JS_Beginning_to_Mastery_Part3_1

April 21, 2023

0.1 Synchronous programming vs asynchronous programming

0.1.1 JS is synchronous programming language and single threaded.

0.1.2 Synchronous programming: One line of code is executed after the other. Synchronous programming is single threaded meaning that only one line of code is executed at a time

0.1.3 Asynchronous programming: One line of code is executed after the other, but not necessarily in order. Asynchronous programming is used to prevent the browser from freezing. Asynchronous programming is multi threaded meaning that multiple lines of code can be executed at the same time

Synchronous programming and asynchronous programming are two different programming paradigms in JavaScript.

In synchronous programming, each task is executed in sequence, one after the other. This means that if a task takes a long time to complete, the program will be blocked until that task is finished.

In asynchronous programming, on the other hand, multiple tasks can be executed concurrently, and the program doesn't wait for a task to finish before moving on to the next one. As a result, the program is not blocked, and it can continue to execute other tasks while waiting for a task to complete. Asynchronous programming is commonly used for tasks that involve I/O operations or network requests, which can take a long time to complete.

In JavaScript, asynchronous programming is typically implemented using callbacks, promises, or async/await syntax. Overall, asynchronous programming allows for more efficient and responsive programs, especially when dealing with tasks that involve waiting for external resources. However, it can be more complex to implement than synchronous programming, and it requires a good understanding of JavaScript's asynchronous mechanisms.

All the code we have written so far are synchronous.

```
// synchronous programming example
console.log("script start");

for(let i=0; i<10000; i++) {
    console.log("inside for loop");
}

console.log("script end");
```

In the code example above, the for loop section acts as a blocking section as it prevents `script end` to be printed in the console. This gets printed only after the for loop is finished. Here asynchronous concept will help. If an API takes time in getting loaded, then it will prevent other parts of the page to load in synchronous proggramming, here also asynchronous programming is of great help.

0.1.4 Asynchronous programming example

Call a function after 1 second

```
console.log("script start");
function hello(){
    console.log("inside setTimeout");
}
setTimeout(hello, 1000);
console.log("script end");
```

We can also pass an arrow function to `setTimeout`

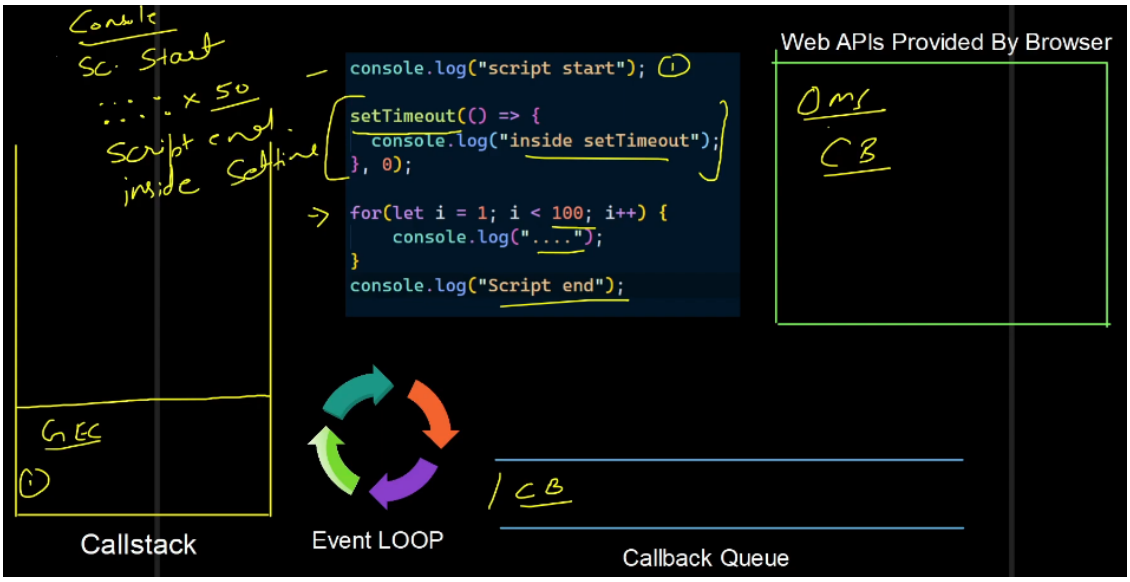
```
console.log("script start");
setTimeout(()=>{
    console.log("inside setTimeout");
}, 1000);
console.log("script end");
```

```
script start                                    130.js:30
script end                                      130.js:34
inside setTimeout                               130.js:32
```

We can see that `hello` gets printed after `script end` is printed.

```
console.log("script start");
setTimeout(()=>{
    console.log("inside setTimeout");
}, 0);

for(let i=0; i<100; i++){
    console.log("....");
}
console.log("script end");
```

```
script start                                    130.js:39
100 ....                                        130.js:45
script end                                      130.js:47
inside setTimeout                               130.js:41
```



`setTimeout()` returns an ID and we can store it in a variable.

```
console.log("script start");
const id = setTimeout(()=>{
    console.log("inside setTimeout");
}, 0);

for(let i=0; i<100; i++){
    console.log("....");
}
console.log("setTimeout id is: ", id);
console.log("script end");
```

```
script start                                    130.js:50
100 ....                                        130.js:56
setTimeout id is:  3                            130.js:58
script end                                      130.js:59
inside setTimeout                               130.js:52
```

Using the id of the setTimeout

```
console.log("script start");
const id = setTimeout(()=>{
    console.log("inside setTimeout");
}, 0);

for(let i=0; i<100; i++){
    console.log("....");
}
console.log("setTimeout id is: ", id);
console.log("clearing the timeout");
clearTimeout(id); // using id
console.log("script end");
```

0.1.5 setInterval

In JavaScript, `setInterval()` is a built-in method that allows you to repeatedly execute a specified function at a set interval of time. It takes two arguments: the function to be executed and the time interval in milliseconds.

Here's an example of how to use `setInterval()` to display a message every 1 second:

```
setInterval(() => {
  console.log("Hello, world!");
}, 1000);
```

In this example, the `setInterval()` function is used to call an anonymous function that logs the message `"Hello, world!"` to the console every 1 second (`1000` milliseconds).

