Tags

A lowercase tag, like `<div>`, denotes a regular HTML element. A capitalised tag, such as `<Widget>` or `<Namespace.Widget>`, indicates a *component*.

```
<script>
    import Widget from './Widget.svelte';
</script>

<div>
    <Widget/>
</div>
```

Attributes and props

By default, attributes work exactly like their HTML counterparts.

```
<div class="foo">
    <button disabled>can't touch this</button>
</div>
```

As in HTML, values may be unquoted.

```
<input type=checkbox>
```

Attribute values can contain JavaScript expressions.

```
<a href="page/{p}">page {p}</a>
```

Or they can *be* JavaScript expressions.

```
<button disabled={!clickable}>...</button>
```

Boolean attributes are included on the element if their value is truthy and excluded if it's falsy.

All other attributes are included unless their value is nullish (`null` or `undefined`).

```
<input required={false} placeholder="This input field is not required">
<div title={null}>This div has no title attribute</div>
```

An expression might include characters that would cause syntax highlighting to fail in regular HTML, so quoting the value is permitted. The quotes do not affect how the value is parsed:

```
<button disabled="{number !== 42}">...</button>
```

When the attribute name and value match (`name={name}`), they can be replaced with `{name}` .

```
<!-- These are equivalent -->
<button disabled={disabled}>...</button>
<button {disabled}>...</button>
```

By convention, values passed to components are referred to as *properties* or *props* rather than *attributes*, which are a feature of the DOM.

As with elements, `name={name}` can be replaced with the `{name}` shorthand.

```
<Widget foo={bar} answer={42} text="hello"/>
```

Spread attributes allow many attributes or properties to be passed to an element or component at once.

An element or component can have multiple spread attributes, interspersed with regular ones.

```
<Widget {...things}/>
```

`$$props` references all props that are passed to a component, including ones that are not declared with `export` . It is not generally recommended, as it is difficult for Svelte to optimise. But it can be useful in rare cases – for example, when you don't know at compile time what props might be passed to a component.

```
<Widget {...$$props}/>
```

`$$restProps` contains only the props which are *not* declared with `export` . It can be used to pass down other unknown attributes to an element in a component. It shares the same optimisation problems as `$$props` , and is likewise not recommended.

```
<input {...$$restProps}>
```

> *The `value` attribute of an `input` element or its children `option` elements must not be set with spread attributes when using `bind:group` or `bind:checked` . Svelte needs to be able to see the element's `value` directly in the markup in these cases so that it can link it to the bound variable.*

Text expressions

```
{expression}
```

Text can also contain JavaScript expressions:

> *If you're using a regular expression (`RegExp`) literal notation, you'll need to wrap it in parentheses.*

```
<h1>Hello {name}!</h1>
<p>{a} + {b} = {a + b}.</p>
```

```
<div>{(/^[A-Za-z ]+$/).test(value) ? x : y}</div>
```

Comments

You can use HTML comments inside components.

```
<!-- this is a comment! -->
<h1>Hello world</h1>
```

Comments beginning with `svelte-ignore` disable warnings for the next block of markup. Usually these are accessibility warnings; make sure that you're disabling them for a good reason.

```
<!-- svelte-ignore a11y-autofocus -->
<input bind:value={name} autofocus>
```

{#if ...}

```
{#if expression}...{/if}
```

```
{#if expression}...{:else if expression}...{/if}
```

```
{#if expression}...{:else}...{/if}
```

Content that is conditionally rendered can be wrapped in an if block.

```
{#if answer === 42}
    <p>what was the question?</p>
{/if}
```

Additional conditions can be added with `{:else if expression}`, optionally ending in an `{:else}` clause.

```
{#if porridge.temperature > 100}
    <p>too hot!</p>
{:else if 80 > porridge.temperature}
    <p>too cold!</p>
{:else}
    <p>just right!</p>
{/if}
```

{#each ...}

```
{#each expression as name}...{/each}
```

```
{#each expression as name, index}...{/each}
```

```
{#each expression as name (key)}...{/each}
```

```
{#each expression as name, index (key)}...{/each}
```

```
{#each expression as name}...{:else}...{/each}
```

Iterating over lists of values can be done with an each block.

```
<h1>Shopping list</h1>
<ul>
    {#each items as item}
        <li>{item.name} x {item.qty}</li>
    {/each}
</ul>
```

You can use each blocks to iterate over any array or array-like value — that is, any object with a `length` property.

