Week 2 WDD330

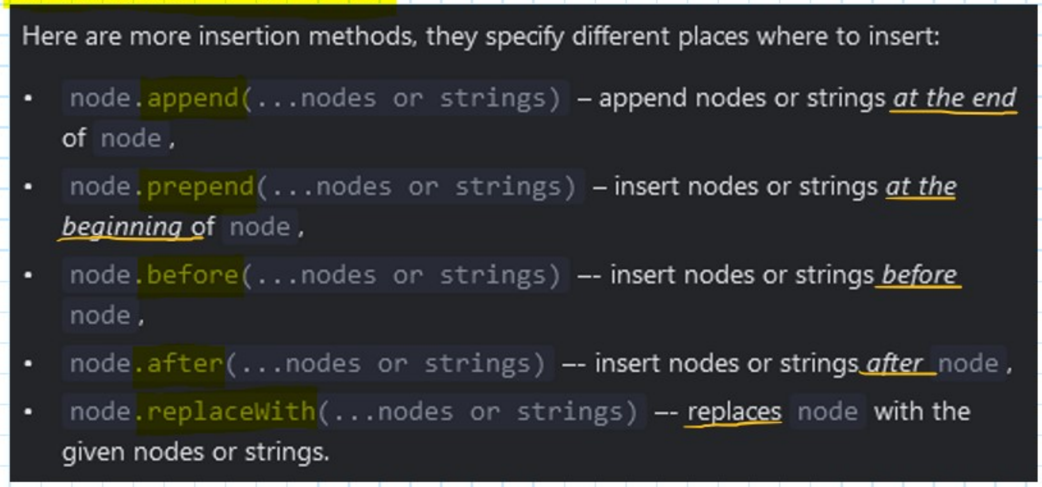
lunes, 24 de abril de 2023 13:12

[illegible] RUTH CARDONA!

MODIFYING THE DOM

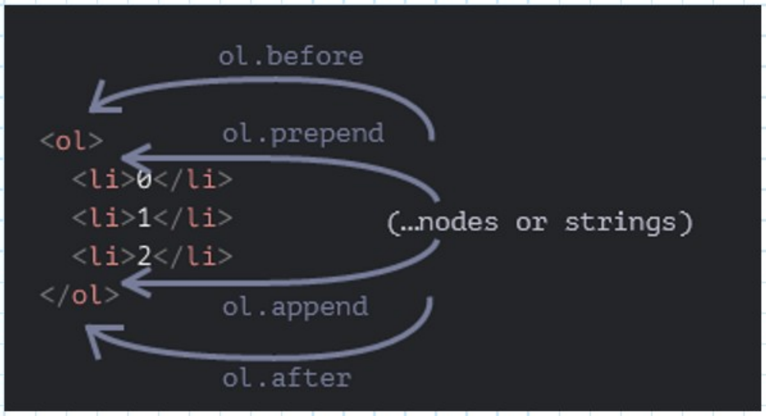
- DOCUMENT.CREATEELEMENT(TAG); ⇒ CREATE A NEW ELEMEMENT
- DOCUMENT.CREATETEXTNODE(TEXT); ⇒ CREATE A NEW TEXT NODE

INSERTION METHODS:



Here are more insertion methods, they specify different places where to insert:

