**Shopping list: From idea to user stories to pseudocode**

**Objective:** By the end of this checkpoint, you can plan an app with pseudocode.

In the previous checkpoints, you practiced DOM manipulation and event listeners with jQuery. You also took your first stab at creating a simple JavaScript app from scratch. One thing we did not discuss, however, is how to think about moving from an app idea to writing code for that app. When you're first learning to code, the way forward is not always obvious.

We also didn't discuss best practices for structuring your app. If you're like most students, for your shopping list app, you probably ended up with so-called [**spaghetti code**](https://en.wikipedia.org/wiki/Spaghetti_code), which is to say, a single document ready function that handles everything.

For a simple app like the shopping list, this is not a huge problem, but as you start to build apps with greater complexity, it becomes important to separate out your code into distinct functions that have narrow, specific responsibilities.

Over the next few checkpoints, we'll continue to explore interactive web apps with jQuery. We'll focus less on learning new jQuery methods, and instead on how to better use the tools we learned about in the previous checkpoints, and how to move from an understanding of what an app should do (stated in plain language) to actually implementing the app.

We'll build another shopping list app from scratch. Along the way you'll learn a process you can use in other projects to iteratively build out client-side apps. You'll learn about writing **user stories** (short, plain language descriptions of what your app does). You'll also be exposed to a **functional** approach to creating applications that leads to clean, maintainable code.

In this checkpoint, we discuss how to move from a high-level app idea to user stories to creating function stubs and pseudocode that you eventually implement to create a working app.

**Key Terms**

* Spaghetti code
* User stories
* Functional programming

**Versus spaghetti code**

To begin, let's establish why [**spaghetti code**](https://en.wikipedia.org/wiki/Spaghetti_code) is less than ideal.

Here's an example of one solution to the challenge in the previous checkpoint:

$(**function**(){

$('#js-shopping-list-form').submit(**function**(event) {

event.preventDefault();

**const** listItem = $('.js-shopping-list-entry').val();

$('#shopping-list-entry').val('');

$('.shopping-list').append(

`<li>

<span class="shopping-item">${listItem}</span>

<div class="shopping-item-controls">

<button class="shopping-item-toggle">

<span class="button-label">check</span>

</button>

<button class="shopping-item-delete">

<span class="button-label">delete</span>

</button>

</div>

</li>`);

});

$('.shopping-list').on('click', '.shopping-item-delete', **function**(event) {

$(**this**).closest('li').remove();

});

$('.shopping-list').on('click', '.shopping-item-toggle', **function**(event) {

$(**this**).closest('li').find('.shopping-item').toggleClass('shopping-item\_\_checked');

});

});

Now before listing out the problems with this code, it's worth appreciating the fact that *it works!*. In conjunction with the right HTML and CSS, this code will render a shopping list to the DOM and allow users to add, remove, and check/uncheck shopping list items. Chances are your solution to the shopping list challenge looked something like this example, and that's nothing to be ashamed of. At the end of the day, it's important to *ship code*, keeping in mind that pretty-but-incomplete-and-unreleased code has zero value for users.

On the other hand, it's important to develop good coding habits and a sense of what makes for good code architecture. So with no further ado, here are some of the deficiencies with this solution:

**You have to read every line of code to understand how the app works.**

The app is composed entirely of a single, anonymous document ready function that has jQuery event listeners for submit and click events that in turn call additional anonymous functions, which appear to do something to the DOM, but without inspecting the HTML, it's hard to know what they do.

As the number of contributors to and number of lines in a code base increase, it becomes essential that someone unfamiliar with the app can skim it and get a high-level sense of what it does and how it works. When you have small, well-named functions, it's possible to understand the story of what an app does without reading all the details. The function names tell you *what* is happening, while the internals of each function are implementation details, which you only need to understand if there's a specific reason to (for instance, if there's a bug related to a particular function).

**It's difficult to reason about the shopping list data**

Imagine you encounter a bug with the check/uncheck functionality in this app — perhaps the feature works only some of the time. You throw a debugger into this part of the code:

$('.shopping-list').on('click', '.shopping-item-toggle', **function**(event) {

$(**this**).closest('li').find('.shopping-item').toggleClass('shopping-item\_\_checked');

**debugger**;

});

You'd like to understand why the DOM is not displaying the "real" state of the shopping list (which is to say, you "know" the item should be checked, but for some reason, it's not). So where is the "real" state of the shopping list in this app? What should you be inspecting while in debug mode?

