CSCE 120: Learning To Code

Manipulating Data I

Introduction

This module is designed to get you started working with data by understanding and using variables and data types in JavaScript. It will also introduce you to basic operators and familiarize you with basic syntax and coding style.

Variables

Previously, we worked with data represented using JSON. JSON is actually a subset of the programming language JavaScript. In the full JavaScript, data is not merely static, it is dynamic. That is, data can be accessed, changed, moved around etc.

The first mechanism for working with data is a *variable*. A variable in a programming language is much like a variable in algebra: it has a *name* (formally called an *identifier*) and can be *assigned* a value. This is very similar to how we worked with JSON data using key-value pairs. With variables, the variable name acts as a key while it can be assigned a value.

Variable Declaration, Naming

In JavaScript, you can create a variable using the keyword `var` and then providing a name for the variable. There are a few simple rules that must be followed when you name a variable.

1. Variable names can include letters (upper or lowercase), digits, and underscores (no spaces are allowed)
2. Variable names must begin with a letter (upper or lower case)[1]

[1] Variable names can also begin with `$` or `_` but this is discouraged as we'll see later.

3. Variable names are case sensitive so `total`, `Total` and `TOTAL` are three *different* variables
4. The JavaScript language has several *reserved words* that are used by the language itself and cannot be used for variable names
5. Variable names in the same context (same code block for example) must be unique

A complete list of reserved words can be found here: http://www.w3schools.com/js/js_reserved.asp. Some examples of variable declarations and names:

```
1 //variable declarations:
2 var total;
3 var numberOfStudents;
4 var averageGpa;
5 var index;
```

Semicolons

You'll notice in the previous example each line was ended with a semicolon. This is a common aspect of many modern "C-style syntax" programming languages. Each executable line or statement is ended with a semicolon to make it unambiguous and clear where the statement ends. In JavaScript, semicolons are not absolutely necessary in many cases. However, it is still considered good programming practice and style to include semicolons where they would traditionally be required in other languages. For more information on when semicolons are necessary and when they are not, see the following article, http://inimino.org/~inimino/blog/javascript_semicolons. In this course, we will always include semicolons.

Naming Conventions

When naming variables, its best practice to give them names which describe their purpose. The names used above were full English words or phrases that described what the variables represented. You should avoid cryptic or generic variable names as it makes your code less readable and more difficult to manage.

Moreover, we used a common modern naming convention called lower camel casing. In this convention, variable names consisting of multiple words are written together with no spaces, but the first letter of each new word is capitalized. The first word and the rest of the letters are all lower case. This convention has gained favor because of its relatively easy to both read and type.

Scoping

The *scope* of a variable is the section of code in which the variable is valid. In general, a variable is valid within the block of code in which you declare it. We'll explore this issue in greater depth later on, but it is necessary at this point to understand two issues involving scoping in JavaScript.

The first issue is that some variables have *global* scope. For example, when JavaScript is run within an HTML document, there is a `document` variable that refers to the document itself. When a variable has a global scope, it can be accessed *anywhere* in the code. Making changes to such variables is risky unless you know what you are doing.

The second issue is that failure to use the `var` keyword in front of your variables is not an error. Instead, JavaScript creates your variable in the global context, making the scope of your variable global. In general this is bad practice. Using global variables "pollutes" the name space. Variable names must be unique, creating two different variables with the same name in the global context will lead to conflicts. Though it won't be an error, it may lead to unexpected or bad results.

Comments

The final thing to notice about the above example is the first line. This is a *comment*. Software developers place comments throughout their code as a form of documentation. Notes that document why they did things a certain way or other information intended for other developers (such as copyright declarations).

In general, good code should be "self-documenting". That is, well-written code should be readable enough that an experienced developer can discern *how* the code works. Comments in code should tell a developer the *what* and they *why*. Good comments provide a summary of what certain functions or code blocks do. Good comments also inform other developers of why things were done in a certain way (to be compatible with other code, performance, etc.).

Single-line comments are denoted with two forward slashes: `//`. Anything that follows is ultimately ignored by the computer. Multi-line comments are denoted with an opening forward-slash-star: `/*` and closed with a `*/`. Anything that appears between them is a comment and is ignored. Another example:

```
/*
  John Student
  CSCE 120
  Fall 2014
*/
var total;
//used for pre-tax total:
var subTotal;
```

Operators

Now that we have variables, we need a way to manipulate the values in the variables. To do this, we use *operators* that act on variables (in this context often referred to as *operands*) and values (referred to as *literals*).

Assignment Operator

We first need a way to assign values to variables. To do so, we use the *assignment* operator which is the single equals sign. Any type of value can be assigned to a variable. The proper usage of the assignment operator is to write:

```
var variable = value;
```

This has the meaning "assign the value on the right hand side to the variable on the left hand side". Take care not to confuse the usage of a single equals sign as it is used in algebra. This is not an assertion of equality, but an *assignment*. Some examples:

```
var x;
x = 10;
var y;
y = "Hello";
var z;
z = {
  "lastName": "Student",
  "firstName":"John"
};
var myArray;
myArray = [10, 20, 30];
```

To save space, you can do a *compound* declaration-assignment in one line. You can even do multiple variable declaration-assignments on a single line using a comma as a separator.

```
var x = 10;
var y = "Hello";
var a = 5, b = "five";
```

You can also *reassign* variable values. Reassigning a variable will erase the old value and take on the new value. You do not redeclare a variable when reassigning its value.

```
var x = 10;
...
x = 20;
...
x = "hello";
```

In the previous example observe that `x` was originally assigned a number and later it was

assigned a string. Variables in JavaScript have no fixed *type*: the type (numeric, string, object, etc.). Instead, the type of a variable is whatever the type of variable has been assigned to it last. Reassigning a variable value to a different type changes the variables type. This is known as *dynamic typing* and JavaScript is a dynamically typed language. In contrast, other languages are *statically typed*. Variables in a statically typed languages such as Java and C are declared by specifying their type and only values of that type may be assigned to the variable. That is, you cannot assign a string value to a numeric type.

