Google Apps Script Overview

Alan Ponte

4/2014

© 2014 Alan Ponte
Released under CC-BY-SA. If you want to reuse it, please respect the atttribution clause by linking to the original document (emailing me would also be nice).

Many Thanks to:

- LBNL Collaborate Team.

Chapter 1: A Quick Introduction to JavaScript

Although it may not appear to be, JavaScript is an immensely powerful language. It is ”weakly typed” and as such many pass it off as not being as powerful as Java or C++. However, it allows you to do so much within its confines.

1.1: Names, types and values

JavaScript does not have ”strict” defined types. That is, you do not need to specify to the compiler what type of data you are working with. For example, in Java, if we wanted to write a function to find the average from a list of doubles, we would have to be very careful of what type of data we pass in.

```
/** Given a list L of values, returns the average of L. */
public static void getAverage(List<Double> L) {
    double avg = 0;
    double total = 0;
        for (int i = 0; i < L.length(); i += 1) {
          total += L.get(i);
        }
        return total/(L.length());
}
```

Note that we had to specify that the method only accepted a List, of which all the values must be a double.

Now, if we look at the same method in JavaScript, you will see startk differences:

```
/** Given a list L of values, returns the average of L. */
fucntion getAverage(L) {
    var avg = 0;
    var total = 0;
        for (var i = 0; i < L.length; i += 1) {
          total += L[i]
        }
        return total/(L.length);
}
```

As you can see, we did not have to tell the JavaScript compiler thatL L was a list or that it could only accept values of type Double. The compiler ”knew” before we ran the code. The compiler also would not care if it added say 1.5 to 2. It would cast the result to a double.

1.2: Functions as Data

A very powerful feature of JavaSript is its functions. If you have used any Lisp dialect such as Scheme, then you know that functions are just data. The same idea holds in JavaScript. We can return functions ”on the fly”[1]

Here is an example of such a function:

```
/** Returns the sum of all the first n/2 elements of L */
function outputHalfSum(L) {
       var sum = function (a, b) {return a + b;}
       var midpoint = Math.floor(L.length/2);
       var total = 0;
       for (var i = 1; i < midpoint; i += 1) {
              total += sum(L[i], L[i - 1]);
       }
       return total;
}
```

Here, *sum* was our anonymous function. That is, we were able to define it within the scope of *outputHalfSum*.

1.3: Objects and Prototypes

JavaScript has all the necessary elements to make it useful for the Object Oriented Programming paradigm[2] There are multiple ways to define a JavaSctipt Object. The most common of which is to create a dictionary, create a function, or use the new Object method. For example, the following code creates an Employee object with two attributues, a NAME, and an EMPLOYEE ID, as well as a method to get their name, using the function declaration.

```
/** A new Employee with a NAME and EMPID. */
function Employee (name, empId) {
       this._name = name;
       this._empId = empId;
}

/** Returns THIS Employee's name. */
Employee.prototype.getName = function() {
       return this._name;
}
```

The following also creates an Employee using the dictionary declaration:

[1]These functions have different names, such as anonymous functions or Lambda functions. The name is not important, only the idea that they do not have to be defined outside another function’s scope.

[2]A programming paradigm as in functional, declarative, imperative, etc.

```
var _name = null;
var _empId = null;
/** Sets the NAME and EMPID data for THIS Employee. */
function setInstance(name, empId) {
       _name = name;
       _empId = empId
}
/** A new Employee with a NAME and EMPID. */
var Employee = {"name" : _name, "EmpId" : _empId};

/** Returns THIS Employee's Name. */
function getName() {
       return Employee["name"];
}
```

Here is a way to create the Object using new Object:

```
var _name = null;
var _empId = null;

/** Sets the Employee's name to NAME and employee ID to EMPLID. */
function getEmployeeInfo(name, empId) {
       _name = name;
       _empId = empId;
}

/* Create the new Employee Object. */
var Employee = new Object();

/* Set the instance variables. */
Employee.name = _name;
Employee.empId = empId;

/** Returns THIS Employee's name. */
Employee.getName = function() {
       return this.name;
}
```

Clearly the mothodologies are different, but very similar. The choice of which one to use is very dependent on the application you are working on. However, for the majority of this document, I will be using the functional declaration to define Objects.