`setInterval()` returns an ID value that can be used to stop the execution of the function using the `clearInterval()` method. For example, the following code stops the execution of the `setInterval()` function defined above after 5 seconds:

```
const intervalId = setInterval(() => {
  console.log("Hello, world!");
}, 1000);

setTimeout(() => {
  clearInterval(intervalId);
}, 5000);
```

In this example, the `setTimeout()` function is used to call the `clearInterval()` method after 5 seconds, passing in the `intervalId` returned by `setInterval()` as its argument. This stops the execution of the `setInterval()` function after 5 seconds.

`setInterval()` is commonly used for animations, real-time updates, and polling for new data from a server. However, it's important to use it with caution, as it can consume a lot of resources if the interval is set too low or if the function being executed takes a long time to complete.

0.1.6 callback function

callback function is a function that is passed as an argument to another function

In JavaScript, a callback is a function that is passed as an argument to another function, and is intended to be executed after some specific task or event has occurred.

Callbacks are commonly used in asynchronous programming, where a function takes some time to complete and does not block the execution of other parts of the program. In this scenario, a callback function is used to specify what should happen after the asynchronous task has completed.

For example, in the following code, the setTimeout function takes two arguments: a callback function and a time delay in milliseconds. The `setTimeout` function schedules the execution of the

callback function after the specified delay:

```
setTimeout(function() {
  console.log('This will be printed after 1000ms');
}, 1000);
```

In this case, the anonymous function passed as the first argument is the callback function that will be executed after 1000 milliseconds.

Callbacks can also be used to handle events, such as mouse clicks or keyboard presses. For instance, the `addEventListener` method allows you to register an event listener that executes a callback function when the event occurs:

```
document.addEventListener('click', function() {
  console.log('The document was clicked');
});
```

In this example, the anonymous function passed as the second argument is the callback function that will be executed when the `click` event occurs on the `document` object.

Callbacks are a fundamental concept in JavaScript and are widely used in modern web development, especially in combination with other asynchronous programming techniques such as promises and async/await.

[1]:
```
function myFunc(callback){
    console.log("Function is doing task 1");
    callback();
}

function myFunc2(){
    console.log("Function is doing task 2");
}

myFunc(myFunc2);
```

```
Function is doing task 1
Function is doing task 2
```

We can also write function inside function like the following:

```
myFunc(function(){
    console.log("function is doing task 2");
});
```

We can also use arrow function:

```
myFunc(()=> {
    console.log("Function inside is doing task 2");
})
```

[2]:
```
function getTwoNumbersAndAdd(number1, number2, callback){
    console.log(number1, number2);
    callback(number1, number2);
```

```
}

function addTwoNumbers(number1, number2){
    console.log(number1 + number2);
}

getTwoNumbersAndAdd(4, 5, addTwoNumbers);
```

```
4 5
9
```

0.1.7 Callback hell, pyramid of doom

we want first to change the text of the heading1 to “Heading 1” and then change the color of the heading1 to “violet” after 1 second, after that to change the text of the heading2 to “Heading 2” and then change the color of the heading2 to “purple” after 3 seconds, and so on.

```
const heading1 = document.querySelector('.heading1');
const heading2 = document.querySelector('.heading2');
const heading3 = document.querySelector('.heading3');
const heading4 = document.querySelector('.heading4');
const heading5 = document.querySelector('.heading5');
const heading6 = document.querySelector('.heading6');
const heading7 = document.querySelector('.heading7');
const heading8 = document.querySelector('.heading8');
const heading9 = document.querySelector('.heading9');
const heading10 = document.querySelector('.heading10');

setTimeout(()=>{
    heading1.textContent = "Heading1";
    heading1.style.color = "violet";
    setTimeout(()=>{
        heading2.textContent = "Heading2";
        heading2.style.color = "purple";
        setTimeout(()=>{
            heading3.textContent = "Heading3";
            heading3.style.color = "red";
            setTimeout(()=>{
                heading4.textContent = "Heading4";
                heading4.style.color = "pink";
                setTimeout(()=>{
                    heading5.textContent = "Heading5";
                    heading5.style.color = "green";
                    setTimeout(()=>{
                        heading6.textContent = "Heading6";
                        heading6.style.color = "blue";
                        setTimeout(()=>{
                            heading7.textContent = "Heading7";
                            heading7.style.color = "brown";
```

```
                    }, 1000);
                }, 3000);
            }, 2000);
        }, 1000);
    }, 2000);
}, 1000);
```

The above can also be done using function.

```
function changeText(element, text, color, time, onSuccessCallback, onFailureCallback){
    setTimeout(()=>{
        if(element){
            element.textContent = text;
            element.style.color = color;
            if (onSuccessCallback){
                onSuccessCallback();
            };
        } else{
            if(onFailureCallback){
                onFailureCallback();
            }
        }
    }, time)
}

// Pyramid of doom

changeText(heading1, "one", "green", 1000, ()=>{
    changeText(heading2, "two", "purple", 2000, ()=>{
        changeText(heading3, "three", "red", 1000, ()=>{
            changeText(heading4, "four", "pink", 1000, ()=>{
                changeText(heading5, "five", "green", 2000, ()=>{
                    changeText(heading6, "six", "blue", 1000, ()=>{
                        changeText(heading7, "seven", "brown", 1000, ()=>{
                            changeText(heading8, "eight", "cyan", 1000, ()=>{
                                changeText(heading9, "nine", "#cda562", 1000, ()=>{
                                    changeText(heading10, "ten", "#dca652", 1000, ()=>{

                                    }, ()=>{
                                        console.log("Heading9 doesn't exist")
                                    });
                                }, ()=>{
                                    console.log("Heading9 doesn't exist")
                                });
                            }, ()=>{
                                console.log("Heading8 doesn't exist")
                            });
```

```
                    }, ()=>{
                        console.log("Heading7 doesn't exist")
                    });
                }, ()=>{
                    console.log("Heading6 doesn't exist")
                });
            }, ()=>{
                console.log("Heading5 doesn't exist")
            });
        }, ()=>{
            console.log("Heading4 doesn't exist")
        });
    }, ()=>{
        console.log("Heading3 doesn't exist")
    });
}, ()=>{
    console.log("Heading2 doesn't exist")
});
}, ()=>{
    console.log("Heading1 doesn't exist")
});
```

Promise help us to avoid this nested structure.

0.1.8 Promise

In JavaScript, a Promise is an object that represents the eventual completion (or failure) of an asynchronous operation and its resulting value. It allows you to handle asynchronous operations in a more structured way, making your code more readable and maintainable.