An each block can also specify an *index*, equivalent to the second argument in an `array.map(...)` callback:

```
{#each items as item, i}
    <li>{i + 1}: {item.name} x {item.qty}</li>
{/each}
```

If a *key* expression is provided — which must uniquely identify each list item — Svelte will use it to diff the list when data changes, rather than adding or removing items at the end. The key can be any object, but strings and numbers are recommended since they allow identity to persist when the objects themselves change.

```
{#each items as item (item.id)}
    <li>{item.name} x {item.qty}</li>
{/each}

<!-- or with additional index value -->
{#each items as item, i (item.id)}
    <li>{i + 1}: {item.name} x {item.qty}</li>
{/each}
```

You can freely use destructuring and rest patterns in each blocks.

```
{#each items as { id, name, qty }, i (id)}
    <li>{i + 1}: {name} x {qty}</li>
{/each}

{#each objects as { id, ...rest }}
```

```
    <li><span>{id}</span><MyComponent {...rest}/></li>
{/each}

{#each items as [id, ...rest]}
    <li><span>{id}</span><MyComponent values={rest}/></li>
{/each}
```

An each block can also have an `{:else}` clause, which is rendered if the list is empty.

```
{#each todos as todo}
    <p>{todo.text}</p>
{:else}
    <p>No tasks today!</p>
{/each}
```

{#await ...}

```
{#await expression}...{:then name}...{:catch name}...{/await}
```

```
{#await expression}...{:then name}...{/await}
```

```
{#await expression then name}...{/await}
```

```
{#await expression catch name}...{/await}
```

Await blocks allow you to branch on the three possible states of a Promise — pending, fulfilled or rejected. In SSR mode, only the pending state will be rendered on the server.

```
{#await promise}
    <!-- promise is pending -->
    <p>waiting for the promise to resolve...</p>
{:then value}
    <!-- promise was fulfilled -->
    <p>The value is {value}</p>
{:catch error}
    <!-- promise was rejected -->
    <p>Something went wrong: {error.message}</p>
{/await}
```

The `catch` block can be omitted if you don't need to render anything when the promise rejects (or no error is possible).

```
{#await promise}
    <!-- promise is pending -->
    <p>waiting for the promise to resolve...</p>
```

```
{:then value}
    <!-- promise was fulfilled -->
    <p>The value is {value}</p>
{/await}
```

If you don't care about the pending state, you can also omit the initial block.

```
{#await promise then value}
    <p>The value is {value}</p>
{/await}
```

Similarly, if you only want to show the error state, you can omit the `then` block.

```
{#await promise catch error}
    <p>The error is {error}</p>
{/await}
```

{#key ...}

```
{#key expression}...{/key}
```

Key blocks destroy and recreate their contents when the value of an expression changes.

This is useful if you want an element to play its transition whenever a value changes.

```
{#key value}
    <div transition:fade>{value}</div>
{/key}
```

When used around components, this will cause them to be reinstantiated and reinitialised.

```
{#key value}
    <Component />
{/key}
```

{@html ...}

```
{@html expression}
```

In a text expression, characters like `<` and `>` are escaped; however, with HTML expressions, they're not.

The expression should be valid standalone HTML — `{@html "<div>"}content{@html "</div>"}` will *not* work, because `</div>` is not valid HTML. It also will *not* compile Svelte code.

> *Svelte does not sanitize expressions before injecting HTML. If the data comes from an untrusted source, you must sanitize it, or you are exposing your users to an XSS vulnerability.*

```
<div class="blog-post">
    <h1>{post.title}</h1>
    {@html post.content}
</div>
```

{@debug ...}

```
{@debug}
```

```
{@debug var1, var2, ..., varN}
```

The `{@debug ...}` tag offers an alternative to `console.log(...)`. It logs the values of specific variables whenever they change, and pauses code execution if you have devtools open.

```
<script>
    let user = {
        firstname: 'Ada',
        lastname: 'Lovelace'
    };
</script>

{@debug user}

<h1>Hello {user.firstname}!</h1>
```

`{@debug ...}` accepts a comma-separated list of variable names (not arbitrary expressions).

```
<!-- Compiles -->
{@debug user}
{@debug user1, user2, user3}

<!-- WON'T compile -->
{@debug user.firstname}
{@debug myArray[0]}
{@debug !isReady}
{@debug typeof user === 'object'}
```

The `{@debug}` tag without any arguments will insert a `debugger` statement that gets triggered when *any* state changes, as opposed to the specified variables.

