# Custom Element Best Practices

Custom elements allow you to extend HTML and define your own tags. They're an incredibly powerful feature, but they're also low-level, which means it's not always clear how best to implement your own element.

To help you create the best possible experiences we've put together this checklist which breaks down all the things we think it takes to be a well behaved custom element.

## Checklist

### Shadow DOM

#### Create a shadow root to encapsulate styles.

| | |
|---|---|
| **Why?** | Encapsulating styles in your element's shadow root ensures that it will work regardless of where it is used. This is especially important if a developer wishes to place your element inside of another element's shadow root. This applies to even simple elements like a checkbox or radio button. It might be the case that the only content inside of your shadow root will be the styles themselves. |
| **Example** | The `<howto-checkbox>` element. |

#### Create your shadow root in the constructor.

| | |
|---|---|
| **Why?** | The constructor is when you have **exclusive knowledge** of your element. It's a great time to setup implementation details that you don't want other elements messing around with. Doing this work in a later callback, like the **`connectedCallback`**, means you will need to guard against situations where your element is detached and then reattached to the document. |
| **Example** | The `<howto-checkbox>` element. |

#### Place any children the element creates into its shadow root.

| | |
|---|---|
| **Why?** | Children created by your element are part of its implementation and should be private. Without the protection of a shadow root, outside JavaScript may inadvertently interfere with these children. |
| **Example** | The `<howto-tabs>` element. |

### Set a `:host` display style (e.g. `block`, `inline-block`, `flex`) unless you prefer the default of `inline`.

| | |
|---|---|
| **Why?** | Custom elements are **`display: inline`** by default, so setting their **`width`** or **`height`** will have no effect. This often comes as a surprise to developers and may cause issues related to laying out the page. Unless you prefer an **`inline`** display, you should always set a default **`display`** value. |
| **Example** | The `<howto-checkbox>` element. |

### Add a `:host` display style that respects the hidden attribute.

| | |
|---|---|
| **Why?** | A custom element with a default **`display`** style, e.g. **`:host { display: block }`**, will override the lower specificity built-in hidden attribute. This may surprise you if you expect setting the **`hidden`** attribute on your element to render it **`display: none`**. In addition to a default **`display`** style, add support for **`hidden`** with **`:host([hidden]) { display: none }`**. |
| **Example** | The `<howto-checkbox>` element. |

## Attributes and properties

### Do not override author-set, global attributes.

| | |
|---|---|
| **Why?** | Global attributes are those that are present on all HTML elements. Some examples include **`tabindex`** and **`role`**. A custom element may wish to set its initial **`tabindex`** to 0 so it will be keyboard focusable. But you should always check first to see if the developer using your element has set this to another value. If, for example, they've set **`tabindex`** to -1, it's a signal that they don't wish for the element to be interactive. |
| **Example** | The `<howto-checkbox>` element. This is further explained in Don't override the page author. |

### Always accept primitive data (strings, numbers, booleans) as either attributes or properties.

| | |
|---|---|
| **Why?** | Custom elements, like their built-in counterparts, should be configurable. Configuration can be passed in declaratively, via |

| | |
|---|---|
| | attributes, or imperatively via JavaScript properties. Ideally every attribute should also be linked to a corresponding property. |
| **Example** | The `<howto-checkbox>` element. |

### Aim to keep primitive data attributes and properties in sync, reflecting from property to attribute, and vice versa.

| | |
|---|---|
| **Why?** | You never know how a user will interact with your element. They might set a property in JavaScript, and then expect to read that value using an API like `getAttribute()`. If every attribute has a corresponding property, and both of them reflect, it will make it easier for users to work with your element. In other words, calling `setAttribute('foo', value)` should also set a corresponding `foo` property and vice versa. There are, of course, exceptions to this rule. You shouldn't reflect high frequency properties, e.g. `currentTime` in a video player. Use your best judgment. If it seems like a user will interact with a property or attribute, and it's not burdensome to reflect it, then do so. |
| **Example** | The `<howto-checkbox>` element. This is further explained in Avoid reentrancy issues. |

### Aim to only accept rich data (objects, arrays) as properties.

| | |
|---|---|
| **Why?** | Generally speaking, there are no examples of built-in HTML elements that accept rich data (plain JavaScript objects and arrays) through their attributes. Rich data is instead accepted either through method calls or properties. There are a couple obvious downsides to accepting rich data as attributes: it can be expensive to serialize a large object to a string, and any object references will be lost in this stringification process. For example, if you stringify an object which has a reference to another object, or perhaps a DOM node, those references will be lost. |