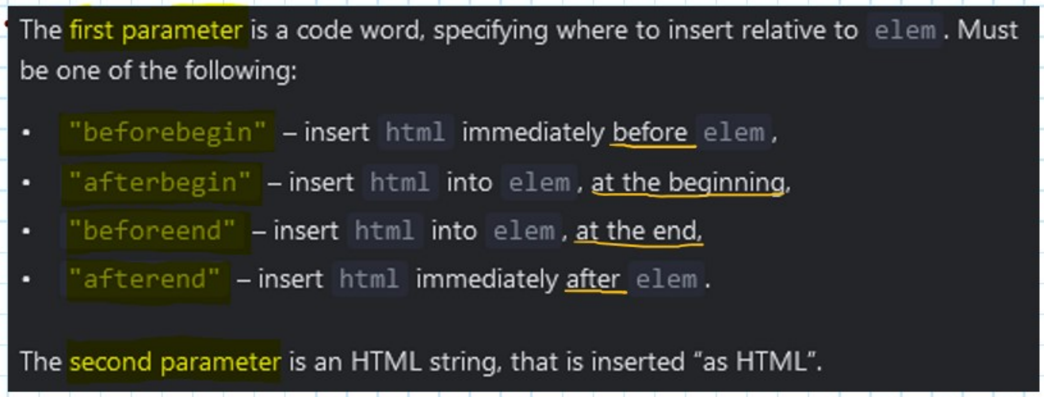
- `node.append(...nodes or strings)` – append nodes or strings *at the end* of `node`,
- `node.prepend(...nodes or strings)` – insert nodes or strings *at the beginning* of `node`,
- `node.before(...nodes or strings)` –- insert nodes or strings *before* `node`,
- `node.after(...nodes or strings)` –- insert nodes or strings *after* `node`,
- `node.replaceWith(...nodes or strings)` –- replaces `node` with the given nodes or strings.



THIS METHODS CAN ONLY BE USED TO INSERT DOM NODES OR TEXT PIECES (ELEMENT.TEXTCONTENT)

insertAdjacentHTML/Text/Element METHOD

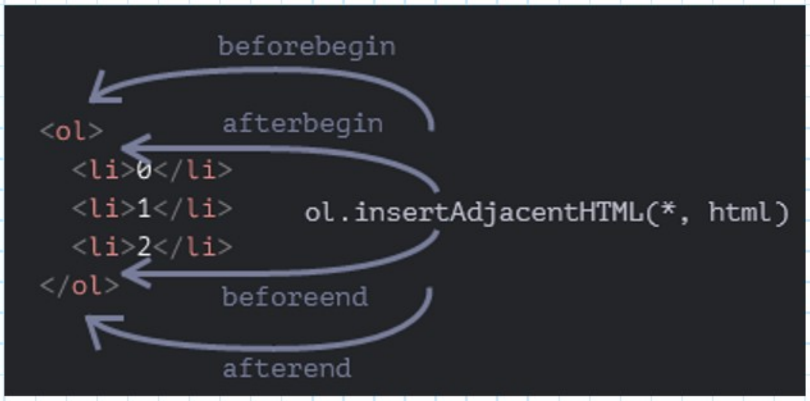
- ELEMENT.INSERTADJACENTHTML(WHERE, HTML);
- ELEMENT.INSERTADJACENTHTML(WHERE, TEXT);
- ELEMENT.INSERTADJACENTHTML(WHERE, ELEM);



The first parameter is a code word, specifying where to insert relative to `elem`. Must be one of the following:

- `"beforebegin"` – insert `html` immediately before `elem`,
- `"afterbegin"` – insert `html` into `elem`, at the beginning,
- `"beforeend"` – insert `html` into `elem`, at the end,
- `"afterend"` – insert `html` immediately after `elem`.

The second parameter is an HTML string, that is inserted "as HTML".



```
<ol>
  <li>0</li>
  <li>1</li>
  <li>2</li>
</ol>
```

THIS METHOD IS USED TO INSERT HTML IN THE SAME MANNER AS Elem.innerHTML DOES IT.

NODE REMOVAL: NODE.REMOVE()

- All insertion methods automatically remove the node from the old place
- if we want to move an element is not necessary remove it

CLONING NODES:

- ELEM.CLONENODE(TRUE) ⇒ CREATE A CLONE OF THE ELEMENT AND ITS ATTRIBUTES AND SUBELEMENTS.
- ELEM.CLONENODE(FALSE) ⇒ CREATE A CLONE WITHOUT CHILD ELEMENTS

DOCUMENT FRAGMENT: SERVES AS A WRAPPER TO PASS AROUND A LIST OF NODES

OLD METHODS :