Promises have three states:

- Pending: The initial state. The promise is neither fulfilled nor rejected.
- Fulfilled: The operation completed successfully, and the promise has a resulting value.
- Rejected: The operation failed, and the promise has a reason for the failure. When you create a new Promise, you pass a function that defines the asynchronous operation to be performed. This function takes two arguments: `resolve` and `reject`. If the operation is successful, you call `resolve` and pass the result value. If the operation fails, you call `reject` and pass an error object or message.

Here's an example of how to create and use a Promise:

```
const myPromise = new Promise((resolve, reject) => {
  // Perform an asynchronous operation, such as an API call
  // If the operation is successful, call resolve with the result
  // If the operation fails, call reject with an error object or message
});

myPromise.then(result => {
  // Handle the successful completion of the operation
```

```
}).catch(error => {
  // Handle the failure of the operation
});
```

In this example, `myPromise` is a new Promise that you create by passing a function with `resolve` and `reject` arguments. You can then use the `then()` method to handle the successful completion of the operation, and the `catch()` method to handle any errors that occur.

Promise can both be saved as a variable and returned as a function. We can rewrite the heading change code using Promise in the following way:

```
function changeText(element, text, color, time) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      if (element) {
        element.textContent = text;
        element.style.color = color;
        resolve();
      } else {
        reject("Element doesn't exist");
      }
    }, time);
  });
}

changeText(heading1, "one", "green", 1000)
  .then(() => changeText(heading2, "two", "purple", 2000))
  .then(() => changeText(heading3, "three", "red", 1000))
  .then(() => changeText(heading4, "four", "pink", 1000))
  .then(() => changeText(heading5, "five", "green", 2000))
  .then(() => changeText(heading6, "six", "blue", 1000))
  .then(() => changeText(heading7, "seven", "brown", 1000))
  .then(() => changeText(heading8, "eight", "cyan", 1000))
  .then(() => changeText(heading9, "nine", "#cda562", 1000))
  .then(() => changeText(heading10, "ten", "#dca652", 1000))
  .catch((error) => {
    console.log(error);
  });
```

Simple Promise example We promise to prepare fried rice if we have all the ingredients in the bucket

Produce promise

```
const friedRicePromise = new Promise((resolve, reject)=>{
    if (bucket.includes("vegetables") && bucket.includes("rice") && bucket.includes("salt") &&
        resolve("Fried Rice")
    } else{
        reject("No Fried Rice");
```

```
    }
});
```

Consume promise the argument passed to the resolve function is passed to the then function as we passed `Fried Rice` to the `resolve` function, the value of `myFriedRice` is `"Fried Rice"`.

```
friedRicePromise.then((myFriedRice)=>{
    // the argument passed to the resolve function is passed to the then function
    // as we passed "Fried Rice" to the resolve function, the value of myFriedRice
    // is "Fried Rice"
    console.log("let's eat ", myFriedRice);
})
```

Now, if we don't have all the elements, for example, missing rice, the promise will not be kept.

```
const bucket = ['coffee', 'chips', 'vegetables', 'salt']
```

We will see this error message: **Uncaught (in promise) No Fried Rice**

So, we can also catch this error by passing another callback function inside then in the following way:

```
friedRicePromise.then((myFriedRice)=>{
    console.log("let's eat ", myFriedRice);
}, (error)=>{console.log(error)})
```

Now, we will see no error, and a message **"No Fried Rice"** in the console.

```
const friedRicePromise = new Promise((resolve, reject)=>{
    if (bucket.includes("vegetables") && bucket.includes("rice") && bucket.includes("salt") &&
        resolve({value: "Fried Rice"}) // we can also pass an object or an array
    } else{
        reject(new Error("something missing from bucket"));
    }
});

friedRicePromise.then((myFriedRice)=>{
    console.log("let's eat ", myFriedRice);
}).catch((error)=>{console.log(error)})
```

will show the following if not all elements are not in the bucket:

```
Error: something missing from bucket                                    134.js:70
    at 134.js:64:16
    at new Promise (<anonymous>)
    at 134.js:60:26
```

Instead of this `reject(new Error("something missing from bucket"));`, we can also write `reject("something missing from bucket");` in the following way:

```
const friedRicePromise = new Promise((resolve, reject)=>{
    if (bucket.includes("vegetables") && bucket.includes("rice") && bucket.includes("salt") &&
        resolve({value: "Fried Rice"}) // we can also pass an object or an array
    } else{
```

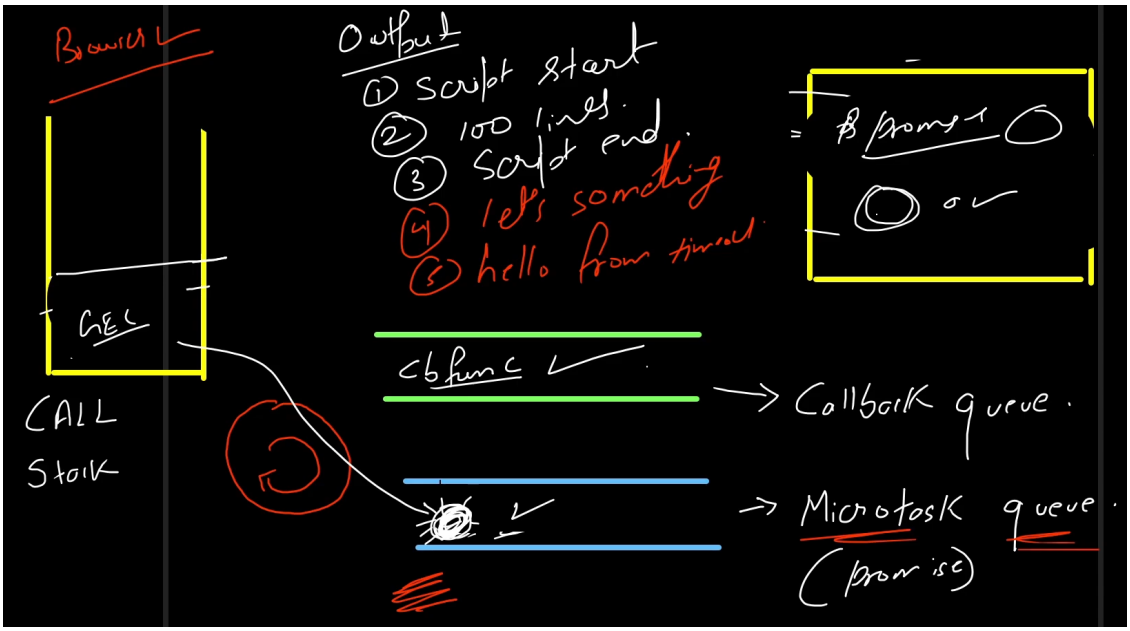
```
        reject("something missing from bucket"); // this is different
    }
});

friedRicePromise.then((myFriedRice)=>{
    console.log("let's eat ", myFriedRice);
}).catch((error)=>{console.log(error)})
```

```
something missing from bucket                                    134.js:86
>
```

