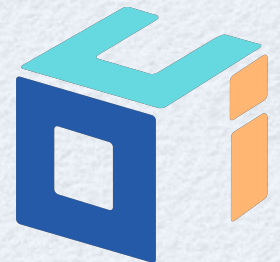




# Svelte

slides at <https://github.com/mvolkmann/talks>

**R. Mark Volkmann**  
Object Computing, Inc.  
<http://objectcomputing.com>  
Email: [mark@objectcompuing.com](mailto:mark@objectcompuing.com)  
Twitter: @mark\_volkman  
GitHub: mvolkmann



OCI | TRAINING



# Why Consider Svelte?

- Better performance
  - reactivity without using a virtual Dom
- Smaller bundle sizes
  - much smaller framework library needed at runtime
  - example Todo app is 13% size of same app in React
    - see <https://github.com/mvolkmann/svelte-todo> and <https://github.com/mvolkmann/react-todo>
- Easier state management
  - component variables, context, stores, and module context
- But ...
  - no TypeScript support yet, but it is coming
    - <https://github.com/sveltejs/svelte/issues/1639>
  - “Svelte Native” perhaps not yet as mature as React Native
    - builds on NativeScript for creating mobile apps



# Svelte Overview ...

- Alternative to web frameworks like React, Vue, and Angular
- A web application **compiler**, not a runtime library
  - compiles `.svelte` files to `.js` files
  - no Svelte runtime **dependencies**, only **devDependencies**
- Developed by **Rich Harris**
  - formerly at "The Guardian"; current at "The New York Times"
  - previously created **Ractive** web framework - <https://ractive.js.org/>
    - used at "The Guardian"
    - inspired parts of Vue
  - created **Rollup** module bundler - <https://rollupjs.org/>
    - alternative to Webpack and Parcel
- Can build entire app or  
add components to existing apps implemented with other frameworks



# ... Svelte Overview

- Supports reactivity, not a virtual DOM
  - top-level component variables are watched for changes
- Faster than apps built with most other frameworks
  - mostly because it doesn't need to build a virtual DOM and perform DOM diffing
  - more suitable for running on low-powered devices
- Delivered code is much smaller
  - uses Rollup by default for module bundling, but can also use Webpack or Parcel
  - create production build with `npm run build`
  - <https://www.freecodecamp.org/news/a-realworld-comparison-of-front-end-frameworks-with-benchmarks-2019-update-4be0d3c78075/>
    - gzipped app size in KBs: **Angular+ngrx**: 134, **React+Redux**: 193, **Vue**: 41.8, **Svelte**: 9.7
- Provides live reload
- Provides runtime warnings for accessibility issues
  - ex. missing an `alt` attribute on an `img` element; using `autofocus`



# Does It Disappear?

- Some say Svelte disappears once an app is built
- The Svelte library is mostly defined by `.js` files in `node_modules/svelte`
  - main functions are defined in `internal.js` which is currently ~1400 line of code
  - other library files used for specific features
    - `easing.js`, `motion.js`, `register.js`, `store.js`, `transition.js`
- `npm run build` produces files in `public` directory
  - including `bundle.js`
- Svelte library functions that are used by the app are copied to the top of `bundle.js`
  - in the case of the Todo app shown later, this is ~500 lines of code
- So Svelte doesn't disappear, it is just very small



# Svelte Resources

- **"Rethinking reactivity"** - <https://www.youtube.com/watch?v=AdNJ3fydeao>
  - **talk by Rich Harris** at "You Gotta Love Frontend (YGLF) Code Camp 2019"
- Home page - <https://svelte.dev>
- **Tutorial** - <https://svelte.dev/tutorial>
- **API** - <https://svelte.dev/docs>
- **Examples** - <https://svelte.dev/examples>
- **Online REPL** - <https://svelte.dev/repl>
  - great for trying small amounts of Svelte code
  - can save for sharing and submitting issues
- Blog - <https://svelte.dev/blog>
- **Discord chat room** - <https://discordapp.com/invite/yy75DKs>
- GitHub - <https://github.com/sveltejs/svelte>



# Getting Started

- Install Node.js from <https://nodejs.org>
  - installs `node`, `npm`, and `npm` commands
- `npm degit sveltejs/template app-name`
  - `degit` is useful for project scaffolding
- `cd app-name`
- `npm install`
- `npm run dev`
  - provides live reload, unlike `npm run start`
  - syntax errors are reported in window where this is running, not in browser because it doesn't produce a new version of the app if there are errors
- Browse `localhost:5000`
- Just outputs "Hello world!" in purple

`degit` was created by Rich Harris. It downloads a git repo, by default the master branch. In this case `sveltejs` is the user name and `template` is the repo. The second argument is the name of the directory to create.