Arithmetic Operators

Basic mathematical expressions can be formed using the common arithmetic operators: `+, -, *, /` which are used for addition, subtraction, multiplication and division respectively. These are *binary* operators in that they are applied to two variables, literals, or expressions. In addition, more complex expressions can be formed by using parentheses, `( , )` just as you would in algebra. Also as in algebra, these operators have an order of precedence: multiplication and division are evaluated first, then addition and subtraction. Moreover, expressions are evaluated in a left-to-right manner.

```
1 var x = 10 + 20 * 30; //610
2 var y = (10 + 20) * 30; //900
3 var average = (x + y) / 2;
4 var a = 3, b = 4;
5 var c = a * a + b * b;
6 var d = (a - b) * (a + b);
```

In each of the examples above, an expression was formed on the right-hand-side of the assignment operator. The value of the result of the expression was then placed into the variable on the left-hand-side. It would be invalid to put an expression on the left-hand-side. However, you can write an expression without an assignment operator such as:

```
(a - b) * c;
```

which is evaluated, but has no effect. We've written a valid expression which is evaluated and has a value but we didn't end up doing anything with it (like placing the value into a variable).

Another useful arithmetic operator is the integer division operator, `a % b` which gives the value of the *remainder* of a divided by b. For example, `23 % 5` would have a value 3 since 23 divided by 5 has a remainder of 3.

Concatenation

When working with strings, it is often useful to combine several strings into one single string. Or, it is useful to combine a string and the value stored in another variable (say

to print it out to the user). This operation is known as *concatenation* and is achieved by using the plus symbol `+`.

Note: we previously used `+` when we wanted to perform addition. However, when used in the context of strings, `+` has the *implicit* meaning of concatenation. Consider the following:

```
var c = a + b;
```

In this line, if `a, b` are both numbers, then `+` is interpreted as addition and the result stored in `c` is a number. However, if either `a, b` are strings (or they both are), then `+` is interpreted as concatenation and the result in `c` is a string.

```
1 var message = "Hello, " + "World";
2 var name = "John";
3 var greet = "Greetings, " + name;
4 var a = 10;
5 var s = "The value stored in the variable a is " + a;
```

Dot Accessor

Recall that objects contain several key-value pairs. Given an object, we can manipulate each key-value pair by using the dot-operator, `.`. If an object is stored in a variable `a`, then we can access a value in that object by using:

```
a.key
```

Where *key* is replaced with the string of the key name you wish to access (this is why using spaces in a key is discouraged). Using the dot operator you can both assign values and access them. A full example:

```
1  var student = {
2    "firstName": "John",
3    "lastName": "Student"
4  };
5  //access the names and concatenate them:
6  var name = student.lastName + ", " + student.firstName;
7  //reassign a value to an existing key:
8  student.firstName = "Jonathan";
9  //set a new key-value pair in the object:
10 student.gpa = 3.75;
```

Note in the last line above we created a new key value pair in an existing object by simply setting it. We did not need to use the keyword `var` to create this new variable inside the object.

Advanced Tip: If you don't know what keys are part of your object `x`, you can get an array of strings containing them by calling `Object.keys(x)`.

Array Indexing

When elements are stored in an array, they can be accessed by specifying *which* element in the array you want. To do this, you must specify the *index* of the element. JavaScript, like most languages, uses *zero-indexing* to store elements meaning that the first element in the array is stored at index 0, the second is stored at index 1, etc. The last element is stored at index $n - 1$ where n is the number of elements stored in the array.

To access a single element in an array you use a combination of the array name, square brackets, and an index:

`arr[index]`

A full example:

```
var arr = [4, 7, 3, 9, 2, 1];
var first = arr[0]; //4
var last = arr[5]; //1
//reassign an existing element:
arr[3] = 42; //now arr is [4, 7, 3, 42, 2, 1]
```

We'll work with arrays in more depth later, but some other useful items:

- To determine the size (length) of an array, you can use `arr.length`
- Attempting to set a value of a negative index will transform your array into an object!
- Attempting to set a value of an index larger than the current size of the array will work. However, the elements in the array will no longer be contiguous. Indices between the new element and the previous old element are undefined (they do not exist; this is different from being `null`).
- To easily add elements to the end of an array, use the *push* operation: `arr.push(100);`

Math Library

Sometimes its necessary to use mathematical functions more complex than simple addition, multiplication, etc. Fortunately, JavaScript provides a library of mathematical functions to help you compute square roots, powers, exponentials, etc. It also defines several useful mathematical constants such as π, e, etc. These convenient functions are part of the `Math` object and can be accessed using the dot operator (we'll study functions more in depth later on). Some examples:

```
//area of a circle
var radius = 1.5;
var area = Math.PI * radius * radius;

//quadratic formula:
var x1 = (-b + Math.sqrt(b*b - 4*a*c) ) / (2*a);
var x2 = (-b - Math.sqrt(b*b - 4*a*c) ) / (2*a);

//annual percentage rate and annual percentage yield
var apr = 0.08;
var apy = Math.exp(apr) - 1;
```

A complete list of math functions and constants is available here: http://www.w3schools.com/js/js_math.asp.