{@const ...}

```
{@const assignment}
```

The `{@const ...}` tag defines a local constant.

```
<script>
  export let boxes;
</script>

{#each boxes as box}
  {@const area = box.width * box.height}
    {box.width} * {box.height} = {area}
{/each}
```

`{@const}` is only allowed as direct child of `{#each}`, `{:then}`, `{:catch}`, `<Component />` or `<svelte:fragment />`.

Element directives

As well as attributes, elements can have *directives*, which control the element's behaviour in some way.

on:*eventname*

```
on:eventname={handler}
```

```
on:eventname|modifiers={handler}
```

Use the `on:` directive to listen to DOM events.

```
<script>
    let count = 0;

    function handleClick(event) {
        count += 1;
    }
</script>

<button on:click={handleClick}>
    count: {count}
</button>
```

Handlers can be declared inline with no performance penalty. As with attributes, directive values may be quoted for the sake of syntax highlighters.

```
<button on:click="{() => count += 1}">
    count: {count}
</button>
```

Add *modifiers* to DOM events with the `|` character.

```
<form on:submit|preventDefault={handleSubmit}>
    <!-- the `submit` event's default is prevented,
         so the page won't reload -->
</form>
```

The following modifiers are available:

- `preventDefault` — calls `event.preventDefault()` before running the handler
- `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
- `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.

Modifiers can be chained together, e.g. `on:click|once|capture={...}`.

If the `on:` directive is used without a value, the component will *forward* the event, meaning that a consumer of the component can listen for it.

```
<button on:click>
    The component itself will emit the click event
</button>
```

It's possible to have multiple event listeners for the same event:

```
<script>
    let counter = 0;
    function increment() {
        counter = counter + 1;
    }

    function track(event) {
        trackEvent(event)
    }
</script>

<button on:click={increment} on:click={track}>Click me!</button>
```

bind:*property*

```
bind:property={variable}
```

Data ordinarily flows down, from parent to child. The `bind:` directive allows data to flow the other way, from child to parent. Most bindings are specific to particular elements.

The simplest bindings reflect the value of a property, such as `input.value`.

```
<input bind:value={name}>
<textarea bind:value={text}></textarea>

<input type="checkbox" bind:checked={yes}>
```

If the name matches the value, you can use a shorthand.

```
<!-- These are equivalent -->
<input bind:value={value}>
<input bind:value>
```

Numeric input values are coerced; even though `input.value` is a string as far as the DOM is concerned, Svelte will treat it as a number. If the input is empty or invalid (in the case of `type="number"`), the value is `undefined`.

```
<input type="number" bind:value={num}>
<input type="range" bind:value={num}>
```

On `<input>` elements with `type="file"`, you can use `bind:files` to get the `FileList` of selected files. It is readonly.

```
<label for="avatar">Upload a picture:</label>
<input
    accept="image/png, image/jpeg"
    bind:files
    id="avatar"
    name="avatar"
    type="file"
/>
```

`bind:` can be used together with `on:` directives. The order that they are defined in determines the value of the bound variable when the event handler is called.

```
<script>
    let value = 'Hello World';
</script>

<input
    on:input="{() => console.log('Old value:', value)}"
    bind:value
    on:input="{() => console.log('New value:', value)}"
/>
```

Binding `<select>` value

A `<select>` value binding corresponds to the `value` property on the selected `<option>`, which can be any value (not just strings, as is normally the case in the DOM).

```
<select bind:value={selected}>
    <option value={a}>a</option>
    <option value={b}>b</option>
    <option value={c}>c</option>
</select>
```

A `<select multiple>` element behaves similarly to a checkbox group.

```
<select multiple bind:value={fillings}>
    <option value="Rice">Rice</option>
    <option value="Beans">Beans</option>
    <option value="Cheese">Cheese</option>
    <option value="Guac (extra)">Guac (extra)</option>
</select>
```

When the value of an `<option>` matches its text content, the attribute can be omitted.

```
<select multiple bind:value={fillings}>
    <option>Rice</option>
    <option>Beans</option>
    <option>Cheese</option>
    <option>Guac (extra)</option>
</select>
```

Elements with the `contenteditable` attribute support `innerHTML` and `textContent` bindings.

```
<div contenteditable="true" bind:innerHTML={html}></div>
```

`<details>` elements support binding to the `open` property.

```
<details bind:open={isOpen}>
    <summary>Details</summary>
    <p>
        Something small enough to escape casual notice.
    </p>
</details>
```

Media element bindings

Media elements (`<audio>` and `<video>`) have their own set of bindings — six *readonly* ones...