# Generated `index.html`

- In `public/index.html`

```
<!doctype html>
<html>
<head>
  <meta charset='utf8'>
  <meta name='viewport' content='width=device-width'>

  <title>Svelte app</title>

  <link rel='icon' type='image/png' href='favicon.png'>
  <link rel='stylesheet' href='global.css'>
  <link rel='stylesheet' href='bundle.css'>
</head>

<body>
  <script src='bundle.js'></script>
</body>
</html>
```

the Svelte logo

non-scoped CSS

scoped CSS from `.svelte` files

bundled JavaScript code  
from `.js` and `.svelte` files



# Generated Source Files

uses tabs for indentation by default

- In `src/main.js`

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

const app = new App({
  target: document.body,
  props: {
    name: 'world'
  }
});

export default app;
```

where to render

data passed to  
top component

- In `src/App.svelte`

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

<style>
  h1 {
    color: purple;
  }
</style>

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

exported variables  
can be set as props

Curly braces (interpolation)  
are used to output the value  
of a JavaScript expression.  
They are also used for  
dynamic attribute values.



# Defining Components

- **Angular** uses classes
- **React** uses functions or classes
- **Vue** uses object literals
- **Svelte** doesn't use any container
- JavaScript, CSS, and HTML in source files  
is combined to form the component definition  
which is automatically becomes the default export



# Svelte Components

- Implemented in `.svelte` files under `src` directory
- Three sections, all optional

```
<script>  
  // JavaScript goes here.  
</script>  
  
<style>  
  /* Scoped CSS rules go here. */  
</style>  
  
<!-- HTML goes here. -->
```

Note the different styles of comments that can be used in each section.



# Component Names

- Svelte component definitions do not specify a component name
  - not provided inside source file by a class name, function name, or property value like in other frameworks
- Name is associated when imported and must start uppercase
- Lowercase names are reserved
  - for non-custom elements like HTML and SVG
- Example

```
// Somewhat confusing
import AnyNameIWant from './some-name.svelte';

// Less confusing
import SameName from './SameName.svelte';
```

# Sharing Data

- Four ways to share data between Svelte components
- **Props**
  - pass data from parent components to child components
- **Contexts**
  - ancestor components make data available to descendant components
- **Stores**
  - store data outside any component and make available to all
- **Module Scope**
  - store data in component modules and make available to all instances of the component

These builtin approaches are so good that there is no need for state management libraries!



# Props ...

- Components can accept input through props
- Specified as attributes on a component element in parent components
- Declared in `<script>` section of component with **export** keyword
  - using valid JavaScript syntax in a Svelte-specific way
- Example
  - in parent component

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

<Hello name="Mark" />
```

prop values that are expressions or non-string literals are surrounded by curly braces instead of quotes

- in child component defined in `Hello.svelte`

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

<div>
  Hello, {name}!
</div>
```

must use `let`, not `const`;  
can assign a default value

# ... Props

- Currently there is no prop type checking like in React, Vue, and Angular



# Attributes

- Attribute values can be supplied from JavaScript expressions

```
<tag attr={expression}>
```

- Can embed expressions in string values

- example 

```
<tag attr="foo{v1}bar{v2}baz">
```

- Shorthand syntax if value is in variable with same name as attribute

- ```
<tag foo={foo}>
```

 is same as 

```
<tag {foo}>
```

- Can use spread operator

- if attributes are in an object where keys are attribute names and values are their values

- example

```
<script>
  let score = 0;
  const inputAttrs = {
    type: 'number',
    max: 10,
    min: 0,
    value: score
  };
</script>
```

```
<input {...inputAttrs} bind:value={score} />
```

bind simulates two-way data binding;  
covered starting on slide 28

# Styling

- Styles in `<style>` tags of `.svelte` files are automatically scoped to component
  - adds same CSS class named `svelte-hash` to all rendered tags
- Global styles go in `public/global.css`
- Can use `/* */` comments, but not `//`
- “svelte3” ESLint plugin warns about unused CSS selectors
- Can conditionally add a CSS class to an element
  - example `<div class:error={status > 0}>{result}</div>`

CSS class `error` is only added if `status` is greater than zero



# Importing Components

- Components can import others inside their `script` tag
  - example `import Other from './Other.svelte';`
- Imported components can then be used in HTML section

# Inserting HTML



- To render a JavaScript expression whose value is an HTML string

```
{@html expression}
```

- Example

- suppose `markup` is a variable that holds a string of HTML

```
<p>{@html markup}</p>
```

- Cross-site Scripting

- you must escape HTML from untrusted sources to avoid this



# Reactivity

- Changes to top-level variables referenced in interpolations automatically cause those interpolations to be reevaluated
- Example

```
<script>
  let count = 0;
  const increment = () => count++;
</script>

<div>count = {count}</div>
<button on:click={increment}>+</button>
```

- Must assign a new value to trigger
  - pushing new elements onto an array doesn't do this

```
myArr = myArr.concat(newValue);
```

works

```
// Alternative trick
myArr.push(newValue);
myArr = myArr;
```

works

# Reactive Declarations

- `$:` is a “labelled statement” with label name “`$`” that Svelte treats as a “reactive declaration”

interestingly it is not an error in JavaScript to use same label more than once in same scope

- Add as a prefix on top-level statements that should be repeated whenever any referenced variables change

- Examples