Promise is not a JS feature, rather a feature of browser. Promise is executed by the browser. Promise is done asynchronously. Browser will consume promise. Promise is added in microtask queue. Steps under promises are sent to microtask queue and other asynchronous steps are sent for execution by callback queue. If there are actions to be performed both stacked in callback queue and microtask queue, microtask queue gets more priority and the task in microstask queue is moved to call stack queue first.

```
// Promise
console.log("script start");
const bucket = ['coffee', 'chips','vegetables','salt','rice'];



const friedRicePromise = new Promise((resolve,reject)=>{
    if(bucket.includes("vegetables")&& bucket.includes("salt") && bucket.includes("rice")){
        resolve({value:"friedrice"});
    }else{
        reject("could not do it");
    }
})

// produce



// consume
// how to consume

friedRicePromise.then(
    // jab promise resolve hoga
    (myfriedRice)=>{
    console.log("lets eat ", myfriedRice);
    }
    ).catch(
    (error)=>{
        console.log(error)
    })
```

```
setTimeout(()=>{
    console.log("hello from settimeout")
},0)

for(let i = 0; i <=100; i++){
    console.log(Math.random(), i);
}

console.log("script end!!!!")
```




```
0.802859518886889 87                                    134 2.js:40
0.2336602490976154 88                                   134 2.js:40
0.9806019129789179 89                                   134 2.js:40
0.3658045071582523 90                                   134 2.js:40
0.5858031750440631 91                                   134 2.js:40
0.7739992209078201 92                                   134 2.js:40
0.3761961700469083 93                                   134 2.js:40
0.8493964071839006 94                                   134 2.js:40
0.06786330434862986 95                                  134 2.js:40
0.13516532383643498 96                                  134 2.js:40
0.9130042616742517 97                                   134 2.js:40
0.3608363380175552 98                                   134 2.js:40
0.2137275353668675 99                                   134 2.js:40
0.6877394741870324 100                                  134 2.js:40
script end!!!!                                          134 2.js:43
lets eat ▶ {value: 'friedrice'}                         134 2.js:27
hello from settimeout                                   134 2.js:36
```

0.1.9 Function returning promise

```
// function returning promise
```

```
function friedRicePromise(){
    const bucket = ['coffee', 'chips', 'vegetables', 'salt', 'rice'];
    return new Promise((resolve, reject) => {
        if (bucket.includes("vegetables") && bucket.includes("rice") && bucket.includes("salt")
            resolve({value: "Fried Rice"});
        } else{
            reject("Not enough ingredients");
        }
    })
}

friedRicePromise().then((myFriedRice)=>{
    console.log("let's eat ", myFriedRice);
}).catch((error)=>{
    console.log(error);
});
```

0.1.10 AJAX

AJAX (Asynchronous JavaScript and XML) is a technique used in JavaScript programming to send and receive data from a web server without reloading the entire page. This allows for faster, more interactive and dynamic web applications.

To implement AJAX in JavaScript, you need to use the XMLHttpRequest (XHR) object, which is built into most modern browsers. Here's an example of how to create an XHR object:

```
var xhr = new XMLHttpRequest();
```

Once you have created an XHR object, you can use it to make an HTTP request to the server using the `open()` method. Here's an example of how to make a `GET` request to a server:

```
xhr.open('GET', '/data.json', true);
```

The first parameter of the open() method specifies the HTTP method (`GET`, `POST`, `PUT`, `DELETE`, etc.), the second parameter specifies the URL of the server, and the third parameter specifies whether the request should be asynchronous (`true`) or synchronous (`false`).

Next, you need to define a callback function that will be called when the server responds to the request. This function is usually defined using the onreadystatechange event. Here's an example:

```
xhr.onreadystatechange = function() {
  if (xhr.readyState == 4 && xhr.status == 200) {
    var data = JSON.parse(xhr.responseText);
    // Do something with the data
  }
};
```

The readyState property of the XHR object indicates the current state of the request, which can have the following values:

- 0: uninitialized
- 1: loading

- 2: loaded
- 3: interactive
- 4: complete The status property indicates the HTTP status code returned by the server.

Finally, you can send the request to the server using the `send()` method:

```
xhr.send();
```

Here's a complete example that demonstrates how to use AJAX to load data from a server and display it on a web page:

```
var xhr = new XMLHttpRequest();
xhr.open('GET', '/data.json', true);
xhr.onreadystatechange = function() {
  if (xhr.readyState == 4 && xhr.status == 200) {
    var data = JSON.parse(xhr.responseText);
    var list = document.getElementById('list');
    for (var i = 0; i < data.length; i++) {
      var item = document.createElement('li');
      item.innerHTML = data[i].name + ': ' + data[i].value;
      list.appendChild(item);
    }
  }
};
xhr.send();
```

In this example, the code loads data from a JSON file called '`data.json`', creates a list of items from the data, and appends the list to an HTML element with an id of `'list'`.

Here are some more examples and use cases for AJAX in JavaScript:

1. **Form submission:** You can use AJAX to submit a form to the server without reloading the entire page. This allows for a smoother user experience and can save time by not requiring the user to wait for the page to reload. Here's an example:

```
var form = document.getElementById('myForm');
form.addEventListener('submit', function(event) {
  event.preventDefault();
  var xhr = new XMLHttpRequest();
  xhr.open('POST', '/submit', true);
  xhr.setRequestHeader('Content-Type', 'application/x-www-form-urlencoded');
  xhr.onreadystatechange = function() {
    if (xhr.readyState == 4 && xhr.status == 200) {
      var response = JSON.parse(xhr.responseText);
      // Do something with the response
    }
  };
  xhr.send(new FormData(form));
});
```

In this example, the code prevents the default form submission behavior, creates an XHR object, sets the HTTP method to POST, sets the content type header to 'application/x-www-form-urlencoded',

and sends the form data as a FormData object. The server can then process the form data and return a response that can be handled in the callback function.