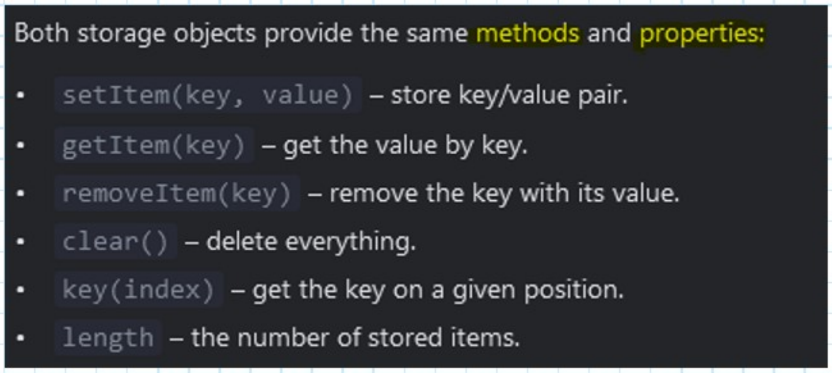
- PARENTELEM . APPENDCHILD(NODE) ⇒ APPEND A NODE AS THE LAST CHILD OF PARENTELEM.
- PARENTELEM . INSERTBEFORE(NODE, NEXTSIBLING) ⇒ INSERTS NODE BEFORE NEXT-SIBLING INTO PARENTELEM.
- PARENTELEM . REPLACECHILD(NODE, OLDCHILD) ⇒ REPLACES OLD CHILD WITH NODE AMONG CHILDREN OF PARENTELEM.
- PARENTELEM . REMOVECHILD(NODE) ⇒ REMOVES NODE FROM PARENTELEM
- DOCUMENT. WRITE(HTML) ⇒ WRITES THE HTML INTO THE PAGE (DOM). ONLY WORKS WHEN PAGE IS LOADING

localStorage, sessionStorage

ALLOW TO SAVE KEY / VALUE PAIRS IN THE BROWSER

- DATA SURVIVES A PAGE REFRESH OR FULL BROWSER RESTART

UNLIKE COOKIES :
- WEB STORAGE OBJECTS ARE NOT SEND TO SERVER WITH EACH REQUEST
- ALLOW AT LEAST 5 MB OF DATA
- EVERYTHING IS DONE WITH JS
- STORAGE IS BOUND TO THE ORIGIN (THEY CAN'T ACCESS DATA FROM EACH OTHER)


Both storage objects provide the same methods and properties:

- `setItem(key, value)` – store key/value pair.
- `getItem(key)` – get the value by key.
- `removeItem(key)` – remove the key with its value.
- `clear()` – delete everything.
- `key(index)` – get the key on a given position.
- `length` – the number of stored items.

LOCAL STORAGE:

- SHARED BETWEEN ALL TABS AND WINDOWS FROM THE SAME ORIGIN.
- DATA DOES NOT EXPIRES. SURVIVES A BROWSER RESTART AND OS REBOOT !

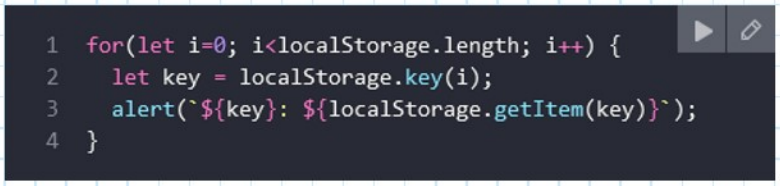
- IS SHARED BETWEEN ALL WINDOWS WITH THE SAME ORIGIN
  ↳ IF WE SET DATA IN ONE WINDOW, THE CHANGE BECOME VISIBLE IN ANOTHER ONE.

