Polymer library

EDIT ON GITHUB

The Polymer library provides a set of features for creating custom elements. These features are designed to make it easier and faster to make custom elements that work like standard DOM elements. Similar to standard DOM elements, Polymer elements can be:

- Instantiated using a constructor or `document.createElement`.
- Configured using attributes or properties.
- Populated with internal DOM inside each instance.
- Responsive to property and attribute changes.
- Styled with internal defaults or externally.
- Responsive to methods that manipulate its internal state.

A basic Polymer element definition looks like this:

```
import {PolymerElement, html} from '@polymer/polymer/polymer-element

// Define the element's API using an ES2015 class
class XCustom extends PolymerElement {

  // Define optional shadow DOM template
  static get template() {
    return html`
      <style>
        /* CSS rules for your element */
      </style>

        <!-- shadow DOM for your element -->

      <div>[[greeting]]</div> <!-- data bindings in shadow DOM -->
    `;
  }

  // Declare properties for the element's public API
  static get properties() {
    return {
```

```
    }
  }

  constructor() {
    super();
    this.greeting = 'Hello!';
  }

  // Add methods to the element's public API
  greetMe() {
    console.log(this.greeting);
  }

}

// Register the x-custom element with the browser
customElements.define('x-custom', XCustom);
```

This guide divides the features into the following groups:

- **Custom elements**. Registering an element associates a class with a custom element name. The element provides callbacks to manage its lifecycle. Polymer also lets you declare properties, to integrate your element's property API with the Polymer data system.
- **Shadow DOM**. Shadow DOM provides a local, encapsulated DOM tree for your element. Polymer can automatically create and populate a shadow tree for your element from a DOM template.
- **Events**. Polymer provides a declarative syntax for attaching event listeners to shadow DOM children. It also provides an optional library for handling gesture events.
- **Data system**. The Polymer data system provides data binding to properties and attributes; property observers; and computed properties.

If you're upgrading an existing 2.x element to 3.x, see the **Upgrade guide** for advice.

If you're looking for the latest changes in this release, see the **Release notes**.

Polymer Project

Custom element concepts

EDIT ON GITHUB

▶ **Contents**

Custom elements provide a component model for the web. The custom elements specification provides:

- A mechanism for associating a class with a custom element name.
- A set of lifecycle callbacks invoked when an instance of the custom element changes state (for example, added or removed from the document).
- A callback invoked whenever one of a specified set of attributes changes on the instance.

Put together, these features let you build an element with its own public API that reacts to state changes. Polymer provides a set of features on top of the basic custom element specification.

This document provides an overview of custom elements as they relate to Polymer. For a more detailed overview of custom elements, see: **Custom Elements v1: Reusable Web Components** on Web Fundamentals.

To define a custom element, you create an ES6 class and associate it with the custom element name. For the full set of Polymer features, extend the `PolymerElement` class:

```
import {PolymerElement} from '@polymer/polymer/polymer-element.js';

  export class MyPolymerElement extends PolymerElement {
    ...
  }

  customElements.define('my-polymer-element', MyPolymerElement);
```

Exporting the custom element class is optional, but recommended.

Import the element into an HTML file using `<script type="module">`.
Use the `import` statement (as shown above) to import it from another ES6 module.

```
<script type="module" src="./my-polymer-element.js">
```

The element's class defines its behavior and public API.

> **Custom element names.** By specification, the custom element's name **must start with a lower-case ASCII letter and must contain a dash (-)**. There's also a short list of prohibited element names that match existing names. For details, see the Custom elements core concepts section in the HTML specification.

Polymer adds a set of features to the basic custom element:

- Instance methods to handle common tasks.
- Automation for handling properties and attributes, such as setting a property based on the corresponding attribute.
- Creating shadow DOM trees for element instances based on a supplied template.
- A data system that supports data binding, property change observers, and computed properties.

The `PolymerElement` class is made up of a set of *class expression mixins* that add individual features. You can also use these mixins individually if you want to use a subset of Polymer's features. See the API documentation for a list of individual mixins.

Polymer element lifecycle

Polymer elements follow the standard lifecycle for custom elements. The custom element spec provides a set of callbacks called "custom element reactions" that allow you to run user code in response to certain lifecycle changes.

