember = ({ email, address = {}}) => {
    const validEmail = /.+\@.+\..+/.test(email)
    if (!validEmail) throw new Error("Valid email pls")

    return {
        email,
        address: address ? address : null
    }
}

const member = createMember({ email: "my@email.com" })
console.log(member)
```

- A: `{ email: "my@email.com", address: null }`
- B: `{ email: "my@email.com" }`
- C: `{ email: "my@email.com", address: {} }`
- D: `{ email: "my@email.com", address: undefined }`

▼ **Answer**

Answer: C

The default value of `address` is an empty object `{}`. When we set the variable `member` equal to the object returned by the `createMember` function, we didn't pass a value for the address, which means that the value of the address is the default empty object `{}`. An empty object is a truthy value, which means that the condition of the `address ? address : null` conditional returns `true`. The value of the address is the empty object `{}`.

---

155. What's the output?

```
let randomValue = { name: "Lydia" }
randomValue = 23

if (!typeof randomValue === "string") {
    console.log("It's not a string!")
} else {
    console.log("Yay it's a string!")
}
```

- A: `It's not a string!`

- B: `Yay it's a string!`
- C: `TypeError`
- D: `undefined`

▼ **Answer**

Answer: B

The condition within the `if` statement checks whether the value of `!typeof randomValue` is equal to `"string"`. The `!` operator converts the value to a boolean value. If the value is truthy, the returned value will be `false`, if the value is falsy, the returned value will be `true`. In this case, the returned value of `typeof randomValue` is the truthy value `"number"`, meaning that the value of `!typeof randomValue` is the boolean value `false`.

`!typeof randomValue === "string"` always returns false, since we're actually checking `false === "string"`. Since the condition returned `false`, the code block of the `else` statement gets run, and `Yay it's a string!` gets logged.