

## 4.2.7 Organize Functionality

The application works like we want it to. But with the current `createTaskHandler()` function, we'd struggle to add more content to a task item or the form for new features. In this step, we'll split the tasks that function is performing.

Sometimes having more functions that each perform one task is better than combining all tasks into one function. We may want to do this for many reasons, like one of the following:

- Functions with a lot of code performing separate tasks could become difficult to read and understand.
- Setting up a function to do multiple tasks, like getting the form values and then printing them to the page, lowers the potential to reuse the function for another part of the application.

Both reasons above can be hard to diagnose in the moment. That will get easier as you become more adept at predicting what the application may need in the future.

### ON THE JOB

The primary goal is to get the code working first, to avoid having to rethink the steps.

Once the code works, come up with what-if scenarios for how the code could be better or worse—or how someone could accidentally break the application.

The code will typically be imperfect upon the first attempt, and it would be unreasonable to expect otherwise. That's why we usually do a code refactor after we get it working.

Let's take a moment and outline what we'll do to refactor the code:

1. Rename the handler function to be a little more specific to the event it's handling.
2. Create a new function to take in the task's name and title as arguments and create the HTML elements that get added to the page.
3. Move the code that creates and adds HTML elements from the handler function into the newly created function.
4. Update the handler function to send the task name and type values from the form to the newly created function.

We're mostly just reorganizing code that we've already written, so a lot of work has already been done. We just need to get the code to its proper place.

We'll start by updating the name of the handler function. In `script.js` change the `createTaskHandler` function name to `taskFormHandler` wherever the name is used: in the function's variable declaration and in the `addEventListener()` method at the bottom of the file.

If we update the name in both places, the application should still work, but we should test it before we move on, just in case. Save `script.js` and test the code by submitting the form.

If it still works, great! If it doesn't, double-check that nothing was spelled incorrectly. If the name of the handler function was all that changed before it stopped working, then there may be a typo or misspelling somewhere.

### HIDE HINT

Don't forget to use the Chrome DevTools console tab to see if any errors show up.

Next we'll create a new function. We'll add the following code right below the `taskFormHandler()` function and above the `addEventListener()` method:

```
var createTaskEl = function(taskDataObj) {  
  
}
```

We just created a new function called `createTaskEl`. As the name indicates, it will hold the code that creates a new task HTML element. But if we're going to provide this function with both the task's title and type, why is there only one parameter, called `taskDataObj`?

We could set up the function to have two parameters, one for each piece of data. That may limit us in the future, though, as we may end up using more information for each task. So instead of having to add another parameter to the function every time we want to use more data, we can just set up the function to accept an object as an argument.

This way, when we send the task's name and type to the `createTaskEl()` function, it'll look like this:

```
// taskDataObj
{
  name: "Task's name",
  type: "Task's type"
}
```

Before we worry about passing the argument object, let's get the code for `createTaskEl()` in place. We could rewrite all of the code by hand, but it's already in the `taskFormHandler()` function, so we can just copy and paste it into `createTaskEl()`.

Select the following code from `taskFormHandler()`. Cut or copy it, then paste it into `createTaskEl()`:

```
// create list item
var listItemEl = document.createElement("li");
listItemEl.className = "task-item";

// create div to hold task info and add to list item
var taskInfoEl = document.createElement("div");
taskInfoEl.className = "task-info";
taskInfoEl.innerHTML = "<h3 class='task-name'>" + taskNameInput + "</h3>";

listItemEl.appendChild(taskInfoEl);

// add entire list item to list
tasksToDoEl.appendChild(listItemEl);
```

Once it's pasted into `createTaskEl()`, make sure to remove that code from `taskFormHandler()`.

