Class Notes: 1 October CMSC122

Big Picture

We are making the transition from HTML/CSS, which are essentially “mark-up” and “description” languages to JavaScript, which is a general purpose “scripting” language. To that end, we recall that computer programs are merely written (usually written) sequences of instructions that embody “algorithms.” Our slides go into some detail about “algorithms.” Some important points to remember:

- Algorithms consist of a finite set of instructions
- Some clear method of determining success (or failure) is necessary before we can begin
- These instructions must be “realizable” or “executable” on some kind of a “machine,” and we described the “von Neumann” machine which is the basis for modern computing devices.
- Finally, the solutions (or failure states) of these algorithms must be obtainable in “reasonable time.”

The specifics are found on the slide and we will not take up much more space, other than to make the occasional reference as needed.

The Four Essential Behaviors

We observed that algorithms require four basic kinds of “moves” or “behaviors”

- Concurrency: This is the idea that a number of “instructions” may be performed in any particular order without changing the outcome. Think of brushing ones’ teeth: the order in which we brush makes no difference on the eventual success/failure of the exercise.
- Sequentiality: This is the idea that instructions sometimes must proceed in a particular order: Step 1 must complete before beginning Step 2, etc. This seems more like the common understanding people have when given a set of instructions.
- Conditionality: The next “instruction” depends upon the outcome of some “test” or the “state” of some object.
- Iteration: Simply said: this is the idea of repetition. In most algorithms, however, the repetition should come to an end … think of what happens if we never stop brushing our teeth.

The remainder of our time together will be spent, in no small part, exploring the possible kinds of problems that we can solve using these basic ingredients, but through the lens of a particular programming language.

The von Neumann Machine

We spent some time describing the “stored program model,” because we need to understand how modern computers represent problems and their solutions—what do these algorithms look like at a certain level?

In this model, “data” and “program” both reside in “memory” (or the “store”). Each datum is essentially either a 0 or a 1, which are switch states. To bring order to this very long string of bits, we need to know where (as a location) to start reading bits, where (as a location) to stop reading, and what is it we are looking for.

All of this is done for us by the machine—and it is done billions of times a second over billions of bits.

How this appears in JavaScript

We will systemically associate various JavaScript operators and concepts with the ideas that we introduced above. We begin, for simplicity’s sake, with the notion of a “variable,” which are machine locations, and the “assignment” statement, which moves values into these machine locations.

```
var someVariable; // declares the identifier to be a variable

var someVariable = 0; // same as above, but stores a 0 into the location
```

Of particular importance here is the use of the equals sign. In most languages the “equal” sign does not mean the binary relation equals (whose properties should be familiar to you). Instead, read the “=“ as in

```
aVariable = 0;
```

to mean “store” the value `0` into the location (variable) named `aVariable`, displacing whatever value may have been there.[1] Armed with this understanding, the following usage now makes sense where it certainly would not if we interpreted the equals sign as a binary relation that we have seen in mathematics. For example:

```
var myTotal = 0; // stores 0 in the location myTotal
myTotal = myTotal + 1 // stores 1 into myTotal
```

This last statement makes sense when we interpret the “=“ to mean “store” into the location named on the left the value computed on the right.

---

[1] We used the left-facing arrow symbol to show this in pseudocode, which is explained in greater depth in another class document.

This week's activity

Because we have a midterm next week, this week we had a simple activity that is designed to give you practice with assignment statements as well as using some basic arithmetic operators defined in JavaScript, and to reflect the results of the computations in HTML.

It's perhaps best to think of the "workflow" or "interaction" model here as
- "read" (or get) some data from the user,
- "evaluate" or transform that data, and
- "write" new data to the Document, i.e., in HTML.

For a variety of reasons, it's easiest to get data from the user. To do that, we use the Browser to "prompt" (ask) the user for data. Then, we use a sequence of assignment statements to store these data, store the results of some simple arithmetic performed on these data, and, finally we use the "document" model to write these results into the HTML elements themselves, thereby re-writing the original webpage to reflect our new work.

Getting data from the User

We will depend upon the Browser to handle the nuts-and-bolts of asking for input and returning that input to our script where we will then "store" it. In doing this, we will think about what kind of algorithmic "moves" we are making.

The Browser is a container that contains another container, called the "document," which contains HTML, etc. The following uses the Browser to prompt the user for data and returns that data as a String:

```
window.prompt( "Enter a positive integer: " );
```

Typing this into the script portion of your working document (the one we downloaded and read into Komodo on Wednesday) pops up a window showing that message. One big problem remains, however. When that window goes away, so does the data that the user entered. This is why we use variables:

```
var input1 = window.prompt( "Enter a positive integer: " );
```

Entering this and loading your test page seems to have the same effect, but, inspecting (using the Developers Toolkit for your Browser; go to the Console Tab, and type in the name of the variable you just created) the variable input1 should show you the value that the user typed, but as a String! We need that String to be an Integer if the arithmetic is going to be meaningful. Thus:

```
var input1 = window.prompt( "Enter a positive integer: " );
input1 = parseInt( input1 );
```

This uses “sequencing” of instructions to take the result from the first instruction, which is a String, and feed it to a function parseInt that takes valid Strings to Integers.[2]

Recalling our discussion of composition, we can compose these operations into one line of code as:

```
var input1 = parseInt( window.prompt( "Enter a positive integer: " ));
```

Convince yourself that you understand this last statement.

Now, you need to repeat this process so that you have two integers, maybe called `input1` and `input2`, before proceeding to the “evaluation” phase of the exercise.

Doing the arithmetic

Use the Tutorial (JavaScript) to find the Arithmetic operators. You’ll likely define a variable for the results of each of the required operations, sum, difference, product, etc.

[2] It is a separate matter (to be discussed later) of what happens if the user does not enter a String that names an integer.

Outputting the Results

You have two options:
1. Use the Browser to output the results in a box (use `window.alert`), or, preferably
2. Construct some HTML code in the body of the document and use the JavaScript DOM connections to write your results into the HTML in real-time.

Rather than give you the entire solution, consider some fragments.

```
<body>
   <h1>Results</h1>
   <ol>
     <li id="sum"></li>
     . . .
   </pl>
…
</body>
```

Read the entire JavaScript tutorial section marked JS Introduction. Note the use of the accessor (the method named `getElementById( "idName" )`) in the "suggested" approach outlined below:

```
document.getElementById( "sum" ).innerHTML=
     "The sum of " + value1 + " and " + value2 + " is " + sum;
```

It may help you to bear in mind when reading/writing this kind of code that the use of the dot, ".", consistently means "access the member of the thing on the left that is called the thing on the right." In

```
document.getElementById( "sum" ).innerHTML
```

this is saying that the object document has a property (or a function) named `getElementById` that also requires that we specify the name of that id, called "`sum`" in our example. The actual text that we typed above returns the "reference," (location) to that element in the HTML that has the id "`sum`" (which is a list item in our example). Now, that reference "holds" or "contains" other properties, one of them is its contained HTML text, which is referenced as `innerHTML`. Once we have that reference, we can store a new value into that location, overwriting whatever was there with our new String: "`The sum of …`".

Again: reading the Tutorial completely and trying the examples therein will help you here.