**Svelte Notes –Official Docs**

**Introduction:-**

* Shorthand attributes

It's not uncommon to have an attribute where the name and value are the same, like src={src}. Svelte gives us a convenient shorthand for these cases:

e.g: <img {src} alt="A man dances.">

* But sometimes you need to render HTML directly into a component. For example, the words you're reading right now exist in a markdown file that gets included on this page as a blob of HTML.

In Svelte, you do this with the special {@html ...} tag:

e.g: <script>

let string = `this string contains some <strong>HTML!!!</strong>`;

</script>

<p>{@html string}</p>

**Reactivity:-**

* Svelte's reactivity not only keeps the DOM in sync with your application's variables as shown in the previous section, it can also keep variables in sync with each other using reactive declarations. They look like this:

E.g: let count = 0;

$: doubled = count \* 2;

* Svelte's reactivity is triggered by assignments. Methods that mutate arrays or objects will not trigger updates by themselves.

The same rule applies to array methods such as pop, shift, and splice and to objects methods such as Map.set, Set.add, etc.

A simple rule of thumb: the updated variable must directly appear on the left hand side of the assignment. E.g: numbers = numbers OR numbers = […numbers, numbers.length + 1] OR numbers[numbers.length] = numbers.length + 1;

**Props:-**

* In any real application, you'll need to pass data from one component down to its children. To do that, we need to declare properties, generally shortened to 'props'. In Svelte, we do that with the *export* keyword.
* We can easily specify **default values for props** in. e.g;

<script>

export let answer = 'a mystery';

</script>

* if you need to reference all the props that were passed into a component, including ones that weren't declared with export, you can do so by accessing *$$props* directly. It's not generally recommended, as it's difficult for Svelte to optimise, but it's useful in rare cases.

**Logic:-**

* if-else logic:-

{#if user.loggedIn} do something…

{:else if} do something

else…{:else} do something..

{/if}

* loop logic

{#each cats as cat, index} do something… {/each}

* By default, when you modify the value of an each block, it will add and remove items at the end of the block, and update any values that have changed. That might not be what you want.
* You can use any object as the key, as Svelte uses a Map internally — in other words you could do (thing) instead of (thing.id). Using a string or number is generally safer, however, since it means identity persists without referential equality, for example when updating with fresh data from an API server.
* Most web applications have to deal with asynchronous data at some point. Svelte makes it easy to await the value of [promises](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Using_promises) directly in your markup:

E.g: {#await promise}

<p>...waiting</p>

{:then number}

<p>The number is {number}</p>

{:catch error}

<p style="color: red">{error.message}</p>

{/await}

* Only the most recent promise is considered, meaning you don't need to worry about race conditions.
* If you know that your promise can't reject, you can omit the catch block. You can also omit the first block if you don't want to show anything until the promise resolves: e.g;

{#await promise then value}

<p>the value is {value}</p>

{/await}

**DOM Events:-**

* you can listen to any event on an element with the on: directive:
* DOM event handlers can have modifiers that alter their behavior. For example, a handler with a once modifier will only run a single time:
* **The full list of modifiers**:
* preventDefault — calls event.preventDefault() before running the handler. Useful for client-side form handling, for example.
* stopPropagation — calls event.stopPropagation(), preventing the event reaching the next element
* passive — improves scrolling performance on touch/wheel events (Svelte will add it automatically where it's safe to do so)
* nonpassive — explicitly set passive: false
* capture — fires the handler during the ***capture*** phase instead of the ***bubbling*** phase ([MDN docs](https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Building_blocks/Events#Event_bubbling_and_capture))
* once — remove the handler after the first time it runs
* self — only trigger handler if event.target is the element itself
* trusted — only trigger handler if event.isTrusted is true. I.e. if the event is triggered by a user action.
* You can chain modifiers together, e.g. on:click|once|capture={...}.
* **Component Events**:
* Components can also dispatch events. To do so, they must create an event dispatcher e.g; <Inner on:message={handleMessage}/>
* createEventDispatcher must be called when the component is first instantiated — you can't do it later inside e.g. a setTimeout callback. This links dispatch to the component instance.
* **Event Forwarding**:
* Unlike DOM events, component events don't bubble. If you want to listen to an event on some deeply nested component, the intermediate components must forward the event.