2. Dynamic content loading: You can use AJAX to load dynamic content into a web page without reloading the entire page. This allows for a more interactive and dynamic user experience. Here's an example:

```
var button = document.getElementById('loadButton');
button.addEventListener('click', function() {
  var xhr = new XMLHttpRequest();
  xhr.open('GET', '/data.json', true);
  xhr.onreadystatechange = function() {
    if (xhr.readyState == 4 && xhr.status == 200) {
      var data = JSON.parse(xhr.responseText);
      var content = document.getElementById('content');
      content.innerHTML = data.content;
    }
  };
  xhr.send();
});
```

In this example, the code creates an XHR object, sets the HTTP method to GET, and sends a request to the server to load data from a JSON file. When the server responds with the data, the callback function replaces the content of an HTML element with an id of 'content' with the new data.

3. Autocomplete search: You can use AJAX to implement an autocomplete search feature that suggests search terms as the user types. This can improve the user experience and help the user find what they are looking for more quickly. Here's an example:

```
var input = document.getElementById('searchInput');
input.addEventListener('input', function() {
  var xhr = new XMLHttpRequest();
  xhr.open('GET', '/search?q=' + encodeURIComponent(input.value), true);
  xhr.onreadystatechange = function() {
    if (xhr.readyState == 4 && xhr.status == 200) {
      var suggestions = JSON.parse(xhr.responseText);
      // Display the suggestions to the user
    }
  };
  xhr.send();
});
```

In this example, the code creates an XHR object, sets the HTTP method to GET, and sends a request to the server with a query parameter 'q' that contains the user's search term. The server can then return a list of suggested search terms based on the user's input, which can be displayed to the user in real-time.

`encodeURIComponent()` is a built-in JavaScript function that encodes a string as a valid URI (Uniform Resource Identifier) component by replacing all special characters with their corresponding encoded values.

In the context of this AJAX example, `encodeURIComponent(input.value)` is used to encode the user's search term before it is included in the URL query parameter. This is necessary because URL query parameters must be encoded to avoid any special characters or reserved characters that can disrupt the query.

For example, if the user enters the search term "pizza toppings", the resulting URL with the encoded query parameter would be `/search?q=pizza%20toppings`. The `%20` represents a space character that has been encoded.

Without encoding the search term, the resulting URL could look like `/search?q=pizza toppings`, which is not a valid URL and would result in errors. By encoding the search term with `encodeURIComponent()`, we ensure that the resulting URL is valid and can be sent to the server for processing.

0.1.11 AJAX : asynchronous javascript and XML

- HTTP request: is a set of "web development techniques" using many web technologies on the "client-side" to create asynchronous web applications. Browser is considered as client.

With Ajax, web applications can send and retrieve data from a server asynchronously (in the background) without interfering with the display and behaviour of the existing page.

We don't use data in XML format anymore, we use JSON now.

We have 3 most common ways to create and send request to server

1. xmlHTTPRequest (old way of doing)
2. fetch API (new way of doing)
3. axios (this is third party library)

XHR — XMLHttpRequest JSON is different from JS object in these ways: a. JSON key has to be within " " and there can't be any methods.

In AJAX, the onreadystatechange attribute is a callback function that gets called every time the readyState property of the XMLHttpRequest object changes. The readyState property indicates the state of the request, and can have the following values:

- 0: UNSENT - the XMLHttpRequest object has been created, but open() method has not been called yet.
- 1: OPENED - the open() method has been called, but send() method has not been called yet.
- 2: HEADERS_RECEIVED - the send() method has been called, and the server has responded with the headers of the response.
- 3: LOADING - the response body is being received. This value is usually not used in practice.
- 4: DONE - the entire response has been received and is available.

So, the values of 2, 3, and 4 for the `onreadystatechange` attribute correspond to the readyState values of `HEADERS_RECEIVED`, `LOADING`, and `DONE`, respectively.

When the `readyState` value changes to 2 (i.e., `HEADERS_RECEIVED`), the `onreadystatechange` callback function can access the response headers using the `getAllResponseHeaders()` or `getResponseHeader()` methods of the XMLHttpRequest object.

When the `readyState` value changes to 3 (i.e., `LOADING`), the `onreadystatechange` callback function can access the partial response using the `responseText` or `responseXML` properties of the XMLHttpRequest object.

When the `readyState` value changes to 4 (i.e., `DONE`), the `onreadystatechange` callback function can access the entire response using the `responseText` or `responseXML` properties of the XMLHttpRequest object.

HTTP status code All HTTP response status codes are separated into five classes or categories. The first digit of the status code defines the class of response, while the last two digits do not have any classifying or categorization role. There are five classes defined by the standard:

- 1xx informational response – the request was received, continuing process
- 2xx successful – the request was successfully received, understood, and accepted
- 3xx redirection – further action needs to be taken in order to complete the request
- 4xx client error – the request contains bad syntax or cannot be fulfilled
- 5xx server error – the server failed to fulfil an apparently valid request

```
const URL = "https://jsonplaceholder.typicode.com/posts";
const xhr = new XMLHttpRequest();
// console.log(xhr);
// console.log(xhr.readyState);
// console.log(xhr.readyState); // 0
xhr.open("GET", URL); // browser will do this asynchronously
// console.log(xhr.readyState); // 1

xhr.onreadystatechange = function(){
    // console.log(xhr.readyState);
    if(xhr.readyState === 4){
        // console.log(xhr.response);
        // console.log(typeof xhr.response);
        console.log(xhr.status); // gives 200
        const response = xhr.response;
        const data = JSON.parse(response); // converts to JS object
        console.log(typeof data); // object
    }
}

xhr.send();
```

Instead of the above code, we can use the below code using `onload` which runs only when the readyState is 4

```
const URL = "https://jsonplaceholder.typicode.com/posts";
const xhr = new XMLHttpRequest();
// console.log(xhr);
// console.log(xhr.readyState);
// console.log(xhr.readyState); // 0
xhr.open("GET", URL); // browser will do this asynchronously
// console.log(xhr.readyState); // 1
```

```
xhr.onload = function(){
    const response =  xhr.response;
    const data = JSON.parse(response);
    console.log(data);
}

xhr.send();
```

Example of using XHR

```
const URL = "https://jsonplaceholder.typicode.com/posts";

const xhr = new XMLHttpRequest();

xhr.open("GET", URL);

xhr.onload = function(){
    if (xhr.status >= 200 && xhr.status < 300){
        const data = JSON.parse(xhr.response);
        console.log(data);
        const id = data[3].id;
        const xhr2 = new XMLHttpRequest();
        const URL2 = `${URL}/${id}`;
        console.log(URL2);
        xhr2.open("GET", URL2);
        xhr2.onload = ()=>{
            const data2 = JSON.parse(xhr2.response);
            console.log(data2);
        }
        xhr2.send();
    }
    else{
        console.log("something went wrong");
    }
}
```