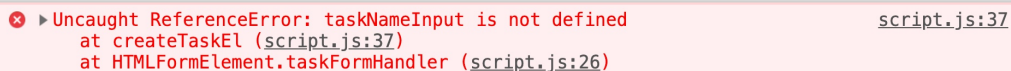
Unfortunately we can't test the code yet, as we haven't updated `taskFormHandler()` to pass the data as arguments. Let's do that now by adding the following code to that function, to look like this:

```
var taskFormHandler = function(event) {  
  event.preventDefault();  
  var taskNameInput = document.querySelector("input[name='task-name']")  
  var taskTypeInput = document.querySelector("select[name='task-type']")  
  
  // package up data as an object  
  var taskDataObj = {  
    name: taskNameInput,  
    type: taskTypeInput  
  };  
  
  // send it as an argument to createTaskEl  
  createTaskEl(taskDataObj);  
}
```

We gathered the form's values and placed them into an object with a `name` and `title` property. Then we inserted it as an argument when we called `createTaskEl()` at the bottom of `taskFormHandler()`. Doing this makes `taskFormHandler()` a lot easier to read, as it shows that it's collecting data and sending it elsewhere.

Lastly, we need to update the code in `createTaskEl()` to stop looking for the `taskNameInput` and `taskTypeInput` variables and start looking for the `taskDataObj` argument's properties instead. If we kept the two variable names there, the application would break when we attempted to submit a form.

The error would look something like this:



```
✖ Uncaught ReferenceError: taskNameInput is not defined script.js:37  
    at createTaskEl (script.js:37)  
    at HTMLFormElement.taskFormHandler (script.js:26)
```

The error states that `taskNameInput` isn't defined; how could this be if it was clearly created in the `taskFormHandler()` function? Remember that any variables created within the curly braces of a function only exist within that function's braces. Any reference to it outside the function will cause

the program to break because it can't find a variable with that name. In this case, since the initialization of `taskNameInput` and `taskTypeInput` are now in `taskFormHandler()`, we don't have access to them in `createTaskEl` except by way of the object we've passed into it.