E.g; <script>import Inner from './Inner.svelte';</script>

<Inner on:message />

* Event forwarding works for DOM events too

E.g; We want to get notified of clicks on our <CustomButton> — to do that, we just need to forward click events on the <button> element in CustomButton.svelte

<button on:click> Click me </button>

**Bindings:-**

* As a general rule, data flow in Svelte is top down — a parent component can set props on a child component, and a component can set attributes on an element, but not the other way around.

we can use the bind  directive to make the element as Controlled element

e.g; <input bind:value={name}>

* In the DOM, everything is a string. That's unhelpful when you're dealing with numeric inputs — type="number" and type="range" — as it means you have to remember to coerce input.value before using it.

With bind:value, Svelte takes care of it for you:

e.g: <input type=number bind:value={a} min=0 max=10>

<input type=range bind:value={a} min=0 max=10>

* Checkboxes are used for toggling between states. Instead of binding to input.value, we bind to input.checked

E.g: <input type=checkbox bind:checked={yes}>

* If you have multiple inputs relating to the same value, you can use bind:group along with the value attribute. Radio inputs in the same group are mutually exclusive; checkbox inputs in the same group form an array of selected values

e.g: <input type=radio bind:group={scoops} name="scoops" value={1}>

* The <textarea> element behaves similarly to a text input in Svelte — use bind:value

e.g: <textarea bind:value={value}></textarea> OR <textarea bind:value></textarea>

* We can also use bind:value with <select> elements

e.g: <select bind:value={selected} on:change="{() => answer = ''"}>

* <option value={question}>. Here question is an object not string. Svelte doesn't mind
* A select can have a multiple attribute, in which case it will populate an array rather than selecting a single value

e.g: <select multiple bind:value={flavours}>

{#each menu as flavour}

<option value={flavour}>

{flavour}

</option>

{/each}

</select>

* Elements with a contenteditable="true" attribute support textContent and innerHTML bindings:

e.g: <div contenteditable="true" bind:innerHTML={html}></div>

* You can even bind to properties inside an each block.

**Note** that interacting with these <input> elements will mutate the array. If you prefer to work with immutable data, you should avoid these bindings and use event handlers instead

* The <audio> and <video> elements have several properties that you can bind to.

Ordinarily on the web, you would track currentTime by listening for timeupdate events. But these events fire too infrequently, resulting in choppy UI. Svelte does better — it checks currentTime using requestAnimationFrame.

The complete set of bindings for <audio> and <video> is as follows — six readonly bindings...

A) duration (readonly) — the total duration of the video, in seconds

B) buffered (readonly) — an array of {start, end} objects

C) seekable (readonly) — ditto

D) played (readonly) — ditto

E) seeking (readonly) — boolean

F) ended (readonly) — boolean

...and five two-way bindings:

A) currentTime — the current point in the video, in seconds

B) playbackRate — how fast to play the video, where 1 is 'normal'

C) paused — this one should be self-explanatory

1. volume — a value between 0 and 1

E) muted — a boolean value where true is muted

Videos additionally have readonly videoWidth and videoHeight bindings.

* Every block-level element has clientWidth, clientHeight, offsetWidth and offsetHeight bindings:

e.g: <div bind:clientWidth={w} bind:clientHeight={h}>

<span style="font-size: {size}px">{text}</span> </div>

These bindings are readonly — changing the values of w and h won't have any effect.

* The readonly this binding applies to every element (and component) and allows you to obtain a reference to rendered elements

e.g:<canvas bind:this={canvas} width={32} height={32} ></canvas>

* Just as you can bind to properties of DOM elements, you can bind to component props

e.g: <Keypad bind:value={pin} on:submit={handleSubmit}/>

* Just as you can bind to DOM elements, you can bind to component instances themselves. For example, we can bind the instance of <InputField> to a variable named field in the same way we did when binding DOM Elements

e.g: <InputField bind:this={field} />

**Lifecycle:-**

* Every component has a lifecycle that starts when it is created, and ends when it is destroyed.
* The one you'll use most frequently is onMount, which runs after the component is first rendered to the DOM

e.g: onMount(async () => {

const res = await fetch(`/tutorial/api/album`);

photos = await res.json(); });

* It's recommended to put the fetch in onMount rather than at the top level of the <script> because of server-side rendering (SSR). With the exception of onDestroy, lifecycle functions don't run during SSR
* Lifecycle functions must be called while the component is initialising so that the callback is bound to the component instance — not (say) in a setTimeout
* If the onMount callback returns a function, that function will be called when the component is destroyed.
* To run code when your component is destroyed, use onDestroy
* For example, we can add a setInterval function when our component initialises, and clean it up when it's no longer relevant. Doing so prevents memory leaks.

