Events

Any component can become source if we attach event handler to it. It is done using "on:event name = {handlefn}". Now, handlefn can be defined in script tag or [illegible] inline (suggested only for smaller fns). These handle fns receive the event object containing various kinds of info. Then we can use it as per our need. In some cases we may want to prevent default behaviour like form submission immediately. In such cases, we should use e.preventDefault(). There are a bunch of such helper fn and values that help us handling the event. But in svelte there is no need to call the preventDefault manually. Rather use one of the modifiers → preventDefault. Similarly we have bunch of such modifiers that alter the normal functionality of the events.

List of such modifiers :-

1. once → forces the event to be called once
2. preventDefault → calls e.preventDefault()

3. stopPropagation → calls e.stopPropagation().
4. passive → improve scrolling performance on touch/wheel events (Svelte adds it automatically wherever it's safe)
5. self → only trigger handler ~~after~~ after first time it runs
6. capture → fires the handler during capture phase instead of bubbling phase.
7. Nonpassive → set passive : false
8. trusted → only trigger handle if event.isTrusted is set to true

components can also dispatch events. To do so the component should have a event dispatcher (provided by svelte)
↳ import {createEventDispatcher} from 'svelte';
const dispatch = createEventDispatcher();
Then dispatch the event.

The component would then be imported and implemented by a parent ~~comp~~ element. Now it's the duty of parent element to ~~[illegible]~~ listen to the event and ~~[illegible]~~ provide a handler for it.

This dispatch fn can be ~~[illegible]~~ called more than once to dispatch multiple events. It takes eventName and event details.

Component event don't bubble. If we want top/parent comp to listen to an event dispatched by a ~~[illegible]~~ deeply nested component, the ~~[illegible]~~ intermediate component must forward the ~~request~~ event.

```
app → has to listen to dispatched event
 ↳ outer
    ↳ inner → dispatch event
```

One soln is simply attaching ~~[illegible]~~ event dispatcher in the outer comp.
Or simple use shorthand → on: message in the nested comp declared in the intermediate comps. If it is not handled explicitly then it is simply forwarded.

We can also forward the DOM event like click, input, etc. from child to parent without explicity handling them in the child comp. Basically child ~~can~~ has just specified that it has an on-click event but this fn is not defined in it. Rather parent comp. implementing the child comp. specified the handler. (there is no need of createEventDispatcher for DOM events but they're reqd. in other event)

Bindings

In general data flow is from top to bottom. But we can also send data from children to parent. Not only that, an element capable of changing ~~a~~ a state can force the elements using this state to use the updated state. All this is implemented using bind. We don't use bind with variable/declaration rather use it with the elements capable of changing ~~it~~ state like input.

ex

<input bind: value = {state-name}

Now, all the elements using this state would show the modified value.

bind also helps in handling numeric values. Everything is considered string in DOM. So, in order to use, we'd have convert it. But ~~bind~~ if we bind ~~these~~ such states then svelte ~~a~~ takes care of it for us

In checkboxes, bind is attached to input.checked

In select elements, bind is attached to the selected element i.e., bind: value = {selected}
→ The selected option is stored here.

The input and select ~~B~~ values need to be bind to submit. Otherwise they won't be submitted as changes made by input and select won't change the ~~old~~ original variable.

for radio and normal checkbox, the selected value is not directly available. (Leave the baggage of checked in checkbox for some time). So, we use group binding. All the radio box of one type are identified using the same name. when we use bind:group = {grp-name} then the selected value would be resolved.
case is similar for multiple checkboxes, only change is that the selected values are added to array.
Select can have multiple values by using attribute multiple and binding the values attribute. (hold ctrl and make selections)
Textarea's value handling is similar to normal input.
In case the variable name is also 'value', we can use shorthand <tA> bind:value </tA>.

life cycle

In svelte, components have a lifecycle that consists of various phases:

1) Mounting (creation) → when component is initiated and rendered for the first time, this is where onMount comes in. This onMount allows us to implement functionities in a component when it is first loaded.
2) Updating → when a reactive value in the component changes, causing a re-render.
3) Unmounting (Destruction) → When component is removed from DOM. This happens when:
   i) Parent comp conditionally renders or destroy this comp
   ii) Navigation changes
   iii) Event triggering destruction of component.

OnMount

svelte specific. Used to run code once after comp has been added to the DOM and rendered.

• It is useful when dealing with side effects like DOM manipulation or external data fetching.

• The code inside onMount runs after comp. is mounted to the DOM, it can return a cleanup f that runs when comp. is destroyed.

Cleanup f^n stops any ongoing processes that might still be running like Timers, Animations, etc.

* beforeUpdate and afterUpdate

---

beforeUpdate f^n schedules work to happen immediately before DOM is updated. afterUpdate is its counterpart used for running code once the DOM is in sync with your data.

Together, they're useful for doing things imperatively that are difficult to achieve in a state driven way, like updat scroll pos^n of an element.

* Tick

↳ It ensures that any pending changes to the DOM are fully applied before we run next ~~bit~~ of code.
It forces the component to wait until the DOM is updated with the next text value.
It is implement using await tick()