```
$: average = total / count;
$: console.log('count =', count);
```

like “computed properties” in Vue

great for debugging

when applied to an assignment to an undeclared variable the `let` keyword is not required

- Can apply to a block

```
$: {
  // statements to be repeated go here
}
```

- Can apply to multiline statements like `if` statements

```
$: if (someCondition) {
  // body statements
}
```

executes if any variables referenced in condition or body change, but of course the body only executes when condition is true

for example, if condition includes calls to functions, they will be called if any references in the body have changed



# Logic in Markup

- Three approaches for conditional and iteration logic
- **React**
  - uses JSX where logic is implemented by JavaScript code in curly braces
- **Angular** and **Vue**
  - support framework-specific attributes for logic
  - ex. `ngIf`, `ngFor`, `v-if`, `v-for`, ...
- **Svelte**
  - supports mustache-like custom syntax that wraps elements
  - ex. `{#if}` and `{#each}`

# Markup `if` Statement

- Begin with `{#if condition}`
  - starting with `#` indicates a block opening tag
- Can use `{:else if condition}` and `{:else}`
  - starting with `:` indicates a block continuation tag
- End with `{/if}`
  - starting with `/` indicates a block ending tag
- Include markup to be conditionally rendered
- Example

```
{#if color === 'yellow'}  
  <div>Nice color!</div>  
{:else if color === 'orange'}  
  <div>That's okay too.</div>  
{:else}  
  <div>Questionable choice.</div>  
{/if}
```



# Markup each Statement ...

- Begin with `{#each iterable as element}`
  - include markup to be rendered for each element
- Optional `{:else}`
  - renders when iterable is empty
- End with `{/each}`

- Examples

```
{#each colors as color}  
  <div style='color: {color}'>{color}</div>  
{/each}
```

red  
green  
blue

```
{#each colors as color, index}  
  <div>{index + 1}) {color}</div>  
{/each}
```

1) red  
2) green  
3) blue

```
{#each people as {name, age}}  
  <div>{name} is {age} years old.</div>  
{:else}  
  <div>There are no people.</div>  
{/each}
```

can use destructuring  
when elements are objects

# ... Markup each Statement

- Need a unique identifier for each element if items will be added, removed, or modified
  - similar to a key in React and Vue
- Example

```
{#each people as person (person.id)}  
  <div>{person.name} is {person.age} years old.</div>  
{/each}
```



# Promises in Markup



- Can wait for promises to resolve or reject in markup and render different markup for each
- Examples
  - assuming `getData` function returns a `Promise`

```
{#await getData()}  
  <div>Waiting for data ...</div>  
{:then result}  
  <div>result = {result}</div>  
{:catch error}  
  <div class="error">Error: {error.message}</div>  
{/await}
```

can use any name for variables  
after `:then` and `:catch`

```
{#await getData() then result}  
  <div>result = {result}</div>  
{:catch error}  
  <div class="error">Error: {error.message}</div>  
{/await}
```

can omit block to render while  
waiting for resolve or reject

can omit `catch` block

# Slots

- Allow child elements to be passed to a component
- Component can decide whether and where to render them
- `<slot />` is a “default slot” that marks the spot where all children are rendered
- `<slot>default content</slot>` is the same, but also specifies content to render if there is no child content
  - note that whitespace counts as child content
- `<slot name="slotName">default content</slot>` is a named slot
  - can have any number of these
  - child elements can specify a target slot with `slot` attribute
    - example 

```
<div slot="address">
  123 Some Street<br>
  Some City, Some State 12345
</div>
```



# Binding Form Elements ...

- Form elements can be bound to a variable
  - simulates two-way data binding
  - provides current value and event handling for updating the variable when user changes value
- For `input` elements with type `number` or `range`, automatically coerces values from strings to numbers
- Example

Name

Happy? ☒

Favorite Flavors ☒ vanilla ☐ chocolate ☒ strawberry

Favorite Season ☒ Spring ☐ Summer ☐ Fall ☐ Winter

Favorite Color

Life Story

Mark likes yellow, Spring, and is happy.

Mark's favorite flavors are vanilla,strawberry.

Story: Once upon a time ...

# ... Binding Form Elements ...

```
<script>
  const colors = ['red', 'orange', 'yellow', 'green', 'blue', 'purple'];
  const flavors = ['vanilla', 'chocolate', 'strawberry'];
  const seasons = ['Spring', 'Summer', 'Fall', 'Winter']
  let favoriteColor = '';
  let favoriteFlavors = [];
  let favoriteSeason = '';
  let happy = true;
  let name = '';
  let story = '';
</script>
```

```
<style>
  div {
    margin-bottom: 10px;
  }

  input,
  select {
    border: solid gray 1px;
    border-radius: 4px;
    padding: 4px;
  }

  input[type='checkbox'],
  input[type='radio'] {
    margin-left: 5px;
  }

  label {
    display: inline-block;
    font-weight: bold;
    margin-right: 5px;
    vertical-align: top;
  }
</style>
```



# ... Binding Form Elements ...

