🏀 📷 📝

- about
- posts
- codes
- talks
- ✨

December 13, 2016

Polymer 1.x Cheat Sheet

This is a cheat sheet for the Polymer 1.x library. It helps you write Web Components, which are pretty 🔥🔥🔥. If you're interested in a Polymer 2.0 cheat sheet, stay tuned: it will come. If you think something is missing from this page, tell me about it!

- Defining an element
- Defining a behaviour
- Lifecycle methods
- Data binding
- Observers
- Listeners
- Properties block
- Observing added and removed children
- Style modules
- Styling with custom properties and mixins
- Binding helper elements

Defining an element

Docs: registering an element, behaviours, shared style modules

```
<dom-module id="element-name">
  <template>
    <!-- Use one of these style declarations, but not both -->
    <!-- Use this if you don't want to include a shared style -->
    <style></style>
    <!-- Use this if you want to include a shared style -->
    <style include="some-style-module-name"></style>
  </template>
  <script>
```

```
    Polymer({
      is: 'element-name',
      // All of these are optional. Only keep the ones you need.
      behaviors: [],
      observers: [],
      listeners: {},
      hostAttributes: {},
      properties: {}
    });
  </script>
</dom-module>
```

Defining a behaviour

Docs: behaviours.

Defining a behavior to share implementation between different elements:

```
<script>
  MyNamespace.MyFancyBehaviorImpl = {
    // Code that you want common to elements, such
    // as behaviours, methods, etc.
  }

  MyNamespace.MyFancyBehavior = [
    MyFancyBehaviorImpl,
    /* You can add other behaviours here */
  ];
</script>
```

Using the behavior in an element:

```
<dom-module id="element-name">
  <template>
    <!-- ... -->
  </template>
  <script>
    Polymer({
      is: 'element-name',
      behaviors: [MyNamespace.MyCustomButtonBehavior]
      /* ... */
    });
  </script>
</dom-module>
```

Lifecycle methods

Docs: lifecycle callbacks.

```
Polymer({
  registered: function() {},
  created: function() {},
  ready: function() {},
  attached: function() {},
  detached: function() {}
});
```

There's an `attributeChanged` callback as well, but that's very rarely used.

Data binding

Docs: data binding, attribute binding, binding to array items, computed bindings.

Don't forget: Polymer camel-cases properties, so if in JavaScript you use `myProperty`, in HTML you would use `my-property`.

One way binding: when `myProperty` changes, `theirProperty` gets updated:

```
<some-element their-property="[[myProperty]]"></some-element>
```

Two way binding: when `myProperty` changes, `theirProperty` gets updated, and vice versa:

```
<some-element their-property="{{myProperty}}"></some-element>
```

Attribute binding: when `myProperty` is `true`, the element is hidden; when it's `false`, the element is visible:

```
<some-element hidden$="[[myProperty]]"></some-element>
```

Computed binding: binding to the `class` attribute will recompile styles when `myProperty` changes:

```
<some-element class$="[[_computeSomething(myProperty)]]"></some-element>

_computeSomething: function(prop) {
  return prop ? 'a-class-name' : 'another-class-name';
}
```

Observers

Docs: observers, multi-property observers, observing array mutations.

Adding an `observer` in the `properties` block lets you observe changes in the value of a property:

```
properties: {
  myProperty: {
    observer: '_myPropertyChanged'
  }
},

// The second argument is optional, and gives you the
// previous value of the property, before the update:
_myPropertyChanged: function(value, /*oldValue */) {
  //...
}
```

In the **observers** block:

```
observers: [
  '_doSomething(myProperty)',
  '_multiPropertyObserver(myProperty, anotherProperty)',
  '_observerForASubProperty(user.name)',
  // Below, items can be an array or an object:'
  '_observerForABunchOfSubPaths(items.*)'
]
```

Listeners

Docs: event listeners, imperative listeners.

```
listeners: {
  'click': '_onClick',
  'input': '_onInput',
  'something-changed': '_onSomethingChanged'
}
```

Properties block

Docs: declared properties, object/array properties, read-only properties, computed properties.

There are all the possible things you can use in the **properties** block. Don't just use all of them because you can; some (like **reflectToAttribute** and **notify**) can have performance implications.

```
properties: {
  basic: {
    type: Boolean | Number | String | Array | Object,

    // Value can be one of the types above, eg:
    value: true,

    // For an Array or Object, you must return it from a function
    // (otherwise the array will be defined on the prototype
    // and not the instance):
    value: function() { return ['cheese', 'pepperoni', 'more-cheese'] },

    reflectToAttribute: true | false,
    readOnly: true | false,
    notify: true | false
  },

  // Computed properties are essentially read-only, and can only be
  // updated when their dependencies change.
  basicComputedProperty: {
    computed: '_someFunction(myProperty, anotherProperty)'
  }
}
```

Observing added and removed children

Docs: DOM distribution, observe nodes.

If you have a content node for distribution:

```
<template>
  <slot id="distributed"></slot>
</template>
```

And you want to be notified when nodes have been added/removed:

```
attached: function() {
  this._observer =
    Polymer.dom(this.$.distributed).observeNodes(function(info) {
    // info is {addedNodes: [...], removedNodes: [...]}
  });
},
detached: function() {
  Polymer.dom(this.$.distributed).unobserveNodes(this._observer);
}
```

Style modules

Docs: shared style modules.

Defining styles that will be shared across different elements, in a file called **`my-shared-styles.html`** (for example):

```
<dom-module id="my-shared-styles">
  <template>
    <style>
      .red { color: red; }
      /* Custom property defined in the global scope */
      html {
        --the-best-red: #e91e63;
      }
    </style>
  </template>
</dom-module>
```

Include the shared style in a custom element:

```
<link rel="import" href="my-shared-styles.html">
<dom-module id="element-name">
  <template>
    <style include="my-shared-styles">
      /* Other styles in here */
    </style>
  </template>
  <script>
    Polymer({ is: 'element-name' });
  </script>
</dom-module>
```

Include the shared style in the main document:

```
<html>
```

```
<head>
  <link rel="import" href="my-shared-styles.html">
  <style is="custom-style" include="my-shared-styles">
    /* Other styles in here */
  </style>
</head>
<body>...</body>
</html>
```

Styling with custom properties and mixins

Docs: styling, CSS properties, CSS mixins, shim limitations

Note that the examples below depend on browser support for custom properties. For how to use the shim (spoilers: it's `<style is="custom-style">`) and its limitations, check the docs linked above.

Defining a custom property:

```
html /* or :host, or :root etc. */{
  --my-custom-radius: 5px;
}
```

Using a custom property:

```
.my-image {
  border-radius: var(--my-custom-radius);
}
```

Using a custom property with a fallback:

```
.my-image {
  border-radius: var(--my-custom-radius, 3px);
}
```

Using a custom property with a custom property fallback:

```
.my-image {
  border-radius: var(--my-custom-radius, var(--my-fallback));
}
```

Defining a mixin:

```
some-custom-element {
  --my-custom-mixin: {
    border-radius: 5px;
  };
}
```

Using a mixin:

```
.my-image {
  @apply --my-custom-mixin;
}
```

Binding helper elements

Docs: dom-repeat, dom-bind, dom-if

`dom-repeat` stamps and binds a template for each item in an array:

```
<template is="dom-repeat" items="{{employees}}">
  <div>First name: <span>{{item.first}}</span></div>
  <div>Last name: <span>{{item.last}}</span></div>
</template>
```

`dom-bind` stamps itself into the main document and adds a binding scope:

```
<html>
<body>
  <template is="dom-bind">
    <paper-input value="{{myText}}"></paper-input>
    <span>You typed: [[myText]]</span>
  </template>
</body>
<html>
```

`dom-if` stamps itself conditionally based on a property’s value:

```
<template is="dom-if" if="{{myProperty}}">
  <span>This content will appear when myProperty is truthy.</span>
</template>
```

thanks for reading! ❤

Tweet