For performance, Polymer defers creating an element's shadow tree and initializing its data system until the first time the element is attached to the DOM. Polymer adds its own `ready` callback for this initialization.

Reaction	Description
constructor	Called when the element is upgraded (that is, when an element is created, or when a previously-created element becomes defined). The constructor is a logical place to set default values, and to manually set up event listeners for the element itself.
connectedCallback	Called when the element is added to a document. Can be called multiple times during the lifetime of an element.

disconnectedCallback	Called when the element is removed from a document. Can be called multiple times during the lifetime of an element. Uses include removing event listeners added in `connectedCallback`.
ready	Called during Polymer-specific element initialization. Called once, the first time the element is attached to the document. For details, see **Polymer element initialization**.
attributeChangedCallback	Called when any of the element's attributes are changed, appended, removed, or replaced. Use to handle attribute changes that *don't* correspond to declared properties. (For declared properties, Polymer handles attribute changes automatically as described in **attribute deserialization**.)

For each reaction, the first line of your implementation must be a call to the superclass constructor or reaction. For the constructor, this is simply the `super()` call.

```
constructor() {
  super();
  // …
}
```

For other reactions, call the superclass method. This is required so Polymer can hook into the element's lifecycle.

```
connectedCallback() {
  super.connectedCallback();
  // …
}
```

The element constructor has a few special limitations:

(`return` or `return this`).

- The constructor can't examine the element's attributes or children, and the constructor can't add attributes or children.

For a complete list of limitations, see **Requirements for custom element constructors** in the WHATWG HTML Specification.

Whenever possible, defer work until the `connectedCallback` or later instead of performing it in the constructor. See **Defer non-critical work** for some suggestions.

Polymer element initialization

The custom elements specification doesn't provide a one-time initialization callback. Polymer provides a `ready` callback, invoked the first time the element is added to the DOM. (If the element is upgraded when it's already in the document, `ready` runs when the element is upgraded.)

```
ready() {
  super.ready();
  // do something that requires access to the shadow tree
  ...

}
```

The `PolymerElement` class initializes your element's template and data system during the `ready` callback, so if you override `ready`, you must call `super.ready()` before accessing the element's shadow tree.

Polymer does several things at `ready` time:

- Creates and attaches the element's shadow DOM tree.
- Initializes the data system, propagating intial values to data bindings.
- Allows observers and computed properties to run (as soon as any of their dependencies are defined).

When the superclass `ready` method returns, the element's template has been instantiated and initial property values have been set. However, light DOM elements may not have been distributed when `ready` is called.

Don't use `ready` to initialize an element based on dynamic values, like property values or an element's light DOM children. Instead, use **observers** to react to property changes,

Related topics:

- DOM templating
- Data system concepts
- Observers and computed properties
- Observe added and removed children

Defer non-critical work

When possible, defer work until after first paint. The `render-status` module provides an `afterNextRender` utility for this purpose.

```
import {PolymerElement} from '@polymer/polymer/polymer-element.js';
import {afterNextRender} from '@polymer/polymer/lib/utils/render-sta


class DeferElement extends PolymerElement {
  ...
  constructor() {
    super();
    // When possible, use afterNextRender to defer non-critical
    // work until after first paint.
    afterNextRender(this, function() {
      this.addEventListener('click', this._handleClick);
    });
  }
}
```

In most cases, you can call `afterNextRender` from either the `constructor` or the `ready` callback with similar results. For anything requiring access to the element's shadow tree, use the `ready` callback.

Element upgrades

By specification, custom elements can be used before they're defined. Adding a definition for an element causes any existing instances of that element to be *upgraded* to the custom class.

For example, consider the following code:

```
<!-- load the element definition -->
<script type="module" src="my-element.js">
```

When parsing this page, the browser will create an instance of `<my-element>` before parsing and executing the script. In this case, the element is created as an instance of `HTMLElement`, not `MyElement`. After the element is defined, the `<my-element>` instance is upgraded so it has the correct class (`MyElement`). The class constructor is called during the upgrade process, followed by any pending lifecycle callbacks.