The problem is that the data for the shopping list itself is stored entirely in the DOM. The only way you can understand the overall state of the shopping list at any time is by inspecting each list item in the DOM. And to make matters worse, the problem you're trying to diagnose involves the DOM being *wrong* about displaying the list!

Ideally, we need to establish one part of our code for storing the underlying data for our shopping list and then have a different part of our code deal with how that data is rendered in the DOM. This will allow us to ask and answer questions about how the DOM contents correspond (or don't correspond) to the data model.

**Spaghetti code doesn't scale**

Our shopping list app is about as simple as apps come in terms of what users can do with it: they can read a shopping list, add items to it, delete them, and check and uncheck them.

Now imagine a complex app like Facebook or Gmail implemented in a single document ready function. The fact is that neither Facebook nor Gmail would be working apps if they were built with spaghetti code.

In short, spaghetti code is difficult to read, maintain, and scale, and for those reasons and more, we need a better approach to architecting our applications. That's what we'll spend the next couple checkpoints learning, starting with...

**Clearly describing your application with user stories**

Many beginners and even some experienced programmers make the mistake of immediately starting to write code at the beginning of a project. That is always a bad idea.

Before opening your text editor, it's critical to get a clear picture of what you're building. That means coming up with clear, concise statements that describe what the app you're building will do. These statements will tell you what you need to build and when a particular feature of the app is complete. They also provide a common language for speaking with non-technical stakeholders on a project.

The idea of a shopping list app may seem intuitive and obvious — users should just be able to make a darn shopping list! — but we need to drill down on that.

Here's how we could break down our shopping list:

* A shopping list should be rendered to the page
* You should be able to add items to the list
* You should be able to check items on the list
* You should be able to delete items from the list

These statements are examples of *user stories*, which are short, plain language descriptions of what a user should be able to do with an app. User stories can be written and discussed by technical and non-technical stakeholders.

Notice that these user stories don't say anything about *how* the app is to be implemented. Instead, they focus on *what* the app should do. Recall that we made a similar observation about how well-named, small functions allow a coder to quickly understand what an app does without having to look at how each function is implemented. This happy coincidence is one we can take advantage of as we start architecting our application.

**From user stories to function stubs with pseudocode**

At this point, we're ready to start writing some code, but for now it will be code that doesn't "do anything". Instead, we're going to write function stubs. These functions will have names, but inside of each one, we won't add any working code. We'll just add a description of what the code should do. We'll also go ahead and hook these functions up to a document ready function since we know we'll definitely need that.

The repl.it below contains the first step in coding up our app. Note that the CSS for this app is identical to the starter files for the original shopping list challenge. The HTML is a little different, as we'll see later in this module, but for now, you don't need to worry about that.

If you have problems accessing the embedded repl.it below, please [**try opening it in a separate browser tab by clicking here**](https://repl.it/@thinkful/jquery-shopping-list-walkthrough-1).

Have a close look at the index.js file in this repl.it. You'll see that we have 5 functions. The first 4 map directly to the user stories we listed out above

* renderShoppingList => "A shopping list should be rendered to the page"
* handleNewItemSubmit => "You should be able to add items to the list"
* handleItemCheckClicked => "You should be able to check items on the list"
* handleDeleteItemClicked() => "You should be able to delete items from the list"

The final function, handleShoppingList, will be responsible for calling all our other ones. It will be the callback function for our app's document ready.

In the main body of each function, we've added a comment that explains what the function should eventually do. These comments are written in [**pseudocode**](https://en.wikipedia.org/wiki/Pseudocode), which is a programming language-agnostic way of starting to describe implementation details that you'll ultimately write in a specific programming language.

We've also added a console log statement so we'll be able to see that these functions do in fact run when we initially wire things up, which is what we'll do next.

If you have problems accessing the embedded repl.it below, please [**try opening it in a separate browser tab by clicking here**](https://repl.it/@thinkful/jquery-shopping-list-walkthrough-2).

This repl.it is identical to the previous one with one difference: we've fully implemented the handleShoppingList function, and we've set it as the callback to the app's document ready function.

Comparing our completed version of handleShoppingList to the pseudocode version of that function, we've accomplished exactly what we set out to do.

To see our embryonic app in action, open DevTools, clear out any existing logs in the console, then click the play button in the repl.it to restart the app. You should see four console log statements, one for each of the four functions called by handleShoppingList. This confirms that everything is wired up correctly, and we're well on our way to implementing our app.

