Components are the building blocks of Svelte applications. They are written into `.svelte` files, using a superset of HTML.

All three sections — script, styles and markup — are optional.

```
<script>
    // logic goes here
</script>

<!-- markup (zero or more items) goes here -->

<style>
    /* styles go here */
</style>
```

<script>

A `<script>` block contains JavaScript that runs when a component instance is created. Variables declared (or imported) at the top level are 'visible' from the component's markup. There are four additional rules:

1. `export` creates a component prop

Svelte uses the `export` keyword to mark a variable declaration as a *property* or *prop*, which means it becomes accessible to consumers of the component (see the section on attributes and props for more information).

```
<script>
    export let foo;

    // Values that are passed in as props
    // are immediately available
    console.log({ foo });
</script>
```

You can specify a default initial value for a prop. It will be used if the component's consumer doesn't specify the prop on the component (or if its initial value is `undefined`) when instantiating the component. Note that whenever a prop is removed by the consumer, its value is set to `undefined` rather than the initial value.

In development mode (see the compiler options), a warning will be printed if no default initial value is provided and the consumer does not specify a value. To squelch this warning, ensure that a default initial value is specified, even if it is `undefined`.

```
<script>
    export let bar = 'optional default initial value';
    export let baz = undefined;
</script>
```

If you export a `const`, `class` or `function`, it is readonly from outside the component. Function *expressions* are valid props, however.

Readonly props can be accessed as properties on the element, tied to the component using `bind:this` syntax.

```
<script>
    // these are readonly
    export const thisIs = 'readonly';

    export function greet(name) {
        alert(`hello ${name}!`);
    }

    // this is a prop
    export let format = n => n.toFixed(2);
</script>
```

You can use reserved words as prop names.

```
<script>
    let className;

    // creates a `class` property, even
    // though it is a reserved word
    export { className as class };
</script>
```

2. Assignments are 'reactive'

To change component state and trigger a re-render, just assign to a locally declared variable.

Update expressions (`count += 1`) and property assignments (`obj.x = y`) have the same effect.

```
<script>
    let count = 0;

    function handleClick () {
        // calling this function will trigger an
        // update if the markup references `count`
        count = count + 1;
    }
</script>
```

Because Svelte's reactivity is based on assignments, using array methods like `.push()` and `.splice()` won't automatically trigger updates. A subsequent assignment is required to trigger the update. This and more details can also be found in the tutorial.

```
<script>
    let arr = [0, 1];

    function handleClick () {
        // this method call does not trigger an update
```

```
        arr.push(2);
        // this assignment will trigger an update
        // if the markup references `arr`
        arr = arr
    }
</script>
```

3. `$:` marks a statement as reactive

Any top-level statement (i.e. not inside a block or a function) can be made reactive by prefixing it with the `$:` JS label syntax. Reactive statements run immediately before the component updates, whenever the values that they depend on have changed.

```
<script>
    export let title;

    // this will update `document.title` whenever
    // the `title` prop changes
    $: document.title = title;

    $: {
        console.log(`multiple statements can be combined`);
        console.log(`the current title is ${title}`);
    }
</script>
```

Only values which directly appear within the `$:` block will become dependencies of the reactive statement. For example, in the code below `total` will only update when `x` changes, but not `y`.

```
<script>
    let x = 0;
    let y = 0;

    function yPlusAValue(value) {
        return value + y;
    }

    $: total = yPlusAValue(x);
</script>

Total: {total}
<button on:click={() => x++}>
    Increment X
</button>

<button on:click={() => y++}>
    Increment Y
</button>
```

If a statement consists entirely of an assignment to an undeclared variable, Svelte will inject a `let` declaration on your behalf.

```
<script>
    export let num;

    // we don't need to declare `squared` and `cubed`
    // — Svelte does it for us
    $: squared = num * num;
    $: cubed = squared * num;
</script>
```

4. Prefix stores with `$` to access their values

A *store* is an object that allows reactive access to a value via a simple *store contract*. The `svelte/store` module contains minimal store implementations which fulfil this contract.