Element upgrades allow you to place elements in the DOM while deferring the cost of initializing them. It's a progressive enhancement feature.

To avoid unstyled content, you can apply styles to undefined elements. See **Style undefined elements** for details.

Extending other elements

In addition to `PolymerElement`, a custom element can extend another custom element:

```
import {MyElment} from './my-element.js';

export class ExtendedElement extends MyElement {
  static get is() { return 'extended-element'; }

  static get properties() {
    return {
      thingCount: {
        value: 0,
        observer: '_thingCountChanged'
      }
    }
  }
  _thingCountChanged() {
    console.log(`thing count is ${this.thingCount}`);
  }
};

customElements.define(ExtendedElement.is, ExtendedElement);
```

> `<button>` and `<input>`. The spec calls these elements *customized built-in elements*. Customized built-in elements provide many advantages (for example, being able to take advantage of built-in accessibility features of UI elements like `<button>` and `<input>`). However, not all browser makers have agreed to support customized built-in elements, so Polymer does not support them at this time.

When you extend custom elements, Polymer treats the `properties` object and `observers` array specially: when instantiating an element, Polymer walks the prototype chain and flattens these objects. So the properties and observers of a subclass are added to those defined by the superclass.

A subclass can also inherit a template from its superclass. For details, see **Inherit a template from another Polymer element**.

To make it easy to extend your elements, the module that defines the element should export it:

```
export class MyElement extends PolymerElement { ... }
```

Legacy elements—elements defined using the legacy `Polymer()` function—don't require you to define your own class. So if you're extending a legacy element, like one of the Polymer paper elements, the module may not export a class.

If you're extending a legacy Polymer element, or a module that doesn't export the element, you can use the `customElements.get` method to retrieve the constructor for any custom element that's been defined.

```
// Import a legacy component
import './legacy-button.js';
// Retrieve the legacy-button constructor
const LegacyButton = customElements.get('legacy-button');
// Extend it!
export class MyExtendedButton extends LegacyButton { ... }
```

Sharing code with class expression mixins

ES6 classes allow single inheritance, which can make it challenging to share code between unrelated elements. Class expression mixins let you share code between elements without adding a common superclass.

```
const fancyDogClass = FancyMixin(dogClass);
const fancyCatClass = FancyMixin(catClass);
```

Using mixins

Add a mixin to your element like this:

```
class MyElement extends MyMixin(PolymerElement) {
  static get is() { return 'my-element' }
}
```

If that isn't clear, it may help to see it in two steps:

```
// Create new base class that adds MyMixin's methods to Polymer.Elem
const PolymerElementPlusMixin = MyMixin(PolymerElement);

// Extend the new base class
class MyElement extends PolymerElementPlusMixin {
  static get is() { return 'my-element' }
}
```

So the inheritance hierarchy is:

```
MyElement <= PolymerElementPlusMixin <= PolymerElement
```

You can apply mixins to any element class, not just `PolymerElement`:

```
class MyExtendedElement extends SomeMixin(MyElement) {
  ...
}
```

You can also apply multiple mixins in sequence:

```
class AnotherElement extends AnotherMixin(MyMixin(PolymerElement)) {
```

Defining mixins

```
  return class extends superClass {
    constructor() {
      super();
      this.addEventListener('keypress', (e) => this._handlePress(e))
    }

    static get properties() {
      return {
        bar: {
          type: Object
        }
      };
    }

    static get observers() {
      return [ '_barChanged(bar.*)' ];
    }

    _barChanged(bar) { ... }

    _handlePress(e) { console.log('key pressed: ' + e.charCode); }
  }
}
```

Or using an ES6 arrow function:

```
MyMixin = (superClass) => class extends superClass {
  ...
}
```

The mixin class can define properties, observers, and methods just like a regular element class. In addition, a mixin can incorporate other mixins:

```
MyCompositeMixin = (base) => class extends MyMixin2(MyMixin1(base))
  ...
}
```

Because mixins are simply adding classes to the inheritance chain, all of the usual rules of inheritance apply. For example, mixin classes can define constructors, can call

documentation tags to property analyze mixins and elements that use them. Without the documentation tags, the tools will log warnings. For details on documenting mixins, see **Class mixins** in Document your elements.

