Advanced bindings

- Content Editable binding :— It allows element to be editable by the user i.e., user can type directly into the element. We can bind the content of any contenteditable element to any variable using bind:innerHTML or bind:textContent innerHTML preserves tag name while textContent is just pure text.

Each block bindings

We can bind properties inside an each block. Basically list items can be individually bind directly to DOM elements.

In the given example, todos arrays had todo objects. All of these todo objects are rendered as input elements with a checkbox and text input to track done & text values of each of the todo.

Handling media elements

We can bind properties of <audio> and <video> elements. This feature can help us understand how to build custom audio players.

The complete set of bindings of <audio> and <video> :-

- duration (readonly) → total duration, in seconds
- buffered (readonly) → array of {start, end} objects
- seekable (readonly) → ditto
- played (readonly) → ditto
- seeking (readonly) → boolean
- readyState (readonly) → no b/w and inc. 0 and 4.

two way bindings :-

- currentTime → current pos' of playhead, in seconds
- playbackRate → speed up or slow down (1 is normal)

↳ paused → this one should be self explanatory

↳ volume → a value b/w 0 & 1.

↳ muted → boolean value where true is muted

Video additionally have readonly videoWidth and videoHeight bindings

Dimensions

Every block level element has clientWidth, clientHeight offsetWidth and offsetHeight bindings.

For particular div's, we can take out their w and h using these bindings.

In the given example, div was dynamic to the contents. Let's say we need to determine the change in dimensions as it happen in real time. We can bind the clientWidth & height to some local variables and use them as and when required.

```
<div bind:clientWidth={w} bind:clientHeight={h}>
  :
</div>
```

* elements like display: inline can't be measured with this approach; nor can elements that can't contain other elements like canvas. In these cases, we need to measure a wrapper element instead.

This

It allows us to bind a DOM element to a variable directly which gives us direct access to the element when it is loaded in the DOM. It elements the need of methods like document.querySelector() which can be error-prone if we have multiple instances of the same element type.

↳ Binding a DOM element: → when we use bind: this={variable} on an element, Svelte will assign

that DOM element to the variable once the component has been mounted.

We can also bind component props.
(Not very recommended)

Classes and Style

We can add classes like any other attribute to a component. Let us we want to specify class based on conditions.
In the given example, we checked flag 'flipped'. If it was true then we added flipped class to card component to conditionally style the card. `<button class="card {flipped ? 'flipped' : ''}"`
Adding or removing a class based on some cond^n is such a common pattern in UI development that Svelte includes a special directive to simplify it.

`<button class="card" class:flipped={flipped}`

⇒ It means add flipped class if flipped value is truthy.
often the name of class is same as name of value, it depends on then just use class:flipped. This shorthand would be enough.

Just like class was added, styles can be added to it (inline). When we have a lots of styles, instead of directly specifying in `style="..."`, we divide the styles in several part and specify each using `style:...` (without quotes o/c). Though the style parameters like transform, d, etc., would still accept value in quotes.

Component styles

often we need to influence the styles inside a child component. One way of achieving it is controlling css of child using global css modified.
for ex → In the given ex., main div with class: boxes can be utilized to set css for its children.

We can access particular child nodes using individual child's name with attribute :nth-child(index)
But the above method is feasible for large applications as parent component becomes a lot more dependant on child component.
Components, though are rendered in a parent comp, should be able to decide their property themselves.
Rather these specific properties should be passed as prop. The child comp's style should prepare itself to accept a prop and make its style. In the given example, we made background-color: var(--color, #ddd)
Then the parent element (here the div. with class = boxes) can provide --color = color-name to all the child comps that are going to render.

Component Composition

Main comp implementing a child component can provide direct elements to be inserted inside the child comp. But child comp would have to be prepared for such use and decide where to keep the upcoming element.
This placeholder is handled by <slot></slot>. Put it in the child comp wherever you want the incomig element to keep. Then simply pass elements from main comp to child comps. The elements will be rendered b/w the slot.

In order to have more control over the placement of incoming element from parent, we can use named slots.
ex. In child comp.

```
<div class='card'>
  <header>
    <slot name="telephone"></slot>
```

<Slot name="company"></Slot>
</header>

<Slot></Slot>

<footer>
<Slot name="address"></Slot>
</footer>

Then in parent comp, the element that required to be placed in specific slot is provided attribute slot="slot name".
In case styles/css is to be applied to a part of element that is inserted in any slot. Though it may feel that the child comp should handle it but the elements are added in the main comp itself. So, its style should be written there only.

We can handle fallbacks for any slots that are left empty by putting content inside <slot> element.
slot props enhance slot by allowing child comp. to pass data back to the slotted content. The data is passed from child to parent through the slot itself.
In the given eg, for each of color, we allotted slot for each row. For each row, we are providing the color info to the parent comp. We can access this data using the let: directive inside the slot. We need to specify a variable name in which we want to keep incoming data.
let:item={row} is the correct syntax for receiving and setting the data.

In some cases, we may want to control part of comp. based on whether slotted content is passed in or not. Basically, it can be case that child comp. render a slot but the parent comp does not provide content for it. In that case, some unwanted elements may appear

on screen. So, we can conditionally specify slots according to availability of content.
We use $$slots.slotname for that.
We simple check if $$slot.slotname available or not. If it's there then only specify slots.

Context API

Context API provides mechanism for components to talk to each other without passing around data & functions as props or dispatching a lot of events.
In svelte, the component defining a particular property or fn imports setContext from svelte. Then use setContext('key', {value}); Now, whoever will import getContext from svelte in the current scenario will have access to the value via key. We use getContext(key) to take out the value stored.
Like lifecycle fns, setContext and getContext must be called during component initialization.
context obj can include anything like stores.