# Classes

## Overview

In this lab, you’ll refactor the “product suggestions” web application from earlier in the course so that it uses a Product class.

The Product class will hold a description, email, recommended price, estimated sales per year, and a timestamp. You will also define a method to format this information as a string and to test if a product’s description matches a regular expression.

## Source folders

* C:/JsDeepDive/Labs/Student/06-Classes
* C:/JsDeepDive/Labs/Solutions/06-Classes

## Roadmap

There are 7 exercises in this lab, of which the last two exercises are "if time permits". Here is a brief summary of the tasks you will perform in each exercise; more detailed instructions follow later:

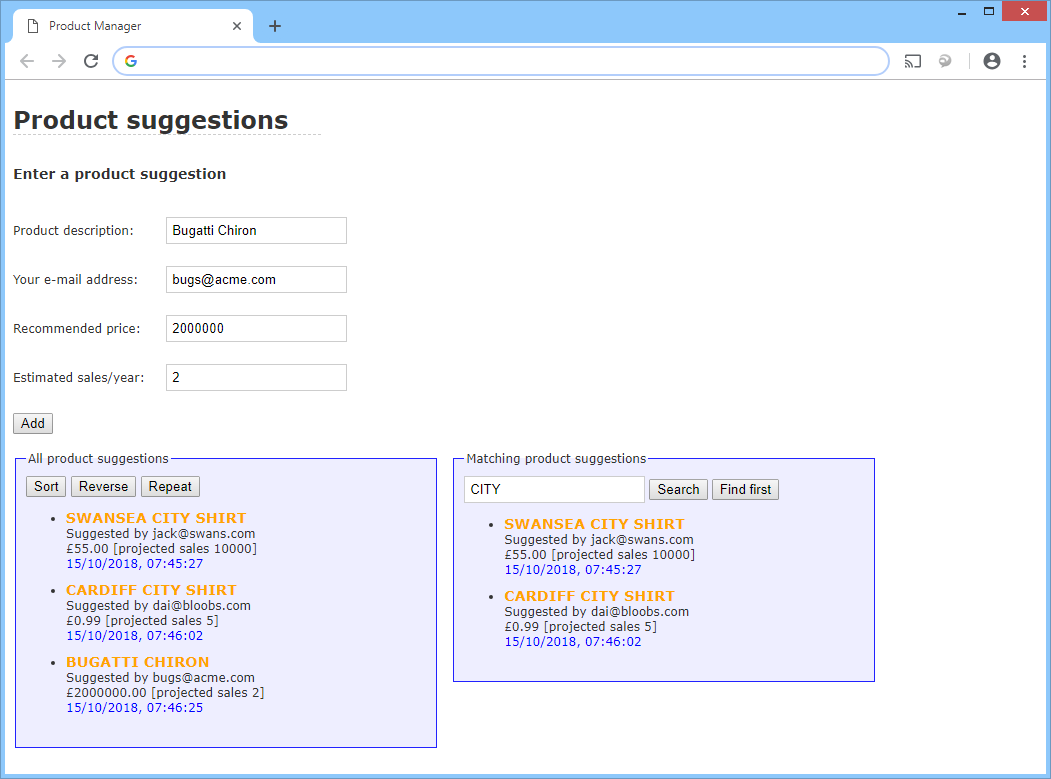
1. Familiarization
2. Getting started defining a Product class
3. Adding functionality to the Product class
4. Refactoring the Web application to use the Product class
5. Customizing the search/find algorithms
6. (If Time Permits) Customizing the sort algorithm
7. (If Time Permits) Defining and using static methods

## Exercise 1: Familiarization

Open a Command Prompt window, go to the C:/JsDeepDive folder, and run the following command to start the Babel transpiler:

npx gulp

Take a look at the files in the *student* folder, to remind yourself how it works. Run the web app and reacquaint yourself with its capabilities.



## Exercise 2: Getting started defining a Product class

Open es6scripts/script.js and define a Product class at the top of the file. Inside the Product class, define a constructor. The constructor should take 4 parameters representing the following product-related info:

* Description
* User’s email address
* Recommended price
* Estimated sales per year

Inside the constructor, assign the following properties to “this” object:

* description
* email
* price
* sales
* ts (this is the timestamp of creation – assign the current date/time)

**Exercise 3: Adding functionality to the Product class**

Now it’s time to add functionality to Product. Add a toString() method as follows:

* The method should create a formatted string that contains the product’s description, email, price, and sales (you can get most of this code from the existing doAdd() function – you might need to make some tweaks).
* The function should then append the formatted timestamp (again, you can get most of this code from the existing doAdd() function, with tweaks).
* The function should then return this concatenated text.

**Exercise 4: Refactoring the Web application to use the Product object**

In this exercise, you’ll refactor your code so that it uses the Product object.

* Refactor doAdd() so that it creates a Product object to contain the information entered by the user, and then add the Product object (rather than a formatted string) in the global allProducts array. Why is it better for the array to hold Product objects rather than formatted strings?
* Refactor displayProducts() so that it makes use of the Product object’s toString() method to return a formatted string representation of each product, ready to be displayed on the form.

Save your script file and verify Babel transpiles it successfully. Then refresh the web page in the browser. Add several product suggestions and verify the Web page displays them correctly. Also verify that the *Sort*, *Reverse, Repeat*, *Search*, and *Find* *First* buttons still seem to work correctly.

**Exercise 5: Customizing the search/find algorithms**

The *Search* and *Find first* algorithms use regular expressions to compare products against a search string entered by the user. Take a look at the doSearch() and doFindFirst() functions to remind yourself how these functions work. Both functions use a regular expression to do the test:

re.test(p)

Here, p is a Product object. When you pass an object into the test() function, it implicitly calls toString() on the object to convert it into a string. The regular expression is then evaluated against the entire string returned by toString(). In our example, the string contains the full product details (description, email, price, sales, and timestamp). To verify this, try to search for any of these parts of a product, e.g. the timestamp – the search/find operations will match on this data.

This probably isn't what you want – it would be better if the search/find operations just tested against product descriptions. Therefore, modify the regular expressions tests in doSearch() and doFindFirst() so that they just test product descriptions.

Refresh your web page in the browser, and verify the search/find operations now just look at product descriptions.

**Exercise 6 (If time permits): Customizing the sort algorithm**

The default behavior of the array sort() function is to call toString() on each object in the array, to compare items lexicographically. This means the sortfunctionality in your web application is sorting the entire string for product objects, whereas it should just sort on product descriptions.

To achieve this effect, you can supply a function to the array sort() function to tell it how to compare two Product objects explicitly. Here’s what we suggest for your sort algorithm:

allProducts.sort( (p1, p2) => {

if (p1.description < p2.description)

return -1

else if (p1.description > p2.description)

return +1

else

return 0

})

Refresh your web page in the browser, and verify the sort operation now sorts by product descriptions.

**Exercise 7 (If time permits): Defining and using static methods**

The application currently has a global function called pad(). Refactor your application so that this becomes a static method in a class named Utils.