Packaging mixins for sharing

When creating a mixin that you intend to share with other groups or publish, a couple of additional steps are recommended:

- Use the `dedupingMixin` function to produce a mixin that can only be applied once.
- Define the mixin in an ES module and export it.

The `dedupingMixin` function is useful because a mixin that's used by other mixins may accidentally be applied more than once. For example if `MixinA` includes `MixinB` and `MixinC`, and you create an element that uses `MixinA` but also uses `MixinB` directly:

```
class MyElement extends MixinB(MixinA(Polymer.Element)) { ... }
```

At this point, your element contains two copies of `MixinB` in its prototype chain. `dedupingMixin` takes a mixin function as an argument, and returns a new, deduplicating mixin function:

mixin-b.js

```
import {dedupingMixin} from '@polymer/polymer/lib/utils/mixin.js';

// define the mixin
let internalMixinB = (base) =>
  class extends base {
    ...
  }

// deduplicate and export it
export const MixinB = dedupingMixin(internalMixinB);
```

Using the mixin

```
class Foo extends MixinB(PolymerElement) { ... }
```

The deduping mixin has two advantages: first, whenever you use the mixin, it memoizes the generated class, so any subsequent uses on the same base class return the same class object—a minor optimization.

More importantly, the deduping mixin checks whether this mixin has already been applied anywhere in the base class's prototype chain. If it has, the mixin simply returns the base class. In the example above, if you used `dedupingMixinB` instead of `mixinB` in both places, the mixin would only be applied once.

Resources

More information: **Custom elements v1: reusable web components** on Web Fundamentals.

Define an element

EDIT ON GITHUB

▶ **Contents**

Define a custom element

To define a custom element, create a class that extends `PolymerElement` and pass the class to the `customElements.define` method.

By specification, the custom element's name **must start with a lower-case ASCII letter and must contain a dash (-)**.

Example:

```
// Import PolymerElement class
import {PolymerElement} from '@polymer/polymer/polymer-element.js';

// define the element's class element
class MyElement extends PolymerElement {

  // Element class can define custom element reactions
  connectedCallback() {
    super.connectedCallback();
    this.textContent = 'I\'m a custom element!';
    console.log('my-element created!');
  }

  ready() {
    super.ready();
    console.log('my-element is ready!');
  }
}

// Associate the new class with an element name
```

```
var el1 = document.createElement('my-element');

// ... or with the constructor:
var el2 = new MyElement();
```

As shown above, the element's class can define callbacks for the custom element reactions as described in Custom element lifecycle.

Extending an existing element

You can leverage native subclassing support provided by ES6 to extend and customize existing elements defined using ES6 syntax:

```
// Subclass existing element
class MyElementSubclass extends MyElement {
  static get is() { return 'my-element-subclass'; }
  static get properties() { ... }
  constructor() {
    super();
    ...
  }
  ...
}

// Register custom element definition using standard platform API
customElements.define(MyElementSubclass.is, MyElementSubclass);
```

For more information on extending elements, see Extending other elements in Custom element concepts.

If you don't provide a template for your subclass, it inherits the superclass's template by default. For more information, see Inherit a template from another Polymer element.

Using mixins

You can share code using *class expression mixins*. You use a mixin to add new features on top of a base class:

```
class MyElementWithMixin extends MyMixin(PolymerElement) {
```

This pattern may be easier to understand if you think of it as two steps:

```
// Create a new base class that adds MyMixin's features to Polymer.E
const BaseClassWithMixin = MyMixin(PolymerElement);

// Extend the new base class
class MyElementWithMixin extends BaseClassWithMixin { ... }
```

Because mixins are simply adding classes to the inheritance chain, all of the usual rules of inheritance apply.

For information on defining mixins, see Sharing code with class expression mixins in Custom element concepts.

Using legacy behaviors with class-style elements

You can add legacy behaviors to your class-style element using the `mixinBehavior` function:

```
import {PolymerElement} from '@polymer/polymer/lib/legacy/class.js';
import {mixinBehaviors} from '@polymer/polymer/polymer-element.js';

class XClass extends Polymer.mixinBehaviors([MyBehavior, MyBehavior2

  ...
}
customElements.define('x-class', XClass);
```

The `mixinBehavior` function also mixes in the Legacy APIs, the same as if you applied the `LegacyElementMixin`. These APIs are required since since legacy behaviors depend on them.

