1)

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

sayHI();
```

Output:

```
undefined
console.log(age);
                        ^

ReferenceError: age is not defined
        at sayHi (/tmp/LwAMscBdU3.js:6:13)
        at Object.<anonymous> (/tmp/LwAMscBdU3.js:11:1)
        at Module._compile (internal/modules/cjs/loader.js:778:30)
        at Object.Module._extensions..js (internal/modules/cjs/loader.js:789:10)
        at Module.load (internal/modules/cjs/loader.js:653:32)
        at tryModuleLoad (internal/modules/cjs/loader.js:593:12)
        at Function.Module._load (internal/modules/cjs/loader.js:585:3)
        at Function.Module.runMain (internal/modules/cjs/loader.js:831:12)
        at startup (internal/bootstrap/node.js:283:19)
        at bootstrapNodeJSCore (internal/bootstrap/node.js:623:3)
```

Answer:

sayHi();

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)

```
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
```

Answer:
The important factor here is that var has a function scope. 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, it 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 keyword are block-scoped . During each iteration, 'i' 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
```

Answer:
Note that the value of diameter is a regular function, whereas the value of perimeter is an arrow function.This is a scope issue- the radius variable is not available inside the perimeter method and so is undefined.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). Changing the function to a regular method resolves the issue.

4)

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

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

Output:

```
Hello
```

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 you change one object, you 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
```

Answer:
Although new Number() looks like a number, its type is Object, so its different from the real number.
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:

```
console.log(freddie.colorChange('orange'));
                                ^

TypeError: freddie.colorChange is not a function
    at Object.<anonymous> (/tmp/LwAMscBdU3.js:13:21)
    at Module._compile (internal/modules/cjs/loader.js:778:30)
    at Object.Module._extensions..js (internal/modules/cjs/loader.js:789:10)
    at Module.load (internal/modules/cjs/loader.js:653:32)
    at tryModuleLoad (internal/modules/cjs/loader.js:593:12)
    at Function.Module._load (internal/modules/cjs/loader.js:585:3)
    at Function.Module.runMain (internal/modules/cjs/loader.js:831:12)
    at startup (internal/bootstrap/node.js:283:19)
    at bootstrapNodeJSCore (internal/bootstrap/node.js:623:3)
```

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. If a static method is called onn instance, TypeError error is thrown indicating that the method does not exist.

7)

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

Output:

Empty

Answer:

This is because the type of function is object!   Function actually has an attribute, which can be called.

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.

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:

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

TypeError: member.getFullName is not a function
        at Object.<anonymous> (/tmp/LwAMscBdU3.js:11:20)
        at Module._compile (internal/modules/cjs/loader.js:778:30)
        at Object.Module._extensions..js (internal/modules/cjs/loader.js:789:10)
        at Module.load (internal/modules/cjs/loader.js:653:32)
        at tryModuleLoad (internal/modules/cjs/loader.js:593:12)
        at Function.Module._load (internal/modules/cjs/loader.js:585:3)
        at Function.Module.runMain (internal/modules/cjs/loader.js:831:12)
        at startup (internal/bootstrap/node.js:283:19)
        at bootstrapNodeJSCore (internal/bootstrap/node.js:623:3)
```

Answer:
We can't directly add properties to constructors like regular objects, so the added properties cannot be used by new instances of new.

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
```

Answer:
Because new is used, a new object member is created, and then this in person points to member, so it becomes member.firstName = 'Lydia'
If new is not used, then this in person points to global, so it becomes global.firstName = 'sarah', but member1 has no value, so it is Undefined

10)

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

Output:

```
0
2
2
```

Answer:

The postfix unary operator ++:
Returns the value (this returns 0)
Increments the value (number is now 1)
The prefix unary operator ++:
Increments the value (number is now 2)
Returns the value (this returns 2)
This returns 0 2 2.