e.g: onDestroy(() => clearInterval(interval));

* The beforeUpdate function schedules work to happen immediately before the DOM is updated. afterUpdate is its counterpart, used for running code once the DOM is in sync with your data.
* Note that beforeUpdate will first run before the component has mounted, so we need to check for the existence of div before reading its properties.
* The tick function is unlike other lifecycle functions in that you can call it any time, not just when the component first initializes. It returns a promise that resolves as soon as any pending state changes have been applied to the DOM (or immediately, if there are no pending state changes).
* When you update component state in Svelte, it doesn't update the DOM immediately. Instead, it waits until the next micro task to see if there are any other changes that need to be applied, including in other components. Doing so avoids unnecessary work and allows the browser to batch things more effectively.

e.g: import { tick } from 'svelte'; await tick();

**Stores:-**

* Not all application state belongs inside your application's component hierarchy. Sometimes, you'll have values that need to be accessed by multiple unrelated components, or by a regular JavaScript module.
* In Svelte, we do this with stores. A store is simply an object with a subscribe method that allows interested parties to be notified whenever the store value changes.
* It's a **writable store**, which means it has set and update methods in addition to subscribe method.
* The store is subscribed to, but never unsubscribed. If the component was instantiated and destroyed many times, this would result in a memory leak
* unsubscribe needs to be called, for example through the onDestroy [lifecycle hook](https://svelte.dev/tutorial/ondestroy):
* It starts to get a bit boilerplatey though, especially if your component subscribes to multiple stores. Instead, Svelte has a trick up its sleeve — you can reference a store value by prefixing the store name with $:
* **Auto-subscription** only works with store variables that are declared (or imported) at the top-level scope of a component.
* Any name beginning with $ is assumed to refer to a store value. It's effectively a reserved character — Svelte will prevent you from declaring your own variables with a $ prefix.
* Not all stores should be writable by whoever has a reference to them. For example, you might have a store representing the mouse position or the user's geolocation, and it doesn't make sense to be able to set those values from 'outside'. For those cases, we have **readable stores**.
* Click over to the stores.js tab. The first argument to readable is an initial value, which can be null or undefined if you don't have one yet. The second argument is a start function that takes a set callback and returns a stop function. The start function is called when the store gets its first subscriber; stop is called when the last subscriber unsubscribes.
* e.g: export const time = readable(new Date(), function start(set) {

const interval = setInterval(() => {

set(new Date());

}, 1000);

return function stop() {

clearInterval(interval);

};

});

* You can create a store whose value is based on the value of one or more other stores with derived
* It's possible to **derive a store** from multiple inputs, and to explicitly set a value instead of returning it (which is useful for deriving values asynchronously). Consult the [API reference](https://svelte.dev/docs#run-time-svelte-store-derived) for more information.
* As long as an object correctly implements the subscribe method, it's a store. Beyond that, anything goes. It's very easy, therefore, to create custom stores with domain-specific logic.
* If a store is writable — i.e. it has a set method — you can bind to its value, just as you can bind to local component state
* In this example we have a writable store name and a derived store greeting. Update the <input> element: <input bind:value={$name}>
* We can also assign directly to store values inside a component. Add a <button> element:

<button on:click="{() => $name += '!'}">

Add exclamation mark!

</button>

* The $name += '!' assignment is equivalent to name.set($name + '!').

**Motion:-**

* Let's start by changing the progress store to a ***tweened*** value:

e.g: import { tweened } from 'svelte/motion';

import { cubicOut } from 'svelte/easing';

const progress = tweened(0, {

duration: 400,

easing: cubicOut

});

* The full set of options available to tweened:
* delay — milliseconds before the tween starts
* duration — either the duration of the tween in milliseconds, or a (from, to) => milliseconds function allowing you to (e.g.) specify longer tweens for larger changes in value
* easing — a p => t function
* interpolate — a custom (from, to) => t => value function for interpolating between arbitrary values. By default, Svelte will interpolate between numbers, dates, and identically-shaped arrays and objects (as long as they only contain numbers and dates or other valid arrays and objects). If you want to interpolate (for example) colour strings or transformation matrices, supply a custom interpolator
* You can also pass these options to progress.set and progress.update as a second argument, in which case they will override the defaults. The set and update methods both return a promise that resolves when the tween completes.
* The spring function is an alternative to tweened that often works better for values that are frequently changing.

