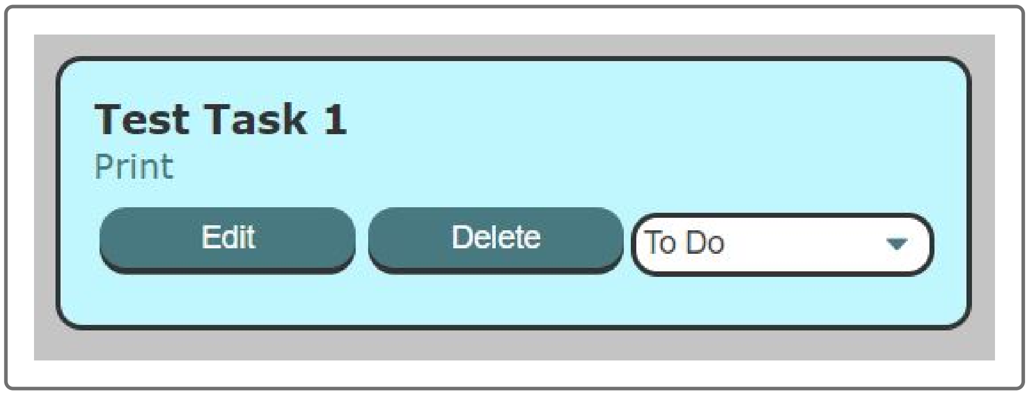
4.3.6 Dynamically Create Buttons

Now that we have an ID for each task, we can start adding buttons and dropdowns that reference these IDs. Remember, each task will have its own set of form elements that perform different actions:



Because the tasks themselves are dynamically created in JavaScript, these form elements will also need to be dynamically created. It will take several lines of code to make all of these elements, so let's offset that logic into its own function.

Add this function beneath `createTaskEl()`:

```
var createTaskActions = function(taskId) {

};
```

Note the parameter called `taskId`. This is how we can pass a different ID into the function each time to keep track of which elements we're creating for which task.

In the `createTaskActions()` function, add these lines to create a new `<div>` element:

```
var actionContainerEl = document.createElement("div");
actionContainerEl.className = "task-actions";
```

This `<div>` will act as a container for the other elements.

Underneath these lines, create two new `<button>` elements and append them to the `<div>`:

```
// create edit button
var editButtonEl = document.createElement("button");
editButtonEl.textContent = "Edit";
editButtonEl.className = "btn edit-btn";
editButtonEl.setAttribute("data-task-id", taskId);

actionContainerEl.appendChild(editButtonEl);

// create delete button
var deleteButtonEl = document.createElement("button");
deleteButtonEl.textContent = "Delete";
deleteButtonEl.className = "btn delete-btn";
deleteButtonEl.setAttribute("data-task-id", taskId);

actionContainerEl.appendChild(deleteButtonEl);
```

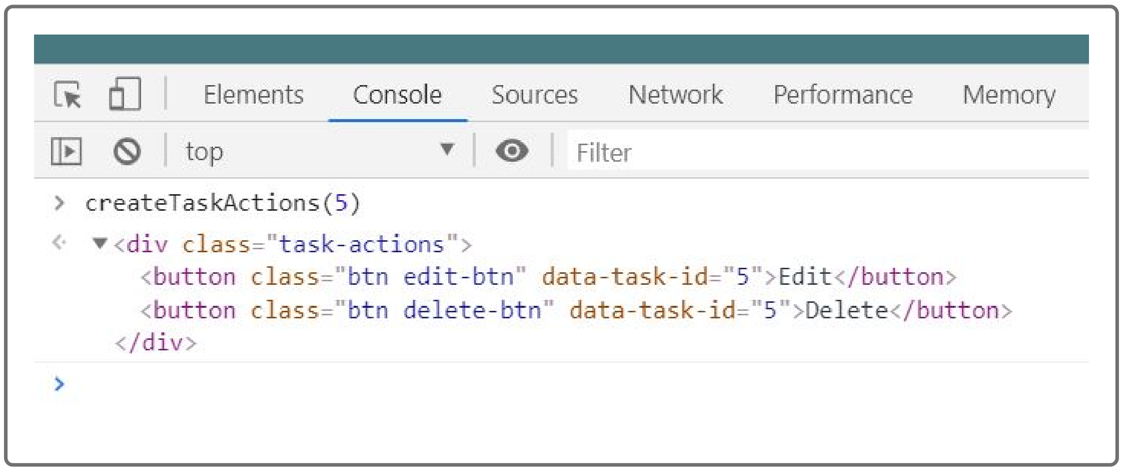
Note that `textContent`, `className`, and `setAttribute()` are properties and methods of the `<button>` elements. The `appendChild()` method will then add this `<button>` to the `<div>`.