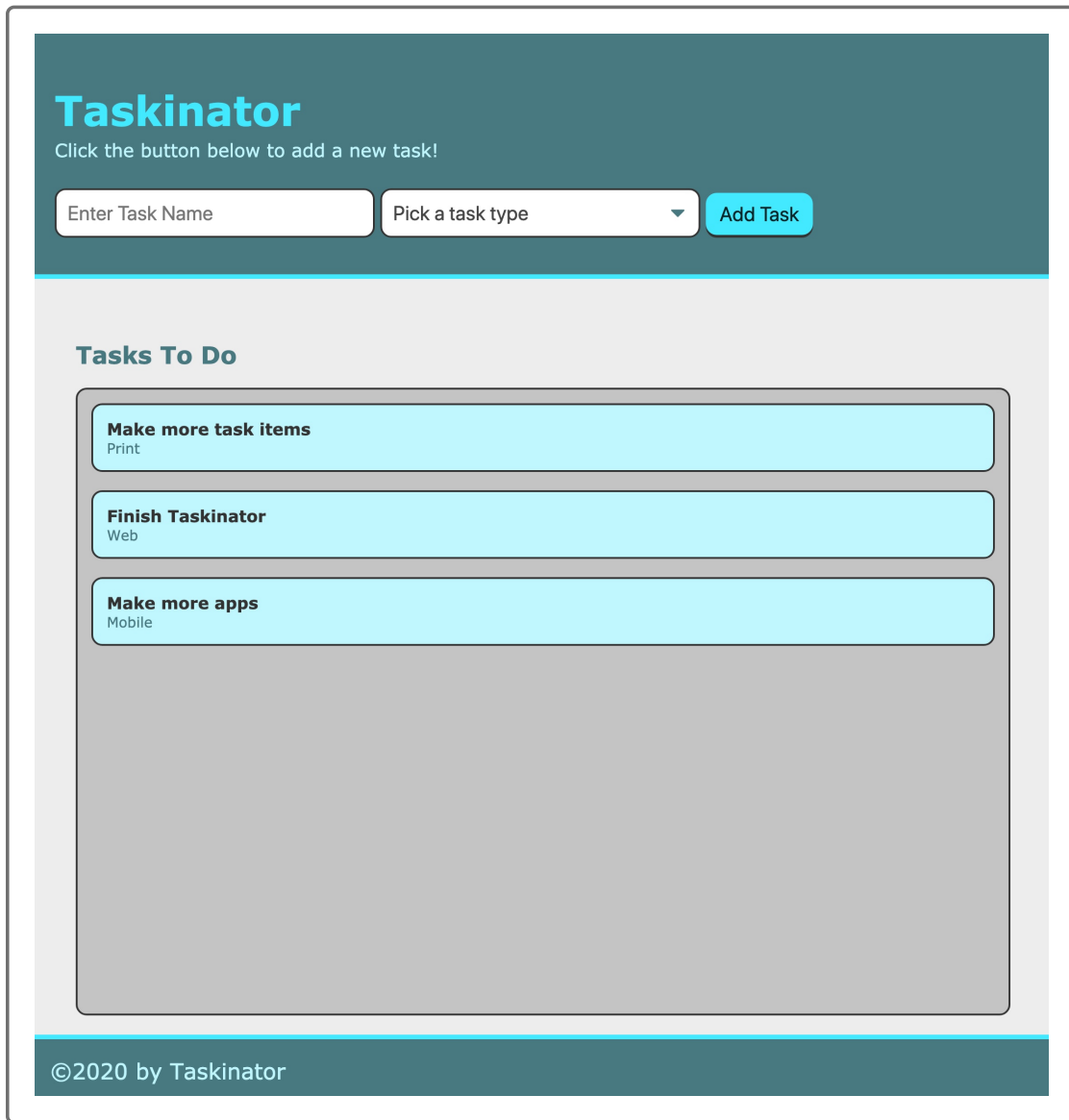
## NERD NOTE

This is known as **lexical scoping**.

Let's fix the problem and update one line in the `createTaskEl()` function to look like this instead:

```
taskInfoEl.innerHTML = "<h3 class='task-name'>" + taskDataObj.name + "
```

Finally, let's save `script.js` and test out a task submission in the browser. It should look like this:



The image shows a web application mockup for 'Taskinator'. The header is a dark teal bar with the title 'Taskinator' in large, bold, light blue font. Below the title, a smaller line of text says 'Click the button below to add a new task!'. The main content area has a light gray background. At the top of this area is a form with three elements: a text input labeled 'Enter Task Name', a dropdown menu labeled 'Pick a task type', and a blue 'Add Task' button. Below the form is a section titled 'Tasks To Do' in bold dark teal font. This section contains a list of three tasks, each in a light blue box with a dark teal border. The first task is 'Make more task items' with a sub-label 'Print'. The second is 'Finish Taskinator' with a sub-label 'Web'. The third is 'Make more apps' with a sub-label 'Mobile'. Below the list is a large gray rectangular placeholder. At the bottom of the application is a dark teal footer bar with the text '©2020 by Taskinator' in white.

If it looks exactly the same as it did before the refactor, great! Remember, we didn't update what the code does on the page; we updated the code to be more maintainable later.

Again, learning to refactor code takes time. It feels unnatural at first: who wants to second-guess code they just wrote and got working? Over time, it will feel not like second-guessing but rather like a personal challenge to be a better developer.

The Taskinator application is taking shape, but as cautious developers, we have to ask ourselves, "How can this break?" We'll tackle that in the next and final step of this lesson, but first let's take a minute to celebrate future-proofing some of the code by separating functionality into two functions.

It's not an easy task to perform or appreciate so early in a development career, but we've really set up the code to be reusable for any future features.

Don't forget to add, commit, and push the code to GitHub!

© 2020 Trilogy Education Services, a 2U, Inc. brand. All Rights Reserved.