PER SCHOLAS

ALAB 308A.1.1:

Practical Use of the Event Loop

Version 1.0, 10/13/23

Click here to open in a separate window.

Introduction

This assignment will introduce some practical implementations of event loop and call stack concepts, as well as emphasize some of the common issues a JavaScript developer may run into due to the implementation of these core features.

Objectives

- Calculate the maximum size of the JavaScript call stack.
- Use “trampolines” to handle stack overflow issues caused by recursion.
- Use deferred execution within the event loop to allow the browser to render between calculations.

Submission

Submit your completed lab using the **Start Assignment** button on the assignment page in Canvas.

Your submission should include:

- A GitHub link to your completed project repository.

Instructions

Initialize a new git repository in a local project folder and create a JavaScript file (.js) to contain your code. Within your code editor of choice (we recommend Visual Studio Code), complete the following activities.

Commit frequently! Every time something works, you should commit it. Remember, you can always go back to a previous commit if something breaks. As an optional goal, create a separate branch for each of the activities and merge it into your main branch when you are finished with that activity.

Part 1: Stack Overflow

Ever wonder where the popular question-and-answer website got its name? When a program attempts to allocate too much memory to the call stack, it results in a “stack overflow” error. In

JavaScript, this error reads “maximum call stack size exceeded.”

So what is the maximum stack size, and when might this become an issue?

Since the call stack holds function calls, stack overflows are most common during recursive function calls. When recursion takes too long to complete, or never does complete, the call stack fills beyond its maximum memory capacity.

This capacity varies based on a number of factors - you will never know *exactly* how many calls you are allowed to put on the stack before it overflows.

You can, however, do the following:

- Declare a global counter variable.
- Create a simple function that increments the variable, and then calls itself recursively.
- Surround the initial function call in a try/catch block.
- Within the catch, log the error and the value of the counter variable.

As your first task, complete the steps above.

Once complete, you should be given a result that is *typically* in the ~15,000 range. This can vary significantly depending on the specifics of the environment that the code is being run in, so do not worry if your number is off by a few thousand.

What you have accomplished is a script that effectively measures the size of the call stack. You recursively put functions onto the stack until it reached overflow, then caught that error to print the count.

This demonstrates another important lesson: when using recursive functions, it is prudent to always wrap them in error-handling logic, just in case the function overflows the call stack.

Part 2: Trampolines

Using try/catch blocks does not solve the root issue of a stack overflow; it just handles the resulting error. What if you are implementing a recursive function that needs to call itself 30,000 or 50,000 times? While the use cases for that are rare, they do exist.

Without getting into the specifics of language-level fixes to this problem, which are being suggested, there is a process called “trampolining,” which eliminates that stack overflow issue if implemented correctly.

There are three steps to writing a trampolined function. An example has been provided below:

Once you have familiarized yourself with the logic behind this process, accomplish the following:

- Write a recursive function that completely flattens an array of nested arrays, regardless of how deeply nested the arrays are.
- Once your recursive function is complete, trampoline it.

Part 3: Deferred Execution

Another practical manipulation of the event loop is through "deferred execution" of tasks when working in a browser environment.

Using setTimeout, you can place tasks into the event queue. This sets the task to be executed after the current call stack is cleared and after the browser has had a moment to render.

Implement the following:

- Create a simple HTML element to hold text. Cache this HTML element into a JavaScript variable.
- Write a function that takes a parameter n and adds a list of all prime numbers between one and n to your HTML element.
- Once complete, use the alert() method to alert the user that the calculation is finished.

Once complete, run your function with n equal to 10,000. Notice how the alert appears before the numbers, and then all of the numbers appear at once? This is because we never gave the browser time to render the results.

Using what you have learned throughout this lesson, modify your function such that each number has an opportunity to render after it is calculated, and the alert only appears once all numbers have been rendered.

Hint: you may want to make use of recursion, and do not forget about setTimeout(..., 0).

How does this change the speed of the program? Are there alternative methods that may be more efficient? What would be the best implementation from a user experience perspective?

Keep these considerations in mind during future learning, as the call stack and event loop factor into everything JavaScript does.