CSCE 120: Learning To Code

Organizing Code I
Hacktivity 9.2

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.

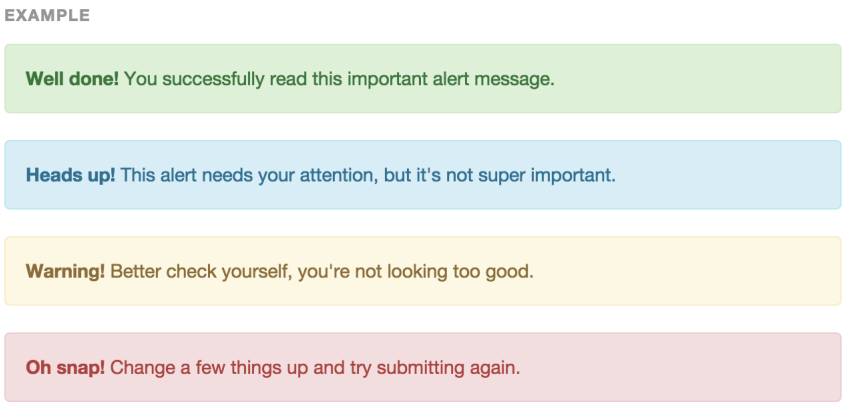
One of you will be the driver and the other will take on the navigator role. Recall that a driver is in charge of the keyboard and computer while the navigator is in charge of the handout and directing the activity. However, you are *both* responsible for contributing and discussing solutions. If you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

1 Making a Factory Function

One common use of a function is to make it easier to construct objects or other elements. Such a function is sometimes referred to as a *factory function*. In general, the parameters required to build the object are passed to the function, the function builds the object, and returns it. This allows you to reuse the function to easily build as many of the objects as you wish.

One of Bootstrap's features is styled *alert* boxes that provide users with color-coded messages. Certain events in an application may raise errors or messages that need to be displayed to a user. The code necessary to build a "dismissable" alert box can be found here: `http://getbootstrap.com/components/#alerts-dismissible`, the examples provided:



Write a function that dynamically creates alert elements (as HTML formatted strings). Your function should be general enough that you can create any type of alert box (success, info, warning, danger), and specify both the "title" (the bolded message) and content of the text box. Place your function in the `alerts/main.js` source file and test your function using the HTML page.

2 Asynchronicity

In computer programs, asynchronous processes or events are those that occur independently of the main program. Asynchronous code is executed in a "non-blocking" manner which allows the main program as well as other asynchronous code to continue processing without "freezing" the entire program.

2.1 Solving a Synchronicity Problem

1. Open/evaluate the `synchronicity/index.html` file. Click the Animate button and observe what happens. The intention is that each alert box will fade out/in one *after* the other. However, that is not what happens. Examine the code and discuss with your partner why that is.
2. Fix the code in the `animate()` function so that the first alert box fades out/in, *then* the second box does so (only after the first box's animation is done) and then the third after that. Use properly nested callbacks to do this. The documentation on the `fadeToggle()` function can be found here: http://api.jquery.com/fadetoggle/ if needed.

2.2 Using Asynchronicity

Behind the scenes, jQuery is using the JavaScript timer functions that allow you to pass in a callback and have it execute after a delay or at specified intervals. Full documentation can be found here: https://developer.mozilla.org/en-US/Add-ons/Code_snippets/Timers.

The `setTimeout()` function allows you to execute a function after a certain delay, specified in milliseconds (1000 milliseconds = 1 second). For example:

```
function sayHello() {
  console.log("Hello!");
}

setTimeout(sayHello, 3000);
```

would have the following effect: it would wait for (approximately) 3 seconds and then execute the `sayHello()` function which would then print `Hello`.

A similar function is the `setInterval()` function which allows you to repeatedly execute a function at specific intervals. For example:

```
setInterval(sayHello, 5000);
```

would setup a recurring call to `sayHello()`, executing it every 5 seconds.

JavaScript also gives us a way to *cancel* an interval (as well as a timeout). The `setTimeout()` and `setInterval()` functions return an ID number that can be used to cancel the operation. You can then use the `clearTimeout()` and `clearInterval()` functions to cancel the operations:

```
var x = setTimeout(sayHello, 100000);
var y = setInterval(sayHello, 5000);

clearTimeout(x);
clearInterval(y);
```

1. **Trying It Out** Run the code in the `countDown.js` file and observe the results. read the comments in the code to understand how it is achieved.
2. **Asynchronicity** With your partner, examine the following code:

```
function print() {
   console.log(1);
   setTimeout(function() { console.log(2); }, 1000);
```

```
    setTimeout(function() { console.log(3); }, 0);
    console.log(4);
}
```

Predict what sequence of numbers will be printed to the console if you call the function `print()`. Then run paste and run this code in Light Table. Were you correct? Discuss the results.

3. **Are You There?** Many secure webapps will include a session timeout so that if a user is inactive for a certain amount of time, they are automatically logged out for security reasons. Often users are prompted whether or not they wish to stay logged. If they answer yes, then the session timeout starts over. If they answer no or if they fail to answer, they will be logged out.

   In this exercise, we will simulate this process. Open the `stayLoggedIn.js` source file. We have provided some starter code, but you need to follow the comments and write code to complete the program.

4. **Animation Done Right** In a previous module we saw that animation is not a simple matter. We revisit that problem here. Open the files in the `animate` folder.

   We have provided two functions that, when called change the color of the two main div elements in the page. The first is progressively changed from blue to red and the second from magenta to green. The two buttons trigger the animation functions, `animate()` and `animate2()`.

   Try placing the following code at the end of the `animate()` function to see if it works. What happens and why?

```
for(var i=0; i<256; i++) {
  increment();
}
```

   Write JavaScript code to make the it animate correctly. That is, it should invoke the `animate()` function once every 50 milliseconds. Run your program to view the gradual change in color.

   You'll find that the function continues to be called even after the color change has completed. Add some additional code to cancel the interval when the animation completes.

   Do the same thing for the second div (which goes from magenta to green). Experiment with different intervals/pauses. Observe that the animation is truly asynchronous by alternately clicking the two animate buttons.