**Modeling our data**

The final step we'll take in this assignment is modeling our shopping list data. Earlier, we pointed out that one of the problems with the spaghetti code solution is that there's not a single place where the application data is stored.

What we need is a single source of truth about the state of our shopping list. We want to be able to store that data someplace, and what the user sees in the DOM should ultimately be a reflection of the current state of our data model.

Specifically, we need a way of modeling a shopping list. How can we best store data about a shopping list?

The name "shopping list" already implies that we'll probably want to use a JavaScript array. That would allow us to store multiple list items, add items, and delete them.

Each item on the list needs at least 2 attributes: the item name and whether or not it is currently checked. We say "at least 2" because there's arguably a third attribute we need: each item needs a unique identifier that we can use to choose a specific item and delete or check/uncheck it. But since we're using an array, we can use each item's array index as its id (for instance, the first item in our shopping list array will be at index 0).

Since we need to store 2 attributes for each list item, JavaScript object literals seem like a natural choice. Each list item can be represented like this:

{name: "bird seed", checked: true}

At the top of index.js file in the repl.it below, take a look at the STORE array.

If you have problems accessing the embedded repl.it below, please [**try opening it in a separate browser tab by clicking here**](https://repl.it/@thinkful/jquery-shopping-list-walkthrough-3).

As you can see, STORE is an array whose items are object literals that represent list items. We call this array "STORE" because it will serve as the single place in our app where we store our data model. Even though STORE is a constant, that does **not** mean that the shopping list array itself should not be altered — indeed, that would make our shopping list store useless. It's okay to alter the underlying array, but it's not okay to reassign the variable name to a new value:

**const** STORE = [

{name: "apples", checked: false},

{name: "oranges", checked: false},

{name: "milk", checked: true},

{name: "bread", checked: false}

];

*// okay*

STORE.push({name: "chocolate", checked: true});

*// not okay!*

STORE = {foo: 'bar'};

The remaining functions in our app will read data from STORE, add items to it, and update the objects representing individual items.

A final note about STORE. In JavaScript, complex data types (aka, objects and arrays) are *passed by reference*, not *by value*. That means that if you pass an array or object to a function as an argument, and the function mutates it, the value of the original variable outside of the function will also be mutated. Press the play button on the repl.it below to see this in action.

If you have problems accessing the embedded repl.it below, please [**try opening it in a separate browser tab by clicking here**](https://repl.it/@thinkful/pass-by-reference-example).

In this repl.it, we have 2 constants: foo, whose value is a number, and bar, whose value is an object. We then have 2 functions: doIt, which takes an argument item and sets its value to 3737, then logs that value; and doThat, which takes an argument item that's an object, and sets item.a to "crazy time!".

We log the beginning values of foo and bar, then pass foo to doIt. Although the item variable inside of doIt gets reassigned in the function body, later, when we log foo again after running the function, its value is still 2.

With bar and the doThat function, however, things are different. Originally, bar.a is set to 'b'. We run it through doThat, and inside that function item.a is set to "crazy time!". When we log the value of bar after running the function, we see that bar.a is still "crazy time!".

This is because bar is an object and gets passed by reference. When a simple variable like foo, whose value is a number, gets passed to a function, it's like a copy of the original variable is made, and that name-value relationship exists only while the function is running.

But when an object or array gets passed to a function as an argument, instead of a copy of the underlying value being made, the function parameter *points* to the original object or array in memory. So when you change that entity inside the function, it has the side effect of changing the entity outside the function.

This is an important thing to understand about JavaScript, but why do we bring it up now? Well, since we're storing our application data in an array, we need to understand that if that array gets passed as an argument and then mutated, there will be side effects.

As a consequence of this, as we build out our app, we're going to intentionally rely on mutating a global variable (STORE) inside of our functions for adding, deleting, and modifying items inside STORE, which will make these side effects explicit.

We mention this now because earlier in the course we warned against relying on global variables. That is still a good rule to follow, but at this point in your learning, we can strategically rely on mutating a global variable in order to create a clean, nicely-architected shopping list app. Later in this course, when you get to React and Redux, you'll learn about how those frameworks allow you to manipulate data state without any side effects. For now though, that's too advanced for our purposes.

**Assignment**

There is no deliverable for this checkpoint. Mark it complete and move on to the next one.