- OBJECT-LIKE ACCESS : => localStorage.TEST = 2
  (NOT RECOMMENDED)
  - THE USE OF A PLAIN OBJECT WAY OF GETTING /SETTING KEYS.
  - WILL FAIL WITH BUILD-IN METHODS OF localStorage OR EVENTS

LOOPING OVER KEYS:

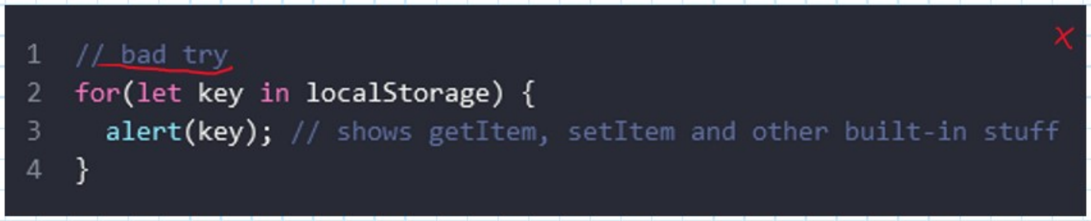
- STORAGE OBJECTS ARE NOT ITERABLE

- OVER AN ARRAY:

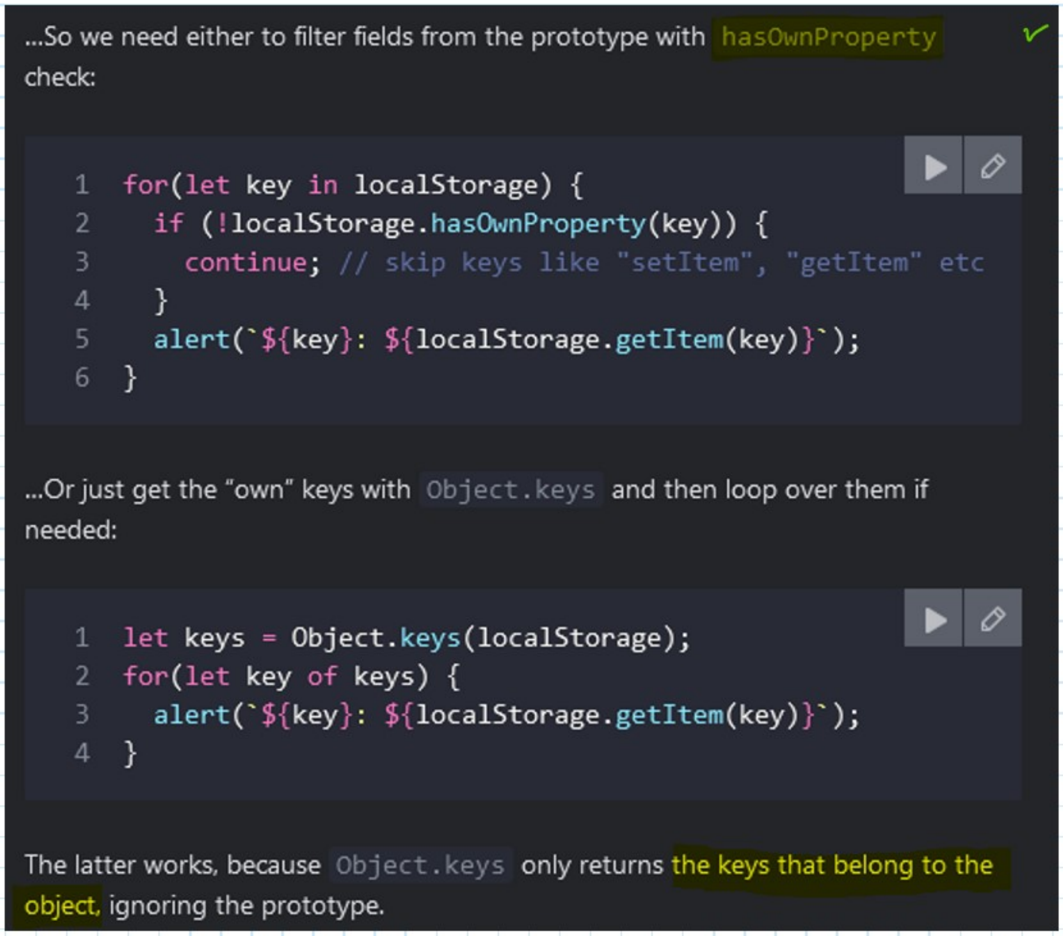


```
for(let i=0; i<localStorage.length; i++) {
  let key = localStorage.key(i);
  alert(`${key}: ${localStorage.getItem(key)}`);
}
```