Now, we will do the same as before using `Promise`

```
const URL = "https://jsonplaceholder.typicode.com/posts";

function sendRequest(method, url) {
    return new Promise(function(resolve, reject) {
        const xhr  = new XMLHttpRequest();
        xhr.open(method, url);
        xhr.onload = function() {
            if(xhr.status >= 200 && xhr.status < 300){
                resolve(xhr.response);
            }
            else{
```

```
                reject(new Error("Something Went wrong"));
            }
        }

        xhr.onerror = function() {
            reject(new Error("Something went wrong"));
        }

        xhr.send();
    })
}

sendRequest("GET", URL)
    .then(response => {
        const data = JSON.parse(response);
        // console.log(data)
        return data;
    })
    .then(data=>{
        const id = data[3].id;
        return id;
    })
    .then(id=>{
        const url = `${URL}/${id}ssss`;
        return sendRequest("GET", url);
    })
    .then(newResponse => {
        const newData = JSON.parse(newResponse);
        console.log(newData);
    })
    .catch(error =>{
        console.log(error);
    })
```

0.1.12 fetch – it does the GET request by default

fetch returns Promise

In JavaScript, `fetch()` is a built-in function that is used to make network requests to a server and retrieve data asynchronously. It returns a Promise that resolves to the Response object representing the response to the request.

Here is an example of how to use `fetch()` to make a GET request and retrieve data from a server:

```
fetch('https://example.com/data.json')
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(error => console.error(error));
```

In this example, `fetch()` is called with the URL of a JSON file on a server. The Promise returned

by `fetch()` resolves to a Response object, which is passed to the first `.then()` method. The `json()` method of the Response object is called to parse the response as JSON, and the resulting data is passed to the second `.then()` method. Finally, any errors that occur during the request are caught by the `.catch()` method and logged to the console.

`fetch()` can also be used to make other types of requests, such as POST or PUT requests, and to include additional options such as headers or authentication credentials.

```
// fetch

const URL = "https://jsonplaceholder.typicode.com/posts";

fetch(URL,{
    method: 'POST',
    body: JSON.stringify({
        title: 'foo',
        body: 'bar',
        userId: 1,
    }),
    headers: {
        'Content-type': 'application/json; charset=UTF-8',
    },
    })
    .then(response => {
        if(response.ok){
            return response.json();
        } else{
            throw new Error("Something went wrong!!!");
        }
    })
    .then(data=>{
        console.log(data);
    })
    .catch(error=>{
        console.log("inside catch");
        console.log(error);
    })
```

0.1.13 `async await`

fetch followed by multiple then can be replaced by using `async await`

`async/await` is a modern approach for handling asynchronous operations in JavaScript. It provides a way to write asynchronous code that looks like synchronous code and is easier to read and understand.

The `async` keyword is used to define a function that returns a Promise. When a function is marked as `async`, it allows us to use the `await` keyword inside it to wait for the resolution of a Promise.

Here's an example:

```
async function getData() {
  const response = await fetch('https://example.com/data.json');
  const data = await response.json();
  return data;
}

getData()
  .then(data => console.log(data))
  .catch(error => console.error(error));
```

In this example, `getData()` is an async function that fetches data from a server using fetch() and returns it as a Promise. The `await` keyword is used to wait for the response to be returned from the server before parsing it as JSON using the `json()` method. The parsed data is then returned from the function.

Note that the `await` keyword can only be used inside an async function.

Using `async/await` can make code more readable and easier to understand, especially when dealing with complex asynchronous operations that involve multiple requests and callbacks.

One thing to keep in mind when using `async/await` is error handling. If an error occurs in an `async function`, it will automatically be caught and converted into a rejected Promise. To handle errors, we can use a `try/catch` block like this:

```
async function getData() {
  try {
    const response = await fetch('https://example.com/data.json');
    const data = await response.json();
    return data;
  } catch (error) {
    console.error(error);
    throw new Error('Failed to fetch data');
  }
}
```

In this example, we use a `try/catch` block to catch any errors that occur during the execution of the function. If an error occurs, it is logged to the console and a new `Error` object is thrown with a custom error message. The `catch` block can also be used to perform other error handling tasks, such as displaying an error message to the user or logging the error to a remote service.

```
const URL = "https://jsonplaceholder.typicode.com/posts";

// writing async before function makes it return a promise
async function getPosts(){

}

const returned = getPosts();
console.log(returned);
```

```
▼ Promise i                                                    144.js:10
  ▶ [[Prototype]]: Promise
    [[PromiseState]]: "fulfilled"
    [[PromiseResult]]: undefined
```

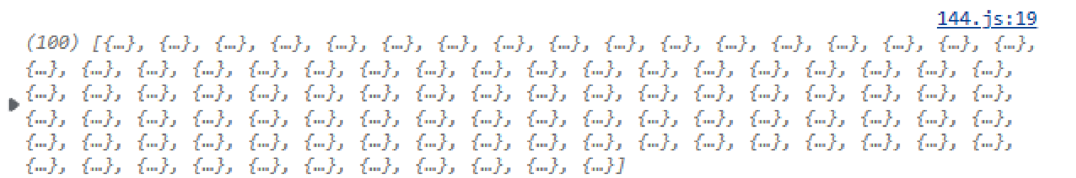
```
const URL = "https://jsonplaceholder.typicode.com/posts";

// writing async before function makes it return a promise
async function getPosts(){
    // fetch(URL) returns a promise
    // we can use the term await before a promise to wait for it to resolve
    const response = await fetch(URL);
    if(!response.ok){
        throw new Error("Something went wrong!");
    }
    // as json method also returns a promise, we can use await before it
    const data = await response.json();
    return data; // we are returning a promise here
}

getPosts()
.then(myData => {
    console.log(myData);
})
.catch(error=>{
    console.log("inside catch");
    console.log(error);
})
```

```
                                                               144.js:19
(100) [{…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…},
{…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…},
▶ {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…},
{…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…},
{…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…},
{…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}]
```

