Getting familiar with Svelte

Yet another web framework.
Yet another component based web framework
But it has many specialities which are not found in other frameworks like its components consist of html, css and js, the components are compiled into js modules, etc.

Getting started

A typical component would consist of markup, css and js. css styling applied on a tag is one component is not propagated automatically to the other components consisting of same tag.

Data handling.

We can declare variable in script tag and use it in the html tags by using ${ }. { } allow js to be injected inside the tags as normal text, data or attribute.
If any string consists of html then while using this string to print it, we should attach @html before the variable.
In order to keep the DOM in sync with the application state, Svelte provides powerful system of reactivity.

Ex.

Lets say we have a count variable =0
And we have a button that displays the count variable.
Then we attach click handler using on:click = {increment} increment fn just increments the count value by 1.
And changes are visible in UI.

If any state value depends on any other state, then use $: dependant variable = independant variable.
This statement is called reactive declaration.
This declaration, though, is not limited to just reactive values. We can run arbitrary statements using $: statement. Example → $: console.log('the count is ' + count);
We can also group statements using $:

```
$:
  s1      Let say s1 renders on change is value
  s2      then s2 re-renders.
```

Statements like if can also be put after $: to implement cond: beyond a value.

This reactivity model clearly specifies that it depends on assignment. In that case, array's operations like push, splice, etc. won't produce any effect as they are not directly assignment ops. So, we have use any of the following methods to help ourselves out:

1. After push, use arr = arr
2. use index. arr[index] = value
3. Spread the array using '...' operator and then specify the elements to be inserted.

Props handling

To pass values from one component to other, use export keyword on the value.
Basically, we pass value in the parent component and receive them in child using export keyword.
The name of props passed is important and export let must be used with the same names to implement props.
Let's say we expect some prop value in a child comp. but no sure of receiving prop. Then we can set default values by simply assigning export let ans = "hello"
If value to be passed is already stored in an array or object, we can use spread operator to pass that obj or arr directly instead of specifying each value of the obj or arr. Conversely, we can refer to props obj using $$ props instead of using export let with every value.

Logic Handling

We have if blocks here to allow conditional rendering

{#if cond}
    markup
{/if}

Similarly we have if-else block

{#if condition}
    markup
{:else}
    markup
{/if}

{:else} can be repeated to mimic elif

- If we want to repeat a component certain no. of times then we can use #each instead of manually writing those components. There has to be an array or array like value for this repetition has to be done.

Ex

{#each colors as color, i} → current index
    <button
        attr = color
    >
    </button>
{/each}

The interesting case #each :

If we create an array of components based on the contents of array
- then we must use key with the array values if the array is dynamic. Otherwise any changes in the original array won't necessarily behave well as it will use the previously created DOM and won't create new DOMs with updations

await promises

Instead of using script, we can handle promise inside markup itself.

```
{#await promise}
   :
   :
{:then ...}
   :
   :
{:catch error}
   :
   :
{/await}
```

Remember that the promise does not need to attached on the source of change. Rather it is for resolving the Promise and returning the result. The source of change should call the fn that returns promise.