- OVER KEYS:



```
// bad try
for(let key in localStorage) {
  alert(key); // shows getItem, setItem and other built-in stuff
}
```



...So we need either to filter fields from the prototype with `hasOwnProperty` check:

```
for(let key in localStorage) {
  if (!localStorage.hasOwnProperty(key)) {
    continue; // skip keys like "setItem", "getItem" etc
  }
  alert(`${key}: ${localStorage.getItem(key)}`);
}
```

...Or just get the "own" keys with `Object.keys` and then loop over them if needed:

```
let keys = Object.keys(localStorage);
for(let key of keys) {
  alert(`${key}: ${localStorage.getItem(key)}`);
}
```

The latter works, because `Object.keys` only returns the keys that belong to the object, ignoring the prototype.

STRINGS ONLY (FOR BOTH STORAGES)

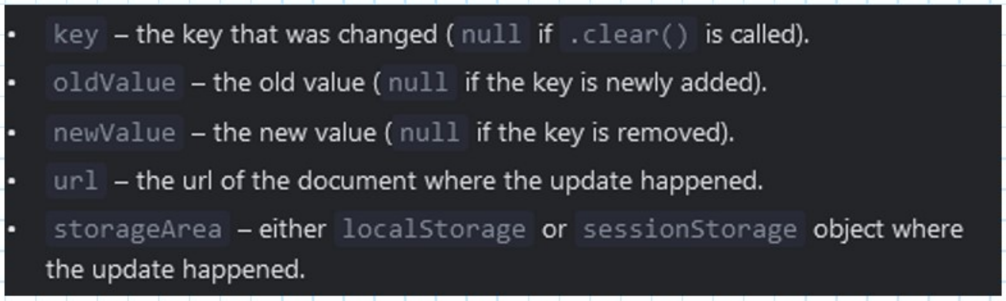
- KEY AND VALUE MUST BE STRINGS
- IF THERE IS ANOTHER TYPE OF DATA, THEY WOULD GET CONVERTED TO A STRING AUTOMATICALLY

SESSION STORAGE:

- EXIST ONLY WITHING THE CURRENT BROWSER TAB
- ANOTHER TAB WITH THE SAME PAGE WILL HAVE DIFFERENT STORAGE
- IT IS SHARED BETWEEN IFRAMES IN THE SAME TAB
- DATA SURVIVES PAGE REFRESH BUT NOT CLOSING/ OPENING THE TAB
- IT'S USED SPARINGLY.

STORAGE EVENT

- THE EVENT TRIGGERS ON ALL WINDOW OBJECTS WHERE THE localStorage IS AVAILABLE, EXCEPT THE ONE THAT CAUSED IT



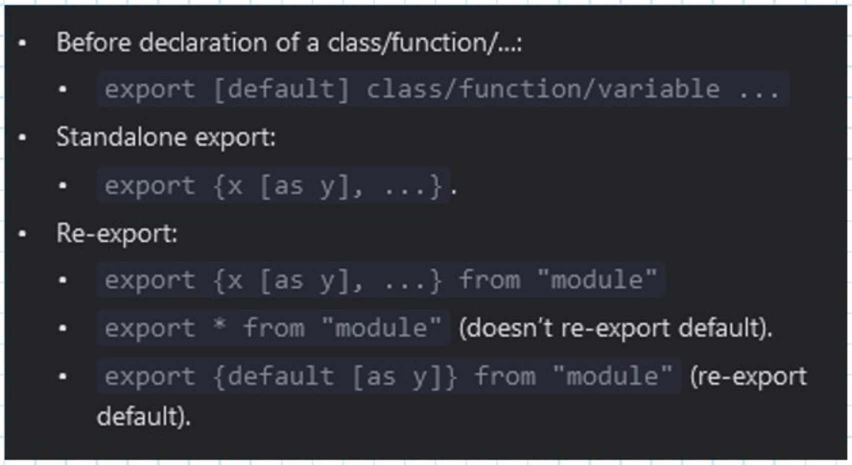
- `key` – the key that was changed (`null` if `.clear()` is called).
- `oldValue` – the old value (`null` if the key is newly added).
- `newValue` – the new value (`null` if the key is removed).
- `url` – the url of the document where the update happened.
- `storageArea` – either `localStorage` or `sessionStorage` object where the update happened.