1.3.1: Function Prototypes

A very powerful feature of Object Oriented Programming in JavaSctipt is that of Prototypes. Every object created is inherited from the Prototype Object. This can be quite confusing at first, but it allows us to add methods to built in objects. For example, say I wanted a new method on array objects to replace all numbers with the string equivalents. So the array [1, 2, 3] would become ['1', '2', '3']. Here is how one might do that task:

```
/** Replaces all elements in THIS Array to its String equvalents. */
Array.prototype.intToString = function() {
	for (var i = 0; i < Array.length; i += 1) {
		Array[i] = Array[i].toString();
	}
}
```

Prototypal inheritance is not only useful for native JavaScript objects, but it is used for objects we create. Take a look at the Employee object we created earlier. When we wanted to create a method to return the Employee's name, we had to add it to the object's prototye with the line Employee.prototype.getName = function()

1.3.2 Prototypal Inheritance

We can have inheritance in the same sense we could in Java. For example, let's say we want to extend our Employee class by adding a Nuclear Engineer. An Nuclear Engineer is an Employee with certain restrictions. First we need to add a method which determines weather or not an Employee can access the Control Room:

As a safety precaution, all Nuclear Engineers have access to the control room, but their Interns do not. To create the neccessary objects, we need to create what's called the **Prototype Chain**.

```
/** A Nuclear Engineer Employee with a NAME and EMPID. */
function NuclearEngineer(name, empId) {
	this._name = name;
	this._empId = empId;
	this.accessControlRoom = true;
}

/** A Nuclear Engineer's Intern with a NAME and EMPID. */
function NuclearEngineerIntern(name, empId) {
	this._name = name;
	this._empId = empId;
	this.accessControlRoom = false;
}

/* Sets up the Prototype chain. */
NuclearEngineer.prototype = new Employee();
NuclearEngineerIntern.prototype = new NuclearEngineer();
```

There is so much more to show with prototypes, unfortunately, it is out of the scope of this tutorial. I encourage you to visit Douglass Crockford's site on Prototypal Inheritance[3].

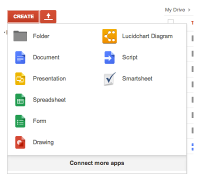
[3]http://www.crockford.com/javascript/inheritance.html

Chapter 2: A Simple Google Apps Script

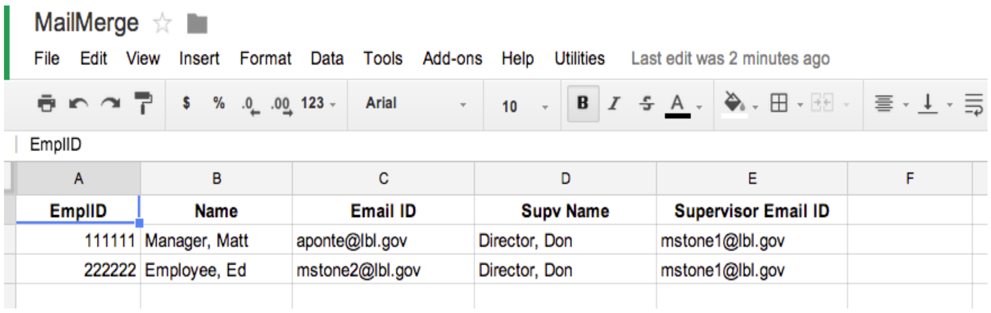
I believe the best way to learn a new technology is to dive in and build something. So let's do that. Say I want to create an App which will automatically send emails out for me via a spreadsheet. The values in the spreadsheet will be injected in the emails.