Writing async function using arrow function

```
// Writing async function using arrow function

const URL = "https://jsonplaceholder.typicode.com/posts";

// writing async before function makes it return a promise
const getPosts = async()=>{
    // fetch(URL) returns a promise
    // we can use the term await before a promise to wait for it to resolve
    const response = await fetch(URL);
    if(!response.ok){
        throw new Error("Something went wrong!");
    }
```

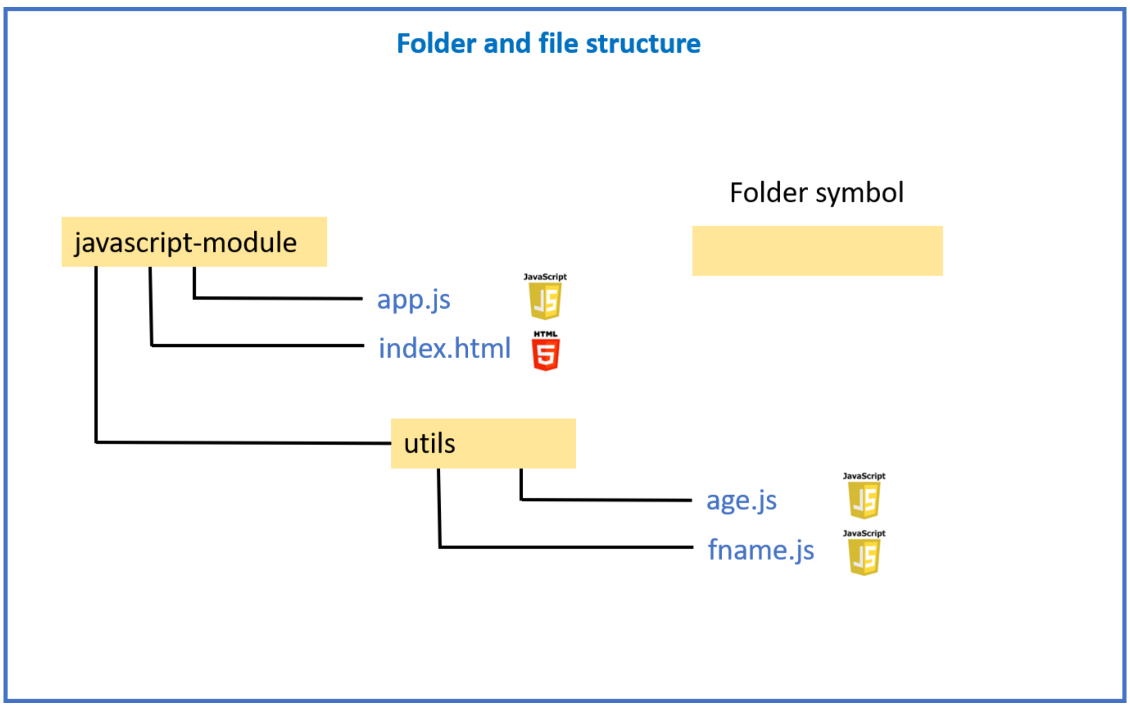
```javascript
    // as json method also returns a promise, we can use await before it
    const data = await response.json();
    return data; // we are returning a promise here
}

getPosts()
.then(myData => {
    console.log(myData);
})
.catch(error=>{
    console.log("inside catch");
    console.log(error);
})
```



We can split our code to multiple files using ES6.

0.1.14 Importing variables defined in multiple JS files

Suppose the folder and file structure we have looks like the following:



The code of **index.html** is like the following:

Note the use of `type="module"` inside `script` tag and dropping defer term.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <script src="./app.js" type="module"></script>
    <title>Document</title>
</head>
<body>
    <h1>JavaScript Module</h1>
</body>
</html>
```

The code of **age.js** is the following. Note the use of `export` before the variable declaration.

```
export const age = 31;
```

The code of **fname.js** is the following:

```
export const firstName = "John";
```

In the **app.js**, we import `firstName` variable from **fname.js** and `age` variable from **age.js** in the following way:

```
// we want to import firstName from fname.js and age from age.js

import {firstName} from './utils/fname.js';
import {age} from './utils/age.js';

console.log(firstName, age);
```

We can also use `export` after declaring a variable. The code of **age.js** is the following. Note the use of `export` after the variable declaration in a separate line.

```
const age = 31;
export {age};
```

We can also shorten the variable name after importing in the JS file and use this shortened name for variable. The code for **app.js** can be also written as:

```
import {firstName as fname} from './utils/fname.js';
import {age} from './utils/age.js';

console.log(fname, age);
```

We can export anything, not only variables.

Suppose, inside **utils** subfolder, we have another JS file named **Person.js**.

```
export class Person{
    constructor(firstName, lastName, age){
        this.firstName = firstName;
        this.lastName = lastName;
```

```
        this.age = age;
    }

    info(){
        console.log(this.firstName, this.lastName, this.age)
    }
}
```

Now, we change the file **app.js** to the following:

```
import {firstName} from './utils/fname.js';
import {age} from './utils/age.js';
import {Person} from './utils/Person.js';

console.log(firstName, age);

const person = new Person("John", "Doe", 31);
person.info();
console.log(person);
```

If we open **index.html** and look at the console, we will see the following:

Inside the **Person.js**, if we write `export default class Person{ ... }` instead of `export class Person{ ... }`, we can import class `Person` without curly braces inside **app.js**.

In this case, the code for **Person.js** looks like the following:

```
export default class Person{
    constructor(firstName, lastName, age){
        this.firstName = firstName;
        this.lastName = lastName;
        this.age = age;
    }

    info(){
        console.log(this.firstName, this.lastName, this.age)
    }
}
```

In this case, the code for **app.js** will look like the following:

```
import {firstName} from './utils/fname.js';
import {age} from './utils/age.js';
import Person from './utils/Person.js';
```

```
console.log(firstName, age);

const person = new Person("John", "Doe", 31);
person.info();
console.log(person);
```

Remember that more than one class or variable can't use export default in a single JS file. One file allows only one `export default`.

Suppose, we have more than one functions to export and we want at least one to have `export default`, then the rest have to use `export` only like the following (**Person.js**):

```
export default class Person{
    constructor(firstName, lastName, age){
        this.firstName = firstName;
        this.lastName = lastName;
        this.age = age;
    }

    info(){
        console.log(this.firstName, this.lastName, this.age)
    }
}

export class Person2{
    constructor(firstName, lastName, age){
        this.firstName = firstName;
        this.lastName = lastName;
        this.fullName = `${firstName} ${lastName}`;
        this.age = age;
    }

    info(){
        console.log(this.fullName, this.age)
    }
}
```

Now, the file **app.js** will look like this:

```
// we want to import firstName from fname.js and age from age.js

import {firstName} from './utils/fname.js';
import {age} from './utils/age.js';
import Person from './utils/Person.js';
import { Person2 } from './utils/Person.js';

console.log(firstName, age);

const person = new Person("John", "Doe", 31);
person.info();
```

```
console.log(person);

const person2 = new Person2("Milon", "Shah", 21);
person2.info();
console.log(person2);
```

```
John 31                                                                    app.js:8
John Doe 31                                                                Person.js:9
▶ Person {firstName: 'John', lastName: 'Doe', age: 31}                     app.js:12
Milon Shah 21                                                              Person.js:22
▶ Person2 {firstName: 'Milon', lastName: 'Shah', fullName: 'Milon Shah', age: 21}   app.js:16
```

Instead of writing {Person} and {Person2} in two lines as shown before:

```
import Person from './utils/Person.js';
import { Person2 } from './utils/Person.js';
```

We can also write them in a single line like this `import Person, {Person2} from './utils/Person.js';`

Now, suppose the file **Person.js** has three classes like the following:

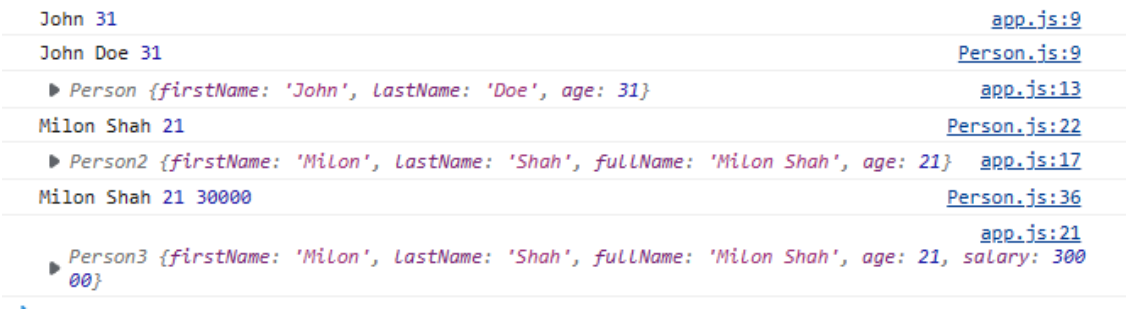
```
export default class Person{
    constructor(firstName, lastName, age){
        this.firstName = firstName;
        this.lastName = lastName;
        this.age = age;
    }

    info(){
        console.log(this.firstName, this.lastName, this.age)
    }
}

export class Person2{
    constructor(firstName, lastName, age){
        this.firstName = firstName;
        this.lastName = lastName;
        this.fullName = `${firstName} ${lastName}`;
        this.age = age;
    }

    info(){
        console.log(this.fullName, this.age)
    }
}

export class Person3{
    constructor(firstName, lastName, age, salary){
        this.firstName = firstName;
```

```
        this.lastName = lastName;
        this.fullName = `${firstName} ${lastName}`;
        this.age = age;
        this.salary = salary
    }

    info(){
        console.log(this.fullName, this.age, this.salary)
    }
}
```

Now, to use these all 3 classes, we can rewrite **app.js** in the following way:

```
// we want to import firstName from fname.js and age from age.js

import {firstName} from './utils/fname.js';
import {age} from './utils/age.js';
// import Person from './utils/Person.js';
// import { Person2 } from './utils/Person.js';
import Person, {Person2, Person3} from './utils/Person.js';

console.log(firstName, age);

const person = new Person("John", "Doe", 31);
person.info();
console.log(person);

const person2 = new Person2("Milon", "Shah", 21);
person2.info();
console.log(person2);

const person3 = new Person3("Milon", "Shah", 21, 30000);
person3.info();
console.log(person3);
```

Now, the output for **index.html** in the console will be:



Now, suppose for the **Person.js** file, we now want to export a variable as `default` in addition to the three classes we already have defined. In the **app.js** we don't necessarily need to name the variable as it is and can import with any arbitrary name that we want.

The code for **Person.js** now:

```
export class Person{
    constructor(firstName, lastName, age){
        this.firstName = firstName;
        this.lastName = lastName;
        this.age = age;
    }

    info(){
        console.log(this.firstName, this.lastName, this.age)
    }
}

export class Person2{
    constructor(firstName, lastName, age){
        this.firstName = firstName;
        this.lastName = lastName;
        this.fullName = `${firstName} ${lastName}`;
        this.age = age;
    }

    info(){
        console.log(this.fullName, this.age)
    }
}

export class Person3{
    constructor(firstName, lastName, age, salary){
        this.firstName = firstName;
        this.lastName = lastName;
        this.fullName = `${firstName} ${lastName}`;
        this.age = age;
        this.salary = salary
    }

    info(){
        console.log(this.fullName, this.age, this.salary)
    }
}

const hello = "Hello World";
export default hello;
```

The code for **app.js** is modified as follwos. Please take a note of how we have imported default variable and other functions using this line `import something, {Person, Person2, Person3} from './utils/Person.js';`

```
// we want to import firstName from fname.js and age from age.js
```

```
import {firstName} from './utils/fname.js';
import {age} from './utils/age.js';
// import Person from './utils/Person.js';
// import { Person2 } from './utils/Person.js';
import something, {Person, Person2, Person3} from './utils/Person.js';

console.log(firstName, age);

const person = new Person("John", "Doe", 31);
person.info();
console.log(person);

const person2 = new Person2("Milon", "Shah", 21);
person2.info();
console.log(person2);

const person3 = new Person3("Milon", "Shah", 21, 30000);
person3.info();
console.log(person3);

console.log(something);
```

Now, the output for **index.html** in the console will be:

```
John 31                                                                    app.js:9
John Doe 31                                                                Person.js:9
▶ Person {firstName: 'John', lastName: 'Doe', age: 31}                     app.js:13
Milon Shah 21                                                              Person.js:22
▶ Person2 {firstName: 'Milon', lastName: 'Shah', fullName: 'Milon Shah', age: 21}   app.js:17
Milon Shah 21 30000                                                        Person.js:36
                                                                           app.js:21
▶ Person3 {firstName: 'Milon', lastName: 'Shah', fullName: 'Milon Shah', age: 21, salary: 300
00}
Hello World                                                                app.js:23
```

[ ]: