**pre-reads: JS for React**

To understand React better we need to understand three things really well. Many times when we want to do React the syntax comes in the way. React is heavily influenced by the functional programming paradigm. With the hooks, you don't need class components anymore.

Again, pre-reads are good to read before the class. But don't ditch the class if you can't. **Live classes are the most important** part of the camp.

**js is async**

Understanding this is the most important part and this is a classic video that explains this. <https://youtu.be/8aGhZQkoFbQ>

You can read about it. If you don't understand all the code, no issues, have a look at least so that you know what you don't know: <https://eloquentjavascript.net/11_async.html>

**functional programming**

You don't need to be an expert in FP. Just understand that JS supports the functions of a first-class citizen. Read this and be ready. I'll explain everything about FP in the session.

<https://eloquentjavascript.net/05_higher_order.html>

**es6+**

This is mostly syntax. Fewer concepts. So try to read it up, and if you can do some exercises on your console that would be great.

1. [Arrow Functions](https://javascript.info/arrow-functions-basics)
2. [Destructuring](https://javascript.info/destructuring-assignment)
3. [let/cost variables](https://javascript.info/variables)
4. [Rest Parameters](https://javascript.info/rest-parameters-spread). (Not important, but you might run into this)
5. [Template Literals](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template_literals)
6. [Modules](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Modules). No need to get into depth but understand the basic syntax.
7. [Shorthand Object](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Object_initializer#property_definitions)
8. [Computed Property Names](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Object_initializer#computed_property_names)
9. [Temporal Dead Zone](https://stackoverflow.com/questions/33198849/what-is-the-temporal-dead-zone)
10. [Import/Export](https://javascript.info/import-export)

<aside> 📎 **overwhelming?**

Well! I said levelOne is not going to be easy. However, if you are truly pressed against time then follow a simple strategy of attending all the live classes and coding assignments. Do these extra things in your own time.

</aside>

<aside> ⛔ **privacy warning**

Before we end, I must say that any resource I share with you, you should not share it with anyone else. If we find anyone violating this policy we will expel that person from the camp.

**Do not break my trust.**

</aside>

Notes:

## [Rest parameters ...](https://javascript.info/rest-parameters-spread" \l "rest-parameters)

A function can be called with any number of arguments, no matter how it is defined.

function showName(firstName, lastName, ...titles) {

alert( firstName + ' ' + lastName ); // Julius Caesar

// the rest go into titles array

// i.e. titles = ["Consul", "Imperator"]

alert( titles[0] ); // Consul

alert( titles[1] ); // Imperator

alert( titles.length ); // 2

}

showName("Julius", "Caesar", "Consul", "Imperator");

##

let arr = [3, 5, 1];

alert( Math.max(...arr) ); // 5 (spread turns array into a list of arguments)

#######################################################################3

Template literals are literals delimited with backticks (`), allowing embedded expressions called substitutions.

* *Untagged* template literals result in strings, which makes them useful for string interpolation (and multiline strings, since unescaped newlines are allowed).
* *Tagged* template literals call a function (the *tag function*) with an array of any text segments from the literal followed by arguments with the values of any substitutions,

### [**Multi-line strings**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template_literals#multi-line_strings)

Any newline characters inserted in the source are part of the template literal.

Using normal strings, you would have to use the following syntax in order to get multi-line strings:

console.log('string text line 1\n' +

'string text line 2');

// "string text line 1

// string text line 2"

Copy to Clipboard

Using template literals, you can do the same like this:

console.log(`string text line 1

string text line 2`);

// "string text line 1

// string text line 2"

### [**Expression interpolation**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template_literals#expression_interpolation)

In order to embed expressions within normal strings, you would use the following syntax:

let a = 5;

let b = 10;

console.log('Fifteen is ' + (a + b) + ' and\nnot ' + (2 \* a + b) + '.');

// "Fifteen is 15 and

// not 20."

Now, with template literals, you are able to make use of the syntactic sugar, making substitutions like this more readable:

let a = 5;

let b = 10;

console.log(`Fifteen is ${a + b} and

not ${2 \* a + b}.`);

// "Fifteen is 15 and

// not 20."

### [**Tagged templates**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template_literals#tagged_templates)

A more advanced form of template literals are tagged templates.

Tags allow you to parse template literals with a function. The first argument of a tag function contains an array of string values. The remaining arguments are related to the expressions.

let person = 'Mike';

let age = 28;

function myTag(strings, personExp, ageExp) {

let str0 = strings[0]; // "That "

let str1 = strings[1]; // " is a "

let str2 = strings[2]; // "."

let ageStr;

if (ageExp > 99){

ageStr = 'centenarian';

} else {

ageStr = 'youngster';

}

// We can even return a string built using a template literal

return `${str0}${personExp}${str1}${ageStr}${str2}`;

}

let output = myTag`That ${ person } is a ${ age }.`;

console.log(output);

// That Mike is a youngster.

#### **Duplicate property names**

When using the same name for your properties, the second property will overwrite the first.

let a = {x: 1, x: 2}

console.log(a) // {x: 2}

#########

// Computed property names (ES2015)

let i = 0

let a = {

['foo' + ++i]: i,

['foo' + ++i]: i,

['foo' + ++i]: i

}

console.log(a.foo1) // 1

console.log(a.foo2) // 2

console.log(a.foo3) // 3

**###########3**

**The time between let/const is hoisted and value is assigned to is called Temporal dead zone.**

Import and Export

Here we first declare, and then export:

// 📁 say.js

function sayHi(user) {

alert(`Hello, ${user}!`);

}

Let a=10

function sayBye(user) {

alert(`Bye, ${user}!`);

}

export {sayHi, sayBye,a}; // a list of exported variables

**Import**

## [Import \*](https://javascript.info/import-export" \l "import)

Usually, we put a list of what to import in curly braces import {...}, like this:

// 📁 main.js

import {sayHi, sayBye, a} from './say.js';

sayHi('John'); // Hello, John!

sayBye('John'); // Bye, John!

But if there’s a lot to import, we can import everything as an object using import \* as <obj>, for instance:

// 📁 main.js

import \* as say from './say.js';

say.sayHi('John');

say.sayBye('John');

## [Import “as”](https://javascript.info/import-export" \l "import-as)

We can also use as to import under different names.

For instance, let’s import sayHi into the local variable hi for brevity, and import sayBye as bye:

// 📁 main.js

import {sayHi as hi, sayBye as bye} from './say.js';

hi('John'); // Hello, John!

bye('John'); // Bye, John!

## [Export “as”](https://javascript.info/import-export" \l "export-as)

The similar syntax exists for export.

Let’s export functions as hi and bye:

// 📁 say.js

...

export {sayHi as hi, sayBye as bye};

 import needs curly braces for named exports and doesn’t need them for the default one.

| **Named export** | **Default export** |
| --- | --- |
| export class User {...} | export default class User {...} |
| import {User} from ... | import User from |