Section 2.1 Creating a New Project

To begin, go to your Google Drive, click create, and click spreadsheet.

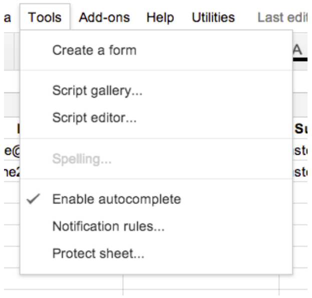


Name the spreadsheet MailMerge. Next, create 5 columns; EMPLID, NAME, EMAILID, SUPV NAME, SUPERVISOR EMAIL ID.



	A	B	C	D	E	F
	EmplID	Name	Email ID	Supv Name	Supervisor Email ID	
	111111	Manager, Matt	aponte@lbl.gov	Director, Don	mstone1@lbl.gov	
	222222	Employee, Ed	mstone2@lbl.gov	Director, Don	mstone1@lbl.gov	

The spreadsheet is pretty dumb right now. We still need a script in the background to do anything useful. Go to tools –> script editor to access the script for this spreadsheet. You should see an almost empty script.



Section 2.2: Menus and Functions

You will need an onOpen function for every Spreadsheet script. OnOpen is basically anologous to Java's public static void main() method. G.A.S. looks for onOpen when running a spreadsheet script. The following code will add a menu and point that menu to the function we will be using:

```
/** onOpen is part of the Google Apps Script library. It runs whenever the
  * spreadsheet is opened. Adds script menu to the spreadsheet.
  * Sometimes this seems not to run by itself when the script is installed
  * and needs to be run manually the first time to prompt script
  * authorization.
  */
function onOpen() {
  var ss = SpreadsheetApp.getActiveSpreadsheet();
  var menuEntries4 = [];
  menuEntries.push({name: "Send Merged Email", functionName:
      "sendMergedEmail"});
  ss.addMenu("Utilities", menuEntries);
}
```

The function that will run when the menu is selected is sendMergedEmail:

```
/** The result of "Send Merged Email" menu selection.
 * Looks through the user's drafts for an email
 * with a specified subject.
 */
function sendMergedEmail() {
  var mergeTags = [];
  var ss = SpreadsheetApp.getActiveSpreadsheet();
  var sheet = ss.getActiveSheet();
  var sheetName = sheet.getName();
  PropertiesService.getScriptProperties().setProperty("sheetName", sheetName);
  var draftSubjName = Defaults.DRAFT_SUBJECT_NAME;
  var draft = lookForDraft(draftSubjName);
  var bodyString = draft.getBody();
  var subjectString = draft.getSubject();
  var headers = fetchSheetHeaders(sheetName);
  var rowFormatsArray = sheet.getDataRange().getNumberFormats();
  var sheetValues = sheet.getDataRange().getValues();

  /* The recipients of the emails. 'Email ID' in this case. */
  var recipientsIndex = headers.indexOf('Email ID');

  var normalizedHeaders = normalizeHeaders(headers);

  /* Prepend a '$' to each header name. */
  mergeTags.push("$"+normalizedHeaders.join(",$"));

  /* Split the headers into an array. */
  mergeTags = mergeTags.toString().split(",");
  for (var i = 1; i < sheetValues.length; i += 1) {
    var recipient = sheetValues[i][recipientsIndex];
    var rowFormats = rowFormatsArray[i];
    var rowValues = sheetValues[i];
    var newBody = replaceStringFields(bodyString, rowValues, rowFormats,
        headers, mergeTags);
    var newSubject = replaceStringFields(draft.getSubject(), rowValues,
        rowFormats, headers, mergeTags);
    GmailApp.sendEmail(recipient, newSubject, "", {htmlBody: newBody});
  }
}
```

The functions normalizeHeaders, fetchSheetHeaders, etc. are simple enough that they are abstracted out. If you want to see the full code for this script, visit ocf.berkeley.edu/ aponte