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

- `buffered` (readonly) — an array of `{start, end}` objects
- `played` (readonly) — ditto
- `seekable` (readonly) — ditto
- `seeking` (readonly) — boolean
- `ended` (readonly) — boolean

...and five *two-way* bindings:

- `currentTime` — the current playback time in the video, in seconds
- `playbackRate` — how fast or slow to play the video, where 1 is 'normal'
- `paused` — this one should be self-explanatory
- `volume` — a value between 0 and 1
- `muted` — a boolean value indicating whether the player is muted

Videos additionally have readonly `videoWidth` and `videoHeight` bindings.

```
<video
    src={clip}
    bind:duration
    bind:buffered
    bind:played
    bind:seekable
    bind:seeking
    bind:ended
    bind:currentTime
    bind:playbackRate
    bind:paused
    bind:volume
    bind:muted
    bind:videoWidth
    bind:videoHeight
></video>
```

Block-level element bindings

Block-level elements have 4 readonly bindings, measured using a technique similar to this one:

- `clientWidth`
- `clientHeight`
- `offsetWidth`
- `offsetHeight`

```
<div
    bind:offsetWidth={width}
    bind:offsetHeight={height}
>
    <Chart {width} {height}/>
</div>
```

bind:group

```
bind:group={variable}
```

Inputs that work together can use `bind:group`.

```
<script>
    let tortilla = 'Plain';
    let fillings = [];
</script>

<!-- grouped radio inputs are mutually exclusive -->
<input type="radio" bind:group={tortilla} value="Plain">
<input type="radio" bind:group={tortilla} value="Whole wheat">
<input type="radio" bind:group={tortilla} value="Spinach">

<!-- grouped checkbox inputs populate an array -->
<input type="checkbox" bind:group={fillings} value="Rice">
<input type="checkbox" bind:group={fillings} value="Beans">
<input type="checkbox" bind:group={fillings} value="Cheese">
<input type="checkbox" bind:group={fillings} value="Guac (extra)">
```

bind:this

```
bind:this={dom_node}
```

To get a reference to a DOM node, use `bind:this`.

```
<script>
    import { onMount } from 'svelte';

    let canvasElement;

    onMount(() => {
        const ctx = canvasElement.getContext('2d');
        drawStuff(ctx);
    });
</script>

<canvas bind:this={canvasElement}></canvas>
```

class:*name*

```
class:name={value}
```

```
class:name
```

A `class:` directive provides a shorter way of toggling a class on an element.

```
<!-- These are equivalent -->
<div class="{active ? 'active' : ''}">...</div>
<div class:active={active}>...</div>

<!-- Shorthand, for when name and value match -->
<div class:active>...</div>

<!-- Multiple class toggles can be included -->
<div class:active class:inactive={!active} class:isAdmin>...</div>
```

style:*property*

```
style:property={value}
```

```
style:property="value"
```

```
style:property
```

The `style:` directive provides a shorthand for setting multiple styles on an element.

```
<!-- These are equivalent -->
<div style:color="red">...</div>
<div style="color: red;">...</div>

<!-- Variables can be used -->
<div style:color={myColor}>...</div>

<!-- Shorthand, for when property and variable name match -->
<div style:color>...</div>

<!-- Multiple styles can be included -->
<div style:color style:width="12rem" style:background-color={darkMode ? "black" :
"white"}>...</div>
```

When `style:` directives are combined with `style` attributes, the directives will take precedence:

```
<div style="color: blue;" style:color="red">This will be red</div>
```

use:*action*

```
use:action
```

```
use:action={parameters}
```

```
action = (node: HTMLElement, parameters: any) => {
    update?: (parameters: any) => void,
    destroy?: () => void
}
```

Actions are functions that are called when an element is created. They can return an object with a `destroy` method that is called after the element is unmounted:

```
<script>
    function foo(node) {
        // the node has been mounted in the DOM

        return {
            destroy() {
                // the node has been removed from the DOM
            }
        };
    }
</script>

<div use:foo></div>
```

An action can have a parameter. If the returned value has an `update` method, it will be called whenever that parameter changes, immediately after Svelte has applied updates to the markup.

