CSCE 120: Learning To Code

Processing Data II
Hacktivity 10.2

Introduction

Prior to engaging in this hacktivity, you should have completed all of the pre-class activities as outlined in this module. At the start of class, you will be randomly assigned a partner to work with on the entirety of this hacktivity as a *peer programming* activity. Your instructor will inform you of your partner for this class.

One of you will be the driver and the other will take on the navigator role. Recall that a driver is in charge of the keyboard and computer while the navigator is in charge of the handout and directing the activity. However, you are *both* responsible for contributing and discussing solutions. If you were a driver/navigator in the prior activity, switch roles for this activity.

You will use the same project from the previous Hacktivity. If you need to, you can re-download it from GitHub using the URL, https://github.com/cbourke/AlbumApp.

1 Functional Array Functions

The `forEach()`, `map()`, `reduce()`, and `filter()` array functions all provide a way to process arrays by providing callback functions. You will get some hands-on practice using these functions by completing the following exercises.

1. Use the `map()` array function that operates on an array of strings that contain numerical values (such as `["100.5", "-1", "8"]`) and produces an array of numeric types with the same values (ex: `[100.5, -1, 8]`)
2. Recall that we can use the `split()` function to split a comma-separated value string into its *tokens*. Let's reverse that operation. Given an array of strings, we

want to "reduce" it down to a single comma-separated value string. However, some of the strings may contain commas themselves.

To distinguish commas that are part of the data and commas that are part of the data delimiter, tokens are usually surrounded by double quotes. In turn, if the tokens contain a double quote, they are usually *escaped* by adding a backslash in front of them.

For example, the array

`["Hello", "World, how are you?", "I'm feeling good."]`

would be "flattened" to

`"Hello,World, how are you?,I'm feeling good."`

Use a combination of a previous function you wrote and the `reduce` function to flatten an array of strings into a single CSV string.

3. Suppose we have an array of *mixed types*: it may contain strings, numbers, and objects. Use the `filter` function that filters the elements of an array to only include the strings (or numbers or objects). Hint: given a variable `x`, you can test what type of value it holds using the `typeof` operator. Examples:

```
var x;
...
if(typeof x === "number") {
  console.log("x is a number");
} else if(typeof x === "string") {
  console.log("x is a string");
} else if(typeof x === "object") {
  console.log("x is an object");
}
```

4. Consider the following code snippet:

```
var arr = [5, 10, 15];

arr.forEach(function(element, index, arr) {
  console.log(element);
  arr.push(element);
});
```

Without executing the code, predict what it will do. What *might* be a potential issue with this code? Execute it and see if your prediction is right. What does this tell you about how the `forEach()` function works?

2 Producer-Consumer Problem

Recall that queues are data structures that hold elements in a first-in first-out (FIFO) order. You can use an array to act as a queue by inserting elements at one end of the array and removing them from the other.

Queues are often used in a problem known as the *Consumer-Producer* problem. This problem involves (many) independent *producer* agents (threads, processes, users, etc.) that generate data or requests that must be processed or handled by *consumer* agents. Since each producer and consumer are independent, they act asynchronously. To facilitate interaction between producers and consumers, we can establish a shared *queue* data structure. Producers enqueue requests that do not require immediate processing and are free to continue in their own execution. Consumers dequeue and process requests as they are able to, enabling asynchronous and independent processing.

As an example, consider the following scenario. On a website, many different users may be submitting data through a web form. The data posts to a server which processes the data, writes to a database, updates a log, and sends an email notification to the user and serves a response page back to the user. If this process were synchronous, each user would have to wait for the previous user's submission to complete before it could submit their data. This would mean that each user would have to wait (a possibly long time) to interact with the web site. Obviously, this is a terrible UX.

In this activity, we'll complete and experiment with a consumer-producer app.

1. Open the files in the `ConsumerProducer` folder.
2. We have provided two functions that can be called with the two buttons, one to "produce" a request and one to "consume" the request. Most of the code has been provided. However, you need to complete it by properly enqueing/dequeing the randomly generated request IDs. Test your code using the buttons. Does it do FIFO ordering properly?
3. Once the two functions are working, simulate consumer and producer "threads" by setting up intervals for the two functions (do so inside a jQuery `ready()` function so that they start when the page loads). The length of the interval for the consumers will simulate how long it takes for the request to be serviced while the length of the interval for the producer will simulate the (average) interval between requests.

   a) Set the produce function to a 1 second interval and the consume to a 2 second interval and observe the results.

   b) Set the produce function to a 2 second interval and the consume to a 1 second interval and observe the results.

   c) Set the producer/consumer to an "equilibrium state" by setting them to very close intervals

   Discuss the results with your partner.

4. Simulate a multithreaded environment by setting up *multiple* consumers and producers (set multiple intervals). Experiment with various intervals on each and different numbers of consumers/producers and observe the results. In general, to ensure an *equilibrium* state, what should be done?

3 Advanced Activity: Album App

You do not need to complete this activity for credit. You may complete this activity as a personal challenge, but it is optional.

We've done plenty of exercises where we take data an present it to a user. For example, taking CSV or JSON data and populating a table. We will now do an exercise in which we *reverse* the process.

We have provided a couple of starter files for an Album Collection App (see `albums`). The idea is that a user can enter album information and add it to the table. They can also remove albums from the table.

3.1 Parsing The Table

Complete the `tableToObjects()` function which will traverse the table of albums, pull the data for each row, and create album objects. It should return an array containing album objects, one for each row in the table.

Be sure to rigorously test your function.

3.2 Saving The Data

Persistence is a characteristic of data that outlives the program that produces or processes it. Some applications use a Relation Database Management System to persist data, which is beyond the scope of this module. However, HTML5 does define a mechanism called *local storage* that allows an application to save data that is persisted beyond the execution of the browser. That is, the data is saved and available even if a user quits their browser, shuts down the computer, etc.

Local storage works on key-value pairs; both of which must be strings. To save data with local storage you use the following:

```
localStorage.setItem("key", "value");
```

Then, to load data, you use

```
var value = localStorage.setItem("key");
```

Since both key and value must be strings, if you want to store more complex data such

as numbers, objects, or arrays, you need to "stringify" them. The obvious format to stringify JavaScript objects is to convert them to JSON format. That is exactly what `JSON.stringify()` does. For example:

```
var str = JSON.stringify(["hello", 42, "goodbye"]);
```

After this code, `str` would contain a string formatted with JSON data. The reverse operation, parsing a string to a JavaScript object, can be achieved using `JSON.parse()`

We have already written the code that *loads* any saved albums from local storage. Write the function, `saveAlbums()` that saves them to local storage.