



- [about](#)
- [posts](#)
- [codes](#)
- [talks](#)
- 

May 31, 2017

Polymer 2.x Cheat Sheet

This is a cheat sheet for the [Polymer 2.x](#) library. If you're looking for the Polymer 1.x cheat sheet, it is [here](#). If you think something is missing from this page, [tell me](#) about it!

- [Defining an element](#)
- [Extending an element](#)
- [Defining a mixin](#)
- [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: [1.x -> 2.x upgrade guide](#), [registering an element](#), [shared style modules](#).

```
<link rel="import" href="bower_components/polymer/polymer-element.html">
<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>
  class MyElement extends Polymer.Element {
    static get is() { return 'element-name'; }
    // All of these are optional. Only keep the ones you need.
    static get properties() { ... }
    static get observers() { ... }
  }

  // Associate the new class with an element name
  customElements.define(MyElement.is, MyElement);
</script>
</dom-module>

```

To get the class definition for a particular custom tag, you can use `customElements.get('element-name')`.

Extending an element

Docs: [extending elements](#), [inherited templates](#).

Instead of `Polymer.Element`, a custom element can extend a different element):

```

class ParentElement extends Polymer.Element {
  /* ... */
}
class ChildElement extends ParentElement {
  /* ... */
}

```

To change or add to the parent's template, override the `template` getter:

```

<dom-module id="child-element">
  <template>
    <style> /* ... */ </style>
    <span>bonus!</span>
  </template>
  <script>
    var childTemplate;
    var childTemplate = Polymer.DomModule.import('child-element', 'template');
    var parentTemplate = ParentElement.template.cloneNode(true);
    // Or however you want to assemble these.
    childTemplate.content.insertBefore(parentTemplate.firstChild, parentTemplate);

    class ChildElement extends ParentElement {
      static get is() { return 'child-element'; }
      // Note: the more work you do here, the slower your element is to
      // boot up. You should probably do the template assembling once, in a
      // static method outside your class (like above).
      static get template() {
        return childTemplate;
      }
    }
    customElements.define(ChildElement.is, ChildElement);
  </script>
</dom-module>

```

If you don't know the parent class, you can also use:

```
class ChildElement extends customElements.get('parent-element') {
  /* ... */
}
```

Defining a mixin

Docs: [mixins](#), [hybrid elements](#).

Defining a class expression mixin to share implementation between different elements:

```
<script>
  MyMixin = function(superClass) {
    return class extends superClass {
      // Code that you want common to elements.
      // If you're going to override a lifecycle method, remember that a) you
      // might need to call super but b) it might not exist
      connectedCallback() {
        if (super.connectedCallback) {
          super.connectedCallback();
        }
      }
    }
  }
  /* ... */
}
</script>
```

Using the mixin in an element definition:

```
<dom-module id="element-name">
  <template><!-- ... --></template>
  <script>
    // This could also be a sequence:
    //class MyElement extends AnotherMixin(MyMixin(Polymer.Element)) { ... }
    class MyElement extends MyMixin(Polymer.Element) {
      static get is() { return 'element-name' }
      /* ... */
    }
    customElements.define(MyElement.is, MyElement);
  </script>
</dom-module>
```

Using hybrid behaviors (defined in the 1.x syntax) as mixins:

```
<dom-module id="element-name">
  <template><!-- ... --></template>
  <script>
    class MyElement extends Polymer.mixinBehaviors([MyBehavior, MyBehavior2], Polymer.Element) {
      static get is() { return 'element-name' }
      /* ... */
    }
    customElements.define('element-name', MyElement);
  </script>
</dom-module>
```

Lifecycle methods

Docs: [lifecycle callbacks](#), [ready](#).

```

class MyElement extends Polymer.Element {
  constructor() { super(); /* ... */}
  ready() { super.ready(); /* ... */}
  connectedCallback() { super.connectedCallback(); /* ... */}
  disconnectedCallback() { super.disconnectedCallback(); /* ... */}
  attributeChangedCallback() { super.attributeChangedCallback(); /* ... */}
}

```

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. The difference between attribute and property binding is that property binding is equivalent to `someElement.someProp = value`, whereas attribute binding is equivalent to: `someElement.setAttribute(someProp, value)`

```

<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>
<script>
  _computeSomething: function(prop) {
    return prop ? 'a-class-name' : 'another-class-name';
  }
</script>

```

Observers

Docs: [observers](#), [multi-property observers](#), [observing array mutations](#), [adding observers dynamically](#).

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

```
static get properties() {
  return {
    myProperty: {
      observer: '_myPropertyChanged'
    }
  }
}

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

In the **observers** block:

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

Adding an observer dynamically for a property **otherProperty**:

```
// Define a method
_otherPropertyChanged(value) { /* ... */ }
// Call it when `otherProperty` changes
this._createPropertyObserver('otherProperty', '_otherPropertyChanged', true);
```

Listeners

In Polymer 2.0, we recommend that rather than using the **listeners** block, you `#useThePlatform` and define event listeners yourself:

```
ready() {
  super.ready();
  window.addEventListener('some-event', () => this.someFunction());
}
```

There is a [PR](#) out to add a declarative listener block as a mixin. Stay tuned!

Properties block

Docs: [declared properties](#), [object/array properties](#), [read-only properties](#), [computed properties](#), [adding computed properties dynamically](#).

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.

```
static get properties() {
  return {
    basic: {
      type: Boolean | Number | String | Array | Object,

      // Default value of the property 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)'
    }
  }
}
```

Adding a computed property dynamically:

```
this._createComputedProperty('newProperty', '_computeNewProperty(prop1,prop2)', true);
```

Observing added and removed children

Docs: [Shadow DOM distribution](#), [observe nodes](#).

If you have a content node for distribution:

```
<template>
  <slot></slot>
</template>
```

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

```
<!-- You need to import the observer -->
<link rel="import" href="/bower_components/polymer/lib/utils/flattened-nodes-observer.html">

<script>
class MyElement extends Polymer.Element {
  /* ... */
  connectedCallback: function() {
    super.connectedCallback();
    this._observer = new Polymer.FlattenedNodesObserver(function(info) {
      // info is {addedNodes: [...], removedNodes: [...]}
    });
  }
  disconnectedCallback: function() {
    super.disconnectedCallback();
  }
}
```

```

    this._observer.disconnect();
  }
}
</script>

```

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>
    class MyElement extends Polymer.Element {
      /* ... */
    }
  </script>
</dom-module>

```

Include the shared style in the main document:

```

<html>
<head>
  <!-- Import the custom-style element -->
  <link rel="import" href="components/polymer/lib/elements/custom-style.html">
  <link rel="import" href="my-shared-styles.html">
  <custom-style>
    <style include="my-shared-styles">
      /* Other styles in here */
    </style>
  </custom-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 and mixins.

Defining a custom property:

```
html /* or :host 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));  
}
```

If you want to use mixins, you need to include the CSS mixins shim. For how to use the shim and its limitations, check the docs linked at the beginning of the section.

```
<link rel="import" href="/bower_components/shadycss/apply-shim.html">
```

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](#)

There are two ways to use the helper elements:

- inside a Polymer element/Polymer managed template: just use the `<template is=...>` syntax, without the wrapper, for example:

```
<template is="dom-repeat">
  ...
</template>
```

- outside of a Polymer managed template: use the `<dom-...>` wrapper element around a `<template>`, for example:

```
<dom-repeat>
  <template>
    ...
  </template>
</dom-repeat>
```

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


```
<link rel="import" href="components/polymer/lib/elements/dom-repeat.html">
<dom-repeat items="[[employees]]">
  <template>
    <div>First name: <span>[[item.first]]</span></div>
    <div>Last name: <span>[[item.last]]</span></div>
  </template>
</dom-repeat>
```

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

```
<link rel="import" href="components/polymer/lib/elements/dom-bind.html">
<html>
<body>
  <dom-bind>
    <template>
      <paper-input value="{{myText}}"></paper-input>
      <span>You typed: [[myText]]</span>
    </template>
  </dom-bind>
</body>
</html>
```

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

```
<link rel="import" href="components/polymer/lib/elements/dom-if.html">
<dom-if if="[[myProperty]]">
  <template>
    <span>This content will appear when myProperty is truthy.</span>
  </template>
</dom-if>
```

thanks for reading! 

[Tweet](#)