Declare Properties

EDIT ON GITHUB

▶ **Contents**

You can declare properties on an element to add a default value and enable various features in the data system.

Declared properties can specify:

- Property type.
- Default value.
- Property change observer. Calls a method whenever the property value changes.
- Read-only status. Prevents accidental changes to the property value.
- Two-way data binding support. Fires an event whenever the property value changes.
- Computed property. Dynamically calculates a value based on other properties.
- Property reflection to attribute. Updates the corresponding attribute value when the property value changes.

Many of these features are tightly integrated into the **data system**, and are documented in the data system section.

In addition, a declared property can be configured from markup using an attribute (see **attribute deserialization** for details).

In most cases, a property that's part of your element's public API should be declared in the `properties` object.

To declare properties, add a static `properties` getter to the element's class. The getter should return an object containing property declarations.

Example

```
class XCustom extends PolymerElement {

  static get properties() {
    return {
      user: String,
      isHappy: Boolean,
```

```
        notify: true
      }
    }
  }
}

customElements.define('x-custom', XCustom);
```

The `properties` object supports the following keys for each property:

Key	Details
`type`	Type: constructor Attribute type, used for deserializing from an attribute. Polymer supports deserializing the following types: `Boolean`, `Date`, `Number`, `String`, `Array` and `Object`. You can add support for other types by overriding the element's `_deserializeValue` method. Unlike 0.5, the property's type is explicit, specified using the type's constructor. See **attribute deserialization** for more information.
`value`	Type: `boolean`, `number`, `string` or `function`. Default value for the property. If `value` is a function, the function is invoked and the return value is used as the default value of the property. If the default value should be an array or object unique to the instance, create the array or object inside a function. See **Configuring default property values** for more information.
`reflectToAttribute`	Type: `boolean` Set to `true` to cause the corresponding attribute to be set on the host node when the property value changes. If the property value is Boolean, the attribute is created as a standard HTML boolean attribute (set if true, not set if false). For other property types, the attribute value is a string

`readOnly`	Type: `boolean` If `true`, the property can't be set directly by assignment or data binding. See **Read-only properties**.
`notify`	Type: `boolean` If `true`, the property is available for two-way data binding. In addition, an event, *`property-name`*`-changed` is fired whenever the property changes. See **Property change notification events (notify)** for more information.
`computed`	Type: `string` The value is interpreted as a method name and argument list. The method is invoked to calculate the value whenever any of the argument values changes. Computed properties are always read-only. See **Computed properties** for more information.
`observer`	Type: `string` The value is interpreted as a method name to be invoked when the property value changes. See **Property change callbacks (observers)** for more information.

Property name to attribute name mapping

For data binding, deserializing properties from attributes, and reflecting properties back to attributes, Polymer maps attribute names to property names and the reverse.

When mapping attribute names to property names:

- Attribute names are converted to lowercase property names. For example, the attribute `firstName` maps to `firstname`.

The same mappings happen in reverse when converting property names to attribute names (for example, if a property is defined using `reflectToAttribute: true`.)

> **Compatibility note:** In 0.5, Polymer attempted to map attribute names to corresponding properties. For example, the attribute `foobar` would map to the property `fooBar` if it was defined on the element. This **does not happen in 1.0**—attribute to property mappings are set up on the element at registration time based on the rules described above.

Attribute deserialization

If a property is configured in the `properties` object, an attribute on the instance matching the property name will be deserialized according to the type specified and assigned to a property of the same name on the element instance.

If no other `properties` options are specified for a property, the `type` (specified using the type constructor, e.g. `Object`, `String`, etc.) can be set directly as the value of the property in the `properties` object; otherwise it should be provided as the value to the `type` key in the `properties` configuration object.

Boolean properties are set based on the *presence* of the attribute: if the attribute exists at all, the property is set to `true`, regardless of the attribute *value*. If the attribute is absent, the property gets its default value.

