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

sayHi();
```

Sample Answer:
Actual Output : undefined (error: Reference Error)
Reason: For “let” variables, memory is not initialized. Age was USED before it was declared which is illegal for let variables.
Whereas for “var” variables, memory space is setup during the creation phase itself” [HOISTING]

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

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

Output:

```
3
3
3
0
1
2
```

Sample Answer:
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 . During each iteration, we will have a new value, and each value is scoped inside the loop

---

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

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

Output:

```
20
NaN
```

Sample Answer:

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 .

There is no value radius on that object, which returns NaN.

4.
```
let c = { greeting: 'Hey!' };
let d;
d = c;
c.greeting = 'Hello';
console.log(d.greeting);
```

Output:

```
Hello
```

Sample Answer:

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 we change one object, we change all of them.

5.
```
let a = 3;
let b = new Number(3);
let c = 3;

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

Output:

```
true
false
false
```

Sample Answer:
new Number() is a built-in function constructor. Although it looks like a number, it's not really a number: it has a bunch of extra features and is an object.

When we use the == operator, it only checks whether it has the same value. They both have the value of 3, so it returns true.

However, when we use the === operator, both value and type should be the same. It's not: new Number() is not a number, it's an object. Both return false.

```
6. 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'));
```

Output:

```
TypeError: freddie.colorChange is not a function
```

Sample Answer:
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. Since freddie is a child, the function is not passed down, and not available on the freddie instance: a TypeError is thrown.

```
7. function bark() {
console.log('Woof!');
bark.animal = 'dog';
```

Output:

```
Uncaught SyntaxError: Unexpected end of input
```

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 we write ourself isn’t the actual function. The function is an object with properties. This property is invocable.

---

8. 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());
```

Output:

TypeError: member.getFullName is not a function

Sample Answer:

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 we want a method to be available to all object instances, we have to add it to the prototype property:

---

9. 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);
```

Output:

Person { firstName: 'Lydia', lastName: 'Hallie' }

undefined

Sample Answer:

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.

---

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

Output:

0

2

2

Sample Answer:

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)