### Do not reflect rich data properties to attributes.

| | |
|---|---|
| **Why?** | Reflecting rich data properties to attributes is needlessly expensive, requiring serializing and deserializing the same JavaScript objects. Unless you have a use case that can only be solved with this feature, it's probably best to avoid it. |

### Consider checking for properties that may have been set before the element upgraded.

| | |
|---|---|
| **Why?** | A developer using your element may attempt to set a property on the element before its definition has been loaded. This is especially true if |

| | |
|---|---|
| | the developer is using a framework which handles loading components, stamping them to the page, and binding their properties to a model. |
| **Example** | The `<howto-checkbox>` element. Further explained in Make properties lazy. |

### Do not self-apply classes.

| | |
|---|---|
| **Why?** | Elements that need to express their state should do so using attributes. The `class` attribute is generally considered to be owned by the developer using your element, and writing to it yourself may inadvertently stomp on developer classes. |

## Events

### Dispatch events in response to internal component activity.

| | |
|---|---|
| **Why?** | Your component may have properties that change in response to activity that only your component knows about, for example, if a timer or animation completes, or a resource finishes loading. It's helpful to dispatch events in response to these changes to notify the host that the component's state is different. |

### Do not dispatch events in response to the host setting a property (downward data flow).

| | |
|---|---|
| **Why?** | Dispatching an event in response to a host setting a property is superfluous (the host knows the current state because it just set it). Dispatching events in response to a host setting a property may cause infinite loops with data binding systems. |
| **Example** | The `<howto-checkbox>` element. |

# Explainers

## Don't override the page author

It's possible that a developer using your element might want to override some of its initial state. For example, changing its ARIA `role` or focusability with `tabindex`. Check to see if these and any other global attributes have been set, before applying your own values.

```
connectedCallback() {
  if (!this.hasAttribute('role'))
    this.setAttribute('role', 'checkbox');
  if (!this.hasAttribute('tabindex'))
    this.setAttribute('tabindex', 0);
```

## Make properties lazy

A developer might attempt to set a property on your element before its definition has been loaded. This is especially true if the developer is using a framework which handles loading components, inserting them into to the page, and binding their properties to a model.

In the following example, Angular is declaratively binding its model's **`isChecked`** property to the checkbox's **`checked`** property. If the definition for howto-checkbox was lazy loaded it's possible that Angular might attempt to set the checked property before the element has upgraded.

```
<howto-checkbox [checked]="defaults.isChecked"></howto-checkbox>
```

A custom element should handle this scenario by checking if any properties have already been set on its instance. The **`<howto-checkbox>`** demonstrates this pattern using a method called **`_upgradeProperty()`**.

```
connectedCallback() {
  ...
  this._upgradeProperty('checked');
}

_upgradeProperty(prop) {
  if (this.hasOwnProperty(prop)) {
    let value = this[prop];
    delete this[prop];
    this[prop] = value;
  }
}
```

**`_upgradeProperty()`** captures the value from the unupgraded instance and deletes the property so it does not shadow the custom element's own property setter. This way, when the element's definition does finally load, it can immediately reflect the correct state.

## Avoid reentrancy issues

It's tempting to use the **`attributeChangedCallback()`** to reflect state to an underlying property, for example:

```
// When the [checked] attribute changes, set the checked property to match.
attributeChangedCallback(name, oldValue, newValue) {
  if (name === 'checked')
    this.checked = newValue;
}
```

But this can create an infinite loop if the property setter also reflects to the attribute.

```
set checked(value) {
  const isChecked = Boolean(value);
  if (isChecked)
    // OOPS! This will cause an infinite loop because it triggers the
    // attributeChangedCallback() which then sets this property again.
    this.setAttribute('checked', '');
  else
    this.removeAttribute('checked');
}
```

An alternative is to allow the property setter to reflect to the attribute, and have the getter determine its value based on the attribute.

```
set checked(value) {
  const isChecked = Boolean(value);
  if (isChecked)
    this.setAttribute('checked', '');
  else
    this.removeAttribute('checked');
}

get checked() {
  return this.hasAttribute('checked');
}
```

In this example, adding or removing the attribute will also set the property.

Finally, the **`attributeChangedCallback()`** can be used to handle side effects like applying ARIA states.

```
attributeChangedCallback(name, oldValue, newValue) {
  const hasValue = newValue !== null;
  switch (name) {
    case 'checked':
      // Note the attributeChangedCallback is only handling the *side effects*
```

```
      // of setting the attribute.
      this.setAttribute('aria-checked', hasValue);
      break;
    ...
  }
}
```

---

*Except as otherwise noted, the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code samples are licensed under the Apache 2.0 License. For details, see our Site Policies. Java is a registered trademark of Oracle and/or its affiliates.*

*Last updated September 26, 2017.*