Example:

```
<script type="module">
  import {PolymerElement} from '@polymer/polymer/polymer-element.js'

  class XCustom extends PolymerElement {

    static get properties() {
      return {
        user: String,
        manager: {
          type: Boolean,
          notify: true
        }
      }
```

```
      super.connectedCallback();

      // render
      this.textContent = 'Hello World, my user is ' + (this.user ||
        'This user is ' + (this.manager ? '' : 'not') + ' a manager.
    }
  }

  customElements.define('x-custom', XCustom);
</script>

<x-custom user="Scott" manager></x-custom>
<!--
<x-custom>'s text content becomes:
Hello World, my user is Scott.
This user is a manager.
-->
```

To configure camel-case properties of elements using attributes, dash- case should be used in the attribute name.

Example:

```
<script type="module">
  import {PolymerElement} from '@polymer/polymer/polymer-element.js'

  class XCustom extends Polymer.Element {

    static get properties() {
      return {
        userName: String
      }
    }
  }

  customElements.define('x-custom', XCustom);
</script>

<x-custom user-name="Scott"></x-custom>
<!-- Sets <x-custom>.userName = 'Scott';  -->
```

the attribute is changed using `setAttribute`). However, it is encouraged that attributes only be used for configuring properties in static markup, and instead that properties are set directly for changes at runtime.

Configuring boolean properties

For a Boolean property to be configurable from markup, it must default to `false`. If it defaults to `true`, you cannot set it to `false` from markup, since the presence of the attribute, with or without a value, equates to `true`. This is the standard behavior for attributes in the web platform.

If this behavior doesn't fit your use case, you can use a string-valued or number-valued attribute instead.

Configuring object and array properties

For object and array properties you can pass an object or array in JSON format:

```
<my-element book='{ "title": "Persuasion", "author": "Austen" }'></m
```

Note that JSON requires double quotes, as shown above.

Custom deserializers

The type system includes built-in support for Boolean and Number values, Object and Array values expressed as JSON, or Date objects expressed as any Date-parsable string representation. To support other types, you can override the element's `_deserializeValue` method.

```
_deserializeValue(value, type) {
  if (type == MyCustomType) {
    return stringToMyCustomType(value);
  } else {
    return super._deserializeValue(value, type);
  }
}
```

Default values for properties may be specified in the `properties` object using the `value` field, **or** set imperatively in the element's `constructor`.

The value in the `properties` object may either be a primitive value, or a function that returns a value.

If you provide a function, Polymer calls the function once *per element instance*.

When initializing a property to an object or array value, either initialize the property in the constructor, or use a function to ensure that each element gets its own copy of the value, rather than having an object or array shared across all instances of the element.

Default in properties object

```
class XCustom extends PolymerElement {

  static get properties() {
    return {
      mode: {
        type: String,
        value: 'auto'
      },

      data: {
        type: Object,
        notify: true,
        value: function() { return {}; }
      }
    }
  }
}
```

Default in constructor

```
constructor() {
  super();
  this.mode = 'auto';
  this.data = {};
}

static get properties() {
  return {
```

```
      type: Object,
      notify: true
    }
  }
}
```

Property change notification events (notify)

When a property is set to `notify: true`, an event is fired whenever the property value changes. The event name is:

property-name-changed

Where *property-name* is the dash-case version of the property name. For example, a change to `this.firstName` fires `first-name-changed`.

These events are used by the two-way data binding system. External scripts can also listen for events (such as `first-name-changed`) directly using `addEventListener`. Property change events don't bubble, so the event listener must be added directly to the element generating the event.

For more on property change notifications and the data system, see **Data flow**.

Read-only properties

When a property only "produces" data and never consumes data, this can be made explicit to avoid accidental changes from the host by setting the `readOnly` flag to `true` in the `properties` property definition. In order for the element to actually change the value of the property, it must use a private generated setter of the convention `_set*Property*(value)` where *Property* is the property name, with the first character converted to uppercase (if alphabetic). For example, the setter for `oneProperty` is `_setOneProperty`, and the setter for _privateProperty is `_set_privateProperty`.

```
class XCustom extends PolymerElement {

  static get properties() {
    return {
      response: {
        type: Object,
```

```
    }
  }

  responseHandler(response) {
    // set read-only property
    this._setResponse(response);
  }
}
```

For more on read-only properties and data binding, see **How data flow is controlled**.