e.g: import { spring } from 'svelte/motion';

let coords = spring({ x: 50, y: 50 });

let size = spring(10);

* Both springs have default stiffness and damping values, which control the spring's, well... springiness. We can specify our own initial values:

e.g: let coords = spring({ x: 50, y: 50 }, {

stiffness: 0.1,

damping: 0.25

* });

**Transitions:-**

* We can make more appealing user interfaces by gracefully transitioning elements into and out of the DOM. Svelte makes this very easy with the ***transition*** directive.

e.g: <script>

import { fade } from 'svelte/transition';

let visible = true;

</script>

<p transition:fade>Fades in and out</p>

* Transition functions can accept parameters. Replace the fade transition with ***fly***...

e.g: <p transition:fly="{{ y: 200, duration: 2000 }}">

Flies in and out

</p>

* Instead of the transition directive, an element can have an ***in*** or an ***out*** directive, or both together. Import fade alongside fly

e.g: <p in:fly="{{ y: 200, duration: 2000 }}" out:fade>

Flies in, fades out

</p>

* The svelte/transition module has a handful of built-in transitions, but it's very easy to create your own.

e.g: function fade(node, {

delay = 0,

duration = 400

}) {

const o = +getComputedStyle(node).opacity;

return {

delay,

duration,

css: t => `opacity: ${t \* o}`

};

}

* The function takes two arguments — the node to which the transition is applied, and any parameters that were passed in — and returns a transition object which can have the following properties:
* delay — milliseconds before the transition begins
* duration — length of the transition in milliseconds
* easing — a p => t easing function (see the chapter on [tweening](https://svelte.dev/tutorial/tweened))
* css — a (t, u) => css function, where u === 1 - t
* tick — a (t, u) => {...} function that has some effect on the node
* While you should generally use CSS for transitions as much as possible, there are some effects that can't be achieved without JavaScript, such as a typewriter effect:

e.g: function typewriter(node, { speed = 1 }) {

const valid = (

node.childNodes.length === 1 &&

node.childNodes[0].nodeType === Node.TEXT\_NODE

);

if (!valid) {

throw new Error(`This transition only works on elements with a single text node child`);

}

const text = node.textContent;

const duration = text.length / (speed \* 0.01);

return {

duration,

tick: t => {

const i = Math.trunc(text.length \* t);

node.textContent = text.slice(0, i);

}

};

}

<p transition:typewriter>

The quick brown fox jumps over the lazy dog

</p>

* It can be useful to know when transitions are beginning and ending. Svelte dispatches events that you can listen to like any other DOM event:

e.g: <p

transition:fly="{{ y: 200, duration: 2000 }}"

on:introstart="{() => status = 'intro started'}"

on:outrostart="{() => status = 'outro started'}"

on:introend="{() => status = 'intro ended'}"

on:outroend="{() => status = 'outro ended'}"

>

Flies in and out

</p>

* Ordinarily, transitions will play on elements when any container block is added or destroyed. In the example here, toggling the visibility of the entire list also applies transitions to individual list elements.

Instead, we'd like transitions to play only when individual items are added and removed — in other words,

We can achieve this with a local transition, which only plays when the block with the transition itself is added or removed:

e.g: <div transition:slide|local>

{item}

</div>

* A particularly powerful feature of Svelte's transition engine is the ability to *defer* transitions, so that they can be coordinated between multiple elements.
* Key blocks destroy and recreate their contents when the value of an expression changes.

e.g.: {#key value}

<div transition:fade>{value}</div>

{/key}

**Animations:-**

* the animate directive
* First, import the flip function — flip stands for ['First, Last, Invert, Play'](https://aerotwist.com/blog/flip-your-animations/) — from svelte/animate: as import { flip } from 'svelte/animate';

e.g.: <label

in:receive="{{key: todo.id}}"

out:send="{{key: todo.id}}"

animate:flip="{{duration: 200}}"

>

duration can also be a d => milliseconds function, where d is the number of pixels the element has to travel

* Note that all the transitions and animations are being applied with CSS, rather than JavaScript, meaning they won't block (or be blocked by) the main thread.

