SVELTE • RUNES

$props

ON THIS PAGE

The inputs to a component are referred to as *props*, which is short for *properties*. You pass props to components just like you pass attributes to elements:

App.svelte

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

<MyComponent adjective="cool" />
```

On the other side, inside `MyComponent.svelte`, we can receive props with the `$props` rune...

MyComponent.svelte

```
<script>
  let props = $props();
</script>

<p>this component is {props.adjective}</p>
```

...though more commonly, you'll *destructure* your props:

MyComponent.svelte

```
<script>
  let { adjective } = $props();
</script>
```

Destructuring allows us to declare fallback values, which are used if the parent component does not set a given prop:

```
let { adjective = 'happy' } = $props();
```

> Fallback values are not turned into reactive state proxies (see Updating props for more info)

Renaming props

We can also use the destructuring assignment to rename props, which is necessary if they're invalid identifiers, or a JavaScript keyword like `super`:

```
let { super: trouper = 'lights are gonna find me' } = $props();
```

Rest props

Finally, we can use a *rest property* to get, well, the rest of the props:

```
let { a, b, c, ...others } = $props();
```

Updating props

References to a prop inside a component update when the prop itself updates — when `count` changes in `App.svelte`, it will also change inside `Child.svelte`. But the child component is able to temporarily override the prop value, which can be useful for unsaved ephemeral state (demo):

App.svelte

Docs

```
import Child from './Child.svelte';

  let count = $state(0);
</script>

<button onclick={() => (count += 1)}>
  clicks (parent): {count}
</button>

<Child {count} />
```

Child.svelte

```
<script>
  let { count } = $props();
</script>

<button onclick={() => (count += 1)}>
  clicks (child): {count}
</button>
```

While you can temporarily *reassign* props, you should not *mutate* props unless they are bindable.

If the prop is a regular object, the mutation will have no effect (demo):

App.svelte

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

<Child object={{ count: 0 }} />
```

Child.svelte

```
<script>
  let { object } = $props();
</script>
```

```
  clicks: {object.count}
</button>
```

If the prop is a reactive state proxy, however, then mutations *will* have an effect but you will see an `ownership_invalid_mutation` warning, because the component is mutating state that does not 'belong' to it (demo):

App.svelte

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

  let object = $state({count: 0});
</script>

<Child {object} />
```

Child.svelte

```
<script>
  let { object } = $props();
</script>

<button onclick={() => {
  // will cause the count below to update,
  // but with a warning. Don't mutate
  // objects you don't own!
  object.count += 1
}}>
  clicks: {object.count}
</button>
```

The fallback value of a prop not declared with `$bindable` is left untouched — it is not turned into a reactive state proxy — meaning mutations will not cause updates (demo)

Child.svelte

```
<script>
```

```
  // has no effect if the fallback value is used
  object.count += 1
}}>
  clicks: {object.count}
</button>
```

In summary: don't mutate props. Either use callback props to communicate changes, or — if parent and child should share the same object — use the `$bindable` rune.

Type safety

You can add type safety to your components by annotating your props, as you would with any other variable declaration. In TypeScript that might look like this...

```
<script lang="ts">
  let { adjective }: { adjective: string } = $props();
</script>
```

...while in JSDoc you can do this:

```
<script>
  /** @type {{ adjective: string }} */
  let { adjective } = $props();
</script>
```

You can, of course, separate the type declaration from the annotation:

```
<script lang="ts">
  interface Props {
    adjective: string;
  }

  let { adjective }: Props = $props();
</script>
```

discover which props they should provide.

Edit this page on GitHub

PREVIOUS
$effect

NEXT
$bindable