> *Don't worry about the fact that we're redeclaring the `foo` function for every component instance — Svelte will hoist any functions that don't depend on local state out of the component definition.*

```
<script>
    export let bar;

    function foo(node, bar) {
        // the node has been mounted in the DOM

        return {
            update(bar) {
                // the value of `bar` has changed
            },

            destroy() {
                // the node has been removed from the DOM
            }
        };
    }
</script>

<div use:foo={bar}></div>
```

transition:*fn*

```
transition:fn
```

```
transition:fn={params}
```

```
transition:fn|local
```

```
transition:fn|local={params}
```

```
transition = (node: HTMLElement, params: any) => {
    delay?: number,
    duration?: number,
    easing?: (t: number) => number,
    css?: (t: number, u: number) => string,
    tick?: (t: number, u: number) => void
}
```

A transition is triggered by an element entering or leaving the DOM as a result of a state change.

When a block is transitioning out, all elements inside the block, including those that do not have their own transitions, are kept in the DOM until every transition in the block has completed.

The `transition:` directive indicates a *bidirectional* transition, which means it can be smoothly reversed while the transition is in progress.

```
{#if visible}
    <div transition:fade>
        fades in and out
    </div>
{/if}
```

> *By default intro transitions will not play on first render. You can modify this behaviour by setting `intro: true` when you create a component.*

Transition parameters

Like actions, transitions can have parameters.

(The double `{{curlies}}` aren't a special syntax; this is an object literal inside an expression tag.)

```
{#if visible}
    <div transition:fade="{{ duration: 2000 }}">
        flies in, fades out over two seconds
    </div>
{/if}
```

Custom transition functions

Transitions can use custom functions. If the returned object has a `css` function, Svelte will create a CSS animation that plays on the element.

The `t` argument passed to `css` is a value between `0` and `1` after the `easing` function has been applied. *In* transitions run from `0` to `1`, *out* transitions run from `1` to `0` — in other words `1` is the element's natural state, as though no transition had been applied. The `u` argument is equal to `1 - t`.

The function is called repeatedly *before* the transition begins, with different `t` and `u` arguments.

```
<script>
    import { elasticOut } from 'svelte/easing';

    export let visible;

    function whoosh(node, params) {
        const existingTransform = getComputedStyle(node).transform.replace('none',
'');

        return {
            delay: params.delay || 0,
            duration: params.duration || 400,
            easing: params.easing || elasticOut,
            css: (t, u) => `transform: ${existingTransform} scale(${t})`
        };
    }
</script>

{#if visible}
    <div in:whoosh>
        whooshes in
    </div>
{/if}
```

A custom transition function can also return a `tick` function, which is called *during* the transition with the same `t` and `u` arguments.

> *If it's possible to use `css` instead of `tick`, do so — CSS animations can run off the main thread, preventing jank on slower devices.*

```
<script>
    export let visible = false;

    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 = ~~(text.length * t);
                node.textContent = text.slice(0, i);
            }
        };
    }
</script>

{#if visible}
    <p in:typewriter="{{ speed: 1 }}">
        The quick brown fox jumps over the lazy dog
    </p>
{/if}
```

If a transition returns a function instead of a transition object, the function will be called in the next microtask. This allows multiple transitions to coordinate, making crossfade effects possible.

Transition events

An element with transitions will dispatch the following events in addition to any standard DOM events:

- `introstart`
- `introend`
- `outrostart`
- `outroend`

```
{#if visible}
    <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>
{/if}
```

Local transitions only play when the block they belong to is created or destroyed, *not* when parent blocks are created or destroyed.

```
{#if x}
    {#if y}
        <p transition:fade>
            fades in and out when x or y change
        </p>

        <p transition:fade|local>
            fades in and out only when y changes
        </p>
    {/if}
{/if}
```

in:*fn*/out:*fn*

```
in:fn
```

```
in:fn={params}
```

```
in:fn|local
```

```
in:fn|local={params}
```

```
out:fn
```

```
out:fn={params}
```

```
out:fn|local
```

```
out:fn|local={params}
```

Similar to `transition:`, but only applies to elements entering (`in:`) or leaving (`out:`) the DOM.