```
<div class="form">
  <div>
    <label>Name</label>
    <input type="text" bind:value={name} />
  </div>
  <div>
    <label>Happy?</label>
    <input type="checkbox" bind:checked={happy} />
  </div>
  <div>
    <label>Favorite Flavors</label>
    {#each flavors as flavor}
      <label>
        <input type="checkbox" value={flavor} bind:group={favoriteFlavors} />
        {flavor}
      </label>
    {/each}
  </div>
```

for checkboxes, bind to **checked** property rather than **value**

using **bind:group** with a set of related checkboxes makes the value an array of strings

# ... Binding Form Elements ...

```
<div>
  <label>Favorite Season</label>
  {#each seasons as season}
    <label>
      <input type="radio" value={season} bind:group={favoriteSeason} />
      {season}
    </label>
  {/each}
</div>
<div>
  <label>Favorite Color</label>
  <select bind:value={favoriteColor}>
    <option />
    {#each colors as color}
      <option>{color}</option>
    {/each}
  </select>
</div>
<div>
  <label>Life Story</label>
  <textarea bind:value={story} />
</div>
```

using `bind:group` with a set of related radio buttons makes the value a single string

add `multiple` attribute to a `select` to change it to a scrollable list that allows selecting multiple options

option elements can have a `value` attribute and its value can be a string, number, or object



# ... Binding Form Elements

```
{#if name}  
  <div>  
    {name} likes {favoriteColor}, {favoriteSeason},  
    and is {happy ? 'happy' : 'unhappy'}.  
  </div>  
  <div>{name}'s favorite flavors are {favoriteFlavors}.</div>  
  <div>Story: {story}</div>  
{/if}  
</div>
```

This part just reports variable values set by binds, but only if `name` has a value.

- In addition to binding to primitive variables, form elements can also bind to object properties
  - user input causes those objects to be mutated

# Binding Custom Props

- Can bind a child component prop to a variable in parent component
- Allows child component to change value of a parent component variable
- Example

```
Parent.svelte
<script>
  import Child from './Child.svelte';
  let pValue = 1;
</script>

<div>pValue = {pValue}</div>
<Child bind:cValue={pValue} />
```

```
Child.svelte
<script>
  export let cValue = '';
  const double = () => cValue *= 2;
</script>

<div>cValue = {cValue}</div>
<button on:click={double}>Double</button>
```

- when `child` button is pressed, `cValue` is doubled and that becomes the new value of `pValue` because it is bound to `cValue`



# Event Handling

- Specify with `on:event-name` attribute set to a function
- Examples

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

reference to function define in `script` section

```
<button on:click={event => clicked = event.target}>Press Me</button>
```

inline event handling

- Event object is passed to event handling function
- Can register multiple event listeners for same event

- example 

```
<button on:click={doOneThing} on:click={doAnother}>Press Me</button>
```

# Event Modifiers

- Can specify any number of event modifiers
  - with vertical bars preceding modifier names
  - ex. `<button on:click|once|preventDefault={handleClick}>Press Me</button>`
- Modifiers are
  - **capture** - invokes handler only in capture phase instead of default bubbling phase
  - **once** - removes handler after first occurrence
  - **passive** - can improve scrolling performance
  - **preventDefault** - prevents default action from occurring
  - **stopPropagation** - prevents subsequent handlers in capture/bubbling flow from being invoked



# Event Forwarding

- Omitting event handling function is a shorthand to forward events up to parent
- Example
  - if component hierarchy is **A** > **B** > **C** and **C** emits event "foo", **B** can forward it up to **A** with `<C on:foo />`
  - just omits value of "on" attribute which otherwise is an event handling function
  - also works with DOM events

# Dispatching Events

- Components can dispatch events

```
<script>
  import {createEventDispatcher} from 'svelte';
  const dispatch = createEventDispatcher();
  function sendEvent() {
    dispatch('event-name', optionalData);
  }
</script>
```

must be called when component is instantiated

data can be a primitive or an object

event names with dashes  
may not work

- These events only go to parent component

- they do not automatically bubble further up
- parent listens for events on child instance

```
<Child on:event-name={handleEvent} />
```

`handleEvent` is a parent component function



# Lifecycle Functions

- **onMount**
  - registers a function to be called when component is added to DOM
- **beforeUpdate**
  - registers a function to be called immediately before component DOM updates
  - first run is before component is mounted, and no component DOM will be present yet
- **afterUpdate**
  - registers a function to be called immediately after component DOM updates
- **onDestroy**
  - registers a function to be called when component is removed from DOM
- Import these from svelte package
  - example

```
<script>  
  import {afterUpdate, onMount} from 'svelte';  
  ...  
</script>
```

# onMount

- Most commonly used lifecycle function
- Some uses
  - move focus into a given form element
  - retrieve data from a REST service (recommended place to do this)
- Example - moves focus to an input when component is mounted

```
<script>
  import {onMount} from 'svelte';
  let name = '';
  let nameInput;
  onMount(() => nameInput.focus());
</script>

<input bind:this={nameInput} bind:value={name} />
```



# onDestroy



- Two options to run a function when component is destroyed
  - register function with `onDestroy`
  - return function from `onMount`
    - similar to React `useEffect` hook, but doesn't run function on updates

# Sharing Lifecycle Functions



- Lifecycle functions be called from helper functions
  - similar to defining custom React hooks
  - multiple components import and call helper functions
- Recommended to name these functions starting with `"on"`
  - similar to how React hook names start with `"use"`



# Actions ...

- Specified on elements with attribute `use: fnName={args}`
  - when element is added to DOM the function **fnName** is called, passing it the DOM element and arguments, if any
  - omit `= {args}` if no arguments other than the element are needed
- Somewhat related to **onMount**
  - **onMount** registers a function to call when each instance of a component added to the DOM
  - actions are called when specific elements are added to DOM

- Example

```
<script>
  let name = '';
  const focus = element => element.focus();
</script>

<input bind:value={name} use:focus />
```

# ... Actions



- Action functions can optionally return an object with **update** and **destroy** properties that are functions
  - **rarely used**
  - **update** is called every time an argument value changes
    - of course this doesn't apply if there are no arguments
  - **destroy** is called when element is removed from DOM



# Context ...

- Alternative to props and stores for making data available in a descendant component

```
import {getContext, setContext} from 'svelte';
```

- Ancestor components set context associated with the component

```
setContext(key, value);
```

- must be called during component initialization
- Descendant components get context from closest ancestor that has context with given key

```
const value = getContext(key);
```

- must be called during component initialization
- Keys can be any kind of value, not just strings
- Values can be any kind of value including objects with methods

# ... Context

- Descendant components don't receive updates
  - only see what is available during component initialization
  - not reactive!



# Context Example

A.svelte

```
<script>
  import {setContext} from 'svelte';
  import B from './B.svelte';
  setContext('favorites', {color: 'yellow', number: 19});
</script>

<div>
  This is in A.
  <B />
</div>
```

## Output

```
This is in A.
This is in B.
This is in C.
favorite color is yellow
favorite number is 19
```

B.svelte

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

<div>
  This is in B.
  <C />
</div>
```

C.svelte

```
<script>
  import {getContext} from 'svelte';
  const {color, number} = getContext('favorites');
</script>

<div>
  This is in C.
  <div>favorite color is {color}</div>
  <div>favorite number is {number}</div>
</div>
```

# Stores

- Hold application state outside any component
- Alternative to using props or context to make data available in components
- Where to define?
  - for stores that should be available to any component, define and export them in a file like `src/stores.js` and import them from that file wherever needed
  - for stores that should only be available to descendants of component `x`, define them in `x` and pass them to descendants using props or context



# Kinds of Stores

- Three kinds of provided stores
  - **writable** stores - only kind that can be modified by components
  - **readable** stores - handle computing their data
  - **derived** stores - derive data from current values of other stores not described further here
  - all have **subscribe** method which returns function to call to unsubscribe
- Can implement custom stores
  - any object with properly implemented **subscribe** method
  - see example at <https://svelte.dev/tutorial/custom-stores>

# Writable Stores

- Call **writable** function to create
  - import from `svelte/store` package
- Pass initial value and optionally a function that takes **set** function
- Writable stores have these methods

- **set**(*newValue*)
- **update**(*currentValue* => *newValue*) calculates new value from current value

- Examples

```
stores.js
import {writable} from 'svelte/store';
export const dogStore = writable([]);
```

initial value

- Can bind to a writable store

- example `<input bind:value={$someStore}>`
- user changes to `input` also update the store

\$ syntax is explained ahead

```
export const fancyStore = writable(
  initialValue,
  set => {
    // Called when subscribe count goes from 0 to 1.
    // Compute initial value and pass to set function.
    return () => {
      // Called when subscriber count goes to 0.
    }
  }
);
```

using optional  
second argument



# Readable Stores

- Call **readable** function to create
  - import from `svelte/store` package
- Pass initial value AND a function that takes **set** function
  - function is called when first subscriber subscribes
  - determine value and pass to **set**
- Optionally return a function
  - called when last subscriber unsubscribes to perform cleanup
- Example

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

export const catStore = readable(
  [], // initial value
  set => {
    const res = await fetch('/cats');
    const cats = await res.json();
    set(cats);
    // Can return cleanup function here.
  }
);
```

can use **setInterval** to continuously change value

# Using Stores

- Import from where defined
  - typically `stores.js`
- Two options to access
  - **subscribe** method - very verbose!
  - `$` auto-subscription shorthand - much better!
    - requires store to be imported at top-level of component
    - all variables whose names begin with `$` must be stores
    - automatically subscribes when first used and unsubscribes when removed from DOM

```
<script>
  import {onDestroy} from 'svelte';
  import {dogStore} from './stores.js';
  let dogs;
  const unsubscribe = dogStore.subscribe(value => dogs = value);
  onDestroy(unsubscribe);
</script>
```

uses **subscribe** method

```
<!-- Use dogs in HTML. -->
```

```
<script>
  import {dogStore} from './stores.js';
</script>
```

uses auto-subscription

```
<!-- Use $dogStore in HTML. -->
```



# Auto-subscription Updates

- Two ways get updates
  - can reference `$storeName` in HTML

```
<script>
  import {dogStore} from './stores.js';
</script>

<div>
  {$dogStore.name} is a
  {$dogStore.breed} that is
  {$dogStore.age} years old.
</div>
```

- can extract data in `<script>`

```
<script>
  import {dogStore} from './stores.js';
  $: ({age, breed, name} = $dogStore);
</script>
<div>{name} is a {breed} that is {age} years old.</div>
```

parens are needed so it knows open curly brace isn't start of a block!

# Module Context

- To run JavaScript code in a component source file only once instead of once for each component instance created, include code in

```
<script context="module">
  ...
</script>
```

- When `script` tag doesn't specify `context`, it is "instance context"
- Can specify both kinds of `script` tags in a component source file
  - can export values from both contexts
  - cannot specify a default export because the component itself is automatically treated as default export
- Can declare variables and define functions in module context
  - accessible in instance context of all component instances
  - allows sharing data between all instances
  - instance context variables and functions are **not** accessible in module context

It's not important to move functions that don't access component state to module context because "Svelte **will hoist** any functions that don't depend on local state out of the component definition."



# Batched DOM Updates

- “When you invalidate component state in Svelte, by changing component variables it doesn't update the DOM immediately. Instead, it waits until the next microtask to see if there are any other changes that need to be applied, including in other components. Doing so avoids unnecessary work and allows the browser to batch things more effectively.”
- The **tick** function “returns a promise that resolves as soon as any pending state changes have been applied to the DOM (or immediately, if there are no pending state changes).”
- To make state changes after DOM updates have been applied

```
<script>
  import {tick} from 'svelte';
  ...
  // Make some state changes.
  await tick();
  // Make more state changes after DOM updates.
  ...
</script>
```

prevents batching of updates  
that occur after call to **tick**

also useful in tests to wait  
for a change to be processed  
before testing for the effect

# Animation ...

- **svelte/animate** provides
  - **flip**
- **svelte/motion** provides
  - **spring**
  - **tweened**
- **svelte/transition** provides
  - **crossfade**
  - **draw** - for SVG elements
  - **fade**
  - **fly** - set **x** and/or **y**
  - **scale**
  - **slide**
- Also see **svelte/easing**

basic example

```
<script>
  import {fade} from 'svelte/transition';
  ...
</script>

<li transition:fade>
  <!-- some content -->
</li>
```

when mounted this fades in;  
when unmounted this fades out



# ... Animation

- Can implement custom transitions
  - see example at <https://svelte.dev/tutorial/custom-css-transitions>
- Can listen for events to know when a transition starts and ends
  - `on:introstart`
  - `on:introend`
  - `on:outrostart`
  - `on:outbound`
- Also see “local transitions” and “delayed transitions”

# Special Elements ...

- `<svelte:component this={expression} optionalProps>`
  - expression value must be a component to render
  - renders nothing if *expression* is falsy
  - optional props are passed to component that is rendered
- `<svelte:self props>`
  - allows a component to render an instance of itself
  - supports recursive components
  - needed because a component cannot import itself



# ... Special Elements ...



- `<svelte:window on:eventName={handler}>`
  - listens for events on `window` object
- `<svelte:window bind:propertyName={variable}>`
  - binds a variable to a `window` property (ex. `innerWidth`)
- `<svelte:body on:eventName={handler}>`
  - listens for events on `body` object (ex. `mouseenter` and `mouseleave`)
- `<svelte:head>elements</svelte:head>`
  - inserts elements in `head` of document (ex. `link` and `script` tags)
  - When is it useful for a component to do this? Is it discouraged?



# ... Special Elements



- `<svelte:options option={value} />`
  - placed at top of file, not inside `script` tag
  - specifies compiler options including:
    - `immutable` means props will be treated as immutable (an optimization)
      - default is false
      - means parent components will create new objects for object props rather than modify properties of existing object
      - allows Svelte to determine whether a prop has changed by comparing object references to rather than object properties
      - if parent component modifies object properties, the child component will not detect the change and will not re-render
  - `accessors` adds getter and setter methods for the component props
    - default is false
  - `namespace="value"` specifies namespace of component
    - useful for SVG components with a namespace of "svg"; other common uses?
  - `tag="value"` specifies name to use when compiled as a custom element
    - allows Svelte components to be used in non-Svelte apps as web components

"If a component is compiled with `accessors: true`, each instance will have getters and setters corresponding to each of the component's props. Setting a value will cause a synchronous update."



# Debugging

- To break when given variables change and output their values in devtools console

```
{@debug var1, var2, var3}
```

place at top of HTML section, not inside `script` tag

- Variables can refer to any kind of value including objects and arrays
- To break when any state changes

```
{@debug}
```

# ESLint Setup

- Create `.eslintrc.json` file
- `npm install -D name` where *name* is
  - `eslint`
  - `eslint-plugin-html`
  - `eslint-plugin-import`
  - `eslint-plugin-svelte3`

```
{
  "env": {
    "browser": true,
    "es6": true
  },
  "extends": [
    "eslint:recommended",
    "plugin:import/recommended"
  ],
  "parserOptions": {
    "ecmaVersion": 2019,
    "sourceType": "module"
  },
  "plugins": ["svelte3"],
  "rules": {
    "no-console": "off",
    "svelte3/lint-template": true
  }
}
```

- Add npm script

```
"lint": "eslint --fix --quiet src --ext .js,.svelte",
```

- Run with `npm run lint`
- For more info see <https://github.com/sveltejs/eslint-plugin-svelte3>



# Prettier Setup

- Create `.prettierrc` file

```
{  
  "bracketSpacing": false,  
  "singleQuote": true  
}
```

- `npm install -D name` where *name* is

- `prettier`
- `prettier-plugin-svelte`

enforces section order of  
`<script>`, `<style>`, and HTML

- Add npm script

```
"format": "prettier--write '{public,src}/**/*.{css,html,js,svelte} '",
```

- Run with `npm run lint`

# Todo App ...

**To Do List**

1 of 2 remaining Archive Completed

Add

☒ learn Svelte Delete

☐ build a Svelte app Delete

```
src/main.js
import ToDoList from './ToDoList.svelte';
const app = new ToDoList({target: document.body});
export default app;
```



# ... Todo App ..

```
<script>
  import {createEventDispatcher} from 'svelte';
  const dispatch = createEventDispatcher();
  export let todo; // the only prop
</script>

<style>
  .done-true {
    color: gray;
    text-decoration: line-through;
  }
</style>

<li>
  <input
    type="checkbox"
    checked={todo.done}
    on:change={() => dispatch('toggleDone')}
  />
  <span class={'done-' + todo.done}>{todo.text}</span>
  <button on:click={() => dispatch('delete')}>Delete</button>
</li>
```

src/ToDo.svelte

# ... Todo App ...

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

  let lastId = 0;
  const createTodo = (text, done = false) => ({id: ++lastId, text, done});

  let todoText = '';
  let todos = [
    createTodo('learn Svelte', true),
    createTodo('build a Svelte app')
  ];

  let uncompletedCount = 0;
  $: uncompletedCount = todos.filter(t => !t.done).length;
  $: status = `${uncompletedCount} of ${todos.length} remaining`;

  function addTodo() {
    todos = todos.concat(createTodo(todoText));
    todoText = '';
  }

  const archiveCompleted = () => todos = todos.filter(t => !t.done);

  const deleteTodo = todoId => todos = todos.filter(t => t.id !== todoId);

  function toggleDone(todo) {
    const {id} = todo;
    todos = todos.map(t => t.id === id ? {...t, done: !t.done} : t);
  }
</script>
```

src/ToDoList.svelte



# ... Todo App ...

```
<style> src/TodoList.svelte
  body {
    font-family: sans-serif;
    padding-left: 10px;
  }

  button {
    margin-left: 10px;
  }

  li {
    margin-top: 5px;
  }

  ul.unstyled {
    list-style: none;
    margin-left: 0;
    padding-left: 0;
  }
</style>
```

# ... Todo App

```
<div>
  <h2>To Do List</h2>
  <div>
    {status}
    <button on:click={archiveCompleted}>Archive Completed</button>
  </div>
  <br />
  <form>
    <input
      type="text"
      size="30"
      autofocus
      placeholder="enter new todo here"
      bind:value={todoText}
    />
    <button disabled={!todoText} on:click={addTodo}>
      Add
    </button>
  </form>
  <ul class="unstyled">
    {#each todos as todo}
      <Todo
        todo={todo}
        on:delete={() => deleteTodo(todo.id)}
        on:toggleDone={() => toggleDone(todo)}
      />
    {/each}
  </ul>
</div>
```

src/ToDoList.svelte



# Unit Tests...

- Using Jest and Svelte Testing Library
- `npm install -D` the following
  - `@babel/core`
  - `@babel/preset-env`
  - `@testing-library/svelte`
  - `babel-jest`
  - `jest`
  - `jest-transform-svelte`
- Add npm script in `package.json`

