**Microtasks:**

Microtask is the one that will complete after our code block is completed. It is basically used to execute asynchronous tasks.

In javascript we have an event loop, and it checks two kinds of queues.

* Microtask Queue
* Callback Queue

We can execute Asynchronous code by using setTimeout, promises.

**If we use promises** =>

Once there is a promise the task is added to the microtask queue and once that is resolved then the task is kept into the microtask queue and once the call stack is empty then only the task will be executed.

**In case of Timeout** =>

The callback function will be kept aside with a timer, once the time is ended, then the callbacks of the setTimeout will be added to the callback queue and once the call stack is empty then they are executed.

**The difference** **=>** the Microtask queue is maintained for the promises, and they have higher priority to the callback queue. So after executing all the code once the call stack is empty then,

The Event loop will first check the microtask queue and if there is any task it will execute all them first.

Then it will check callback queue and if there are any task they will be executed then.

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

In JavaScript, we have are two types of object fields (properties and methods):

* Public: accessible from anywhere. They comprise the external interface.
* Private: accessible only from inside the class. These are for the internal interface.

To understand this we will take an example of coffee machine:

If we continue with the coffee machine – what’s hidden inside: a boiler tube, heating element, and so on – is its internal interface.

An internal interface is used for the object to work, its details use each other. For instance, a boiler tube is attached to the heating element.

But from the outside a coffee machine is closed by the protective cover, so that no one can reach those. Details are hidden and inaccessible. We can use its features via the external interface.

class CoffeeMachine {

waterAmount = 0; // the amount of water inside

constructor(power) {

this.power = power;

alert( `Created a coffee-machine, power: ${power}` );

}

}

// create the coffee machine

let coffeeMachine = new CoffeeMachine(100);

// add water

coffeeMachine.waterAmount = 200;

Here the properties of waterAmount and power are public. We can easily get/set them from the outside to any value. Let’s change waterAmount property to protected to have more control over it. For instance, we don’t want anyone to set it below zero.

**Protected**: In javascript Protected properties are usually prefixed with an underscore \_.

class CoffeeMachine {

\_waterAmount = 0;

set waterAmount(value) {

if (value < 0) {

value = 0;

}

this.\_waterAmount = value;

}

get waterAmount() {

return this.\_waterAmount;

}

constructor(power) {

this.\_power = power;

}

}

// create coffee machine

let coffeeMachine = new CoffeeMachine(100);

// add water

coffeeMachine.waterAmount = -10; // \_waterAmount will become 0, not -10

**Privates:** in Js privateshould start with #. They are only accessible from inside the class.

Here # is a special sign that the field is private. We can’t access it from outside or from inheriting classes.

Private fields do not conflict with public ones. We can have both private #waterAmount and public waterAmount fields at the same time.

class CoffeeMachine {

#waterAmount = 0;

get waterAmount() {

return this.#waterAmount;

}

set waterAmount(value) {

if (value < 0) value = 0;

this.#waterAmount = value;

}

}

let machine = new CoffeeMachine();

machine.waterAmount = 100;

alert(machine.#waterAmount); // Error