Question number 1

What are microtasks? What is a microtask queue? What is their role in Promises and how are they different from callbacks?

Answer-

MicroTask:

A microtask is a short function which is executed after the function or program which created it exits *and* only if the JavaScript execution stack is empty, but before returning control to the event loop being used by the user agent to drive the script's execution environment.

Microtasks are created by promises: an execution of .then/catch/finally handler becomes a microtask. Microtasks are used “under the cover” of await as well, as it’s another form of promise handling. Immediately after every macrotask, the engine executes all tasks from Microtask queue, prior to running any other Macrotasks or rendering or anything else. The queue is first-in-first-out: tasks enqueued first are run first. Execution of a task is initiated only when nothing else is running. Or, to say that simply, when a promise is ready, its .then/catch/finally handlers are put into the queue. They are not executed yet. JavaScript engine takes a task from the queue and executes it, when it becomes free from the current code.

Microtask Queue

Microtask Queue is like the Callback Queue, but Microtask Queue has higher priority. All the callback functions coming through *Promises* and *Mutation Observer* will go inside the Microtask Queue. For example, in the case of .fetch(), the callback function gets to the Microtask Queue. Promise handling always has higher priority so the JavaScript engine executes all the tasks from Microtask Queue and then moves to the Callback Queue.

Its role in Promises

The main reason to use microtasks is that: to ensure consistent ordering of tasks, even when results or data is available synchronously, but while simultaneously reducing the risk of user-discernible delays in operations.
The promise handling process is always asynchronous because all the promise actions pass through the internal microtask queue. The handlers such as .then/catch/finally should be called after the current code is over. To make sure that these handlers are executed, it is required to add them to the .then call. The

microtasks concepts in JavaScript are closely linked to the event loop macrotasks, as well.

How are they different from callbacks?

Tasks coming through promises and Mutation Observer are known as microtasks. While on the other hand tasks coming from setInterval, setInterval, setImmediate are known as callbacks/macrotasks/Tasks.

Macrotasks: setTimeout, setInterval, setImmediate, requestAnimationFrame, I/O, UI rendering
Microtasks: process.nextTick, Promises, queueMicrotask, MutationObserver

One go-around of the event loop will have **exactly one** task being processed from the **macrotask queue** (this queue is simply called the *task queue*. After this macrotask has finished, all available **microtasks** will be processed, namely within the same go-around cycle. While these microtasks are processed, they can queue even more microtasks, which will all be run one by one, until the microtask queue is exhausted.

If a **microtask** recursively queues other microtasks, it might take a long time until the next macrotask is processed. This means, you could end up with a blocked UI, or some finished I/O idling in your application.
However, at least concerning Node.js's process.nextTick function (which queues **microtasks**), there is an inbuilt protection against such blocking by means of process.maxTickDepth. This value is set to a default of 1000, cutting down further processing of **microtasks** after this limit is reached which allows the next **macrotask** to be processed)

Basically, use microtasks when you need to do stuff asynchronously in a synchronous way (i.e. when you would say *perform this (micro-)task in the most immediate future*). Otherwise, stick to **macrotasks.**

Question number 2

Explain with examples how private, protected variables can be implemented in classes and how can they be used in subclasses?

Answer-

The three major keywords at play are public, protected, and private.

1. **Public:** These members of the class are available to everyone that can access the (owner) class instance.

2. **Private**: These members are only accessible within the class that instantiated the object.
3. **Protected:** This keyword allows a little more access than private members but a lot less than the public. A protected member is accessible within the class (similar to private) and any object that inherits from it. A protected value is shared across all layers of the prototype chain. It is not accessible by anybody else.

Private

There are multiple ways of creating private variables in JavaScript. First is closures.

```
function carModule() {
    var speed = 0;

    return {
        accelerate: function () {
            return speed++;
        }
    }
}

var car = new carModule();
var redCar = new carModule()
console.log(car.accelerate()); // 0
console.log(car.accelerate()); // 1
console.log(redCar.accelerate()); // 0
console.log(redCar.accelerate()); // 1
console.log(car.accelerate()); // 2
console.log(redCar.accelerate()); // 2
console.log(speed); // speed is not defined
```

The second way is by using the # notation in a class.

```
class ObjectCreator {
    #meaningOfLife;

    constructor(name) {
        this.#meaningOfLife = 42;
    }

    returnMeaningOfLife() {
        return this.#meaningOfLife;
    }
```

```
    #returnAMessage() {
        return "You will do great things in life";
    }
}

const myObject = new ObjectCreator("Parwinder");
console.log(myObject.returnMeaningOfLife()); // 42
console.log(myObject["#meaningOfLife"]); // undefined
console.log(myObject.#meaningOfLife); // SyntaxError
console.log(myObject.#returnAMessage); // SyntaxError
```

The language enforces encapsulation. It is a syntax error to refer to # names from out of scope. Public and private fields do not conflict. We can have both private #meaningOfLife and public meaningOfLife fields in the same class.

The # method for declaring private members of a class is in part of ES2019/ES10.

Protected

Protected is the hardest of all 3 (private, public, protected data) to implement in JavaScript. The only way that I can think of doing this is by using a class that has a getter for a property without a setter. The property will be read-only, and any object will inherit it from the class, but it will only be change-able from within the class itself.

```
class NameGenerator {
    _name;

    constructor(name) {
        this._name = name;
    }

    get name() {
        return this._name;
    }
}

let nameGenerator = new NameGenerator("John");
console.log(`My name is ${nameGenerator.name}`); /* My name is John*/
nameGenerator.name = "Jane"; /* Cannot assign to 'name' because it is a read-only property. */
```

How can they be used in subclasses?

- To access private variables of parent class in subclass you can use protected or add getters and setters to private variables in parent class. You can't directly access any private variables of a class directly from outside. You can access private member's using **getter** and **setter**.
- When calling subclass on a constructor, a new class is formed that extends both the public and protected prototypes and makes them available to the subclass definition. It also stores a property called super that points to the parent class' respective prototypes for easy super method invocation. The protected key is also passed to the subclass allowing all instances of this class hierarchy to access it.