These elements still only exist in memory, though, so nothing has changed on the page yet. It might seem like it's too soon to be able to test if this function is working, but we can at least verify that it returns the correct data.

Add a `return` statement to the bottom of the `createTaskActions()` function:

```
return actionContainerEl;
```

Here's the fun part: we're going to run this function directly in the browser to test it out. Open Chrome DevTools and navigate to the Console panel. In the console, type `createTaskActions(5)` and press Enter. The results should look like this:



```
> createTaskActions(5)
< ▼<div class="task-actions">
     <button class="btn edit-btn" data-task-id="5">Edit</button>
     <button class="btn delete-btn" data-task-id="5">Delete</button>
   </div>
>
```

In this example, we ran our function, passing in the number 5 to act as a fake ID. The function did what it was supposed to, however, and returned a `<div>` element with two `<button>` elements inside. The buttons also have the correct `class` and `data-task-id` attributes. Looks like everything is good to go!

HIDE PRO TIP

Any function that you create in the global scope can be accessed by the Chrome DevTools console. This is a great way to test if functions are working rather than trying to find a place in your code to call it.

The last thing we're missing in `createTaskActions()` is the dropdown, or to be exact, the `<select>` element. After the delete button is appended but before the `return` statement, add the following block of code:

```
var statusSelectEl = document.createElement("select");
statusSelectEl.className = "select-status";
statusSelectEl.setAttribute("name", "status-change");
statusSelectEl.setAttribute("data-task-id", taskId);

actionContainerEl.appendChild(statusSelectEl);
```

This will add an empty `<select>` element to the `<div>` container, but looking at our HTML file again, we know that `<select>` elements are made up of child `<option>` elements. We'll need to add three options to this dropdown: To Do, In Progress, and Completed. We could create and add these one after the other, but we'd end up with some very similar-looking code.

PAUSE

What JavaScript feature could you use to execute a block of code a certain number of times?

`for` loops

Hide Answer

While a `for` loop isn't required to make these `<option>` elements, any chance to make the code more DRY is always welcome. To facilitate this looping, create the following array after the `statusSelectEl` expressions:

```
var statusChoices = ["To Do", "In Progress", "Completed"];
```

Using an array also provides the benefit of being able to easily add new options later on without changing much code. Another win for the DRY principle!

After the array declaration, write the following `for` loop logic:

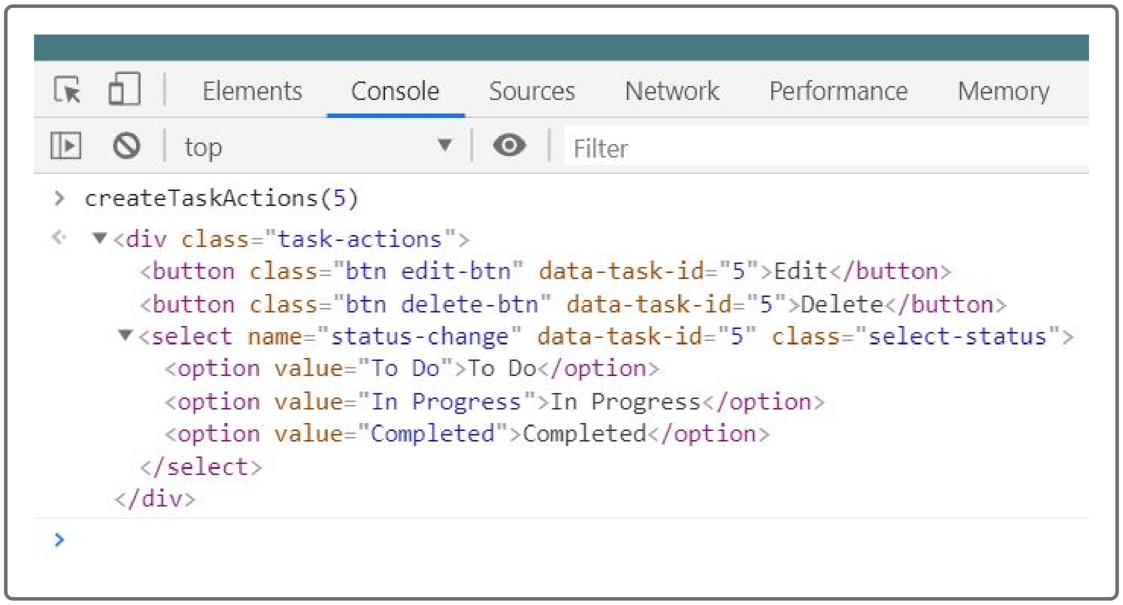
```
for (var i = 0; i < statusChoices.length; i++) {
  // create option element
  var statusOptionEl = document.createElement("option");
  statusOptionEl.textContent = statusChoices[i];
  statusOptionEl.setAttribute("value", statusChoices[i]);

  // append to select
  statusSelectEl.appendChild(statusOptionEl);
}
```

REWIND

There are several pieces in that `for` loop that might warrant a refresher. Let's break it down:

- `var i = 0` defines an initial counter, or iterator, variable
- `i < statusChoices.length` keeps the `for` loop running by checking the iterator against the number of items in the array (`length` being the property that returns the number of items)
- `i++` increments the counter by one after each loop iteration
- `statusChoices[i]` returns the value of the array at the given index (e.g., when `i = 0`, or `statusChoices[0]`, we get the first item)

Try running the `createTaskActions()` function in the console again to verify that the `<select>` element is being created correctly:



```
> createTaskActions(5)
<· ▼<div class="task-actions">
      <button class="btn edit-btn" data-task-id="5">Edit</button>
      <button class="btn delete-btn" data-task-id="5">Delete</button>
    ▼<select name="status-change" data-task-id="5" class="select-status">
        <option value="To Do">To Do</option>
        <option value="In Progress">In Progress</option>
        <option value="Completed">Completed</option>
      </select>
    </div>
>
```

Now that we've verified that the function works, add the following lines to the `createTaskEl()` function, right before the `tasksToDoEl.appendChild()` method:

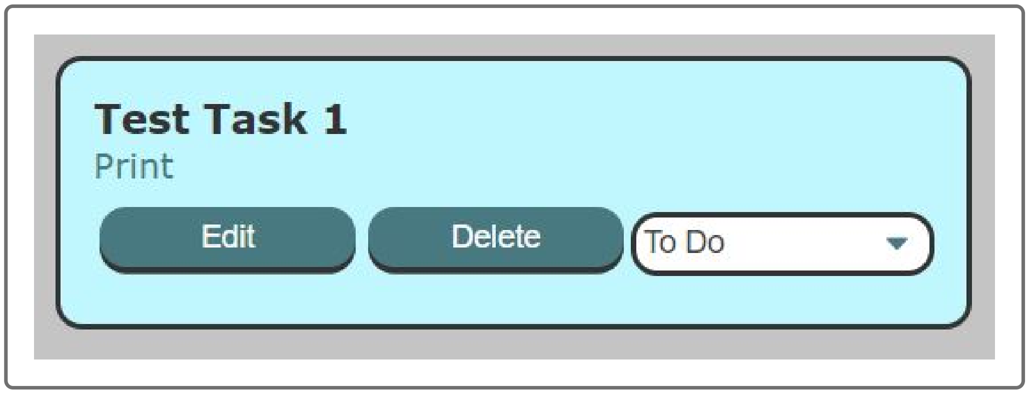
```
var taskActionsEl = createTaskActions(taskIdCounter);
console.log(taskActionsEl);
```

Note that we're using `taskIdCounter` as the argument now to create buttons that correspond to the current task ID. Because `createTaskActions()` returns a DOM element, we can store that element in a variable (`taskActionsEl`). Console logging the variable is another way to quickly verify that the function is working in this new context. Save your work and refresh the browser, then run the program to see if the element is logged to the console when you submit the form.

If the `console.log()` statement looks good, we can remove it and finalize our function by appending `taskActionsEl` to `listItemEl` before `listItemEl` is appended to the page:

```
var taskActionsEl = createTaskActions(taskIdCounter);
listItemEl.appendChild(taskActionsEl);

tasksToDoEl.appendChild(listItemEl);
```

Save `script.js`, refresh the browser, and make a few tasks. The tasks should now look like this:



If not, check the Chrome DevTools console for errors or use the script debugger to verify that functions and methods are being called in the right order. When you're happy with the results, save your progress with Git!