Q: WHEN IS COMMON TO USED STORAGE EVENTS? (WHICH SITUATIONS)

Export and Import

- WE CAN PUT IMPORT/EXPORT STATEMENTS AT THE TOP OR AT THE BOTTOM OF A SCRIPT

EXPORT:



- Before declaration of a class/function/…:
  - `export [default] class/function/variable ...`
- Standalone export:
  - `export {x [as y], ...}`.
- Re-export:
  - `export {x [as y], ...} from "module"`
  - `export * from "module"` (doesn't re-export default).
  - `export {default [as y]} from "module"` (re-export default).

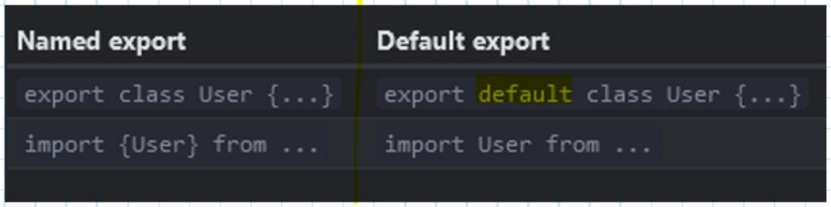
EXPORT BEFORE DECLARATION:

- EXPORT LET ARRAY = [` ', '', ...];
- EXPORT CONST CONSTANT_VARIABLE = 21;
- EXPORT CLASS USER { ... }
- EXPORT {VARIABLE AS 1, ARRAY AS AR2}

LIST OF EXPORTED DECLARATIONS:

- EXPORT { VARIABLE 1, VARIABLE2 }

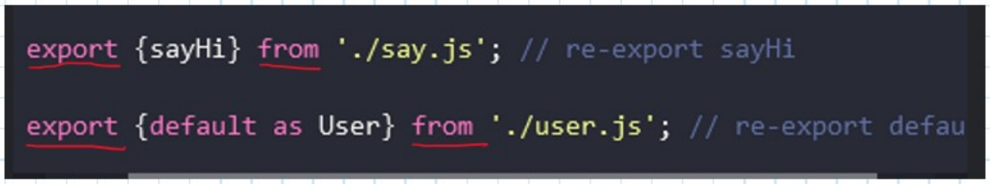
EXPORT DEFAULT:



Named export	Default export
`export class User {...}`	`export default class User {...}`
`import {User} from ...`	`import User from ...`

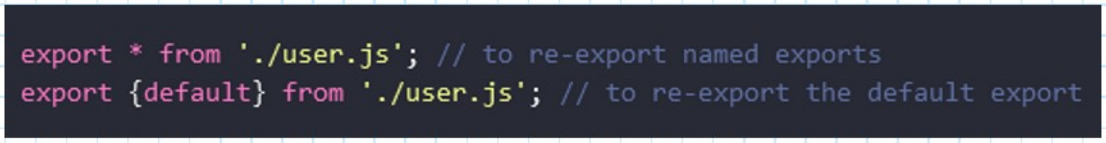
- THERE MAY BE ONLY ONE EXPORT DEFAUT PER FILE
- IT CAN BE EXPORTED WITHOUT A VARIABLE NAME:
  - EXPORT DEFAULT FUNCTION (x) {...} (NO NAME)
- FOR A DEFAULT EXPORT, WE ALWAYS CHOOSE THE NAME WHEN IMPORTING

RE-EXPORT: EXPORT ... FROM ...



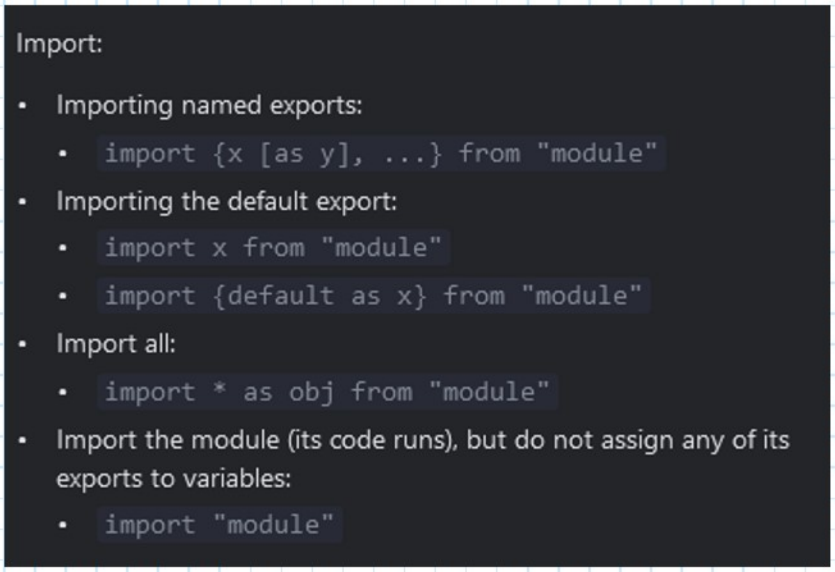
```
export {sayHi} from './say.js'; // re-export sayHi

export {default as User} from './user.js'; // re-export defau
```

- TWO STATEMENT ARE NEEDED TO RE-EXPORT DEFAULT EXPORTS



```
export * from './user.js'; // to re-export named exports
export {default} from './user.js'; // to re-export the default export
```

IMPORTS



Import:

- Importing named exports:
  - `import {x [as y], ...} from "module"`
- Importing the default export:
  - `import x from "module"`
  - `import {default as x} from "module"`
- Import all:
  - `import * as obj from "module"`
- Import the module (its code runs), but do not assign any of its exports to variables:
  - `import "module"`

- WE PUT A LIST OF WHAT TO IMPORT IN CURLY BRACES
  import { . . . } FROM "MODULE";
- EXPLICIT LIST OF IMPORTS : 1) GIVES SHORTER NAMES

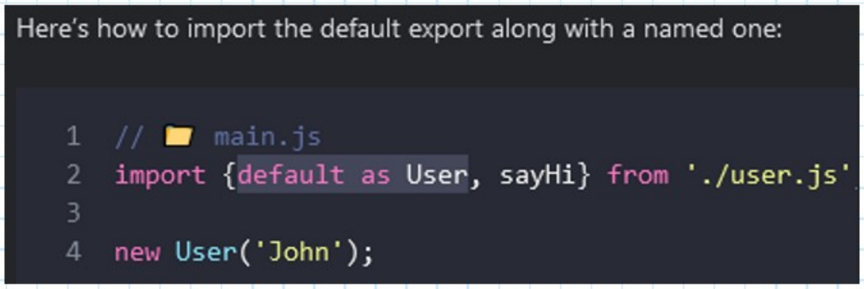
2) Gives better overview of the code structure

3) It makes code support and refactoring easier.

Import default:

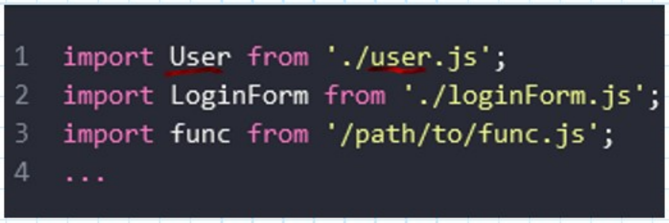
Imports are usually at the start of the file

- Imports without curly braces => import something from "module";


Here's how to import the default export along with a named one:

```
// 📁 main.js
import {default as User, sayHi} from './user.js'

new User('John');
```

- Imported variables should correspond to file names



```
import User from './user.js';
import LoginForm from './loginForm.js';
import func from '/path/to/func.js';
...
```