Reflecting properties to attributes

In specific cases, it may be useful to keep an HTML attribute value in sync with a property value. This may be achieved by setting `reflectToAttribute: true` on a property in the `properties` configuration object. This causes any observable change to the property to trigger an update to the corresponding attribute (as described in **Property name to attribute name mapping**). Since attributes only take string values, the property value is serialized to a string, as described in **Attribute serialization**.

```
class XCustom extends PolymerElement {

  static get properties() {
    return {
      loaded: {
        type: Boolean,
        reflectToAttribute: true
      }
    }
  }

  _onLoad() {
    this.loaded = true;
    // results in this.setAttribute('loaded', true);
  }
}
```

Attribute serialization

By default, values are serialized according to value's **current type**, *regardless of the property's `type` value*:

- `String`. No serialization required.
- `Date` or `Number`. Serialized using `toString`.
- `Boolean`. Results in a non-valued attribute to be either set (`true`) or removed (`false`).
- `Array` or `Object`. Serialized using `JSON.stringify`.

To add custom serialization for other data types, override your element's `_serializeValue` method.

```
_serializeValue(value) {
  if (value instanceof MyCustomType) {
    return value.toString();
  }
  return super._serializeValue(value);
}
```

Implicitly declared properties

A property is declared *implicitly* if you add it to a data binding or add it as a dependency of an observer, computed property, or computed binding.

Polymer automatically creates setters for these implicitly declared properties. However, implicitly declared properties can't be configured from markup.

Private and protected properties

JavaScript doesn't have any true protection for properties. By convention, Polymer elements usually use a single underscore (`_protectedProp`) to indicate a protected property or method (intended to be used or overridden by subclasses, but not for public use), and a double underscore (`__privateProp`) for members that are private to the class.

Work with legacy elements

EDIT ON GITHUB

▶ **Contents**

Define a legacy element

Legacy elements can use use the `Polymer` function to register an element. The function takes as its argument the prototype for the new element. The prototype must have an `is` property that specifies the HTML tag name for your custom element.

By specification, the custom element's name **must start with an ASCII letter and contain a dash (-)**.

Example:

```
// register an element
MyElement = Polymer({

  is: 'my-element',

  // See below for lifecycle callbacks
  created: function() {
    this.textContent = 'My element!';
  }

});

// create an instance with createElement:
var el1 = document.createElement('my-element');

// ... or with the constructor:
var el2 = new MyElement();
```

The `Polymer` function registers the element with the browser and returns a constructor that can be used to create new instances of your element via code.

between elements.

Legacy lifecycle callbacks

Legacy elements use a different set of lifecycle callbacks than standard Polymer 3.x elements. These callbacks are based on the custom elements v0 lifecycle that was supported in Polymer 1.x.

Legacy callback	Description
`created`	Called when the element has been created, but before property values are set and local DOM is initialized. Use for one-time set-up before property values are set. Equivalent to the native constructor.
`ready`	Called after property values are set and local DOM is initialized. Use for one-time configuration of your component after its shadow DOM tree is initialized. (For configuration based on property values, it may be preferable to use an **observer**.)
`attached`	Called after the element is attached to the document. Can be called multiple times during the lifetime of an element. The first `attached` callback is guaranteed not to fire until after `ready`. Uses include adding document-level event listeners. (For listeners local to the element, you can use declarative event handling, such as **annotated event listeners**.) Equivalent to native `connectedCallback`.
`detached`	Called after the element is detached from the document. Can be called multiple times during the lifetime of an element. Uses include removing event listeners added in `attached`. Equivalent to native `disconnectedCallback`.

Use to handle attribute changes that *don't* correspond to declared properties. (For declared properties, Polymer handles attribute changes automatically as described in **attribute deserialization**.)

Equivalent to the native `attributeChangedCallback`.

Legacy behaviors

Legacy elements can share code in the form of *behaviors*, which can define properties, lifecycle callbacks, event listeners, and other features.

For more information, see **Behaviors** in the Polymer 1.x docs.