HowTo: Components – howto-checkbox

By Ewa Gasperowicz

Ewa is a contributor to Web**Fundamentals**

By Rob Dodson

Rob is a contributor to Web**Fundamentals**

By Surma

Surma is a contributor to Web**Fundamentals**

Summary

A `<howto-checkbox` represents a boolean option in a form. The most common type of checkbox is a dual-type which allows the user to toggle between two choices – checked and unchecked.

The element attempts to self apply the attributes `role="checkbox"` and `tabindex="0"` when it is first created. The `role` attribute helps assistive technology like a screen reader tell the user what kind of control this is. The `tabindex` attribute opts the element into the tab order, making it keyboard focusable and operable. To learn more about these two topics, check out What can ARIA do? and Using tabindex.

When the checkbox is checked, it adds a `checked` boolean attribute, and sets a corresponding `checked` property to `true`. In addition, the element sets an `aria-checked` attribute to either `"true"` or `"false"`, depending on its state. Clicking on the checkbox with a mouse, or space bar, toggles these checked states.

The checkbox also supports a `disabled` state. If either the `disabled` property is set to true or the `disabled` attribute is applied, the checkbox sets `aria-disabled="true"`, removes the `tabindex` attribute, and returns focus to the document if the checkbox is the current `activeElement`.

The checkbox is paired with a `howto-label` element to ensure it has an accessible name.

Warning: Just because you *can* build a custom element checkbox, doesn't necessarily mean that you *should*. As this example shows, you will need to add your own keyboard, labeling, and ARIA support. It's also important to note that the native `<form>` element will NOT submit values from a custom element. You will need to wire that up yourself using AJAX or a hidden `<input>` field. For these reasons it can often be preferable to use the built-in `<input type="checkbox">` instead.

Reference

- HowTo: Components on GitHub
- Checkbox Pattern in ARIA Authoring Practices 1.1
- What can ARIA Do?
- Using tabindex

Demo

View live demo on GitHub

Example usage

```
<style>
  howto-checkbox {
    vertical-align: middle;
  }
  howto-label {
    vertical-align: middle;
    display: inline-block;
    font-weight: bold;
    font-family: sans-serif;
    font-size: 20px;
    margin-left: 8px;
  }
</style>

<howto-checkbox id="join-checkbox"></howto-checkbox>
<howto-label for="join-checkbox">Join Newsletter</howto-label>
```

Code

```
(function() {
```

Define key codes to help with
handling keyboard events.

```
const KEYCODE = {
```

```
  SPACE: 32,
};
```

Cloning contents from a <template>
element is more performant than
using innerHTML because it avoids
additional HTML parse costs.

```
const template = document.createElement('template');
```

```
template.innerHTML = `
  <style>
    :host {
      display: inline-block;
      background: url('../images/unchecked-checkbox.svg') no-repeat;
      background-size: contain;
      width: 24px;
      height: 24px;
    }
    :host([hidden]) {
      display: none;
    }
    :host([checked]) {
      background: url('../images/checked-checkbox.svg') no-repeat;
      background-size: contain;
    }
    :host([disabled]) {
      background:
        url('../images/unchecked-checkbox-disabled.svg') no-repeat;
      background-size: contain;
    }
    :host([checked][disabled]) {
      background:
        url('../images/checked-checkbox-disabled.svg') no-repeat;
      background-size: contain;
    }
  </style>
`;
```

```
class HowToCheckbox extends HTMLElement {
  static get observedAttributes() {
    return ['checked', 'disabled'];
  }
```

The element's constructor is run anytime a new instance is created. Instances are created either by parsing HTML, calling document.createElement('howto-checkbox'), or calling new HowToCheckbox(); The constructor is a good place to create shadow DOM, though you should avoid touching any attributes or light DOM children as they may not be available yet.

```
  constructor() {
```

```
    super();
    this.attachShadow({mode: 'open'});
    this.shadowRoot.appendChild(template.content.cloneNode(true));
  }
```

`connectedCallback()` fires when the element is inserted into the DOM. It's a good place to set the initial **`role`**, **`tabindex`**, internal state, and install event listeners.

```
  connectedCallback() {
```

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

A user may set a property on an *instance* of an element, before its prototype has been connected to this class. The **`_upgradeProperty()`** method will check for any instance properties and run them through the proper class setters. See the lazy properties section for more details.

```
      this._upgradeProperty('checked');
      this._upgradeProperty('disabled');
```

```
      this.addEventListener('keyup', this._onKeyUp);
      this.addEventListener('click', this._onClick);
    }
```

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

`disconnectedCallback()` fires when the element is removed from the DOM. It's a good place to do clean up work like releasing references and removing event listeners.

```
    disconnectedCallback() {
```

```
      this.removeEventListener('keyup', this._onKeyUp);
      this.removeEventListener('click', this._onClick);
    }
```

Properties and their corresponding

attributes should mirror one another. The property setter for **checked** handles truthy/falsy values and reflects those to the state of the attribute. See the avoid reentrancy section for more details.

```
set checked(value) {
```

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

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

  set disabled(value) {
    const isDisabled = Boolean(value);
    if (isDisabled)
      this.setAttribute('disabled', '');
    else
      this.removeAttribute('disabled');
  }

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

`attributeChangedCallback()` is called when any of the attributes in the **`observedAttributes`** array are changed. It's a good place to handle side effects, like setting ARIA attributes.

```
attributeChangedCallback(name, oldValue, newValue) {
```

```
    const hasValue = newValue !== null;
    switch (name) {
      case 'checked':
        this.setAttribute('aria-checked', hasValue);
        break;
        case 'disabled':
          this.setAttribute('aria-disabled', hasValue);
```

The **`tabindex`** attribute does not provide a way to fully remove focusability from an element. Elements with **`tabindex=-1`** can still be focused with a mouse or by calling **`focus()`**. To make sure an element is disabled and not focusable, remove the **`tabindex`** attribute.

```
          if (hasValue) {
            this.removeAttribute('tabindex');
```

If the focus is currently on this element, unfocus it by calling the **`HTMLElement.blur()`** method.

```
            this.blur();
          } else {
            this.setAttribute('tabindex', '0');
          }
          break;
      }
    }

  _onKeyUp(event) {
```

Don’t handle modifier shortcuts typically used by assistive technology.

```
    if (event.altKey)
      return;

    switch (event.keyCode) {
      case KEYCODE.SPACE:
        event.preventDefault();
        this._toggleChecked();
        break;
```

Any other key press is ignored and passed back to the browser.

```
        default:
          return;
      }
    }

    _onClick(event) {
      this._toggleChecked();
    }
```

`_toggleChecked()` calls the `checked` setter and flips its state. Because `_toggleChecked()` is only caused by a user action, it will also dispatch a change event. This event bubbles in order to mimic the native behavior of `<input type=checkbox>`.

```
    _toggleChecked() {
```

```
      if (this.disabled)
        return;
      this.checked = !this.checked;
      this.dispatchEvent(new CustomEvent('change', {
        detail: {
          checked: this.checked,
        },
        bubbles: true,
      }));
    }
  }

  window.customElements.define('howto-checkbox', HowToCheckbox);
})();
```