Unlike with `transition:`, transitions applied with `in:` and `out:` are not bidirectional — an in transition will continue to 'play' alongside the out transition, rather than reversing, if the block is outroed while the transition is in progress. If an out transition is aborted, transitions will restart from scratch.

```
{#if visible}
    <div in:fly out:fade>
        flies in, fades out
    </div>
{/if}
```

animate:*fn*

```
animate:name
```

```
animate:name={params}
```

```
animation = (node: HTMLElement, { from: DOMRect, to: DOMRect } , params: any) => {
    delay?: number,
    duration?: number,
    easing?: (t: number) => number,
    css?: (t: number, u: number) => string,
    tick?: (t: number, u: number) => void
}
```

```
DOMRect {
    bottom: number,
    height: number,
    left: number,
    right: number,
    top: number,
    width: number,
    x: number,
    y: number
}
```

An animation is triggered when the contents of a keyed each block are re-ordered. Animations do not run when an element is added or removed, only when the index of an existing data item within the each block changes. Animate directives must be on an element that is an *immediate* child of a keyed each block.

Animations can be used with Svelte's built-in animation functions or custom animation functions.

```
<!-- When `list` is reordered the animation will run-->
{#each list as item, index (item)}
    <li animate:flip>{item}</li>
{/each}
```

Animation Parameters

As with actions and transitions, animations can have parameters.

(The double `{{curlies}}` aren't a special syntax; this is an object literal inside an expression tag.)

```
{#each list as item, index (item)}
    <li animate:flip="{{ delay: 500 }}">{item}</li>
{/each}
```

Custom animation functions

Animations can use custom functions that provide the `node`, an `animation` object and any `parameters` as arguments. The `animation` parameter is an object containing `from` and `to` properties each containing a DOMRect describing the geometry of the element in its `start` and `end` positions. The `from` property is the DOMRect of the element in its starting position, the `to` property is the DOMRect of the element in its final position after the list has been reordered and the DOM updated.

If the returned object has a `css` method, Svelte will create a CSS animation that plays on the element.

The `t` argument passed to `css` is a value that goes from `0` and `1` after the `easing` function has been applied. The `u` argument is equal to `1 - t`.

The function is called repeatedly *before* the animation begins, with different `t` and `u` arguments.

```
<script>
    import { cubicOut } from 'svelte/easing';

    function whizz(node, { from, to }, params) {

        const dx = from.left - to.left;
        const dy = from.top - to.top;

        const d = Math.sqrt(dx * dx + dy * dy);

        return {
            delay: 0,
            duration: Math.sqrt(d) * 120,
            easing: cubicOut,
            css: (t, u) =>
                `transform: translate(${u * dx}px, ${u * dy}px)
rotate(${t*360}deg);`
        };
    }
</script>

{#each list as item, index (item)}
    <div animate:whizz>{item}</div>
{/each}
```

A custom animation function can also return a `tick` function, which is called *during* the animation with the same `t` and `u` arguments.

> *If it's possible to use `css` instead of `tick`, do so — CSS animations can run off the main thread, preventing jank on slower devices.*

```
<script>
    import { cubicOut } from 'svelte/easing';

    function whizz(node, { from, to }, params) {

        const dx = from.left - to.left;
        const dy = from.top - to.top;
```

```
        const d = Math.sqrt(dx * dx + dy * dy);

        return {
        delay: 0,
        duration: Math.sqrt(d) * 120,
        easing: cubicOut,
        tick: (t, u) =>
            Object.assign(node.style, {
                color: t > 0.5 ? 'Pink' : 'Blue'
            });
    };
    }
</script>

{#each list as item, index (item)}
    <div animate:whizz>{item}</div>
{/each}
```

Component directives

on:*eventname*

```
on:eventname={handler}
```

Components can emit events using createEventDispatcher, or by forwarding DOM events. Listening for component events looks the same as listening for DOM events:

```
<SomeComponent on:whatever={handler}/>
```

As with DOM events, if the `on:` directive is used without a value, the component will *forward* the event, meaning that a consumer of the component can listen for it.

```
<SomeComponent on:whatever/>
```

--style-props

```
--style-props="anycssvalue"
```

You can also pass styles as props to components for the purposes of theming, using CSS custom properties.

Svelte's implementation is essentially syntactic sugar for adding a wrapper element. This example:

```
<Slider
  bind:value
  min={0}
  --rail-color="black"
```

```
  --track-color="rgb(0, 0, 255)"
/>
```

Desugars to this:

```
<div style="display: contents; --rail-color: black; --track-color: rgb(0, 0, 255)">
  <Slider
    bind:value
    min={0}
    max={100}
  />
</div>
```

Note: Since this is an extra `<div>`, beware that your CSS structure might accidentally target this. Be mindful of this added wrapper element when using this feature.