Any time you have a reference to a store, you can access its value inside a component by prefixing it with the `$` character. This causes Svelte to declare the prefixed variable, subscribe to the store at component initialization and unsubscribe when appropriate.

Assignments to `$`-prefixed variables require that the variable be a writable store, and will result in a call to the store's `.set` method.

Note that the store must be declared at the top level of the component — not inside an `if` block or a function, for example.

Local variables (that do not represent store values) must *not* have a `$` prefix.

```
<script>
    import { writable } from 'svelte/store';

    const count = writable(0);
    console.log($count); // logs 0

    count.set(1);
    console.log($count); // logs 1

    $count = 2;
    console.log($count); // logs 2
</script>
```

Store contract

```
store = { subscribe: (subscription: (value: any) => void) => (() => void), set?:
(value: any) => void }
```

You can create your own stores without relying on `svelte/store`, by implementing the *store contract*:

1. A store must contain a `.subscribe` method, which must accept as its argument a subscription function. This subscription function must be immediately and synchronously called with the store's current value

upon calling `.subscribe`. All of a store's active subscription functions must later be synchronously called whenever the store's value changes.
2. The `.subscribe` method must return an unsubscribe function. Calling an unsubscribe function must stop its subscription, and its corresponding subscription function must not be called again by the store.
3. A store may *optionally* contain a `.set` method, which must accept as its argument a new value for the store, and which synchronously calls all of the store's active subscription functions. Such a store is called a *writable store*.

For interoperability with RxJS Observables, the `.subscribe` method is also allowed to return an object with an `.unsubscribe` method, rather than return the unsubscription function directly. Note however that unless `.subscribe` synchronously calls the subscription (which is not required by the Observable spec), Svelte will see the value of the store as `undefined` until it does.

<script context="module">

A `<script>` tag with a `context="module"` attribute runs once when the module first evaluates, rather than for each component instance. Values declared in this block are accessible from a regular `<script>` (and the component markup) but not vice versa.

You can `export` bindings from this block, and they will become exports of the compiled module.

You cannot `export default`, since the default export is the component itself.

> *Variables defined in `module` scripts are not reactive — reassigning them will not trigger a rerender even though the variable itself will update. For values shared between multiple components, consider using a store.*

```
<script context="module">
    let totalComponents = 0;

    // this allows an importer to do e.g.
    // `import Example, { alertTotal } from './Example.svelte'`
    export function alertTotal() {
        alert(totalComponents);
    }
</script>

<script>
    totalComponents += 1;
    console.log(`total number of times this component has been created:
${totalComponents}`);
</script>
```

<style>

CSS inside a `<style>` block will be scoped to that component.

This works by adding a class to affected elements, which is based on a hash of the component styles (e.g. `svelte-123xyz`).

```
<style>
    p {
        /* this will only affect <p> elements in this component */
        color: burlywood;
    }
</style>
```

To apply styles to a selector globally, use the `:global(...)` modifier.

```
<style>
    :global(body) {
        /* this will apply to <body> */
        margin: 0;
    }

    div :global(strong) {
        /* this will apply to all <strong> elements, in any
             component, that are inside <div> elements belonging
             to this component */
        color: goldenrod;
    }

    p:global(.red) {
        /* this will apply to all <p> elements belonging to this
             component with a class of red, even if class="red" does
             not initially appear in the markup, and is instead
             added at runtime. This is useful when the class
             of the element is dynamically applied, for instance
             when updating the element's classList property directly. */
    }
</style>
```

If you want to make @keyframes that are accessible globally, you need to prepend your keyframe names with `-global-`.

The `-global-` part will be removed when compiled, and the keyframe then be referenced using just `my-animation-name` elsewhere in your code.

```
<style>
    @keyframes -global-my-animation-name {...}
</style>
```

There should only be 1 top-level `<style>` tag per component.

However, it is possible to have `<style>` tag nested inside other elements or logic blocks.

In that case, the `<style>` tag will be inserted as-is into the DOM, no scoping or processing will be done on the `<style>` tag.

```
<div>
    <style>
        /* this style tag will be inserted as-is */
        div {
            /* this will apply to all `<div>` elements in the DOM */
            color: red;
        }
    </style>
</div>
```