**Actions:-**

* Actions are essentially element-level lifecycle functions. They're useful for things like:
  + interfacing with third-party libraries
  + lazy-loaded images
  + tooltips
  + adding custom event handlers

e.g.: click the button to show the modal and then click outside it to close it.

* Like transitions and animations, an action can take an argument, which the action function will be called with alongside the element it belongs to. e.g: <button use:longpress={duration}

If you need to pass multiple arguments to an action, combine them into a single object, as in use:longpress={{duration, spiciness}}

**Advanced Styling:-**

* Like any other attribute, you can specify classes with a JavaScript attribute, seen here:

e.g. <button

class:selected="{current === 'foo'}"

on:click="{() => current = 'foo'}"

>foo</button>

* Often, the name of the class will be the same as the name of the value it depends on: e.g.<div class:big={big}></div>

In those cases we can use a shorthand form: e.g.<div class:big></div>

* Apart from adding styles inside style tags, you can also add styles to individual elements using the style attribute. Usually you will want to do styling through CSS, but this can come in handy for dynamic styles, especially when combined with CSS custom properties.

e.g. <p style="color: {color}; --opacity: {bgOpacity};">This is a paragraph.</p>

* Being able to set CSS properties dynamically is nice. However, this can get unwieldy if you have to write a long string. Mistakes like missing any of the semicolons could make the whole string invalid. Therefore, Svelte provides a nicer way to write inline styles with the style directive.

e.g. <p style:color={color} style:--opacity="{bgOpacity}">This is a paragraph.</p>

**Component Composition:-**

* Just like elements can have children so can components. Before a component can accept children, though, it needs to know where to put them. We do this with the <slot> element

e.g <div class="box"><slot></slot></div>

* A component can specify fallbacks for any slots that are left empty, by putting content inside the <slot> element:

e.g: <div class="box"> <slot><em>no content was provided</em></slot></div>

We can now create instances of <Box> without any children:

* The previous example contained a default slot, which renders the direct children of a component. Sometimes you will need more control over placement, such as with this <ContactCard>. In those cases, we can use named slots.

In ContactCard.svelte, add a name attribute to each slot:

e.g. <ContactCard>

<span slot="name">

P. Sherman

</span>

<span slot="address">

42 Wallaby Way<br>

Sydney

</span>

</ContactCard>

* In some cases, you may want to control parts of your component based on whether the parent passes in content for a certain slot. Perhaps you have a wrapper around that slot, and you don't want to render it if the slot is empty. Or perhaps you'd like to apply a class only if the slot is present. You can do this by checking the properties of the special $$slots variable.

$$slots is an object whose keys are the names of the slots passed in by the parent component. If the parent leaves a slot empty, then $$slots will not have an entry for that slot.

e.g. <article class:has-discussion={$$slots.comments}> and then

{#if $$slots.comments}

<div class="discussion">

<h3>Comments</h3>

<slot name="comments"></slot>

</div>

{/if}

* In this app, we have a <Hoverable> component that tracks whether the mouse is currently over it. It needs to pass that data back to the parent component, so that we can update the slotted contents.

For this, we use slot props. In Hoverable.svelte, pass the hovering value into the slot:

e.g <div on:mouseenter={enter} on:mouseleave={leave}>

<slot hovering={hovering}></slot></div>

Then, to expose hovering to the contents of the <Hoverable> component, we use the let directive:

You can rename the variable, if you want — let's call it active in the parent component:

Named slots can also have props; use the let directive on an element with a slot="..." attribute, instead of on the component itself.

**Context API:-**

* The context API provides a mechanism for components to 'talk' to each other without passing around data and functions as props, or dispatching lots of events. It's an advanced feature, but a useful one.
* There are two halves to the context API — setContext and getContext. If a component calls setContext(key, context), then any child component can retrieve the context with const context = getContext(key).
* The context object can be anything you like. Like [lifecycle functions](https://svelte.dev/tutorial/onmount), setContext and getContext must be called during component initialisation. Calling it afterwards - for example inside onMount - will throw an error.
* **Contexts vs. stores**
* Contexts and stores seem similar. They differ in that stores are available to ***any*** part of an app, while a context is only available to ***a component and its descendants***. This can be helpful if you want to use several instances of a component without the state of one interfering with the state of the others.
* In fact, you might use the two together. Since context is not reactive, values that change over time should be represented as stores:

e.g. const { these, are, stores } = getContext(...);