`"test": "jest --watch src",`
- Run with `npm test`

# ... Unit Tests ...

- Create `babel.config.js`

```
module.exports = {
  presets: [
    [
      '@babel/preset-env',
      {
        targets: {
          node: 'current'
        }
      }
    ]
  ]
};
```

will get "regenerator-runtime not found" without this

- Create `jest.config.js`

```
module.exports = {
  transform: {
    '^.+\\.jsx?$': 'babel-jest',
    '^.+\\.svelte$': 'jest-transform-svelte'
  },
  moduleFileExtensions: ['js', 'svelte'],
  bail: false,
  verbose: true
};
```

means don't exit a suite  
after a test fails

shows result of each test  
rather than just each suite



# ... Unit Tests ...

src/TODO.spec.js

```
import {cleanup, render} from '@testing-library/svelte';

import Todo from './Todo.svelte';

describe('Todo', () => {
  const text = 'buy milk';
  const todo = {text};

  beforeEach(cleanup);

  test('should render', () => {
    const {getByText} = render(Todo, {props: {todo}});
    const checkbox = document.querySelector('input[type="checkbox"]');
    expect(checkbox).not.toBeNull();
    expect(getByText(text)); // the todo text
    expect(getByText('Delete')); // the button
  });

  // There is no easy way to test that events are fired
  // when the checkbox state is changed
  // or when the "Delete" button is pressed.
  // These are covered by tests in TodoList.spec.js.
});
```

# ... Unit Tests ...

```
import {tick} from 'svelte';
import {cleanup, fireEvent, render, wait} from '@testing-library/svelte';
import {src/ToDoList.spec.js}
import ToDoList from './ToDoList.svelte';

describe('ToDoList', () => {
  const PREDEFINED_TODOS = 2;

  beforeEach(cleanup);

  function expectTodoCount(count) {
    return wait(() => {
      // Each todo has an <li> root element.
      const lis = document.querySelectorAll('li');
      expect(lis.length).toBe(count);
    });
  }

  test('should render', async () => {
    const {getByText} = render(ToDoList);
    expect(getByText('To Do List'));
    expect(getByText('1 of 2 remaining'));
    expect(getByText('Archive Completed')); // button
    await expectTodoCount(PREDEFINED_TODOS);
  });
});
```



# ... Unit Tests ...

src/ToDoList.spec.js

```
test('should add a todo', async () => {
  const {getByTestId, getByText} = render(ToDoList);

  const input = getByTestId('todo-input');
  const value = 'buy milk';
  fireEvent.input(input, {target: {value}});
  fireEvent.click(getByText('Add'));

  await expectTodoCount(PREDEFINED_TODOS + 1);
  expect(getByText(value));
});

test('should archive completed', async () => {
  const {getByText} = render(ToDoList);
  fireEvent.click(getByText('Archive Completed'));
  await expectTodoCount(PREDEFINED_TODOS - 1);
  expect(getByText('1 of 1 remaining'));
});
```

# ... Unit Tests

```
test('should delete a todo', async () => {  
  const {getAllByText, getByText} = render(TodoList);  
  const text = 'learn Svelte'; // first todo  
  expect(getByText(text));  
  
  const deleteBtns = getAllByText('Delete');  
  fireEvent.click(deleteBtns[0]); // deletes first todo  
  await expectTodoCount(PREDEFINED_TODOS - 1);  
});  
  
test('should toggle a todo', async () => {  
  const {container, getByText} = render(TodoList);  
  const checkboxes = container.querySelectorAll('input[type="checkbox"]');  
  
  fireEvent.click(checkboxes[1]); // second todo  
  await tick();  
  expect(getByText('0 of 2 remaining'));  
  
  fireEvent.click(checkboxes[0]); // first todo  
  await tick();  
  expect(getByText('1 of 2 remaining'));  
});  
});
```

src/TodoList.spec.js



# Cypress E2E Tests ...

- Setup

- `npm i -D cypress`
- edit `package.json` and add script `"test:e2e": "cypress open",`
- `npm test:e2e`
  - creates `cypress` directory with subdirectories
    - `fixtures` - can hold data used by tests; typically `.json` files that are imported into tests
    - `integration` - your tests go here; can have subdirectories
    - `plugins` - extend Cypress functionality; ex. <https://github.com/bahmutov/cypress-svelte-unit-test>
      - Cypress automatically runs code in `index.js` in this directory before running each spec file
    - `screenshots` - holds screenshots produced by calling `cy.screenshot()`
    - `support` - can add custom Cypress commands, making them available in tests
      - Cypress automatically runs code in `index.js` in this directory before running each spec file
  - opens browser window and runs all the provided tests
- close Cypress browser window
- delete all sample files in `cypress` subdirectories



# ... Cypress E2E Tests

- Create test files in `cypress/integration` with file extension of `.spec.js`
- `npm run test:e2e`
- Press “Run all specs” button
- Example
  - `TODO`



# Related Tools

- **Svelte VS Code extension**
- **Sapper** - <https://sapper.svelte.dev/>
  - “application framework powered by Svelte”
  - name may be a contraction of “Svelte” and “Application”
  - similar to Next and Gatsby
  - provides routing, server-side rendering, and code splitting
- **Svelte Native** - <https://svelte-native.technology/>
  - for implementing native mobile apps
  - based on nativescript-vue
  - community-driven project
- **Svelte GL** - <https://github.com/Rich-Harris/svelte-gl>
  - in-work Svelte version of Three.js
- **Svelte Testing Library** - <https://testing-library.com/docs/svelte-testing-library/intro>
- **Storybook** with Svelte - <https://storybook.js.org/docs/guides/guide-svelte/>

# Conclusion