Svelte Cheatsheet

A concise reference for Svelte syntax, directives, lifecycle methods, and best practices. This cheat sheet provides a quick guide to building reactive web applications with Svelte.

Svelte Basics

Component Syntax

Svelte components are defined in `.svelte` files, containing HTML, CSS, and JavaScript.

```
<!-- MyComponent.svelte -->
<script>
  let name = 'world';
</script>

<h1>Hello {name}!</h1>

<style>
  h1 {
    color: blue;
  }
</style>
```

Key Features:

- Declarative components
- Reactive statements (`$:`)
- Built-in transitions and animations
- Scoped CSS

Reactivity

`let` keyword	Declares a reactive variable. Svelte tracks assignments to these variables and updates the DOM accordingly.
	`let count = 0;`
	`function increment() { count += 1; }`

`$:` reactive statements	Re-executes the code block whenever any of the referenced variables change.
	`let count = 0;`
	`$: doubled = count * 2;`

Updating Arrays and Objects	Arrays and objects require special handling to trigger reactivity. Use methods like `push`, `pop`, `shift`, `unshift`, `splice` for arrays, or reassign the entire object/array.
	`// Array update`
	`myArray = [...myArray, newValue];`
	`// Object update`
	`myObject = { ...myObject, newProperty: value };`

Basic Directives

- `bind:` : Binds an element property to a component variable.
- `on:` : Attaches an event listener to an element.
- `class:` : Conditionally adds or removes a CSS class.
- `style:` : Conditionally applies a CSS style.

Example:

```
<input bind:value={name}>
<button on:click={handleClick}>Click me</button>
<div class:highlighted={isHighlighted}></div>
<p style:color={textColor}></p>
```

Control Flow & Components

Conditional Rendering

Svelte uses `{#if}`, `{:else if}`, `{:else}`, and `{/if}` blocks for conditional rendering.

```
{#if user.loggedIn}
  <p>Welcome, {user.name}!</p>
{:else}
  <p>Please log in.</p>
{/if}
```

List Rendering

Use `{#each}` blocks to iterate over arrays and render lists. Include a `key` attribute for efficient updates.

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

Component Communication

Props

Pass data from parent to child components using props.

```
<!-- Parent.svelte -->
<Child name="{parentName}" />

<!-- Child.svelte -->
<script>
  export let name;
</script>
<h1>Hello {name}!</h1>
```

Events

Dispatch custom events from child to parent components.

```
<!-- Child.svelte -->
<script>
  import {
createEventDispatcher } from
'svelte';
  const dispatch =
createEventDispatcher();

  function handleClick() {
    dispatch('message', {
text: 'Hello from Child!' });
  }
</script>
<button on:click=
{handleClick}>Send
Message</button>

<!-- Parent.svelte -->
<Child on:message=
{handleMessage} />
```

Context API

Share data between components without explicitly passing props through every level. Use `setContext` and `getContext`.

```
// Parent.svelte
import { setContext } from
'svelte';
setContext('theme', { color:
'dark' });

// Child.svelte
import { getContext } from
'svelte';
const theme =
getContext('theme');
```

Lifecycle and Advanced Features

Lifecycle Methods

- `onMount(callback)`: Runs after the component is first rendered to the DOM.
- `onDestroy(callback)`: Runs when the component is unmounted.
- `beforeUpdate(callback)`: Runs immediately before the component updates.
- `afterUpdate(callback)`: Runs immediately after the component updates.
- `tick()`: Returns a promise that resolves once any pending state changes have been applied to the DOM.

Example:

```
import { onMount } from 'svelte';

onMount(() => {
  console.log('Component mounted');
  return () => {
    console.log('Component unmounted');
  };
});
```

Transitions

`transition:`	Apply transitions when elements enter or leave the DOM. `<div transition:fade>...`
`in:` and `out:`	Apply different transitions for entering and leaving. `<div in:fly={{ y: 200, duration: 2000 }} out:fade>...`
Built-in transitions	`fade`, `fly`, `slide`, `scale`, `draw`

Animations

`animate:`	Animate changes to numeric values. `<div animate:flip>...`
Custom animations	Use the `animate:` directive with a custom animation function for more control.

Stores and Actions

Stores

Stores are a way to manage state outside of components.

- **Writable:** Allows setting and updating values.
- **Readable:** Allows reading values but not directly setting them.
- **Derived:** A store that derives its value from other stores.

```
// store.js
import { writable } from 'svelte/store';

export const count = writable(0);


<!-- Component.svelte -->
<script>
  import { count } from './store.js';
  import { increment } from
'./actions.js';
</script>

<h1>{$count}</h1>
<button on:click={() =>
$count++}>Increment</button>
<button on:click={increment}>Increment
with Action</button>
```

Actions

Actions	Actions are functions that run when an element is created and can return an object with a `destroy` method that runs when the element is unmounted.
	`// actions.js`
	`export function increment() {`
	`  count.update(n => n + 1);`
	`}`

Using Actions	`<button use:increment>Increment with Action</button>`

Advanced Directives

- `use:`: Applies an action to an element.
- `transition:`: Applies a transition when an element enters or leaves the DOM.
- `animate:`: Animates changes to numeric values.