Svelte's CSS Variables support allows for easily themable components:

```
<!-- Slider.svelte -->
<style>
  .potato-slider-rail {
    background-color: var(--rail-color, var(--theme-color, 'purple'));
  }
</style>
```

So you can set a high level theme color:

```
/* global.css */
html {
  --theme-color: black;
}
```

Or override it at the consumer level:

```
<Slider --rail-color="goldenrod"/>
```

bind:*property*

```
bind:property={variable}
```

You can bind to component props using the same syntax as for elements.

```
<Keypad bind:value={pin}/>
```

bind:this

```
bind:this={component_instance}
```

Components also support `bind:this`, allowing you to interact with component instances programmatically.

> *Note that we can't do `{cart.empty}` since `cart` is `undefined` when the button is first rendered and throws an error.*

```
<ShoppingCart bind:this={cart}/>

<button on:click={() => cart.empty()}>
    Empty shopping cart
</button>
```

`<slot>`

```
<slot><!-- optional fallback --></slot>
```

```
<slot name="x"><!-- optional fallback --></slot>
```

```
<slot prop={value}></slot>
```

Components can have child content, in the same way that elements can.

The content is exposed in the child component using the `<slot>` element, which can contain fallback content that is rendered if no children are provided.

```
<!-- Widget.svelte -->
<div>
    <slot>
        this fallback content will be rendered when no content is provided, like in
the first example
    </slot>
</div>

<!-- App.svelte -->
<Widget></Widget> <!-- this component will render the default content -->

<Widget>
    <p>this is some child content that will overwrite the default slot content</p>
</Widget>
```

`<slot name="name">`

Named slots allow consumers to target specific areas. They can also have fallback content.

```
<!-- Widget.svelte -->
<div>
    <slot name="header">No header was provided</slot>
    <p>Some content between header and footer</p>
    <slot name="footer"></slot>
</div>

<!-- App.svelte -->
<Widget>
    <h1 slot="header">Hello</h1>
    <p slot="footer">Copyright (c) 2019 Svelte Industries</p>
</Widget>
```

Components can be placed in a named slot using the syntax `<Component slot="name" />`. In order to place content in a slot without using a wrapper element, you can use the special element `<svelte:fragment>`.

```
<!-- Widget.svelte -->
<div>
    <slot name="header">No header was provided</slot>
    <p>Some content between header and footer</p>
    <slot name="footer"></slot>
</div>

<!-- App.svelte -->
<Widget>
    <HeaderComponent slot="header" />
    <svelte:fragment slot="footer">
        <p>All rights reserved.</p>
        <p>Copyright (c) 2019 Svelte Industries</p>
    </svelte:fragment>
</Widget>
```

`$$slots`

`$$slots` is an object whose keys are the names of the slots passed into the component by the parent. If the parent does not pass in a slot with a particular name, that name will not be present in `$$slots`. This allows components to render a slot (and other elements, like wrappers for styling) only if the parent provides it.

Note that explicitly passing in an empty named slot will add that slot's name to `$$slots`. For example, if a parent passes `<div slot="title" />` to a child component, `$$slots.title` will be truthy within the child.

```
<!-- Card.svelte -->
<div>
    <slot name="title"></slot>
    {#if $$slots.description}
        <!-- This <hr> and slot will render only if a slot named "description" is
provided. -->
        <hr>
        <slot name="description"></slot>
```

```
    {/if}
</div>

<!-- App.svelte -->
<Card>
    <h1 slot="title">Blog Post Title</h1>
    <!-- No slot named "description" was provided so the optional slot will not be
rendered. -->
</Card>
```

`<slot key={value}>`

Slots can be rendered zero or more times, and can pass values *back* to the parent using props. The parent exposes the values to the slot template using the `let:` directive.

The usual shorthand rules apply — `let:item` is equivalent to `let:item={item}`, and `<slot {item}>` is equivalent to `<slot item={item}>`.

```
<!-- FancyList.svelte -->
<ul>
    {#each items as item}
        <li class="fancy">
            <slot prop={item}></slot>
        </li>
    {/each}
</ul>

<!-- App.svelte -->
<FancyList {items} let:prop={thing}>
    <div>{thing.text}</div>
</FancyList>
```

Named slots can also expose values. The `let:` directive goes on the element with the `slot` attribute.

```
<!-- FancyList.svelte -->
<ul>
    {#each items as item}
        <li class="fancy">
            <slot name="item" {item}></slot>
        </li>
    {/each}
</ul>

<slot name="footer"></slot>

<!-- App.svelte -->
<FancyList {items}>
    <div slot="item" let:item>{item.text}</div>
    <p slot="footer">Copyright (c) 2019 Svelte Industries</p>
</FancyList>
```

`<svelte:self>`

The `<svelte:self>` element allows a component to include itself, recursively.

It cannot appear at the top level of your markup; it must be inside an if or each block or passed to a component's slot to prevent an infinite loop.

```
<script>
    export let count;
</script>

{#if count > 0}
    <p>counting down... {count}</p>
    <svelte:self count="{count - 1}"/>
{:else}
    <p>lift-off!</p>
{/if}
```

`<svelte:component>`

```
<svelte:component this={expression}/>
```

The `<svelte:component>` element renders a component dynamically, using the component constructor specified as the `this` property. When the property changes, the component is destroyed and recreated.

If `this` is falsy, no component is rendered.

```
<svelte:component this={currentSelection.component} foo={bar}/>
```

`<svelte:window>`

```
<svelte:window on:event={handler}/>
```

```
<svelte:window bind:prop={value}/>
```

The `<svelte:window>` element allows you to add event listeners to the `window` object without worrying about removing them when the component is destroyed, or checking for the existence of `window` when server-side rendering.

Unlike `<svelte:self>`, this element may only appear the top level of your component and must never be inside a block or element.

```
<script>
    function handleKeydown(event) {
        alert(`pressed the ${event.key} key`);
```

```
    }
</script>

<svelte:window on:keydown={handleKeydown}/>
```

You can also bind to the following properties:

- `innerWidth`
- `innerHeight`
- `outerWidth`
- `outerHeight`
- `scrollX`
- `scrollY`
- `online` — an alias for window.navigator.onLine

All except `scrollX` and `scrollY` are readonly.

```
<svelte:window bind:scrollY={y}/>
```

> *Note that the page will not be scrolled to the initial value to avoid accessibility issues. Only subsequent changes to the bound variable of `scrollX` and `scrollY` will cause scrolling. However, if the scrolling behaviour is desired, call `scrollTo()` in `onMount()`.*

`<svelte:body>`

```
<svelte:body on:event={handler}/>
```

Similarly to `<svelte:window>`, this element allows you to add listeners to events on `document.body`, such as `mouseenter` and `mouseleave`, which don't fire on `window`. It also lets you use actions on the `<body>` element.

`<svelte:body>` also has to appear at the top level of your component.

```
<svelte:body
    on:mouseenter={handleMouseenter}
    on:mouseleave={handleMouseleave}
    use:someAction
/>
```

`<svelte:head>`

```
<svelte:head>...</svelte:head>
```

This element makes it possible to insert elements into `document.head`. During server-side rendering, `head` content is exposed separately to the main `html` content.

As with `<svelte:window>` and `<svelte:body>`, this element has to appear at the top level of your component and cannot be inside a block or other element.

```
<svelte:head>
    <link rel="stylesheet" href="/tutorial/dark-theme.css">
</svelte:head>
```

`<svelte:options>`

```
<svelte:options option={value}/>
```

The `<svelte:options>` element provides a place to specify per-component compiler options, which are detailed in the compiler section. The possible options are:

- `immutable={true}` — you never use mutable data, so the compiler can do simple referential equality checks to determine if values have changed
- `immutable={false}` — the default. Svelte will be more conservative about whether or not mutable objects have changed
- `accessors={true}` — adds getters and setters for the component's props
- `accessors={false}` — the default
- `namespace="..."` — the namespace where this component will be used, most commonly "svg"; use the "foreign" namespace to opt out of case-insensitive attribute names and HTML-specific warnings
- `tag="..."` — the name to use when compiling this component as a custom element

```
<svelte:options tag="my-custom-element"/>
```

`<svelte:fragment>`

The `<svelte:fragment>` element allows you to place content in a named slot without wrapping it in a container DOM element. This keeps the flow layout of your document intact.

```
<!-- Widget.svelte -->
<div>
    <slot name="header">No header was provided</slot>
    <p>Some content between header and footer</p>
    <slot name="footer"></slot>
</div>

<!-- App.svelte -->
<Widget>
    <h1 slot="header">Hello</h1>
    <svelte:fragment slot="footer">
        <p>All rights reserved.</p>
        <p>Copyright (c) 2019 Svelte Industries</p>
    </svelte:fragment>
</Widget>
```